Digitale Systemen

Ontwerpen met behulp van VHDL

vierde druk

Wim Dolman

Informatie over dit boek en andere uitgaven kunt u verkrijgen bij:
Wim Dolman
info@dolman-wim.nl
http://www.dolman-wim.nl/di/

Digitale Systemen is geschreven door Wim Dolman en in eigen beheer uitgegeven bij Lecturium uitgeverij.

eerste druk, 2011
tweede druk, 2012
derde druk, 2014

Vormgeving en opmaak is verzorgd door de auteur.

ISBN: 978-90-484-2020-9
NUR: 173/959

Voorwoord

Digitaal Ontwerpen is een vak dat gegeven wordt in het tweede studiejaar van de opleiding E-technology van de Hogeschool van Amsterdam. Dit vak behandelt het ontwerp van digitale systemen met behulp van de hardwarebeschrijvingstaal VHDL. Studenten, die dit vak volbracht hebben, kunnen een ontwerp van een complex digitaal systeem maken, dit ontwerp in VHDL implementeren en het testen met een simulatieprogramma. Om dit te kunnen, is kennis nodig van digitale systemen, ontwerpmethodieken, algoritmes, testmethodieken en van de hardwarebeschrijvingstaal VHDL.

Hoewel het vak reeds vele jaren bij de opleiding E-technology wordt gegeven, is het gebruik van dit boek nieuw. In het verleden zijn met wisselend succes diverse Engelstalige boeken gebruikt. Vanaf 2009 is de onderwijsvorm van het vak 'Digitale Systemen' aangepast en is als lesmateriaal een dictaat met dezelfde naam gebruikt. Dit dictaat is in 2010 uitgebreid en vormt de basis voor dit boek.

Er zijn veel Engelstalige boeken over het ontwerpen van digitale systemen beschikbaar. Deze zijn te verdelen in twee groepen: boeken die digitale systemen in het algemeen bespreken en boeken die een hardwarebeschrijvingstaal behandelen. De eerste groep heeft een grote overlap met boeken over digitale techniek en microcontrollers. De tweede groep is vooral gericht op de syntax van de hardwarebeschrijvingstaal en minder op het ontwerpen van een digitaal systeem.

Dit boek sluit aan bij het vak Digitale Techniek en het vak Microcontrollers van de opleiding E-technology. Basiskennis over digitale techniek en microcontrollers wordt bekend verondersteld. In plaats hiervan richt dit boek zich vooral op het ontwerpen en testen van digitale systemen. De hardwarebeschrijvingstaal, die in dit boek gebruikt wordt, is VHDL. De syntax van deze taal wordt niet uitgebreid beschreven. VHDL is een zeer rijke taal met heel veel mogelijkheden. Wel geeft dit boek veel complete beschrijvingen en besteedt het ruime aandacht aan de synthese- en simulatieresultaten. Synthese is het omzetten van een VHDL-beschrijving naar een fysieke schakeling. Daarnaast bevat het boek veel informatie over de technologie en over de verschillende implementatiemogelijkheden.
Dit boek is bijzonder omdat het zich richt op zowel de hardwarebeschrijvingstaal VHDL, als op het ontwerpen van digitale systemen en de implementatiemogelijkheden.

Het boek Digitale Systemen bestaat uit elf hoofdstukken. De eerste vier hoofdstukken behandelen de basisaspecten van het ontwerpen van een digitaal systeem met VHDL.

- Hoofdstuk 1 is een inleiding over digitale systemen. Uitgelegd wordt wat een digitaal systeem is en welke varianten er bestaan. Een digitaal systeem wordt — net als een microprocessor of microcontroller — toegepast bij intelligente systemen. De verschillen en overeenkomsten tussen al deze systemen en mogelijkheden worden uitgelegd.
- Hoofdstuk 2 is een inleiding in VHDL. Voor eenvoudige combinatorische en sequentiële schakelingen worden verschillende beschrijvingen in VHDL gemaakt. Daarbij wordt vooral gekeken naar de verschillende methoden van beschrijven en het effect op het synthese- en simulatieresultaat.
- Hoofdstuk 3 bespreekt het simulatiemodel van VHDL. Met VHDL is het mogelijk het parallelle gedrag hardware vast te leggen en dit parallelle gedrag te simuleren. Het verschil tussen parallelle en sequentiële constructies uit de taal VHDL komen daarbij aan de orde.
- Hoofdstuk 4 bespreekt de synthese met VHDL. Niet alle VHDL is synthetiseerbaar. Eerst wordt verteld wat wel en wat niet synthetiseerbaar is en daarna worden er een paar sjablonen gegeven voor synthetiseerbare combinatorische en synthetiseerbare sequentiële beschrijvingen. Deze sjablonen worden bij de ontwerpvoorbeelden toegepast. Het hoofdstuk besluit met een groot aantal adviezen, afspraken en ontwerpregels.

Na de bestudering van deze vier hoofdstukken, kan de lezer een synthetiseerbare VHDL-beschrijving van een combinatorische en een sequentiële schakeling maken en een testbench voor deze schakelingen opstellen in VHDL.

Hoofdstuk 5 en 6 bespreken de methodieken, die nodig zijn voor het maken van een VHDL-beschrijving van een complex digitaal systeem.

- Hoofdstuk 5 bespreekt de toestandsmachines. Er wordt aandacht besteed aan het opstellen van een toestandsdiagram en aan de implementatie van het toestandsdiagram in VHDL.
- Hoofdstuk 6 gaat over het ontwerpen van digitale systemen en is het hart van het boek. Verschillende methodieken worden besproken, daarbij ligt de nadruk op de methodiek met een gescheiden dataverwerking en besturing. Belangrijke onderdelen van deze methodiek zijn de tekening van het dataverwerkingsdeel en het toestandsdiagram voor de besturing.

Na afloop is de lezer in staat een ontwerp te maken met behulp van de methodiek met een gescheiden dataverwerking en besturing. Bovendien kan de lezer van dit ontwerp een VHDL-beschrijving maken op basis van combinatorische en sequentiële schakelingen.

In hoofdstuk 7 en 8 komt het ontwikkelen van een algoritme aan bod.

- Hoofdstuk 7 legt uit wat een algoritme is en onderzoekt voor het berekenen van een grootste gemene deler verschillende algoritmes, die in C geschreven zijn. Het meest belovende algoritme wordt in hardware geïmplementeerd. Het hoofdstuk eindigt met een lijst van adviezen voor het vinden van een bruikbare algoritme.

- Hoofdstuk 8 bestudeert verschillende algoritmes voor BCD-bewerkingen, zoals een BCD-opteller, een BCD-teller en een omzetter van binaire getallen naar BCD.

Na de bestudering van deze twee hoofdstukken, is de lezer op de hoogte met de aanpak om een geschikt algoritme te vinden en is in staat om dat algoritme in VHDL te implementeren.

De laatste drie hoofdstukken geven meer achtergrond informatie over de verificatie van digitale systemen, de mogelijkheden van de beschikbare VHDL-bibliotheken en over de technologie van de digitale systemen.

- Hoofdstuk 9 gaat over verificatie en behandelt met name de simulatie, maar bespreekt ook de statische en dynamische tijdsanalyses. Besproken worden verschillende modellen voor een testbench en het opstellen van een test.
- Hoofdstuk 10 bespreekt de verschillende packages, die bij het ontwerpen in VHDL gebruikt worden.
- Hoofdstuk 11 behandelt de verschillende implementatievormen voor digitale systemen in de ruimste zin van het woord. Naast de populaire FPGA worden de CPLD, de ASIC's en andere opties besproken. De technologie achter deze implementatievormen is steeds anders en wordt eveneens toegelicht. Het hoofdstuk sluit af met een beschouwing over de toekomstige technologische mogelijkheden.

Digitale Systemen is geschreven in LaTeX. Teksten met programmacode kunnen veel beter en eenvoudiger gemaakt worden met LaTeX, dan met een programma als Word. De gebruikte compiler is pdfeTeX, die standaard bij de Cygwin-omgeving zit. De hoofdtekst is gezet in Garamond; voor de wiskundige formules is Math Design en voor de programmacode is Bera Mono gebruikt. Dit laatste font is de TeX-variant van Bitstream Vera Mono. Voor de opmaak van de code is de *lstlisting*-stijl gebruikt. De gereserveerde namen zijn in de code en in de tekst vet gedrukt. De meeste tekeningen zijn gemaakt met Mayura Draw en als PDF-bestand toegevoegd.
Het boek bevat meer dan vijfhonderd figuren, tabellen en complete programma's. Daarnaast bevat het boek ook nog honderden codefragmenten. Een groot deel van de programmacodes is, zonder enige aanpassing, direct te gebruiken.

Alle VHDL-beschrijvingen in het boek zijn getest met de IEEE-standaard VHDL-2002. Voor de simulatie zijn verschillende versies van Modelsim gebruikt. De synthese is gedaan met Leonardo Spectrum v2007.a, Precision RTL Synthesis 2006a en diverse versies van Quartus II.
VHDL-2002 is niet de meeste recente standaard. In 2008 heeft de IEEE een nieuwe standaard gepubliceerd. Op dit moment is VHDL-2008 niet of maar gedeeltelijk in de nieuwe ontwikkelomgevingen opgenomen. Omdat de nieuwe aanpassingen aanzienlijk zijn en deze veel interessante mogelijkheden biedt, wordt deze standaard wel in bijlage C besproken.

Bij het boek Digitale Systemen hoort een internetsite met een studiewijzer, een toolbox, practicumopdrachten en informatie over de gebruikte software:
http://www.dolman-wim.nl/di/.

Inhoud

Bijlagen

1

Digitale Systemen

Doelstelling	In dit hoofdstuk leer je wat een digitaal systeem is, hoe deze is opgebouwd en waar digitale systemen worden toegepast.
Onderwerpen	De behandelde onderwerpen zijn: ▪ De definitie van een digitaal systeem. ▪ Het begrip embedded systeem en de verschillende implementatievormen voor de embedded systemen: microprocessorsystemen, programmeerbare bouwstenen, digitale schakelingen en applicatiespecifieke geïntegreerde schakelingen. ▪ De overeenkomst tussen programmeerbare bouwstenen en applicatiespecifieke geïntegreerde schakelingen. ▪ Het verschil tussen programmeerbare logische bouwstenen en een microcontroller of een microprocessor. ▪ Toepassingsgebieden voor programmeerbare logische bouwstenen. ▪ De mogelijkheden van IP- of softcores.

Het begrip *digitale systemen* heeft meerdere betekenissen met een wisselende inhoud en reikwijdte. Het wordt gebruikt voor zowel complexe embedded systemen als voor systemen met eenvoudige digitale componenten.

De Engelse term *embedded system* betekent letterlijk ingebed systeem.

Digitaal betekent dat de informatie uit een discreet aantal signaalniveaus bestaat. Meestal zijn dat twee niveaus: hoog en laag. Het hoeft echter niet altijd om een spanningsniveau te gaan. Het kan ook gaan om aan of uit, geladen of ongeladen, geleidend of niet geleidend, ja of nee, '1' of '0'.

Het woord *digitaal* verbindt men tegenwoordig direct aan moderne, intelligente apparaten, die bijvoorbeeld gebruikt worden in de computertechniek, automatisering, robotica, medische wereld en bij wetenschappelijk onderzoek. Ook denkt men direct aan allerlei elektronische apparatuur in en om het huis, zoals een radio, televisie, fotocamera, dvd, wii of ipod. Maar ook een kamerthermostaat, magnetron, scheerapparaat, vaatwasser, wasmachine en een elektrische tandenborstel bevatten vaak een of meer digitale componenten.

Er zijn veel verschillende definities van een embedded systeem. Een fraaie definitie is deze:
'Een embedded systeem is een intelligent systeem dat er niet uit ziet als een computer! Dus zonder muis, beeldscherm en toetsenbord.'.
De gebruiker van een embedded systeem ervaart niet dat er een computer in zit.

Een *systeem* is een verzameling componenten, die zo georganiseerd zijn dat ze specifieke functies kunnen vervullen. Een digitaal systeem is een systeem dat informatie digitaal verwerkt. In veel gevallen wordt hier het begrip *embedded system* voor gebruikt.

1.1 Embedded systemen

Embedded systemen zijn intelligente systemen. Het hart van een embedded systeem is daarom vaak een microprocessor of microcontroller. De software of firmware zorgt voor de intelligentie. Maar er zijn ook andere digitale bouwstenen beschikbaar om een intelligent systeem te maken.

Firmware is de software waarmee een systeem geprogrammeerd is. Embedded systemen hebben meestal een vast programma dat wordt uitgevoerd. Bij een pc is dat anders. De gebruiker bepaalt de applicatie die gestart wordt. Bij een magnetron geeft de gebruiker bijvoorbeeld via een eenvoudige interface aan wat hij wil. Het programma dat de processor uitvoert, is altijd hetzelfde.

Microprocessorsysteem

De kern van een microprocessor of microcontroller is de centrale verwerkingseenheid of CPU, *central processor unit*. Deze is opgebouwd uit een centrale rekeneenheid — ALU (*arithmetic logic unit*) — die de rekenkundige (*arithmetic*) en logische (*logic*) bewerkingen uitvoert en een aantal registers, zoals de programmateller *program counter* en het statusregister.
In het programmageheugen staat de machinecode met de opdrachten voor de centrale verwerkingseenheid. De enen en nullen van de machinecode vormen de instructies voor de processor.

Het programma voor de processor wordt in een taal als Java, C of C++ geschreven en met een compiler naar machinecode vertaald. Soms wordt er een operating system toegevoegd, bijvoorbeeld Linux of Windows CE, of wordt er een real time operating system gebruikt.

PLD

Het hart van een embedded systeem kan ook een programmeerbare logische bouwsteen of PLD, *programmable logical device* zijn. Dat is een component waarvan de functionaliteit van de hardware nog niet vastligt. Het is opgebouwd uit een grote hoeveelheid flipfloppen en eenvoudige logische functies, die aan elkaar gekoppeld kunnen worden tot een complexe digitale schakeling.
De enen en nullen uit het bestand waarmee de PLD geprogrammeerd wordt, leggen de configuratie van de PLD vast. De configuratie definieert de logische functies, flipfloppen en verbindingen, die nodig zijn voor het functionele gedrag van de PLD.

De naam PAL is een geregistreerd handelsmerk van Monolithic Memories Inc. (MMI), dat in 1987 overgenomen is door AMD. Daarom gebruiken andere fabrikanten alternatieve namen. Lattice spreekt van Generic Array Logic (GAL) en ICT van Programmable Electrically Erasable Logic (PEEL).

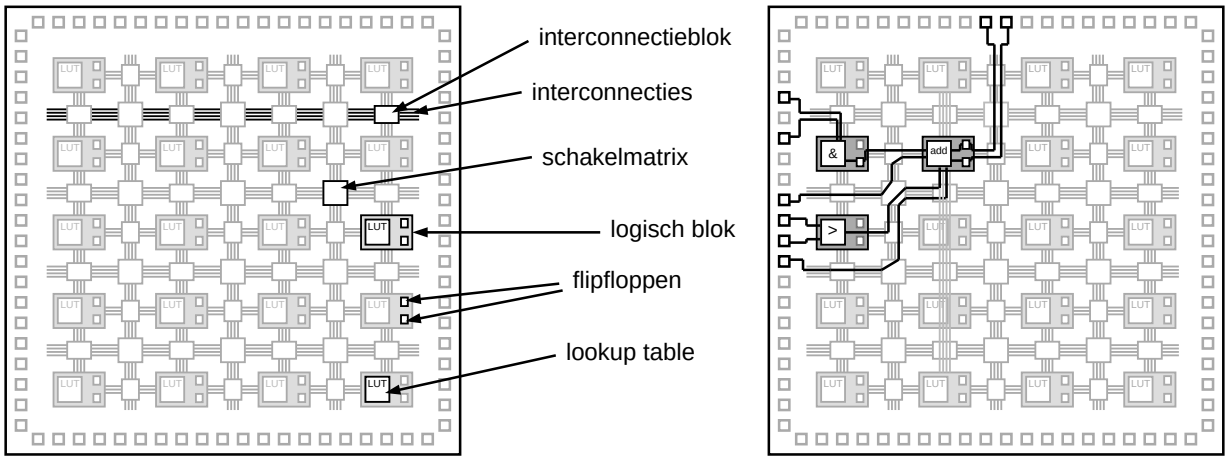
Men onderscheid drie soorten PLD's:

- FPGA, *field programmable gate array*,
- CPLD, *complex programmable logic device*,
- PAL, *programmable array logic*.

Let op: PAL is ook een standaard voor analoge televisieuitzendingen. In dat geval betekent het *phase alternating line*.

De PAL is de kleinste en eenvoudigste component en wordt soms aangeduid met *simple programmable logic device* of SPLD. In nieuwe producten wordt de PAL niet meer toegepast. CPLD's zijn gebaseerd op EEPROM- of Flash-technologie. FPGA's gebruiken RAM- of antifuse-technieken. In hoofdstuk 11 worden de verschillende soorten PLD's uitgebreid besproken.

In figuur 1.1 staat een vereenvoudigde voorstelling van een FPGA. Een FPGA is een groot array van logische blokken, die via interconnectieblokken en schakelmatrices met elkaar verbonden kunnen worden. Elk logisch blok bevat een LUT (*lookup table*) voor de logische functie en een paar D-flipfloppen als geheugenfuncties. FPGA's bevatten een enorme hoeveelheden logische blokken. De kleinste FPGA's hebben al enkele duizenden logische blokken. Grote FPGA's bestaan uit honderdduizenden logische blokken.



Figuur 1.1 : Een vereenvoudigde voorstelling van een FPGA.
De FPGA is een array van programmeerbare logische blokken en interconnecties. Links staat een ongeprogrammeerde FPGA en rechts staat een geprogrammeerde FPGA met drie LUT's, zes ingangen en twee uitgangen.

Het programmeren van een FPGA bestaat uit het selecteren van de logische blokken met de juiste logische functies en het maken van de benodigde verbindingen. De programmeur bouwt in de FPGA dus digitale schakelingen. Vanwege de enorme omvang van FPGA's gaat het uiteindelijk om complexe digitale hardware.

Sommige schrijvers reserveren de term PLD specifiek voor PAL. De hele groep programmeerbare bouwstenen wordt dan vaak met ASIC aangeduid.
Een fraaie, maar minder gangbare term, voor de hele familie programmeerbare bouwstenen is FPL: *field programmable logic*.

Systeem met discrete componenten

In plaats van een systeem te ontwerpen voor een programmeerbare bouwsteen kan het natuurlijk ook opgebouwd worden uit discrete componenten. Bijvoorbeeld een schakeling met TTL-logica uit de 7400-serie of CMOS-logica uit de 4000-serie.

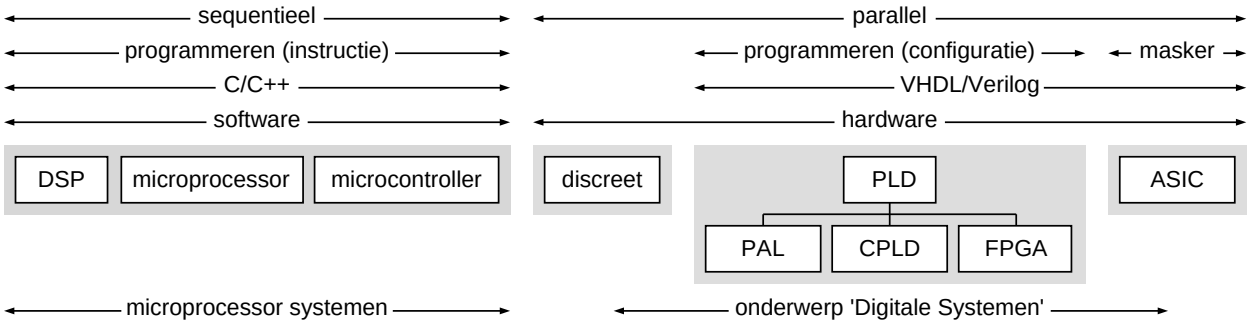
ASIC

Een andere implementatie is een eigen digitale geïntegreerde schakeling. De schakeling wordt dan in een IC-fabriek direct op het silicium aangebracht. Een eigen specifieke geïntegreerde schakeling wordt een ASIC — *application specific integrated circuit* — genoemd. Hoofdstuk 11 bespreekt de technologie voor digitale systemen en daarbij is ruimschoots aandacht voor de ASIC.

Een IC is een *integrated circuit* of geïntegreerde schakeling. Een ander term voor een IC is chip.

1.2 Overzicht embedded systemen

Hoewel de verschillen tussen een oplossing met discrete componenten en met een PLD of met een ASIC groot zijn, wordt er in al deze gevallen hardware ontwikkeld en zijn er bij het ontwerp veel overeenkomsten.
Figuur 1.2 geeft een overzicht van de diverse implementatiemogelijkheden voor een embedded systeem. Dit boek beperkt zich voornamelijk tot de programmeerbare bouwstenen of PLD's. Een aantal onderwerpen zijn ook van toepassing bij het ontwerp met discrete componenten en de meeste onderwerpen zijn ook relevant bij het ontwikkelen van ASIC's.



Figuur 1.2 : Een overzicht van de implementatiemogelijkheden voor een embedded systeem.
Hardware is van nature parallel en software sequentieel. Voor software zijn talen als C en C++ geschikt. Voor hardware zijn hardwarebeschrijvingstalen als VHDL en Verilog nodig. Microprocessorsystemen hebben machinecode met instructies nodig. PLD's moeten geconfigureerd worden.
'Digitale Systemen' behandelt vooral de programmeerbare bouwstenen. De nadruk ligt op hardware en op de taal VHDL.

Microprocessorsysteem versus PLD

Ontwerpers, die voor het eerst te maken krijgen met programmeerbare bouwstenen, benaderen het ontwikkelen van een systeem met PLD's vaak hetzelfde als het ontwikkelen van een microprocessorsysteem. Dit beeld wordt versterkt doordat PLD's, net als bijvoorbeeld microcontrollers, geprogrammeerd worden en doordat er een programmeertaal nodig is. Men denkt daardoor dat de overstap van een microcontroller naar een FPGA bestaat uit het leren van een andere programmeertaal.

Het verschil tussen een microprocessorsysteem en een systeem met een PLD is groot. Een microprocessorsysteem is sequentieel. De hardware van de processor ligt vast. Zolang het geen *multiple core* processor betreft, worden de instructies sequentieel na elkaar uitgevoerd. Het programma wordt stap voor stap doorlopen.

Bij het configureren van een programmeerbare bouwsteen worden, net als bij het programmeren van een processor, programmeertalen gebruikt. Het gaat hier echter niet om instructies, maar om het bouwen van digitale hardware in de programmeerbare logische bouwsteen. Hardware bestaat altijd uit onderdelen die de taken parallel, dus tegelijkertijd, uitvoeren. Elk onderdeel, op elk niveau, bepaalt op ieder moment wat de nieuwe toestand is.

Het ontwerpen van parallelle hardware voor PLD's verschilt conceptueel sterk met het ontwerpen van sequentiële programma's voor microprocessorsystemen. Dit betekent dat er andere strategieën en methodieken nodig zijn voor het ontwerp van embedded systemen op basis van PLD's.

Om een digitaal ontwerp op een gewone PC met een sequentiële processor te kunnen simuleren, zijn speciale hardwarebeschrijvingstalen ontwikkeld. Dit boek gebruikt VHDL als hardwarebeschrijvingstaal.

De overeenkomst tussen PLD's en ASIC's
Bij PLD's wordt een van de hardwarebeschrijvingstalen VHDL of Verilog gebruikt om een configuratiebestand te maken waarmee de PLD geprogrammeerd wordt. Het configuratiebestand legt vast welke flipfloppen, welke logica en welke verbindingen nodig zijn.
Een ASIC wordt niet geprogrammeerd. Voor het maken van een IC zijn fotografische maskers nodig. Daarmee wordt het silicium op de juiste plaatsen belicht om verbindingen en transistoren te creëren. Voor het maken van de maskers of een deel van de maskers moet men weten welke componenten en verbindingen er op het silicium nodig zijn. Dit verschilt niet wezenlijk met het programmeren van een PLD. Sommige ASIC-technieken worden daarom *mask programmable* genoemd. Hoofdstuk 11 licht de technologie van de ASIC verder toe.

1.3 Toepassingsgebieden voor programmeerbare logische bouwstenen

Al de implementatietechnieken uit figuur 1.2 zijn zinvolle alternatieven voor het maken van een embedded systeem. De keuze voor een bepaalde techniek hangt af van de eisen die er aan het embedded systeem worden gesteld. Er zijn vele soorten microprocessoren, microcontrollers, CPLD, FPGA's en ASIC's. Dit boek behandelt vooral de programmeerbare logische bouwstenen. De voordelen van deze bouwstenen zijn:

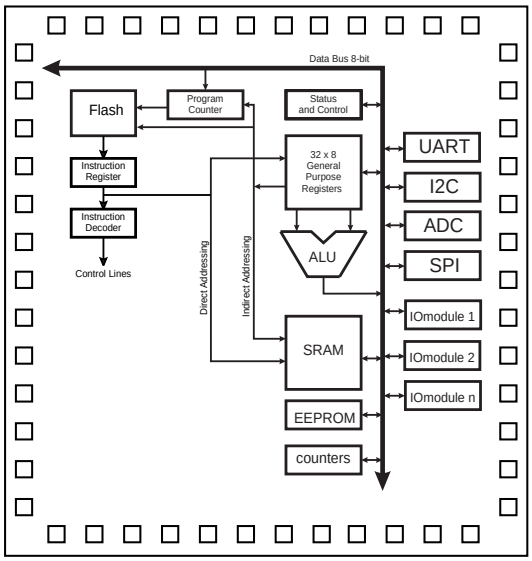
- Elke digitale schakeling kan gerealiseerd worden met een PLD.
- PLD's zijn zeer geschikt om taken parallel uit te voeren.
- PLD's hebben zeer veel aansluitpinnen.
- De oplossing wordt direct in hardware gerealiseerd met minimale overhead.
- PLD's zijn zeer geschikt voor systemen met veel dataverwerking.
- Met PLD's zijn grote dataverwerkingssnelheden mogelijk.
- In vergelijking met een microprocessor zijn er minder klokslagen nodig.
- De integratie van alle digitale componenten in een PLD vereenvoudigt het printed circuit board.
- Ten opzichte van discrete componenten en ASIC's zijn PLD's flexibel.
- PLD's worden bij de prototyping van ASIC's gebruikt.
- Er zijn veel kant-en-klare commerciële en niet-commerciële deeloplossingen (IP-cores of softcores) beschikbaar.

De nadelen van programmeerbare logische bouwstenen zijn:

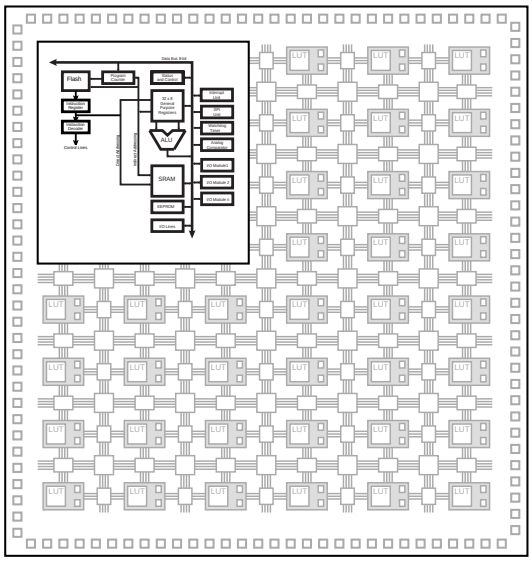
- PLD's zijn minder geschikt voor ontwerpen met heel veel besturing.
- PLD's hebben weinig mogelijkheden wat betreft analoge interfacing.
- PLD's zijn bijna alleen verkrijgbaar als *surface mounted device*.
- De ontwerper moet een hardwarebeschrijvingstaal leren.
- Ontwerpen in hardware zijn minder flexibel dan de software voor een microprocessorsysteem.

De kracht van PLD's en met name FPGA's zit hem in het feit dat de oplossing geïmplementeerd wordt in hardware en niet in software. Systemen met veel dataverwerking en met hoge snelheden zijn geschikt voor een FPGA. Typische applicatiegebieden zijn: beeldbewerking, visionsystemen, communicatienetwerken,

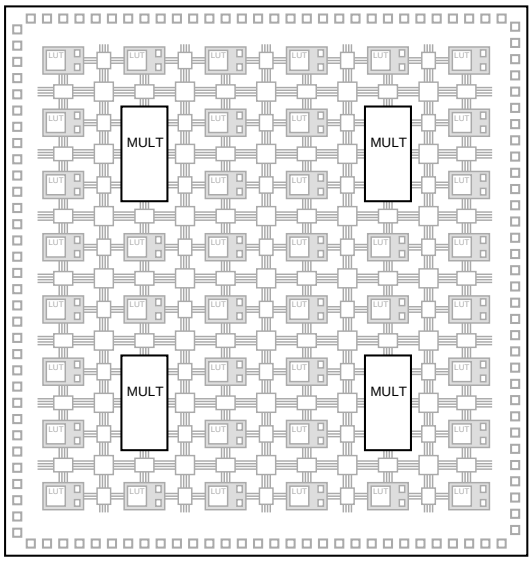
interfacing van systemen met verschillende klokken of protocollen, spraakherkenning, analyse DNA-structuren, cryptografie en ASIC-prototyping.



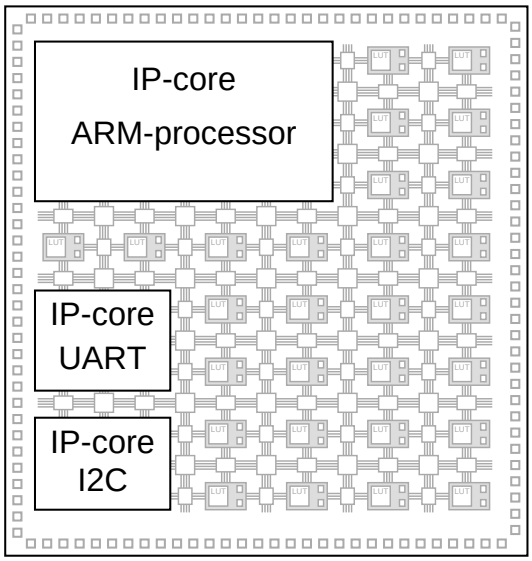
(a) Microcontroller



(b) Microcontroller en FPGA in één behuizing



(c) FPGA met extra multipliers



(d) FPGA met IP-cores

Figuur 1.3 : Alternatieven voor en varianten van een FPGA.

Figuur 1.3 toont een aantal alternatieven of varianten van FPGA's. Voor een systeem met veel besturing en analoge interfacing is een microcontroller vaak een beter alternatief. Het schrijven van een programma in C is eenvoudiger dan het ontwikkelen van een complete digitale schakeling met een hardwarebeschrijvingstaal. Atmel heeft een bouwsteen FPSLIC met een AVR-microcontroller en een FPGA-structuur in één behuizing. Digitale signaalprocessoren (DSP) hebben naast de

processor extra rekeneenheden om snel data te kunnen verwerken. Op een FPGA is vanwege dezelfde reden vaak een groot aantal vermenigvuldigers en extra RAM aanwezig.

Een zeer interessant aspect van FPGA's is dat er veel kant-en-klare oplossingen zijn. De meeste ontwikkelomgevingen voor FPGA's kennen het gebruik van geparametriseerde onderdelen, zoals: tellers, RAM, lifo's en fifo's.

lifo en fifo zijn beide geheugens. lifo staat voor een *last in first out* en fifo staat voor een *first in first out*. de fifo wordt als buffer gebruikt en met een lifo wordt een stack geïmplementeerd.

Voor allerlei andere onderdelen zijn er complete oplossingen te koop of als freeware te downloaden van het internet. Een dergelijke oplossing noemt men softcore of IP-core. IP staat voor *intellectual property* oftewel intellectueel eigendom. Voorbeelden van beschikbare softcores zijn: een CAN-bus, een USB-interface, een DMA-controller en een I^2C -controller.

USB staat voor *universal serial bus*. CAN is een seriële bus die vooral toegepast wordt in auto's. DMA staat voor *direct memory access*. I^2C is een serieel tweedraadsinterface voor printed circuit boards.

Figuur 1.3d toont een FPGA met een IP-core voor een USB-interface, een I^2C-controller en een ARM-processor. De ARM-processor is een populaire processorstructuur, die de basis is voor veel microprocessoren, zoals de AT91SAM-familie van Atmel en de i.MX-familie van Freescale. Een belangrijke ARM-softcore voor FPGA's is de Cortex-M1.

Een softcore is meestal configureerbaar. Dat betekent dat de ontwerper precies de functionaliteit kan selecteren die nodig is voor het ontwerp. Overbodige onderdelen worden dan niet geïmplementeerd.

Er bestaan ook softcores van de PIC-controller van Microchip en van de AVR-controller van Atmel. Het gaat bij een softcore van een microcontroller voornamelijk om de processor en niet om de randmodulen. Analoge modulen, zoals een ADC, kunnen op een FPGA niet geïmplementeerd worden en voor digitale interfaces, zoals CAN of I^2C, bestaan al andere oplossingen.

1.4 Digitale Systemen

In het overzicht van figuur 1.2 is aangegeven wat het onderwerp van 'Digitale Systemen' is. Bij de ontwerpvoorbeelden in dit boek gaat het voornamelijk om programmeerbare bouwstenen en dan vooral die voor een FPGA. De hardwarebeschrijvingstaal die gebruikt wordt, is VHDL.

Hoewel de nadruk op FPGA en VHDL ligt, zijn de gebruikte ontwerpmethodieken ook geschikt voor PAL, CPLD, ASIC en voor ontwerpen met discrete componenten en eveneens toepasbaar bij andere hardwarebeschrijvingstalen, zoals Verilog. In alle gevallen gaat het bij het ontwerp van digitale systemen om het ontwerp van hardware.

2

De taal VHDL

Doelstelling	In dit hoofdstuk leer je wat het belang van de taal VHDL is, maak je kennis met voorbeelden in VHDL en maak je kennis met simulatie en met synthese.
Onderwerpen	De behandelde onderwerpen zijn: ▪ De kracht van de taal VHDL. ▪ De achtergronden en geschiedenis van de taal VHDL. ▪ Alternatieven voor VHDL. ▪ De ontwerp-eenheden van VHDL: **`entity`** en **`architecture`**. ▪ Diverse gedrags-, dataflow- en structuurbeschrijvingen van een full-adder. ▪ Het ontwerptraject met VHDL. ▪ VHDL-beschrijvingen van synchrone schakelingen: de D-flipflop en dataregisters.

De hardwarebeschrijvingstaal VHDL is een krachtig hulpmiddel bij het ontwerp van digitale systemen. Deze kracht ligt vooral in het feit dat deze taal op bijna alle ontwerpniveaus toepasbaar is: VHDL kan de specificatie van een digitaal systeem vastleggen; door zijn modulaire opbouw is VHDL geschikt voor het systematisch opdelen in subsystemen; VHDL wordt gebruikt voor gedragsbeschrijvingen, netwerkbeschrijvingen en beschrijvingen op poortniveau; de testomgeving kan volledig in VHDL worden gemaakt; een subset van VHDL is synthetiseerbaar en alle belangrijke ontwikkelomgevingen voor digitale systemen ondersteunen VHDL.

VHDL kent veel vrijheden en ondersteunt vele soorten van beschrijvingen. De taal is onafhankelijk van de technologie, zij is geschikt voor alle digitale systemen, dus voor CPLD's, FPGA's én ASIC's. Allerlei ontwerpstrategieën en methodieken zijn mogelijk: synchroon en asynchroon, *bottom up* en *top down*, een verdeel-en-heersstrategie of bij het gebruik van het V-model.

De brede toepasbaarheid en de grote ontwerpvrijheid zorgen er wel voor dat het een taal is die op veel manieren gebruikt wordt en dat er veel verschillende ontwerpstijlen zijn. Anderzijds is VHDL slechts een hulpmiddel bij het systematisch ontwerpen van digitale systemen. Een hardwarebeschrijvingstaal, zoals VHDL, geeft de ontwerper meer inzicht in het systematisch ontwerpen.

2.1 Ontstaan en ontwikkeling van VHDL

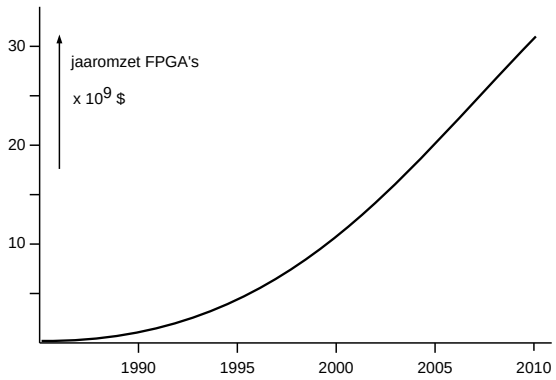
VHDL staat dus niet voor *very high description language*. Soms noemt men het — terecht of niet terecht — *very huge description language*.

Andere omschrijven VHDL — met recht — als *verification & hardware description language*.

Dit boek beperkt zich tot VHDL-2002. Bijlage C geeft de belangrijkste aanpassingen van VHDL-2008.

IEEE staat voor *Institute of Electrical and Electronics Engineers* en is een internationale organisatie die elektrotechnische standaarden ontwikkelt. IEEE wordt uitgesproken als *eye-triple-e*.

De ontwikkeling van VHDL is begin jaren tachtig gestimuleerd door het Amerikaanse ministerie van defensie (*Department of Defense*). Er kwamen steeds meer militaire producten op de markt met complexe geïntegreerde schakelingen. Er was behoefte aan duidelijke, uniforme beschrijvingen van de gebruikte componenten.
In eerste instantie is VHDL ontwikkeld als beschrijvingstaal voor digitale geïntegreerde schakelingen. De letters VHDL staan voor *VHSIC Hardware Description Language* en daarin staat VHSIC voor *Very High Speed Integrated Circuit*. Bij de ontwikkeling van de taal is al besloten om rekening te houden met de mogelijkheid om VHDL te gebruiken voor simulaties. VHDL is een IEEE-standaard. In 1987 is de eerste officiële standaard uitgebracht. In 1993, in 2002 en in 2008 zijn er een aantal belangrijke wijzigingen aangebracht.
Eind jaren zeventig kwamen ook de eerste programmeerbare bouwstenen op de markt. In de loop van de jaren tachtig zijn daar eenvoudige talen als PALASM en ABEL, *a boolean equation language*, voor ontwikkeld. Met deze talen werd de digitale schakeling beschreven en daarna door een compiler omgezet naar een bestand met enen en nullen waarmee de bouwstenen geprogrammeerd konden worden. Deze ontwerpstap wordt synthese genoemd en deze compilers worden synthesizers genoemd.
In loop van de jaren tachtig is het idee ontstaan dat VHDL ook voor synthese gebruikt zou moeten worden. Begin jaren negentig kwamen de eerste synthesizers voor VHDL beschikbaar. Halverwege de jaren negentig nam de omzet van FPGA's en daarmee ook het gebruik van VHDL enorm toe. Figuur 2.1 toont de omzetgroei van FPGA's gedurende de laatste twintig jaar.



Figuur 2.1 : De omzetgroei van FPGA's in miljarden dollars.

2.2 Alternatieven voor VHDL

VHDL is niet de enige hardwarebeschrijvingstaal. Alternatieven zijn onder andere Verilog, SystemVerilog, Handle-C en SystemC. Handle-C en SystemC zijn varianten op basis van de taal C. Het voordeel van deze talen is dat co-design — het samen ontwikkelen van software en hardware — eenvoudiger is. Deze talen worden echter niet door alle ontwikkelomgevingen ondersteund.

Verilog is, net als VHDL, een echte hardwarebeschrijvingstaal en wordt door alle ontwikkelomgevingen voor digitale systemen ondersteund. Sterker, het is mogelijk om Verilog en VHDL te mixen. Een VHDL of Verilog-ontwerp kan bestaan uit componenten die in VHDL of die in Verilog zijn geschreven.
Hoewel de talen verschillend ogen, zijn de verschillen tussen Verilog en VHDL niet enorm groot. Verilog lijkt meer op C en VHDL meer op Ada, Pascal en Delphi. Ontwerpen in Verilog zijn beknopter; er zijn minder regels code nodig. VHDL is daarentegen beter leesbaar en heeft meer mogelijkheden. Verilog wordt veel gebruikt in de Verenigde Staten en de rest van Amerika. VHDL is juist heel populair in Europa en Japan.

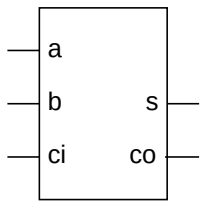
2.3 Het basisconcept van VHDL

Een digitaal systeem of digitaal subsysteem wordt in VHDL een *design unit* of ontwerpeenheid genoemd. Een ontwerpeenheid bestaat uit twee delen: een *entity* en een *architecture*. De entity definieert de in- en uitgangsrelaties van het systeem met zijn omgeving. De architectuur legt de functionaliteit van het systeem vast. Deze paragraaf gebruikt, net als paragraaf 2.7, een full-adder als voorbeeld van een ontwerp-eenheid. Een full-adder is een 1-bit opteller met carry-in en carry-out. De full-adder is een combinatorische schakeling met drie ingangen en twee uitgangen. Het symbool staat in figuur 2.2. De functietabel met het functionele gedrag staat in tabel 2.1.

Het Engelse woord *entity* komt overeen met het Nederlandse begrip entiteit, dat zoiets als 'iets wezenlijks bestaands' betekent.
Dit begrip komt in veel vakgebieden voor, zoals in de systeemtheorie, filosofie en psychologie.

Entity

De entity van de full-adder staat in code 2.1. Een entity begint met het sleutelwoord **`entity`** met direct daar achter de naam van de entity en het sleutelwoord **`is`**. De entity wordt afgesloten met de sleutelwoorden **`end entity`** en daarna de entitynaam en een puntkomma.



Figuur 2.2 : Het symbool van de full-adder.

Tabel 2.1 : De functietabel van de full-adder.

a	b	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Code 2.1 : De entity van een full-adder.

```
1  entity fulladder is
2    port (
3      a, b : in  std_logic;
4      ci   : in  std_logic;
5      co   : out std_logic;
6      s    : out std_logic
7    );
8  end entity fulladder;
```

De entity bevat een *port list* met de in- en uitgangen. De port-list begint met het sleutelwoord **`port`** en een ronde openingshaak en wordt afgesloten met een ronde sluithaak en een puntkomma. De port-list bestaat uit vier declaraties. Op regel 3 wordt de ingangen `a` en `b` gedeclareerd en daarna ingang `ci` en de uitgangen `s` en `co`. Na de dubbele punt staat de *mode* of modus van de aansluiting en het signaaltype. De modus van de ingangen is **`in`** en die van de uitgangen is **`out`**. Het signaaltype

is voor al de vijf aansluitingen `std_logic`. De declaraties worden gescheiden door puntkomma's en na de laatste declaratie mag geen puntkomma staan.
In de afgebeelde code zijn de sleutelwoorden vetgedrukt. VHDL is ongevoelig voor boven- of onderkast; hoofd- en kleine letters kunnen door elkaar gebruikt worden. In VHDL zijn de namen **`EntiTY`**, **`ENTITY`**, **`entITY`** en **`entity`** identiek.
In dit boek worden constanten meestal met hoofdletters geschreven en worden in namen van variabelen soms hoofdletters gebruikt. De sleutelwoorden worden altijd met kleine letters geschreven.
Vetgedrukte sleutelwoorden in onderkast maken de code beter leesbaar. Bovenkast is hiervoor niet nodig. Moderne ontwikkelomgevingen voor digitale systemen en veel platte teksteditors gebruiken syntaxkleuren; sleutelwoorden zijn bijvoorbeeld blauw, commentaar rood, tekst purper en karakters donkergroen. Er is zodoende geen enkele reden hoofdletters te gebruiken bij sleutelwoorden.

Architecture

De entity legt de in- en uitgangsrelaties van een systeem vast. De architectuur legt de functionaliteit van het systeem vast. De functionaliteit kan altijd op veel verschillende manieren beschreven worden. Daarom kunnen er bij een entity meerdere architecturen horen.

Code 2.2 geeft een mogelijke architectuur voor de full-adder. Tabel 2.1 is hier met behulp van een case-statement geïmplementeerd. De code begint met het sleutelwoord **`architecture`** met daarna achtereenvolgens de naam van de architectuur, het sleutelwoord **`of`**, de naam van de entity waar deze architectuur bijhoort en het sleutelwoord **`is`**. Hierna volgt een declaratiedeel dat wordt afgesloten met het sleutelwoord **`begin`**. Deze architectuur heeft geen declaraties. Na **`begin`** start de feitelijke beschrijving van de full-adder en bestaat in dit geval uit één proces.

Code 2.2 : Een gedragsbeschrijving voor de full-adder met een case-statement

```
1  architecture gedrag of fulladder is
2  begin
3    p1: process (a,b,ci) is
4      variable v : std_logic_vector(2 downto 0);
5    begin
6      v := a & b & ci;
7      case v is
8        when "000"  =>   co <= '0';  s <= '0';
9        when "001"  =>   co <= '0';  s <= '1';
10       when "010"  =>   co <= '0';  s <= '1';
11       when "011"  =>   co <= '1';  s <= '0';
12       when "100"  =>   co <= '0';  s <= '1';
13       when "101"  =>   co <= '1';  s <= '0';
14       when "110"  =>   co <= '1';  s <= '0';
15       when "111"  =>   co <= '1';  s <= '1';
16       when others =>   co <= '0';  s <= '0';
17     end case;
18   end process p1;
19 end architecture gedrag;
```

Een proces is een essentieel onderdeel van VHDL. In een proces worden de toewijzingen sequentieel — na elkaar — doorlopen. In dit geval is dat de toekenning

op regel 6 een case-statement dat op de volgende regels verder gaat. In een architectuur mogen meerdere processen staan. Deze processen staan parallel naast elkaar en worden tegelijkertijd uitgevoerd. Hoofdstuk 3 bespreekt het simulatiemodel van VHDL en de rol die processen daarin spelen.
Het proces begint met **process** en direct daarachter een gevoeligheidslijst of *sensitivity list*. Voor **process** staat een dubbele punt en een label p1. De gevoeligheidslijst staat tussen ronde haken en bevat een of meer ingangssignalen gescheiden door komma's. Het proces wordt uitgevoerd als een van deze ingangssignalen van waarde verandert. Voor **begin** staan lokale declaraties en na **begin** staan de toewijzingen. Het proces wordt afgesloten met **end process**, het label en een puntkomma.
VHDL kent signalen en variabelen. De in- en uitgangen van de entity zijn signalen. In dit geval zijn er vijf signalen: a, b, ci, s en co. In het proces is een variabele v gedeclareerd. Het type is een std_logic_vector en representeert een digitale bus, die uit meerdere waarden bestaat.

De bits van de variable v zijn genummerd 2, 1 en 0. De index loopt van de busbreedte - 1 naar nul. VHDL kent naast **downto** ook **to**. Andere mogelijkheden zijn: 0 **to** 2, 1 **to** 3, 3 **downto** 1. In de digitale techniek is het gebruikelijk om bits te nummeren van hoog naar nul.

Op regel 6 worden aan de variabele v de waarden van de signalen a, b en ci toegekend. Dit is gedaan met het concatenatie-teken (&). Als de waarden van a, b en ci bijvoorbeeld '1', '1' en '0' zijn, krijgt v de waarde "110".
Het case-statement op regel 7 kent afhankelijk van de waarde van v een '1' of een '0' toe aan de uitgangssignalen s en co. Bij de toewijzing aan een variabele wordt het symbool := gebruikt en bij de toewijzing aan een signaal wordt <= gebruikt. VHDL maakt onderscheid tussen signalen en variabelen. Signalen vormen de verbindingen tussen de processen. Ze zijn vergelijkbaar met de fysieke verbindingen tussen geïntegreerde schakelingen op een printed circuit board en de connecties in de geïntegreerde schakelingen zelf. Net als bij echte signalen hebben de signalen in VHDL ook tijdsaspecten. Variabelen spelen in VHDL dezelfde rol als in andere programmeertalen. Hoofdstuk 3 behandelt de timing bij VHDL. Het verschil tussen signalen en variabelen wordt daar uitgebreid besproken.

In VHDL moet de opsomming bij een **case** volledig zijn. Drie std_logic-bits geven veel meer combinaties dan de genoemde "000" tot en met "111". Het type std_logic is een negenlaags logica. Naast de '0' en de '1' kent het zeven andere logische waarden, zoals de hoge impedantie 'Z'. Drie bits van negen niveaus geven 9^3=729 mogelijkheden. Daarom is de **when others** nodig.

Het case-statement begint met het sleutelwoord **case**, de naam van een signaal of variabele en een **is**. Het case-statement wordt afgesloten met **end case** en een puntkomma. De keuzes beginnen met het sleutelwoord **when**, de keuze en het symbool =>. Daarna volgen er een of meer toewijzingen. In dit geval zijn dat steeds twee signaaltoewijzingen. Het sleutelwoord **others** bij de laatste keuze betekent dat voor alle andere — nog niet genoemde — gevallen dit de keuze is.

Bestandsorganisatie

De **entity** en de **architecture** zijn basisonderdelen of primaire ontwerpeenheden (*primairy design units*). Elke primaire ontwerpeenheid mag bij VHDL in een apart bestand staan. In figuur 2.3 is de beschrijving van de full-adder gesplitst in twee bestanden: een bestand *fa_entity.vhd* met de entity en een tweede bestand *fa_gedrag.vhd* met de architectuur.
In de entity en in de architectuur wordt en de typen std_logic en std_logic_vector gebruikt. Deze typen zijn gedefinieerd in het package std_logic_1164 uit de IEEE-bibliotheek. Beide bestanden beginnen daarom met deze regels:

```
library ieee;
use ieee.std_logic_1164.all;
```

De eerste regel met het sleutelwoord **library** maakt de bibliotheek ieee toegankelijk. De tweede regel met het sleutelwoord **use** vertelt welke onderdelen uit de

bestand: fa_entity.vhd

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4  entity fulladder is
 5    port (
          ⋮
10    );
11  end entity fulladder;
```

bestand: fa_gedrag.vhd

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4  architecture gedrag of fulladder is
 5  begin
 6    p1: process (a,b,ci) is
          ⋮
20    end process p1;
21  end architecture gedrag;
```

Figuur 2.3 : **De entity en de architectuur in twee aparte VHDL-bestanden.** Links staat het bestand *fa_entity.vhd* met de entity van de full-adder. Rechts staat het bestand *fa_gedrag.vhd* met een architectuur van de full-adder.

bibliotheek toegankelijk zijn. In dit geval zijn dat alle (`all`) declaraties en definities uit het package `std_logic_1164`.
De entity en de architectuur kunnen ook in één bestand worden geplaatst. De twee regels voor het gebruik van de bibliotheek staan dan alleen aan het begin van het bestand; de declaraties en definities uit de bibliotheek zijn dan binnen het hele bestand bekend.

bestand: fa.vhd

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4  entity fulladder is
 5    port (
          ⋮
10    );
11  end entity fulladder;
12
13  architecture gedrag of fulladder is
14  begin
15    p1: process (a,b,ci) is
          ⋮
30    end process p1;
31  end architecture gedrag;
```

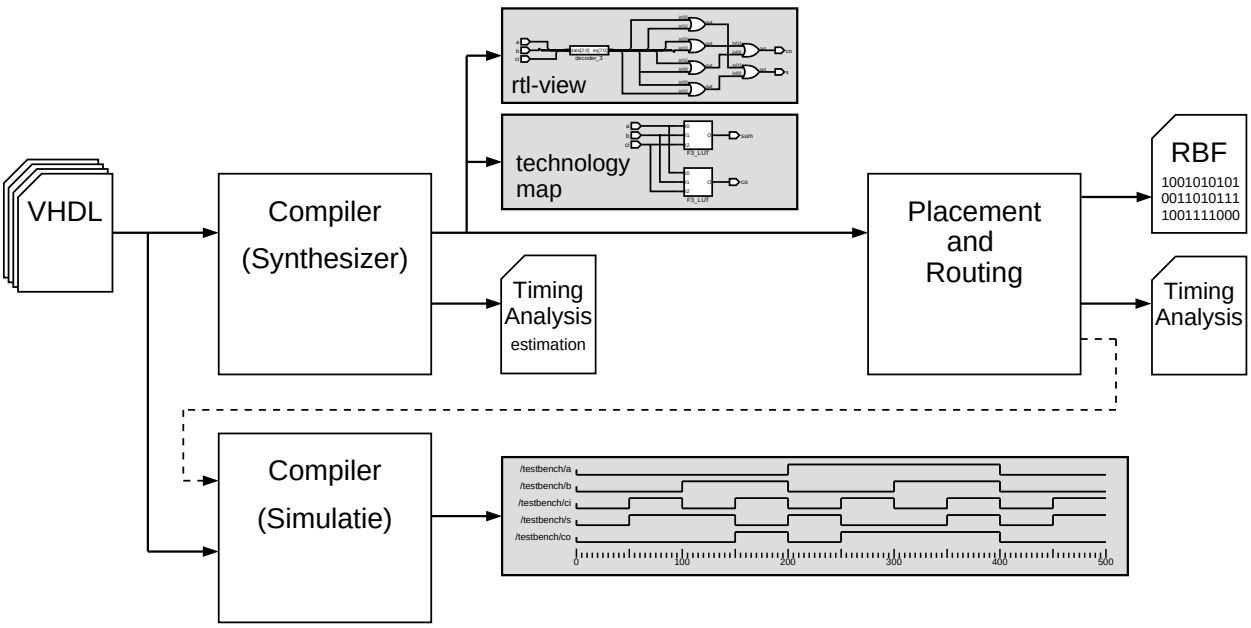
Figuur 2.4 : **De entity en de architectuur staan hier samen in één bestand.**

2.4 Het ontwerptraject

Bij hardwarebeschrijvingstalen worden altijd twee compilers gebruikt: één voor de simulatie en één voor de synthese. Het ontwerptraject met de twee compilers voor VHDL staat in figuur 2.5

De compiler voor de simulator maakt net als een compiler voor een microprocessor een binair objectbestand en linkt deze met de *kernel* van de simulator. Het simuleren is dan vergelijkbaar met het uitvoeren van een programma. Of beter het uitvoeren van een simulatie lijkt op het debuggen van software. Met de simulator kunnen breekpunten worden gezet en kan er door de code worden gestapt. Het resultaat van een simulatie is vaak een waveform of signaaldiagram. Dat is een grafische afbeelding van de signalen in de loop van de tijd. De veranderingen van de signalen zijn dan goed te volgen.

Een *kernel* is in de kern van een besturingssysteem of van een complex programma.



Figuur 2.5 : **Het ontwerptraject voor een digitaal systeem.** Voor het ontwerp zijn twee compilers nodig: één voor de synthese en één voor de simulatie.

De compiler voor de synthese wordt ook een synthesizer genoemd. De compiler vertaalt de VHDL in logische functies, die samengevoegd worden tot een grote digitale schakeling. De synthesizer vertaalt de code eerst naar een netwerkbeschrijving met logische functies en registers. Deze weergave noemt men de RTL-view of *register transfer logic view*, en bestaat uit registers, flipfloppen, RAM en ROM waarin gegevens worden opgeslagen en uit relatief eenvoudige logische bewerkingen, die op deze gegevens worden toegepast. Voorbeelden zijn de logische poorten, decoders, encoders, multiplexers, adders, buffers en vermenigvuldigers. De RTL-view is onafhankelijk van de gekozen technologie en onafhankelijk van het type bouwsteen.

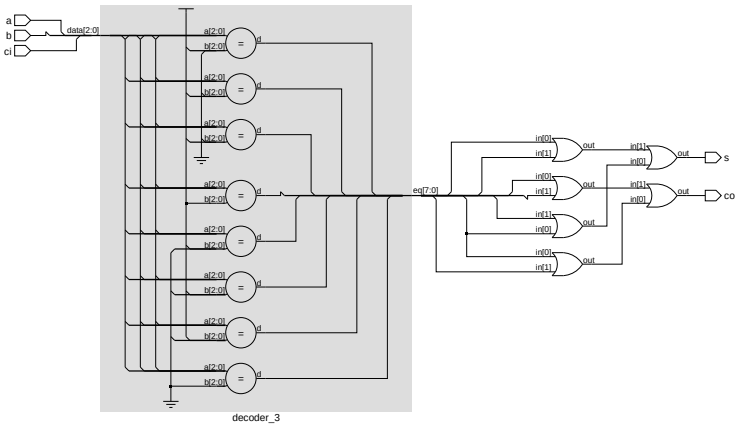
De synthesizer vertaalt de beschrijving ook naar de gebruikte technologie. Bij FPGA's worden de logica en de benodigde registers verdeeld over *lookup tables*, flipfloppen en geparametriseerde modulen. Het resultaat wordt een *technology map* genoemd. Tenslotte wordt het ontwerp geplaatst en bedraad. Het ontwerp wordt verdeeld over de beschikbare LUT's en flipfloppen en met elkaar verbonden. Vervolgens wordt dit omgezet naar een RBF-bestand met enen en nullen, waarmee de FPGA geprogrammeerd wordt.

RBF staat voor *raw bit file*.

Na de synthese en na het plaatsen en bedraden (*placement and routing*) is informatie over het tijdsgedrag beschikbaar in de vorm van een tijdsanalyse (*timing analysis*). Daarin staat onder andere wat de maximaal haalbare frequentie is en wat het meest ongunstige pad (*worst case path*) is. Na de plaatsing en bedrading is er tevens gedetailleerde informatie beschikbaar over het tijdsgedrag dat in de simulatie verwerkt kan worden.

2.5 Synthese

Het resultaat van de synthese van de beschrijving uit code 2.2 staat in figuur 2.6. De synthesizer — het programma dat de VHDL-code vertaalt naar hardware — herkent in de VHDL een decoder en een aantal logische poorten. Dit is de RTL-view. RTL staat voor (*register transfer logic*) en is een beschrijving van een digitaal systeem met registers en digitale logica. De RTL-view is een schema met flipflop-pen, registers, optellers, vermenigvuldigers, multiplexers, decoders en elementaire logische poorten.

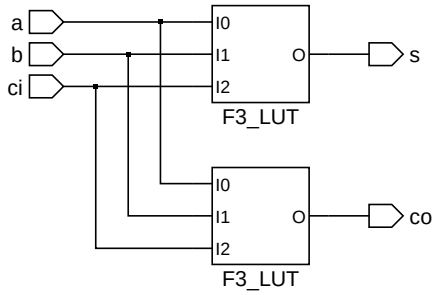


Figuur 2.6 : **De RTL-view van de full-adder met het case-statement.** Links staat een decoder, die uit acht drie-bits vergelijkers bestaat. Rechts staan zes OR-poorten, die acht mogelijkheden coderen naar een twee-bits code voor de uitgangen `s` en `co`.

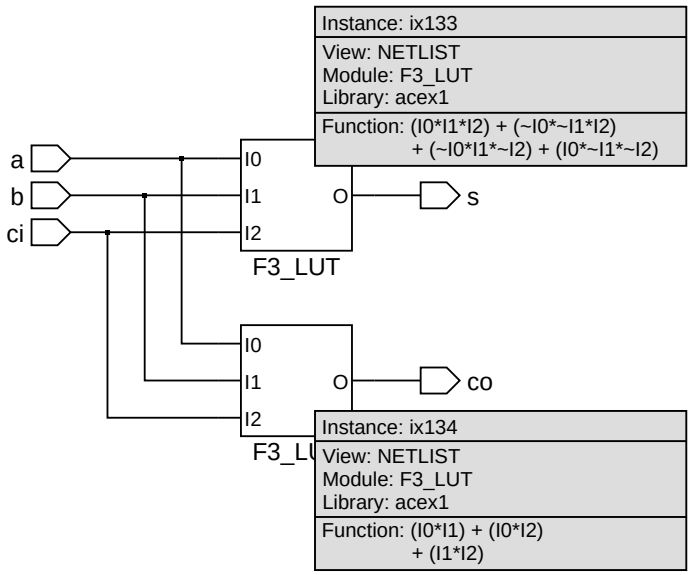
De decoder van figuur 2.6 beschrijft de selectiemogelijkheden van het case-statement uit code 2.2 en bestaat uit acht drie-bits vergelijkers. De zes OR-poorten zetten de acht keuzes om naar een twee-bits code. De meest significante bit van deze code is uitgang `co` en en het minst significante bit is uitgang `s`.

De RTL-view is technologie onafhankelijk. Een FPGA of CPLD bevat geen losse decoders en geen losse OR-poorten. Een FPGA heeft alleen de beschikking over LUT's (*lookup tables*) voor het maken van digitale logica. De synthesizer vertaalt de code ook naar logische functies die wel in de LUT's passen. De synthesizer optimaliseert daarbij het ontwerp. In figuur 2.7 staat het resultaat. Er zijn slechts twee LUT's nodig: een voor uitgang `s` en een voor uitgang `co`.

Figuur 2.7 toont alleen de LUT's. De logische vergelijkingen zijn niet zichtbaar. De viewer van de synthesizer kan de vergelijkingen van een LUT wel zichtbaar maken. Figuur 2.8 toont dezelfde twee LUT's met daaraan toegevoegd de inhoud van de beide LUT's.



Figuur 2.7 : De *technology-map* van de full-adder met het case-statement.



Figuur 2.8 : De *technology-map* van de full-adder met de inhoud van de LUT's.

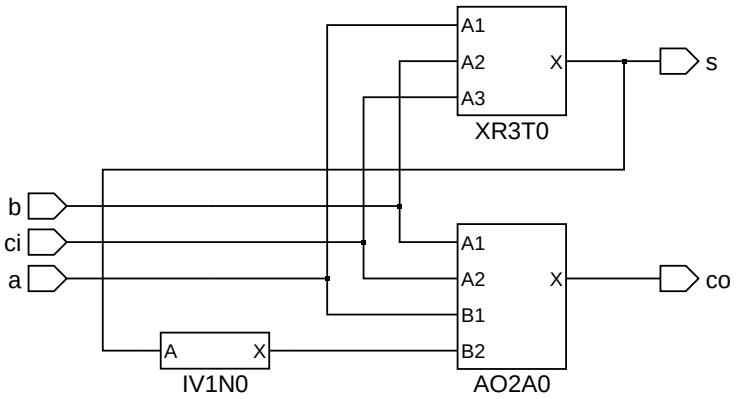
Er is een FPGA uit de ACEX-1K familie van Altera gebruikt. De bijbehorende *library* is *acex1*. De vergelijkingen zijn:

```
s  = (I0*I1*I2) + (~I0*~I1*I2) + (~I0*I1*~I2) + (I0*~I1*~I2)
co = (I0*I1) + (I0*I2) + (I1*I2)
```

Dit is geen VHDL. In deze notatie is een `*` de AND-functie, een `+` de OR-functie en de tilde (`~`) geeft aan dat het signaal geïnverteerd wordt. De ingangen `a`, `b` en `ci` zijn aan beide LUT's aangesloten op de ingangen `I0`, `I1` en `I2`. In VHDL zou dit als volgt geschreven kunnen worden:

```
s  <= (a and b and c) or ((not a) and (not b) and c) or
      ((not a) and b and (not c)) + (a and (not b) and (not c));
co <= (a and b) or (a and c) or (b and c);
```

Een bouwsteen uit een andere familie of een totaal andere technologie toont dezelfde RTL-view, maar geeft een hele andere *technology-map*. Figuur 2.9 geeft het resultaat voor een ASIC met een standaardceltechnologie.



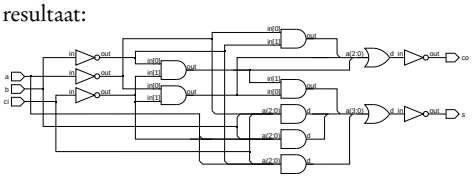
Figuur 2.9 : De *technology map* van de full-adder met de inhoud van de LUT's.

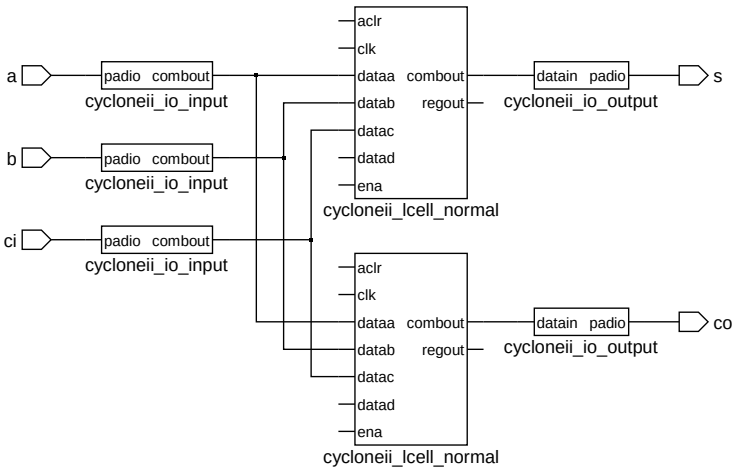
Het resultaat is een schakeling met een 3-input XOR, een and-or-functie en een inverter. In VHDL komt dit overeen met deze vergelijkingen:

```
y   <= a xor b xor ci ;
co  <= (b and ci) or (a and (not y));
s   <= y;
```

Ook met een andere familie van Altera kan het resultaat er anders uitzien. Figuur 2.10 toont het resultaat voor de Cyclone-II familie. Er zijn net als in figuur 2.7 weer twee blokken te zien. Alleen worden deze nu geen LUT maar *lcell* of *logical cell* genoemd. In figuur 2.11 staat meer informatie over de inhoud van de

De RTL-views in dit boek zijn gemaakt met Leonardo Spectrum van Mentor Graphics. Andere synthesizers zullen andere oplossingen geven.
Precision RTL van Mentor Graphics geeft bijvoorbeeld dit resultaat:

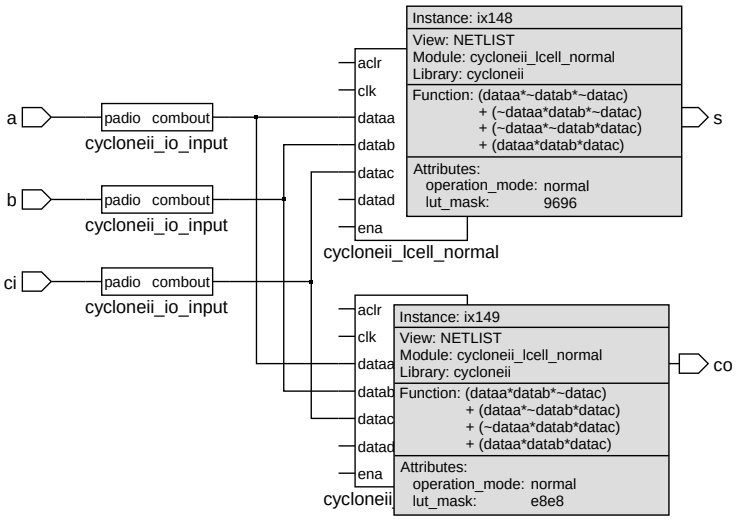




Figuur 2.10 : De *technology map* van de full-adder met de inhoud van de LUT's.

logische cellen. De vergelijkingen zijn nu niet geoptimaliseerd. Het is een rechtstreekse vertaling van de RTL-view naar de technologie.

Iedere technologie — dus alle PLD-families van alle fabrikanten — heeft zijn eigen interne structuur. De *technology map* zal daarom per familie vaak anders zijn. Bij eenzelfde VHDL-beschrijving is de RTL-view voor verschillende families identiek. De RTL-view is onafhankelijk van de gekozen technologie. Daarom wordt in dit boek bij syntheseresultaten meestal alleen de RTL-view gegeven.



Figuur 2.11 : De *technology map* van de full-adder met de inhoud van de LUT's.

2.6 Simulatie

Een essentieel onderdeel uit het ontwerptraject van een digitale systeem is de simulatie. Zonder simulaties is het heel moeilijk — zo niet onmogelijk — een correct ontwerp te maken. Het synthesetraject van VHDL tot een geprogrammeerde bouwsteen neemt flink wat tijd in beslag. Bovendien kan een geprogrammeerde bouwsteen alleen compleet getest worden in een testschakeling. Onderdelen uit het ontwerp zijn zonder speciale testschakelingen niet apart te testen.
Bij simulatie kan elk onderdeel uit het ontwerp apart getest worden. De simulatie van deelontwerpen nemen relatief weinig tijd in beslag. Simulaties van complexe ontwerpen vergen daarentegen veel tijd. In de praktijk blijkt een ontwerp, dat bij de simulatie correct functioneert, na de synthese een goed werkende bouwsteen te leveren. Alleen bij kritische ontwerpen, die het uiterste uit een programmeerbare bouwsteen halen, kan de synthese problemen geven die bij de simulatie niet te zien waren. Voor de ontwerper is het de uitdaging om een test te creëren, die de VHDL-code van het ontwerp zo goed mogelijk test.
Het gaat bij simulaties vooral om het functionele gedrag van het ontwerp. Het onderzoek naar het tijdsafhankelijk gedrag beperkt zich in eerste instantie tot het aantal klokslagen dat er voor bepaalde bewerkingen nodig is. Gegevens over exacte vertragingstijden zijn bij een FPGA- en ASIC-ontwerp pas beschikbaar na de synthese.

Test bench betekent letterlijk proefbank. Ook in de meeste Engelstalige literatuur over digitale systemen wordt *testbench* aan elkaar geschreven.

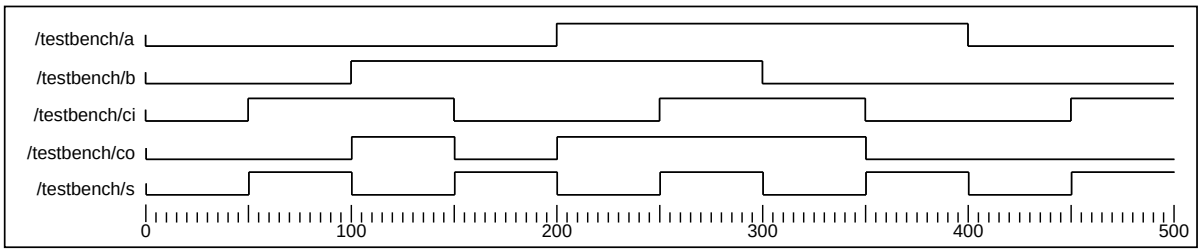
In code 2.3 staat de testbench voor de full-adder. Deze testbench is in VHDL geschreven en roept op regel 28 het te testen ontwerp als component aan. Het aanroepen van componenten wordt in paragraaf 2.7 besproken. Op regel 37 staat een proces `signal_generator` dat de signalen genereert waarmee het ontwerp — de *unit under test* — getest wordt. Dit proces maakt alle acht mogelijke combinaties van de drie ingangssignalen `a`, `b` en `ci`. Daarbij verandert er altijd maar één ingang tegelijk; de acht combinaties veranderen volgens een Gray-code.
Hoofdstuk 9 bespreekt het maken van een testbench en in hoofdstuk 3 komen de tijdsaspecten en het wait-statement aan de orde.

Code 2.3 : De testbench voor de full-adder.

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4  entity testbench is
 5  end entity testbench;
 6
 7  architecture tb_fulladder of testbench is
 8    component fulladder is
 9      port (
10        a  : in  std_logic;
11        b  : in  std_logic;
12        ci : in  std_logic;
13        co : out std_logic;
14        s  : out std_logic
15       );
16    end component fulladder;
17
18    for uut : fulladder use entity work.fulladder(gedrag);
19
20    signal a  : std_logic;
21    signal b  : std_logic;
22    signal ci : std_logic;
23    signal s  : std_logic;
24    signal co : std_logic;
25
26    constant DELAY : time := 50 ns;
27  begin
28    uut : fulladder port map (
29      a  => a,
30      b  => b,
31      ci => ci,
32      co => co,
33      s  => s
34    );
35
36
37    signal_generator: process is
38    begin
39      a  <= '0';
40      b  <= '0';
41      ci <= '0';
42      wait for DELAY;
43      ci <= '1';
44      wait for DELAY;
45      b  <= '1';
46      wait for DELAY;
47      ci <= '0';
48      wait for DELAY;
49      a  <= '1';
50      wait for DELAY;
51      ci <= '1';
52      wait for DELAY;
53      b  <= '0';
54      wait for DELAY;
55      ci <= '0';
56      wait for DELAY;
57      a  <= '0';
58      wait for DELAY;
59      ci <= '1';
60      wait;
61    end process signal_generator;
62  end architecture tb_fulladder;
```

Elk signaal verandert pas na delta-delay. Delta-delay geeft ook het aantal keer aan dat de processen geëvalueerd zijn. In hoofdstuk 3 over de timing wordt delta-delay uitgebreid besproken.

Het resultaat van de simulatie is in figuur 2.12 weergegeven als waveform en toont aan dat de uitgangen `co` en `s` de correcte waarden krijgen. In figuur 2.13 is de uitvoer van de simulatie op twee manieren in tekst weergegeven: één keer met delta-delay's en één keer zonder delta-delay's. De *listing* zonder delta-delay's is eenvoudig te lezen en te controleren. Bij de listing met delta-delay's en bij de waveform is niet in één oogopslag te zien dat het ontwerp correct is. Enerzijds komt dat doordat de ingangen met een Gray-code zijn gecodeerd. Het patroon van de uitgangen is daardoor tamelijk willekeurig. Anderzijds is het lastig te zien wanneer `a`, `b` en `c` hoog en laag zijn. De listing met delta-delay's bevat te veel gedetailleerde informatie. De listing zonder de delta-delay's is snel te interpreteren. De laatste twee bits stellen het aantal enen in de ingangscombinatie voor.



Figuur 2.12 : De waveform, of signaaldiagram, van de simulatie van de full-adder.

listing met delta-delay's

```
 ns     /testbench/a
  delta   /testbench/b
           /testbench/ci
               /testbench/co
                  /testbench/s
  0  +0            U U U   U U
  0  +1            0 0 0   0 0
 50  +1            0 0 1   0 0
 50  +2            0 0 1   0 1
100  +1            0 1 1   0 1
100  +2            0 1 1   1 0
150  +1            0 1 0   1 0
150  +2            0 1 0   0 1
200  +1            1 1 0   0 1
200  +2            1 1 0   1 0
250  +1            1 1 1   1 0
250  +2            1 1 1   1 1
300  +1            1 0 1   1 1
300  +2            1 0 1   1 0
350  +1            1 0 0   1 0
350  +2            1 0 0   0 1
400  +1            0 0 0   0 1
400  +2            0 0 0   0 0
450  +1            0 0 1   0 0
450  +2            0 0 1   0 1
```

listing zonder delta-delay's

```
 ns /testbench/a
      /testbench/b
       /testbench/ci
           /testbench/co
              /testbench/s
  0            0 0 0   0 0
 50            0 0 1   0 1
100            0 1 1   1 0
150            0 1 0   0 1
200            1 1 0   1 0
250            1 1 1   1 1
300            1 0 1   1 0
350            1 0 0   0 1
400            0 0 0   0 0
450            0 0 1   0 1
```

Figuur 2.13 : Het simulatieresultaat van figuur 2.12 in tabelvorm. Links staat het resultaat met delta-delay's en hierboven zonder.

2.7 Alternatieve beschrijvingen voor de full-adder

Bij een entity hoort minimaal één architectuur, maar het mogen er ook meer zijn. Dit geeft de mogelijkheid om van één entity zowel een abstracte gedragsbeschrijving als een structuurbeschrijving op componentniveau te maken. Het functionele gedrag van een complex systeem wordt vastgelegd met een aantal gedragsbeschrijvingen voor de verschillende onderdelen waaruit het systeem is opgebouwd. Structuurbeschrijvingen met nauwkeurige timingparameters leggen het tijdsafhankelijk gedrag vast. Er zijn drie soorten architectuurbeschrijvingen te onderscheiden:

- de gedragsbeschrijving,
- de dataflowbeschrijving,
- de structuurbeschrijving.

Een gedragsbeschrijving legt vast wat het systeem achtereenvolgens moet doen en bestaat vaak uit één proces. Een dataflowbeschrijving beschrijft ruw hoe het systeem is opgebouwd en bestaat in het algemeen uit een verzameling (parallelle) signaaltoewijzingen. De structuurbeschrijving legt vast uit welke componenten het systeem is opgebouwd en bestaat dan ook uit een verzameling component-aanroepen.

Alternatieve gedragsbeschrijving voor de full-adder met `with select`

De gedragsbeschrijving van code 2.2 bevat een proces met een case-statement. In code 2.4 staat een versie met een *selected signal assignment*. Deze toewijzing begint op regel 8 en eindigt bij de puntkomma op regel 17. Afhankelijk van de waarde van `x` wordt een andere 2-bits vector aan het signaal `sum` toegekend.
Het 3-bits selectiesignaal `x` is de samenvoeging van de drie ingangssignalen `a`, `b` en `ci`. Dit wordt niet met de concatenatie uit code 2.2 gedaan, maar door de bits afzonderlijk toe te kennen. Een index tussen ronde haken achter de vector geeft de betreffende bit. Op regel 5 wordt signaal `a` aan de meest significante bit van `x` toegekend. Op regel 18 wordt aan `co` de meest significante bit van `sum` toegekend en op regel 19 krijgt `s` de waarde van de minst significante bit van `sum`.

Code 2.4 : Een gedragsbeschrijving voor de full-adder met een `with-select`.

```
 1  architecture gedrag2 of fulladder is
 2    signal x   : std_logic_vector(2 downto 0);
 3    signal sum : std_logic_vector(1 downto 0);
 4  begin
 5    x(2) <= a;
 6    x(1) <= b;
 7    x(0) <= ci;
 8    with x select sum <=
 9      "00" when "000",
10      "01" when "001",
11      "01" when "010",
12      "10" when "011",
13      "01" when "100",
14      "10" when "101",
15      "10" when "110",
16      "11" when "111",
17      "00" when others;
18    co <= sum(1);
19    s  <= sum(0);
20  end architecture gedrag2;
```

Het *selected signal assignment* begint met de sleutelwoorden **`with`** en **`select`**; met daar tussenin het signaal dat de voorwaarde geeft. Na de signaaltoewijzing `<=` volgt een lijst met alternatieven gescheiden door komma's. Deze alternatieven bestaan uit de toe te kennen waarde, het sleutelwoord **`when`** en de betreffende conditie. Het laatste alternatief op regel 17 wordt uitgevoerd als het niet één van de voorafgaande alternatieven is.
De synthese geeft een RTL-view — mits de technologie ook ACEX-1K is — die volledig identiek is aan figuur 2.6. De full-adder bestaat wederom uit een decoder en zes OR-poorten. Ook de *technology map* is exact gelijk aan figuur 2.7 en figuur 2.8.

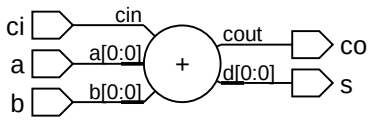
Alternatieve gedragsbeschrijving voor de full-adder met optelfunctie

De beschrijvingen van code 2.2 en code 2.4 zijn letterlijke vertalingen van tabel 2.1. Een andere interpretatie van de tabel kan tot een andere gedragsbeschrijving leiden.

De uitgangen `co` en `s` vormen samen een 2-bits vector, die de som representeert van de drie ingangen. De beschrijving `gedrag3` uit code 2.5 gebruikt dit idee om het gedrag vast te leggen. Het interne signaal `sum` is de som van de interne signalen `ia`, `ib` en `ic`. De interne signalen zijn van het type `unsigned` en twee bits breed. Het type `std_logic_vector` is niet geschikt, omdat voor dit type het optellen niet gedefinieerd is. De meest significante bit van `sum` is de carry-out `co` en de minst significante bit is `s`.

Code 2.5 : Een gedragsbeschrijving voor de full-adder gebaseerd op de optelfunctie.

```
1  architecture gedrag3 of fulladder is
2    signal ia,ib,ic,sum : unsigned(1 downto 0);
3  begin
4    ia  <= '0' & a;
5    ib  <= '0' & b;
6    ic  <= '0' & ci;
7    sum <= ia + ib + ic;
8    co  <= sum(1);
9    s   <= sum(0);
10 end architecture gedrag3;
```



Figuur 2.14 : RTL-view na synthese code 2.5.

Het type `unsigned` is gedefinieerd in het package `numeric_std` uit de IEEE-bibliotheek. Het bestand met de beschrijving van code 2.5 moet beginnen met deze regels:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

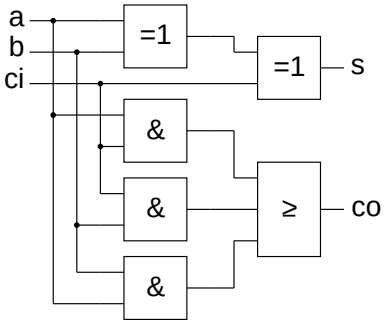
Hoofdstuk 10 bespreekt het package `numeric_std` en andere belangrijke packages uit de IEEE-bibliotheek.

Signaal `a` is van het type `std_logic`. Door `a` met het concatenatie-teken `&` aan `'0'` toe te voegen, ontstaat een 2-bits vector. Het is nodig om voor de rekenkundige bewerking de operanden de juiste breedte te geven. Het rechterlid van de toewijzing op regel 7 moet net als het linkerlid 2-bits breed zijn.

De RTL-view staat in figuur 2.14. De synthesizer herkent in de code een 1-bits opteller met carry-in en carry-out. De technology-map is volledig identiek aan het resultaat van figuur 2.7 en figuur 2.8.

Dataflowbeschrijvingen voor de full-adder

Een ervaren ontwerper van digitale elektronica zal de full-adder op basis van twee optelfuncties of met een case-statement onhandig vinden. Iedereen, die goed bekend is met digitale techniek, weet dat een full-adder gemaakt kan worden met het schema van figuur 2.15. De dataflowbeschrijving van code 2.6 representeert het schema van figuur 2.15 en bestaat uit twee *concurrent signal assignments*. Een *concurrent signal assignment* is een signaaltoewijzing, die direct is de architectuur staat. Deze twee toewijzingen worden parallel, dus tegelijkertijd, uitgevoerd.



Figuur 2.15 : Digitaal schema van full-adder.

Code 2.6 : De dataflowbeschrijving van figuur 2.15.

```
1  architecture dataflow1 of fulladder is
2  begin
3    s  <= (a xor b) xor ci;
4    co <= (a and b) or (b and ci) or (a and ci);
5  end architecture dataflow1;
```

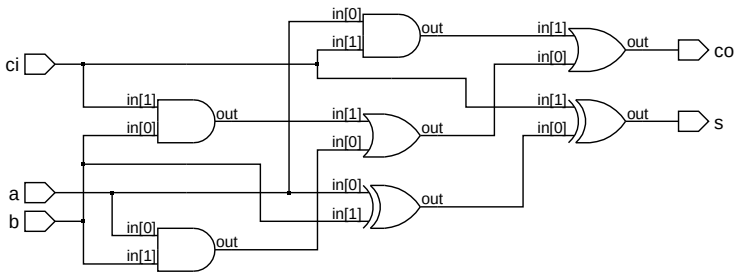
Tabel 2.2 : De logische functies bij VHDL. In de eerste twee kolommen staan de operanden en in de volgende zes kolommen de uitkomst voor de zes logische functies.

a	b	and	nand	or	nor	xor	xnor
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

Tabel 2.3 : De logische functie `not`. De waarde van de ingang wordt door **`not`** geïnverteerd.

a	not
0	1
1	0

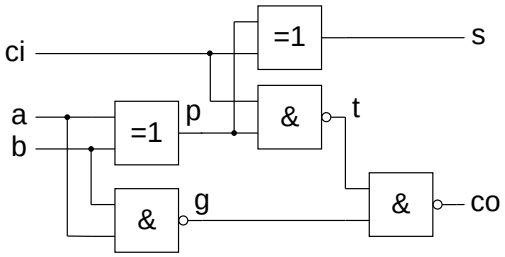
VHDL kent voor `std_logic` en `std_logic_vector` een aantal logische functies: **`and`**, **`nand`**, **`or`**, **`nor`**, **`xor`** en **`xnor`**. In tabel 2.2 staan de waarheidstabellen van deze zes functies. Deze functies hebben de allerlaagste prioriteit en worden allemaal van links naar rechts geëvalueerd. De logische functie **`not`** inverteert het ingangssignaal. De waarheidstabel van **`not`** staat in tabel 2.2. Deze functie **`not`** heeft juist de hoogste prioriteit. Ronde haakjes voorkomen onduidelijkheden in de bewerkingsvolgorde.



Figuur 2.16 : RTL-view van code2.6.

De RTL-view van de synthese staat in figuur 2.16. De technology-map bestaat weer uit twee LUT's en is identiek met figuur 2.7 en figuur 2.8.

Een zeer ervaren ontwerper van digitale elektronica weet dat de full-adder ook gerealiseerd kan worden met twee XOR-functies en drie NAND-functies; in figuur 2.17 staat deze schakeling en in code 2.7 de VHDL-beschrijving. De architectuur bestaat uit vijf *concurrent signal assignments* en kent drie interne signalen `t`, `p` en `g`. De volgorde van de signaaltoewijzingen doet er niet. Het zijn vijf aparte processen die gelijktijdig worden uitgevoerd.

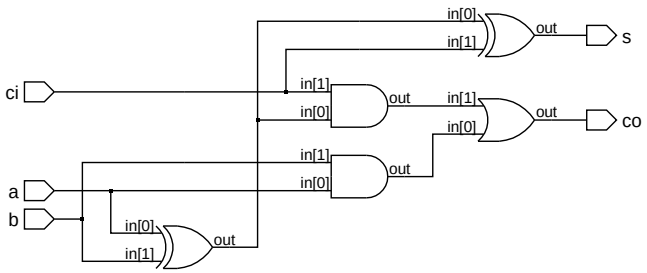


Figuur 2.17 : Geoptimaliseerde versie van full-adder.

Code 2.7 : De dataflowbeschrijving van figuur 2.17.

```
1  architecture dataflow2 of fulladder is
2    signal p, g, t : std_logic;
3  begin
4    s  <= p xor ci;
5    co <= g nand t;
6    t  <= p nand ci;
7    p  <= a xor b;
8    g  <= a nand b;
9  end architecture dataflow2;
```

De RTL-view van het syntheseresultaat staat in figuur 2.18. De NAND-NAND-structuur uit de VHDL-beschrijving is hier vertaald naar een AND-OR. Ook in dit geval is de technology-map weer identiek aan de eerdere beschrijvingen.



Figuur 2.18 : De RTL-view van de synthese van code 2.7.

Structuurbeschrijvingen van de full-adder

De dataflowbeschrijving `dataflow1` uit code 2.6 is opgebouwd uit drie AND's, twee XOR's en één OR en kan dus ook met deze logische poorten worden gerealiseerd. Een FPGA is opgebouwd uit LUT's en flipfloppen. Een standaardcel ASIC bestaat uit logische poorten, zoals ook de technology-map van figuur 2.9 laat zien. Een FPGA-ontwerp kan direct in LUT's en flipfloppen worden gemaakt en een ASIC-ontwerp direct in logische poorten. Voor een VHDL-ontwerp is dan wel een bibliotheek nodig met de betreffende poorten.

Het type `time` voor het beschrijven van tijd kent de eenheden `fs`, `ps`, `ns`, `us`, `ms`, `sec`, `min` en `hr`.

Code 2.8 geeft de beschrijvingen van 2-input AND, een 2-input XOR en een 3-input OR. De beschrijvingen leggen niet alleen het functionele gedrag vast, maar hebben ook een dynamisch gedrag. Elke logische bewerking heeft een vertraging van enkele nanoseconden. Dit is vastgelegd met de **`after`**-clausules. De vertragingstijden bestaan uit een getal en een eenheid. De eenheid is hier nanoseconde (`ns`). Tussen het getal en de eenheid moet een spatie staan.

De structuurbeschrijving van de full-adder figuur 2.19 representeert het schema van figuur 2.15 op basis van de logische poorten uit code 2.8. Er zijn vier interne signalen `w1`, `w2`, `w3` en `w4`. De logische poorten worden direct aangeroepen. Deze vorm van aanroep noemt men *design entity instantiation*. De aanroep bestaat uit een label, een dubbele punt, het sleutelwoord **`entity`**, de binding en een

Code 2.8 : Een bibliotheek met drie logische poorten. De entity's staan links en de bijbehorende architecturen rechts.

```
1  entity and2 is
2    port(i1, i2: in std_logic; o1: out std_logic);
3  end entity and2;
4
5
6  entity or3 is
7    port(i1, i2, i3: in std_logic; o1: out std_logic);
8  end entity or3;
9
10
11 entity nand2 is
12   port(i1, i2: in std_logic; o1: out std_logic);
13 end entity nand2;
14
15
16 entity xor2 is
17   port(i1, i2: in std_logic; o1: out std_logic);
18 end entity xor2;
19
```

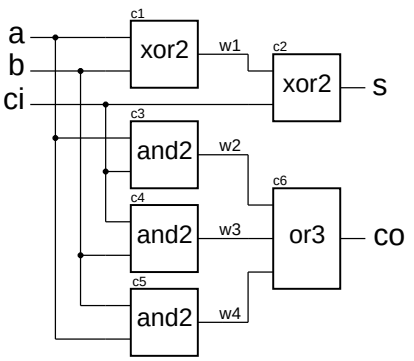
```
21 architecture dataflow of and2 is
22 begin
23     o1 <= i1 and i2 after 5 ns;
24 end architecture dataflow;
25
26 architecture dataflow of or3 is
27 begin
28     o1 <= i1 or i2 or i3 after 6 ns;
29 end architecture dataflow;
30
31 architecture dataflow of nand2 is
32 begin
33     o1 <= i1 xor i2 after 4 ns;
34 end architecture dataflow;
35
36 architecture dataflow of xor2 is
37 begin
38     o1 <= i1 xor i2 after 7 ns;
39 end architecture dataflow;
```

zogenoemde *port map*. De binding begint met de naam van de bibliotheek, een punt, de entitynaam en tussen ronde haken de naam van de architectuur. De naam van de bibliotheek is `work`. Dat is de werkbibliotheek waarin de compiler de gecompileerde bestanden plaatst.

De signalen waarmee de poort verbonden is, staan na de sleutelwoorden **`port map`** tussen twee ronde haken. Ingang `ci` van de full-adder en het interne signaal `w1` zijn verbonden met de ingangen van `i1` en `i2` van component `c2`. De uitgang `o1` van `c2` is verbonden met de uitgang `s` van de full-adder.

Code 2.9 : De structuurbeschrijving van figuur 2.19.

```
1  architecture structuur of fulladder is
2    signal w1, w2, w3, w4 : std_logic;
3  begin
4    c1: entity work.xor2(dataflow) port map (a, b, w1);
5    c2: entity work.xor2(dataflow) port map (w1, ci, s);
6    c3: entity work.and2(dataflow) port map (a, ci, w2);
7    c4: entity work.and2(dataflow) port map (b, ci, w3);
8    c5: entity work.and2(dataflow) port map (a, b, w4);
9    c6: entity work.or3(dataflow)  port map (w2, w3, w4, co);
10 end architecture structuur;
```



Figuur 2.19 : Een full-adder opgebouwd uit componenten van code 2.8.

Meestal wordt bij een hiërarchisch ontwerp in plaats van *design entity instantiation* gebruik gemaakt van *component instantiation*. De entity wordt dan expliciet als component aangeroepen. Code 2.10 bevat de structuurbeschrijving van de full-adder uit figuur 2.19. Voor een structuurbeschrijving met expliciete componenten

zijn drie zaken nodig:

- *Component declaration*
  De componentdeclaratie toont de compiler hoe de te gebruiken component er uit ziet. Het vertelt de compiler welke in- en uitgangen de te gebruiken component heeft. De componentdeclaratie is vergelijkbaar met de prototype bij de programmeertaal 'C'.
- *Component configuration*
  De componentconfiguratie geeft aan welke entity, en welke architectuur er gebruikt wordt. Dit onderdeel mag worden weggelaten. Er kan een standaardconfiguratie of een expliciete *configuration specification* gebruikt worden. Bij de standaardconfiguratie moet dan wel een entity met dezelfde naam in de werkbibliotheek aanwezig zijn. Deze entity met de meest recent gecompileerde architectuur wordt dan met deze component verbonden.
- *Component instantiation*
  De componentinstantiatie is de feitelijke aanroep van de component in de body van de architectuur.

In code 2.10 staan op de regels 3 tot en met 13 drie componentdeclaraties. Een componentdeclaratie is identiek aan de declaratie van de entity, alleen is het sleutelwoord **`entity`** vervangen door het sleutelwoord **`component`**.
De componentconfiguraties staan op regel 16 tot en met 18. Tussen de **`for`** en de dubbele punt staan de labels van de componenten. De labels worden gescheiden door komma's en het sleutelwoord **`all`** geeft aan dat de configuratie voor alle componenten geldt. Na de dubbele punt staat de naam van de component. Achter het sleutelwoord **`use`** staat de binding; deze bestaat uit de naam van de bibliotheek, een punt, de entitynaam en tussen ronde haken de naam van de architectuur.

De instantiatie of de aanroep van de component bestaat uit een label, een dubbele punt, de component en een port map.
Er zijn twee manieren om de actuele parameters met de formele parameters te associëren, namelijk met de posities van de parameters en met de namen van de parameters. In code 2.9 en in code 2.10 zijn de signalen geassocieerd via de posities. Bij de port map staan de parameters achter elkaar in een lijst. Het eerste signaal `a` uit de lijst is verbonden met het eerste signaal `i1` uit de parameterlijst van de component. Het tweede signaal `b` is verbonden met het tweede signaal `i2` uit de parameterlijst van de component en het derde signaal `w1` met het derde signaal `o1` van component `and2`.

Een parameter wordt ook een argument genoemd. Dit boek gebruikt altijd het begrip parameter. Sommigen gebruiken parameter alleen voor een formele parameter en argument alleen voor een actuele parameter.

In code 2.3 is de associatie via de naam van de parameters gebruikt. Het interne signaal `clk` van de testbench is verbonden met het ingangssignaal `clk` van de component `uut`. Regel 23 van code 2.10 had met een naamassociatie geschreven kunnen worden als:

```
c1: xor2 port map (
  i1 => a,
  i2 => b,
  o1 => w1);
```

Het signaal `a` wordt met ingang `i1`, signaal `b` met ingang `i2` en signaal `w1` met uitgang `o1` van de component verbonden. De signalen `a`, `b` en `w1` zijn de actuele parameters en de signalen `i1`, `i2`, `o1` zijn formele parameters.

Code 2.10 : Structuurbeschrijving van full-adder met component instantiatie.

```
1  architecture structuur of fulladder is
2  -- component declarations:
3    component and2 is
4      port (i1, i2 : in std_logic; o1 : out std_logic);
5    end component and2;
6
7    component or3 is
8      port (i1, i2, i3 : in std_logic; o1 : out std_logic);
9    end component or3;
10
11   component xor2 is
12     port (i1, i2 : in std_logic; o1 : out std_logic);
13   end component xor2;
14
15 -- component configurations:
16   for all   : and2 use entity work.and2(dataflow);
17   for c6    : or3  use entity work.or3(dataflow);
18   for c1,c2 : xor2 use entity work.xor2(dataflow);
19
20   signal w1, w2, w3, w4 : std_logic;
21 begin
22 -- component instantiations:
23   c1: xor2 port map (a, b, w1);
24   c2: xor2 port map (w1, ci, s);
25   c3: and2 port map (a, ci, w2);
26   c4: and2 port map (b, ci, w3);
27   c5: and2 port map (a, b, w4);
28   c6: or3  port map (w2, w3, w4, co);
29 end architecture structuur;
```

Commentaar toevoegen kan in VHDL met twee streepjes. Alles dat op een regel na -- komt wordt door de compiler genegeerd.

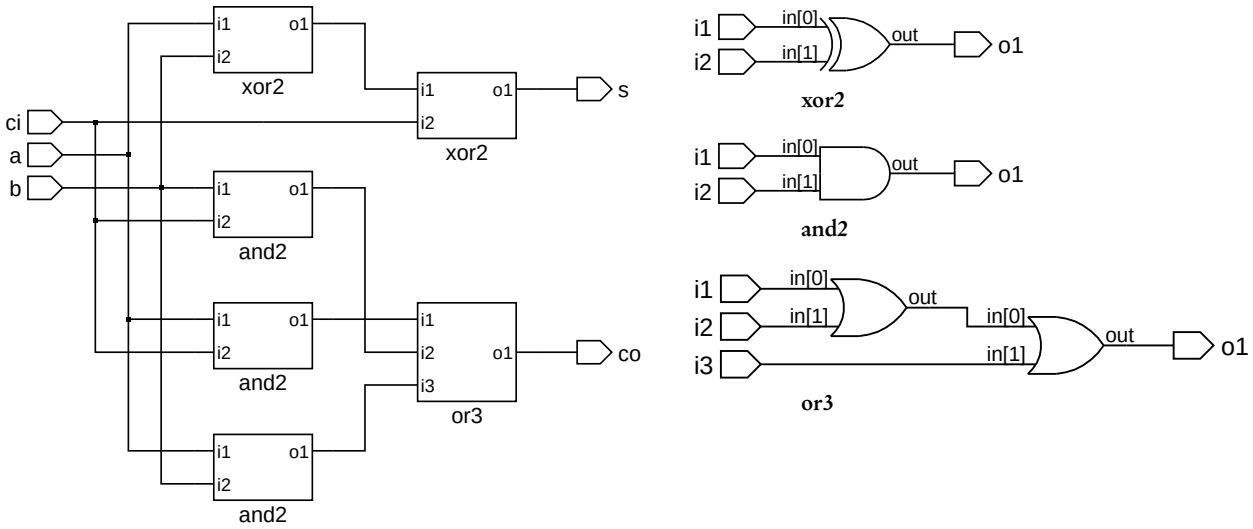
De volgorde van de signaalbindingen doet er niet toe. De onderstaande associatie geeft precies dezelfde binding:

```
c1: xor2 port map (
  o1 => w1,
  i1 => a,
  i2 => b);
```

Het voordeel van de associatie met behulp van de naam is dat de volgorde er niet toe doet. Vooral bij lange parameterlijsten is dat handig. Digitale onderdelen hebben vaak veel aansluitingen. Het is dan veel eenvoudiger om de signalen via de naam met elkaar te verbinden. De associatie met behulp van de positie in de parameterlijst levert een meer compacte beschrijving op.

Associaties met de naam kan gebruikt worden bij de aanroep van entity's, componenten, functies en procedures. Functies en procedures worden meestal geassocieerd via de posities. Bij entity's en componenten wordt vaak de associatie met de naam gebruikt.

De structuurbeschrijvingen van code 2.9 en 2.10 kunnen ook gesynthetiseerd worden. De RTL-view staat in figuur 2.20 en komt — zoals te verwachten is — volledig overeen met het schema van figuur 2.19. De hiërarchie van het syntheseresultaat is hetzelfde als dat van de VHDL-beschrijving. Het gesynthetiseerde netwerk is opgebouwd uit de gesynthetiseerde beschrijvingen van de logische poorten uit figuur 2.8. De synthese geeft een technology-map die identiek is aan de map van figuur 2.7 en 2.8.



Figuur 2.20 : De RTL-view van de synthese van code 2.9 en code 2.10. Links staat de RTL-view van de full-adder. Deze is opgebouwd uit zes componenten. Rechts staan de drie RTL-views van de betreffende componenten.

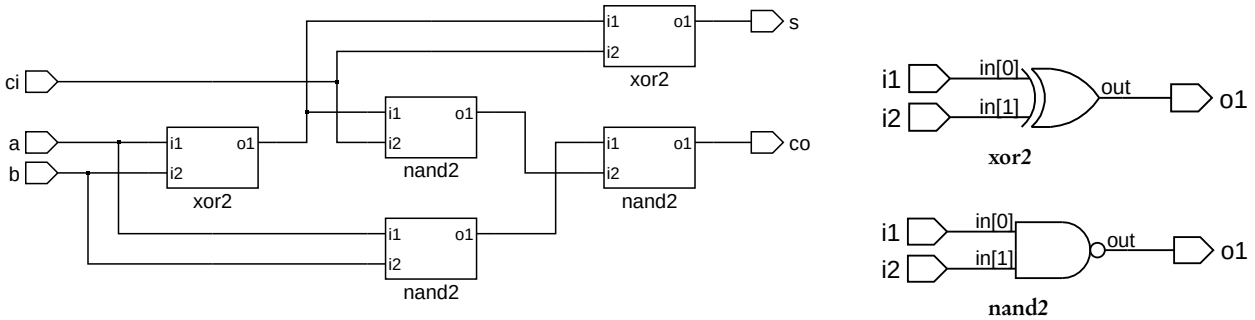
In code 2.11 staat het equivalent van de geoptimaliseerde dataflowbeschrijving met behulp van de logische poorten uit code 2.7. Deze beschrijving heeft een aparte configuratie. De configuratie begint met het **`configuration`**, de naam van de configuratie, **`of`**, de entitynaam en **`is`** en eindigt met **`end configuration`**, de naam van de configuratie en een puntkomma. De beschrijving tussen de regels 23 en 30 geldt voor de architectuur `structuur`. De syntax van de **`for`**-clausule op regel 24 en op regel 27 is gelijk aan de **`for`**-clausules van code 2.10. Alleen worden deze nu met een **`end for`**; afgesloten in plaats van een enkele puntkomma.

Code 2.11 : Alternatieve structuurbeschrijving met aparte configuratie.

```
1  architecture structuur of fulladder is
2  -- component declarations:
3    component nand2 is
4      port (i1, i2 : in std_logic; o1 : out std_logic);
5    end component nand2;
6
7    component xor2 is
8      port (i1, i2 : in std_logic; o1 : out std_logic);
9    end component xor2;
10
11   signal p, g, t : std_logic;
12 begin
13 -- component instantiations:
14   c1: xor2  port map (a, b, p);
15   c2: xor2  port map (p, ci, s);
16   c3: nand2 port map (a, b, g);
17   c4: nand2 port map (p, ci, t);
18   c5: nand2 port map (g, t, co);
19 end architecture structuur;
20
```

```
22 configuration cfg of fulladder is
23   for structuur
24     for c1,c2 : xor2 use entity
25                 work.xor2(dataflow);
26     end for;
27     for c3,c4,c5 : nand2 use entity
28                 work.nand2(dataflow);
29     end for;
30   end for;
31 end configuration cfg;
```

De synthese van deze alternatieve structuurbeschrijving geeft de RTL-view van figuur 2.21 en komt volledig overeen met het schema van figuur 2.20. De hiërarchie van het syntheseresultaat is hetzelfde als dat van de originele VHDL-beschrijving met een zelfde structuur. Het bestaat uit een netwerk van twee xor2's en drie nand2's.



Figuur 2.21 : De RTL-view van de synthese van code 2.11. Links staat de RTL-view van de full-adder en daarnaast staan de twee RTL-views van de componenten waaruit de full-adder is opgebouwd.

De synthese geeft een technology-map die weer gelijk is aan de technology-map van figuur 2.7 en 2.8.

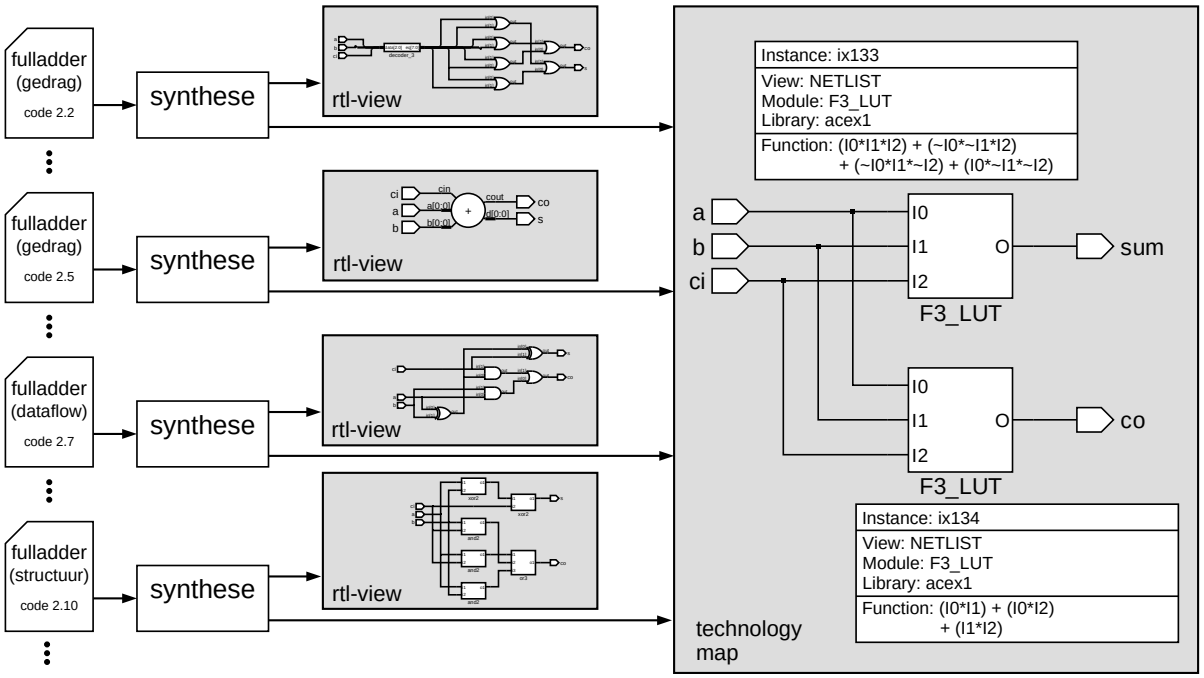
2.8 Evaluatie alternatieven beschrijvingen full-adder

In de vorige paragraaf zijn twee alternatieve gedragsbeschrijvingen, twee dataflowbeschrijvingen en drie structuur-beschrijvingen voor de full-adder besproken. Deze beschrijvingen zijn heel verschillend en bedacht vanuit een ander gezichtspunt. De RTL-views na de synthese zijn bijna allemaal anders. De RTL-view bevat hetzelfde concept als de oorspronkelijke VHDL-beschrijving. De beschrijving met een opteller, zoals code 2.5, geeft in de RTL-view eveneens een opteller. Een beschrijving met een case-statement, zoals code 2.2, levert een oplossing met een decoder op.

Ondanks de verschillende VHDL-beschrijvingen en de totaal andere RTL-views is de technology-map bij een zelfde technologie voor alle beschrijvingen identiek. Het maakt uiteindelijk niet uit welke beschrijving gebruikt wordt, de oplossing voor de full-adder bestaat altijd uit deze twee vergelijkingen:

```
s  = (I0*I1*I2) + (~I0*~I1*I2) + (~I0*I1*~I2) + (I0*~I1*~I2)
co = (I0*I1) + (I0*I2) + (I1*I2)
```

Het heeft bij kleine, combinatorische schakelingen weinig zin veel tijd te stoppen in een discussie over welke beschrijving beter is. Een gedragsbeschrijving legt vast wat het gedrag van een entity is. Dataflow- en structuurbeschrijving zijn oplossingen, die suggereren hoe het gemaakt zou kunnen worden. Het vastleggen van het gedrag is meestal eenvoudiger dan het beschrijven van een implementatie. In het algemeen kan gezegd worden dat gedragsbeschrijvingen eenvoudiger zijn en dat deze beschrijvingen het meest geschikt zijn voor het maken van een ontwerp. Gedrags-, dataflow- en structuurbeschrijvingen mogen ook worden gemengd. In code 2.12 staat een beschrijving van een full-adder met dataflow- en gedragsken-



Figuur 2.22 : Overzicht syntheseresultaat full-adder. Verschillende VHDL-beschrijvingen geven heel andere RTL-views, maar de technology-map is voor allemaal gelijk.

merken. De logische bewerkingen op regel 4 en op regel 5 zijn dataflowbewerkingen. Het proces `mx` beschrijft het gedrag van uitgang `co`. Deze uitgang volgt ingang `ci` als signaal `x` hoog is en ingang `a` als de waarde van `x` laag is. Het proces bevat een if-statement. De voorwaarde waar het if-statement op test, staat tussen **`if`** en **`then`**. Als de voorwaarde waar is, wordt regel 9 uitgevoerd. Als de voorwaarde niet waar is, wordt de regel na de **`else`** uitgevoerd. Het if-statement wordt afgesloten met **`end if`**;.

Code 2.12 : Beschrijving van full-adder met dataflow- en gedragselementen.

```
1  architecture mixed of fulladder is
2    signal x : std_logic;
3  begin
4    x  <= a xor b;
5    s  <= x xor ci;
6    mx: process (x,a,ci) is
7    begin
8      if x='1' then
9        co <= ci;
10     else
11       co <= a;
12     end if;
13   end process mx;
14 end architecture mixed;
```

Het onderscheid tussen vooral dataflow- en gedragselementen is erg arbitrair. Proces `mx` uit code 2.12 beschrijft een multiplexer. Veel ontwerpers zien in code 2.12

drie databewerkingen, namelijk twee xor-functies en een multiplexer; zij zullen het eventueel een dataflowbeschrijving noemen.
Alle beschrijvingen, die een ontwerper maakt, leggen het gedrag vast. De synthesizer maakt er een technology-map van die verder verwerkt wordt door de ontwikkelomgeving tot een bestand met enen en nullen waarmee de FPGA of CPLD geprogrammeerd kan worden. In het algemeen zullen in dit boek daarom alle VHDL-beschrijvingen gedragsbeschrijving worden genoemd. De ontwerper legt immers het gedrag vast. Dat de beschrijving ook dataflow of structuurelementen bevat, doet er niet toe.

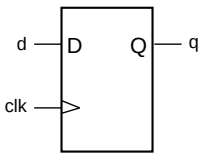
Het onderzoek met de full-adder is erg beperkt. Voor grotere schakelingen kan het wel degelijk uit maken hoe het opgeschreven is. Voor sequentiële schakelingen ligt dit ook anders. Paragraaf 2.10 geeft adviezen voor het schrijven van sequentiële schakelingen.

2.9 D-flipfloppen en dataregisters

Naast combinatorische schakelingen, zoals een full-adder, bestaan er ook sequentiële schakelingen, zoals flipfloppen, registers en tellers. De uitgangen van een combinatorische schakeling hangen alleen af van de huidige waarden van de ingangen. Bij een sequentiële schakeling hangen de uitgangen af van de huidige ingangswaarden en van de toestand waarin de schakeling zich bevindt. Een sequentiële schakeling bevat altijd geheugenelementen.

De D-flipflop

In figuur 2.23 staat het logisch symbool van een flankgevoelige D-flipflop met een positieve, opgaande klokflank. Bij de actieve klokflank van signaal `clk` leest de flipflop de waarde op ingang `d`. De actieve klokflank is bij deze positieve flankgevoelige D-flipflop de opgaande klokflank. Uitgang `q` verandert alleen bij de actieve klokflank. De entity van de flipflop staat in code 2.13. De gedragsbeschrijving van de positieve flankgevoelige D-flipflop staat in code 2.14.



Figuur 2.23 : Het symbool van een D-flipflop.

Code 2.13 : De entity van een D-flipflop.

```
1  entity dff is
2    port (
3      clk : in std_logic;
4      d   : in std_logic;
5      q   : out std_logic
6    );
7  end entity dff;
```

Code 2.14 : De architectuur van een D-flipflop.

```
1  architecture gedrag of dff is
2  begin
3    p1: process (clk) is
4    begin
5      if rising_edge(clk) then
6        q <= d;
7      end if;
8    end process p1;
9  end architecture gedrag;
```

De architectuur bevat een enkel proces `p1`. In de gevoeligheidslijst of de *sensitivity list* van dit proces staat alleen het kloksignaal `clk`. Het proces wordt uitgevoerd als `clk` verandert. De waarde van uitgang `q` hoeft uitsluitend te veranderen bij de opgaande klokflank. De functie `rising_edge` test of signaal `clk` van laag naar hoog is gegaan. Alleen in dat geval krijgt `q` de waarde van `d`. Het if-statement heeft geen else-statement. Er verandert dus niets als `rising_edge` niet waar is. De waarde van `q` blijft in dat geval ongewijzigd. Proces `p1` kent uitsluitend bij de actieve klokflank aan `q` een nieuwe waarde toe.
Een negatieve flankgevoelige D-flipflop reageert op een neergaande klokflank. Code 2.15 gebruikt voor de beschrijving van deze flipflop de functie `falling_edge`.

In plaats van de functie `rising_edge` en `falling_edge` kan de flank ook worden beschreven met het `'event` attribuut. De uitdrukking `clk'event` is waar als de waarde van signaal `clk` verandert. Als het signaal `clk` verandert en tegelijkertijd hoog is geworden, moet het om een opgaande flank gaan. Code 2.16 beschrijft met `'event` een positieve flankgevoelige D-flipflop.

De functies `rising_edge` en `falling_edge` zijn gedefinieerd in `std_logic_1164`.

Code 2.15 : Een negatieve flankgevoelige D-flipflop.

```
1  architecture gedrag of dff is
2  begin
3    p1: process (clk) is
4    begin
5      if falling_edge(clk) then
6        q <= d;
7      end if;
8    end process p1;
9  end architecture gedrag;
```

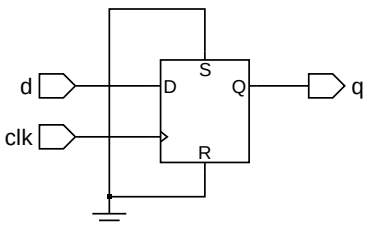
Code 2.16 : Een positieve flankgevoelige D-flipflop met `'event`.

```
1  architecture gedrag of dff is
2  begin
3    p1: process (clk) is
4    begin
5      if clk'event and clk='1' then
6        q <= d;
7      end if;
8    end process p1;
9  end architecture gedrag;
```

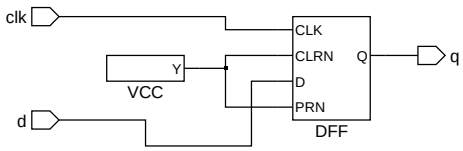
De eerste synthesizers kenden de functies `rising_edge` en `falling_edge` niet. Veel ontwerpers gebruiken daarom nog steeds `'event`. De laatste jaren worden steeds vaker `rising_edge` en `falling_edge` gebruikt.

Het syntheseresultaat van code 2.14 staat in figuur 2.24 en in figuur 2.25. De RTL-view gebruikt een standaard model voor de D-flipflop. De reset en de set van dit model zijn verbonden met de referentie. In de technology-map wordt een D-flipflop gebruikt die in de gegeven technologie beschikbaar is. De actieve lage reset en set van deze flipflop zijn met de voeding verbonden.

Dit boek gebruikt `rising_edge` en `falling_edge`. De leesbaarheid van deze functies is beter dan de beschrijving met `'event`.



Figuur 2.24 : RTL-view van de D-flipflop.



Figuur 2.25 : De technology-map van de D-flipflop.

Dataregisters

Een dataregister bestaat uit de parallel schakeling van een aantal flipfloppen. De beschrijving van een register lijkt daarom heel sterk op de beschrijving van een flipflop. Bij een register zijn de data-ingangen en de data-uitgangen meerdere bits breed. In code 2.17 staat de entity van een 4-bits register. De architectuur staat in code 2.18.

Proces `p1` is identiek aan proces `p1` uit code 2.14. De RTL-view van de synthese staat in figuur 2.26 en de technology-map in figuur 2.27. In beide gevallen bestaat het resultaat uit een register van vier D-flipfloppen.

Het verschil tussen de VHDL-beschrijving van een flipflop en een dataregister is niet groot en het verschil tussen een 4-bits dataregister en een 8-bits dataregister is nog veel kleiner. Dan hoeft namelijk alleen de entitynaam en de breedte van de signalen `d` en `q` te worden aangepast.

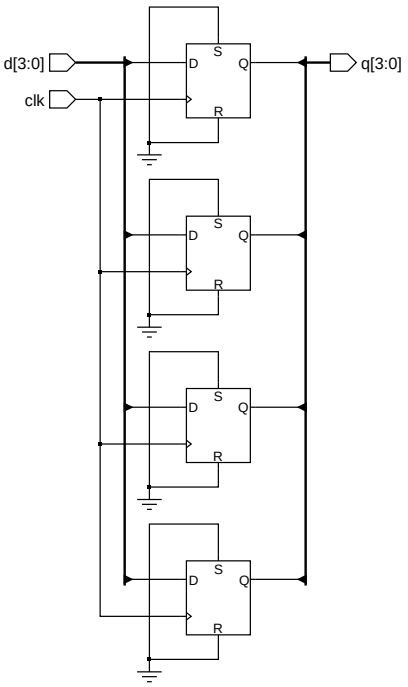
Het ligt voor de hand om een algemene beschrijving van een n-bits register te maken die kan worden ingezet bij iedere bitbreedte. In code 2.19 staat de entity en de architectuur van een n-bits register. Het datasignaal `d` en de uitgang `q` zijn

Code 2.17 : De entity van een 4-bits dataregister.

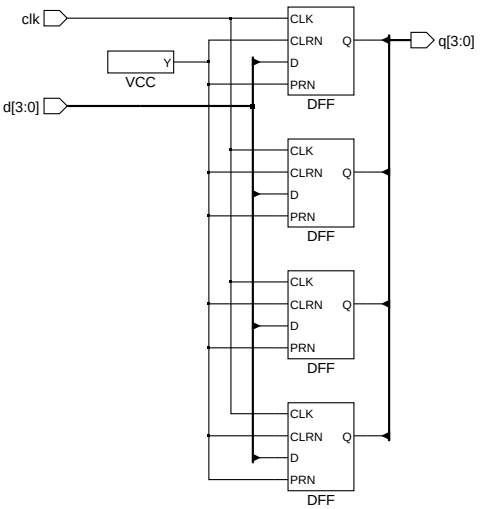
```
1  entity reg4 is
2    port (
3      clk : in  std_logic;
4      d   : in  std_logic_vector(3 downto 0);
5      q   : out std_logic_vector(3 downto 0)
6    );
7  end entity reg4;
```

Code 2.18 : De architectuur van een 4-bits dataregister.

```
1  architecture gedrag of reg4 is
2  begin
3    p1: process (clk) is
4    begin
5      if rising_edge(clk) then
6        q <= d;
7      end if;
8    end process p1;
9  end architecture gedrag;
```



Figuur 2.26 : RTL-view van het 4-bits dataregister.



Figuur 2.27 : De technology-map van het 4-bits dataregister.

Het type `natural` staat voor de natuurlijke getallen $\mathbb{N}$ en is een subset van type `integer` dat staat voor de gehele getallen $\mathbb{Z}$.

beide `n` bits breed; de signalen zijn beide genummerd met `n downto 0`. De constante `n` is gedefinieerd als **`generic`**. Via de signalen uit de *port list* gaat er informatie — enen en nullen — de entity in en uit. Naast deze digitale informatie kunnen er andere waarden worden doorgegeven, bijvoorbeeld de vertragingstijden van de entity. Dit gaat via de generieke parameters van de *generic list*. Deze lijst staat in de entity voor de port-list en begint met het sleutelwoord **`generic`** gevolgd door de lijst met parameters tussen ronde haken. In code 2.19 omvat de lijst één parameter `n` van het type `natural`. Deze parameter is in dit geval de bitbreedte van het register en heeft als standaardwaarde `8`.

Code 2.20 beschrijft een 4-bits register en maakt gebruik van de entity `nreg` uit code 2.19. Het declaratiedeel uit de architectuur bevat de componentdeclaratie en de componentconfiguratie. De componentdeclaratie is in feite een kopie van

Code 2.19 : Een n-bits register.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity nreg is
5    generic (
6      n   : natural := 8
7    );
8    port (
9      clk : in std_logic;
10     d   : in std_logic_vector(n-1 downto 0);
11     q   : out std_logic_vector(n-1 downto 0)
12   );
13 end entity nreg;
14
15 architecture gedrag of nreg is
16 begin
17   p1: process (clk) is
18   begin
19     if rising_edge(clk) then
20       q <= d;
21     end if;
22   end process p1;
23 end architecture gedrag;
```

Code 2.20 : Een 4-bits register op basis van een n-bits register.

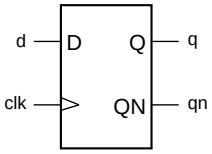
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity reg4 is
5    port (
6      clk : in std_logic;
7      d   : in std_logic_vector(3 downto 0);
8      q   : out std_logic_vector(3 downto 0)
9    );
10 end entity reg4;
11
12 architecture gedrag of reg4 is
13   component nreg is
14     generic (
15       n   : natural := 8
16     );
17     port (
18       clk : in std_logic;
19       d   : in std_logic_vector(n-1 downto 0);
20       q   : out std_logic_vector(n-1 downto 0)
21     );
22   end component nreg;
23
24   for r4: nreg use entity work.nreg(gedrag);
25 begin
26   r4: nreg generic map (
27     n => 4
28   )
29   port map (
30     clk => clk,
31     d => d,
32     q => q
33   );
34 end architecture gedrag;
```

de entity uit code 2.19. De aanroep van `nreg` op regel 26 bevat niet alleen een port map, maar ook een *generic map*. Deze laatste bestaat uit de sleutelwoorden **`generic map`** en vervolgens twee ronde haken met daar tussen een lijst associaties. In dit geval krijgt hier de formele parameter `n` de waarde `4`. Het register zal daarom 4-bits breed zijn.

D-flipflop met geïnverteerde en niet-geïnverteerde uitgang

In figuur 2.28 staat het logisch symbool van een positieve flankgevoelige flipflop met een geïnverteerde en een niet-geïnverteerde uitgang. Code 2.21 geeft de bijbehorende entity.

Code 2.22 bevat een architectuur, die gebruik maakt van een intern signaal `state`. Signaal `state` is de uitgang van het proces `p1` dat een D-flipflop beschrijft. Uitgang `qn` is aangesloten op de geïnverteerde waarde van `state`. De toewijzing op regel 10 verbindt `state` met uitgang `q`. In figuur 2.29 is dit aangegeven met een gestippeld vierkant; alsof er een buffer tussen `state` en `q` zit. In figuur 2.30 en figuur 2.31



Figuur 2.28 : **Het symbool van een D-flipflop met twee uitgangen: geïnverteerd en niet-geïnverteerd.**

Code 2.21 : **De entity van een D-flipflop met twee uitgangen: geïnverteerd en niet-geïnverteerd.**

```
1  entity dff is
2    port (
3      clk : in  std_logic;
4      d   : in  std_logic;
5      q   : out std_logic;
6      qn  : out std_logic
7    );
8  end entity dff;
```

staan de RTL-view en de technology-map van de synthese.

Code 2.22 : **Correcte architectuur voor entity dff2.**

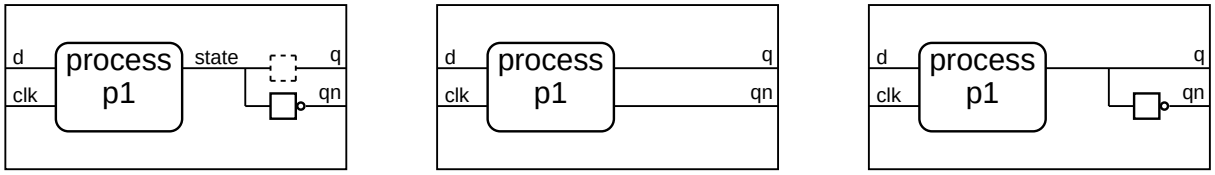
```
1  architecture gedrag of dff is
2    signal state : std_logic;
3  begin
4    p1: process (clk) is
5    begin
6      if rising_edge(clk) then
7        state <= d;
8      end if;
9    end process p1;
10   q  <= state;
11   qn <= not state;
12 end architecture gedrag;
```

Code 2.23 : **Alternatieve architectuur voor entity dff2.**

```
1  architecture gedrag of dff is
2  begin
3    p1: process (clk) is
4    begin
5      if rising_edge(clk) then
6        q  <= d;
7        qn <= not d;
8      end if;
9    end process p1;
10 end architecture gedrag;
```

Code 2.24 : **Foute architectuur voor entity dff2.**

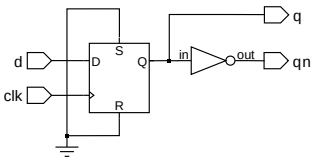
```
1  architecture gedrag of dff is
2  begin
3    p1: process (clk) is
4    begin
5      if rising_edge(clk) then
6        q  <= d;
7      end if;
8    end process p1;
9    qn <= not q;
10 end architecture gedrag;
```



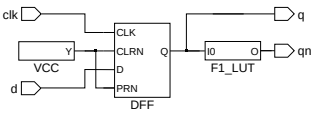
Figuur 2.29 : **Grafische weergave van code 2.22, 2.23 en 2.24.** In de code 2.22 schrijft het proces de uitkomst naar het interne signaal `state`, dat wordt toegekend aan `q` en `qn`. Uitgang `q` wordt niet als ingang voor een ander proces gebruikt. Bij code 2.23 schrijft het proces de uitkomsten direct naar `q` en `qn`. Deze uitgangen worden niet als ingang bij een ander proces gebruikt. In code 2.24 schrijft het proces de uitkomst naar uitgang `q`. Uitgang `q` wordt ook als ingang bij de toekenning aan `qn` gebruikt.

In code 2.23 wordt geen intern signaal gebruikt. De signalen `q` en `qn` krijgen hun waarde in process `p1`. De RTL-view en de technology-map van de synthese bevatten beide nu twee flipfloppen, zoals in figuur 2.32 en 2.33 te zien is. Bij synthese worden alle signalen die *achter* een klokflank zitten geregistreerd. Dat betekent dat er een flipflop of register — meerdere flipfloppen — gebruikt worden. In code 2.22 staat alleen `state` achter de klokflank en wordt er dus één flipflop gebruikt. Bij code 2.22 staan `q` en `qn` beide achter de klokflank. Beide signalen leveren een flipflop op.
De simulaties van beide architecturen zullen ook in lichte mate afwijken. Het signaaldiagram zal hetzelfde zijn, maar de uitvoer op de delta-delay niveau zal anders zijn. Code 2.23 bestaat uit een proces en code 2.22 bestaat uit twee processen.

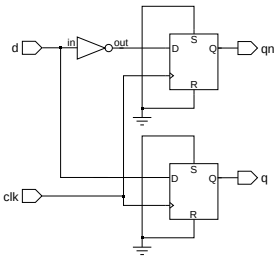
Signaal `qn` zal dan een delta-delay, een simulatieslag, later veranderen.



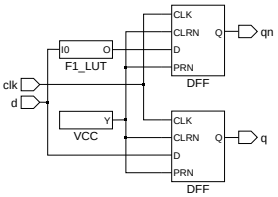
Figuur 2.30 : RTL-view van code 2.22.



Figuur 2.31 : Technology map van code 2.22.



Figuur 2.32 : RTL-view van code 2.23.



Figuur 2.33 : Technology map van code 2.23.

De architectuur van code 2.24 werkt niet goed met de entity van code 2.21. Uitgang `q` van de architectuur is niet alleen een uitgang, maar wordt ook als ingang bij een ander proces gebruikt, namelijk bij de toewijzing op regel 9. In VHDL mag een uitgang nooit als ingang bij een proces gebruikt worden. De compiler van de simulator geeft deze foutmelding:

```
    Error:  Cannot read output "q".
```

De synthesizer geeft dezelfde foutmelding, maar geeft ook een hint om het op te lossen, namelijk om de uitgang een andere modus te geven:

```
    Error, cannot read output: q; use mode buffer or inout.
```

In tabel 2.4 staan de vijf mogelijke modi met een korte uitleg. De modus **`inout`** is de bidirectionele aansluiting. In dat geval kan de aansluiting worden gebruikt om gegevens naar buiten te sturen en om gegevens naar binnen te halen. Het is niet de bedoeling dat er een signaal op de uitgang `q` van de flipflop wordt gezet. Dus de modus **`inout`** is hier niet geschikt.
De modus **`buffer`** is bedoeld voor dit soort situaties. De aansluiting werkt als uitgang, maar intern mag het signaal ook worden gebruikt als ingang voor andere processen.

Tabel 2.4 : De modi van de aansluitingen.

modus	type io	uitleg
`in`	ingang	er kan alleen gelezen worden
`out`	uitgang	er mag alleen geschreven worden, de uitgang kan intern niet als ingang van een ander proces gebruikt worden
`inout`	bidirectioneel	er mag gelezen en geschreven worden
`buffer`	uitgang	er mag alleen geschreven worden, de uitgang mag intern wel als ingang van een ander proces gebruikt worden
`linkage`		is bedoeld om VHDL te koppelen met niet-VHDL systemen; wordt bij gewone ontwerpen niet gebruikt

Met de entity uit code 2.25 werkt de beschrijving uit code 2.24 wel goed. De syntheseresultaten zijn gelijk aan die van figuur 2.30 en van figuur 2.31.
In dit boek wordt de modus **`buffer`** weinig gebruikt. Als een uitgang intern als ingang gebruikt, wordt er net als in code 2.22 een intern signaal gebruikt. De ontwerper schrijft meestal eerst de entity. Op dat moment is er meestal nog geen enkele reden om **`buffer`** als modus te gebruiken. Het is meestal een aanpassing

Code 2.25 : De entity van de D-flipflop met modus `buffer`.

```
1  entity dff is
2    port (
3      clk : in     std_logic;
4      d   : in     std_logic;
5      q   : buffer std_logic;
6      qn  : out    std_logic
7    );
8  end entity dff;
```

achteraf. Er zijn ook ontwerpers die voor de modus van de uitgangen altijd `buffer` gebruiken.

2.10 Evaluatie beschrijvingen flipfloppen en registers

Bij de voorbeelden van de full-adder in paragraaf 2.7 is gevonden dat de technology-maps allemaal hetzelfde zijn. In figuur 2.22 van paragraaf 2.8 is dat gevisualiseerd. In paragraaf 2.9 blijkt dat de beschrijvingen van de flipfloppen en registers wel een andere technology-map kunnen geven.

Een full-adder is een combinatorische schakeling. Flipfloppen en registers zijn sequentiële schakelingen. Bij combinatorische schakelingen hangt de waarde van de uitgangen alleen af van de huidige waarden van de ingangen. Bij sequentiële schakelingen hangt de waarde van de uitgangen niet alleen af van de huidige waarden van de ingangen, maar ook van waarden die de ingangen eerder hadden. Deze schakelingen hebben altijd een klok en bevatten flipfloppen.
Gedragsbeschrijvingen van combinatorische schakelingen zullen over het algemeen hetzelfde syntheseresultaat opleveren. Alleen bij zeer complexe schakelingen kan de optimalisatie anders uitpakken. Bij sequentiële schakelingen ligt dat anders. De synthesizer gebruikt voor elk signaal dat zich *achter* een flipflop of register. Het maakt een groot verschil of een toekenning in een proces na een klokflank staat of dat het buiten een proces of niet achter een klokflank staat. Beschrijvingen van sequentiële schakelingen kunnen daarom heel verschillende uitkomsten geven.

Hoofdstuk 3 bespreekt het tijdsgedrag en het simulatiemechanisme van VHDL. De belangrijkste aspecten en constructies worden in dat hoofdstuk verder uitgelegd. Hoofdstuk 4 behandelt de synthese van VHDL, daar wordt uitgelegd hoe de synthesizer de VHDL-code interpreteert en er een schakeling van maakt.

3

Simulatiemodel VHDL

Doelstelling

In dit hoofdstuk leer je hoe het simulatiemodel van VHDL werkt, leer je wat het verschil is tussen een parallelle en een sequentiële omgeving en tussen parallelle en sequentiële constructies. Verder maak je kennis met signalen en variabelen, met parallelle en gewone signaaltoewijzingen en met tijdvertragingen.

Onderwerpen

De behandelde onderwerpen zijn:

- Het simulatiemodel van de taal VHDL.
- Het concept van een **`process`**.
- Het wacht-statement **`wait`**.
- De impliciete wait en de *sensitivity list* of gevoeligheidslijst.
- Sequentiële en parallelle opdrachten.
- De verschillen tussen variabelen en signalen
- Gewone signaaltoewijzingen en parallelle signaaltoewijzingen.
- Sequentiële en parallelle omgevingen.
- Meervoudige toewijzingen aan een signaal.
- Tijdvertragingen: *delta delay*, *inertial delay* en *transport delay*.
- Samengestelde signaaltoewijzingen.

Hardware kan worden opgevat als een verzameling parallelle processen die met elkaar communiceren. Elk van deze processen bepaalt op basis van zijn ingangssignalen voortdurend wat de uitgangssignalen moeten zijn. Dit is een eigenschap die we op een sequentiële computer met een enkele processor nooit kunnen nabootsen. Als alle processen continu actief moeten zijn, zijn er minstens evenveel processoren nodig als dat er processen zijn. De truc om dit dilemma op te lossen is een proces alleen dóór te rekenen als een van de ingangswaarden verandert. De uitgangen veranderen immers niet als de ingangen stabiel zijn.
Er is dan wel een mechanisme nodig dat regelt welk proces aan de beurt is om te worden doorgerekend. VHDL gebruikt hiervoor, net als bijna alle logische simulatoren, het event-driven simulatiemodel.

3.1 Parallelle versus sequentiële constructies

Het bijzondere van VHDL is dat de gebruiker te maken krijgt met zowel parallelle als sequentiële taalconstructies. Parallelle constructies zijn nodig om netwerkstructuren te beschrijven en sequentiële constructies zijn nodig om gedragsbeschrijvingen te maken.
De parallelle constructies — de *concurrent statements* — zijn alleen toegestaan in een *concurrent body* en de sequentiële constructies — de *sequential statements* — zijn alleen toegestaan in een *sequential body*. In een concurrent body of parallelle omgeving worden de opdrachten en toewijzingen — schijnbaar – gelijktijdig uitgevoerd. In een sequential body of sequentiële omgeving worden alle opdrachten en toewijzingen na elkaar uitgevoerd.
De belangrijkste parallelle omgeving is de architectuur, een ander voorbeeld is het block-statement. In dit boek wordt het block-statement niet gebruikt; parallelle opdrachten en toewijzingen staan daarom altijd in een architectuur. De declaraties, die nodig zijn voor parallelle constructies, staan in de architectuur, in de bijbehorende entity of in een package.
Het proces is het belangrijkste voorbeeld van een sequentiële omgeving; andere voorbeelden zijn de procedure en de functie. Sequentiële opdrachten en toewijzingen staan dus altijd in een procedure, functie of proces. Tabel 3.1 geeft een overzicht van de belangrijkste taalconstructies in VHDL. De parallelle opdrachten en toewijzingen staan direct in een architectuur. De sequentiële opdrachten en toewijzingen staan in een proces, procedure of functie.

Tabel 3.1 : De belangrijkste parallelle en sequentiële opdrachten en toewijzingen.

parallelle opdrachten	sequentiële opdrachten
process statement	—
—	wait statement
—	variable assignment
concurrent signal assignment	signal assignment
concurrent procedure call	procedure call
conditional signal assignment (**`when else`**)	if statement
selected signal assignment (**`with select`**)	case statement
generate statements	loop statements
component instantiation	—

In de nieuwe VHDL2008 vervalt het strenge onderscheid tussen parallelle en sequentiële opdrachten gedeeltelijk. De **`when else`** kan dan ook in een sequentiële omgeving staan. Dit boek gebruikt VHDL2002. Alle ontwikkelomgevingen ondersteunen inmiddels VHDL2002 en geen enkele ondersteunt VHDL2008 volledig.
De norm loopt altijd voor op de standaard die de ontwikkelomgevingen gebruiken. Bovendien stappen de ontwerpers pas over als er belangrijke, fundamentele wijzigingen zijn.

Het conditional signal assignment (**`when else`**) wordt de *parallelle if* genoemd en het selected signal assignment (**`with select`**) wordt de *parallelle case* genoemd.
Code 2.4 gebruikt een parallelle case. Deze parallelle case of selected signal assignment staat direct in de architectuur. Code 2.2 bevat een gewoon case-statement. Dit statement staat in een sequential body, namelijk in process `p1`.
In code 3.1 staat de beschrijving van een tristatebuffer met een if-statement en in code 3.2 staat een beschrijving van dezelfde tristatebuffer met een conditional signal assignment. Het grootste verschil tussen de parallelle if en de gewone if is dat de beschrijving met de parallelle if veel beknopter is.
De tristatebuffer functioneert als een gewone buffer wanneer het signaal `enable` hoog is. Uitgang `outp` is hoog als de ingang `inp` hoog is en laag als de ingang laag is. Als het enable-signaal laag is, is de tristatebuffer hoogimpedant en krijgt de uitgang de waarde `'Z'`.

Code 3.1 : Een beschrijving van een tristatebuffer met een if-statement.

```
1  entity tribuf is
2    port (
3      inp    : in  std_logic;
4      enable : in  std_logic;
5      outp   : out std_logic
6    );
7  end entity tribuf;
8
9  architecture gedrag of tribuf is
10 begin
11   p1: process (inp,enable) is
12   begin
13     if enable = '1' then
14       outp <= inp;
15     else
16       outp <= 'Z';
17     end if;
18   end process p1;
19 end architecture gedrag;
```

Code 3.2 : Een beschrijving van een tristatebuffer met een conditional signal assignment.

```
1  entity tribuf is
2    port (
3      inp    : in  std_logic;
4      enable : in  std_logic;
5      outp   : out std_logic
6    );
7  end entity tribuf;
8
9  architecture gedrag of tribuf is
10 begin
11   outp <= inp when enable = '1' else 'Z';
12 end architecture gedrag;
```

3.2 Het verschil tussen variabelen en signalen

Bij hardware is er altijd sprake van een zekere tijdvertraging. Een systeem — of subsysteem — reageert nooit direct; het reageert altijd pas na enige tijd. Dat is de reden dat de hardwarebeschrijvingstaal VHDL naast constanten en variabelen ook signalen kent. Signalen zijn de verbindingen tussen de verschillende processen waaruit de hardware is opgebouwd.

Dat er altijd een tijdvertraging is, is een fundamentele fysische beperking. Voor een oneindig kleine tijdvertraging is een oneindig grote hoeveelheid energie nodig.
Dit volgt direct uit van één van de onzekerheidsrelaties van Heisenberg:

$$\Delta E \, \Delta t \;\geq\; \frac{h}{4\pi}$$

Hierin is Δt de tijdvertraging, ΔE de energieverandering en h de constante van Planck.

- **`constant`:**
  Een constante heeft een vaste waarde. Constanten staan alleen in het declaratiedeel van een ontwerpeenheid en krijgen eenmalig — bij de initialisatie — een waarde toegekend.
- **`variable`:**
  Een variabele kan tijdens de simulatie een andere waarde krijgen. Variabelen komen alleen voor in een sequentiële omgeving. Ze kunnen alleen lokaal gedeclareerd en gebruikt worden in functies, procedures en processen.
- **`signal`:**
  Een signaal kan tijdens de simulatie een andere *wave form* krijgen. Bij een signaal hoort een lijst met toekomstige waarden en tijdstippen wanneer deze waarden worden toegekend.
  Signalen komen voor in sequentiële en in parallelle omgevingen. Signaaltoewijzingen mogen dus in de architectuur staan, maar ook in functies, procedures en processen. De signalen worden altijd in een parallelle omgeving gedeclareerd, bijvoorbeeld in de port list van een entity, het declaratiedeel van de architectuur of in een package. De declaratie van signalen staat nooit in een functie, procedure of proces.

- **`shared variable`:**
  De *shared* variabele is in 1993 aan VHDL toegevoegd en is een variabele die globaal gebruikt wordt. Deze globale variabelen ontberen, in tegenstelling tot signalen, de mogelijkheid om met parallelle opdrachten om te gaan. De uitkomst van een simulatie is dan onbepaald. Ontwerpen met shared variabelen zijn bovendien niet synthetiseerbaar. Het gebruik van deze variabelen wordt daarom sterk ontraden. Alleen in bijzondere omstandigheden — bijvoorbeeld bij het maken van testbenches — kan een shared variabele nuttig zijn.

Signalen zijn nodig voor de communicatie tussen de processen waaruit een ontwerp is opgebouwd; op dezelfde wijze als er verbindingen nodig zijn tussen de verschillende onderdelen in een fysieke schakeling. Variabelen spelen alleen een rol binnen één proces. Het zijn hulpmiddelen om de beschrijving van het gedrag binnen het proces eenvoudiger te maken. In code 2.2 is in proces `p1` een 3-bits vector nodig voor de keuze van het case-statement. Dit is een voorbeeld waarbij een variabele handig is. In code 2.4 is bij het selected signal assignment om dezelfde reden een signaal `x` gebruikt.
Voor de toewijzing, of *assignment*, aan constanten, variabelen en signalen worden twee verschillende symbolen gebruikt:

`:=` Dit teken wordt gebruikt voor de toewijzing aan constanten, variabelen en bij de initialisatie van signalen.

```
doel := waarde;
```

Men noemt een dergelijke toekenning ook wel de variabele of de directe toewijzing.

`<=` Met dit symbool — het 'pijltje' — wordt aan een signaal een complete *wave form* toegekend. Alleen bij de initialisatie van signalen wordt `:=` gebruikt.

```
doel <=   waarde1 [ after tijdvertraging1 ]
        { ,waardei [ after tijdvertragingi ] };
```

Deze toekenning noemt men een signaaltoewijzing.

Code 3.3 toont enkele voorbeelden van declaraties en toewijzingen aan constanten, variabelen en signalen.

Bij VHDL verschilt het declareren en het toekennen van constanten en variabelen niet wezenlijk van andere programmeertalen. De toewijzingen van meerdere bits aan vectoren kan op een aantal manieren gedaan worden.
Bij de toewijzing aan variabele `v3` wordt de associatie via de positie gebruikt. De linker bit wordt `'0'` en de volgende drie bits worden respectievelijk `'1'`, `'0'` en `'1'`. Aan variabele `v4` wordt een complete bitstring toegekend. Bij variabelen `v5` en `v6` wordt de associatie met de naam toegepast. De bits met bitnummer `0` en `2` van vector `v6` worden `'1'` en de andere twee bits worden `'0'`. Het effect is hetzelfde als de toewijzing aan variabele `v3`. In beide gevallen wordt `"0101"` toegekend. De associatie op naam is vooral handig bij het *nul* maken van een vector. Alle bits van variabele `v5` worden nul.

Signaal `s1` wordt met de toewijzing `:=` geïnitialiseerd met een '0'. Alle andere signalen zijn niet geïnitialiseerd bij de declaratie. Variabelen en signalen van het type `std_logic` — en daarvan afgeleide typen als `std_logic_vector` en `unsigned` — zijn

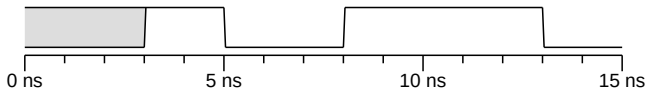
Code 3.3 : Voorbeelden van declaraties en toewijzingen aan constanten, variabelen en signalen.

```
-- constant declarations
-- allowed in the declaration part of
-- either a concurrent or sequential body
constant c1    : std_logic := '0';
constant c2    : integer := 255;

-- variable declarations
-- only allowed
-- in declaration part of a sequential body
variable v1    : std_logic;
variable v2    : integer := 0;
variable v3,v6 : std_logic_vector(3 downto 0);
variable v4,v5 : unsigned(7 downto 0);

-- variable assignments
-- only allowed in a sequential body
v1 := '1';
v2 := 24;
v3 := ('0','1','0','1');
v4 := "1001";
v5 := (others => '0');
v6 := (0 => '1', 2 => '1', others => '0');
```

```
-- signal declarations
-- allowed in the declaration part of
-- a concurrent body
signal s1     : std_logic := '0';
signal s2, s5 : integer;
signal s3, s6 : std_logic_vector(3 downto 0);
signal s4, s7 : std_logic;
signal s8     : unsigned(3 downto 0);

-- signal assignments
-- allowed in either a concurrent or
-- a sequential body
s1 <= '1';
s2 <= 24;
s3 <= ('0','1','0','1');
s4 <= '1' after 4 ns;
s5 <= 24 after 2.57 ns;
s6 <= ('0','1','0','1') after 10 ns;
s7 <= '1' after 3 ns, '0' after 5 ns,
      '1' after 8 ns, '0' after 13 ns;
s8 <= (3 => '1', others => '0'),
      (2 => '1', others => '0') after 2 ns;
```

standaard bij de initialisatie niet geïnitialiseerd. Dat betekent dat `v1`, `s4` en `s7` aanvankelijk de waarde `'U'` hebben en dat de bits van de vectoren allemaal `'U'` zijn.

De signaaltoewijzingen aan `s1`, `s2` en `s3` zien er niet anders uit dan de toewijzingen aan de variabelen `v1`, `v2` en `v3`. Signaal `s4` krijgt de waarde `'1'` na dat er vier nanoseconden verstreken zijn en signaal `s5` wordt `24` na 2,57 ns. Figuur 3.1 toont de waveform die aan `s7` wordt toegekend. Dit signaal blijft ongewijzigd tot er 3 ns zijn verstreken, daarna wordt het signaal hoog. Twee nanoseconden later — namelijk vijf nanoseconden vanaf het moment van toewijzen — wordt het signaal weer laag. Drie nanoseconden later wordt het signaal hoog en tenslotte wordt het signaal weer laag. In figuur 3.1 is deze waveform getekend. De vector `s8` krijgt eerst de waarde `"1000"` en na 2 ns krijgt het de waarde `"0100"`.



Figuur 3.1 : De waveform van signaal `s7`.

3.3 Delta-delay

Tijdens de simulatie veranderen de waarden van signalen nooit direct, dat gebeurt altijd op een later moment. Ook als er geen expliciete **`after`**-clausule wordt meegegeven of als de vertragingstijd expliciet `0 ns` is, wordt de nieuwe waarde niet direct toegekend. Men spreekt dan van een tijdvertraging *delta delay*:

```
t <= '1';
t <= after 0 ns;   }   t <= after delta delay;
```

Delta-delay is overigens geen sleutelwoord in VHDL. Het is alleen een spraakgebruik om aan te geven dat er een tijdvertraging van één simulatieslag is.
Delta-delay wordt vaak beschouwd als een oneindig kleine tijdvertraging. Feitelijk is dat niet correct. Het is geen echte tijdvertraging; de simulatietijd verandert niet. Een vertraging van delta-delay betekent alleen dat de simulatiecyclus helemaal doorlopen wordt voordat de nieuwe waarde wordt toegekend. In de listing van figuur 2.13 met de delta-delay's is te zien dat er steeds twee simulatieslagen nodig zijn voordat de simulatietijd verandert. Delta-delay komt overeen met één simulatieslag.

3.4 Het simulatiemodel

Hardware kan worden opgevat als een verzameling van parallelle processen, die onderling met elkaar communiceren. Een VHDL-simulatie, die op een niet-parallelle computer wordt uitgevoerd, moet het parallel functioneren van processen nabootsen. VHDL gebruikt hiervoor het event-driven simulatiemodel.
Het simulatiemodel en de syntax van VHDL is vastgelegd in de *Language Reference Manual*. In 1987 heeft IEEE de eerste goedgekeurde versie IEEE 1076-1987 uitgebracht. In de loop der jaren is de taal een aantal keren herzien. In 1993, 2002 en 2008 zijn de IEEE 1076-1993, de IEEE 1076-2002 en de de IEEE 1076-2008 uitgegeven.

Het proces

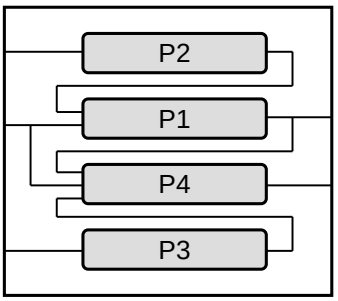
Het proces is het belangrijkste onderdeel van VHDL. Processen vormen de basis van het simulatiemodel. De VHDL-code is altijd te herleiden tot een lijst met processen. Alle processen uit deze lijst worden een voor een geëvalueerd. Ieder proces bevat een of meer wait-statements. De simulatie gaat verder met het volgende proces als een wait-statement bereikt is. Pas als alle processen geëvalueerd zijn, worden de signalen aangepast en worden de processen opnieuw geëvalueerd. Het één keer evalueren van alle processen wordt een simulatieslag, *simulation cycle* of *delta cycle* genoemd.

Hoofdstuk 2 bevat een aantal verschillende beschrijvingen van een full-adder. Sommige van deze beschrijvingen, zoals code 2.2, bevatten expliciet een proces. In andere voorbeelden komt het sleutelwoord **`process`** niet voor: in code 2.6 staan slechts zes concurrent signal assignments en in code 2.10 worden alleen vijf componenten aangeroepen.
Ondanks dat het woord **`process`** soms ontbreekt, bestaan alle beschrijvingen uit een of meer processen. De zes concurrent signal assignments uit code 2.6 zijn feitelijk zes parallelle processen. De vijf aanroepen uit code 2.10 bevatten beschrijvingen uit code 2.8. Elke aanroep bevat in dit geval een concurrent signal assignment, dat op zijn beurt weer een compleet proces voorstelt.
Ieder concurrent statement is te herschrijven met, of is op minst te herleiden tot, een proces. Een architectuur bestaat daardoor altijd uit een samenstelling van één of meer processen.

Het proces is zelf een concurrent statement en staat altijd in een parallelle omgeving. De binnenkant van het proces is daarentegen sequentieel. Alle opdrachten en toewijzingen in het proces zijn sequentieel en worden na elkaar uitgevoerd. In een proces staan geen parallelle opdrachten. Omdat het proces een parallelle op-

Code 3.4 : Architectuur met vier processen.

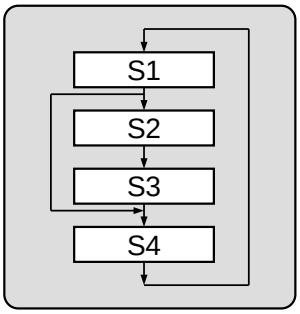
```
1 architecture a of e is
2 begin
3   P2: process ··· end process P2;
4   P1: process ··· end process P1;
5   P4: process ··· end process P4;
6   P3: process ··· end process P3;
7 end architecture a;
```



Figuur 3.2 : De vier processen van architectuur a.

Code 3.5 : Proces met vier statements.

```
1 P1 : process is
2 begin
3     -- statement S1;
4     if ··· then
5       -- statement S2;
6       -- statement S3;
7     end if;
8     -- statement S4;
9 end process P1;
```



Figuur 3.3 : De vier statements van process P1.

Een geneste routine of *nested routine* is een routine die aangeroepen wordt binnen een andere routine.

dracht is, mogen in een proces geen andere processen staan. In tegenstelling tot procedures en functies mogen processen dus niet genest worden.

Architectuur a uit code 3.4 bestaat uit vier processen en is grafisch weergeven in figuur 3.2. De volgorde waarin de processen zijn opgeschreven doet er niet toe. Alle processen worden — schijnbaar — gelijktijdig uitgevoerd. In iedere simulatieslag worden alle processen na elkaar doorgerekend zonder dat de simulatietijd daarbij verandert. Aan de waveform van de signalen worden de nieuwe waarden toegevoegd. De huidige waarden van de signalen veranderen dus nog niet. Pas aan het eind van een simulatiecyclus — nadat alle processen uitgevoerd zijn — veranderen er mogelijk signaalwaarden. Alleen als er na een simulatieslag geen signaalveranderingen zijn, wordt de simulatietijd aangepast.

Elk proces bestaat uit een of meer opdrachten die sequentieel — na elkaar — worden uitgevoerd. Code 3.5 bevat een proces P1 dat uit vier opdrachten S1, S2, S3 en S4 bestaat. Figuur 3.3 laat zien dat de opdrachten na elkaar worden uitgevoerd. Na S1 volgt S2, dan S3, daarna S4 en tenslotte gaat het weer verder bij S1. Als het proces een conditionele opdracht bevat, kunnen er wel opdrachten worden overgeslagen. Ook in dat geval worden de opdrachten na elkaar uitgevoerd.
Het uitvoeren van deze opdrachten kost *geen* simulatietijd. Natuurlijk kost het wel rekentijd van de computer waarop de simulatie wordt uitgevoerd. Nadat bij proces P1 de laatste toewijzing S4 is uitgevoerd, wordt er gewoon verder gegaan bij S1. Omdat dit een eindeloze lus geeft, moet een proces *altijd* een wait-statement bevatten. Met het wait-statement wordt de uitvoering onderbroken en gaat de simulator het volgende proces — bijvoorbeeld P2 — doorrekenen.

Het wait-statement

VHDL kent vier wait-statements:

- `wait for` tijdsinterval;
  De executie van het proces wordt gedurende het aangegeven tijdsinterval onderbroken.
  Voorbeeld : `wait for 3 us;`
- `wait until` Booleaanse uitdrukking;
  Het proces blijft wachten totdat er aan de opgegeven Booleaanse conditie wordt voldaan.
  Voorbeeld : `wait until clk='1';`
- `wait on` sensitivity list;
  De uitvoering van het proces wordt onderbroken totdat een of meer van de signalen in de opgegeven lijst van waarde verandert.
  Voorbeeld : `wait on a,b;`
- `wait;`
  De executie van het proces stopt definitief. Dit wordt gebruikt om een proces precies één keer te doorlopen, bijvoorbeeld om een initialisatie uit te voeren. Een typisch voorbeeld is de signaalgenerator van code 2.3.

In een proces mogen meerdere wait-statements voorkomen, mits er geen impliciete wait wordt gebruikt.

De impliciete wait

Naast de vier genoemde wait-statements kent VHDL ook een impliciete wait. Bij een proces met een impliciete wait staat achter het woord **process** tussen twee ronde haakjes een gevoeligheidslijst — *sensitivity list* — vermeld. Deze lijst impliceert een wait-on aan het einde van het proces.
Het proces `expl_wait` uit code 3.6 bevat op regel 6 een expliciete wait. Proces `impl_wait` uit code 3.7 bevat een impliciete wait: achter het sleutelwoord **process** staat een gevoeligheidslijst. De processen `impl_wait` en `expl_wait` zijn wat hun gedrag betreft volkomen identiek. Als één van de ingangssignalen `x` of `y` van waarde verandert, zal in beide gevallen het uitgangssignaal `z` opnieuw worden berekend. Daarna wachten beide processen weer op een verandering van één van de ingangssignalen.
In een proces met een impliciete wait mogen geen andere wait-statements voorkomen. De impliciete wait is een duidelijke toewijzing; zij staat aan het begin van het proces en het is meteen helder dat er geen andere wait-statements aanwezig zijn. De impliciete wait wordt daarom veel gebruikt.

Code 3.6 : Proces met een expliciete wait.

```
1  architecture a of e is
2  begin
3    expl_wait: process is
4    begin
5      z <= x and y;
6      wait on x,y;
7    end process expl_wait;
8  end architecture a;
```

Code 3.7 : Proces met een impliciete wait.

```
1  architecture a of e is
2  begin
3    impl_wait: process (x,y) is
4    begin
5      z <= x and y;
6    end process impl_wait;
7  end architecture a;
```

Code 3.8 : Concurrent signal assignment.

```
1  architecture a of e is
2  begin
3    conc_signal: z <= x and y;
4  end architecture a;
```

Concurrent signal assignments

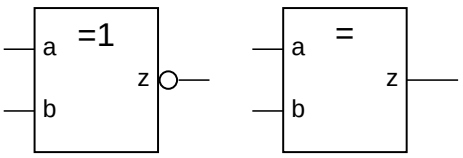
De processen `expl_wait` en `impl_wait` uit code 3.6 en 3.7 zijn identiek en kunnen nog korter worden opgeschreven. In code 3.8 staat de signaaltoewijzing direct in de architectuur. Deze signaaltoewijzing wordt daarom een *concurrent signal assignment* of een parallelle signaaltoewijzing genoemd. Het is meer dan alleen een signaaltoewijzing: het stelt namelijk het proces `expl_wait` voor.
Alleen als ingang `x` of `y` van waarde verandert, wordt de toewijzing uitgevoerd. Dat verschilt niet van proces `expl_wait` of proces `impl_wait`. Ook dan wordt de signaaltoewijzing uitgevoerd als er een verandering is van `x` of `y`.
Omdat de parallelle signaaltoewijzing een proces voorstelt en omdat het direct in een parallelle omgeving staat, kan het net als alle andere parallelle constructies een label krijgen. In dit geval is dat `conc_signal`.

Ondanks dat de syntax exact gelijk is (`z <= x and y;`) zijn er twee soorten signaaltoewijzingen te onderscheiden. De sequentiële of gewone signaaltoewijzing staat in een sequentiële omgeving en kent aan een signaal een waveform toe. De parallelle signaaltoewijzing of *concurrent signal assignment* staat in een parallelle omgeving, kent ook een waveform toe, maar representeert tegelijkertijd een compleet proces.
Elke parallelle signaaltoewijzing, dus ook het *selected signal assignment* en het *conditional signal assignment*, is te herschrijven als een expliciet proces. Code 3.8 is identiek met code 3.6 en code 3.7. Het selected signal assignment van code 2.4 is te herschrijven als proces `P1` van code 2.2 en het conditional signal assignment van code 3.2 is identiek aan proces `P1` uit code 3.1.

3.5 De exnor als voorbeeld

Vanaf 1993 kent VHDL ook de logische functie **`xnor`**.
De exnor kan dus ook met een enkele toewijzing worden opgeschreven:
`z <= a xnor b;`

Deze paragraaf gebruikt een exnor om de verschillen tussen parallelle en sequentiële beschrijvingen en die tussen signalen en variabelen te laten zien. Figuur 3.4 geeft twee symbolen van een exnor. In tabel 3.2 staat de functietabel en in code 3.9 staat de entity. Als de ingangen `a` en `b` gelijk zijn is de uitgang hoog, anders is de uitgang laag.



Figuur 3.4 : **Twee symbolen voor een exnor.**

Tabel 3.2 : **De functietabel van de exnor.**

a	b	z
0	0	1
0	1	0
1	0	0
1	1	1

Code 3.9 : **De entity van de exnor.**

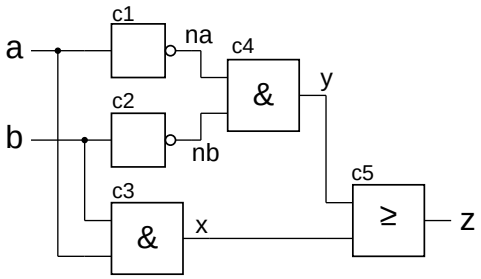
```
1  entity exnor is
2    port (
3      a  : in  std_logic;
4      b  : in  std_logic;
5      z  : out std_logic
6    );
7  end entity exnor;
```

Exnor met vijf parallelle signaaltoewijzingen

De beschrijving van de exnor uit code 3.10 bestaat uit vijf parallelle signaaltoewijzingen. Figuur 3.5 toont het bijbehorende schema.

Code 3.10 : De exnor met vijf parallelle signaaltoewijzingen.

```
1  architecture gedrag of exnor is
2    signal na, nb, x, y : std_logic;
3  begin
4    c1: na <= not a;
5    c2: nb <= not b;
6    c3: x  <= a and b;
7    c4: y  <= na and nb;
8    c5: z  <= x or y;
9  end architecture gedrag;
```



Figuur 3.5 : De exnor opgebouwd uit vijf parallelle processen.

Tabel 3.3 : Het simulatieresultaat van de exnor met vijf parallelle processen. De eerste kolom bevat de simulatietijd en de tweede het aantal delta-delay's. De achtergrond van de signalen is donker grijs als het proces tijdens de betreffende simulatieslag actief is. De waarden van de signalen, die bij deze simulatieslag veranderen, zijn vet gedrukt. In de laatste kolom staan de acties die uitgevoerd worden bij de volgende simulatieslag.

ns	δ	a	b	na	nb	x	y	z	*volgende actie*
0	+0	U	U	U	U	U	U	U	*pas simulatietijd aan*
0	+1	0	0	U	U	U	U	U	*c1,c2,c3*
0	+2	0	0	1	1	0	U	U	*c4,c5*
0	+3	0	0	1	1	0	1	U	*c5*
0	+4	0	0	1	1	0	1	1	*pas simulatietijd aan*
10	+1	0	1	1	1	0	1	1	*c2,c3*
10	+2	0	1	1	0	0	1	1	*c4*
10	+3	0	1	1	0	0	0	1	*c5*
10	+4	0	1	1	0	0	0	0	*pas simulatietijd aan*
20	+1	1	0	1	0	0	0	0	*c1,c2,c3*
20	+2	1	0	0	1	0	0	0	*c4*
20	+3	1	0	0	1	0	0	0	*pas simulatietijd aan*
30	+1	1	1	0	1	0	0	0	*c2,c3*
30	+2	1	1	0	0	1	0	0	*c4,c5*
30	+3	1	1	0	0	1	0	1	*pas simulatietijd aan*
40	+1	0	0	0	0	1	0	1	*c1,c2,c3*
40	+2	0	0	1	1	0	0	1	*c4,c5*
40	+3	0	0	1	1	0	1	0	*c5*
40	+4	0	0	1	1	0	1	1	

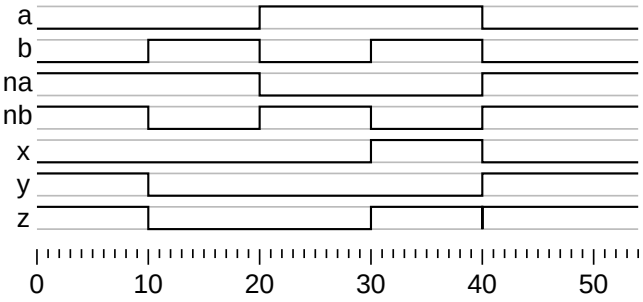
Het simulatieresultaat staat in de vorm van een listing in figuur 3.3. Bij de start van de simulatie zijn de signalen ongeïnitialiseerd; ze hebben de waarde `'U'`. Op tijdstip 0 krijgen de signalen `a` en `b` een nieuwe waarde. Deze veranderingen zorgen ervoor dat de processen `c1`, `c2` en `c3` geëvalueerd worden. Bij de volgende simulatieslag krijgen `na`, `nb` en `x` een nieuwe waarde. De verandering van `na` en `nb` zorgt er voor dat bij de volgende simulatieslag proces `c4` geëvalueerd wordt. Op dezelfde manier triggert de verandering van `x` proces `c5`. De uitgang van proces `c5` blijft `'U'`, omdat `y` nog steeds `'U'` is. In de volgende simulatieslag wordt `y`, de uitgang van proces `c4`, hoog. Dit triggert opnieuw proces `c5`, zodat in de volgende simulatieslag `z` hoog wordt. Daarna verandert er niets meer en wordt de simulatietijd gewijzigd. De eerstvolgende gebeurtenis is het hoog worden van signaal `b` na 10 ns.

Elke keer wanneer er geen veranderingen meer zijn, wordt de simulatietijd opgehoogd naar de eerstvolgende gebeurtenis en worden er één of meer simulatieslagen uitgevoerd. In dit voorbeeld zijn dat steeds drie of vier simulatieslagen.

Spikes zijn zeer ongewenst. Een signaal dat spikes vertoont en op de interrupt van een processor is aangesloten, zal fouten geven.
Spikes worden voorkomen door de uitgangen een register te geven.

Bij 40 ns zijn er vier simulatieslagen nodig. De uitgang blijft aanvankelijk nog hoog, wordt bij delta-cycle 3 laag en bij de laatste simulatieslag toch weer hoog. Er treedt een zogenoemde *spike* of *glitch* op. Spikes kunnen bijna bij alle combinatorische schakelingen voorkomen en worden veroorzaakt door looptijdverschillen tussen verschillende paden. In dit geval loopt het pad via `c2`, `c4` en `c5` langs drie componenten en het pad via `c3` en `c5` langs twee componenten. De waveform van deze simulatie staat in figuur 3.6. De spike is bij 40 ns zichtbaar als een streepje. De

waveform bevat minder informatie dan de listing uit figuur 3.3. De delta-delay's op een bepaald tijdstip vallen samen; alle veranderingen lijken op hetzelfde moment te gebeuren.



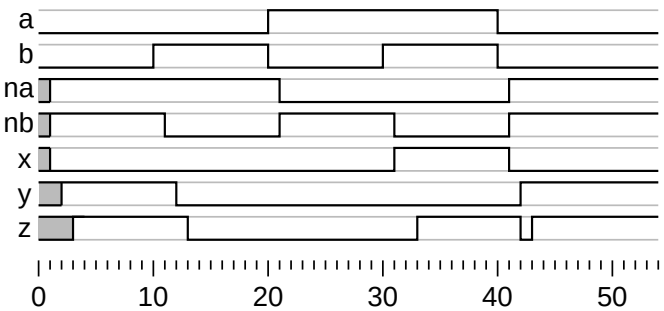
Figuur 3.6 : De waveform bij de simulatie van de exnor met vijf parallelle processen.

Exnor met vijf parallelle signaaltoewijzingen en tijdvertragingen

In code 3.11 is de volgorde van de vijf signaaltoewijzingen verandert en zijn er tijdvertragingen van 1 ns toegevoegd. Bij gedrags- en dataflowbeschrijvingen zijn tijdvertragingen niet erg zinvol. Na synthese ontstaat er meestal een ander netwerk met een eigen karakteristiek tijdsgedrag. Om de effecten van de verschillende simulatieslagen beter in beeld te brengen, worden in deze paragraaf wel tijdvertragingen gebruikt.
De waveform van code 3.11 staat in figuur 3.7. Functioneel is het resultaat exact gelijk aan dat van figuur 3.6. Alleen zijn de veranderingen bij een bepaald tijdstip nu uitgesmeerd over een langere tijd. De spike is nu zichtbaar bij 42 ns en is 1 ns breed.

Code 3.11 : De exnor met vijf parallelle signaaltoewijzingen met tijdvertraging.

```
1  architecture gedrag of exnor is
2    signal na, nb, x, y : std_logic;
3  begin
4    c1: na <= not a after 1 ns;
5    c5: z  <= x or y after 1 ns;
6    c3: x  <= a and b after 1 ns;
7    c4: y  <= na and nb after 1 ns;
8    c2: nb <= not b after 1 ns;
9  end architecture gedrag;
```



Figuur 3.7 : De waveform bij de simulatie van de exnor met vijf parallelle processen met tijdvertraging.

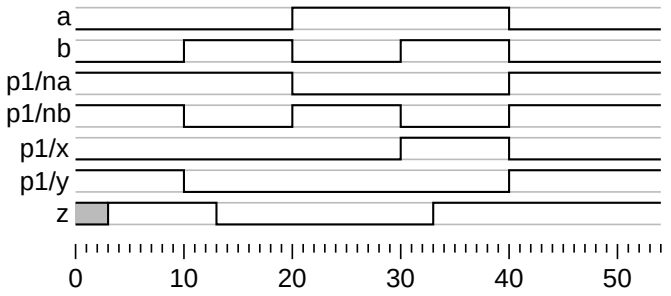
De volgorde van de parallelle signaaltoewijzingen doet er niet toe. De volgorde waarin de processen geëvalueerd worden, kan een hele andere zijn dan de code suggereert. Op tijdstip `0` ns worden `a` en `b` beide laag, dit triggert de processen `c1`, `c2` en `c3`. Bij de evaluatie van de processen worden de huidige waarden van de signalen gebruikt. De nieuwe waarden worden pas aan het eind van een simulatieslag toegekend. De volgorde waarin de toewijzingen staan, heeft daarom geen invloed op het resultaat.

Exnor met sequentieel proces en variabele toewijzingen

In code 3.12 staat een gedragsbeschrijving van de exnor met een proces en met dezelfde logische vergelijkingen als in code 3.10 zijn gebruikt. In plaats van de interne signalen na, nb, x en y worden er nu vier lokale variabelen na, nb, x en y gebruikt. Het simulatieresultaat staat in figuur 3.8. In de waveform staan de ingangssignalen a en b, het uitgangssignaal z, en de lokale variabelen uit proces p1.

Code 3.12 : De exnor met een proces en lokale variabelen.

```
 1  architecture gedrag of exnor is
 2  begin
 3    p1: process (a,b) is
 4      variable na, nb, x, y : std_logic;
 5    begin
 6      na := not a;
 7      nb := not b;
 8      x  := a and b;
 9      y  := na and nb;
10      z  <= x or y after 3 ns;
11    end process p1;
12  end architecture gedrag;
```



Figuur 3.8 : De waveform van de exnor met een proces en lokale variabelen.

De vier variabele toewijzingen worden altijd direct uitgevoerd. In één simulatieslag worden nieuwe waarde berekend voor na, nb, y, x en de toekomstige waarde van signaal z. De toewijzing aan y komt na de toewijzingen aan na en nb. Bij de toewijzing aan y worden dus de nieuwe waarden van na en nb gebruikt. Op dezelfde manier gebruikt de signaaltoewijzing aan z ook de nieuwe waarden van x en y.

Exnor met proces en variabele toewijzingen in de verkeerde volgorde

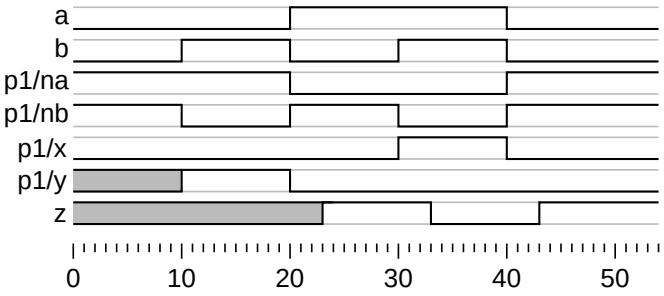
In code 3.13 is de volgorde van de toewijzingen in het proces veranderd. Het simulatieresultaat staat in figuur 3.9 en toont aan dat dit geen correcte beschrijving is van de exnor. De toewijzing aan y staat na de toewijzing aan na en voor die aan nb. Bij het berekenen wordt de oude waarde van nb gebruikt en de nieuwe waarde van na. De toekenning aan z staat voor die aan x en y, zodat hier de oude waarden van x en y worden gebruikt.

Een probleem is dat de synthesizers van de foute beschrijving toch een exnor maken. Dat lijkt in eerste instantie plezierig, maar het veroorzaakt dat de simulaties van de gedragsbeschrijving en die van het gesynthetiseerde netwerk verschillen. Dat is niet wenselijk. De ontwerper moet hier rekening mee houden. Meestal zal een ontwerper de vergelijkingen in de juiste volgorde zetten. Het is namelijk niet logisch om dat anders te doen. Bovendien ziet een goede ontwerper bij de simulatie dat het niet klopt. Belangrijk is dat de ontwerper zich bewust is van het feit dat er bij foutieve beschrijvingen verschillen tussen het gesynthetiseerde netwerk en de simulatie ontstaan.

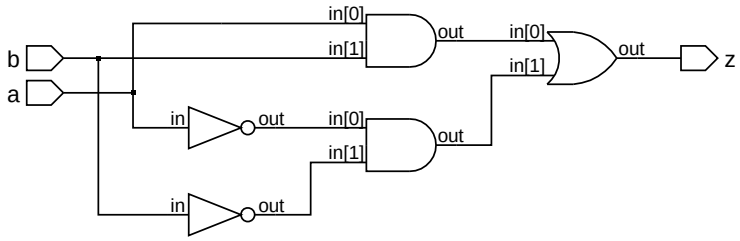
Figuur 3.10 toont het syntheseresultaat van code 3.10, 3.12 en code 3.13. In al deze drie gevallen levert het dezelfde exnor op, dus ook voor de foutieve beschrijving uit code 3.13.

Code 3.13 : De exnor met de toewijzingen in een verkeerde volgorde.

```
1  architecture foutief of exnor is
2  begin
3    p1: process (a,b) is
4      variable na, nb, x, y : std_logic;
5    begin
6      na := not a;
7      z  <= x or y after 3 ns;
8      x  := a and b;
9      y  := na and nb;
10     nb := not b;
11   end process p1;
12 end architecture foutief;
```



Figuur 3.9 : De waveform van de exnor met variabelen in de verkeerde volgorde.



Figuur 3.10 : Het syntheseresultaat van code 3.10, 3.12 en de foute beschrijving van code 3.13 is identiek.

Exnor met sequentieel proces en afhankelijke signaaltoewijzingen

In code 3.14 worden in plaats van variabelen vier interne signalen `na`, `nb`, `x` en `y` gebruikt. Deze vier signalen krijgen hun waarde in proces `p1`. Ondanks dat de logische vergelijkingen overkomen met figuur 3.5 en dat de code veel overeenkomsten heeft met 3.10, is deze beschrijving niet correct. Dat is ook te zien in de waveform van de simulatie van figuur 3.11.

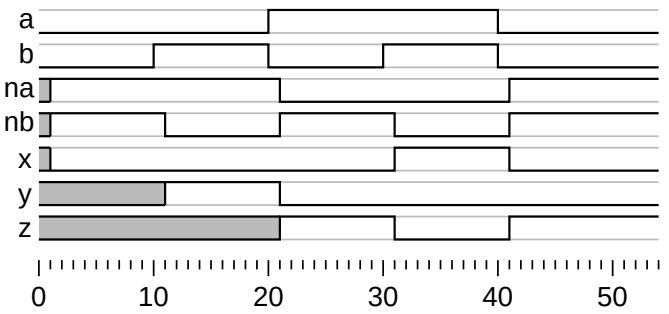
Aanvankelijk zijn alle signalen niet geïnitialiseerd. Op tijdstip `0` ns veranderen de ingangssignalen `a` en `b` en triggeren proces `p1`. Met de nieuwe waarden van `a` en `b` en de huidige waarden van de signalen `na`, `nb`, `x` en `y` worden de nieuwe waarden van `na`, `nb`, `x`, `y` en `z` berekend. De signalen `na`, `nb` en `x` veranderen na 1 ns. Signaal `y` hangt af van `na` en `nb`; omdat bij de toewijzing deze signalen ongeïnitialiseerd waren, blijft `y` ook ongeïnitialiseerd. De uitgang `z` blijft ook ongeïnitialiseerd omdat `x` en `y` ook ongeïnitialiseerd waren.

Daarna gebeurt er niets meer totdat op tijdstip `10` ns signaal `b` hoog wordt. Op dat moment wordt het proces `p1` opnieuw geëvalueerd. Bij de berekening van de waarde van `y` worden de oude waarden van `na` en `nb` gebruikt. Signaal `y` wordt pas 1 ns later hoog. De uitgang `z` blijft ongeïnitialiseerd omdat de oude waarde van `y` ook nog ongeïnitialiseerd was.

Op het tijdstip `20` ns veranderen de signalen `a` en `b` en wordt het proces weer getriggerd. Wederom worden uit de nieuwe waarden van `a` en `b` en de oude waarden

Code 3.14 : De exnor met een proces en afhankelijke signaaltoewijzingen.

```
 1  architecture foutief2 of exnor is
 2    signal na, nb, x, y : std_logic;
 3  begin
 4    p1: process (a,b) is
 5    begin
 6      na <= not a after 1 ns;
 7      nb <= not b after 1 ns;
 8      x  <= a and b after 1 ns;
 9      y  <= na and nb after 1 ns;
10      z  <= x or y after 1 ns;
11    end process p1;
12  end architecture foutief2;
```



Figuur 3.11 : De waveform van de exnor met een proces en afhankelijke signaaltoewijzingen.

van de signalen na, nb, x en y nieuwe signaalwaarden berekend. De waarde van z hangt af van de oude waarde van x en de oude waarde van y. Op tijdstip 21 ns wordt z hoog.
De uitkomst van deze gedragsbeschrijving levert onzin op, het uitgangssignaal had laag moeten zijn. Er worden wel nieuwe waarden voor de signalen uitgerekend, maar die nieuwe waarden worden niet gebruikt. Pas als het proces opnieuw getriggerd wordt, wordt er iets met deze waarden gedaan.
De fout in de code 3.14 wordt gecorrigeerd door aan de gevoeligheidslijst de signalen na, nb, x, y toe te voegen. Bij een verandering van één of meer van deze signalen wordt het proces opnieuw getriggerd en de signalen worden opnieuw berekend met hun meest recente waarden. Het syntheseresultaat is in dat geval gelijk aan dat van code 3.11 en de waveform is identiek aan figuur 3.7.

3.6 Conclusie uit beschrijvingen exnor

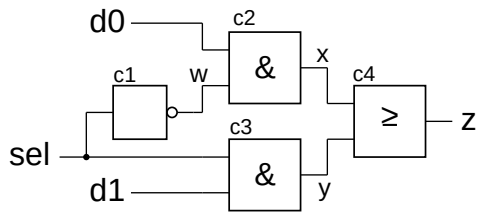
In de vorige paragraaf staan verschillende beschrijvingen van een exnor, die gebaseerd zijn op een aantal logische vergelijkingen. Sommige voorbeelden zijn correct, andere lijken op het eerste gezicht correct, maar zijn dat niet. De dataflowbeschrijving van code 3.10 met parallelle signaaltoewijzingen en de gedragsbeschrijving van code 3.12 met een proces en variabelen in de juiste volgorde zijn beide correct.

Het nadeel van parallelle signaaltoewijzingen is dat het ontwerp uit veel processen bestaat en dat er meer simulatieslagen nodig zijn. De simulatie zal langer duren. Wel geeft dit meer informatie over het werkelijke gedrag. Bij simulatie van de gedragsbeschrijving is in figuur 3.8 geen spike aanwezig. Het modelleren van vertragingstijden en het tijdsgedrag is bij het beschrijven van VHDL overigens niet zinvol, omdat de synthesizer er vaak een ander netwerk van maakt met een totaal ander tijdsgedrag.
Bij de beschrijvingen met een proces staan de toewijzingen in een sequentiële omgeving. De volgorde van de toewijzingen is essentieel voor het functionele gedrag. Het voordeel van deze beschrijvingen is dat er bij de simulatie minder simulatieslagen nodig zijn; de simulatie duurt korter.

Afhankelijke signaaltoewijzingen binnen een proces zijn lastig te interpreteren. Enerzijds worden de opdrachten sequentieel doorlopen; ze staan immers in een proces. Anderzijds worden de toekenningen pas uitgevoerd, nadat alle processen geëvalueerd zijn. Hoofdstuk 4 geeft een aantal voorbeelden van synchrone schakelingen waarbij afhankelijke signaaltoewijzingen binnen een proces wel nuttig zijn, zoals bij de synchronizer van code 4.31.

Code 3.15 : De entity van de multiplexer.

```
1  entity mux is
2    port (
3      sel    : in  std_logic;
4      d0, d1 : in  std_logic;
5      z      : out std_logic
6    );
7  end entity mux;
```



Figuur 3.12 : Het schema van de multiplexer.

Code 3.16 : Een multiplexer met vier parallelle signaaltoewijzingen.

```
1  architecture gedrag of mux is
2    signal w, x, y : std_logic;
3  begin
4    c1 : w <= not sel;
5    c2 : x <= d0 and w;
6    c3 : y <= d1 and sel;
7    c4 : z <= x or y;
8  end architecture gedrag;
```

Code 3.17 : Een multiplexer met een proces en sequentiële toewijzingen.

```
1  architecture gedrag of mux is
2  begin
3    p1: process (sel,d0,d1) is
4      variable w, x, y : std_logic;
5    begin
6      w := not sel;
7      x := d0 and w;
8      y := d1 and sel;
9      z <= x or y;
10   end process p1;
11 end architecture gedrag;
```

Gebruik zodoende voor het vastleggen van het gedrag van een combinatorische schakeling altijd een architectuur met parallelle signaaltoewijzingen of een proces met de variabelen in de juiste volgorde. Code 3.16 en code 3.17 beschrijven een 2-input multiplexer met behulp van deze twee ontwerpstijlen. Beide zijn gebaseerd op het schema van figuur 3.12. In code 3.15 staat de entity van de multiplexer.

Code 3.18 : Een multiplexer met een enkele signaaltoewijzing.

```
1  architecture gedrag of mux is
2  begin
3    cs : z <= (d0 and (not sel)) or (d1 and sel);
4  end architecture gedrag;
```

In het geval dat de logische bewerkingen niet te ingewikkeld zijn, is een beschrijving met een enkele parallelle signaaltoewijzing een goed alternatief. Code 3.18 komt volledig overeen met code 3.17. De signaaltoewijzing `cs` en proces `p1` wachten alle twee op veranderingen van de ingangssignalen `sel`, `d0` en `d1` en voeren de-

zelfde logische bewerking uit. De simulatie zal, ook wat betreft de delta-cycle's, exact gelijk zijn.

3.7 Meervoudige toewijzingen aan hetzelfde signaal

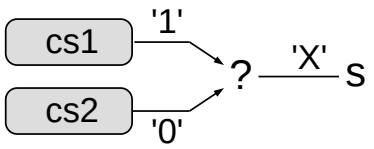
Signalen verschillen in heel veel opzichten van variabelen. Meerdere toewijzingen aan een variabele geeft geen enkele probleem. Na elke toewijzing heeft de variabele een nieuwe waarde. Omdat bij een signaaltoewijzing het signaal nooit direct een nieuwe waarde krijgt, zijn er afspraken nodig voor meervoudige toewijzingen aan één en hetzelfde signaal. Deze afspraken zijn anders binnen een parallelle en een sequentiële omgeving.

Toewijzingen aan hetzelfde signaal in een parallelle omgeving

In een parallelle omgeving is het niet mogelijk meerdere toewijzingen te doen aan hetzelfde signaal. Alle processen worden immers — schijnbaar — gelijktijdig uitgevoerd. Code 3.19 bevat twee parallelle signaaltoewijzingen `cs1` en `cs2`, die beide signaal `s` aansturen. Toewijzing `cs1` maakt signaal `s` hoog en `cs2` maakt het laag. Het is niet duidelijk welke van de twee wint. Omdat `s` van het type `std_logic` is, zal de uitkomst `'X'` zijn.

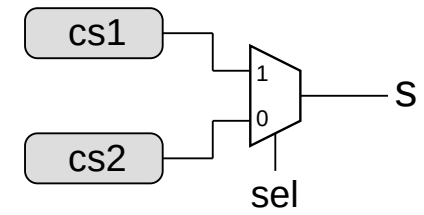
Code 3.19 : Code met twee parallelle signaaltoewijzingen.

```
architecture a of e is
  signal s : std_logic;
begin
    ...
  cs1:  s <= '1';
  cs2:  s <= '0';
    ...
end architecture a;
```



Figuur 3.13 : Het effect van twee parallelle signaaltoewijzingen.

Omdat deze situatie zich juist bij hardware voordoet — bijvoorbeeld een *wired-AND* — is er binnen VHDL een mogelijkheid aanwezig om in een parallelle omgeving toch meerdere toewijzingen aan één signaal te doen. Er is dan een zogenoemd *resolved signal* nodig. De bijbehorende *resolution function* beslist dan wat er bij een conflictsituatie gedaan moet worden.
Signalen van het type `std_logic` zijn *resolved*. Het package `std_logic_1164` bevat een resolutiefunctie, die bepaalt wat de uitkomst is in conflictsituaties. Signalen van het type `std_ulogic` zijn juist niet *resolved*. Als `s` van het type `std_ulogic` is, geeft de compiler de foutmelding: `"Nonresolved signal 's' has multiple sources"`.
Een oplossing om toch vanuit twee processen hetzelfde signaal aan te sturen is om een multiplexer of tristatebuffers te gebruiken. Figuur 3.14 geeft de oplossing met de multiplexer en in code 3.20 is daar een conditional signal assignment voor gebruikt.
De oplossing met de multiplexer werkt ook bij signalen van het *unresolved* type `std_ulogic`. Alleen het proces `cs3` kent immers een waarde aan `s` toe.
De twee tristatebuffers in figuur 3.15 regelen of proces `cs1` of proces `cs2` met signaal

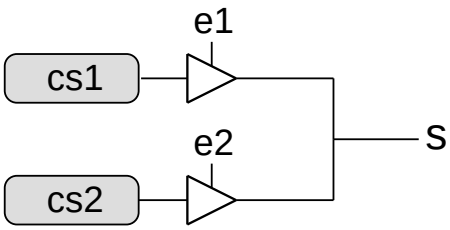


Figuur 3.14 : **Toewijzen vanuit twee processen aan een signaal met behulp van een multiplexer.**

Code 3.20 : **Toewijzen aan een signaal met een multiplexer.**

```
architecture a of e is
  signal s,s1,s2,sel : std_logic;
begin
    ...
  cs1: s1 <= '1';
  cs2: s2 <= '0';
  cs3: s  <= s1 when sel ='1' else s2;
    ...
end architecture a;
```

`s` verbonden is. Als signaal `e1` hoog en `e2` laag is, is signaal `s` hoog. Evenzo is `s` laag als `e2` hoog is en `e1` laag is. Als `e1` en `e2` beide hoog zijn, is een er conflictsituatie en zal `s` ongedefinieerd (`'X'`) zijn. De ontwerper moet er voor zorgen dat deze situatie niet voorkomt. Als `e1` en `e2` allebei laag zijn, is signaal `s` hoogimpedant (`'Z'`).



Figuur 3.15 : **Toewijzen vanuit twee processen aan een signaal met behulp van twee tristatebuffers.**

Code 3.21 : **Toewijzen aan een signaal met behulp van tristatebuffers.**

```
architecture a of e is
  signal s,s1,s2,e1,e2 : std_logic;
begin
    ...
  cs1: s1 <= '1';
  cs2: s2 <= '0';
  cs3: s  <= s1 when e1 ='1' else 'Z';
  cs4: s  <= s2 when e2 ='1' else 'Z';
    ...
end architecture a;
```

Toewijzingen aan hetzelfde signaal in een sequentiële omgeving

Omdat in een sequentiële omgeving de toewijzingen na elkaar gebeuren, leveren toewijzingen aan hetzelfde signaal geen conflicten op. In code 3.22 staat een proces `p1` met twee signaaltoewijzingen aan signaal `p`. In dit geval is de tweede toewijzing dominant en krijgt `p` de waarde '0'.

Een praktische toepassing voor het vaker toewijzen aan hetzelfde signaal staat in code 3.23. Het proces `fsm` bevat een omvangrijk case-statement. In plaats van bij alle keuzemogelijkheden signaal `u` expliciet een waarde te geven, is het eenvoudiger om aan het begin van proces `fsm` `u` laag te maken en alleen bij `S8` signaal `u` hoog te maken.
Een tweede toepassing is het maken van waveforms. In de testbench van code 2.3 wordt aan de signalen `a`, `b` en `ci` meerdere keren een waarde toegekend. Signaal `a` is eerst laag, na 200 ns hoog en na 400 ns weer laag. In code 2.3 is dit gedaan met een proces en wait-statements en in code 3.25 met drie samengestelde signaaltoewijzingen.

Code 3.22 : Proces met twee toewijzingen aan hetzelfde signaal.

```
architecture a of e is
  signal p : std_logic;
begin
    ...
  p1: process is
      ...
    p <= '1';
    p <= '0';
      ...
  end process p1;
    ...
end architecture a;
```

Code 3.23 : Toepassing voor sequentiële toewijzingen aan hetzelfde signaal.

```
architecture a of e is
  signal u : std_logic;
begin
    ...
  fsm: process is
    u <= '0';
    case state is
    when S0 =>
        ...
    when S8 =>
      state <= S9;
      u <= '1';
    when S9 =>
        ...
    end case;
  end process fsm;
end architecture a;
```

3.8 Tijdvertragingen en samengestelde signaaltoewijzingen

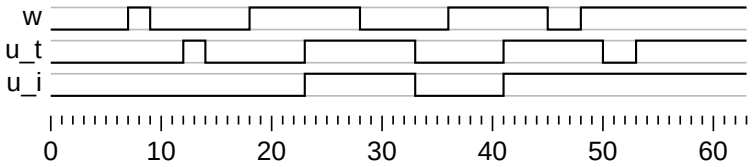
VHDL kent drie soorten tijdvertragingen: de in paragraaf 3.3 besproken *delta delay*, de *inertial delay* en de *transport delay*.

- *delta delay*
  Er is geen echte tijdvertraging. De feitelijk toekenning gebeurt niet direct, maar aan het einde van de huidige simulatieslag.
  Voorbeeld: `u1 <= w;`
- *inertial delay*
  Deze vertraging geeft alleen de veranderingen door, die gedurende de gegeven vertraging stand houden. Dit is het standaard vertragingsmodel en is bedoeld voor de modellering van tijdvertragingen bij logische poorten.
  Voorbeelden:
  ```
  u1 <= w after 5 ns;
  u2 <= inertial w after 5 ns;
  ```
  Het sleutelwoord **`inertial`** is vanaf 1993 aan VHDL toegevoegd.
- *transport delay*
  Het sleutelwoord **`transport`** geeft aan dat alle veranderingen worden doorgegeven. Deze vertraging is bedoeld voor het modelleren van lange lijnen.
  Voorbeeld: `u1 <= transport w after 5 ns;`

Code 3.24 geeft het ingangssignaal `w` op twee manieren door. Bij uitgang `u_i` is een inertial delay en bij uitgang `u_t` is een transport delay gebruikt. Figuur 3.16 laat het verschil tussen deze vertragingen zien.
In code 3.24 wordt het sleutelwoord **`transport`** bij een parallelle signaaltoewijzing toegepast. Het kan ook bij een gewone signaaltoewijzing in een proces gebruikt worden. Bij meerdere toewijzingen aan een enkel signaal in een proces geldt dat, als er inertial delays worden gebruikt, de laatste toewijzing de voorafgaande toewijzingen overbodig maakt. Bij code 3.22 wordt signaal `p` daarom laag.

Code 3.24 : Verschil tussen inertial delay en transport delay.

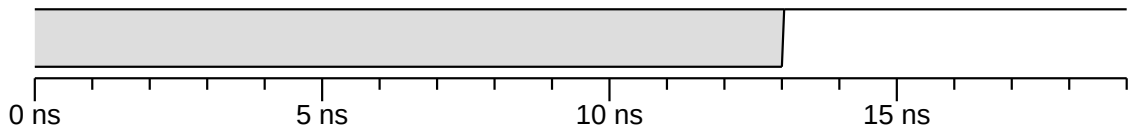
```
1  entity delay is
2    port (
3      w   : in  std_logic;
4      u_t : out std_logic;
5      u_i : out std_logic
6    );
7  end entity delay;
8
9  architecture demo of delay is
10 begin
11   u_t <= transport w after 5 ns;
12   u_i <= w after 5 ns;
13 end architecture demo;
```



Figuur 3.16 : Het verschil tussen inertial delay en transport delay.
Bij signaal `u_t` is transport delay gebruikt. Alle informatie van het signaal `w` wordt met een vertraging van 5 ns aan `u_t` doorgegeven. Bij signaal `u_i` is inertial delay gebruikt. Alleen informatie, die langer dan 5 ns bestaat, wordt doorgegeven aan `u_i`.

Proces `i1` in figuur 3.17 bevat vijf signaaltoewijzingen met een standaard inertial delay. Daardoor overschrijft de laatste toewijzing de voorafgaande vier toewijzingen. Signaal `s1` blijft ongeïnitialiseerd tot 13 ns en wordt daarna hoog. De processen `i2` en `i3` zijn identiek met proces `i1`.

Figuur 3.18 geeft een voorbeeld van een proces met een samengesteld signaal. Proces `t1` bevat vijf signaaltoewijzingen. Bij de laatste vier staat het sleutelwoord **`transport`** en worden daardoor toegevoegd aan de eerste toewijzing. De waveform van `s2` is bij alle drie de processen hetzelfde. De processen `t2` en `t3` met de samengestelde signaaltoewijzingen zijn identiek aan proces `t1`.



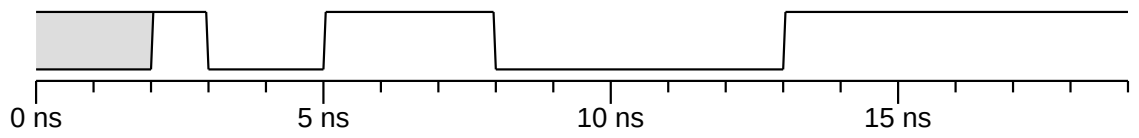

```
i1: process is
begin
  s1 <= '1' after 2 ns;
  s1 <= '0' after 3 ns;
  s1 <= '1' after 5 ns;
  s1 <= '0' after 8 ns;
  s1 <= '1' after 13 ns;
  wait;
end process;
```

```
i2: process is
begin
  s1 <= '0' after 13 ns;
  wait;
end process;
```

```
i3: s1 <= '0' after 13 ns;
```

Figuur 3.17 : Een samengestelde signaaltoewijzing met inertial delay.
Proces `i1` heeft vier signaaltoewijzingen met inertial delay. Proces `i2` en `i3` zijn identiek aan proces `i1`. De drie processen leveren dezelfde waveform op.

Proces `t3` is een parallelle signaaltoewijzing, die exact hetzelfde doet als proces `t1`. Deze signaaltoewijzing is veel eenvoudiger dan het complete proces `t1`. Het is daarom verstandig samengestelde signalen met een parallelle signaaltoewijzing




```
t1: process is
begin
  s2 <= '1' after 2 ns;
  s2 <= transport '0' after 3 ns;
  s2 <= transport '1' after 5 ns;
  s2 <= transport '0' after 8 ns;
  s2 <= transport '1' after 13 ns;
  wait;
end process;
```

```
t2: process is
begin
  s2 <= '1' after 2 ns,
        '0' after 3 ns,
        '1' after 5 ns,
        '0' after 8 ns,
        '1' after 13 ns;
  wait;
end process;
```

```
t3: s2 <= '1' after 2 ns,
          '0' after 3 ns,
          '1' after 5 ns,
          '0' after 8 ns,
          '1' after 13 ns;
```

Figuur 3.18 : Een samengestelde signaaltoewijzing met transport delay.
Proces `i1` heeft vier signaaltoewijzingen met transport delay. Proces `i2` en `i3` zijn alternatieven voor proces `i1`. De drie processen leveren dezelfde waveform op.

te beschrijven. Dit maakt opnieuw duidelijk dat een parallelle signaaltoewijzing, net als alle andere parallelle opdrachten, altijd te herschrijven is met een expliciet proces.

Alternatieve signaalgenerator voor testbench full-adder

De signaalgenerator uit de testbench voor de full-adder uit code 2.3 kan ook gerealiseerd worden met de drie parallelle, samengestelde signaaltoewijzingen uit code 3.25.

Code 3.25 : Alternatieve signaalgenerator voor testbench full-adder.

```
-- signaal_generator
a  <= '0', '1' after 200 ns, '0' after 400 ns;
b  <= '0', '1' after 100 ns, '0' after 300 ns;
ci <= '0', '1' after  50 ns, '0' after 150 ns,
           '1' after 250 ns, '0' after 350 ns, '1' after 450 ns;
```

De waveforms van deze signalen zijn exact gelijk aan die uit figuur 2.12. In principe maakt het daarom niet uit welke methode toegepast wordt. De ervaring leert echter dat de aanpak met samengestelde signalen alleen bruikbaar is bij een eenvoudige test. Met name als er een onderlinge relatie is tussen de samengestelde signalen is het opstellen complex. Het proces `signal_generator` met wait-statements uit code 2.3 is veel flexibeler.

4

Synthese

Doelstelling

In dit hoofdstuk leer je wat synthese is, wat synthetiseerbaarheid is en welke subset van VHDL synthetiseerbaar is. Je leert onderscheid te maken tussen de VHDL-beschrijvingen van synchrone en asynchrone sequentiële schakelingen en combinatorische schakelingen.

Onderwerpen

De behandelde onderwerpen zijn:

- Het concept synthese en de ontwikkelingen op het gebied van synthese.
- Welke VHDL in het algemeen, wel en niet synthetiseerbaar is.
- Een sjabloon voor synthetiseerbare combinatorische schakelingen.
- Sjablonen voor synthetiseerbare sequentiële schakelingen.
- De modellering van een synchroon en een asynchroon resetsignaal bij sequentiële schakelingen.
- De voor- en nadelen van een asynchrone reset.
- De sequentiële conditionele opdrachten **`if`**, **`else`**, **`elsif`**, **`case`**.
- De parallelle conditionele opdrachten **`when else`**, **`with select`**.
- De sequentiële herhalingsopdrachten **`for loop`**, **`while loop`**.
- De parallelle herhalingsopdrachten **`for generate`**, **`if generate`**.

Synthese is het automatisch omzetten van een hogere naar een lagere abstractie. De taal VHDL is geschikt voor allerlei abstracties: systeemniveau, RTL-niveau en logische poortniveau en zelfs transistorniveau.

Het omzetten van het systeemniveau naar het RTL-niveau, wordt *high level synthesis* of *behavorial synthesis* genoemd. Er zijn de aflopen jaren allerlei hulpmiddelen ontwikkeld, die dit traject moeten ondersteunen. Hoewel er zeer interessante ontwikkelingen zijn en er nog steeds nieuwe mogelijkheden bij komen, heeft dit nog geen grote vlucht genomen. Daarvoor zijn een aantal oorzaken: deze hulpmiddelen zijn zeer prijzig; er moeten zowel hardware- als software-ontwikkelaars mee kunnen werken; er zijn nauwelijks standaarden; het traject kan bijna nooit helemaal geautomatiseerd worden en de ontwerper wil en moet bepaalde ontwerpbeslissingen zelf nemen.

Het omzetten van het abstracte ontwerpen naar RTL-niveau naar logische poorten noemt men *RTL synthesis* of logische synthese. Deze synthese van hardwarebeschrijvingstalen heeft wel een grote vlucht genomen. Belangrijke redenen zijn dat VHDL en Verilog IEEE-standaarden zijn: dat er veel — betaalbare — hulp-

middelen beschikbaar zijn; dat er met deze ontwikkelomgevingen heel complexe systemen gemaakt kunnen worden; dat er veel softcores beschikbaar zijn en dat allerlei andere applicaties, zoals Matlab en Labview, VHDL genereren .

EDA staat voor *Electronic Design Automation* en omvat alle softwarepakketten waarmee elektronische systemen ontworpen kunnen worden, zoals PCB-programma's en simulatiepakketten,

Tabel 4.1 geeft de VHDL-syntheseprogramma's van een aantal EDA-producenten en organisaties. De laatste drie zijn algemene ontwikkelomgevingen van FPGA-fabrikanten en de eerste vier zijn algemene RTL-synthesizers.

Tabel 4.1 : De synthese-programma's van een aantal leveranciers.

Synthesizer	Leverancier
Design Compiler	Synopsys
Synplify	Synplicity[1]
Leonardo Spectrum[2]	Mentor Graphics
Precision RTL[2]	Mentor Graphics
Webpack	Xilinx
Quartus II	Altera
Libero	Actel

[1] In maart 2008 is Synplicity door Synopsys overgenomen.
[2] Precision RTL is de opvolger van Leonardo Spectrum.

VHDL is oorspronkelijk ontworpen voor het beschrijven en simuleren van digitale hardware. Later is bedacht dat de taal ook voor synthese gebruikt kan worden. Lang niet alle VHDL is synthetiseerbaar. Bovendien interpreteren de verschillende synthesizers de code soms anders.
Het is zeer belangrijk dat men zich realiseert dat VHDL niet speciaal voor synthese is ontwikkeld en dat deze taal constructies bevat die niet of moeilijk synthetiseerbaar zijn. Een ontwerp maken dat niet synthetiseerbaar is, is zinloos. Het is bovendien zonde van de tijd als er een herontwerp nodig is om het synthetiseerbaar te maken. Een goede ontwerper weet wat wel en niet synthetiseerbaar is en gebruikt een ontwerpstijl die altijd gesynthetiseerd kan worden. In de praktijk wordt voor het ontwerp een subset van VHDL gebruikt. Bijzondere, niet synthetiseerbare VHDL-constructies zijn nuttig bij het schrijven van een testbench.

Hoofdstuk 2 geeft een groot aantal VHDL-beschrijvingen met het syntheseresultaat. Dit hoofdstuk behandelt de regels waaraan de taal VHDL moet voldoen om omgezet te kunnen worden naar een netwerk. Daarnaast geeft dit hoofdstuk een aantal sjablonen voor het opstellen van een synthetiseerbare gedragsbeschrijving van combinatorische en sequentiële schakelingen.

4.1 Synthetiseerbare VHDL

De vraag welke onderdelen van VHDL wel of niet synthetiseerbaar zijn, is om een aantal redenen moeilijk in algemene bewoordingen vast te leggen.
Ten eerste hangt dat af van het gebruikte synthesesysteem. In de meeste gevallen leveren de verschillende synthesizers een vergelijkbaar resultaat op. Echter in het grensgebied, van wat juist wel of niet synthetiseerbaar is, wijken de uitkomsten vaak af.
Ten tweede zijn de synthesesystemen nog steeds in ontwikkeling. Daardoor kunnen er aanzienlijke verschillen bestaan tussen de verschillende versies.

Ten derde is — zoals al eerder is opgemerkt — VHDL niet speciaal voor synthese geschreven. VHDL kan gebruikt worden op verschillende ontwerpniveaus, bij vele ontwerptechnologieën en ondersteunt vele ontwerpmethodieken. VHDL wordt toegepast op systeemniveau en op transistorniveau; voor ASIC's en voor FPGA's; bij synchrone en bij asynchrone ontwerpen; voor een top-down strategie en een bottom-up strategie. Door deze brede inzetbaarheid is het mogelijk om VHDL-code te schrijven die niet synthetiseerbaar is.

Hoewel de laatste jaren de synthesesystemen aanzienlijk verbeterd zijn en er gewerkt is aan standaardisatie, zijn er nog steeds verschillen. Het is vaak makkelijker aan te geven welke constructies niet synthetiseerbaar zijn, dan uit te leggen welke wel synthetiseerbaar zijn. Vandaar dat er in de volgende paragraaf eerst aandacht besteed wordt aan het niet synthetiseerbaar zijn. Een handige vuistregel, die aangeeft of een proces wel of niet synthetiseerbaar is, luidt:

> *Heb je een idee hoe je het proces zou kunnen realiseren, dan is het synthetiseerbaar.*
> *Heb je dat niet, dan is het niet synthetiseerbaar.*

Bij deze vuistregel gaat het om het herkennen van tellers, toestandsmachines, geheugenelementen, multiplexers, comparatoren en alle andere combinatorische of sequentiële logica in de te synthetiseren gedragsbeschrijving.

4.2 Niet synthetiseerbare VHDL

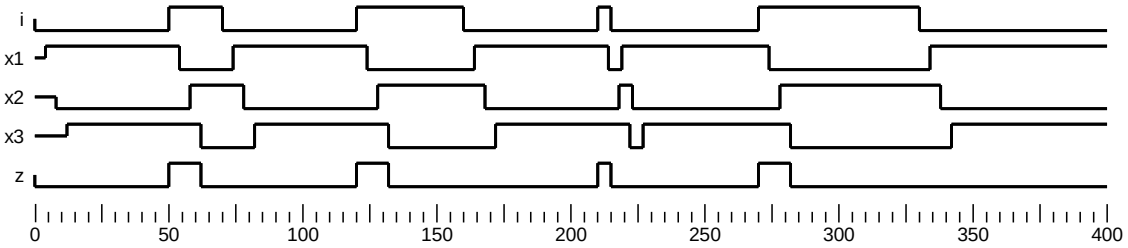
Opvallend is dat juist de onderdelen die te maken hebben met het event-driven simulatiemodel lastig te synthetiseren zijn. Synthesizers hebben vooral moeite met:

- vertragingstijden;
- asynchrone beschrijvingen;
- de initialisatie van variabelen en signalen;
- het simulatiemodel;
- het gebruik van meerdere wait-statements in één proces;
- bepaalde datatypen en sommige rekenkundige bewerkingen;
- dynamische constructies.

Vertragingstijden

Expliciete tijdvertragingen zijn niet synthetiseerbaar. Alle **`after`**-clausules worden bij de synthese genegeerd. De toewijzing `x1 <= i after 4 ns;` wordt vereenvoudigd tot `x1 <= i;`.
In een gedragsbeschrijving heeft een tijdvertraging geen werkelijke betekenis; het maakt het simulatieresultaat eventueel wat realistischer. Het weglaten van de expliciete tijdvertraging heeft tot gevolg dat alle transport- en inertial-delay's worden teruggebracht tot delta-delay's. De sleutelwoorden **`inertial`** en **`transport`** hebben bij synthese geen enkele betekenis.
Dat tijdvertragingen niet synthetiseerbaar zijn, is eenvoudig te begrijpen wanneer men zich realiseert hoe deze gemaakt zouden moeten worden. Er zijn dan buffers of lange lijnen met de gewenste tijdvertraging nodig. Om aan alle mogelijke tijdvertragingen te voldoen, is een extreem grote bibliotheek nodig. Een bijkomend probleem is de nauwkeurigheid van die tijdvertragingen. Voldoet een buffer met een vertraging van 3,8 ns aan de eis van 4 ns of is 4,07 ns ook goed?

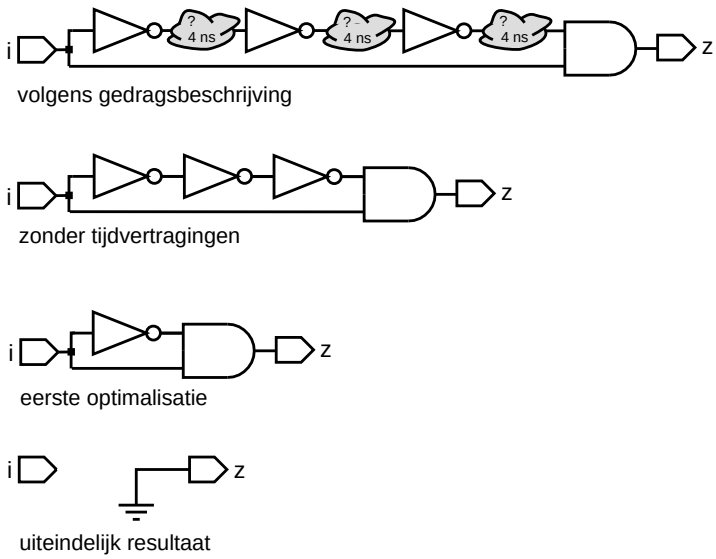
Het negeren van tijdvertragingen veroorzaakt dat de oorspronkelijke VHDL-beschrijving en de gerealiseerde schakeling van elkaar gaan afwijken. Dat is niet wenselijk. Gebruik daarom in het ontwerp nooit tijdvertragingen.



Figuur 4.1 : De simulatie van de gedragsbeschrijving van de one-shot.

Code 4.1 : Gedragsbeschrijving van een one-shot met behulp van tijdvertraging.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity oneshot is
5    port (
6      i : in  std_logic;
7      z : out std_logic
8    );
9  end entity oneshot;
10
11 architecture gedrag of oneshot is
12   signal x1, x2, x3 : std_logic;
13 begin
14   x1 <= not i after 4 ns;
15   x2 <= not x1 after 4 ns;
16   x3 <= not x2 after 4 ns;
17   z  <= i and x3;
18 end architecture gedrag;
```



Figuur 4.2 : Bij synthese worden de tijdvertragingen weggelaten. Dit geeft een opmerkelijke vereenvoudiging.

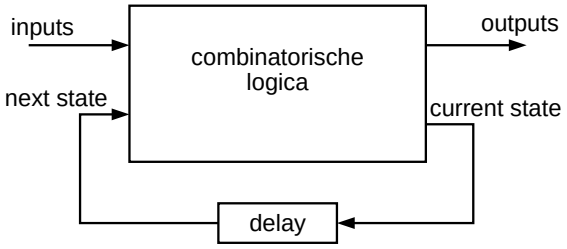
Code 4.1 bevat een gedragsbeschrijving van een one-shot of pulsdetector, die gebaseerd is op tijdvertragingen. In figuur 4.1 staat een simulatieresultaat voor deze pulsdetector. Het idee van de gedragsbeschrijving is dat de drie inverters een inverteerde vertragingslijn vormen.
De synthesizer negeert de **`after`**-clausules en ziet drie inverters, die logisch gezien vervangen kunnen worden door één enkele inverter. De ingangen van de *ideale* AND-functie zijn dan altijd tegengesteld. De uitgang zal dan altijd laag zijn, zoals figuur 4.2 laat zien.

Asynchrone beschrijvingen

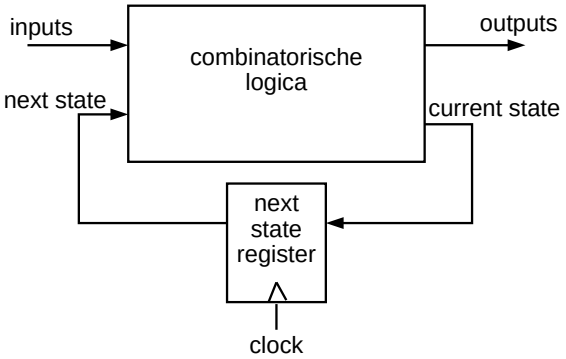
Complexe digitale systemen hebben meestal een kloksignaal. Bij de actieve klokflank zetten deze synchrone systemen nieuw berekende waarden in de registers.

Asynchrone systemen hebben geen kloksignaal. Er is geen vast moment dat er nieuwe informatie verwerkt wordt. Bij asynchrone beschrijving wordt dit opgelost met handshake-protocollen of door vertragingen toe te voegen.

In het Huffmanmodel, zie figuur 4.3, voor asynchrone systemen wordt de huidige toestand via een vertraging teruggekoppeld als nieuwe toestand. Bij synchrone systemen zijn er geklokte registers die de toestand van het systeem onthouden, zoals het model van figuur 4.4 laat zien. De eindige toestandsmachine of *finite state machine* die hiervoor nodig is, komt in hoofdstuk 5 aan de orde.



Figuur 4.3 : Huffmanmodel voor asynchrone systemen.



Figuur 4.4 : Model voor synchrone systemen.

In code 4.2 staat een asynchrone beschrijving van een pulsdetector, die bij de synthese een correct netwerk oplevert. De beschrijving is een asynchrone toestandsmachine. Process `afsm` bepaalt uit de ingang `puls` en de huidige toestand `prs` de volgende toestand `nxt` en de uitgang `puls_detector`. Op regel 39 krijgt de huidige toestand `prs` vertraagd zijn nieuwe waarde.
Het simulatieresultaat staat in figuur 4.5. Omdat er een inertial delay is gebruikt, wordt de puls bij 220 ns niet gezien. Het syntheseresultaat van deze asynchrone beschrijving staat in figuur 4.6. In figuur 4.7 staat een simulatie van het gesynthetiseerde netwerk. Voor alle poorten is de vertragingstijd 2 ns. De puls bij 220 ns van het ingangssignaal levert nu wel een puls in het uitgangssignaal op.

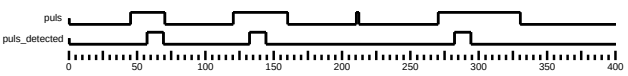
In de gedragsbeschrijving is een eigen typedefinitie gebruikt voor de toestanden. Type `pd_type` is een *enumerate type* dat uit een opsomming van drie waarden bestaat. De synthesizer codeert deze waarden met twee bits: `00`, `01` en `10`. Het gevolg is dat de `WAITP` gelijk is aan `00`, `NEWP` gelijk aan `01` en `OLDP` gelijk aan `10`. Een andere codering levert een ander netwerk op. Als de volgorde van de typedefinitie veranderd wordt in `NEWP`, `WAITP` en `OLDP`, wordt `WAITP` gelijk aan `01` en `NEWP` gelijk aan `00`. De synthese van de beschrijving levert dan het netwerk van figuur 4.8 op. Uitgang `puls_detected` is verbonden met de referentie en is dus altijd laag.

Al in de jaren vijftig zijn er basisconcepten voor asynchrone systemen door David A. Huffman en David E. Muller ontwikkeld. Huffman heeft het vertragingsmodel van figuur 4.3 opgesteld en Muller heeft het naar hem genoemde Muller C-element bedacht dat de basis logische poort is voor asynchrone schakelingen.

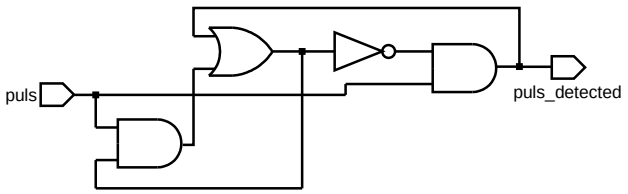
Het gebruik van asynchrone schakelingen wordt absoluut ontraden, omdat deze lastig te begrijpen zijn, bestaande ontwikkelomgevingen niet geschikt zijn voor asynchrone systemen, het testen lastig is en omdat er een overhead aan logica nodig is om het goed te laten functioneren.
Toch zijn er ook redenen om wel asynchroon te ontwerpen: er is geen klok nodig en dus ook geen ingewikkelde klokdistributie; er wordt minder geschakeld en dat betekent een lagere vermogensconsumptie en minder elektromagnetische

Code 4.2 : Gedragsbeschrijving van een asynchrone pulsdetector.

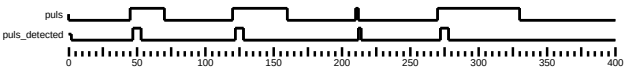
```
1  entity pulsdetector is
2    port (
3      puls          : in  std_logic;
4      puls_detected : out std_logic
5    );
6  end entity pulsdetector;
7
8  architecture gedrag of pulsdetector is
9    type   pd_state is (WAITP, NEWP, OLDP);
10   signal nxt, prs : pd_state;
11 begin
12   afsm: process (prs,puls) is
13   begin
14     case prs is
15     when WAITP =>
16       if puls = '1' then
17         nxt <= NEWP;
18       else
19         nxt <= WAITP;
20       end if;
21       puls_detected <= '0';
22     when NEWP =>
23       if puls = '1' then
24         nxt <= OLDP;
25       else
26         nxt <= WAITP;
27       end if;
28       puls_detected <= '1';
29     when OLDP =>
30       if puls = '1' then
31         nxt <= OLDP;
32       else
33         nxt <= WAITP;
34       end if;
35       puls_detected <= '0';
36     end case;
37   end process afsm;
38
39   prs <= nxt after 5 ns;
40 end architecture gedrag;
```



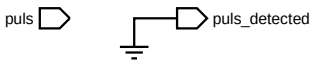
Figuur 4.5 : De simulatie van de pulsdetector uit code 4.2.



Figuur 4.6 : Syntheseresultaat pulsdetector van code 4.2.
De volgorde van toestandscode is hier `WAITP`, `NEWP`, `OLDP`.



Figuur 4.7 : De simulatie van het gesynthetiseerde netwerk.



Figuur 4.8 : Syntheseresultaat pulsdetector met alternatieve toestandscodering.
De volgorde van toestandscode is hier `NEWP`, `WAITP`, `OLDP`.

interferentie. In 1989 is door de VLSI-group van Alain Martin van de California Institute of Technology al een asynchrone microprocessor ontwikkeld. Vanwege het lagere vermogensverbruik gebruikt Philips in consumentenproducten asynchrone technieken.
Ondanks dat er ook redenen zijn om juist wel asynchroon te ontwerpen, zijn de eerder geschetste problemen de reden dat dit boek alleen synchrone ontwerpen toestaat.

Initialisatie van variabelen en signalen

Bij de initialisatie krijgt een variabele of een signaal, als er niet expliciet een beginwaarde is opgegeven, de meest linkse waarde uit het bereik. Voor een `integer` is dat —2147483647 (—$2^{31}+1$); voor een `natural` is dat `0`; voor een `boolean` is dat `false`; en voor `std_ulogic` en `std_logic` is dat `'U'`. De beginwaarde van een niet expliciet geïnitialiseerde integer is standaard dus niet nul.
Synthesizers interpreteren impliciete beginwaarden vaak op een andere manier. Voor de datatypen `std_ulogic` en `std_logic` wordt als beginwaarde `'0'` genomen. Bij integers wordt meestal `0` als beginwaarde genomen.
Door deze andere interpretatie verschilt de gerealiseerde schakeling met die van de oorspronkelijke VHDL-beschrijving. Impliciete en expliciete initialisaties moeten daarom niet gebruikt worden. Voeg in plaats daarvan aan het ontwerp een reset- of presetsignaal toe.

Het simulatiemodel

In paragraaf 3.5 is bij het voorbeeld van de exnor al duidelijk gemaakt dat een gedragsbeschrijving met een fout simulatieresultaat bij de synthese toch een correcte schakeling op kan leveren. De foute beschrijvingen van code 3.13 en 3.14 leveren het netwerk van figuur 3.10 op.

Code 4.3 : Full-adder met variabelen in juiste volgorde.

```
 1  architecture dataflow2 of fulladder is
 2  begin
 3    p1: process (a,b,ci) is
 4      variable p, g, t : std_logic;
 5    begin
 6      p  := a xor b;
 7      t  := p nand ci;
 8      g  := a nand b;
 9      s  <= p xor ci;
10      co <= g nand t;
11    end process p1;
12  end architecture dataflow2;
```

Code 4.4 : Full-adder met toewijzingen in foute volgorde.

```
 1  architecture dataflow2 of fulladder is
 2  begin
 3    p1: process (a,b,ci) is
 4      variable p, g, t : std_logic;
 5    begin
 6      s  <= p xor ci;
 7      co <= g nand t;
 8      t  := p nand ci;
 9      p  := a xor b;
10      g  := a nand b;
11    end process p1;
12  end architecture dataflow2;
```

In code 2.7 staat een dataflow-beschrijving van een full-adder. In code 4.3, 4.4 en 4.5 staan drie alternatieve beschrijvingen met een expliciet proces. Alleen code 4.3 met de variabelen in de juiste volgorde geeft een correcte simulatie. Bij code 4.4 staan de toewijzingen niet in de juiste volgorde en bij code 4.5 worden in het proces afhankelijke signaaltoewijzingen gebruikt.

Bij de synthese wordt er niet naar het simulatiemodel gekeken. De synthesizer ziet in code 2.7, 4.3, 4.4 en 4.5 alleen deze vijf logische relaties:

```
s  ← p xor ci
p  ← a xor b
g  ← a nand b
co ← g nand t
t  ← p nand ci
```

Code 4.5 : Full-adder met afhankelijke signaaltoewijzingen.

```
1  architecture dataflow2 of fulladder is
2    signal p, g, t : std_logic;
3  begin
4    p1: process (a,b,ci) is
5    begin
6      p  <= a xor b;
7      t  <= p nand ci;
8      g  <= a nand b;
9      s  <= p xor ci;
10     co <= g nand t;
11   end process p1;
12 end architecture dataflow2;
```

De volgorde van de toewijzingen doet er toe; het maakt ook niet uit of de toewijzingen in een proces staan en of er variabelen of signalen zijn gebruikt. Sommige synthesizers geven bij code 4.5 de waarschuwing dat de signalen `p`, `t` en `g` ontbreken bij de sensitivity list.

Meerdere wait-statements in één proces

De synthese van het wait-statement levert in een aantal gevallen problemen op.

- **`wait for`** tijdsinterval;
  Dit statement is niet synthetiseerbaar, omdat tijdvertragingen niet synthetiseerbaar zijn.
- **`wait until`** Booleaanse uitdrukking;
  Dit statement is beperkt synthetiseerbaar. Meestal beschrijft dit een klokflank:

  ```
  wait until clk = '1';
  wait until rising_edge(clk);
  wait until clk'event and clk = '1';
  wait until not clk'stable; and clk = '1';
  ```

  In dat geval levert het een synchrone schakeling op. Synthesizers accepteren een soort **`wait until`**'s per proces. In code 4.6 staat een synthetiseerbaar proces dat uit een kloksignaal `t` een tweefasenklok genereert. De fasen veranderen alleen bij de opgaande flank van signaal `t`. Het proces uit code 4.7 is niet synthetiseerbaar omdat de fasen bij de opgaande en bij de neergaande flank van `t` veranderen.
- **`wait on`** sensitivity list;
  Deze wait is in principe synthetiseerbaar, maar wordt meestal alleen toegestaan in de vorm van een impliciete wait.
- **`wait`**;
  Dit statement wordt voornamelijk gebruikt voor het initialiseren van signalen en variabelen en is daarom niet synthetiseerbaar.

Een proces met impliciete wait is over het algemeen synthetiseerbaar. Het is het meest gebruikte wait-statement en wordt door alle VHDL-synthesizers ondersteund.

Code 4.6 : Synthetiseerbaar proces met `wait until`.

```
two_phase: process is
begin
  wait until t = '1';
  phi1 <= '0';
  wait until t = '1';
  phi2 <= '1';
  wait until t = '1';
  phi2 <= '0';
  wait until t = '1';
  phi1 <= '1';
end process two_phase;
```

Code 4.7 : Niet-synthetiseerbaar proces met `wait until`.

```
two_phase: process is
begin
  wait until t = '1';
  phi1 <= '0';
  wait until t = '0';
  phi2 <= '1';
  wait until t = '1';
  phi2 <= '0';
  wait until t = '0';
  phi1 <= '1';
end process two_phase;
```

Datatypen en rekenkundige bewerkingen

Signalen en variabelen van het type `boolean`, `bit`, `integer`, `natural` zijn synthetiseerbaar. Dat geldt eveneens voor signalen en variabelen van het type `std_ulogic` en daarmee ook voor voor de daarvan afgeleide typen `std_logic`, `std_logic_vector`, `unsigned` en `signed`. Hoofdstuk 10 bespreekt de packages `ieee_stdlogic_1164` en `numeric_std` waarin deze typen zijn gedefinieerd.
Niet alle negen logische niveaus van `std_ulogic` hebben dezelfde betekenis bij een simulatie als bij de synthese. Bij een simulatie betekent `'U'` ongeïnitialiseerd en geeft de `X` aan dat er een conflict is en dat het signaal ongedefinieerd is. In werkelijkheid zal een signaal altijd hoog of laag zijn. Een signaal dat `'U'` of `'X'` is, zal dus altijd hoog of laag zijn. Synthesizers gebruiken van `std_ulogic` alleen de `'0'`, de `'1'` en de `'Z'`. De `'X'` en `'U'` worden geïnterpreteerd als een `'0'`.
De *don't care* of `'-'` is ook synthetiseerbaar, maar moet op een speciale manier gebruikt worden. Dit wordt in hoofdstuk 10 bij de bespreking van de functie `std_match` uitgelegd.

De signalen en variabelen van het type `integer` en `natural` zijn synthetiseerbaar. Het is wel verstandig om een range aan te geven. Standaard zijn integers 32 bits breed. Als de breedte niet beperkt wordt, kan de synthesizer voor deze integers 32 bits reserveren. Dat kan betekenen dat er veel meer logica en flipfloppen gemaakt worden dan nodig is. Hieronder staan een aantal declaraties van integers met een range:

Als IEEE een nieuwe VHDL-versie goedkeurt, duurt het vaak nog jaren voordat de leveranciers hun ontwikkelomgevingen daarop hebben aangepast. Bovendien duurt het nog eens vele jaren voor de ontwerpers de nieuwe versie toepassen in nieuwe ontwerpen.

```
signal int8  : integer range 0 to 255;
signal int4  : integer range 0 to 15;
signal bcd   : integer range 0 to 9;
signal index : integer range 1 to 8;
```

De signalen `int8` en `int4` zijn respectievelijk acht en vier bits breed. Voor de signalen `bcd` en `index` zijn ook vier bits nodig. Dit boek representeert gehele getallen meestal niet met integers maar met `unsigned` of `signed`.
De datatypen `real` en `time` zijn niet synthetiseerbaar. VHDL-2008 is uitgebreid met synthetiseerbare definities voor fixed- en floating-point getallen.

De logische functies **`and`**, **`or`**, **`xor`**, **`nand`**, **`nor`**, **`xnor`** en **`not`** zijn synthetiseerbaar. De unaire rekenkundige bewerkingen **`abs`**, `-` en de binaire rekenkundige bewerkingen `+`, `-` en `*` zijn synthetiseerbaar voor datatypen die gehele getallen representeren.

De rekenkundige functies machtverheffen `**`, delen `/`, remainder **`rem`** en modulus **`mod`** zijn beperkt synthetiseerbaar. Voor deze functies bestaat geen optimaal algoritme. Alleen voor bijzondere gevallen zijn deze functies synthetiseerbaar. Delen door een macht van twee (2^n) komt overeen met het n posities naar links schuiven van de bits. Dit geldt ook voor de functies **`mod`** en **`rem`**. Er zijn ontwerpers die in plaats van `unsigned` en `signed` gewone integers gebruiken voor rekenkundige bewerkingen. Om expliciet aan te geven dat de bewerking beperkt blijft tot n bits neemt men dan de modulus 2^n. Bij een 8-bits integer `int8` schrijft men dan:

Synthesizers gebruiken voor het vermenigvuldigen het Booth-algoritme voor two's complement getallen.

```
int8 <= (int8 + 1) mod 256;
```

De relationele operatoren `<=`, `<`, `=`, `/=`, `>`, `>=` zijn synthetiseerbaar. Er zijn wel afspraken en restricties nodig bij het vergelijken van vectoren van het type `std_logic`. Zolang de vectoren dezelfde lengte hebben en de bits alleen enen en nullen zijn, is het vergelijken eenduidig. Maar is vector `"01-ZX"` groter of kleiner `"01ZHL"`? Als `"101"` met een 5-bits vector wordt vergeleken, gaat het dan om `00101` of om `10100`? In hoofdstuk 10 staan de afspraken en de restricties voor het vergelijken van vectoren bij de packages `ieee_stdlogic_1164` en `numeric_std`.

De IEEE-bibliotheek bevat een package `math_real` met de beschrijvingen van allerlei rekenkundige functies, zoals `sin`, `cos` en `log`. Net als alle andere bewerkingen voor reële getallen zijn deze functies niet synthetiseerbaar. Alleen bij constanten zijn deze bewerkingen soms synthetiseerbaar. Deze bewerking is bij Precision RTL wel en bij Leonardo Spectrum niet synthetiseerbaar:

```
constant LEFT : natural := natural(log(real(MAX_CC))/log(2.0)) - 1;
```

Dynamische constructies

Bij software is het gebruikelijk om met dynamische constructies te werken. Nadat de code gecompileerd is, kan er bij het uitvoeren van het programma bijvoorbeeld extra geheugenruimte worden gealloceerd. Dit kan mits er voldoende RAM aanwezig is.

Alloceren is het toewijzen van faciliteiten of middelen. Geheugenallocatie is het toewijzen van een deel van de geheugenruimte voor een speciaal doel.

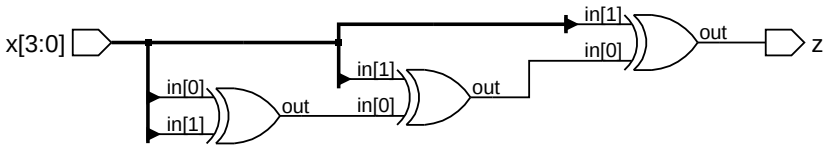
Bij een digitaal systeem ligt alle hardware vast. Tijdens de compilatie bepaalt de synthesizer welke hardware er nodig is voor het functionele gedrag. Nadat de FPGA geprogrammeerd is, zijn er geen mogelijkheden meer om extra registers en extra logica toe te voegen.

Herhalingsopdrachten, zoals de **`for loop`** en de **`for generate`** moeten altijd begrensd zijn. In code 4.8 staat de beschrijving van een paritygenerator. De for-lus wordt vier keer doorlopen. De beschreven functionaliteit komt overeen met:

Het attribuut `range` geeft in de uitdrukking `x'range` het bereik van `x`. In dit geval is dat `3 downto 0`.

```
z <= (((’0’ xor x3) xor x2) xor x1) xor x0;
```

Bij het optimaliseren valt de eerste **`xor`**-functie weg. Uiteindelijk leidt dit tot het syntheseresultaat van figuur 4.9.



Figuur 4.9 : RTL-view van paritygenerator.

Code 4.8 : Synthetiseerbare paritygenerator met for-lus.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity parity is
5    port (
6      x: in std_logic_vector(3 downto 0);
7      z: out std_logic
8    );
9  end entity parity;
10
11 architecture gedrag of parity is
12 begin
13   p1: process (x) is
14     variable p: std_logic;
15   begin
16     p := '0';
17     for i in x'range loop
18       p := p xor x(i);
19     end loop;
20     z <= p;
21   end process p1;
22 end architecture gedrag;
```

Van een for-lus met n iteraties wordt hardware gemaakt door de in de for-lus beschreven functionaliteit n maal te realiseren.
Als de onder- en de bovengrens van een herhalingsopdracht niet vastliggen, is het aantal iteraties onbepaald en weet de synthesizer niet hoe vaak de in de for-lus beschreven functionaliteit gemaakt moet worden. Een constructie met variabele grenzen is niet toegestaan:

```
for i in 0 to variable_limit loop
  ...
end loop;
```

Als de waarde van `variable_limit` variabel is, weet de synthesizer niet hoe vaak de body van de for-lus herhaald moet worden en zal er een foutmelding verschijnen.

4.3 Modellen voor het schrijven van synthetiseerbare VHDL

Een uitleg over de subset van VHDL die synthetiseerbaar is, zal altijd een verhaal met mitsen en maren zijn. Sommige onderdelen zijn in bepaalde situaties juist wel en in andere situaties juist niet synthetiseerbaar. Bovendien interpreteren de synthesizers de code soms verschillend.
Deze paragraaf behandelt alleen de belangrijkste aspecten aan de hand van een aantal sjablonen voor synthetiseerbare sequentiële en combinatorische schakelingen. Deze sjablonen of modellen kunnen direct toegepast worden, maar ze markeren ook de grenzen van de synthetiseerbaarheid.

In hoofdstuk 3 is uitgelegd dat de basis van VHDL de processen zijn. Alle VHDL-constructies zijn te herleiden tot elementaire processen. Na de synthese resulteert een proces in:

- een combinatorische schakeling,
- een sequentiële schakeling,
- een schakeling met latches,
- of geen schakeling als het proces niet synthetiseerbaar is.

Een combinatorische schakeling bestaat alleen uit logische, relationele en rekenkundige bewerkingen. Deze schakelingen hebben geen kloksignaal. Sequentiële schakelingen hebben altijd een klok en bevatten flipfloppen en registers om de huidige toestand van de schakeling te onthouden. Schakelingen met latches zijn vanwege het deels asynchrone gedrag niet gewenst in digitale systemen.

Combinatorisch proces

Een combinatorisch proces is een proces dat een combinatorische schakeling oplevert. Dit proces heeft de volgende kenmerken:

- De gevoeligheidslijst is volledig; dat wil zeggen dat alle ingangen van het proces genoemd worden in de gevoeligheidslijst.
- De beschrijving in het proces is ook volledig. De uitgangen worden voor alle mogelijke ingangscombinaties benoemd.

Het combinatorische proces voldoet aan dit sjabloon:

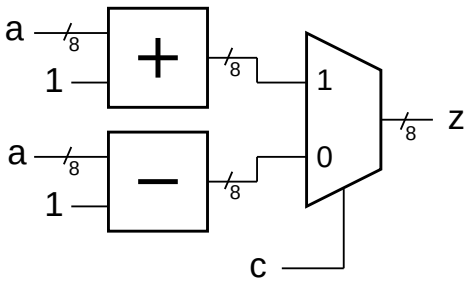
```
proces_label : process ('volledig') is
  ...
begin
  'volledig'
end process proces_label;
```

Als één van de ingangen verandert, moet het proces worden geëvalueerd en daarom moeten alle ingangen in de *sensitivity list* staan. Als niet alle mogelijke combinaties beschreven zijn, onthoudt het proces de huidige toestand. Daarvoor zijn geheugenelementen nodig en is het geen combinatorische schakeling meer.

Code 4.9 geeft een voorbeeld van een combinatorisch proces. In de gevoeligheidslijst van het proces `combinational` staan de twee ingangen `c` en `a` van het proces. Signaal `c` wordt bij de voorwaardelijk opdracht op regel 17 gebruikt. Op regel 18 staat de waarde van `z` als `c` hoog is en op regel 20 staat de waarde van `z` als `c` laag is. De toewijzingen aan `z` berekenen voor elke ingangswaarde van signaal `a` een uitgangswaarde. Alle mogelijke combinaties van `c` en `a` worden door dit proces beschreven. Het proces is dus volledig.
In figuur 4.10 staat een mogelijke RTL-view na de synthese. Het resultaat bestaat uit een incrementer, een decrementer en een multiplexer. De incrementer is nodig voor de bewerking `a + 1`, de decrementer voor `a - 1` en de multiplexer is de implementatie van de voorwaardelijke opdracht.

Code 4.9 : Gedragsbeschrijving met combinatorisch proces.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity model is
6    port (
7      c : in  std_logic;
8      a : in  unsigned(7 downto 0);
9      z : out unsigned(7 downto 0)
10   );
11 end entity model;
12
13 architecture gedrag of model is
14 begin
15   combinational: process (c,a) is
16   begin
17     if c = '1' then
18       z <= a + 1;
19     else
20       z <= a - 1;
21     end if;
22   end process combinational;
23 end architecture gedrag;
```



Figuur 4.10 : RTL-view van het combinatorische proces.

Sequentieel proces zonder reset

Een sequentieel proces is een proces dat een sequentiële schakeling oplevert en heeft altijd een kloksignaal. In dit voorbeeld is er geen reset- of presetsignaal aanwezig. Dit proces heeft de volgende kenmerken:

- De gevoeligheidslijst bevat alleen het kloksignaal.
- De acties in het proces worden alleen uitgevoerd bij de actieve klokflank.

Het sequentiële proces met alleen een klok voldoet aan dit sjabloon:

```
proces_label : process ('klok') is
  ...
begin
  if 'actieve klokflank' then
    ...
  end if;
end process proces_label;
```

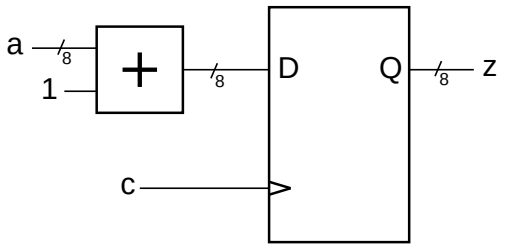
In hoofdstuk 2 en 3 is het begrip sequentieel op een andere manier gebruikt. Daar betekent sequentieel dat de toewijzingen in een proces na elkaar uitgevoerd worden. Hier wordt met een sequentieel proces een proces bedoelt dat een sequentiële schakeling beschrijft.

Als het kloksignaal verandert, wordt het proces geëvalueerd en bij een actieve klokflank worden de acties van de voorwaardelijke opdracht uitgevoerd. Als er geen actieve klokflank is, veranderen de uitgangen niet en zijn er geheugenelementen nodig die de huidige toestand onthouden. Het resultaat is een sequentiële schakeling.

Code 4.10 geeft een voorbeeld van een sequentieel proces met alleen een klok en geen reset of preset. Proces `sequential_no_reset` lijkt sterk op proces `combinational` van code 4.9. In de gevoeligheidslijst staat nu alleen signaal `c` en het else-statement

Code 4.10 : Sequentieel proces zonder resetsignaal.

```
15  sequential_no_reset: process (c) is
16  begin
17    if c = '1' then
18      z <= a + 1;
19    end if;
20  end process sequential_no_reset;
```



Figuur 4.11 : RTL-view van het sequentiële proces zonder resetsignaal.

ontbreekt bij de voorwaardelijke opdracht. De test `c = '1'` wordt alleen uitgevoerd als `c` veranderd is — anders wordt het proces niet uitgevoerd — en als `c` bovendien hoog is, wordt de toewijzing uitgevoerd. Er vindt dus alleen een actie plaats bij de opgaande klokflank. Er is een 8-bits dataregister nodig om de huidige toestand van `z` te onthouden en bij de actieve klokflank wordt de nieuw berekende waarde `a + 1` in het register gezet. In figuur 4.11 staat het syntheseresultaat.

Het nadeel van deze beschrijving is dat niet direct duidelijk is dat het proces alleen afhangt van een klokflank. Het is verstandig om een kloksignaal een duidelijke naam te geven, bijvoorbeeld: `clk`, `clock`, `sys_clk` of `cp`. Dit boek noemt een klok altijd `clk`; alleen in deze paragraaf wordt juist `c` gebruikt om de overeenkomsten tussen de verschillende modellen beter te laten zien. Verder is het verstandig om altijd expliciet aan te geven dat er een actieve klokflank is. In code 4.11 heeft de klok de naam `clk` en is de testvoorwaarde `rising_edge(clk)`. Bij deze beschrijving is direct duidelijk dat er een kloksignaal is met een opgaande flank en dat het proces een sequentiële schakeling voorstelt.

Code 4.11 : Sequentieel proces zonder resetsignaal met een expliciete klok.

```
15  sequential_no_reset: process (clk) is
16  begin
17    if rising_edge(clk) then
18      z <= a + 1;
19    end if;
20  end process sequential_no_reset;
```

Latch-inference

Een proces, dat een schakeling met latches oplevert heeft deze kenmerken:

- Er is geen kloksignaal te herkennen.
- Alle ingangssignalen staan in de gevoeligheidslijst, anders gezegd de *sensitivity list* is volledig.
- De beschrijving in het proces is onvolledig. De uitgangen zijn niet benoemd voor alle mogelijke ingangscombinaties.

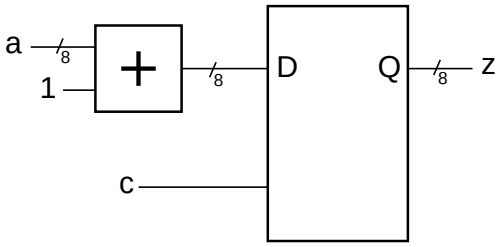
Het toevoegen van latches door een synthesizer aan een schakeling wordt *latch inference* genoemd. Processen met latch-inference voldoen aan dit sjabloon:

```
proces_label : process ('volledig') is
  ...
begin
  'onvolledig'
end process proces_label;
```

Als de synthesizer in de gevoeligheidslijst geen kloksignaal herkent, is het een schakeling zonder flipfloppen of registers. Als de ingangen allemaal genoemd worden en het proces niet voor iedere ingangscombinatie uitgangswaarden benoemd, moet er toch iets worden onthouden. Dit lost de synthesizer op door latches toe te voegen. Omdat in de meeste digitale systemen latches ongewenst zijn, geeft de synthesizer een waarschuwing.

Code 4.12 : Proces met latch-inference.

```
15  latch: process (c,a) is
16  begin
17    if c = '1' then
18      z <= a + 1;
19    end if;
20  end process latch;
```



Figuur 4.12 : RTL-view van het proces met latch-inference.

Code 4.12 geeft een voorbeeld van een proces met latch-inference. Proces `latch` is gelijk aan proces `combinational` van code 4.9, alleen ontbreekt het else-statement. Proces `latch` verschilt ook amper van het proces `sequential_no_reset` uit code 4.10. De gevoeligheidslijst van proces `sequential_latch` bevat naast het signaal `c` ook het ingangssignaal `a`.

Figuur 4.12 geeft het syntheseresultaat en dit is bijna gelijk aan figuur 4.11. In plaats van een dataregister met acht D-flipfloppen worden er acht D-latches gebruikt. De synthesizer geeft de waarschuwing:

```
Warning, z is not always assigned. Storage may be needed..
```

Een niet of slecht synthetiseerbaar proces

Voor een proces, dat niet of slecht synthetiseerbaar is, geldt:

- Er is geen kloksignaal te herkennen.
- Niet alle ingangssignalen staan in de gevoeligheidslijst; de *sensitivity list* is onvolledig.

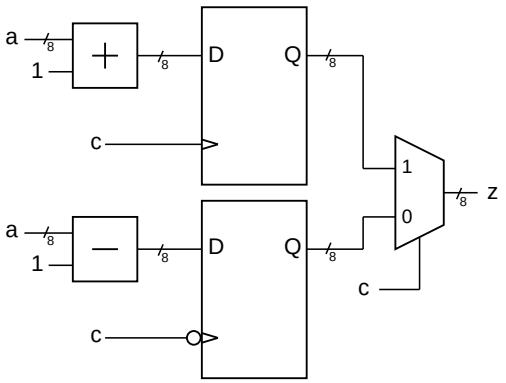
Niet of slecht synthetiseerbare beschrijvingen voldoen aan dit sjabloon:

```
proces_label : process ('onvolledig') is
  ...
begin
  '(on)volledig'
end process proces_label;
```

Als er geen kloksignaal is én één of meer ingangen ontbreken aan de gevoeligheidslijst, reageert het proces niet op een verandering van de ontbrekende ingang.

Code 4.13 : Proces met niet of slecht synthetiseerbare proces.

```
15   bad: process (c) is
16   begin
17     if c = '1' then
18       z <= a + 1;
19     else
20       z <= a - 1;
21     end if;
22   end process bad;
```



Figuur 4.13 : Mogelijke realisatie van code 4.13.

Dat betekent dat de huidige toestand bewaard moet blijven. Dit leidt tot latch-inference, mits er geen onduidelijkheid bestaat over wat de enable-conditie is.
Het verschil tussen proces `bad` uit code 4.13 en proces `combinational` uit code 4.9 is dat signaal `a` aan de gevoeligheidslijst ontbreekt. Proces `bad` wordt alleen getriggerd door signaal `c`. Dit kan geen kloksignaal zijn, omdat — vanwege het else-statement – er een dubbelflankgevoelige flipflop nodig is. ASIC- en PLD-bibliotheken hebben normaal gesproken geen dubbelflankgevoelige flipfloppen. Figuur 4.13 geeft een mogelijke realisatie met een positieve en een negatieve flankgevoelige flipflop.
De synthesizers Leonardo Spectrum en Precision RTL gaan er vanuit dat signaal `a` in de gevoeligheidslijst ontbreekt en maken beide een combinatorische schakeling die overeenkomt met figuur 4.10. Leonardo Spectrum waarschuwt:

```
Warning, a should be declared on the sensitivity list of the process.
```

en Precision RTL meldt:

```
Net a: Although this signal is not part of the sensitivity list
       of this block, it is being read.
       This may lead to simulation mismatch.
```

Algemeen hebben de verschillende synthesizers drie oplossingen voor dit soort beschrijvingen:

- Ze maken registers met combinatorische logica, zoals de code impliceert.
- Ze maken een combinatorische schakeling, zoals de ontwerper het waarschijnlijk bedoelt en geven daarnaast een waarschuwing.
- Ze maken helemaal niets en geven een foutmelding.

De eerste oplossing is niet altijd mogelijk. Als in de gevoeligheid bij proces `bad` alleen signaal `a` staat, is het nog moeilijker te verzinnen hoe de code gerepresenteerd kan worden. Het proces onthoudt dan signaal `c` en heeft daar een flipflop voor nodig die getriggerd wordt door alle flanken van de acht bits van signaal `a`. De tweede oplossing wordt vaak gebruikt, maar heeft als groot nadeel dat de beschrijving en het gesynthetiseerde netwerk functioneel anders zijn. De derde oplossing wordt weinig gebruikt. Het confronteert de ontwerper met de dubbelzinnigheid in het ontwerp.
Een goed ontwerper gebruikt dit type beschrijvingen nooit. Hij controleert alle waarschuwingen van de synthesizer en voegt indien nodig de ontbrekende signalen toe aan de gevoeligheidslijst.

De synchrone en asynchrone reset

Veel VHDL-ontwerpers voegen aan de sequentiële processen een asynchrone reset toe. Dat is vooral handig bij simulaties. Bij de start van de simulatie zijn alle signalen ongeïnitialiseerd (`'U'`). Een bewerking met ongeïnitialiseerde signalen levert vaak een onbekende waarde (`'X'`) op.

Door aan de flipfloppen een reset of een preset toe te voegen, kunnen deze eenvoudig in een bekende toestand worden gebracht. Dit kan met een asynchrone of met een synchrone reset. Voordat de simulatie start wordt eerst de reset een aantal klokslagen actief gemaakt.

Het belangrijkste voordeel van een synchrone reset is dat het ontwerp helemaal synchroon is. Mits de reset aan dezelfde eisen voldoet als de andere ingangen, zal het ontwerp ongevoelig zijn voor glitches en metastabiliteit.

Als de ingang van een flipflop vlak in de beurt van een actieve klokflank verandert, wordt de flipflop metastabiel. Dat betekent dat de flipflop niet hoog of laag wordt, maar er tussen in blijft hangen.
Een ontwerper van digitale systemen moet metastabiliteit voorkomen. Dat kan door alle asynchrone ingangen te synchroniseren.

Het nadeel van een synchrone reset is dat de logica die nodig is voor de reset onderdeel uitmaakt van de functionele beschrijving van het ontwerp. Bovendien is daar extra logica voor nodig, hetgeen leidt tot extra vertragingen in de signaalpaden.

Mits de ASIC- of PLD-bibliotheek flipfloppen met een asynchrone reset ondersteunt, geeft een asynchrone reset geen extra vertragingen in de signaalpaden. Het nadeel is dat deze reset per definitie asynchroon is en dat het systeem metastabiel kan worden bij het loslaten van de reset. Dit probleem is te ondervangen door het resetsignaal te synchroniseren.

Sequentieel proces met synchrone reset

Een proces, dat sequentiële schakeling met synchrone reset beschrijft, voldoet aan deze punten:

- De gevoeligheidslijst bevat alleen het kloksignaal.
- Alle toewijzingen in het proces, dus ook die van de reset, worden bij de actieve klokflank uitgevoerd.

De beschrijving voldoet aan dit sjabloon:

```
proces_label : process (’klok’) is
  ...
begin
  if ’actieve klokflank’ then
    if ’resetvoorwaarde’ then
      alle uitgangen worden laag
    else
      ...
    end if;
  end if;
end process proces_label;
```

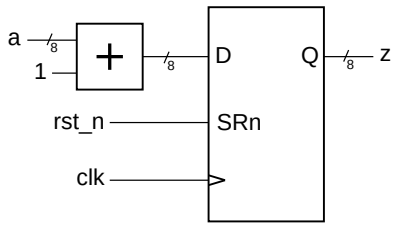
Alleen als er een actieve klok is voert deze beschrijving een actie uit. Als tegelijkertijd de synchrone reset laag is, worden de uitgangen laag gemaakt en als de synchrone reset niet actief is, worden de acties van het else-statement uitgevoerd.

Code 4.14 bevat een voorbeeld van een sequentieel proces met een klok- en een synchroon resetsignaal. De entity `modelr` komt overeen met entity `model` uit code 4.9. Er is alleen een signaal `rst_n` toegevoegd en de naam van `c` is veranderd in `clk`. Proces `sequential_syn_reset` is een uitbreiding op proces `sequential_no_reset` uit code 4.10.

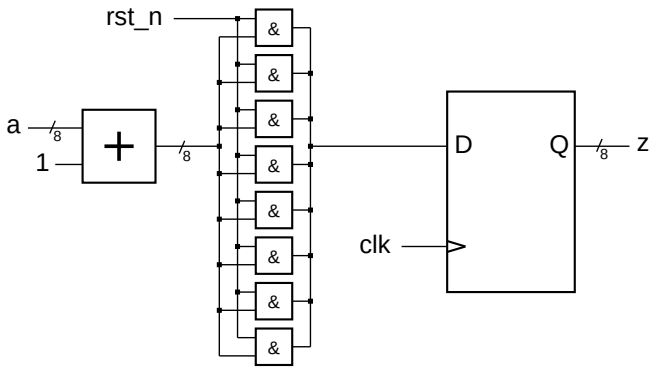
In de gevoeligheidslijst staat alleen signaal `clk`. Het proces wordt alleen getriggerd bij een verandering van `clk` en voert alleen bij de actieve klokflank de voorwaardelijke opdracht op regel 19 uit. Als het resetsignaal `rst_n` laag is, worden alle bits van `z` laag gemaakt, anders krijgt `z` de waarde `a + 1`.

Code 4.14 : Sequentieel proces met synchrone reset.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity modelr is
6    port (
7      clk   : in  std_logic;
8      rst_n : in  std_logic;
9      a     : in  unsigned(7 downto 0);
10     z     : out unsigned(7 downto 0)
11   );
12 end entity modelr;
13
14 architecture gedrag of modelr is
15 begin
16   sequential_syn_reset: process (clk) is
17   begin
18     if rising_edge(clk) then
19       if rst_n = '0' then
20         z <= (others => '0');
21       else
22         z <= a + 1;
23       end if;
24     end if;
25   end process sequential_syn_reset;
26 end architecture gedrag;
```



Figuur 4.14 : Realisatie code 4.14 met register met synchrone reset.



Figuur 4.15 : Realisatie code 4.14 met register zonder synchrone reset.

In figuur 4.14 staat de realisatie van deze beschrijving als de gebruikte bibliotheek flipfloppen en registers bevat met een synchrone reset. Figuur 4.15 geeft de implementatie als de bibliotheek geen flipfloppen en registers met een synchrone reset bevat. De synchrone reset wordt dan met combinatorische logica gemaakt en tussen de opteller en het register ingevoegd.

Sequentieel proces met asynchrone reset

Een proces, dat een sequentiële schakeling met asynchrone reset beschrijft, heeft deze kenmerken:

- De gevoeligheidslijst bevat het klok- en het resetsignaal.
- Het bevat een if-elsif-constructie.
- Bij de if staat de resetvoorwaarde en worden de uitgangen laag gemaakt.
- Bij de elsif staat de actieve klokflank en staan de toewijzingen, die bij de klokflank uitgevoerd moeten worden.

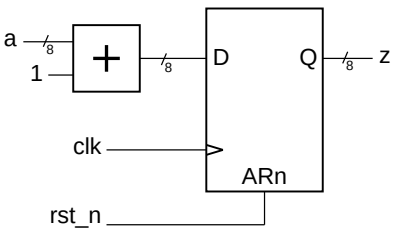
De beschrijving voldoet aan dit sjabloon:

```
proces_label : process ('klok','reset') is
  ...
begin
  if 'resetvoorwaarde' then
    alle uitgangen worden laag
  elsif 'actieve klokflank' then
    ...
  end if;
end process proces_label;
```

Als het reset- of de kloksignaal verandert, wordt het proces getriggerd. Als er aan de resetvoorwaarde wordt voldaan, worden de uitgangen laag gemaakt. Alleen als het resetsignaal niet actief is en er een actieve klokflank is, worden de acties van het elsif-statement uitgevoerd.
Omdat de reset niet achter de elsif met de klokflank staat, is de reset asynchroon. Bovendien is de reset dominant, omdat er eerst op de reset wordt getest.

Code 4.15 : Sequentieel proces met asynchrone reset.

```
1  sequential_asyn_reset: process (clk,rst_n) is
2  begin
3    if rst_n = '0' then
4      z <= (others => '0');
5    elsif rising_edge(clk) then
6      z <= a + 1;
7    end if;
8  end process sequential_asyn_reset;
```



Figuur 4.16 : Realisatie code 4.15.

Code 4.15 bevat een voorbeeld van een sequentieel proces met een klok- en een asynchroon resetsignaal. Proces `sequential_asyn_reset` is een uitbreiding op proces `sequential_no_reset` uit code 4.10. In de gevoeligheidslijst staat naast het kloksignaal `clk` ook het resetsignaal `rst_n` en de test met de resetvoorwaarde is toegevoegd.
Het proces wordt getriggerd bij een verandering van `clk` of `rst_n`. Als `rst_n` laag is, worden de uitgangen laag gemaakt. Als `rst_n` hoog is en er een opgaande klokflank is, krijgt de uitgang `z` de waarde `a + 1`.

In figuur 4.16 staat een realisatie van deze beschrijving. De gebruikte bibliotheek moet dan wel flipfloppen en registers met een asynchrone reset bevatten.

4.4 Sjablonen voor een synthetiseerbare subset van VHDL

Een digitaal systeem bestaat uit combinatorische en sequentiële schakelingen. Met slechts drie processen uit paragraaf 4.3 zijn al deze schakelingen te beschrijven. In figuur 4.17 staan de sjablonen van deze processen: het combinatorische proces, het sequentiële proces zonder reset en het sequentiële proces met een asynchrone reset.

sjablonen voor synthetiseerbare processen

Combinatorisch proces

```
label: process ('volledig') is
  ...
begin
  'volledig'
end process label;
```

Sequentieel proces zonder reset

```
label: process ('klok') is
  ...
begin
  if 'actieve klokflank' then
    ...
  end if;
end process label;
```

Sequentieel proces met asynchrone reset

```
label: process ('klok','reset') is
  ...
begin
  if 'resetvoorwaarde' then
    ...
  elsif 'actieve klokflank' then
    ...
  end if;
end process label;
```

Figuur 4.17 : De drie sjablonen voor het beschrijven van een digitaal systeem.

Dit boek gebruikt het principe van deze drie sjablonen voor het schrijven van synthetiseerbare VHDL. Sequentiële processen met een synchrone reset worden in dit boek weinig toegepast. Schakelingen met latch-inference zijn ongewenst en worden net als de slecht of niet synthetiseerbare processen niet gebruikt.

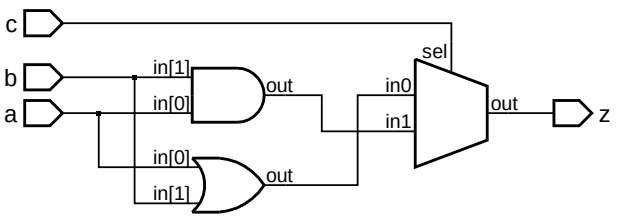
Het gaat bij deze sjablonen om een basale afspraak. Paragraaf 4.5 geeft een aantal variaties op deze afspraak. In hoofdstuk 3 is duidelijk gemaakt dat elke parallelle opdracht herschreven kan worden met een expliciet proces. Een equivalente parallelle opdracht van een combinatorische proces is natuurlijk ook synthetiseerbaar.

Code 4.16 : Combinatorische beschrijving met proces en if-statement.

```
comb_if: process (c,a,b) is
begin
  if c = '1' then
    z <= a and b;
  else
    z <= a or b;
  end if;
end process comb_if;
```

Code 4.17 : Combinatorische beschrijving met conditional signal assignment.

```
comb_cond: z <= a and b when c = '1' else a or b;
```



Figuur 4.18 : RTL-view van code 4.16 en code 4.17.

4.5 Variaties op de sjablonen voor de synthetiseerbare subset

Deze paragraaf laat een aantal variaties op de sjablonen van figuur 4.17 zien. Sommige alternatieven zijn op zijn minst zo goed als het sjabloon. Dat geldt zeker voor de parallelle signaaltoewijzingen die combinatorische schakelingen beschrijven.

Alternatieven voor het combinatorische proces

In code 4.16 staat de beschrijving van een combinatorisch proces: alle ingangssignalen staan in de gevoeligheidslijst en voor alle ingangscombinaties krijgt de uitgang `z` een waarde. De RTL-view is de combinatorische schakeling uit figuur 4.18. Code 4.17 is de parallelle variant van het proces `comb_if`. Als één van de ingangen `c`, `a` of `b` verandert, wordt de toewijzing uitgevoerd. Het proces `comb_cond` is dus gevoelig voor alle ingangen van het proces. Bovendien krijgt `z` een waarde voor alle mogelijke combinaties van `a`, `b` en `c`. Proces `comb_cond` levert dus ook een combinatorische schakeling op. Sterker nog, de processen `comb_if` en `comb-cond` zijn equivalent en leveren bij synthese dezelfde schakeling op.

Sommige ontwerpers gebruiken in code 4.18 en code 4.19 het null-statement:
`when others => null;`
`null when others;`
Dit wordt gedaan om de metawaarden van `std_logic` af te vangen. **`null`** geeft aan dat er bij die waarden niets gedaan wordt.

Voor de simulatie maakt dat geen verschil, maar bij de synthese leidt dit tot latch-inference.

Daarom is hier steeds expliciet een bewerking vermeld.

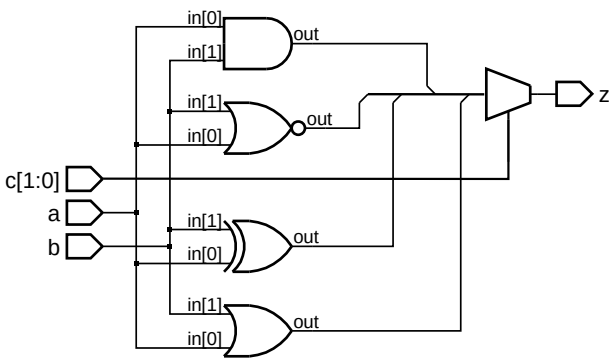
Code 4.18 bevat een proces met case-statement. Dit proces is eveneens combinatorisch: alle ingangssignalen staan in de gevoeligheidslijst en voor alle ingangscombinaties krijgt uitgang `z` een waarde. De RTL-view staat in figuur 4.19. In code 4.19 staat de parallelle variant van proces `comb_case`. Als één van de ingangen `c`, `a` of `b` verandert, wordt de toewijzing getriggerd. De parallelle opdracht `comb_select` beschrijft dezelfde functionaliteit als proces `comb_case`. De synthese geeft in beide gevallen de combinatorische schakeling uit figuur 4.19.

Code 4.18 : Combinatorische beschrijving met proces en case-statement.

```
comb_case: process (c,a,b) is
begin
  case c is
  when "00"   =>  z <= a and b;
  when "01"   =>  z <= a or  b;
  when "10"   =>  z <= a xor b;
  when "11"   =>  z <= a nor b;
  when others =>  z <= a and b;
  end case;
end process comb_case;
```

Code 4.19 : Combinatorische beschrijving met selected signal assignment.

```
comb_select: with c select z <=
  a and b when "00",
  a or  b when "01",
  a xor b when "10",
  a nor b when "11",
  a and b when others;
```



Figuur 4.19 : RTL-view van code 4.18 en code 4.19.

De parallelle opdrachten `comb_cond` en `comb_select` zijn feitelijk verkorte schrijfwijzen voor de processen `comb_if` en `comb_case`. Het zijn impliciete procesbeschrijvin-

gen met hetzelfde functionele gedrag als de expliciete procesbeschrijvingen en zijn dientengevolge eveneens synthetiseerbaar.

Alternatieve beschrijvingen voor sequentieel proces zonder reset

In figuur 4.20 staan zeven equivalente beschrijvingen voor een sequentieel proces. Bij synthese geven al deze beschrijvingen hetzelfde resultaat, namelijk het netwerk dat in figuur 4.20 staat.

```
sequential1: process (clk) is
begin
  if clk = '1' then
    z <= a + b;
  end if;
end process sequential1;
```

```
sequential2: process (clk) is
begin
  if rising_edge(clk) then
    z <= a + b;
  end if;
end process sequential2;
```

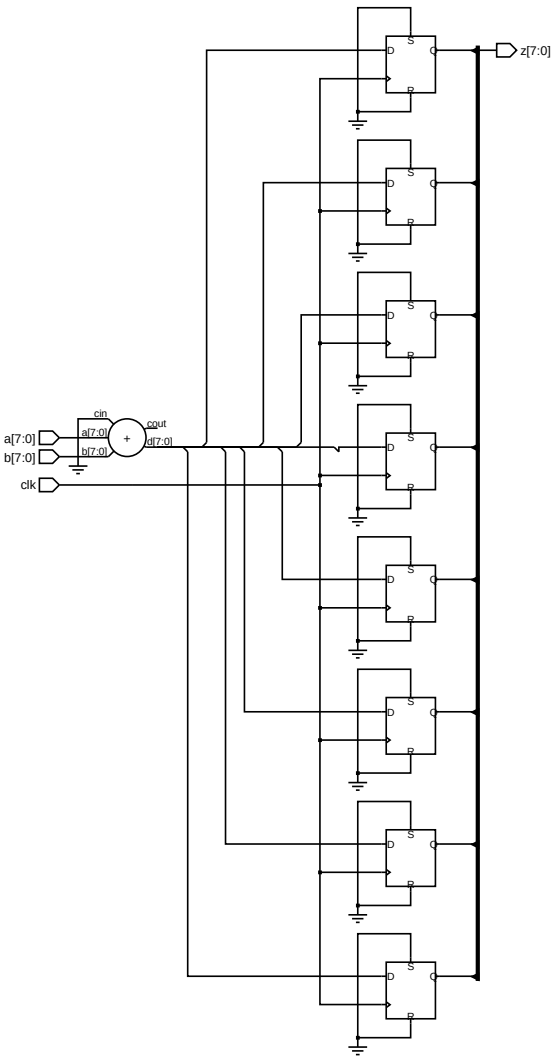
```
sequential3: process (clk) is
begin
  if clk'event and clk = '1' then
    z <= a + b;
  end if;
end process sequential3;
```

```
sequential4: process is
begin
  wait until clk = '1';
  z <= a + 1;
end process sequential4;
```

```
sequential5: process is
begin
  wait until clk'event and clk = '1';
  z <= a + b;
end process sequential5;
```

```
sequential6: process is
begin
  wait until not clk'stable and clk='1';
  z <= a + b;
end process sequential6;
```

```
sequential7: z <= a + b when clk'event and clk = '1';
```



Figuur 4.20 : Zeven beschrijvingen van een sequentieel proces met het syntheseresultaat.

Proces `sequential1` komt overeen met proces `sequential_no_reset` uit code 4.10. Dit proces wordt getriggerd door het kloksignaal `clk`. Als deze klok bovendien hoog is, wordt aan `z` de som van `a` en `b` toegekend. Bij deze beschrijving is niet direct te zien dat het om een sequentieel proces gaat.

Proces `sequential2` wordt in dit boek gebruikt om een sequentieel proces zonder reset te beschrijven. De functie `rising_edge` laat duidelijk zien dat het een sequentieel proces is en dat de schakeling op een opgaande klokflank reageert. De functie `falling_edge` beschrijft een schakeling met een neergaande klokflank.

Aanvankelijk herkenden synthesizers de functie `rising_edge` niet. Ontwerpers gebruikten de uitdrukking `clk'event and clk = '1'` voor een opgaande klokflank. Proces `sequential3` gebruikt deze uitdrukking. Het attribuut `'event` van het signaal `clk` is waar als `clk` ten opzichte van de vorige simulatieslag gewijzigd is. Als het signaal tegelijkertijd hoog is, moet het signaal hoog zijn geworden en is er een opgaande klokflank. Op dezelfde manier beschrijft `clk'event and clk = '0'` een neergaande klokflank. De functies `rising_edge` en `falling_edge` hebben tegenwoordig de voorkeur in plaats van de `'event`-constructies.

Proces `sequential4` heeft geen gevoeligheidslijst, maar in plaats daarvan een expliciete wachtopdracht. Het proces wacht totdat het kloksignaal `clk` hoog is en voert dan de bewerking uit. Proces `sequential5` is identiek aan `sequential4`, maar gebruikt een expliciete beschrijving voor de klokflank. Sommige ontwerpers geven de voorkeur aan een beschrijving met de wait-statements. Het proces `sequential6` is identiek aan `sequential4`, maar gebruikt het attribuut `'stable`.

Naast het attribuut `'event` en `'stable` kent VHDL nog tien andere signaalattributen. De meeste van deze attributen hebben bij synthese geen betekenis.

Proces `sequential7` is een parallelle signaaltoewijzing. Dit is de kortste beschrijving. Toch wordt deze vorm weinig gebruikt. Niet alle synthesizers herkennen hier een sequentieel proces in. Vooral als er een extra voorwaardelijke opdracht bij staat, is het slecht synthetiseerbaar:

```
not1: z <= a + b when rising_edge(clk) and (enable='1');
```

Ingewikkelde constructies zoals deze zullen zeker niet synthetiseerbaar zijn:

```
not2: z <= a + b when rising_edge(clk) and (mode='1') else
           a - b when rising_edge(clk) and (mode='0');
```

De conditionele signaaltoewijzing wordt daarom weinig gebruikt voor sequentiële schakelingen. Alleen voor dataregisters zonder *clear*- of *load*-functie is dit bruikbaar.

Alternatieve beschrijvingen voor sequentieel proces met reset

In figuur 4.21 staan drie equivalente beschrijvingen voor een sequentieel proces met een synchrone reset. De beschrijvingen geven alle drie het syntheseresultaat dat ook in figuur 4.21 staat.

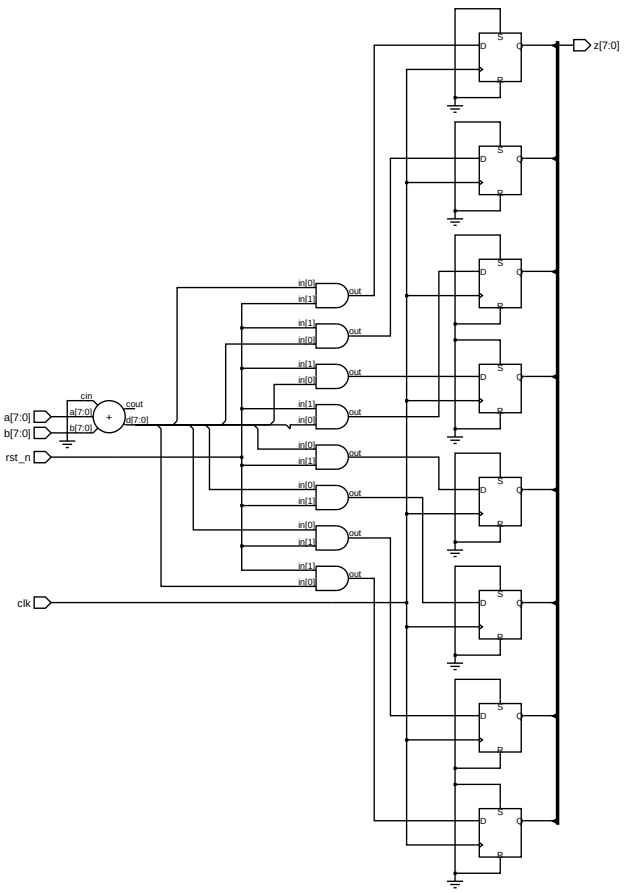
Proces `seq_sr1` komt overeen met proces `sequential_syn_reset` uit code 4.14. Dit proces wordt getriggerd door het kloksignaal `clk`. Er is een opgaande klokflank als `clk` hoog is. Bij deze klokflank worden als het resetsignaal laag is alle bits van het signaal `z` laag. Anders krijgt `z` de som van `a` en `b` toegekend.

De processen `seq_sr2` en `seq_sr3` wachten op een opgaande klokflank. Proces `seq_sr2` voert bij de klokflank hetzelfde if-statement uit van proces `seq_sr1`. In proces `seq_sr3` krijgt `z` de som van `a` en `b` toegekend en wordt daarna getest of er een reset is. Als `rst_n` laag is, worden de bits van `z` als nog laag gemaakt.

```
seq_sr1: process (clk) is
begin
  if rising_edge(clk) then
    if rst_n = '0' then
      z <= (others => '0');
    else
      z <= a + b;
    end if;
  end if;
end process seq_sr1;
```

```
seq_sr2: process is
begin
  wait until clk = '1';
  if rst_n = '0' then
    z <= (others => '0');
  else
    z <= a + b;
  end if;
end process seq_sr2;
```

```
seq_sr3: process is
begin
  wait until clk = '1';
  z <= a + b;
  if rst_n = '0' then
    z <= (others => '0');
  end if;
end process seq_sr3;
```

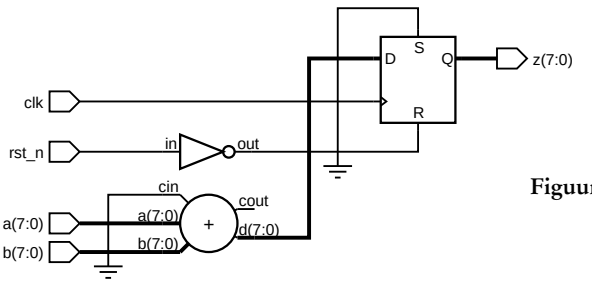


Figuur 4.21 : Drie beschrijvingen voor een sequentieel proces met een synchrone reset.
Het syntheseresultaat heeft acht extra NAND-poorten nodig voor de synchrone reset.

Sequentiële schakelingen met een asynchrone reset moeten over het algemeen voldoen aan het sjabloon van figuur 4.17. Code 4.20 voldoet aan dit sjabloon en geeft het resultaat van figuur 4.22. Veel synthesizers hebben moeite met de constructie uit code 4.21. Leonardo Spectrum meldt:

```
Error, clock expression should contain only one signal
```

Precision RTL herkent wel een asynchrone reset in deze beschrijving en maakt hetzelfde netwerk als bij code 4.20.



Figuur 4.22 : RTL-view van proces met asynchrone reset.

Code 4.20 : Synthetiseerbaar sequentieel proces met een asynchrone reset.

```
seq_ar1: process (clk,rst_n) is
begin
  if rst_n = '0' then
    z <= (others => '0');
  elsif rising_edge(clk) then
    z <= a + b;
  end if;
end process seq_ar1;
```

Code 4.21 : Slecht synthetiseerbaar sequentieel proces met een asynchrone reset.

```
seq_ar2: process is
begin
  wait until (rst_n = '0') or rising_edge(clk);
  if rst_n = '0' then
    z <= (others => '0');
  elsif rising_edge(clk) then
    z <= a + b;
  end if;
end process seq_ar2;
```

4.6 Beschrijving van een 4-bits teller

In code 4.22 staat de entity van een 4-bits teller met een clear, een enable en een asynchrone reset. Alle ingangen zijn van het type `std_logic` en het uitgangssignaal `count` is een `std_logic_vector` van 4 bits.

Code 4.22 : De entity van een 4-bits teller met clear en enable.

```
1  entity count4 is
2    port (
3      clk   : in  std_logic;
4      rst_n : in  std_logic;
5      clear : in  std_logic;
6      enable: in  std_logic;
7      count : out std_logic_vector(3 downto 0)
8    );
9  end entity count4;
```

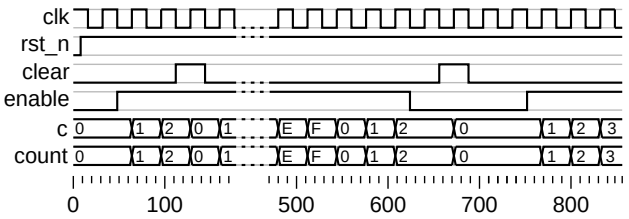
In code 4.23 staat een gedragsbeschrijving. De clear is dominant over de enable. De beschrijving voldoet aan het sjabloon van een sequentieel proces met de asynchrone reset uit figuur 4.17.

Het interne signaal `c` wordt gebruikt als teller. Het type van `c` is `unsigned`. De optelfunctie voor dit type staat in het package `numeric_std`. Bij een actieve klokflank zijn er drie mogelijkheden: `clear` is hoog; `clear` is laag en `enable` is hoog of beide signalen zijn laag. In het eerste geval wordt de teller `c` nul gemaakt, in het tweede geval wordt `c` opgehoogd en in het laatste geval gebeurt er niets en behoudt `c` de huidige waarde.

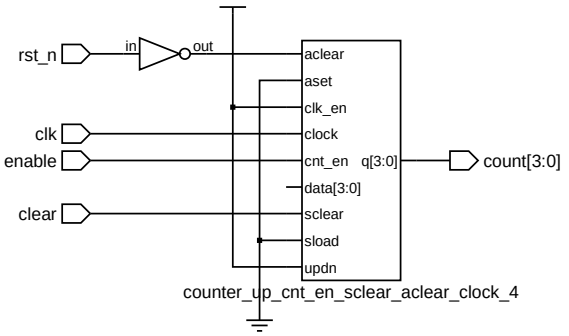
Het uitgangssignaal `count` kan in proces `p` niet gebruikt worden als teller omdat bij het ophogen van de teller `count` ook een ingang is voor het proces. In paragraaf 2.9 is uitgelegd waarom dit niet toegestaan is. Bij de toewijzing van `c` aan `count` wordt het signaal getypecast naar `std_logic_vector`.

Code 4.23 : Beschrijving van een 4-bits teller.

```
1  architecture gedrag of count4 is
2    signal c : unsigned(3 downto 0);
3  begin
4    p: process (clk, rst_n) is
5    begin
6      if rst_n = '0' then
7        c <= (others => '0');
8      elsif rising_edge(clk) then
9        if clear = '1' then
10         c <= (others => '0');
11       elsif enable = '1' then
12         c <= c + 1;
13       end if;
14     end if;
15   end process p;
16
17   count <= std_logic_vector(c);
18 end architecture gedrag;
```



Figuur 4.23 : Simulatie van 4-bits teller.



Figuur 4.24 : RTL-view van 4-bits teller.

De simulatie laat zien dat als `enable` hoog is, de teller telt en dat als `clear` hoog is, de teller nul wordt gemaakt. Bij 128 ns is te zien dat de clear dominant is over het enable-signaal.
Uitgangssignaal `count` volgt signaal `c` en geeft de uitkomst, die bij een 4-bits teller met clear en enable hoort.

Code 4.24 : Beschrijving van een 4-bits teller met behulp van een variabele.

```
1  architecture gedrag of count4 is
2  begin
3    v: process (clk,rst_n) is
4      variable c : unsigned(3 downto 0);
5    begin
6      if rst_n = '0' then
7        c := (others => '0');
8      elsif rising_edge(clk) then
9        if clear = '1' then
10         c := (others => '0');
11       elsif enable = '1' then
12         c := c + 1;
13       end if;
14     end if;
15     count <= std_logic_vector(c);
16   end process v;
17 end architecture gedrag;
```

Code 4.24 gebruikt in plaats van een signaal een variabele als teller. De toewijzing aan `count` moet nu in het proces staan, omdat de variabele `c` alleen lokaal in het proces bekend is. De if-constructie tussen regel 6 en regel 14 bepaalt de nieuwe waarde van `c` en deze wordt vervolgens op regel 15 toegekend aan `count`.

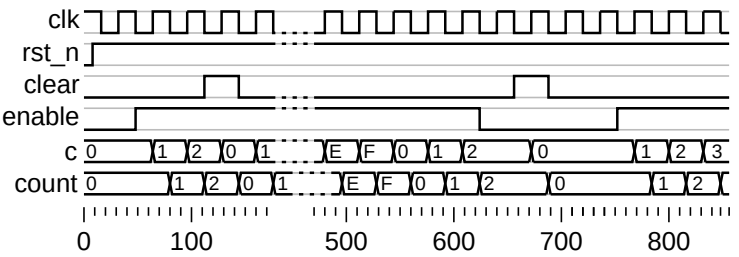
De simulatie van code 4.24 geeft hetzelfde resultaat als figuur 4.23 en de synthese is identiek aan figuur 4.24. Op het niveau van de delta-cycles verschillen de beide simulaties wel. Bij code 4.23 verandert signaal `count` één delta-cycle later.

In code 4.25 staat een mengvorm van code 4.23 en code 4.24. Er wordt voor `c` een signaal gebruikt en de toewijzing aan `count` staat in het proces.

Code 4.25 : Een 4-bits teller die op de verkeerde klokflank reageert.

```
1  architecture gedrag of count4 is
2    signal c : unsigned(3 downto 0);
3  begin
4    s: process (clk,rst_n) is
5    begin
6      if rst_n = '0' then
7        c <= (others => '0');
8      elsif rising_edge(clk) then
9        if clear = '1' then
10         c <= (others => '0');
11       elsif enable = '1' then
12         c <= c + 1;
13       end if;
14     end if;
15     count <= std_logic_vector(c);
16   end process s;
17 end architecture gedrag;
```

In figuur 4.25 staat het simulatieresultaat van code 4.25. Uitgang `count` verandert bij de neergaande klokflank. Dit komt doordat de toewijzing aan `count` afhangt van de toewijzing aan `c`. Signaal `c` krijgt zijn nieuwe waarde bij de opgaande klokflank, maar de feitelijk toekenning gebeurt pas nadat alle processen geëvalueerd zijn. Bij de toewijzing aan `count` op regel 15 heeft `c` nog de oude waarde en verandert `count` dus ook niet. Bij de neergaande klokflank wordt het proces opnieuw getriggerd. Ondertussen heeft `c` de nieuwe waarde gekregen. De voorwaarden op regel 6 en 8 zijn beide niet waar. De enige actie die uitgevoerd wordt, is de toewijzing van regel 15. Het signaal krijgt nu de waarde van `c`.



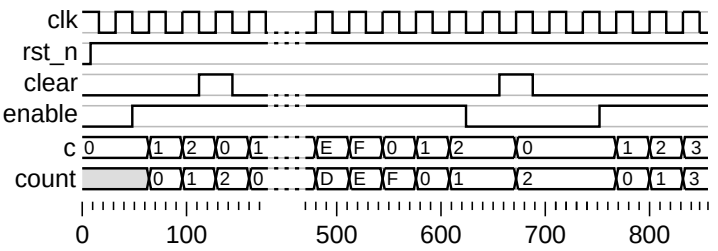
Figuur 4.25 : Simulatie van 4-bits teller van code 4.25.

Het simulatieresultaat van figuur 4.25 is anders dan dat van figuur 4.23. Uitgang `count` verandert een halve klokslag later. Ondanks dit verschil genereert de synthesizer toch het netwerk van figuur 4.24. Het syntheseresultaat van code 4.25 heeft een ander gedrag dan de gedragsbeschrijving.

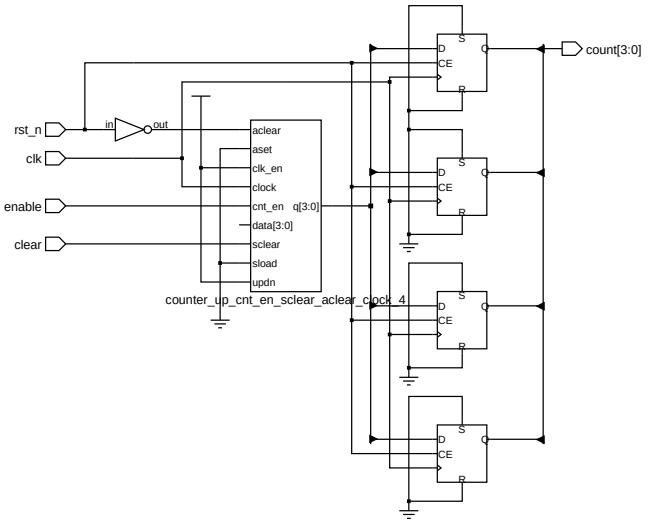
In code 4.26 is de toewijzing binnen de voorwaardelijk opdracht met de klokflank geplaatst. De toewijzing staat op regel 14 na het if-else-statement regel 9 met de toewijzingen aan c. Signaal count valt nu net als c achter de klokflank. Elk signaal achter een klokflank krijgt een register, dus ook het signaal count.

Code 4.26 : Een 4-bits teller met register.

```
1  architecture gedrag of count4 is
2    signal c : unsigned(3 downto 0);
3  begin
4    d: process (clk,rst_n) is
5    begin
6      if rst_n = '0' then
7        c <= (others => '0');
8      elsif rising_edge(clk) then
9        if clear = '1' then
10         c <= (others => '0');
11       elsif enable = '1' then
12         c <= c + 1;
13       end if;
14       count <= std_logic_vector(c);
15     end if;
16   end process d;
17 end architecture gedrag;
```



Figuur 4.26 : Simulatie van code 4.26.



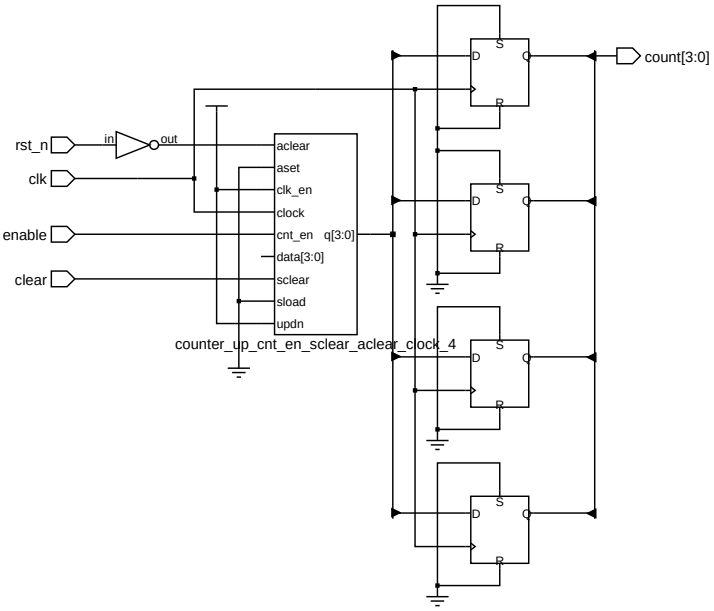
Figuur 4.27 : RTL-view van 4-bits teller.

Het simulatieresultaat in figuur 4.26 toont aan dat het signaal count één klokslag naijlt op het interne signaal c. Het syntheseresultaat in figuur 4.27 laat zien dat er een register, in de vorm van vier flipfloppen, is toegevoegd.

Proces d uit code 4.26 bevat de beschrijving van een teller en een dataregister. Dit kan ook met twee aparte processen worden gedaan. In code 4.27 staat een proces t voor de beschrijving van de teller en een proces r voor de beschrijving van het dataregister.
Het voordeel van aparte processen is dat de bedoeling van de ontwerper veel duidelijker is. Er kan geen misverstand zijn over het idee. Er is een proces t dat de teller en een proces r dat het register beschrijft. Een bijkomend voordeel is dat het sjabloon ook kan worden aangepast. Proces t heeft het sjabloon van een sequentieel proces met een asynchrone reset en proces r dat van een sequentieel proces zonder reset.

Code 4.27 : Een 4-bits teller en register met twee processen.

```
 1  architecture gedrag of count4 is
 2    signal c : unsigned(3 downto 0);
 3  begin
 4    t: process (clk,rst_n) is
 5    begin
 6      if rst_n = '0' then
 7        c <= (others => '0');
 8      elsif rising_edge(clk) then
 9        if clear = '1' then
10          c <= (others => '0');
11        elsif enable = '1' then
12          c <= c + 1;
13        end if;
14      end if;
15    end process t;
16
17    r: process (clk) is
18    begin
19      if rising_edge(clk) then
20        count <= std_logic_vector(c);
21      end if;
22    end process r;
23  end architecture gedrag;
```



Figuur 4.28 : RTL-view van code 4.27.

Bij proces `d` in code 4.26 gebeurt er niets met signaal `count` als `rst_n` laag is. Dat is de reden dat in het resultaat van figuur 4.27 flipfloppen met een enable-functie worden gebruikt. Bij de beschrijving van code 4.27 heeft proces `r` alleen een klok en worden in het resultaat van figuur 4.28 flipfloppen zonder enable-functie gebruikt.

In code 4.28 staat het enable-signaal niet achter de klokflank, maar is het met een and-functie aan de klokvoorwaarde toegevoegd. Veel synthesizers hebben moeite met deze constructie. Precision RTL vertaalt dit wel naar een schakeling, maar Leonardo geeft deze foutmelding:

```
Error, clock expression should contain only one signal.
```

Omdat niet alle synthesizers deze constructie accepteren, is het raadzaam deze methode niet te gebruiken.

In code 4.29 is het enable-signaal met een and-functie aan de kloklijn toegevoegd. Signaal `clke` geeft het kloksignaal `clk` door als `enable` hoog is. Alle synthesizers kunnen met deze zogenoemde *gated clock* over weg. Toch is het beter geen gated clocks te gebruiken. FPGA's bevatten speciale, geoptimaliseerde verbindingen voor kloksignalen. Bovendien gebruiken synthesizers speciale algoritmes om deze kloklijnen zo gebalanceerd en efficiënt mogelijk te gebruiken. Als er logica in een kloklijn wordt aangebracht worden deze speciale voorzieningen niet gebruikt en zal het tijdsgedrag van de schakeling slechter zijn.
Behandel de enable daarom als een gewoon synchroon signaal en plaats deze in de processen achter de klokflank.

Code 4.28 : Een 4-bits teller met enable in sjabloon.

```
1  architecture gedrag of count4 is
2    signal c : unsigned(3 downto 0);
3  begin
4    e1: process (clk,rst_n) is
5    begin
6      if rst_n = '0' then
7        c <= (others => '0');
8      elsif rising_edge(clk) and (enable = '1') then
9        c <= c + 1;
10       if clear = '1' then
11         c <= (others => '0');
12       end if;
13     end if;
14   end process e1;
15
16   count <= std_logic_vector(c);
17 end architecture gedrag;
```

Code 4.29 : Een 4-bits teller met enable in kloklijn.

```
1  architecture gedrag of count4 is
2    signal c    : unsigned(3 downto 0);
3    signal clke : std_logic;
4  begin
5    clke <= clk and enable;
6
7    e2: process (clk, rst_n) is
8    begin
9      if rst_n = '0' then
10       c <= (others => '0');
11     elsif rising_edge(clk) then
12       c <= c + 1;
13       if clear = '1' then
14         c <= (others => '0');
15       end if;
16     end if;
17   end process e2;
18
19   count <= std_logic_vector(c);
20 end architecture gedrag;
```

4.7 Afhankelijke signaaltoewijzingen in een proces

In paragraaf 3.6 en in paragraaf 4.2 is er op gewezen dat afhankelijke signaaltoewijzingen moeilijk te interpreteren zijn. Code 3.14 en code 4.5 geven bij de synthese een resultaat met een andere functionaliteit dan het gedrag bij de simulatie.
Toch zijn er ook zinvolle toepassingen van afhankelijke signaaltoewijzingen in een proces. Code 4.26 is zinvol als de ontwerper het extra register bij de uitgang nodig acht. In code 4.30 staat een beschrijving van een sequentiële schakeling met een hulpvariabele `tmp` en in code 4.31 staat dezelfde beschrijving maar nu met signaal `tmp`.

Code 4.30 : Proces met variabele `tmp`.

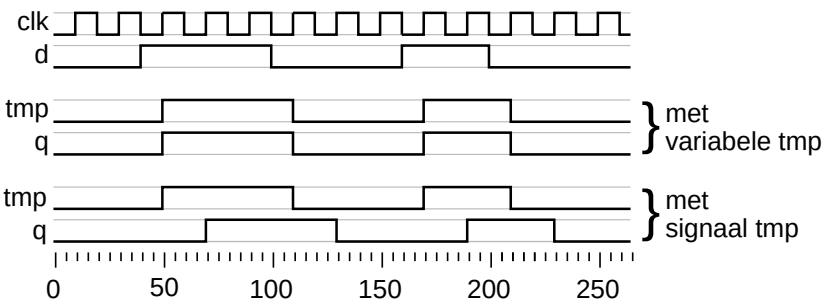
```
1  architecture gedrag of ex is
2  begin
3    v: process (clk) is
4      variable tmp : std_logic;
5    begin
6      if rising_edge(clk) then
7        tmp := d;
8        q   <= tmp;
9      end if;
10   end process v;
11 end architecture gedrag;
```

Code 4.31 : Proces met intern signaal `tmp`.

```
1  architecture gedrag of ex is
2    signal tmp : std_logic;
3  begin
4    s: process (clk) is
5    begin
6      if rising_edge(clk) then
7        tmp <= d;
8        q   <= tmp;
9      end if;
10   end process s;
11 end architecture gedrag;
```

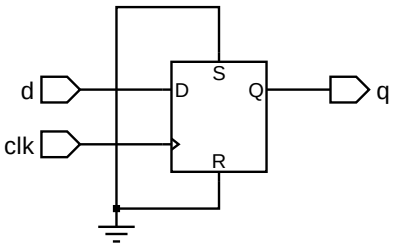
De simulatie van beide beschrijvingen staat in figuur 4.29. Uitgang `q` van proces `v` volgt direct de variabele `tmp`. Een verandering van `d` is bij de eerstvolgende klokslag

zichtbaar bij de uitgang q. Uitgang q van proces s volgt niet direct het signaal tmp. Een verandering van d veroorzaakt dat signaal tmp bij de eerst volgende klokslag een nieuwe waarde krijgt. Alleen wordt deze waarde toegekend aan het einde van de simulatieslag. Bij de toewijzing aan q heeft tmp nog de oude waarde. Het effect is dat alleen tmp verandert en dat q ongewijzigd blijft. Pas bij de volgende klokslag als de toewijzingen weer doorlopen worden krijgt q de nieuwe waarde. Signaal q ijlt één klokslag na op signaal tmp.

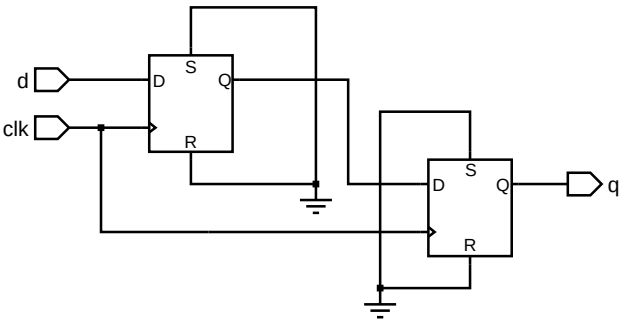


Figuur 4.29 : De simulatie van code 4.30 en code 4.31. In de waveform staan de variable tmp en q van code 4.30 en de signalen tmp en q van code 4.31.

In figuur 4.30 en in figuur 4.31 staan de syntheseresultaten van code 4.30 en code 4.31. Proces v beschrijft een flipflop. De variabele v is feitelijk overbodig, signaal d had ook direct aan q toegekend kunnen worden. Proces s beschrijft twee in serie-geschakelde flipfloppen. Het beschrijft een 2-bits schuifregister: de eerste flipflop bewaard signaal tmp en de tweede flipflop signaal q. Anders gezegd, signaal tmp en d staan beide achter de klokflank en krijgen daarom allebei een flipflop.



Figuur 4.30 : RTL-view van code 4.30.



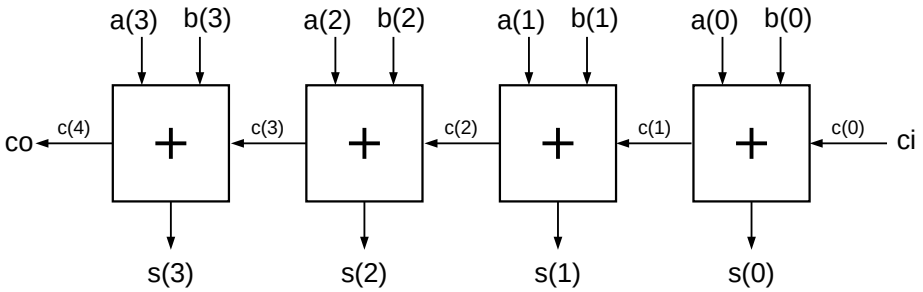
Figuur 4.31 : RTL-view van code 4.31.

Code 4.31 met proces s is een zeer nuttige beschrijving. Het 2-bits schuifregister kan gebruikt worden als synchronizer om asynchrone ingangssignalen te synchroniseren. Elke IO-cel van een FPGA heeft altijd twee flipfloppen, die gebruikt kunnen worden voor de synchronisatie van het ingangssignaal.

4.8 Herhalingsopdrachten

In paragraaf 4.2 is bij de dynamische structuren aangegeven dat herhalingsopdrachten alleen synthetiseerbaar zijn als voor de synthese bekend is hoe vaak een

bepaalde structuur herhaald moet worden. Een while-lus wordt juist gebruikt als het aantal iteraties niet bekend is. Beschrijvingen met while-lussen zullen daarom niet synthetiseerbaar zijn of kunnen ook als for-lus worden geschreven.



Figuur 4.32 : Schema van opteller met vier full-adders.

Een 4-bits opteller met een for-lus

Een opteller kan worden gemaakt door een aantal full-adders te combineren. In figuur 4.32 staat het schema van een 4-bit *ripple adder*. Deze opteller wordt zo genoemd omdat de carry-in zich van de laagste bit naar de hoogste bit door de opteller voortplant. In code 4.32 staat een beschrijving van deze 4-bits opteller, die gebaseerd is op een for-lus en de beschrijving van een full-adder.

Code 4.32 : Een 4-bits opteller opgebouwd uit vier full-adders met behulp van een for-lus.

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4  entity adder4 is
 5    port (
 6      a, b : in  std_logic_vector(3 downto 0);
 7      ci   : in  std_logic;
 8      s    : out std_logic_vector(3 downto 0);
 9      co   : out std_logic
10    );
11  end entity adder4;
```

```
13  architecture gedrag of adder4 is
14    signal c : std_logic_vector(4 downto 0);
15  begin
16    c(0) <= ci;
17    f: process (a,b,c) is
18      variable p,g,t:  std_logic;
19    begin
20      for j in 3 downto 0 loop
21        p      := a(j) xor b(j);
22        g      := a(j) nand b(j);
23        t      := p nand c(j);
24        c(j+1) <= g nand t;
25        s(j)   <= p xor c(j);
26      end loop;
27    end process f;
28    co <= c(4);
29  end architecture gedrag;
```

Er is een 5-bits signaal `c` gedeclareerd dat gebruikt wordt voor de verbindingen tussen de full-adders. De carry-in van de eerste full-adder is `c(0)` en komt overeen met signaal `ci`, de carry-in van de opteller. De carry-out van de laatste full-adder is `c(4)` en is hetzelfde als signaal `co`, de carry-out van de opteller.

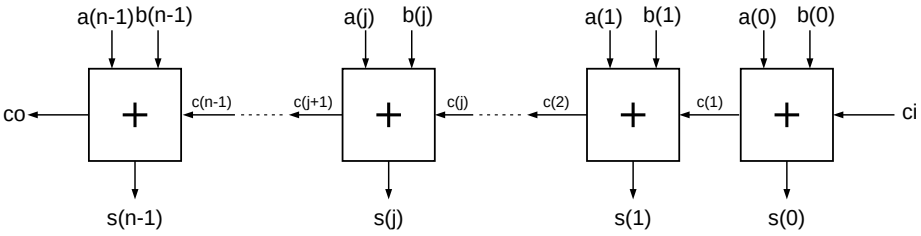
De for-lus berekent bij iedere iteratie — dus voor elke bit — uit de ingangen `a(j)`, `b(j)` en `c(j)` van de full-adder de uitgangen `s(j)` en `c(j+1)`.

Code 4.33 : Een 4-bits opteller met vier componenten.

```
1  architecture gedrag of adder4 is
2    component fulladder is
3      port (
4        a,b : in  std_logic;
5        ci  : in  std_logic;
6        s   : out std_logic;
7        co  : out std_logic
8      );
9    end component fulladder;
10
11   for all : fulladder use entity work.fulladder(gedrag);
12
13   signal c : std_logic_vector(3 downto 1);
14 begin
15   f3:  fulladder port map (a(3), b(3), c(3), s(3), co);
16   f2:  fulladder port map (a(2), b(2), c(2), s(2), c(3));
17   f1:  fulladder port map (a(1), b(1), c(1), s(1), c(2));
18   f0:  fulladder port map (a(0), b(0),   ci, s(0), c(1));
19 end architecture gedrag;
```

Een 4-bits opteller met vier full-adders

In plaats van opnieuw een full-adder in de gedragsbeschrijving van de opteller te beschrijven, kan ook de full-adder uit paragraaf 2.3 of een van de alternatieven uit paragraaf 2.7 als component gebruikt worden. In code 4.33 staat een 4-bits opteller die opgebouwd is uit vier full-adders. Signaal `c` is nu drie bits breed. Signaal `ci` is de carry-in van component `f0` en signaal `co` is de carry-out van component `f3`.



Figuur 4.33 : Schema van opteller met n full-adders. De in- en uitgangen van bijna alle full-adders worden met een index j beschreven. Alleen de eerste en de laatste full-adder wijken daarvan af.

Een n-bits opteller met het generate-statement

De for-lus en de while-lus mogen alleen in een sequentiële omgeving gebruikt worden. Zoals de opteller laat zien, bestaat hardware vaak uit een herhaling van dezelfde structuren. VHDL kent daarom ook een herhalingsopdracht voor de parallelle omgeving. Deze herhalingsopdracht is vooral interessant voor beschrijvingen die algemeen zijn. Als de ontwerper naast een 4-bits opteller ook een 10-bits opteller nodig heeft, moet er een tweede opteller gemaakt worden met tien aanroepen van de full-adder. In plaats daarvan kan de ontwerper een algemene n-bits opteller maken, waar pas bij de aanroep wordt aangegeven uit hoeveel bits deze bestaat.

Code 4.34 : De entity van een n-bits opteller.

```
1  entity adder is
2    generic (n : natural := 4);
3    port (
4      a, b   : in  std_logic_vector(n-1 downto 0);
5      ci     : in  std_logic;
6      s      : out std_logic_vector(n-1 downto 0);
7      co     : out std_logic
8    );
9  end entity adder;
```

Code 4.35 : Een n-bits opteller met behulp van een generate-statement.

```
1  architecture gedrag of adder is
2    component fulladder is
3      port (
4        a,b : in  std_logic;
5        ci  : in  std_logic;
6        s   : out std_logic;
7        co  : out std_logic
8      );
9    end component fulladder;
10
11   for all : fulladder use entity work.fulladder(gedrag);
12
13   signal c : std_logic_vector(n-1 downto 1);
14 begin
15   f: for j in n-1 downto 0 generate
16     m: if j=n-1 generate
17       msb : fulladder port map (a(j), b(j), c(j), s(j), co);
18     end generate m;
19     r: if (j>0) and (j<n-1) generate
20       rst : fulladder port map (a(j), b(j), c(j), s(j), c(j+1));
21     end generate r;
22     l: if j=0 generate
23       lsb : fulladder port map (a(j), b(j), ci, s(j), c(j+1));
24     end generate l;
25   end generate f;
26 end architecture gedrag;
```

De simulator Modelsim geeft een waarschuwing. Het generate-statement is dynamisch, daarbij is de statische configuratie van regel 11 formeel niet toegestaan.
Het alternatief is de componentconfiguratie op regel 11 weg te laten of een aparte configuratie te gebruiken, zoals in code 2.11 is toegepast.

In code 4.34 staat de entity van een n-bits opteller. In de *generic list* is een generieke constante `n` gedeclareerd. De breedte van de vectoren `a`, `b` en `s` hangt af `n`. Bij de aanroep van de component wordt de waarde van `n` meegegeven. Als dat niet gedaan wordt, is de standaardwaarde 4.

```
a4:  adder port map (a1,a2,ci,sum,open);
a10: adder generic map (10); port map (x,y,'0',z,cout);
```

Bij de aanroep van component `a4` staat geen **`generic map`**, de breedte van de opteller is dan 4. De signalen `a1`, `a2` en `ci` zijn verbonden met de ingangen `a`, `b` en `ci` en signaal `sum` met uitgang `s`. Het sleutelwoord **`open`** geeft aan dat de carry-out van de opteller niet aangesloten is. Bij de aanroep `a10` is er een **`generic map`**, die de opteller 10 bits breed maakt.

Het generate-statement herhaalt de beschreven hardware voor het opgeven aantal. Bij de herhaling is bij het begin en bij het eind vaak een andere oplossing nodig dan in rest van de schakeling, daarom kan er binnen het generate-statement gebruik worden gemaakt van een speciale if.
In code 4.35 staat de architectuur van de n-bits opteller. Op regel 15 begint het generate-statement. De variabele `j` is een index voor de te herhalen onderdelen en ligt in het bereik `n-1 downto 0`. Een label is bij het generate-statement verplicht. Het generate-statement eindigt op regel 25.
Binnen het generate-statement staan drie voorwaardelijke opdrachten. Op regel 16 staat een if-generate-statement dat wordt uitgevoerd als index `j` gelijk is aan de index van de meest significante bit. Het if-generate-statement op regel 22 wordt uitgevoerd als index `j` gelijk is aan de index van de minst significante bit. In alle andere gevallen is de voorwaarde van regel 19 waar.
Binnen het generate-statement mogen alleen parallelle opdrachten staan, zoals aanroepen van componenten, parallelle signaaltoewijzingen, processen, functies en procedures.

Code 4.36 : Een n-bits opteller met een procedure en een generate-statement.

```
1  architecture gedrag of adder is
2    procedure fulladder (a,b,ci      : in  std_logic;
3                         signal s,co : out std_logic) is
4      variable p,g,t : std_logic;
5    begin
6      p  := a xor b;
7      g  := a nand b;
8      t  := p nand ci;
9      co <= g nand t;
10     s  <= p xor ci;
11   end procedure fulladder;
12
13   signal c : std_logic_vector(n-1 downto 1);
14 begin
15   f: for j in n-1 downto 0 generate
16     m : if j=n-1 generate
17       msb : fulladder (a(j), b(j), c(j), s(j), co);
18      end generate m;
19     r: if (j>0) and (j<n-1) generate
20       rst : fulladder (a(j), b(j), c(j), s(j), c(j+1));
21     end generate r;
22     l: if j=0 generate
23       lsb : fulladder (a(j), b(j), ci, s(j), c(j+1));
24     end generate l;
25   end generate f;
26 end architecture gedrag;
```

Een n-bits opteller met een generate-statement en een procedure

In code 4.36 staat het voorbeeld van een n-bits opteller met een generate-statement en een procedure `fulladder`. De procedure heeft drie ingangen `a`, `b` en `ci` en twee uitgangen `s` en `co`. Omdat de procedure in een parallelle omgeving wordt gebruikt, moeten de uitgangen signalen zijn. Daarom staat het sleutelwoord **`signal`** voor de namen `s` en `co`. Bij de ingangen moet voor `a`, `b` en `c` niets staan of het sleutelwoord **`constant`**.

Het generate-statement van code 4.36 verschilt nauwelijks van die uit code 4.35. Bij de aanroep van de procedures staan nu niet de sleutelwoorden **port map**.

Een n-bits opteller met standaard optelfuncties

Een opteller die gebouwd is uit een keten van full-adders is een *ripple adder*. Een kritiek pad of *critical path* is het pad waar langs de totale tijdvertraging het grootst is. Bij een ripple adder is dat het pad van de carry-in via de keten van full-adders naar de carry-out of een van andere uitgangen van de laatste full-adder. Vooral bij veel bits geeft dit een slechte performance. Voor dit probleem zijn andere, ingewikkelder oplossingen bedacht: de *carry look ahead adder* en de *carry select adder*.
De bespreking van optellers met een alternatief algoritme valt buiten de context van dit boek. Er is een veel belangrijker reden om geen eigen opteller te maken op basis van eigen full-adders. FPGA's hebben de beschikking over speciale constructies om carry-signalen snel door te geven. Daarvoor moet de synthesizer de carry kunnen herkennen. Dit lukt het eenvoudigst door een abstracte optelfunctie te gebruiken. Een synthesizer zal de optelfunctie altijd op de meest optimale wijze implementeren.

Code 4.37 : Een n-bits opteller met een '+' uit numeric_std.

```
 1  architecture gedrag of adder is
 2    signal c : unsigned(n downto 0);
 3    subtype t is unsigned(c'range);
 4  begin
 5    c  <= ('0' & unsigned(a))
 6        + ('0' & unsigned(b))
 7        + t'(0 => ci, others => '0');
 8    s  <= std_logic_vector(c(n-1 downto 0));
 9    co <= c(n);
10  end architecture gedrag;
```

De simulator Modelsim waarschuwt bij de constructuie op regel 7 dat **others** alleen toegestaan is, als dit de enige keuze is.
Op pagina 251 worden diverse alternatieven besproken.

In code 4.37 en in code 4.38 staan twee beschrijvingen voor een binaire opteller. Deze beschrijvingen zijn geschikt het optellen van de natuurlijke getallen, het zijn *unsigned binary adders*. In hoofdstuk 10 worden het package `numeric_std` besproken en komen de gewone binaire en de two's complement bewerkingen aan de orde.

Het package `numeric_std` kent zes optelfuncties, onder andere een functie die twee getallen van het type `unsigned` optelt. In code 4.37 wordt de vector met `unsigned()` omgezet naar een vector van het type `unsigned`. De bewerking `unsigned()` is geen functie maar een zogenoemde *type casting*. De enen en nullen van vector `a` veranderen niet; alleen het type verandert van `std_logic_vector` naar `unsigned`. Met het concatenatie-teken wordt de vector uitgebreid met een `'0'`. Het type van de uitdrukking op regel 5 is dus `unsigned(n downto 0)`.

Op regel 6 wordt bij deze uitdrukking een op dezelfde wijze herschreven vector `b` opgeteld. De carry-in `ci` wordt op regel 7 daar weer bijgeteld. Het omzetten van een `std_logic` naar een `unsigned` is hier gedaan met een *qualified expression*. Van een bit kan eenvoudig een vector gemaakt worden door de bit aan de minst significante bit van de vector toe te kennen (`0 => ci`) en de andere bits hoog te maken (`others => '0'`). Het probleem is dat het type van deze uitdrukking niet bekend is. Een array van nullen en enen kan een `std_logic_vector`, een `unsigned`, een `signed`

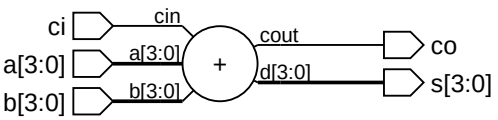
Code 4.38 : Een n-bits opteller met de standaard '+'.

```
1  architecture gedrag of adder is
2    signal uci          : unsigned(0 downto 0);
3    signal na,nb,nc,sum : natural;
4    signal u            : unsigned(n downto 0);
5  begin
6    na     <= to_integer(unsigned(a));
7    nb     <= to_integer(unsigned(b));
8    uci(0) <= ci;
9    nc     <= to_integer(unsigned(uci));
10   sum    <= na + nb + nc;
11   u      <= to_unsigned(sum, n+1);
12   s      <= std_logic_vector(u(n-1 downto 0));
13   co     <= u(n);
14 end architecture gedrag;
```

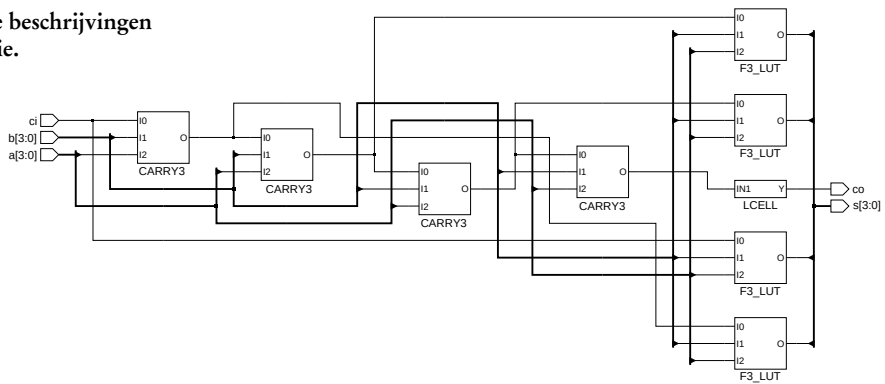
en zelfs een string zijn. Op regel 3 is subtype `t` gedefinieerd die overeenkomt met het type van signaal `c`. Door op regel 7 voor de bewerking `t'` te plaatsen, wordt de uitdrukking van het type `unsigned(n downto 0)`.
Het signaal `c` is de som van `a`, `b`, `ci`. De meest significante bit is de carry-out `co`. De andere bits vormen uitgang `s`. Omdat `c` van het type `unsigned` en `s` een `std_logic_vector` is, wordt `c` getypecast naar een `std_logic_vector`.

De nadelen van de standaard optelfunctie zijn tweeledig. Ten eerste kunnen de getallen nooit breder dan 32-bits breed zijn en ten tweede genereren synthesizers soms een 32-bits opteller terwijl er veel minder bits nodig zijn. Gebruik voor rekenkundige bewerkingen altijd `unsigned` of `signed`.

Code 4.38 gebruikt de standaard optelfunctie voor integers. Ook hier zijn conversies nodig. Op regel 6 wordt vector `a` getypecast naar een `unsigned` en vervolgens met de functie `to_integer()` uit het package `numeric_std` omgezet naar een integer. Regel 2 definieert een 1-bits vector `uci`, waaraan op regel 8 de carry-in wordt toegekend. Op regel 11 wordt met de functie `to_unsigned` de uitkomst `sum` omgezet naar een vector van het type `unsigned`. Deze vector wordt op de volgende regels toegekend aan `s` en `co`.

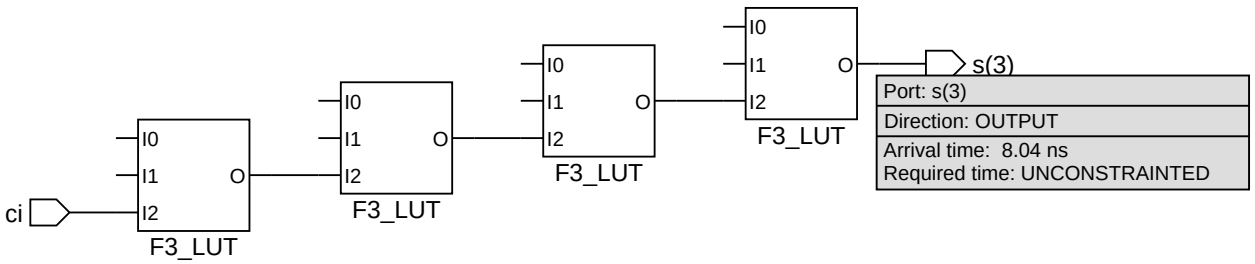


Figuur 4.34 : RTL-view van de beschrijvingen met de '+'-functie.

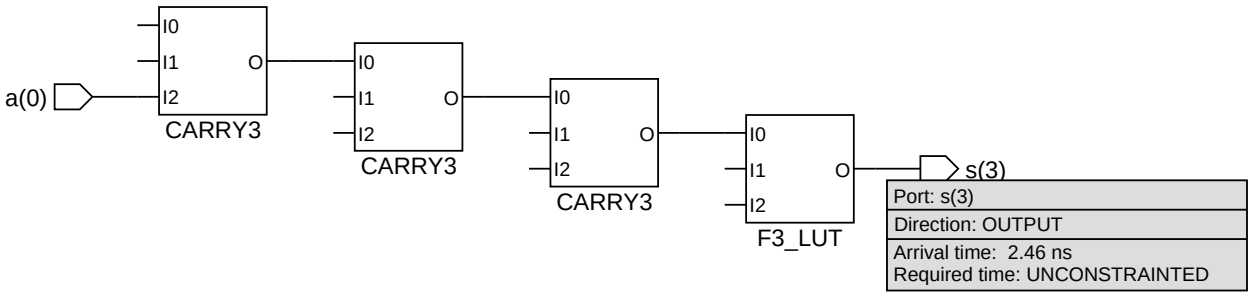


Figuur 4.35 : De technology map van de beschrijvingen met de '+'-functie.

In de twee voorbeelden uit code 4.37 en code 4.38 zijn helaas een aantal lastige conversies nodig. Dat komt omdat VHDL een streng getypeerde taal is en toekenningen alleen mogelijk zijn als de typen overeenkomen. Meer informatie over conversies staat in hoofdstuk 10 over de bibliotheken en packages.
Toch zijn deze twee voorbeelden te prefereren boven de optellers op basis van een eigen full-adder. Beide genereren dezelfde RTL-view en dezelfde *technology map*. Figuur 4.34 geeft de RTL-view en figuur 4.35 de technology map. Zoals de RTL-view laat zien, herkent in beide gevallen de synthesizer een elementaire optelfunctie. In de technology-map worden niet alleen LUT's gebruikt, maar ook speciale carry-blokken. In figuur 4.36 staat het kritieke pad van de beschrijvingen met eigen full-adders en in figuur 4.37 staat het kritieke pad van de beschrijving met de optelfuncties. Het eerste kritieke pad bestaat uit vier LUT's en is 8,04 ns lang. Het tweede kritieke pad is veel korter. Het bevat drie carry-blokken en een LUT en is slechts 2,46 ns



Figuur 4.36 : Het kritieke pad van een opteller met eigen full-adders.



Figuur 4.37 : Het kritieke pad van een opteller met de '+'-functies.

Bij de synthese is de full-adder uit code 2.7 gebruikt en een FPGA uit de ACEX-1K familie met een *speed grade -1*. Het verschil tussen de beide tijdvertragingen kan nog groter zijn als een van de andere full-adders gebruikt wordt en zal zeker veel groter zijn als er meer bits gebruikt worden. Bij 32-bits verschillen de vertragingen meer dan een factor acht, namelijk 65,7 ns tegen 8,06 ns.

Het is verstandig om de code zodanig te schrijven dat de synthesizer in staat is om basale logische en rekenkundige bewerkingen te herkennen. Bovendien is het nodig om de synthesizer en de mogelijkheden van de gebruikte bouwsteen goed te kennen.

4.9 Adviezen, conventies en ontwerpregels

In dit hoofdstuk zijn veel adviezen gegeven en veel bezwaren genoemd bij bepaalde constructies. Hieronder zijn de adviezen uit dit hoofdstuk samengevat met een aantal conventies die dit boek gebruikt om het ontwerpen met VHDL zo eenvoudig mogelijk te houden.

- Ontwerp in algemeen synthetiseerbare VHDL, die door alle synthesizers verwerkt kan worden.
- Gebruik de sjablonen uit figuur 4.17 van paragraaf 4.4.
- De alternatieve beschrijvingen voor combinatorische schakelingen moeten aan dezelfde eisen voldoen als het sjabloon.
- Voeg aan sequentiële schakelingen een asynchrone reset toe. Alleen bij dataregisters is een reset soms overbodig.
- Synchroniseer de asynchrone reset van het systeem.
- Gebruik een actieve lage reset.
- Gebruik alleen een preset- of een resetsignaal.
- Noem het resetsignaal `resetn` of `rst_n` en geen `clearn` of `clr_n`.
- Gebruik de naam `clear` voor een synchrone clear van een teller en register.
- Gebruik altijd een klokflank; bij voorkeur de opgaande flank.
- Gebruik de functie `rising_edge` om een klokflank te beschrijven.
- Plaats geen extra logica naast de functie `rising_edge` in de klokvoorwaarde.
- Plaats geen logica in de kloklijnen; gebruik dus geen zogenoemde *gated clock*.
- Gebruik geen dubbelflankgevoelige flipfloppen.
- Wees in een proces voorzichtig met afhankelijk signaaltoewijzingen.
- Bedenk dat elk signaal achter een klokflank een register krijgt.
- Houd processen klein en overzichtelijk.
- Splits processen eventueel in logische, digitale blokken, zoals: decoders, optellers, tellers, dataregisters en schuifregisters.
- Meng geen combinatorische en sequentiële aspecten in één proces.
- Zorg dat de simulatie van de gedragsbeschrijving en het syntheseresultaat identiek is.
- Controleer altijd of het syntheseresultaat overeenkomt met de beschrijving.
- Gebruik voor de typen van de in- en uitgangssignalen van een entity altijd `std_logic` of `std_logic_vector`.
- Gebruik voor de rekenkundige en relationele bewerkingen altijd `unsigned` of `signed`.
- Gebruik bij `unsigned` en `signed` altijd het package `numeric_std`.

De adviezen en conventies beperken het synthetiseerbare deel van VHDL meer dan strikt noodzakelijk is. Door een bepaalde ontwerpstijl te gebruiken wordt de code veel beter leesbaar en worden er minder fouten gemaakt en zijn fouten veel beter te vinden.

5

Toestandsmachines

Doelstelling

In dit hoofdstuk leer je wat een toestandsmachine is en wat daarbij het verschil is tussen een Moore-machine en een Mealy-machine. Je leert hoe je een toestandsdiagram tekent en hoe je een transitietabel opstelt. Daarnaast leer je verschillende methoden om een toestandsmachine in VHDL te beschrijven.

Onderwerpen

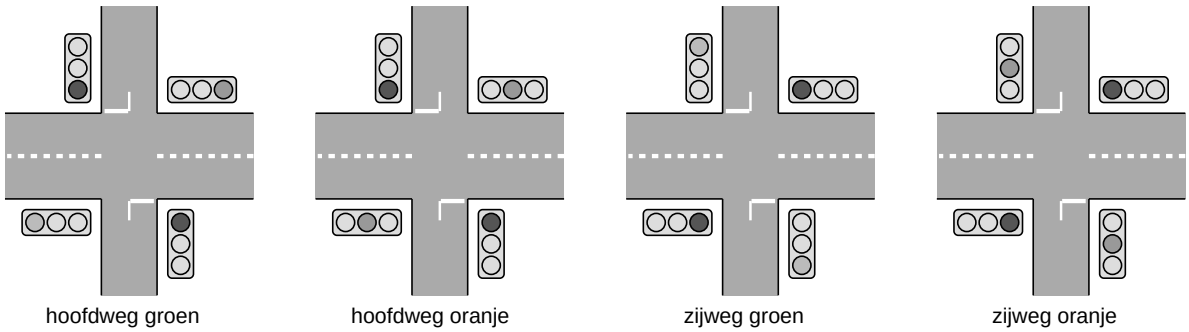
De behandelde onderwerpen zijn:

- Het concept van de toestandsmachine.
- De Moore-machine.
- De Mealy-machine.
- Het toestandsdiagram.
- De toestandstransitietabel.
- Het tijdsgedrag van een Moore-machine en een Mealy-machine.
- Het opstellen van een toestandsdiagram voor een Moore-machine.
- Het omzetten van een Mealy-machine naar een Moore-machine.
- Het beschrijven van toestandsmachines in VHDL.
- Het *enumerated type*.
- Toestandscodering en toestandsoptimalisatie.
- Veilige toestandsmachines.
- De ASM-chart.
- Hiërarchie en gekoppelde toestandsmachines.

Een digitaal systeem kan theoretisch verdeeld worden in twee delen, namelijk een besturingsdeel en een dataverwerkingsdeel. De dataverwerking manipuleert gegevens en bestaat typisch uit registers en logica voor de gegevensbewerking. De besturing zorgt er voor dat het dataverwerkingsdeel op het juiste moment de juiste functie uitvoert. Het levert de stuursignalen, die er voor zorgen dat de registers worden geladen, dat tellers worden geactiveerd en multiplexers de juiste selectie hebben. Om de juiste beslissingen te maken gebruikt de besturing de statussignalen van het dataverwerkingsdeel en externe ingangssignalen.

Het digitale systeem zal zich altijd in een bepaalde toestand bevinden. Het systeem laadt de gegevens die verwerkt moeten worden, voert de bewerking uit of schrijft het resultaat weg. Een typisch voorbeeld is de instructiecyclus bij een microprocessor met *fetch*, *decode* en *execute*.

Een ander voorbeeld, dat in veel handboeken over digitale systemen beschreven wordt, is de *traffic light controller* of verkeerslichtinstallatie. De kruising uit figuur 5.1 tussen een hoofdweg en een zijweg kent een eenvoudige verkeersregeling met vier toestanden: de hoofdweg heeft groen, de hoofdweg heeft oranje of geel, de zijweg heeft groen en de zijweg heeft oranje of geel.

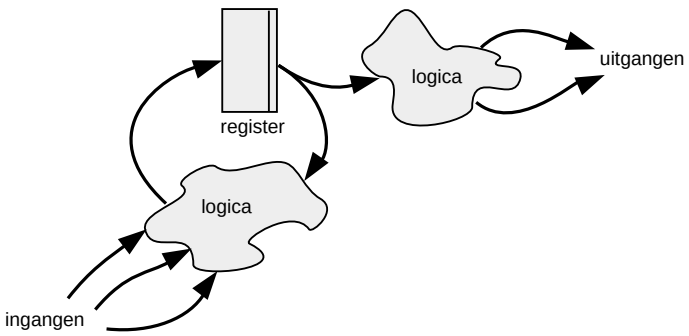


Figuur 5.1 : **De kruising tussen een hoofdweg en een zijweg.** Er zijn vier verschillende toestanden te onderscheiden, die elkaar opvolgen.

De vier toestanden worden in een vaste volgorde na elkaar doorlopen. Als er geen auto op de zijweg is, heeft de hoofdweg groen. Als er een auto op de zijweg verschijnt, worden de verkeerslichten op de hoofdweg enkele seconden oranje en krijgt de zijweg groen. Na een vaste tijd van een tiental seconde worden de verkeerslichten op de zijweg enkele seconden oranje en krijgt de hoofdweg weer groen.

5.1 Het concept van een toestandsmachine

Het besturingsdeel van een digitaal systeem kan beschreven worden met een zogenoemde toestandsmachine en bestaat uit drie delen: een register om de huidige toestand te bewaren, een stuk logica om de volgende toestand te berekenen en een stuk logica om de uitgangswaarden te berekenen.



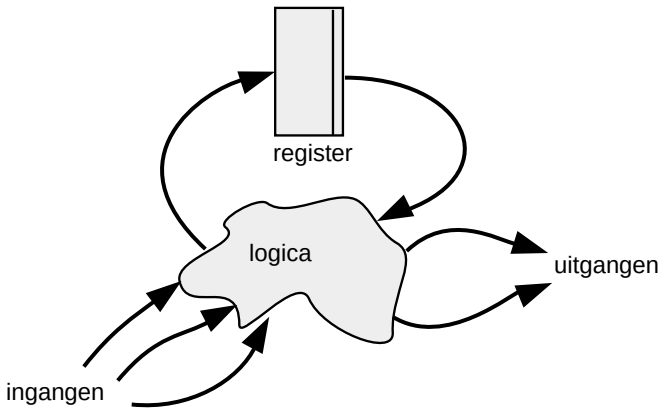
Figuur 5.2 : **Het principe van een toestandsmachine.** Het register onthoudt de huidige toestand en de logica berekent uit de huidige toestand en de ingangswaarden de volgende toestand en de uitgangswaarden.

Het register bewaart de toestand van de machine. Bij de verkeerslichtinstallatie zijn vier toestanden mogelijk en is er een register nodig met minimaal twee bits.
De volgende toestand wordt door de logica bepaalt uit de huidige toestand en uit de waarden van de ingangen. De ingangen mogen externe signalen zijn, die van buiten komen, of statussignalen uit het dataverwerkingsdeel van het systeem waar de toestandsmachine de besturing van is.
Uit de huidige toestand berekent de logica de waarden van de uitgangssignalen. Dat kunnen externe signalen zijn of de stuursignalen die het dataverwerkingsdeel van het digitale systeem besturen.

Een toestandsmachine of *state machine* wordt ook FSM of *finite state machine* genoemd. Het begrip *finite*, of eindig, betekent dat er een eindig aantal toestanden is.

Een oneindige toestandsmachine kan ook worden gedefinieerd, maar heeft geen praktische betekenis.

Het register bewaart de toestand van de machine. Daarom noemt men dit het toestandsregister of *state register*. De huidige toestand wordt ook de *present state* of *current state* genoemd en de volgende toestand noemt men ook de *next state*.



Figuur 5.3 : Een toestandsmachine met een gecombineerde toestandsdecoder en uitgangsdecoder.

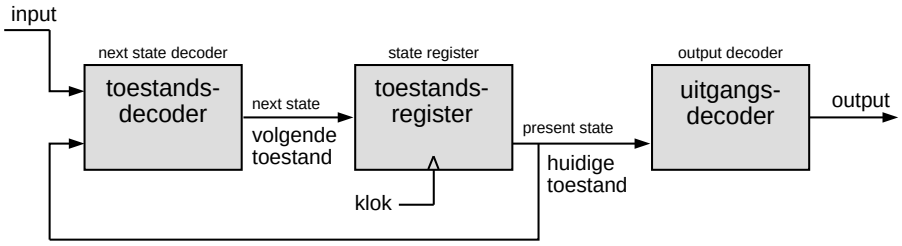
De logica, die de volgend toestand uitrekent, is de toestandsdecoder of *next state decoder*. De logica, die de uitgangen bepaalt is de uitgangsdecoder of *output decoder*. De logische blokken van de toestandsdecoder en de uitgangsdecoder kunnen ook samen genomen worden. In figuur 5.3 heeft één blok logica dat, uit de huidige toestand en de ingangssignalen, de volgende toestand en de uitgangssignalen berekent.

5.2 De Moore-machine en de Mealy-machine

Een consequentie van de gecombineerde toestandsdecoder en uitgangsdecoder van figuur 5.3 is, dat er een combinatorisch pad van de ingangen naar de uitgangen kán zijn. Het tijdsgedrag van een toestandsmachine met een combinatorische verbinding tussen de in- en uitgangen verschilt essentieel met dat van een toestandsmachine zonder een dergelijke verbinding. Daarom worden er twee soorten toestandsmachines onderscheiden: de Moore-machine en de Mealy-machine.
In figuur 5.4 staat het blokschema van een Moore-machine. Er is geen directe verbinding tussen de in- en uitgangen. Als een ingang verandert, berekent de

toestandsdecoder de volgende toestand, die bij de eerstvolgende klokflank in het toestandsregister wordt geplaatst. Uit de nieuwe huidige toestand berekent de toestandsdecoder nieuwe uitgangswaarden.

De Moore-machine is genoemd naar Edward F. Moore (1925-2003), een wiskundige die — net als Mealy — een van de grondleggers van de automatentheorie is.



Figuur 5.4 : **De Moore-machine.** Er kan geen combinatorisch pad tussen de in- en uitgangen zijn. Alle signalen lopen via het toestandsregister.

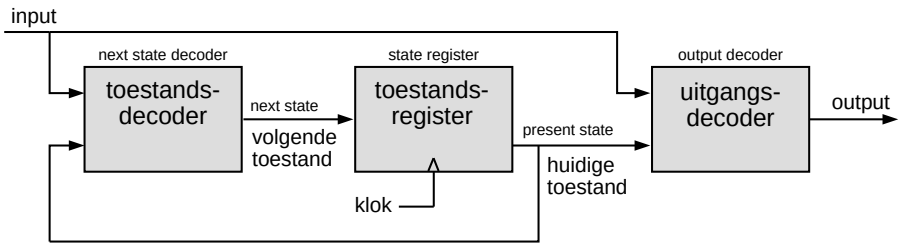
Bij een Moore-machine hangen de uitgangswaarden alleen af van de huidige toestand. De uitgang is slechts een functie van de huidige toestand:

$$\texttt{output} = f(\texttt{present state}) \tag{5.1}$$

De uitgangen van een Moore-machine veranderen pas als er een nieuwe waarde in het toestandsregister staat. Met als gevolg dat een Moore-machine altijd synchroon is met de klok van de toestandsmachine.

Figuur 5.5 geeft het blokschema van een Mealy-machine. Er kan nu wel een directe verbinding zijn tussen één of meer van de ingangen en één of meer van de uitgangen. Als een ingang verandert, berekent de toestandsdecoder — net als bij de Moore-machine — de volgende toestand, die bij de eerstvolgende klokflank in het toestandsregister geplaatst wordt. Tegelijkertijd berekent de uitgangsdecoder nieuwe waarden voor de uitgangen. Deze uitgangen veranderen direct, dus voordat het toestandsregister zijn nieuwe huidige waarde heeft gekregen.

De Mealy-machine is genoemd naar George H. Mealy (1927-), die in de jaren vijftig bij Bell Labs werkzaam was en net als Moore een van de grondleggers van de automatentheorie is.

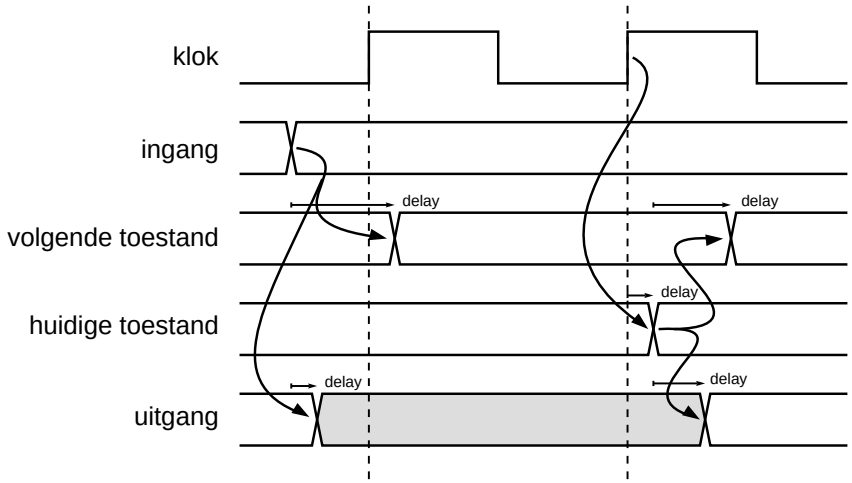


Figuur 5.5 : **De Mealy-machine.** Er kan een combinatorisch pad tussen de in- en uitgangen zijn.

Het grote verschil in het gedrag tussen een Moore-machine en een Mealy-machine is dat de uitgangen van de Moore-machine synchroon en die van een Mealy-machine asynchroon zijn. De uitgangen van een Moore-machine veranderen altijd na de actieve klokflank. De uitgangen van een Mealy-machine reageren direct op een ingangsverandering en krijgen hun nieuwe waarden voor de actieve klokflank. Bij een Mealy-machine zijn de uitgangen een functie van de huidige toestand en van de ingangen:

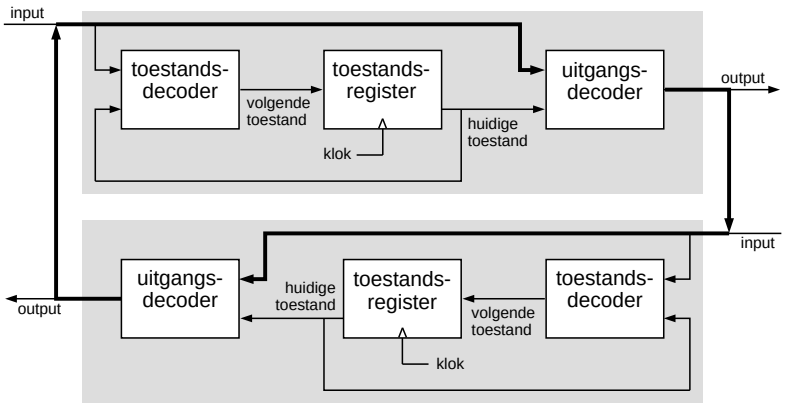
$$\texttt{output} = f(\texttt{present state}, \texttt{input}) \tag{5.2}$$

De Mealy-machine kent vanwege het asynchrone gedrag twee problemen. Ten eerste kan als een ingang in de nabijheid van een actieve klokflank verandert, de toestand pas bij de volgende actieve klokflank veranderen. Terwijl de uitgang al voor de huidige klokflank gewijzigd is. Figuur 5.6 laat zien dat de uitgang niet één klokflank, maar twee klokflanken hoog kan zijn. Als dit uitgangssignaal de enable van een teller is, wordt de teller met twee in plaats van met één verhoogd.



Figuur 5.6 : Het gedrag van een Mealy-machine nabij een klokflank. Door verschillen in de tijdvertragingen verandert het uitgangssignaal voor de eerste actieve klokflank en de volgende toestand na deze flank. De huidige toestand verandert daardoor na de tweede klokflank. Het effect is dat de uitgang bij beide klokflanken actief (grijs) is.

Het tweede probleem kan ontstaan bij een digitaal systeem met twee Mealy-machines. Als de ingangen van de ene machine aan de uitgangen van de andere gekoppeld zijn, ontstaat er een combinatorisch pad, zoals in figuur 5.7 is getekend. Een gesloten combinatorische lus kan altijd gaan oscilleren en dat is ongewenst.

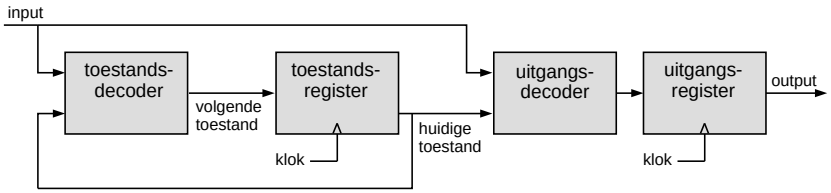


Figuur 5.7 : Het combinatorische pad bij twee tegengekoppelde Mealy-machines.

Bij het ontwerp van digitale systemen verdient de Moore-machine vanwege het gunstiger tijdsgedrag de voorkeur boven een Mealy-machine. In dit boek wordt bij toestandsmachines bijna altijd een Moore-machine gebruikt. Dit hoofdstuk gebruikt om het onderscheid duidelijk te maken een enkele keer een Mealy-machine. In paragraaf 7.2 is ook voor een Mealy-machine gekozen.

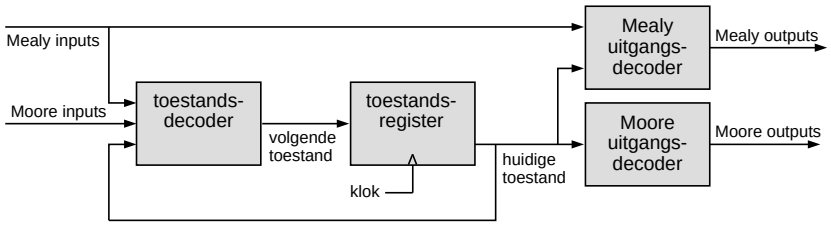
Bij de ontwikkeling van software worden eveneens toestandsmachines gebruikt. Daarbij is het opvallend dat software-ontwikkelaars vaak Mealy-machines toepassen. Omdat alle bewerkingen via de processor lopen, is dat geen probleem. De processor plaatst alle resultaten immers in een register. Er kunnen dan geen oscillaties optreden en er zijn geen andere problemen met het tijdsgedrag.

In hardware mag alleen een Mealy-machine worden gebruikt als de uitgangen een flipflop krijgen, zoals in figuur 5.8 is getekend. De uitgangen zijn dan gesynchroniseerd en mogen dan weer een ingang van een andere Mealy-machine zijn.



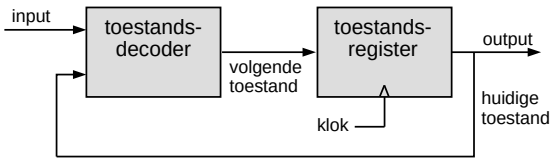
Figuur 5.8 : **Een Mealy-machine met een uitgangsregister.**

In het voorafgaande is steeds verondersteld dat alle uitgangen van een Mealy-machine ook echte Mealy-uitgangen zijn. Dat hoeft echter niet. Een toestandsmachine kan uitgangen hebben die alleen van de huidige toestand afhangen en ook uitgangen hebben die van de huidige toestand en een of meer ingangen afhangen. In figuur 5.9 staat een toestandsmachine met zowel Mealy- als Moore-uitgangen.



Figuur 5.9 : **Een gemengde Mealy- en Moore-machine.** Deze machine heeft twee aparte toestandsdecoders voor de Mealy- en de Moore-uitgangen.

Voor sommige toepassingen is het ook mogelijk om een toestandsmachine zonder uitgangsdecoder te gebruiken. Ieder uitgang komt dan overeen met een bepaalde toestand van de machine. In figuur 5.10 staat deze zogenoemde Medvedev-machine.

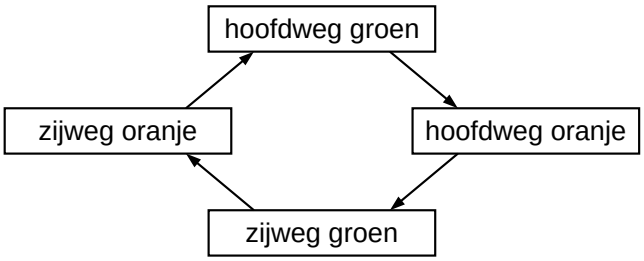


Figuur 5.10 : **De Medvedev-machine.** De uitgangen zijn tevens de huidige toestand van de machine.

Zoals al eerder in deze paragraaf is opgemerkt, is het verstandig bij hardware altijd een Moore-machine te gebruiken.

5.3 Toestandsdiagrammen en transitietabellen

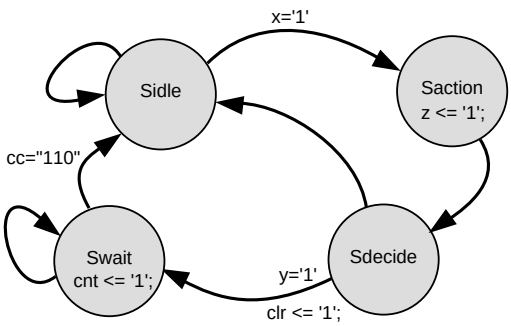
In de vorige paragraaf is uitgelegd dat een toestandsmachine opgebouwd is uit een toestandsregister, toestandsdecoder en uitgangsdecoder. Toch is dat niet de methode waarmee de functionaliteit van de toestandsmachine wordt beschreven. Dit beschrijft alleen de manier waarop een toestandsmachine wordt gerealiseerd. De functionaliteit van een toestandsmachine wordt vastgelegd met een toestandstransitietabel, een toestandsdiagram of een hardwarebeschrijvingstaal. Deze notaties leggen de toestanden vast en definiëren de volgorde waarin deze toestanden worden doorlopen.
Figuur 5.11 toont de vier toestanden van de verkeerslichtinstallatie uit figuur 5.1. De pijlen geven de volgorde aan waarin deze toestanden worden doorlopen.



Figuur 5.11 : De vier toestanden van de verkeerslichtinstallatie.

Aan het diagram van figuur 5.11 ontbreken de overgangsvoorwaarden en uitgangsspecificaties.

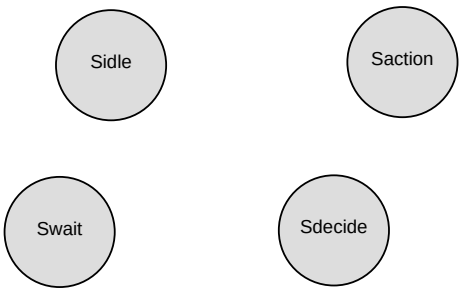
Er bestaat geen officiële notatie voor de toestandsdiagrammen. Hoewel de verschillen meestal niet groot zijn, gebruikt ieder boek en ieder programma voor het maken van toestandsdiagrammen zijn eigen notatie. Meestal worden er cirkels, bollen of *bubbles* voor de toestanden gebruikt. Men noemt een toestandsdiagram of *state diagram* daarom ook wel bolletjesdiagram of *bubble diagram*.
In figuur 5.12 staat een toestandsdiagram voor een willekeurige toestandsmachine. Bij dit voorbeeld gaat het niet om de functionaliteit, maar om de betekenis van de symbolen. Het diagram beschrijft een gemengde Mealy- en Moore-machine.



Figuur 5.12 : Een toestandsdiagram met vier toestanden.

Het toestandsdiagram omvat alle informatie die nodig is om de toestandsmachine te beschrijven. Het bevat de toestanden, de toestandsovergangen met de overgangsvoorwaarden en de uitgangsspecificaties.

In dit voorbeeld heeft het toestandsdiagram vier toestanden. Iedere toestand heeft een unieke naam. Veel boeken gebruiken namen als `S0`, `S1`, `S2` en zo verder. Bij complexe machines en bij de VHDL-beschrijvingen van een toestandsdiagram is dat niet overzichtelijk. Het is beter om een functionele naam te kiezen. Te lange namen maken het diagram eveneens onduidelijk. In dit geval bestaan de toestandsnamen uit de hoofdletter `S` van *State* en korte titel die de toestand beschrijft.

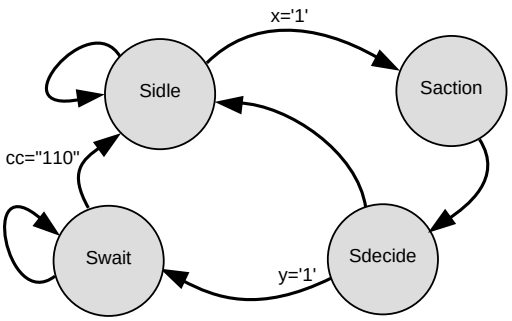


Figuur 5.13 : De vier toestanden van de toestandsmachine.

Het Engelse woord *Idle* betekent ijdel, nietsdoen of niet aan het werk.

De toestand `Sidle` is de naam van de begintoestand. Hier doet het systeem niets; de toestand is *idle*. Bij de toestand `Saction` wordt een bepaalde actie uitgevoerd. Bij de toestand `Sdecide` wordt een beslissing genomen en bij `Swait` wordt gewacht.

De pijlen geven de mogelijke richtingen waarin het toestandsdiagram doorlopen wordt. Ieder pijl is een overgang of *transition* van de ene toestand naar een andere toestand. Als er vanuit één toestand meerdere pijlen vertrekken, moet duidelijk zijn onder welke omstandigheid in welke richting gegaan wordt.



Figuur 5.14 : De overgangen en de overgangsvoorwaarden.

De overgangsvoorwaarden zijn hier in VHDL-formaat genoteerd. Veel programma's voor het tekenen van toestandsdiagrammen gebruiken deze notatie. Deze voorwaarden worden dan één op één bij de conditionele toekenningen van de gegenereerde code gebruikt.

Vanuit toestand `Sidle` wijzen twee pijlen: één wijst naar `Saction` en één wijst naar de toestand `Sidle`. Zolang signaal `x` laag is blijft de machine in de toestand `Sidle`, als `x` hoog is, wordt de volgende toestand `Saction`. Er staat alleen een overgangsvoorwaarde of *transition condition* bij de pijl die naar `Saction` wijst.
Bij toestand `Saction` staat één uitgaande pijl zonder overgangsvoorwaarde. Dit betekent dat de machine slechts één klokslag in toestand `Saction` blijft en direct doorgaat naar `Sdecide`. Bij de toestand `Sdecide` gaat de machine na één klokslag door naar toestand `Swait` als `y` hoog is en anders terug naar `Sidle`. De machine blijft in toestand `Swait` zolang signaal `cc` ongelijk aan `110` is. Pas als `cnt` gelijk aan `110` is, gaat de machine terug naar `Sidle`.

De complete toestandsmachine van figuur 5.14 bevat twee Moore-uitgangen en één Mealy-uitgang. Bij toestand `Saction` staat de VHDL-toewijzing `z<='1';`. Als de machine in deze toestand is, is deze uitgang hoog en in alle andere toestanden is deze uitgang laag. Op dezelfde manier is uitgang `cnt` hoog als de toestandsmachine zich in toestand `Swait` bevindt.
De toekenning aan het uitgangssignaal `clr` staat bij de overgang van toestand `Scompare` naar `Swait`. Bij toestand `Scompare` zijn er twee mogelijkheden. Als ingangssignaal `y` laag is, is de volgende toestand `Sidle` en blijft signaal `clr` laag. Als `y` hoog is, is de volgende toestand `Swait` en wordt het uitgangssignaal `clr` hoog. De waarde van `clr` hangt dus af van de toestand en van de waarde van ingang `y`. Signaal `clr` is dientengevolge een Mealy-uitgang.

Toestandstransitietabel

Een andere methode om het gedrag van een toestandsmachine vast te leggen is een toestandstransitietabel. Dat is een functietabel met de huidige toestand, de ingangen, de volgende toestand en de uitgangen. In tabel 5.1 staat de toestandstransitietabel van de toestandsmachine waarvan het toestandsdiagram in figuur 5.12 staat.

Tabel 5.1 : De toestandstransitietabel van het toestandsdiagram uit figuur 5.12. In de kolom `prs` staat de huidige toestand of *present state* en in de kolom `nxt` staat de volgende toestand of *next state*.

prs	x	y	cc	nxt	z	cnt	clr
Sidle	0	-	---	Sidle	0	0	0
Sidle	1	-	---	Saction	0	0	0
Saction	-	-	---	Sdecide	1	0	0
Sdecide	-	0	---	Sidle	0	0	0
Sdecide	-	1	---	Swait	0	0	1
Swait	-	-	$\neq$ 110	Swait	0	1	0
Swait	-	-	110	Sidle	0	1	0

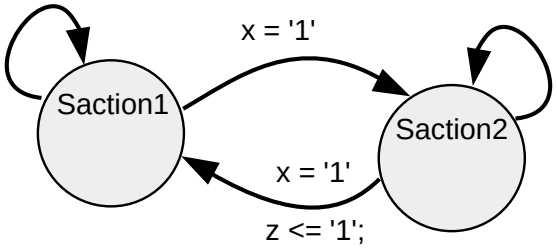
De toestandstransitietabel bevat dezelfde informatie als het toestandsdiagram. De tabel is compacter dan het diagram. Het diagram is op zijn beurt beter leesbaar en eenvoudiger te interpreteren.
De tabel laat — net als het diagram — zien dat de uitgangen `z` en `cnt` van het type Moore zijn en dat de uitgang `clr` van het type Mealy is. De uitgangssignalen `z` en `cnt` hangen alleen van de toestand af: `z` is alleen in toestand `Saction` hoog en signaal `cnt` is alleen in toestand `Swait` hoog. Uitgangssignaal `clr` is alleen hoog in toestand `Sdecide` als bovendien tegelijkertijd ingangssignaal `y` hoog is.

Toestandsmachines vastleggen met vergelijkingen en schema's

Naast tabellen en diagrammen kan een toestandsmachine ook gerepresenteerd worden door middel van logische vergelijkingen en met elektrische schema's. Om dit te kunnen doen, moet er een toestandscodering voor de toestanden gekozen worden. De vergelijkingen van de toestandsdecoder en de uitgangsdecoder van de machine van het diagram uit figuur 5.12 zijn anders als de toestandscodering van `Sidle`, `Saction`, `Sdecide` en `Swait` `00`, `01`, `10` en `11` of als de codering `0001`, `0010`, `0100` en `1000` is.

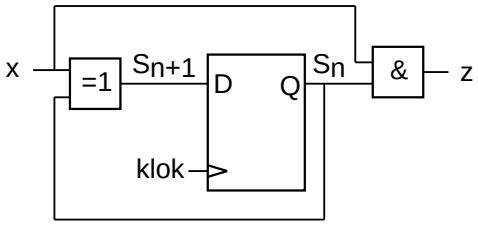
De toestandscodering bij toestandsmachines wordt in paragraaf 5.8 toegelicht.

In figuur 5.15 staat een toestandsdiagram van een Mealy-machine. Uitgang `z` is een Mealy-uitgang, omdat `z` afhangt van ingang `x`. Signaal `z` is alleen hoog wanneer de machine in toestand `Saction2` is en tegelijkertijd ingang `x` hoog is.



Figuur 5.15 : Voorbeeld van Mealy-machine met twee toestanden.

In figuur 5.16 staan drie andere representaties van dezelfde toestandsmachine, namelijk: een schema, een set met logische vergelijkingen en een toestandstransitietabel. Om een schema te maken en om de vergelijkingen op te stellen, moet er een toestandscodering voor de toestanden gekozen worden. In dit geval is er voor de twee toestanden één D-flipflop gebruikt. De toestand `Saction1` komt overeen met een `0` en de toestand `Saction2` komt overeen met `1`.

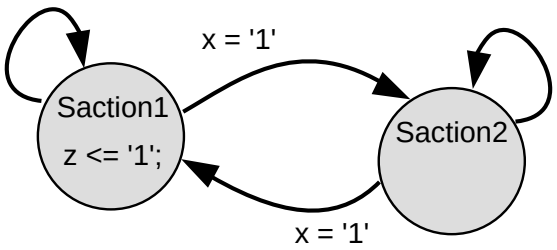


$S_{n+1} \leftarrow S_n$ **xor** x;
z $\leftarrow S_n$ **and** z;

S_n	x	S_{n+1}	clr
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figuur 5.16 : Verschillende representaties van het Mealy-diagram van figuur 5.15.
Rechts staat de toestandstransitietabel en daarnaast staan het schema en de logische vergelijking voor de situatie wanneer de toestanden `Saction1` en `Saction2` door respectievelijk een `0` en een `1` worden gerepresenteerd.

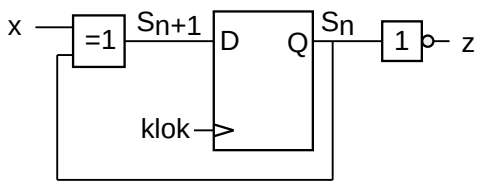
In het schema van figuur 5.16 zijn de onderdelen van de toestandsmachine duidelijk te herkennen: de D-flipflop is het toestandsregister, de XOR-poort is de toestandsdecoder en de AND-poort is de uitgangsdecoder.
In alle representaties is duidelijk te zien dat de uitgang niet alleen afhangt van de huidige toestand S_n maar ook van ingang `x` en dat het een Mealy-machine betreft.



Figuur 5.17 : Voorbeeld van Moore-machine met twee toestanden.

In figuur 5.17 staat een toestandsdiagram van een Moore-machine. Uitgang `z` is een Moore-uitgang, omdat `z` alleen afhangt van de toestand. Signaal `z` is hoog als de machine in toestand `action1` is en laag als deze in `Saction2` is.

Figuur 5.18 geeft drie andere representaties van deze toestandsmachine, namelijk: een schema, een set met logische vergelijkingen en een toestandstransitietabel. Voor het schema en de vergelijkingen is gekozen om de twee toestanden weer met één D-flipflop vast te leggen. Ook nu komt de toestand `Saction1` overeen met een `0` en de toestand `Saction2` met `1`.



$S_{n+1} \leftarrow S_n$ **xor** x;
z $\leftarrow$ **not** S_n

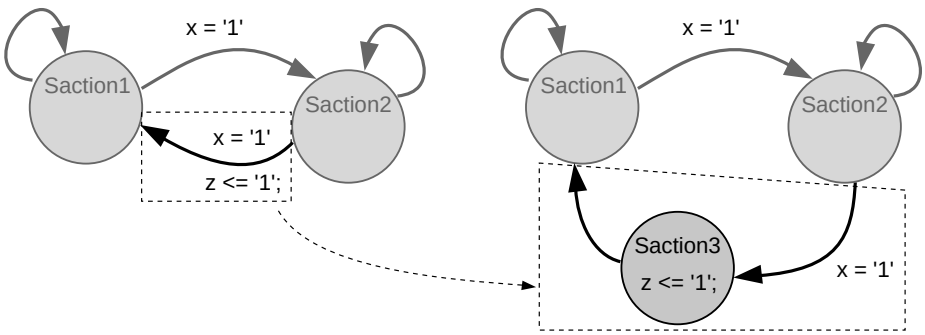
S_n	x	S_{n+1}	clr
0	0	0	1
0	1	1	1
1	0	1	0
1	1	0	0

Figuur 5.18 : Verschillende representaties van het Moore-diagram van figuur 5.17.
Rechts staat de toestandstransitietabel en daarnaast staan het schema en de logische vergelijking voor de situatie wanneer de toestanden `Saction1` en `Saction2` door respectievelijk een `0` en een `1` worden gerepresenteerd.

In het schema van figuur 5.18 zijn de onderdelen van de toestandsmachine duidelijk te herkennen: de D-flipflop is het toestandsregister, de XOR-poort is de toestandsdecoder en de inverter is de uitgangsdecoder.
In alle drie de representaties is zichtbaar dat de uitgang alleen van de huidige toestand s_n afhangt en dat het een Moore-machine is.

5.4 Een Mealy-model omzetten naar een Moore-model

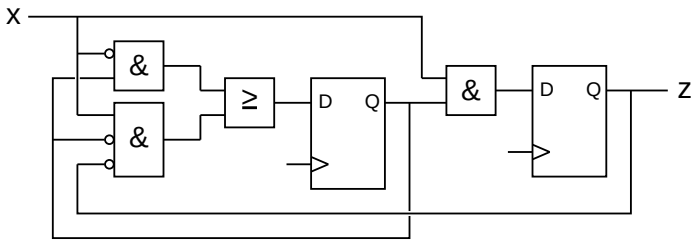
Ieder Mealy-machine kan worden herschreven als een Moore-machine. Het toestandsdiagram van figuur 5.15 beschrijft een Mealy-machine, omdat de uitgang `z` bij de overgang van `Saction2` naar `Saction1` staat.



Figuur 5.19 : Het omzetten van een Mealy-diagram naar een Moore-diagram. Links staat het Mealy-diagram van figuur 5.15. Rechts staat een Moore-versie van het diagram.

Door de pijl te veranderen in twee pijlen met een extra toestand — zoals in figuur 5.19 is getekend — wordt het een Moore-diagram. De uitgangsspecificatie staat niet meer bij een pijl, maar bij toestand `Saction3`. De overgangsvoor-

waarde staat nu bij de pijl van `Saction2` naar `Saction3`. Bij de pijl van `Saction3` naar `Saction1` staat geen voorwaarde. De toestandsmachine blijft één klokslag in toestand `Saction3` en signaal `z` is precies één klokslag hoog.

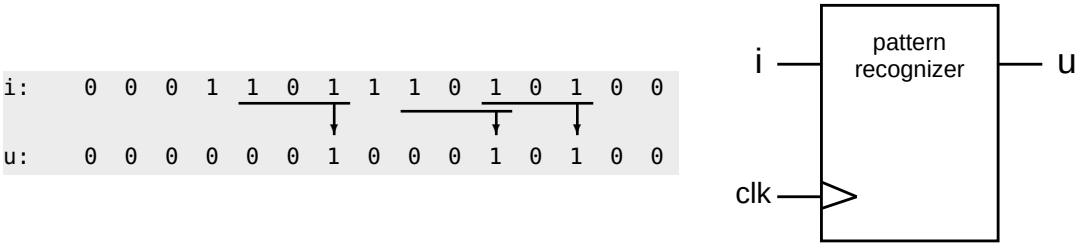


Figuur 5.20 : Een implementatie van het Moore-diagram van figuur 5.19 met twee flipfloppen.

Bij de implementatie van het toestandsdiagram uit figuur 5.19 zijn minimaal twee flipfloppen nodig om de drie toestanden mee vast te leggen. In figuur 5.20 is een realisatie getekend met twee flipfloppen, waarbij de toestanden `Saction1`, `Saction2`, en `Saction3` gecodeerd zijn met `00`, `01` en `10`. Signaal `z` is hoog in `Saction3`. Dit is alleen het geval als de tweede flipflop hoog is. Signaal `z` komt overeen met de uitgang van deze flipflop. Bij deze toestandscodering is de Moore-machine toevallig geïmplementeerd als Medvedev-machine.

5.5 Opstellen van een toestandsdiagram voor een Moore-machine

Er zijn verschillende manieren om een toestandsmachine op te stellen. Een methode is om eerst alle scenario's op te schrijven en dan voor ieder scenario alle toestanden op te schrijven. Een andere aanpak is om vanuit de begintoestand stap voor stap alle toestanden toe te voegen.



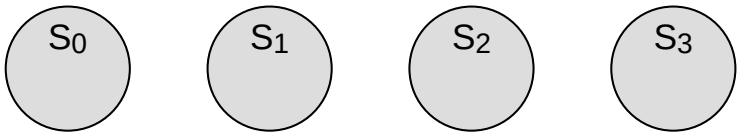
Figuur 5.21 : De seriële patroonherkenner. Links staat een voorbeeld van een ingangspatroon met het bijbehorende uitgangssignaal. Rechts staat het symbool van de herkenner.

In deze paragraaf wordt het Moore-diagram van een seriële patroonherkenner opgesteld. Deze herkenner heeft een ingang `i` en een uitgang `u`. Iedere klokslag leest de herkenner de ingangswaarde `i`. De uitgang `u` wordt hoog indien de laatste drie gelezen ingangswaarden achtereenvolgens 1, 0 en 1 zijn. In alle andere situaties is de uitgang laag. Figuur 5.21 geeft een voorbeeld van dit gedrag en laat zien dat de patronen elkaar ook mogen overlappen.

Bij een Moore-machine is het handig om eerst alle toestanden te definiëren. De seriële herkenner detecteert een patroon van drie waarden, daarbij zijn vier situaties te onderscheiden:

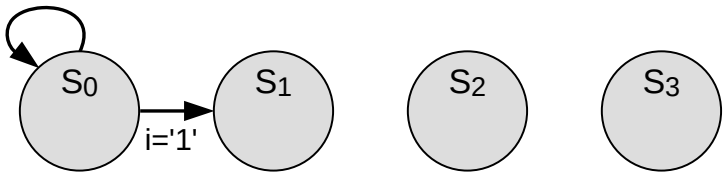
- er is nog niets van het patroon gevonden;
- de eerste waarde van het patroon is herkend;
- de eerste twee waarden van het patroon zijn herkend;
- het hele patroon is gelezen.

De Moore-machine van de patroonherkenner heeft dus vier toestanden waarin deze machine zich kan bevinden.



Figuur 5.22 : De vier toestanden van het toestandsdiagram van de seriële herkenner.

In figuur 5.22 zijn de vier toestanden s_0, s_1, s_2 en s_3 getekend. De gekozen toestandsnamen komen overeen met het aantal bits dat van het patroon gelezen is. In de begintoestand s_0 — wanneer er nog niets van het patroon is gevonden — zijn er twee mogelijkheden: er staat een `0` of een 1 op de ingang. Als er een 1 op de ingang staat, is de eerste bit uit het patroon gelezen en wordt de volgende toestand s_1. Als er op de ingang een `0` staat, blijft de toestandsmachine in toestand s_0.



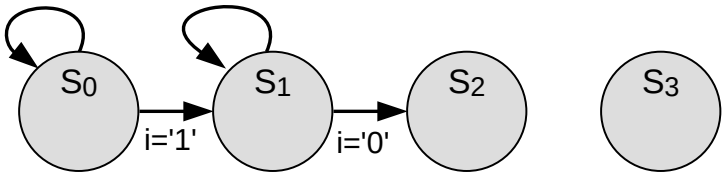
Figuur 5.23 : De twee toestandsovergangen bij toestand s_0.

In ontwerpprogramma's voor toestandsmachines wordt de standaardconditie vaak met een speciale code aangeduid, bijvoorbeeld: `@else`.

Figuur 5.23 laat zien dat er vanuit de begintoestand twee pijlen zijn getekend: één naar toestand s_1 en één naar zichzelf. Bij de pijl naar s_1 staat de overgangsvoorwaarde of overgangsconditie `i='1'`. Bij de andere pijl staat niets. Dat is de overgang die wordt genomen als alle andere — niet genoemde — condities niet waar zijn. In dit geval is deze standaardconditie gelijk aan `i='0'`.

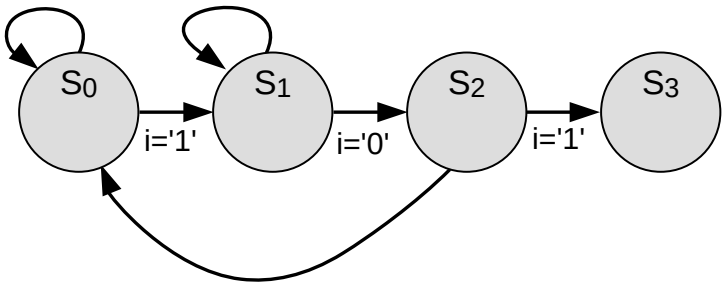
Figuur 5.24 toont dat er vanuit toestand s_1 weer twee mogelijkheden zijn. Als de ingang `0` is, is de tweede bit uit het patroon gevonden en wordt de volgende toestand s_2. Als de ingang 1 is, is het gelezen patroon gelijk aan 11. Omdat de patronen mogen overlappen, kan de laatst gelezen 1 de eerste waarde van een nieuw patroon zijn. Daarom blijft de toestandsmachine in toestand s_1. De standaardconditie voor deze overgang is `i='1'`.

In figuur 5.25 zijn de toestandsovergangen voor toestand s_2 aan het diagram toegevoegd. Als er in toestand s_2 op de ingang een 1 verschijnt, is de derde waarde van het patroon gevonden. Er zijn dan drie correcte waarden gelezen en de volgende toestand is dan s_3. Als er een `0` op de ingang staat, zijn de laatste drie gelezen



Figuur 5.24 : De twee toestandsovergangen bij toestand s_1.

waarden 1, 0 en 0. Hierin zit niet een begin van een nieuw patroon. De volgende toestand is dan de begintoestand s_0.



Figuur 5.25 : De toestandsovergangen bij toestand s_2.

In figuur 5.27 zijn alle toestandsovergangen van het diagram getekend. In toestand s_3 zijn de laatste drie waarden 0, 1 en 1 als de ingang hoog is en 0, 1 en 0 als de ingang laag is. In het eerste geval is er weer één waarde van het patroon gelezen en in het tweede geval zijn er twee waarden gelezen, zoals figuur 5.26 laat zien.

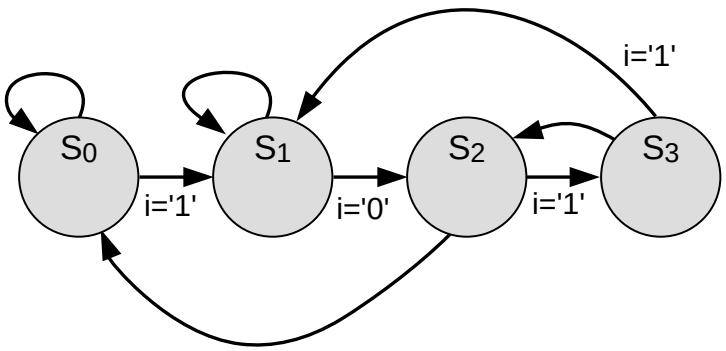
s_3

1	0	1	1	·	·

s_3

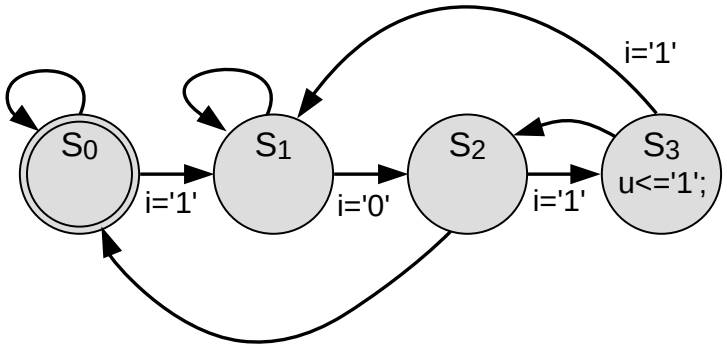
1	0	1	0	·

Figuur 5.26 : Patronen na toestand s_3. In s_3 is het patroon gelezen. Als de volgende waarde een 1 is, kan dat de eerste waarde van een nieuw patroon zijn. Als dat een 0 is, kan dat de tweede waarde van een nieuw patroon zijn.



Figuur 5.27 : Alle toestandsovergangen van het diagram zijn getekend.

Nadat alle overgangen en overgangscondities zijn getekend, kunnen de uitgangswaarden worden toegevoegd. In dit voorbeeld is de uitgang hoog als alle drie de waarden uit het patroon gelezen zijn. Dat is het geval in toestand s_3. Figuur 5.28 toont het complete toestandsdiagram. Bij toestand s_3 staat dat de uitgang hoog wordt. Bij de drie andere toestanden staat niets. Uitgangen, die niet genoemd zijn bij een toestand, hebben de standaardwaarde. In dit geval is uitgang u standaard laag.



Figuur 5.28 : Het complete toestandsdiagram van de seriële herkenner.

In het diagram is geen expliciete standaardwaarde voor uitgang `u` gedefinieerd. Dat hoeft ook niet. Het feit dat bij toestand s_3 de uitgang hoog is, impliceert dat de standaardwaarde laag moet zijn. Bij ontwerpprogramma's voor toestandsdiagrammen moet de ontwerper voor alle uitgangen de standaardwaarden definiëren.

Bij sommige tekenprogramma's voor toestandsdiagrammen geeft een dubbele cirkel een hiërarchische toestand aan.

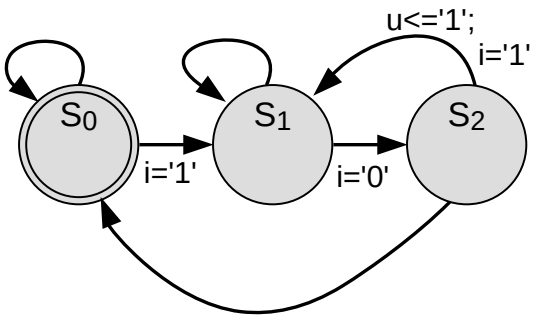
De dubbele cirkel bij toestand s_0 geeft aan dat dit de begintoestand is. Dat is de toestand waarin de toestandsmachine komt als de machine aan wordt gezet of als de machine een reset krijgt.

Het voorafgaande demonstreert dat het opstellen van een Moore-diagram uit drie stappen bestaat:

- het definiëren van alle toestanden;
- het aanbrengen van alle overgangen en overgangscondities;
- het opstellen van de uitgangsspecificaties.

Het Mealy-diagram van de seriële patroonherkenner

In figuur 5.29 staat het Mealy-diagram van de seriële herkenner. Het meest opvallende is dat er slechts drie toestanden zijn. De vierde toestand uit het Moore-diagram ontbreekt. De uitgangsspecificatie staat nu bij de overgang van toestand s_2 naar toestand s_1.

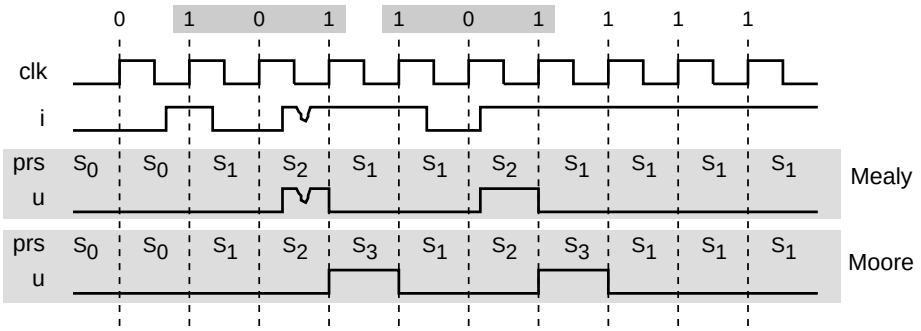


Figuur 5.29 : Het Mealy-diagram van de seriële herkenner.

Doordat er ook acties bij de pijlen staan is de relatie tussen de situaties waarin de machine zich kan bevinden en de toestanden van het diagram niet meer zo strikt. Het opstellen van een Mealy-diagram is daardoor wat meer intuïtief en wat minder systematisch.

5.6 Het tijdsgedrag van een Moore en Mealy-machine

Het Moore-diagram van figuur 5.28 en het Mealy-diagram van figuur 5.29 beschrijven allebei de seriële patroonherkenner. Het tijdsgedrag van de beide oplossingen is echter verschillend. In figuur 5.30 staat het signaaldiagram met zowel de uitgang `u` en de huidige toestand `prs` van de Moore-machine als die van de Mealy-machine.



Figuur 5.30 : Het tijdsgedrag van de seriële herkenner. Het signaaldiagram bevat zowel de huidige toestand `prs` en de uitgang `u` van de Moore-machine als die van de Mealy-machine. Het ingangssignaal `i` bevat twee keer het gezochte patroon. Bij de actieve klokflank staat de huidige waarde van `i`. De gezochte patronen hebben een grijze achtergrond. De uitgang van de Moore-machine verandert na de actieve klokflank en die van de Mealy-machine verandert voor de actieve klokflank.

Het ingangspatroon bevat twee maal het gezochte patroon. Nadat er twee bits van het gezochte patroon gelezen zijn, bevinden beide machines zich in toestand s_2. Als `i` hoog wordt reageert de uitgang `u` van de Mealy-machine direct. Het uitgangssignaal `u` van de Moore-machine verandert pas na de actieve klokflank. De Mealy-machine geeft alle verstoringen, *spikes* en *glitches* door aan de uitgang en heeft een asynchroon gedrag. De Moore-machine is altijd synchroon met de klok en verdient daarom de voorkeur bij digitale systemen.

5.7 VHDL-beschrijvingen van een toestandsmachine

Voor het beschrijven van een toestandsmachine in VHDL zijn een groot aantal verschillende stijlen beschikbaar. Het belangrijkste onderscheid tussen deze stijlen is het aantal processen dat gebruikt wordt. De Moore-machine van figuur 5.4 bestaat uit drie blokken: de toestandsdecoder, het toestandsregister en de uitgangsdecoder. Het ligt voor de hand om voor ieder blok een apart proces te gebruiken. De toestandsmachine bestaat dan uit drie processen. Door twee of meer blokken te combineren, kan de toestandsmachine ook opgebouwd zijn uit een of twee processen.

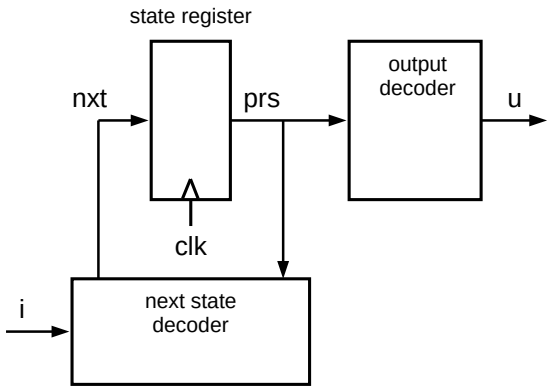
Onafhankelijk van de gebruikte stijl is de entity van de toestandsmachine bij alle stijlen hetzelfde. Deze paragraaf beschrijft het Moore-diagram van de seriële patroonherkenner uit figuur 5.28 in VHDL. De entity staat in code 5.1. Naast de ingang `i`, de uitgang `u` en het kloksignaal `clk` heeft de herkenner ook een actief lage asynchrone reset `rst_n`.

Code 5.1 : De entity van de seriële patroonherkenner.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity pattern_recognizer is
5    port (
6      clk   : in  std_logic;
7      rst_n : in  std_logic;
8      i     : in  std_logic;
9      u     : out std_logic
10   );
11 end entity pattern_recognizer;
```

Een Moore-machine met drie processen

In code 5.2 staat een VHDL-beschrijving, die opgebouwd is uit drie processen en figuur 5.31 geeft de samenhang tussen deze drie processen.



Figuur 5.31 : De beschrijving van de seriële herkenner met drie processen.

De signaalnamen `prs` en `nxt` zijn kort en voldoende duidelijk. Alternatieven zijn `present_state` en `next_state`. Niet geschikt is **`next`** omdat dit een gereserveerde naam is. Ook de namen `ps` en `ns` zijn onhandig omdat dat de tijdseenheden voor de pico- en nanoseconde zijn.

Het toestandsdiagram uit figuur 5.28 heeft vier toestanden $\texttt{S}_0$, $\texttt{S}_1$, $\texttt{S}_2$ en $\texttt{S}_3$. In VHDL kunnen de toestanden vastgelegd worden met een *enumerated type* of opsommingstype. In dit geval is in code 5.2 op regel 14 een opsommingstype `state_type` gedefinieerd die uit vier elementen bestaat: `S0`, `S1`, `S2` en `S3`. Met dit type worden op regel 15 de huidige toestand `prs` en de volgende toestand `nxt` gedefinieerd.

Op regel 17 tot en met 24 staat het proces `state_register` dat het toestandsregister beschrijft. Iedere klokslag wordt de volgende toestand `nxt` aan `prs` toegekend. Bij een asynchrone reset wordt de begintoestand `S0` aan `prs` toegekend.

Vanaf regel 26 tot en met regel 54 staat het proces `next_state_decoder` dat de toestandsdecoder beschrijft. De gevoeligheidslijst van deze decoder bevat altijd de huidige toestand en alle ingangssignalen. In dit geval zijn dat `prs` en het ingangssignaal `i`. Het proces is een groot case-statement waarin voor iedere toestand `prs` wordt beschreven wat de volgende toestand `nxt` is.

Code 5.2 : De architectuur van de seriële patroonherkenner met drie processen.

```
13 architecture gedrag of  pattern_recognizer is
14   type state_type is (S0, S1, S2, S3);
15   signal prs, nxt : state_type;
16 begin
17   state_register: process (clk, rst_n) is
18   begin
19     if rst_n = '0' then
20       prs <= S0;
21     elsif rising_edge(clk) then
22       prs <= nxt;
23     end if;
24   end process state_register;
25
26   next_state_decoder: process (prs, i) is
27   begin
28     case prs is
29     when S0 =>
30       if i = '1' then
31         nxt <= S1;
32       else
33         nxt <= S0;
34       end if;
35     when S1 =>
36       if i = '0' then
37         nxt <= S2;
38       else
39         nxt <= S1;
40       end if;
41     when S2 =>
42       if i = '1' then
43         nxt <= S3;
44       else
45         nxt <= S0;
46       end if;
47     when S3 =>
48       if i = '1' then
49         nxt <= S1;
50       else
51         nxt <= S2;
52       end if;
53     end case;
54   end process next_state_decoder;
55
56   output_decoder: u <= '1' when prs = S3 else '0';
57 end architecture gedrag;
```

De uitgangsdecoder staat op regel 56 en bestaat uit één conditionele signaaltoewijzing. Omdat deze signaaltoewijzing een proces voorstelt, heeft het een label `output_decoder` gekregen.

Het case-statement en de standaardoptie

Het proces `next_state_decoder` dat de toestandsdecoder beschrijft, bestaat uit één groot case-statement. Normaal gesproken heeft een case-statement altijd de standaardoptie **`when others`**, zoals in code 2.2. Zeker bij het signaaltype `std_logic` moeten de metawaarden, zoals `'Z'` en `'X'`, worden afgevangen. In dit geval is de standaardoptie niet nodig. Er zijn slechts vier expliciete toestanden, die alle vier bij het case-statement aan bod komen.

Als de standaardoptie wordt toegevoegd, maakt dat voor de simulatie niet uit. Bij de synthese zullen sommige synthesizers de **`when others`** negeren en andere geven een waarschuwing. Quartus van Altera negeert het en Leonardo Spectrum meldt:

```
Info, others clause is never selected for synthesis.
```

en waarschuwt:

```
Warning, Integer range is less than necessary range to cover others clause,
         may produce bad logic.
```

Bij een case-statement voor het opsommingstype is het beter om de standaardoptie weg te laten.

Alternatieven voor de uitgangsdecoder

In code 5.2 is voor de uitgangsdecoder een parallelle signaaltoewijzing gebruikt. Als er meerdere uitgangen zijn, bestaat de uitgangsdecoder uit meer processen:

```
u <= '1' when prs = S3 else '0';
v <= '1' when prs = S2 else '0';
w <= '1' when (prs = S1) or (prs = S0) else '0';
```

Als er drie uitgangen zijn, betekent dit dat de toestandsmachine niet uit drie processen, maar uit vijf processen bestaat. Toch spreken we bij dit model altijd over het drie-processenmodel. De uitgangen `w`, `y` en `z` kunnen immers ook met een enkel proces worden beschreven. In code 5.3 staat een uitgangsdecoder met één proces en een case-statement. Bij iedere toestand worden steeds alle uitgangssignalen beschreven. Als er veel uitgangen zijn, wordt dit omvangrijk en minder goed leesbaar.

Code 5.3 : Uitgangsdecoder met een proces.

```
56  output_decoder: process (prs) is
57  begin
58    case prs is
59    when S0 =>
60      w <= '1';  v <= '0';  u <= '0';
61    when S1 =>
62      w <= '1';  v <= '0';  u <= '0';
63    when S2 =>
64      w <= '0';  v <= '1';  u <= '0';
65    when S3 =>
66      w <= '0';  v <= '0';  u <= '1';
67    end case;
68  end process output_decoder;
```

Code 5.4 : Uitgangsdecoder met standaard uitgangswaarden.

```
56  output_decoder: process (prs) is
57  begin
58    u <= '0';
59    v <= '0';
60    w <= '0';
61    case prs is
62    when S0 =>  w <= '1';
63    when S1 =>  w <= '1';
64    when S2 =>  v <= '1';
65    when S3 =>  u <= '1';
66    end case;
67  end process output_decoder;
```

Een alternatieve beschrijving staat in code 5.4. Op regel 58, 59 en 60 worden de standaardwaarden voor de drie uitgangen gedefinieerd. Bij de toestanden staan alleen de uitgangen, die dan actief zijn. Signaal `u` is standaard laag en in toestand `S3` is dit signaal hoog.

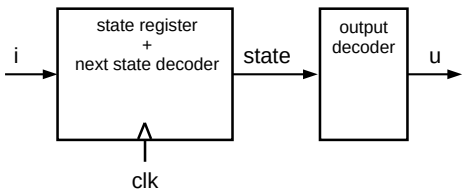
Het voordeel van de parallelle signaaltoewijzingen is dat de toestand, de standaardwaarde en de actieve waarde direct bij elkaar staan. In veel gevallen komen de namen van de uitgangen en toestandsnamen met elkaar overeen.

```
clear <= '1' when prs = Sclear else '0';
load  <= '1' when prs = Sload  else '0';
pop   <= '1' when (prs = Spop1) or (prs = Spop2) else '0';
```

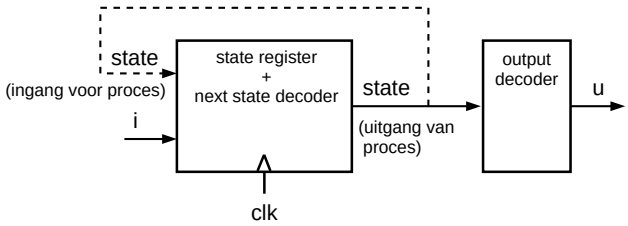
De uitgangsdecoder is dan een lijst, die eenvoudig opgesteld kan worden. Bij het debuggen kan een uitgang ook tijdelijk worden verwijderd door er commentaarstreepjes voor te zetten.

Een Moore-machine met twee processen en aparte uitgangsdecoder

Een andere stijl, die veel toegepast wordt, is een toestandsmachine met een gecombineerd proces voor de toestandsdecoder en het toestandsregister. Voor de seriële patroonherkenner is dit in figuur 5.32 getekend.



Figuur 5.32 : De beschrijving van de seriële herkenner met twee processen.



Figuur 5.33 : Het gedrag van signaal `state`. Signaal `state` is zowel uitgang als ingang van het proces.

In plaats van twee signalen `prs` en `nxt` voor de huidige en volgende toestand is er slechts één signaal `state`. Op regel 15 van code 5.5 is dit signaal gedeclareerd. Signaal `state` speelt zowel de rol van de huidige toestand als die van de volgende toestand en is dus enerzijds een uitgang en anderzijds een ingang van het proces `state_register_next_state_decoder`.

In figuur 5.32 is `state` alleen uitgang van het proces en geen ingang. Ondanks dat dit feitelijk niet correct is, wordt het toch vaak zo getekend, omdat het proces dit `state` oplevert en deze alleen van `i` afhangt. Figuur 5.33 past beter bij code 5.5. Signaal `state` is nu een uitgang en een ingang van het proces.
In code 5.5 is signaal `state` de volgende toestand — en een uitgang van het proces — als deze in het linkerlid van een toewijzing staat. Voorbeelden zijn de signaaltoewijzingen op regel 25 en op regel 27. Signaal `state` is de huidige toestand — en een ingang van het proces — als deze in het rechterlid van een toewijzing staat of als deze bij een voorwaarde in een conditionele toekenning wordt gebruikt. In code 5.5 wordt alleen bij het case-statement op regel 22 `state` als ingang gebruikt.

De impliciete else en expliciete else

Sommige if-statements in code 5.5, zoals die bij toestand `S0`, bevatten een expliciet else-statement dat in feite redundante informatie is. In figuur 5.34 staat links de situatie bij toestand `S0` met het expliciete else en rechts een verkorte notatie met een zogenoemde impliciete else. Als de toestandsmachine zich in toestand `S0` bevindt en de voorwaarde `i = 1` niet waar is, wordt er geen nieuwe waarde aan `state` toegekend en blijft de toestandsmachine in toestand `S0`.

Code 5.5 : Patroonherkenner met gecombineerde toestandsregister en toestandsdecoder.

```
13  architecture gedrag of pattern_recognizer is
14    type state_type is (S0, S1, S2, S3);
15    signal state : state_type;
16  begin
17    state_register_next_state_decoder: process (clk, rst_n) is
18    begin
19      if rst_n = '0' then
20        state <= S0;
21      elsif rising_edge(clk) then
22        case state is
23        when S0 =>
24          if i = '1' then
25            state <= S1;
26          else
27            state <= S0;
28          end if;
29        when S1 =>
30          if i = '0' then
31            state <= S2;
32          else
33            state <= S1;
34          end if;
35        when S2 =>
36          if i = '1' then
37            state <= S3;
38          else
39            state <= S0;
40          end if;
41        when S3 =>
42          if i = '1' then
43            state <= S1;
44          else
45            state <= S2;
46          end if;
47        end case;
48      end if;
49    end process state_register_next_state_decoder;
50
51    output_decoder: u <= '1' when state = S3 else '0';
52  end architecture gedrag;
```

```
case state is
when S0 =>
  if i = '1' then
    state <= S1;
  else
    state <= S0;
  end if;
when S1 =>
```

```
case state is
when S0 =>
  if i = '1' then
    state <= S1;
  end if;
when S1 =>
```

Figuur 5.34 : De expliciete en impliciete else. Links staat een fragment uit code 5.5 met een expliciete else. Rechts staat hetzelfde fragment met een impliciete else.

Bij het drie-processenmodel kan een impliciete else niet zomaar gebruikt worden. In figuur 5.35 staat links een fragment uit code 5.31 waar het else-statement bij toestand `S0` is weggelaten. Dit lijkt een impliciete else.

```
next_state_decoder: process (prs,i) is
begin
  case prs is
  when S0 =>
    if i = '1' then
      nxt <= S1;
    end if;
  when S1 =>
     ⋮
  end case;
end process next_state_decoder;
```

```
next_state_decoder: process (prs,i) is
begin
  nxt <= prs;   -- always a value for signal nxt
  case prs is
  when S0 =>
    if i = '1' then
      nxt <= S1;
    end if;
  when S1 =>
     ⋮
  end case;
end process next_state_decoder;
```

Figuur 5.35 : De impliciete else bij een toestandsmachine met drie processen. Links staat een fragment dat een variant is op een deel van code 5.2. De else van het if-statement is ten onrechte weggelaten. Rechts staat hetzelfde fragment met een standaardwaarde voor signaal `nxt`. De else mag nu wel worden weggelaten.

Omdat het drie-processenmodel voor de huidige en volgende toestand twee signalen gebruikt, krijgt signaal `nxt` niet altijd een waarde. Bij de synthese worden er latches aan de schakeling toegevoegd. Alle syntheseprogramma's geven een waarschuwing. Leonardo Spectrum waarschuwt:

```
Warning, nxt is not always assigned. Storage may be needed..
```

en meldt verder:

```
Info, DLATCH nxt(1) implemented using combinational logic
Info, DLATCH nxt(0) implemented using combinational logic
```

Quartus van Altera waarschuwt:

```
Warning (10631): VHDL Process Statement warning
  inferring latch(es) for signal or variable "nxt",
  which holds its previous value in one or more paths through the process
```

In figuur 5.35 staat rechts hetzelfde fragment met het if-statement zonder else, maar voor het case-statement krijgt signaal `nxt` eerst de waarde van de huidige toestand `prs`. Na het case-statement kan `nxt` veranderd zijn, maar in alle gevallen wordt er nu een waarde aan `nxt` toegekend. Het weglaten van de else impliceert nu dat de toestand niet verandert.

Een Moore-machine met twee processen en apart toestandsregister

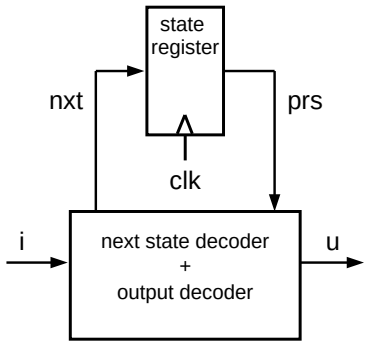
In code 5.6 staat een VHDL-beschrijving, die is opgebouwd uit twee processen. Er is een proces voor het toestandsregister en een proces voor een gecombineerde toestandsdecoder en uitgangsdecoder. In figuur 5.36 zijn de twee processen getekend.

Omdat dit model — net als het drie-processenmodel — een apart proces heeft voor het toestandsregister worden er twee aparte signalen voor de huidige en de vol-

Code 5.6 : Patroonherkenner met gecombineerde toestandsdecoder en uitgangsdecoder.

```
13  architecture gedrag of  pattern_recognizer is
14    type state_type is (S0, S1, S2, S3);
15    signal prs, nxt : state_type;
16  begin
17    state_register: process (clk, rst_n) is
18    begin
19      if rst_n = '0' then
20        prs <= S0;
21      elsif rising_edge(clk) then
22        prs <= nxt;
23      end if;
24    end process state_register;
25
26    logic: process (prs, i) is
27    begin
28      u <= '0';
29      case prs is
30      when S0 =>
31        if i = '1' then
32          nxt <= S1;
33        else
34          nxt <= S0;
35        end if;
36      when S1 =>
37        if i = '0' then
38          nxt <= S2;
39        else
40          nxt <= S1;
41        end if;
42      when S2 =>
43        if i = '1' then
44          nxt <= S3;
45        else
46          nxt <= S0;
47        end if;
48      when S3 =>
49        if i = '1' then
50          nxt <= S1;
51        else
52          nxt <= S2;
53        end if;
54        u <= '1';
55      end case;
56    end process logic;
57
58  end architecture gedrag;
```

gende toestand gebruikt. Het proces `logic` is gevoelig voor de huidige toestand `prs` en het ingangssignaal `i`. Proces `logic` komt voor een groot deel overeen met proces `next_state_decoder` uit code 5.2. De uitgangsdecoder is geïntegreerd door op regel 28 de standaardwaarde voor `u` toe te voegen en op regel 54 bij toestand `S3` `u` hoog te maken.



Figuur 5.36 : De patroonherkenner die is opgebouwd uit logica en een toestandsregister.

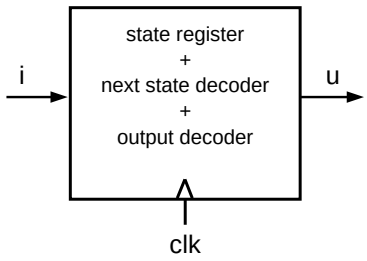
Het grote nadeel van de structuur van dit model is, dat er nu geen wezenlijk verschil tussen een Moore-machine en een Mealy-machine is. In figuur 5.36 kan `u` van het type Moore zijn als deze uitgang alleen van `prs` afhangt, maar kan eveneens van het type Mealy zijn als het ook van ingang `i` afhangt.

```
when S3 =>
  if i = '1' then
    nxt <= S1;
  else
    nxt <= S2;
  end if;
  u <= '1';
```

```
when S3 =>
  if i = '1' then
    nxt <= S1;
    u <= '1';
  else
    nxt <= S2;
  end if;
```

Figuur 5.37 : Signaal `u` als Moore-uitgang en als Mealy-uitgang. Links staat het fragment uit code 5.6 bij toestand `S3`. Signaal `u` hangt niet van `i` af en is een Moore-uitgang. Rechts staat een situatie waar `u` een Mealy-uitgang is, die afhangt van ingang `i`.

Figuur 5.37 laat zien, dat als in code 5.6 de toewijzing aan `u` verplaatst wordt naar het if-statement, uitgang `u` afhankelijk van ingang `i` wordt. De toestandsmachine verandert dan van een Moore-machine in een Mealy-machine. Overigens verandert hiermee tegelijkertijd het functionele gedrag en beschrijft de code niet meer het gedrag van de seriële patroonherkenner.



Figuur 5.38 : De patroonherkenner die is opgebouwd uit één proces.

Een Moore-machine met een proces

In code 5.7 staat een VHDL-beschrijving, die is opgebouwd uit één proces. Figuur 5.38 geeft de opbouw van deze patroonherkenner. Het proces definieert zowel het toestandsregister, de toestandsdecoder en als de uitgangsdecoder.

Net als bij het model met een gecombineerde toestandsdecoder en uitgangsdecoder is het bij dit model niet duidelijk of de uitgangen van het Mealy- of van het Moore-type zijn. Er is nog een groot verschil met de andere beschrijvingen. De uitgangen staan in dit model achter de klokflank en krijgen daarom allemaal een flipflop. De uitgangssignalen zijn geregistreerd en veranderen één klokslag later.

Code 5.7 : Patroonherkenner met één proces.

```
13 architecture gedrag of  pattern_recognizer is
14   type state_type is (S0, S1, S2, S3);
15   signal state : state_type;
16 begin
17   state_machine: process (clk, rst_n) is
18   begin
19     if rst_n = '0' then
20       state <= S0;
21     elsif rising_edge(clk) then
22       u <= '0';
23       case state is
24       when S0 =>
25         if i = '1' then
26           state <= S1;
27         else
28           state <= S0;
29         end if;
30       when S1 =>
31         if i = '0' then
32           state <= S2;
33         else
34           state <= S1;
35         end if;
36       when S2 =>
37         if i = '1' then
38           state <= S3;
39         else
40           state <= S0;
41         end if;
42       when S3 =>
43         if i = '1' then
44           state <= S1;
45         else
46           state <= S2;
47         end if;
48         u <= '1';
49       end case;
50     end if;
51   end process state_machine;
52 end architecture gedrag;
```

Keuze van het model voor de toestandsmachine

De modellen met een aparte uitgangsdecoder, die alleen afhangt van de huidige toestand, hebben de voorkeur. Deze modellen zijn altijd van het Moore-type. De ontwerper kan er niet per ongeluk een Mealy-machine van maken.
Het drie-processenmodel en het twee-processenmodel met een gecombineerde

toestandsregister en toestandsdecoder hebben een aparte uitgangsdecoder. De laatste heeft één signaal voor de toestand en levert een beschrijving die meer beknopt is dan het drie-processenmodel. Het voordeel van het drie-processenmodel is dat er aparte signalen zijn voor de huidige en voor volgende toestand. Dat maakt de code beter leesbaar.

De andere modellen worden niet aanbevolen, omdat er met het één-procesmodel en met het twee-processenmodel met een gecombineerde toestandsdecoder en uitgangsdecoder bewust of onbewust ook Mealy-uitgangen gebruikt worden. Als de uitgangen een register moeten krijgen, is het handiger om aan de uitgangsdecoder een register toe te voegen.

Code 5.8 : Uitgangsdecoder met register.

```
output_decoder: process (clk, rst_n) is
begin
  if rst_n = '0' then
    u <= '0';
  elsif rising_edge(clk) then
    if prs = S3 then
      u <= '1';
    else
      u <= '0';
    end if;
  end if;
end process output_decoder;
```

Code 5.9 : Uitgangsdecoder met een apart register.

```
output_decoder:  u_in <= '1' when prs = S3 else '0';

output_register: process (clk, rst_n) is
begin
  if rst_n = '0' then
    u <= '0';
  elsif rising_edge(clk) then
    u <= u_in;
  end if;
end process output_register;
```

De uitgangsdecoder uit code 5.2 is in code 5.8 herschreven met een if-statement en in een sequentieel proces geplaatst. In code 5.9 is bij de parallelle signaaltoewijzing `u` vervangen door een intern signaal `u_in`. Het proces `output_register` kent dit signaal toe aan uitgang `u`.

5.8 Toestandscodering

Het toestandsregister bewaart de toestand van de toestandsmachine. In dit hoofdstuk zijn tot nu toe vooral abstracte beschrijvingen gebruikt. De toestandsdiagrammen en de VHDL-beschrijvingen met een opsommingstype voor de toestandssignalen bevatten alleen het aantal toestanden en de toestandsnamen. Bij de implementatie moet iedere toestand een unieke code krijgen. Alleen in figuur 5.16 en in figuur 5.18 is voor het opstellen van het schema gekozen om een flipflop te gebruiken, die laag is in toestand s_0, en in figuur 5.20 zijn twee flipfloppen gebruikt. Alle andere voorbeelden doen geen uitspraak over de wijze waarop de toestanden gecodeerd worden.

Bij een binaire codering geldt dat als k het aantal te coderen toestanden en n het aantal bits van het toestandsregister is, dat het aantal mogelijkheden overeenkomt met:

$$\frac{2^n!}{(2^n - k)!}$$

Om de vier toestanden van de seriële patroonherkenner vast te leggen zijn minimaal twee bits nodig. Er zijn verschillende toestandscoderingen mogelijk. In het totaal zijn er bij vier toestanden en bij een twee bits-register 24 mogelijkheden. Vier van deze mogelijkheden staan in tabel 5.2. Het ligt voor de hand om de begintoestand `S0` met `00` te coderen. Bij het opstarten zal de schakeling in deze toestand beginnen. In dat geval blijven er nog zes mogelijkheden over.
Het aantal mogelijkheden groeit explosief bij een groter aantal toestanden. Bij zes toestanden en 3-bits zijn er 20160 mogelijkheden en bij negen toestanden en 4-bits zijn er meer dan 4 miljard mogelijkheden.

Tabel 5.2 : Verschillende toestandscoderingen.

toestand	vier mogelijke coderingen			
S0	00	00	00	11
S1	01	01	10	00
S2	10	11	11	01
S3	11	10	01	10

Tabel 5.3 : One-hot-toestandscodering.

toestand	one hot	one hot with zero
S0	0001	0000
S1	0010	0011
S2	0100	0101
S3	1000	1001

De binaire toestandscode

Het coderen van de toestanden met een zo klein mogelijk toestandsregister noemt men *minimal bits encoding* of binaire codering. Er bestaan verschillende varianten en er worden allerlei namen gebruikt:

- **Binary** of **minimal bits**
  Alle oplossingen uit tabel 5.2 zijn voorbeelden van een binaire codering.
- **Sequential**
  Alle toestanden worden binair genummerd in de volgorde waarin ze gedefinieerd zijn. De eerste mogelijkheid uit tabel 5.2 is sequentieel gecodeerd.
- **Gray**
  De toestandscode van twee naburige toestanden verschilt maximaal één bit. Deze coderingsstrategie reduceert de grootte van de toestandsdecoder.
- **Adjacent**
  De toestandscode van twee naburige toestanden verschilt, zover dat mogelijk is, maximaal één bit. Dit is een variant op gray-code.
- **Johnson**
  De toestandscode van twee naburige toestanden verschilt maximaal één bit. Iedere volgende toestand wordt verkregen door de bits van de huidige toestand één positie op te schuiven.

De Gray-code is genoemd naar de natuurkundige Frank Gray, die bij Bell Labs gewerkt heeft aan de ontwikkeling van de televisie.

Adjacent betekent aanliggend, aangrenzend of naburig.

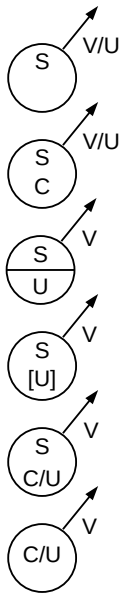
De Johnson counter en de Johnson code zijn genoemd naar Robert Johnson, die ook de uitvinder is van het printen met magnetische inkt.

De Johnson, Gray en de sequentiële toestandscode zijn geschikt voor toestandsdiagrammen met weinig vertakkingen. Een ringvormig toestandsdiagram met acht toestanden is realiseerbaar met een 3-bits binaire teller.
Omdat er bij de Gray- of de adjacency-codering bij een overgang slechts één of een beperkt aantal bits verandert, leiden deze twee coderingen tot een kleine toestandsdecoder.

One-hot codering

Een andere coderingsstrategie is het aantal bits van toestandsregister niet minimaal, maar juist maximaal te maken. Iedere toestand krijgt een eigen flipflop. Deze flipflop is hoog als de toestandsmachine zich in die toestand bevindt, en laag in alle andere toestanden. Deze methode wordt *one hot* genoemd. In tabel 5.3 staat het voorbeeld voor de patroonherkenner met de vier toestanden. Het toestandsregister is 4-bits breed en er is altijd één flipflop hoog, oftewel *hot*.

Het voordeel van one-hot codering is dat er veel minder logica voor de toestandsdecoder en de uitgangsdecoder nodig is dan bij de binaire coderingen. Voor FPGA's, die veel flipfloppen en relatief weinig logica hebben, is de one-hot codering zeer efficiënt. Alleen bij grote toestandsmachines met meer dan 32 toestanden is een binaire codering gunstiger.



Figuur 5.39 : **Symbolen.** Er zijn geen officiële regels voor het tekenen van een toestandsdiagram. In de literatuur worden verschillende notaties gebruikt. De toestandsnaam (`S`), de overgangsvoorwaarde (`V`), de uitgangsspecificatie (`U`) en de toestandscodering (`C`) staan vaak op andere plaatsen. Omdat er zowel voor `V`, `U` en `C` alleen nullen en enen gebruikt worden, kan dit verwarring geven.

Een nadeel van de one-hot codering is dat de begintoestand niet uit enkel nullen bestaat. Daarom wordt er vaak een variant gebruikt, waarbij de minst significante bit geïnverteerd wordt. Tabel 5.3 geeft naaste de gewone one-hot ook deze *one hot with zero*. Bij het opstarten komt de machine bij deze codering automatisch in toestand s_0.

Advies voor toestandscodering

Bij het ontwerp van een digitaal systeem is het verstandig om voor toestandsmachines abstracte beschrijvingen te gebruiken en de keuze voor de toestandscodering aan het syntheseprogramma over te laten. Een toestandsdiagram en een VHDL-beschrijving met een opsommingstype voor de toestanden, leggen allebei het gedrag van de toestandsmachine vast zonder toestandscodes.

De syntheseprogramma's hebben altijd een optie waarmee de toestandscodering bepaald kan worden. Quartus van Altera kent onder andere de optie: `auto`, `Gray`, `sequential` en `one hot`. Bij de optie `auto` bepaalt de synthesizer wat de beste codering is. In het geval van een FPGA-ontwerp zal dat one-hot zijn en dat is bij Quartus altijd de *one hot with zero*.

De opties mogen ook als attribuut in de VHDL-code geplaatst worden. Door na de declaratie van het toestandstype `state_type` de volgende regels op te nemen, gebruikt de synthesizer een Gray-codering voor de toestandscodering:

```
attribute syn_encoding : string;
attribute syn_encoding of state_type : type is "gray";
```

Op een zelfde manier kan elke toestand ook een specifieke toestandscode krijgen:

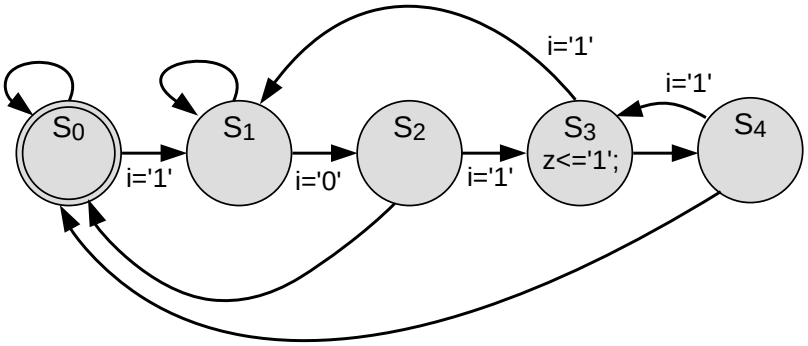
```
attribute syn_encoding : string;
attribute syn_encoding of state_type : type is "11 00 01 10";
```

In het algemeen is er meestal geen reden om een bijzondere codering te gebruiken en kan bij de synthese het beste de standaardinstelling gebruikt worden.

5.9 Toestandsoptimalisatie

Een toestandsdiagram kan ook overbodige of redundante toestanden hebben. Moderne syntheseprogramma's gebruiken verschillende toestandsoptimalisatie-technieken om overbodige toestanden te verwijderen.

Een toestand is overbodig als de volgende toestanden en de uitgangsspecificaties bij dezelfde overgangsvoorwaarden identiek zijn. De seriële patroonherkenner uit paragraaf 5.5 herkent overlappende patronen.
Een andere ontwerper kan redeneren dat in de toestand s_3 weer de eerste `1` van een nieuw patroon is gelezen en dat als de volgende waarde laag is, de machine naar een toestand s_4 gaat waarbij weer de eerste twee waarden van het patroon zijn herkend. Het resultaat is een toestandsdiagram met vijf toestanden, zoals in figuur 5.40 is getekend.
Dit diagram bevat redundante informatie. De volgende toestand is vanuit toestanden s_2 en s_4 in beide gevallen s_3, als `x` hoog is en s_0, als `x` laag is . Bovendien is in beide toestanden de uitgang laag en dus ook identiek.



S_n	x	S_{n+1}	x
S0	1	S1	0
S0	0	S0	0
S1	1	S1	0
S1	0	S2	0
S2	1	S3	0
S2	0	S0	0
S3	1	S1	1
S3	0	S4	1
S4	1	S3	0
S4	0	S0	0

Figuur 5.40 : De seriële herkenner met vijf toestanden. Links staat het toestandsdiagram en rechts de toestandstransitietabel. Bij toestand S_2 en toestand S_4 zijn de volgende toestand en de uitgangswaarden identiek.

Toestand S_4 kan vervallen en worden vervangen door S_2. De pijl van toestand S_3 naar S_4 moet dan naar S_2 gaan wijzen. Het resultaat van deze toestandsoptimalisatie is het toestandsdiagram met de vier toestanden uit figuur 5.28.

5.10 Veilige toestandsmachines

Alleen bij een binaire toestandscodering met een aantal toestanden dat een macht van twee is, worden alle toestanden van de toestandsmachine beschreven in het toestandsdiagram en in de VHDL-beschrijving. Bij een binaire codering met een ander aantal toestanden en bij one-hot worden niet alle toestanden beschreven. Het is in principe mogelijk dat de toestandsmachine zich dan in een ongedefinieerde toestand bevindt.
Sommige ontwerpers voegen daarom bij het case-statement in de toestandsdecoder een **when others** toe die naar de begintoestand verwijst. Bij de meeste synthesizers is dit zinloos, omdat alle voor het functionele gedrag overbodige logica wordt verwijderd.

De meeste synthesizers hebben een optie, die er voor zorgt dat deze extra logica automatisch wordt aangebracht. Gebruik bijvoorbeeld bij Quartus van Altera deze attributen in de VHDL-code:

```
attribute syn_encoding : string;
attribute syn_encoding of state_type : type is "safe";
```

of als er tegelijkertijd ook een Gray-code nodig is:

```
attribute syn_encoding : string;
attribute syn_encoding of state_type : type is "gray, safe";
```

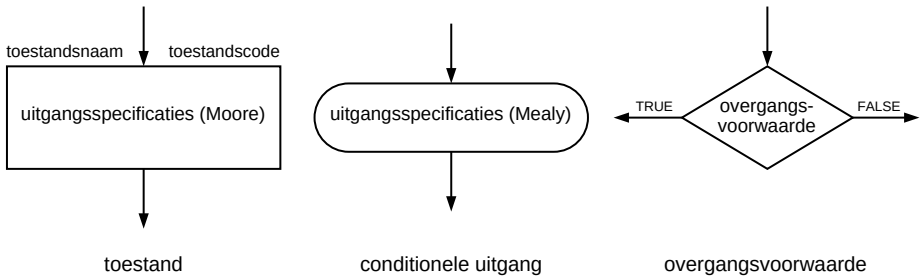
De veilige optie is alleen zinvol bij een binaire code. Bij one-hot levert de veilige optie enorm veel extra logica op.

Een andere methode om een veilige toestandsmachine te krijgen is om er voor te zorgen dat het aantal toestanden exact een macht van twee is door extra toestanden toe te voegen.

Een toestandsmachine zal onder normale condities nooit in een ongedefinieerde toestand komen. De belangrijkste voorwaarden voor een normaal gedrag zijn dat alle signalen synchroon zijn met de klok en dat de clock-skew minimaal is. Als daaraan voldaan is, kan alleen door externe factoren — zoals kosmische straling — de schakeling in een ongedefinieerde toestand komen. De meeste FPGA-ontwerpers gebruiken de one-hot codering en zorgen er voor dat de schakeling synchroon is en dat de klok geen clock-skew bevat. Bij extreme condities kan men er beter voor zorgen dat het aantal toestanden een macht van twee is en tevens een binaire codering gebruiken.

5.11 ASM-chart

Sommige ontwerpers gebruiken in plaats van toestandsdiagrammen ASM-charts. ASM staat voor *algorithmic state machine* en een ASM-chart is een tekening van een ASM. Een toestandsmachine en een ASM komen volledig met elkaar overeen. Een toestandsdiagram is altijd om te zetten naar een ASM-chart en omgekeerd. Een ASM-chart is bij grote toestandsmachines soms overzichtelijker dan een toestandsdiagram.
Een ASM-chart kent drie symbolen voor: de toestand, de overgangsvoorwaarde en de conditionele uitgang. Figuur 5.41 toont deze symbolen.

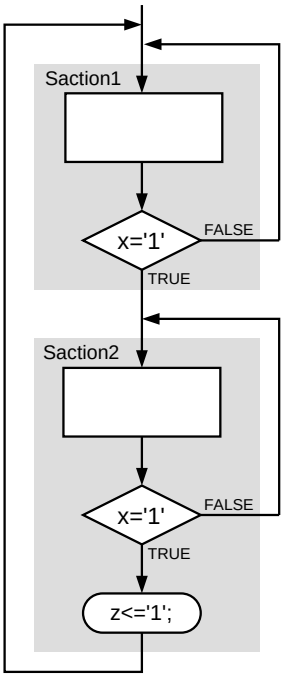


Figuur 5.41 : De symbolen van een ASM-chart.

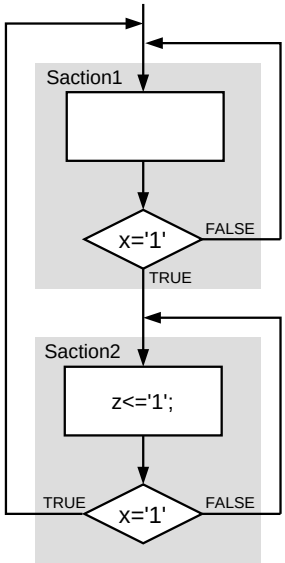
Het toestandssymbool heeft een naam, eventueel een toestandscodering en de uitgangsspecificaties. Bij deze laatsten gaat het altijd om Moore-uitgangen, omdat deze specificatie bij de toestand staat. Bij de conditionele uitgang gaat het juist altijd om specificaties voor Mealy-uitgangen. Deze uitgangsspecificaties horen bij een toestandsovergang.

In figuur 5.42 staat de ASM-chart van het Mealy-diagram uit figuur 5.16 en in figuur 5.43 staat de ASM-chart van het Moore-diagram uit figuur 5.18.
De grijze vlakken in figuur 5.42 en figuur 5.43 worden ASM-blokken of *ASM blocks* genoemd. Ieder blok bevat één toestand met alle symbolen voor conditionele uitgangen en overgangsvoorwaarden, die bij de uitgaande overgangen horen. Elk ASM-blok heeft slechts één ingang, maar mag meerdere uitgangen hebben.

De verschillen tussen een ASM-chart en een toestandsdiagram zijn niet groot. Een ASM-chart is wat formeler en meer geschikt voor complexe machines. Daarentegen past een toestandsdiagram beter bij een creatieve ontwerpstrategie. Het tekenen van bolletjes en pijlen is met een pen-en-papier-ontwerpmethode zeer natuurlijk.



Figuur 5.42 : De ASM-chart van een Mealy-machine.

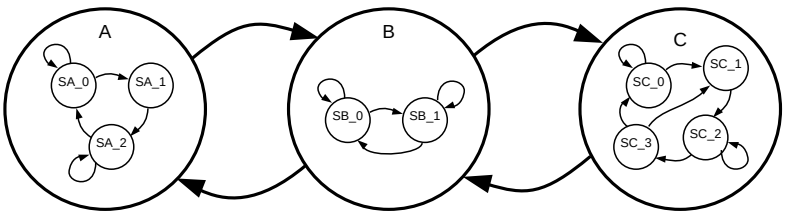


Figuur 5.43 : De ASM-chart van een Moore-machine.

Een ASM-chart heeft veel overeenkomsten met een stroomdiagram of *flowchart*. Het grote verschil is dat bij een ASM-chart de overgangen tussen de toestanden — net als bij een toestandsdiagram — klokgestuurd zijn. Tussen twee opeenvolgende ASM-blokken zit minimaal één klokslag. Een stroomdiagram legt alleen de volgorde van de verschillende bewerkingen vast.

5.12 Hiërarchie en gekoppelde toestandsmachines

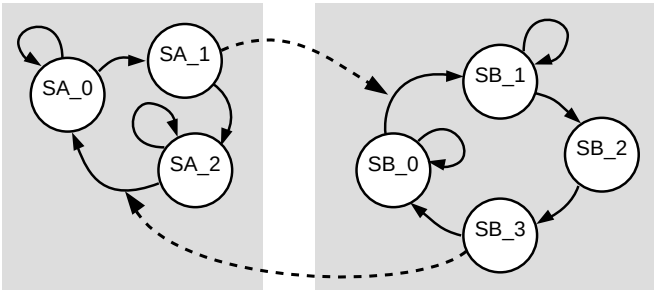
Met de meeste tekenpakketten voor toestandsdiagrammen kunnen ook hiërarchische en gekoppelde toestandsmachines gemaakt worden. Figuur 5.44 geeft een schets van een hiërarchische toestandsmachine. De bovenste laag van de machine bestaat uit drie toestanden en bij iedere toestand hoort een aparte toestandsmachine.



Figuur 5.44 : Een schets van een hiërarchische toestandsmachine.

In figuur 5.45 staan twee gekoppelde toestandsmachines. Toestandsmachine `B` wacht in toestand `SB_0` op een signaal van toestandsmachine `A`. Als toestandsmachine `A` in toestand `SA_1` is, wordt het signaal actief en wordt toestandsmachine `B` doorlopen. Toestandsmachine `A` gaat tegelijkertijd naar toestand `SA_2` en wacht op

een signaal van toestandsmachine B. Dit signaal wordt actief, als toestandsmachine B in toestand SB_3 is.



Figuur 5.45 : Twee gekoppelde toestandsmachines.

Een toestandsmachine kan ook worden gesplitst in meerdere gekoppelde toestandsmachines. In dit boek wordt hier nauwelijks gebruik van gemaakt. De ontwerpstrategie gaat uit van de databewerkingen. De hiërarchische blokken die daar uit ontstaan bevatten een dataverwerkingsdeel en een toestandsmachine. De toestandsmachines van de verschillende hiërarchische blokken zijn in feite gekoppelde toestandsmachines.

5.13 Resumé

Een toestandsmachine is het gedeelte van een digitaal systeem dat het dataverwerkingsdeel bestuurt en kan worden gerepresenteerd door een toestandsdiagram. Omdat een Moore-machine synchroon is, heeft deze de voorkeur boven een Mealy-machine die asynchroon is.

Voor het opstellen van een toestandsdiagram, kan de ontwerper het best alle scenario's en daarna voor ieder scenario de toestanden tekenen. Hij kan het toestandsdiagram met een speciaal ontwerpprogramma tekenen en het getekende diagram automatisch omzetten naar een VHDL-beschrijving. Een alternatief is om het toestandsdiagram met pen en papier te tekenen en het daarna direct in VHDL of een andere hardwarebeschrijvingstaal op te schrijven.

In VHDL of in een andere hardwarebeschrijvingstaal heeft het drie-processenmodel of het twee-processenmodel met aparte uitgangsdecoder de voorkeur. Kies bij FPGA's altijd voor een one-hot codering of voor de *one hot with zero*. Nog beter is het om de keuze van de toestandscodering aan de synthesizer over te laten. Houd het ontwerp daarom zo abstract mogelijk, bijvoorbeeld door de toestanden als een opsommingstype te definiëren.

Het toevoegen van extra code om onveilige situaties af te vangen, is zinloos omdat de synthesizer dit bij de synthese verwijderd. Een veilige toestandsmachine kan het beste worden gemaakt door er voor te zorgen dat het aantal toestanden een macht van twee is en door bovendien een binaire codering te gebruiken.

6

Ontwerpmethoden

Doelstelling

In dit hoofdstuk maak je kennis met een aantal ontwerpmethodieken en leer je werken met behulp van de methode met een gescheiden dataverwerking en besturing.

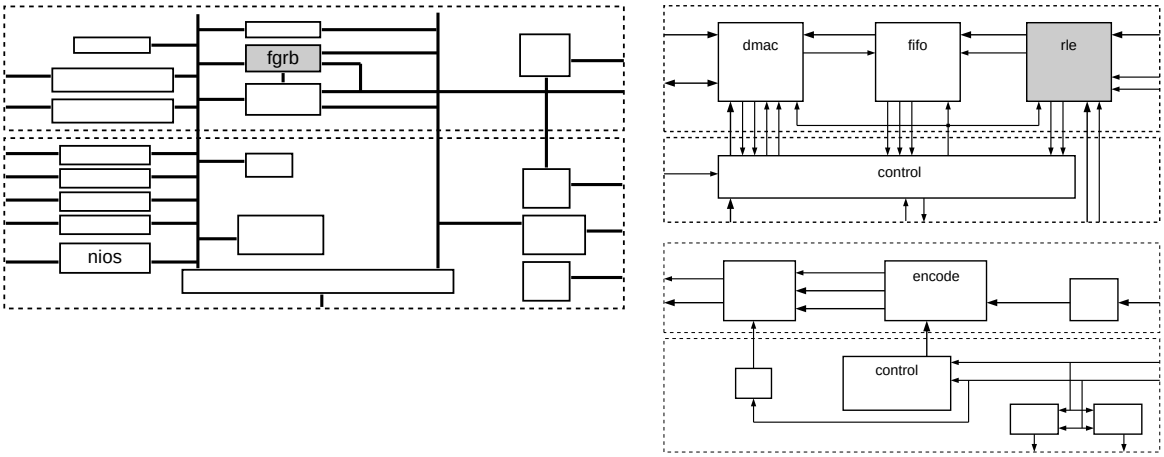
Onderwerpen

De behandelde onderwerpen zijn:

- De architectuur van een complex digitaal systeem.
- De hiërarchie in een complex digitaal systeem.
- De opdeling van een digitaal systeem in een control- of besturingsdeel en een datapad of dataverwerkingsdeel.
- De stuursignalen en statussignalen.
- Twee methoden voor het bepalen van de frequentie: de periodetijdmeting en de frequentiemeting.
- Het ontwerpen met de iteratieve, softwarematige aanpak.
- Het ontwerpen met behulp van de methode met een gescheiden dataverwerking en besturing.
- Het tekenen van het dataverwerkingsgedeelte van een digitaal systeem.
- Het omzetten van een ontwerp met gescheiden dataverwerking en besturing naar een VHDL-beschrijving.
- Het ontwerp met de FSMD-methode.
- Het type `boolean`.
- Toestandsdiagrammen met datapad-aspecten (FSMD).
- Een ASM-chart met datapad-aspecten (ASMD).
- De twee-processenmethode.

Grote complexe digitale systemen bestaan uit veel verschillende onderdelen, die onderling met elkaar communiceren. Deze onderdelen of subsystemen bestaan zelf weer uit een aantal deelsystemen, die allerlei gegevens aan elkaar doorgeven. Ieder deelsysteem kan op zichzelf weer een compleet digitaal systeem zijn.
Kenmerkend voor een complex digitaal systeem is de hiërarchische opbouw en het parallellisme van al deze deelsystemen. Een digitaal ontwerp heeft een architectuur, die met behulp van blokschema's, stroomdiagrammen en andere grafische tekeningen wordt vastgelegd.
In figuur 6.1 staat een voorbeeld van een complex digitaal systeem. Het hoogste niveau kent achttien deelsystemen. Een van deze onderdelen, het blok `fgrb`, is verder uitgewerkt en bestaat weer uit vier blokken. Daarvan bevat het blok `rle` op zijn beurt weer zeven onderdelen.

Blokschema's, zoals die in figuur 6.1 zijn getekend, zijn essentieel om het overzicht over het complexe systeem te houden. Een dergelijk groot complex systeem wordt ontworpen door een team van ontwerpers, die allemaal verantwoordelijk zijn voor één of meer subsystemen.



Figuur 6.1 : **Een hiërarchisch ontwerp van een complex digitaal systeem.** Links staat het blokschema van het hoogste niveau met het complete systeem. Het blokschema van het onderdeel `fgrb` staat rechtsboven en rechtsonder is het onderdeel `rle` verder uitgewerkt.

Op ieder niveau van het ontwerp is een dataverwerkingsdeel en een besturingsgedeelte te onderscheiden. In de blokschema's van figuur 6.1 bevindt het dataverwerkingsdeel zich steeds bovenaan en de besturing onderaan de tekeningen. Ook de embedded NIOS-processor, linksonder in het blokschema van het hoogste niveau, bevat een dataverwerkingsdeel met een rekeneenheid en dataregisters en een besturingsdeel dat de programmacode vertaalt naar functionele bewerkingen.
Om een complex digitaal systeem te maken, zijn een vaste ontwerpmethodiek, een zorgvuldige documentatie, een systematische aanpak en eenduidige blokschema's essentieel.

In hoofdstuk 2 tot en met hoofdstuk 4 zijn relatief eenvoudige, combinatorische en sequentiële bouwblokken besproken. Dit hoofdstuk behandelt verschillende ontwerpmethoden voor een middelgroot digitaal systeem. Dat is een deelontwerp op het laagste niveau van figuur 6.1. Deze systemen zijn opgebouwd uit zowel combinatorische als sequentiële processen, waarvan de functionaliteit met een paar tekeningen kan worden vastgelegd en waarvan de hoeveelheid VHDL-code niet groter is dan duizend regels.
Niet alles hoeft vanaf niets te worden opgebouwd. Softcores en megafuncties geven vaak complete oplossingen voor deelproblemen. In figuur 6.1 representeert bijvoorbeeld het blok NIOS een complete NIOS-processor.
Dit hoofdstuk richt zich op methodieken om een compleet, nieuw systeem te maken. De besproken methodieken zijn:

- de iteratieve softwarematige aanpak,
- de methode met gescheiden dataverwerking en besturing,
- de FSMD-methode,
- de ASMD-methode,
- de twee-processenmethode.

6.1 Ontwerpvoorbeeld : de elektronische personenweegschaal

Dit hoofdstuk gebruikt één specifiek voorbeeld om al de verschillende ontwerpmethodieken te bespreken.

Het voorbeeld van de elektronische personenweegschaal is ontleend aan *Ontwerpen van digitale MOS-IC's* van J. van Dijken et. al., dat in 1994 is uitgegeven door Nijgh & Van Ditmar.

De specificatie: de eisen aan de elektronische personenweegschaal

Een elektronische personenweegschaal heeft een slimme druksensor, die een pulstrein afgeeft waarvan de frequentie f evenredig is met het gewicht G dat de sensor detecteert:

$$f = k\,G \tag{6.1}$$

Ieder ontwerp begint met een studie naar de mogelijkheden. In dit voorbeeld is dat de meetmethode, maar in een andere situatie is dat een studie van de bestaande toestand of een studie naar een nieuw te gebruiken ontwerptechniek. Hier is de term analyse gebruikt, anderen spreken liever van een onderzoek of vooronderzoek.

De constante k is een evenredigheidsconstante en is in dit voorbeeld 100 Hz/kg. Het bereik van de weegschaal ligt tussen 10 en 150 kg. Het gewicht wordt met een nauwkeurigheid van 0,1 kg op een vier-cijferig digitaal display afgebeeld.
De aanduiding op het display moet minimaal iedere halve seconde ververst kunnen worden. De sensor reageert binnen 10 ms op een gewichtsverandering.

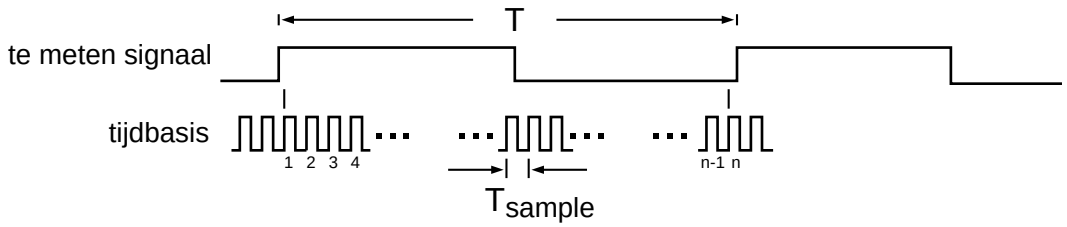
De analyse: het bepalen van een periodetijd of een frequentie

Het bepalen van de periodetijd of de frequentie van een periodiek signaal is in wezen hetzelfde. Als de frequentie f bekend is, is de periodetijd T ook bekend en omgekeerd. Er geldt:

$$f = \frac{1}{T} \tag{6.2}$$

In een digitaal systeem wordt tijd altijd gerelateerd aan het kloksignaal of aan een van de klok afgeleid signaal. Voor het bepalen van de periodetijd en het meten van de frequentie is altijd een teller nodig. Er zijn twee principieel verschillende meetmethoden:

- **Periodetijdmeting**
  Bij deze meting wordt gedurende een periode van het periodieke signaal het aantal klokslagen n geteld.



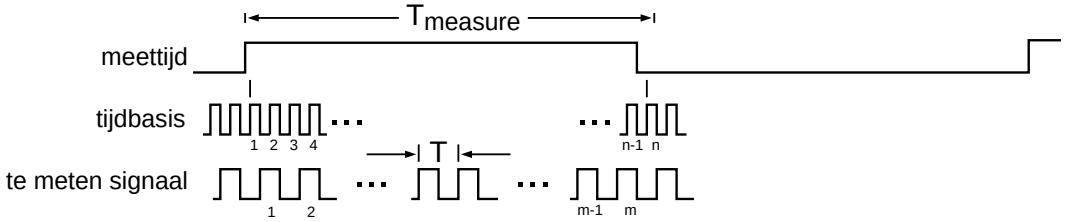
Figuur 6.2 : **De periodetijdmeting.** Het aantal pulsen van de tijdbasis is evenredig met de periodetijd van het te meten signaal.

Als T_{sample} de periodetijd van de systeemklok is of een daarvan afgeleide klok en de periodetijd van het te meten signaal T is, geldt:

$$T = n\,T_{\text{sample}} \tag{6.3}$$

De periodetijd van het te meten signaal is evenredig met het aantal gemeten *samples* of klokslagen

- **Frequentiemeting**
  Bij deze meting wordt gedurende een vooraf vastgestelde meettijd het aantal perioden van het te meten signaal geteld.



Figuur 6.3 : **De frequentiemeting.** De frequentie van het te meten signaal is evenredig met het aantal samples.

Figuur 6.3 geeft een grafische weergave van deze meetmethode. Er zijn twee tellers nodig. Eén teller definieert de meettijd T_{measure}, door bijvoorbeeld n klokslagen te tellen. De andere teller telt het aantal perioden m van het te meten signaal.

$$mT = T_{\text{measure}} \qquad \text{of} \qquad f = m f_{\text{measure}} \tag{6.4}$$

De frequentie van het te meten signaal is evenredig met het aantal gemeten *samples* of klokslagen m.

De periodetijdmeting is snel; de periodetijd is bekend na één periode van het te meten signaal. Een nadeel is dat bij een hoge nauwkeurigheid de samplefrequentie hoog moet zijn. Voor snelle signalen is deze meting minder geschikt.
De frequentiemeting is geschikt voor signalen met een hoge frequentie. In een korte tijd kunnen er dan heel veel perioden geteld worden. Voor langzame signalen — en dat geldt zeker bij nauwkeurige metingen — wordt de meettijd te lang.
Voor signalen, die niet snel en die niet langzaam zijn, kan een variant op de periodetijdmeting worden toegepast. In plaats van over één periode kan er over meerdere perioden worden geteld. De gezochte periodetijd is dan het gemiddelde over de verschillende perioden.

Voor de periodetijdmeting volgt uit formule 6.1, 6.2 en 6.3 dat het gewicht G omgekeerd evenredig is met het aantal getelde klokslagen n:

$$G = \frac{1}{n} \frac{f_{\text{sample}}}{k} \tag{6.5}$$

Naast het gegeven dat de samplefrequentie hoog moet zijn om de gewenste nauwkeurigheid te halen, is er voor de berekening een deling nodig om het gewicht te bepalen. De deling is niet standaard synthetiseerbaar en als deze wel gesynthetiseerd kan worden, is daar veel logica voor nodig.

Bij de frequentiemeting is het gewicht evenredig met het aantal getelde klokslagen. Uit formule 6.1 en 6.4 volgt:

$$G = m \frac{f_{\text{measure}}}{k} \tag{6.6}$$

Bij deze methode is het gewicht evenredig met het aantal getelde klokslagen. Bovendien kan het ontwerp verder vereenvoudigd worden door f_{measure} slim te kiezen. De meettijd kan zo gekozen worden dat het gemeten aantal klokslagen overeenkomt met het gewicht in tiende kilogrammen. Voor die situatie is het gewicht G een tiende van het gemeten aantal klokslagen m. Voor een k van 100 Hz/kg moet f_{measure} gelijk zijn aan 10 Hz. De meettijd T_{measure} is dan gelijk aan 0,1 s.

Het aantal perioden van het ingangssignaal dat in 0,1 s past, geeft het gewicht in tiende kilogrammen. Deze meettijd voldoet aan de eis dat het display minimaal iedere halve seconde ververst moet worden.

6.2 De iteratieve softwarematige aanpak

Deze paragraaf geeft een gedragsbeschrijving voor de elektronische personenweegschaal met behulp van een iteratieve softwarematige aanpak. Deze aanpak beschrijft het gedrag eerst in gewone, Nederlandstalige zinnen. Stap voor stap wordt de beschrijving steeds verder verfijnd en tenslotte leidt dit tot een VHDL-beschrijving van de weegschaal.
De verwerkingseenheid van de elektronische personenweegschaal doet voortdurend twee dingen: het telt het aantal pulsen gedurende de meettijd en zet het resultaat daarna op het display. Een eerste aanzet kan dus zijn:

```
Tel het aantal pulsen gedurende de meettijd van 0,1 seconde
Zet het resultaat op het display
```

De hier gebruikte aanpak heeft als voordeel dat het bedenken van de oplossing vanuit het Nederlands gedaan wordt.

Het achterliggende idee van deze aanpak is dat als je het probleem niet in gewoon Nederlands kunt uitleggen, je het dan zeker niet in een taal als C, Java of VHDL kunt uitleggen.

De Nederlandse teksten zijn schuingedrukt en staan tussen dubbele aanhalingstekens.

De beslissing of er geteld wordt of dat het resultaat op het display gezet wordt, kan gerealiseerd worden met een if-statement:

```
if "meettijd < 100 ms" then
  "tel het aantal pulsen"
else
  "zet resultaat op het display"
  "begin opnieuw met meten"
end if;
```

Een digitaal systeem voor een FPGA zal altijd synchroon zijn. Dit gedrag kan met een proces met een wachtopdracht worden gerepresenteerd. Verder wordt het aantal pulsen alleen verhoogd als er een nieuwe puls is en wordt dit aantal weer nul gemaakt voor er een nieuwe meting uitgevoerd wordt.

```
aanzet_3 : process is
begin
  if "meettijd < 100 ms" then
    if "nieuwe puls" then
      aantalpulsen <= aantalpulsen + 1;
    end if;
  else
    "zet resultaat op het display"
    aantalpulsen <= 0;
    "begin opnieuw met meten"
  end if;
  wait for sample_time;
end process aanzet_3;
```

Het wait-for-statement is — net als andere toewijzingen met een tijdsaspect — niet synthetiseerbaar. Een digitaal systeem heeft altijd een systeemklok nodig voor de definitie van tijd. Het meten van een bepaalde tijdsduur komt altijd overeen met het tellen van een aantal klokslagen. Voor een systeemklok van 100 kHz is een meettijd van 100 ms gelijk aan 10000 klokslagen. In het ontwerp zijn dus twee tellers nodig: één voor het aantal pulsen en één voor het aantal klokslagen. Nadat het resultaat op het display is gezet, worden beide tellers nul gemaakt.
In code 6.1 staat de vierde aanzet voor de gedragsbeschrijving van de elektronische personenweegschaal. Twee aspecten zijn nog niet goed beschreven: het detecteren van een nieuwe puls en het op het display zetten van het resultaat.

De pulsdetectie kan worden beschreven met het attribuut `'last_value`. Dit signaalattribuut geeft de vorige waarde van een signaal. Als de laatste waarde laag en de huidige waarde hoog is, is er een nieuwe puls. De vijfde aanzet uit code 6.2

Code 6.1 : Vierde aanzet voor gedragsbeschrijving van elektronische personenweegschaal.

```
architecture gedrag of epw is
  signal cc           : integer;
  signal aantalpulsen : integer;
begin
  aanzet_4 : process (clk) is
  begin
    if rising_edge(clk) then
      if cc < 10000 then
        if "nieuwe puls" then
          aantalpulsen <= aantalpulsen + 1;
        end if;
        cc <= cc + 1;
      else
        "zet resultaat op het display"
        aantalpulsen <= 0;
        cc <= 0;
      end if;
    end if;
  end process aanzet_4;
end architecture gedrag;
```

bevat een procedure `send_to_display`, die het aantal getelde pulsen op de juiste manier op een 4-cijferig digitaal display afbeeldt. Ook is aan het proces een actief lage asynchrone reset toegevoegd.

Code 6.2 : Vijfde aanzet voor gedragsbeschrijving van elektronische personenweegschaal.

```
 1  architecture gedrag of epw is
 2    signal cc           : integer;
 3    signal aantalpulsen : integer;
 4  begin
 5    aanzet_5 : process (clk, rst_n) is
 6    begin
 7      if rst_n = '0' then
 8          aantalpulsen <= 0;
 9          cc <= 0;
10      elsif rising_edge(clk) then
11        if cc < 10000 then
12          if (puls'last_value = '0') and (puls = '1') then
13            aantalpulsen <= aantalpulsen + 1;
14          end if;
15          cc <= cc + 1;
16        else
17          send_to_display(aantalpulsen);
18          aantalpulsen <= 0;
19          cc <= 0;
20        end if;
21      end if;
22    end process aanzet_5;
23  end architecture gedrag;
```

In code 6.2 heeft de procedure `send_to_display` geen uitgangen. Normaal gesproken zouden dat de aansluitingen van het display, dat het gewicht afbeeldt, moeten zijn. In dit geval schrijft de procedure `send_to_display` bij het simuleren het gewicht als tekst naar het scherm. Deze vijfde aanzet is ook om deze reden niet synthetiseerbaar.

In paragraaf 10.11 wordt het textio-package waarmee tekst naar het scherm geschreven kan worden besproken.

Mits de procedure `send_to_display` uit code 6.2 het aantal pulsen op een juiste wijze converteert naar vier cijfers, beschrijft deze code het gedrag van de elektronische personenweegschaal. Helaas bevat deze code twee lastige problemen. Het attri-

buut `last_value` is niet synthetiseerbaar en het afbeelden van een binair getal als BCD-gecodeerd getal is niet triviaal.

Voor de detectie van een nieuwe puls moet de vorige waarde bewaard worden, daar is een extra intern signaal voor nodig. In code 6.3 krijgt bij iedere klokslag het signaal `vorige_puls` de waarde van `puls`. Het signaal `vorige_puls` bevat dus altijd de vorige waarde van `puls`. Hiermee is een nieuw puls eenvoudig te detecteren.
De pulsen worden BCD-gecodeerd geteld. De functie `incrementBCD` verhoogt het aantal pulsen met één en de functie `to_display` zet de vier BCD-cijfers om naar een 28-bits signaal `gewicht` dat de vier 7-segmentsdisplays aanstuurt.

Code 6.3 : Zesde aanzet voor gedragsbeschrijving van elektronische personenweegschaal.

```
12    aanzet_6 : process (clk, rst_n) is
13    begin
14      if rst_n = '0' then
15          aantalpulsen <= (others => '0');
16          cc <= 0;
17      elsif rising_edge(clk) then
18        if cc < 10000 then
19          if (vorige_puls  = '0') and (puls = '1') then
20            aantalpulsen <= incrementBCD(aantalpulsen);
21          end if;
22          cc <= cc + 1;
23        else
24          gewicht <= to_display(aantalpulsen);
25          aantalpulsen <= (others => '0');
26          cc <= 0;
27        end if;
28        vorige_puls <= puls;
29      end if;
30    end process aanzet_6;
31
32    dig_dec  <= gewicht(6 downto 0);
33    dig_unit <= gewicht(13 downto 7);
34    dig_ten  <= gewicht(20 downto 14);
35    dig_hund <= gewicht(27 downto 21);
```

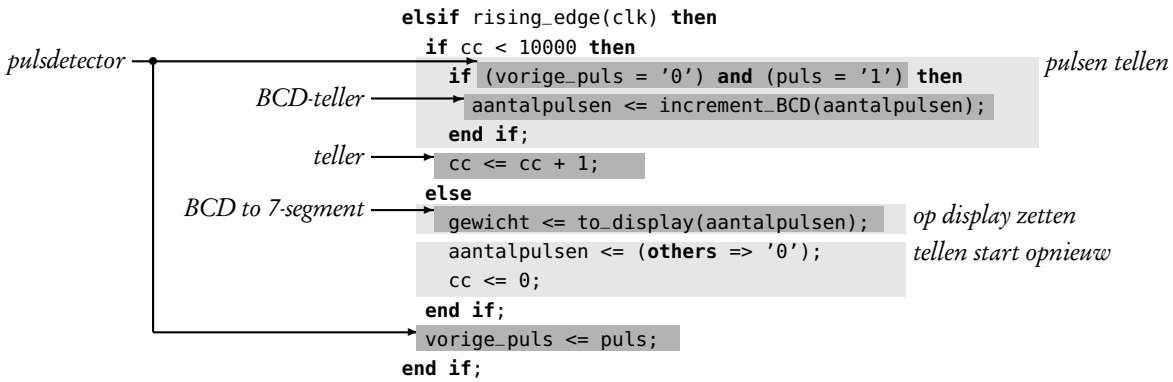
Er is gekozen om het aantal pulsen BCD-gecodeerd te tellen, omdat het omzetten van een binair getal naar een BCD-gecodeerd getal lastig is. Deze conversie kan op twee manieren worden opgelost: parallel en sequentieel. Bij de parallelle methode is er veel logica nodig en bij de sequentiële oplossing zijn er meerdere klokslagen nodig voor de conversie. Bovendien is het niet eenvoudig om de sequentiële variant in het proces te implementeren: functies mogen bijvoorbeeld in VHDL geen wachtopdrachten bevatten. Fundamenteel hierbij is dat het proces iedere klokslag nieuwe waarden voor `vorige_puls` en `cc` moet bepalen en dat er voor de conversie meer klokslagen nodig zijn.

De keuze voor BCD-gecodeerd tellen is bij het ontwikkelen van hardware een logische keuze. Een BCD-teller is een standaard component die deel uitmaakt van iedere logische familie. Een gedragsbeschrijving van een BCD-teller is niet eenvoudig, maar BCD-tellers zijn binnen veel ontwikkelomgevingen wel beschikbaar als standaard bouwsteen. De hier gebruikte functie `incrementBCD` komt in paragraaf 8.3 aan de orde en de BCD-teller in paragraaf 8.4.

Bespreking softwarematige aanpak

Een volgende, logische stap in het ontwerpproces zou een nieuwe opzet kunnen zijn, die gebruik maakt van een aparte BCD-teller, pulsdetector en displayeenheid. Dit voortschrijdend inzicht ontstaat bijna altijd als geprobeerd wordt een gedragsbeschrijving met één proces te maken. Omdat het ontwerp uiteindelijk als hardware gerealiseerd wordt, is het verstandig het ontwerp direct in functionele blokken in te delen.
Als er bij de zesde aanzet gekozen was om binair te tellen en het resultaat te converteren naar een BCD-gecodeerd signaal, kan de beschrijving zeer ingewikkeld worden. Het is bijna ondoenlijk om de seriële BCD-conversie in deze oplossing in te bouwen. Dat is een fundamenteel probleem. Een digitaal systeem bevat altijd aspecten die te maken hebben met de verwerking van gegevens en aspecten die deze verwerking besturen. De verwerking van gegevens bevat typisch parallellisme. De tellers voor het tellen van het aantal klokslagen en het aantal pulsen voeren hun taken gelijktijdig uit. De besturing is daarentegen juist sequentieel. Na het tellen van de pulsen wordt het resultaat op het display gezet en start het tellen opnieuw. In veel gevallen is de besturing niet meer dan een toestandsmachine.

De dataverwerking van de elektronische personenweegschaal bestaat uit een pulsdetector, een BCD-teller voor het aantal pulsen en een omzetting van de BCD-waarde naar de vier 7-segmentsdisplays. De besturing bestaat uit drie acties: het tellen van de pulsen, het op het display zetten van het resultaat en het opnieuw starten van de meting. In code 6.3 staan deze twee aspecten — namelijk de dataverwerking en de besturing — allemaal door elkaar. In figuur 6.4 is dit gevisualiseerd. Onderdelen van de dataverwerking zijn over de hele code verdeeld en vaak in verschillende stukken geknipt. De pulsdetector bestaat uit twee stukken: het testen op de flank staat bij het if-statement en het toekennen van signaal `puls` aan `volgende_puls` staat aan het einde van de code.



Figuur 6.4 : Dataverwerking en besturing bij de softwarematige aanpak. De verschillende aspecten staan door elkaar en overlappen elkaar. De dataverwerkingsdelen hebben een donker grijze achtergrond. De besturingsdelen hebben een licht grijze achtergrond.

Voor beginnende VHDL-ontwerpers is het moeilijk deze twee aspecten te onderscheiden. Zeker in combinatie met de parallelle en sequentiële constructies van de taal levert dat vaak een onleesbare, moeilijk te begrijpen code op. Ervaren VHDL-ontwerpers kunnen hier vaak wel goed mee omgaan en maken wel leesbare beschrijvingen met grote complexe processen.

Het voordeel van grote geïntegreerde blokken is dat de simulatietijd kort kan zijn. Een beschrijving, die uit veel kleine processen bestaat, simuleert langzamer. Er zijn dan veel interne signalen. Voor ieder signaal heeft de simulator een databasestructuur nodig waarin allerlei attributen worden bijgehouden. Voor een beschrijving met een klein aantal grote processen zijn minder interne signalen nodig. De interne databasestructuur is dan veel eenvoudiger en de simulatie zal sneller zijn.

6.3 De methode met gescheiden dataverwerking en besturing

Het voorbeeld met de elektronische personenweegschaal laat zien dat een digitaal systeem bestaat uit onderdelen, die typisch te maken hebben met dataverwerking en uit onderdelen bestaan die vooral te maken hebben met besturing.

Het dataverwerkingsdeel wordt ook *data path* of datapad genoemd.

De dataverwerking bestaat uit verschillende blokken die tegelijkertijd naast elkaar hun taak uitvoeren. De personenweegschaal bevat een pulsdetector, een BCD-teller en een omzetter voor het gemeten resultaat. Deze blokken bestaan naast elkaar en voeren hun taken parallel uit.

In dit boek wordt meestal dataverwerking of dataverwerkingsdeel gebruikt omdat de uitspraak en de betekenis van *path* en pad verwarring geeft.
Het Engelse woord *path* betekent in het Nederlands pad in betekenis van route of traject. Het Engelse woord *pad* betekent kussentje.

De besturing zorgt ervoor dat de gegevens op een correcte manier door het dataverwerkingsdeel worden geleid. De besturing is meestal een toestandsmachine die op het juiste moment de stuursignalen van de dataverwerking hoog maakt. Het besturingsdeel kenmerkt zich door het sequentiële karakter van de toestanden die het digitale systeem moet doorlopen.

Omdat dataverwerking vooral parallelle kenmerken heeft en het besturingsdeel vooral sequentieel is, ligt het voor de hand deze twee te scheiden. Deze paragraaf behandelt de methode met een gescheiden dataverwerking en besturing. Bij deze methode wordt eerst het dataverwerkingsdeel getekend en daarna het bijbehorende toestandsdiagram. Bij het tekenen van het dataverwerkingsdeel worden alle tijdsaspecten buiten beschouwing gelaten. In de tekening van de dataverwerking staan alleen de registers waarmee de verschillende gegevens bewaard worden en de bewerkingen, die met deze gegevens worden uitgevoerd. Naderhand worden de verschillende scenario's dit het systeem kan doorlopen opgesteld en wordt dit verder uitgewerkt in een toestandsdiagram.

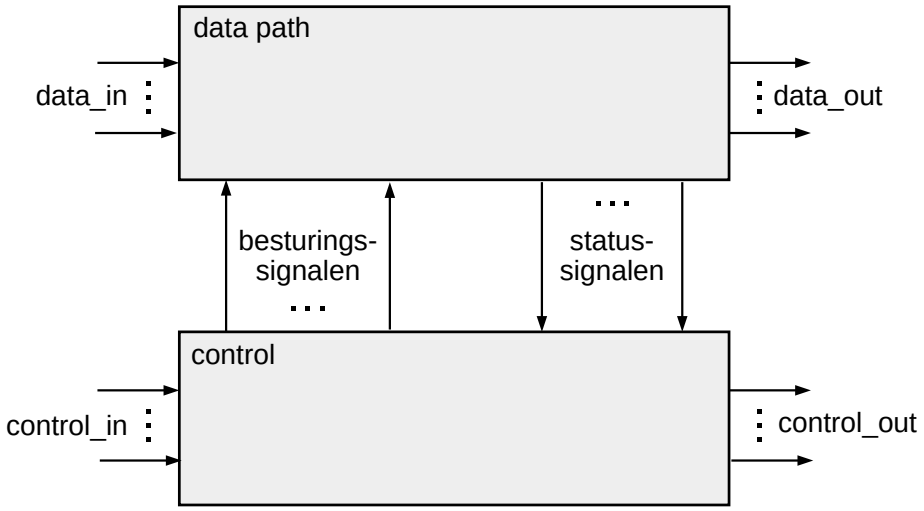
Het Engelse woord voor besturing is *control*. Het Nederlandse woord controle is afgeleid van controleren, dat naast beheersen ook nagaan, nazien of checken kan betekenen.
In dit boek wordt meestal besturing gebruikt, omdat dit in het Nederlands meer eenduidig is.

Model voor gescheiden dataverwerking en besturing

In figuur 6.5 staat een blokschema van een digitaal systeem dat opgebouwd is uit een dataverwerkingsdeel en een besturingsdeel. Aan de linkerzijde staan de ingangssignalen van het systeem. Een deel van deze signalen bevat de gegevens die door het dataverwerkingsdeel bewerkt worden, het andere deel stuurt het besturingsdeel van het systeem aan. Aan de rechterzijde staan de uitgangssignalen van het systeem. Een deel van de uitgangssignalen bevat het resultaat van de dataverwerking. Het andere deel van de uitgangssignalen geeft informatie over de toestand van het systeem.

De methode met gescheiden dataverwerking en besturing wordt ook de *data path/control-* of *datapad/control*-methode genoemd.

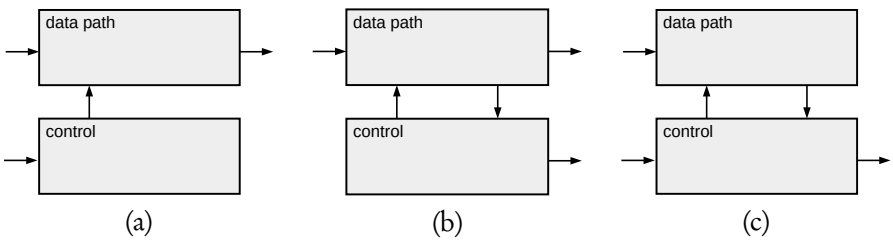
De besturingssignalen of stuursignalen van het besturingsdeel naar het dataverwerkingsdeel zorgen dat de enable-signalen van registers en tellers en de selectiesignalen van multiplexers op het juiste moment actief zijn. Informatie over de



Figuur 6.5 : Een digitaal systeem is opgebouwd uit een dataverwerkingsdeel of *data path* en een besturingsdeel of *control*.
Naast de ingang- en uitgangssignalen zijn er besturingssignalen van het besturingsdeel naar het dataverwerkingsdeel en kunnen er statussignalen zijn van het dataverwerkingsdeel naar het besturingsdeel.

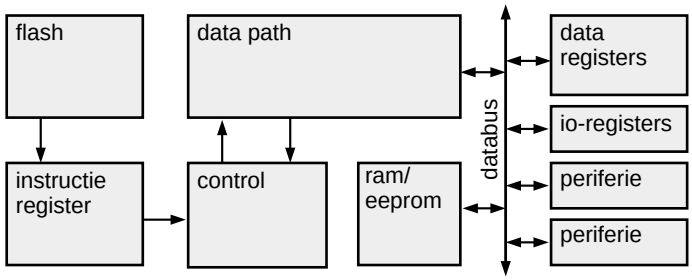
status van de bewerkingen wordt via de statussignalen aan het besturingsdeel doorgegeven. Voorbeelden zijn de overflow bij de optelfunctie en de overflow van een teller.

Niet alle signalen hoeven in ieder digitaal systeem aanwezig te zijn. Figuur 6.5 is een algemeen model die in veel varianten voorkomt. In figuur 6.6.a staat een voorbeeld van een systeem zonder statussignalen en met een besturing zonder uitgangssignalen. Figuur 6.6.b is een systeem met een autonome besturing. Het besturingsdeel heeft geen externe ingangssignalen Het systeem uit figuur 6.6.c heeft geen uitgangssignalen bij de dataverwerking.



Figuur 6.6 : Drie varianten op het digitale systeem met gescheiden dataverwerking en besturing.

Een microprocessor en een microcontroller voldoen ook aan het model met een gescheiden dataverwerking en besturing. In figuur 6.7 staat een vereenvoudigd schema van een microcontroller. Het dataverwerkingsdeel van de microcontroller bevat onder andere de ALU of rekeneenheid en het besturingsdeel krijgt de opdrachten uit het instructieregister. De dataverwerking is bij een microcontroller aangesloten op de databus, die ook is verbonden met de dataregisters, het datageheugen, de io-registers en allerlei perifere blokken, zoals een UART en ADC.



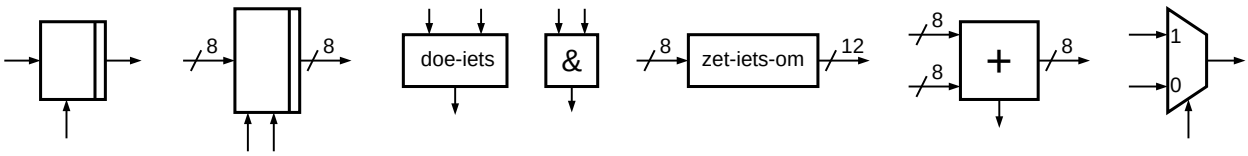
Figuur 6.7 : **Vereenvoudigd schema van een microcontroller.** Een microcontroller bevat een dataverwerkingsdeel en een besturing. De dataverwerking omvat ondermeer de ALU en de besturing krijgt de instructies uit het instructieregister.

6.4 Tekenen dataverwerkingsdeel

Het dataverwerkingsdeel bestaat uit allerlei sequentiële componenten, zoals dataregisters, schuifregisters, tellers en flipfloppen en uit combinatorische componenten, zoals multiplexers, optellers, vermenigvuldigers en andere rekenkundige en logische bewerkingen.
Van het ontwerp van het dataverwerkingsdeel wordt eerst een tekening gemaakt, die aan de volgende eisen voldoet:

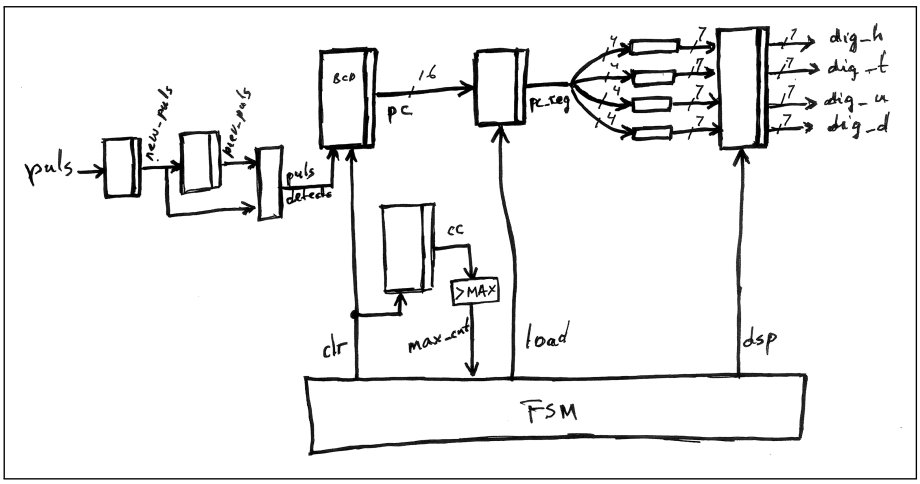
- de tekening bevat alleen functionele signalen;
- het kloksignaal en de globale asynchrone reset worden niet getekend;
- sequentiële componenten krijgen aan de rechter zijde een verticale streep;
- de tekening bevat simpele blokken en symbolen;
- bij de blokken en symbolen staat eventueel een beknopte toelichting;
- alle signalen hebben een naam;
- voor alle signalen is de busbreedte aangegeven.

De tekening, blokken en symbolen moeten eenvoudig zijn. Het is een hulpmiddel bij het ontwerp. De tekening hoeft niet volledig te zijn. Figuur 6.8 geeft een aantal voorbeelden van symbolen.



Figuur 6.8 : **Symbolen voor een tekening van de dataverwerking.** Rechts staan een flipflop en een 8-bits dataregister. Daarnaast staan vijf verschillende combinatorische functies.

Met behulp van potlood en papier is een schets van de dataverwerking snel te maken. In figuur 6.9 staat de schets voor de dataverwerking van de elektronische personenweegschaal. Samen met het toestandsdiagram beschrijft deze tekening het hele ontwerp. Zonder dat er over VHDL gesproken wordt, kunnen belangrijke ontwerpbeslissingen genomen worden. Bij het tekenen is besloten een BCD-teller voor het aantal pulsen te gebruiken. Dit is expliciet gemaakt door in het symbool van de teller *bcd* te zetten.



Figuur 6.9 : Een schets van de dataverwerking voor de elektronische personenweegschaal.

Er is een register `pc_reg` toegevoegd om het aantal getelde pulsen te bewaren. Strikt genomen is dit register overbodig. De reden om dit register toe te voegen is dat hierna een flink stuk combinatoriek volgt, namelijk vier BCD-to-7segment-converters. Zonder register `pc_reg` veranderen de converters voortdurend van waarde en wordt er voortdurend energie gedissipeerd. Met register `pc_reg` erbij veranderen de ingangen alleen als er een nieuwe waarde naar het display wordt geschreven.

De entity is hier weggelaten. Deze bevat naast de ingang `puls` en de vier 7-bits uitgangen `dig_hund`, `dig_ten`, `dig_unit`, `dig_dec` een kloksignaal `clk` en een asynchrone reset `rst_n`.

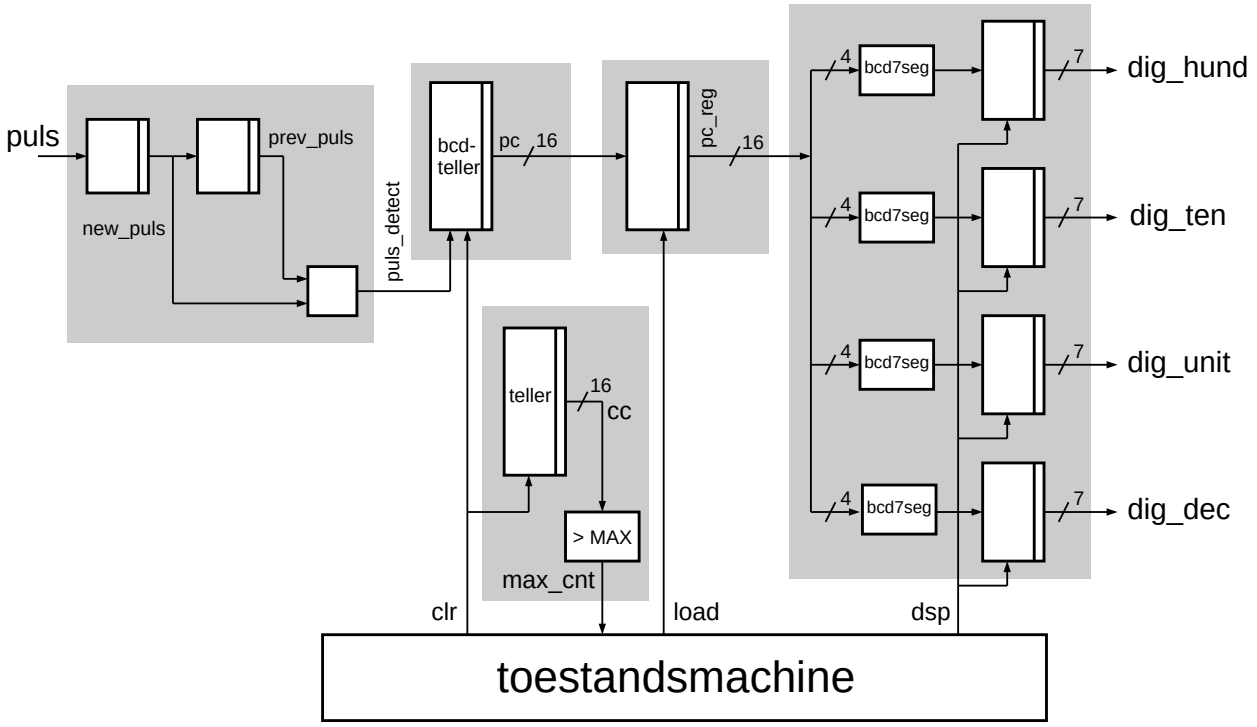
Tijdens het ontwerptraject wordt de schets voortdurend aangepast of moet zelfs opnieuw getekend worden. Alle tussenstappen worden in het projectdossier bewaard. Pas bij het documenteren is het nodig een nette tekening van de dataverwerking te maken. In figuur 6.10 staat het definitieve datapad.

Code 6.4 : De interne signalen van het dataverwerkingsdeel.

```
30   signal pc_reg        : std_logic_vector(15 downto 0);
31   signal pc            : unsigned(15 downto 0);
32   signal cc            : unsigned(15 downto 0);
33   signal new_puls      : std_logic;
34   signal prev_puls     : std_logic;
35   signal puls_detect   : std_logic;
36   signal clr           : std_logic;
37   signal dsp           : std_logic;
38   signal load          : std_logic;
39   signal max_cnt       : std_logic;
```

6.5 Uitwerking dataverwerkingsgedeelte in VHDL

De onderdelen uit het dataverwerkingsdeel kunnen één op één naar VHDL vertaald worden. De tekening bevat tien interne signalen, onder andere de drie stuursignalen `clr`, `dsp` en `load` die van de toestandsmachine komen en het statussignaal `max_cnt` dat naar de toestandsmachine wijst. Samen met de zes andere interne signalen staan deze in het fragment van code 6.4.



Figuur 6.10 : De dataverwerking voor de elektronische personenweegschaal.
Er zijn vijf onderdelen te onderscheiden: een pulsdetector, een pulsteller, een register voor het aantal getelde pulsen, een teller voor de meettijd en een deel dat het resultaat op het display afzet.

In code 6.5 staat de VHDL-beschrijving van het deel met de pulsdetector. De signalen `new_puls` en `prev_puls` staan beide achter de klokflank van regel 73. Voor beide signalen wordt daarom een flipflop gebruikt. De conditionele signaaltoewijzing van regel 79 beschrijft het combinatorische blok dat uit de signalen `new_puls` en `prev_puls` detecteert dat er een nieuwe puls is.

Code 6.5 : De pulsdetector uit het datapad van de elektronische personenweegschaal.

```
68  pulsdetector : process (clk, rst_n) is
69  begin
70    if rst_n = '0' then
71      new_puls  <= '0';
72      prev_puls <= '0';
73    elsif rising_edge(clk) then
74      new_puls  <= puls;
75      prev_puls <= new_puls;
76    end if;
77  end process pulsdetector;
78
79  puls_detect <= '1' when (new_puls = '1') and (prev_puls = '0') else '0';
```

In code 6.6 en code 6.7 staan de VHDL-beschrijvingen van het register `pc_reg` en de teller `cc`. De teller en de verschillende dataregisters krijgen allemaal een eigen proces. Alleen als de voorwaarde, waarop de signalen een andere waarde krijgen,

hetzelfde is, is het handig om de signalen in één proces te plaatsen. De flipfloppen `new_puls` en `prev_puls` krijgen allebei iedere klokslag een nieuwe waarde. Daarom zijn deze in één proces geplaatst.
De teller wordt nul gemaakt als signaal `clr` hoog is en het signaal `pc_reg` verandert alleen als `load` hoog is. Omdat deze voorwaarden verschillend zijn, krijgen deze signalen een eigen sequentieel proces. Signaal `max_cnt` wordt hoog als het aantal getelde klokpulsen groter is dan tienduizend.

Code 6.6 : Het dataregister `pc_reg` uit het datapad.

```
106  pc_register : process (clk,rst_n)  is
107  begin
108    if rst_n = '0' then
109      pc_reg <= (others =>'0');
110    elsif rising_edge(clk) then
111      if load = '1' then
112        pc_reg <= std_logic_vector(pc);
113      end if;
114    end if;
115  end process pc_register;
```

Code 6.7 : De teller voor de klokslagen.

```
117  clockcounter : process (clk, rst_n) is
118  begin
119    if rst_n = '0' then
120      cc <= (others => '0');
121    elsif rising_edge(clk) then
122      if clr = '1' then
123        cc <= (others => '0');
124      else
125        cc <= cc + 1;
126      end if;
127    end if;
128  end process clockcounter;
129
130  max_cnt <= '1' when cc > MAX  else '0';
```

De BCD-teller en de uitgangsregisters hebben verschillende stuursignalen en zijn dus weer een aparte processen, die respectievelijk in code 6.8 en 6.9 staan. Het algoritme voor het incrementeren van een BCD-getal is te complex voor deze uitleg van de implementatie van het model met gescheiden dataverwerking en besturing. In paragraaf 8.3 wordt dit algoritme besproken en in code 8.16 staat het ontbrekende deel van het proces `pulscounter`. In plaats van een eigen BCD-teller te gebruiken, kan de ontwerper ook een BCD-teller uit de bibliotheek van de ontwikkelomgeving toepassen door de betreffende component in de code aan te roepen.

Code 6.8 : De 4-digit BCD-teller.

```
81   pulscounter : process (clk,rst_n)  is
82     variable c : std_logic;
83   begin
84     if rst_n = '0' then
85       pc <= (others =>'0');
86     elsif rising_edge(clk) then
87       if clr = '1' then
88         pc <= (others => '0');
89       elsif puls_detect = '1' then
           ⋮ het algoritme voor
           ⋮ met één ophogen van
           ⋮ BCD-gecodeerde signaal pc
102      end if;
103    end if;
104  end process pulscounter;
```

Code 6.9 : Het uitgangsregister van het datapad.

```
132  outputregisters : process (clk, rst_n) is
133  begin
134    if rst_n = '0' then
135      dig_dec  <= (others => '0');
136      dig_unit <= (others => '0');
137      dig_ten  <= (others => '0');
138      dig_hund <= (others => '0');
139    elsif rising_edge(clk) then
140      if dsp = '1' then
141        dig_dec  <= bcd7seg(pc_reg(3 downto 0));
142        dig_unit <= bcd7seg(pc_reg(7 downto 4));
143        dig_ten  <= bcd7seg(pc_reg(11 downto 8));
144        dig_hund <= bcd7seg(pc_reg(15 downto 12));
145      end if;
146    end if;
147  end process outputregisters;
```

Voor de conversie van de vier BCD-gecodeerde signalen naar de vier signalen, die de 7-segmentsdisplays aansturen, wordt in het proces `outputregisters` een functie `bcd7seg` gebruikt. Deze functie staat in code 6.10.

In het boek van van Dijken wordt niet altijd het gemeten resultaat afgebeeld.

Als het gewicht kleiner is dan 10 kg wordt er niets afgebeeld. De weegschaal lijkt dan uit. Als het gewicht groter is dan 150 kg toont de weegschaal de tekst `err` en als het gewicht kleiner is dan 100 kg wordt het honderdtal niet weergegeven.

Om de letters `e`, `r` en een leeg display weer te geven zijn deze tekens toegevoegd aan `bcd7seg`.

Het algoritme voor deze correcties kan in proces `pc_reg` plaats vinden. In code 6.6 kan regel 112 door deze pseudocode vervangen worden:

```
if pc >= 150 then
  pc_reg <= errE;
elsif pc < 10 then
  pc_reg <= EEEE;
else
  pc_reg <= pc;
  if pc < 100 then
    pc_reg(15 downto 12)<= E;
  end if;
end if;
```

Deze functionaliteit van de weegschaal is voor de uitleg van de ontwerpmethodieken niet heel relevant en is daarom weggelaten.

Wel is het interessant dat het extra register `pc_reg` hier voor gebruikt wordt. Zonder dit register, zou er in het datapad op deze plaats een combinatorisch blok geplaatst moeten worden met dit algoritme.

Code 6.10 : De functie `bcd7seg`.

```
43  function bcd7seg (bcd : in std_logic_vector(3 downto 0))
44                                  return std_logic_vector is
45  begin
46    case bcd is
47      -- digits
48      when "0000" => return "1111110";
49      when "0001" => return "0110000";
50      when "0010" => return "1101101";
51      when "0011" => return "1111001";
52      when "0100" => return "0110011";
53      when "0101" => return "1011011";
54      when "0110" => return "1011111";
55      when "0111" => return "1110000";
56      when "1000" => return "1111111";
57      when "1001" => return "1111011";
58      -- characters
59      when "1100" => return "1101111";  -- e
60      when "1101" => return "1000110";  -- r
61      when "1111" => return "0000000";  -- empty
62      when others => return "0000000";
63    end case;
64  end function bcd7seg;
```

Conclusie over ontwerp en implementatie dataverwerking

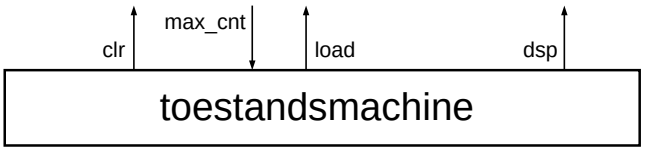
Het ontwerp van het dataverwerkingsdeel van een digitaal systeem bestaat uit eenvoudige digitale componenten. Een schets of tekening van het dataverwerkingsdeel is snel gemaakt en is goed bespreekbaar binnen een ontwerpteam. Belangrijke ontwerpbeslissingen kunnen al in een vroeg stadium van het ontwerptraject gemaakt worden.

De tekening kan één op één naar een VHDL-beschrijving vertaald worden. Het levert een beschrijving op die uit een groot aantal, relatief eenvoudige processen bestaat. Processen van lastig te beschrijven onderdelen kunnen eenvoudig worden vervangen door een aanroep van een bibliotheekcomponent met de gewenste functionaliteit.

Tijdens het hele ontwerp van het dataverwerkingsdeel is alleen vastgelegd welke functies nodig zijn en hoe deze functies met elkaar verbonden zijn. Nergens is ter sprake gekomen wanneer de tellers nul gemaakt moeten worden of wanneer register `pc_reg` zijn nieuwe waarde krijgt. Er is gefocust op de datastructuur en de bewerkingen. De toestandsmachine levert de signalen waarmee de registers en tellers worden aangestuurd. De toestandsmachine bepaalt de volgorde waarin dat gebeurt en regelt daarmee hoe de gegevens verwerkt worden.

6.6 Ontwerp van de toestandsmachine

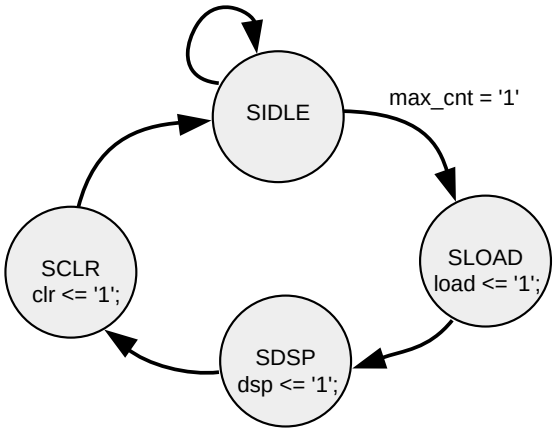
In het voorbeeld van de elektronische personenweegschaal is de toestandsmachine autonoom. Er zijn geen externe besturingssignalen. Figuur 6.11 toont een detail van figuur 6.10 met de toestandsmachine.



Figuur 6.11 : De in- en uitgangssignalen van de toestandsmachine.

De toestandsmachine heeft slechts één ingangssignaal: het statussignaal `max_cnt` waarmee de dataverwerking aangeeft dat de meettijd verstreken is. De toestandsmachine genereert de drie stuursignalen `load`, `dsp`, en `clr`.

In figuur 6.12 staat het toestandsdiagram voor de elektronische personenweegschaal. Er zijn vier toestanden. Samen met de tekening van het datapad uit figuur 6.10 legt het toestandsdiagram uit figuur 6.12 het gedrag van de elektronische personenweegschaal volledig vast.



Figuur 6.12 : Het toestandsdiagram voor de elektronische personenweegschaal.

In de toestand `SIDLE` worden de pulsen van het signaal `puls` geteld. Nadat de meettijd verstreken is, als `max_cnt = '1'`, wordt in toestand `SLOAD` signaal `load` hoog gemaakt. Het aantal getelde pulsen wordt in register `pc_reg` gezet. Eén klokslag later wordt `dsp` hoog en krijgen de uitgangsregisters de nieuwe waarden. Tenslotte wordt in toestand `SCLR` signaal `clr` hoog gemaakt, waardoor bij de volgende klokslag de tellers voor de meettijd `cc` en het aantal getelde pulsen `pc` nul gemaakt worden.

In code 6.11 staat de VHDL-beschrijving van het toestandsdiagram. Bij het coderen is gebruikt gemaakt van het model voor een toestandsmachine met drie processen uit paragraaf 5.7. Er zijn aparte processen voor het toestandsregister en voor de toestandsdecoder. De uitgangsdecoder bestaat uit drie conditionele signaaltoewijzingen.

Code 6.11 : De toestandsmachine voor de elektronische personenweegschaal.

```
27    type state_type is (SIDLE, SLOAD, SDSP, SCLR);
28    signal pres_state, next_state : state_type;
```

```
149   state_reg : process (clk,rst_n) is
150   begin
151     if rst_n = '0' then
152       pres_state  <= SIDLE;
153     elsif rising_edge(clk) then
154       pres_state  <= next_state;
155     end if;
156   end process state_reg;
157
158   next_state_decoder: process (pres_state, max_cnt) is
159   begin
160     case pres_state is
161     when SIDLE =>
162       if max_cnt = '1' then
163         next_state <= SLOAD;
164       else
165         next_state <= SIDLE;
166       end if;
167     when SLOAD =>
168       next_state <= SDSP;
169     when SDSP =>
170       next_state <= SCLR;
171     when SCLR =>
172       next_state <= SIDLE;
173     end case;
174   end process next_state_decoder;
175
176   load <= '1' when pres_state = SLOAD else '0';
177   dsp  <= '1' when pres_state = SDSP  else '0';
178   clr  <= '1' when pres_state = SCLR  else '0';
```

De code van de dataverwerking en de besturing staan in één architectuur en die hoort bij de entity van de personenweegschaal.

Het is niet handig om een aparte entity voor de dataverwerking en een aparte entity voor de besturing te maken. Het testen van het dataverwerkingsdeel zonder besturing is lastig omdat er dan een testbench nodig is die alle besturingssignalen genereert.

De testbench voor een complete personenweegschaal is relatief eenvoudig. Naast het kloksignaal en de reset is er alleen een signaal `puls` nodig.

6.7 Statussignalen en booleans

De manier waarop de status van het dataverwerkingsdeel bij de methode met gescheiden dataverwerking en besturing wordt vastgelegd, vinden sommige ontwerpers onhandig. Deze paragraaf geeft hiervoor een aantal alternatieven.

Bij de methode met gescheiden dataverwerking en besturing zijn alle statussignalen van het type `std_logic`. De status van de teller uit code 6.7 wordt op regel 130 met de conditionele signaaltoewijzing doorgegeven aan de toestandsmachine:

```
131   max_cnt <= '1' when cc > MAX  else '0';
```

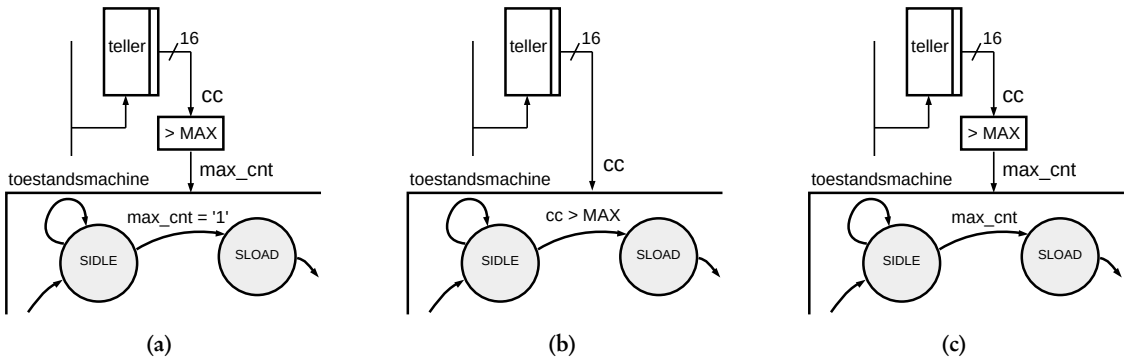
Vanaf VHDL-2008 is er een andere notatie met de operator `??` mogelijk:

```
if ?? max_cnt then
```

Meer informatie over de nieuwe operator `??` staat in bijlage C.6.

In figuur 6.13a staat een detail uit figuur 6.10 met de teller en dit statussignaal. De bijbehorende conditie bij de overgang van toestand `SIDLE` naar toestand `SLOAD` uit de beschrijving van de toestandsmachine van code 6.11 is dan:

```
163       if max_cnt = '1' then
```



Figuur 6.13 : Drie oplossingen voor een statussignaal: voorbeeld a gebruikt een 1-bits signaal `max_cnt`, in voorbeeld b gebruikt de toestandsmachine het 16-bits signaal `cc` en bij voorbeeld c is het signaal van het type `boolean`.

In figuur 6.13b gaat de waarde van de teller `cc` naar de toestandsmachine. De voorwaarde bij de toestandsovergang is dan gelijk aan:

```
163     if cc > MAX then
```

Een andere aanpak is om een signaal van het type `boolean` te gebruiken:

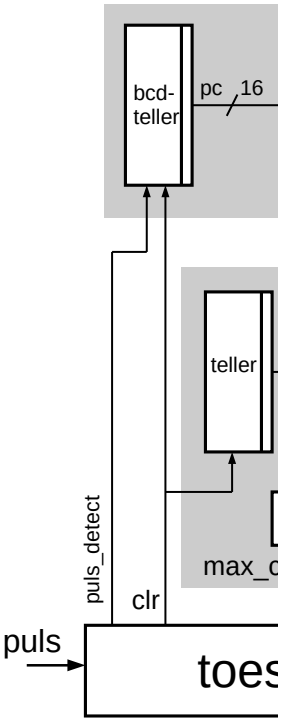
```
        signal max_cnt : boolean;
             ⋮
        max_cnt <= cc > MAX;
```

De waarde van `max_cnt` is, afhankelijk van de waarde die signaal `cc` heeft, gelijk aan `false` of `true`. De voorwaarde bij de toestandsovergang is dan:

```
163     if max_cnt then
```

Figuur 6.13c toont deze aanpak.

Ondanks het feit dat er geen principieel verschil is tussen een `std_logic` en een `boolean` en de toewijzing aan `max_cnt` en de overgangsvoorwaarde beknopter is, levert het gebruik van booleans geen beter leesbare code op. Integendeel, het is een voordeel als alle signalen `std_logic`, of een daarvan afgeleide type als `signed`, `unsigned` of `std_logic_vector` zijn.



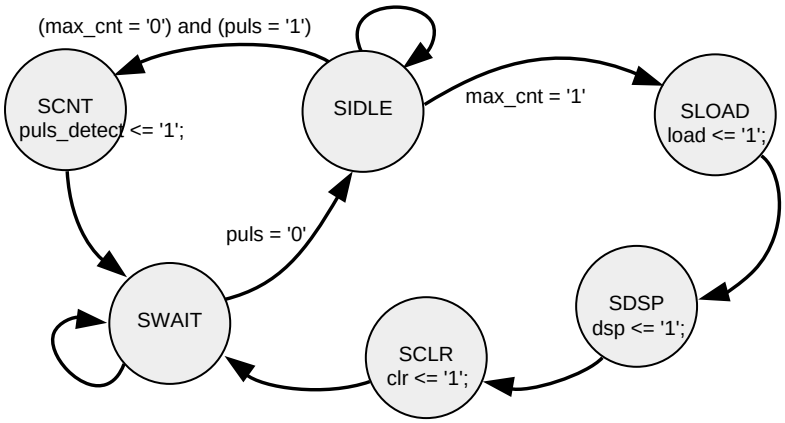
Figuur 6.14 : Detail van alternatief datapad.
Signaal `puls` is een ingang voor de toestandsmachine. Signaal `puls_detect` is hoog als er een nieuwe puls is.

6.8 Alternatieve dataverwerking en besturing

Bij het tekenen van het datapad is besloten het signaal `puls` als een datasignaal te beschouwen. Dat hoeft niet per se. Dit signaal kan ook een ingang van het besturingsdeel zijn. In dat geval moet uit figuur 6.10 het deel met de pulsdetector verwijderd worden. Het signaal `puls_detect` komt dan uit de toestandsmachine, zoals dat in figuur 6.14 is getekend.

Het toestandsdiagram voor deze alternatieve versie staat in figuur 6.15. De toestandsmachine kent nu twee scenario's. Ten eerste moeten de pulsen geteld worden en ten tweede moet het resultaat op het display gezet worden als de meettijd verstreken is. In toestand `SIDLE` doet het systeem niets. Iedere keer als signaal `puls` hoog en de meettijd nog niet verstreken is, gaat de toestandsmachine naar toestand

SCNT. In deze toestand wordt signaal puls_detect hoog. Bij de volgende klokslag komt de machine in toestand SWAIT en wacht totdat signaal puls laag is.



Figuur 6.15 : Het toestandsdiagram voor de alternatieve beschrijving. Signaal puls is hier een ingang van de toestandsmachine.

Nadat de meettijd verstreken is, worden achtereenvolgens de signalen load, dsp en clr hoog. Ook bij dit scenario komt de machine in toestand SWAIT. Dit zorgt ervoor dat een puls nooit meerdere keren geteld wordt.

Als de besturing van een digitaal systeem meer taken doet, wordt de toestandsmachine vaak complexer en lastiger te overzien. Bij de alternatieve beschrijving moeten nu twee dingen tegelijkertijd worden gedaan: het tellen van de pulsen en het afbeelden van de pulsen. Bij de oorspronkelijke beschrijving kent de toestandsmachine slechts één scenario. Het datapad van de oorspronkelijk beschrijving is complexer. Het omvat ook de pulsdetectie, wel zijn in het dataverwerkingsdeel de verschillende functies eenvoudiger te onderscheiden.

6.9 De FSMD-methode

Het is mogelijk om meer functionaliteiten van het dataverwerkingsdeel naar het besturingsdeel te verplaatsen. Zelfs alle functies uit het datapad kunnen in de beschrijving van de toestandsmachine opgenomen worden. Men spreekt dan van een toestandsmachine met dataverwerking oftewel een FSMD, een *finite state machine with a data path*.
In het algemeen geeft een FSMD een meer compacte beschrijving met minder processen dan een ontwerp met een gescheiden dataverwerking en besturing. Een ander voordeel is dat de simulatietijd korter zal zijn. De nadelen van een FSMD zijn dat dit ontwerp minder flexibel is, dat het moeilijker is typisch hardware oplossingen te gebruiken, dat er gemakkelijk foute VHDL-constructies worden toegepast, dat het een slecht leesbare code geeft en dat er geen goede grafische mogelijkheden zijn.

In code 6.12 staat een FSMD-beschrijving van de elektronische personenweegschaal. De opzet is identiek aan de toestandsmachine van de alternatieve beschrijving uit figuur 6.15. De signalen cc, pc, pc_reg en alle uitgangssignalen zijn aan het

Code 6.12 : FSMD-beschrijving van de elektronische personenweegschaal.

```
27    type state_type is (SIDLE, SCNT, SWAIT, SLOAD, SDSP, SCLR);
28    signal state : state_type;
```

```
89    fsmd : process (clk,rst_n) is
90    begin
91      if rst_n = '0' then
92        state     <= SIDLE;
93        cc        <= (others => '0');
94        pc        <= (others => '0');
95        pc_reg    <= (others => '0');
96        dig_hund  <= (others => '0');
97        dig_ten   <= (others => '0');
98        dig_unit  <= (others => '0');
99        dig_dec   <= (others => '0');
100     elsif rising_edge(clk) then
101       cc <= cc + 1;
102       case state is
103       when SIDLE =>
104         if cc > MAX then
105           state <= SLOAD;
106         elsif puls = '1' then
107           state <= SCNT;
108         else
109           state <= SIDLE;
110         end if;
111       when SCNT =>
112         pc <= incrementBCD(pc);
113         state <= SWAIT;
114       when SWAIT =>
115         if puls = '0' then
116           state <= SIDLE;
117         else
118           state <= SWAIT;
119         end if;
120       when SLOAD =>
121         pc_reg <= pc;
122         state <= SDSP;
123       when SDSP =>
124         dig_dec  <= bcd7seg(pc_reg(3 downto 0));
125         dig_unit <= bcd7seg(pc_reg(7 downto 4));
126         dig_ten  <= bcd7seg(pc_reg(11 downto 8));
127         dig_hund <= bcd7seg(pc_reg(15 downto 12));
128         state <= SCLR;
129       when SCLR =>
130         pc <= (others => '0');
131         cc <= (others => '0');
132         state <= SWAIT;
133       end case;
134     end if;
135   end process fsmd;
```

proces fsmd toegevoegd. Omdat deze signalen achter de klokflank staan worden bij synthese hiervoor registers gebruikt.
De functie bcd7seg is identiek aan die uit code 6.10. De functie incrementBCD verhoogt het BCD-gecodeerde signaal pc met één. Deze functie is net als de beschrijving van de BCD-teller uit code 6.8 behoorlijk complex. Het schrijven van de functie is iets lastiger dan de BCD-teller en wordt ook in paragraaf 8.4 besproken. Bij het ontwerp met een gescheiden dataverwerking en besturing ligt het voor de hand om in plaats van een eigen BCD-teller te schrijven, een component uit de bibliotheek te gebruiken. Bij de FSMD-methode ziet de ontwerper niet zo snel dat hier een standaard BCD-teller gebruikt kan worden.

Een nadeel van het FSMD-model is dat er eenvoudig fouten in de VHDL ontstaan. In figuur 6.16 staan twee verschillende versies van code 6.12. In de linker versie wordt signaal cc opgehoogd en daarna volgt het case-statement. In de rechter versie staat eerst het case-statement en daarna wordt het signaal cc opgehoogd.

```
  elsif rising_edge(clk) then
    cc <= cc + 1;
    case state is
    when SIDLE =>
          ⋮
    when SCLR =>
      pc <= (others => '0');
      cc <= (others => '0');
      state <= SWAIT;
    end case;

  end if;
end process fsmd;
```

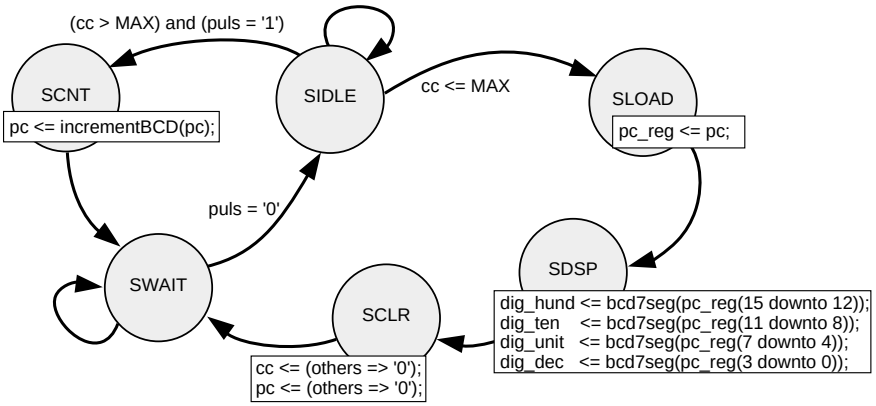
```
  elsif rising_edge(clk) then

    case state is
    when SIDLE =>
          ⋮
    when SCLR =>
      pc <= (others => '0');
      cc <= (others => '0');
      state <= SWAIT;
    end case;
    cc <= cc + 1;
  end if;
end process fsmd;
```

Figuur 6.16 : **Mogelijke onduidelijkheden bij FSMD-model.** Links staat de beschrijving van code 6.12. Het signaal cc wordt voor het case-statement opgehoogd. Rechts staat een alternatief, waarbij het ophogen na het case-statement komt. Hierdoor wordt signaal cc nooit nul gemaakt.

Bij de rechter beschrijving heeft het nul maken van signaal cc in toestand SCLR geen effect. De nieuwe waarde voor het signaal cc wordt in toestand SCLR nul gemaakt. Omdat dit een signaaltoewijzing is, wordt deze waarde niet direct toegekend. Signaal cc heeft nog de oude waarde. Deze waarde wordt gebruikt bij het ophogen. De nieuwe waarde voor signaal cc is de oude waarde van cc, die met één is opgehoogd. Na de evaluatie van alle processen krijgt signaal cc die waarde en dus niet 1 wat een onervaren VHDL-ontwerper misschien verwacht.
Bij de linker versie wordt signaal cc eerst opgehoogd. De nieuwe waarde voor het signaal cc is dan de huidige waarde plus één. In toestand SCLR wordt deze nieuwe waarde overruled met nul. Na de evaluatie van alle processen krijgt signaal cc dan de waarde nul.

Om deze problemen te omzeilen, gebruikt men in plaats van signalen variabelen. Er is niets tegen het gebruik van variabelen. Integendeel, bij het simuleren is voor variabelen geen interne datastructuur nodig en zullen de simulaties sneller zijn. Het nadeel van variabelen is dat het effect bij synthese niet eenduidig is. Signalen achter een klokflank leveren altijd een register op. Variabelen achter

een klokflank kunnen een register opleveren. Vaak leidt dit tot een ingewikkelde beschrijving met variabelen en signalen en extra toestanden om het tijdsgedrag in orde te krijgen.



Figuur 6.17 : Het FSMD-model voor de elektronische personenweegschaal.

In figuur 6.17 staat een tekening van het FSMD-model voor de elektronische personenweegschaal. De dataverwerkingsaspecten staan als signaaltoewijzingen bij de betreffende toestanden. Bij complexere systemen kan dit zeer onoverzichtelijk worden.

6.10 De ASMD-methode

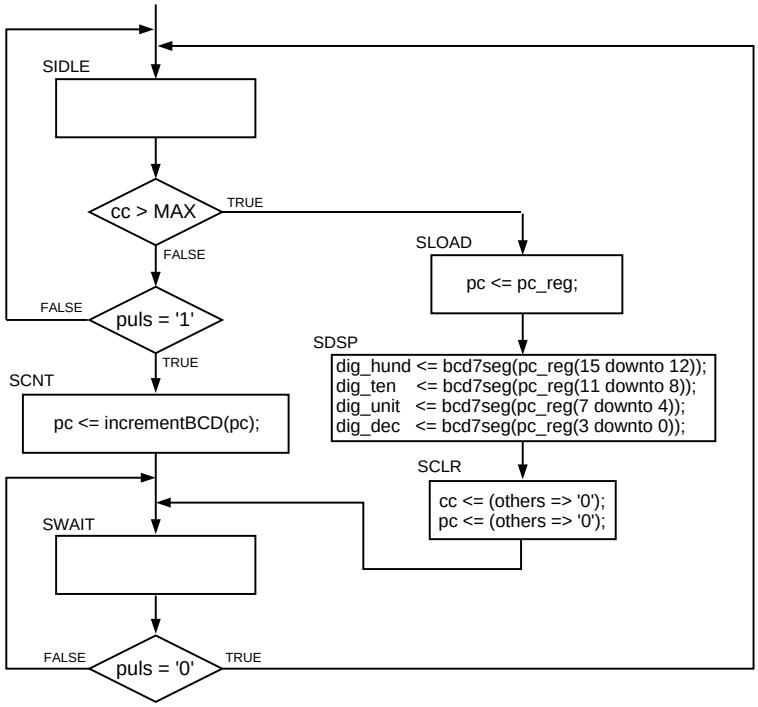
Het FSMD-model kan ook als ASM-chart worden getekend. In plaats van een toestandsdiagram kan de ontwerper ook een ASM-chart met dataverwerking tekenen oftewel een ASMD-chart. De FSMD-methode zou men evengoed de ASMD-methode kunnen noemen.

Voor deze ASMD-methode gelden dezelfde voor- en nadelen als voor de FSMD-methode, alleen levert dit een tekening op die beter leesbaar is. In figuur 6.18 staat de ASMD-chart voor de elektronische personenweegschaal.

Veel ontwikkelomgevingen voor FPGA's bevatten grafische programma's waarmee toestandsdiagrammen en ASM-charts getekend kunnen worden. Aan deze tekeningen kunnen meestal ook dataverwerkingsaspecten worden toegevoegd. Een beginnend VHDL-ontwerper is bij het gebruik van deze programma's snel geneigd alles met het toestandsdiagram of met het ASM-chart op te lossen. Men vergeet dan om een BCD-teller voor het ophogen van de te tellen pulsen te gebruiken.

6.11 De twee-processenmethode

VHDL voert de processen en de parallelle signaaltoewijzingen niet uit in de volgorde waarin deze opgeschreven zijn. De volgorde wordt bepaald door de gebeurtenissen. Als een signaal verandert triggert dat andere signaaltoewijzingen of processen. Deze wijze van uitvoeren staat dicht bij de hardware, maar maakt de interpretatie bij veel processen en toewijzingen moeilijk.

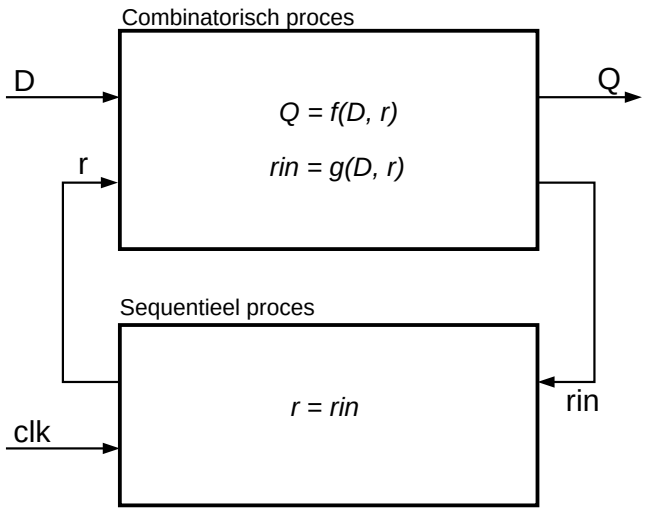


Figuur 6.18 : De ASMD-chart voor de elektronische personenweegschaal.

Jiri Gaisler van Gaisler Research AB was eerder werkzaam bij het ESA, European Space Agency. Als systeemontwerper heeft hij daar gewerkt aan de ontwikkeling van fouttolerante microprocessoren voor de ruimtevaart. In het document "Fault-tolerant Microprocessors for Space Applications- uit 2008 "stelt hij in hoofdstuk 5, "A structured VHDL design method- de twee-processenmethode voor.

Om de eenduidigheid en de leesbaarheid van de VHDL te verbeteren, heeft Jiri Gaisler de twee-processenmethode voorgesteld. Iedere entity heeft bij dit model slechts twee processen: een combinatorisch proces dat alle combinatorische functies bevat en een sequentieel proces dat alle registers beschrijft. Het combinatorische proces bestaat uit variabele- en signaaltoewijzingen die op een sequentiële manier doorlopen worden.

Voor de lucht- en ruimtevaart is het maken van soft- en hardware zonder fouten essentieel. Er worden in dat vakgebied extreem hoge eisen gesteld aan de betrouwbaarheid en aan de fouttolerantie. Het is dus niet vreemd dat er vanuit dit vakgebied veel aandacht is voor methodieken die tot minder ontwerpfouten leiden.



Figuur 6.19 : Het algemene blokschema voor de twee-processenmethode.

Verwar de twee-processenmethode niet met het twee-processenmodel voor een toestandsmachine uit figuur 5.32.
De twee-processenmethode stemt wel overeen met het model uit figuur 5.36.

Figuur 6.19 toont het blokschema voor deze aanpak. De ingangen `D` van de entity zijn verbonden met het combinatorische proces. De ingangen van het sequentiële

proces zijn de interne signalen `rin` die van het combinatorische proces komen. Het sequentiële proces kopieert bij de actieve klokflank deze interne signalen naar de interne signalen `r`, die weer ingangen zijn van het combinatorische proces. De uitgangssignalen van de entity zijn uitgangssignalen van het combinatorische proces. Twee vergelijkingen leggen de functionaliteit van het combinatorische proces vast:

$$\mathtt{Q} = f(\mathtt{D}, \mathtt{r}) \tag{6.7}$$

$$\mathtt{rin} = g(\mathtt{D}, \mathtt{r}) \tag{6.8}$$

Op pagina 156 staat in code 6.14 het voorbeeld van een 8-bits teller dat Jiri Gaisler bij zijn voorstel gebruikt. Het combinatorische proces uit deze beschrijving heeft twee uitgangen, namelijk de signaaltoewijzing aan `rin` op regel 28 en de signaaltoewijzing aan `q` op regel 29. Signaal `q` hangt alleen van signaal `r` af. De waarde van `rin` hangt af van de signalen `load`, `count`, `d` en `r`. Om problemen met meervoudige signaaltoewijzingen te voorkomen, is er een hulpvariabele `tmp` gebruikt.

Code 6.13 geeft de beschrijving van de elektronische personenweegschaal met behulp van de twee-processenmethode. Het proces `sequential` beschrijft in feite een hele verzameling dataregisters. Bij de actieve klokflank worden de nieuw berekende waarden in deze registers gezet.
Alle ingangssignalen van het proces `combinational` worden bij het begin van het proces, op regel 103 tot en met 109, aan variabelen toegekend. Afhankelijk van de toestand waarin het systeem zich bevindt, past het case-statement op regel 110 tot en met 141 deze variabelen aan. Aan het eind van het proces worden de variabelen aan de uitgangssignalen van dit proces toegekend. Op deze manier staan alle veranderingen sequentieel achter elkaar. Er zijn in het process geen afhankelijke meervoudige signaaltoewijzingen en bovendien is de combinatoriek gescheiden van de registers. Proces `combinational` is een proces dat sequentieel doorlopen wordt.

6.12 Keuze methodiek

Formeel lijkt het twee-processenmethode de juiste strategie voor het beschrijven van digitale systemen, alleen voor hardware-ontwerpers is deze aanpak nogal geforceerd. Verschillende functionaliteiten van de specifieke onderdelen zijn nu verdeeld over twee processen. Van de pulsteller staat de eigenschap dat het een register is in proces `sequential` en staan het ophogen op regel 120 en het nul maken op regel 138 van proces `combinational`. Dit kan voor hardware-ontwerpers zeer verwarrend zijn.

De FSMD- en de ASMD-methode geven een compacte beschrijving, maar lenen zich minder goed voor typische hardware oplossingen. Bovendien worden er gemakkelijk fouten in de beschrijvingen gemaakt. De sequentiële en parallelle aspecten van VHDL komen op een onduidelijke manier bij elkaar te staan.

De methode met een gescheiden dataverwerking en besturing levert een relatief grote beschrijving met veel processen op. Het voordeel van deze methode is dat met twee tekeningen — de schets van de dataverwerking en het toestandsdiagram — het complete ontwerp vastligt. Bovendien zijn deze tekeningen één op één over te zetten naar synthetiseerbare VHDL. Voor beginnende VHDL-ontwerpers, mits ze de basis kennen van de digitale techniek, geeft deze methode

```
 91 combinational : process (puls, curr_state,
 92                          curr_cc, curr_pc, curr_pc_reg,
 93                          curr_dig_hund, curr_dig_ten,
 94                          curr_dig_unit, curr_dig_dec) is
 95   variable v_cc       : unsigned(15 downto 0);
 96   variable v_pc       : std_logic_vector(15 downto 0);
 97   variable v_pc_reg   : std_logic_vector(15 downto 0);
 98   variable v_dig_dec  : std_logic_vector(6 downto 0);
 99   variable v_dig_unit : std_logic_vector(6 downto 0);
100   variable v_dig_ten  : std_logic_vector(6 downto 0);
101   variable v_dig_hund : std_logic_vector(6 downto 0);
102 begin
103   v_cc       := curr_cc + 1;
104   v_pc       := curr_pc;
105   v_pc_reg   := curr_pc_reg;
106   v_dig_dec  := curr_dig_dec;
107   v_dig_unit := curr_dig_unit;
108   v_dig_ten  := curr_dig_ten;
109   v_dig_hund := curr_dig_hund;
110   case curr_state is
111   when SIDLE =>
112     if curr_cc > MAX then
113       next_state <= SLOAD;
114     elsif puls = '1' then
115       next_state <= SCNT;
116     else
117       next_state <= SIDLE;
118     end if;
119   when SCNT =>
120     v_pc    := incrementBCD(curr_pc);
121     next_state <= SWAIT;
122   when SWAIT =>
123     if puls = '0' then
124       next_state <= SIDLE;
125     else
126       next_state <= SWAIT;
127     end if;
128   when SLOAD =>
129     v_pc_reg := curr_pc;
130     next_state  <= SDSP;
131   when SDSP =>
132     v_dig_dec  := bcd7seg(curr_pc_reg(3 downto 0));
133     v_dig_unit := bcd7seg(curr_pc_reg(7 downto 4));
134     v_dig_ten  := bcd7seg(curr_pc_reg(11 downto 8));
135     v_dig_hund := bcd7seg(curr_pc_reg(15 downto 12));
136     next_state <= SCLR;
137   when SCLR =>
138     v_pc    := (others => '0');
139     v_cc    := (others => '0');
140     next_state <= SWAIT;
141   end case;
142   next_cc       <= v_cc;
143   next_pc       <= v_pc;
144   next_pc_reg   <= v_pc_reg;
145   next_dig_hund <= v_dig_hund;
146   next_dig_ten  <= v_dig_ten;
147   next_dig_unit <= v_dig_unit;
148   next_dig_dec  <= v_dig_dec;
149 end process combinational;
```

Code 6.13 : De twee-processenmethode voor de elektronische personenweegschaal. Links staat het combinatorische proces dat uit de huidige waarden de nieuwe waarden berekent. Hieronder staat het sequentiële proces dat de nieuwe waarden toekent aan de huidige waarden.

```
151 sequential : process (clk,rst_n) is
152 begin
153   if rst_n = '0' then
154     curr_state     <= SIDLE;
155     curr_cc        <= (others => '0');
156     curr_pc        <= (others => '0');
157     curr_pc_reg    <= (others => '0');
158     curr_dig_hund  <= (others => '0');
159     curr_dig_ten   <= (others => '0');
160     curr_dig_unit  <= (others => '0');
161     curr_dig_dec   <= (others => '0');
162   elsif rising_edge(clk) then
163     curr_state     <= next_state;
164     curr_cc        <= next_cc;
165     curr_pc        <= next_pc;
166     curr_pc_reg    <= next_pc_reg;
167     curr_dig_hund  <= next_dig_hund;
168     curr_dig_ten   <= next_dig_ten;
169     curr_dig_unit  <= next_dig_unit;
170     curr_dig_dec   <= next_dig_dec;
171   end if;
172 end process sequential;
173
174 dig_hund      <= curr_dig_hund;
175 dig_ten       <= curr_dig_ten;
176 dig_unit      <= curr_dig_unit;
177 dig_dec       <= curr_dig_dec;
```

Code 6.14 : Een 8-bits teller volgens de twee-processenmethode.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity count8 is
6    port (
7      clk   : in  std_logic;
8      load  : in  std_logic;
9      count : in  std_logic;
10     d     : in  std_logic_vector(7 downto 0);
11     q     : out std_logic_vector(7 downto 0));
12 end entity count8;
13
14 architecture twoproc of count8 is
15   signal r, rin : unsigned(7 downto 0);
16 begin
17
18   combinational : process (load, count, d, r) is
19     variable tmp : unsigned(7 downto 0);
20   begin
21     if load = '1' then
22       tmp := unsigned(d);
23     elsif count = '1' then
24       tmp := r + 1;
25     else
26       tmp := r;
27     end if;
28     rin <= tmp;
29     q   <= std_logic_vector(r);
30   end process combinational;
31
32   sequential : process (clk) is
33   begin
34     if rising_edge(clk) then
35       r <= rin;
36     end if;
37   end process sequential;
38
39 end architecture twoproc;
```

het beste resultaat. Het grootste voordeel van deze methode is dat in een vroeg stadium het ontwerp besproken kan worden aan de hand van tekeningen, zonder dat er al een VHDL-code is.

Het programma Ease van HDL Works is wel heel interessant voor de methode met een gescheiden dataverwerking en besturing.

Met de meeste grafische ontwerpprogramma's kunnen toestandsdiagrammen en blokschema's getekend worden. Mits de flexibiliteit groot genoeg is, zijn deze programma's bruikbaar bij de FSMD-methode. Voor de methode met een gescheiden dataverwerking en besturing zijn deze programma's meestal minder geschikt. Het tekenen van blokken, processen en verbindingen kost veel tijd. Bovendien geeft het toevoegen van de VHDL-code aan de verschillende processen meer problemen als het schrijven van code in één enkel VHDL-bestand.

7

Algoritmes

Doelstelling	In dit hoofdstuk en het volgende hoofdstuk leer je hoe je een algoritme kunt opstellen. Dit hoofdstuk bespreekt wat een algoritme is en hoe je voor een wiskundig probleem een optimaal algoritme kunt vinden. Bij het onderzoek naar algoritmes wordt de taal C gebruikt en leer je een algoritme in C om te zetten naar een VHDL-beschrijving.
Onderwerpen	De behandelde onderwerpen zijn: ▪ De definitie van een algoritme. ▪ Een voorbeeld van een algoritme uit het dagelijks leven, zoals het zetten van koffie. ▪ Een voorbeeld van een wiskundig algoritme, namelijk het vermenigvuldigen van twee getallen. ▪ De grootste gemeenschappelijke deler. ▪ Een intuïtief algoritme voor de grootste gemeenschappelijke deler. ▪ Het algoritme van Euclides voor de grootste gemeenschappelijke deler. ▪ Het gebruik van recursie bij algoritmes ▪ Het algoritme van Josef Stein voor de grootste gemeenschappelijke deler. ▪ Het selecteren van een geschikt algoritme. ▪ De implementatie van een algoritme in VHDL met de methode van gescheiden dataverwerking en besturing. ▪ De implementatie van een algoritme in VHDL met de FSMD-methode. ▪ Een lijst met adviezen voor het vinden van een bruikbaar algoritme.

Een digitaal systeem heeft niet alleen een bepaalde hiërarchische structuur, die met blokschema's beschreven kan worden. Een digitaal systeem is vaak de implementatie van een bepaald algoritme dat opgelost moet worden en bestaat uit delen, die één of meer taken uitvoeren. Deze taken zijn dan implementaties van algoritmes.

Een algoritme is een voorschrift hoe een bepaald probleem opgelost kan worden. In veel gevallen zijn dit wiskundige problemen, maar in het dagelijks leven zijn er ook algoritmes te herkennen. Ieder kookboek staat vol met recepten waarmee gerechten bereid kunnen worden. Elk recept legt vast welke stappen de kok

achtereenvolgens moet doen. Een voorbeeld is het zetten van koffie met behulp van een koffiezetapparaat. Afhankelijk van het type apparaat zijn er verschillende procedures mogelijk. Dit is een mogelijke recept:

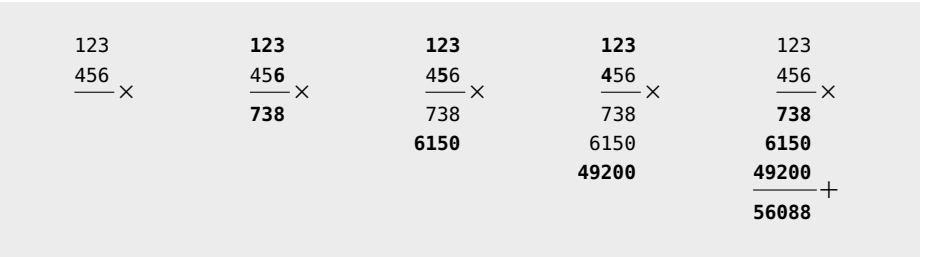
- spoel de koffiekan om en zet deze in het apparaat;
- pak een koffiefilterzakje en plaats deze in het koffiefilter;
- pak de koffie en schep de koffie in het filter;
- vul het waterreservoir met water;
- zet het koffiezetapparaat aan;
- wacht tot de koffie klaar is.

Algoritmes zijn dikwijls te beschrijven met flow-charts of stroomdiagrammen. Zeker als het de besturing van een proces beschrijft, kan een toestandsdiagram het algoritme vastleggen.

In het algemeen hangen algoritmes af van bepaalde gegevens. Bij het koffiezetten is dat bijvoorbeeld het aantal kopjes dat gezet moet worden. Het algoritme bevat dan herhalingsopdrachten als: doe acht schepjes koffie in het koffiefilter.

Wiskundige algoritmes

Iedereen kent verschillende wiskundige algoritmes, zoals vermenigvuldigen en delen met een staartdeling of een andere methode. Figuur 7.1 geeft de klassieke methode om twee getallen met elkaar te vermenigvuldigen.



Figuur 7.1 : Het klassieke algoritme voor een vermenigvuldiging.
Zet de getallen onder elkaar. Vermenigvuldig het meest rechtse cijfer van het tweede getal met het eerste getal. Neem vervolgens steeds het volgende cijfer en voeg aan het resultaat een extra nul toe totdat de cijfers van het tweede getal op zijn. Tel al deze uitkomsten op voor het eindresultaat.

Voor optellen, aftrekken en vermenigvuldigen bestaan veel verschillende hardware-oplossingen. Iedere synthesizer herkent — voor zowel `unsigned` als `signed` — de onderstaande bewerkingen en zal deze synthetiseren:

```
x <= a + b;
y <= a - b;
z <= a * b;
```

Voor vermenigvuldigen wordt meestal een variant van het Booth-algoritme gebruikt. In 1951 is dit algoritme, dat gebaseerd is op schuiven en optellen, door Andrew Donald Booth voorgesteld.

De synthesizer gebruikt bij optellen en aftrekken de mogelijkheden om de berekening van de `carry` snel door te geven en zorgt er voor dat de hoeveelheid logica toch beperkt blijft. Tegenwoordig bevatten bijna alle FPGA's naast een grote hoeveelheid logische blokken ook meerdere *multipliers* voor het vermenigvuldigen.

Hoewel er veel gepubliceerd is over het optimaliseren van optellers en vermenigvuldigers zijn de details voor de gemiddelde FPGA-ontwerper minder interessant. Andere rekenkundige bewerkingen worden daarentegen door de synthesizers niet herkend en dus niet gesynthetiseerd. Wel bevatten tegenwoordig de ontwikkelomgevingen van FPGA's bibliotheken met blokken voor allerlei rekenkundige bewerkingen. Ook zijn er IP- of softcores voor rekenkundige functies beschikbaar.

Een belangrijke reden waarom de bewerkingen als delen, logaritme, exponentiële en goniometrische functies niet synthetiseerbaar zijn. is dat hiervoor gebroken getallen nodig zijn. In VHDL-2008 zijn door IEEE afspraken gemaakt over fixed-point- en floating-point getallen. VHDL-2008 zal de komende jaren in de verschillende ontwikkelomgevingen geïmplementeerd worden en voor deze IEEE-bibliotheek zullen steeds meer standaardoplossingen beschikbaar zijn.

Een andere probleem bij de implementatie van rekenkundige bewerkingen is dat bij veel algoritmes meerdere klokslagen nodig zijn. Dit betekent dat er meer signalen bij betrokken zijn. De bewerking moet met een signaal gestart worden en maakt een signaal actief als de bewerking klaar is.
Het is onwaarschijnlijk dat in de nabije toekomst deze bewerkingen direct synthetiseerbaar zijn:

```
r <= a / b;
s <= a % b;
t <= sin(a);
u <= cos(b);
v <= exp(a);
w <= log(a);
```

Algoritmes zijn niet alleen van belang bij rekenkundige bewerkingen. Ze spelen een hoofdrol bij allerlei digitale systemen, zoals: digitale filters, digitale regelsystemen, Fouriertransformaties, encryptie, digitale audio, beeldbewerking en navigatiesystemen. De meeste van deze onderwerpen zijn te complex om hier te behandelen. Dit hoofdstuk behandelt het kiezen en opstellen van algoritmes aan de hand van het bepalen van de grootste gemeenschappelijke deler. Het volgende hoofdstuk sluit hierbij aan met de verschillende rekenkundige bewerkingen voor Binary-Coded-Decimal-getallen, zoals de conversie van binaire getallen naar BCD-getallen.

7.1 De grootste gemeenschappelijke deler

De grootste gemeenschappelijke deler of GGD van twee getallen is het grootste gehele getal waardoor beide getallen gedeeld kunnen worden. De GGD van 48 en 36 is bijvoorbeeld gelijk aan 12. Hoewel op het eerste gezicht een grootste gemeenschappelijke deler in dagelijks leven niet belangrijk lijkt te zijn, wordt het intensief gebruikt bij het versleutelen van gegevens. Het populaire RSA-algoritme gebruikt de GGD voor de encryptie.

Vroeger sprak men van grootste gemene deler, tegenwoordig gebruiken wiskundeboeken meestal grootste gemeenschappelijke deler. In het Engels is het de *greatest common divisor* of GCD.

Deze paragraaf gebruikt naast VHDL ook de taal C om algoritmes te beschrijven. Er zijn meerdere redenen om C te gebruiken:

- Veel algoritmes zijn al in C of C++ opgeschreven.
- Voor VHDL moet veel zelf gemaakt of herschreven worden.
- De implementatie in synthetiseerbare VHDL is vaak complexer.

- Bij de implementatie zijn in VHDL vaak nieuwe keuzes te maken, waardoor het lastig is een uitspraak te doen over de snelheid van het algoritme.
- Het testen van de C-code gaat veel sneller.

Op internet en in de literatuur zijn veel verschillende algoritmes voor de grootste gemeenschappelijke deler te vinden. Bij een aantal oplossingen, die hier gepresenteerd worden, is geen bron vermeld. Een oorspronkelijke bron is dan niet te achterhalen. Bovendien zijn de oplossingen aangepast, zodat ze qua stijl en opmaak meer op elkaar lijken en de verschillen beter zichtbaar zijn.

Een intuïtief algoritme

Een intuïtief algoritme voor het vinden van de GGD is beide getallen te delen door het grootst mogelijke getal en als dat niet lukt dit deeltal steeds met één te verlagen, totdat beide getallen wel deelbaar zijn door het deeltal. In pseudocode is dit het algoritme:

```
neem als deeltal een van de getallen
zolang beide getallen niet deelbaar zijn door het deeltal
  verlaag het deeltal met één
```

De afspraak is dat `gcd(0,y)` gelijk is aan `y` en dat `gcd(x,0)` gelijk is aan `x`.

Na afloop bevat het deeltal de grootste gemeenschappelijke deler. In code 7.1 staat in de taal C een functie `gcd`, die volgens dit algoritme de GGD berekent. De functie heeft twee parameters `x` en `y`. De hulpvariabele `d` krijgt de waarde van `x`. Als `d` nul is, wordt de waarde van `y` teruggegeven. Als beide getallen deelbaar zijn door `d` stopt het verlagen van `d` en wordt `d` als resultaat teruggegeven.

Code 7.1 : Een C-algoritme voor de berekening van de GGD.

```
 1  int gcd(int x, int y)
 2  {
 3    int d = x;
 4
 5    if ( d == 0 ) {
 6      return y;
 7    }
 8
 9    while( (x%d) || (y%d) ) {
10      d--;
11    }
12
13    return d;
14  }
```

Code 7.2 : Een verbeterde variant van code 7.1.

```
 1  int gcd(int x, int y)
 2  {
 3    int d = (x > y) ? y : x;
 4
 5    if ( d == 0 ) {
 6      return ((x > y) ? x : y);
 7    }
 8
 9    while( (x%d) || (y%d) ) {
10      d--;
11    }
12
13    return d;
14  }
```

Het maximaal aantal stappen dat het algoritme moet doorlopen, is gelijk aan het grootste getal waarvoor de GGD bepaald moet worden. In code 7.2 wordt als deeltal `y` gebruikt als deze kleiner is dan `x`. Het maximaal aantal stappen is dan altijd gelijk aan het kleinste getal. Hoewel dit algoritme uit code 7.2 meestal sneller is dan dat van code 7.1, zijn beide algoritmes voor grote getallen inefficiënt.
Bovendien gebruiken beide oplossingen de modulus-operator om te bepalen of de getallen deelbaar zijn door het deeltal. De modulus is geen eenvoudig functie. Een microprocessor heeft hiervoor veel meer klokslagen nodig dan voor een sommatie of voor een schuiffunctie. Een bijkomend probleem voor een hardware implementatie is dat de modulus-operator niet zonder meer synthetiseerbaar is.

Het algoritme van Euclides

Het werk 'Elementen' van de Griekse wiskundige Euclides is het belangrijkste wiskundige werk uit de klassieke oudheid en bestaat uit dertien boeken. Ieder boek bespreekt enige tientallen proposities. Een propositie is een stelling of theorema.
Propositie 1 en 2 uit boek 7 behandelen de grootste gemeenschappelijke deler.

Een Engelse vertaling van de 'Elementen' vertaald door Richard Fitzpatrick staat op deze internetpagina:
http://farside.ph.utexas.edu/euclid.html

De intuïtieve methode is typisch een oplossing die je ook zelf bedacht zou kunnen hebben. Over heel veel problemen is door anderen al eerder nagedacht. Er zijn daarom vaak elegante en slimme oplossingen beschikbaar.
Rond 300 voor Christus heeft de Griekse wijsgeer Euclides al een slim en eenvoudig algoritme voor het bepalen van de grootst gemeenschappelijke deler opgeschreven.
Figuur 7.2 geeft voor drie voorbeelden alle stappen van het algoritme van Euclides. Dit algoritme is gebaseerd op het idee dat de grootste gemeenschappelijke deler van twee getallen hetzelfde blijft als het kleinere getal van het grotere aantal afgetrokken wordt.
Het linker voorbeeld uit figuur 7.2 bepaalt de GGD voor de getallen 48 en 36. Het kleinste getal (36) kan één keer van het grootste getal (48) worden afgetrokken. Er blijft dan 12 over. Bij de volgende stap is het grootste getal 36 en het kleinste getal 12. Dit laatste getal past precies 3 keer in 36 en er blijft niets over. Het grootste getal wordt nu 12 en het kleinste getal 0. Omdat het kleinste getal nul is, stopt het algoritme en is het resultaat 12.

```
 x   y  x/y rest          x   y   x/y rest          x   y   x/y rest
48  36    1   12        144  15    9    9         55  34    1   21
36  12    3    0         15   9    1    6         34  21    1   13
12   0                    9   6    1    3         21  13    1    8
                          6   3    2    0         13   8    1    5
                          3   0                    8   5    1    3
                                                   5   3    1    2
                                                   3   2    1    1
                                                   2   1    1    0
                                                   1   0

GGD = 12               GGD = 3               GGD = 1
```

Figuur 7.2 : Drie voorbeelden om de GGD te bepalen met de methode van Euclides. Afhankelijk van de gekozen getallen heeft het algoritme meer of minder stappen nodig. Bij elk voorbeeld wordt steeds de rest van de deling van x en y bepaald en krijgt de nieuwe x de waarde van y en de nieuwe y de waarde van de rest. Het algoritme stopt als y nul is, x bevat dan het eindresultaat.

De getallen van Fibonacci vormen een reeks, beginnend met 0 en 1, waarbij het volgende getal de som van de voorgaande twee getallen is. De reeks begint met: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

Het rechter voorbeeld geeft de meest ongunstige situatie voor het algoritme van Euclides. Bij twee opeenvolgende getallen uit de reeks van Fibonacci zijn de meeste stappen nodig. In dit voorbeeld, met 55 en 34, zijn er acht stappen nodig.

Code 7.3 geeft de functie in C voor het algoritme van Euclides. Het herhaald aftrekken van het kleinste getal van het grootste getal is een deling en de rest kan worden gevonden met de modulus-operator.
In code 7.4 staat een variant op de methode van Euclides, die in de praktijk een klein beetje sneller is. Het aantal iteraties is de helft van het aantal iteraties dat bij Euclides nodig is, maar bij iedere iteratie moet wel twee keer een modulus worden uitgerekend.

Bij een implementatie in hardware is er bij beide oplossingen een modulusfunctie nodig. In code 7.4 hangt de tweede modulusberekening af van de eerste. Beide bewerkingen moeten na elkaar worden uitgevoerd. Als de bewerkingen onafhankelijk zouden zijn, zou de implementatie met twee parallelle modulusoperatoren

Code 7.3 : Het algoritme van Euclides in de taal C.

```
1  int gcd(int x, int y)
2  {
3    int rest;
4
5    while (y != 0) {
6      rest = x % y;
7      x = y;
8      y = rest;
9    }
10
11   return x;
12 }
```

Code 7.4 : Een alternatieve versie voor het algoritme van Euclides.

```
1  int gcd(int x, int y)
2  {
3    if ( x==0 ) return y;
4    if ( y==0 ) return x;
5
6    while (1) {
7      x = x % y;
8      if ( x == 0 ) return y;
9      y = y % x;
10     if ( y == 0 ) return x;
11   }
12 }
```

twee keer zo snel kunnen zijn. Omdat de bewerkingen na elkaar moeten worden uitgevoerd, geeft de implementatie van twee parallelle modulusfuncties geen snelheidswinst.

Het algoritme van Euclides met recursie

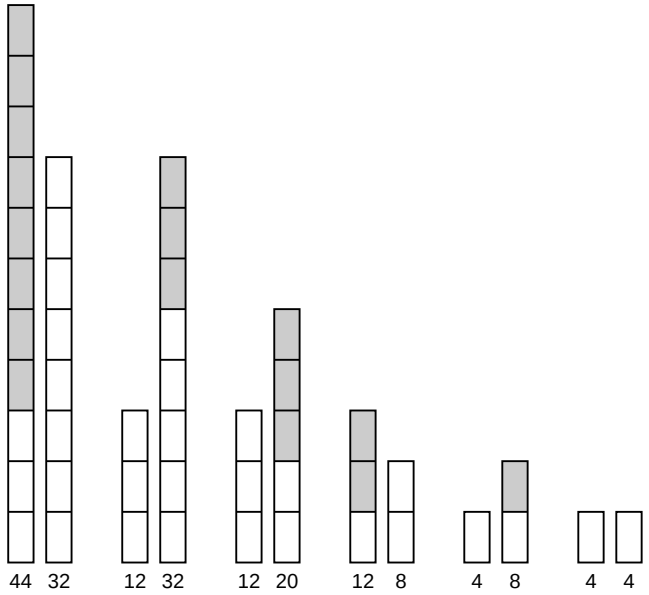
Veel oplossingen voor het bepalen van GGD maken gebruik van recursie. Code 7.5 geeft de recursieve versie van het algoritme van Euclides. Recursieve oplossingen zijn vaak elegant en beknopt. Bij iedere volgende stap roept de functie `gcd` zichzelf aan met als nieuwe waarde voor `x` de oude `y` en als nieuwe waarde voor `y` de modulus van de oude `x` en `y`. Als `y` gelijk is aan nul stopt het algoritme en geeft het `x` terug.

Code 7.5 : Het recursieve algoritme van Euclides.

```
1  int gcd(int x, int y)
2  {
3    if ( y == 0 ) return x;
4
5    return gcd(y,x%y);
6  }
```

Het recursieve algoritme van Euclides is een zogenoemde staartrecursie of *tail recursion*. Dit betekent dat de recursieve aanroep van de functie de laatste actie van de functie is. Functies met staartrecursie kunnen altijd iteratief worden gemaakt. Code 7.5 en de iteratieve oplossing uit code 7.3 zijn feitelijk identiek.

Het nadeel van recursie is dat bij iedere stap een functieaanroep of *function call* nodig is. Dat maakt een recursieve oplossing meestal iets langzamer dan een overeenkomstige iteratieve oplossing. De meeste C-compilers herkennen staartrecursie en implementeren het iteratieve equivalent. Recursieve algoritmes zonder staartrecursie zijn meestal veel langzamer en hebben veel geheugen nodig.
Een recursieve implementatie in hardware is niet mogelijk. Vooraf is niet bekend hoe vaak de functie aangeroepen moet worden. Recursieve functies zijn niet synthetiseerbaar. De oplossing hiervoor is om de functie één keer te maken en deze



Figuur 7.3 : **Een grafische weergave van het algoritme van code 7.6.** Bij iedere stap wordt het kleinste getal van het grootste getal afgetrokken. Het deel dat van het grootste getal af gaat, is grijs gekleurd.

herhaald aan te roepen. De bewerkingen worden dan niet parallel maar sequentieel uitgevoerd en is het in feite een iteratieve oplossing geworden. Dit is ook de reden dat in de rest van dit hoofdstuk geen recursieve algoritmes aan de orde komen, maar alleen iteratieve algoritmes.

Algoritme met herhaald aftrekken

Tot nu toe is bij alle besproken algoritmes voor het bepalen van de grootste gemeenschappelijke deler minimaal één modulusfunctie nodig. Zoals eerder is gezegd, is de modulus niet synthetiseerbaar. Bij een implementatie in hardware is het interessant om een oplossing zonder modulus of deling te gebruiken.
De modulus van de twee getallen is de rest van de deling van deze getallen. Om deze te bepalen is een deling nodig en delen is weer herhaald aftrekken. In code 7.6 staat een variant, zonder modulus, die gebaseerd is op herhaald aftrekken. Bij ieder iteratie wordt het kleinste getal van het grootste getal afgetrokken. In figuur 7.4 zijn alle stappen voor het bepalen van de GGD met behulp van het algoritme uit code 7.6 voor de getallen 44 en 32 opgeschreven en in figuur 7.3 staat grafische weergave.
Voor willekeurige getallen is dit een gunstig algoritme. Er zijn meestal weinig iteraties nodig en de hoeveelheid logica voor een implementatie in hardware is minimaal. Alleen zijn er een aantal combinaties van getallen waarvoor erg veel iteraties nodig zijn. De meest extreme situaties treden op bij `gcd(n,n-1)` en bij `gcd(n,1)`. Er zijn dan `n-1` iteraties nodig.

Het rechtsschuivende binaire algoritme van Josef Stein

In 1961 heeft Josef Stein een binair algoritme ontwikkeld dat in 1967 is gepubliceerd. Dit algoritme gebruikt — net als code 7.6 — herhaald aftrekken. Stein

Code 7.6 : Een GGD-algoritme gebaseerd op herhaald aftrekken.

```
 1  int gcd(int x, int y)
 2  {
 3    while (x != y) {
 4      if ( x == 0 ) return y;
 5      if ( y == 0 ) return x;
 6
 7      if (x > y) {
 8        x = x - y;
 9      } else {
10        y = y - x;
11      }
12    }
13
14    return x;
15  }
```

```
 x   y
44  32    x > y  dus  x = 44 - 32 = 12
12  32    x < y  dus  y = 32 - 12 = 20
12  20    x < y  dus  y = 20 - 12 =  8
12   8    x > y  dus  x = 12 -  8 =  4
 4   8    x < y  dus  y =  8 -  4 =  4
 4   4    x = y  zodat  GGD = 4
```

Figuur 7.4 : Het algoritme van code 7.6 voor de getallen 44 en 32.

heeft er aan toegevoegd dat even getallen door twee gedeeld kunnen worden. Delen door twee is in hardware eenvoudig te realiseren door alle bits één positie naar rechts te schuiven.
Het algoritme van Stein gebruikt de volgende eigenschappen van de grootste gemeenschappelijke deler:

1. Als `x` en `y` beide even zijn, is: `gcd(x,y) = 2 gcd(x/2,y/2).`
2. Als `x` even is en `y` oneven is, is: `gcd(x,y) = gcd(x/2,y).`
3. Als `x` oneven is en `y` even is, is: `gcd(x,y) = gcd(x,y/2).`
4. Als `x` en `y` beide oneven zijn, is: `gcd(x,y) = gcd(x,(y-x)/2)` voor `x < y`
   `gcd(x,y) = gcd(y,(x-y)/2)` voor `x >= y`.

Donald Knuth meldt in het tweede deel van *The Art of Computer Programming* dat Stein in 1961 het algoritme heeft bedacht. Anderen zeggen dat Silver en Tersian in 1962 het algoritme hebben bedacht. In ieder geval is het door Stein als eerste gepubliceerd in *Journal of Computational Physics 1 (1967) 397-405*.

Het delen door twee is de schuifoperatie `>> 1` en vermenigvuldigen met twee is de schuifoperatie `<< 1`. Een getal is even als de minst significante bit nul is. In C wordt dit met de bitsgewijze-en gedetecteerd. Het algoritme van Stein gebruikt zodoende alleen schuifoperaties, aftrekfuncties en de bitsgewijze-en. Al deze functies zijn synthetiseerbaar of eenvoudig in hardware te beschrijven.

In code 7.7 staat het algoritme van Stein. De while-lus op regel 17 komt overeen met eigenschap 2: zolang `x` even is, wordt `x` door twee gedeeld. Op dezelfde manier komt eigenschap 3 overeen met de while-lus van regel 21. Het stuk code van regel 24 tot en met 31 is eigenschap 4. Door het omwisselen van `x` en `y` als `x` groter of gelijk is aan `y`, is `x` in de do-while-lus altijd oneven. Daarom kunnen de acties van eigenschap 2 voor de do-while staan.
Eigenschap 1 is in de code verdeeld over twee delen: het schuiven voor de berekening van `gcd(x/2,y/2)` staat in de while-lus op regel 10 en het vermenigvuldigen met 2 is de schuifactie bij de return op regel 35. Hoe vaak er naar links geschoven wordt, hangt af van het aantal keer `i` dat de while-lus van regel 10 doorlopen is.

De snelheid van het algoritme van Stein wordt bepaald door het aantal keer dat er geschoven moet worden. Dat hangt natuurlijk af van de waarden van `x` en `y`. Bij de eigenschappen 2, 3 en 4 schuift bij iedere iteratie hetzij `x` of `y`. Bij eigenschap 1 schuiven `x` en `y`. Het lijkt dat het algoritme hier twee keer zo snel convergeert, maar daar tegenover staat dat bij de return `x` weer naar links geschoven moet worden. Uiteindelijk komt het er op neer, dat voor iedere stap `x` of `y` schuift. Het

algoritme stopt als y nul is. Als n het maximaal aantal bits van x en y is, zijn er voor de meest ongunstige situatie $2n - 1$ stappen nodig.

Omdat het algoritme van Stein op het naar rechts schuiven gebaseerd is, wordt het — en alle varianten van dit algoritme — een rechtsschuivende binaire algoritme, *right-shift binary algorithm*, genoemd. Bij RSA-encryptie worden ook *left-shift binary algorithms* toegepast. De laatste jaren is er veel gepubliceerd over deze *left-shift binary algorithms*.

Code 7.7 : Het binaire algoritme van Stein.

```
1  int gcd(int x, int y)
2  {
3    int i, tmp;
4
5    if ( x == 0 ) return y;
6    if ( y == 0 ) return x;
7
8    // actions characteristic 1
9    i = 0;
10   while ( ((x & 1) == 0) && ((y & 1) == 0) ) {
11     x >>= 1;
12     y >>= 1;
13     i++;
14   }
15
16   // actions characteristic 2
17   while ((x & 1) == 0) x >>= 1;
18
19   do {
20     // actions characteristic 3
21     while ((y & 1) == 0) y >>= 1;
22
23     // actions characteristic 4
24     if (x < y) {
25       y -= x;
26     } else {
27       tmp = x - y;
28       x = y;
29       y = tmp;
30     }
31     y >>= 1;
32   } while (y != 0);
33
34   // actions characteristic 1
35   return x << i;
36 }
```

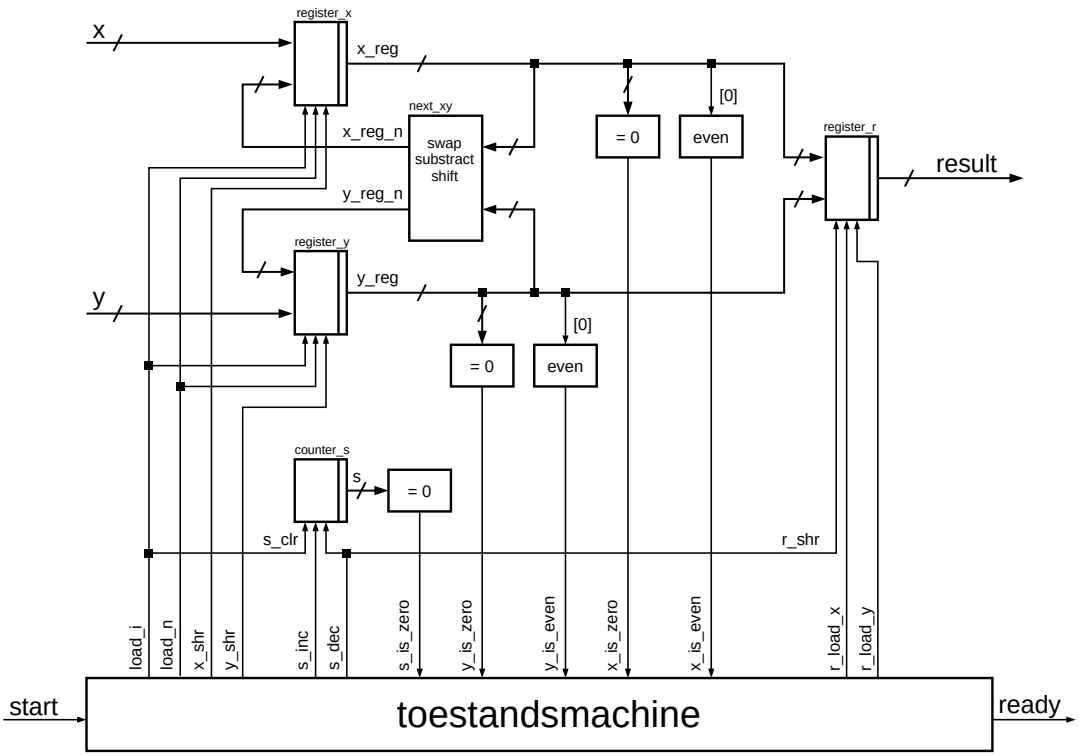
Code 7.8 : Right-shift binary algorithm gebaseerd op Stein.

```
1  int gcd(int x, int y)
2  {
3    int i = 0;
4    int tmp;
5
6    while ( 1 ) {
7      if ( x == 0 ) return y;
8      if ( y == 0 ) return x << i;
9
10     if ( ((x & 1) == 0) && ((y & 1) == 0) ) {
11       x >>= 1;
12       y >>= 1;
13       i++;
14     } else if ( (x & 1) == 0 ) {
15       x >>= 1;
16     } else if ( (y & 1) == 0 ) {
17       y >>= 1;
18     } else {
19       if ( x < y ) {
20         y = y - x;
21       } else {
22         tmp = x - y;
23         x = y;
24         y = tmp;
25       }
26       y >>= 1;
27     }
28
29   }
30 }
```

Een compacte versie van het algoritme van Stein

In code 7.8 staat een variant van Stein. Alle stappen zijn in een oneindige while-lus geplaatst. Het doel hiervan is het algoritme te vereenvoudigen. Er is maar één toestand, namelijk de while-lus, waarbij — afhankelijk van de status van x en y — steeds een andere actie uitgevoerd wordt.

Als x nul is stopt het algoritme en is y de grootste gemeenschappelijke deler. Als y nul is, geeft Stein op regel 6 x en op regel 35 x<<i terug. Code 7.8 geeft op regel 8 x<<i terug als y nul is.
De if-else-if op regel 10 bevat de vier eigenschappen van de methode van Stein. Zowel eigenschap 1 als eigenschap 2 staan nu ook in de while-lus.



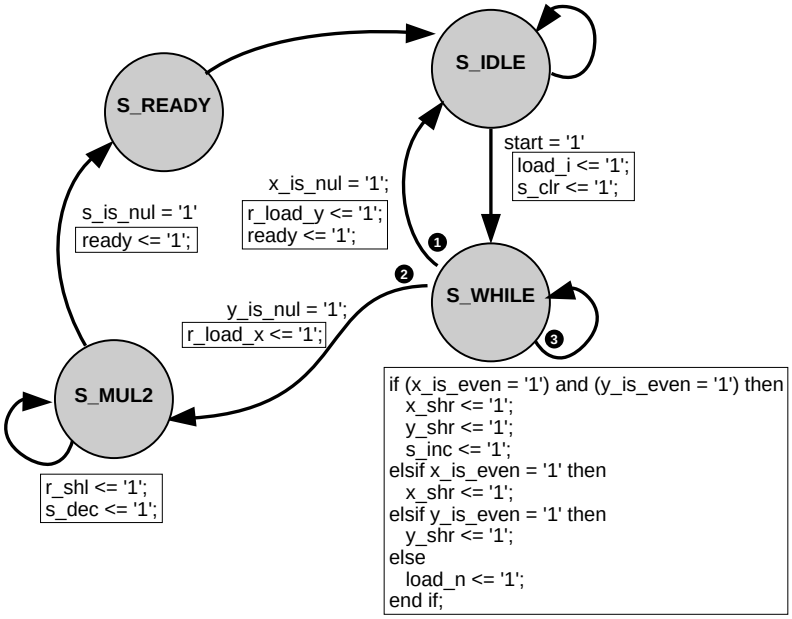
Figuur 7.5 : Het dataverwerkingsdeel voor het algoritme van code 7.8.

7.2 De implementatie van de GGD in hardware

De ontwerpmethode met een gescheiden dataverwerking en besturing legt eerst het dataverwerkingsdeel vast. Uit code 7.8 volgt dat er schuifregisters nodig zijn voor de getallen x en y waarvoor de grootste gemeenschappelijke deler moet worden uitgerekend. Verder is er een teller nodig voor i. De uitgang kan direct met een multiplexer op de schuifregisters worden aangesloten of er kan een uitgangsregister gebruikt worden. In het dataverwerkingsdeel van figuur 7.5 is ervoor gekozen om een uitgangsregister te gebruiken.
Er zijn vijf combinatorische blokken die detecteren of getallen nul zijn of dat ze even zijn. Het blok next_xy bevat de *swap-substract-shift* -berekening van regel 19 tot en met 26 uit code 7.8.
Het dataverwerkingsdeel is zowel geschikt voor het algoritme van Stein uit code 7.8 als voor het algoritme van code 7.7. De toestandsmachine heeft om de bewerking te starten een ingangssignaal start en signaleert met het uitgangssignaal ready dat de bewerking klaar is. Een Moore-implementatie van het algoritme van Stein staat in figuur 7.8. Het is een complexe toestandsmachine met zestien toestanden. Dit grote aantal toestanden bij een Moore-machine is het gevolg van het feit dat

De teller heet in het softwarealgoritme i. In figuur 7.5 is deze naam veranderd in s.
Signaal s is het aantal posities dat er naar rechts geschoven moet worden.

De namen van de ingangen zijn x en y. Intern zijn dat de geregistreerde signalen x_reg en y_reg.



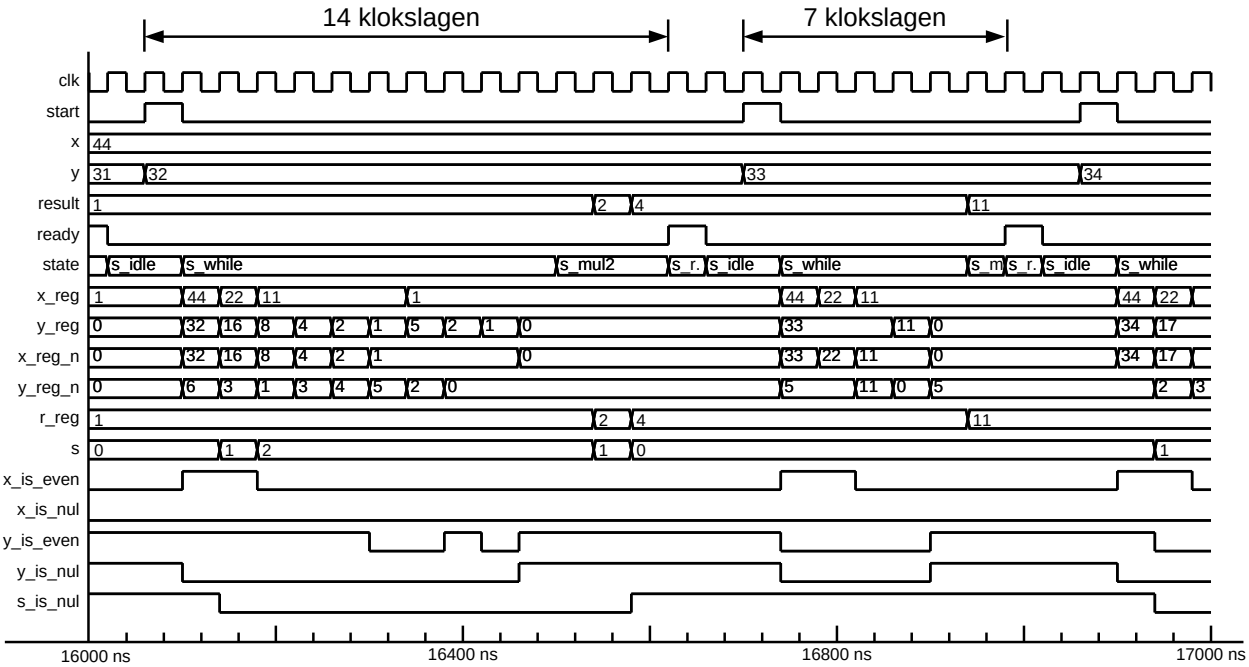
Figuur 7.6 : Een Mealy-implementatie van de toestandsmachine voor het algoritme van Stein.

iedere actie zijn eigen toestand heeft. Bovendien zijn er extra toestanden nodig om eerst het laatste resultaat te verwerken alvorens een beslissing te nemen wat de volgende actie is. Tenslotte geeft de letterlijke interpretatie van het algoritme van Stein ook extra toestanden.
Voor een Mealy-implementatie van code 7.8 zijn minder toestanden nodig. Dit is typisch een voorbeeld van een probleem waarbij overwogen moet worden om een Mealy-machine te maken. In figuur 7.6 staat een Mealy-implementatie met vier toestanden.

Toestand `S_while` komt overeen met de while-lus uit code 7.8. Als `x_reg` laag is stopt het algoritme en wordt `y_reg` in het uitgangsregister gezet. Als `y_reg` laag is stopt het algoritme eveneens. Signaal `x_reg` wordt in het uitgangsregister gezet en de volgende toestand wordt nu `S_mul2`, waarin `x_reg s` keer met twee wordt vermenigvuldigd. In alle andere gevallen wordt één van de vier bewerkingen uit het algoritme van Stein uitgevoerd.

Bij deze Mealy-oplossing is ervoor gekozen om het uitgangssignaal `ready` van het type Moore te maken. Daarvoor is de extra toestand `S_ready` nodig. Dit is gedaan om te garanderen dat signaal `ready` één hele klokperiode hoog is.

In bijlage D staat de complete VHDL-beschrijving voor een realisatie van de GGD met de dataverwerking uit figuur 7.5 en de Mealy-machine uit figuur 7.6. Figuur 7.7 geeft het simulatieresultaat van deze realisatie voor de grootste gemeenschappelijke delers van 44 en 32 en die van 44 en 33. Voor deze berekeningen zijn respectievelijk 14 en 7 klokslagen nodig. Voor 6-bits getallen is 14 de meest ongunstige situatie. Als n het aantal bits van de signalen is, is het maximaal aantal klokslagen gelijk aan $2n+2$. Naast de $2n-1$ stappen, die het algoritme maximaal nodig heeft, zijn er drie stappen nodig voor het binnenhalen van de getallen, en het weergeven van het resultaat.



Figuur 7.7 : Het simulatieresultaat van de Mealy-implementatie voor `gcd(44,32)` en `gcd(44,33)`. De GGD van 44 en 32 is 4. Nadat signaal `start` hoog is gemaakt, duurt het 14 klokslagen voordat `ready` hoog is. De GGD van 44 en 33 is 11. Voor deze berekening zijn 7 klokslagen nodig.

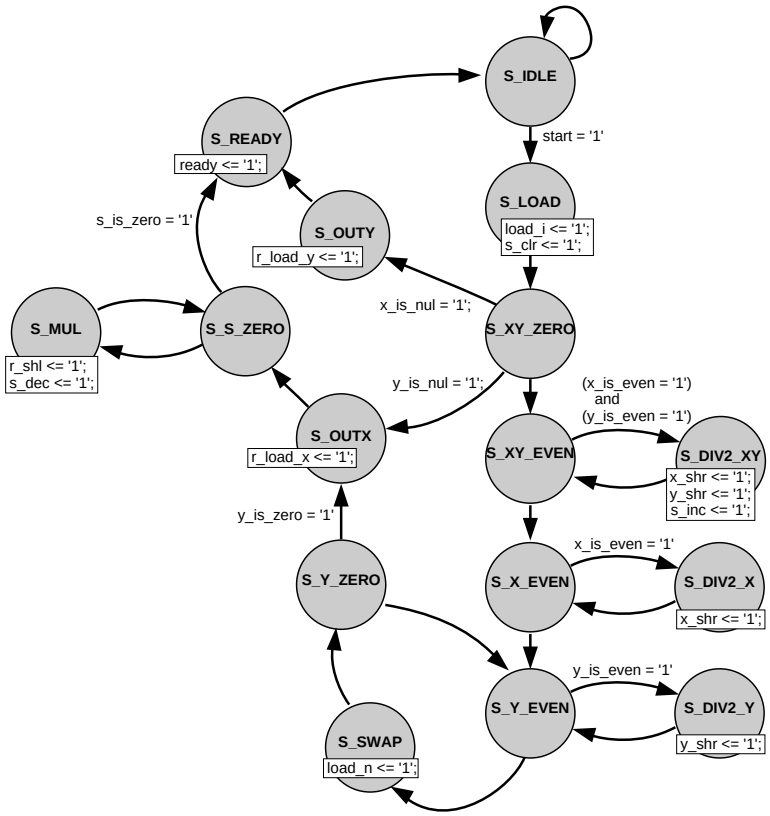
Als voor de toestandsmachine de Moore-machine van figuur 7.8 in plaats van de Mealy-machine gebruikt wordt, zijn $5n + 5$ klokslagen nodig. De versie met de Mealy-machine is ongeveer 2,5 zo snel als die met de Moore-machine. Dit rechtvaardigt de keuze voor de Mealy-machine.

De implementatie met de FSMD-methode

In bijlage D staat ook een deel van de implementatie met behulp van de FSMD-methode. De body van de architectuur staat in code D.8.

De complete code van de FSMD-beschrijving is fors kleiner dan die bij de methode met gescheiden dataverwerking en besturing: 103 regels tegenover 218 regels. In beide gevallen is dat veel meer dan de 29 regels van de C-functie uit code 7.8. Bij de beschrijving in VHDL zijn extra regels nodig voor de entity en de extra toestanden `S_idle` en `S_ready`. Bovendien is VHDL minder beknopt dan C. Bij de methode met een gescheiden dataverwerking en besturing zijn meer declaraties voor interne signalen en wordt de dataverwerking met alle registers en combinatorische functies expliciet opgeschreven. De volledige dataverwerking staat in code D.3, code D.4 en code D.5.

Hoewel de FSMD-beschrijving korter is, hoeft dat niet te betekenen dat het syntheseresultaat sneller of kleiner is. Voor de grootte van het ontwerp zal het niet veel uitmaken. In alle gevallen zijn er twee schuifregisters, een teller en is er logica nodig voor de beslissingen die de toestandsmachine maakt.



Figuur 7.8 : **Een Moore-versie van de toestandsmachine voor het algoritme van Stein.**

Er zijn vier aspecten waarop de gekozen implementatie mogelijk invloed heeft:

- **De snelheid of het maximaal aantal klokslagen**
  De beide methoden zijn functioneel identiek. Het aantal klokslagen is exact gelijk en de snelheid van de gerealiseerde systemen zijn ook hetzelfde.
- **De simulatieduur**
  Omdat de FSMD-methode minder interne signalen heeft en dus minder complex is, zal deze sneller simuleren. Bij dit voorbeeld is dat ongeveer een factor twee.
- **De grootte van het syntheseresultaat**
  De tabellen D.1, D.2 en D.3 uit bijlage D laten zien dat de grootte van het resultaat afhangt van de gebruikte synthesizer en de gekozen FPGA. In een aantal gevallen levert de FSMD-methode zelfs een groter netwerk op.
- **De maximaal haalbare kloksnelheid**
  De maximaal haalbare kloksnelheid hangt naast de keuze van de synthesizer en de FPGA ook af van het aantal bits dat nodig is. Voor een groot aantal bits is de FSMD-methode gunstig. Voor een klein aantal bits is de methode met een gescheiden dataverwerking en besturing voordeliger.

De keuze van de gebruikte methodiek beïnvloedt het resultaat slechts in geringe mate en is niet altijd gunstiger voor de FSMD-methode. Het voordeel van de FSMD-methode is dat de VHDL-beschrijving meer op de beschrijving in C lijkt. Het voordeel van de methodiek met gescheiden dataverwerking en besturing is dat het een verdeel-en-heersstrategie is en dat de dataverwerking zichtbaar is.

7.3 Adviezen voor de zoektocht naar een bruikbaar algoritme

Paragraaf 7.1 en 7.2 geven stap voor stap de zoektocht naar een geschikt algoritme voor de berekening van de grootste gemeenschappelijke deler. Deze zoektocht is samen te vatten in een aantal adviezen:

- **Denk eerst zelf na.**
  Voor veel problemen bestaat een intuïtieve oplossing. Meestal zal dit niet de beste oplossing geven, maar het geeft wel inzicht in de problematiek. Het leidt tot een pakket van eisen waar de oplossing aan moet voldoen. Bovendien kan er al een testomgeving gemaakt worden en dat is eenvoudiger voor een probleem dat goed begrepen wordt.
- **Onderzoek wat anderen er over hebben gepubliceerd.**
  De meeste problemen zijn niet nieuw. Veel mensen hebben er eerder al over nagedacht. Het algoritme van Euclides is rond 300 voor Christus opgeschreven en is waarschijnlijk nog veel ouder.
- **Denk er aan dat het in hardware gerealiseerd moet worden.**
  Niet ieder oplossing is geschikt voor een digitaal systeem. Herken de mogelijke syntheseproblemen die in sommige algoritmes zitten.
- **Herdefinieer eventueel de eisen.**
  Nadat gevonden was dat de methode van Euclides minder geschikt was, is de zoektocht herstart met de eis dat er geen modulus gebruikt mocht worden. Er is opnieuw een eenvoudige, bijna intuïtieve, methode met herhaald aftrekken onderzocht. Voortschrijdend inzicht is een goede eigenschap.
- **Speel met de gevonden algoritmes.**
  Test de algoritmes grondig en vergelijk verschillende alternatieve mogelijkheden. Herschrijf eventueel het algoritme. Het algoritme van code 7.8 leidt tot een andere implementatie dan het oorspronkelijk algoritme van Stein uit code 7.7.
- **Kies een algoritme dat geschikt is.**
  Een algoritme dat voldoet aan de eisen is prima. Het zoeken van het allerbeste algoritme is zeer tijdrovend en vaak overbodig.
- **Begrijp het te implementeren algoritme volledig.**
  Een algoritme dat niet goed of maar gedeeltelijk begrepen wordt, kan bijna niet foutloos worden geïmplementeerd.
- **Maak een VHDL-beschrijving die dicht bij het gekozen algoritme ligt.**
  Probeer bij het maken van een VHDL-beschrijving zo dicht mogelijk bij het oorspronkelijke algoritme te blijven. Pas het algoritme ook aan en test het opnieuw.
- **Maak een testbench die flexibel is en die veel aspecten test.**
  Volledige tests zijn niet mogelijk, maar test in ieder geval alle belangrijke situatie, zoals voor `gcd(0,0)`, `gcd(x,0)`, `gcd(0,x)` en vooral de meest ongunstige situaties. Gebruik `textio` om extra informatie af te drukken, bijvoorbeeld het aantal klokslagen dat nodig is.

De zoektocht naar een bruikbaar algoritme is geen vast omschreven reis. Elk probleem heeft zijn specifieke aspecten. Er zijn meer of minder oplossingen voorhanden of de oplossingen zijn moeilijker of eenvoudiger naar hardware om te zetten. Bovenstaande adviezen ondersteunen de reiziger bij zijn of haar ontdekkingstocht.

8

Algoritmes voor BCD

Doelstelling	Dit hoofdstuk bespreekt verschillende algoritmes voor bewerkingen met BCD-getallen en sluit daarom goed aan bij het hoofdstuk over algoritmes.
Onderwerpen	De behandelde onderwerpen zijn: ▪ BCD-getallen. ▪ Het optellen van twee BCD-getallen. ▪ Het ophogen van een BCD-getal met één. ▪ Een 4-digit BCD-teller en een n-digit BCD-teller. ▪ De conversie van binair naar BCD. ▪ De toepassing van functies en het overloaden van functies. ▪ Het gebruik van *unconstrained* vectoren. ▪ Het gebruik van het generate-statement en het gebruik van for-lussen voor het maken van regelmatige structuren. ▪ Het assert-statement om fouten af te vangen. ▪ Een toepassing van de functies `log` en `ceil` uit het `math_real`-package voor het berekenen van het aantal BCD-cijfers. ▪ De voor- en nadelen van seriële en parallelle oplossingen voor het beschrijven van algoritmes.

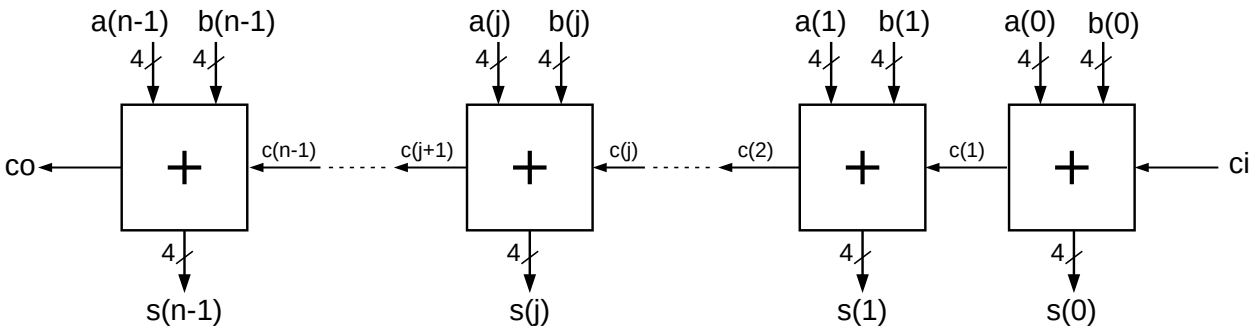
In hoofdstuk 7 is — als voorbeeld van een algoritme — de grootste gemeenschappelijke deler onderzocht. Dit hoofdstuk onderzoekt de algoritmes die nodig zijn voor het ontwikkelen van componenten en functies om met BCD-getallen te rekenen. BCD staat voor *binary coded decimal*, dit betekent dat de cijfers van getal decimaal gecodeerd worden met vier bits. Alle tien decimale cijfers `0` tot en met `9` worden gecodeerd met `0000` tot en met `1001`. De binaire waarden `1010` tot en met `1111` worden bij BCD niet gebruikt.

Het grote voordeel van BCD-getallen is dat het een tientallig stelsel is. Daartegenover staat dat bewerkingen met BCD-getallen complexer zijn en dat er voor deze getallen meer bits nodig zijn. In paragraaf 6.2 en in paragraaf 6.4 is er bij het ontwerp van de elektronische weegschaal voor gekozen om de pulsen op een BCD-manier te tellen. Deze keuze vereenvoudigde de aansturing van vier 7-segmentsdisplays.

8.1 De BCD-opteller

Deze paragraaf bespreekt de BCD-opteller. Andere rekenkundige bewerkingen met BCD-getallen, zoals aftrekken, vermenigvuldigen en delen worden in dit boek niet besproken. Deze functies komen alleen in zeer specifieke situaties voor, zoals een rekenmodule die gebruik maakt van BCD-getallen. De volgende paragrafen behandelen twee onderwerpen, die in de praktijk wel regelmatig toegepast worden, namelijk de BCD-teller en het algoritme voor de conversie van binaire getallen naar BCD-getallen.

Het optellen van BCD-getallen verschilt niet principieel met het optellen van binaire getallen. Paragraaf 4.8 over de herhalingsopdrachten geeft verschillende beschrijving voor 4- en n-bits optellers. Deze optellers zijn opgebouwd uit meerdere full-adders, die de som uit de twee binaire waarden en een carry-in-signaal bepalen. Op dezelfde wijze bestaat een BCD-opteller uit een verzameling optelfuncties, die de som van twee 1-digit BCD-getallen en een carry-in berekent. Figuur 8.1 geeft het schema voor een n-bits BCD-opteller. Dit schema komt bijna helemaal overeen met het schema van de n-bits binaire opteller uit figuur 4.33, alleen heeft de 1-digit BCD-opteller twee 4-bits ingangen en een 4-bits uitgang.



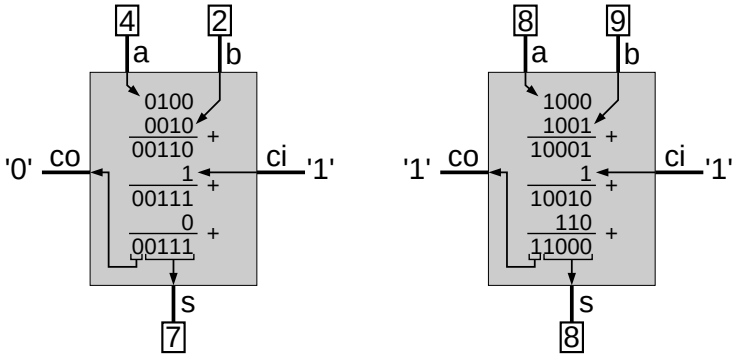
Figuur 8.1 : **Schema voor een BCD-opteller met n cijfers.** De opteller is opgebouwd uit een rij van 1-digit BCD-optellers.

Voor de realisatie van de BCD-opteller met n cijfers is een 1-digit BCD-opteller nodig met een carry-in en een carry-out.

Een 1-digit BCD-opteller

Het optellen van twee BCD-cijfers kan gerealiseerd worden door de waarden binair op te tellen en de uitkomst te corrigeren als deze groter is dan 9. Twee 4-bits BCD-cijfers en een carry-in zijn samen nooit groter dan 19.

In figuur 8.2 staan twee voorbeelden. De som van 4 en 2 is 6, met een carry-in wordt dat 7. Dit is kleiner of gelijk aan 9, zodat dit ook de uitkomst is. De som van 8 en 9 is 17, met een carry-in wordt dat 18. Binair is dat 10010. Dit is groter dan 9 en daarom wordt er 6, oftewel 110 binair, extra bijgeteld. Samen geeft dat 11000 binair, zodat de carry-out hoog is en de uitkomst 8 is.



Figuur 8.2 : Het optellen van twee 1-digit BCD-getallen. Links staat een voorbeeld zonder correctie en rechts een voorbeeld met correctie.

Het algoritme voor het optellen van twee BCD-cijfers `a` en `b` en een carry-in `ci` luidt in pseudocode:

```
som := a + b + ci;
if som > 9 then
  som := som + 6;
end if;
```

Na afloop bevat de meest significante bit van hulpvariabele `som` de carry-out `co` en bevatten de andere vier bits de som `s`.

In code 8.1 is het algoritme als functie geïmplementeerd. Een functie heeft alleen ingangsparameters en geen uitgangsparameters. Er is alleen een retourwaarde. De functie `bcd_add` geeft een 5-bits vector `som` terug.
De signalen `a` en `b` zijn van het type `std_logic_vector` en de variabele `som` is van het type `unsigned`, zodat de binaire optelfunctie uit het package `numeric_std` gebruikt kan worden. Via typecasting worden `a` en `b` omgezet naar `unsigned`. Signaal `ci` is een 1-bits signaal. Door aan deze bit — met behulp van het concatenatie-teken — een 4-bits vector `0000` te plakken wordt daar een 5-bits vector van gemaakt. Daarmee is de rechterzijde van de toekenning 5-bits en hoeft de lengte van `a` en `b` niet aangepast te worden.

Code 8.1 : De 1-digit opteller als functie.

```
1  function bcd_add(a,b : in std_logic_vector(3 downto 0); ci : in std_logic) return std_logic_vector is
2    variable som : unsigned (4 downto 0);
3  begin
4    som := unsigned(a) + unsigned(b) + ("0000" & ci);
5    if som > 9 then
6      som := som + 6;
7    end if;
8    return std_logic_vector(som);
9  end function bcd_add;
```

Een n-digit BCD-opteller

Met behulp van de functie `bcd_add` kan een n-digit BCD-opteller gemaakt worden. De configuratie van figuur 8.1 is hier niet helemaal geschikt voor. De functie heeft immers één 5-bits uitgang in plaats van twee aparte uitgangen voor de som en de

carry-out. In het schema zijn alle cijfers apart genummerd. Het ligt voor de hand om een eigen type te definiëren voor de 4-bits BCD-cijfers:

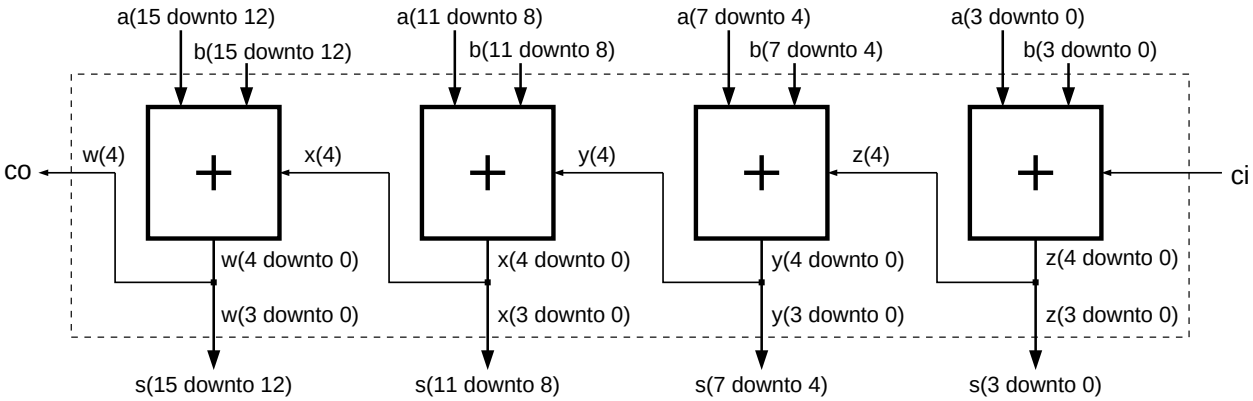
```
type bcd is array ( natural range <> ) of std_logic_vector(3 downto 0);
signal x, y : bcd(3 downto 0);
```

De signalen x en y zijn dan zestien bits breed. Het nadeel is dat het een nieuw type is en dat er weer typecasting nodig is bij de aansluitingen van normale vectoren. Dit is vooral handig bij het ontwikkelen van een BCD-bibliotheek en ook bij het schrijven van een testbench voor een BCD-component.

Code 8.2 : Een 4-digit BCD-opteller met behulp van de functie `bcd_add`.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity bcd4adder is
6    port (
7      a,b : in  std_logic_vector(15 downto 0);
8      ci  : in  std_logic;
9      s   : out std_logic_vector(15 downto 0);
10     co  : out std_logic
11   );
12 end entity bcd4adder;
13
14 architecture gedrag of bcd4adder is
15   function bcd_add(a, b : std_logic_vector(3 downto 0); ci : std_logic) return std_logic_vector is
16     variable som : unsigned (4 downto 0);
17   begin
18     som := unsigned(a) + unsigned(b) + ("0000" & ci);
19     if som > 9 then
20       som := som + 6;
21     end if;
22     return std_logic_vector(som);
23   end function bcd_add;
24
25   function bcd4add (a, b : std_logic_vector(15 downto 0); ci: std_logic) return std_logic_vector is
26     variable z,y,x,w : std_logic_vector(4 downto 0);
27   begin
28     z := bcd_add(a(3 downto 0),b(3 downto 0),ci);
29     y := bcd_add(a(7 downto 4),b(7 downto 4),z(4));
30     x := bcd_add(a(11 downto 8),b(11 downto 8),y(4));
31     w := bcd_add(a(15 downto 12),b(15 downto 12),x(4));
32     return (w(4 downto 0) & x(3 downto 0) & y(3 downto 0) & z(3 downto 0));
33   end bcd4add;
34
35   signal som : std_logic_vector(16 downto 0);
36 begin
37    som <= bcd4add(a,b,ci);
38    s   <= som(15 downto 0);
39    co  <= som(16);
40 end architecture gedrag;
```

Een andere aanpak is om signalen van het type `std_logic_vector` te gebruiken In figuur 8.3 staat een schema van een 4-digit BCD-opteller waarbij de signalen `a`, `b` en `s` 16-bits vectoren zijn.



Figuur 8.3 : **Schema voor een 4-digit BCD-opteller.** De ingangssignalen `a`, `b` en het uitgangssignaal `s` zijn 16-bits vectoren, die ieder een 4-digit BCD-getal representeren.

In code 8.2 staat de beschrijving van de 4-digit BCD-opteller. In dit geval is ervoor gekozen om een aparte functie `bcd4add` te maken. Deze functie roept vier keer de functie `bcd_add` aan, die een 1-digit BCD-opteller beschrijft. Er zijn vier 5-bits hulpvariabelen `w`, `x`, `y` en `z` nodig om de functies met elkaar en met uitgang `s` te verbinden. De functie retourneert een 17-bits vector, namelijk de som `s`, die overeenkomt met de vier minst significante vier bits van `w`, `x`, `y` en `z`, en de carry-out `co`, die overeenkomt met de carry-out `w(4)` van het laatste blok.

Het is interessanter om een algemene n-digit BCD-opteller te maken. De lengte van de ingangssignalen `a` en `b` en het uitgangssignaal `s` hangt dan af van het aantal cijfers dat opgeteld moet worden. Er zijn twee manieren om een entity te maken voor signalen met een variabele lengte, namelijk met een generieke parameter of door middel van een *unconstrained* vector. De generieke parameter is al eerder toegepast bij het n-bit register uit code 2.19 en de n-bits binaire opteller uit code 4.34.

Omdat een vector in feite array van `std_logic` is, spreekt men in dit verband ook over een unconstrained array.
Het Engelse woord unconstrained betekent hier: onbeperkt, onbegrensd, zonder beperking of zonder grenzen.

Code 8.3 geeft de entity voor de n-digit BCD-opteller met een generieke parameter `n`, die het aantal cijfers van de opteller definieert. De standaardwaarde van deze generieke parameter is 8. De lengte van de ingangssignalen `a` en `b` en het uitgangssignaal `s` is 4 maal het aantal cijfers. Als er bij de aanroep geen waarde voor de parameter `n` wordt meegegeven, wordt de standaardwaarde gebruikt en zijn de signalen 32-bits.

In code 8.4 hebben de vectoren `a`, `b` en `s` geen afmeting. Bij de aanroep van de entity wordt het aantal BCD-cijfers berekend uit de lengte van de vectoren die aangesloten zijn op `a` of `b`. De functie `max` bepaalt wat de grootste lengte is. Deze lengte wordt met één verlaagd, door vier gedeeld en met één verhoogd. Deze berekening zorgt ervoor dat er altijd voldoende cijfers zijn om de bits van de BCD-getallen, die op `a` en `b` zijn aangesloten, kwijt te kunnen. Voor een vectorlengte van 16, 15, 14 en 13 is het aantal cijfers 4 en voor een vectorlengte van 12, 11, 10 en 9 is bijvoorbeeld het aantal cijfers 3.

De declaraties van de constante `n` en van de functie `max` staan in het declaratiegedeelte van de entity. Dit heeft geen dwingende reden, anders dan dat `n` nauw gerelateerd is aan de ingangssignalen.

Code 8.3 : De entity voor een n-digit BCD-opteller met behulp van een generieke parameter.

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity bcd_n_adder is
 6   generic (
 7     n   : natural := 8
 8   );
 9   port (
10     a,b : in  std_logic_vector(4*n-1 downto 0);
11     ci  : in  std_logic;
12     s   : out std_logic_vector(4*n-1 downto 0);
13     co  : out std_logic
14   );
15 end entity bcd_n_adder;
```

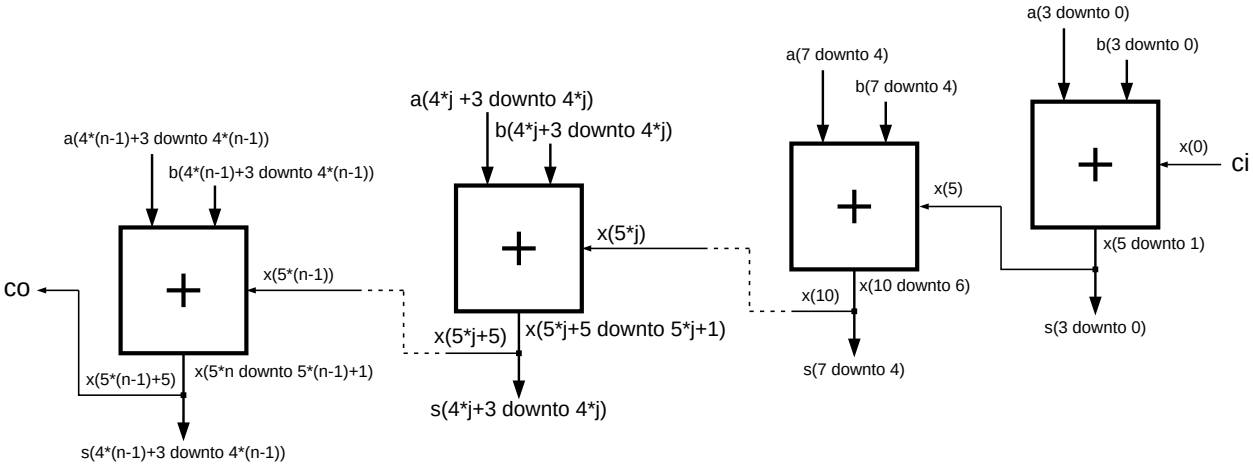
Code 8.4 : De entity voor een n-digit BCD-opteller met unconstrained vectoren.

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity bcd_n_adder is
 6   port (
 7     a,b : in  std_logic_vector;
 8     ci  : in  std_logic;
 9     s   : out std_logic_vector;
10     co  : out std_logic
11   );
12
13   function max (x, y: integer) return integer is
14   begin
15     if x > y then return x;
16     else return y;
17     end if;
18   end function max;
19
20   constant n : natural :=
21                (max(a'length,b'length)-1)/4+1;
22 end entity bcd_n_adder;
```

De 4-digit BCD-opteller uit figuur 8.3 gebruikt vier interne 5-bits hulpvariabelen. Voor een opteller van `n` cijfers zijn `n` hulpvariabelen nodig, of `n` interne signalen — als het over een structuurbeschrijving gaat. Een optie is om een array te gebruiken. Maak een eigen datatype voor het 5-bits signaal en definieer een n-bits array van dat type.

```
type connect_type is array (natural range <>) of std_logic_vector(4 downto 0);
signal x : connect_type(n-1 downto 0);
```

Een alternatief is om een intern signaal te declareren van `5*n+1` bits, namelijk: vijf bits voor ieder cijfer en één extra voor de carry-out. In figuur 8.4 staat het schema voor een n-digit BCD-opteller met een intern signaal `x`. In de figuur zijn de optellers voor cijfers `0`, `1`, `j` en `n-1` getekend. Bij het j^e cijfer zijn alle indices van de signalen uitgedrukt in `j`. Bij de cijfers `0`, `1` en `n-1` zijn de indices in de specifieke waarde uitgedrukt. Deze kunnen in alle drie de gevallen gevonden worden door voor `j` respectievelijk `0`, `1` en `n-1` in te vullen.



Figuur 8.4 : **Schema voor een n-digit BCD-opteller.** De ingangssignalen `a`, `b` en het uitgangssignaal `s` zijn 16-bits vectoren, die ieder een 4-digit BCD-getal representeren.

Het is dus voldoende om één element met de algemene index `j` te beschrijven en deze `n` keer aan te roepen. Dat kan in een structuur met een generate-statement en in een functionele beschrijving met een herhalingslus. In code 8.5 staat de structuurbeschrijving met het generate-statement. De toewijzingen op regel 38 en 39 komen exact overeen met de signalen `x` en `s` van het j^e cijfer uit figuur 8.4. Deze twee toewijzingen staan in een generate-statement dat dit `n` keer reproduceert. De carry-in `ci` wordt toegekend aan `x(0)` en de carry-out `co` krijgt de waarde `x(5*n)`.

Code 8.5 : **De structuurbeschrijving voor een n-digit BCD-opteller.**

```
24  architecture gedrag of bcd_n_adder is
      ⋮  hier staat de functie bcd_add uit code 8.1
33
34    signal x : std_logic_vector(5*n downto 0);
35  begin
36    x(0) <= ci;
37    f: for j in n-1 downto 0 generate
38      x(5*j+5 downto 5*j+1) <= bcd_add(a(4*j+3 downto 4*j),b(4*j+3 downto 4*j),x(5*j));
39      s(4*j+3 downto 4*j)   <= x(5*j+4 downto 5*j+1);
40    end generate f;
41    co <= x(5*n);
42  end architecture gedrag;
```

Bij code 8.5 is er voor gekozen om de opteller als een entity met een structuurbeschrijving te realiseren. In code 8.6 is met hetzelfde algoritme een functie `bcd_n_add` gedefinieerd. Het generate-statement is vervangen door een for-lus. De lusvariabele is hier van `0` naar `n-1` gedefinieerd. Cijfer `0` moet als eerste worden doorgerekend, net zoals in figuur 8.4 de signalen ook van rechts naar links gaan. Bij code 8.5 doet de volgorde er niet toe. Alle aanroepen van de functie `bcd_add` worden daar parallel uitgevoerd.

Code 8.6 : De n-digit BCD-opteller als functie.

```
1   function bcd_n_add (a,b : in std_logic_vector; ci: std_logic) return std_logic_vector is
2     constant n : natural := (a'length-1)/4+1;
3     variable x : std_logic_vector(5*n downto 0);
4     variable s : std_logic_vector(4*n-1 downto 0);
5   begin
6     x(0) := ci;
7     for j in 0 to n-1 loop
8       x(5*j+5 downto 5*j+1) := bcd_add(a(4*j+3 downto 4*j), b(4*j+3 downto 4*j), x(5*j));
9       s(4*j+3 downto 4*j)   := x(5*j+4 downto 5*j+1);
10    end loop;
11
12    return x(5*n) & s;
13  end bcd_n_add;
```

Bij de definitie van de functie + staat + tussen dubbele aanhalingstekens. Dit betekent dat bij deze functie de infix-notatie gebruikt wordt.

```
z <= x + y;
```

VHDL ondersteunt voor een beperkt aantal gevallen *overloading*. Standaardfuncties, zoals `+`, `-`, **`and`** en **`or`**, mogen opnieuw worden gedefinieerd. De functie **`and`** bestaat standaard voor de typen `bit` en `bit_vector`. In het package `std_logic_1164` is deze functie opnieuw gedefinieerd voor `std_logic` `std_logic_vector` en in het package `numeric_std` voor `unsigned` en `signed`. Als de functie geen carry-in heeft, kan de functie `bcd_n_adder` de standaardfunctie `+` overloaden.
De functie `"+"` uit code 8.7 is bijna identiek aan de functie `bcd_n_adder` uit code 8.6. De naam is alleen veranderd in `"+"`, de carry-in `ci` is uit de parameterlijst verwijderd en in plaats van `ci` wordt aan `x(0)` de waarde `'0'` toegekend.

Code 8.7 : Een 4-digit BCD-opteller met overloading

```
24  architecture gedrag of bcd4adder is
      ⋮  hier staat de functie bcd_add uit code 8.1
33    function "+" (a,b : in std_logic_vector) return std_logic_vector is
34      constant n : natural := (a'length-1)/4+1;
35      variable x : std_logic_vector(5*n downto 0);
36      variable s : std_logic_vector(4*n-1 downto 0);
37    begin
38      x(0) := '0';
39      for j in 0 to n-1 loop
40        x(5*j+5 downto 5*j+1) := bcd_add(a(4*j+3 downto 4*j), b(4*j+3 downto 4*j), x(5*j));
41        s(4*j+3 downto 4*j)   := x(5*j+4 downto 5*j+1);
42      end loop;
43
44      return x(5*n) & s;
45    end "+";
46
47    signal z : std_logic_vector(16 downto 0);
48  begin
49    z  <= a + b;
50    s  <= z(15 downto 0);
51    co <= z(16);
52  end architecture gedrag;
```

8.2 Werkwijze bij het ontwerp

In deze paragraaf is er voor gekozen om de 1-digit BCD-opteller als functie te beschrijven. Dit had evengoed met een proces, een procedure of een complete entity gedaan kunnen worden.
Iedere aanpak heeft zijn voor- en nadelen. Een beschrijving met een proces kent veel vrijheden, maar is niet echt modulair. Voor een bibliotheekfunctie is een proces niet geschikt. Een complete entity met een architectuur kan als component worden aangeroepen. Deze kan dan in een speciale bibliotheek staan of in de werkbibliotheek.
Het voordeel van procedures en functies is dat deze compact zijn. Een nadeel van procedures is dat de uitgangsparameters verschillend gedefinieerd worden voor variabelen en signalen. Bij een *concurrent procedure call* moeten de uitgangsparameters de modus `signal` hebben. Dezelfde procedure is dan niet geschikt om als gewone procedure met variabelen aan te roepen.
Een functie heeft alleen ingangsparameters en slechts één retourwaarde. Bij de functie `bcd_add` uit code 8.1 is dat opgelost door een 5-bits vector terug te geven, die bestaat uit de som en de carry-out.

Stappen voor het implementeren van een algoritme

Het schrijven van VHDL met generate-statements en for-lussen is bij signalen met veel indices vaak complex. Het helpt om dat stapsgewijs te doen. Verander niet alles tegelijk en maak duidelijke tekeningen. De werkwijze voor de modulair opgebouwde BCD-opteller is hier samengevat in een aantal stappen:

- Bepaal de essentiële bouwsteen en kies een geschikte vorm voor de implementatie: een procedure, een functie, een proces of een entity. In dit geval is de essentiële bouwsteen een 1-digit BCD-opteller en is de implementatievorm een functie.
- Zoek een algoritme dat het gedrag van de essentiële bouwsteen beschrijft.
- Beschrijf deze essentiële bouwsteen, maak een entity en architectuur, die deze bouwsteen aanroept, creëer hiervoor een testbench en test het geheel.
- Check of de beschrijving synthetiseerbaar is en kijk of de synthese het gewenste resultaat geeft.
- Maak een tekening van de 4-digit BCD-opteller met alle indices
- Beschrijf de 4-digit BCD-opteller, maak weer een testbench en controleer het simulatieresultaat.
- Maak een tekening van de n-digit BCD-opteller met alle indices.
- Beschrijf de n-digit BCD-opteller en test deze met de testbench van de 4-digit BCD-opteller.
- Maak een testbench om de n-digit BCD-opteller ook in andere situaties te testen, bijvoorbeeld als 1-digit en als 9-digit BCD-opteller.
- Check of de beschrijving van de BCD-optellers synthetiseerbaar is.

De focus van de voorafgaande paragraaf ligt op het algoritme en de opbouw van verschillende BCD-optellers. De testbenches, de simulatieresultaten en de resultaten van de synthese zijn achterwege gelaten.

8.3 Een BCD-getal met één verhogen

Veel programmeertalen gebruiken een aparte functie voor het ophogen met één en verlagen met één. De taal C gebruikt hiervoor `i++` en `i--`.

Het ontwerp van de elektronische personenweegschaal uit paragraaf 6.2 met behulp van de iteratieve softwarematige aanpak gebruikt een functie `incrementBCD`, die het aantal getelde pulsen met één ophoogt. In het Engels heet het ophogen van een getal met één *increment* en het verlagen met één *decrement*.

Het ophogen van een BCD-cijfer met één

De essentiële bouwsteen voor het maken van een functie, die een n-digit BCD-getal met één ophoogt, is een functie die een BCD-cijfer met één verhoogt. Het ophogen met één is hetzelfde als bij het betreffende getal 1 optellen. In principe kan daarom hetzelfde algoritme gebruikt worden als bij het optellen van twee BCD-cijfers gebruikt is. Toch is het ophogen van een BCD-cijfer met één wat eenvoudiger dan het optellen van twee BCD-cijfers. Bij het optellen wordt er eerst opgeteld en daarna gecorrigeerd. Bij het ophogen met één wordt er gecorrigeerd als het cijfer groter of gelijk is aan negen en anders wordt er één bij geteld. In pseudocode geeft dat:

```
if digit >= 9 then
   result := 10;
else
   result := digit + 1;
end if;
```

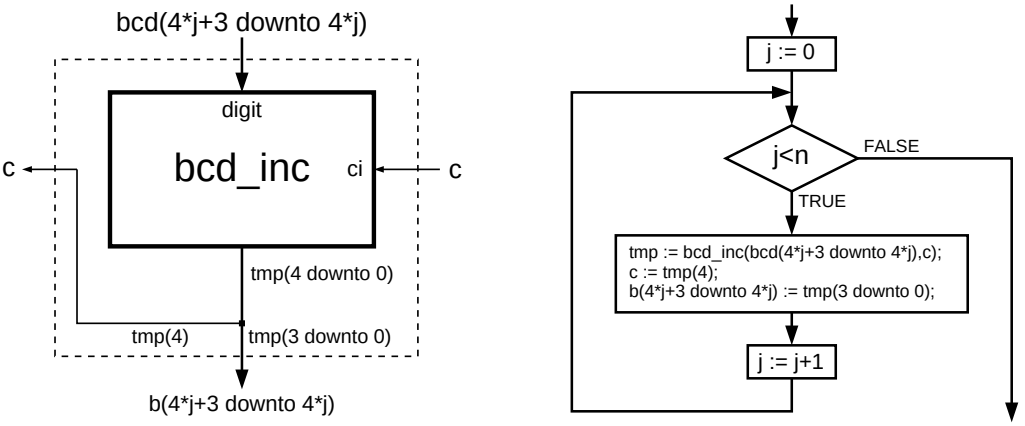
Het algoritme van de BCD-opteller bevat minimaal twee binaire optelfuncties. Als de BCD-opteller ook een carry-in heeft, zijn er zelfs drie binaire optelfuncties nodig. Het algoritme voor het ophogen heeft slechts één optelfunctie nodig.
Als het cijfer gelijk is aan 9, is het nieuwe cijfer `0` en de carry-out hoog. De carry-out is de carry-in van het volgende cijfer en als deze laag is, blijft dat cijfer en alle volgende cijfers ongewijzigd. De carry-in functioneert bij een 1-digit BCD-opteller als enable. De bewerking wordt alleen uitgevoerd als carry-in hoog is.

Code 8.8 : De functie `bcd_inc` die een BCD-cijfer met één ophoogt.

```
1  function bcd_inc (digit : in std_logic_vector(3 downto 0);
2                    ci    : in std_logic) return std_logic_vector is
3  begin
4    if ci = '1' then
5      if unsigned(digit) >= 9 then
6        return "10000";
7      else
8        return std_logic_vector(unsigned('0' & digit) + 1);
9      end if;
10   end if;
11
12   return '0' & digit;
13 end function bcd_inc;
```

In code 8.8 staat de functie `bcd_inc` die een BCD-cijfer met één ophoogt. De ingangsparameter `ci` is de carry-in. De carry-out is de meest significante bit van de 5-bits waarde, die de functie teruggeeft.

De functie `bcd_inc` wordt vervolgens gebruikt om een functie te maken die een n-digit BCD-getal met één ophoogt. Dat kan door de functie `bcd_inc` herhaald aan te roepen in een for-lus.



Figuur 8.5 : **De opbouw van de functie `incrementBCD`.** Links staat de aanroep van de functie `bcd_inc` voor het j^e cijfer. Rechts staat het stroomdiagram van de herhalingslus.

In figuur 8.5 is de aanroep van de functie voor het j^e cijfer weergegeven. De variable `bcd` is het getal dat opgehoogd wordt en `b` is de uitkomst. Omdat de functie `bcd_inc` vijf bits retourneert, is er een variabele `tmp` nodig. De carry-out is `tmp(4)` en dat is de carry-in voor het volgende cijfer. Het stroomdiagram uit figuur 8.5 beschrijft een herhalingslus, die de functie herhaald aanroept.

Code 8.9 : **De functie `incrementBCD` die een n-digit BCD-getal met één ophoogt.**

```
1  function incrementBCD(bcd : std_logic_vector) return std_logic_vector is
2    constant n   : natural := (bcd'length-1)/4+1;
3    variable b   : std_logic_vector(bcd'length-1 downto 0);
4    variable tmp : std_logic_vector(4 downto 0);
5    variable c   : std_logic;
6  begin
7    c := '1';
8    for j in 0 to n-1 loop
9      tmp := bcd_inc(bcd(4*j+3 downto 4*j),c);
10     c := tmp(4);
11     b(4*j+3 downto 4*j) := tmp(3 downto 0);
12   end loop;
13
14   return b;
15 end function incrementBCD;
```

Code 8.9 beschrijft de functie `incrementBCD`, die in code 6.3 van paragraaf 6.2 gebruikt is. In de functie is de herhalingslus van het stroomdiagram uit figuur 8.5 geïmplementeerd met behulp van een for-statement. De functie `incrementBCD` heeft geen carry-in en geen carry-out. De carry-in van het eerste cijfer is op regel 7 hoog gemaakt. Bij de return op regel 14 wordt alleen `b` teruggegeven. De carry-out `tmp(4)` van het laatste cijfer is niet toegevoegd aan de retourwaarde.

8.4 Een BCD-teller

In paragraaf 6.5 is bij het ontwerp van de elektronische personenweegschaal met behulp van de methodiek met gescheiden dataverwerking en besturing een 4-digit BCD-teller gebruikt. Een BCD-teller is net als een binaire teller een sequentiële schakeling met een kloksignaal en kan opgebouwd worden uit vier 1-digit BCD-tellers. In code 8.10 staat de entity van een 1-digit BCD-teller. Naast de 4-bits uitgang `q` en de carry-out `co` heeft deze entity een kloksignaal `clk` en een actief lage asynchrone reset `rst_n`.

Code 8.10 : De entity van een 1-digit BCD-teller.

```
 5  entity bcd_cnt is
 6    port (
 7      clk    : in  std_logic;
 8      rst_n  : in  std_logic;
 9      q      : out std_logic_vector(3 downto 0);
10      co     : out std_logic
11    );
12  end entity bcd_cnt;                          \Index{teller!1-digit BCD-}
```

De architectuur van de 1-digit BCD-teller staat in code 8.11. Deze beschrijving voldoet aan het sjabloon uit paragraaf4.4 van een sequentieel proces met synchrone reset. Het interne signaal `c` bevat het cijfer van de 1-digit BCD-opteller. Na de klokflank op regel 21 volgt het algoritme dat het BCD-cijfer met één ophoogt.

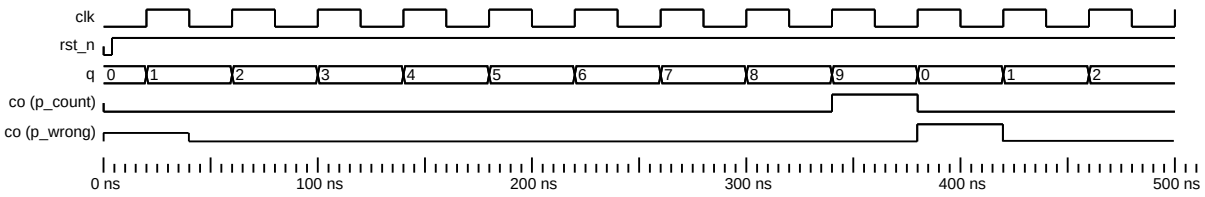
Code 8.11 : De architectuur van een 1-digit BCD-teller.

```
14  architecture gedrag of bcd_cnt is
15    signal c : unsigned(3 downto 0);
16  begin
17    p_count: process (clk, rst_n) is
18    begin
19      if rst_n = '0' then
20        c <= (others =>'0');
21      elsif rising_edge(clk) then
22        if c = 9 then
23          c <= (others =>'0');
24        else
25          c <= c + 1;
26        end if;
27      end if;
28    end process p_count;
29
30    q  <= std_logic_vector(c);
31    co <= '1' when c = 9 else '0';
32  end architecture gedrag;
33
```

Code 8.12 : Een foutieve 1-digit BCD-teller.

```
14  architecture gedrag of bcd_cnt is
15    signal c : unsigned(3 downto 0);
16  begin
17    p_wrong: process (clk, rst_n) is
18    begin
19      if rst_n = '0' then
20        c <= (others =>'0');
21      elsif rising_edge(clk) then
22        if c = 9 then
23          c  <= (others =>'0');
24          co <= '1';
25        else
26          c  <= c + 1;
27          co <= '0';
28        end if;
29      end if;
30    end process p_wrong;
31
32    q  <= std_logic_vector(c);
33  end architecture gedrag;
```

De carry-out staat buiten het proces `p_count` dat de teller beschrijft. Dan is `co` hoog als de waarde van c gelijk is aan 9. Code 8.12 geeft een incorrect alternatief. Omdat `c` een signaal is, krijgt dit signaal in proces `p_wrong` zijn nieuwe waarde nadat alle processen zijn geëvalueerd. Eén klokslag ná de klokflank waarbij signaal `c` de waarde 9 krijgt, is de test van regel 22 waar. Dat is bij de klokflank waarbij `c` weer



Figuur 8.6 : Het simulatieresultaat voor `bcd_cnt`. Signaal `co (p_count)` is de carry-out van de correcte beschrijving uit code 8.11. Signaal `co (p_wrong)` is de carry-out van de foutieve beschrijving uit code 8.12.

0 is geworden. Figuur 8.6 toont het signaaldiagram van de correcte beschrijving uit code 8.11 en de incorrecte beschrijving uit code 8.12.
De synthese van de incorrecte beschrijving geeft een netwerk met vijf flipfloppen, namelijk: vier voor de teller en één voor de carry-out. De correcte beschrijving geeft een schakeling met vier flipfloppen, zoals voor een 1-digit BCD-teller te verwachten is.

Code 8.13 : De 1-digit BCD-teller met enable en synchrone clear.

```
 5  entity bcd_cnt_c_e is
 6    port (
 7      clk    : in  std_logic;
 8      rst_n  : in  std_logic;
 9      clear  : in  std_logic;
10      enable : in  std_logic;
11      q      : out std_logic_vector(3 downto 0);
12      co     : out std_logic
13    );
14  end entity bcd_cnt_c_e;
15
16  architecture gedrag of bcd_cnt_c_e is
17    signal c : unsigned(3 downto 0);
18  begin
19    p_count : process (clk,rst_n) is
20    begin
21      if rst_n = '0' then
22        c <= (others =>'0');
23      elsif rising_edge(clk) then
24        if clear = '1' then
25          c <= (others =>'0');
26        elsif enable = '1' then
27          if c = 9 then
28            c <= (others =>'0');
29          else
30            c <= c + 1;
31          end if;
32        end if;
33      end if;
34    end process p_count;
35
36    q  <= std_logic_vector(c);
37    co <= '1' when c = 9 else '0';
38  end architecture gedrag;
```

De hier vóór beschreven 1-digit BCD-teller is niet geschikt voor een meercijferige teller. Op dezelfde manier, als op bladzijde 180 bij de beschrijving van de functie `bcd_cnt` is aangegeven, moet de bouwsteen een carry-in hebben. Om de 1-digit BCD-teller ook geschikt te maken om als BCD-teller bij de elektronische personenweegschaal te gebruiken, is bij de entity uit code 8.13 naast een enable ook een synchrone clear toegevoegd.

Code 8.14 : Een 4-digit BCD-teller met enable en synchrone clear.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity bcd4count_c_e is
5    port (
6      clk    : in  std_logic;
7      rst_n  : in  std_logic;
8      clear  : in  std_logic;
9      enable : in  std_logic;
10     q      : out std_logic_vector(15 downto 0);
11     co     : out std_logic
12   );
13 end entity bcd4count_c_e;
14
15 architecture gedrag of bcd4count_c_e is
16   component bcd_cnt_c_e is
17     port (
18       clk    : in  std_logic;
19       rst_n  : in  std_logic;
20       clear  : in  std_logic;
21       enable : in  std_logic;
22       q      : out std_logic_vector(3 downto 0);
23       co     : out std_logic
24     );
25   end component bcd_cnt_c_e;
26
27   for all : bcd_cnt_c_e use entity work.bcd_cnt_c_e(gedrag);
28
29   signal c : std_logic_vector(2 downto 0);
30 begin
31   b3:  bcd_cnt_c_e  port map (clk, rst_n, clear, c(2),   q(15 downto 12), co);
32   b2:  bcd_cnt_c_e  port map (clk, rst_n, clear, c(1),   q(11 downto 8),  c(2));
33   b1:  bcd_cnt_c_e  port map (clk, rst_n, clear, c(0),   q(7 downto 4),   c(1));
34   b0:  bcd_cnt_c_e  port map (clk, rst_n, clear, enable, q(3 downto 0),   c(0));
35 end architecture gedrag;
```

Met de 1-digit BCD-teller van code 8.13 kan een meercijferige teller worden gemaakt. In code 8.14 staat een 4-digit BCD-teller. De vier 1-digit BCD-tellers worden als component aangeroepen. De volgorde waarin de statements staan doet er in een parallelle omgeving niet toe. De component `b0` is het laatst genoemd en is in dit geval het minst significante cijfer. Aan de enable-ingang van deze component is het signaal `enable` aangesloten. De 3-bits van signaal `c` verbinden de carry-out van een component met de carry-in van de volgende component. De carry-out van het meest significante cijfer, ofwel component `b3`, is verbonden met de carry-out `co` van de 4-digit teller.

Deze 4-digit BCD-teller kan bij de beschrijving van de elektronische personenweegschaal uit paragraaf 6.5 proces `pulscounter` vervangen. De aanroep van de teller uit code 8.15 vervangt dan het proces `pulscounter` uit code 6.8. Alleen moet signaal `pc` niet van het type `unsigned`, maar van het type `std_logic_vector` zijn. Uitgang `co` wordt niet gebruikt en is daarom niet verbonden. Dit is aangegeven met het sleutelwoord **`open`**.

Code 8.15 : De aanroep van de 4-digit BCD-teller bij de elektronische personenweegschaal.

```
pulscounter : bcd4count_c_e port map (
  clk => clk,
  rst_n => rst_n,
  clear => clr,
  enable => puls_detect,
  q => pc,
  co => open
);
```

In code 8.17 zijn de vier componentinstantiaties uit code 8.14 vervangen door een generate-statement. De lusvariabele hangt af van het aantal cijfers `n`. Zo is het meteen een beschrijving van een n-digit BCD-teller geworden. Uitgang `q` is *unconstrained*. Het aantal cijfers is afgeleid uit de lengte van `q`. De lengte van het interne signaal `c` hangt af van `n`.

Bij deze aanpak moet `n` groter of gelijk aan twee zijn. Als `n` één is, is het bereik van signal `c` ongeldig. Bovendien worden er dan twee 1-digit optellers geïnstantieerd in plaats van één. Daarnaast moet het signaal dat aangesloten wordt aan uitgang `q` een veelvoud van vier zijn. Deze twee gebruiksvoorwaarden zijn in code 8.17 afgevangen in de entity met twee assert-statements.

In paragraaf 9.2 wordt op bladzijde 204 het assert-statement besproken.

De pulsteller van de elektronische personenweegschaal

Bij het ontwerpvoorbeeld met de elektronische personenweegschaal is in paragraaf 6.4 een BCD-teller gebruikt. In code 6.8 ontbreekt bij het proces `pulscounter` de code die het signaal `pc` met één ophoogt. In code 8.16 staat het ontbrekende deel. Het is een for-lus met daarin het algoritme voor het verhogen van een BCD-cijfer.

Code 8.16 : Het ontbrekende deel van proces `pulscounter` uit code 6.8.

```
90          c := '1';
91          for j in 0 to 3 loop
92            if c = '1' then
93              if pc(4*j+3 downto 4*j) = 9 then
94                pc(4*j+3 downto 4*j) <= (others =>'0');
95                c := '1';
96              else
97                pc(4*j+3 downto 4*j) <= pc(4*j+3 downto 4*j) + 1;
98                c := '0';
99              end if;
100           end if;
101         end loop;
```

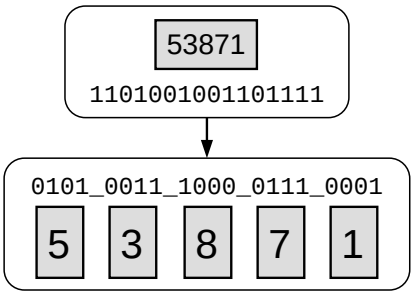
Code 8.17 : Een n-digit BCD-teller met behulp van het generate-statement.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity bcd_n_count_c_e is
5    port (
6      clk    : in  std_logic;
7      rst_n  : in  std_logic;
8      clear  : in  std_logic;
9      enable : in  std_logic;
10     q      : out std_logic_vector;
11     co     : out std_logic
12   );
13   constant n : natural := q'length/4;
14 begin
15   assert q'length mod 4 = 0 report "length of q must be a multiple of 4" severity failure;
16   assert n > 1              report "q must be larger than 4 bits" severity failure;
17 end bcd_n_count_c_e;
18
19 architecture gedrag of bcd_n_count_c_e is
20   component bcd_cnt_c_e is
21     port (
22       clk    : in  std_logic;
23       rst_n  : in  std_logic;
24       clear  : in  std_logic;
25       enable : in  std_logic;
26       q      : out std_logic_vector(3 downto 0);
27       co     : out std_logic
28     );
29   end component bcd_cnt_c_e;
30
31   for all : bcd_cnt_c_e use entity work.bcd_cnt_c_e(gedrag);
32
33   signal c : std_logic_vector(n-2 downto 0);
34 begin
35   f: for j in n-1 downto 0 generate
36     m: if j=n-1 generate
37       msd : bcd_cnt_c_e  port map (clk, rst_n, clear, c(j-1), q(4*j+3 downto 4*j), co);
38     end generate;
39     x: if (j>0) and (j<n-1) generate
40       nsd : bcd_cnt_c_e  port map (clk, rst_n, clear, c(j-1), q(4*j+3 downto 4*j), c(j));
41     end generate;
42     l: if j=0 generate
43       lsd : bcd_cnt_c_e  port map (clk, rst_n, clear, enable, q(4*j+3 downto 4*j), c(0));
44     end generate;
45   end generate;
46 end architecture gedrag;
```

In tegenstelling tot de beschrijving uit code 8.17 is de beschrijving voor alle cijfers gelijk en loopt de lusvariabele `j` op. De carry-in van het eerste, minst significante, cijfer wordt op regel 90 hoog gemaakt. De carry-out wordt steeds toegekend aan de carry-in van het volgende cijfer. De carry-out van het laatste, minst significante, cijfer wordt niet gebruikt.

8.5 De conversie van binair naar BCD

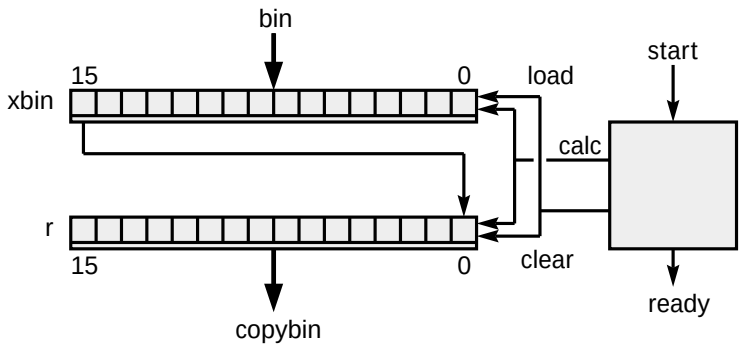
Het algoritme om van een binair getal een BCD-getal te maken is niet eenvoudig. Figuur 8.7 geeft de binaire en de BCD-representatie van het getal 53871. Het 16-bits binaire getal wordt een vijfcijferig BCD-getal dat uit twintig bits bestaat.



Figuur 8.7 : De omzetting van het 16-bits binaire getal 53871 naar een BCD-getal.

De opzet voor een serieel conversie-algoritme

De meest gebruikte methode voor de conversie van een binair getal naar een BCD-getal is om met twee schuifregisters te werken. Eén voor het binaire getal en één voor het te berekenen BCD-getal. De truc van dit seriële algoritme is om bit voor bit een kopie van het binaire getal te maken en voortdurend een correctie te maken voor het BCD-getal.

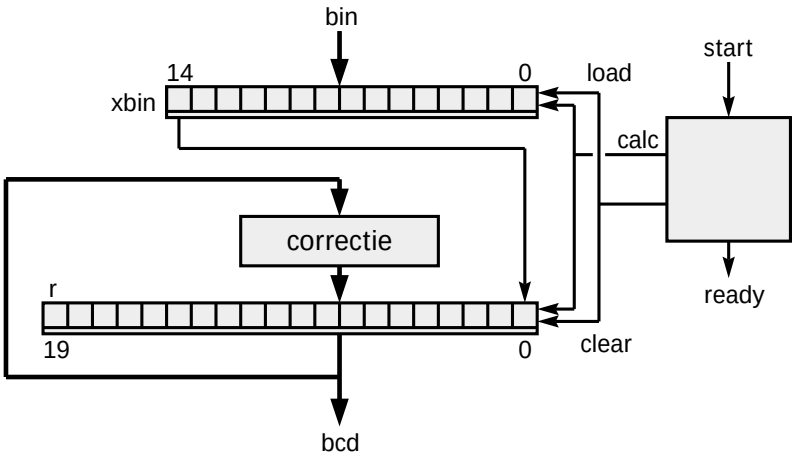


Figuur 8.8 : Een schakeling die van een binair signaal een seriële kopie maakt. Bij de start wordt signal `bin` in het schuifregister `xbin` geladen. Na 16 keer schuiven staat de waarde van `bin` in het schuifregister `r`.

In figuur 8.8 staat het schema met twee schuifregisters zónder correctie. Deze schakeling maakt van het binaire getal `bin` een exacte kopie `copybin` door tegelijkertijd:

- het origineel naar links te schuiven,
- de kopie naar links te schuiven,
- de meest significante bit van het origineel als minst significante aan de kopie toe te voegen.

De bewerking start als signaal `start` hoog is. Na 16-bits keer schuiven staat de waarde van `bin` in schuifregister `r`. Het signaal `ready` wordt vervolgens hoog gemaakt. De oorspronkelijke waarde van register `xbin` is door het schuiven verloren gegaan. Eigenlijk maakt deze bewerking geen kopie, maar verplaatst het de inhoud van `xbin` naar `r`.



Figuur 8.9 : **Een schakeling die een binair getal serieel omzet naar een BCD-getal.** Bij de start wordt signal `bin` in het schuifregister `xbin` geladen. Na 15 keer schuiven staat de BCD-waarde van `bin` in het schuifregister `r`.

Figuur 8.9 is gelijk aan figuur 8.8 maar nu met correctie. Omdat er voor een BCD-getal meer bits nodig zijn dan voor een binair getal zijn de bitbreedte van de schuifregisters verschillend. Signaal `xbin` is vijftien bits. De maximale waarde van een 15-bits binair getal is 32768. Om dit getal BCD-gecodeerd weer te geven zijn vijf cijfers nodig. De breedte van register `r` is daarom 20 bits.

Op dezelfde manier als bij de schakeling uit figuur 8.8 wordt er een BCD-kopie van de binaire getal gemaakt door:

- het origineel naar links te schuiven,
- de kopie naar links te schuiven,
- indien noodzakelijk de BCD-waarde te corrigeren,
- de meest significante bit van het origineel als minst significante aan de BCD-kopie toe te voegen.

Het schuiven van `xbin` en `r` gaat gelijktijdig. Deze twee signaaltoewijzingen berekenen de volgende waarden:

```
xbin <= xbin(13 downto 0) & '0';
r    <= correctie(r(18 downto 0)) & xbin(14);
```

Het meest significante bit van `xbin` wordt met een concatenatie aan `r` toegevoegd. De functie `correctie` corrigeert de volgende waarde, zodat de waarde in `r` altijd een BCD-getal is.

Het algoritme voor de correctie

Het schema uit figuur 8.9 geeft slechts de opbouw hoe er met twee schuifregisters een binair getal kan worden omgezet naar een BCD-getal. Essentieel in dit schema is de correctie, die bij iedere schuifactie nodig is.
Tabel 8.1 maakt de correctie die nodig is als de bits één positie worden opgeschoven. Naar links schuiven is hetzelfde als vermenigvuldigen met twee. De verwachte waarde is dus eenvoudig te bepalen. Zonder correctie ontstaan er cijfers met een binaire waarde, die groter is dan negen. De laatste twee kolommen uit de tabel tonen de correctie die ervoor zorgt dat het BCD-cijfers blijven.

Tabel 8.1 : De correctie die na het schuiven van het BCD-cijfer nodig is.
In de eerste kolom staan alle BCD-cijfers met de bijbehorende binaire waarde. In de tweede kolom staat de verwachte waarde in de decimale en binaire notatie. In de derde en de vierde kolom staat het resultaat als de oorspronkelijke waarde alleen geschoven wordt. Kolom drie geeft de uitkomst (hexa-)decimaal en binair. In kolom vier zijn de opgeschoven bits cursief gedrukt. De vijfde kolom is een kopie van de vierde kolom met de correctie voor de laatste vijf waarden . De cursief gedrukte bits in kolom vier en vijf tonen de correctie die nodig is.

cijfer oorspronkelijk	2×cijfer verwacht	cijfer << 1 alleen schuiven	cijfer << 1 alleen schuiven	correctie schuiven+correctie
0 0000	0 0 0000	0 0 0000	*0000*0	*0000*0
1 0001	2 0 0010	2 0 0010	*0001*0	*0001*0
2 0010	4 0 0100	4 0 0100	*0010*0	*0010*0
3 0011	6 0 0110	6 0 0110	*0011*0	*0011*0
4 0100	8 0 1000	8 0 1000	*0100*0	*0100*0
5 0101	10 1 0000	A 0 1010	*0101*0	10000
6 0110	12 1 0010	C 0 1100	*0110*0	10010
7 0111	14 1 0100	E 0 1110	*0111*0	10100
8 1000	16 1 0110	10 1 0000	*1000*0	10110
9 1001	18 1 1000	12 1 0010	*1001*0	11000

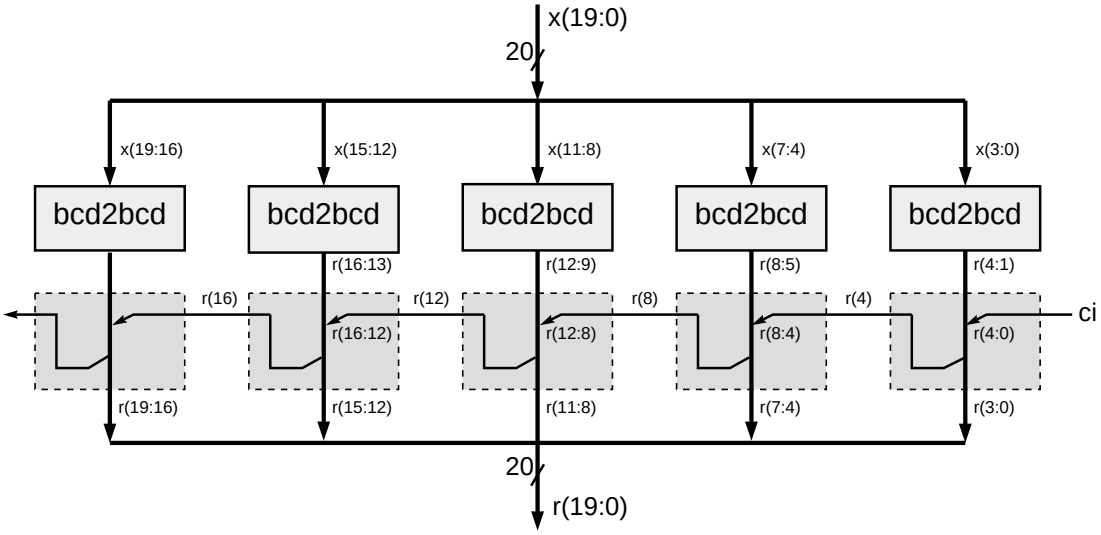
In code 8.18 staat een functie, die de gecorrigeerde waarde van een BCD-cijfer `d` teruggeeft. Het switch-statement vergelijkt `d` met de te corrigeren waarden. Als `d` gelijk is aan één van deze waarden geeft het de gecorrigeerde waarde terug. In alle andere gevallen is er geen correctie nodig en retourneert de functie de originele `d`.

Code 8.18 : De functie die de correctie uit tabel 8.1 uitvoert.

```
20  function bcd2bcd(d : in std_logic_vector(3 downto 0))
21                                   return std_logic_vector is
22  begin
23    case d is
24       when "0101"  =>  return("1000");
25       when "0110"  =>  return("1001");
26       when "0111"  =>  return("1010");
27       when "1000"  =>  return("1011");
28       when "1001"  =>  return("1100");
29       when others  =>  return(d);
30    end case;
31  end bcd2bcd;
```

De correctie voor een BCD-getal met meerdere cijfers

BCD-getallen bestaan natuurlijk uit meerdere cijfers. Met de functie `bcd2bcd` kan ieder cijfer apart gecorrigeerd worden. Als de carry-out van een bepaald cijfer hoog is, moet het volgende cijfer met één worden opgehoogd. Tabel 8.1 laat zien dat de minst significante bit van de verschoven en gecorrigeerde getallen altijd nul is. Ophogen betekent in dit geval dat de minst significante bit één wordt, niet ophogen betekent dat deze bit nul blijft. Anders gezegd, de minst significante bit is steeds de carry-out van het vorige cijfer.



Figuur 8.10 : De correctie, inclusief het schuiven, voor een vijfcijferig getal.
Ieder BCD-cijfer van signaal `x` wordt met de functie `bcd2bcd` aangepast en daarna één positie naar links geschoven. De gecorrigeerde waarde van `x(7:4)` wordt dus toegekend aan `r(8:5)` De carry-out van het vorige cijfer, `r(4)`, is de minst significante bit van het huidige cijfer.

In figuur 8.10 staat de correctie, inclusief het schuiven, voor een vijfcijferig getal. Na de aanroep van `bcd2bcd` wordt bij ieder cijfer de carry-out van het vorige cijfer toegevoegd en is de meest significante bit de carry-in van het volgende cijfer.

Code 8.19 : De schuif- en correctiefunctie `shlbcd`.

```
33    function shlbcd( x  : in std_logic_vector;
34                     ci : in std_logic) return std_logic_vector is
35      variable r : std_logic_vector(x'left+1 downto 0);
36    begin
37      r(0) := ci;
38      for i in 0 to x'left/4 loop
39        r(4*i+4 downto 4*i) := bcd2bcd(x(4*i+4-1 downto 4*i)) & r(4*i);
40      end loop;
41       return (r(x'left downto 0));
42    end shlbcd;
```

De functie `shlbcd` uit code 8.19 beschrijft het gedrag van de schakeling van figuur 8.10. De ingangsvector `x` is *unconstrained*. Signaal `ci` is de carry-in. De functie geeft een vector terug, die dezelfde lengte heeft als vector `x`. De carry-out wordt niet gebruikt. Bij de synthese zal dit signaal niet gemaakt worden en leidt dus niet tot overbodig logica.

De gedragsbeschrijving voor de seriële omzetting van binair naar BCD

De gedragsbeschrijving voor de omzetting van een binair getal naar een BCD-getal volgens het seriële algoritme uit figuur 8.9 van pagina 188 staat in code 8.20 en code 8.21. De beschrijving bestaat uit twee processen en drie signaaltoewijzingen. Ingangsvector `bin` is het binaire getal dat geconverteerd wordt en uitgangsvector `bcd` is dit zelfde getal in BCD-notatie.

Code 8.20 : De seriële conversie van een binair getal naar een BCD-getal. Deze beschrijving bevat de entity en het declaratiedeel van de architectuur. De body van de architectuur staat in code 8.21

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3  use ieee.numeric_std.all;
 4  use ieee.math_real.all;
 5
 6  entity bin2bcd is
 7    port (
 8      clk   : in  std_logic;
 9      start : in std_logic;
10      bin   : in  std_logic_vector;
11      bcd   : out std_logic_vector;
12      ready : out std_logic
13    );
14  end entity bin2bcd;
15
16  architecture gedrag of bin2bcd is
17    constant nbin : natural := bin'length;
18    constant nbcd : natural := 4 * (natural(ceil( real(nbin) * log10(2.0) )));
        ⋮ Hier staan de declaraties van de functie bcd2bcd uit code 8.18
        ⋮ en de functie shlbcd uit code 8.19
43    signal r       : std_logic_vector(nbcd-1 downto 0);
44    signal c       : natural range 0 to nbcd+1;
45    signal xbin    : std_logic_vector(nbin-1 downto 0);
46    signal convert : std_logic;
```

De vectoren zijn *unconstrained*. Constante `nbin` is het aantal bits n van het binaire getal dat op `bin` is aangesloten. Het aantal bits van `bcd` hangt af van n. Voor een groter getal zijn meer cijfers en dus meer bits nodig. Het bereik van het binaire getal is gelijk aan $2^n - 1$. Het bereik van het BCD-getal is afhankelijk van het aantal cijfers d en is gelijk aan $10^d - 1$. Het bereik van het BCD-getal moet groter of gelijk zijn aan het bereik van het binaire getal:

$$10^d \;\geq\; 2^n \tag{8.1}$$

Voor het aantal cijfers d van het BCD-getal geldt:

$$d \;\geq\; n\log(2) \tag{8.2}$$

Het minimaal aantal BCD-cijfers d_{min} dat nodig is om een `n`-bits binair getal wéér te geven is zodoende gelijk aan:

$$d_{\text{min}} \;=\; \lceil n\log(2)\rceil \tag{8.3}$$

Naar beneden afronden, wordt gedaan met de functie `floor`. Naar boven afronden, wordt gedaan met de functie `ceil`. In de wiskunde geeft men met deze functies aan respectievelijk $\lfloor\ \rfloor$ en $\lceil\ \rceil$.

Het minimaal aantal bits is vier zo groot als het minimaal aantal cijfers d_{min}. Op regel 18 is formule 8.3 toegepast om uit `nbin` de constante `nbcd` te berekenen, die het aantal bits van het BCD-getal definieert.

Proces `xbin_r` beschrijft de registers `xbin` en `r` uit figuur 8.9 en vormt met de signaaltoewijzing aan `bcd` het dataverwerkingsdeel. Op regel 56 gebruikt proces `xbin_r` de functie `shlbcd` om de bits van register `r` één positie naar links te schuiven en gelijktijdig te corrigeren. De carry-in van deze functie is de meest significante bit van register `xbin`.

Code 8.21 : Vervolg beschrijving seriële conversie van een binair getal naar een BCD-getal. Deze beschrijving bevat de body van de architectuur. De entity en het declaratiedeel van de architectuur staan in code 8.20

```
47  begin
48    xbin_r: process (clk) is
49    begin
50      if clk'event and clk = '1' then
51        if start='1' then
52          xbin <= bin;
53          r <= (others => '0');
54        elsif convert = '1' then
55          xbin <= xbin(nbin-2 downto 0) & '0';
56          r <= shlbcd(r, xbin(nbin-1));
57        end if;
58      end if;
59    end process xbin_r;
60    bcd <= r;
61
62    control: process (clk) is
63    begin
64      if clk'event and clk = '1' then
65        if start = '1' then
66          c <= nbin+1;
67        elsif c > 0 then
68          c <= c - 1;
69        end if;
70      end if;
71    end process control;
72
73    convert <= '1' when c > 1 else '0';
74    ready   <= '1' when c = 1 else '0';
75  end architecture gedrag;
```

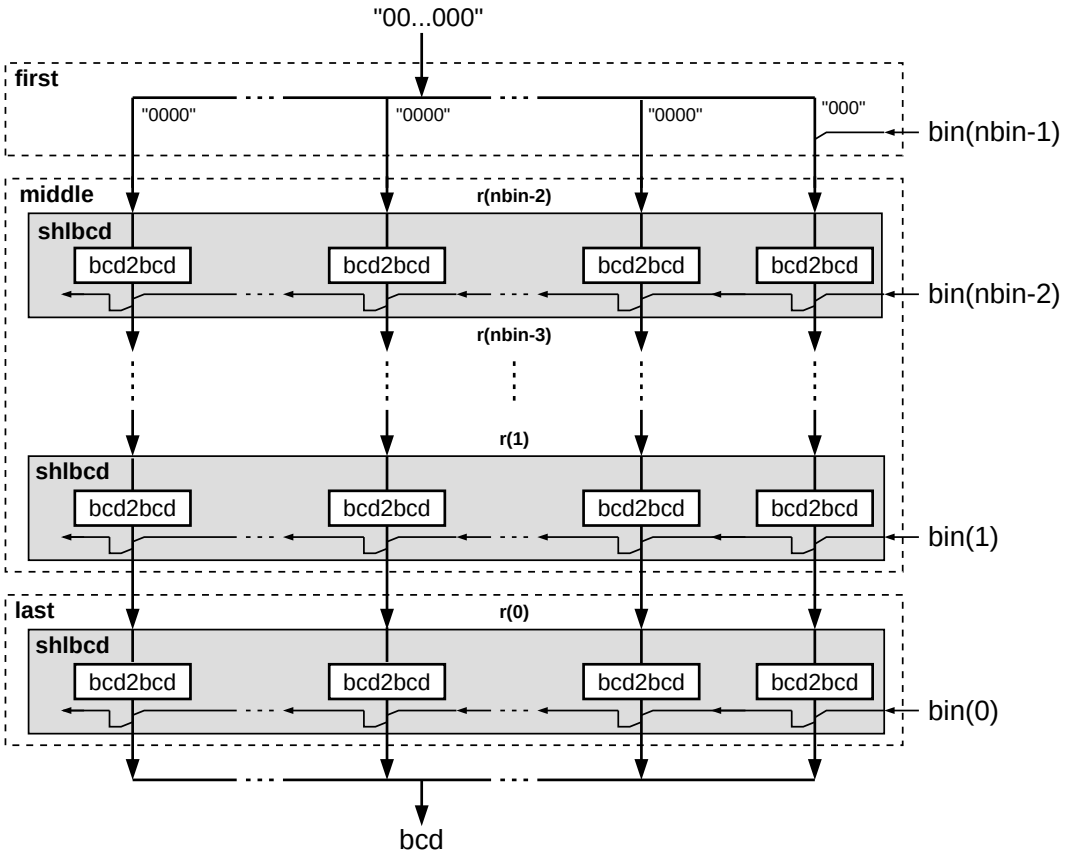
Proces `control` is een teller en is samen met de signaaltoewijzingen aan `convert` en `ready` het besturingsdeel van de omzetter. De conversie begint als signaal `start` hoog is. Register `xbin` krijgt de waarde van signaal `bin` en geeft tegelijkertijd de teller zijn beginwaarde `nbin+1`. Vervolgens telt de teller af totdat deze nul is. Signaal `convert` is hoog zolang de waarde van de teller groter is dan één. Als de teller gelijk aan één is, is signaal `ready` hoog.

Figuur 8.11 : De teller van proces `control` is de toestandsmachine voor de conversie. De toestanden van de teller zijn in drie groepen verdeeld.

De teller telt vanaf `nbin+1` terug naar nul en is een toestandsmachine met `nbin+2` toestanden. In figuur 8.11 zijn deze toestanden getekend en gegroepeerd in drie groepen. Er zijn `nbin` toestanden nodig voor het schuiven, één toestand waarbij `ready` hoog is en een toestand `idle` waarbij niets gebeurt.

Gedragsbeschrijving voor een parallelle omzetting van binair naar BCD

De beschrijving van code 8.20 heeft meerdere klokslagen nodig om een binair getal te converteren naar een BCD-getal. Het aantal klokslagen is evenredig met het aantal bits van het binaire getal. In figuur 8.11 start de teller van de toestandsmachine bij `nbin+1` en is de conversie klaar bij `1`.
In plaats van de bewerkingen na elkaar uit te voeren kunnen deze ook parallel worden uitgevoerd. Er zijn dan geen schuifregisters nodig. Wel is er voor elke stap een eigen schuif- en correctiefunctie nodig. In figuur 8.12 is een opzet getekend.



Figuur 8.12 : De structuur voor de parallelle conversie van een binair naar een BCD-getal. Rechts staan de `nbin` bits van het binaire getal. Er zijn net zoveel schuif- en correctiefuncties `shlbcd` als het aantal binaire bits. De uitkomst van de bewerkingen wordt niet in een register bewaard, maar wordt doorgegeven aan de volgende functie `shlbcd`. Er zijn `nbin` interne signalen. Hiervoor is een array `r` van vectoren nodig.

Bij de seriële omzetting van figuur 8.9 is de eerste bit dat register `r` ingeschoven wordt de meest significante bit van ingangssignaal `bin`. De andere bits van register `r` zijn bij de start van de conversie nul gemaakt. Bij de parallelle omzetting gaat dit op een overeenkomstige wijze. De eerste stap uit figuur 8.12 maakt ook alle bits nul en de minst significante bit gelijk aan `bin(nbin-1)`. Het resultaat — vector `r(nbin-2)` — staat nu niet in een register, maar wordt aan een schuif- en correctiefunctie `shlbcd` aangeboden. De index van het interne signaal komt overeen met de index van de betreffende bit dat de functie `shlbcd` in geschoven wordt. Bij de

volgende stappen wordt de uitkomst `r(i)` van een `shlbcd` samen met de volgende bit bin(i) van `bin` aangeboden aan de volgende functie `shlbcd`. De uitkomst van de laatste rij is het geconverteerde BCD-getal en is toegekend aan uitgang `bcd`.

In code 8.22 staat de VHDL-beschrijving voor de parallelle conversie. De entity heeft — in tegenstelling tot de seriële omzetting — één ingang en één uitgang. Er is geen klok, zodat het een combinatorische schakeling beschrijft.

Code 8.22 : De parallelle conversie van een binair getal naar een BCD-getal.

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3  use ieee.numeric_std.all;
 4  use ieee.math_real.all;
 5
 6  entity bin2bcd_p is
 7    port (
 8      bin   : in  std_logic_vector;
 9      bcd   : out std_logic_vector
10    );
11  end entity bin2bcd_p;
12
13  architecture gedrag of bin2bcd_p is
14    constant nbin : natural := bin'length;
15    constant nbcd : natural := 4 * (natural(ceil( real(nbin) * log10(2.0) )));
        ⋮ Hier staan de declaraties van de functie bcd2bcd uit code 8.18
        ⋮ en de functie shlbcd uit code 8.19
40    type bcd_array is array (natural range <>) of std_logic_vector(nbcd-1 downto 0);
41
42    signal r : bcd_array(nbin-2 downto 0);
43  begin
44    f : for i in nbin-1 downto 0 generate
45       first: if i=nbin-1 generate
46         r(i-1)(nbcd-1 downto 1) <= (others => '0');
47         r(i-1)(0)               <= bin(i);
48       end generate;
49       middle: if (i>0) and (i<nbin-1) generate
50         r(i-1) <= shlbcd(r(i), bin(i));
51       end generate;
52       last: if i=0 generate
53         bcd <= shlbcd(r(0), bin(0));
54       end generate;
55    end generate;
56  end architecture gedrag;
```

Er is gebruik gemaakt van een array van `std_logic_vector`. De typedeclaratie op regel 40 beschrijft een array `bcd_array` die uit `std_logic_vector` met een lengte `nbcd` bestaat. Regel 42 definieert een signaal `r` dat een array is met een lengte `nbin` van type `bcd_array`.

Het generate-statement `f` bestaat uit drie delen, `first`, `middle` en `last`. Deze drie delen zijn in figuur 8.12 aangegeven. De eerste deel kent aan de minst significante bit van `r-1` signaal `bin(i)` toe en maakt de andere bits nul. Signaal `r` is feitelijk een tweedimensionaal array van `std_logic`. Het toewijzen van een specifiek bit gaat

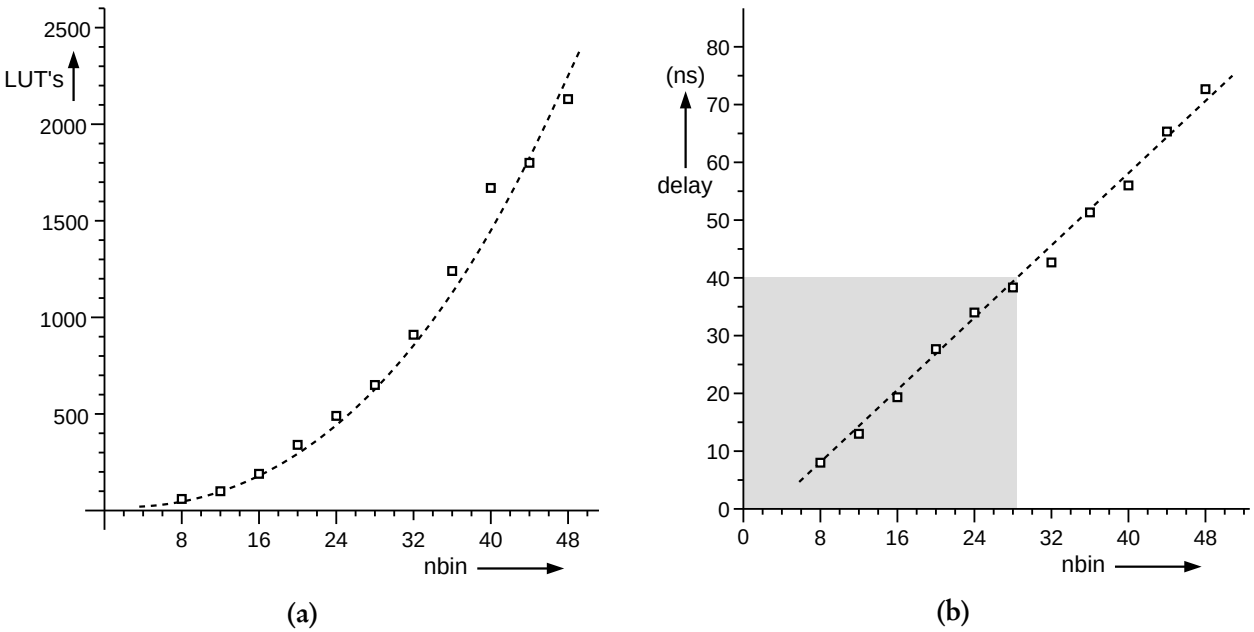
met twee indices. De eerste index is de vector en de tweede index is de betreffende bit van deze vector. De toewijzingen van regel 46 en 47 kunnen ook met één enkele signaaltoewijzing:

```
r(i-1) <= (0 => bin(i), others => '0');
```

Alleen geeft de simulator Modelsim een waarschuwing dat **others** alleen niet lokaal gebruikt mag worden als de keuze uniek is. Het tweede deel van het generate-statement met de label `middle` genereert voor de bits `1` tot en met `nbin-1` de schuif- en conversiefunctie. Het derde deel bevat de laatste aanroep van `shlbcd` en kent het resultaat aan `bcd` toe.

Een flink deel van de beschrijving uit code 8.22 is overbodig. De functie `bin2bcd` linksboven uit figuur 8.12 heeft bijvoorbeeld altijd een ingangswaarde `"0000"`. De conversie verandert daar niets aan. De waarde van de uitgang is dus ook "0000". Dit geldt voor alle functies die linksboven in de tekening staan. Een betere beschrijving laat het aantal functies in iedere rij afhangen van het rijnummer. De VHDL-beschrijving wordt dan veel complexer, terwijl een goede synthesizer de overbodige functies toch uit het netwerk verwijdert. Daarom is er gekozen voor de regelmatige structuur van figuur 8.12.

Het voordeel van deze parallelle beschrijving boven de seriële beschrijving van code 8.20 is dat het een combinatorische beschrijving is en dat in één klokslag de binaire waarde geconverteerd kan worden. Het nadeel is dat voor grote getallen er veel logica nodig is. Bovendien wordt de vertragingstijd in dat geval erg groot.



Figuur 8.13 : **Het syntheseresultaat als functie van de vectorlengte** `nbin`**.** In figuur (a) staat het aantal LUT's als functie van het aantal bits `nbin`. In figuur (b) staat de vertragingstijd als functie van het aantal bits `nbin`.

In figuur 8.13 staat het resultaat van de synthese van code 8.22 met Leonardo Spectrum. De FPGA is een Cyclone-II EP2C5F256C van Altera en de vereiste klokfrequentie is 25 MHz. Figuur 8.13a geeft het aantal LUT's als functie van de

vectorlengte `nbin`. De grootte van het gesynthetiseerde oppervlak neemt kwadratisch toe met de vectorlengte. Figuur 8.13b geeft de vertragingstijd als functie van de vectorlengte `nbin`. De vertragingstijd neemt lineair toe met de vectorlengte.

Het is lastig om algemene uitspraken te doen over de syntheseresultaten, omdat deze afhangen van het gebruikte syntheseprogramma en de opgegeven condities. In dit voorbeeld is de eis dat de klokfrequentie minimaal 25 MHz is. Dit betekent dat de vertragingstijd maximaal 40 ns is. Uit figuur 8.13.b volgt dan dat de maximale vectorlengte van het te converteren binaire getal gelijk is aan 28. Zoals in de meeste gevallen is de beperking niet het aantal LUT's van de FPGA. De kleinste Cyclone-II heeft immers al 4608 LUT's. De beperkende factor is tegenwoordig de tijdvertraging.
Bewerkingen met veel combinatoriek zijn groot en daardoor te traag. Seriële oplossingen zijn vaak klein, hebben weinig tijdvertraging, maar hebben wel meer klokslagen nodig om het antwoord te berekenen. Voor een seriële 32-bits omzetter zijn slechts 81 LUT's en 78 flipfloppen nodig.

8.6 Resumé

Veel wiskundige algoritmes kunnen parallel en serieel worden uitgevoerd. Seriële oplossingen zijn vaak gebaseerd op schuifregisters, zijn relatief klein en snel en zijn bruikbaar bij hoge kloksnelheden. Het nadeel is dat er meer klokslagen nodig zijn. Parallelle beschrijvingen berekenen de uitkomst in één enkele klokslag, maar zijn relatief groot en traag.
De regelmatige structuren, die bij wiskundige bewerkingen nodig zijn, kunnen uitstekend met for-lussen of generate-statements beschreven worden. Door gebruik van generieke parameters of door *unconstrained* vectoren toe te passen wordt de flexibiliteit van de beschrijving groter. Het algoritme kan daarmee voor een willekeurig aantal bits worden vastgelegd.
Werk bij het ontwerp van een algoritme systematisch en gebruik de werkwijze en het stappenplan uit paragraaf 8.2. Bij wiskundige algoritmes is het belangrijk om nette tekeningen te maken, zodat de indices van de for-lussen en generate-statements correct en eenvoudig te controleren zijn.

9

Verificatie

Doelstelling	In dit hoofdstuk leer je wat verificatie, validatie en testen is. Je maakt kennis met de testmogelijkheden voor FPGA's en ASIC's. Je leert wat simulatie is, hoe je een testbench maakt, wat goede stimuli zijn. Daarnaast maak je kennis met dynamische en statische tijdsanalyses.
Onderwerpen	De behandelde onderwerpen zijn: ▪ Validatie, verificatie en testen. ▪ Het verschil tussen functionele en structurele verificatie. ▪ De mogelijkheden van functionele simulatie. ▪ De verschillende testbenchconfiguraties ▪ De toepassing van het report- en het assert-statement. ▪ Het opstellen van stimuli en het kiezen van testvectoren. ▪ Statische tijdsanalyse. ▪ De invloed van een *false path* en van een *multicycle path*. ▪ Dynamische tijdsanalyse of timingssimulatie. ▪ Verificatie van de vermogensdissipatie.

Verificatie, testen, debuggen, valideren en simuleren zijn termen die elkaar deels overlappen en soms ten onrechte als synoniem worden gebruikt. Het onderscheid is vaak subtiel. Zo is verificatie van een ontwerp het aandragen van bewijs dat een ontwerp voldoet aan de gestelde specificaties. Validatie van een ontwerp is daarentegen het aandragen van bewijs dat een ontwerp voldoet aan de gestelde specificatie in een situatie waarvoor het ontwerp bedoeld is. Verificatie is het controleren van alle deelontwerpen en het bestuderen van alle verschillende ontwerpstappen. Simulatie is een belangrijk hulpmiddel bij verificatie. Bij validatie test men het uiteindelijke product op de gestelde specificaties onder echte condities. Een testrit is een goed voorbeeld van een validatie.

9.1 Verificatie en testen

Verificatie is het controleren of een ontwerp overeenkomt met de gewenste specificatie. Haalt het ontwerp het juiste prestatievermogen en heeft het de gewenste performance? Functionele verificatie is de controle of het ontwerp de vereiste functionaliteit heeft. Hebben de uitgangen de juiste uitgangswaarden? Bij de verificatie van het prestatievermogen gaat het vooral om het tijdsgedrag van het ontwerp. Reageert het ontwerp snel genoeg en wordt de gewenste kloksnelheid

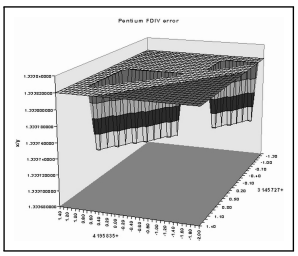
gehaald. Vermogensdissipatie is tegenwoordig een ander belangrijk aspect. Produceert het ontwerp niet te veel warmte en verbruikt het ontwerp niet te veel vermogen.

Figuur 9.1 : De explosie van de Ariana-5.
In 1996 ontplofte vlak na de start de Ariana-5 raket van de European Space Agency. Bij de besturing van de raket werd een 64-bits floating-point getal omgezet naar een 16-bits integer. Er was echter geen rekening gehouden met overflow.

Functionele verificatie

Het ontwerp van een digitaal systeem moet voldoen aan het functionele gedrag zoals dat in de specificatie is beschreven. Soms wordt er eerst een pure gedragsbeschrijving gemaakt, die het gedrag van het hele systeem vastlegt. In andere gevallen wordt er direct een architectuur gemaakt. Het ontwerp wordt opgesplitst in deelontwerpen en er worden functionele specificaties voor alle deelsystemen gemaakt. De verschillende onderdelen worden stap voor stap verder ontwikkeld en uiteindelijk samengevoegd. Dit leidt tot een groot, gesynthetiseerd netwerk dat voldoet aan de functionele specificatie.
Bij het opdelen in blokken en bij het uitwerken van deze blokken worden fouten gemaakt. Vaak zijn dat kleine fouten, die eenvoudig herkend en gerepareerd kunnen worden. In andere situaties is het ontwerp ogenschijnlijk correct, maar bevat het een subtiele fout met een desastreus gevolg voor de werking.
In het ideale geval is de eerste aanzet van het ontwerp al *correct by construction* en worden er bij het ontwerpproces geen fouten gevonden. Tijdens het verfijnen van het ontwerp zullen er nieuwe fouten geïntroduceerd worden, die met functionele verificatie bij de diverse ontwerpstappen, uit het ontwerp gehaald moeten worden.

Simulatie is het hulpmiddel om een ontwerp functioneel te verifiëren. Een ontwerp dat bij de simulatie niet correct functioneert, zal in werkelijkheid ook niet functioneren. Het bedenken van goede simulaties, waarmee alle functionele aspecten van een complex ontwerp getest worden, is een grote uitdaging.
Simulaties kunnen nooit bewijzen dat een ontwerp correct is. Hooguit kan een simulatie aantonen dat een ontwerp één of meer fouten bevat. In de praktijk is het immers onmogelijk om alle situaties te testen. Voor een component met twee 32-bits ingangen zijn er al 2^{64} verschillende ingangscombinaties. Als er voor iedere test 1 ns nodig is, duurt de simulatie meer dan 500 jaar. Daarom kunnen nooit alle mogelijkheden worden gesimuleerd.



Figuur 9.2 : De FDIV-bug van de eerste Pentium.
De afbeelding toont een deel van de getalruimte waarin zichtbaar is dat 4195835 gedeeld door 3145727 niet correct is.

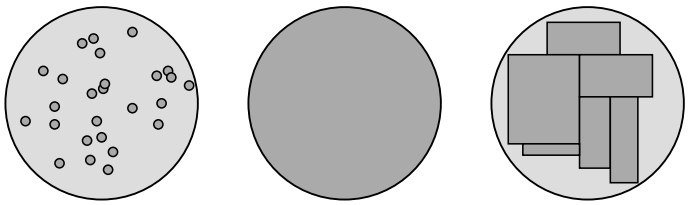
Formele verificatie

Met *formal verification* of formele verificatie kan een ontwerp of een deel van een ontwerp wel uitputtend worden getest. Bij formele verificatie worden logische technieken gebruikt, die werkelijk kunnen bewijzen dat een ontwerp correct is. Er zijn drie vormen van formele verificatie:

- **Equivalence checking**
  Hierbij wordt aangetoond dat twee ontwerpen identiek zijn. Dat kunnen twee ontwerpen van een ander abstractieniveau zijn, bijvoorbeeld een RTL-ontwerp en een gate-level-ontwerp. Per definitie detecteert deze methode geen ontwerpfouten, maar fouten tijdens het implementatietraject.
- **Model checking**
  Deze methode controleert of de eigenschappen van een ontwerp voldoen aan het logische model. Voor toestandsmachines kan deze methodiek volledig geautomatiseerd worden. In de praktijk is dat alleen haalbaar voor kleine systemen.

- **Theorem Proving**
  In dit geval wordt het probleem vastgelegd in een set vergelijkingen, hulpstellingen en regels. Via deductie wordt aangetoond dat het ontwerp correct is.

Het voordeel van formele verificatie is dat er honderd procent zekerheid is dat een ontwerp of een deel van een ontwerp correct is. De FDIV-bug van de eerste Pentium-processor van Intel was met behulp van formele verificatie zeker gevonden. Een nadeel van formele verificatie is dat de drie methoden niet in alle gevallen bruikbaar zijn en dat een complex ontwerp nooit volledig geverifieerd kan worden. In figuur 9.3 is dat gevisualiseerd.



Figuur 9.3 : **De dekkingsgraad bij simulatie en bij formele verificatie.** De grote cirkels representeren alle mogelijke situaties, die getest moeten worden. De niet geteste delen zijn licht grijs. De delen die wel gedekt worden door de tests zijn donker grijs.

Verificatie van het tijdsgedrag

Verificatie van het tijdsgedrag controleert of de schakeling de juiste performance heeft wat betreft het tijdsgedrag. Deze performance wordt uitgedrukt in een minimaal haalbare propagatietijd of in een maximale haalbare klokfrequentie. Op register-transfer niveau kan de propagatietijd tussen een ingang en een uitgang worden berekend door verschillende tijdvertragingen van de individuele onderdelen langs het signaalpad te sommeren. Dit geeft altijd een ruwe schatting. Verdere verfijning van het ontwerp en de synthese van de onderdelen heeft invloed op de propagatietijden. Op poortniveau wordt de propagatietijd niet alleen beïnvloed door de vertraging van de individuele componenten, maar ook door de lengte van de verbindingen. Pas na de plaatsing en de bedrading kunnen hiervoor exacte waarden worden gegenereerd. Nieuwe procestechnieken zorgen ervoor dat de transistorafmetingen steeds kleiner worden en daardoor wordt de vertraging ten gevolge van de plaatsing en bedrading relatief belangrijker en moeilijker vooraf te schatten.
Of een ontwerp voldoet aan de gestelde eisen wat betreft het tijdsgedrag kan met diverse methoden geverifieerd worden: door middel van een timingssimulatie of dynamische tijdsanalyse, met een zogenoemde *timing analysis* of statische tijdsanalyse en door middel van hardware-emulatie.

Vermogensverificatie

Bij vermogensverificatie moet het gedissipeerde vermogen van het ontwerp aan de gestelde eisen voldoen. Dit onderwerp wordt steeds belangrijker. Er zijn steeds meer draagbare producten en apparaten die draadloos communiceren en een eigen

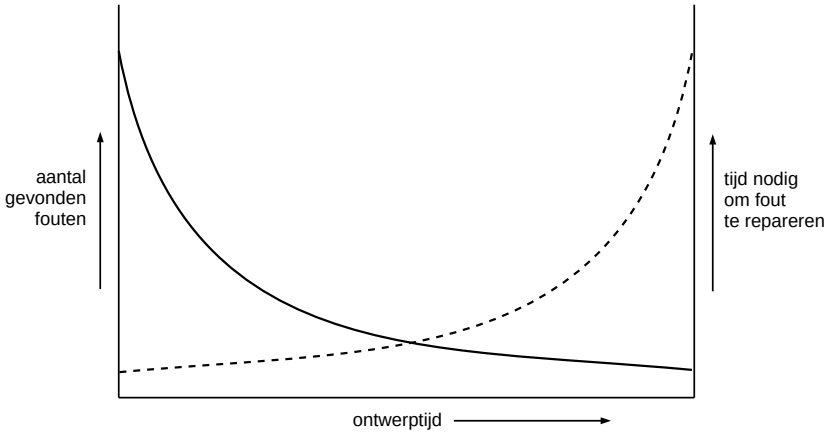
voedingsbron hebben. Batterijen moeten steeds langer gebruikt kunnen worden voordat deze weer opgelaad moeten worden, terwijl de eisen, die er aan de performance van nieuwe producten gesteld worden, steeds hoger worden.
Een ander bijkomend aspect is dat het gedissipeerde vermogen consequenties heeft voor de warmtehuishouding. Een hoge dissipatie maakt een component heet. Voor de gebruiker kan dat onplezierig zijn. Minstens zo belangrijk daarbij is dat het gedrag, zoals dat van de tijdskarakteristieken, verandert en dat daardoor het product uiteindelijk niet meer aan de gestelde eisen voldoet.

Functionele en structurele verificatie

De ontwerper van een digitaal systeem is vooral gericht op het functionele gedrag en op de performance van het systeem. Als het ontwerp op de markt wordt gebracht, moet bij de productie het systeem getest en gecontroleerd worden of het nog steeds correct werkt. Bij de productie treden altijd fouten op: een printspoor op een PCB maakt kortsluiting; een via is niet goed gemetalliseerd of een component is niet goed gemonteerd. Bij de productie van ASIC's zorgt een stofdeeltje ervoor dat een verbinding niet aanwezig is of krijgt een transistor niet de juiste eigenschappen door een belichtingsfout.
Bij de productie moeten IC's en PCB's tijdens het fabricageproces continu getest worden. Dat kan door deze systemen functioneel te verifiëren. Een nadeel van functioneel testen is dat dit lang duurt.
Een andere methode is om het systeem structureel te verifiëren. Er wordt fysiek getest of op het PCB de verbindingen tussen de IC's correct zijn; of er geen kortsluitingen of open contacten zijn. Bij de productie van ASIC's en bij de productie van complexe PCB's wordt daarvoor JTAG gebruikt. JTAG staat voor *Joint Test Action Group* en is een seriële interface die gebruikt wordt om componenten intern te testen, de verbindingen tussen componenten te verifiëren en om programmeerbare bouwstenen en geheugens te programmeren.
Met behulp van ATPG, *automatic test pattern generation*, worden automatisch testsignalen gegenereerd. Deze testsignalen zijn gebaseerd op de fysieke structuur van het systeem en niet op het functionele gedrag. Het meest gebruikte foutmodel is het *single stuck-at fault model*. Dit model veronderstelt dat steeds één van de interne verbindingen vastzit (*stuck at*) aan een bepaald logisch niveau. Met ATPG worden testsignalen gegenereerd, die voor belangrijke interne verbindingen deze *stuck-at*-fouten zichtbaar maken.

Verificatie en optimalisatie

Moderne digitale systemen zijn functioneel complex. Desalniettemin dienen ze klein te zijn, snel te zijn en weinig vermogen te dissiperen. Principieel zijn dat tegengestelde eisen. Een systeem dat snel is, zal veel parallellisme bevatten om berekeningen gelijktijdig uit te voeren. Daardoor zal het systeem groot zijn en meer dissiperen. Bij een digitaal ontwerp gaat het dus niet alleen om het vastleggen van het functionele gedrag, maar ook om de andere ontwerpaspecten.
Het functionele gedrag wordt initieel meestal geverifieerd met behulp van simulatie. Het maken van een testbench en het bedenken van de signalen waarmee het ontwerp wordt getest is een belangrijk, maar tijdrovend, werk. Figuur 9.4 laat zien dat in de loop van het ontwerptraject het steeds lastiger wordt om fouten uit het ontwerp te halen en dat het tegelijkertijd steeds meer tijd kost om fouten te repareren.



Figuur 9.4 : **Gevonden fouten en reparatietijd versus testtijd.** Tijdens het ontwerpen wordt het ontwerp voortdurend getest. De getrokken lijn geeft aan dat het tijdens het ontwerptraject steeds lastiger is om fouten uit het ontwerp te halen. De streeplijn laat zien dat tegelijkertijd de reparatie van fouten steeds meer tijd vergt.

Voor het tijdsgedrag is een statische tijdsanalyse, *statical timing analysis* belangrijk. De ontwerper krijgt daarmee snel inzicht in het tijdsgedrag van de schakeling. Een statische tijdsanalyse wordt uitgevoerd na de synthese. Om op systeemniveau vooraf een gedetailleerde uitspraak te doen over het tijdsgedrag is niet mogelijk.

Moderne ontwikkelomgevingen voor FPGA's en CPLD's omvatten behalve een *timing analyzer* allerlei hulpprogramma's die de ontwerper ondersteunen. Deze programma's adviseren om het tijdsgedrag te verbeteren door bijvoorbeeld aan de synthesizer bepaalde opties mee te geven.

De adviesprogramma's voor de vermogensdissipatie geven naast adviezen om bepaalde opties te gebruiken ook adviezen om het ontwerp aan te passen, zodat de synthesizer in staat is een fifo in de RAM-blokken van de FPGA te plaatsen in plaats van de fifo op te bouwen uit de flipfloppen van de logische blokken.

9.2 Simulatie

Het begrip vector betekent hier alleen een combinatie van enen, nullen en eventueel don't cares. Er is dus geen sprake van een richting zoals gebruikelijk is bij een vector in de wiskunde en de natuurkunde.

Een ontwerp wordt getest door de ingangssignalen te veranderen en dan te kijken of het ontwerp op de juiste manier reageert. Een combinatie van ingangswaarden voor een test, die op een bepaald moment wordt aangeboden, wordt een testvector genoemd.

Stimulus is iets wat het gedrag beïnvloedt; het is datgene dat prikkelt of dat aanspoort. Meestal zijn er meerdere ingangssignalen en spreekt men over de stimuli van de simulatie.

Een set testvectoren wordt bij een simulatie de stimuli genoemd. Het bedenken van zinvolle stimuli is de taak van de ontwerper. Goede stimuli zijn zo volledig mogelijk.

Het is niet mogelijk om bij een complexe systeem alle situaties te testen. Het aantal mogelijkheden is veel te groot. De test zou te lang duren. Voor een systeem met N ingangen en M flipfloppen zijn $2^{(N+M)}$ combinaties mogelijk. Een goede simulatie test in ieder geval alle belangrijke situaties. Daarbij spelen twee fundamentele aspecten een rol: de observeerbaarheid en de aanstuurbaarheid. Kan een bepaald signaal hoog of laag gemaakt worden en kan de toestand van een bepaald signaal worden waargenomen.

De Engelse termen voor observeerbaarheid en aanstuurbaarheid zijn: *observability* en *controllability*.

Simulatie en testmethodieken

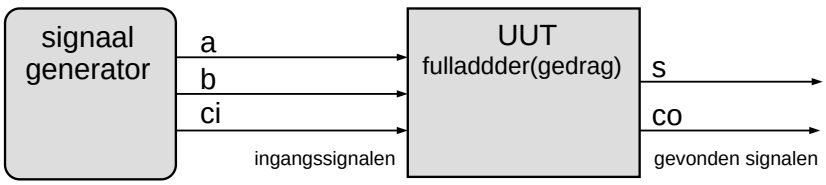
Er zijn verschillende testmethodieken:

- **Interactief**
  Bij interactief testen worden op basis van de huidige situatie van het te testen systeem de ingangssignalen gewijzigd. Bij de meeste simulatoren kan dat grafisch en met behulp van tekstuele commando's. Deze werkwijze is ad hoc; het leidt tot slechte documentatie en is naderhand slecht reproduceerbaar.
- **Met commandobestand**
  Door vooraf alle testvectoren als commando's in een bestand te zetten, ligt de test vast en kan deze later gereproduceerd worden. Vooraf nadenken over de stimuli, zal een betere test opleveren. Deze werkwijze is wel systeem en compiler afhankelijk. De commando's moeten vertaald worden naar de commandotaal van die andere simulator.
- **Met een waveformgenerator**
  Een waveformgenerator is een grafische omgeving waarmee testvectoren gegenereerd worden. Deze methode is ad hoc en bovendien systeem en compiler afhankelijk.
- **Met een testbench**
  Bij de methode met de testbench wordt de test en de testomgeving in dezelfde taal geschreven als het ontwerp. De test ligt van tevoren vast. De methode is systeem en compiler onafhankelijk, is reproduceerbaar, is goed te hergebruiken en is eenvoudig te documenteren. Bovendien kan de testbench door een ander team ontwikkeld worden.

De signalen worden met een waveformviewer bestudeerd. Deze viewer toont als functie van de tijd de signaalveranderingen en de onderlinge relaties tussen de signalen. Een viewer kan in- en uitzoomen op een deel van de signalen of op een deel van de tijd. Signalen kunnen gerepresenteerd worden als bitstrings of als decimale of hexadecimale integers. Bovendien kan de tijd tussen twee signaalovergangen gemeten worden.

Testbenchconfiguraties

In paragraaf 2.6 staat een testbench voor de full-adder. Deze testbench bestaat uit twee delen: een proces `signal_generator` die de ingangssignalen `a`, `b` en `ci` levert en de aanroep van de component `fulladder`. In figuur 9.5 staat een grafische weergave van de architectuur van deze testbench.

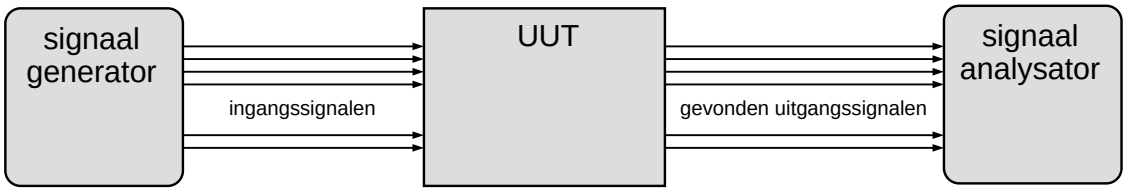


Figuur 9.5 : **De testbench voor de full-adder.** De testbench bestaat uit een signaalgenerator, die de ingangssignalen `a`, `b` en `ci` levert en de aanroep van de `uut`, de *unit under test*. In dit geval is dat de entity full-adder.

Het proces `signal_generator` levert de stimuli. Vooraf is bedacht welke ingangscombinaties van `a`, `b` en `ci` aan de full-adder worden aangeboden. Het te testen onderdeel wordt de *unit under test*, UUT, genoemd. De uitgangssignalen zijn niet aangesloten en kunnen net als alle andere signalen in de waveformviewer bekeken worden.

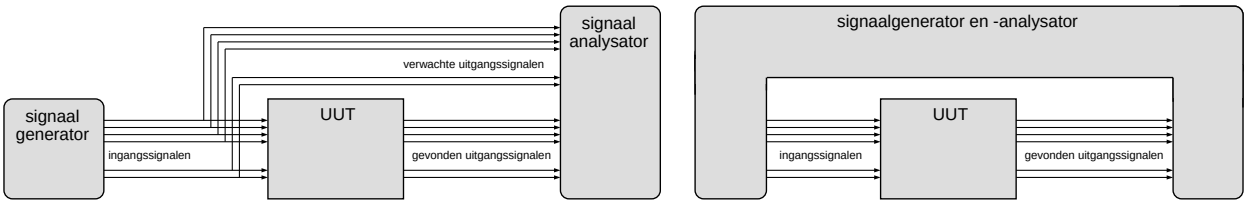
De term *UUT* is een variant op de, in de testwereld, gangbare term DUT *Device Under Test*.

Bij complexe digitale systemen zijn heel veel signalen betrokken en moeten deze signalen over een lange tijd gevolgd worden. Bovendien worden bij het debuggen simulaties vaak herhaald om het effect van veranderingen in de code te observeren. Het loont de moeite om, zoals in figuur 9.6 is getekend, een aparte signaalanalysator te maken, die de uitgangssignalen interpreteert en de ontwerper feedback geeft.



Figuur 9.6 : De testbench met een aparte signaalgenerator en signaalanalysator.

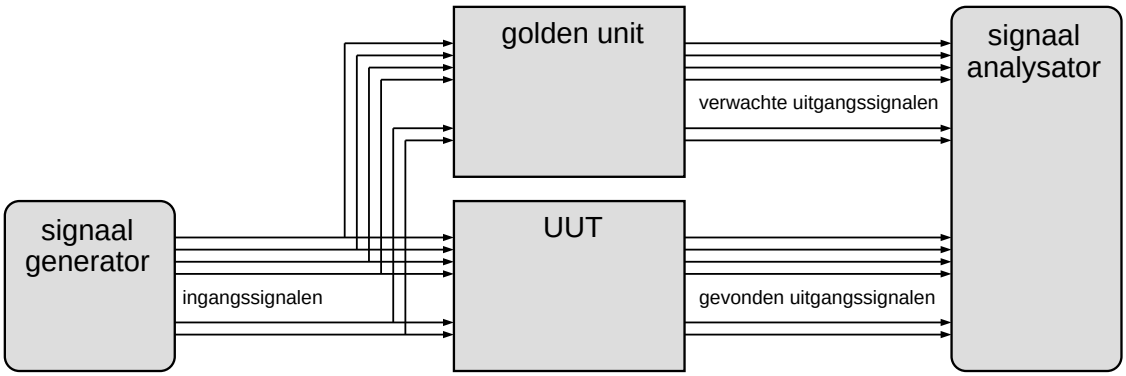
Een signaalanalysator maken op basis van alleen de uitgangssignalen is lastig. In figuur 9.7 staat een voorbeeld van een testconfiguratie met een signaalanalysator die ook de ingangswaarden gebruikt en een voorbeeld van een testbench met een gecombineerde signaalgenerator en -analysator.



Figuur 9.7 : Twee configuraties voor een testbench met analyse op basis van de in- en uitgangssignalen.

De signaalgenerator uit het voorbeeld van figuur 9.8 biedt de ingangssignalen aan aan zowel de *unit under test* als aan een *golden unit*. Dat laatste is een ontwerp dat het correct gedrag heeft. De *golden unit* is bijvoorbeeld een functionele beschrijving van het ontwerp en de *unit under test* is een beschrijving van het gesynthetiseerde ontwerp inclusief de informatie over het tijdsgedrag.

Bij het maken van een beschrijving voor een signaalgenerator en een signaalanalysator wordt vaak tekstinvoer of tekstuitvoer gebruikt. In het uitvoervenster van de simulator worden mededelingen geplaatst, die bijvoorbeeld melden dat een bepaalde test geslaagd of niet geslaagd is. In paragraaf 10.11 wordt het schrijven van een testbench op basis van het *textio*-package en de conversie van integer en vectoren naar tekststrings besproken.



Figuur 9.8 : Een testbench met als referentie een *golden unit*. De *golden unit* is bijvoorbeeld de functionele beschrijving en de *unit under test* het gesynthetiseerde netwerk.

Het report- en assert-statement

Zonder het *textio*-package kan er met het assert- en het report-statement ook informatie naar het uitvoervenster worden geschreven. Het report-statement drukt een tekststring af naar de standaarduitvoer. Proces `compare_result` drukt een mededeling af als de signalen `expect` en `found` verschillend zijn:

```
compare_result: process (expect,found)
begin
  if expect <> found then
    report "De gevonden waarde 'found' wijkt af van de verwachte waarde 'expect'.";
  end if;
end process compare_result;
```

Het assert-statement of assertie kan net als het report-statement in een sequentiële en in een parallelle omgeving worden gebruikt. Onderstaande opdracht is identiek met proces `compare_result`:

```
assert expected = found report "De gevonden waarde 'found' wijkt af van de verwachte waarde 'expect'."
                        severity note;
```

Wanneer de signalen verschillen, drukken beide processen deze mededeling af:

```
Note: De gevonden waarde 'found' wijkt af van de verwachte waarde 'expect'.
```

In beide gevallen staat voor de af te drukken tekst de mededeling *Note:*. De assertie kan een bepaalde ernst — *severity* — hebben. Er zijn vier niveaus: `note`, `warning`, `failure` of `error`. Bij `failure` en `error` stopt de simulatie. Bij `note` en `warning` wordt de assertie uitgevoerd en gaat de simulator verder.

Assertie betekent bevestiging, verklaring of bewering.

De assertie wordt uitgevoerd als de Booleaanse uitdrukking achter **`assert`** *niet* waar is. Achter **`assert`** staat de normale situatie. In dit geval hoort signaal `expect` gelijk te zijn aan signaal `found`. Als dat niet het geval is, wordt de mededeling afgedrukt. De opdracht **`report`** is identiek met een assertie die altijd `false` is en waarvan de severity `note` is.

```
assert false report "mededeling" severity note;
report "mededeling";
```

Bij FPGA's is het gebruik van een SR-latch af te raden. Het is een niet-combinatorische schakeling zonder klokflank. Gebruik in plaats van een SR-latch een D-flipflop met een synchrone set en clear.

De assertie wordt ook in de body van de entity gebruikt. Bij een SR-latch mogen de reset en de set niet allebei actief zijn. Door de assertie in de body van de entity te plaatsen is dit onafhankelijk van de architectuur, zoals code 9.1 laat zien. De beschrijving van de architectuur hoeft nu geen rekening te houden met een actieve set en reset. Bij deze gedragsbeschrijving van de SR-latch is de set dominant.

Code 9.1 : Een SR-latch met een assertie in de body van de entity.

```
1  entity sr_latch is
2    port (
3      s  : in  std_logic;
4      r  : in  std_logic;
5      q  : out std_logic;
6      qn : out std_logic
7    );
8  begin
9    assert s='0' or r='0'
10     report "r and s are both 1. Signal r, or s, or both, must be 0"
11     severity warning;
12 end entity sr_latch;
13
14 architecture gedrag of sr_latch is
15 begin
16   sr: process (s,r) is
17   begin
18     if s='1' then
19       q  <= '1';
20       qn <= '0';
21     elsif r='1' then
22       q  <= '0';
23       qn <= '1';
24     end if;
25   end process sr;
26 end architecture gedrag;
```

9.3 Stimuli

De uitdaging bij het maken van een simulatie is goede testvectoren te bedenken, die het ontwerp zo goed mogelijk testen. Een goede simulatie:

- is zo volledig mogelijk;
- test in ieder geval alle belangrijke, normale situaties;
- test ook de uitzonderlijke situaties, dat zijn vaak situaties die in eerste instantie niet belangrijk lijken;
- is eenvoudig te interpreteren.

Het normale gedrag van het ontwerp moet in ieder geval worden onderzocht: werkt de fifobuffer correct, kunnen er waarden in gezet worden en eruit gehaald worden? Wat gebeurt er als er gelezen wordt uit een lege buffer? Wat gebeurt er als er geschreven wordt naar een volle buffer? Ook als de buffer niet gebruikt wordt om gelijktijdig iets naar de buffer te schrijven en uit de buffer te lezen, is het toch interessant om te weten wat er gebeurt. Door fouten in een ander deel van het systeem zou deze situatie toch op kunnen treden. Daarom zijn uitzonderlijke situaties altijd belangrijk.

9.4 Testbench voor het normale gedrag van de 74163

De 74163 is een *presettable synchronous 4-bit binary counter* met een synchrone reset. In code 9.2 staat de entity van deze teller. Signaal `cp` is de klok, `mr_n` is de actief lage synchrone reset, `pe_n` is de parallelle load, `cep` en `cet` zijn twee enable-signalen. De 4-bits signalen `d` en `q` zijn de ingangsdata en de waarde van de teller Signaal `tc` is de *terminal count* oftewel de *ripple carry output*.

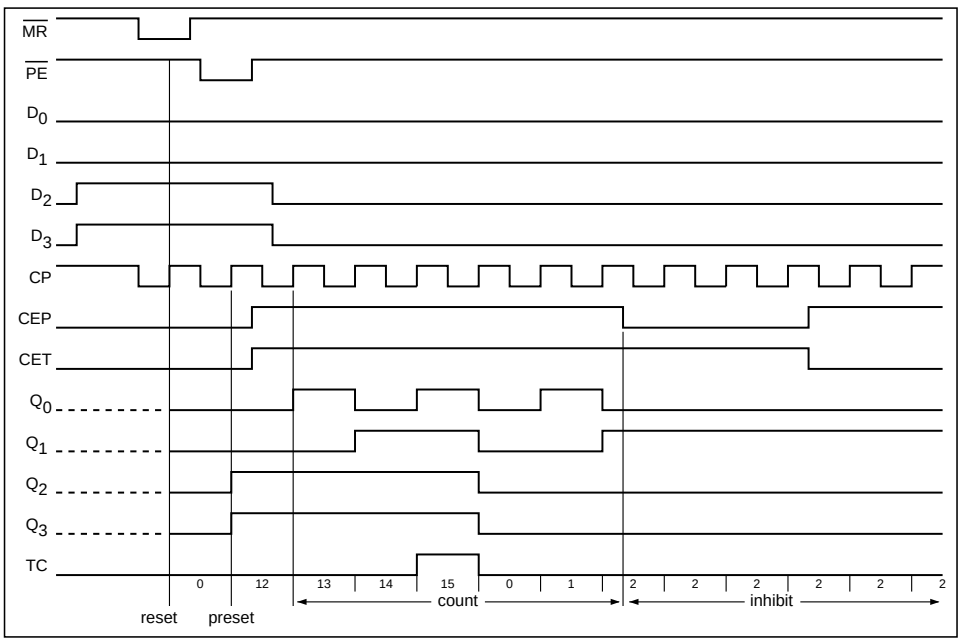
Tabel 9.1 : De functies van de 74163.

mr_n	pe_n	cep	cet	functie
0	–	–	–	clear (q=0)
1	0	–	–	laad (q=d)
1	1	1	1	tellen (q=q+1)
1	1	0	–	vasthouden
1	1	–	0	vasthouden

Code 9.2 : De entity van de 74163.

```
 1  entity w74163 is
 2    port (
 3      cp    : in  std_logic;
 4      mr_n  : in  std_logic;
 5      pe_n  : in  std_logic;
 6      cep   : in  std_logic;
 7      cet   : in  std_logic;
 8      d     : in  std_logic_vector(3 downto 0);
 9      q     : out std_logic_vector(3 downto 0);
10      tc    : out std_logic
11    );
12  end entity w74163;
```

De teller kent vier functies: clearen, laden, tellen en vasthouden. Deze functies worden geselecteerd met de ingangen `mr_n`, `pe_n`, `cep` en `cet`. In tabel 9.1 staan de relaties tussen deze signalen en de vier functies. Figuur 9.9 geeft het signaaldiagram uit de datasheet. Dit diagram is de basis voor de test. Achtereenvolgens worden de vier situaties uit de tabel geëvalueerd. Eerst wordt de teller gecleard; daarna wordt de waarde 12 in de teller gezet; vervolgens wordt er geteld tot de teller op 2 staat en tenslotte wordt deze waarde vastgehouden door `cep` of `cet` laag te maken. Tijdens het tellen is te zien dat bij 15 de uitgang `tc` hoog is.



Figuur 9.9 : Het signaaldiagram uit de datasheet van de 74163.

In code 9.3 staat het deel van de testbench dat deze ingangssignalen genereert. De stimuli komen overeen met die van het signaaldiagram uit de datasheet. Het simulatieresultaat zal, als de gedragsbeschrijving correct is, exact overeenkomen met het signaaldiagram uit de datasheet.

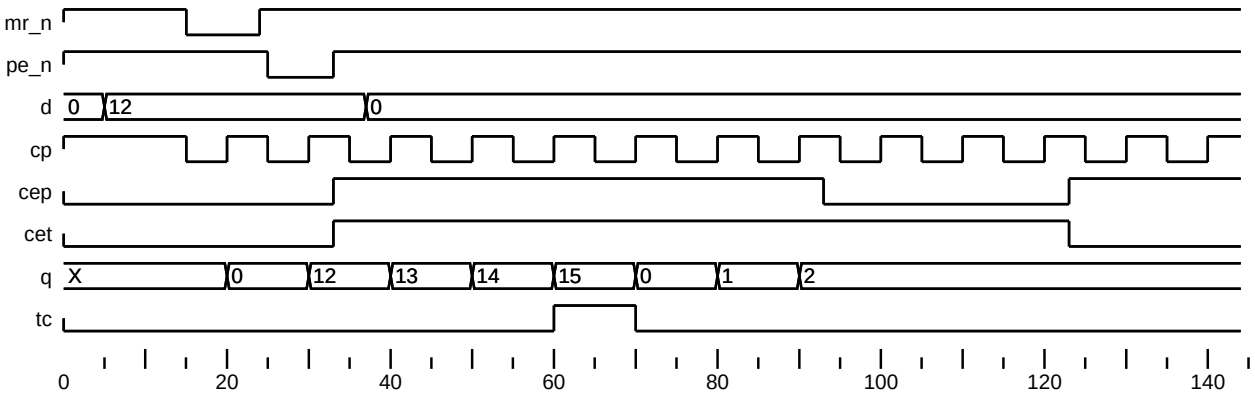
De klok `cp` wordt apart van de datasignalen gegenereerd. De klokperiode is als een constante `CLOCK_PERIOD` gedefinieerd. De periode van 10 ns komt overeen met een frequentie van 100 MHz. De opzet van de test is dat deze het signaaldiagram uit de datasheet reproduceert. Daarin is de klok aanvankelijk niet actief. In de test is dit bereikt met een continu actieve klok `clk` en een enable `cp_enable`. Na anderhalve klokslag (`1.5*CLOCK_PERIOD`) wordt de enable actief en daarmee wordt de klokpuls `cp` ook actief.

Code 9.3 : Het deel van de testbench voor de 74163 met de signaalgenerator.

```
30    constant CLOCK_PERIOD : time := 10 ns;
31
32    signal clk       : std_logic := '1';
33    signal cp_enable : std_logic;
34  begin
35    clk       <= not clk after 0.5*CLOCK_PERIOD;
36    cp_enable <= '0', '1' after 1.5*CLOCK_PERIOD;
37    cp        <= clk when cp_enable='1' else '1';
38
39    signal_generator: process is
40    begin
41      mr_n <= '1';                -- initialize
42      pe_n <= '1';
43      cep  <= '0';
44      cet  <= '0';
45      d    <= "0000";
46      wait for 0.5*CLOCK_PERIOD;
47      d    <= "1100";
48      wait for CLOCK_PERIOD;
49      mr_n <= '0';                -- clear
50      wait for 0.9*CLOCK_PERIOD;
51      mr_n <= '1';
52      wait for 0.1*CLOCK_PERIOD;
53      pe_n <= '0';                -- parallel load
54      wait for 0.8*CLOCK_PERIOD;
55      pe_n <= '1';                -- enable
56      cep  <= '1';
57      cet  <= '1';
58      wait for 0.4*CLOCK_PERIOD;
59      d    <= "0000";
60      wait for 5.6*CLOCK_PERIOD;
61      cep  <= '0';                -- hold
62      wait for 3*CLOCK_PERIOD;
63      cep  <= '1';                -- hold
64      cet  <= '0';
65      wait;
66    end process signal_generator;
```

In dit voorbeeld zijn de tijdvertragingen in de signaalgenerator zo gekozen dat de signaaltransities overeenkomen met die uit het signaaldiagram van de datasheet. Het gevolg is dat de beschrijving wat ingewikkelder is, dan strikt genomen nodig is.

Alle andere ingangssignalen van de teller worden beschouwd als data en krijgen de waarden die het proces `signal_generator` genereert. Eerst worden alle signalen geïnitialiseerd: de stuursignalen zijn inactief en `d` heeft de waarde `0`. Nadat `d` 12 gemaakt is, worden de verschillende functies van de teller getest: vanaf regel 49 wordt de teller gecleard, na regel 53 wordt de waarde van `d` geladen, en na regel 55 zijn `cep` en `cet` actief en telt de teller. Tenslotte wordt vanaf regel 61 de teller gestopt door eerst `cep` laag en daarna `cet` laag te maken.



Figuur 9.10 : Het simulatieresultaat van de 74163.

Het resultaat van de simulatie staat in figuur 9.10 en komt overeen met het signaaldiagram uit figuur 9.9. Figuur 9.10 toont niet de losse bits van de 4-bits signalen `d` en `q` maar de decimale waarde van deze vier bits.

Code 9.4 : De gedragsbeschrijving van de 74163.

```
1  architecture gedrag of w74163 is
2    signal qi : unsigned(3 downto 0);
3  begin
4    count: process (cp) is
5    begin
6      if rising_edge(cp) then
7        if mr_n = '0' then
8          qi <= (others => '0');
9        elsif pe_n = '0' then
10         qi <= unsigned(d);
11       elsif cep='1' and cet='1' then
12         qi <= qi + 1;
13       end if;
14     end if;
15   end process count;
16
17   q  <= std_logic_vector(qi);
18   tc <= cet and qi(3) and qi(2) and qi(1) and qi(0);
19 end architecture gedrag;
```

Code 9.5 : Een foutieve beschrijving van de 74163.

```
1  architecture gedrag of w74163 is
2    signal qi : unsigned(3 downto 0);
3  begin
4    count_wrong: process (cp) is
5    begin
6      if rising_edge(cp) then
7        if cep='1' and cet='1' then
8          qi <= qi + 1;
9        elsif pe_n = '0' then
10         qi <= unsigned(d);
11       elsif mr_n = '0' then
12         qi <= (others => '0');
13       end if;
14     end if;
15   end process count_wrong;
16
17   q  <= std_logic_vector(qi);
18   tc <= qi(3) and qi(2) and qi(1) and qi(0);
19 end architecture gedrag;
```

9.5 Testbench voor het niet normale gedrag van de 74163

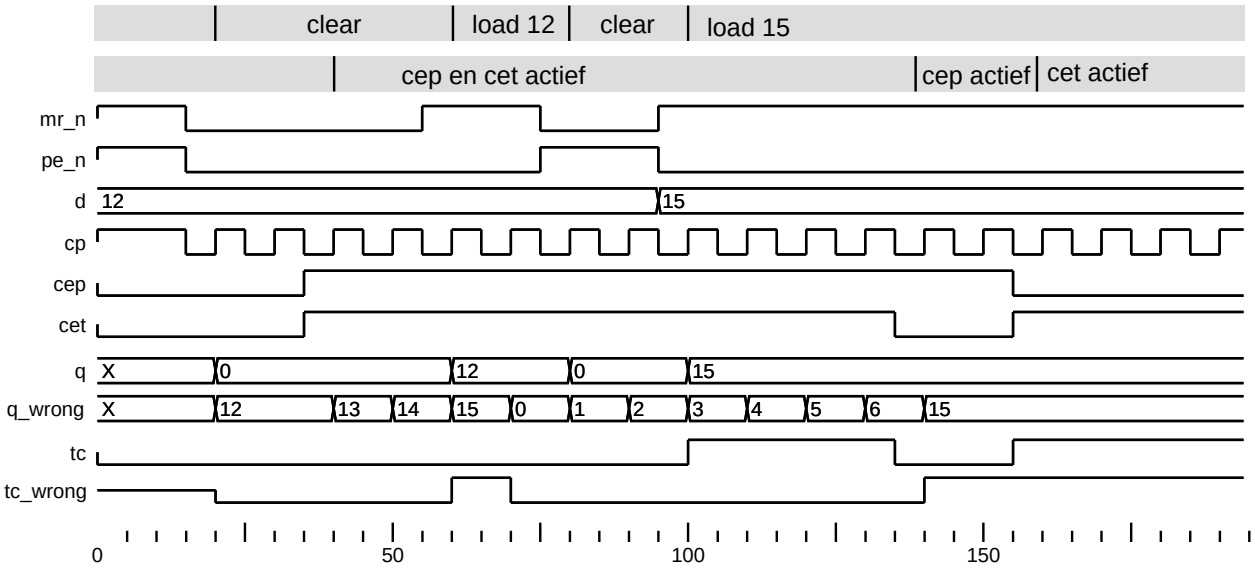
De test laat zien dat het gewone gedrag van de teller klopt. Dat niet alle waarden van de teller bestudeerd worden, is geen probleem. De meest interessante waarde — de waarde 15 — en de belangrijkste overgang — die van 15 naar 0 — zijn onderzocht. Bij 15 is het signaal tc hoog en na 15 gaat de teller verder met 0.
Het resultaat uit figuur 9.10 toont alleen het *normale* gedrag. De teller telt, cleart, laadt en houdt vast. De vraag is of de teller ook goed functioneert in bijzondere omstandigheden. Wat gebeurt er als mr_n en pe_n beide actief zijn en is tc alleen hoog als de teller 15 is en cet actief is?
In code 9.4 en code 9.5 staan twee beschrijvingen van de 74163. De eerste heeft de juiste prioriteit voor de stuursignalen. Bij de tweede wijkt de prioriteit af en hangt het uitgangssignaal tc niet van cet af.

De testbench uit code 9.3 geeft voor de foutieve beschrijving exact hetzelfde resultaat als voor de juiste beschrijving. Het signaaldiagram van de foute beschrijving is identiek aan het diagram van de juiste beschrijving uit figuur 9.10.
Deze test detecteert dus niet de fouten in de beschrijving van code 9.5. De testbench maakt steeds mr_n of pe_n of de enables actief. Er wordt niet gekeken naar wat er gebeurt als bijvoorbeeld mr_n en pe_n beide actief zijn. Bovendien wordt uitgang tc alleen getest tijdens het tellen als cet eveneens hoog is.

Code 9.6 : Signaalgenerator voor het niet normale gedrag.

```
39   signal_generator: process is
40   begin
41     mr_n <= '1';
42     pe_n <= '1';
43     cep  <= '0';
44     cet  <= '0';
45     d    <= "1100";
46     wait for 1.5*CLOCK_PERIOD;
47     mr_n <= '0';                -- clear and load
48     pe_n <= '0';
49     wait for 2*CLOCK_PERIOD;
50     cep  <= '1';                -- clear and load and enable
51     cet  <= '1';
52     wait for 2*CLOCK_PERIOD;
53     mr_n <= '1';                -- load and enable
54     wait for 2*CLOCK_PERIOD;
55     mr_n <= '0';                -- clear and enable
56     pe_n <= '1';
57     wait for 2*CLOCK_PERIOD;
58     d    <= "1111";             -- load and enable
59     mr_n <= '1';
60     pe_n <= '0';
61     wait for 4*CLOCK_PERIOD;
62     cet  <= '0';                -- load and hold
63     wait for 2*CLOCK_PERIOD;
64     cet  <= '1';                -- load and hold
65     cep  <= '0';
66     wait;
67   end process signal_generator;
```

De signaalgenerator uit code 9.6 test het niet normale gedrag. Voor verschillende situaties wordt onderzocht of de prioriteit van de functies van de teller correct is. Vanaf regel 58 worden een aantal bijzondere condities van uitgang `tc` getest.
Het signaaldiagram van figuur 9.11 geeft zowel de uitgangen van de correcte beschrijving uit code 9.4 als die van de foute beschrijving uit code 9.5. Het diagram toont aan dat de foute beschrijving afwijkt van de correcte beschrijving.



Figuur 9.11 : **Het simulatieresultaat voor het niet normale gedrag van de 74163.** De bovenste grijze balk laat het juiste gedrag zien. Signaal `q`, de uitgang van de correcte beschrijving, voldoet hieraan. Signaal `q_wrong`, de uitgang van de foute beschrijving, vertoont een ander gedrag. Bij de foute beschrijving is de clear niet dominant en is de parallel load niet dominant boven de enable-signalen.
De onderste grijze balk laat zien wanneer `cep` en `cet` actief zijn. Bij de correcte beschrijving wordt `tc` hoog als `q` 15 is en als `cet` tegelijkertijd hoog is. Bij de foute beschrijving is `tc_wrong` ook hoog als `q` 15 is en `cet` laag is.

De signaalgenerator uit code 9.6 vindt fouten, die de signaalgenerator uit code 9.3 niet vindt. Om alle mogelijke fouten in een ontwerp te vinden, moet de ontwerper niet alleen het normale gedrag, maar ook het niet normale gedrag, testen.
In de praktijk zal de teller alleen gebruikt worden onder normale condities. Als de component onder niet normale omstandigheden gebruikt wordt en de component niet geteste ontwerpfouten bevat, kan dat tot gevolg hebben dat het complete systeem niet goed functioneert. Achteraf, op een later moment in het ontwerptraject, zijn deze fouten lastig terug te vinden.

9.6 Klok, asynchrone reset en data

Bij de test van de 74163 is geprobeerd het signaaldiagram uit de datasheet na te maken. De klokgeneratie is daardoor relatief ingewikkeld. Bovendien heeft de teller geen asynchrone reset. De meeste testbenches hebben een eenvoudigere klok en een asynchrone reset.

Een klok `clk` kan worden gegenereerd met een concurrent signal assignment:

```
clk <= not clk after HALF_CLOCK_PERIOD;
```

De constante `HALF_CLOCK_PERIOD` moet net als het signaal `clk` in de testbench gedeclareerd worden:

```
constant HALF_CLOCK_PERIOD : time := 20 ns;
signal clk : std_logic := '0';
```

In dit geval begint de klok laag en wordt alternerend na 20 ns hoog en weer laag. De klokperiode is dan 40 ns en de bijbehorende klokfrequentie is 25 MHz.
Als de asynchrone reset `rst_n` actief laag is, wordt dit signaal gegenereerd met:

```
rst_n <= '0', '1' after RESET_PERIOD;
```

De constante `RESET_PERIOD` is dan:

```
constant RESET_PERIOD : time := 50 ns;
```

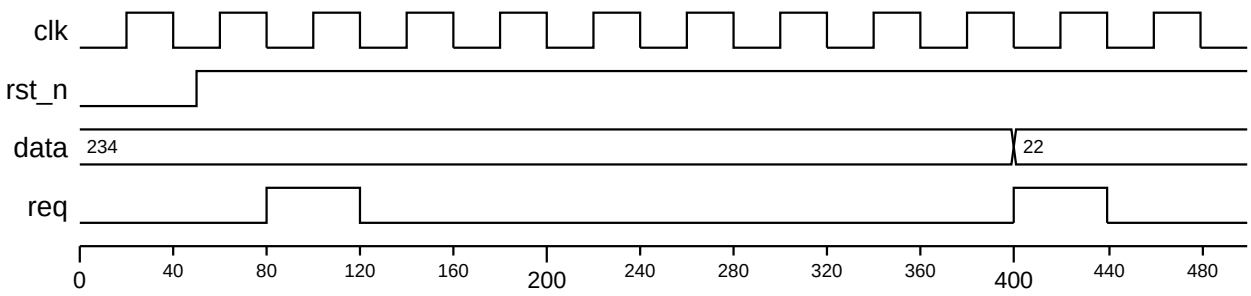
De resetperiode is in dit voorbeeld 50 ns. Het resetsignaal is nodig om het systeem in een gedefinieerde toestand te brengen. Daarom is de reset direct actief bij de start van de simulatie. In dit voorbeeld wordt de reset na 50 ns losgelaten. De duur van de reset is bij een functionele simulatie zonder tijdvertragingen niet belangrijk. Het is verstandig om het loslaten van de reset niet samen te laten vallen met een klokflank. Dit kan tot gevolg hebben dat het niet duidelijk is of het systeem in de resettoestand begint of dat er al een klokslag is geweest.

Code 9.7 : Voorbeeld van een signaalgenerator.

```
signal_generator: process is
begin
  data <= std_logic_vector(to_unsigned(234,data'length));
  req  <= '0';
  wait for 2*CLOCK_PERIOD;
  req  <= '1';
  wait for CLOCK_PERIOD;
  req  <= '0';
  wait for 7*CLOCK_PERIOD;
  data <= std_logic_vector(to_unsigned(22,data'length));
  req  <= '1';
  wait for CLOCK_PERIOD;
  req  <= '0';
  wait;
end process signal_generator;
```

De andere ingangssignalen kunnen door één of meer signaalgeneratoren geproduceerd worden. Meestal is er een bepaald patroon waarmee de ingangssignalen worden aangeboden. Het is dan handig om één signaalgenerator te gebruiken, zoals die uit code 9.6 en code 9.7. De constante `CLOCK_PERIOD` uit code 9.7 is twee keer de halve klokperiode `HALF_CLOCK_PERIOD` en dus gelijk aan 40 ns.

In figuur 9.12 staat het signaaldiagram met de ingangssignalen. De signalen `data` en `req` veranderen steeds bij de neergaande klokflank. Bij de derde klokflank, dus bij 100 ns, is het signaal `req` actief. Acht klokslagen verder, bij 420 ns, is dit signaal opnieuw actief. Blijkbaar zijn voor de *unit under test* acht klokslagen voldoende om de volgende data te verwerken en de volgende test uit te voeren.



Figuur 9.12 : De klok, de reset en de signalen, die door de signaalgenerator worden gegenereerd.

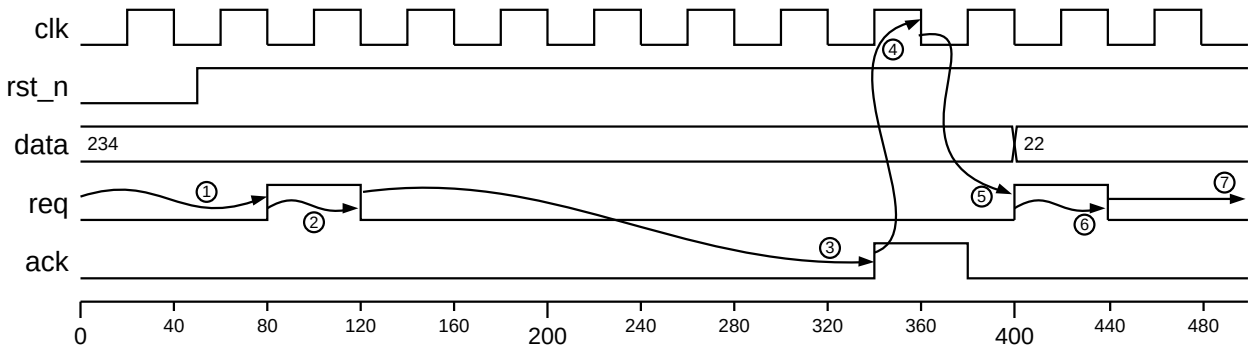
In code 9.7 ligt het initiatief voor het aanleveren van nieuwe gegevens bij de signaalgenerator. Deze signaalgenerator werkt alleen goed als de *unit under test* met de verwerking van de gegevens klaar is, voordat er nieuwe gegevens worden aangeboden.
Bij de signaalgenerator uit code 9.8 ligt het initiatief voor het aanleveren bij de *unit under test*. Deze component heeft een uitgangssignaal `ack`, dat een klokslag hoog is als de verwerking van de gegevens klaar is. De signaalgenerator uit code 9.8 heeft zeven wachtopdrachten. Bij de derde wachtopdracht, na het laag worden van `req`, wacht de generator op het hoog worden van signaal `ack`. Bij de vierde **`wait`** wordt er gewacht op de neergaande klokflank. De signalen `data` en `req` worden bij de volgende neergaande klokflank actief en bij de daaropvolgende opgaande klokflank is `req` weer actief.

Code 9.8 : Voorbeeld signaalgenerator2.

De wait-statements bij de cijfers in code 9.8 komen overeen met de pijlen bij de cijfers in figuur 9.13.

```
signal_generator2: process is
begin
  data <= std_logic_vector(to_unsigned(234,data'length));
  req  <= '0';
  wait for 2*CLOCK_PERIOD;                                  -- 1
  req  <= '1';
  wait for CLOCK_PERIOD;                                    -- 2
  req  <= '0';
  wait until ack='1';                                       -- 3
  wait until falling_edge(clk);                             -- 4
  wait for CLOCK_PERIOD;                                    -- 5
  data <= std_logic_vector(to_unsigned(22,data'length));
  req  <= '1';
  wait for CLOCK_PERIOD;                                    -- 6
  req  <= '0';
  wait;                                                     -- 7
end process signal_generator2;
```

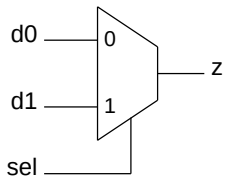
Bij de signaalgenerator van code 9.8 gaat het aanbieden van gegevens in principe vanzelf goed. Alleen als er fouten in het ontwerp of in de signaalgenerator zitten, stopt de simulatie en krijgt de ontwerper ook geen informatie over de rest van de test. Bij de signaalgenerator van code 9.7 gaat de simulatie gewoon door en geeft de rest van de simulatie ook informatie over de werking van de generator en van de te testen component.



Figuur 9.13 : Het signaaldiagram met `signal_generator2`. De pijlen met nummers komen overeen met de **`wait`**'s van het proces `signal_generator2` uit code 9.8.

9.7 Test multiplexer: observeerbaarheid versus aanstuurbaarheid

Een multiplexer is een combinatorische schakeling met drie ingangen en een uitgang. Figuur 9.14 geeft het symbool. In tabel 9.2 staat de functietabel. Als ingang `sel` hoog is, wordt ingang `d1` doorgegeven aan uitgang `z` en als `sel` laag is, wordt `d0` doorgegeven.



Figuur 9.14 : Het symbool van een multiplexer.

Tabel 9.2 : De functietabel van de multiplexer.

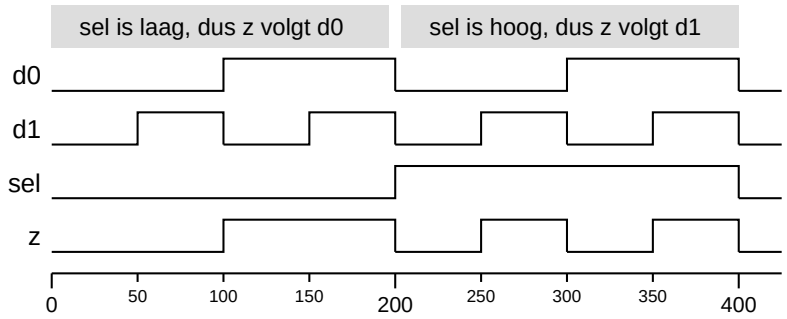
sel	d0	d1	z
0	0	-	0
0	1	-	1
1	-	0	0
1	-	1	1

Code 9.9 toont een signaalgenerator om de multiplexer te testen. Alle acht ingangscombinaties worden gegenereerd. De test is dus volledig. De volgorde van de testvectoren is zo gekozen dat de interpretatie van het simulatieresultaat zo eenvoudig mogelijk is.

Code 9.9 : Signaalgenerator multiplexer.

```
45    signal_generator_mux: process is
46    begin
47      sel <= '0';
48      d0  <= '0';
49      d1  <= '0';
50      wait for DELAY;
51      d1  <= '1';
52      wait for DELAY;
53      d0  <= '1';
54      d1  <= '0';
55      wait for DELAY;
56      d1  <= '1';
57      wait for DELAY;
58      sel <= '1';
59      d0  <= '0';
61      d1  <= '0';
62      wait for DELAY;
63      d1  <= '1';
64      wait for DELAY;
65      d0  <= '1';
66      d1  <= '0';
67      wait for DELAY;
68      d1  <= '1';
69      wait for DELAY;
70      sel <= '0';
71      d0  <= '0';
72      d1  <= '0';
73      wait;
74    end process signal_generator_mux;
75
```

Figuur 9.15 toont het simulatieresultaat. De constante `DELAY` is 50 ns genomen. In de figuur is duidelijk te zien dat `z` signaal `d0` volgt als `sel` laag is en dat het signaal `d1` volgt als `sel` hoog is.



Figuur 9.15 : Het simulatieresultaat van de multiplexer. Als `sel` laag is, volgt de uitgang `z` signaal `d0` en als `sel` hoog is, volgt `z` signaal `d1`.

Ogenschijnlijk is de test uit code 9.9 volledig en correct. Toch ontdekt deze test niet alle fouten in de beschrijving. In code 9.10 en 9.11 staan twee beschrijvingen van de multiplexer.

Code 9.10 : Correcte beschrijving multiplexer.

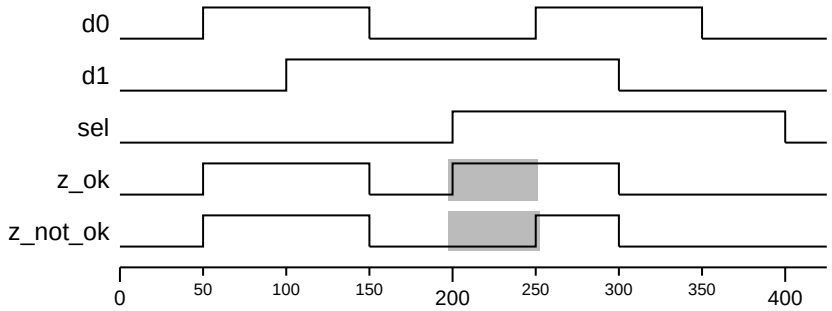
```
mux_ok : process (sel,d0,d1) is
begin
  z <= ((not sel) and d0) or (sel and d1);
end process mux_ok;
```

Code 9.11 : Multiplexer met niet volledige sensitivity list.

```
mux_not_ok: process (d0,d1) is
begin
  z <= ((not sel) and d0) or (sel and d1);
end process mux_not_ok;
```

Beide gebruiken een expliciet proces met een gevoeligheidslijst. Het proces `mux_ok` uit code 9.10 is correct. Het proces `mux_not_ok` uit code 9.11 is niet correct: het signaal `sel` ontbreekt aan de gevoeligheidslijst.

De simulatie van de foute beschrijving met de signaalgenerator uit code 9.9 geeft — ondanks het feit dat het signaal `sel` aan de gevoeligheidslijst ontbreekt — exact hetzelfde signaaldiagram. De fout wordt niet gedetecteerd, ondanks het feit dat alle mogelijke ingangscombinaties zijn aangeboden.

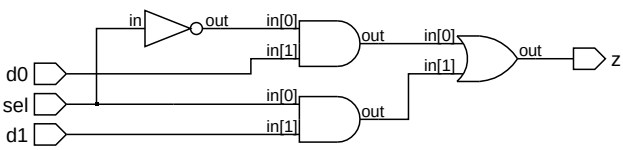


Figuur 9.16 : Het simulatieresultaat van de multiplexer voor de test met de Gray-code. Op de plaats van de grijze vlakken verschilt de correcte beschrijving van de foute beschrijving.

De fout kan alleen gevonden worden bij een verandering van `sel`. Signaal `sel` verandert slechts twee keer van waarde. Bij het begin op tijdstip `0` ns wordt `sel` laag gemaakt en halverwege op tijdstip 200 ns wordt `sel` hoog gemaakt. Op die momenten veranderen `d0` en `d1` ook. Bij de foute beschrijving wordt het proces `mux_not_ok` getriggerd door de overgangen van `d0` en `d1`. Het proces wordt gewoon uitgevoerd en de juiste waarde van `z` wordt berekend.

De Gray-code is een binaire codering waarbij twee opeenvolgende getallen altijd slechts één bit verschillen. Er zijn altijd meerdere Gray-coderingen mogelijk.

Om dit soort fouten te vinden, is het nodig dat de signaalgenerator altijd maar één signaal tegelijkertijd veranderd. In code 9.12 verandert elke keer slechts één van de signalen `d0`, `d1` of `sel`. De testvectoren veranderen volgens een Gray-code.



Figuur 9.17 : De RTL-view van de multiplexer.

Code 9.12 : Signaalgenerator multiplexer met gray-code.

```
45    signal_generator_mux_gray: process is
46    begin
47      sel <= '0';
48      d0  <= '0';
49      d1  <= '0';
50      wait for DELAY;
51      d0  <= '1';
52      wait for DELAY;
53      d1  <= '1';
54      wait for DELAY;
55      d0  <= '0';
56      wait for DELAY;
57      sel <= '1';
58      wait for DELAY;
59      d0  <= '1';
60      wait for DELAY;
61      d1  <= '0';
62      wait for DELAY;
63      d0  <= '0';
64      wait for DELAY;
65      sel <= '0';
66      wait;
67    end process signal_generator_mux_gray;
```

De meeste synthesizers, zoals Leonardo Spectrum, Quartus en Precision RTL, genereren voor de foute beschrijving van code 9.11 het netwerk van figuur 9.17 en geven daarbij een waarschuwing.
Leonardo Spectrum meldt bijvoorbeeld: `"Warning, sel should be declared on the sensitivity list of the process."`.

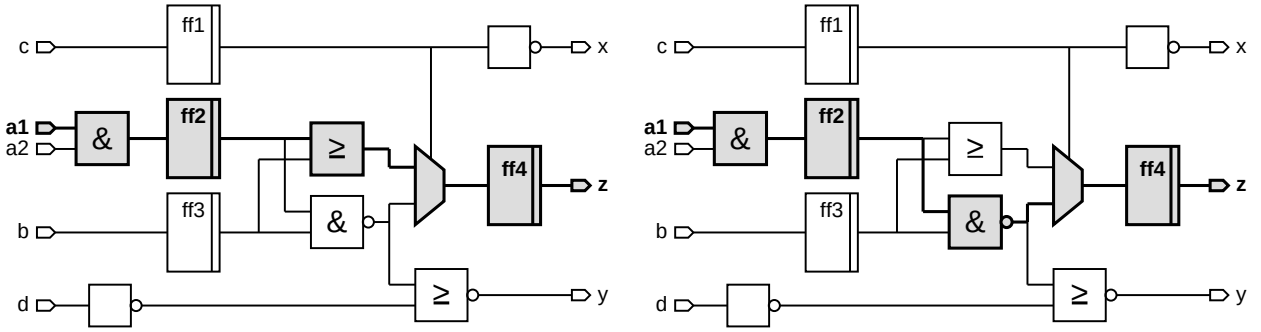
Figuur 9.16 geeft het resultaat van deze test voor zowel de correcte als voor de foute beschrijving van de multiplexer. Bij 200 ns verandert signaal `sel` en blijft de uitgang bij de foute beschrijving ten onrechte laag. Dit komt omdat process `mux_not_ok` niet gevoelig is voor veranderingen van signaal `sel`.
De test met `signal_generator_mux_gray` detecteert deze fout in de gevoeligheidslijst, maar in tegenstelling tot de test met `signal_generator_mux` is het correct functioneren veel lastiger te herkennen. Bij `signal_generator_mux` is één oogopslag zichtbaar dat als `sel` hoog is de uitgang `d1` volgt en dat als `sel` laag is de uitgang signaal `d0` met de dubbele frequentie van `d1` volgt. Dat is bij figuur 9.16 niet het geval.

Observeerbaarheid en aanstuurbaarheid kunnen elkaar dus tegenwerken. Een test, die eenvoudig te interpreteren is, is soms niet volledig en een test, die alle situaties op een juiste wijze test, kan lastig te interpreteren zijn. Duidelijk is dat een simulatie nooit kan bewijzen dat een systeem goed werkt. Een simulatie maakt hooguit aannemelijk dat het systeem correct functioneert.

9.8 Statische tijdsanalyse

Statische tijdsanalyse oftewel STA, *Statical Timing Analysis* is een methode om te onderzoeken of een systeem aan de gestelde tijdseisen voldoet, zonder het te simuleren. Hierbij wordt niet naar het functionele gedrag gekeken. Statische tijdsanalyse gaat veel sneller dan simulatie en een testbench is niet nodig.

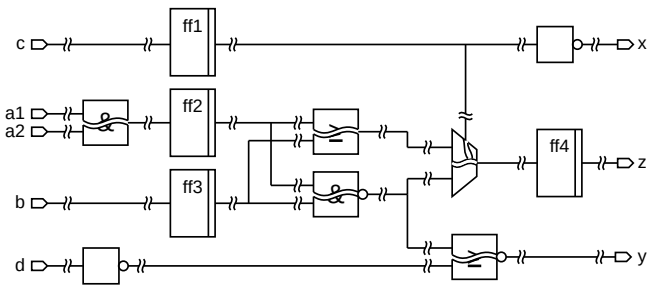
Figuur 9.18 toont een schakeling met twee verschillende signaalpaden. De tijd die er voor een signaalverandering nodig is, hangt af van de tijdvertraging van de logische poorten langs deze paden. Voor een CMOS-technologie zal de vertraging via de NAND kleiner zijn dan via de NOR-poort.



Figuur 9.18 : Een digitaal systeem met twee verschillende signaalpaden. In het linker figuur loopt het pad via de nor-poort en in de rechter figuur via de nand-poort.

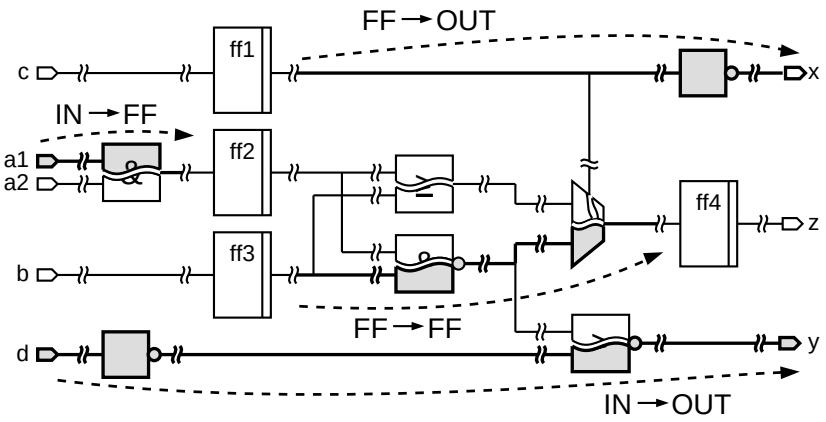
Een statische tijdsanalyse kan alleen worden uitgevoerd als het netwerk bekend is. Bij FPGA's kan deze analyse pas worden uitgevoerd na de synthese. Het te evalueren netwerk bestaat bij FPGA's uit LUT's en flipfloppen. Na het plaatsen en bedraden, de zogenoemde *technology mapping*, zijn de vertragingen ten gevolge van de bedrading ook beschikbaar. De statische tijdsanalyse wordt daarom in het algemeen pas uitgevoerd na de plaatsing en de bedrading.

Figuur 9.19 laat zien dat het te beschouwen netwerk in stukken geknipt wordt, waarvan de tijdvertraging bekend is. Voor alle signaalpaden kan dan het tijdsgedrag worden geëvalueerd.




```
c → ff1 → inv → x
c → ff1 → mux → ff4 → z
a1 → and → ff2 → or → mux → ff4 → z
a1 → and → ff2 → nand → mux → ff4 → z
a1 → and → ff2 → nand → nor → y
a2 → and → ff2 → or → mux → ff4 → z
a2 → and → ff2 → nand → mux → ff4 → z
a2 → and → ff2 → nand → nor → y
b → ff3 → or → mux → ff4 → z
b → ff3 → nand → mux → ff4 → z
b → ff3 → nand → nor → y
d → inv → nor → y
```

Figuur 9.19 : Het netwerk wordt in stukken geknipt. Bij ieder stuk hoort een bepaalde tijdvertraging. In dit voorbeeld zijn elf verschillende paden te herkennen.



Figuur 9.20 : **Het voorbeeld uit deze paragraaf met de vier verschillende soorten paden.** Ingang naar flipflop, ingang naar uitgang, flipflop naar uitgang en flipflop naar flipflop.

Het kritieke pad en de maximale klokfrequentie

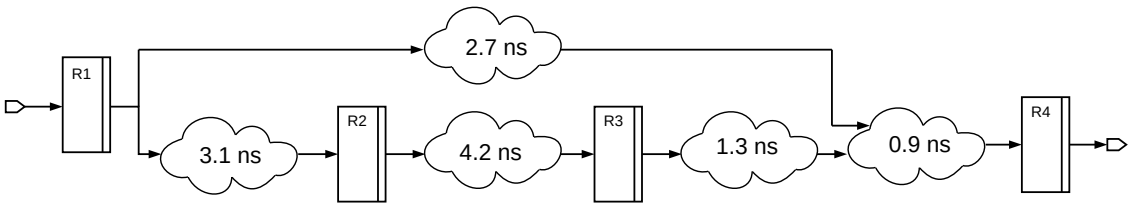
Clock skew is het verschijnsel dat een kloksignaal door looptijdverschillen op verschillende momenten bij de flipfloppen aankomt.

Figuur 9.19 suggereert dat alle paden van iedere ingang naar iedere uitgang apart onderzocht worden. Voor complexe systemen zou dat betekenen dat er oneindig veel paden onderzocht moeten worden. De statisch tijdsanalyse onderzoekt daarom geen complete paden, maar delen van paden. In het netwerk worden vier verschillende soorten paden onderscheiden:

- van een ingang naar een flipflop of register
- van een ingang naar een uitgang
- van een flipflop of register naar een uitgang
- van een flipflop of register naar een flipflop of register

De setuptijd of *setup time* is de tijd dat een ingangssignaal bij een flipflop stabiel moet zijn voor de actieve klokflank. De houdtijd of *hold time* is de tijd dat een ingangssignaal bij een flipflop stabiel moet zijn na de actieve klokflank.

Met name het laatste type is belangrijk voor het tijdsgedrag. Als de zogenoemde *clock skew* en de setup- en houdtijden van de flipfloppen en registers buiten beschouwing worden gelaten, is de maximaal haalbare klokfrequentie omgekeerd evenredig met het traagste pad tussen twee flipfloppen.



Figuur 9.21 : **De tijdvertragingen langs twee verschillende signaalpaden.** De grootste vertragingstijd bevindt zich tussen register 2 en register 3 en is gelijk aan 4,2 ns.

Figuur 9.21 toont een vereenvoudigd voorbeeld met twee signaalpaden. Bij de rechtstreekse verbinding tussen register R1 en register R4 is de vertraging 3,6 ns. Langs register R2 en R3 is de totale vertraging 9,5 ns. Deze tijd is niet relevant voor de snelheid van het systeem.

De signalen bij de registers moeten stabiel zijn voor de volgende klokslag. Het pad met de grootste tijdvertraging tussen twee registers of flipfloppen wordt het kritieke pad, *critical path*, genoemd. De maximale frequentie f_{max} van het systeem is omgekeerd evenredig met de som van de tijdvertraging t_{delay} van het kritieke pad, de setup van het tweede register t_{su} en de vertraging $t_{clk \rightarrow q}$ van het eerste register:

In het Nederlands wordt *critical path* vaak ten onrechte vertaald met kritisch(e) pad. Kritiek(e) pad is de juiste vertaling.

$$f_{max} = \frac{1}{t_{clk \rightarrow q} + t_{su} + t_{delay}} \tag{9.1}$$

Eventuele clock-skew is bij formule 9.1 buiten beschouwing gelaten.

De tijdvertraging tussen twee opeenvolgende registers is tussen register R2 en R3 het grootst. De minimale tijd waarmee de registers geklokt kunnen worden, is met een t_{su} van 0,8 ns gelijk aan 5,0 ns. De maximale frequentie is dan 200 MHz. Een timing analyzer genereert een lijst met de kritieke paden. De ontwerper geeft voor de analyse een gewenste klokfrequentie op. Het resultaat van de statische tijdsanalyse voor het netwerk uit figuur 9.21 staat in tabel 9.3. In dit geval is voor de gewenste klokfrequentie 125 MHz genomen en is de klokperiode 8,0 ns.

In dit voorbeeld is $t_{clk \rightarrow q}$ verwaarloosd, hetgeen bij FPGA's meestal gebeurd.

Tabel 9.3 : Het resultaat van een statische tijdsanalyse.

index	slack	delay	start pin	end pin
1	3.0	4.2	R2.clk	R3.d
2	3.6	3.6	R1.clk	R4.d
3	4.1	3.1	R1.clk	R2.d
4	5.0	2.2	R3.clk	R4.d

In de tweede kolom van de tabel staat de zogenoemde *slack*. Dat is het verschil tussen de klokperiode en de tijd die er voor de bewerking nodig is:

$$\text{slack} = \frac{1}{f_{clk}} - t_{su} - t_{delay} \tag{9.2}$$

Slack laat zich het beste vertalen met speling of marge. Het is de ruimte tussen de beschikbare tijd en de benodigde tijd.

Het kritieke pad is het pad met de kleinste slack.

Latency

Een ander belangrijk begrip is de zogenoemde latentie of *latency*. Dat is de tijd die nodig is om informatie van de ingang naar de uitgang van een systeem te brengen. In figuur 9.21 zijn drie klokslagen nodig om de informatie van register R1 via R2 en R3 naar register R4 te krijgen. Bij een gewenste klokfrequentie van 125 MHz is dat 24 ns. Voor de maximaal haalbare klokfrequentie van 200 MHz is de latentie 15 ns.

Voor- en nadelen van statische tijdsanalyse

De voordelen van een statische tijdsanalyse of *timing analysis* ten opzichte van simulatie zijn:

- Alle paden worden meegenomen. Bij simulatie is dat voor complexe systemen niet mogelijk.
- Het analyseren gaat veel sneller dan een simulatie.
- Er hoeft geen testbench te worden gemaakt.

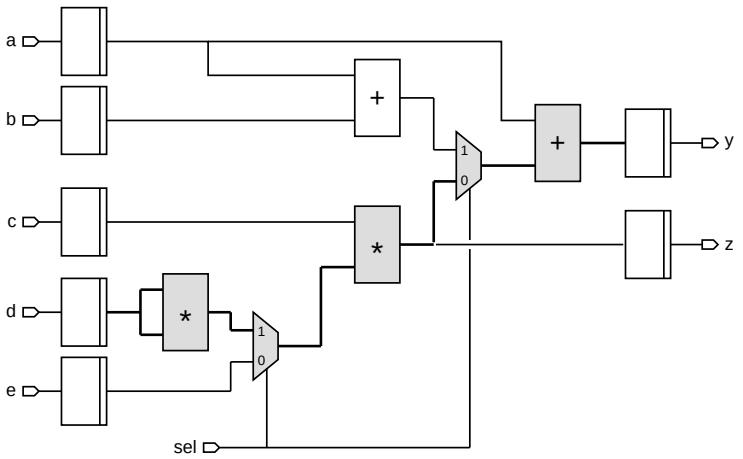
- Er kan gelijktijdig een *worst* en *best case* analyse worden gemaakt.
- De grootste tijdvertraging wordt altijd gevonden. De maximale frequentie waarbij het systeem nog functioneert is daardoor bekend.

Nadelen van een statische tijdsanalyse of *timing analysis* zijn:

- Niet alle paden hoeven functioneel voor te komen. Het ontwerp kan zogenoemde *false paths* bevatten. De analyse kan daardoor te pessimistisch zijn.
- Voor het signaalpad tussen twee registers kunnen meer klokslagen gebruikt worden. Deze zogenoemde *Multicycle paths* geven een te pessimistische analyse.
- Statische tijdsanalyse controleert het functionele gedrag niet.
- Het is niet geschikt voor asynchrone systemen.
- Het is alleen geschikt voor systemen met één klok. Wel is het mogelijk meerkloksystemen te analyseren als deze kloksignalen afgeleid zijn van een systeemklok.
- Er mogen geen combinatorische lussen in het ontwerp voorkomen.

False paths

Valse paden of *false paths* zijn signaalpaden in een schakeling, die functioneel nooit worden doorlopen. Een voorbeeld van een false path staat in figuur 9.22. Het kritieke pad in deze schakeling ligt tussen het register van ingang `d` en het register van uitgang `y`. Functioneel wordt dit pad nooit doorlopen. Als ingang `sel` hoog is, is `y` gelijk aan `a+b+a` en als de ingang laag is, is uitgang `y` gelijk aan `a+c+e`.



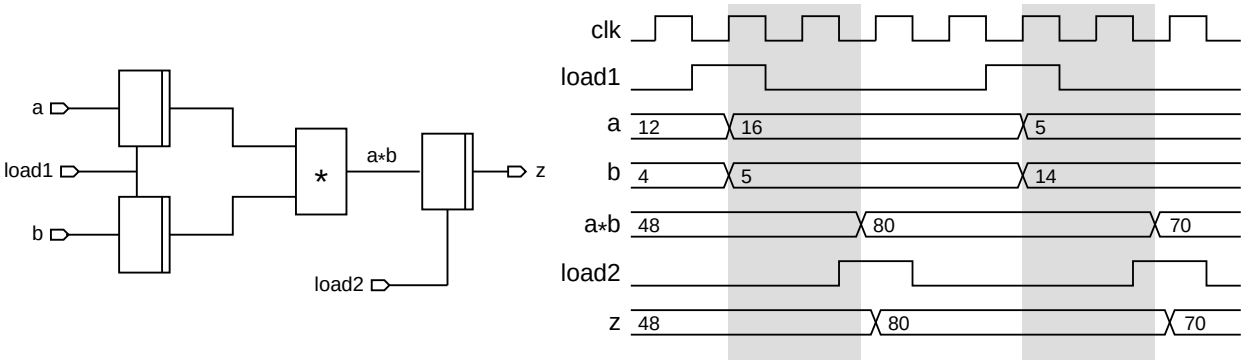
Figuur 9.22 : Een voorbeeld van een netwerk met een *false path*.

Voor het doorlopen van het *false path* moet de ingang van de eerste multiplexer hoog en die van de tweede multiplexer laag zijn. In dit netwerk is dat nooit het geval. Daarom kan dit pad genegeerd worden bij de statische tijdsanalyse. Timing analyzers hebben een optie om de *false paths* te oormerken, zodat deze bij de tijdsanalyse buiten beschouwing worden gelaten.

Multicycle paths

Een *multicycle path* is een combinatorisch pad tussen twee registers of flipfloppen waarvoor meerdere klokslagen beschikbaar zijn. De maximale klokfrequentie hangt dan niet alleen af van de vertraging van de combinatoriek maar ook af van het aantal klokslagen N dat beschikbaar is. De tijdvertraging wordt dan verdeeld over N klokslagen. Voor een *multicycle path* luidt formule 9.3:

$$f_{\max} = \frac{1}{t_{\text{clk}\rightarrow\text{q}} + t_{\text{su}} + \frac{1}{N} t_{\text{delay}}} \tag{9.3}$$



Figuur 9.23 : **Een voorbeeld met een *multicycle path*.** Links staat de schakeling en rechts het signaaldiagram. De grijze vlakken geven het venster waarbinnen het signaal `a*b` instabiel is. Signaal `z` is stabiel als er twee klokslagen verstreken zijn nadat `a` en `b` zijn ingeladen.

In de schakeling van figuur 9.23 hebben de ingangsregisters een stuursignaal `load1` en heeft het uitgangsregister een stuursignaal `load2`. Als deze twee stuursignalen van een toestandsmachine afkomen, die ervoor zorgt dat `load2` twee klokslagen na `load1` hoog wordt, zijn er twee klokslagen beschikbaar voordat signaal `a*b` stabiel moet zijn. In de figuur is het signaaldiagram getekend. De grijze vlakken geven het venster aan waarbinnen signaal `a*b` niet stabiel is. Bij de actieve opgaande klokflank waarbij signaal `load2` hoog is, is `a*b` stabiel. Omdat er tussen `load1` en `load2` altijd twee klokslagen zitten, kan de vermenigvuldiging langer duren dan één klokslag.

9.9 Dynamische tijdsanalyse of timingssimulatie

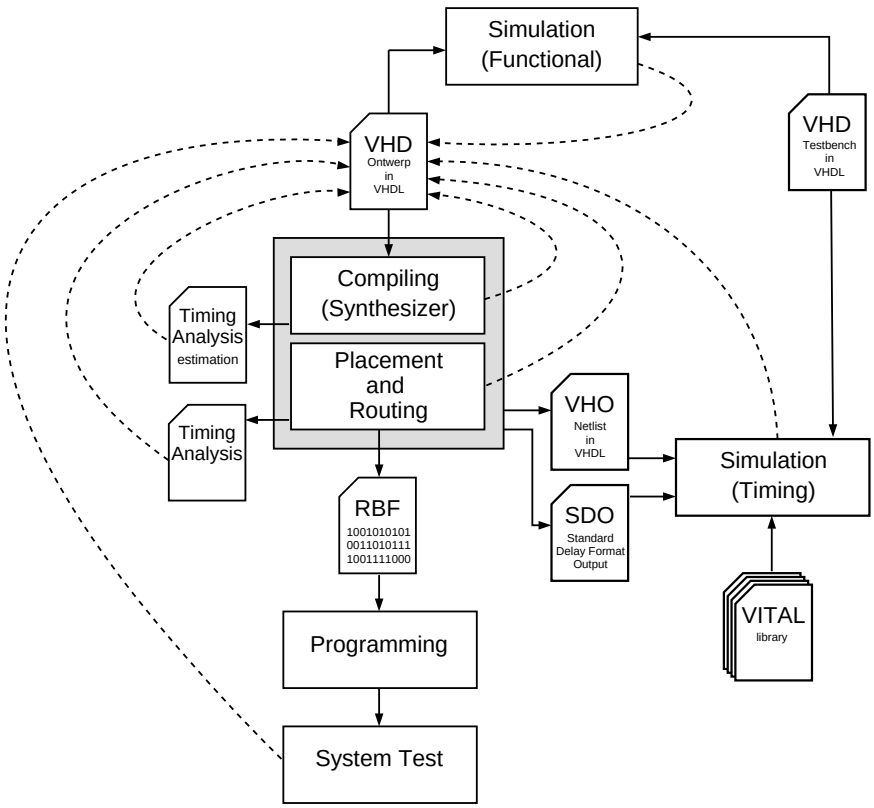
Een timingssimulatie wordt ook postsimulatie genoemd. Bij ASIC's gebruikt men ook de term postlayoutsimulatie en bij FPGA's en CPLD's heet dat een postmappingsimulatie. De functionele simulatie noemt men dan prelayout- of premappingsimulatie.

Dynamische tijdsanalyse, *Dynamic Timing Analysis* of DTA, bestudeert het functionele gedrag en gebruikt daarbij de tijdvertragingen in het ontwerp. Het is een simulatie op basis van het gesynthetiseerde netwerk inclusief gedetailleerde informatie over het tijdsgedrag. Na de synthese en *mapping* is bekend welke *lookup tables* en flipfloppen er gebruikt worden en na de plaatsing en bedrading zijn alle gedetailleerde gegevens over de tijdvertraging bekend.

Het extraheren van tijdvertragingen uit een gesynthetiseerd netwerk noemt men *back-annotation*. Omdat bij FPGA's en ASIC's het gesynthetiseerde netwerk altijd anders is dan het oorspronkelijk VHDL-ontwerp, kan de oorspronkelijk gedragsbeschrijving niet voor de dynamische tijdsanalyse gebruikt worden. Naast

de timingparameters is ook een VHDL-beschrijving van het gesynthetiseerde netwerk nodig. Een dynamische tijdsanalyse is een zogenoemde *gate level*-simulatie oftewel simulatie op poortniveau.

De syntheseprogramma's kunnen een VHDL-beschrijving (VHO) van het gesynthetiseerde netwerk genereren met een apart bestand (SDO) dat de vertragingstijden bevat. VHO staat voor *VHDL output file* en SDO staat voor *standard delay output format file*. Begin jaren 90 is door OVI, *Open Verilog International* het *standard delay format* ontwikkeld voor het vastleggen van vertragingstijden. Tegelijkertijd heeft een IEEE werkgroep een standaard ontwikkeld voor het beschrijven van logische poorten in VHDL met een gedetailleerd tijdsgedrag. Deze standaard heet VITAL, *VHDL initiative toward asic libraries*, en maakt gebruik van het door OVI ontwikkelde *standard delay format*.

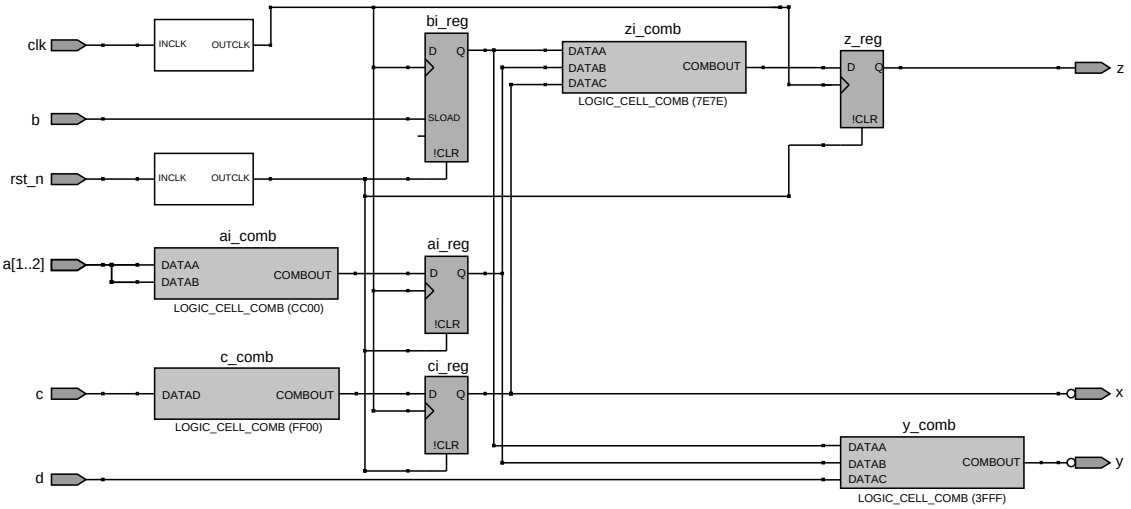


Figuur 9.24 : **De plaats van de timingssimulatie in het ontwerptraject.** De timingssimulatie of dynamische tijdsanalyse gebruikt de netwerkbeschrijving (`VHO`), het bijbehorende bestand (`SDO`) met de tijdvertragingen, de VITAL-bibliotheek en de testbench. De onderbroken pijlen gegeven de terugkoppeling naar het ontwerp aan.

Figuur 9.24 geeft de plaats van de dynamische tijdsanalyse in het ontwerptraject. De synthesizer genereert een VHO- en een SDO-bestand. Met de testbench, waarmee eerst het functionele gedrag met het ontwerp is getest, wordt ook het gesynthetiseerde netwerk onderzocht. De VHDL-bibliotheek bevat beschrijvingen van de componenten van FPGA op basis van het VITAL-formaat.

Voorbeeld timingssimulatie

Deze subparagraaf gebruikt als voorbeeld voor de timingssimulatie de schakeling van figuur 9.18. Van dit ontwerp is een gedragsbeschrijving en een testbench gemaakt. Het ontwerp is gesimuleerd en vervolgens gesynthetiseerd. Het syntheseresultaat staat in staat in figuur 9.25.



Figuur 9.25 : Het syntheseresultaat van de figuur 9.18.

De gebruikte technology in dit voorbeeld is een Cyclone-II FPGA van Altera, de EP2C5T144C8. Volgens Altera is de maximale frequentie waarbij deze component nog goed functioneert 340 MHz. Voor de synthese en de statische tijdsanalyse is Quartus II versie 9.1 gebruikt.

Bij de synthese is een statische tijdsanalyse uitgevoerd en zijn de VHO- en SDO-bestanden voor de timingssimulatie geëxtraheerd. Voor de postsimulatie is configuratie uit figuur 9.8 gebruikt. De *golden unit* is de gedragsbeschrijving en de *unit under test* is het gesynthetiseerde netwerk uit de VHO- en SDO-bestanden. De gedragsbeschrijving van het netwerk en de structuurbeschrijving van het gesynthetiseerde netwerk worden zo gelijktijdig getest.

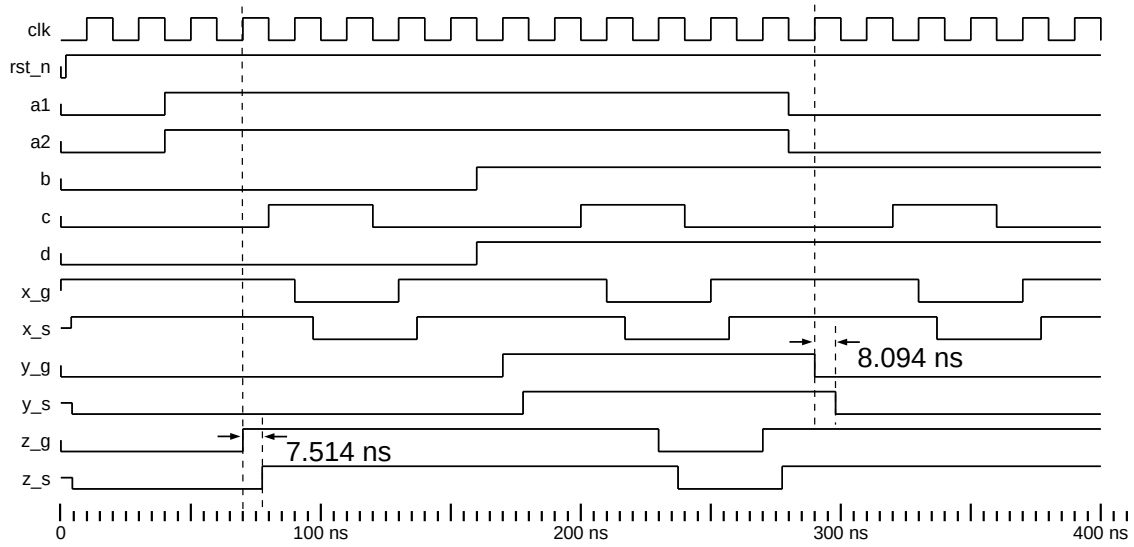
Figuur 9.26 geeft een simulatieresultaat. De signalen `x_g`, `y_g` en `z_g` zijn de uitgangssignalen `x`, `y` en `z` van de gedragsbeschrijving en de signalen `x_s`, `y_s` en `z_s` zijn de uitgangssignalen van de structuurbeschrijving van het gesynthetiseerde netwerk.
De grootste vertragingstijd in het signaaldiagram is te vinden rond 290 ns bij het laag worden van uitgang `y`. Uitgang `y_s` van de structuurbeschrijving ijlt 8.094 ns na op uitgang `y_g` van de gedragsbeschrijving. De ontwerper zou ten onterechte kunnen beredeneren dat dit ongeveer overeenkomt met een 125 MHz en dat de maximale frequentie 125 MHz is.

Quartus II geeft iets andere resultaten dan in tabel 9.4 staan. In plaats van kolom *delay+setup* geeft de classical timing analyzer een kolom met f_{max}. In deze kolom is de maximaal gegarandeerde frequentie van 340 MHz ingevuld.
Daarnaast is de vormgeving van de tabel aangepast.

Tabel 9.4 : De statische tijdsanalyse voor de vertraging van klok naar klok.

index	slack	delay	delay+setup	start pin	end pin
1	18.522	1.214	1.478	bi.clk	z reg0.clk
2	18.565	1.171	1.435	ai.clk	z reg0.clk
3	18.815	0.921	1.185	ci.clk	z reg0.clk

Uit de statische timinganalyse blijkt dat het traagste pad tussen twee flipfloppen 1,478 ns is en dat deze ligt tussen de klokflank van flipflop `bi` en die van flipflop `z~reg0`. Deze vertraging komt overeen met een frequentie van 676 MHz. Dit is aanzienlijk hoger dan 125 MHz, die de simulatie suggereert.



Figuur 9.26 : Het simulatieresultaat van een functioneel en een tijdsgedrag. De simulatie bevat zowel de gedragsbeschrijving van het netwerk uit figuur 9.18 als de structuurbeschrijving van het gesynthetiseerde netwerk uit figuur 9.25. De signalen `x_g`, `y_g` en `z_g` zijn de uitgangssignalen van de gedragsbeschrijving en de signalen `x_s`, `y_s` en `z_s` zijn de uitgangssignalen van de structuurbeschrijving.

De maximale frequentie wordt bepaalt door de vertraging tussen de flipfloppen en de registers. De vertragingen in figuur 9.26 zijn vertragingen van de klokflank van het uitgangsregister naar de uitgangspin. In dit geval is uitgang `y` gekoppeld aan pin 8 van de FPGA. De vertraging in de simulatie bevat ook de vertragingen van de uitgangsbuffer van deze pin.

t_{co} is de *clock to output delay*.

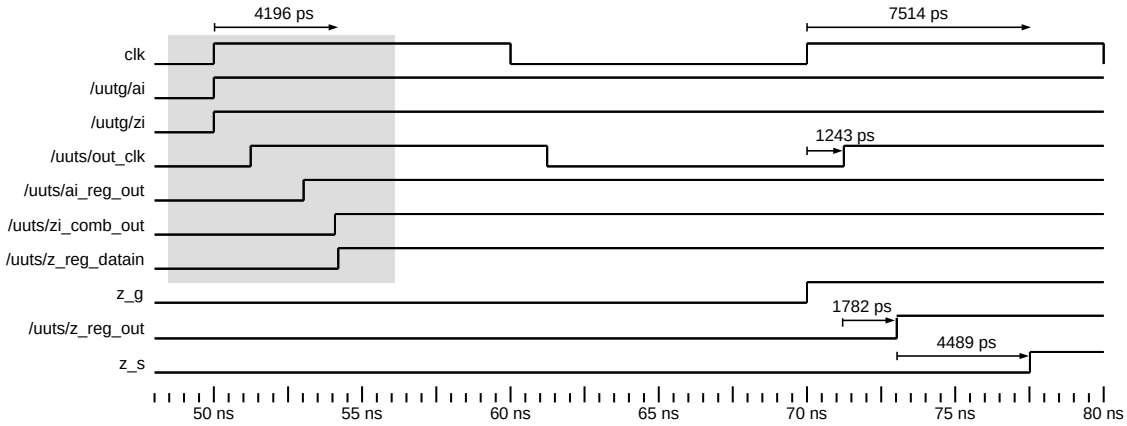
De tijdsanalyse geeft ook de vertragingen t_{co} van klok naar uitgang. Tabel 9.5 toont het overzicht uit de tijdsanalyse.

Tabel 9.5 : De statische tijdsanalyse voor de vertraging van klok naar uitgang.

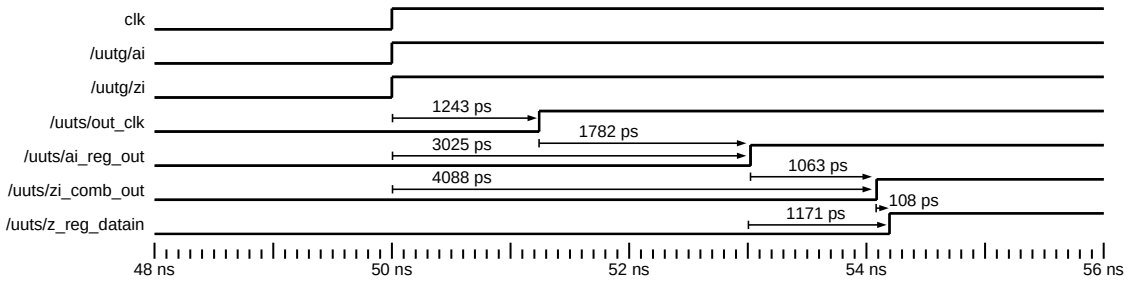
t_{co}	start pin	end pin
8.094	`ai.clk`	y
7.848	`bi.clk`	y
7.514	`z_reg.clk`	z
7.093	`ci.clk`	x

Om het tijdsgedrag tussen flipfloppen te bestuderen, moet aan het signaaldiagram een aantal interne signalen worden toegevoegd. In figuur 9.27 zijn de signalen toegevoegd, die op het pad liggen dat van register `ai` langs het combinatorische blok `zi~0` en flipflop `z~reg0` naar uitgang `z` loopt. Bij de actieve klokflank van 50 ns verandert de data-ingang `z_reg_datain` van flipflop `z_reg_out` na 4,196 ns. Bij de volgende klokflank wordt deze verandering doorgegeven aan de uitgang `z_s` van de structuurbeschrijving. Dit gebeurt na 7,514 ns. Deze vertraging komt overeen met de t_{co} uit tabel 9.5.

Figuur 9.28 toont het grijze deel uit figuur 9.27 met het signaalgedrag vanaf de actieve klokflank bij 50 ns naar de data-ingang `z_reg_datain` van flipflop `z_reg0`.



Figuur 9.27 : Een deel van het signaaldiagram met een aantal interne signalen. Het betreft de signalen:
`out_clk` is de uitgang van het klokcircuit.
`ai_reg_out` is de uitgang van flipflop `ai`.
`zi_comb_out` is de uitgang van het combinatorische blok `zi_comb`.
`z_reg_datain` is de ingang van flipflop `z_reg`.
`z_reg_out` is de uitgang van flipflop `z_reg`.
Het grijze gebied is in figuur 9.28 uitvergroot.

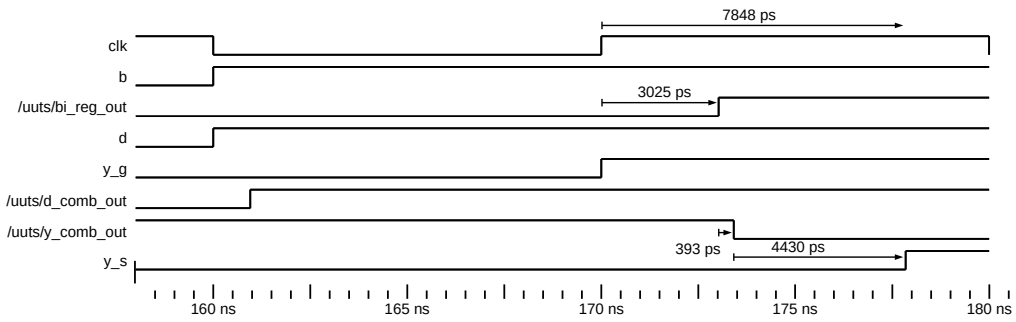


Figuur 9.28 : Een deel van het signaaldiagram nog meer uitvergroot. Deze figuur bevat het grijze gebied uit het signaaldiagram van figuur 9.28. Het laat de tijdvertragingen van de klokflank naar de dataingang van flipflop `z_reg` zien.

De signaalverandering langs het pad van flipflop `ai_reg` naar flipflop `z_reg` gaat in vier stappen:

- De klokingang komt de FPGA binnen via een speciaal klokcircuit. De uitgang van dit circuit is signaal `out_clk` en is 1,243 ns vertraagd. Omdat alle flipfloppen deze klok gebruiken, heeft dit geen invloed op de maximaal haalbare frequentie.
- Na de actieve flank van `out_clk` duurt het 1,782 ns voordat de uitgang `ai_reg_out` van de flipflop `ai` verandert. Deze vertraging hoort bij de flipflop. Alle flipfloppen, dus ook flipflop `z_reg`, veranderen pas na deze vertraging. Per saldo heeft dat geen effect op de maximaal haalbare frequentie.
- Tussen flipflop `ai` en flipflop `z_reg` zit het combinatorische blok `zi_comb`. De tijdvertraging van dit blok is 1,063 ns.
- De dataingang geeft nog een extra tijdvertraging van 0,108 ns.

Totaal is 4,196 ns nodig om na de actieve flank van `clk` de data-ingang `z_reg_datain` van flipflop `z_reg` te wijzigen. Toch is de maximaal haalbare frequentie niet de reciproke van 4,196. Alleen de laatste twee stappen zijn relevant voor het kritieke pad. Daarom vermeldt tabel 9.4 als vertraging 1,171 ns.



Figuur 9.29 : Het deel van het signaaldiagram waarbij uitgang `z` verandert.

De schakeling bevat ook een combinatorisch pad van ingang `d` via `y_comb` naar uitgang `y`. Figuur 9.29 toont een deel uit het signaaldiagram bij 170 ns. De verandering van `y_g` is hier niet het gevolg van de verandering van `d` maar van `b`. De signalen `b` en `d` worden tegelijk hoog. Uitgang `y` moet dan laag worden. Het diagram laat zien dat de uitgang `bi_reg_out` van flipflop `bi` pas na de eerst volgende klokslag verandert. De tijdvertraging uit het diagram komt hier weer overeen met de betreffende t_{co} uit tabel 9.5.

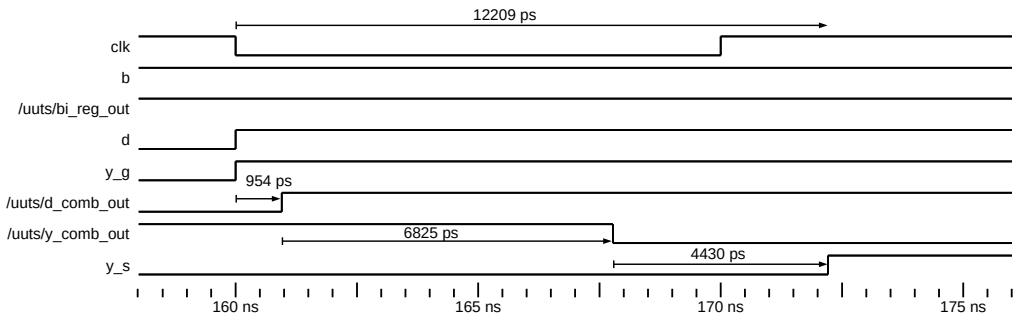
Tabel 9.6 : De statische tijdsanalyse voor de combinatorische paden.

t_{pd}	from	to
12.209	d	y

Om het effect te zien van een verandering langs dit combinatorische pad moet ingang `b` al hoog zijn, zoals in figuur 9.30 staat. De verandering gaat in drie stappen:

- De ingangsbuffer van ingang `d` geeft een vertraging van 0,954 ns.
- De vertraging van het combinatorische blok `y_comb` is 6,825 ns.
- De uitgangsbuffer van uitgang `y` draagt 4,430 ns bij.

De totale vertraging van dit logische pad is 12,209 ns en komt overeen met de propagatietijd t_{pd} uit tabel 9.6.



Figuur 9.30 : Het deel van het signaaldiagram waarbij uitgang `z` verandert.

9.10 Statische tijdsanalyse versus dynamische tijdsanalyse

Een dynamische tijdsanalyse of timingssimulatie geeft heel veel informatie over het tijdsgedrag van een ontwerp. Een probleem is dat de ontwerper daarvoor een testbench nodig heeft. In principe kan dat de testbench zijn, die bij de functionele simulatie is gebruikt. Om alle kritieke paden te vinden moeten extra testvectoren toegevoegd worden. De aanstuurbaarheid van deze paden en de observeerbaarheid is daarbij vaak een probleem.
Het voordeel van de statische tijdsanalyse is dat alle kritieke paden gevonden worden. Een nadeel is dat de analyse te pessimistisch kan zijn door *false paths* en *multicycle paths*.

Voor een synchroon systeem voor FPGA's en CPLD's met één systeemklok zijn functionele simulaties en een statische tijdsanalyse voldoende. De ervaring leert dat het ontwerp altijd het gedrag van de simulatie uitvoert. Als er bij de systeemtest toch fouten zijn, blijken deze ook met een functionele simulatie te kunnen worden gereproduceerd. Timingssimulaties kunnen bij synchrone systemen achterwege gelaten worden.

Asynchrone systemen kunnen niet met een statische tijdsanalyse onderzocht worden. Deze systemen kunnen alleen afdoende getest worden met timingssimulaties. Overigens zijn timing analyzers tegenwoordig wel geschikt voor meerkloksystemen. De aansluitingen van een digitaal systeem zijn meestal asynchroon. Een timingssimulatie, die dit gedrag onderzoekt, is wel nuttig. Datzelfde geldt ook voor een onderzoek naar het signaalgedrag tussen deelsystemen met een verschillende klok. Omdat, vanwege de vermindering van de vermogensdissipatie, meerkloksystemen steeds vaker toegepast worden, blijven dynamische tijdsanalyses zinvol.

9.11 Verificatie van de vermogensdissipatie

Er zijn twee verschillende soorten vermogensdissipaties bij digitale geïntegreerde schakelingen, namelijk statische en dynamische vermogensdissipatie. Beide soorten dissipatie hebben een andere fysische oorsprong en moeten anders bestreden worden.

Oorzaak van statische vermogensdissipatie

Lekstromen veroorzaken de statische vermogensdissipatie. Deze zorgen er voor dat FPGA's en andere digitale componenten altijd vermogen dissiperen. Ook als een schakeling niets doet, in de rusttoestand, wordt er energie verbruikt.
De afmeting van de sporen en dus die van de transistoren wordt bij iedere generatie FPGA's kleiner. De componenten hebben een steeds lagere voedingsspanning nodig. De *core*, de binnenkant van een FPGA, werkt tegenwoordig met 1,2 V en binnen een aantal jaren zal dat 0,8 V zijn. Een neveneffect van deze verdere miniaturisatie is dat de ruismarges relatief groot worden en dat daarmee de lekstromen toenemen.
Statische vermogensdissipatie is met name vervelend voor ontwerpers van mobiele apparatuur, omdat ook in stand-by het apparaat relatief veel vermogen verbruikt en dus vaker opgeladen moet worden.

De verbindingen tussen de transistoren hebben een capaciteit en de gates van de transistoren zijn te beschouwen als vlakke plaatcondensatoren. De hoeveelheid opgeslagen energie van een capaciteit is $\frac{1}{2}CV^2$.
Voor het opladen is CV^2 nodig; er dissipeert dus een $\frac{1}{2}CV^2$. Bij het ontladen komt de opgeslagen energie vrij. Iedere oplaad-ontlaad-cyclus heeft dus CV^2 nodig. Voor een frequentie f worden de capaciteiten f keer per seconde geladen en ontladen. Het dynamische gedissipeerde vermogen $P_{\text{dynamisch}}$ is daarmee fCV^2

Dat de dissipatie niet afhangt van de weerstand gaat tegen je gevoel in. Men denkt vaak dat de dissipatie afhangt van de weerstand. De weerstanden bepalen alleen hoe snel de capaciteiten opgeladen wordt. Als de weerstand groot is, vindt de dissipatie over een langere periode plaats. Als de weerstand klein is, worden de capaciteiten snel opgeladen en wordt er in korte tijd veel gedissipeerd. Als de weerstand klein is, kan de klokfrequentie f groter en zal $P_{\text{dynamisch}}$ ook groter zijn. $P_{\text{dynamisch}}$ is immers evenredig met f.

Als de frequentie en de voedingsspanning beide 20% lager genomen worden, wordt het gedissipeerde vermogen gehalveerd.
Laptops gebruiken deze methode om energie te sparen als de laptop niet verbonden is met een voedingsadapter.

Oorzaak van dynamische vermogensdissipatie

Dynamische dissipatie is het gevolg van het opladen en ontladen van interne capaciteiten in de component. Als alle transistoren en verbindingen in één klokslag een keer opgeladen en ontladen worden, dan hangt de dynamische dissipatie af van de klokfrequentie f, de totale capaciteit C van de transistoren en de verbindingen en van de voedingsspanning V:

$$P_{\text{dynamisch}} = fCV^2 \tag{9.4}$$

In de praktijk doen niet alle transistoren elke klokslag mee. De werkelijke dynamische dissipatie zal daarom lager zijn. Dit wordt aangegeven met een activiteitenfactor of vormfactor α. Deze factor ligt tussen 0 en 1. De dynamische dissipatie is dan:

$$P_{\text{dynamisch}} = \alpha fCV^2 \tag{9.5}$$

De dynamische dissipatie wordt kleiner als de transistorafmetingen en de spoorbreedten kleiner en de voedingsspanning lager wordt. In de praktijk blijkt dat de totale capaciteit bij verdere integratie niet kleiner wordt doordat de digitale IC's steeds meer transistoren krijgen. Digitale IC's voeren steeds complexere functies uit bij een eveneens toenemende klokfrequentie. Per saldo wordt de dynamische dissipatie juist groter.

Optimalisatie voor dissipatie

Bij verificatie van vermogensdissipatie gaat het er om dat het ontwerp zo weinig mogelijk vermogen verbruikt. Er zijn drie verschillende aspecten waarop een ontwerper kan optimaliseren, namelijk snelheid, oppervlak en vermogen. De synthesizers en andere programma's binnen de ontwikkelomgevingen voor ASIC en FPGA's hebben opties waarmee op één van deze aspecten kan worden geoptimaliseerd.
Er is een groot aantal factoren, die invloed hebben op de vermogensdissipatie. Sommige horen bij de technologie en zijn niet beïnvloedbaar door de ontwerper, andere zijn wel instelbaar. In het algemeen wordt een lage vermogensdissipatie verkregen door:

- Een lage voedingsspanning te gebruiken. De *core* — dat is de binnenkant van FPGA's zonder de in- en uitgangscellen — heeft een lagere voedingsspanning, bijvoorbeeld 1,2 V. De in- en uitgangscellen hebben dan een instelbare voedingsspanning van 2,5 V of 3,3 V.
- Een lage klokfrequentie te kiezen. De dynamische dissipatie is recht evenredig met de frequentie. Het is vaak verstandig om niet automatisch de maximale haalbare klokfrequentie te kiezen. Als 200 MHz haalbaar is en het systeem werkt ook goed bij 20 MHz, is de dynamische dissipatie bij een klokfrequentie van 20 MHz slechts 10%.
- Het ontwerp op te delen in domeinen met een eigen kloksignaal. Sommige domeinen kunnen dan een klok met een lagere frequentie krijgen. FPGA's en CPLD's hebben meerdere globale kloklijnen, die hiervoor gebruikt kunnen worden. De klok van een domein kan zelfs tijdelijk uitgezet worden. Hoewel de statische dissipatie aanwezig blijft, is de dynamische dissipatie voor het betreffende klokdomein dan nul.
- Het ontwerp op te delen in domeinen met een eigen voeding. De dissipatie van een deel van het ontwerp, dat tijdelijk niet gebruikt wordt, is nul als het geen voeding krijgt. Bij FPGA's en CPLD's is dit niet altijd mogelijk.

- Verschillende transistoren te gebruiken binnen digitale IC's. Transistoren met een hoge spanning en een lage lekstroom worden voor flipfloppen gebruikt en snelle transistoren met een lage spanning worden voor logische poorten gebruikt. De spanning op de logische poorten kan dan worden uitgeschakeld terwijl de spanning op de flipfloppen aanblijft. De status van het ontwerp wordt dan onthouden in deze laagvermogensmodus.
- De adviesfuncties van de ontwerpprogramma's te gebruiken. Alle ontwikkelomgevingen voor FPGA's bevatten adviesprogramma's voor laagvermogensontwerp.

De ontwerper kan bij elke ontwerpbeslissing zelf rekening houden met de grootte, de snelheid en het gedissipeerd vermogen van het ontwerp, onder andere door:

- Het gebruik van enable-signalen bij tellers en schuifregisters. Een teller die niet telt, gebruikt veel minder vermogen dan een teller die telt.
- Het voorkomen van glitches of spikes. Bij een signaal dat even hoog en daarna weer laag is, loopt er kortstondig een oplaadstroom en een ontlaadstroom. Dat geeft extra vermogensdissipatie. Grote combinatorische blokken hebben statistisch gezien meer kans op glitches.
- Het voorkomen van overspraak.
- Het gebruik van speciale RAM-blokken voor fifo's en buffers in plaats van flipfloppen uit de logische blokken. De speciale technologie van de RAM-blokken zorgt ervoor dat deze minder dissiperen.
- Het aantal globale signalen te beperken. De capaciteit van een lange, globale lijn is immers groot. Er is meer vermogen nodig om deze lange lijnen op te laden dan een korte lijn.

9.12 Resumé

Dit hoofdstuk toont aan dat verificatie een veelzijdig onderwerp is. Simulatie is een belangrijk aspect bij het ontwerpen van een digitaal systeem. Daarbij is het opstellen van de stimuli en het kiezen van de juiste testvectoren geen eenvoudige stap in het ontwerptraject. Veel aandacht is nodig voor de bijzondere situaties die het functionele gedrag van het digitale systeem altijd heeft.
Statische en dynamische tijdsanalyses krijgen in de rest van dit boek minder aandacht, maar zijn zeker niet onbelangrijk. De statische tijdsanalyse is een stap in het ontwerptraject die nooit overgeslagen mag worden.
Zoals in dit hoofdstuk op verschillende plaatsen is opgemerkt hebben moderne ontwikkelomgevingen voor FPGA's speciale hulpprogramma's om het ontwerp te verifiëren en verschillende adviesprogramma's, die de ontwerper ondersteunen bij het optimaliseren van het ontwerp. De adviezen van deze programma's kunnen de prestatie van het ontwerp flink verbeteren.

10

Bibliotheken

Doelstelling

Dit hoofdstuk bespreekt de verschillende packages uit de standaardbibliotheek en uit de IEEE-bibliotheek, die je bij het ontwerpen met VHDL nodig hebt. Je leert hoe je deze packages kunt gebruiken en je maakt kennis met de verschillende packages die in dit boek worden toegepast.

Onderwerpen

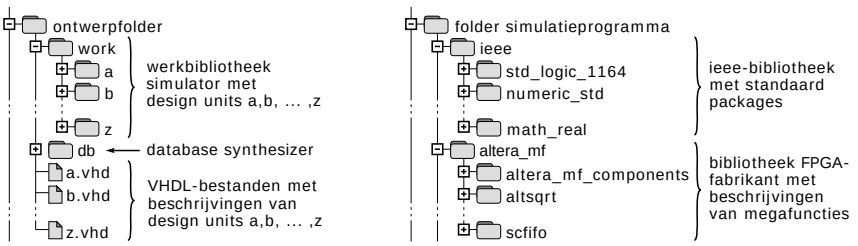
De behandelde onderwerpen zijn:

- De begrippen **`library`**, **`package`** en **`package body`**.
- Het standaard package `standard`.
- De functie `now` die huidige simulatietijd geeft.
- Het IEEE-package `standard_logic`.
- Het IEEE-package `numeric_std`.
- De conversiefunctie uit `numeric_std`: `to_unsigned`, `to_signed` en `to_integer`.
- De resize-functies uit `numeric_std`.
- Rekenkundige bewerkingen met `numeric_std`.
- Het verschil tussen **`mod`** en **`rem`**.
- Relationele bewerkingen met `numeric_std`.
- De schuiffunctie uit `numeric_std`: **`sll`**, **`srl`**, **`rol`** en **`ror`**.
- De verschillen tussen de packages `numeric_std` en `std_logic_arith`.
- De overflow- en underflowdetectie.
- Het package `textio` voor de in- en uitvoer van tekst.
- Het gebruik van het access-type en de functie **`new`**, `deallocate`.
- Alternatieven packages en bibliotheken.
- Het begrip overloading.

VHDL gebruikt bibliotheken met kant-en-klare onderdelen. Deze bevatten entities, die als componenten in een ontwerp opgenomen worden, of packages met vooraf gedefinieerde constanten, typedefinities, componentdeclaraties, functies en procedures. De belangrijkste bibliotheken zijn de werkbibliotheek, standaardbibliotheek en de IEEE-bibliotheek. Daarnaast kent elke FPGA-ontwikkelomgeving speciale bibliotheken met de beschrijvingen van eigen componenten, softcores en onderdelen voor de beschrijving van gesynthetiseerde netwerken.

De ontwerper plaatst zijn VHDL-bestanden meestal bij elkaar in één *ontwerp*-folder. De simulator en de synthesizer creëren bij deze bestanden folders voor de simulatie- en syntheseresultaten. Het simulatieprogramma Modelsim maakt een zogenoemde werkbibliotheek met de naam *work*. Na compilatie staan daar de

zogenoemde *primary design units* of primaire ontwerpeenheden. Fysiek zijn dat folders met gecompileerde bestanden voor de entity's, de packages en de configuraties. Op dezelfde manier plaats Quartus, de ontwikkelomgeving van Altera, de syntheseresultaten in een folder *db*.



Figuur 10.1 : De structuur van een ontwerpfolder en de folder van het simulatieprogramma.

Een voorbeeld van de inhoud van een ontwerpfolder staat in figuur 10.1 samen met een voorbeeld van de folder van het simulatieprogramma. Deze laatste bevat in ieder geval de IEEE-bibliotheek met de verschillende IEEE-packages. Daarnaast bevat het bibliotheken met vooraf gedefinieerde *design units*.

In eerdere hoofdstukken zijn de packages `std_logic_1164` en `numeric_std` uit de IEEE-bibliotheek gebruikt. Met het sleutelwoord **`library`** wordt de bibliotheek toegankelijk gemaakt en met **`use`** wordt aangegeven welke onderdelen uit de bibliotheek gebruikt worden:

```
library ieee;
library altera_mf,lpm;

use ieee.std_logic_1164.all;
use altera_mf.altera_mf_components.all, lpm.lpm_components.all;
```

Het use-statement is opgebouwd uit één of meer verwijzingen. Een verwijzing is opgebouwd uit de bibliotheeknaam, de naam van de *design unit* en het sleutelwoord **`all`**. Dit sleutelwoord geeft aan dat alles uit de betreffende *design unit* toegankelijk is. Eventueel kan ook één enkel onderdeel uit de *design unit* toegankelijk worden gemaakt, zoals de componentdeclaratie `dffp` uit het package `altera_mf_components`:

```
use altera_mf.altera_mf_components.dffp;
```

VHDL definieert geen fysieke paden voor de bibliotheek. Het simulatieprogramma verbindt de bibliotheeknamen met de fysieke plaats.

De bibliotheken `work` en `std` zijn altijd toegankelijk. De werkbibliotheek `work` bevat de *design unit* van de ontwerper en de standaardbibliotheek `std` is altijd gedefinieerd. Het package `standard` uit de standaardbibliotheek is eveneens altijd toegankelijk. Zonder deze declaraties expliciet te vermelden, zijn bij alle beschrijvingen deze library- en use-statements impliciet aanwezig:

```
library work;
library std;
use std.standard.all;
```

Dit hoofdstuk bespreekt de packages uit de standaardbibliotheek en de belangrijkste packages uit de IEEE-bibliotheek.

10.1 Het package standard

Het package standard wordt bij elke compilatie/simulatie automatisch meegenomen. De definities uit dit package zijn te beschouwen als *predefined types*. Voor dit package is geen use-clausule nodig.

In bijlage B staat in code B.1 de complete beschrijving van het package standard.

Alle basale datatypen in VHDL zijn enumerated types

Alle typen zijn bij VHDL gedefinieerd als een enumerated type, met behulp van een *range* of een subtype. Het type bit is een enumerated type en een integer is gedeclareerd als range:

```
type bit is ('0', '1');
type integer is range -2147483648 to 2147483647;
subtype natural is integer range 0 to integer'high;
```

Het type natural is een subtype van integer.

Bij de initialisatie van een variabele krijgt deze impliciet de meest linkse waarde. Dit kan worden voorkomen met een expliciete initialisatie. Voorbeelden van declaratie met standaardtypen zijn:

```
signal a : bit;              -- a krijgt waarde '0'
signal b : bit := '1';       -- b krijgt waarde '1'
signal i : integer;          -- a krijgt waarde -2147483647
signal j : integer := 0;     -- b krijgt waarde 0
```

Een initialisatie heeft bij de synthese geen enkele betekenis. De synthesizer laat deze beginwaarde weg. In het package is ook het type time gedefinieerd en een bijbehorend record **units** dat de relaties tussen picoseconden, microseconden, seconden, minuten, etc. vastlegt. De basiseenheid is de femtoseconde (10^{-15} seconde).

De typen bit en bit_vector kennen slechts twee signaalniveaus: '0' en '1'. Andere signaalniveaus, zoals X en Z kent het type bit niet. Het type std_logic uit het package std_logic_1164 kent deze signaalniveaus wel. Dat is de reden dat dit boek geen bit en bit_vector gebruikt, maar std_logic en std_logic_vector.

Synthetiseerbaarheid van de verschillende datatypen

Tabel 10.1 geeft aan welke typen uit standard synthetiseerbaar zijn en welke niet. Het niet synthetiseerbaar zijn van een type is meestal goed te begrijpen. Hoe kan een synthesizer een tijd van bijvoorbeeld exact 2,43 ns realiseren? Het type time is dus niet synthetiseerbaar.

Tabel 10.1 : De synthetiseerbaarheid van de typen uit het package standard.

Type	Synthetiseerbaar	Type	Synthetiseerbaar
boolean	ja	severity level	nee
bit	ja	string	nee, tenzij
character	ja	bit_vector	ja
integer	ja	integer_vector	ja
natural	ja	real_vector	nee
positive	ja	time_vector	nee
real	nee	file_open_kind	nee
time	nee	file_open_status	nee

De algoritmes, die bij floating-point berekeningen nodig zijn, zijn complex. Reële getallen zijn daarom niet synthetiseerbaar. Wel zijn er vanaf VHDL-2008 synthetiseerbare packages beschikbaar voor floating-point- en fixed-point getallen.
Een severity-level heeft voor de synthese geen betekenis. Dat is typisch iets voor de simulatie. Zelfgemaakte strings zijn eveneens niet synthetiseerbaar. Daarentegen zijn strings van bijvoorbeeld bit of std_logic wel synthetiseerbaar.

De functie `now`

In het package `standard` wordt naast alle typedefinities één functie gedefinieerd, namelijk de functie `now`. Deze functie geeft de huidige simulatietijd terug. Dat is de tijd, die verstreken is nadat de simulatie gestart is. De functie `now` is vooral handig bij testbenches.

Tabel 10.2 : De standaardbewerkingen van VHDL. In de derde kolom is voor iedere bewerking *globaal* aangegeven wat de typen van de operanden en van de uitkomst zijn.

soort bewerkingen	operatoren	typen retourwaarde en operanden				
rekenkundige bewerkingen	**abs** +, -	numeriek	←		⊠[1]	numeriek
	+, -, *, /	numeriek	←	numeriek	⊠	numeriek
	mod, **rem**	integer	←	integer	⊠	integer
	**	numeriek	←	numeriek	**	integer
schuiffuncties	**sll**, **srl**, **sla**, **sra**, **rol**, **ror**	bitvector	←	bitvector	⊠	integer
relationele bewerkingen	=, /=, <, <=, >, >=	boolean	←	alle typen	⊠	alle typen
logische bewerkingen	**and**, **or**, **nand**, **nor**, **xor**, **xnor**	boolean	←	boolean	⊠	boolean
		bit	←	bit	⊠	bit
		bitvector	←	bitvector	⊠	bitvector

[1] Het symbool ⊠ is een plaatsvervanger voor één van de genoemde operatoren.

De standaardbewerkingen van VHDL

In tabel 10.2 staan de bewerkingen, die standaard in VHDL aanwezig zijn. Deze functies kennen allemaal een infix-notatie. Dat wil zeggen dat de operator tussen de beide operanden staat:

```
z   <= x   +   y;
bv3 <= bv2 and bv1;
b3  <= b2  or  b1;
```

In VHDL kunnen andere functies de standaardfuncties vervangen. De herdefinitie van een functie noemt men *overloading*. In code 10.3 op bladzijde 235 staat de definitie van de **and**-functie voor `std_logic_vector`. Er bestaan zodoende meerdere **and**-functies:

```
// bv1, bv2, bv3 are bit_vector and sv1, sv2, sv3 are std_logic_vector
bv3 <= bv2 and bv1;   -- uses standard VHDL and-function
sv3 <= sv2 and sv1;   -- uses and-function from package std_logic_1164
```

In tabel 10.2 staan globaal de typen van de operanden en de uitkomst vermeld. Bewust zijn hier niet expliciet de typen van de standaardfuncties genoemd. In plaats daarvan zijn algemene namen gebruikt en deze ook toepasbaar zijn bij de *overloaded* functies. Een bit in de tabel is niet alleen een `bit`, maar ook een `std_logic`. De bitvector staat voor een `std_logic_vector`, `unsigned` of een `signed` en een numerieke waarde kan een `real`, `integer`, `unsigned` of `signed` zijn.

10.2 Het package `standard_logic_1164`

1164 is het nummer van de IEEE-standaard waar dit package op gebaseerd is.

Het bekendste en belangrijkste IEEE-package is `std_logic_1164`. Het bevat de definities voor het type `std_logic` en is een alternatief voor het standaardtype `bit`. De definities uit dit package worden toegankelijk door vóór de entity, of vóór de architectuur, deze twee regels toe te voegen:

```
library ieee;
use ieee.std_logic_1164.all;
```

Het package bestaat uit twee delen: een **package**-deel en een **package body**. Het package-deel is vergelijkbaar met een headerbestand uit C. Het bevat onder meer: vooraf gedefinieerde constanten, typedefinities, componentdeclaraties, en de declaraties van de interfaces van functies en procedures. Deze interfaces komen overeen met de prototypes uit de taal C en geven alleen de in- en uitgangsparameters van de functies en procedures. De **package body** bevat de complete declaraties van de functies en procedures. Hierin staan de definities van de functies en procedures, die horen bij de declaraties van de interfaces van functies en procedures uit het package-deel.

Het **package**-deel van std_logic_1164 bevat de typedefinities voor onder andere std_ulogic, std_logic, std_ulogic_vector, std_ulogic_vector en de interfacedeclaraties van de bewerkingen die bij deze typen horen. Code B.2 uit bijlage B geeft de broncode van het package-deel.

std_ulogic versus std_logic

Het package std_logic_1164 definieert niet alleen std_logic, maar ook het type std_ulogic. Sterker, std_logic is een subtype van std_ulogic. Het type std_ulogic kent negen logische niveaus:

```
type std_ulogic is ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
```

Het type std_logic kent dezelfde logische niveaus, maar gebruikt bij conflictsituaties — die optreden bij parallelle toewijzingen aan een zelfde signaal — een zogenoemde resolutiefunctie. Code 10.2 is identiek aan code 3.19 uit hoofdstuk 3 en in code 10.1 is signaal s van het type std_ulogic.

Code 10.1 : Twee parallelle signaaltoewijzingen met std_ulogic.

```
architecture a of e is
  signal s : std_ulogic;
begin
    ...
  cs1:  s <= '1';
  cs2:  s <= '0';
    ...
end architecture a;
```

Code 10.2 : Twee parallelle signaaltoewijzingen met std_logic.

```
architecture a of e is
  signal s : std_logic;
begin
    ...
  cs1:  s <= '1';
  cs2:  s <= '0';
    ...
end architecture a;
```

In het geval dat s van het type std_ulogic is, geeft de compiler een foutmelding:

```
Nonresolved signal 's' has multiple sources.
```

Als s van het type std_logic is, is er geen foutmelding en krijgt signaal s de waarde 'X'. Het belangrijkste voordeel van *unresolved* typen is dat de compiler een fout-

melding geeft als signalen verkeerd worden aangesloten. Bij *resolved* typen is dat pas zichtbaar bij de simulatie. Net als de meeste ontwerpers beperkt dit boek zich tot std_logic. De definitie van std_logic is:

```
subtype std_logic is resolved std_ulogic;
```

In de package body is de resolutiefunctie resolved gedeclareerd, die de conflictsituaties op de juiste wijze afhandelt.

Bewerkingen met std_logic_1164

Het package std_logic_1164 definieert voor de typen std_logic en std_logic_vector de logische functies: **and**, **nand**, **or**, **nor**, **xor**, **xnor** en **not**.

Rekenkundige bewerkingen, zoals optellen en aftrekken, ontbreken in dit package. Voor al deze bewerkingen geldt dat er een representatie gekozen moet worden. Moeten de bits van de std_logic_vector geïnterpreteerd worden als unsigned binair, two's complement, signed magnitude of nog anders? Voor rekenkundige bewerkingen zijn er speciale packages beschikbaar, die een aanvulling zijn op het package std_logic_1164. De belangrijkste aanvulling is het package numeric_std.

In std_logic_1164 zijn bovendien geen relationele functies gedefinieerd. In plaats hiervan gebruikt VHDL de relationele functies voor het vergelijken van strings. Dat is mogelijk omdat std_logic_vector een array van std_ulogic is en deze laatste in feite een enumerated type van character is. Dat kan merkwaardige resultaten geven:

```
v1 <= "01100";                  -- v1 is 12
v2 <= "L1100";                  -- v1 is 12
w  <= "01110";                  -- v2 is 14
b1 <= '1' when v1 < w else '0'; -- b1 is '1'
b2 <= '1' when v2 < w else '0'; -- b2 is '0'
```

Signaal b1 wordt hoog, omdat "01100" kleiner is "01110". Alfabetisch gezien is "L1100" groter dan "01110" en zal b2 dus laag zijn.
Daarom is het vergelijken van vectoren niet altijd veilig. Alleen als de beide vectoren even lang zijn en er alleen 1-en en 0-en in het spel zijn, is dit geen probleem. Het is verstandig om de vectoren voor het vergelijken even lang te maken. Het package numeric_std bevat voor unsigned en signed relationele bewerkingen, die wel een correcte oplossing geven. In paragraaf 10.7 worden de relationele bewerkingen van het package numeric_std toegelicht.

Bij de definitie van de functie **and** staat **and** tussen dubbele aanhalingstekens. Dit betekent dat bij de functie ook de infix-notatie gebruikt kan worden:

```
z <= and(x,y);
z <= x and y;
```

De infix-notatie is alleen mogelijk bij de functies uit tabel 10.2.

De and-functie uit std_logic_1164

De **package body** bevat alle functiedeclaraties van het package. De logische functies zijn met opzoektabellen gerealiseerd. Als voorbeeld wordt hier de and-functie voor variabelen en signalen van het type std_logic_vector besproken. In het package-deel staat deze interfacedeclaratie:

```
function "and"  ( l, r : std_logic_vector) return std_logic_vector;
```

In de body van het package staat een constante and_table. Het is een tweedimensionaal array. Het type stdlogic_table is ook lokaal in de **package body** gedefinieerd. De constante and_table is een 9x9-matrix, die alle mogelijke uitkomsten voor de ingangscombinaties van twee std_logic bits vastlegt.

Code 10.3 : De and-functie uit std_logic_1164.

```
174   TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;
            ⋮
215   constant and_table : stdlogic_table := (
216   --      ---------------------------------------------------
217   --      |  U    X    0    1    Z    W    L    H    -        |   |
218   --      ---------------------------------------------------
219           ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
220           ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
221           ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
222           ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
223           ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
224           ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
225           ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
226           ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
227           ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
228   );
            ⋮
311   function "and"  ( l,r : std_logic_vector ) return std_logic_vector is
312       alias lv : std_logic_vector ( 1 to l'LENGTH ) is l;
313       alias rv : std_logic_vector ( 1 to r'LENGTH ) is r;
314       variable result : std_logic_vector ( 1 to l'LENGTH );
315   begin
316       if ( l'LENGTH /= r'LENGTH ) then
317           assert false
318           report "arguments of overloaded 'and' operator are not of the same length"
319           severity failure;
320       else
321           for i in result'RANGE loop
322               result(i) := and_table (lv(i), rv(i));
323           end loop;
324       end if;
325       return result;
326   end "and";
```

Code 10.3 gebruikt op regel 312 en op regel 313 een **alias** om de leesbaarheid van de code te verbeteren. Ook als de ingangsvectoren `l` en `v` een verschillende indexering hebben, hebben `lv` en `rv` overeenkomstige indices. Dat maakt de for-lus eenvoudig en overzichtelijk.

De and-functie van twee vectoren is de and-functie van de afzonderlijke bits. In code 10.3 staat op regel 321 een for-lus die voor de afzonderlijke bits de and-functie bepaalt door de betreffende waarde uit de 9x9-matrix aan het resultaat toe te kennen. In de matrix is direct te zien dat de uitkomst alleen `'1'` is als beide bits `'1'` of `'H'` zijn.

De flankgevoelige functies `rising_edge` en `falling_edge`

De functies `rising_edge` en `falling_edge` detecteren voor een signaal respectievelijk de actieve opgaande en neergaande flank. Deze functies zijn in paragraaf 2.9 aan de orde gekomen bij de beschrijving van D-flipfloppen en dataregisters.
Deze flankgevoelige functies zijn bedoeld voor het beschrijven van klokflanken. De synthesizer herkent deze functies en interpreteert het betreffende signaal als een kloksignaal. Hoewel `rising_edge` en `falling_edge` ook bij datasignalen gebruikt

kunnen worden om een flank te beschrijven, moeten deze functies daar niet voor gebruikt worden. De synthesizer zal het datasignaal interpreteren als een kloksignaal. Het gevolg is dat het synchrone systeem met één klok verandert in een systeem met meer klokken en daarmee een asynchroon gedrag in het systeem introduceert.

10.3 IEEE-packages voor rekenkundige bewerkingen

Het IEEE-package `std_logic_1164` definieert voor `std_logic` en `std_logic_vector` alleen de logische bewerkingen. De rekenkundige en relationele bewerkingen ontbreken in dit package. Bij het opstellen van `std_logic_1164` heeft men bewust besloten dit niet toe te voegen. De reden is dat er bij de implementatie van deze bewerkingen verschillende keuzes gemaakt kunnen worden. Een vector kan op allerlei manieren een getal representeren. Het kan bijvoorbeeld een unsigned binair, een two's complement of een signed magnitude voorstellen.

Code B.3 uit bijlage B geeft de broncode van het package-deel van `numeric_std`.

In 1995 is het package `numeric_std` — de officiële aanvulling op de `std_logic_1164` — door de IEEE goedgekeurd. Omdat dit relatief laat gebeurde, zijn er door de industrie allerlei alternatieven ontwikkeld voor rekenkundige bewerkingen. Hoewel dit boek altijd `numeric_std` gebruikt, kan het — omdat sommige ontwerpers wel een alternatief package toepassen — toch verwarrend zijn. In principe zijn er vier alternatieven voor het maken van rekenkundige bewerkingen:

- `numeric_std`
- `standard_logic_arith`
- `standard_logic_unsigned`
- `standard_logic_signed`

De package `numeric_std` en `standard_logic_arith` doen min of meer hetzelfde. Beide declareren ze twee nieuwe vectortypen: een type `unsigned` voor de unsigned binaire en een type `signed` voor de signed binaire berekeningen:

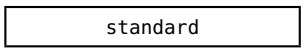
```
type unsigned is array (natural range <>) of std_logic;
type signed is array (natural range  <>) of std_logic;
```

Beide packages bevatten voor zowel het type `unsigned` als voor het type `signed` de rekenkundige en relationele bewerkingen. Vectoren van het type `signed` worden geïnterpreteerd als two's complement getallen en vectoren van het type `unsigned` als unsigned binair. De packages `standard_logic_unsigned` en `standard_logic_signed` interpreteren alle vectoren van het `std_logic_vector` als unsigned of als signed.

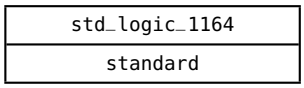
Overzicht gebruik van de diverse packages voor vectoren

De verschillende packages voegen steeds andere functionaliteiten toe. Het is een gelaagde structuur. Package `std_logic_1164` maakt gebruik van `standard` en het package `numeric_std` heeft op zijn beurt het package `std_logic_1164` nodig.

Hierna worden zes combinaties besproken. In de kantlijn zijn de lagen gevisualiseerd. De voorbeelden in dit boek gebruiken steeds één van de drie eerstgenoemde combinaties. De andere komen in de praktijk voor, maar worden afgeraden omdat deze verwarring kunnen geven of omdat ze niet zinvol zijn.



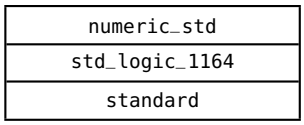
Als er geen bibliotheken aangeroepen worden, is alleen de standaardbibliotheek beschikbaar. Deze bibliotheek gebruikt de `bit` en de `bit_vector` om signalen weer te geven. Het is gebruikelijk om in plaats van deze typen altijd `std_logic` en `std_logic_vector` toe te passen.



Met deze aanroep kent het ontwerp alleen vectoren van het type `std_logic_vector`:

```
library ieee;
use ieee.std_logic_1164.all;
```

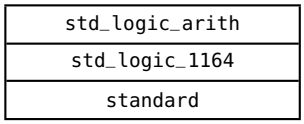
Met alleen `std_logic_1164` zijn geen rekenkundige bewerkingen mogelijk. Het package kent ook geen relationele bewerkingen. In plaats hiervan worden deze vectoren vergeleken als strings. Dit gaat goed mits de vectoren even lang zijn en deze geen metawaarden bevatten.



Het package `numeric_std` is een aanvulling op het package `std_logic_1164` en moet na `std_logic_1164` worden aangeroepen:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

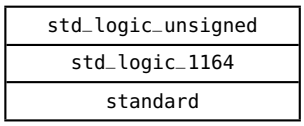
Met het package `numeric_std` zijn er voor `std_logic_vector` nog steeds geen rekenkundige en relationele bewerkingen mogelijk. Wel kent dit package vectoren van het type `unsigned` en van het type `signed` met de bijbehorende rekenkundige en relationele bewerkingen.



Het package `std_logic_arith` is net als `numeric_std` een aanvulling op het package `std_logic_1164` en moet ook na `std_logic_1164` worden aangeroepen:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
```

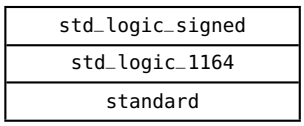
Functioneel gezien is dit identiek aan de combinatie met `numeric_std`. Ook in dit geval is `std_logic_vector` alleen zonder rekenkundige en relationele bewerkingen beschikbaar en kent dit package deze bewerkingen voor vectoren van het type `unsigned` en `signed`.



Met het package `std_logic_unsigned` wordt iedere `std_logic_vector` als geïnterpreteerd als een unsigned binair getal.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

Dit package gebruikt intern het package `std_logic_arith` voor de rekenkundige en relationele bewerkingen. Met deze combinatie zijn alle bewerkingen unsigned. Om signed bewerkingen te uitvoeren moet tussen de packages `std_logic_1164` en `std_logic_unsigned` expliciet het package `std_logic_arith` aangeroepen worden.



Met het package `std_logic_signed` wordt iedere `std_logic_vector` als geïnterpreteerd als een signed binair getal.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
```

Het package `std_logic_signed` gebruikt net als `std_logic_unsigned` intern het package `std_logic_arith` bij de rekenkundige en relationele bewerkingen. Ontwerpen met enkel signed bewerkingen zijn zeldzaam, zodat dit package overbodig is.

De packages numeric_std en std_logic_arith komen functioneel grotendeels met elkaar overeen, maar mogen absoluut niet gelijktijdig gebruikt worden. Het package std_logic_signed is, zoals al eerder is aangegeven, overbodig. Het package std_logic_unsigned voegt niet veel extra functionaliteit toe. Met unsigned uit numeric_std of std_logic_arith kan hetzelfde worden bereikt.

Package std_logic_arith is ontwikkeld door Synopsys en was veel eerder beschikbaar dan numeric_std.
Dit package had daarom jarenlang de status van een standaard en is naderhand in de officiële IEEE-bibliotheek opgenomen zonder dat het een echte standaard is.

Dit boek gebruikt alleen de packages std_logic_1164 en numeric_std. Daarmee kunnen al rekenkundige en relationele bewerkingen voor signed en unsigned getallen worden gedaan. De belangrijkste reden dat std_logic_arith in dit boek niet toegepast is, is dat het door elkaar gebruiken van numeric_std en std_logic_arith zeer verwarrend is.

Verder heeft numeric_std minder mogelijkheden dan std_logic_arith. Enerzijds is dat een nadeel, anderzijds is numeric_std daardoor eenvoudiger te begrijpen en komen er minder snel situaties voor die dubbelzinnig zijn. Een laatste reden is dat numeric_std het officiële IEEE-package voor unsigned en signed is.
De volgende vijf paragrafen bespreken de verschillende onderdelen van het package numeric_std. Hierbij wordt ook uitgelegd wat de overeenkomstige bewerkingen van std_logic_arith zijn. Voorbeelden op internet en sommige boeken gebruiken nog steeds std_logic_arith. Het is zinvol om te zien hoe de overeenkomstige bewerkingen eruit zien en op welke wijze de bewerkingen van std_logic_arith omgezet worden naar bewerkingen met numeric_std.

Als s2 en s1 beide signed zijn, heeft std_logic_arith voor deze toekenning
```
s2 <= s1 + "1001";
```
twee optelfuncties: één waarbij "1001" unsigned (9) is en één waarbij "1001" signed (-7) is. Dit is een situatie die ambigu of dubbelzinnig is.

10.4 Het package numeric_std: conversiefuncties

Bij VHDL, en met het package numeric_std in het bijzonder, kan een signaal op veel manieren een getal representeren. Het kan bijvoorbeeld een integer, natural, std_logic_vector, unsigned of signed zijn. Omdat VHDL een streng getypeerde taal is, moet een signaal soms omgezet worden van het ene type naar het andere. Het converteren is te groeperen in drie verschillende categorieën:

1. automatische typeconversie,
2. typecasting,
3. conversiefuncties.

Automatische typeconversie vindt plaats tussen integer en natural. Een natural is een subtype van integer en dus ook een integer.

```
-- i is integer and n is natural
i <= n;
n <= i;  -- i must >=0  else this results in a fatal error
```

Omdat de typen std_logic_vector, unsigned en signed alle drie een array van het type std_logic zijn, kunnen de bits van deze vectoren direct aan elkaar en aan een std_logic worden toegewezen:

```
-- sl is std_logic, sv is std_logic_vector,
-- u is unsigned and s is signed
sv(2) <= u(3);
sl    <= s(2);
u(2)  <= sl;
s(1)  <= u(1);
```

Tabel 10.3 : De conversiefuncties uit numeric_std en std_logic_arith.

	numeric_std	std_logic_arith
van unsigned naar natural	`n <= to_integer(u);`	`n <= conv_integer(u);`
van natural naar unsigned	`u <= to_unsigned(n, u'length);`	`u <= conv_unsigned(n, u'length);`
van signed naar integer	`i <= to_integer(s);`	`i <= conv_integer(s);`
van integer naar signed	`s <= to_signed(i, s'length);`	`s <= conv_signed(i, s'length);`

Om dezelfde reden kan een unsigned en een signed met behulp van *type casting* omgezet worden naar een std_logic_vector en omgekeerd:

```
-- sv1,sv2,sv3,sv4 are std_logic_vector,
-- u1,u2 are unsigned and s1,s2 are signed
sv1 <= std_logic_vector(u1);
sv2 <= std_logic_vector(s1);
u2  <= unsigned(sv3);
s2  <= signed(sv4);
```

Het converteren van unsigned naar signed en omgekeerd kan ook. In het algemeen is dat niet zinvol omdat de betekenis van de enen en nullen voor beide typen anders is. Aftrekken van twee natuurlijke getallen kan een negatief getal opleveren. Als het resultaat unsigned is, leidt dat tot een foute uitkomst. Door u1 en u2 met één bit uit te breiden en met typecasting om te zetten naar een signed, is het resultaat wel correct.

```
-- u1,u2,u are unsigned(7 downto 0) and s is signed(8 downto 0)
u <= u1 - u2;                                -- 12 - 23 result in 245
s <= signed('0' & u1) - signed('0' & u2);    -- 12 - 23 result in -11
```

Voor het omzetten van een natural naar unsigned en van een integer naar een signed en omgekeerd zijn in het package numeric_std conversiefuncties gedefinieerd. Tabel 10.3 geeft deze conversiefuncties met de overeenkomstige conversiefuncties uit std_logic_arith.

```
-- n is natural, i is integer and sv1, sv2 are std_logic_vector
n  <= to_integer(unsigned(sv1));
i  <= to_integer(signed(sv2));
```

Omgekeerd kan een natural of integer omgezet worden naar std_logic_vector door eerst het getal te converteren naar unsigned of een signed en vervolgens deze via typecasting om te zetten naar std_logic_vector:

```
-- n is natural, i is integer and sv1, sv2 are std_logic_vector
sv1 <= std_logic_vector(to_unsigned(n, sv1'length));
sv2 <= std_logic_vector(to_signed(i, sv2'length));
```

Het package std_logic_arith kent veel meer conversiefuncties dan numeric_std. De functies conv_unsigned en conv_signed geven ook een std_logic_vector terug:

```
-- only valid with std_logic_arith
-- n is natural, i is integer and sv1, sv2 are std_logic_vector
sv1 <= conv_unsigned(n, sv1'length);
sv2 <= conv_signed(i, sv2'length);
```

De conversiefuncties uit package std_logic_arith controleren niet op foutieve invoer. Onderstaand voorbeeld zet een negatief getal om naar een std_logic_vector:

```
-- only valid with std_logic_arith, sv is 4-bits std_logic_vector
sv <= conv_unsigned(-3, sv'length);
```

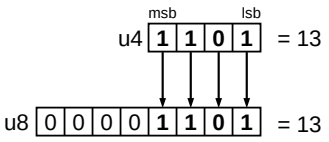
De functie `conv_unsigned` uit het package `std_logic_arith` geeft geen waarschuwing en `sv` krijgt de waarde `"1101"`. Met `numeric_std` luidt hetzelfde voorbeeld:

```
-- sv is 4-bits std_logic_vector
sv <= std_logic_vector(to_unsigned(-3, sv'length));
```

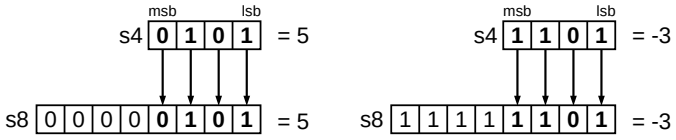
De functie `to_unsigned` uit het package `numeric_std` herkent dat `-3` geen unsigned getal kan zijn, en geeft bij de compilatie een foutmelding:

```
Error: Value does not belong to type/subtype 'natural'.
       -3 is not between [0, 2147483647]
```

```
-- numeric_std
u8 <= resize(u4,u8'length);

-- std_logic_arith
u8 <= conv_unsigned(u4,u8'length);
```




```
-- numeric_std
s8 <= resize(s4,s8'length);

-- std_logic_arith
s8 <= conv_signed(s4,s8'length);
```



Figuur 10.2 : Het uitbreiden van unsigned en signed getallen. Bij een `unsigned` worden vooraan nullen toegevoegd. Een `signed` krijgt nullen als het getal positief is en enen als het negatief is.

10.5 Het package `numeric_std`: resize-functies

Om onduidelijkheden bij het manipuleren met vectoren te voorkomen, is het nodig dat de lengtes van vectoren gelijk zijn. Een vector moet soms worden uitgebreid door één of meer bits toe te voegen of worden ingekort door bits te verwijderen. Dit is bijvoorbeeld noodzakelijk voor over- en underflowdetectie. Het package `numeric_std` gebruikt de functie `resize` voor het uitbreiden en inkorten van unsigned en signed getallen. Het package `std_logic_arith` gebruikt hiervoor de functies `conv_unsigned` en `conv_signed`.

In figuur 10.2 staat voor de typen `unsigned` en `signed` het uitbreiden en in figuur 10.3 staat het inkorten van deze vectoren. Opmerkelijk is het verschil bij het verkleinen van signed-vectoren. Het package `std_logic_arith` kapt de vector gewoon af en `numeric_std` bewaart het teken van de vector.

Het inkorten van vectoren kan ook zonder een speciale resize-functie worden bereikt door bits weg te laten:

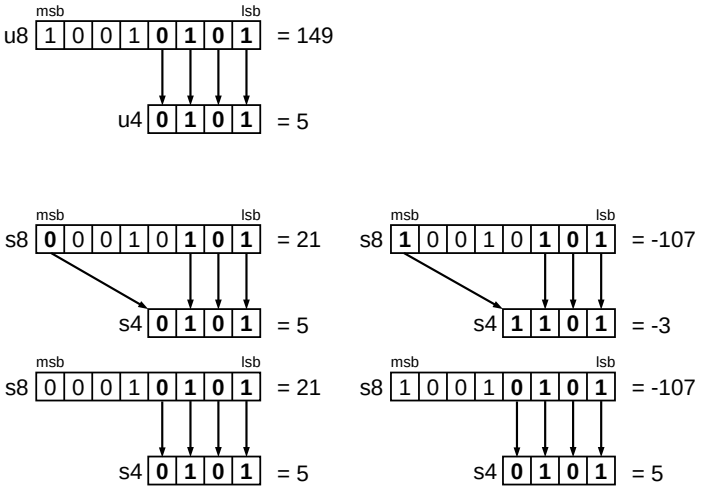
```
u4 <= u8(u4'length-1 downto 0);
s4 <= s8(7) & s8(s4'length-2 downto 0);  -- numeric_std style
s4 <= s8(s4'length-1 downto 0);          -- std_logic_arith style
```

```
-- numeric_std
u4 <= resize(u8,u4'length);
```

```
-- std_logic_arith
u4 <= conv_unsigned(u8,u4'length);
```

```
-- numeric_std
s4 <= resize(s8,s4'length);
```

```
-- std_logic_arith
s4 <= conv_signed(s8,s4'length);
```



Figuur 10.3 : Het inkorten van unsigned en signed getallen. Bij een `signed` verandert bij `numeric_std` de meest significante bit niet. Een negatief getal blijft negatief en een positief getal blijft positief. Bij `std_logic_arith` worden vooraan de enen en nullen verwijderd.

Het uitbreiden van vectoren kan eveneens zonder resize-functie gedaan worden met behulp van het concatenatie-teken (`&`):

```
u8 <= "0000" & u4;
s8 <= "1111" & s4 when s4(3)='1' else "0000" & s4;
```

Een nadeel van deze methode is dat het aantal toe te voegen nullen en enen in de bitstrings exact moet kloppen.

Tabel 10.4 : De rekenkundige bewerkingen uit `numeric_std`.

operator	bewerking	synthetiseerbaar
abs	absolute waarde	ja
-	negatie (unaire min)	ja
+	optellen	ja
-	aftrekken	ja
*	vermenigvuldigen	ja
/	delen	nee
rem	*remainder*, de rest van een deling	nee
mod	modulus	nee

De rekenkundige bewerkingen uit `numeric_std` zijn standaard in VHDL al gedefinieerd voor de typen `natural` en `integer`.

10.6 Het package `numeric_std`: rekenkundige bewerkingen

Rekenkundige bewerkingen zijn in te delen in unaire en binaire bewerkingen. Een unaire bewerking heeft één operand en een binaire bewerking heeft twee operanden. Het package `numeric_std` bevat voor vectoren van het type `signed` twee unaire functies: de negatie - en de absolute waarde **`abs`**:

Binair heeft hier twee verschillende betekenissen. Een binair getal is een getal dat uit enen en nullen bestaat. Een binaire operator is een operator met twee operanden.

```
negation      <= -s;
absolute_value <= abs(s);
```

Deze twee unaire bewerkingen zijn synthetiseerbaar en zijn alleen gedeclareerd voor het type `signed`. De absolute waarde van een natuurlijk getal is zinloos en binnen het bereik van de natuurlijke getallen is de negatie ongedefinieerd.

In tabel 10.4 staan alle rekenkundige bewerkingen uit het package `numeric_std`. De bewerkingen `+`, `-` en `*` zijn synthetiseerbaar. De andere drie bewerkingen zijn niet synthetiseerbaar.

Er is vaak verwarring rond **`rem`** en **`mod`**. Dit komt omdat in veel talen, zoals C, een operator `%` wordt gebruikt, die de rest van een deling berekent, maar ten onrechte aangeduid wordt als modulus.

De functies **`mod`** en **`rem`** lijken sterk op elkaar. De functie **`rem`** geeft de rest van een deling en de functie **`mod`** geeft de modulus. Hier staan een paar voorbeelden met de **`mod`** en de **`rem`** die in VHDL standaard voor integers beschikbaar zijn:

```
r1 <=  13 rem  4;   -- r1 is  1
r2 <= -13 rem  4;   -- r2 is -1
r3 <=  13 rem -4;   -- r3 is  1
r4 <= -13 rem -4;   -- r4 is -1
```

```
m1 <=  13 mod  4;   -- m1 is  1
m2 <= -13 mod  4;   -- m2 is  3
m3 <=  13 mod -4;   -- m3 is -3
m4 <= -13 mod -4;   -- m4 is -1
```

Als het teken van de operanden gelijk is, zijn de uitkomsten van **`mod`** en **`rem`** identiek. De uitkomst van de beide functies is dus identiek voor positieve gehele getallen.

De typen van de operanden bij de binaire rekenkundige bewerkingen

Voor iedere binaire bewerking heeft `numeric_std` zes verschillende versies, drie leveren een `unsigned` op en drie een `signed`. Tabel 10.5 geeft voor deze zes versies de mogelijke typen van de linker en de rechter operand en het teruggegeven resultaat.

Tabel 10.5 : De typen bij de zes verschillende versies van de rekenkundige bewerkingen.

resultaat		linker operand		rechter operand
`unsigned`	←	`unsigned`	⊠[1]	`unsigned`
`unsigned`	←	`unsigned`	⊠	`natural`
`unsigned`	←	`natural`	⊠	`unsigned`
`signed`	←	`signed`	⊠	`signed`
`signed`	←	`signed`	⊠	`integer`
`signed`	←	`integer`	⊠	`signed`

[1] Het symbool ⊠ is een plaatsvervanger voor één van de operatoren uit tabel 10.4.

Rekenkundige bewerkingen met vectoren van ongelijke lengte

De vectoren van de operanden hoeven niet dezelfde lengte te hebben. De uitkomst van de bewerking krijgt de lengte van de grootste vector. Het optellen van een 4-bits vector en een 8-bits vector levert een 8-bits vector op.

```
-- u4 is 4-bits, u8 and uu8 are 8-bits and
-- u16 is 16-bits unsigned
   uu8 <= u8 + u4;
-- u16 <= u8 + u4;  -- compile error
   u16 <= resize(u8,16) + u4;
   u16 <= u8 + resize(u4,16);
```

```
-- s4 is 4-bits, s8 and ss8 are 8-bits and
-- s16 is 16-bits signed
   ss8 <= s8 + s4;
-- s16 <= s8 + s4;  -- compile error
   s16 <= resize(s8,16) + s4;
   s16 <= s8 + resize(s8,16);
```

Daarentegen moet bij een toekenning het aantal bits aan beide kanten van het toekenningsteken gelijk zijn. Bij het toekennen van de som van een 4-bits vector en een 8-bits vector aan een 16-bits vector moet één van de operanden 16-bits worden gemaakt, anders treedt er een compilerfout op:

```
Error: Length of expected is 16; length of actual is 8.
```

Rekenkundige bewerkingen met expliciete vectoren

De operanden van de rekenkundige bewerkingen kunnen ook *literals*, ofwel expliciet uitgeschreven bitvectoren, zijn. Bij numeric_std zijn deze toewijzingen aan de unsigned vector u1 en aan de signed vectoren s1 en s2 correct:

```
-- u,u1,u2 are 8-bits unsigned
   u1 <= u + "1001";
   u2 <= u + 9;
```

```
-- s,s1,s2,s3,s4,s5 are 8-bits signed
   s1 <= s + "1001";          -- -7 (ambiguous for std_logic_arith)
   s2 <= s + "00001001";      -- +9 (ambiguous for std_logic_arith)
   s3 <= s + 9;               -- +9
-- s4 <= s + signed("1001");  -- compile error
   s5 <= s + signed'("1001"); -- + -7
```

Literal betekent letterlijk. Bij computertalen representeert een *literal* een vaste waarde.

Bij het package std_logic_arith is de toekenning aan s1 en s2 ambigu en geeft de compiler een foutmelding. De expliciete bitvector moet bij dit package een type krijgen. Dat kan niet met een typecasting zoals bij s4; dat levert eveneens een foutmelding op. Het type moet dan aangegeven worden met een *qualified expression*, zoals bij de toewijzing aan s5 is gedaan.
Bij signed vectoren wordt de *literal* als signed geïnterpreteerd. De vector "1001" stelt dan -7 voor en geen 9. De extra nullen bij signaal s2 zorgen er voor dat s2 wel met 9 wordt verhoogd. In plaats van expliciete bitvectoren kunnen er ook integers toegepast worden. Vanwege de betere leesbaarheid is dit zelfs te prefereren boven expliciet uitgeschreven bitvectoren.

10.7 Het package numeric_std: relationele bewerkingen

Zoals in paragraaf 10.2 bij de bespreking van het package std_logic_1164 is uitgelegd, zijn de relationele bewerkingen voor het std_logic_vector niet gedefinieerd. In plaats daarvan worden deze vectoren bij het vergelijken behandeld als gewone strings. Alleen wanneer de vectoren even lang zijn en er alleen maar 1-en en 0-en in het spel zijn, zal het resultaat correct zijn.
Het package numeric_std definieert de relationele operatoren opnieuw voor de type unsigned en signed. Voor iedere operator, =, <, <=, >, >=, /=, zijn er — net als bij de rekenkundige bewerkingen — zes verschillende mogelijkheden. Deze komen overeen met de zes mogelijkheden bij rekenkundige bewerkingen uit tabel 10.5 alleen is de retourwaarde bij de relationele bewerking altijd een boolean.

Het package numeric_std kent ook een functie std_match die twee vectoren of twee std_logic vergelijkt met don't cares.

```
p : process (s) is
begin
  if std_match(s,"10-")
  then z <= '1';
  else z <= '0';
  end if;
end process p;
```

Dit kan — net als in de meeste gevallen — ook anders opgelost worden:

```
z <= s(2) and not s(1);
```

Tabel 10.6 : De relationele bewerking > bij verschillende vectortypen.

x	y	std_logic_vector	unsigned	signed
"1111"	"0000"	true	true	false
"1001"	"0100"	true	true	false
"0000"	"1111"	false	false	true
"0000"	"XXXX"	true	false	false
"1111"	"XXXX"	true	false	false
"0000"	"----"	false	false	false
"1111"	"----"	false	false	false
"111-"	"000-"	false	false	false

Omdat de relationele bewerkingen bij numeric_std anders zijn gedefinieerd dan de stringvergelijkingen die bij std_logic_vector gebruikt worden, zijn de uitkomsten voor vectoren van het type std_logic_vector en voor vectoren van het type

unsigned en signed verschillend. Tabel 10.6 laat de verschillende interpretaties voor de functie > bij deze drie vectortypen zien.

De metawaarden, of *metavalues*, van std_logic zijn 'U', 'X', 'Z', 'W' en '-'.

Het vergelijken van vectoren die metawaarden bevatten

Als één van de beide operanden bij het vergelijken een metawaarde bevat, maakt het package numeric_std de uitkomst altijd false. Bij de simulatie waarschuwen de relationele functies als er een metawaarde aanwezig is:

```
Warning: NUMERIC_STD.">": metavalue detected, returning FALSE
```

Meestal kunnen dit soort waarschuwingen genegeerd worden. Een typisch voorbeeld treedt op bij de conditionele signaaltoewijzingen:

```
u <= '1' when u1 > u2 else '0';
```

Als u1 en u2 interne signalen zijn, zullen deze zonder speciale voorzieningen ongedefinieerd zijn bij de start van de simulatie. De vergelijking u1 > u2 is dan altijd false en er wordt gewaarschuwd dat er een metawaarde is. Een alternatief is om de interne signalen een initiële waarde te geven:

De simulator Modelsim heeft een optie om alle waarschuwingen uit het package numeric_std uit te zetten.

```
signal u1 : unsigned(7 downto 0) := (others => '0');
signal u2 : unsigned(7 downto 0) := (others => '0');
```

Het nadeel van deze aanpak is dat de synthesizer deze initialisaties negeert. De meeste ontwerpers nemen geen speciale actie en negeren de waarschuwingen over metawaarden.

Tabel 10.7 : De vergelijking x > y bij vectoren van het type std_logic_vector met een ongelijke lengte.

vectortype	vergelijking	interpretatie	in getallen	resultaat
sv1 > sv2	"110" > "1010"	"1100" > "1010"	12 > 10	true
('0' & sv1) > sv2	('0' & "110") > "1010"	"0110" > "1010"	6 > 10	false
unsigned(sv1) > unsigned(sv2)	"110" > "1010"	"0110" > "1010"	6 > 10	false
signed(sv1) > signed(sv2)	"110" > "1010"	"1110" > "1010"	-2 > -6	true

Het vergelijken van vectoren met ongelijke lengte

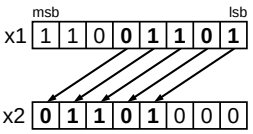
Vergelijken van vectoren, die een ongelijke lengte hebben, gaat met de relationele operatoren uit numeric_std op een correcte manier. Bij het vergelijken met vectoren van het type std_logic_vector kan dat fout gaan. Deze worden behandeld als gewone strings en worden bij een ongelijke lengte rechts aangevuld en daarna vergeleken. Dat zal altijd een onzinnig resultaat opleveren.

Bij het vergelijken van vectoren van het type std_logic_vector moeten de vectoren eerst van gelijke lengte gemaakt worden, zoals het tweede voorbeeld uit tabel 10.7 laat zien. De beste oplossing is om de vectoren sv1 en sv2 met typecasting om te zetten naar unsigned of naar signed.

10.8 Het package numeric_std: schuiffuncties

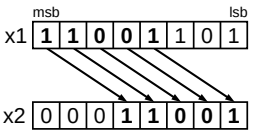
VHDL kent voor het type bit_vector een aantal schuifoperaties. Vanaf VHDL93 zijn dat: **sll**, **srl**, **rol**, **ror**, **sla** en **sra**. In het package numeric_std zijn deze functies **sll**, **srl**, **rol**, **ror** opnieuw gedefinieerd voor unsigned en signed. De linker operand van deze functie is een unsigned of een signed en de rechter operand is een integer die het aantal posities aangeeft dat er geschoven moet worden.

```
-- numeric_std
x2 <= x1 sll 3;

-- std_logic_arith
x2 <= shl(x1,"11");

-- without a package
x2 <= x1(4 downto 0) & "000";
```



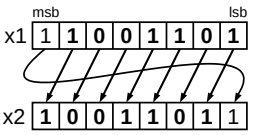
a. De functie `sll`, *shift left logical*.

```
-- numeric_std
x2 <= x1 srl 3;

-- std_logic_arith
x2 <= shr(x1,"11");

-- without a package
x2 <= "000" & x1(7 downto 3);
```



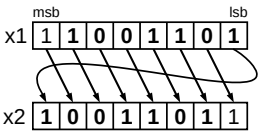
b. De functie `srl`, *shift right logical*.

```
-- numeric_std
x2 <= x1 rol 1;

-- without a package
x2 <= x1(6 downto 0) & x1(7);
```



c. De functie `rol`, *rotate left*.

```
-- numeric_std
x2 <= x1 ror 1;

-- without a package
x2 <= x1(6 downto 0) & x1(7);
```



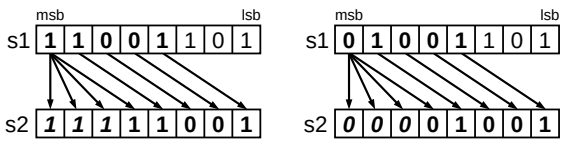
d. De functie `ror`, *rotate right*.

Figuur 10.4 : De standaard schuiffuncties uit `numeric_std`. Vermeld zijn de overeenkomstige schuiffuncties uit `std_logic_arith` en de notatie zonder speciale functies met het concatenatie-teken. De functie `shr` uit `std_logic_arith` geeft voor `signed` een andere uitkomst. Signaal `x2` wordt dan aangevuld met de *msb*-bit van `x1`, zie ook figuur 10.5.

Figuur 10.4 geeft een overzicht van de schuiffuncties. De schuiffuncties **`sll`** en `shl` schuiven de bits respectievelijk naar links en naar rechts. De leeggekomen posities worden aangevuld met nullen. De rotatiefuncties **`rol`** en de **`ror`** doen hetzelfde alleen wordt bij **`rol`** de leeggekomen plaats opgevuld met de meest significante bit en bij **`ror`** met de minst significante bit.

Het package `std_logical_arith` kent een functie `shl` en `shr` voor het naar links en naar rechts schuiven. In figuur 10.4 zijn deze alternatieven meegenomen. Alleen is het effect van `shr` voor het type `signed` anders. Deze functie komt overeen met het gedrag van de functie `shift_right` uit `numeric_std`. Dit laatste package kent naast de standaard schuiffunctie nog een aantal schuiffuncties, namelijk: `shift_left`, `shift_right`, `rotate_left` en `shift_right`. In grote lijnen komen deze functies overeen met de standaard schuiffuncties. De grootste verschillen zijn dat de standaard schuiffuncties een infix-notatie hebben en dat de tweede operand ook negatief mag zijn. De functie `shift_right` is — net als de functie `shr` uit `std_logical_arith` — anders gedefinieerd voor het type `signed`. Bij het schuiven worden de leeggekomen bits niet aangevuld met nullen, maar met de waarde van de meest significante bit.

```
-- numeric_std
-- s2 and s1 are signed
s2 <= shift_right(s1, 3);

-- std_logic_arith
-- s2 and s1 are signed
s2 <= shr(s1,"11");
```



Figuur 10.5 : De schuiffunctie `shift_right` en `shr` voor signed getallen. De leeggekomen posities worden opgevuld met de meest significante bit.

De schuifbewerkingen zijn ook realiseerbaar zonder de standaard schuiffuncties of de schuiffuncties uit `numeric_std`. De bits die verschoven moeten worden kunnen expliciet worden benoemd en met het concatenatie-teken kunnen er bits worden toegevoegd. In code 10.4 staat een 8-bits schuifregister, die zowel naar links als rechts kan schuiven. De gegevens worden serieel naar binnen geschoven en er is een parallelle 8-bits uitgang. Er is een intern signaal `s` van het type `std_logic_vector` voor het schuiven van de bits. Voor het beschrijven van een schuifregister is het package `numeric_std` dus niet noodzakelijk.

Code 10.4 : Een 8-bits schuifregister.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity shift_register is
5    port (
6        clk      : in  std_logic;
7        rst_n    : in  std_logic;
8        left     : in  std_logic;
9        left_in  : in  std_logic;
10       right_in : in  std_logic;
11       q        : out std_logic_vector(7 downto 0)
12   );
13 end entity shift_register;
14
15 architecture gedrag of shift_register is
16   signal s : std_logic_vector(q'length-1 downto 0);
17 begin
18   process (clk, rst_n) is
19   begin
20     if rst_n = '0' then
21       s <= (others =>'0');
22     elsif rising_edge(clk) then
23       if left = '1' then
24         s <= s(s'length-2 downto 0) & right_in;
25       else
26         s <= left_in & s(s'length-1 downto 1);
27       end if;
28     end if;
29   end process;
30
31   q <= s;
32 end architecture gedrag;
```

10.9 Gehele getallen met behulp van integers

Standaard VHDL-operatoren kunnen dezelfde berekeningen doen als de numerieke operatoren uit het package `numeric`. Sommige ontwerpers gebruiken signalen van het type `integer` en `natural`. Het voordeel is dat er dan geen speciaal numeriek package nodig is.
Toch is het niet verstandig integers te gebruiken. Het probleem is dat een integer gedefinieerd is als een 32-bits getal. Signalen van het type `integer` zijn dan

automatisch 32-bits en signalen van het type `natural` zijn 31-bits. Zonder speciale voorzieningen geeft de optelling van twee integers bij de synthese altijd een 32-bits opteller. Dat levert vaak veel overbodige logica op. Een ander bezwaar is dat de getallen maximaal 32-bits groot kunnen zijn.

Code 10.5 : Een 32-bits opteller met integers zonder range.

```
1  entity som is
2    port (
3      a : in  integer;
4      b : in  integer;
5      z : out integer
6    );
7  end entity som;
8
9  architecture gedrag of som is
10 begin
11   z <=  a + b;
12 end architecture gedrag;
```

Code 10.6 : Een 4-bits opteller met integers met een range.

```
1  entity som is
2    port (
3      a : in  integer range 0 to 15;
4      b : in  integer range 0 to 15;
5      z : out integer range 0 to 15
6    );
7  end entity som;
8
9  architecture gedrag of som is
10 begin
11   z <=  a + b;
12 end architecture gedrag;
```

Een beperkt bereik, *range*, bij de integer verhelpt het eerste bezwaar. Code 10.5 geeft een opteller zonder beperkt bereik voor de in- en uitgangssignalen. De synthesizer maakt in dit geval een 32-bits opteller. In code 10.6 is de breedte van de signalen beperkt door een **`range`** toe te voegen. In dit geval is het bereik `0` tot en met 15 en dit betekent dat de synthesizer van deze beschrijving een 4-bits opteller maakt.

Code 10.7 : Een 4-bits teller met behulp van een integer.

```
1  architecture gedrag of count4 is
2    signal c : integer range 0 to 15;
3  begin
4    c4: process (clk,rst_n) is
5    begin
6      if rst_n = '0' then
7        c <= 0;
8      elsif rising_edge(clk) then
9        if clear = '1' then
10         c <= 0;
11       elsif enable = '1' then
12         c <= (c + 1) mod 16;
13       end if;
14     end if;
15   end process c4;
16
17   count <= std_logic_vector(to_unsigned(c, count'length));
18 end architecture gedrag;
```

In code 10.7 staat een gedragsbeschrijving van de 4-bits teller uit code 4.25 met behulp van een integer in plaats van een `unsigned`. Bij de declaratie van de integer `c` op regel 2 is het bereik van `c` beperkt tot de waarden `0` tot en met 15. Bovendien wordt bij het verhogen van `c` de waarde nooit hoger dan 15 doordat van het resultaat de modulus 16 wordt genomen. De modulus-functie is alleen synthetiseerbaar als de waarde een macht van twee is.

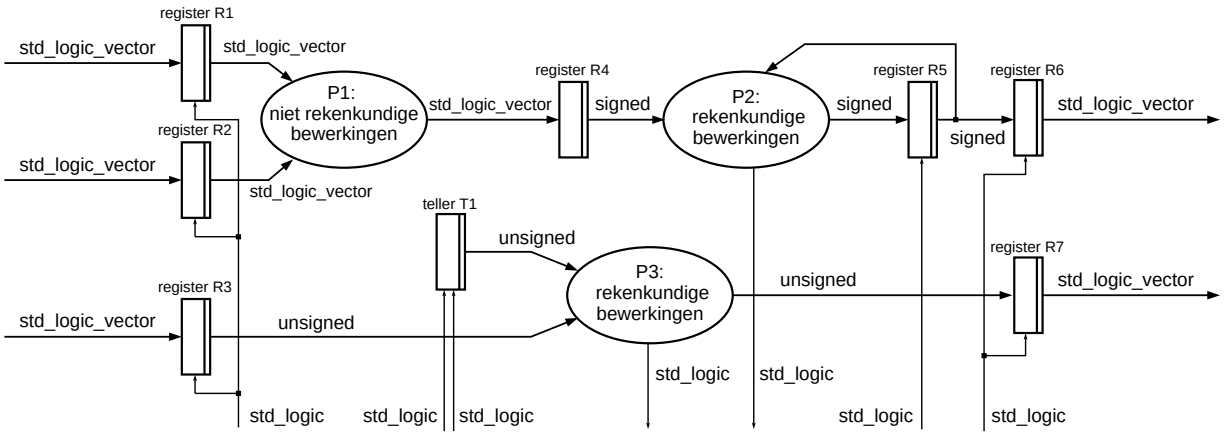
Als op regel 2 de range of op regel 12 de modulus weggelaten wordt of niet de juiste waarde heeft, leidt dat tijdens de uitvoering van de simulatie tot een fatale fout. De synthesizer genereert als beide onderdelen weggelaten worden een 32-bits teller waarvan alleen de vier minst significante bits aangesloten zijn op uitgang `count`.

Het gebruik van integers lijkt op het eerste gezicht eenvoudiger dan `unsigned` en `signed`. In werkelijkheid is het correct toepassen van integers voor numerieke bewerkingen minstens even complex. Het advies is daarom in plaats van een `integer` altijd een `unsigned` of een `signed` te gebruiken. Alleen bij expliciete bitvectoren, oftewel *literals*, kan een integer duidelijker zijn:

```
z1 <= x + "011111011011";
z2 <= x + 2011;
m1 <= '1' when c = "1111" else '0';
m2 <= '1' when c = 15 else '0';
```

10.10 Adviezen bij vectoren en tips bij numerieke bewerkingen

Hoewel er in dit boek ook voorbeelden staan zonder packages, zijn er bij een realistisch ontwerp altijd extra packages nodig. Het is verstandig om het gebruik te beperken. Het package `std_logic_1164` definieert de meerlaagslogica en is altijd noodzakelijk. Met `numeric_std` zijn er twee vectortypen `unsigned` en `signed` beschikbaar voor het vergelijken en rekenen met natuurlijke en gehele getallen. Dit package is nodig bij een ontwerp met numerieke bewerkingen.



Figuur 10.6 : De signaaltypen bij een willekeurige ontwerpeenheid. De in- en uitgangssignalen zijn allemaal van het type `std_logic` of `std_logic_vector`. De ingangsregisters `R1` en `R2` zijn van het type `std_logic_vector`. Ingangsregister `R3` is `unsigned`. De uitgangsregisters zijn allemaal van het type `std_logic_vector`. De interne registers `R4` en `R5` zijn `signed` en teller `T1` is `unsigned`. De rekenkundige bewerkingen zijn in dit voorbeeld van het type `signed` of `unsigned`.

Het is een goede gewoonte om alle in- en uitgangen van een ontwerpeenheid van het type `std_logic` of `std_logic_vector` te maken. Op een hoger niveau kunnen deze signalen dan verbonden worden zonder dat er extra conversies nodig zijn. De syntheseprogramma's genereren in VHDL netwerkbeschrijvingen waarbij de in-

en uitgangen ook altijd std_logic of std_logic_vector zijn. In principe kunnen dan de originele testbenches voor het ontwerp en voor het gesynthetiseerde netwerk worden gebruikt.

Binnen een ontwerpeenheid moeten vectoren, die getallen voorstellen en waarmee gerekend wordt, omgezet worden naar unsigned of signed. De numerieke uitkomst wordt weer geconverteerd naar std_logic_vector. De omzettingen van std_logic_logic naar signed of unsigned en vice versa gaat met typecasting. Signalen, die geen getallen representeren of waar niet mee gerekend wordt, hoeven niet te worden geconverteerd.
Figuur 10.6 geeft een willekeurig voorbeeld van een ontwerp met drie groepen bewerkingen. De combinatorische bewerkingen P1 zijn niet-rekenkundig. De in- en uitgangen van dit cluster bewerkingen kunnen std_logic_vector blijven. Bij de bewerkingen P2 en P3 stellen de in- en uitgangssignalen wel getallen voor. Omdat de bewerkingen bij P2 geheeltallig zijn, zijn de signalen van de registers R4 en R5 van het type signed. Bij P3 gaat het om natuurlijke getallen en zijn teller T1 en ingangsregister R3 daarom unsigned.
De in- en uitgangssignalen van de combinatorische processen, die rekenkundige bewerkingen representeren, zijn op deze manier altijd van het type unsigned of signed. Binnen deze processen zijn geen conversies nodig.

Overflowdetectie unsigned getallen

Besteed bij rekenkundig bewerkingen altijd aandacht aan bijzondere situaties. Als de operanden van een opteller 16-bits zijn en het resultaat is ook 16-bits, geeft dat een fout resultaat als de som van de twee operanden groter is dan 65535.

```
-- signal u1, u2, sum: unsigned(15 downto 0)
u1  <= 30000;
u1  <= 40000;
sum <= u2 + u1;    -- u is 4464 (is equal with 70000 - 65536)
```

Bij VHDL gaan de bits die niet in het resultaat passen — net als bij een hardwareversie van de opteller — verloren. Dit hoeft geen fout te zijn. De ontwerper moet hiermee rekening houden door bijvoorbeeld een overflowdetectie te maken of door het signaal sum breder te maken.

Bij een unsigned optelling kan overflow worden gedetecteerd als de opteller één bit breder is. De uitkomst van de som van twee n-bits getallen is maximaal n+1 bits breed. Door de som voor n+1 bits uit te rekenen, past de uitkomst altijd. Als de extra bit van het resultaat hoog is, is er sprake van overflow. Deze bit wordt aan signaal co toegekend en de overige bits aan sum:

```
-- signal u1, u2, sum: unsigned(n-1 downto 0);
-- signal tmp: unsigned(n downto 0);
-- signal co: std_logic
tmp <= ('0' & u2) + ('0' & u1);
sum <= tmp(sum'length-1 downto 0);
co  <= tmp(sum'length);
```

Signaal co is de carry-out van de optelling. De toestandsmachine van het ontwerp kan, als signaal co hoog is, besluiten het resultaat sum niet verder te verwerken en iets anders te doen. De uitbreiding van u2 en u1 en het inkrimpen van tmp kan ook met behulp van de functie resize gerealiseerd worden:

```
tmp <= resize(u2, tmp'length) + resize(u1, tmp'length);
```

```
sum <= resize(tmp, sum'length);
```

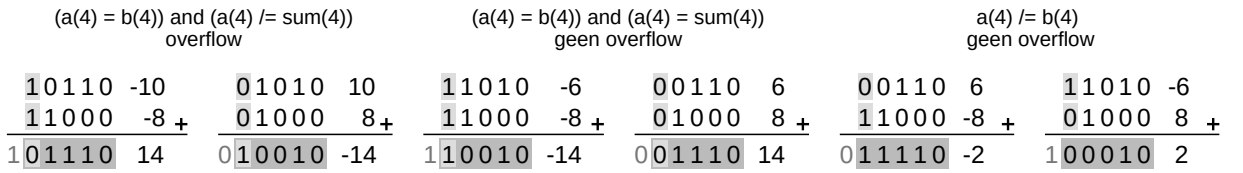
Overflowdetectie bij aftrekken met unsigned getallen is niet zinvol. De eerste operand moet dan sowieso groter of gelijk zijn aan de tweede operand. Het resultaat is dan altijd kleiner dan de eerste operand en kan nooit buiten het bereik vallen.

Overflowdetectie signed getallen

Bij signed getallen is een overflowdetectie complexer. De uitkomst van een optelling kan te hoog zijn bij twee positieve getallen en te laag bij twee negatieve getallen. Een oplossing is om de vectoren uit te breiden en daarna te verhogen. Als de waarde lager is dan de kleinst mogelijke waarde of groter dan de grootst mogelijke waarde van een n-bits getal is er overflow:

```
-- constant MAXVALUE : integer :=  2**(n-1)-1;
-- constant MINVALUE : integer := -2**(n-1);
-- signal s1, s2, sum: signed(n-1 downto 0)
-- signal tmp: signed(n downto 0);
-- signal overflow: std_logic;
tmp <= resize(s2, tmp'length) + resize(s1, tmp'length);
sum  <= tmp(sum'length-1 downto 0);
overflow <= '1' when (tmp > MAXVALUE) or (tmp < MINVALUE) else '0';
```

Deze oplossing is niet efficiënt. Een andere aanpak is de vectoren juist niet uit te breiden en eerst bij elkaar op te tellen. Er kan alleen overflow optreden als beide operanden negatief of beide positief zijn. Als beide operanden positief zijn, zal bij overflow de meest significante bit juist laag zijn. Als beide operanden negatief zijn en er overflow — of beter underflow — optreedt, zal de meest significante bit van de som juist hoog zijn.



Figuur 10.7 : Het optellen van twee 5-bits unsigned getallen `a` en `b`. Het 5-bits resultaat `sum` heeft een donker grijze achtergrond. De meest significante bits hebben een licht grijze achtergrond. De zesde bit bij het resultaat wordt niet gebruikt en heeft een grijze tint.
Er zijn drie situaties te onderscheiden. Als de msb-bits van `a` en `b` ongelijk zijn is er geen overflow. Als deze bits gelijk zijn én bovendien de msb-bit van het resultaat `sum` hieraan tegengesteld is, is er ook geen overflow, anders is er wel overflow.

Figuur 10.7 toont voor een 5-bits optelling de situaties die kunnen voorkomen. Overflow treedt op als het meest significante bit van de operanden identiek is en bovendien de meest significante bit van het resultaat daar niet gelijk aan is.

```
-- signal s1, s2, sum: signed(n-1 downto 0)
-- signal overflow: std_logic;
sum <= s2 + s1;
overflow  <= '1' when (s1(n-1) = s2(n-1)) and (s1(n-1) /= sum(n-1)) else '0';
```

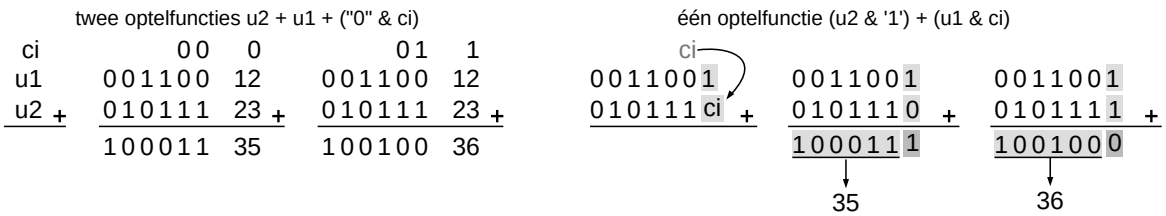
Bij aftrekken werkt de overflowdetectie op een soortgelijke wijze. Overflow treedt op als de meest significante bit van de operanden verschillend is en als bovendien de meest significante bit van het resultaat identiek is aan dat van de *substrahend*, ofwel de tweede operand:

```
-- signal s1, s2, diff: signed(n-1 downto 0)
-- signal overflow: std_logic;
diff <= s2 - s1;
overflow  <= '1' when (s1(n-1) /= s2(n-1)) and (s1(n-1) = diff(n-1)) else '0';
```

Het resultaat van een vermenigvuldiging heeft maximaal de breedte van de som van het aantal bits van de beide operanden. Als er voor het eindresultaat minder bits beschikbaar zijn, wordt overflow gevonden door het tussenresultaat te vergelijken met de minimale en maximale waarde:

```
-- constant MAXVALUE : integer :=  2**(n-1)-1;
-- constant MINVALUE : integer := -2**(n-1);
-- signal s1, s2, mul: signed(n-1 downto 0)
-- signal tmp: signed(2*n-1 downto 0);
-- signal overflow: std_logic;
tmp <= s2 * s1;
mul <= tmp(mul'length-1 downto 0);
overflow <= '1' when (tmp > MAXVALUE) or (tmp < MINVALUE) else '0';
```

Voor een deling, een modulo-bewerking en een restfunctie is overflowdetectie niet nodig omdat het resultaat altijd kleiner of gelijk is aan één van de operanden.



Figuur 10.8 : **De verwerking van een carry-in met twee optellers en met één opteller.** Links staat een voorbeeld met twee optellers. Rechts staat hetzelfde voorbeeld met één opteller. De operanden zijn aan de rechter kant uitgebreid met een extra bit: `u1` wordt uitgebreid met een `'1'` en `u2` met de carry-in. Als de carry-in laag is, is de som van de minst significante bits `'1'`. Als de carry-in hoog is, is deze som `"10"`. Deze carry-out is de carry-in voor de andere bits. Het gezochte antwoord wordt gevonden door de minst significante bit weg te laten.

Carry-in bij een optelling met unsigned getallen

Een carry-in kan direct met een extra `+` aan een optelling worden toegevoegd. Het enige probleem is dat `ci` van het `std_logic` is en er geen optelfunctie is van een `unsigned` met een `std_logic`. De oplossing is om van `ci` een vector te maken met behulp van een subtype of met een concatenatie `ci` aan een vector toe te voegen:

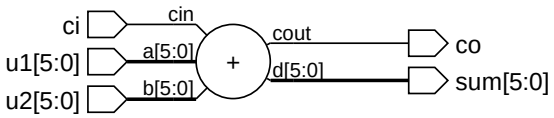
```
-- u1, u2, s1 and s2 are unsigned and t is a subtype of type s1
-- c1 is std_logic
s1 <= u1 + u2 + t'(0 => ci, others => '0');
s2 <= u1 + u2 + ("0" & ci);
```

In code 4.37 is de versie met het subtype gebruikt. Een alternatief is om één operand met `'1'` te verlengen en de andere operand met `ci`.

```
-- u1, u2, tmp and s3 are unsigned and c1 is std_logic
tmp <= (u1 & '1') + (u2 & ci);
s3  <= tmp(tmp'length-1 downto 1);
```

Aan de hand van een voorbeeld laat figuur 10.8 zien hoe deze methode werkt. Als `ci` laag is, is de uitkomst van de toegevoegde bits `'1'` en is er geen effect op de meer significante bits. Als `ci` hoog is, is de uitkomst van de toegevoegde bits `"10"` en de `'1'` is een carry-out, die de carry-in is voor de rest van de optelling. De uitkomst wordt gevonden door de minst significante bit weg te laten.

De meeste synthesizers geven beide situaties een vergelijkbaar resultaat. In beide gevallen maakt Leonardo Spectrum de RTL-view van figuur 10.9. De beschrijving met één optelfunctie geeft geen beter resultaat. De synthesizers herkennen in de beschrijving met twee optellers de carry-in en behandelen de beschrijving als een beschrijving van één optelfunctie met een carry-in.



Figuur 10.9 : Het syntheseresultaat met Leonardo Spectrum voor de opteller met een carry-in. Beide methoden voor het beschrijvingen van een carry-in geven deze RTL-view.

Ondanks het feit dat er geen verschil lijkt te zijn, is de beschrijving met één optelfunctie toch een interessante optie omdat er dan nooit twee optellers gebruikt zullen worden.

10.11 Het package `textio` voor de in- en uitvoer van tekst

Het package `textio` is een standaard package dat de mogelijkheid geeft om tekst af te drukken op het scherm, om teksten te lezen uit bestanden en teksten te schrijven naar bestanden. Het manipuleren van tekst is nuttig bij het maken van testbenches. Voor synthese heeft het gebruik van tekst geen enkele betekenis; het wordt genegeerd of leidt tot foutmeldingen. Voor een digitaal ontwerp kan dit package nooit gebruikt worden.

Software programmeurs zijn gewend om tijdens het debuggen extra regels tussen de code te plaatsen die informatie over variabelen in een bestand zetten of op het scherm afdrukken. Bij VHDL gaat dat niet zo simpel als bij Java of C. Het is ook niet echt nodig. Elk signaal kan immers altijd geobserveerd worden in een tabel of in een signaaldiagram. Bovendien zijn professionele simulatoren in feite debuggers; er kunnen breakpoints worden gezet en variabelen kunnen zo gevolgd worden.

Tekstuitvoer zonder package `textio`

Op bladzijde 204 van paragraaf 9.2 is het report-statement besproken. Met dit statement wordt tekst op het scherm afgedrukt. In code 10.8 staat een proces `p1` met een if-elsif-statement. Aan iedere tak van het if-statement is een report-statement toegevoegd die rapporteert welke tak gekozen is.
Helaas kan het report-statement alleen een string afdrukken. Door de strenge typedefinities in VHDL kan een `std_logic_vector` niet zo maar als string gebruikt worden.

Code 10.8 : Een proces met drie report-statements.

```
1  p1: process (a,b,x,y) is
2  begin
3    if a = '0' then
4      z <= x and y;
5      report "if";
6    elsif b = '1' then
7      z <= (0 => '1', others => '1');
8      report "elsif";
9    else
10     z <= y;
11     report "else";
12   end if;
13 end process p1;
```

Voor scalaire typen, zoals `character` of `integer`, kan met behulp van het attribuut `image` van een variabele of signaal een stringrepresentatie worden gemaakt en daarmee aan de af te drukken string worden toegevoegd. In code 10.9 is op regel 6 de integerwaarde van variabele `v` toegevoegd. In `p2` is een hulpvariabele `v` gebruikt om er voor te zorgen dat **`report`** de actuele waarde van `x` **`and`** `y` afdrukt. In code 10.8 en in code 10.9 krijgt `z` de nieuwe waarde pas nadat alle processen geëvalueerd zijn.

Code 10.9 : Een report-statement, die de waarde van een variabele afdrukt.

```
1  p2: process (a,b,x,y) is
2    variable v : std_logic_vector(7 downto 0);
3  begin
4    if (a = '0') then
5      v := x and y;
6      report "if not a :  v = " & integer'image(to_integer(unsigned(v)));
7      report "if not a :  v = " & to_string(v);
8    elsif b = '1' then
9      v := (others => '1');
10     report "elsif b :   v = " & to_string(v);
11   else
12     v := y;
13     report "else        v = " & to_string(v);
14   end if;
15   z <= v;
16 end process p2;
```

Om een `std_logic_vector` met **`report`** af te drukken is een functie nodig, die deze vector converteert naar een string. Er zijn verschillende methoden om dat te doen, in code 10.10 staat een functie `to_string`, die daarvoor het type-attribuut `pos` gebruikt. Voor iedere bit `i` uit `slv` wordt de positie van de betreffende waarde in de typedefinitie van `std_logic` bepaald. De constante `STD_CHAR` is een string met alle negen waarden van `std_logic`. De volgorde van de karakters is identiek aan die van `std_logic`. Omdat de index van een string altijd bij `1` begint, is de positie met één verhoogd. Het betreffende karakter uit `STD_CHAR` wordt aan string `s` toegevoegd.

Code 10.10 : De functie `to_string`, die een `std_logic_vector` omzet naar een `string`.

```
1   function to_string(slv : std_logic_vector) return string is
2     constant STD_CHAR: string := "UX01ZWLH-";
3     variable s : string(1 to slv'length);
4     variable j : integer;
5   begin
6     j := 1;
7     for i in slv'range loop
8       s(j) := STD_CHAR(std_logic'pos(slv(i))+1);
9       j  := j + 1;
10    end loop;
11    return s;
12  end function to_string;
```

Als signaal `a` laag is, geven de report-statements op regel 6 en 7 uit code 10.9 dit resultaat:

```
# ** Note: if not a :  v = 128
#    Time: 30 ns  Iteration: 1  Instance: /test
# ** Note: if not a :  v = 10000000
#    Time: 30 ns  Iteration: 1  Instance: /test
```

Iedere simulator zal dit anders weergeven. Modelsim plaatst bij iedere **`report`** automatisch ook de simulatietijd en het aantal delta-delay's of *iterations*.

Code 10.11 : Het afdrukken van tekst met de functie `write` uit het package `textio`.

```
1   p3: process (a,b,x,y) is
2     variable v : std_logic_vector(7 downto 0);
3     variable L : line;
4   begin
5     write(L, now, right, 6);
6     if (a = '0') then
7       v := x and y;
8       write(L, string'("  if not a :  v = "));
9     elsif b = '1' then
10      v := (others => '1');
11      write(L, string'("  elsif b  :  v = "));
12    else
13      v := y;
14      write(L, string'("  else     :  v = "));
15    end if;
16    write(L, to_bitvector(v));
17    writeline(output, L);
18    z <= v;
19  end process p3;
```

Het afdrukken van tekst met behulp van het package `textio`

De mogelijkheden van het standaard package `textio` zijn beperkt. Het definieert alleen lees- en schrijffuncties voor variabelen en signalen van het type: `bit`, `bit_vector`, `boolean`, `integer`, `real`, `time`, `string` en `character`.

Code 10.11 gebruikt in plaats van **report** de procedures write en writeline om tekst op het scherm af te drukken. Op regel 3 is een variable L van het type line gedeclareerd. Het type line is een pointer, die naar een string wijst. De procedure write op regel 8 schrijft een tekststring naar de locatie in het geheugen waar L naar wijst. De write van regel 16 voegt de variable v hieraan toe en gebruikt daarvoor de functie to_bitvector.
Op regel 5 is met de functie now de huidige simulatietijd bepaald en met write naar de tekstbuffer geschreven.
De procedure writeline schrijft de inhoud van de buffer naar een bestand. In dit voorbeeld schrijft de writeline van regel 17 de tekst waar L naar wijst naar output. In textio is de file-variabele output gedefinieerd als de standaarduitvoer "STD_OUTPUT"; dit is de uitvoer die ook door het report-statement wordt gebruikt.

In VHDL gebruikt men de term *pointer* niet. Meestal spreekt men van een *access type*. Dit boek gebruikt juist wel het begrip pointer, omdat dit wijd verbreid is en ook in andere talen toegepast wordt.

Het resultaat hangt af van de waarden van de ingangssignalen, maar kan er bijvoorbeeld zo uitzien:

```
#   0 ns  else     :  v = 11001100
#   0 ns  if not a :  v = 10000000
#  10 ns  else     :  v = 11001100
#  20 ns  elsif b  :  v = 11111111
#  30 ns  if not a :  v = 10000000
```

In code 10.11 is op regel 16 een std_logic_vector met behulp van de procedure write en de functie to_bitvector omgezet naar een string. Deze methode kan ook als alternatief voor de functie to_string uit code 10.10 worden gebruikt. In code 10.12 staat een alternatieve versie. De functie schrijft de vector s naar een lokale pointervariabele l en retourneert met l.**all** de inhoud van deze pointer.

Code 10.12 : De functie to_string met behulp van het package textio.

```
1  function to_string(s: std_logic_vector) return string is
2    variable b: bit_vector(s'range);
3    variable l: line;
4  begin
5    write(l, to_bitvector(s));
6    return l.all;
7  end function to_string;
```

Pointers en bewerkingen met pointers

De variabele l uit code 10.12 en L uit code 10.11 zijn beide van het type line. Dit type is een pointer en in het package textio gedefinieerd als:

```
type line is access string;
```

Interessant is dat in een digitaal ontwerp pointers geen rol spelen en dat bij microprocessoren pointers juist heel belangrijk zijn.
Een pointer is een geheugenadres en juist in die zin zouden pointers veel met hardware te maken moeten hebben.

Pointers worden in een VHDL-ontwerp nooit gebruikt, omdat deze niet synthetiseerbaar zijn. Wel kunnen pointers effectief zijn in een testbench en met name bij de manipulatie van tekst.

De toepassing van pointers verschilt bij VHDL aanzienlijk met het gebruik van pointers bij de taal C. VHDL kent door de strenge typering veel minder mogelijkheden voor het manipuleren met pointers:

- VHDL kan niet rekenen met pointers, VHDL kent dus geen *pointer arithmetic*;

- VHDL kent geen *address referencing*, anders gezegd VHDL kent geen adres-operator. De constructie `&x` uit C, die adres van variabele `x` geeft, heeft dus geen equivalent in VHDL.
- In VHDL kunnen — door de strenge typering — pointers niet getypecast worden.

Toch zijn er ook veel overeenkomsten Figuur 10.10 geeft een aantal pointerbewerkingen in VHDL met de bijbehorende bewerking uit C. Op regel 5 gebruikt C de functie `malloc` en VHDL **`new`** voor het alloceren van geheugen.

```
1   variabele t : string := "tekst";
2   variable p  : line;
3   variable L  : line;
4   -- something missing
5   p     := new string (1 to t'length);
6   p.all := t;
7   -- do something with p
8   write(L, p.all);
9   writeline(output, L);
10  deallocate(p);
```

```
1   char t[] = "tekst";
2   char *p;
3   char L[64];
4   // something missing
5   p = (char *) malloc(sizeof(t)+1);
6   strcpy(p, t);
7   // do something with p
8   sprintf(L, "%s", p);
9   fprintf(stdout, L);
10  free(p)
```

Figuur 10.10 : Overeenkomstige pointerbewerkingen in VHDL en in C. Links staat op iedere regel een VHDL-bewerking en rechts staat op dezelfde regel het equivalent in C.

Pointerrekenen kent VHDL niet maar op regel 6 wordt met **`.all`** een kopie van `t` gemaakt op de plaats waar `p` naar wijst. In C kan dat met de functie `strcpy`. Op regel 8 wordt de hele string `p.all` afgedrukt.
De procedures `write` en `writeline` van regel 8 en 9 zijn vergelijkbaar met de functie `sprintf` en `fprintf` in C. De functie `deallocate` van regel 10 geeft net als de functie `free` uit `C` het geheugen vrij waar `p` naar wist.

Lezen uit een bestand met het package `textio`

Met de procedures `read` en `readline` uit het package `textio` kunnen ook gegevens uit een bestand gelezen en naar een bestand geschreven worden. In code 10.13 staat een proces `signal_generator` dat de signaalgenerator uit de testbench van code 2.3 uit hoofdstuk 2 kan vervangen. Deze signaalgenerator leest uit een bestand de ingangssignalen `a`, `b` en `ci` en de verwachte waarden voor de uitgangen `s` en `ci`.
Het bestand *fulladder.txt* staat in tabel 10.8. Op iedere regel staan vijf bits: de eerste drie zijn de waarden voor `a`, `b` en `ci` en de laatste twee zijn de verwachte waarden voor `s` en `ci`.
Op regel 2 wordt een file `f` gedeclareerd met de modus `read_mode`, die bestand *fulladder.txt* opent. Deze definitie komt overeen met de declaraties voor de standaard in- en uitvoer uit het package `textio`.

Tabel 10.8 : Het tekstbestand `fulladder.txt`.

```
000 00
001 01
010 01
011 10
100 01
101 10
110 10
111 11
000 00
```

```
file input : text open read_mode is "STD_INPUT";
file output: text open write_mode is "STD_OUTPUT";
```

De procedure `readline` van regel 11 leest één regel uit het bestand en plaatst deze in de buffer `data`. Daarna leest de procedure `read` achtereenvolgens de waarde van `a`, `b` en `c` uit de buffer. De vierde `read` leest in één keer de verwachte waarde van `s` en `co` als een 2-bits vector.

Code 10.13 : Het lezen van stimuli uit een bestand met het package textio.

```
1  signal_generator: process is
2    file f : text open read_mode is "fulladder.txt";
3    variable data : line;
4    variable va, vb, vci: bit;
5    variable ve: bit_vector(1 downto 0);
6  begin
7    if endfile(f) then
8      deallocate(data);
9      wait;
10   end if;
11   readline(f,data);
12   read(data, va);
13   read(data, vb);
14   read(data, vci);
15   read(data, ve);
16   a  <= to_stdulogic(va);
17   b  <= to_stdulogic(vb);
18   ci <= to_stdulogic(vci);
19   expected <= to_stdlogicvector(ve);
20   wait for 50 ns;
21 end process signal_generator;
```

Voor het lezen van de gegevens worden vier variabelen gebruikt. Aangezien `read` een procedure is en de uitgang gedefinieerd is voor variabelen en niet voor signalen, moet de functie in een sequentiële omgeving worden gebruikt en moet de uitgang altijd een variabele zijn. De vier variabelen zijn van het type `bit` en `bit_vector`, omdat het package geen lees- en schrijffuncties heeft voor `std_logic`. De vier variabelen worden met behulp van de conversiefuncties `to_stdulogic` en `to_stdlogicvector` aan de signalen `a`, `b`, `c` en `expected` toegekend.
Na het lezen en verwerken van de gegevens, die op één regel in het bestand staan, wacht het proces 50 ns. Op regel 7, aan het begin van het proces, test het proces met de functie `endfile` of het einde van het bestand is bereikt. Als dat het geval is, stopt het proces.

Code 10.14 : Het vergelijken van de gevonden en de verwachte uitkomst.

```
1  found <= co & s;
2
3  compare: process is
4  begin
5    wait on a,b,ci;
6    wait for 1 ns;
7    assert found = expected
8      report "voldoet niet" severity note;
9  end process compare;
10
11 diff <= '1' after 1 ns when found /= expected else '0' after 1 ns;
```

Door het gevonden resultaat te vergelijken met het verwachte resultaat en deze uitkomst als tekst in het scherm zichtbaar te maken of in het signaaldiagram weer

te geven, is het zoeken van fouten eenvoudiger. In code 10.14 is het signaal `found` de concatenatie van de signalen `co` en `s`. Dit proces wacht op een verandering van één van de ingangssignalen `a`, `b` of `ci` en wacht daarna 1 ns. Dit laatste voorkomt dat bij iedere delta-cycle de verwachte waarde met het tussenresultaat wordt vergeleken. Als `found` en `expected` verschillen, drukt het assert-statement een waarschuwing af.

Dit soort informatie kan ook in een signaaldiagram getoond worden. Het signaal `diff` is hoog als `found` en `expected` verschillen. De tijdvertraging zorgt er weer voor dat het verschil alleen zichtbaar is als `found` niet meer verandert.

10.12 Het package `std_logic_textio` voor tekst met `std_logic`

Het package `std_logic_textio` is een aanvulling op het package `textio`. Dit package definieert lees- en schrijfprocedures voor `std_logic` en `std_logic_vector`. In code B.6 uit bijlage B staat de broncode van het package-deel.
De beschrijvingen uit paragraaf 10.11 zijn met dit package iets eenvoudiger op te schrijven. Regel 16 uit code 10.11:

```
17      write(L, to_bitvector(v));
```

kan dan worden vervangen door:

```
17      write(L, v);
```

Met dit package mogen de variabelen `va`, `vb`, `vci` en `ve` uit code 10.13 van het type `std_logic` en `std_logic_vector` zijn:

```
 5    variable va, vb, vci: std_logic;
 6    variable ve: std_logic_vector(1 downto 0);
          ⋮
13    read(data, va);
14    read(data, vb);
15    read(data, vci);
16    read(data, ve);
17    a  <= va;
18    b  <= vb;
19    ci <= vci;
20    expected <= ve;
```

Omdat bij de procedure `read` de modus van de uitgang **`variable`** is, blijven de hulpvariabelen noodzakelijk.

Het package `std_logic_textio` definieert tevens een aantal procedures voor het lezen en het schrijven van octale en hexadecimale getallen. De procedures `oread` en `hread` lezen respectievelijk een octaal en hexadecimaal getal uit een buffer:

```
      -- variable o, h and s are of type std_logic_vector(11 downto 0);
      readline(f, L);   -- file f contains this line: 0053 02B 43
      oread(L, o);      -- o is "000000101011"
      hread(L, h);      -- h is "000000101011"
      read(L, s);       -- s is "000000101011"
```

Omdat `o` en `h` octale en hexadecimale getallen zijn, moet het aantal bits van variabele `o` een veelvoud zijn van drie en dat van `h` een veelvoud van vier.

De procedures owrite en hwrite schrijven respectievelijk een octaal en hexadecimaal getal naar een buffer. In onderstaand voorbeeld wordt een vector octaal, hexadecimaal, binair en decimaal afgedrukt:

```
-- variable s is of type std_logic_vector(11 downto 0);
s := "000000101010";
owrite(L, s);
write(L, string'(" "));
hwrite(L, s);
write(L, string'(" "));
write(L, s);
write(L, string'(" "));
write(L, to_integer(unsigned(s)));
writeline(output, L);   -- print to output: 0052 02A 000000101010 42
```

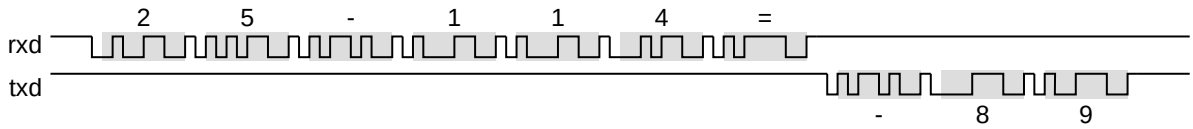
Voor het decimaal afdrukken van de vector s moet deze eerst worden omgezet naar een integer.

10.13 Voorbeeld gebruik textio bij het opstellen van een testbench

Tekstuitvoer is nuttig bij het schrijven van een testbench, met name bij de verwerking van het simulatieresultaat. Samen met procedures die de signaalinvoer vereenvoudigen geeft dat een professionele testomgeving.

Probleemstelling: het testen van een seriële rekenmachine

Deze paragraaf bespreekt de testbench voor een rekenmachine met een seriële ingang en een seriële uitgang. Deze rekenmachine kent alleen gehele getallen en een paar eenvoudige bewerkingen: +, -, *, / en de unaire min en de wortelfunctie. Voor deze laatste twee functies worden de symbolen m en s gebruikt. De uitkomst van de rekenmachine is een geheel getal. De in- en uitvoer bestaan beide uit een rij ASCII-waarden. De invoer is bijvoorbeeld 25-114=, de uitvoer is -89 en de rekenmachine via de seriële uitgang geeft achtereenvolgens de ASCII-waarden '-', '8' en '9'.

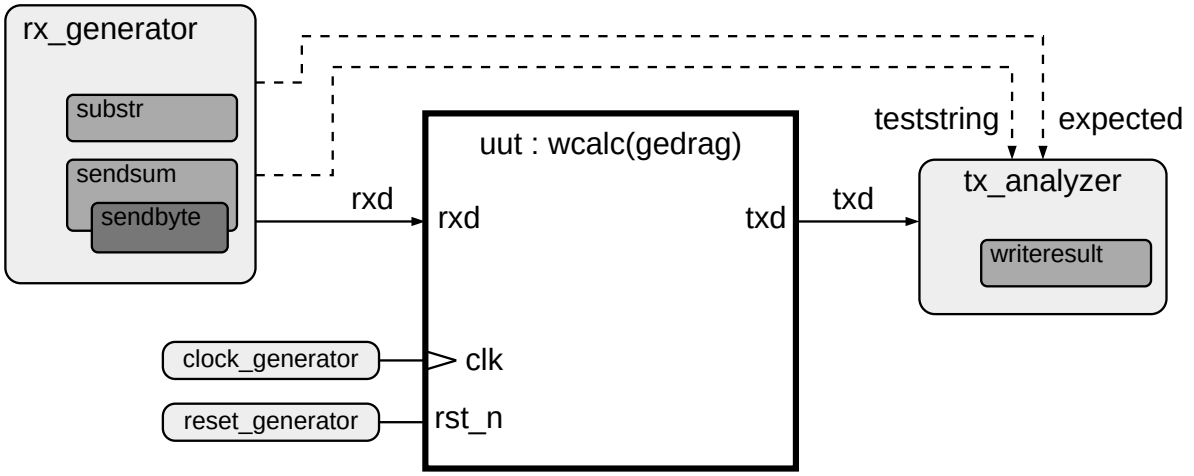


Figuur 10.11 : Het signaaldiagram van de rekenmachine. Het ingangssignaal is de som 25-114= en het uitgangssignaal is de uitkomst -89.

Hoewel de rekenmachine slechts één data-ingang heeft, is de signaalgeneratie en de interpretatie van het resultaat is niet eenvoudig. Uit het signaaldiagram van figuur 10.11 is de som en de uitkomst niet in één oogopslag te zien. Bovendien zal de rekenmachine voor diverse situaties getest moeten worden. Het loont daarom de moeite om voor deze rekenmachine een flexibele testomgeving te maken.

De testomgeving voor de rekenmachine

In figuur 10.12 staat de testomgeving voor de rekenmachine. Naast een klokgenerator en resetgenerator is er een signaalgenerator rx_generator die het ingangssignaal rxd creëert.



Figuur 10.12 : De testomgeving voor de rekenmachine. De signaalgenerator gebruikt twee procedures `sendsum` en `sendbyte` voor het omzetten van een teststring naar een serieel signaal. Daarnaast geeft de signaalgenerator via twee shared variabelen de teststring en verwachte uitkomst `expected` door aan het proces `tx_analyzer`. Dit proces drukt het resultaat af en vergelijkt `expected` met dit resultaat.

Het proces `tx_analyzer` analyseert de uitkomst van de rekenmachine, het vergelijkt de uitkomst met de verwachte waarde en drukt de uitkomst samen met de teststring af. De teststring en de verwachte uitkomst worden door de signaalgenerator gecreëerd en als shared variabelen door gegeven aan de analysator.

Het versturen van een teststring

Het seriële ingangssignaal `rxd` voor de rekenmachine bestaat uit één lange reeks enen en nullen. Het signaaldiagram van figuur 10.11 toont de reeks, die de som `25-114=` representeert. Signaal `txd` is de uitkomst van deze som en stelt de waarde `-89` voor.
In principe kan de signaalgenerator van de testbench een proces zijn dat steeds iedere baudperiode het ingangssignaal `rxd` hoog of laag maakt:

```
        rxd ='1'
        wait for BAUD_PERIOD;
        rxd ='0'                    -- startbit
        wait for BAUD_PERIOD;
        rxd ='0'                    -- bit 0 van databyte 001100110 ('2')
        wait for BAUD_PERIOD;
        rxd ='1'                    -- bit 1 van databyte 001100110 ('2')
        wait for BAUD_PERIOD;
```

Bij het testen moet de rekenmachine met meerdere sommen getest worden. In plaats van alle bits één voor één te beschrijven, is het duidelijker om de som als tekststring aan te bieden.

In code 10.15 staat de procedure `sendbyte`, die een byte naar de rekenmachine verstuurt. Uitgang `rx` is een signaal en ingang `c` is een constante. Op regel 120 wordt het karakter omgezet naar de overeenkomstige ASCII-waarde en vervolgens geconverteerd naar een unsigned getal. Het attribuut `pos` geeft de positie van het karakter in de typedefinitie van `character` en deze positie is gelijk aan de ASCII-waarde.

Code 10.15 : De procedure sendbyte die een byte serieel verstuurt.

```
117  procedure sendbyte(signal rx : out std_logic; constant c : in character) is
118    variable u : unsigned(7 downto 0);
119  begin
120      u := to_unsigned(character'pos(c), u'length);
121      rx <= '0';       -- startbit
122      wait for BAUD_PERIOD;
123      for i in u'reverse_range loop
124        rx <= u(i);    -- databits
125        wait for BAUD_PERIOD;
126      end loop;
127      rx <= '1';       -- stopbit/idle
128      wait for BAUD_PERIOD;
129  end procedure sendbyte;
```

De procedure sendbyte verstuurt eerst de startbit, daarna de acht databits en tenslotte de stopbit. Na ieder bit wordt er een periode BAUD_PERIOD gewacht. Deze tijd is omgekeerd evenredig met de baudsnelheid BAUD_RATE. Naast de BAUD_RATE en BAUD_PERIOD bevat de testbench nog drie andere constanten:

Code 10.16 : Diverse constanten uit de testbench.

```
49  constant n : natural := 64;
50  constant NUM_DIGITS  : natural := natural(floor(real(n-1)*log10(2.0))) + 1;
51  constant BAUD_RATE   : natural := 115200;
52  constant BAUD_PERIOD : time    := 1 sec / BAUD_RATE; -- is 8680 ns
53  constant DELAY       : time    := ((1+NUM_DIGITS+2+1)*10+1)*BAUD_PERIOD;
```

De constante n is de bitbreedte van de rekenmachine. Het aantal decimale cijfers NUM_DIGITS is een functie van de bitbreedte n. Na het verzenden van een complete teststring moet de signaalgenerator wachten totdat de rekenmachine de som heeft verwerkt. Deze tijd DELAY hangt af van het aantal karakters dat de rekenmachine verstuurt.

Code 10.17 : De procedure sendsum die een som verstuurt.

```
131  procedure sendsum(signal rx : out std_logic; constant sum : in string) is
132    variable i : integer := sum'left;
133  begin
134    while i <= sum'right loop
135      sendbyte(rx, sum(i));
136      i := i + 1;
137    end loop;
138    wait until txd='0';
139    wait for DELAY;
140  end procedure sendsum;
```

De procedure sendsum verstuurt een rij karakters naar de rekenmachine. Het while-statement string sum verstuurt met de procedure sendbyte één voor één de karakters uit sum totdat het einde van de string is bereikt. Daarna wacht de procedure totdat txd laag is en daarna nog een vaste tijd DELAY. Deze vertragingstijd hangt af van de bitbreedte van de rekenmachine en het aantal karakters dat maximaal verstuurd kan worden.

De datastructuur voor de testgegevens

De rekenmachine moet bij allerlei situaties worden getest: er zijn verschillende operatoren; de getallen kunnen positief en negatief zijn; er kan overflow optreden; de prioriteit van operatoren moet correct zijn en haakjes moeten op de juiste wijze gebruikt worden. Voor al deze situaties genereert het proces `rx_generator` een tekststring. Een methode is om een array te gebruiken waarin deze teststrings met de verwachte waarden zijn opgeslagen. De signaalgenerator verstuurt één voor één de teststrings naar de rekenmachine en stuurt deze gelijktijdig samen met de verwachte waarde naar de signaalanalysator .

Het nadeel van een array is dat de items hetzelfde formaat moeten hebben. In dit voorbeeld zijn de teststrings verschillend van lengte. Bij een streng getypeerde taal als VHDL is dat lastig.
In plaats van een array met strings kan er ook één grote string gebruikt worden. De teststrings en verwachte waarden staan achter elkaar en worden bijvoorbeeld gescheiden door een puntkomma. Figuur 10.13 geeft een voorbeeld.

```
"25-114=;-89;123+10*456-678=;4005;m(3+5)=;-8;...;..."
```

Figuur 10.13 : De string met de tests voor de rekenmachine. De teststrings hebben een licht grijze achtergrond, de verwachte waarden hebben een donker grijze achtergrond en het scheidingsteken is een puntkomma.

In code 10.18 staat de string waar de pointer `stimulus` naar wijst. Alle teststrings en verwachte waarden zijn apart vermeld. Het concatenatie-teken knoopt deze losse delen aan elkaar. Om de leesbaarheid te verbeteren staan de onderdelen in twee kolommen gescheiden door puntkomma's. De eerste kolom bevat de teststrings en de tweede het verwachte resultaat.

Code 10.18 : De pointer `stimulus` naar de string met de teststrings en de verwachte waarden.

```
57  shared variable stimulus : line := new string'(
58    "25-114="                        &";"& "-89"                      &";"&
59    "123+10*456-678="                &";"& "4005"                     &";"&
60    "m(3+5)="                        &";"& "-8"                       &";"&
61    "s(m(4-13))="                    &";"& "3"                        &";"&
62    "(123*10/1*10)+(456*678*1/10)="  &";"& "43216"                    &";"&
63    "5+m((3+7)*(8-5))+10="           &";"& "-15"                      &";"&
64    "a:?7="                          &";"& "?"                        &";"&
          ⋮
95    "123456787654321/11111111="      &";"& "11111111"                 &";"&
96    "10+9+8+7+6+5+4+3+2+1="          &";"& "10+9+8+7+6+5+4+3+2+1=55"  &";"&
97    "30+2*8-3="                      &";"& "43"
98  );
```

Bij het testen van de rekenmachine moeten uit `stimulus` alle teststrings en verwachte waarden worden gehaald. De procedure `substr` uit code 10.19 haalt uit een string een substring. De procedure heeft drie parameters met de modus **`inout`**. De pointers `s` en `sub` wijzen naar de plaats waar de string staat en waar de substring moet komen te staan. De variabele `pos` bevat de positie waar in `s` de substring gelezen moet worden en bevat na het lezen de positie van het volgende veld.

Code 10.19 : De procedure substr die een substring uit een string haalt.

```
101 procedure substr(s: inout line; sub: inout line;
102                  pos: inout natural; sep: in character) is
103 begin
104   loop
105     if pos > s'length then
106       exit;
107     end if;
108     if s(pos) = sep then
109       pos := pos+1;
110       exit;
111     end if;
112     write(sub,s(pos));
113     pos := pos+1;
114   end loop;
115 end procedure substr;
```

Code 10.20 : De C-versie van substr.

```
char *substr(char *s, char *sub,
                              char c)
{
  while (*s != '\0') {
    if (*s == ';') {
       *sub = '\0';
       return ++s;
    }
    *sub++ = *s++;
  }
  return s;
}
```

Omdat VHDL geen pointerrekenen kent, is bij de procedure substr een variabele pos nodig om de positie in de string s bij te houden. Voor het toevoegen van karakters aan de substring is de procedure write gebruikt. Het voordeel van de procedure write is dat deze automatisch de geheugenruimte vergroot.
De C-versie van substr uit code 10.20 bevat geen variabele pos en geen functie write. In plaats daarvan gebruikt C pointerrekenen en de dereferentie-operator *.

De signaalgenerator: proces rx_generator

Proces rx_generator beschrijft de signaalgenerator en staat in code 10.21. Aanvankelijk is signaal rxd hoog. Zolang er nog stimuli aanwezig zijn, wordt er een teststring met de bijbehorende verwachte waarde met behulp van de procedure substr uit stimulus gehaald en wordt de teststring met de procedure sendsum naar de rekenmachine gestuurd.

Code 10.21 : Het proces rx_generator dat de signaalgenerator beschrijft.

```
169   rx_generator : process is
170     variable pos : natural;
171   begin
172     rxd <= '1';
173     wait for BAUD_PERIOD;
174     pos   := 1;
175     while pos < stimulus'length loop
176       substr(stimulus,teststring,pos,';');
177       substr(stimulus,expected,pos,';');
178       sendsum(rxd, teststring.all);
179     end loop;
180     wait;
181   end process rx_generator;
```

Nadat alle stimuli zijn verwerkt, blijft het proces voor altijd wachten. De variabele pos bevat voortdurend de positie van het eerstvolgende veld uit de string stimulus die nog niet gelezen is.

De procedure `substr` schrijft de teststring en de verwachte waarde naar de locaties waar de shared variabelen `teststring` en `expected` naar wijzen. De definities van deze twee pointers staan in code 10.22.

Code 10.22 : De shared variabelen voor de teststring en de verwachte waarde.

```
55    shared variable expected   : line;
56    shared variable teststring : line;
```

De signaalanalysator: proces `tx_analyzer`

De signaalanalysator vergelijkt iedere uitkomst van de rekenmachine met de verwachte waarde en drukt de teststring en de gevonden waarde af. Als de verwachte en de gevonden waarde overeenkomen, wordt `OK` afgedrukt, als dat niet het geval is wordt **`NOT`** `OK` met de verwachte waarde afgedrukt.

Het proces `tx_analyzer` dat de signaalanalysator beschrijft, staat in code 10.23. Het proces wacht op een neergaande flank van het uitgangssignaal `txd` van de rekenmachine. Dit is de startbit van de eerste ASCII-waarde die verstuurd is. Na anderhalve baudperiode is de minst significante bit van deze ASCII-waarde beschikbaar. Achtereenvolgens worden alle acht bits opgeslagen in de vector `u`. Op regel 195 wordt deze vector omgezet naar een integer en omgezet naar de overeenkomstige ASCII-waarde `c` uit de ASCII-tabel. Het type-attribuut `val` geeft voor een integer het bijbehorende karakter.

Code 10.23 : Het proces `rx_generator` dat de signaalgenerator beschrijft.

```
183   tx_analyzer : process is
184     variable u     : unsigned(7 downto 0);
185     variable c     : character;
186     variable found : line;
187   begin
188     wait until falling_edge(txd);
189     wait for 1.5*BAUD_PERIOD;
190     u := (others => '0');
191     for i in u'range loop
192       u := txd & u(7 downto 1);
193       wait for BAUD_PERIOD;
194     end loop;
195     c := character'val(to_integer(unsigned(u(7 downto 0))));
196     if ((c >= '0') and (c <= '9')) or (c = '-') then
197       write(found, c);
198     elsif c = LF then
199       writeresult(found);
200       deallocate(found);
201       deallocate(teststring);
202       deallocate(expected);
203     end if;
204   end process tx_analyzer;
```

Als het karakter een cijfer of het minteken is, wordt het met de procedure `write` aan `found` toegevoegd. De variabele `found` is een lokaal gedeclareerde pointer naar een buffer. De procedure `write` vergroot automatisch de geheugenruimte van de buffer.

De rekenmachine sluit de karakters van de som af met de ASCII-waarden LF en CR. Als het karakter c gelijk is aan LF, kan het resultaat worden afgedrukt. Alle andere karakters worden genegeerd.
De procedure writeresult drukt het resultaat af en gebruikt hiervoor de shared variabelen teststring en expected. Na het afdrukken worden de geheugenruimte van de buffers teststring, expected en found vrijgemaakt. Bij het afdrukken van tekst is dat normaal gesproken niet nodig, omdat de procedure writeline de gebruikte buffer vrijmaakt.

Het maken van een geformatteerd overzicht met simulatieresultaten

In code 10.24 staat de procedure writeresult die het simulatieresultaat geformatteerd afdrukt. Eerst wordt met een concatenatie de inhoud van found aan die van teststring toegevoegd en naar een buffer L geschreven. Deze tekst wordt links uitgelijnd en aangevuld met spaties tot totaal 40 karakters.

Code 10.24 : De procedure writeresult die het simulatieresultaat afdrukt.

```
143   procedure writeresult(variable found : in line) is
144     variable L : line;
145   begin
146       write(L, teststring.all & found.all, LEFT, 40);
147       if found.all = expected.all then
148         write(L,string'("==> OK"));
149       else
150         write(L,string'("==> NOT OK expected "));
151         write(L,expected.all);
152       end if;
153       writeline(output,L);
154   end procedure writeresult;
```

Als de gevonden waarde gelijk is aan de verwachte waarde wordt aan L de string "==> OK" toegevoegd en anders wordt aan L de string "==> NOT OK expected " en de verwachte waarde toegevoegd. De procedure writeline stuurt de opgebouwde regel naar de standaarduitvoer. De simulator Modelsim geeft dit resultaat in het transcriptvenster:

```
# 25-114=-89                              ==> OK
# 123+10*456-678=4005                     ==> OK
# m(3+5)=-8                               ==> OK
# m3+5=2                                  ==> OK
# s(m(4-13))=3                            ==> OK
# (123*10/1*10)+(456*678*1/10)=43216      ==> OK
# 5+m((3+7)*(8-5))+10=-15                 ==> OK
# a:?7=7                                  ==> NOT OK expected ?
        ⋮
# 123456787654321/11111111=11111111       ==> OK
# 10+9+8+7+6+5+4+3+2+1=55                 ==> OK
# 30+2*8-3=43                             ==> OK
```

Met de waveformviewer van de simulator kunnen de interne signalen van de rekenmachine zeer gedetailleerd bestudeerd worden. De tekstuitvoer geeft direct een duidelijk overzicht van een flink aantal verschillende tests. Het grootste ge-

vaar tijdens het debuggen is dat de ontwerper zich focust op een specifiek probleem en dat repareert, maar daarbij tegelijkertijd een ander probleem creëert. Met het overzicht uit het transcriptvenster is dit direct zichtbaar.

De invoer van de teststrings is flexibel. Bij de declaratie van de variable `stimulus` in code 10.18 kunnen eenvoudig tests worden toegevoegd en verwijderd door voor een test twee commentaarstrepen te zetten. De volgorde kan met plakken en knippen ook relatief eenvoudig worden aangepast.

10.14 Het package `math_real` en andere numerieke packages

Hoewel er nieuwe ontwikkelingen zijn, zijn de meeste rekenkundige bewerkingen niet synthetiseerbaar. Bewerkingen als delen en de modulusfunctie zijn niet of maar gedeeltelijk synthetiseerbaar. Voor gehele getallen is delen door een macht van twee, hetzelfde als het naar rechts schuiven van de bits. De wortel en de exponentiële, logaritmische en goniometrische functies zijn ook niet synthetiseerbaar.

In het package `math_real` zijn allerlei constanten en rekenkundige functies gedefinieerd voor het type `real`. In bijlage B.4 staat een overzicht. Het package is niet bestemd voor synthese. Het heeft geen zin de wortelfunctie `sqrt` of de sinusfunctie `sin` in een ontwerp te gebruiken. Synthesizers kunnen deze functies niet implementeren.

Het package `math_real` kan in een ontwerp gebruikt worden bij de definities van constanten. De gedragsbeschrijving voor een parallelle omzetting van binair naar BCD uit code 8.22 gebruikt een constante `nbcd` dat het aantal BCD-cijfers vastlegt. Deze constante is met de functies `log10` en `floor` berekend uit het aantal bits `nbin` van de binaire representatie:

```
16   constant nbcd : natural := 4 * (natural(ceil(real(nbin)*log10(2.0))));
```

Voor code die niet gesynthetiseerd hoeft te worden, zoals bijvoorbeeld een testbench, kan het package `math_real` wel gebruikt worden.

VHDL-2008 kent synthetiseerbare fixed- en floating-point packages

In 2008 is door IEEE een nieuwe VHDL-standaard goedgekeurd. Een belangrijke toevoeging van VHDL-2008 zijn packages voor fixed-point- en floating-point- berekeningen. Het package `fixed_pkg` definieert twee typen `sfixed` en `ufixed` voor vastekommagetallen met en zonder teken. Het werken met vastekommagetallen wijkt niet wezenlijk af van dat met gehele getallen, omdat een vastekommagetal feitelijk een geheel getal is dat door een macht van twee is gedeeld. Het package `float_pkg` definieert een type `float` voor een willekeurig drijvendekommagetal en drie typen `float32`, `float64` en `float128` voor respectievelijk 32-, 64-, en 128-bits drijvendekommagetallen.

Het feit dat de fixed- en floating-point packages door IEEE goedgekeurd zijn, betekent niet dat alle nieuwe mogelijkheden beschikbaar zijn. In de ontwikkelomgevingen voor FPGA's zijn deze fixed- en floating-point packages niet of slechts gedeeltelijk geïmplementeerd. De goedkeuring door IEEE betekent dat de bouwers van deze ontwikkelomgevingen met de nieuwe standaard aan de slag kunnen. De komende jaren zullen bij iedere release van de software nieuwe aspecten van de VHDL-2008 beschikbaar zijn.

11

Technologie

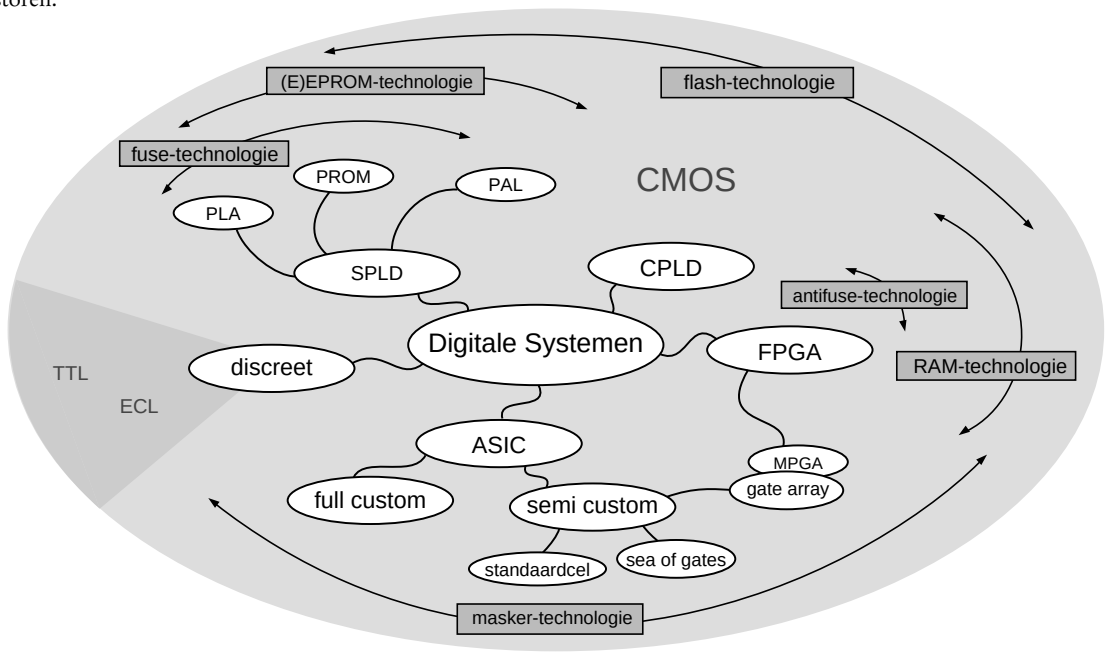
Doelstelling	In dit hoofdstuk leer je welke technieken er bestaan om digitale systemen te maken. Van elke implementatiemogelijkheid leer je de achterliggende technologie en de gebruikte architectuur kennen.
Onderwerpen	De behandelde onderwerpen zijn: ▪ De NMOS- en PMOS-transistor. ▪ De CMOS-technologie en de CMOS-inverter. ▪ Eenvoudige PLD's als PLA, PAL en PROM. ▪ De fuse-technologie. ▪ Het maskergeprogrammeerd ROM. ▪ De floating-gate-transistor. ▪ EPROM-, EEPROM- en flashtechnologie. ▪ Complexe PLD's. ▪ Antifuse-technologie en SRAM-technologie. ▪ FPGA's en *Look up tables*. ▪ Procestechnologie en het fullcustom IC-ontwerpen. ▪ De semicustom-ASIC met gate-array- en standaardceltechnologie. ▪ De wet van Moore en de toekomstige ontwikkelingen.

In hoofdstuk 1 zijn beknopt verschillende implementatiemogelijkheden voor digitale systemen besproken. Dit hoofdstuk gaat verder in op de technologie van de diverse realisatievormen. Door de grote verscheidenheid aan soorten bouwstenen en verschillende soorten architecturen is het een omvangrijk geheel. Figuur 1.2 geeft een overzicht met de belangrijkste typen bouwstenen.
In de loop der jaren zijn er veel programmeerbare bouwstenen en vele soorten ASIC's geïntroduceerd. Sommige daarvan waren succesvol, andere waren dat niet of zijn dat inmiddels niet meer. Dit hoofdstuk zet de verschillende mogelijkheden op een rij. De minder gangbare of verouderde bouwstenen worden om het overzicht compleet te houden wel besproken, maar krijgen minder aandacht.

Een complicerende factor is dat er bij de verschillende bouwstenen veel verschillende technieken worden toegepast. Dat maakt dat de bespreking al snel omvangrijk en onoverzichtelijk wordt. Toch is het handig om een overzicht te hebben over de gangbare en minder gangbare technieken. Zij zijn fundamenteel en vormen met elkaar de grondslag voor het digitaal ontwerpen.

Sommige componenten gebruiken BICMOS, dat is een combinatie van bipolaire transistoren en CMOS-transistoren.

De basis voor bijna alle technieken is CMOS. Alle moderne digitale elektronica wordt met deze technologie gemaakt. De basis van CMOS is de PMOS- en de NMOS-transistor. Alleen oudere technieken zijn gebaseerd op alleen NMOS of alleen PMOS. Oudere programmeerbare bouwstenen en discrete componenten zijn opgebouwd met bipolaire transistoren.



Figuur 11.1 : **Overzicht van de implementatiemogelijkheden voor digitale systemen.** De witte ovalen geven de verschillende soorten implementaties. De grijze rechthoeken geven de bijbehorende technologie. De onderliggende technologie is in bijna alle gevallen CMOS.

TTL staat voor *transistor transistor logica* en ECL staat voor *emitter transistor logica*. Beide technieken gebruiken bipolaire transistoren voor de logica.

Figuur 11.1 geeft een overzicht van de verschillende mogelijkheden om digitale systemen te realiseren. De belangrijkste basisvormen staan ook in figuur 1.2: SPLD's, CPLD, FPGA, ASIC en een schakeling met discrete componenten.
SPLD staat voor *simple programmable logical device* en wordt onderverdeeld in drie typen: PLA, PROM en PAL. De meest gebruikte programmeerbare bouwsteen is de PAL. Dit is de reden dat alleen deze naam in figuur 1.2 is genoemd.
Voor SPLD's werd aanvankelijk de fuse-technologie gebruikt. Later zijn daar de EPROM- en EEPROM-technologie bij gekomen. Tegenwoordig wordt voor geheugens flash gebruikt. Deze technologie wordt ook bij de CPLD's toegepast.

Hoewel veel aspecten bij CPLD's en FPGA's anders zijn, worden de verschillen tussen deze componenten kleiner. Er bestaan sinds kort ook FPGA's met flash-technologie naast de bij FPGA's gangbare antifuse- en RAM-technologie.

De basis voor ASIC's is CMOS. Bij een fullcustom-IC ontwerpt de ontwerper zelf alle maskers die nodig zijn voor het productieproces. Bij een semicustom-ontwerp gebruikt de ontwerper bibliotheken met vooraf ontworpen cellen. Bij een gate-array en een sea-of-gates zijn de transistoren al op het IC aangebracht. De ontwerper levert alleen de maskers voor de verbindingen aan. Voor een standaardcel-IC moeten alle maskers opnieuw ontworpen worden. De gate-array's hebben een overlap met de FPGA's. Sommige FPGA-fabrikanten geven de mogelijkheid om een maskergebaseerde versie van de FPGA te maken.

11.1 CMOS

De basis voor moderne digitale geïntegreerde schakelingen is CMOS. CMOS staat voor *complementair metal oxide semiconductor*. CMOS-schakelingen zijn opgebouwd uit zowel PMOS- als NMOS-transistoren.

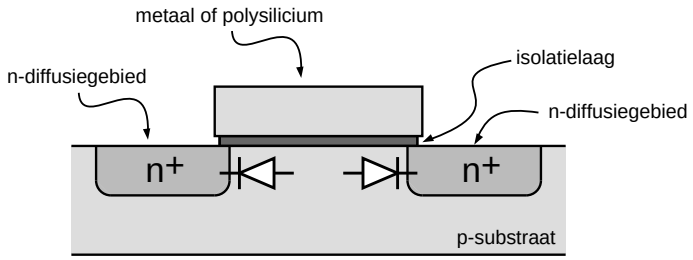
De MOS-transistor wordt ook bij analoge elektronica gebruikt. De functionaliteit komt overeen met die van een JFET. Men noemt de MOS-transistor daarom ook wel MOSFET.

De MOS-transistor als schakelaar

Een MOS-transistor bestaat uit een substraat van halfgeleider materiaal met daarop een dunne isolatielaag en daar boven een geleider. Het substraat is zwak gedoteerd silicium. Voor PMOS is dat n-materiaal en voor NMOS is dat p-materiaal. De isolatielaag bestaat uit siliciumoxide, SiO_2. Oorspronkelijk werd als geleider aluminium gebruikt, later is dit metaal vervangen door polysilicium. Het acroniem MOS staat voor *metal oxide semiconductor* en representeert de drie lagen van de transistor. Het metaal/polysilicium is de *gate* van de MOS-transistor.

Doteren is het opzettelijk in zeer geringe mate verontreinigen met vreemde atomen.

Als verbinding tussen de transistoren werd meestal aluminium gebruikt. Tegenwoordig is dat vaak koper vanwege de betere geleidbaarheid.

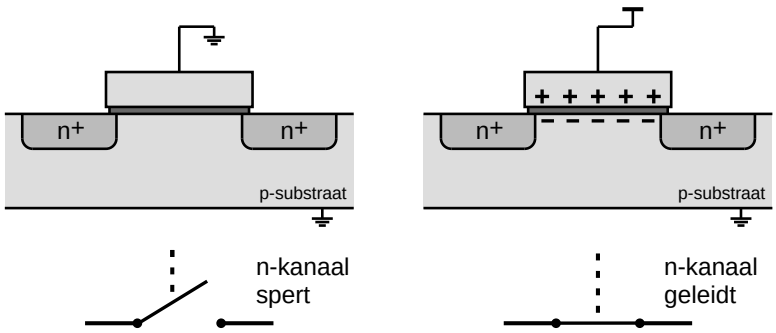


Figuur 11.2 : **Een dwarsdoorsnede van een NMOS-transistor.** De pn-overgangen tussen de n-diffusiegebieden en het p-substraat vormen twee diodes die tegengesteld gericht zijn.

De isolatielaag was meestal SiO_2. Bij moderne IC's is dat tegenwoordig een materiaal met een hogere diëlektrische constante, zoals TiO_2 en HfO_2.

In figuur 11.2 staat een schematische dwarsdoorsnede van een NMOS-transistor. In het zwak gedoteerde p-substraat liggen naast de gate twee sterk gedoteerde n-gebieden. Bij de productie van de transistor worden deze gebieden door middel van diffusie aangebracht, vandaar dat deze diffusiegebieden worden genoemd.

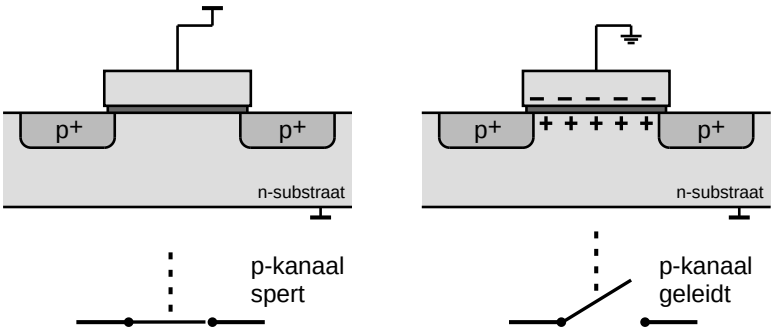
De pn-overgangen tussen de n-diffusiegebieden en het p-substraat vormen twee diodes, die tegengesteld gericht zijn. Zonder gate kan er nooit een stroom tussen de twee diffusiegebieden lopen.



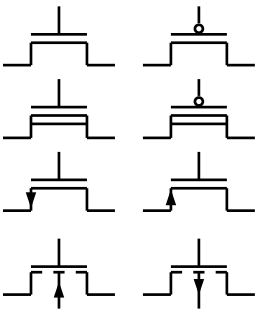
Figuur 11.3 : **De werking van een NMOS-transistor.** Als de gate-spanning hoog is, geleidt de transistor en als deze laag is, spert de transistor.

Figuur 11.3 laat het gedrag van de NMOS-transistor zien. Als de gate van een NMOS-transistor ten opzichte van het substraat een positieve spanning krijgt,

wordt de gate positief geladen. Als de gatespanning groter is dan de drempelspanning U_{th}, ontstaat er tussen de diffusiegebieden direct onder de gate een geleidend kanaal met negatieve ladingdragers. Met als gevolg dat er nu wel een stroom tussen de diffusiegebieden kan lopen.
De NMOS-transistor werkt dus als een schakelaar, als de gate hoog is geleidt de transistor en als de gate laag is spert de transistor.

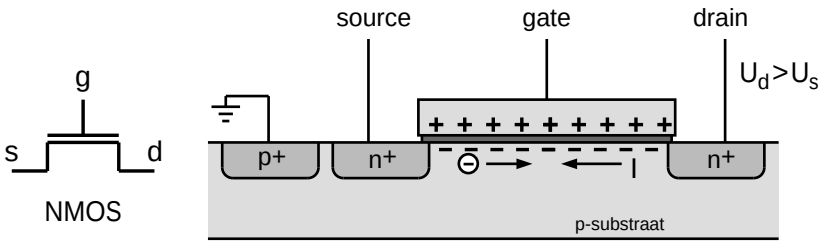


Figuur 11.4 : De werking van een PMOS-transistor Als de gate-spanning laag is ten opzichte van van het substraat, geleidt de transistor en als deze hoog is, spert de transistor.

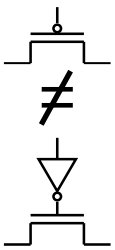


Figuur 11.5 : Alternatieve symbolen voor MOS-transistoren. Links staat steeds het NMOS- en rechts het bijbehorende PMOS-symbool. De onderste symbolen hebben het substraat of back-gate als vierde aansluiting.

Voor een PMOS-transistor geldt hetzelfde. Alleen geleidt de PMOS-transistor als de gate juist laag is ten opzichte van het n-substraat. Er ontstaat onder de gate een geleidend kanaal met positieve ladingdragers. Het gevolg is dat er dan een stroom tussen de diffusiegebieden kan lopen. In dit geval gaat het niet om een elektronenstroom maar om een gatenstroom.
De PMOS-transistor werkt — op dezelfde manier als de NMOS-transistor — als een schakelaar. Alleen geleidt de PMOS-transistor als de gate laag is en spert de transistor als de gate hoog is. Het gedrag van een PMOS-transistor is complementair aan dat van een NMOS-transistor.

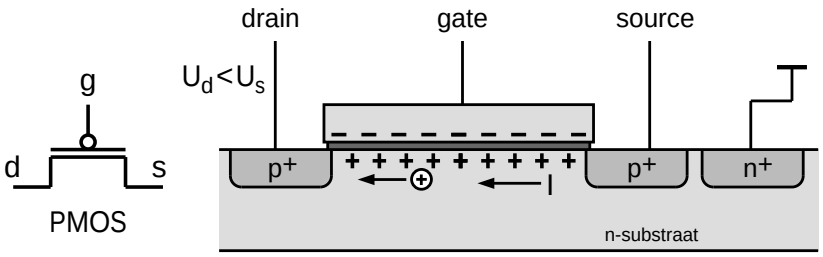


Figuur 11.6 : Symbool en namen voor de aansluitpunten bij de NMOS-transistor.



Figuur 11.7 : Let op! Een PMOS is niet hetzelfde als een inverter met een NMOS.

Een MOS-transistor is symmetrisch. De twee diffusiegebieden zijn inwisselbaar. De aansluiting waar de ladingdragers in het kanaal vandaan komen, wordt de *source* genoemd en de aansluiting waar de ladingdragers naar toe gaan is de *drain*. Figuur 11.6 laat zien dat bij een NMOS-transistor de elektronen van de source naar de drain gaan en dat de stroom van de drain naar de source loopt. Figuur 11.8 laat zien dat bij een PMOS-transistor de gaten van de source naar de drain gaan en dat de stroom van de source naar de drain loopt.

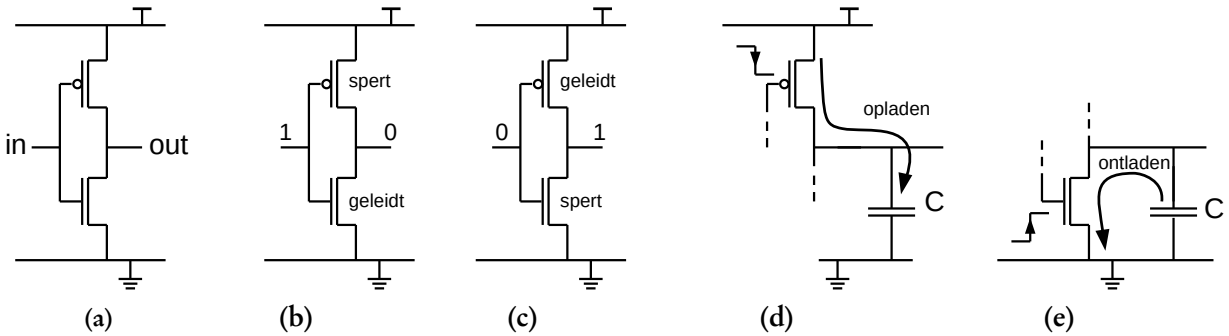


Figuur 11.8 : Symbool en namen voor de aansluitpunten bij de PMOS-transistor.

Het substraat van een MOS-transistor wordt de bulk of de *back gate* genoemd. Bij digitale IC's zijn de back gates van de NMOS-transistoren met de referentie verbonden en die van de PMOS-transistoren met de voedingsspanning.

De CMOS-inverter

De letter 'C' in het acroniem CMOS staat voor complementair. In een CMOS-schakeling komen evenveel NMOS- als PMOS-transistoren voor.
Het schema van de CMOS-inverter staat links in figuur 11.9. Als de ingang hoog is, geleidt de NMOS-transistor en spert de PMOS-transistor. De uitgang zal dan laag zijn. Als de ingang laag is, geleidt de PMOS-transistor en spert de NMOS-transistor. De uitgang zal dan hoog zijn. Er lopen alleen stromen als de ingang van niveau verandert. Als de ingang niet verandert, loopt er geen stroom en is de dissipatie nul.



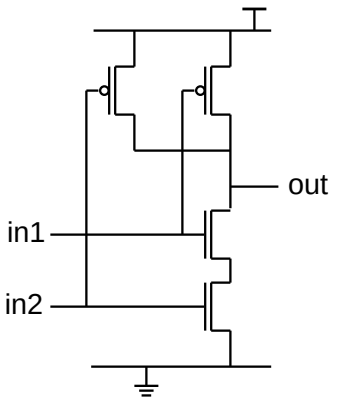
Figuur 11.9 : De bouw en de werking van de CMOS-inverter. Figuur a geeft de bouw van de CMOS-inverter. De figuren b en c laten zien dat de uitgang laag is als de ingang hoog is en omgekeerd. Figuur d laat zien dat als de ingang laag wordt, de NMOS spert en de capaciteit C via de PMOS-transistor opgeladen wordt. Figuur e laat zien dat als de ingang hoog wordt, de PMOS spert en de capaciteit C via de NMOS-transistor ontladen wordt.

De capaciteit C representeert de uitgangscapaciteit van de inverter, de ingangscapaciteiten van de aangesloten poorten en de bedradingscapaciteiten.

Figuur 11.9 laat verder zien dat bij het omlaag en omhoog gaan van de ingang de capaciteit C wordt opgeladen of wordt ontladen. Als de ingang laag wordt, dan zal de NMOS-transistor sperren en gaat de PMOS-transistor geleiden. Via deze PMOS-transistor wordt de uitgangscapaciteit C opgeladen en wordt de uitgang hoog.
Omgekeerd zal als de ingang hoog wordt, de PMOS-transistor gaan sperren en de NMOS-transistor gaan geleiden. De uitgangscapaciteit C wordt dan via de NMOS-transistor ontladen.

CMOS-logica

In figuur 11.10 is een NAND met twee ingangen getekend. Er is alleen een pad naar de referentie als de beide ingangen hoog zijn. In alle andere gevallen is er een pad naar de voeding. De uitgang is daarom alleen laag als de beide ingangen hoog zijn. Dit is ook te zien in de waarheidstabel, die naast de NAND staat.

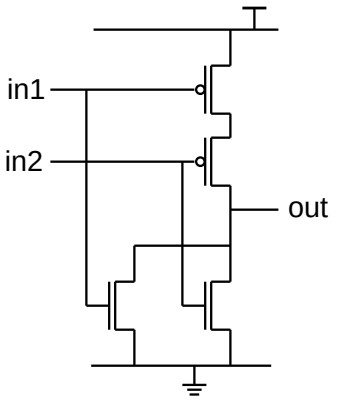


in1	in2	out
0	0	1
0	1	1
1	0	1
1	1	0

Figuur 11.10 : **Een CMOS 2-input NAND.** Links staat het schema van een NAND met twee ingangen en rechts staat de waarheidstabel. Als de ingangen allebei hoog zijn, geleiden de NMOS-transistoren en sperren de PMOS-transistoren. De uitgang is dan laag. In alle andere gevallen is de uitgang hoog.

In figuur 11.11 is een NOR met twee ingangen getekend. Er is alleen een pad naar de voeding als de beide ingangen laag zijn. In alle andere gevallen is er een pad naar de referentie. De uitgang is daarom alleen hoog als de beide ingangen laag zijn. Dit is ook te zien in de waarheidstabel, die naast de NOR staat.

Omdat de geleiding van elektronen beter gaat dan die van gaten, is het verstandig om de PMOS-transistoren niet in serie te gebruiken. Bij CMOS hebben NAND-poorten daarom de voorkeur boven NOR-poorten.



in1	in2	out
0	0	1
0	1	0
1	0	0
1	1	0

Figuur 11.11 : **Een CMOS 2-input NOR.** Links staat het schema van een NOR met twee ingangen en rechts staat de waarheidstabel. Als de ingangen allebei laag zijn, geleiden de PMOS-transistoren en sperren de NMOS-transistoren. De uitgang is dan hoog. In alle andere gevallen is de uitgang laag.

Het is niet ingewikkeld om een NAND met drie of vier ingangen te maken. Een NAND met drie ingangen heeft drie NMOS-transistoren die in serie staan en drie PMOS-transistoren die parallel staan.

11.2 SPLD

Zoals eerder is gezegd staat SPLD voor *simple programmable logical device*. Voor het ontwerp van digitale systemen zijn deze bouwstenen achterhaald. Wel vormen ze de basis voor verschillende andere componenten en technieken: de CPLD is gebaseerd op een PAL; de opbouw van de PLA wordt gebruikt bij fullcustom-IC's en de PROM is verder geëvolueerd naar EEPROM en flash. Daarom blijft het interessant om op de hoogte te zijn van de opbouw en de technologie van de diverse soorten SPLD's.

Som van producten

Iedere combinatorische schakeling is te schrijven als een som van producten. Voor elke combinatorische schakeling is immers een waarheidstabel op te stellen.

Een productterm is de AND-functie van de waarden of de geïnverteerde waarden. De term `abc'` is de AND-functie van `a`, `b` en de geïnverteerde `c`.

Een somterm is de OR-functie van de waarden of hun geïnverteerde waarde. De term `a+b'+c` is de OR-functie van `a`, `c` en de geïnverteerde `b`.

Tabel 11.1 geeft de waarheidstabel voor een schakeling met drie ingangen `a`, `b` en `c` en twee uitgangen `f` en `g`. In de tabel staan alle combinaties voor de ingangen met de bijbehorende uitgangswaarden. Als `a`, `b` en `c` respectievelijk `0`, `1` en `0` zijn, is uitgang `f` laag en uitgang `g` hoog.

Tabel 11.1 : Voorbeeld waarheidstabel met de bijbehorende logische vergelijkingen.

a	b	c	f	g
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

```
f = a'bc + ab'c' + ab'c + abc' + abc    (11.1)
g = ab'c + a'bc + ab'c + abc            (11.2)
```

In de digitale techniek worden veel verschillende notaties gebruikt voor de logische vergelijkingen:

```
g = a'b + ac
g = /ab + ac
g = ā.b + a.c
g = !a&b | a&c
```

De prioriteit van de productterm is hier hoger dan die van de somterm. Als daar onduidelijkheid over bestaat, worden haakjes gebruikt:

```
g = ((/a)b) + (ac)
g = (ā).b + (a.c)
```

Signaal `f` is hoog als één van de laatste vijf producttermen uit de tabel hoog is. Uitgang `f` is de somterm van deze vijf producttermen, zoals logische vergelijking 11.1 laat zien. Op dezelfde manier is de functie van `g` gelijk aan vergelijking 11.2 en is het de som van vier producttermen.

Tabel 11.2 : Voorbeeld functietabel met de bijbehorende logische vergelijkingen.

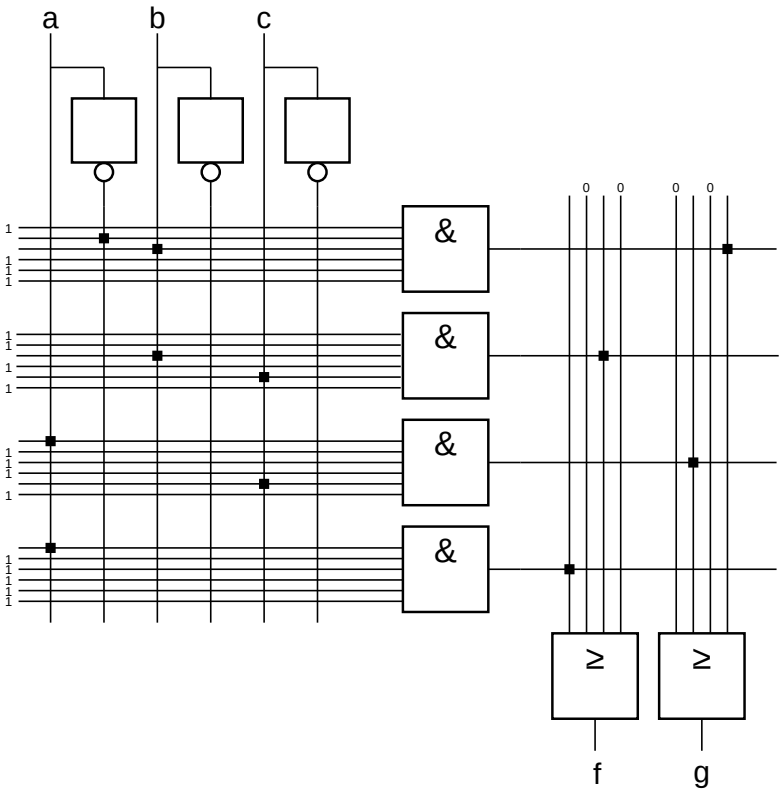
a	b	c	f	g
0	1	-	.	1
-	1	1	1	.
1	-	1	.	1
1	-	-	1	.

```
f = a + bc      (11.3)
g = a'b + ac    (11.4)
```

De waarheidstabel kan met optimalisatietechnieken geminimaliseerd worden. De geminimaliseerde versie van waarheidstabel 11.1 is functietabel 11.2. Het minteken in de functietabel betekent dat de betreffende ingang niet meedoet bij de productterm. Een nul betekent dat de ingang geïnverteerd wordt. De eerste productterm is `01-` en komt overeen met `a'b`. Ingang `a` is geïnverteerd, `b` is niet geïnverteerd en `c` doet niet mee.

Een punt in het rechter deel van de functietabel betekent dat de betreffende productterm niet meedoet met de somterm. De logische vergelijkingen 11.3 en 11.4 zijn de geoptimaliseerde versies van de vergelijkingen 11.1 en 11.2 en komen overeen met functietabel 11.2.

Iedere combinatorische schakeling is te beschrijven met een functietabel. Het linker deel van de tabel bevat producttermen en in het rechter deel staan de somtermen. Als de ingangssignalen ook geïnverteerd beschikbaar zijn, is iedere combinatorische schakeling te schrijven als een som van producten en te realiseren met tweelaagslogica. De producttermen worden gerealiseerd met AND-functies en de somtermen met OR-functies.

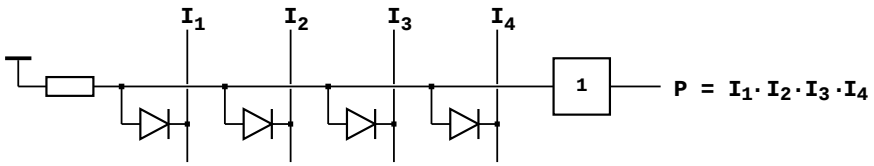


Figuur 11.12 : **Realisatie met behulp van AND- en OR poorten.** De schakeling bestaat uit vier 6-input AND-poorten en twee 4-input OR-poorten. Niet gebruikte ingangen van de AND's zijn hoog en die van de OR's zijn laag. De drie ingangen zijn, hoewel deze niet allemaal gebruikt worden, ook geïnverteerd beschikbaar. De verbindingen tussen de signaallijnen zijn met stippen aangegeven.

In figuur 11.12 staat de implementatie van functietabel 11.2. De structuur met deze AND- en OR-poorten is flexibel. Elke combinatorische functie met maximaal drie ingangen, twee uitgangen en vier producttermen kan met deze structuur worden gerealiseerd, alleen zullen voor een andere functionaliteit de verbindingen tussen de signaallijnen bij andere knooppunten zitten.

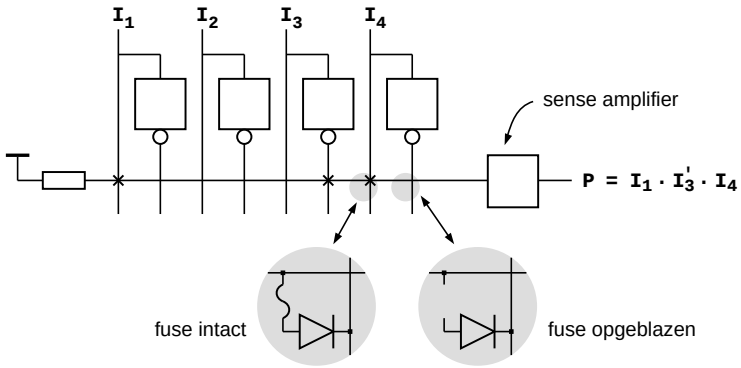
Fuse-technologie

Figuur 11.12 geeft het logische model voor zowel de PLA, de PROM, als voor de PAL, alleen gebruiken deze componenten geen AND- en OR-poorten, maar zogenoemde *wired-AND*-constructies. In figuur 11.13 is een *wired-AND* getekend. De horizontale lijn is via een pullup-weerstand verbonden met VCC. Tussen de horizontale en verticale lijnen zit een diode. Als één of meer van de ingangen I_1, I_2, I_3 en I_4 laag is, wordt de spanning van de horizontale lijn omlaag getrokken. Alleen als alle ingangen hoog zijn, blijft de spanning op de lijn hoog. Uitgang P is de AND-functie van de vier ingangen I_1, I_2, I_3 en I_4.



Figuur 11.13 : De *wired-AND* constructie.

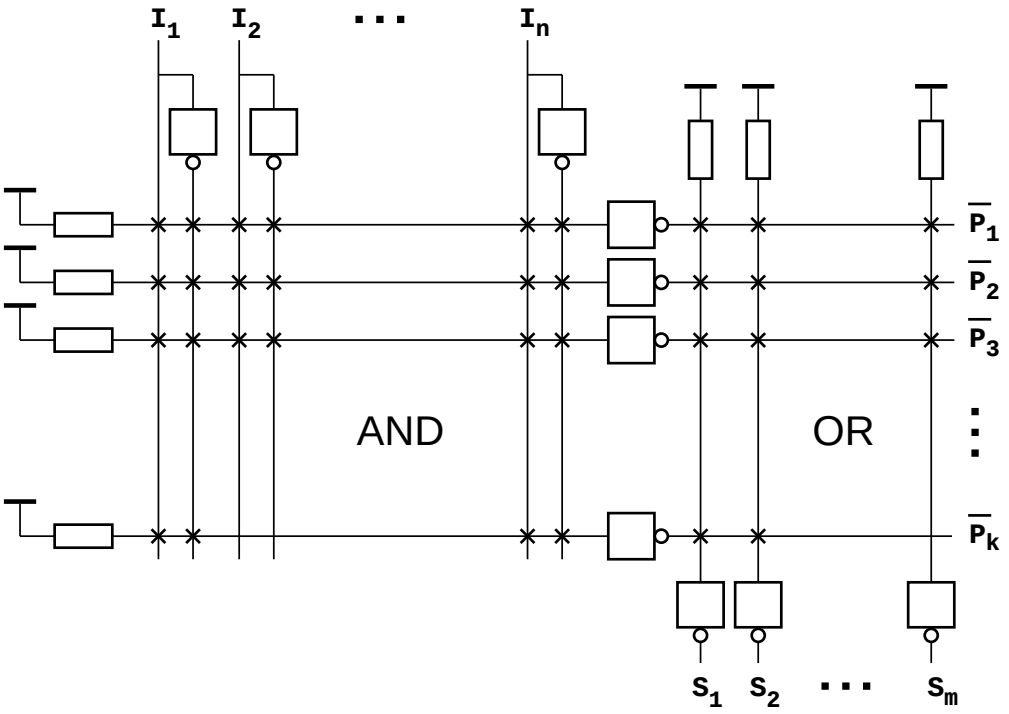
Figuur 11.14 geeft dezelfde wired-AND als figuur 11.13, alleen zijn de ingangen ook geïnverteerd beschikbaar en zijn de knooppunten programmeerbaar. Tussen de diodes en de lange lijn zijn *fuses* of zekeringen, aangebracht. Deze zekering werkt in principe op dezelfde manier als een smeltzekering. Als er een grote stroom door loopt, smelt de verbinding weg. Bij deze *fuse link*-technologie wordt dit bereikt door een hoge programmeerspanning aan te brengen.



Figuur 11.14 : De wired-AND zoals deze gebruikt wordt in een SPLD. De ingangssignalen worden geïnverteerd en niet geïnverteerd aangeboden. Een kruis geeft aan dat de fuse intact is en dat er een verbinding is.

Een kruis bij een knooppunt in figuur 11.14 geeft aan dat fuse nog intact is. Bij de andere knooppunten zijn de fuses opgeblazen en is er geen verbinding tussen de lange lijn en de diodes. Het effect is dat alleen I_1, I_4 en de geïnverteerde van I_3 uitgang P laag kunnen maken. Zodoende is de uitgang hoog, als deze signalen alle drie hoog zijn. Signaal P is gelijk aan $I_1 \cdot I_3' \cdot I_4$

Een wired-AND uit een SPLD heeft meestal veel meer aansluitingen dan de acht uit figuur 11.14. De capaciteit van de lange lijn is dan relatief hoog. Omdat de tijdvertraging evenredig is met de capaciteit, zou de SPLD traag worden als de lijn helemaal omlaag getrokken moet worden. Daarom is de buffer bij de uitgang een zogenoemde *sense amplifier*, die een kleine verandering van de spanning detecteert en de uitgang veel sneller laat reageren.



Figuur 11.15 : De AND-OR-structuur gerealiseerd met behulp van wired NAND's.

Een AND-OR-functie is met de regels van De Morgan ook te schrijven in de vorm van een NAND-NAND-functie:

$$A \cdot B + C \cdot D \;=\; \overline{\overline{A \cdot B + C \cdot D}} \;=\; \overline{\overline{A \cdot B} \cdot \overline{C \cdot D}} \tag{11.5}$$

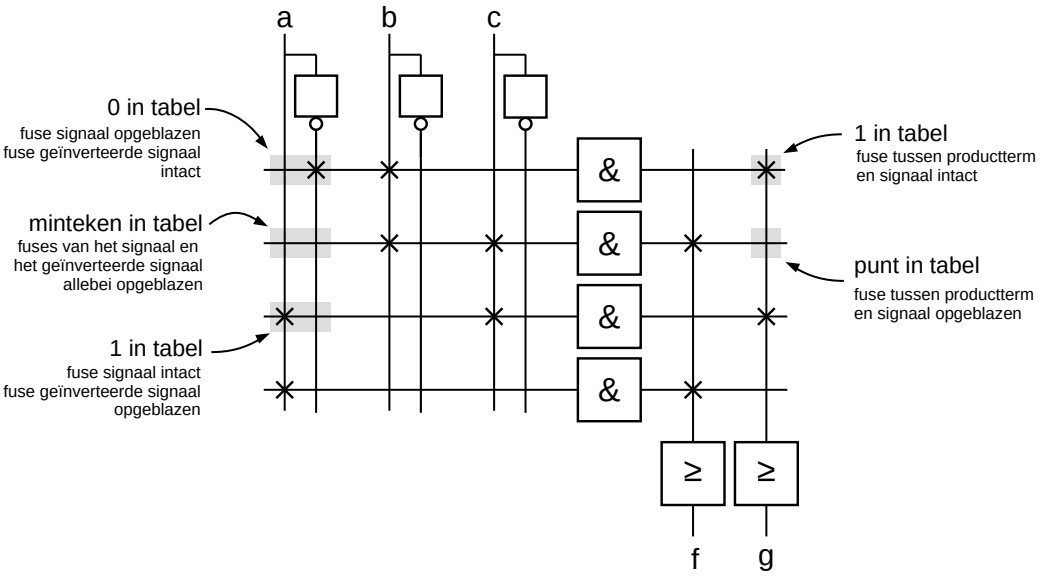
De AND-OR uit figuur 11.12 kan ook gerealiseerd worden met de programmeerbare NAND-NAND uit figuur 11.15. Het AND-vlak en het OR-vlak zijn beide gerealiseerd met wired-AND's, of eigenlijk wired-NAND's. Deze implementatie heeft n ingangen, k producttermen en m somtermen en is geschikt voor alle schakelingen met n ingangen en m uitgangen mits er niet meer dan k producttermen nodig zijn om het gedrag te beschrijven.

PLA

De AND-OR-structuur uit figuur 11.15 noemt men een PLA, *programmable logic array*. De schakeling van functietabel 11.2 kan gerealiseerd worden met een PLA met drie ingangen, twee uitgangen en vier producttermen.
In figuur 11.16 staat een kopie van tabel 11.2 en een implementatie van deze tabel met een PLA. De getekende PLA is een vereenvoudigde vorm van figuur 11.15. De pullup-weerstanden zijn niet getekend, de sense-amplifiers zijn niet-inverterend en hebben het AND-teken of het OR-teken gekregen om te benadrukken dat de PLA in feite uit een AND- en een OR-vlak bestaat.

a	b	c	f	g
0	1	-	.	1
-	1	1	1	.
1	-	1	.	1
1	-	-	1	.

de functietabel



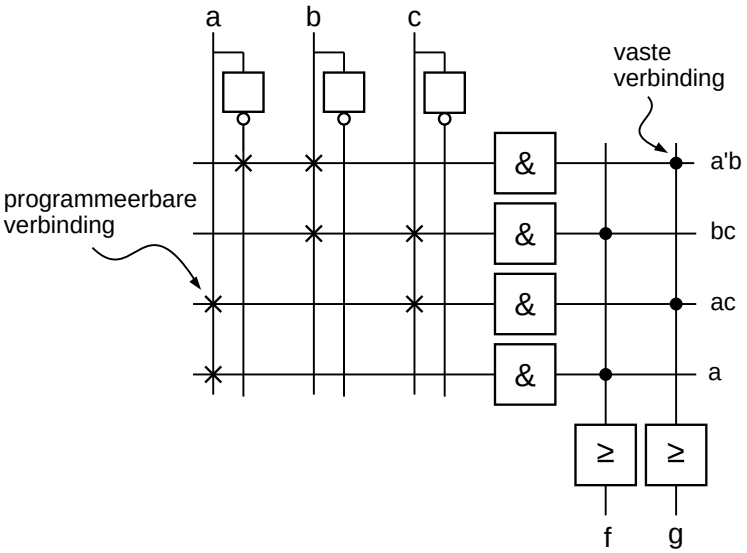
Figuur 11.16 : De realisatie van de functietabel van tabel 11.2 met een PLA.

Een 1 in het linker deel van de tabel komt overeen met een intacte fuse in het AND-vlak bij de betreffende signaallijn. Een 0 in het linker deel van de tabel komt overeen met een intacte fuse in het AND-vlak bij de geïnverteerde lijn van het betreffende ingangssignaal. Een minteken in het linker deel van de tabel betekent dat de fuses van het betreffende signaal en het geïnverteerde signaal allebei opgeblazen zijn.
Een 1 in het rechter deel van de tabel komt overeen met een intacte fuse in het OR-vlak voor de betreffende productterm en het bijbehorende uitgangssignaal. Een · in het rechter deel van de tabel betekent dat de fuse in het OR-vlak voor de betreffende productterm en het bijbehorende uitgangssignaal opgeblazen is.

Eind jaren zeventig waren de eerste PLA's beschikbaar. Begin jaren tachtig zijn door Signetics, toen een onderdeel van Philips, aan de PLA registers toegevoegd. Daarmee was er een programmeerbare bouwsteen beschikbaar waarmee eenvoudig toestandsmachines gerealiseerd konden worden. Het register is het toestandsregister en de PLA is het toestandsdecoder en de uitgangsdecoder. Deze bouwsteen wordt PLS, *programmable logic sequencer* of kortweg sequencer genoemd.
Ondanks dat de PLA als bouwsteen verouderd is, worden bij het fullcustom ontwerpen van geïntegreerde schakelingen nog steeds PLA-generatoren gebruikt om grote blokken logica te maken. De gegenereerde layout is gebaseerd op een AND-OR-constructie van de PLA.

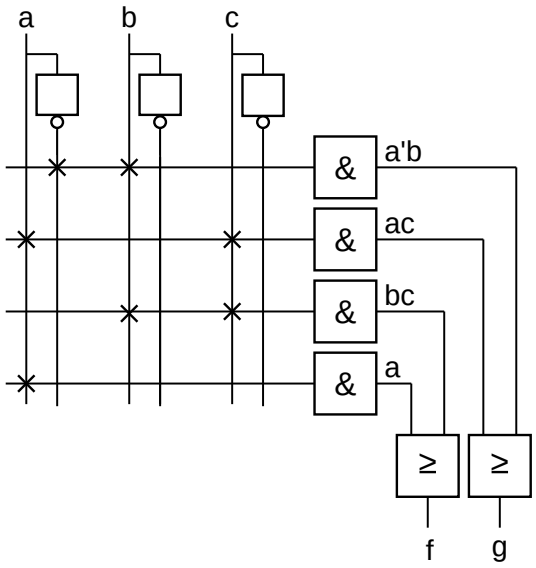
PAL

In de jaren negentig kwamen er SPLD's op de markt, die gebaseerd waren op de van de PLA-structuur afgeleide PAL, *programmable array logic*, Een PLA heeft een programmeerbaar AND-vlak en een programmeerbaar OR-vlak. De PAL heeft alleen een programmeerbaar AND-vlak. Het OR-vlak ligt vast. In figuur 11.17 is de schakeling van tabel 11.2 als PAL getekend. Het AND-vlak is programmeerbaar en is identiek aan dat van de PLA uit figuur 11.16.



Figuur 11.17 : De realisatie van tabel 11.2 met een PAL-structuur. Het AND-vlak is programmeerbaar. Het OR-vlak is niet programmeerbaar.

Het OR-vlak met vaste verbindingen is vervangbaar door vaste OR-poorten. In figuur 11.18 is dat gedaan en is de volgorde van de producttermen aangepast. Een nadeel van een PAL is dat per uitgang alle producttermen apart gemaakt moeten worden. Bij een PLA is elke productterm bij elke uitgang beschikbaar.

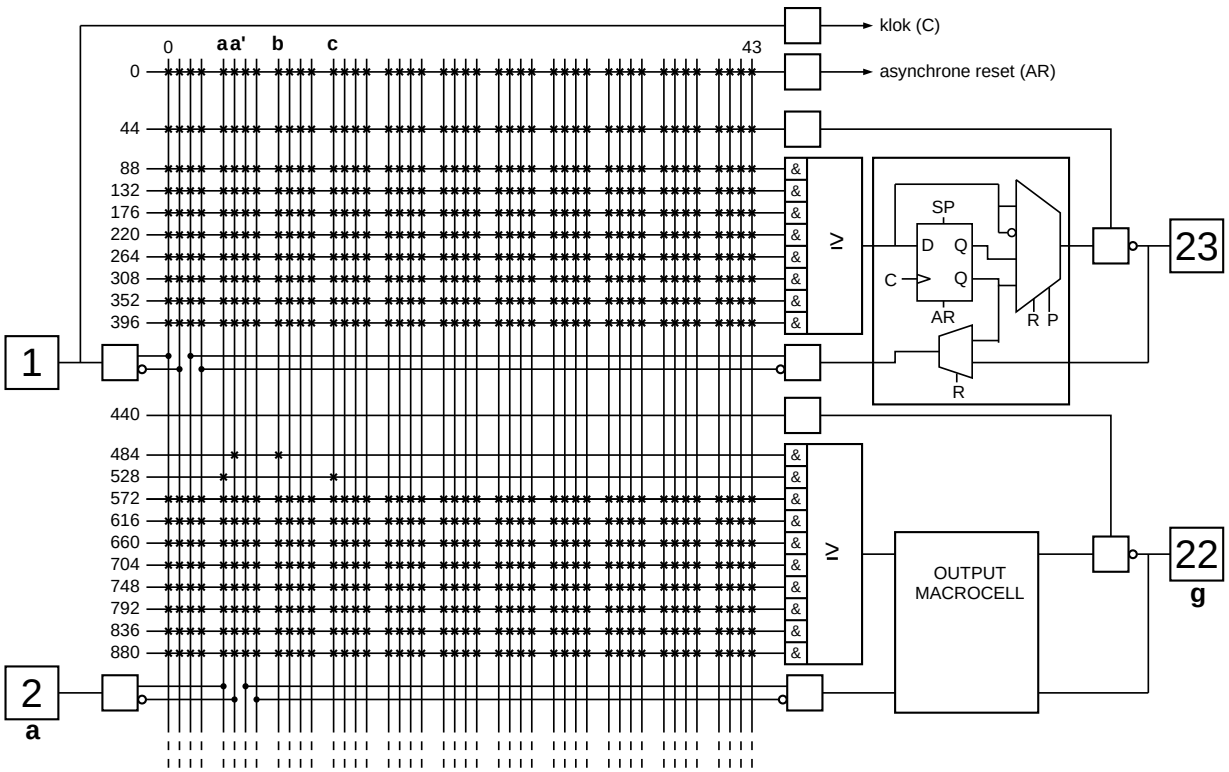


Figuur 11.18 : De realisatie van tabel 11.2 met een PAL met vaste OR-poorten.

Een geregistreerde uitgang is de letterlijke vertaling van de Engelse uitdrukking *registered output*.

Populaire PAL's waren de 16V8 en de 22V10. Deze bouwstenen hadden een zogenoemde macrocel met een D-flipflop bij elke uitgangspin. Met twee fuses kon deze uitgang op vier manieren gebruikt, als combinatorische of als geregistreerde uitgang en geïnverteerd of niet-geïnverteerd. Een geregistreerde uitgang is een uitgang met een flipflop.

De 22V10 heeft 22 bruikbare pinnen. In een 24-pins behuizing is pin 24 de VCC en pin 12 de referentie (GND). Pin 14 tot en met 23 zijn de 10 pinnen die of in- of uitgang kunnen zijn. De overige twaalf pinnen zijn altijd ingangen. Van deze PAL is in figuur 11.19 het deel getekend met de ingangspinnen 1 en 2 en de in- of uitgangspinnen 22 en 23.

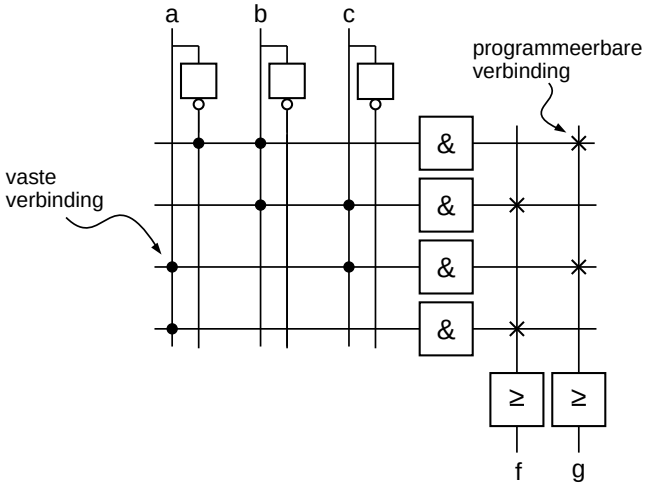


Figuur 11.19 : **Een realisatie van tabel 11.2 met een PAL22V10.** Er is slechts een deel getekend. Pin 23 wordt niet gebruikt en pin 22 is de uitgang voor signaal `g`. Pin 2, 3 en 4 komen overeen met respectievelijk de signalen `a`, `b` en `c`. Fuse 489 is verbonden met de inverterende ingang van pin 2. Fuse 532 is verbonden met de niet-inverterende ingang van pin 2. Fuse 492 en 536 zijn respectievelijk verbonden met pin 3 en pin 4, die beide niet zichtbaar zijn. De uitgang van pin 22 is dan de som van de producttermen `a'b` en `ac`.

De figuur toont de acht producttermen van pin 23 en de tien producttermen van pin 22. Pin 23 wordt niet gebruikt; alle fuses zijn nog intact. Omdat alle ingangen ook geïnverteerd met het AND-vlak zijn verbonden, zijn er altijd signalen die de lijnen omlaag trekken. Het signaal AR, de asynchrone reset van de flipfloppen, is in dit voorbeeld dus altijd laag. Ook de lijn met fuse 44 is altijd laag, zodat de tristate-inverter bij pin 23 hoogimpedant is. Deze pin is dus geen uitgang, maar een ingang, die niet gebruikt wordt.
Omdat de fuses van de enable van de tristate-inverter van pin 22 allemaal zijn opgeblazen is deze altijd hoog. Pin 22 is daarom een uitgang. Niet zichtbaar is dat de twee fuses van de macrocel op combinatorisch en niet-inverterend zijn ingesteld. De lijnen met de fuses 572 tot en met 880 zijn intact en zijn dus altijd laag. De andere twee producttermen zijn verbonden via de pinnen 2, 3 en 4 met respectievelijk de signalen `a`, `b` en `c`. De uitgang van pin 22 is gelijk aan `a'b + ac`.

PROM

Bij de PAL is het AND-vlak programmeerbaar en het OR-vlak ligt vast. Bij een PROM, *Programmable read only memory*, is het OR-vlak programmeerbaar en ligt het AND-vlak vast. In figuur 11.20 is deze constructie nogmaals voor de schakeling van tabel 11.2 getekend.

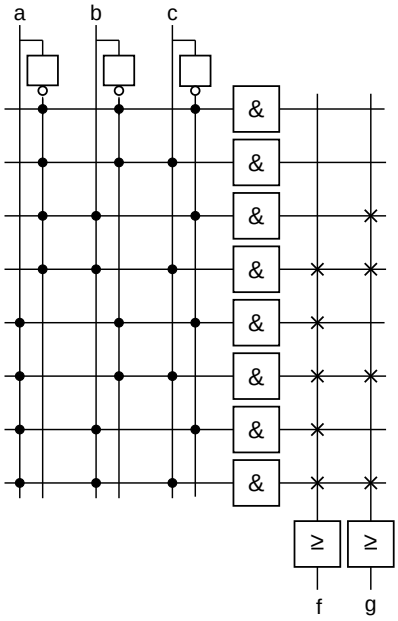


Figuur 11.20 : **De realisatie van tabel 11.2 met een PROM-structuur.** Het AND-vlak is niet programmeerbaar. Het OR-vlak is programmeerbaar.

Bij een PROM zijn normaal gesproken alle producttermen beschikbaar. De waarheidstabel 11.1 kan dan rechtstreeks, zonder optimalisatie, met een PROM gerealiseerd worden. Figuur 11.21 laat zien dat het rechter deel uit deze waarheidstabel overeenkomt met het OR-vlak van de PROM.

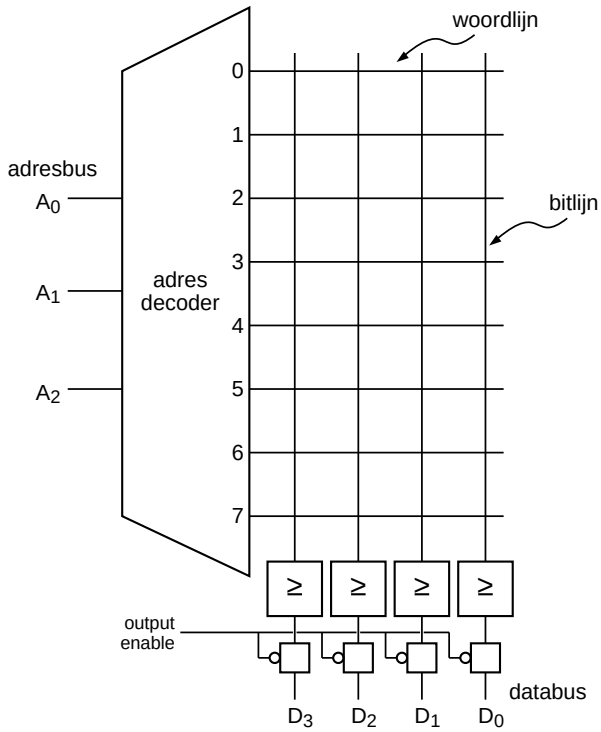
a	b	c	f	g
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

de waarheidstabel



Figuur 11.21 : **De realisatie van tabel 11.1 met een PROM-structuur.** Het AND-vlak is niet programmeerbaar en bevat alle acht mogelijke producttermen voor een combinatie met drie ingangen. Het OR-vlak is programmeerbaar en komt overeen met het rechter deel van de waarheidstabel.

Het AND-vlak van een PROM is een decoder en wordt, omdat een PROM meestal als geheugen gebruikt wordt, de adresdecoder genoemd. De ingangen zijn dan de adreslijnen of vormen de adresbus en de uitgang zijn dan de datalijnen of databus. Een horizontale producttermlijn wordt bij geheugens woordlijn, *word line*, genoemd en de verticale datalijn wordt bitlijn, *bit line*, genoemd.
Omdat de uitgangen vaak worden aangesloten op een databus hebben de datalijnen meestal een tristatebuffer met een output-enable-signaal.



Figuur 11.22 : **De PROM als geheugenbouwsteen.** Deze PROM heeft drie adreslijnen en vier datalijnen. De adresdecoder is het AND-vlak en ligt vast. Het OR-vlak is programmeerbaar en de uitgangen hebben een tristatebuffer.

Het aantal geheugen bits hangt af van het aantal adresbits n en het aantal databits m en is gelijk aan $m \cdot 2^n$ bits. Het PROM uit figuur 11.22 is 32 bits groot. Een 27C64 met 13 adreslijnen en 8 datalijnen heeft een omvang van 65536 bits.
Moderne geheugens zijn gebaseerd op EEPROM- of flashtechnologie en hebben soms alleen een seriële aansluiting. De op RAM-gebaseerde FPGA's gebruiken een seriële EEPROM om de FPGA bij het opstarten mee te configureren.

Overeenkomsten en verschillen PLA, PAL en PROM

De PLA, PAL en PROM zijn programmeerbare bouwstenen, die veel overeenkomsten hebben. Alledrie hebben ze een programmeerbaar AND- en/of OR-vlak. De PLA en de PAL worden tot (S)PLD's gerekend, omdat deze bouwstenen gebruikt worden om digitale schakelingen te realiseren. Hoewel met een PROM ook schakelingen gemaakt kunnen worden, wordt een PROM meestal als geheugen gebruikt en dus niet tot de (S)PLD's gerekend.

Vanwege de overeenkomsten in de structuur is de PROM hier wel samen met de PLA en de PAL besproken. Tabel 11.3 geeft een overzicht van de verschillende combinaties met een programmeerbaar AND- en OR-vlak.

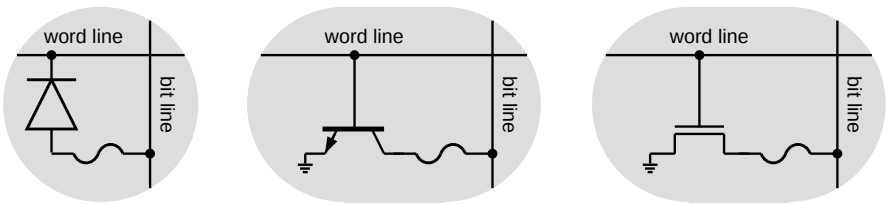
Tabel 11.3 : Overzicht mogelijkheden met AND- en OR-vlak.

	AND-vlak	OR-vlak
PLA	programmeerbaar	programmeerbaar
PAL	programmeerbaar	vast
PROM	vast	programmeerbaar
ROM	vast	vast

Aan het overzicht is de ROM, *read only memory* toegevoegd. In feite is dat een PROM, die in de fabriek is *voorgeprogrammeerd*. Het OR-vlak van de ROM ligt daarmee ook vast. Bij de fabricage van de ROM worden de verbindingen met behulp van het metaalmasker aangebracht. Men noemt dit ook wel een *mask PROM* of maskergeprogrammeerd ROM.

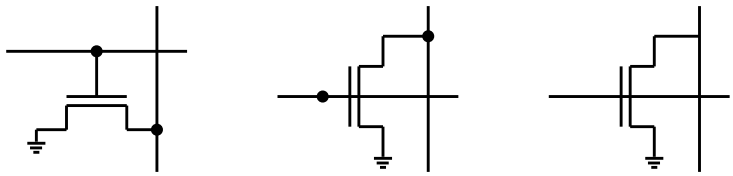
Maskergeprogrammeerd ROM

In figuur 11.14 is de fuse gecombineerd met een diode. Er bestaan ook PROM's met een bipolaire transistor of met een MOS-transistor in plaats van een diode. Figuur 11.23 toont de programmeerbits uit drie verschillende implementaties. Bij de fuse met de diode is de woordlijn actief laag. De woordlijnen bij de fuses met de transistoren zijn daarentegen actief hoog. Als de fuse intact is en de basis van de bipolaire transistor of de gate van de NMOS-transistor hoog is, geleiden de transistoren en wordt de bitlijn omlaag getrokken.



Figuur 11.23 : Verschillende configuraties van een bit met de fuse-technologie. De woordlijn met de diode is actief laag en de woordlijnen bij de bipolaire transistor en de NMOS-transistor zijn actief hoog.

Bij een *mask* PROM of maskergeprogrammeerd ROM is een fuse overbodig. Een intacte fuse komt overeen met een transistor en bij een opgeblazen fuse wordt de hele verbinding met de transistor weggelaten.
In figuur 11.24 staat het programmeerbit van een ROM met twee alternatieve notaties. Deze notaties zijn meer beknopt en passen beter bij de fysieke realisatie van de transistor. Omdat deze notaties minder duidelijk zijn, worden ze in dit boek niet gebruikt. Bij de realisatie van een maskergeprogrammeerd ROM worden alle



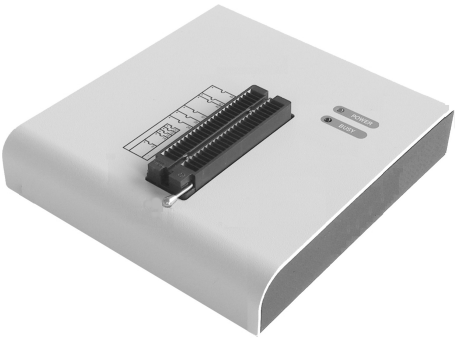
Figuur 11.24 : **De NMOS-transistor als programmeerbit bij een ROM.** Links staat de notatie die in dit boek wordt gebruikt. In het midden en rechts staan twee alternatieve notaties. De transistor is 90^0 gedraaid en de woordlijn ligt over de gate heen.

transistoren aangebracht. Dus ook de transistoren van de bits die niet geprogrammeerd zijn. Daarmee wordt de regelmaat van de compacte array met transistoren niet onderbroken. Bij de niet-geprogrammeerde transistoren ontbreekt alleen de verbinding tussen de woordlijn en de gate van de transistor.

ZIFF staat voor *zero insertion force and friction*. Dat is een IC-voet waarin het IC geplaatst en verwijderd kan worden zonder dat de pinnen beschadigd worden.

Een hex-bestand is een bestand met de te programmeren gegevens in het Intel HEX-formaat. Dit bestandstype wordt vooral bij geheugens en microcontrollers gebruikt.

Een JEDEC-bestand is een bestand met de te programmeren gegevens in JEDEC-formaat. JEDEC staat voor *Joint Electron Device Engineering Council* en is een organisatie die standaardisatie bevorderd. Tegenwoordig luidt de officiële naam *JEDEC Solid State Technology Association*. JEDEC-bestanden worden met name bij PLD's gebruikt.



Figuur 11.25 : **Een programmer voor programmeerbare bouwstenen.** In de ZIFF-socket wordt de PROM, PAL of microcontroller geplaatst. Met een programmeerprogramma wordt het hex- of het JEDEC-bestand met de fuse-map geopend en wordt de component geprogrammeerd.

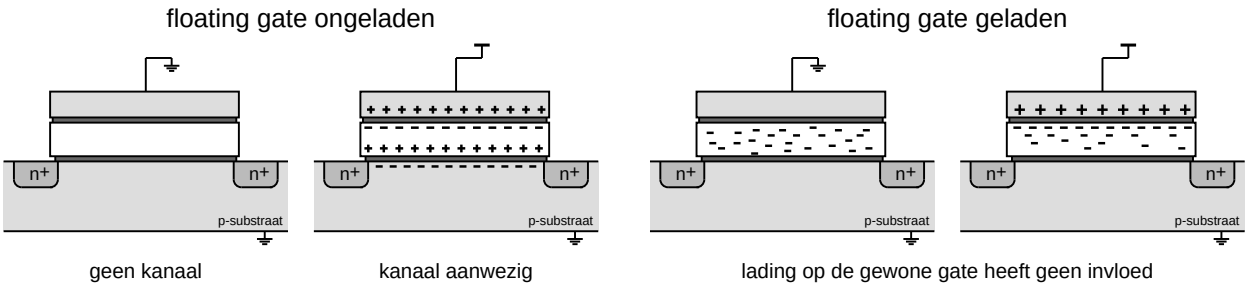
11.3 EPROM-, EEPROM- en flashtechnologie

In paragraaf 11.2 is de algemeen opbouw van de PLA, de PAL, en de PROM besproken. Daarbij is uitgegaan van de fuse-technologie, die op bladzijde 275 is geïntroduceerd. Aanvankelijk was dat de technologie voor al deze bouwstenen. Later zijn daar achtereenvolgens de EPROM-, de EEPROM- en de flashtechnologie bijgekomen. Het gemeenschappelijk kenmerk van deze technieken is dat ze alle drie gebaseerd zijn op de zogenoemde *floating gate*-transistor.

De floating-gate-transistor

Een floating-gate-transistor is een MOS-transistor met een extra gate tussen de normale gate en het substraat. De floating gate, of zwevende gate, heeft geen aansluitingen en is volledig omsloten door een isolator. Ondanks dat er geen aansluiting is, is het toch mogelijk om lading op deze gate aan te brengen.
In figuur 11.26 is een NMOS-transistor met een floating gate getekend. Als de floating gate ongeladen is en de normale gate is hoog, induceert dit direct onder de normale gate een negatieve lading in de floating gate. De onderkant van de floating gate wordt daardoor positief. Deze lading zorgt ervoor dat er een kanaal

in het substraat ontstaat. Als de normale gate laag is, is er geen ladingsverschuiving in de floating gate en is er ook geen kanaal. De floating-gate-transistor werkt, als de floating gate ongeladen is, als een normale NMOS-transistor.



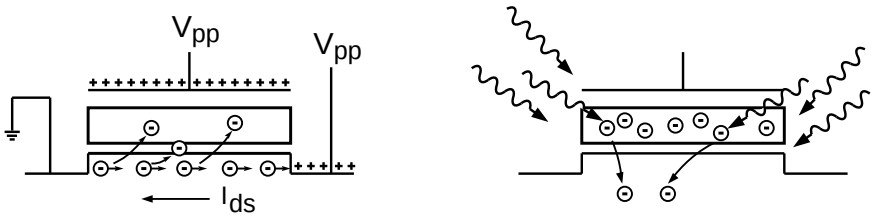
Figuur 11.26 : **De floating-gate-transistor.** Als de floating gate ongeladen is, werkt de transistor als een gewone MOS-transistor. Wanneer er dan lading op de gate wordt aangebracht, ontstaat er een kanaal. Als de floating gate geladen is, verhindert deze negatieve lading dat er een kanaal ontstaat.

Als de floating gate negatief geladen wordt, schermt deze negatieve lading de invloed van de lading op de normale gate af. Als de normale gate laag is, versterkt de negatieve lading van de floating gate hooguit het sperren van de NMOS-transistor. Wanneer de normale gate hoog is, wordt de bovenkant van de floating gate negatief. Maar als er voldoende negatieve ladingen zijn, zal de onderkant van de floating gate nog steeds negatief zijn en blijft de transistor gesperd.

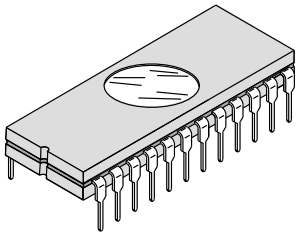
EPROM

In de jaren tachtig en negentig is de fuse-technology geleidelijk vervangen door EPROM, *erasable programmable read only memory*. Een EPROM gebruikt voor het opslaan van een bit een floating-gate-transistor in plaats van een transistor en een fuse. Als de floating gate ongeladen is, is de verbinding intact en is de bit geprogrammeerd. Een negatief geladen floating gate komt overeen met een opgeblazen fuse. Er is geen verbinding tussen de woordlijn en de bitlijn. De bit is dan ongeprogrammeerd.

De EPROM gebruikt een FAMOS, *floating gate avalanche injection MOS*. Deze transistor wordt geprogrammeerd met een techniek die gebaseerd is op *hot electrons*. Dit is een fenomeen waarbij elektronen voldoende kinetische energie krijgen om een potentiaalbarrière te overbruggen.

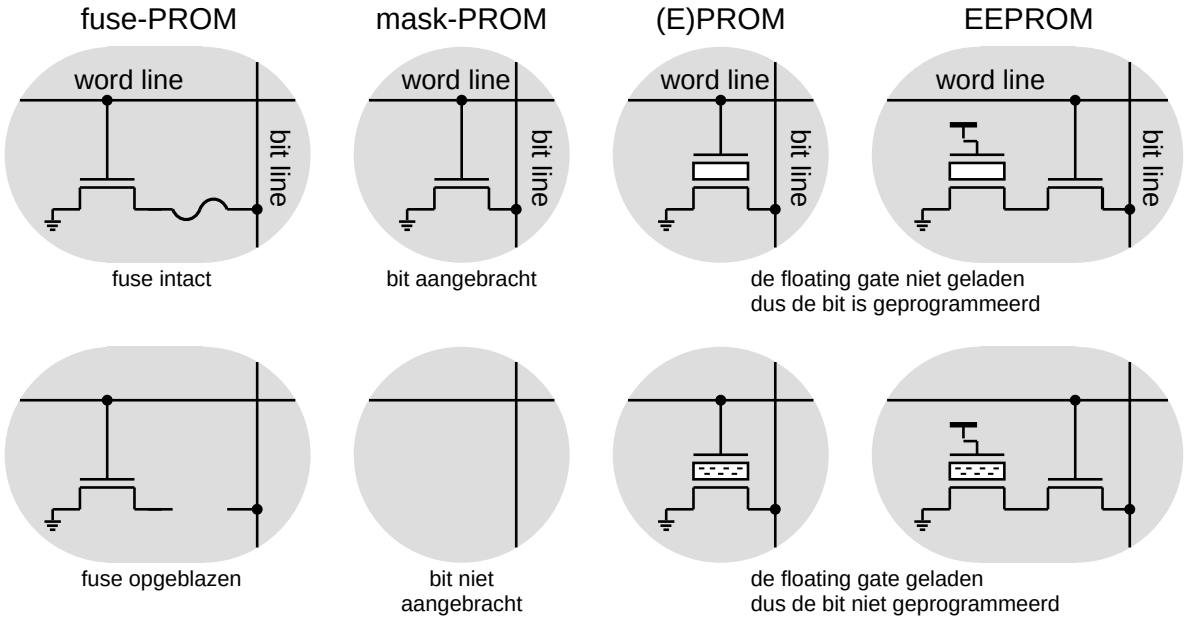


Figuur 11.27 : **Het programmeren en het wissen van een EPROM.** Om de EPROM te programmeren wordt de programmeerspanning V_{pp} aangesloten aan de drain en de gate van de transistor. De kinetische energie van de elektronen is zó hoog dat ze onder invloed van de positieve gate ook in de floating gate terechtkomen.
Voor het wissen wordt uv-licht gebruikt. De energie van de fotonen is zó groot dat deze in staat zijn de elektronen uit de floating gate te verwijderen.



Figuur 11.28 : **Een wisbare EPROM.** De uv-wisbare IC's zijn herkenbaar aan het venster.

Om de EPROM te programmeren is een hoge programmeerspanning V_{pp} nodig. Afhankelijk van de fabricagetechniek is dat bijvoorbeeld 12, 15 of 21 V. Deze programmeerspanning zorgt voor een kanaal met veel ladingdragers en grote drain-source-spanning. De elektronen krijgen een hoge snelheid en kunnen door de hoge gatespanning in de floating gate terechtkomen.
Een belangrijke eigenschap van de EPROM is dat deze ook wisbaar, *erasable*, is met ultraviolet licht. De constructie van de geïntegreerde schakeling en van de transistoren is zodanig dat er licht bij de floating gate kan komen. Zichtbaar licht heeft geen invloed, maar voor ultraviolet licht is de energie van de fotonen voldoende groot om de elektronen uit de floating gate te stoten. De uv-wisbare EPROM's hebben speciaal hiervoor een venster in de behuizing. Om de EPROM's te wissen wordt een speciale uv-wisser gebruikt. Dit is een kastje met een uv-lamp waarin de EPROM tien à twintig minuten moet liggen om gewist te worden.
EPROM's zijn ook beschikbaar in een behuizing zonder venster. Ze zijn dan niet wisbaar en worden OTP, *one time programmable*, genoemd. Deze OTP-PROM's zijn niet handig bij het ontwerp, maar zijn wel goedkoper en daarom zeer geschikt voor de productie.

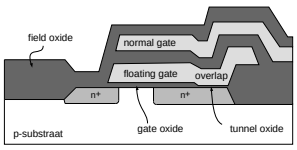


Figuur 11.29 : **Verschillende technieken om een bit vast te leggen.** Achtereenvolgens zijn dat een fuse-PROM-, mask-PROM-, EPROM en EEPROM. Bij de bovenste rij is de verbinding tussen de woordlijn en de bitlijn intact. De bit is *geprogrammeerd*. Bij de onderste rij is er geen verbinding tussen de woordlijn en de bitlijn. De bit is *ongeprogrammeerd*.

In figuur 11.29 zijn de intacte en *opgeblazen* programmeerbits voor de diverse PROM-implementaties getekend. Als de verbinding intact is, zegt men dat de bit *geprogrammeerd* is. Als de verbinding niet intact is, zegt men dat de bit *niet geprogrammeerd* of *ongeprogrammeerd* is.

De woorden programmeren en wissen hebben bij een EPROM en bij EEPROM twee tegenovergestelde betekenissen. Dit is verwarrend. Bij een nieuwe EPROM of EEPROM zijn de floating gates ongeladen. Alle verbindingen zijn dus aanwezig en alle bits zijn geprogrammeerd. Bij het programmeren van de EPROM krijgen

sommige floating gates een lading. Bij deze bits is er geen verbinding meer en zijn ze dus juist ongeprogrammeerd.
Als je een EPROM, of een EEPROM of flash programmeert, verwijder je een deel van de verbindingen. Als je een EPROM, of een EEPROM of flash wist, breng je juist alle verbindingen weer terug.

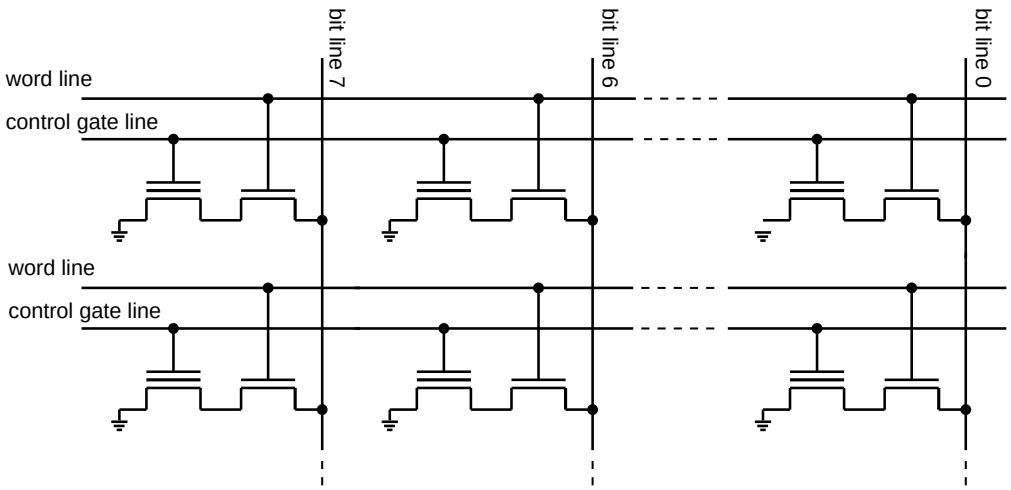


Figuur 11.30 : De EEPROM floating-gate-transistor.

De transistor bij EEPROM is een FLOTOX, *floating-gate tunneling oxide*. Deze transistor wordt geprogrammeerd en gewist met behulp van Fowler-Nordheim tunneling. Dit is een quantummechanisch tunneleffect dat ontstaat als elektronen zich in een sterk elektrisch veld bevinden.

EEPROM

Een variant van de EPROM is de EEPROM, *electrical erasable programmable read only memory*, die elektrisch gewist kan worden.
De floating-gate-transistoren van een EEPROM zijn op een ander fysisch mechanisme gebaseerd en hebben een andere interne bouw. Zowel bij het laden als bij het ontladen van de floating gate wordt gebruik gemaakt van Fowler-Nordheim-tunneling. De bouw van de transistor is ook anders: de floating gate en de normale gate steken over de drain van de transistor heen. Een nadeel van deze tunneltransistor is dat, doordat de drempelspanning van een ongeprogrammeerde floating-gate-transistor negatief is, er ook een stroom kan lopen als de normale gate laag is. Dit is de reden dat er in figuur 11.29 twee transistoren zijn gebruikt: één gewone MOS-transistor om te kunnen selecteren en één floating-gate-transistor waarvan de gewone gate de voedingsspanning VCC of de speciale programmeerspanning is.



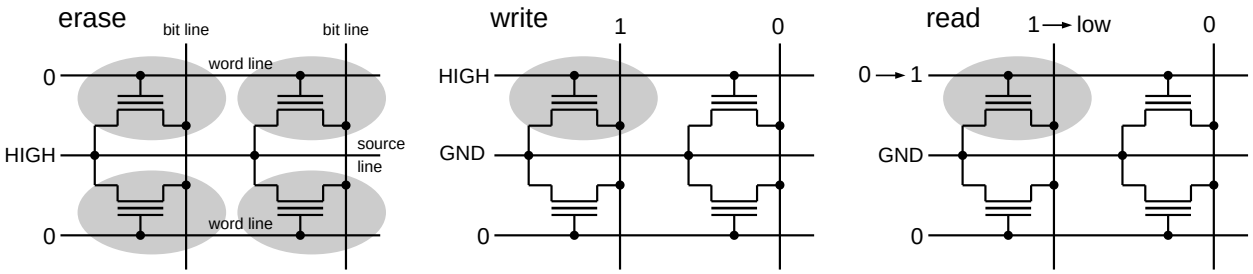
Figuur 11.31 : Twee bytes uit een EEPROM-architectuur. De controllijnen zijn verbonden met VCC of met de hoge programmeerspanning.

Figuur 11.31 toont een deel van het geheugen van een EEPROM. De gates van de floating-gate-transistoren hebben per byte een gemeenschappelijke programmeerlijn: de *control gate line*. Om de floating gate te laden wordt een hoge programmeerspanning op de floating gate gezet en tegelijkertijd de betreffende woordlijn hoog gemaakt en de betreffende bitlijn laag gemaakt. De floating gate wordt ontladen door de hoge programmeerspanning op de bitlijn te zetten en tegelijkertijd de woordlijn hoog en de gate van de floating gate laag te maken. Bij de normale werking is de floating gate aangesloten op de voedingsspanning.
Omdat acht floating-gate-transistoren een gemeenschappelijke programmeerlijn hebben, is de EEPROM meestal per byte te programmeren en te wissen.

Flashtechnologie

De transistor bij flash is de ETOX, *EPROM tunnel oxide*. Deze transistor wordt geprogrammeerd met *hot electrons* en gewist met Fowler-Nordheim tunneling. De transistor lijkt op een FAMOS, maar is met de techniek van de FLOTOX ook elektrisch wisbaar.

De flashtechnologie is gebaseerd op een verbeterde versie van de FAMOS-transistor, die net als de FAMOS geprogrammeeerd wordt met een hoge gatespanning. De sterk positief geladen gate trekt dan elektronen uit de source naar de floating gate. Deze verbeterde transistor is ook elektrisch wisbaar. Daarvoor is een negatieve gatespanning nodig. De negatief geladen gate duwt dan de elektronen als het ware uit de floating gate. Een negatieve gatespanning wordt verkregen door de source sterk positief te maken ten opzichte van de floating gate.
Om iedere individuele floating gate te ontladen, is per transistor een apart ontlaadcircuit nodig. Bij de flashtechnologie is daar niet voor gekozen, maar is er een ontlaadcircuit voor alle transistoren. Het gevolg is dat bij een flash-IC alle transistoren gelijktijdig gewist worden of dat ze in grote blokken tegelijkertijd gewist worden.



Figuur 11.32 : Vier bits uit een flash-architectuur. Links worden alle floating gates van de transistoren gelijktijdig gewist door de potentiaal van de sourcelijn hoog te maken. In het midden wordt van één transistor de floating gate geladen door de betreffende woordlijn een hoge spanning te geven en de bitlijn gewoon hoog te maken. Rechts staat de situatie voor als de flash wordt uitgelezen. Als de woordlijn hoog is en de bit geprogrammeerd is, dus als de floating gate ontladen is, wordt de betreffende bitlijn omlaag getrokken.

Figuur 11.32 toont vier bits uit een flash-architectuur. Een hoge programmeerpotentiaal op de sourcelijn wist alle floating gates van de transistoren die op deze lijn zijn aangesloten. De transistoren kunnen wel afzonderlijk worden geladen. De woordlijn moet dan de hoge programmeerspanning krijgen en de bitlijn moet gewoon hoog zijn. Het lezen gaat op dezelfde manier als bij de EPROM's. Als de woordlijn hoog is en als de floating gate van de transistor ongeladen is, wordt de bitlijn omlaag getrokken.

De flashtechnologie heeft net als de EEPROM de mogelijkheid om de floating-gate-transistoren elektrisch te ontladen, maar gebruikt als transistor een variant op die van de EPROM-technologie. De term *flash* geeft aan dat het geheugen in grote blokken tegelijk wordt gewist. Hiermee onderscheid een flashgeheugen zich van een EEPROM, die meestal per byte gewist wordt.

Bij eenvoudige en complexe PLD's worden dezelfde technieken gebruikt als bij EPROM, EEPROM en flashgeheugens. Alleen is niet het OR-vlak programmeerbaar, maar het AND-vlak. Deze bouwstenen noemt men soms EPLD en EEPLD en flash-PLD, analoog aan respectievelijk EPROM, EEPROM en flash-PROM. Cypress levert bijvoorbeeld een PALC22V10, die op EPROM is gebaseerd, en een PALCE22V10, die op EEPROM is gebaseerd. De PALC22V10 was leverbaar met en zonder venster, dus als een uv-wisbare of als een eenmalig programmeerbare bouwsteen.

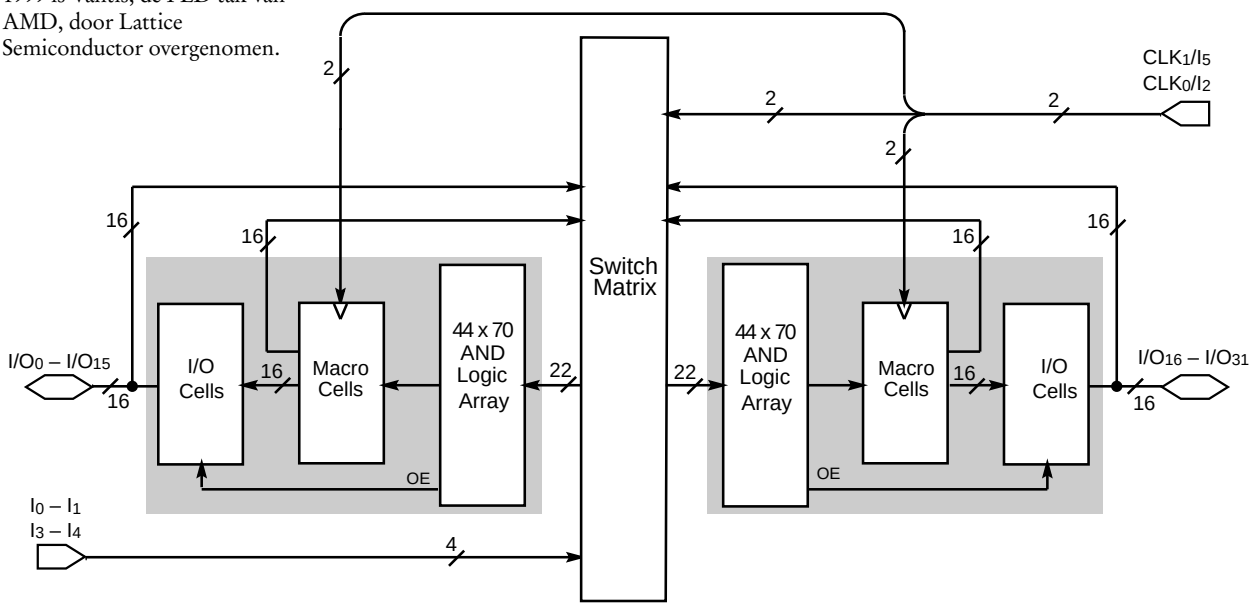
Een verschil met de PROM's is dat de meeste SPLD's en CPLD's programmeerbare macrocellen hebben die ook configureerbaar zijn. Een PAL22V10 heeft naast een AND-vlak met 5280 programmeerbare bits nog eens 528 bits voor de reset- en enable-lijnen en 20 bits voor de tien macrocellen.

11.4 CPLD

De PAL16V8 en PAL22V10 waren zeer populaire programmeerbare bouwstenen. Hoewel er ook grotere varianten — zoals een 26V12 — op de markt zijn gebracht, zijn deze nooit erg populair geworden. Een groter programmeerbaar AND-vlak impliceert dat de wired-AND's een grotere capaciteit hebben en dus trager zijn. Bovendien waren de afzetmogelijkheden kleiner, zodat de prijs ook hoger was. Ontwerpers gebruikten liever twee 22V10's dan één 26V12.

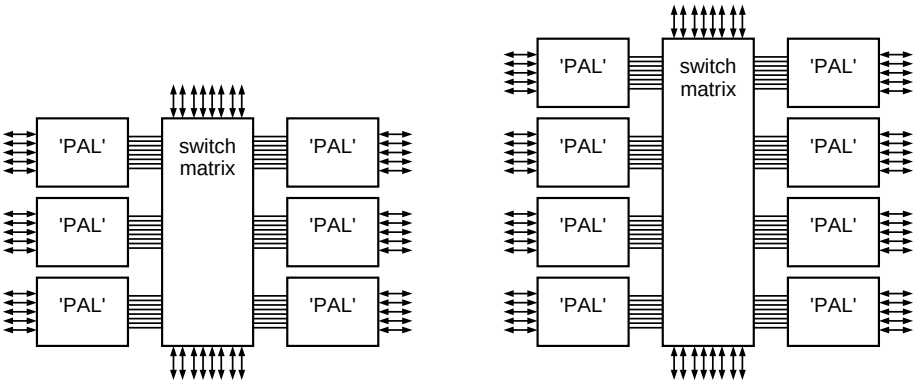
Altera en AMD waren de eerste PLD-fabrikanten, die met een CPLD op de markt kwamen. Voor de Altera was dat de MAX5000 en voor de AMD was dat de MACH1. In 1999 is Vantis, de PLD-tak van AMD, door Lattice Semiconductor overgenomen.

Als alternatief kwamen rond 1990 programmeerbare bouwstenen op de markt met twee of meer PAL-achtige structuren in één behuizing. Deze worden CPLD's genoemd. CPLD staat voor *complex programmable logical device* en is altijd gebaseerd op een EEPROM- of flashtechnologie.



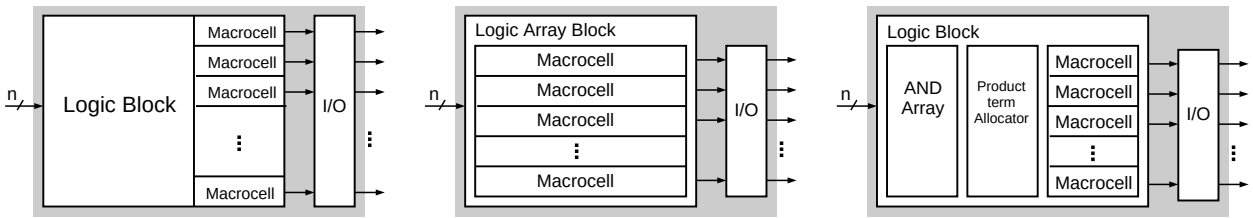
Figuur 11.33 : Een CPLD met twee PAL-achtige structuren. De grijs gekleurde delen geven de twee PAL-structuren aan waaruit deze CPLD is opgebouwd. Elke PAL-structuur bestaat uit een AND-vlak, een aantal macrocellen en in- en uitgangen. In het midden bevindt zich de *switch matrix*, die de aansluitingen en de PAL's met elkaar kan verbinden.

Figuur 11.33 toont een CPLD met twee PAL-achtige structuren. Dit is ruwweg de structuur van de MACH 110, één van de eerste CPLD's van de firma AMD. De *switch matrix* of schakelmatrix verbindt de zes ingangen en de twee PAL's met elkaar. Iedere PAL heeft 32 in- en/of uitgangen. Deze aansluitingen hebben net als de 22V10 een I/O-cell met een tristatebuffer en een macrocel met een selecteerbare flipflop. Het AND-vlak bevat ook een zogenoemde allocator en voor iedere uitgang een OR-poort.



Figuur 11.34 : Een CPLD met zes en een CPLD met acht PAL-achtige structuren.

Figuur 11.34 toont twee CPLD's met zes en acht PAL-achtige structuren. Alle fabrikanten gebruiken andere namen voor de schakelmatrix en voor de PAL-achtige blokken. De schakelmatrix heet bijvoorbeeld switch matrix (AMD), *programmable interconnect array* (Altera), *programmable interconnect matrix* (Cypress) of *zero-power interconnect array* (Xilinx). Voor de PAL-achtige structuur of onderdelen van deze structuur worden ook verschillende namen gebruikt.



Figuur 11.35 : Drie uitvoeringen van de PAL-achtige structuur bij een CPLD. Links staat een structuur die Xilinx gebruikt, in het midden staat een LAB van Altera en rechts staat de opbouw van een Cypress CPLD.

Figuur 11.35 toont de logische blokken van drie verschillende CPLD's. Soms bevat het logische blok alleen het AND- en OR-vlak en is een macrocel niet meer dan een instelbare flipflop. In andere gevallen bestaat het logisch blok uit macrocellen en bevat de macrocel ook producttermen en een OR-poort. Bij enkele fabrikanten bevat het logische blok de hele PAL-structuur inclusief de macrocellen met de instelbare flipfloppen.

De belangrijkste kenmerken van CPLD's zijn:

- CPLD's zijn gebaseerd op EEPROM- en flashtechnologie.
- CPLD's zijn niet-vluchtig.
- Het tijdsgedrag van de CPLD is deterministisch, of anders gezegd de vertragingen zijn voorspelbaar. De functionaliteit ligt vast in de AND-vlakken van de logische blokken. Iedere productterm heeft dezelfde tijdvertraging. De vertraging van de OR-poort ligt, net als de vertraging langs de signaalpaden door de macrocellen, ook vast.

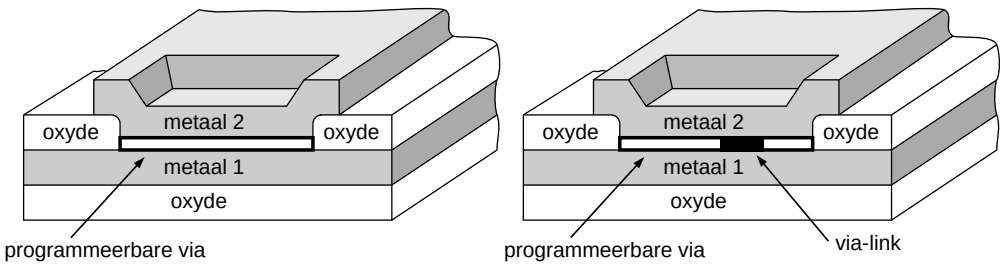
- CPLD's hebben veel logica en relatief weinig flipfloppen.
- CPLD's hebben geen speciale connecties voor carry-signalen en hebben geen speciale functieblokken, zoals RAM en multipliers.
- CPLD's zijn geschikt voor middelgrote ontwerpen met veel logica, zoals toestandsmachines en adresdecoders.
- CPLD's zijn minder geschikt voor grote ontwerpen en voor ontwerpen met veel flipfloppen, zoals signaalverwerking, digitale filters, buffers, fifo's, lifo's, en schuifregisters.

Het alternatief voor een CPLD is een FPGA. FPGA's zijn over het algemeen veel krachtiger, maar meestal gebaseerd op SRAM en zijn dus vluchtig. Daarom is het niet-vluchtig zijn van de op EEPROM of op flash gebaseerde CPLD een belangrijk voordeel.

11.5 Antifuse-technologie

FPGA's zijn leverbaar met antifuse-, SRAM- en flashtechnologie. De op SRAM-gebaseerde FPGA's van Xilinx en Altera zijn het meest populair. De antifuse-technologie is minder gangbaar, maar voor sommige toepassingen heel interessant. Sinds 2000 zijn er ook FPGA's van Actel en Lattice, die de flashtechnologie gebruiken.

De antifuse-technologie is ontwikkeld en wordt toegepast door Actel en Quicklogic. Antifuse is het tegengestelde van de fuse-technologie. Bij de fuse-technologie worden bij het programmeren de verbindingen verbroken. Bij antifuse worden bij het programmeren de verbindingen juist aangebracht. De te programmeren verbinding tussen twee geleidende lagen wordt een via genoemd. Door een hoge spanning over een programmeerbare via aan te brengen, ontstaat er een permanente doorslag. Na het verwijderen van de hoge spanning blijft de doorslag of de zogenoemde *via-link* intact.



Figuur 11.36 : **De antifuse-technologie.** Links staat een ongeprogrammeerde verbinding of via. Het oxide van deze via is nog intact. Rechts staat een geprogrammeerde verbinding. Er is een verbinding of een via-link in het oxide tussen metaal 1 en metaal 2.

FPGA's op basis van antifuse-technologie zijn eenmalig programmeerbaar en dus niet herprogrammeerbaar. Ze zijn niet vluchtig en er is geen serieel EEPROM bij het opstarten nodig om de FPGA te configureren. Antifuse-FPGA's hebben een hoge dichtheid en verbruiken weinig vermogen. De geprogrammeerde fuses zijn bijna niet te onderscheiden van ongeprogrammeerde fuses. Daardoor kan het ontwerp bijna niet uit deze FPGA's worden uitgelezen.

Een belangrijk nadeel van de antifuse-technologie is de complexiteit van het productieproces. Dit is een belangrijke drempel bij de verbetering en de vernieuwing van deze FPGA's. Een bezwaar voor de ontwerper is dat deze FPGA's niet herprogrammeerbaar zijn en dat ze geprogrammeerd moeten worden met een speciale programmer.

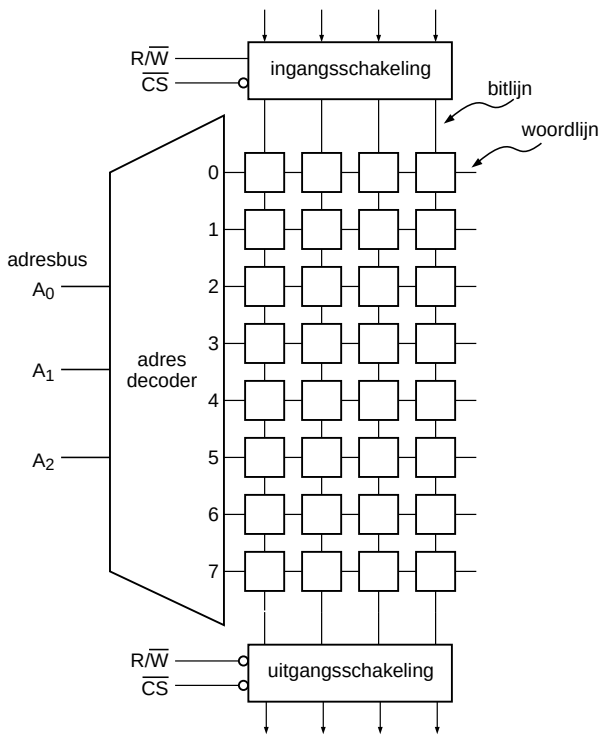
11.6 RAM-technologie

De term RAM, *random access memory*, wordt gebruikt voor geheugens waaruit even gemakkelijk geschreven als gelezen kan worden. Dit onderscheidt het RAM van ROM-geheugens waaruit alleen gelezen kan worden en de (E)EPROM- en flashgeheugens waaruit de informatie veel eenvoudiger en sneller te lezen is dan dat er gegevens naar toe kunnen worden geschreven.

Een betere term voor de RAM-technologie is RWM, *read write memory*. Het zijn geheugens waar ongeveer even snel uit gelezen als naar geschreven kan worden. Bij de introductie van de eerste RAM-geheugens was het grote verschil met de toen bestaande geheugens, zoals bijvoorbeeld tape-recorders, dat het RAM willekeurig toegankelijk is. Bij een tape-recorder moet de tape gelezen worden tot de plaats waar iets gelezen of geschreven moet worden. Bij een RAM kan men direct naar een willekeurige adres springen.

De belangrijkste toepassing van RAM is het gebruik als werkgeheugen bij computers. Omdat de snelheid en de omvang van dit geheugen essentieel zijn voor de gigantische ontwikkeling die computers de afgelopen dertig jaar hebben doorgemaakt, is de ontwikkeling, de verbetering en de schaalverkleining van RAM-geheugens enorm. Van deze technologische groei profiteren de op SRAM-gebaseerde FPGA's enorm.

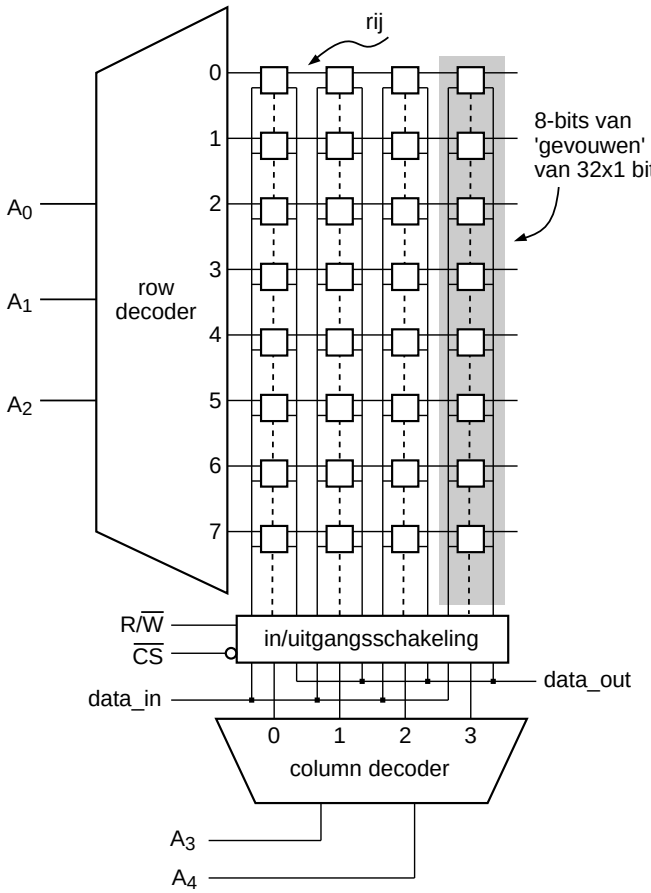
Het nadeel van RAM is dat de opgeslagen gegevens verloren gaan als de voedingsspanning wegvalt. Daarom hebben de op SRAM-gebaseerde FPGA's een extern serieel EEPROM met de configuratie van de FPGA. Bij het opstarten wordt de seriële EEPROM automatisch uitgelezen en wordt de FPGA geconfigureerd.



Figuur 11.37 : **De structuur van een RAM.** Er zijn drie adreslijnen en vier datalijnen. Er zijn acht woordlijnen en de omvang is 32 bits.

Opbouw RAM

De algemene opbouw van een RAM staat in figuur 11.37. Deze lijkt op de opbouw van de (P)ROM uit figuur 11.22. De RAM heeft ook een adresdecoder en een array met geheugenelementen. De RAM heeft naast een uitgangsschakeling ook een schakeling voor de ingangen. De RAM heeft naast een 3-bits adresingang en een 4-bits dataingang twee stuursignalen: $\overline{\texttt{CS}}$ en $\texttt{R}/\overline{\texttt{W}}$.
Het signaal $\texttt{R}/\overline{\texttt{W}}$, *read write*, regelt of er uit het RAM gelezen wordt of dat er naar het RAM geschreven wordt. Omdat een RAM meestal in een busstructuur met andere IC's gebruikt wordt, bestaat de mogelijkheid om een RAM te selecteren of te deselecteren door het signaal $\overline{\texttt{CS}}$, *chip select*, respectievelijk laag of hoog te maken.



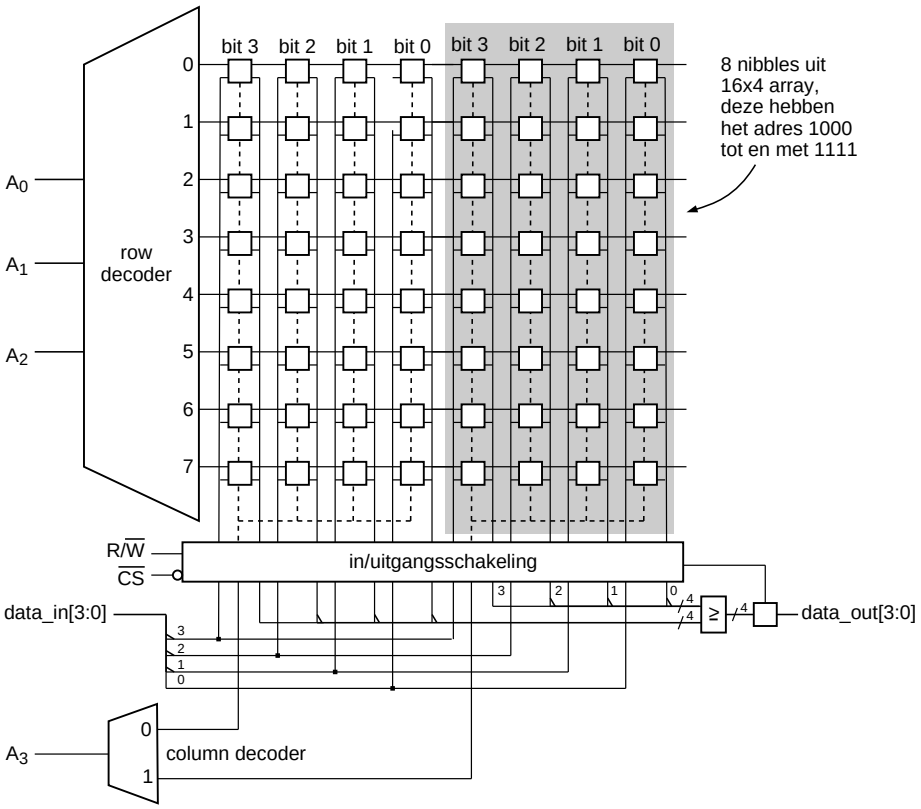
Figuur 11.38 : **Een 32x1 RAM met 3-bits rij- en een 2-bits kolomdecoder.** Er zijn drie adressignalen voor de rijen en twee voor de kolommen. Er zijn acht rijen en vier kolommen. De omvang van de array is 32 bits.

SRAM staat *static RAM* en DRAM staat voor *dynamic RAM*. In deze paragraaf wordt vanaf pagina 294 het DRAM besproken en vanaf pagina 295 het SRAM.

Figuur 11.38 is een variant van figuur 11.37 en lijkt wat meer op een echte RAM. De ingangs- en uitgangsschakeling zijn samengevoegd tot een circuit. Ieder geheugenelement heeft een ingang en uitgang voor de data. Bij een SRAM en bij een DRAM is dit anders georganiseerd. In de configuratie van figuur 11.38 kan het geheugenelement een D-flipflop zijn. Ieder geheugenelement, dus iedere bit, is afzonderlijk te schrijven en te lezen. Dit kan door de betreffende rij en de betreffende kolom te selecteren. De kolomselectie is gevisualiseerd met streeplijnen.

Het RAM uit figuur 11.38 is een 32x1 RAM. Dat wil zeggen dat er 32 adressen zijn en dat de data-signalen 1-bit breed zijn. Bij grote geheugens, zoals een SRAM van 2Mx8, is een adresdecoder met 2 miljoen adressen niet handig. Met de structuur van figuur 11.37 zou dat een array van 2 miljoen bij 8 geven. Om de structuur van de array meer vierkant te maken, wordt deze in stukken geknipt en naast elkaar gezet. In figuur 11.38 is de array in vier stukken van acht geknipt. Ieder stuk is een kolom uit de array, dat geselecteerd kan worden met behulp van de kolomdecoder.

Dit voorbeeld heeft drie adressignalen voor de rijen en dus acht rijen en het heeft twee adressignalen voor de kolommen en dus vier kolommen. Met het gegeven dat de databreedte 1 is, is de grootte van de array 32. In het algemeen geldt dat de grootte van een RAM weergegeven wordt met $2^{i+j} \times k$, waarbij i het aantal adresingangen voor de rijen, j het aantal adresingangen voor de kolommen en k het aantal databits is. In dit geval spreken van 32x1 RAM.



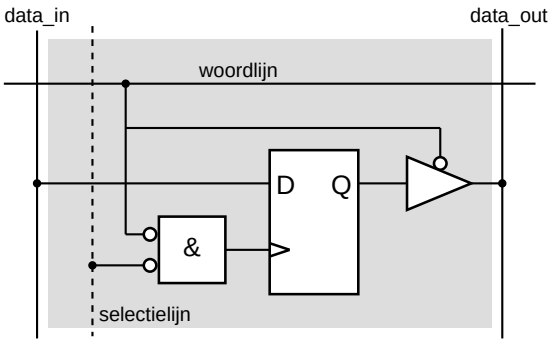
Figuur 11.39 : Een 16x4 RAM met 3 adressignalen voor de rijen en 1 adressignaal voor de kolommen. Er zijn acht rijen en twee kolommen met steeds vier databits. De omvang van de array is 64 bits. De nibbles bij het grijze vlak hebben adres 1000 tot en met 1111. De andere nibbles hebben adres 0000 tot en met 0111.

In figuur 11.39 staat een 16x4 RAM. Dit array heeft acht rijen en twee kolommen met ieder vier databits. Er zijn drie adressignalen voor de rijen en één voor de kolommen.

RAM-technologie met behulp van D-flipfloppen

Figuur 11.38 en 11.39 geven in grote lijnen de architectuur van een RAM-geheugen. De precieze invulling van de geheugenelementen waaruit deze RAM's zijn opgebouwd is in het midden gelaten. Afhankelijk van het type geheugencel zal de aansturing anders zijn, maar voor de organisatie van de structuur maakt dat niet uit.

Een mogelijkheid is om voor de geheugenelementen een D-flipflop te gebruiken. De implementatie van de geheugencel van figuur 11.40 past direct in de structuren van figuur 11.38 en 11.39. De geheugencel heeft een aansluiting met de horizontale woordlijn en drie aansluitingen voor de verticale datalijnen en selectielijn. De woordlijn en de selectielijn zijn actief laag. Als deze signalen beide laag zijn, krijgt de flipflop de waarde van het ingangssignaal `data_in`. De uitgang van de flipflop is via een tristatebuffer aangesloten op `data_out`. Als de woordlijn laag is, zal de uitgang van de flipflop deze signaallijn omlaag trekken.

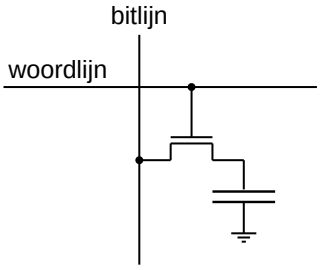


Figuur 11.40 : Een geheugencel voor een RAM met behulp van een D-flipflop.

Een RAM-geheugen dat opgebouwd is uit D-flipfloppen wordt ook wel een *register file* genoemd. Microcontrollers hebben vaak een registerfile ter grootte van bijvoorbeeld 32 bytes waar de rekeneenheid of ALU, *arithmetic logic unit* rechtstreeks mee kan communiceren.

DRAM-technologie

Figuur 11.41 toont een geheugencel met slechts één transistor en één capaciteit. Dit is de basisconfiguratie voor de geheugencel van een DRAM, *dynamic RAM*, of dynamisch RAM.



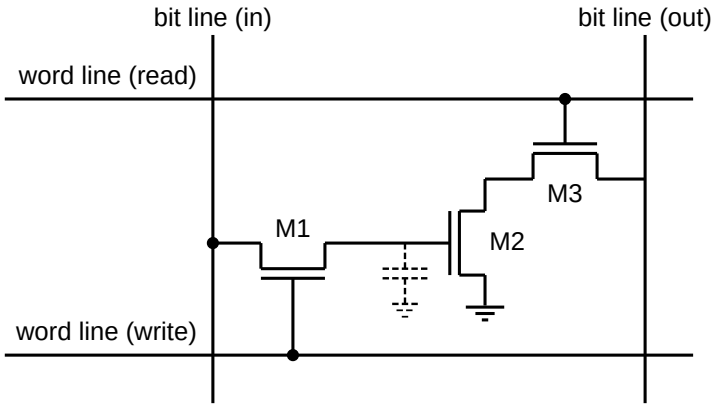
Figuur 11.41 : De 1T-DRAM geheugencel met één transistor en één capaciteit.

Door de woordlijn hoog te maken, wordt de capaciteit geladen als tegelijkertijd de bitlijn hoog is en ontladen als de bitlijn laag is. Nadat de woordlijn laag gemaakt

is, zal de lading van de capaciteit langzaam weglekken. De inhoud van deze geheugencel moet daarom regelmatig worden ververst. Dit wordt gedaan door iedere milliseconde alle waarden van de geheugencellen te lezen en weer terug te schrijven.
Een bit wordt gelezen door de bitlijn de halve voedingsspanning te geven en de woordlijn hoog te maken. De lading op de condensator verstoort het spanningsniveau op de bitlijn. Een speciale sense-amplifier kan uit de verstoring detecteren of de capaciteit geladen was of niet.

Het voordeel van een 1T-DRAM is dat er maar één transistor nodig is en dat een chip met veel geheugencellen daardoor relatief klein en goedkoop is. Het belangrijkste nadeel van dit type geheugencel is dat de informatie ververst moet worden. Hierdoor is om de paar milliseconden de chip een tiental microseconden niet bereikbaar. Andere nadelen zijn dat capaciteiten relatief groot zijn en dat er voor iedere bitlijn een sense-amplifier nodig is, die de kleine variaties in de spanning moet kunnen detecteren.
Er bestaan verschillende alternatieve configuraties voor de 1T-DRAM cel. Figuur 11.42 toont een versie met drie transistoren. Er zijn twee aparte woordlijnen voor het schrijven en het lezen. Er zijn ook twee bitlijnen: een ingang en een uitgang.

Omdat een DRAM-geheugen meestal veel bits bevat zijn er veel adresingangen. Om het aantal aansluitpinnen bij het IC te beperken is er een gemeenschappelijke adresbus voor de rijen en de kolommen. De adressen van rijen en de kolommen worden na elkaar ingelezen. Deze IC's hebben twee extra stuursignalen en twee extra registers voor het rij- en kolomadres. Het signaal `RAS`, *row address strobe*, selecteert het register voor het rij-adres en `CAS`, *column address strobe*, selecteert het register voor het kolomadres.



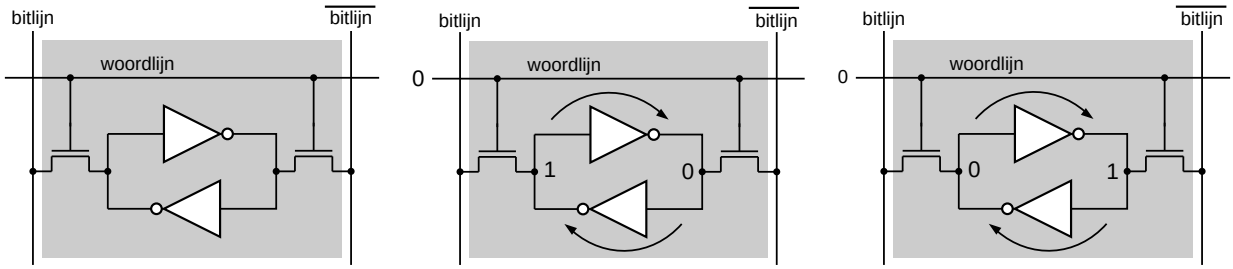
Figuur 11.42 : **Een 3T-DRAM cel met drie transistoren.** De gatecapaciteit van transistor M2 is met streeplijnen aangegeven.

De gatecapaciteit van transistor M2 speelt dezelfde rol als de capaciteit bij de 1T-DRAM cel. De gate wordt opgeladen of ontladen door de woordlijn voor het schrijven hoog te maken. Als de gate geladen is en de woordlijn voor het lezen hoog wordt, wordt de uitgangsbitlijn omlaag getrokken.
Het voordeel van de 3T-DRAM is dat de gatecapaciteit eenvoudiger te produceren is en dat het lezen niet destructief is. De lading blijft bij het lezen op de gate aanwezig. Deze cel moet eveneens regelmatig ververst worden omdat de gate door lekstromen zijn lading verliest.

SRAM-technologie

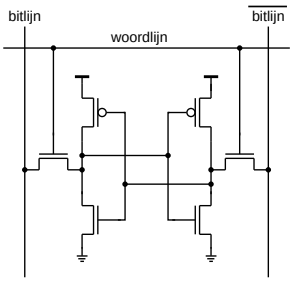
Figuur 11.43 geeft de basisconfiguratie van een SRAM-geheugencel. Deze geheugencel bestaat uit twee inverters, die samen een lus vormen, en twee zogeheten

passtransistoren. SRAM staat voor *static RAM*, oftewel statisch RAM. Als de woordlijn laag is, houdt de lus met de inverters de waarde vast. Een hoge ingang bij de bovenste inverter maakt de uitgang van deze inverter laag en een lage ingang bij de onderste inverter maakt de uitgang van deze inverter hoog. De cel SRAM-cel houdt de waarde vast zonder dat er ververst hoeft worden. Alleen als de voedingsspanning wegvalt, verliest een SRAM-cel zijn informatie.

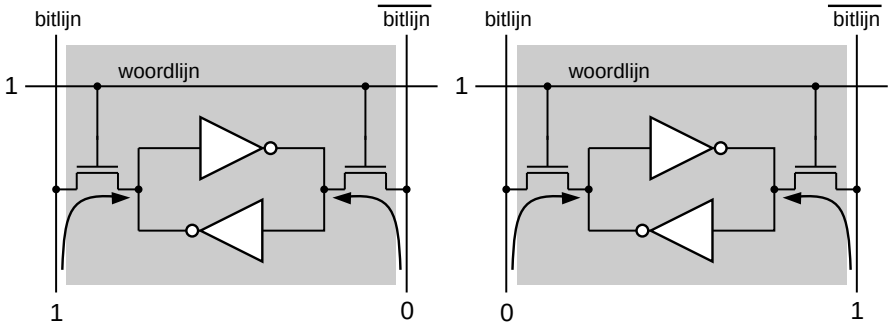


Figuur 11.43 : **De basisconfiguratie van een SRAM-cel.** Links staat de cel met twee inverters en twee transistoren. In het midden en rechts is de woordlijn laag. In het midden wordt een `1` vastgehouden en rechts wordt een `0` vastgehouden

De SRAM heeft een bitlijn en een geïnverteerde bitlijn. Om een `1` naar deze geheugencel te schrijven, moet de bitlijn hoog zijn en de geïnverteerde bitlijn laag zijn, zoals in figuur 11.45 is getekend. Als de woordlijn hoog is, wordt de ingang van de bovenste inverter hoog en die van de onderste laag. Om een `0` naar de geheugencel te schrijven moet de bitlijn juist laag en de geïnverteerde bitlijn hoog zijn.

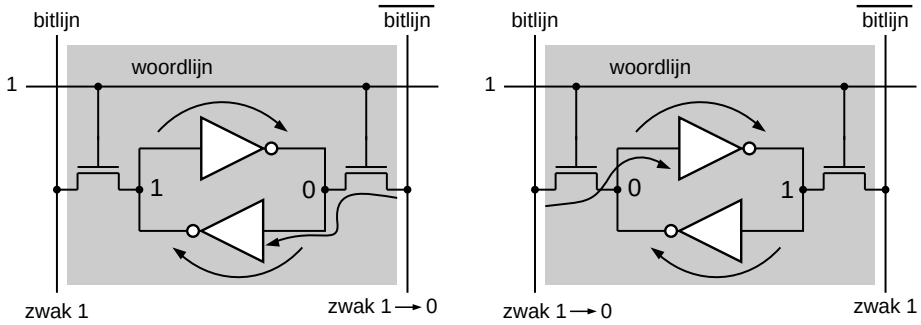


Figuur 11.44 : **6T-SRAM.** Deze SRAM-cel is opgebouwd uit zes transistoren.



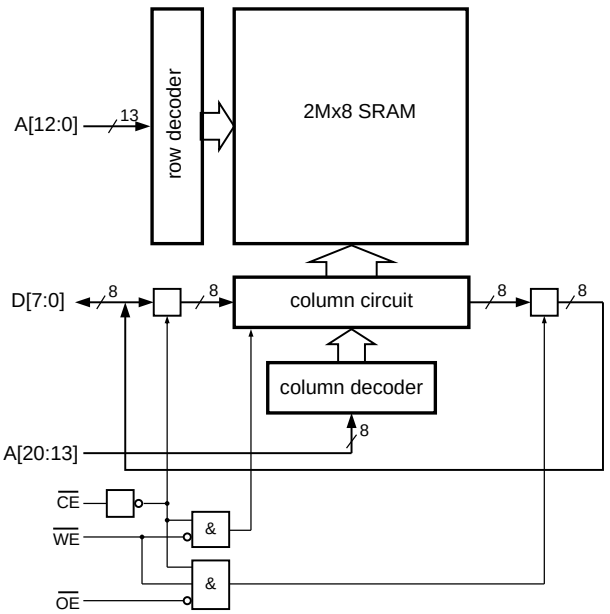
Figuur 11.45 : **Het schrijven naar een SRAM-cel.** In de linker figuur wordt een `1` naar de SRAM-cel geschreven en bij de rechter figuur wordt een `0` naar de SRAM-cel geschreven.

Het lezen van een bit uit een SRAM-cel is in figuur 11.46 gevisualiseerd. De woordlijn wordt hoog en de bitlijnen worden zwak hoog gemaakt. Als de cel een `1` bevat, zal de geïnverteerde bitlijn enigszins omlaag getrokken worden en de sense-amplifier van deze bitlijn detecteert deze kleine verandering. Als daarentegen de cel een `0` bevat, zal de niet-geïnverteerde bitlijn omlaag getrokken worden en wordt dat door de sense-amplifier van deze lijn gedetecteerd.
In figuur 11.44 staat een implementatie van een SRAM-cel met zes transistoren. Bij het lezen loopt er een stroom van de bitlijn via de passtransistor en een van de NMOS-transistoren naar de referentie.



Figuur 11.46 : **Het lezen uit een SRAM-cel.** De linker SRAM-cel bevat een `1`. Bij het hoog worden van de woordlijn, wordt de geïnverteerde bitlijn omlaag getrokken. De rechter SRAM-cel bevat een `0`. Bij het hoog worden van de woordlijn, wordt de niet-geïnverteerde bitlijn omlaag getrokken.

De SRAM-cel heeft in tegenstelling tot de geheugencel uit figuur 11.40 geen aparte ingang en uitgang voor de data, maar een geïnverteerde en een niet-geïnverteerde bitlijn. Als de SRAM-cel in de array van figuur 11.39 wordt gebruikt, kunnen de selectielijnen worden weggelaten en moeten de twee datalijnen geïnterpreteerd worden als de bitlijn en geïnverteerde bitlijn.
Omdat dit soort details de bespreking lastig maakt, worden in datasheets deze bijzonderheden meestal weggelaten. In figuur 11.47 staat een blokschema van een 8Mx8 SRAM zoals dat typisch in een datasheet voorkomt.



Figuur 11.47 : **Een blokschema van een 2Mx8 SRAM.** Het kolomcircuit stuurt bij het schrijven de bitlijnen en geïnverteerde bitlijnen aan en bevat sense-amplifiers, die bij het lezen de bitwaarden uit de spanningsniveaus van de bitlijnen detecteren. De SRAM heeft drie actief lage stuursignalen: de chip enable `CE`, de output enable `OE` en de write enable `WE`. Als `CE` en `WE` laag zijn, worden de ingangsdata in het geheugen gezet en om een waarde uit de SRAM te lezen, moeten `CE` en `OE` laag zijn en moet `WE` hoog zijn.

DRAM versus SRAM

Door de enorme ontwikkeling van computers, laptops en andere elektronische apparaten zijn vele varianten van DRAM- en SRAM-geheugens ontwikkeld. De besproken SRAM- en DRAM-geheugens zijn asynchroon. Aan zowel SRAM als DRAM worden vaak extra registers met een klok toegevoegd, zodat de geheugens synchroon zijn. Een volledige bespreking van al deze varianten valt buiten de context van dit boek. Ook een gedetailleerde bespreking van de elektrische schakelingen, zoals de sense-amplifiers, is hier weggelaten.
Wel is de algemene opbouw van deze geheugens en zijn de belangrijkste technologische eigenschappen aan bod gekomen. De consequenties die daaruit volgen, zijn hier samengevat.

De kenmerken van DRAM-geheugens zijn:

- De opgeslagen gegevens moeten regelmatig ververst worden.
- Er zijn weinig transistoren nodig.
- De bitdichtheid is groot.
- De geheugens zijn relatief klein.
- De geheugens zijn goedkoop.
- Door het verversen zijn de geheugens niet snel.
- Het stroomverbruik is relatief groot.

De kenmerken van SRAM-geheugens zijn:

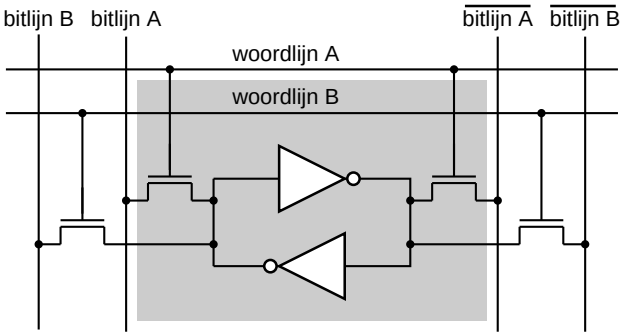
- De opgeslagen gegevens blijven bewaard zolang de voedingsspanning aanwezig is.
- Er zijn meer transistoren nodig.
- De bitdichtheid is kleiner.
- De geheugens zijn relatief groot.
- De geheugens zijn relatief duur.
- De geheugens zijn snel.
- Het stroomverbruik is klein.

SRAM wordt verkozen boven DRAM als snelheid of stroomverbruik belangrijk zijn. SRAM's zijn in vele soorten en maten verkrijgbaar. Een snelle SRAM is veel sneller dan een DRAM en een zuinige SRAM verbruikt veel minder vermogen dan een DRAM.
Ruwweg geldt dat een DRAM de voorkeur heeft boven een SRAM als de grootte en de prijs belangrijk zijn. DRAM wordt toegepast bij computergeheugens en snellere maar duurdere SRAM's worden juist voor cache-geheugens gebruikt.
Voor de ontwikkeling van moderne digitale systemen zijn beide geheugens belangrijk. Een digitaal systeem bestaat immers uit minimaal een microcontroller, een FPGA of een FPGA met een softcore processor en één of meer geheugens. Bovendien is de belangrijkste technologie voor de FPGA's gebaseerd op SRAM.

Dual port SRAM

In figuur 11.48 staat een SRAM-cel met twee woordlijnen, twee stellen bitlijnen en twee extra passtransistoren. Met deze cel kan een geheugen worden gemaakt waarmee gelijktijdig op twee verschillende adressen waarden gelezen en geschreven kunnen worden. Een geheugen met deze 8T-SRAM-cellen wordt een *dual port*

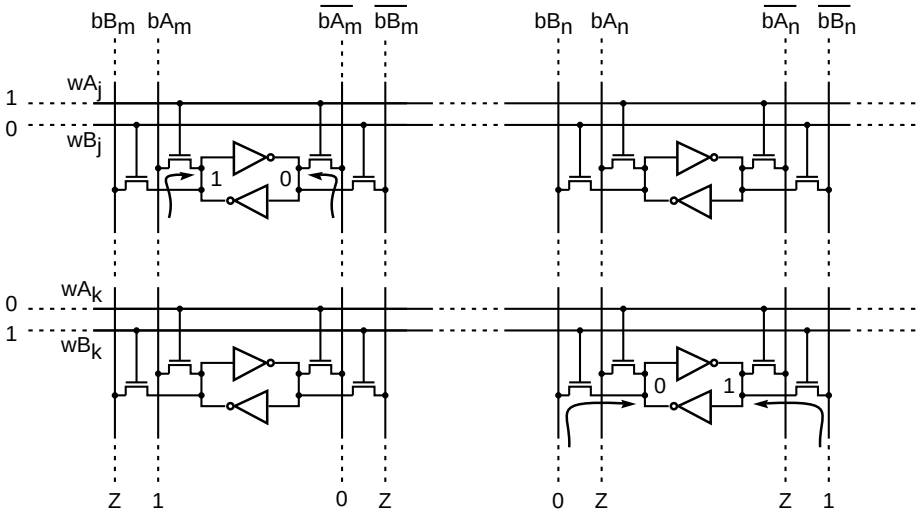
SRAM genoemd. Een geheugen op basis van de 6T-SRAM-cel uit figuur 11.43 is een *single port* SRAM.



Figuur 11.48 : **Een dual-port SRAM-cel met twee inverters en vier passtransistoren.** Het grijze vlak komt overeen met het grijze vlak van de single-port cel uit figuur 11.43.

Als de woordlijnen A en B bij een dual-SRAM-cel laag zijn, houdt de lus met de inverters de waarde vast. Als één van de woordlijnen hoog is, kan de bitcel geschreven worden door de bijbehorende bitlijnen tegengesteld aan te sturen of kan de bitcel uitgelezen worden door de bitlijnen de halve voedingsspanning te geven en de spanningsvariatie op de bitlijn met een sense-amplifier te detecteren.

In figuur 11.49 krijgt de bitcel van rij `j` en kolom `m` gelijktijdig met de bitcel van rij `k` en kolom `n` een nieuwe waarde. Om $\text{bitcel}_{j,m}$ hoog te maken, moet woordlijn wA_j hoog, bitlijn bA_m hoog en bitlijn $\overline{bA_m}$ laag zijn. Tegelijkertijd moet om $\text{bitcel}_{k,n}$ laag te maken woordlijn wB_k hoog, bitlijn bB_n laag en bitlijn $\overline{bB_n}$ hoog zijn. De andere bitlijnen worden niet aangedreven en zijn hoogimpedant.

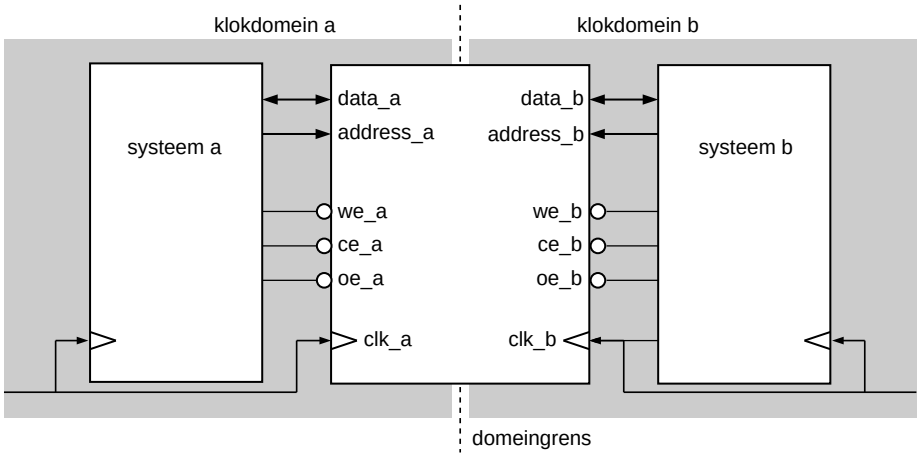


Figuur 11.49 : **De geheugenarray van dual-port SRAM.**

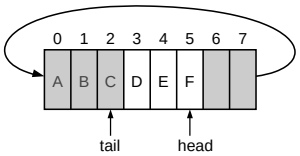
Het gelijktijdig lezen gaat op een zelfde manier door de woordlijnen van twee verschillende rijen hoog te maken en aan twee stellen bitlijnen de halve voedingsspanning aan te bieden. De sense-amplifiers van de betreffende bitlijnen detecteren gelijktijdig de waarden uit twee verschillende bitlijnen.

Ook kan er simultaan naar een bitcel worden geschreven en uit een andere bitcel worden gelezen.

Er treedt een conflictsituatie op bij een dual-SRAM als twee bitcellen met hetzelfde adres gelijktijdig benaderd worden. Zeker bij het simultaan schrijven en het simultaan schrijven en lezen, is ongedefinieerd wat er zal gebeuren.
Om dit probleem op te lossen zijn er twee opties: niets doen en de SRAM gebruiken onder de restrictie dat de twee adressen altijd verschillend zijn of een arbiter gebruiken, die de voorrang regelt. Dual-port SRAM's zijn verkrijgbaar met en zonder arbiter. De componenten met arbiter hebben extra stuursignalen, zoals: semaforen, interrupts of *busy*-signalen.
Synchrone SRAM's hebben extra registers voor de datasignalen en de rij- en kolomadressen. Een synchrone dual-port-SRAM heeft twee extra kloksignalen: één voor de registers van poort A en één voor de registers van poort B.



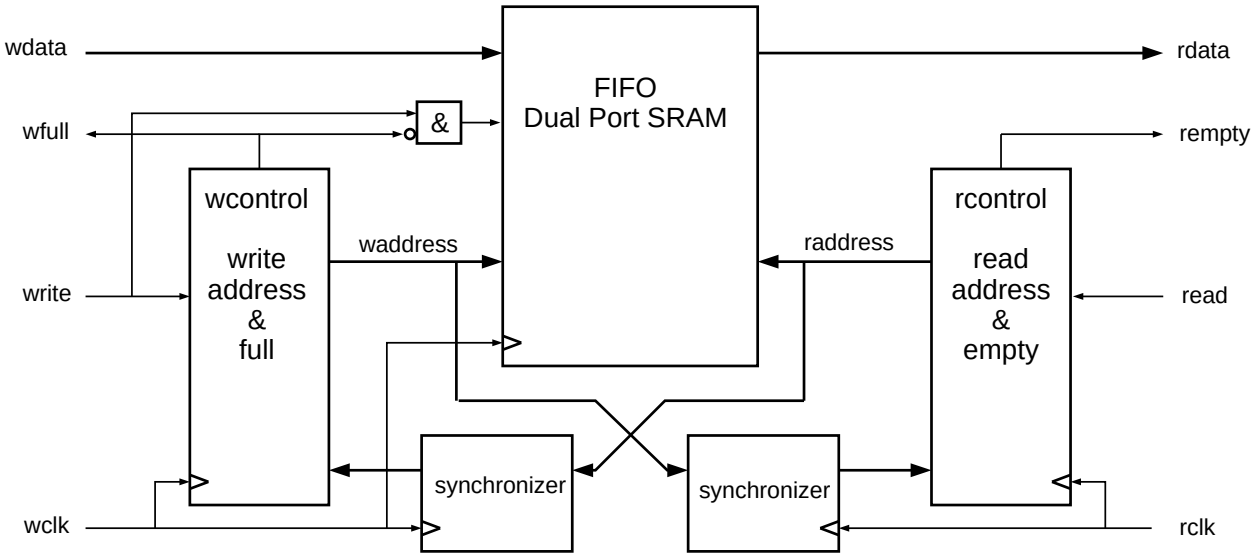
Figuur 11.50 : Twee klokdomeinen met een dual-port SRAM. Met een dual-port geheugen kunnen gegevens tussen twee asynchrone klokdomeinen uitgewisseld worden.



Figuur 11.51 : Een fifo is een circulaire buffer.
Het geheugen is acht bytes groot. De index `head` wijst naar de laatst geschreven byte en de index `tail` wijst naar de laatst gelezen byte.

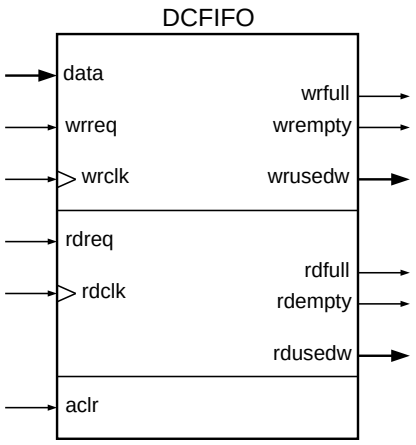
Een dual-port RAM is zeer nuttig bij de data-overdracht tussen twee klokdomeinen. In figuur 11.50 kunnen beide domeinen informatie in het geheugen zetten of uit het geheugen halen. Meestal zijn er extra stuursignalen nodig, die met synchronizers worden gesynchroniseerd.
In figuur 11.52 staat een fifo die gebaseerd is op een dual-port SRAM en die gebruikt wordt voor de dataoverdracht van het klokdomein `wclk` naar het klokdomein `rclk`. Een fifo is een circulaire buffer. Figuur 11.51 laat zien dat er twee adressen of indices zijn die naar de kop en naar de staart van de gegevens in het geheugen wijzen. Bij het schrijven wordt de index van de kop één opgehoogd en bij het lezen wordt de index van de staart één opgehoogd. De fifo is leeg als de indices gelijk zijn en de fifo is vol als het verschil tussen de indices 1 is.

Bij een schrijfopdracht controleert het blok `wcontrol` of er plaats is in het geheugen en zet de nieuwe gegevens in het geheugen en hoogt de index `waddress` met één op. Er mag alleen geschreven worden als de buffer niet vol is. Dit kan het schrijfgedeelte alleen verifiëren als de index `raddress` ook daar bekend is. Via een synchronizer krijgt het schrijfgedeelte deze index van het leesgedeelte.



Figuur 11.52 : **Een fifo op basis van een dual-port SRAM.** De linkerzijde van het schema functioneert met een klok `wclk` en de rechterzijde functioneert met een klok `rclk`. Informatie gaat via het dual-port SRAM van de linkerkant naar de rechterkant.

Bij een leesopdracht controleert het controlblok `rcontrol` of de buffer niet leeg is en haalt de gegevens uit de buffer en hoogt de index `raddress` één op. Er mag alleen gelezen worden als de buffer niet leeg is. Dat kan het leesgedeelte alleen verifiëren als de index `waddress` ook aan deze zijde bekend is. Via een synchronizer krijgt het leesgedeelte deze index van het schrijfdeel.



aansluiting	verplicht	commentaar
`wrclk`	ja	klok schrijfgedeelte
`rdclk`	ja	klok leesgedeelte
`data`	ja	ingangsdata
`q`	ja	uitgangsdata
`wrreq`	ja	write request
`rdreq`	ja	read request
`wrfull`	nee	full synchroon met `wrclk`
`rdfull`	nee	full synchroon met `rdclk`
`wrempty`	nee	empty synchroon met `wrclk`
`rdempty`	nee	empty synchroon met `rdclk`
`wrusedw`	nee	aantal gegevens synchroon met `wrclk`
`rdusedw`	nee	aantal gegevens synchroon met `rdclk`
`aclr`	nee	asynchrone clear voor de statussignalen

Figuur 11.53 : **De dual-clock implementatie van de fifo bij Altera.** Links staat het symbool uit de datasheet en in de tabel staat een omschrijving van de signalen.

De RAM-blokken van FPGA's zijn gebaseerd op dual-SRAM en zijn op verschillende manieren configureerbaar. Ze kunnen gebruikt worden als single-port of dual-port, synchroon of asynchroon en met en zonder enable-signalen.
De ontwikkelomgevingen voor FPGA's hebben functiegeneratoren die met deze RAM-blokken stacks, fifo's en andere bufferstructuren maken. Figuur 11.53 geeft de configureerbare dual-clock fifo van Altera.

11.7 FPGA

Het concept voor FPGA's is begin jaren tachtig bedacht door Ross Freeman en Jim Barnett. Om dit idee verder te ontwikkelen hebben zij in 1984 Xilinx opgericht. De eerste FPGA, de XC2064, was in 1986 beschikbaar. De FPGA is een alternatief voor de zogenoemde gate-array. Dit is een structuur met vooraf aangebrachte transistoren, waaraan alleen de metaallaag voor de verbinding tussen de transistoren ontbreekt. De ontwerper maakt een ontwerp op basis van logische poorten en de onderlinge verbinding. De ontwerpsoftware vertaalt dit naar een metaalmasker. In de IC-fabriek wordt de metaallaag aan de geïntegreerde schakeling toegevoegd.

Spreek Xilinx uit als *zai-links*.

Nadelen van gate-array's zijn de lange doorlooptijd en het hoge volume dat nodig is om het rendabel te maken. Het ontwerp is relatief duur en moet in één keer correct zijn. Het concept van Xilinx is dat de transistoren en de verbindingen al compleet op het IC aanwezig zijn en dat de functionaliteit van de FPGA elektrisch, met behulp van programmeerbare transistoren, wordt vastgelegd.

Xilinx gebruikte oorspronkelijk de term LCA, *logic cell array* voor de FPGA.

Omdat de ontwerper dit zelf op zijn werkplek doet, wordt deze bouwsteen een FPGA, *field programmable gate array*, genoemd. Dit in tegenstelling tot de klassieke gate-array, die ook aangeduid wordt met MPGA, *mask programmable gate array*.

In 1975 voorspelde Gordon E. Moore dat de integratie van IC's zich ontwikkelt met een factor 2 per twee jaar. Later heeft hij deze voorspelling aanscherpt tot een factor 2 in anderhalf jaar. Deze voorspelling wordt de wet van Moore genoemd. Paragraaf 11.9 bespreekt de wet van Moore en de toekomstige ontwikkelingen.

Xilinx heeft al jarenlang meer dan 50% van de markt voor FPGA's in handen. De tweede grote speler op deze markt is Altera met een aandeel van ongeveer 35%. Andere fabrikanten zijn Atmel, Cypress, Actel en Lattice. De ontwikkelingen van FPGA's volgt vanaf het begin de wet van Moore. Elke anderhalf jaar verdubbelen het aantal transistoren en de mogelijkheden.

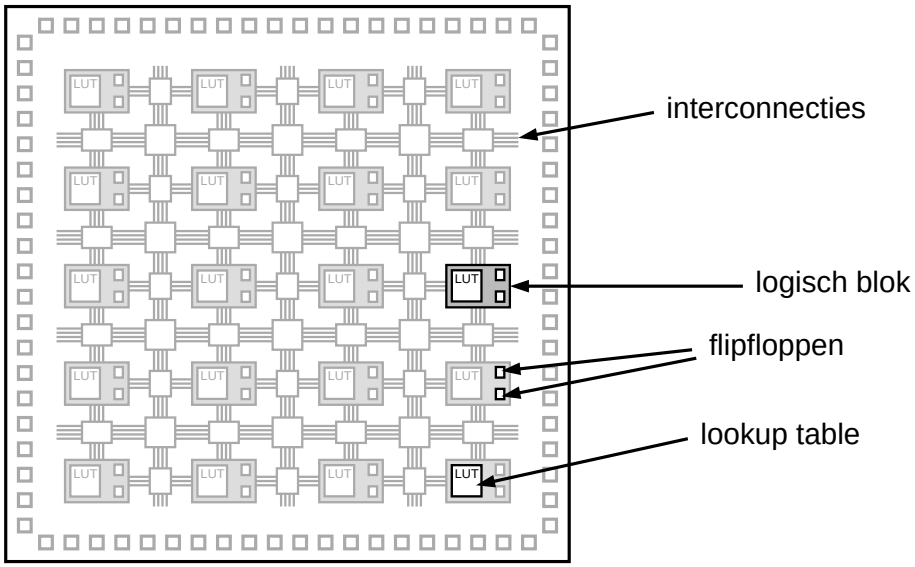
De snelle ontwikkeling van de integratiedichtheid is zeer bijzonder. De afgelopen tientallen jaren zijn er enorme technologische stappen gemaakt. De omvangrijkste FPGA's bevatten nu enkele miljarden transistoren. De kleinste afmetingen op een IC zijn nu ongeveer 25 nm. Dat is niet meer dan tien keer de atoomafstand. Als de wet van Moore doorzet, zullen in 2030 de grootste FPGA's 1000 miljard transistoren bevatten.

De algemene structuur van een FPGA

In paragraaf 1.1 is op pagina 3 de FPGA al geïntroduceerd. De opbouw van figuur 1.1 is een zeer ruwe schets van een echte FPGA. In figuur 11.54 is deze opbouw nogmaals getekend.

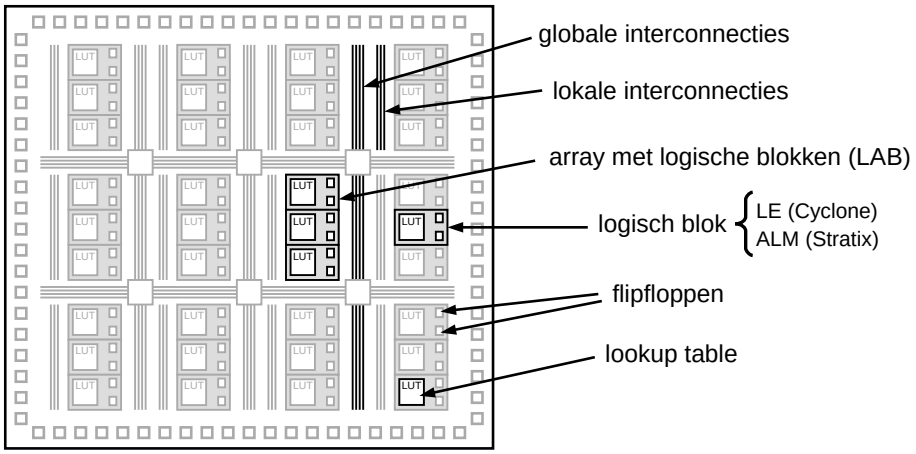
Een FPGA bestaat uit logische blokken en programmeerbare verbindingen. De logische blokken bevatten één of meer LUT's, *lookup tables* waarmee logica gemaakt kan worden en één of meer D-flipfloppen om gegevens op te slaan. Deze bouwblokken zijn veel kleiner dan de PAL-achtige blokken van de CPLD's. De XC2064, de eerste FPGA van Xilinx, bevatte 64 logische blokken en elk van deze blokken bestond uit een 4-input LUT en een flipflop. Dat is al veel meer dan de structuur uit figuur 11.54 suggereert. Moderne FPGA's bevatten nog veel meer logische blokken. De XC6SLX4, de kleinste FPGA uit de Spartan-6 familie van Xilinx, bevat 600 logische blokken, *slices*, met vier 6-input LUT's en acht flipfloppen. De Spartan-6 familie is de meest recente eenvoudige reeks die door Xilinx is uitgebracht. De grootste FPGA uit de complexere Virtex-6 familie, de XC6VLX760, bevat 118560 logische blokken, *slices*, met eveneens vier 6-input LUT's en acht flipfloppen.

Vergelijken van de grootte van FPGA's uit verschillende families is lastig. De definitie van een logisch blok is anders. Xilinx gebruikt nog steeds de naam CLB, *configurable logic block*. Oorspronkelijk kwam dat overeen met wat hier bedoeld wordt met een logisch blok. Bij Spartan en Virtex wordt tegenwoordig de term *slice* gebruikt. Een CLB bestaat bij de Spartan-6 en bij de Virtex-6 uit twee slices.



Figuur 11.54 : De structuur van een FPGA.

Altera gebruikt bij de Cyclone-serie de LE, *logic element*, en bij de Stratix-serie de ALM, *adaptive logic module* als basiselement. Altera groepeert deze basiselementen in LAB's, *logic array blocks*. In figuur 11.55 staat een sterk vereenvoudigde afbeelding van een FPGA van Altera. De EP1C3, de kleinste FPGA uit de inmiddels verouderde Cyclone-I reeks, heeft 24 kolommen en 13 rijen met totaal 291 LAB's met ieder 10 LE's.

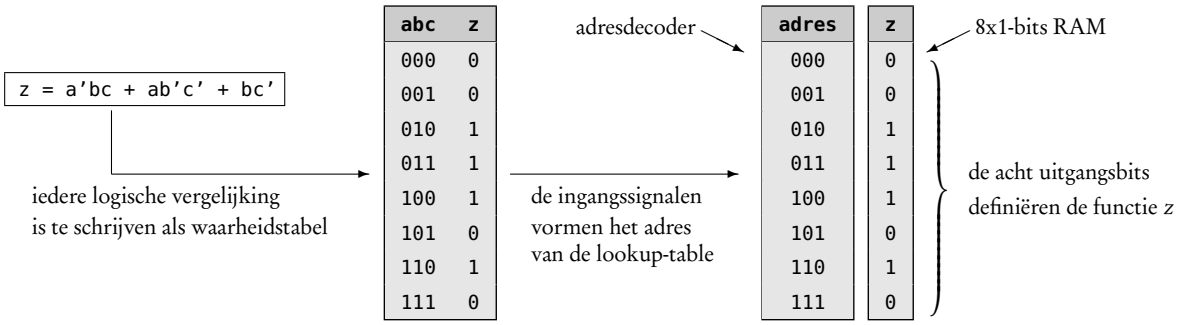


Figuur 11.55 : De structuur van een FPGA van Altera.

Door de LAB's lijkt de structuur van Altera op het eerste gezicht op de structuur van CPLD's. Het grote verschil met CPLD's is dat de basisbouwsteen niet de LAB is, maar het logische blok. Het onderscheid tussen CPLD's en FPGA's is dat de laatste een veel fijne structuur hebben en dat ze heel veel logica en flipfloppen hebben. De EP3C120 uit de Cyclone-III heeft 119088 logische blokken en daarnaast nog bijna 4 miljoen bits in de vorm van RAM-blokken en 288 18x18-vermenigvuldigers.

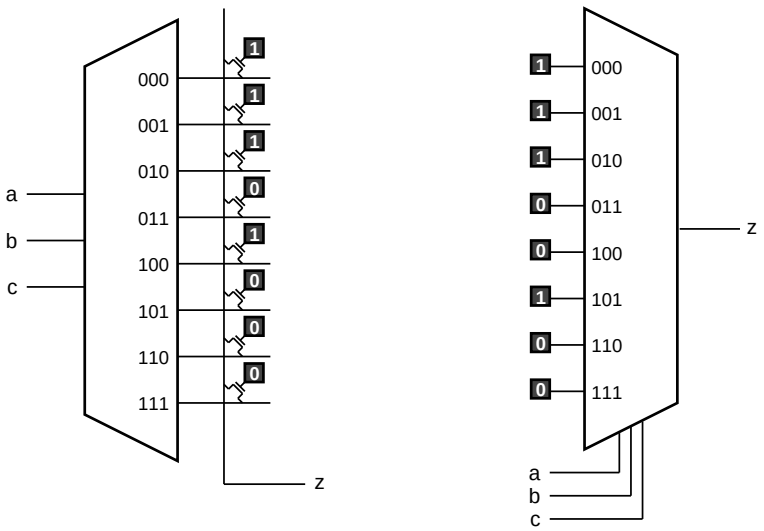
Lookup table

Een logisch blok bevat in ieder geval minimaal een LUT en een D-flipflop. De LUT, *lookup table*, is het meest kenmerkende onderdeel voor een FPGA.
Iedere logische vergelijking is te schrijven in de vorm van een waarheidstabel. Bij een vergelijking met n variabelen heeft de tabel een lengte van 2^n. De variabelen vormen samen een adres. Afhankelijk van de logische functie hoort bij ieder adres een specifieke waarde. Met een $2^n \times 1$-bits RAM en een adresdecoder, kan zo iedere functie gemaakt worden.



Figuur 11.56 : **De implementatie van een logische vergelijking met een lookup-table.** De waarheidstabel van de logische vergelijking is in feite de combinatie van een 3x8-adresdecoder en een 8x1-bits RAM

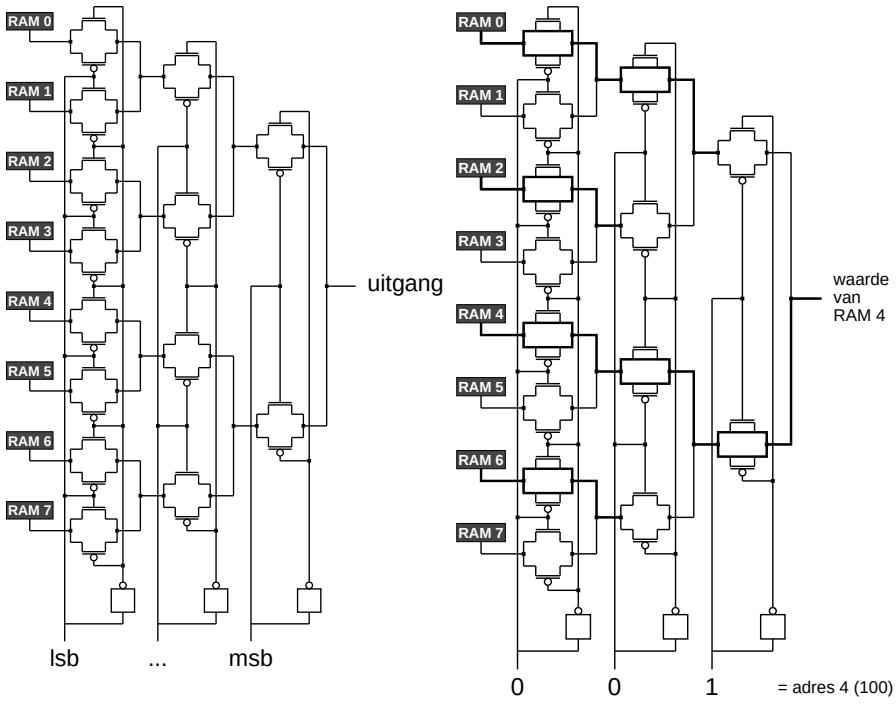
In figuur 11.56 staat de waarheidstabel die hoort bij een logische vergelijking met drie ingangen `a`, `b` en `c`. De uitgang is `z` en de waarheidstabel is acht regels groot. Deze functie kan gerealiseerd worden met een `3x8`-adresdecoder en een `8x1`-bits RAM.



Figuur 11.57 : **Twee implementaties van een lookup-table.** Links staat een LUT met `3x8`-adresdecoder en `8x1`-bits RAM en rechts staat een LUT met `8-to-1` multiplexer en `8x1`-bits RAM. Beide LUT's representeren de functie `z = a'c'+b'c`.

Figuur 11.56 suggereert dat een LUT altijd bestaat uit een adresdecoder en een $2^n \times 1$-bits RAM. In het algemeen wordt de LUT gerealiseerd met een multiplexer

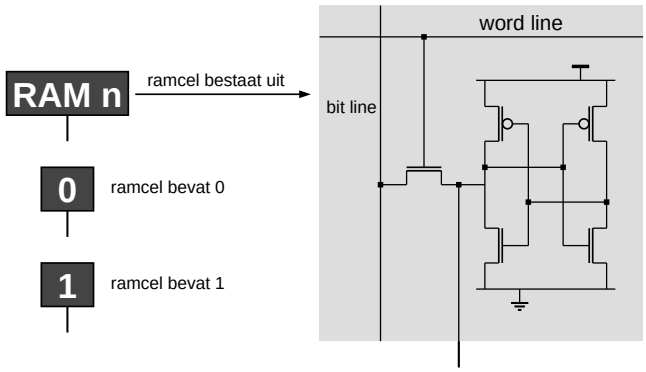
en een $2^n \times 1$-bits RAM. Figuur 11.57 geeft de RAM/ROM-achtige implementatie van een LUT met een adresdecoder en een implementatie met een multiplexer. Voor de RAM/ROM-achtige structuur zou per LUT — net als bij een gewoon RAM — een sense-amplifier nodig zijn. Bij de structuur met de multiplexer is dit niet nodig. Daarom wordt deze structuur bij FPGA's gebruikt.



Figuur 11.58 : **Een LUT met een multiplexer op basis van transmissiepoorten.** De linker figuur geeft de structuur van de LUT en de rechter figuur toont de selectie voor de ingangswaarde `100`.

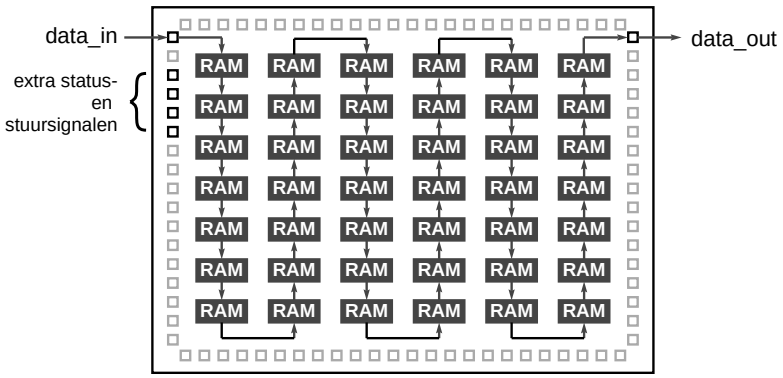
De multiplexer wordt met behulp van passtransistoren of met behulp van transmissiepoorten gerealiseerd. In figuur 11.58 is een LUT met drie ingangen en een 8-to-1 multiplexer met transmissiepoorten getekend. Bij alle acht ingangscombinaties verschijnt één van de acht waarden van de RAM-cellen op de uitgang. In figuur 11.58 is dit voor de ingangscombinatie `100` getekend.

De RAM-cellen van een LUT zijn gebaseerd op de SRAM-cel uit de figuren 11.43 en 11.44 en is in figuur 11.59 getekend. De uitgang van de SRAM-cel wordt bij de LUT niet uitgelezen via de bitlijnen, maar wordt direct aangesloten op één van de ingangen van de multiplexer. De geïnverteerde bitlijn is daarom weggelaten. De transistoren worden zo gedimensioneerd dat bij het schrijven de waarde van de SRAM-cel overschreven kan worden.
De woordlijn en de bitlijn zijn nodig voor het programmeren van de SRAM-cel. Dit gebeurt bij de configuratie of bij de reconfiguratie van de FPGA. Voor de gewone functionaliteit zijn deze lijnen niet belangrijk en worden daarom bij een LUT niet getekend. In deze paragraaf worden bij de tekeningen van de LUT's voor de SRAM-cellen abstracte symbolen gebruikt, die eveneens in figuur 11.59 zijn getekend.



Figuur 11.59 : **De SRAM-cel van een FPGA.** Links staan het symbool van de SRAM-cel, zoals in figuur 11.58 is gebruikt, en de symbolen voor een geprogrammeerde SRAM-cel uit figuur 11.57. Rechts staat de transistorschakeling van de SRAM-cel.

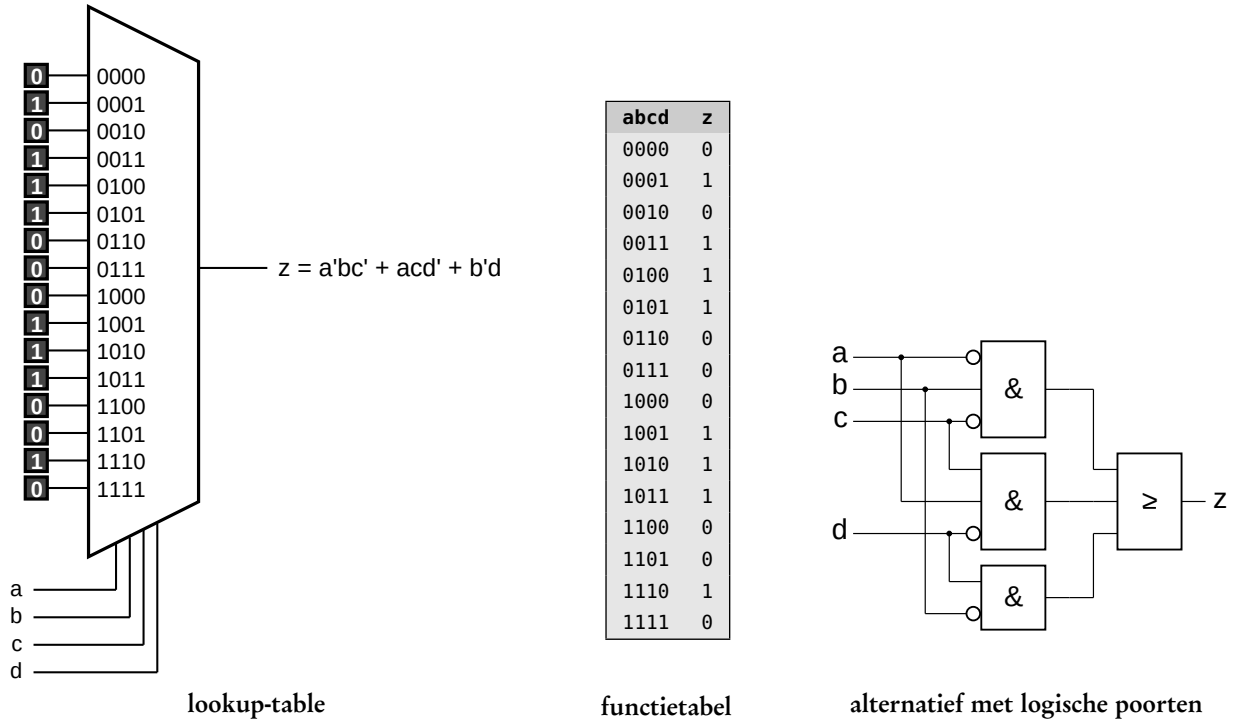
Bij de configuratie worden de enen en nullen waarmee de FPGA geprogrammeerd wordt serieel naar binnen geschoven. Hoewel in werkelijkheid de structuur complexer is, kan de ontwerper bij de configuratie de FPGA als één lang schuifregister opvatten. De SRAM-cellen van de LUT's en van de programmeerbare interconnecties vormen een lange rij. Het adres van de SRAM-cellen is zowel bij het schrijven als bij het lezen dus niet essentieel. De woordlijn en de bitlijn kunnen daarom ook als een *write enable* en als een datalijn geïnterpreteerd worden.



Figuur 11.60 : **Bij de configuratie vormen de SRAM-cellen een lange ketting.**

In figuur 11.60 is voor een FPGA de seriële verbinding tussen de SRAM-cellen gevisualiseerd. Naast een aansluiting voor het ingangssignaal is er een uitgangssignaal, zodat meerdere FPGA's vanuit een programmer geprogrammeerd kunnen worden en zijn er een aantal status- en stuursignalen.

In het algemeen bevat een logisch blok van een FPGA een of twee LUT's met drie of vier ingangen. Figuur 11.61 toont een LUT met vier ingangen en zestien SRAM-cellen. De zestien bits `0101110001110010` representeren de schakeling die ook in figuur 11.61 is getekend. Als deze functie met NAND-poorten gerealiseerd wordt, zijn er 30 transistoren nodig. Voor de 4-input LUT zijn meer dan 150 transistoren nodig; dat is dus vijf maal zoveel. De overhead is nog veel groter als de LUT gebruikt wordt om alleen een 2-input NAND te maken; daar zijn

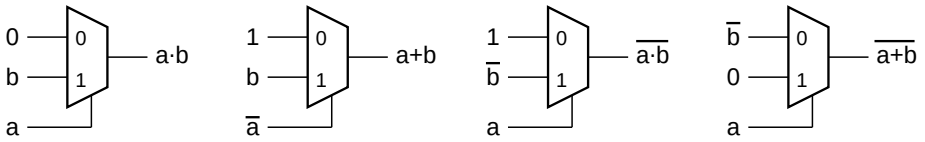


Figuur 11.61 : **Een LUT met vier ingangen.** Links staat de implementatie van de LUT met een 16-to-1 multiplexer en een 16x1 SRAM. De functietabel van de implementatie staat in het midden en rechts staat de alternatieve schakeling met AND- en OR-poorten.

slechts vier transistoren voor nodig. Het aantal transistoren van een LUT met n ingangen is evenredig met 2^n. Dat is de reden dat de logische blokken in het algemeen geen lookup-tables hebben met vijf of zes ingangen. Wel hebben de logische blokken soms twee of meer LUT's, die afzonderlijk gebruikt kunnen worden of gecombineerd kunnen worden tot een LUT met vijf of zes ingangen.

Een alternatief voor de lookup-table

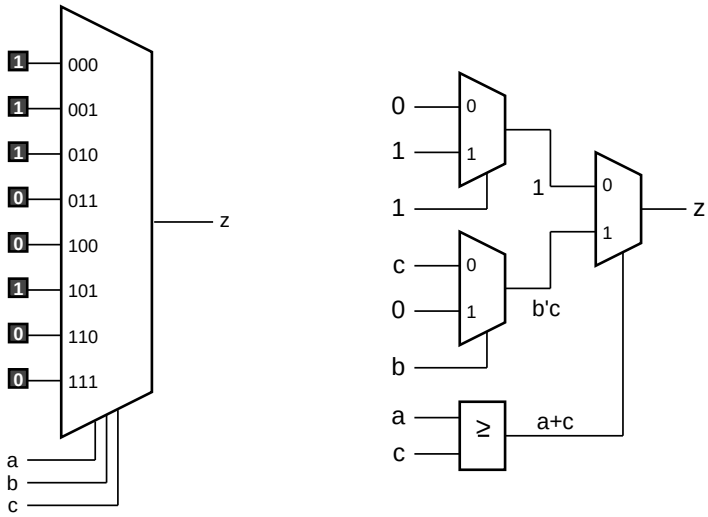
Sommige FPGA's, zoals die van Actel, gebruiken geen lookup-table om de logische functies te realiseren, maar gebruiken multiplexers en logische poorten. Als de ingangssignalen geïnverteerd en niet-geïnverteerd beschikbaar zijn, kan met een 4-to-1 multiplexer iedere logische functie met drie ingangen worden gecreëerd.



Figuur 11.62 : **De realisatie van diverse logische functies met behulp van een multiplexer.**

In figuur 11.63 staat een voorbeeld van een logische cel van een Actel FPGA en een overeenkomstige implementatie met behulp van een LUT. De logische cel bestaat uit een drie multiplexers en een OR-poort.

Voor de oplossing met de LUT is eenvoudig te zien dat de geïmplementeerde logische functie overeenkomt met `a'c'+b'c`. Bij de oplossing met de multiplexers is dat wat lastiger te zien. De uitgang is hoog als `a+c` laag is; dus als `a'c'` hoog is. De uitgang is ook hoog als `b'c` hoog is.

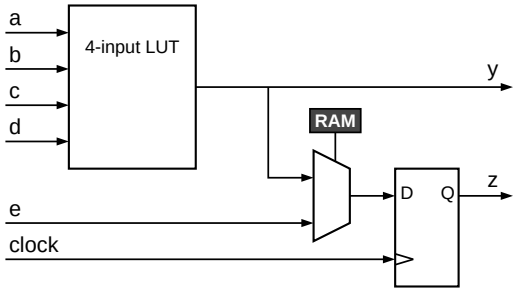


Figuur 11.63 : Een logische functie met behulp van een LUT en met multiplexers. Links staat een LUT met een `8-to-1` multiplexer en een `8x1`-bits RAM. Rechts staat de alternatieve implementatie van Actel met multiplexers en een OR-poort. In beide gevallen wordt de functie `z = a'c'+b'c` gerepresenteerd.

Omdat de FPGA's van Altera en Xilinx gebaseerd zijn op LUT's en deze leveranciers verreweg het grootste marktaandeel hebben, wordt in de rest van dit boek altijd een LUT gebruikt als logische component in de logische blokken.

Het logische blok

Een logisch blok bevat naast een of twee LUT's minimaal een of twee D-flipflop-pen. De flipfloppen uit de verschillende logische blokken kunnen gecombineerd woorden tot dataregisters, schuifregisters en tellers. De combinatie van een relatief groot aantal kleine logische functies en veel flipfloppen maken de FPGA's zeer krachtig.

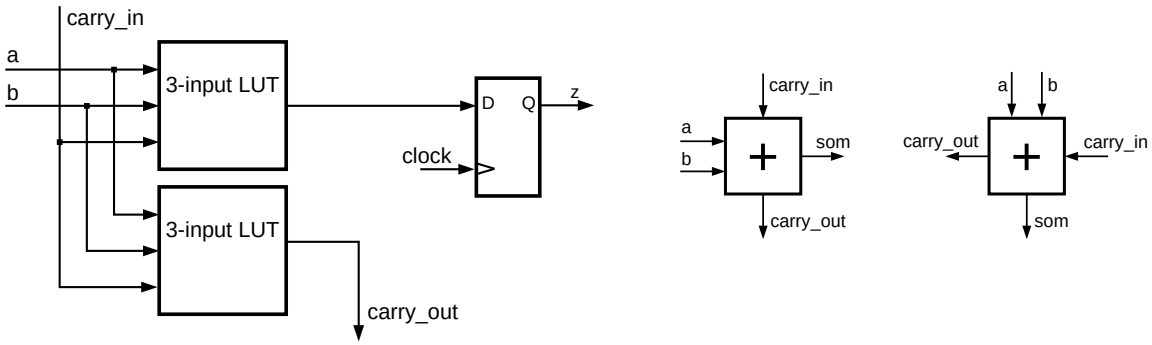


Figuur 11.64 : Een logische blok met een 4-input LUT en een D-flipflop.

In figuur 11.64 staat een logisch blok met een 4-input LUT en een D-flipflop. Naast de vier ingangen van de LUT heeft het logische blok een vijfde dataingang en een klokingang. Het blok heeft een combinatorische uitgang en een uitgang met een D-flipflop.

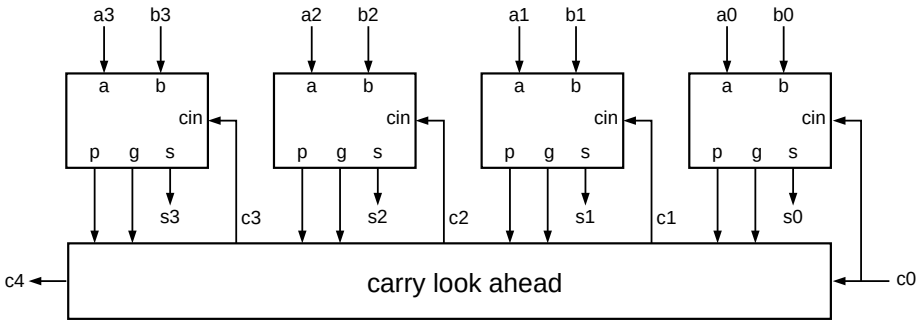
De multiplexer selecteert of de uitgang van de LUT of de vijfde dataingang op de flipflop is aangesloten. De functionaliteit van dit logische blok wordt bepaald door de zestien SRAM-cellen van de LUT en de SRAM-cel van de multiplexer.

Het logische blok uit figuur 11.65 heeft een carry-in en twee LUT's. Eén wordt er gebruikt om de som te berekenen en één om de carry-out uit te rekenen.



Figuur 11.65 : Een logisch blok met twee 3-input LUT's en een carry-signaal. Links staat het logische blok en rechts staan twee symbolen voor de logische functie van de LUT.

Met de full-adder uit figuur 11.65 kan de n-bits opteller uit figuur 4.33 gemaakt worden. In principe levert dat een zogenoemde *ripple adder* op. De verandering van de carry-in `c0` plant via de vier full-adders door naar de carry-out.

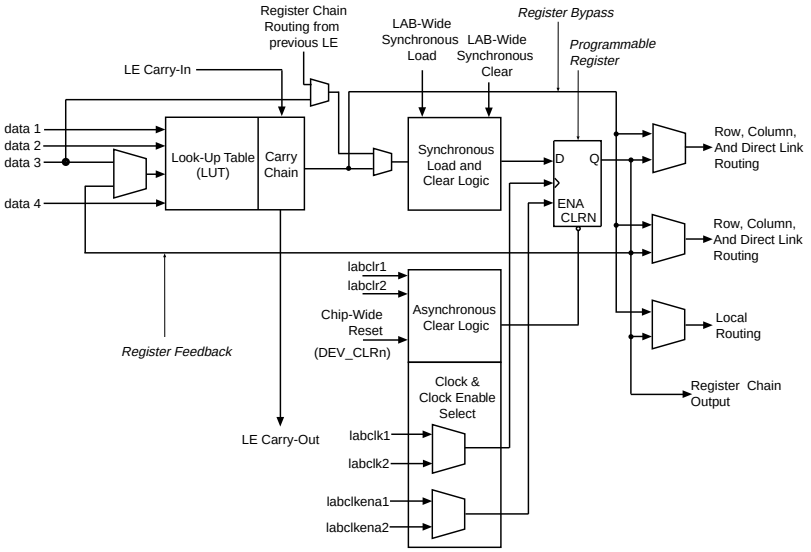


Figuur 11.66 : Een carry-lookahead adder.

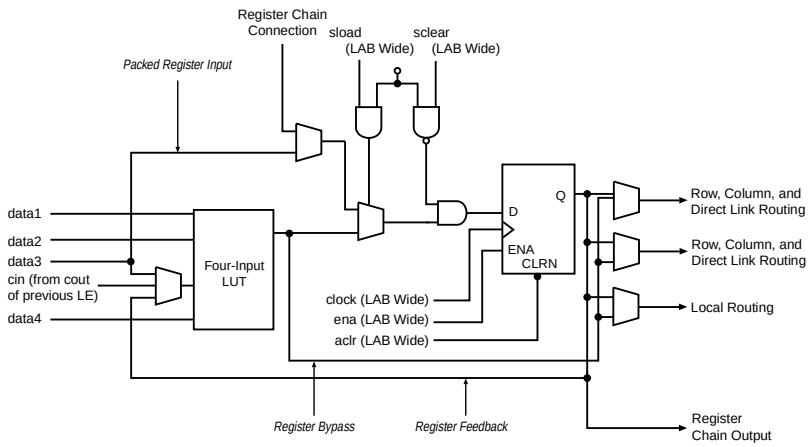
Om de opteller sneller te maken, kan de *carry-lookahead adder* uit figuur 11.66 gebruikt worden. Aan de full-adders ontbreekt een deel van de logica van de carry-out. In plaats daarvan hebben deze niet-complete full-adders twee extra uitgangssignalen `p` en `g`. Deze signalen komen overeen met de interne signalen `p` en `g` uit de VHDL-beschrijving van code 2.7.

Ieder logisch blok van de diverse FPGA-families van de verschillende fabrikanten hebben hun eigen specifieke mogelijkheden. Het is niet mogelijk om hier alle aspecten van alle verschillende logische blokken volledig te bespreken. Omdat de synthese tools en de plaatsings- en bedradingsprogramma's al deze details kennen en daar gebruik van maken bij de synthese en bij de plaatsing en bedrading, is het voor de ontwerper ook niet nodig om dit in detail te weten.

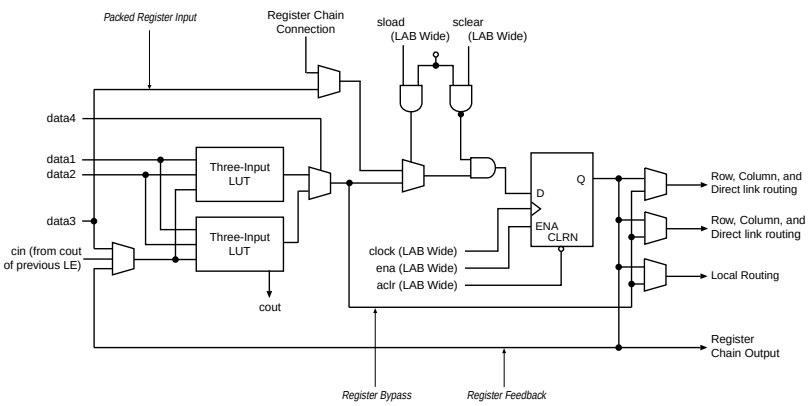
Voor een ontwerper is het belangrijk dat deze op de hoogte is van de belangrijkste aspecten van de logische blokken en van de andere faciliteiten van de FPGA.



Het logische element van de Cyclone-III familie.



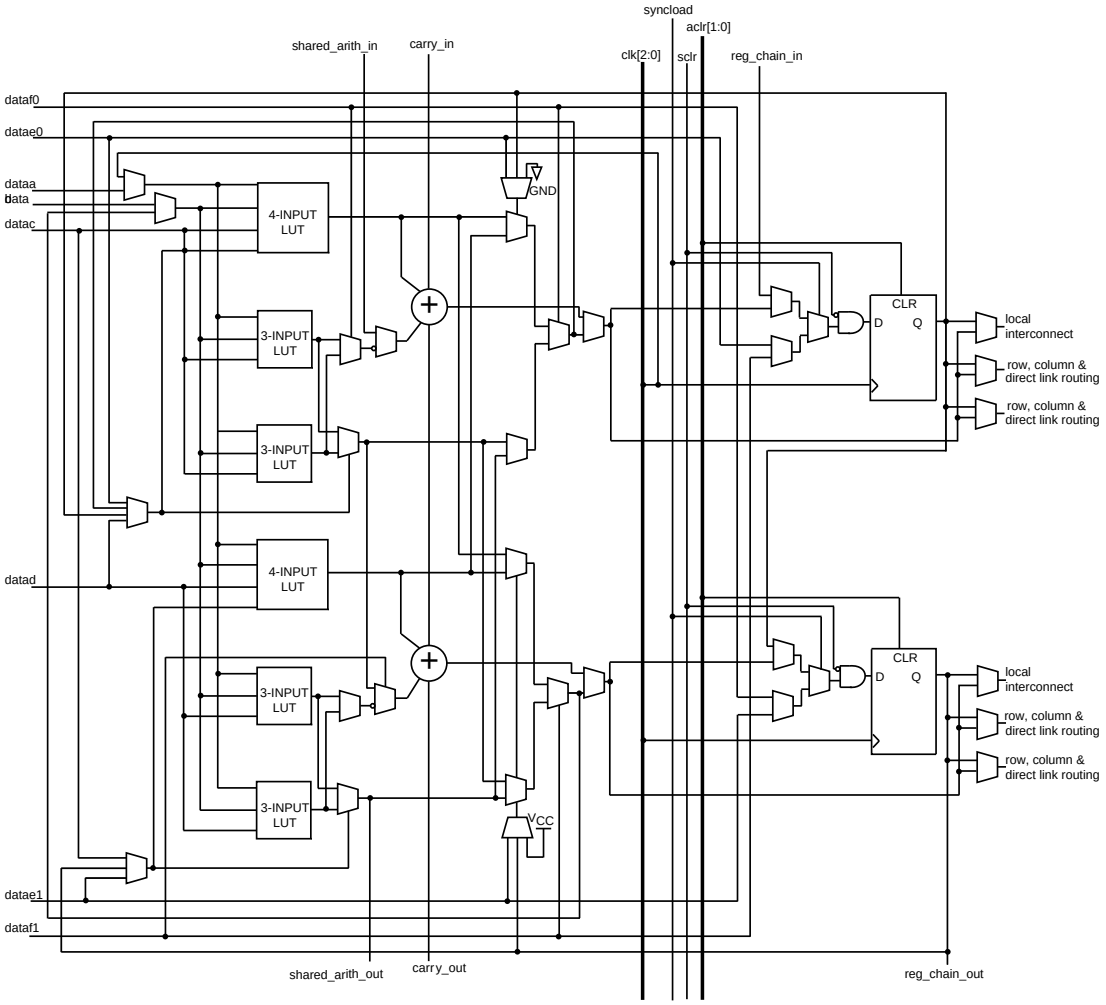
Het logische element van de Cyclone-III familie in de normale modus.



Het logische element van de Cyclone-III familie in de rekenkundige modus.

Figuur 11.67 : Het logische element van de Cyclone-III familie van Altera. De bovenste schakeling geeft de generieke structuur. De andere twee schakelingen tonen de schakeling in de normale modus en in de rekenkundige modus.

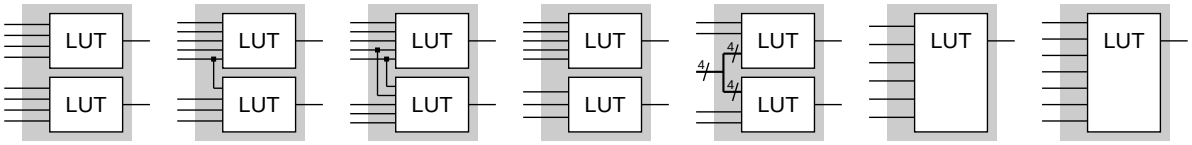
Als voorbeeld staat in figuur 11.67 het logische blok van de Cyclone-III familie van Altera. Altera noemt dit blok een LE, *logic element*. Het bevat een 4-input LUT, een D-flipflop, speciale logica voor een carry-in en carry-out en een speciale instelling voor load-, clear- en enable-signalen. Deze LE kan gebruikt worden als normaal logisch blok met een 4-input LUT en een D-flipflop, maar heeft ook een speciale modus voor rekenkundige bewerkingen. In het laatste geval wordt de LUT gesplitst in twee 3-input LUT's met een carry-in en een carry-out. De functionaliteit van het logische element komt dan overeen met die van het logische blok uit figuur 11.65.



Figuur 11.68 : **Het logische blok van de Stratix-III familie van Altera.** Deze ALM, *adaptive logic module*, bevat meerdere LUT's, die gecombineerd kunnen worden, twee flipfloppen, een *fast carry* met twee full-adders en een speciale aansluiting om met de flipfloppen registers te maken.

In figuur 11.68 staat het logische blok uit de Stratix-III familie van Altera. Altera noemt dit logische blok een ALM, *adaptive logic module*. Het bevat meerdere LUT's die op een aantal manieren gecombineerd kunnen worden. Verder heeft iedere ALM twee flipfloppen, een *fast carry* met twee full-adders en een speciale aansluiting om met de flipfloppen uit naburige ALM's registers te maken.

Figuur 11.69 laat de zeven combinaties zien die met de LUT's uit de ALM gecreeerd kunnen worden. Met de 7-input LUT zijn niet alle functies mogelijk.

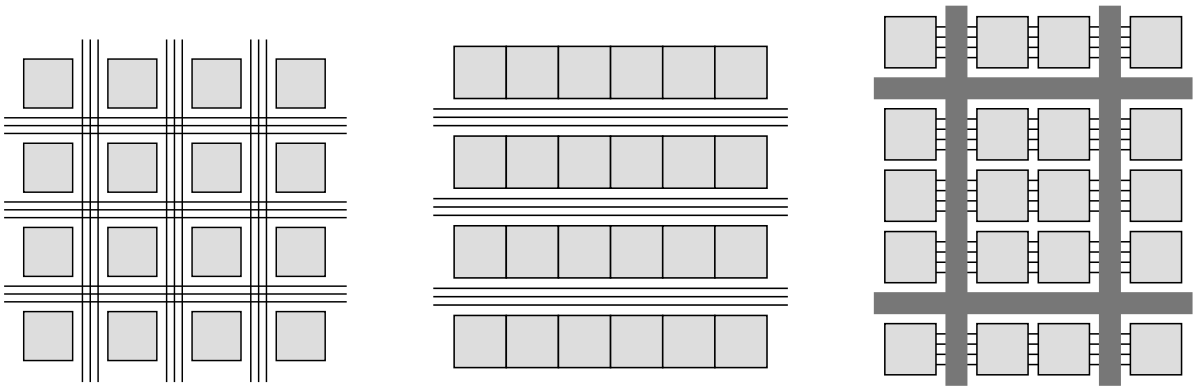


Figuur 11.69 : De zeven combinaties die met de LUT's van de ALM mogelijk zijn.

Al deze verschillende mogelijkheden en instellingen worden gerealiseerd met behulp van multiplexers, die net als de multiplexer uit figuur 11.64, met SRAM-cellen configureerbaar zijn.

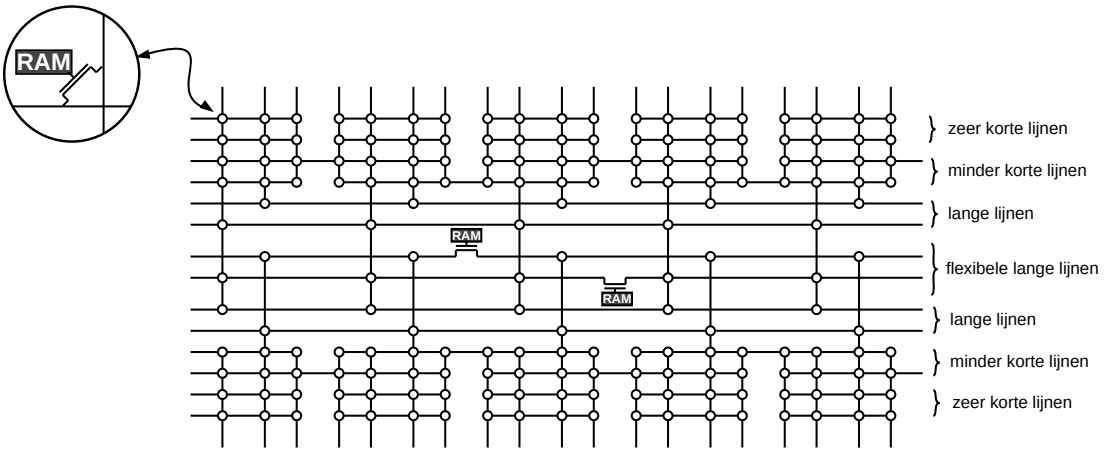
De structuur en de programmeerbare interconnecties van een FPGA

De vele logische blokken op FPGA's moeten met elkaar verbonden kunnen worden om complexe functies te maken. Net als iedere fabrikant een eigen oplossing heeft voor de logische blokken, zijn er ook verschillende oplossingen voor de structuur en de manier waarop de blokken verbonden zijn. Figuur 11.70 toont de drie verschillende basisstructuren.



Figuur 11.70 : Verschillende bedradingsstructuren voor FPGA's.

De eilandstructuur wordt door Xilinx gebruikt. Bij deze structuur liggen de logische blokken in een matrix en zijn er verticale en horizontale connecties, die ook onderling met elkaar verbonden kunnen worden.
De rijstructuur van Actel lijkt op de structuur van gate-array's en is opgebouwd uit rijen met logische blokken met daar tussen de bedradingskanalen. De in- en uitgangen van de verschillende logische blokken kunnen via de bedradingskanalen met elkaar verbonden worden. Via speciale schakelblokken worden de bedradingskanalen ook onderling verbonden.
Bij de hiërarchische structuur hebben de logische blokken een rij- en kolom-structuur. Binnen een kolom of rij zijn er lokale verbindingen en om logische blokken met elkaar te verbinden zijn er lange verticale en horizontale verbindingslijnen. Altera gebruikt een hiërarchische structuur.



Figuur 11.71 : **Een voorbeeld van een bedradingskanaal.** De horizontale verbindingslijnen hebben verschillende lengtes. Er zijn korte, minder korte en lange lijnen. De witte stippen geven de programmeerbare knooppunten. Het knooppunt linksboven is uitvergroot.

Om met logische blokken complexere functies te maken, zijn de verbindingen op een FPGA programmeerbaar. De logische blokken kunnen met programmeerbare knooppunten wel of niet met de interconnecties verbonden worden en de verticale en horizontale interconnecties kunnen ook onderling verbonden worden. Een programmeerbaar knooppunt bestaat uit bijvoorbeeld een passtransistor en een SRAM-cel.

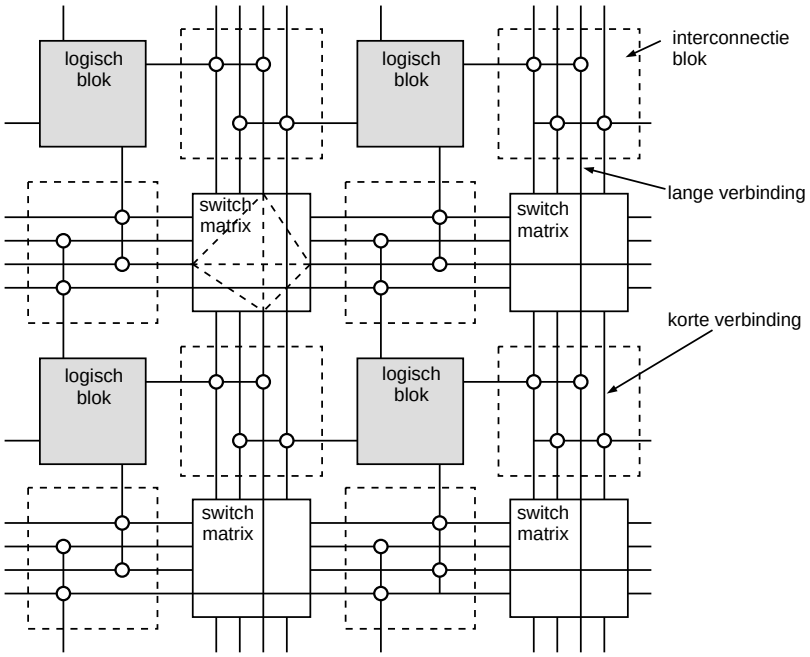
Figuur 11.71 geeft een voorbeeld van een horizontaal bedradingskanaal uit een FPGA met een rij-structuur. Er is geen optimale keuze voor het aantal verbindingen en de lengte van de verbindingen. Korte horizontale lijnen geven veel flexibiliteit, maar bevatten vaak veel knooppunten en leveren extra tijdvertraging op. Bij lange horizontale lijnen zijn veel lijnen nodig om alle verbindingen mogelijk te maken. Voor korte verbindingen is dat inefficiënt. Er is dan relatief veel chipoppervlak nodig. Een alternatief is om flexibele lange lijnen te gebruiken met programmeerbare verbindingen. Alleen leveren de passtransistoren weer een extra tijdvertraging op.

Hoewel er veel verschillende oplossingen bestaan, zijn ruwweg de volgende verbindingen te onderscheiden:

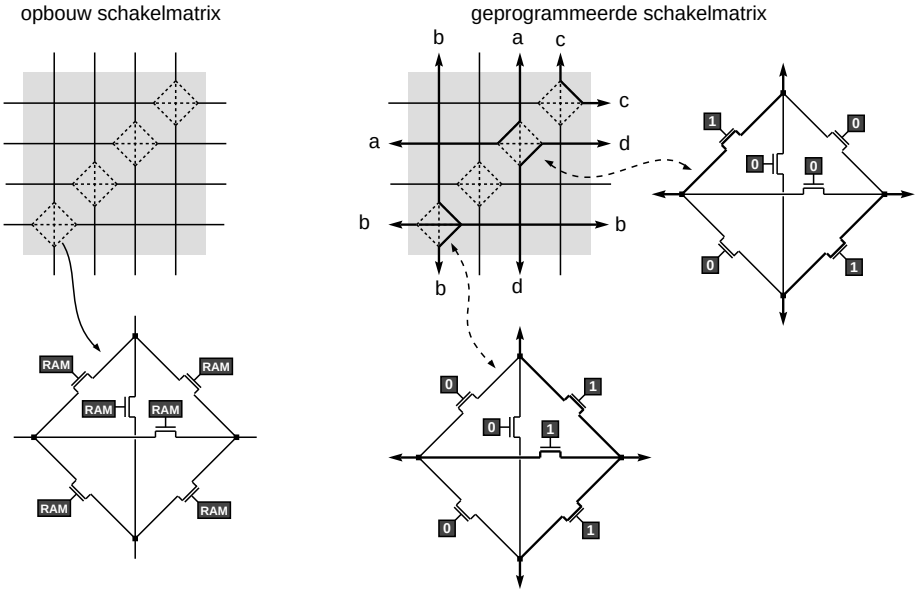
- speciale, snelle verbindingen voor de kloklijnen;
- lange lijnen;
- flexibele lange lijnen;
- directe verbindingen tussen naburige logische blokken;
- verbindingen tussen dichtbij gelegen logische blokken

Meestal wordt, net als in figuur 11.71, een combinatie van lijnlengtes gebruikt. Iedere familie FPGA's kent hiervoor zijn eigen specifieke oplossing.

Figuur 11.72 geeft een detail van een FPGA met een eilandstructuur, volgens het concept dat toegepast wordt door Xilinx. Tussen de logische blokken bevinden zich blokken met programmeerbare verbindingen. De logische blokken worden

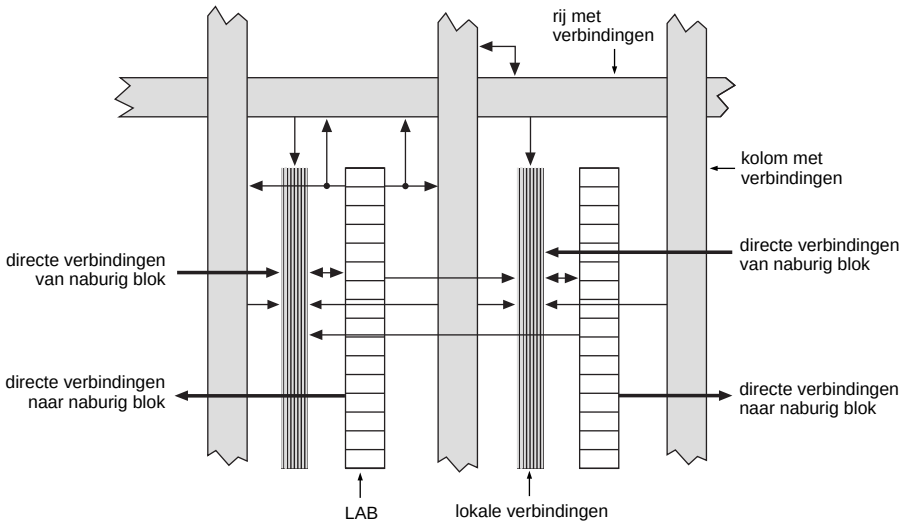


Figuur 11.72 : **Een detail van een FPGA met een eilandstructuur.** Tussen de logische blokken bevinden zich de blokken met de interconnecties en op kruisingen van de horizontale en verticale verbindingen liggen de schakelmatrices.



Figuur 11.73 : **De schakelmatrix.** Links staat de opbouw van een schakelmatrix en de uitvergroting van een programmeerbaar knooppunt. Deze bestaat uit zes passtransistoren met een SRAM-cel. Vier liggen er in een ruit en twee liggen langs de diagonalen. Rechts staat een geprogrammeerde schakelmatrix met de uitvergrotingen van twee knooppunten.

met behulp van de programmeerbare verbindingen met de verticale en horizontale lijnen verbonden. Op de kruisingen tussen de horizontale en verticale banen liggen de schakelmatrices.
Met een schakelmatrix, *switch matrix*, kunnen de verticale en horizontale lijnen met verschillende andere horizontale of verticale lijnen verbonden worden. In figuur 11.73 is een implementatie van een schakelmatrix getekend. In dit geval zijn er vier horizontale en vier verticale lijnen. Bij vier kruispunten bevindt zich een programmeerbaar knooppunt.



Figuur 11.74 : Een detail van een FPGA uit de Cyclone-III familie van Altera.

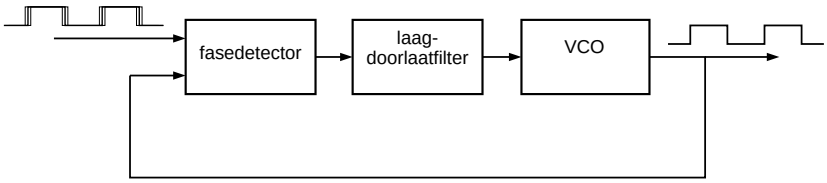
In figuur 11.74 staat een detail van een Cyclone-III FPGA van Altera. De logische blokken zijn gegroepeerd in een LAB, *logic array block*. Bij iedere LAB hoort een interconnectieblok voor de lokale verbindingen met de logische blokken. Tussen de LAB's bevinden zich lange verticale en horizontale verbindingen. Verder heeft ieder LAB directe verbindingen van en naar de naburige LAB's.

Kloklijnen en klokdomeinen

Een digitaal systeem is normaal gesproken synchroon. De hele FPGA functioneert met een klok. Alle D-flipfloppen veranderen bij dezelfde klokflank.
Omdat een FPGA zoveel flipfloppen bevat en omdat de frequenties relatief hoog zijn, kunnen er problemen rond het tijdsgedrag ontstaan. De klokflank is dan niet bij iedere flipflop op hetzelfde moment actief. Dit noemt men *clock skew*. Om hier geen last van te hebben heeft een FPGA speciale globale lijnen voor de kloksignalen en voor andere belangrijke globale signalen. Met deze globale verbindingen is het mogelijk de clock-skew klein te houden, zodat de flipfloppen op ongeveer hetzelfde moment veranderen.
De kloksignalen moeten altijd op één van de speciale klokpinnen van de FPGA worden aangesloten. Via deze pinnen zijn de speciale kloklijnen direct bereikbaar. Een FPGA heeft altijd meerdere klokpinnen. Een ontwerp mag daarom opgebouwd zijn uit meerdere domeinen met ieder een eigen kloksignaal. De signalen tussen de verschillende klokdomeinen moeten gesynchroniseerd worden.

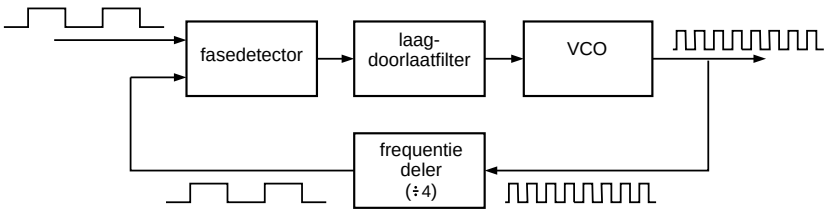
De ontwerpsoftware herkent de kloksignalen in de VHDL-code en zal, mits er geen andere restricties zijn, deze signalen automatisch toewijzen aan de beschikbare klokpinnen.

Met een PLL, *phase-locked loop*, of DLL, *delay-locked loop* kan de clock-skew nog verder geminimaliseeerd worden. De FPGA's van Xilinx gebruiken DLL's en die van Altera PLL's. De PLL en de DLL zijn beide teruggekoppelde regelsystemen. Met de terugkoppeling kan een stabiele klok met weinig clock-skew worden gemaakt.



Figuur 11.75 : **Het basisprincipe van een PLL.** De lus met de fasedetector, het laagdoorlaatfilter en de VCO stelt zich in op een stabiele vaste frequentie, die overeenkomt met de frequentie van het ingangssignaal.

Figuur 11.75 laat zien dat een PLL bestaat uit een fasedetector, een laagdoorlaatfilter en een VCO, *voltage controlled oscillator* of spanningsgestuurde oscillator. Het terugkoppelsignaal is de uitgang van de VCO, die bestuurd wordt door de uitgang van de fasedetector. Als het faseverschil tussen het ingangssignaal en het teruggekoppelde signaal nul is, is het regelsysteem op slot, *locked*. De uitgang van de VCO heeft de frequentie van het ingangssignaal. Als er een faseverschil optreedt, doordat de fase of de frequentie van het ingangssignaal kleine variaties heeft, compenseert het systeem deze afwijking. Het effect is een zeer stabiel uitgangssignaal. Een kloksignaal dat via een PLL en de speciale kloklijnen naar de flipfloppen gaat, vertoont nauwelijks clock-skew en heeft weinig last van temperatuur en spanningsvariaties.



Figuur 11.76 : **De PLL als frequentievermenigvuldiger.** De PLL zorgt ervoor dat het uitgangssignaal van de frequentiedeler dezelfde fase en frequentie heeft als het ingangssignaal. In dit voorbeeld is de deelfactor factor 4. Daarom is hier de uitgangsfrequentie vier keer zo groot als de frequentie van het ingangssignaal.

Een andere belangrijke toepassing van een PLL bij FPGA's is frequentievermenigvuldiging. Voor de communicatie met moderne, externe componenten — zoals de DDR2 SDRAM, DDR SDRAM en QDRII SRAM — zijn relatief hoge kloksnelheden noodzakelijk. De snelheden zijn vaak hoger dan de kloksnelheid, die op de FPGA haalbaar is. Het ontwerp wordt dan opgedeeld in verschillende klokdomeinen. De kern van de FPGA is een domein dat functioneert met de systeemklok en de normale taken verricht. Voor de interfacing met de snelle, externe componenten zijn aparte klokdomeinen nodig met de benodigde frequentie. Deze — in

het algemeen — hoge klokfrequentie wordt met een PLL gecreëerd afgeleid uit de systeemklok volgens het principe van figuur 11.76 De besturings- en datasignalen tussen het kerndomein en de domeinen van de interfaces moeten gesynchroniseerd worden.

In- en uitgangen van een FPGA

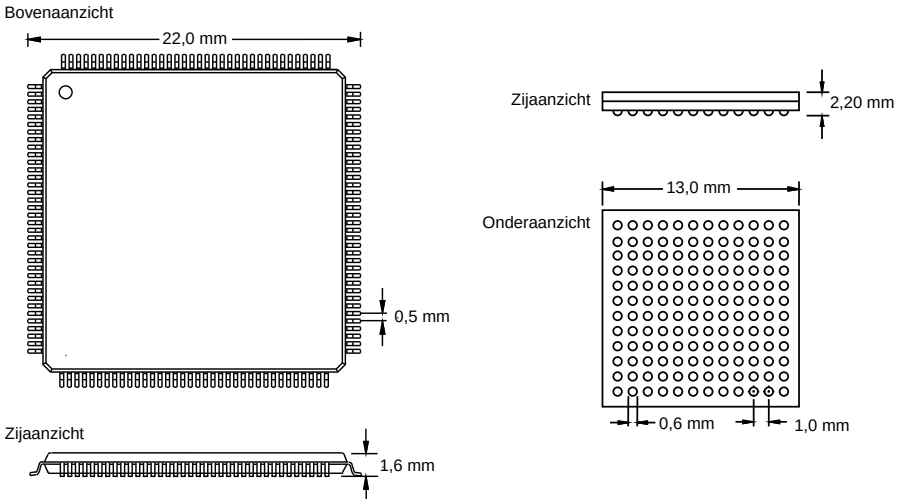
Een FPGA is een component met heel veel aansluitpinnen. Tabel 11.4 geeft voor de Cyclone-III van Altera en de Spartan-6 van Xilinx de behuizing en het aantal aansluitingen voor de kleinste en de grootste component.

Tabel 11.4 : Het aantal pinnen voor een paar FPGA's.

familie	component	behuizing	aantal pinnen	bruikbare pinnen[1]
Cyclone-III	EP3C5	TQFP	144	94 (22)
Cyclone-III	EP3C780	BGA	780	531 (233)
Spartan-6	XC6SLX4	TQFP	144	102 (51)
Spartan-6	XC6SLX150	BGA	900	576 (288)

[1] De getallen tussen haakjes geven het aantal differentiële ingangen dat beschikbaar is.

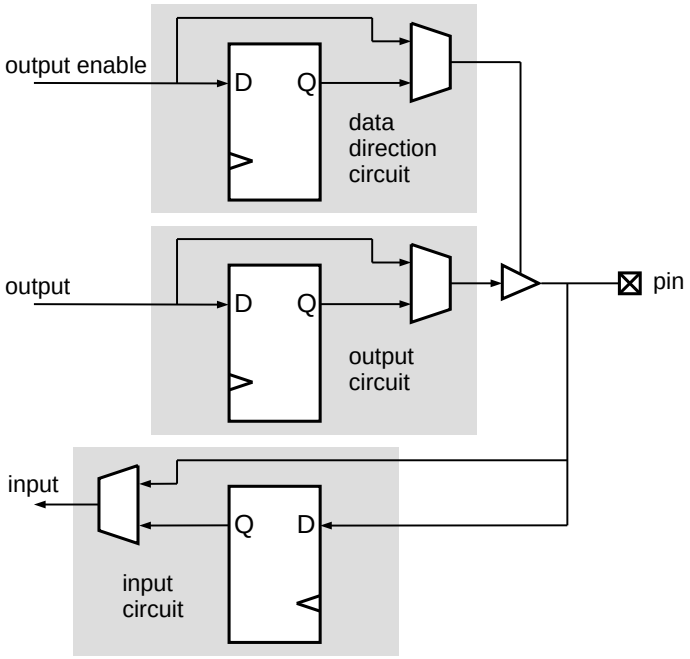
De niet bruikbare pinnen zijn de aansluitingen waarmee de FPGA geconfigureerd wordt en de aansluitingen voor de voeding. De EP3C5 heeft naast de 94 bruikbare pinnen, 24 aansluitingen voor de voedingsspanning (VCC), 14 aansluitingen voor de referentie (GND), en 12 speciale aansluitingen voor het configureren.



Figuur 11.77 : Een 144-pins TQFP-behuizing (links) en een 144-pins BGA-behuizing (rechts). In werkelijkheid zijn de behuizingen twee keer zo klein als hier getekend is.

De kleinste FPGA's zijn beschikbaar in een TQFP-behuizing en de grootste zijn alleen verkrijgbaar in een BGA-behuizing. TQFP staat voor *thin quad flat pack* en BGA staat voor *ball grid array*. De grootste Stratix FPGA's van Altera hebben zelfs een BGA met 1932 aansluitingen. In figuur 11.77 staat een 144-pins TQFP- en een 144-pins BGA-behuizing. De BGA is bijna een factor 1,7 kleiner dan TQFP. Omdat de aansluitingen bij een BGA in een twee dimensionaal vlak liggen, zal dat verschil bij meer aansluitingen alleen maar groter zijn. Grote FPGA's zijn daarom alleen verkrijgbaar in een BGA-behuizing.

De aansluitingen, die beschikbaar zijn voor het ontwerp, zijn: de gewone, generieke in- en uitgangen, de speciale kloklijnen en de aansluitingen voor speciale functies. Zowel Altera als Xilinx heeft FPGA's met speciale transceivers voor bijvoorbeeld Ethernet-verbindingen.

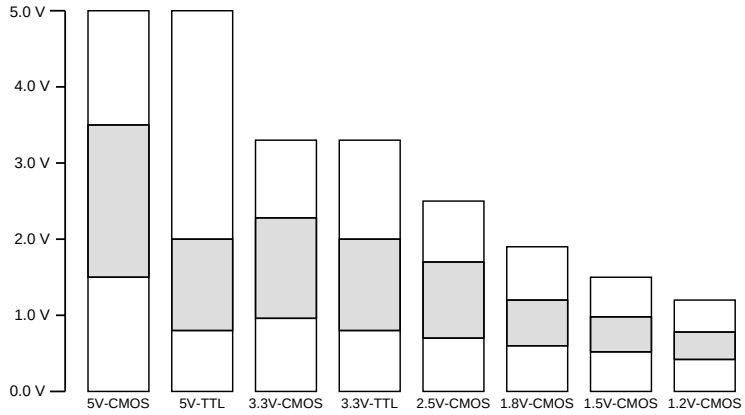


Figuur 11.78 : **Een generiek in- en uitgangsblok.** Als de flipflop van het *data direction circuit* hoog is, functioneert het blok als uitgang, anders is het een ingang.

De normale aansluitingen kunnen geprogrammeerd worden als ingang, uitgang en als bidirectionele in- en uitgang. Het basisconcept van de in- en uitgangen wijkt niet af van bijvoorbeeld de generieke aansluitingen bij microcontrollers. In figuur 11.78 staat het algemeen schema voor een in- en uitgangsblok. In het algemeen bevat dit blok drie flipfloppen: één voor het uitgangssignaal, één voor het ingangssignaal en één voor de selectie van de richting van de gegevensstroom. Als deze laatste flipflop hoog is, functioneert het blok als uitgangsblok en als het laag is, functioneert het als ingangsblok.

In figuur 11.78 zijn allerlei details niet getekend. De in- en uitgangen zijn geschikt voor verschillende interfaces. Onder ander beschikken de in- en uitgangsblokken over programmeerbare elektrische aanpassingen, zoals pullup-weerstanden en weerstanden om transmissielijnen op de juiste wijze af te sluiten.

Bovendien zijn de in- en uitgangen geschikt voor verschillende signaalniveaus. De eerste FPGA van Xilinx, de XC2064, werkte gewoon met 5 V. Omdat de dynamische vermogensdissipatie kwadratisch toeneemt met de voedingsspanning, wordt de voedingsspanning steeds lager. Begin jaren negentig bracht Xilinx een 3,3 V versie van de XC3000-familie op de markt. Tegenwoordig werkt de *core* — de binnenkant van de FPGA — bij eenvoudige FPGA's met 1,2 V en bij high performance FPGA's is dat 1,0 V of lager.
Figuur 11.79 geeft de signaalniveaus voor de aansluitingen met TTL- en CMOS-componenten. De signaalniveaus van de in- en uitgangen zijn standaard vaak 3.3V-



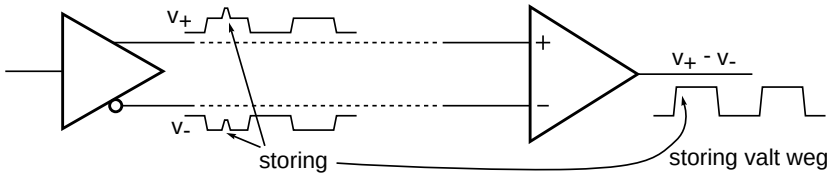
Figuur 11.79 : **De signaalniveaus voor verschillende technologieën.** Het witte gebied boven een grijs gebied definieert een hoog signaal en het witte gebied onder een grijs gebied definieert een hoog signaal.

CMOS. Voor snelle verbindingen zijn de lagere niveaus belangrijk. Voor de interfacing bestaan verschillende standaarden. Tabel 11.5 noemt een aantal standaarden. Vaak bestaan er meerdere varianten voor de verschillende signaalniveaus, zoals: SSTL2 en SSTL1.8 voor 2,5 V en 1,8 V en HSTL1.8, HSTL1.5 en HSTL1.2 voor respectievelijk 1,8, V1,5 V en 1,2 V.

Tabel 11.5 : Standaarden voor de interfacing.

standaard	soort	omschrijving
LVTTL	enkel	low voltage TTL
LVCMOS	enkel	low voltage CMOS
PCI	enkel	peripheral component interconnect
SSTL	enkel/differentieel	stub series terminated logic
HSTL	enkel/differentieel	high speed transceiver logic
LVDS	differentieel	low voltage differential signaling

Veel snelle verbindingen zijn differentieel. Dit betekent dat er voor een signaal twee verbindingen gebruikt worden. en dat de informatie verstopt zit in het verschil van deze twee signalen. In tabel 11.4 staat bij de genoemde FPGA's ook het aantal differentiële aansluitingen.



Figuur 11.80 : **Een differentiële verbinding.** De bron levert het geïnverteerde v_- en het niet-geïnverteerde signaal v_+. De ontvanger bepaalt het verschilsignaal $v_+ - v_-$. De storing op de twee verbindingen valt bij het verschilsignaal weg.

Differentiële verbindingen zijn veel minder gevoelig voor storing en worden veel gebruikt. De bij consumentenproducten veel gebruikte USB-verbinding is een

ander voorbeeld van een differentiële verbinding. Figuur 11.80 laat zien dat de bron de signalen geïnverteerd v_- en niet-geïnverteerd v_+ aanbiedt. De ontvanger herleidt hieruit het verschilsignaal $v_+ - v_-$. Storingen zullen vaak bij beide signalen optreden. Doordat de ontvanger naar het verschilsignaal kijkt, worden deze fouten er uitgefilterd.

FPGA's zijn verkrijgbaar met verschillende *speed grades*. Niet alle snelle verbindingen zijn beschikbaar bij iedere *speed grade*. Bij de productie van FPGA zijn niet alle wafers gelijk en de FPGA's van een wafer zijn onderling ook niet helemaal gelijk. Een FPGA uit het midden van een wafer voldoet beter aan de specificatie dan een FPGA van de rand.

Een wafer is de schijf silicium waarop de geïntegreerde schakelingen worden gemaakt. Een wafer heeft typisch een doorsnede van enkele tientallen centimeters en bevat vele honderden IC's.

Bij de productie worden de FPGA's gemeten en verdeeld in snellere en langzamere componenten. Bij Altera is de *speed grade* een cijfer dat lager is voor de snellere FPGA's en bij Xilinx is het een cijfer dat juist hoger is bij snellere FPGA's.

Gespecialiseerde onderdelen van een FPGA

Microcontrollers bestaan in vele soorten en maten, met allerlei gespecialiseerde onderdelen, zoals ADC's, timers voor PWM en seriële verbindingen als I^2C, SPI, CAN en USB.

Voor FPGA's zijn er veel minder mogelijkheden. Er zijn geen extra gespecialiseerde digitale onderdelen voor timers en seriële verbindingen. Deze componenten worden direct met de logica van FPGA gerealiseerd. Er bestaan ook bijna geen FPGA's met een ADC of DAC. Analoge conversies worden met externe componenten gedaan.

Een FPGA kan wel één of meer van deze extra functionaliteiten hebben:

Tabel 11.6 : De afmetingen van het M4K RAM-blok van Altera.

diepte	breedte
4096	1
2048	2
1024	4
512	8
512	9
256	16
256	18
128	32
128	36

Tabel 11.7 : De afmetingen van het M9K RAM-blok van Altera.

diepte	breedte
8192	1
4096	2
2048	4
1024	8
1024	9
512	16
512	18
256	32
256	36

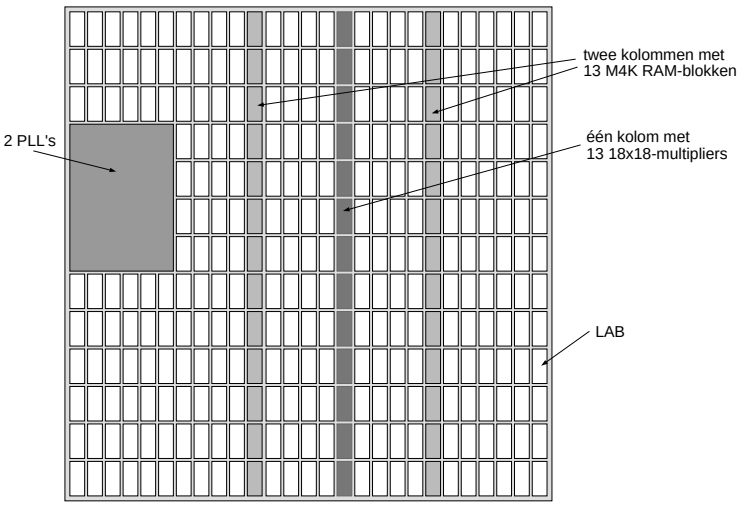
- **RAM-blokken**
  Ondanks dat een FPGA meestal uit SRAM bestaat en de flipfloppen ook voor een generiek geheugen gebruikt kunnen worden, heeft een FPGA extra RAM-blokken. Deze worden gebruikt voor allerlei geheugens, zoals een stack, een lifo en een fifo.
  Een belangrijke toepassing is die van een fifo om data te synchroniseren, zoals op bladzijde 300 is uitgelegd. De RAM-blokken zijn programmeerbaar. De diepte en de breedte van het RAM is configureerbaar. In tabel 11.6 staan de mogelijke configuraties van de 4608 bits van het M4K RAM-blok van Altera. Tabel 11.7 geeft die voor de 9216 bits van het M9K RAM-blok. Deze RAM-blokken worden gebruikt als single-port-, simple-dual-port-, true-dual-port RAM, maar ook als schuifregister, ROM en fifo.
- **Vermenigvuldigers en andere rekenkundige bewerkingen**
  FPGA's zijn bijzonder geschikt voor digitale signaalbewerkingen. Omdat daar veel rekenkracht voor nodig is, is het zinvol om extra rekenfaciliteiten toe te voegen. De logische blokken van FPGA's hebben extra logica voor snelle carry-signalen en soms ook snelle optellers. Een FPGA bevat ook extra logica voor vermenigvuldigers of *multipliers*. De vermenigvuldiger van een eenvoudige FPGA is meestal een 18×18-bits vermenigvuldiger, die ook te gebruiken is als twee afzonderlijke 9×9-bits vermenigvuldigers.
- **DSP-blokken**
  De grotere FPGA's bevatten DSP-blokken, *digital signal processor blocks* met meerdere vermenigvuldigers, optellers en registers. Deze blokken zijn

geschikt voor het maken van complexe digitale bewerkingen, zoals FFT, *fast fourier transform* en voor digitale filters, zoals FIR, *finite impulse response* en IIR, *infinite impulse response*.

- **Communicatieblokken**

  Voor moderne communicatieprotocollen zijn snelle seriële transceivers nodig. De grotere en duurdere *high end* FPGA's zijn leverbaar met on-chip transceivers. Er zijn belangrijke voordelen om in de FPGA geïntegreerde transceivers te gebruiken. Het PCB-ontwerp wordt kleiner en eenvoudiger. Bovendien zijn er minder problemen met het tijdsgedrag.

Figuur 11.81 geeft de architectuur van de kleinste *low end* Cyclone-II FPGA van Altera. Deze EP2C5 heeft twee kolommen met dertien M4K RAM-blokken en een kolom met dertien 18×18 vermenigvuldigers. In het totaal bevat de component 26 RAM-blokken van 4608-bits. Bij elkaar betekent dat 119808 bits RAM.



Figuur 11.81 : De architectuur van de Cyclone-II.

Naast de vermenigvuldigers en de RAM-blokken bevat de core van deze FPGA ook twee PLL's, die de plaats innemen van 24 LAB's. Er zijn 13 rijen en 24 kolommen met LAB's. Totaal betekent dit 288 LAB's met ieder 16 logische blokken. Bij elkaar heeft deze FPGA dus 4608 logische blokken.

Voor veel digitale oplossingen is — naast de RAM-blokken, de vermenigvuldigers en de speciale snelle in- en uitgangen — geen extra digitale hardware nodig. Veel bijzondere functionaliteiten kunnen met de logica van de FPGA gemaakt worden. Bij de ontwerpsoftware van de fabrikanten zitten altijd programma's om speciale blokken te genereren. Bij Xilinx is dat CoreGen en bij Altera is dat de MegaWizard. Deze programma's worden gebruikt om bijvoorbeeld een UART, een I^2C-interface, een dual-port RAM-geheugen te genereren. De ontwikkelomgeving van Altera kent ook de SOPC-builder, waarmee een compleet systeem kan worden gebouwd. SOPC staat voor *system on a programmable chip*. Hiermee kan bijvoorbeeld een microprocessorsysteem gegenereerd worden voor de FPGA.

Toekomst FPGA's

Deze paragraaf over FPGA's beschrijft voornamelijk de op SRAM-gebaseerde FPGA's van Xilinx en Altera. Concurrenten gebruiken ook andere technieken. Actel levert FPGA's op basis van antifuse en van flash. Het gebruik van flash voor FPGA's is een nieuwe ontwikkeling. Ook Lattice biedt naast SRAM een nieuwe FPGA-familie aan op basis van flash. Een aantal aspecten is in deze en in de voorgaande paragrafen al besproken. Het overzicht van tabel 11.8 completeert de belangrijkste verschillen en overeenkomsten.

Tabel 11.8 : Overzicht FPGA-technologieën.

familie	SRAM	antifuse	flash
herprogrammeerbaar	ja	nee	ja
vluchtig	ja	nee	nee
extern geheugen nodig	ja	nee	nee
werkt direct	nee	ja	ja
vermogensverbruik	gemiddeld	laag	gemiddeld
grootte basiselememt	groot	klein	gemiddeld
aantal basiselememten	hoog	laag	gemiddeld
in systeem programmeerbaar	ja	nee	ja
productieproces	eenvoudig	complex	gemiddeld
productbescherming	acceptabel	goed	goed
stralingsgevoeligheid	gemiddeld	laag	hoog
testbaarheid	goed	slecht	gemiddeld

Het zwakke punt van de op SRAM-gebaseerde FPGA's blijft de vluchtigheid. Als de spanning wegvalt, moet de FPGA opnieuw geprogrammeerd worden en is er een extra geheugen nodig voor de configuratiebits. Dit levert ook een probleem op rond de productbescherming. De configuratiebits kunnen bij het opstarten ook door anderen uitgelezen worden en gebruikt worden om een illegale kopie te maken. Xilinx en Altera gebruiken encryptietechnieken om dit probleem te voorkomen. Wel wordt hiervoor een deel van de logica gebruikt en zijn er licentiekosten aan verbonden.

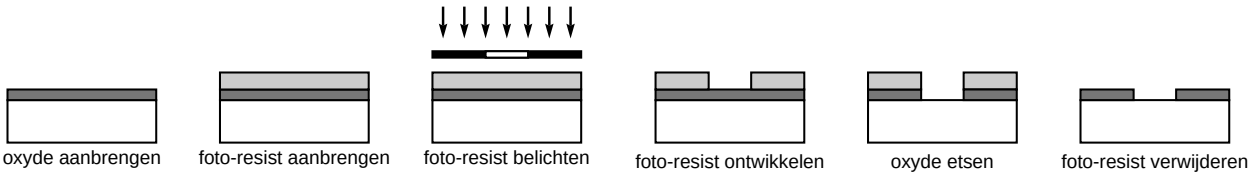
De belangrijkste reden dat de op SRAM-gebaseerde FPGA's van Altera en Xilinx zo succesvol zijn, is dat deze FPGA's uit gewone CMOS-transistoren zijn opgebouwd. Het sterke punt is dat de productie relatief goedkoop is en dat de technologie altijd de meest moderne is met de kleinste transistorafmetingen. Feitelijk lopen op SRAM-gebaseerde FPGA's steeds een paar generaties voor op de antifuse- en flash-gebaseerde FPGA's. De komende jaren zullen Xilinx en Altera hun grote marktaandeel vasthouden.

11.8 ASIC

CMOS is de belangrijkste technologie voor het maken van geïntegreerde digitale schakelingen. Voor het begrijpen en classificeren van full- en semicustom-IC's is het noodzakelijk een beeld te hebben van de fabricage van CMOS-schakelingen. Deze paragraaf begint met een ruwe uitleg van het productieproces en de fysieke bouw van CMOS-schakelingen. De CMOS-technologie is al eerder in paragraaf 11.1 geïntroduceerd.

Procestechnologie

Fysiek is een CMOS-schakeling opgebouwd uit verschillende lagen of *layers*. De IC-ontwerper legt de patronen die deze lagen moeten krijgen vast. De totale beschrijving van deze patronen noemt men de layout. Met behulp van fotolithografische processen worden deze patronen op het silicium overgebracht.



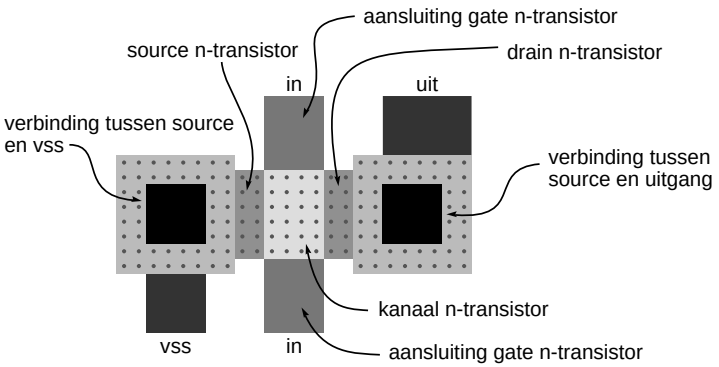
Figuur 11.82 : Het foto-lithografisch proces. De processtappen die nodig zijn om een laag op de wafer aan te brengen. In dit voorbeeld is dit de eerste oxidelaag.

Een wafer is de schijf silicium waarop de geïntegreerde schakelingen worden aangebracht.

Figuur 11.82 toont het fotolithografische proces om een laag op een wafer aan te brengen. Eerst wordt een egale laag materiaal aangebracht. Dat kan door opdampen of via een chemische reactie als oxideren. Daarbovenop wordt een fotogevoelige laag — het foto-resist — aangebracht. Via een fotografisch masker met het gewenste patroon wordt de fotogevoelige laag belicht. De onbedekte delen van het materiaal worden weggeëtst en tenslotte wordt het foto-resist verwijderd.

Er bestaat positief en negatief foto-resist. Positief foto-resist lost op onder invloed van licht. Negatief foto-resist wordt juist hard onder invloed van licht.

De layout van een CMOS-inverter staat in figuur 11.84 en is opgebouwd uit een groot aantal rechthoeken. Ieder rechthoek is een deel van het patroon van een fotografisch masker of van een combinatie van fotografische maskers. In dit voorbeeld worden zeven maskers gebruikt. Het patroon van ieder masker is in figuur 11.84 ook afzonderlijk afgebeeld. De layout is in feite de som van de zeven maskers.

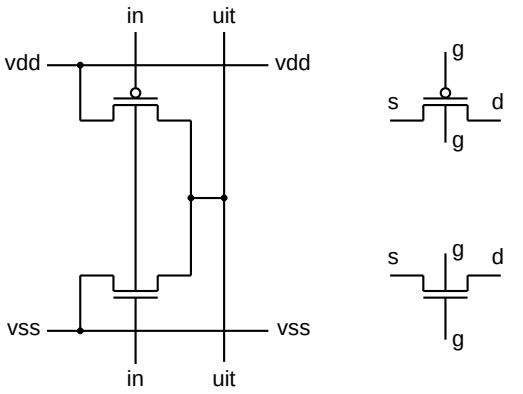


Figuur 11.83 : De functionele betekenis van de layout.

Figuur 11.83 toont het deel met de n-transistor uit de layout en geeft de functionaliteiten van de verschillende lagen. De overeenkomst tussen dit fragment en het betreffende deel van het schema van de CMOS-inverter uit figuur 11.84 is goed te zien.

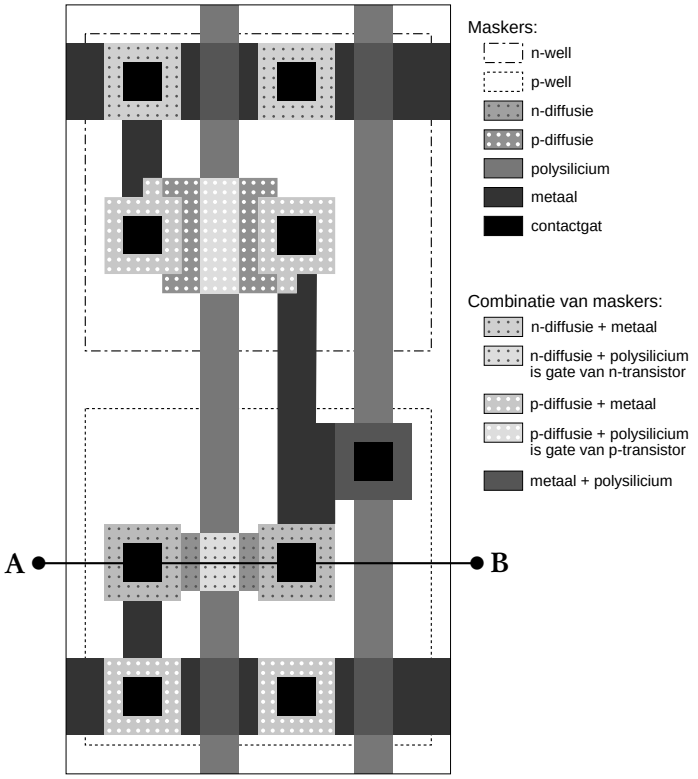
In de ASIC-wereld, waar CMOS de standaard is, spreekt men niet over NMOS- en PMOS-transistoren, maar over n- en p-transistoren.

In figuur 11.84 is de p-transistor twee zo breed als de n-transistor. Bij de p-transistor vindt de geleiding plaats door gaten in plaats van elektronen. Omdat gatengeleiding twee tot drie keer zo traag is als elektronengeleiding, is de breedte van de p-transistor twee zo groot gemaakt. De weerstand is dan twee zo klein en de snelheid waarmee de inverter wordt opgeladen komt dan ongeveer overeen met de snelheid van het ontladen.

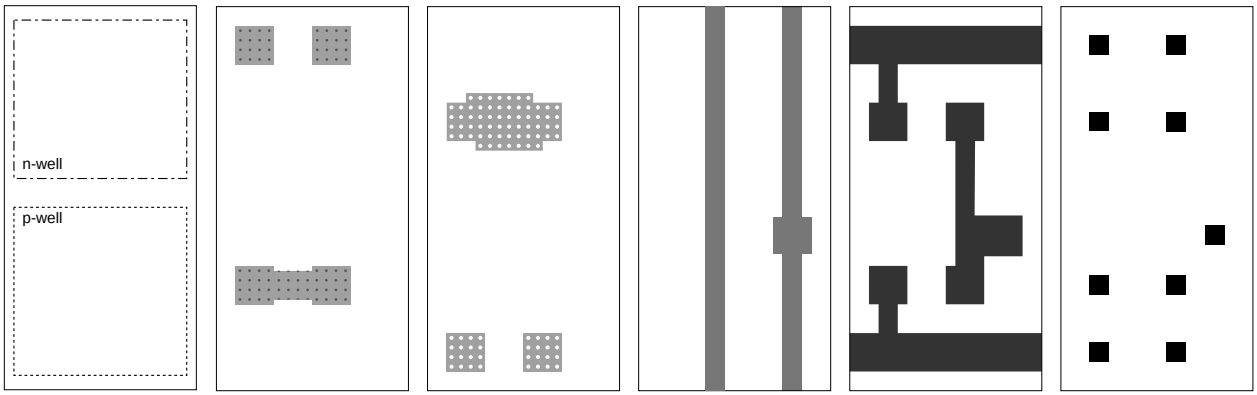


(a) inverterschakeling

(b) transistorsymbolen



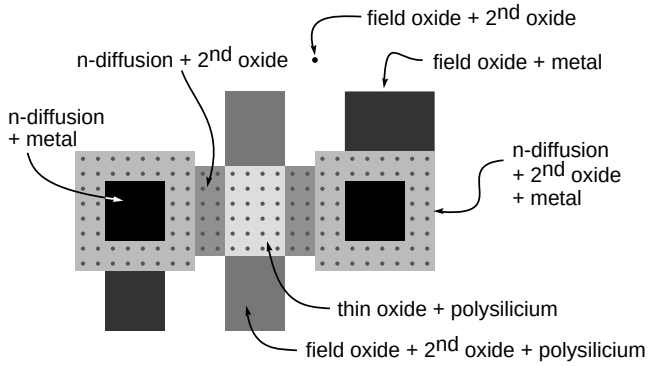
(c) layout van inverter met legenda



p-well en n-well | n-diffusie | p-diffusie | polysilicium | metaal | contactgaten

Figuur 11.84 : **De layout van en de benodigde fotografische maskers voor een CMOS-inverter.** Bovenaan (a) staat het schema van de inverter. De transistoren (b) hebben twee aansluitingen voor de gate. Dit past beter bij de fysieke realisatie van de inverter.
Rechts (c) staat de layout van de inverter. De doorsnede langs de lijn tussen de punten A en B staat in figuur 11.86. Naast de layout van de inverter staat de legenda. Behalve de zeven maskers geeft de legenda ook de combinatie van de verschillende maskers.
Onderaan zijn de zeven maskers apart getekend. De maskers voor de n-well en p-well zijn hier in een figuur gecombineerd.

De aanwezigheid van een masker betekent niet dat op die plaats ook het betreffende materiaal aanwezig is. Op de plaatsen waar het polysiliciummasker over het n-diffusie- of het p-diffusie-masker heen ligt, komen juist geen diffusiegebieden. Op deze plaatsen bevinden zich juist de gates van de transistoren.



Figuur 11.85 : De fysieke lagen voor de layout.

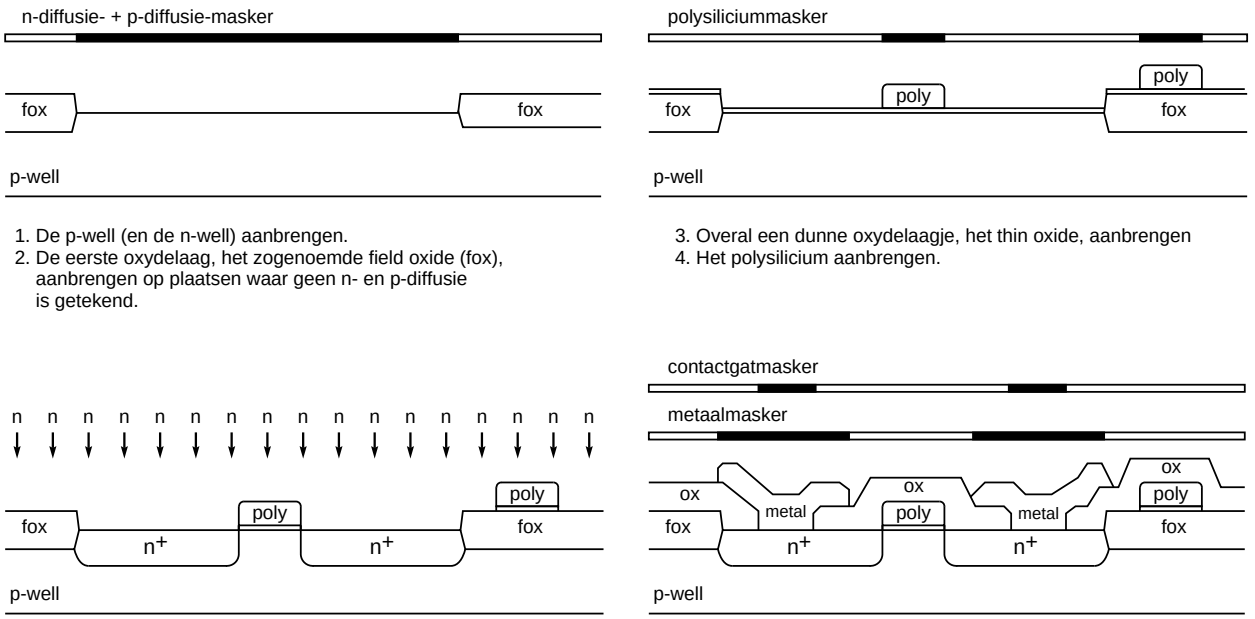
Figuur 11.85 is een kopie van het detail van figuur 11.83 met de namen van de lagen waaruit dit deel van de inverter bestaat. Tussen de verschillende geleiders worden isolerende oxidelagen aangebracht. De eerst oxidelaag is het zogenoemde *field oxide*. Deze laag ligt overal waar geen diffusie is aangebracht. Het is de isolatie tussen het substraat — in dit geval is dat de p-well – en het polysilicium. De tweede dikke oxidelaag is de isolatie tussen de eerste metaallaag en het polysilicium en de isolatie tussen de eerste metaallaag en de diffusie.

In figuur 11.84 zijn ook speciale verbindingen aangebracht tussen de voedingslijnen en de onderliggende n-well of p-well. Deze well-contacten zorgen ervoor dat de spanning van de n-well hoog (vdd) is en de p-well laag (vss) is.

In figuur 11.86 is vier maal — bij steeds een andere fase uit het fabricageproces — de dwarsdoorsnede van de inverter langs de lijn AB uit figuur 11.84 getekend. Bij de fases staan steeds de belangrijkste stappen uit het productieproces.
Eerst worden door middel van diffusie de zwak-gedoteerde n- en p-wellgebieden op de wafer (het substraat) aangebracht. Vervolgens wordt de eerste oxidelaag — het *field oxide* (FOX) — aangebracht die een isolatie vormt tussen het substraat en de geleidende lagen daarboven.
Daarna wordt het *thin oxide* aangebracht die het oxidelaagje onder de gate moet gaan vormen. Nadat het polysilicium is aangebracht wordt het dun oxide weggeëtst. Alleen onder het polysilicium blijft dun oxide achter. Aangezien de te diffunderen atomen niet door de oxidelagen doordringen, ontstaan er alleen diffusiegebieden waar geen field oxide is en waar geen gate-kanaal is. Belangrijk is dat deze stap *selfaligning* is. De plaats van het dun oxide — dus de positie van het polysiliciummasker — doet er niet zoveel toe. De diffusiegebieden komen altijd direct naast de gate te liggen.

De n- en p-diffusie-maskers worden ten onrechte diffusiemaskers genoemd. Deze maskers leggen vast waar het field oxide *niet* ligt. Op de plaatsen waar de n- en p-diffusie getekend is, liggen de diffusiegebieden, mits er geen polysiliciummasker getekend is. Anders bevindt zich daar de gate van een transistor.

Het contactgatmasker wordt vervolgens gebruikt om de openingen in de volgende oxidelaag te definiëren. Daarna wordt het metaalmasker gebruikt om het patroon van de eerste metaallaag aan te brengen. De contactgaten maken de verbindingen tussen het metaal en de diffusiegebieden en het polysilicium dat daar onder ligt.



Figuur 11.86 : De dwarsdoorsnede van de n-transistor met de belangrijkste fabricagestappen. Het is een doorsnede langs de lijn AB uit de layout van figuur 11.84.

De voorstelling die hier van het productieproces is gegeven, is sterk vereenvoudigd en niet volledig. Iedere stap uit figuur 11.86 bestaat uit verschillende handelingen, zoals foto-resist aanbrengen, belichten, etsen en foto-resist verwijderen. Bovendien hebben moderne IC's meerdere metaallagen. Tussen metaallagen liggen dikke oxidelagen met openingen voor verticale verbindingen. Over het geheel ligt weer een dikke oxidelaag met openingen voor de *bonding wires*. Dit zijn de verbindingen tussen de chip en de pinnen van de behuizing.

Er zijn verschillende maten voor de processen. De *pitch* is de kleinste afstand tussen twee sporen. De helft daarvan is de *half-pitch* en komt vaak overeen met de kleinste spoorbreedte. λ is dan de helft van de *half-pitch*.

Begin jaren negentig was de half-pitch nog ruim een 1μm. Tegenwoordig is de half-pitch 45 nm en 28 nm. Voor een 45 nm-proces is λ 22 nm.

Bij het fabricageproces kunnen de fotografische maskers slechts met een bepaalde nauwkeurigheid worden gepositioneerd. De layout moet aan de ontwerpregels voldoen die de toleranties beschrijven. Deze ontwerpregels of *design rules* zijn onafhankelijk van de afmetingen van het proces en worden uitgedrukt in λ's. Eén λ komt overeen met de helft van de minimale spoorbreedte. Bij het kleiner worden van de spoorbreedte blijven de ontwerpregels hetzelfde. Alle afmetingen worden uitgedrukt in λ. Om te profiteren van de voortdurende ontwikkeling in de chipfabricage hoeft een ontwerp niet te worden gewijzigd. Het is voldoende om alleen λ kleiner te maken om een kleinere schakeling te krijgen.

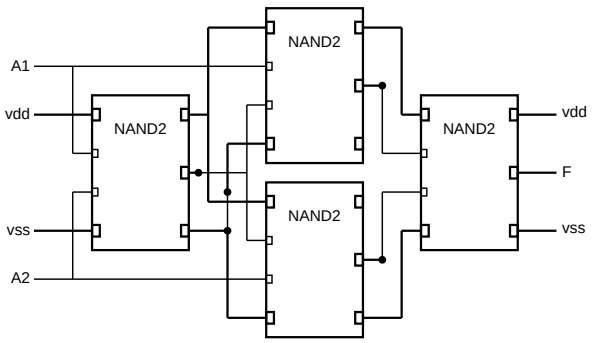
De CMOS-ontwerpstijl en het stickdiagram

Bij een groot ontwerp hoeven niet alle transistoren steeds opnieuw te worden getekend. Er bestaan bibliotheken met standaardcomponenten. De layout van een geïntegreerde schakeling zal altijd hiërarchisch zijn. Het symbool van een bibliotheekcel is een rechthoek — de *bounding box* — met de naam van de cel en de plaats van de aansluitingen.

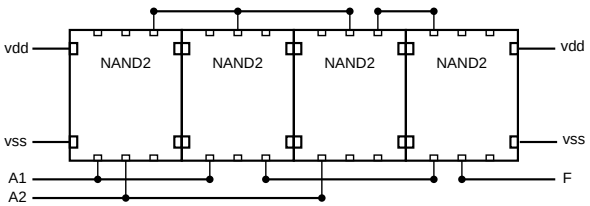
Bij het maken van een bibliotheekcel moet men rekening houden met de manier waarop de verschillende cellen verbonden worden. Behalve dat de verbindingen elektrisch correct zijn, moet er voor gezorgd worden dat:

- het aantal verbindingen minimaal is;
- de verbindingen zo kort mogelijk zijn;
- het layertype correct is;
- kruisende lijnen van een verschillend layertype zijn;
- lange lijnen in metaal worden uitgevoerd.

Een gevolg van deze eisen is dat bij een bibliotheekcel de posities van de aansluitingen heel belangrijk zijn.



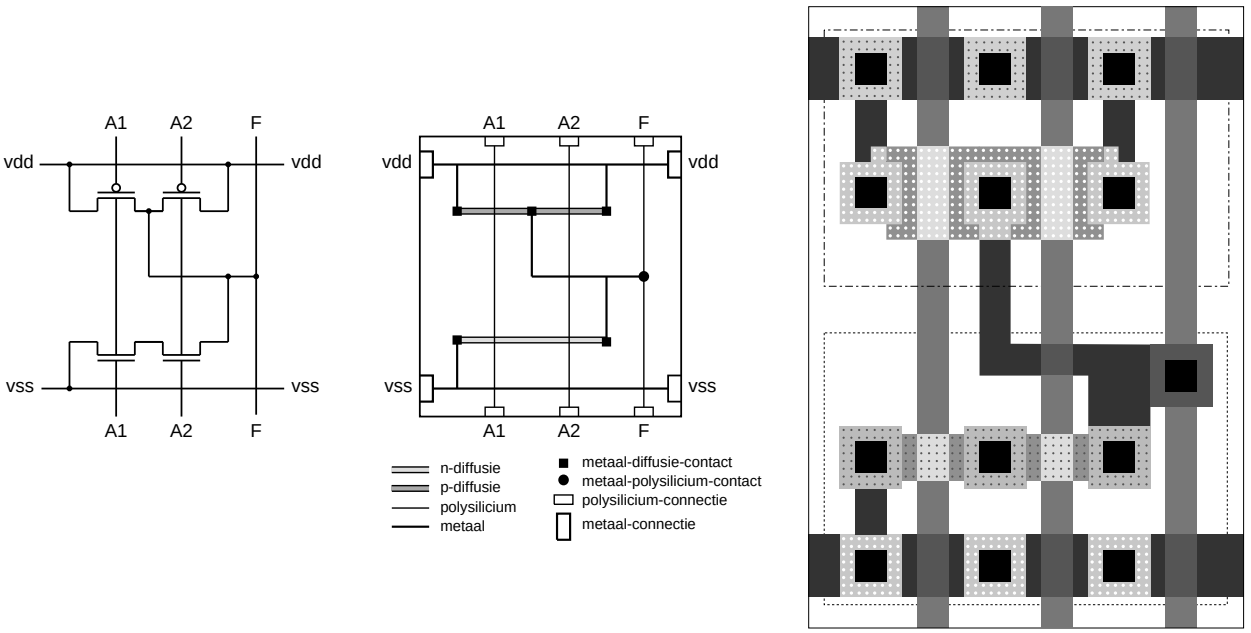
Figuur 11.87 : Een XOR met vier NAND-poorten zonder speciale ontwerpstrategie.



Figuur 11.88 : Een XOR met vier NAND-poorten met de speciale ontwerpstrategie.

In figuur 11.87 en figuur 11.88 zijn twee XOR-poorten getekend, die zijn opgebouwd uit vier NAND-poorten. De NAND uit figuur 11.87 is niet ontworpen met een speciale layoutstrategie. De ingangen zitten, net als in de schakeling van figuur 11.10, links en de uitgang rechts. De NAND uit figuur 11.88 is wel ontworpen met een speciale strategie. De voedingslijnen lopen horizontaal en de datalijnen verticaal. Bij CMOS zijn er altijd evenveel p- als n-transistoren. Het ligt voor de hand om de transistoren in elkaars verlengde te plaatsen, zoals ook bij de inverter uit figuur 11.84 is gedaan. De signaalaansluitingen komen dan automatisch aan de boven- en de onderkant. Deze strategie wordt ook bij standaardceltechnologie toegepast. De layout van de XOR met de speciale strategie is veel compacter en eenvoudiger dan de layout zonder ontwerpstrategie.

Voor het handmatig ontwerpen van een layout gebruikt men een layout-editor. Deze editor lijkt sterk op een tekenprogramma. Het resultaat van een layoutontwerp is een tekening waarin alle informatie van de maskers vastligt. Omdat er rekening met de ontwerpregels gehouden moet worden, is dit een consciëntieus werk. In plaats daarvan tekent men eerst een *stick diagram*, dat is een vereenvoudigde versie van de layout. De verbindingen zijn geen rechthoeken maar lijnen

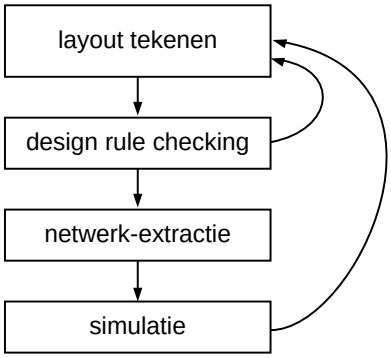


Figuur 11.89 : Het schema, het stickdiagram en de layout van een NAND met twee ingangen.

of *sticks*. Er hoeft geen rekening te worden gehouden met de ontwerpregels en het hoeft ook niet op schaal te zijn. Het stickdiagram kan automatisch omgezet worden naar een correcte layout. Figuur 11.89 geeft het schema, het stickdiagram en de bijbehorende layout van een 2-input-NAND.

Design rule checking en circuitextractie

Zoals gezegd moet een layout voldoen aan de ontwerpregels van het proces dat gebruikt wordt. Met speciale *design rule checkers* wordt gecontroleerd of het ontwerp aan de ontwerpregels voldoet.



Figuur 11.90 : Het ontwerptraject voor een fullcustom-layout.

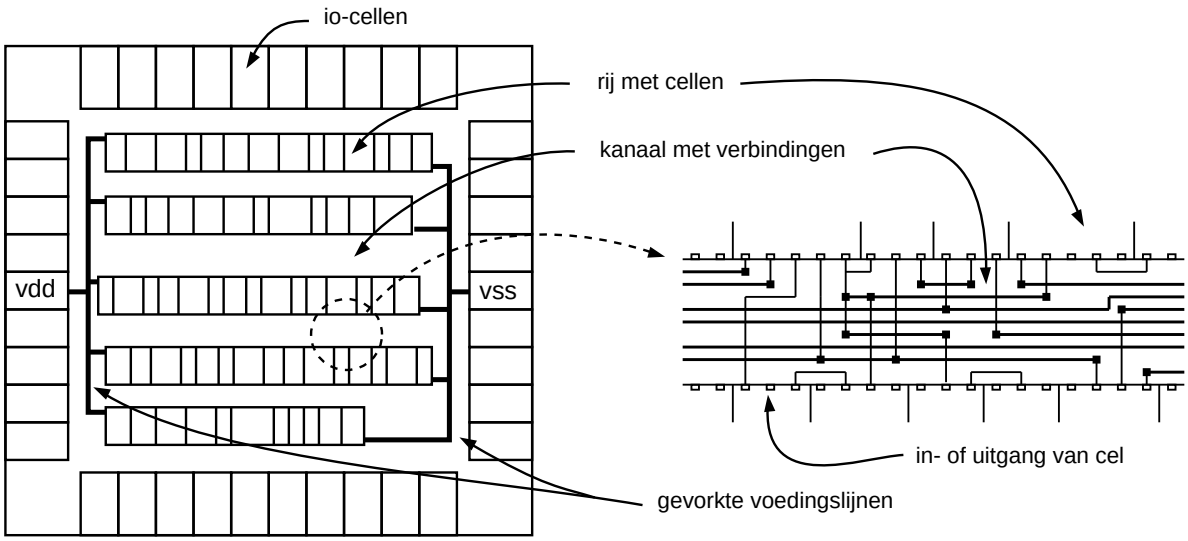
Het ontwerp moet bovendien ook aan de elektrische specificaties voldoen. Daarvoor wordt uit de layout het bijbehorende elektrische netwerk geëxtraheerd, dat vervolgens wordt geverifieerd door middel van simulatie. Figuur 11.90 geeft de stappen die achtereenvolgens moeten worden uitgevoerd bij het maken van een fullcustom-layout.

Fullcustom- en semicustom-ASIC's

Een fullcustom-IC is een geïntegreerde schakeling waarbij de ontwerper de layout helemaal zelf maakt. Mits voldaan wordt aan alle ontwerpregels heeft hij daarbij alle vrijheden. Hoewel er ook veel hulpmiddelen zijn — zoals bijvoorbeeld PLA-generatoren, waarmee grote blokken logica gemaakt kunnen worden — is het fullcustom-ontwerp een complex en arbeidsintensief werk. Daarom wordt dit alleen door specialisten gedaan.

Voor een semicustom-IC hoeft de ontwerper geen layout te tekenen. De ontwerper maakt net als bij een FPGA een VHDL-, SystemC- of Verilog-beschrijving en zet deze om naar een netwerkbeschrijving met eenvoudige logische poorten. Met een plaatsing- en bedradingsprogramma worden — net als bij een FPGA — de poorten geplaatst en met elkaar verbonden.

Er zijn verschillende soorten semicustom-IC's. De belangrijkste zijn de gate-array en het standaardcel-IC. De layout van het standaardcel-IC staat in figuur 11.91.



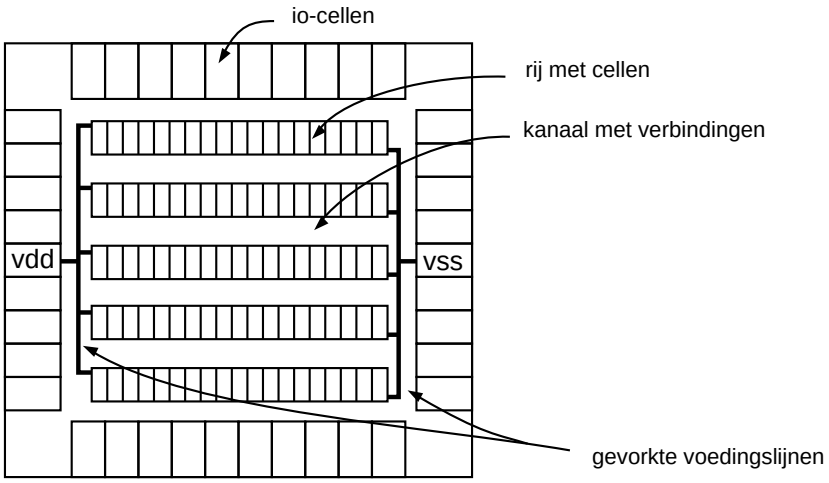
Figuur 11.91 : De layout van een standaardcel-IC. Links staat het IC en rechts een uitvergroting van een deel van een bedradingskanaal.

Bij de standaardceltechnologie hoort een bibliotheek met basiscomponenten. De ontwerpstijl van de NAND uit figuur 11.89 komt overeen met die van de componenten of cellen uit de standaardceltechnologie. De horizontale voedingslijnen hebben altijd dezelfde breedte en dezelfde afstand. De cellen kunnen, net als bij de XOR-poort van figuur 11.88, tegen elkaar aan worden geplaatst.

Het standaardcel-IC bestaat zodoende uit rijen met standaardcellen met daartussen de kanalen voor de verbindingen. Omdat bij de standaardceltechnologie alle maskers door de ontwerper gemaakt worden, kunnen transistoren overal liggen, is de breedte van de kanalen variabel en is een standaardcel-IC altijd te bedraden. Een ander voordeel is de gevorkte structuur van de voedingslijnen, zodat de voedingslijnen elkaar niet kruisen en dat ze met één metaallaag aangebracht kunnen worden.

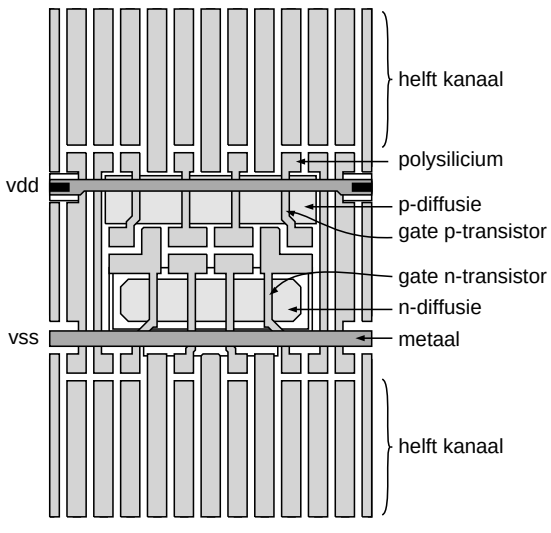
In tegenstelling tot de standaardceltechnologie liggen bij de gate-array de posities van de transistoren vooraf vast. De wafer is bijna helemaal klaar, alleen de laatste

metaallaag of metaallagen moeten nog worden aangebracht. Deze technologie is veel goedkoper, maar op een gate-array passen per oppervlakte-eenheid wel minder transistoren. Omdat het proces eenvoudiger is, is de productietijd veel korter en krijgt de ontwerper de IC's eerder terug van de IC-fabrikant.

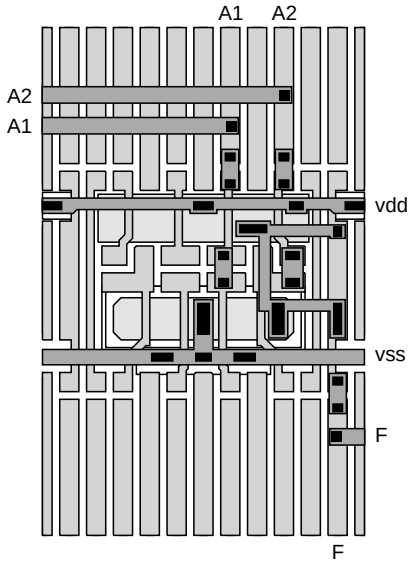


Figuur 11.92 : De layout van een gate-array.

Omdat bij een gate-array de plaats van de transistoren op het IC vooraf is vastgelegd, liggen de kanaalbreedtes vast. In figuur 11.92 zijn de kanalen allemaal even breed. Dit kan een probleem geven bij de plaatsing en bedrading. Bovendien ligt de afmeting van het IC vooraf vast. Zelfs als er een productlijn is met meerdere afmetingen zal altijd een deel van de gate-array niet gebruikt worden. Net zoals een FPGA, zal een gate-array nooit helemaal vol zijn.



Figuur 11.93 : Een lege gate-arraycel met acht transistoren.



Figuur 11.94 : Een gate-arraycel met het metaalmasker voor een 2-input-NAND.

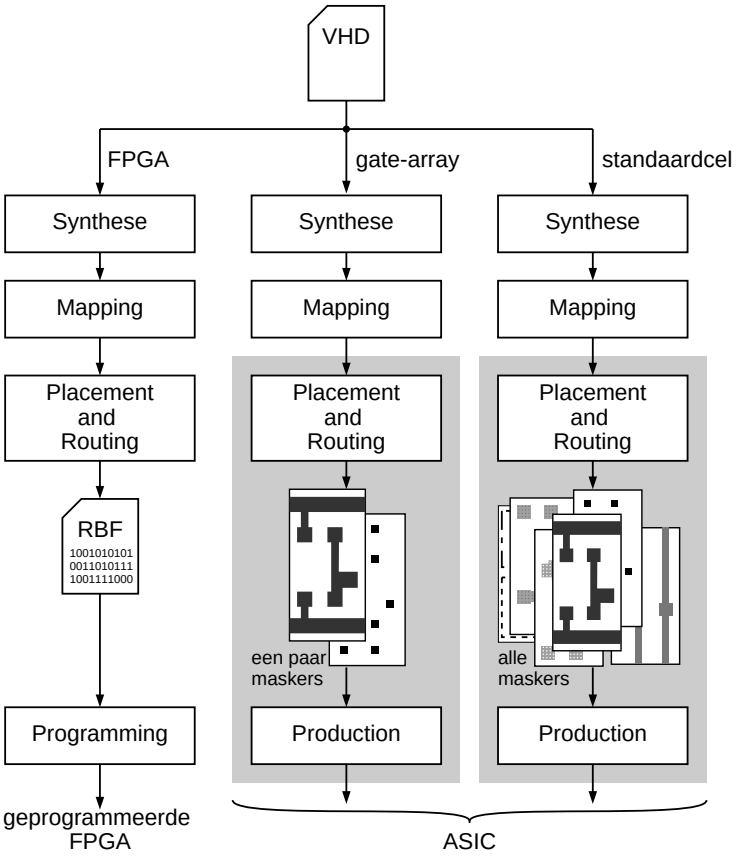
De cellen van een gate-array zijn allemaal even groot. In figuur 11.93 staat een voorbeeld van een cel met acht transistoren zonder het metaalmasker dat de functionaliteit geeft. De twee metaalsporen voor de beide voedingslijnen zijn wel getekend, omdat een lege cel in ieder geval de voedingsstroom naar de aanliggende cellen door moet geven. Overal waar een polysiliciumspoor over de n- of de p-diffusie ligt, bevindt zich de gate van een n- of p-transistor. Deze cel heeft vier n-transistoren en vier p-transistoren.

De boven- en de onderkant van de cel bevatten de helft van het kanaal. De cellen boven en onder deze cel zien er identiek uit en liggen tegen de cel aan. De gates van de transistoren zijn bij de lege cel niet verbonden. Met het metaalmasker kunnen de transistoren op verschillende manieren met elkaar gecombineerd worden en de polysilicium sporen in één van de kanaalhelften verbonden worden. Het metaalmasker wordt ook gebruikt om horizontale verbindingen door de kanalen te maken. Figuur 11.94 geeft een gate-arraycel met het metaalmasker voor een 2-input NAND.

Ontwerptraject standaardcel-IC, gate-array en FPGA

De gate-array heeft veel overeenkomsten met de FPGA. Beide bezitten een kant-en-klare structuur. In beide gevallen zijn de transistoren al aangebracht in vaste cellen of logische blokken.
De ontwerper definieert bij een gate-array en bij een FPGA de functionaliteit van de cellen of logische blokken en definieert de verbindingen tussen deze cellen of logische blokken. Bij de gate-array brengt de IC-fabriek deze functies en verbindingen aan. De informatie hiervoor ligt vast in de patronen op de maskers. Bij de FPGA brengt de ontwerper zelf de functies en verbindingen aan met behulp van een programmer en programmeersoftware. De informatie voor het ontwerp bevindt zich in het configuratiebestand.
Vanwege de overeenkomsten met de FPGA, de *field programmable gate array*, noemt men een gate-array ook MPGA of *mask programmable gate array*. Het onderscheid is dat de MPGA masker geprogrammeerd is en de FPGA in het werkveld, in het *field*, geprogrammeerd wordt.

Het ontwerptraject voor de standaardceltechnologie, de gate-array's en FPGA's heeft veel overeenkomsten. In alle drie de gevallen wordt meestal een schema of een hardwarebeschrijvingstaal omgezet naar een netwerk van logische poorten. Het verschil is dat er in deze drie gevallen steeds op een andere technologie wordt gemapt en dat er andere plaatsing- en bedradingsmogelijkheden zijn.
In figuur 11.95 staan de ontwerptrajecten voor de drie technieken. Voor de ontwerper is het begin van het ontwerptraject het belangrijkst. Het ontwerp moet synthetiseerbaar zijn en optimaal gebruik maken van de toegepaste techniek.
Bij een gate-array- en een standaardcelontwerp gebeurt de realisatie in een IC-fabriek. Omdat alle IC's bij de productie in één keer geproduceerd worden, mag er geen enkele fout in het ontwerp zitten. Daarom worden de plaatsing en bedrading en alle testen, die daarbij nodig zijn, door specialisten uitgevoerd. Grote bedrijven hebben daar speciale afdelingen voor en kleine bedrijven besteden dit werk vaak uit aan gespecialiseerde ontwerphuizen.



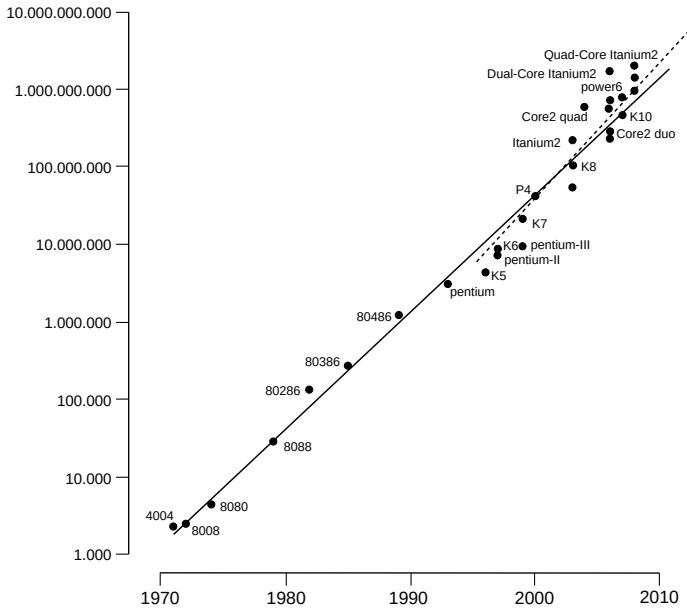
Figuur 11.95 : De ontwerptrajecten voor de FPGA, de gate-array en de standaardceltechnologie.

Xilinx en Altera bieden beide de mogelijkheid aan om van een FPGA-ontwerp ook een hardcopy te maken. Voor Altera kan dat met bouwstenen uit de Stratix-familie.

11.9 De wet van Moore en de toekomstige ontwikkelingen

Gordon Moore voorspelde in 1965 dat het aantal transistoren op een geïntegreerde schakeling iedere 24 maanden zou verdubbelen. Deze voorspelling wordt de wet van Moore genoemd en is later bijgesteld naar een verdubbeling in 18 maanden. Figuur 11.96 geeft de wet van Moore voor het aantal transistoren voor populaire microprocessoren van 1970 tot 2010.

Geen enkele ontwikkeling gaat zo snel als het kleiner worden van de transistoren. De technische inspanningen, die dit mogelijk maken, zijn gigantisch. Op allerlei gebieden zijn grenzen verlegd en zijn slimme oplossingen bedacht. De verbeteringen op het gebied van de optica, mechanica, materiaalkunde en chemie zijn groot.



Figuur 11.96 : **De wet van Moore.** Het aantal transistoren bij microprocessoren verdubbelt volgens de getrokken lijn iedere 24 maanden. De streeplijn volgt een verdubbeling van 18 maanden.

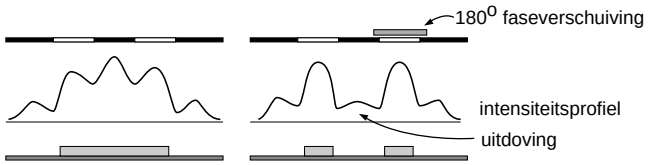
Fysische en chemische hoogstandjes

Vanwege dezelfde fysische beperking kan een gewone lichtmicroscoop geen details zichtbaar maken die kleiner zijn dan de golflengte van het licht.
Een elektronenmicroscoop kan dat wel. De golflengte van een elektronenbundel kan veel kleiner zijn dan zichtbaar licht.

Het maken van een geïntegreerde schakeling is een fysisch en chemisch proces waarbij fotolithografische technieken gebruikt worden. Fysisch is het onmogelijk om een patroon af te beelden met een nauwkeurigheid kleiner dan de golflengte die gebruikt wordt. Door buiging en interferentie vervaagt de afbeelding. Bij de belichting wordt ultraviolet licht toegepast omdat de golflengte hiervan veel kleiner is dan zichtbaar licht. Om patronen van enkele tientallen nm's te maken zou röntgenstraling toegepast moet worden. Dat klinkt simpel, maar is het niet. Het maskerpatroon wordt met een lenzenstelsel op het IC-oppervlak afgebeeld. Ofschoon er vooruitgang is, bestaan er voor röntgenstraling nog geen lenzen met de vereiste kwaliteit.

Toch lukt het om patronen te maken die veel kleiner zijn dan de golflengte van diep uv-licht. Daarvoor zijn drie oplossingen bedacht:

- **Fasecontrasttechnieken**
  Door faseverschuivingen aan te brengen kan de plaats waar de buiging en interferentie optreedt, beïnvloed worden. Figuur 11.97 laat zien dat zonder

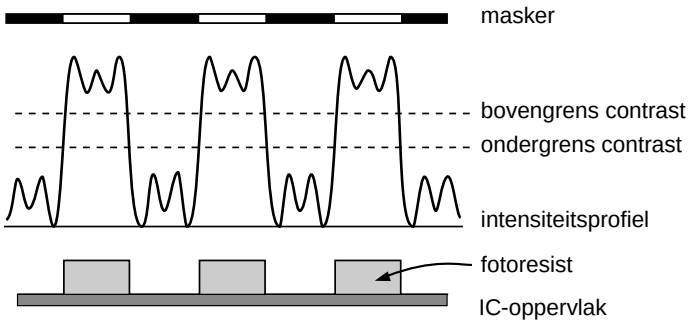


Figuur 11.97 : **De fasecontrasttechniek.** Zonder de faseverschuiving vallen de beide afbeeldingen van de sporen samen. Met de faseverschuiving zijn er wel twee sporen zichtbaar.

Frits Zernicke heeft in 1953 de Nobelprijs voor de Natuurkunde ontvangen voor zijn onderzoek naar de fasecontrastmicroscoop.

faseverschuiving de afbeelding van de twee lijnen samenvalt en met faseverschuiving er twee aparte sporen ontstaan. Deze fasecontrasttechniek is in 1930 door Frits Zernicke gebruikt bij de fasecontrastmicroscoop.

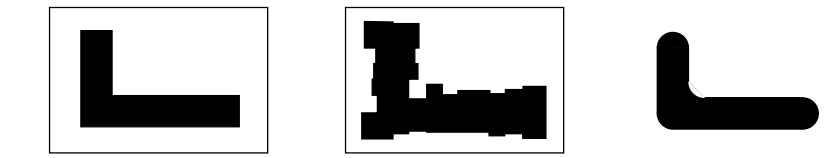
- **Fotogevoelig materiaal met een hoge contrastgevoeligheid**
  Een belangrijke stap is de ontwikkeling van fotogevoelige materialen die heel erg onderscheidend werken. Boven een bepaalde grenswaarde is het materiaal bijvoorbeeld totaal ongevoelig voor het licht en onder een andere grenswaarde is het volledig gevoelig voor het licht en verdwijnt het materiaal volledig.



Figuur 11.98 : Fotogevoelig materiaal met een scherpe contrastgevoeligheid.

Figuur 11.98 laat zien dat het materiaal of verdwijnt of helemaal aanwezig blijft. Het proces is ongevoelig voor kleine interferentie-effecten. Er ontstaat een scherpe afbeelding.

- **Optische correctietechnieken**
  Door correctiepatronen aan de maskers toe te voegen kunnen optisch fouten worden verbeterd.

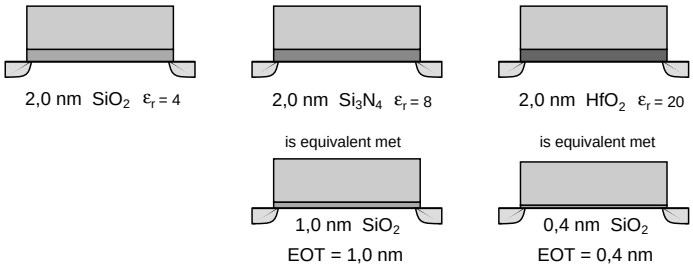


Figuur 11.99 : Optische benaderingscorrectie.

De ongecorrigeerde L-vorm in figuur 11.99 geeft geen goed resultaat. De gecorrigeerde L-vorm met de rafelige randen geeft een patroon met afgeronde hoeken dat wel voldoet.

De transistoren zijn bij iedere nieuwe generatie IC's weer een slag kleiner. De schaalverkleining verandert niet alleen de grootte van de patronen op de geïntegreerde schakelingen, maar de lagen worden ook steeds dunner. De dunste laag op het IC is de oxidelaag onder de gate van de transistoren. De dikte van deze laag is heel belangrijk voor het elektrische gedrag en zal — vanwege doorslag en tunneleffecten — altijd een aantal atoomlagen dik moeten zijn. Bij de huidige stand van zaken zou volgens Moore deze laag kleiner moeten zijn dan de atoomafstand van SiO_2. Om dit probleem op te lossen worden tegenwoordig andere oxides toegepast, zoals bijvoorbeeld Si_3N_4, HfO_2 en TiO_2. Deze materialen heb-

ben een veel hogere diëlektrische constante dan SiO_2, maar zijn moeilijker op het siliciumsubstraat aan te brengen of hebben soms andere minder goede elektrische eigenschappen. TiO_2 heeft bijvoorbeeld een erg lage doorslagspanning.



Figuur 11.100 : Drie transistoren met verschillende materialen voor het gate-oxide. De onderste twee transistoren tonen de effectieve oxidedikte (EOT).

Figuur 11.100 geeft de dwarsdoorsnede voor drie transistoren met een 2 nm gate-oxide van respectievelijk SiO_2, Si_3N_4, HfO_2. De diëlektrische constante van Si_3N_4 is ruim twee keer zo groot als die van SiO_2 en voor HfO_2 is dat ongeveer een factor vijf. Het 2 nm dikke oxide van Si_3N_4 komt, door de hogere ε_r, effectief overeen een 1 nm dik oxide van SiO_2. Het 2 nm dikke oxide van HfO_2 komt overeen met een laagje oxide van 0,4 nm SiO_2.

Om de verschillende oxides met elkaar te kunnen vergelijken wordt de dikte van het oxidelaagje uitgedrukt in EOT, *effective oxide thickness*. Dat is dikte van een overeenkomstige laag SiO_2.

De grenzen aan de schaalverkleining

Hoewel de wet van Moore nog steeds geldt en zelfs lijkt te versnellen, komen de grenzen van de schaalverkleining wel in zicht. Gordon Moore heeft zelf in diverse interviews gezegd dat de wet niet eeuwig bruikbaar is. Hij stelt onder andere: 'We naderen grenzen die niet overschreden kunnen worden. Misschien duurt dat nog wel tien jaar of langer, maar de verkleining stuit onherroepelijk op fundamentele grenzen'.
Aan de andere kant demonstreren de oplossingen om kleinere structuren te kunnen maken dan de golflengte van het licht en de vervanging van het gate-oxide door materialen met een hoge diëlektrische constante, dat het onmogelijke toch mogelijk gemaakt kan worden.

In ieder geval geldt de wet van Moore de komende jaren zeker nog. De schaalverkleining zal minimaal tot 2020 doorgaan. Er zullen dan IC's zijn met meer dan 100 miljard transistoren. Na 2020 is de trend moeilijker te voorspellen.
Er zullen fundamentele barrières overwonnen moeten worden. Sommigen denken dat zeker zal gebeuren en dat men overstapt naar een compleet nieuwe technologie, zoals Germanium III-V of de toepassing van zogenoemde nanotubes en moleculaire IC's. Het probleem bij een grote overstap is dat de nieuwe techniek direct concurrerend moet zijn met de huidige siliciumtechnologie. De snelle ontwikkeling van de afgelopen veertig jaar was ook mogelijk omdat de stappen in het veranderingsproces niet te groot waren. Nieuwe ontwikkelingen moeten immers ook economisch haalbaar zijn. Bedrijven investeren alleen als op een redelijke termijn winst te verwachten is.

Drie belangrijke factoren, die een verdere schaalverkleining in de weg staan, zijn:

- de minimale afmetingen die geproduceerd kunnen worden;
- een slechter schakelgedrag en een afnemende betrouwbaarheid;
- de toename van statische dissipatie door grotere lekstromen.

De eerste factor heeft vooral te maken met de fysische beperkingen bij de lithografische technieken. Zelfs als deze allemaal overwonnen worden zal de gate van een transistor nooit kleiner kunnen worden dan de grootte van een molecuul. Bovendien nemen de kosten exponentieel toe met het kleiner worden van de afmetingen. Bij het verkleinen treden allerlei effecten op die het elektrisch gedrag en de betrouwbaarheid nadelig beïnvloeden. Een kanaallengte van enkele atoomafstanden leidt tot variaties in de drempelspanning, een slechtere mobiliteit van de ladingdragers en elektronenmigratie.
De derde factor geeft misschien wel het grootste probleem. De lekstromen nemen exponentieel toe met het afnemen van de transistorafmetingen. Dit wordt deels veroorzaakt door het verlagen van de voedingsspanning. Als de halve pitch 25 nm is, is de statische dissipatie ongeveer even groot als de dynamische dissipatie.

Tabel 11.9 : Roadmap 2009 van ITRS. Per jaar zijn gegeven: de verwachte procesafmeting uitgedrukt in de half-pitch, de voedingsspanning van de core, het maximaal vermogen[1], het aantal pinnen, de maximale haalbare klokfrequentie van de core[2], het maximaal aantal verbindingslagen, het aantal transistoren voor nieuwe en gangbare componenten.

Year	Half-pitch (nm)	Vdd (V)	Power[1] (W)	Pins	Frequency[2] (GHz)	Layers	Number of transistors $\times 10^9$ introduction	production
2009	54	1	143	4620	5,5	12	1,5	0,8
2010	45	0,97	146	4851	5,9	12	1,5	0,8
2011	38	0,93	161	5094	6,3	12	3,1	1,5
2012	32	0,9	158	5348	6,8	12	3,1	1,5
2013	27	0,87	149	5616	7,3	13	3,1	3,1
2014	24	0,84	152	5896	7,9	13	6,2	3,1
2015	21	0,81	143	6191	8,5	13	6,2	3,1
2016	18,9	0,78	130	6501	9,2	13	6,2	6,2
2017	16,9	0,76	130	6826	9,9	14	12,4	6,2
2018	15	0,73	136	7167	10,7	14	12,4	6,2
2019	13,4	0,71	133	7525	11,5	14	12,4	12,4
2020	11,9	0,68	130	7902	12,3	14	24,7	12,4
2021	10,6	0,66	130	8297	13,3	15	24,7	12,4
2022	9,5	0,64	130	8712	14,3	15	24,7	24,7
2023	8,4	0,62		9148	15,5	15	49,5	24,7
2024	7,5	0,6		9605	16,6	15	49,5	24,7

[1] Dit is het maximaal haalbare vermogen voor high-end ASIC's en microprocessoren. Er is altijd een *heat sink* nodig om de warmte af te voeren.
[2] Dit is de maximale klokfrequentie voor high-end ASIC's en processoren. Voor gangbare FPGA's is de maximaal haalbare frequentie veel lager.

Tabel 11.9 geeft een aantal parameters uit de zogenoemde *road map* die door ITRS in 2009 is uitgebracht. ITRS staat voor *International Technology Roadmap for Semiconductors* en wordt gesponsord door de belangrijkste halfgeleiderfabrikanten in de wereld. Zij brengen regelmatig een voorspelling uit hoe de markt van halfgeleiders zich gaat ontwikkelen.
Deze voorspelling maakt duidelijk dat de ontwikkelingen door zullen gaan. Dit geldt des te meer, omdat de industrie deze voorspellingen ook als richtlijn gebruikt bij de investeringen in nieuwe technieken.

A

Gereserveerde namen

Een gereserveerde naam is in een programmeertaal een naam met een vaste betekenis. Een dergelijke naam noemt men ook wel een gereserveerd woord, een sleutelwoord of een *keyword*. In VHDL mogen de woorden uit de tabel A.1 niet voor andere doeleinden worden gebruikt.
Een naam moet in VHDL beginnen met een letter en mag verder bestaan uit letters, cijfers en het liggend streepje (_). Omdat VHDL ongevoelig is voor hoofd- en kleine letters, zijn er feitelijk veel meer namen, die niet gebruikt mogen worden. De namen `is`, `Is`, `iS` en `IS` zijn alle vier gereserveerd.

Tabel A.1 : De gereserveerde woorden van VHDL-2002.

abs	downto	library	postponed	sra
access	else	linkage	procedure	srl
after	elsif	literal	process	subtype
alias	end	loop	protected	then
all	entity	map	pure	to
and	exit	mod	range	transport
architecture	file	nand	record	type
array	for	new	register	unaffected
assert	function	next	reject	units
attribute	generate	nor	rem	until
begin	generic	not	report	use
block	group	null	return	variable
body	guarded	of	rol	wait
buffer	if	on	ror	when
bus	impure	open	select	while
case	in	or	severity	with
component	inertial	others	shared	xnor
configuration	inout	out	signal	xor
constant	is	package	sla	
disconnect	label	port	sll	

Naast gereserveerde namen gebruikt VHDL de symbolen uit tabel A.2. Andere symbolen zijn niet toegestaan.

Tabel A.2 : De symbolen bij VHDL-2002.

"	#	&	'	(	)
*	+	-	/		
,	.	:	;		
<	=	>			
=>	<=	<>	/=	**	/=

Behalve de gereserveerde namen uit tabel A.1 zijn er nog veel meer namen of woorden, die verwarring kunnen geven. In de verschillende packages worden ook typen en variabelen gedeclareerd. Deze namen kunnen ook tot conflicten leiden. Een fraai voorbeeld is de eenheid van tijd: `fs`, `ps` `ns`, `us`, `ms`, `sec`, `min` en `hr`. Zolang er geen tijden worden gebruikt, kunnen `ps` en `ns` ook de namen van een signaal of een variabele zijn. De signalen `ps` en `ns` kunnen de huidige toestand, *present state*, en volgende toestand, *next state*, van een toestandsmachine zijn. In de beschrijving van de toestandsmachine, levert deze signaaltoewijzing:

```
ps <= ns;
```

geen problemen op.
Daarentegen zal ieder simulator bij deze toekenning:

```
ps <= ns after 1 ns;
```

een foutmelding geven. Het is verstandig geen namen van typen, constanten, functies uit de packages, die voor het ontwerp nodig zijn, te gebruiken.

B

Standaard en IEEE-packages

Deze bijlage bevat de belangrijkste packages uit de standaard- en uit de IEEE-bibliotheek. Als het package een package body heeft is deze weggelaten. Bovendien is uit de packages een deel van het commentaar verwijderd en de opmaak hier en daar aangepast.

Code B.1 : Het package standard uit de standaardbibliotheek.

```
1  package standard is
2          type boolean is (false,true);
3          type bit is ('0', '1');
4          type character is (
5                  nul, soh, stx, etx, eot, enq, ack, bel,
6                  bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
7                  dle, dc1, dc2, dc3, dc4, nak, syn, etb,
8                  can, em,  sub, esc, fsp, gsp, rsp, usp,
9                  ' ', '!', '"', '#', '$', '%', '&', ''',
10                 '(', ')', '*', '+', ',', '-', '.', '/',
11                 '0', '1', '2', '3', '4', '5', '6', '7',
12                 '8', '9', ':', ';', '<', '=', '>', '?',
13                 '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
14                 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
15                 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
16                 'X', 'Y', 'Z', '[', '\', ']', '^', '_',
17                 '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
18                 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
19                 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
20                 'x', 'y', 'z', '{', '|', '}', '~', del,
21                 c128, c129, c130, c131, c132, c133, c134, c135,
22                 c136, c137, c138, c139, c140, c141, c142, c143,
23                 c144, c145, c146, c147, c148, c149, c150, c151,
24                 c152, c153, c154, c155, c156, c157, c158, c159,
25                 'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç',
26                 'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
27                 'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×',
28                 'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',
29                 'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç',
30                 'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
31                 'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷',
32                 'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ' );
33         type severity_level is (note, warning, error, failure);
34         type integer is range -2147483648 to 2147483647;
```

Code B.1 (vervolg) : Het package standard uit de standaardbibliotheek.

```
35          type real is range -1.0E308 to 1.0E308;
36          type time is range -2147483647 to 2147483647
37                  units
38                          fs;
39                          ps = 1000 fs;
40                          ns = 1000 ps;
41                          us = 1000 ns;
42                          ms = 1000 us;
43                          sec = 1000 ms;
44                          min = 60 sec;
45                          hr = 60 min;
46                  end units;
47          subtype delay_length is time range 0 fs to time'high;
48          impure function now return delay_length;
49          subtype natural is integer range 0 to integer'high;
50          subtype positive is integer range 1 to integer'high;
51          type string is array (positive range <>) of character;
52          type boolean_vector is array (natural range <>) of boolean;
53          type bit_vector is array (natural range <>) of bit;
54          type integer_vector is array (natural range <>) of integer;
55          type real_vector is array (natural range <>) of real;
56          type time_vector is array (natural range <>) of time;
57          type file_open_kind is (
58                  read_mode,
59                  write_mode,
60                  append_mode);
61          type file_open_status is (
62                  open_ok,
63                  status_error,
64                  name_error,
65                  mode_error);
66          attribute foreign : string;
67  end standard;
```

Code B.2 : Het package `std_logic_1164` uit de IEEE-bibliotheek.

```
1   PACKAGE std_logic_1164 IS
2       ------------------------------------------------------------------
3       -- logic state system  (unresolved)
4       ------------------------------------------------------------------
5       TYPE std_ulogic IS ( 'U',  -- Uninitialized
6                            'X',  -- Forcing  Unknown
7                            '0',  -- Forcing  0
8                            '1',  -- Forcing  1
9                            'Z',  -- High Impedance
10                           'W',  -- Weak     Unknown
11                           'L',  -- Weak     0
12                           'H',  -- Weak     1
13                           '-'   -- Don't care
14                         );
15      ------------------------------------------------------------------
16      -- unconstrained array of std_ulogic for use with the resolution function
17      ------------------------------------------------------------------
18      TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;
```

Code B.2 (vervolg) : Het package `std_logic_1164` uit de IEEE-bibliotheek.

```
20      -- resolution function
21      ------------------------------------------------------------------
22      FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;
23      ------------------------------------------------------------------
24      -- *** industry standard logic type ***
25      ------------------------------------------------------------------
26      SUBTYPE std_logic IS resolved std_ulogic;
27      ------------------------------------------------------------------
28      -- unconstrained array of std_logic for use in declaring signal arrays
29      ------------------------------------------------------------------
30      TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;
31      ------------------------------------------------------------------
32      -- common subtypes
33      ------------------------------------------------------------------
34      SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')
35      SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
36      SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1')
37      SUBTYPE UX01Z   IS resolved std_ulogic RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')
38      ------------------------------------------------------------------
39      -- overloaded logical operators
40      ------------------------------------------------------------------
41      FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
42      FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
43      FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
44      FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
45      FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
46      function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
47      FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;
48      ------------------------------------------------------------------
49      -- vectorized overloaded logical operators
50      ------------------------------------------------------------------
51      FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
52      FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
53      FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN std_logic_vector;
54      FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
55      FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN std_logic_vector;
56      FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
57      FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
58      FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
59      FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
60      FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
61      function "xnor" ( l, r : std_logic_vector  ) return std_logic_vector;
62      function "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;
63      FUNCTION "not"  ( l : std_logic_vector  ) RETURN std_logic_vector;
64      FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector;
65      ------------------------------------------------------------------
66      -- conversion functions
67      ------------------------------------------------------------------
68      FUNCTION To_bit       ( s : std_ulogic;        xmap : BIT := '0') RETURN BIT;
69      FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT := '0') RETURN BIT_VECTOR;
70      FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT := '0') RETURN BIT_VECTOR;
71      FUNCTION To_StdULogic       ( b : BIT               ) RETURN std_ulogic;
72      FUNCTION To_StdLogicVector  ( b : BIT_VECTOR        ) RETURN std_logic_vector;
```

Code B.2 (vervolg) : Het package `std_logic_1164` uit de IEEE-bibliotheek.

```
73      FUNCTION To_StdLogicVector  ( s : std_ulogic_vector ) RETURN std_logic_vector;
74      FUNCTION To_StdULogicVector ( b : BIT_VECTOR        ) RETURN std_ulogic_vector;
75      FUNCTION To_StdULogicVector ( s : std_logic_vector  ) RETURN std_ulogic_vector;
76      ----------------------------------------------------------------------
77      -- strength strippers and type convertors
78      ----------------------------------------------------------------------
79      FUNCTION To_X01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
80      FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
81      FUNCTION To_X01  ( s : std_ulogic        ) RETURN  X01;
82      FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
83      FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
84      FUNCTION To_X01  ( b : BIT               ) RETURN  X01;
85      FUNCTION To_X01Z ( s : std_logic_vector  ) RETURN  std_logic_vector;
86      FUNCTION To_X01Z ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
87      FUNCTION To_X01Z ( s : std_ulogic        ) RETURN  X01Z;
88      FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
89      FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
90      FUNCTION To_X01Z ( b : BIT               ) RETURN  X01Z;
91      FUNCTION To_UX01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
92      FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
93      FUNCTION To_UX01  ( s : std_ulogic        ) RETURN  UX01;
94      FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
95      FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
96      FUNCTION To_UX01  ( b : BIT               ) RETURN  UX01;
97      ----------------------------------------------------------------------
98      -- edge detection
99      ----------------------------------------------------------------------
100     FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
101     FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;
102     ----------------------------------------------------------------------
103     -- object contains an unknown
104     ----------------------------------------------------------------------
105     FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN;
106     FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN;
107     FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN;
108 END std_logic_1164;
```

Code B.3 : Het package `numeric_std` uit de IEEE-bibliotheek.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  package NUMERIC_STD is
4    constant CopyRightNotice: STRING := "Copyright 1995 IEEE. All rights reserved.";
5    --============================================================================
6    -- Numeric array type definitions
7    --============================================================================
8    type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
9    type SIGNED is array (NATURAL range <>) of STD_LOGIC;
10   --============================================================================
11   -- Arithmetic Operators:
12   --============================================================================
13   function "abs" (ARG: SIGNED) return SIGNED;
14   function "-" (ARG: SIGNED) return SIGNED;
```

Code B.3 (vervolg) : Het package `numeric_std` uit de IEEE-bibliotheek.

```
16    function "+" (L, R: UNSIGNED) return UNSIGNED;
17    function "+" (L, R: SIGNED) return SIGNED;
18    function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
19    function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
20    function "+" (L: INTEGER; R: SIGNED) return SIGNED;
21    function "+" (L: SIGNED; R: INTEGER) return SIGNED;
22    --synopsys synthesis_off
23    function "-" (L, R: UNSIGNED) return UNSIGNED;
24    function "-" (L, R: SIGNED) return SIGNED;
25    function "-" (L: UNSIGNED;R: NATURAL) return UNSIGNED;
26    function "-" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
27    function "-" (L: SIGNED; R: INTEGER) return SIGNED;
28    function "-" (L: INTEGER; R: SIGNED) return SIGNED;
29    --synopsys synthesis_off
30    function "*" (L, R: UNSIGNED) return UNSIGNED;
31    function "*" (L, R: SIGNED) return SIGNED;
32    function "*" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
33    function "*" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
34    function "*" (L: SIGNED; R: INTEGER) return SIGNED;
35    function "*" (L: INTEGER; R: SIGNED) return SIGNED;
36    --synopsys synthesis_off
37    function "/" (L, R: UNSIGNED) return UNSIGNED;
38    function "/" (L, R: SIGNED) return SIGNED;
39    function "/" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
40    function "/" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
41    function "/" (L: SIGNED; R: INTEGER) return SIGNED;
42    function "/" (L: INTEGER; R: SIGNED) return SIGNED;
43    --synopsys synthesis_off
44    function "rem" (L, R: UNSIGNED) return UNSIGNED;
45    function "rem" (L, R: SIGNED) return SIGNED;
46    function "rem" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
47    function "rem" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
48    function "rem" (L: SIGNED; R: INTEGER) return SIGNED;
49    function "rem" (L: INTEGER; R: SIGNED) return SIGNED;
50    --synopsys synthesis_off
51    function "mod" (L, R: UNSIGNED) return UNSIGNED;
52    function "mod" (L, R: SIGNED) return SIGNED;
53    function "mod" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
54    function "mod" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
55    function "mod" (L: SIGNED; R: INTEGER) return SIGNED;
56    function "mod" (L: INTEGER; R: SIGNED) return SIGNED;
57    --synopsys synthesis_off
58    -- Comparison Operators
59    --synopsys synthesis_off
60    function ">" (L, R: UNSIGNED) return BOOLEAN;
61    function ">" (L, R: SIGNED) return BOOLEAN;
62    function ">" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
63    function ">" (L: INTEGER; R: SIGNED) return BOOLEAN;
64    function ">" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
65    function ">" (L: SIGNED; R: INTEGER) return BOOLEAN;
66    --synopsys synthesis_off
67    function "<" (L, R: UNSIGNED) return BOOLEAN;
68    function "<" (L, R: SIGNED) return BOOLEAN;
```

Code B.3 (vervolg) : Het package `numeric_std` uit de IEEE-bibliotheek.

```
    function "<" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
    function "<" (L: INTEGER; R: SIGNED) return BOOLEAN;
    function "<" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
    function "<" (L: SIGNED; R: INTEGER) return BOOLEAN;
    --============================================================================
    function "<=" (L, R: UNSIGNED) return BOOLEAN;
    function "<=" (L, R: SIGNED) return BOOLEAN;
    function "<=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
    function "<=" (L: INTEGER; R: SIGNED) return BOOLEAN;
    function "<=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
    function "<=" (L: SIGNED; R: INTEGER) return BOOLEAN;
    --============================================================================
    function ">=" (L, R: UNSIGNED) return BOOLEAN;
    function ">=" (L, R: SIGNED) return BOOLEAN;
    function ">=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
    function ">=" (L: INTEGER; R: SIGNED) return BOOLEAN;
    function ">=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
    function ">=" (L: SIGNED; R: INTEGER) return BOOLEAN;
    --============================================================================
    function "=" (L, R: UNSIGNED) return BOOLEAN;
    function "=" (L, R: SIGNED) return BOOLEAN;
    function "=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
    function "=" (L: INTEGER; R: SIGNED) return BOOLEAN;
    function "=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
    function "=" (L: SIGNED; R: INTEGER) return BOOLEAN;
    --============================================================================
    function "/=" (L, R: UNSIGNED) return BOOLEAN;
    function "/=" (L, R: SIGNED) return BOOLEAN;
    function "/=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
    function "/=" (L: INTEGER; R: SIGNED) return BOOLEAN;
    function "/=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
    function "/=" (L: SIGNED; R: INTEGER) return BOOLEAN;
    --============================================================================
    -- Shift and Rotate Functions
    --============================================================================
    function SHIFT_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
    function SHIFT_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
    function SHIFT_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
    function SHIFT_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
    --============================================================================
    function ROTATE_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
    function ROTATE_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
    function ROTATE_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
    function ROTATE_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
    --============================================================================
    function "sll" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
    function "sll" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
    function "srl" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
    function "srl" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
    function "rol" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
    function "rol" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
    function "ror" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
    function "ror" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
```

Code B.3 (vervolg) : Het package `numeric_std` uit de IEEE-bibliotheek.

```
122   --============================================================================
123   --   RESIZE Functions
124   --============================================================================
125   function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL) return SIGNED;
126   function RESIZE (ARG: UNSIGNED; NEW_SIZE: NATURAL) return UNSIGNED;
127   --============================================================================
128   -- Conversion Functions
129   --============================================================================
130   function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
131   function TO_INTEGER (ARG: SIGNED) return INTEGER;
132   function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
133   function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;
134   --============================================================================
135   -- Logical Operators
136   --============================================================================
137   function "not" (L: UNSIGNED) return UNSIGNED;
138   function "and" (L, R: UNSIGNED) return UNSIGNED;
139   function "or" (L, R: UNSIGNED) return UNSIGNED;
140   function "nand" (L, R: UNSIGNED) return UNSIGNED;
141   function "nor" (L, R: UNSIGNED) return UNSIGNED;
142   function "xor" (L, R: UNSIGNED) return UNSIGNED;
143   function "xnor" (L, R: UNSIGNED) return UNSIGNED;
144   function "not" (L: SIGNED) return SIGNED;
145   function "and" (L, R: SIGNED) return SIGNED;
146   function "or" (L, R: SIGNED) return SIGNED;
147   function "nand" (L, R: SIGNED) return SIGNED;
148   function "nor" (L, R: SIGNED) return SIGNED;
149   function "xor" (L, R: SIGNED) return SIGNED;
150   function "xnor" (L, R: SIGNED) return SIGNED;
151   --============================================================================
152   -- Match Functions
153   --============================================================================
154   function STD_MATCH (L, R: STD_ULOGIC) return BOOLEAN;
155   function STD_MATCH (L, R: UNSIGNED) return BOOLEAN;
156   function STD_MATCH (L, R: SIGNED) return BOOLEAN;
157   function STD_MATCH (L, R: STD_LOGIC_VECTOR) return BOOLEAN;
158   function STD_MATCH (L, R: STD_ULOGIC_VECTOR) return BOOLEAN;
159   --============================================================================
160   -- Translation Functions
161   --============================================================================
162   function TO_01 (S: UNSIGNED; XMAP: STD_LOGIC := '0') return UNSIGNED;
163   function TO_01 (S: SIGNED; XMAP: STD_LOGIC := '0') return SIGNED;
164 end NUMERIC_STD;
```

Code B.4 : Het package `math_real` uit de IEEE-bibliotheek.

```
1  package MATH_REAL is
2      constant CopyRightNotice: STRING := "Copyright 1996 IEEE. All rights reserved.";
3      --
4      -- Constant Definitions
5      --
6      constant  MATH_E : REAL := 2.71828_18284_59045_23536;           -- Value of e
7      constant  MATH_1_OVER_E : REAL := 0.36787_94411_71442_32160;    -- Value of 1/e
```

Code B.4 (vervolg) : Het package `math_real` uit de IEEE-bibliotheek.

```
 8      constant  MATH_PI : REAL := 3.14159_26535_89793_23846;          -- Value of pi
 9      constant  MATH_2_PI : REAL := 6.28318_53071_79586_47693;        -- Value of 2*pi
10      constant  MATH_1_OVER_PI : REAL := 0.31830_98861_83790_67154;   -- Value of 1/pi
11      constant  MATH_PI_OVER_2 : REAL := 1.57079_63267_94896_61923;   -- Value of pi/2
12      constant  MATH_PI_OVER_3 : REAL := 1.04719_75511_96597_74615;   -- Value of pi/3
13      constant  MATH_PI_OVER_4 : REAL := 0.78539_81633_97448_30962;   -- Value of pi/4
14      constant  MATH_3_PI_OVER_2 : REAL := 4.71238_89803_84689_85769; -- Value 3*pi/2
15      constant  MATH_LOG_OF_2 : REAL := 0.69314_71805_59945_30942;    -- Natural log of 2
16      constant  MATH_LOG_OF_10 : REAL := 2.30258_50929_94045_68402;   -- Natural log of 10
17      constant  MATH_LOG2_OF_E : REAL := 1.44269_50408_88963_4074;    -- Log base 2 of e
18      constant  MATH_LOG10_OF_E: REAL := 0.43429_44819_03251_82765;   -- Log base 10 of e
19      constant  MATH_SQRT_2: REAL := 1.41421_35623_73095_04880;       -- square root of 2
20      constant  MATH_1_OVER_SQRT_2: REAL := 0.70710_67811_86547_52440; -- square root of 1/2
21      constant  MATH_SQRT_PI: REAL := 1.77245_38509_05516_02730;      -- square root of pi
22      constant  MATH_DEG_TO_RAD: REAL := 0.01745_32925_19943_29577;
23                                                 -- Conversion factor from degree to radian
24      constant  MATH_RAD_TO_DEG: REAL := 57.29577_95130_82320_87680;
25                                                 -- Conversion factor from radian to degree
26      --
27      -- Function Declarations
28      --
29      function  SIGN (X: in REAL ) return REAL;
30      function  CEIL (X : in REAL ) return REAL;
31      function  FLOOR (X : in REAL ) return REAL;
32      function  ROUND (X : in REAL ) return REAL;
33      function  TRUNC (X : in REAL ) return REAL;
34      function  "MOD" (X, Y: in REAL ) return REAL;
35      function  REALMAX (X, Y : in REAL ) return REAL;
36      function  REALMIN (X, Y : in REAL ) return REAL;
37      procedure UNIFORM(variable SEED1,SEED2:inout POSITIVE; variable X:out REAL);
38      function  SQRT (X : in REAL ) return REAL;
39      function  CBRT (X : in REAL ) return REAL;
40      function  "**" (X : in INTEGER; Y : in REAL) return REAL;
41      function  "**" (X : in REAL; Y : in REAL) return REAL;
42      function  EXP (X : in REAL ) return REAL;
43      function  LOG (X : in REAL ) return REAL;
44      function  LOG2 (X : in REAL ) return REAL;
45      function  LOG10 (X : in REAL ) return REAL;
46      function  LOG (X: in REAL; BASE: in REAL) return REAL;
47      function  SIN (X : in REAL ) return REAL;
48      function  COS ( X : in REAL ) return REAL;
49      function  TAN (X : in REAL ) return REAL;
50      function  ARCSIN (X : in REAL ) return REAL;
51      function  ARCCOS (X : in REAL ) return REAL;
52      function  ARCTAN (Y : in REAL) return REAL;
53      function  ARCTAN (Y : in REAL; X : in REAL) return REAL;
54      function  SINH (X : in REAL) return REAL;
55      function  COSH (X : in REAL) return REAL;
56      function  TANH (X : in REAL) return REAL;
57      function  ARCSINH (X : in REAL) return REAL;
58      function  ARCCOSH (X : in REAL) return REAL;
59      function  ARCTANH (X : in REAL) return REAL;
60  end  MATH_REAL;
```

Code B.5 : Het package `textio` uit de standaardbibliotheek.

```
1  package TEXTIO is
2      type LINE is access string;
3      type TEXT is file of string;
4      type SIDE is (right, left);
5      subtype WIDTH is natural;
6
7      file input : TEXT open read_mode is "STD_INPUT";
8      file output : TEXT open write_mode is "STD_OUTPUT";
9
10     procedure READLINE(file f: TEXT; L: out LINE);
11
12     procedure READ(L:inout LINE; VALUE: out bit; GOOD : out BOOLEAN);
13     procedure READ(L:inout LINE; VALUE: out bit);
14     procedure READ(L:inout LINE; VALUE: out bit_vector; GOOD : out BOOLEAN);
15     procedure READ(L:inout LINE; VALUE: out bit_vector);
16     procedure READ(L:inout LINE; VALUE: out BOOLEAN; GOOD : out BOOLEAN);
17     procedure READ(L:inout LINE; VALUE: out BOOLEAN);
18     procedure READ(L:inout LINE; VALUE: out character; GOOD : out BOOLEAN);
19     procedure READ(L:inout LINE; VALUE: out character);
20     procedure READ(L:inout LINE; VALUE: out integer; GOOD : out BOOLEAN);
21     procedure READ(L:inout LINE; VALUE: out integer);
22     procedure READ(L:inout LINE; VALUE: out real; GOOD : out BOOLEAN);
23     procedure READ(L:inout LINE; VALUE: out real);
24     procedure READ(L:inout LINE; VALUE: out string; GOOD : out BOOLEAN);
25     procedure READ(L:inout LINE; VALUE: out string);
26     procedure READ(L:inout LINE; VALUE: out time; GOOD : out BOOLEAN);
27     procedure READ(L:inout LINE; VALUE: out time);
28
29     procedure WRITELINE(file f : TEXT; L : inout LINE);
30
31     procedure WRITE(L : inout LINE; VALUE : in bit;
32                     JUSTIFIED: in SIDE := right; FIELD: in WIDTH := 0);
33     procedure WRITE(L : inout LINE; VALUE : in bit_vector;
34                     JUSTIFIED: in SIDE := right; FIELD: in WIDTH := 0);
35     procedure WRITE(L : inout LINE; VALUE : in BOOLEAN;
36                     JUSTIFIED: in SIDE := right; FIELD: in WIDTH := 0);
37     procedure WRITE(L : inout LINE; VALUE : in character;
38                     JUSTIFIED: in SIDE := right; FIELD: in WIDTH := 0);
39     procedure WRITE(L : inout LINE; VALUE : in integer;
40                     JUSTIFIED: in SIDE := right; FIELD: in WIDTH := 0);
41     procedure WRITE(L : inout LINE; VALUE : in real;
42                     JUSTIFIED: in SIDE := right; FIELD: in WIDTH := 0; DIGITS: in NATURAL := 0);
43     procedure WRITE(L : inout LINE; VALUE : in string;
44                     JUSTIFIED: in SIDE := right; FIELD: in WIDTH := 0);
45     procedure WRITE(L : inout LINE; VALUE : in time;
46                     JUSTIFIED: in SIDE := right; FIELD: in WIDTH := 0; UNIT: in TIME := ns);
47
48    -- is implicit built-in:
49    -- function ENDFILE(file F : TEXT) return boolean;
50 end TEXTIO ;
```

Code B.6 : Het package `std_logic_textio` uit de IEEE-bibliotheek.

```
use STD.textio.all;
library IEEE;
use IEEE.std_logic_1164.all;

package STD_LOGIC_TEXTIO is
        -- Read and Write procedures for STD_ULOGIC and STD_ULOGIC_VECTOR
        procedure READ(L:inout LINE; VALUE:out STD_ULOGIC);
        procedure READ(L:inout LINE; VALUE:out STD_ULOGIC; GOOD: out BOOLEAN);
        procedure READ(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
        procedure READ(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR; GOOD: out BOOLEAN);
        procedure WRITE(L:inout LINE; VALUE:in STD_ULOGIC;
                        JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
        procedure WRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;
                        JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);

        -- Read and Write procedures for STD_LOGIC_VECTOR
        procedure READ(L:inout LINE; VALUE:out STD_LOGIC_VECTOR);
        procedure READ(L:inout LINE; VALUE:out STD_LOGIC_VECTOR; GOOD: out BOOLEAN);
        procedure WRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                        JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);

        --
        -- Read and Write procedures for Hex and Octal values.
        -- The values appear in the file as a series of characters
        -- between 0-F (Hex), or 0-7 (Octal) respectively.
        --

        -- Hex
        procedure HREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
        procedure HREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR; GOOD: out BOOLEAN);
        procedure HWRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;
                        JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
        procedure HREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR);
        procedure HREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR; GOOD: out BOOLEAN);
        procedure HWRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                        JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);

        -- Octal
        procedure OREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
        procedure OREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR; GOOD: out BOOLEAN);
        procedure OWRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;
                        JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
        procedure OREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR);
        procedure OREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR; GOOD: out BOOLEAN);
        procedure OWRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                        JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
end STD_LOGIC_TEXTIO;
```

C

VHDL-2008

In 2008 is de standaard voor VHDL door IEEE vernieuwd. Het duurt altijd een flink aantal jaren voor een nieuwe standaard gangbaar is. Eerdere aanpassingen uit 1993 en 2002 waren niet erg groot. VHDL-2008 bevat een aantal fundamentele wijzigingen, die niet onbelangrijk zijn en vermoedelijk snel ingeburgerd raken. In 2011 worden nieuwe ontwikkelomgevingen nog geleverd met VHDL-2002 als standaard. Fabrikanten hebben een aantal jaren nodig om de nieuwe standaard volledig te implementeren.

C.1 Packages voor drijvende- en vastekommagetallen

Bij de standaard van VHDL-2008 zijn afspraken gemaakt over packages voor drijvende- en vastekommagetallen. Het package `fixed_pkg` definieert twee typen `sfixed` en `ufixed` voor respectievelijk vastekommagetallen met een teken en zonder een teken. Het package bevat verder alle gebruikelijke rekenkundige, relationele en logische bewerkingen voor deze getallen en verschillende conversiefuncties naar integers, `unsigned` en `signed`.

Het signaal `uf` declareert een 5-bits geheel getal met een 3-bits fractie:

```
        signal uf : ufixed(4 downto -3);
```

De vijf meest significante bits `(4 downto 0)` zijn het getal en de drie minst significante bits `(-1 downto -3)` vormen de fractie. Door aan dit vastekommagetal de expliciete vector "01100011"toe te kennen krijgt het de waarde 12,375:

```
        uf <= "01100011";   -- represents 12.375
```

Het werken met vastekommagetallen wijkt niet wezenlijk af van dat met gehele getallen, omdat een vastekommagetal in feite een geheel getal is dat gedeeld is door een macht van twee. De vector "01100011"stelt als unsigned getal 99 voor. Signaal `uf` heeft drie cijfers achter de komma, zodat 99 door 8 gedeeld moet worden voor de juiste betekenis.

Het package `float_pkg` definieert het type `float` voor een willekeurig drijvendekommagetal en drie typen `float32`, `float64` en `float128` voor respectievelijk 32-, 64-, en 128-bits drijvende-kommagetallen. Het package bevat daarnaast alle gebruikelijke rekenkundige, relationele en logische bewerkingen voor deze getallen en verschillende conversiefuncties naar `std_logic_vector`, `unsigned`, `signed`, `sfixed` en `ufixed`. De definities voor de drijvendekomagetallen zijn gebaseerd op de IEEE-standaarden voor floating-point getallen, namelijk: *IEEE Std 754* en *IEEE Std 854*.

De leveranciers van ontwikkelomgevingen zijn bezig om alle vernieuwingen in deze programma's aan te brengen. Op de intranetpagina van David W. Bishop, http://www.vhdl.org/fphdl/, staan verwijzingen naar de packages voor fixed-point en floating-point berekeningen, die gebruikt kunnen worden met VHDL-1993. Omdat deze packages de basis vormen van fixed-point en floating-point IEEE-packages van VHDL-2008 zijn de overeenkomsten met deze packages groot. De uitbreiding van VHDL met drijvende- en vastekommagetallen is een belangrijke aanvulling en zeer interessant voor ontwerpers. Het is een belangrijke reden om over te stappen naar VHDL-2008.

C.2 Vereenvoudigde gevoeligheidslijst

Het sleutelwoord **`all`** mag gebruikt worden in een gevoeligheidslijst. Dat maakt de beschrijvingen van combinatorische schakelingen eenvoudiger. Bij de beschrijving van code 3.17 staan alle ingangen `sel`, `d0` en `d1` van de multiplexer in de gevoeligheidslijst. In VHDL-2008 kan dit vervangen worden door het sleutelwoord **`all`**, zoals code C.1 laat zien.

Code C.1 : Een multiplexer met een proces met in de gevoeligheidslijst `all`.

```
 1  architecture gedrag of mux is
 2  begin
 3    p1: process (all) is
 4      variable w, x, y : std_logic;
 5    begin
 6      w := not sel;
 7      x := d0 and w;
 8      y := d1 and sel;
 9      z <= x or y;
10    end process p1;
11  end architecture gedrag;
```

Deze wijziging is zinvol. Het voorkomt namelijk dat de ontwerper een signaal vergeet en het maakt de code beter leesbaar. In één oogopslag is zichtbaar dat het proces een combinatorische schakeling beschrijft.

C.3 Hiërarchische namen

Een andere interessante vernieuwing is de toepassing van hiërarchische namen. Dit is niet van belang voor het ontwerp maar wel voor het testen van een ontwerp. In veel gevallen zal een test-engineer een testbench willen maken, die automatisch allerlei zaken controleert. Tot VHDL-2008 was het soms zeer lastig om een signaal te benaderen dat diep weggestopt in een ontwerp zit. Sommige simulatoren, zoals Modelsim, hadden hier al wel een mechanisme voor. In VHDL-2008 kan in een testbench deze toewijzing staan:

```
<<signal .uut.wcalc.wdivide.c : unsigned(4 downto 0)>> <= "00011";
```

Signaal `c` uit component `wdivide` van component `wcalc` van de *unit under test* krijgt de waarde `"00011"`. De signaalnaam tussen de tekens `<<` en `>>` wordt een externe naam genoemd.

Met deze externe namen is het bij het testen eenvoudig om fouten in een ontwerp aan te brengen en zo het ontwerp te controleren op allerlei bijzondere condities.

C.4 Conditionele sequentiële toewijzingen

Tot VHDL-2008 was er een streng onderscheid tussen parallelle en sequentiële constructies. De if-statements en het case-statement mogen niet in een parallelle omgeving staan en een when-else-statement en het with-select-statement mogen niet in een sequentiële omgeving staan. Code 3.1 geeft de gedragsbeschrijving van een tristatebuffer met een proces en een if-statement en in code 3.2 staat een equivalente beschrijving met een when-else-statement.
Het omzetten van een sequentiële naar een parallelle beschrijving en omgekeerd is vanwege de afwijkende syntax relatief veel werk. In VHDL-2008 mogen de when-else-statement en het with-select-statement ook in een sequentiële omgeving staan, In figuur C.1 staan proces `p1` uit code 3.1 en de conditionele signaaltoewijzing van code 3.2. Bij VHDL-2008 is de constructie van proces `p1_2008` ook toegestaan.

```
1   p1: process (inp,enable) is
2   begin
3     if enable = '1' then
4       outp <= inp;
5     else
6       outp <= 'Z';
7     end if;
8   end process p1;
```

```
1   outp <= inp when enable = '1' else 'Z';
```

```
1   p1_2008: process (inp,enable) is
2   begin
3     outp <= inp when enable = '1' else 'Z';
4   end process p1_2008;
```

Figuur C.1 : Drie methoden om het gedrag van een tristatebuffer vast te leggen. Links staat het sequentiële proces `p1` en rechts het concurrent signal assignment en het sequentiële proces `p1_2008` met een when-else-statement.

C.5 Vectors in *aggregates*

In code 3.3 staan voorbeelden van toekenningen aan variabelen en signalen. Daarbij zijn ook zogenoemde *aggregates* gebruikt. Met VHDL-2008 mogen bij deze *aggregates* ook bitstrings gebruikt worden:

```
v6 := ((2 downto 0) => "101", others => '0');
v7 := ((2 downto 0) => "101", (7 downto 4) => "1011", others => '0');
```

Bovendien kan aan een *aggregate* ook een waarde worden toegekend:

```
(x(7 downto 0), x(15 downto 8)) <= x;
```

De bovenstaande bewerking verwisselt het hoge en het lage byte van het 16-bits signaal `x`. Met naamassociatie kan het lage byte van `x` worden toegekend aan een 8-bits signaal `lowbyte` en het hoge byte van `x` aan een 8-bits signaal `highbyte`:

```
(7 downto 0 => lowbyte, 15 downto 8 => highbyte) <= x;
```

C.6 Nieuwe relationele operatoren

VHDL kent ook een aantal nieuwe relationele operatoren. Code 6.5 gebruikt op regel 79 bij de conditionele signaaltoewijzing de voorwaarde dat de nieuwe

waarde van de puls hoog en de vorige waarde van de puls laag is:

```
80   puls_detect <= '1' when (new_puls = '1') and (prev_puls = '0') else '0';
```

In VHDL-2008 mag dat ook zó worden geformuleerd:

```
81   puls_detect <= '1' when ?? (new_puls and not prev_puls) else '0';
```

De operator `??` geeft aan dat de uitkomst van de bewerking `(new_puls and not prev_puls)` een boolean is.

Het packages `std_logic_1164` is uitgebreid met een verzameling relationele functies, die het resultaat van de relationele bewerking vertaalt naar het basistype van de vector. Als `sv1`, `sv2` van het type `std_logic_vector` zijn, zal signaal `equal` de waarde `'1'` krijgen als deze vectoren gelijk zijn en `'0'` als dat niet het geval is.

```
-- equal, less, geq is std_logic
-- sv1 and sv2 are std_logic_vector, unsigned or signed
equal <= sv1 ?= sv2;
less  <= sv1 ?< sv2;
geq   <= sv1 ?>= sv2;
```

Deze functies behandelen de `'H'` als een `'1'` en de `L` als een `'0'` en de `'-'` als een echte *don't care*. In VHDL-2008 zijn alle zes vergelijkingsoperatoren ook in deze nieuwe vorm beschikbaar. Er zijn dus zes nieuwe operatoren: `?=`, `?/=`, `?<`, `?<=`, `?>`, `?>=`. Deze operatoren zijn ook in de packages `fixed_pkg`, `float_pkg` en in het nieuwe `numeric_std` package gedefinieerd en dus beschikbaar voor `unsigned`, `signed`, `ufixed`, `sfixed` en `float`.

Daarnaast zijn er aan `std_logic_1164` en aan `numeric_std` zogenoemde reductieoperatoren toegevoegd. Deze operatoren passen de logische bewerking op de bits van de vector toe:

```
-- s is std_logic
-- sv is std_logic_vector, unsigned or signed
s <= and sv;
```

Als `sv` een 3-bits vector is en alle drie de bits hoog zijn, is `s` hoog. Bij alle andere combinaties zal `s` laag zijn.

C.7 Meer mogelijkheden voor expliciet uitgeschreven bitvectoren

In dit boek is bij het toekennen van expliciete vectoren altijd de complete bitvector opgeschreven. VHDL kent ook andere opties door een `B`, `O`, of `X` voor de vector te zetten. De `B` betekent een binaire representatie, de `O` betekent octaal en `X` hexadecimaal:

```
// s is std_logic_vector, unsigned or signed
s <= "00001101";
s <= B"0000_1101";
s <= X"0D";
s <= O"014";                    -- Error
s <= resize(O"014",s'length);
```

Bij deze representaties mag de `_` als scheidingsteken gebruikt worden om de leesbaarheid te verbeteren. Het nadeel van de octale en hexadecimale notatie is dat deze altijd een veelvoud zijn van respectievelijk drie en vier bits. Dit is dan ook de reden dat in dit boek bewust geen reclame is gemaakt voor deze wijze van toekennen.

In VHDL-2008 mogen de expliciete bitvectoren een expliciete breedte krijgen, gedeclareerd worden als unsigned of signed en metawaarden bevatten:

```
// a,b,c,d,e,f,g are std_logic_vector(5 downto 0);
a <= 6X"0D";     -- "001101"
b <= 6X"XD";     -- "XX1101"
c <= 6SX"D";     -- "111101"
d <= 6UX"D";     -- "001101"
e <= 6SB"111";   -- "111111"
f <= 6UO"7";     -- "000111"
g <= 6UX"7";     -- "000007"
```

Met deze extra mogelijkheden is het expliciet uitschrijven van bitvectoren met behulp van `B`, `O`, of `X` veel aantrekkelijker geworden.

C.8 Uitbreiding van het generate-statement

VHDL-2008 staat naast een **`if`** ook een **`else`** en een **`elsif`** toe bij het generate-statement. Het generate-statement uit code 4.35 kan met VHDL-2008 ook zo worden geformuleerd:

```
16  f: for j in n-1 downto 0 generate
17    g: if j=n-1 generate
18      msb : fulladder port map (a(j), b(j), c(j), s(j), co);
19    elsif (j>0) and (j<n-1) generate
20      rst : fulladder port map (a(j), b(j), c(j), s(j), c(j+1));
21    else
22      lsb : fulladder port map (a(j), b(j), ci, s(j), c(j+1));
23    end generate g;
24  end generate f;
```

Bovendien mag in VDHL-2008 ook een case-statement bij het generate-statement worden gebruikt:

```
15  f: for j in n-1 downto 0 generate
16    g: case j generate
17       when n-1 =>
18       msb : fulladder port map (a(j), b(j), c(j), s(j), co);
19       when 0 =>
20       lsb : fulladder port map (a(j), b(j), ci, s(j), c(j+1));
21       when others =>
22       rst : fulladder port map (a(j), b(j), c(j), s(j), c(j+1));
23    end generate g;
24  end generate f;
```

C.9 De standaard IEEE-packages behoren bij de officiële standaard

Tot nu was in de officiële definitie van VHDL alleen het package `standard` uit bijlage B opgenomen. Vanaf VHDL-2008 zijn deze packages aan de officiële standaard toegevoegd:

`ieee.std_logic_1164`

`ieee.numeric_std`

`ieee.numeric_bit`

`ieee.math_real`

`ieee.math_complex`

Dit betekent alleen dat het officiële standaarden zijn. Bij de beschrijvingen moeten deze packages nog steeds op de gebruikelijke wijze worden ingevoegd:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

De packages `std_logic_unsigned` en `std_logic_signed` zijn, zoals al eerder is opgemerkt, vervangen door de packages `numeric_std_unsigned` en `numeric_std_signed`. De functionaliteit van het package `std_logic_textio` is toegevoegd aan het package `std_logic_1164` en lijkt daarmee overbodig geworden.

Het package `textio` is echter nog wel nodig en is zelfs uitgebreid. Voor het schrijven van een expliciete string was een zogenoemde *qualified expression* nodig omdat het anders onduidelijk is welke schrijffunctie nodig is. VHDL-2008 kent een nieuwe functie `swrite` voor het schrijven van een string. Regel 150 uit code 10.24:

```
16    write(L,string'("==> NOT OK expected "));
```

kan met VHDL-2008 worden vervangen door:

```
17    swrite(L, "==> NOT OK expected ");
```

De lees- en schrijffuncties voor `std_logic` en `std_logic_vector` en voor `unsigned` en `signed` zijn ook toegevoegd aan de packages `std_logic_1164` en `numeric_std`. Deze functies maken gebruik van `textio` en moet dus ook gedeclareerd zijn. Met deze bibliotheekdeclaraties zijn alle lees- en schrijffuncties voor `unsigned` en `signed` beschikbaar:

```
use std.textio.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

C.10 De functies `to_string` en `justify` toegevoegd

Dit boek definieert twee maal een functie `to_string` in code 10.10 en in code 10.12. VHDL-2008 bevat de functie `to_string` voor alle normale datatypen en een aantal varianten daarop, zoals functies `to_hstring` en `to_ostring` voor het omzetten naar een hexadecimale en octale representatie. Met het package VHDL-2008 zijn de zelfgeschreven functies `to_string` uit code 10.10 en code 10.12 overbodig.
Het package `textio` kent in VHDL-2008 een functie `justify` die het uitlijnen van tekst vereenvoudigt:

```
report "result:" &
  justify(to_string(now, ns), width => 10) & "  " &
  justify(to_string(a), LEFT, 8) &
  justify(to_hstring(a), width => 6, side => LEFT) &
  to_string(z);
```

Bovenstaand report-statement levert bijvoorbeeld dit resultaat op:

```
result:    100 ns  110101  FD3A  11010101000
result:    200 ns  111001  104E  10001101001
result:    250 ns  001001  222B  01101010001
result:    300 ns  110010  22EE  10011111001
result:    330 ns  001001  3EFB  11111010001
```

C.11 Functies minimum en maximum toegevoegd

VHDL-2008 definieert twee nieuwe functies `minimum` en `maximum`, die het minimum en het maximum tussen twee getallen bepalen:

```
-- i1 and i2 are integers
i1 <= minimum(632,234)           -- i1 is 234
i2 <= maximum(632,234)           -- i2 is 632
-- u1 is unsigned and s1 is signed
u1 <= minimum("1000", "0111")    -- u1 is "0111"
s1 <= minimum("1000", "0111")    -- u1 is "1000"
```

Deze getallen mogen van het type `integer`, `unsigned` en `signed` zijn en van het type `ufixed`, `sfixed` en `float`.

C.12 Nieuw case- en select-statement toegevoegd

Er is aan VHDL-2008 een nieuwe versie van het case-statement en een select-statement toegevoegd, waarbij de *don't care* gebruikt kan worden:

```
case? x is
when "1--"  =>  z <= a;
when "001"  =>  z <= c;
when others =>  z <= d;
when "01-"  =>  z <= b;
end case?;
```

```
with x select? z <=
  a when "1--";
  b when "01-";
  c when "001";
  d when others;
```

Bij een gewoon case-statement mag een uitdrukking maar één keer voorkomen. Bij deze *matching* case dient de ontwerper er op te letten dat de verschillende keuzes niet overlappen.

C.13 Uitbreiding van de generieke parameters

VHDL kent generieke parameters. Tot nu toe waren deze alleen toegestaan bij entity's en moesten dat constanten zijn. VHDL-2008 staat generieke parameters ook toe bij packages en subprogramma's, dus bij functies en procedures.
Bovendien mogen bij VHDL-2008 de generieke parameters ook typen, subprogramma's en packages zijn.

C.14 De mogelijkheid voor een commentaarblok toegevoegd

Net als C kent VHDL-2008 het commentaarblok. Alles tussen `/*` en `*/` wordt als commentaar beschouwd:

```
/*
  x <= "0000";
  y <= '1';
  wait for DELAY;
  x <= "1111";
  y <= '0';
  wait for DELAY;
*/
```

Deze verandering is niet echt belangrijk. Moderne platte teksteditors hebben vaak de mogelijkheid alle regels van een geselecteerd tekstblok in één keer van commentaar te voorzien.

C.15 Packages: `numeric_std_unsigned` en `numeric_std_unsigned`

De niet-officiële packages `std_logic_unsigned` en `std_logic_signed` zijn vervangen door twee nieuwe packages `numeric_std_unsigned` en `numeric_std_unsigned`.

C.16 De toekomst van VHDL-2008

Naast de genoemde toevoegingen zijn er nog meer veel kleine en grote aanvullingen. Een compleet overzicht valt buiten de context van dit boek. Sommige vernieuwingen hebben betrekking op aspecten die weinig gebruikt worden en zijn daarom weggelaten. De vorige standaarden zijn in dit boek ook niet volledig besproken. Het aantal wijzigingen bij VHDL-2008 is groot en zal zeker impact hebben. Omdat VHDL-2008 nog niet of nog niet volledig aan ontwikkelomgevingen is toegevoegd, ontbreekt de ervaring welke aanpassingen gebruikt gaan worden en welke niet.

In ieder geval zijn sommige toevoegingen zeer zinvol en zullen zeker gebruikt gaan worden, bijvoorbeeld het gebruik van drijvende- en vastekommagetallen en de toepassing van hiërarchische namen bij het maken van een testbench.
Andere wijzigingen zijn minder belangrijk en zullen niet of door een kleine groep ontwerpers gebruikt worden. Het *matching* case-statement en het gebruik van een commentaarblok zijn minder gelukkige aanpassingen.

In ieder geval zijn er genoeg aanvullingen, die voldoende belangrijk zijn om op termijn over te stappen naar VHDL-2008. Een geschikt tijdstip is het moment dat de gebruikte ontwikkelomgeving automatisch ingesteld is op de nieuwe versie. Daarmee geeft de leverancier aan dat de programma's voldoende aangepast zijn op VHDL-2008.

D

VHDL voor bepaling GGD

Deze bijlage bevat de implementatie van het alternatieve algoritme van Stein voor de berekening van de grootste gemeenschappelijke deler. Dit algoritme staat in code 7.8, een tekening van de dataverwerking staat in figuur 7.5 en het toestandsdiagram staat in figuur 7.6.
De entity staat in code D.1 en heeft vier ingangen en twee uitgangen. De signalen `x` en `y` bevatten de getallen waarvoor het GGD berekend moet worden en het signaal `result` bevat het resultaat. Er is een ingangssignaal `start` waarmee de berekening gestart wordt en het uitgangssignaal `ready` wordt één klokslag hoog als de berekening klaar is. Signaal `clk` is het kloksignaal. Er is geen asynchrone reset, de registers en de teller krijgen de juiste waarde als `start` actief is. Alleen moet de toestandsmachine bij de initialisatie in de begintoestand zijn. De synthesizer zorgt ervoor dat de juiste codering gebruikt wordt.

Code D.1 : De entity voor het bepalen van de GGD.

```
1  --
2  -- Project: GCD (Greatest Common Divisor)
3  -- Datum:   maart 2011
4  -- Versie:  6.0
5  --          Berekening GCD met eigen variant algoritme van Stein.
6  --          De gebruikte toestandsmachine is van het Mealy-type.
7  -- Auteur:  Wim Dolman
8  --
9  Library ieee;
10 use ieee.std_logic_1164.all;
11 use ieee.numeric_std.all;
12 use ieee.math_real.all;
13
14 entity gcd is
15   port (
16     clk     : in  std_logic;
17     start   : in  std_logic;
18     x       : in  std_logic_vector;
19     y       : in  std_logic_vector;
20     result  : out std_logic_vector;
21     ready   : out std_logic
22   );
23 end entity gcd;
24
```

De signalen x, y en result zijn van het type std_logic_vector. Bij de declaratie staat geen bereik. De constante n bepaalt de lengte van de signalen en wordt bij de aanroep bepaald uit het bereik van het signaal dat op x aangesloten is. Het bereik van signaal s is afhankelijk van n. De maximale tellerlengte m wordt met de log-functie uit het package math_real berekend uit n. Code D.2 geeft het declaratiedeel van de architectuur met de declaraties van de constanten en de interne signalen.

Code D.2 : De type- en signaaldeclaratie voor het bepalen van de GGD.

```
25  architecture gedrag of gcd is
26    constant n : natural := x'length;
27    constant m : natural := natural(log(real(n))/log(2.0));
28    constant ZERO  : unsigned(n-1 downto 0) := (others => '0');
29
30    signal x_reg   : unsigned(n-1 downto 0);
31    signal y_reg   : unsigned(n-1 downto 0);
32    signal x_reg_n : unsigned(n-1 downto 0);
33    signal y_reg_n : unsigned(n-1 downto 0);
34    signal r_reg   : unsigned(n-1 downto 0);
35    signal s       : unsigned(m downto 0);
36
37    signal x_is_even : std_logic;
38    signal x_is_nul  : std_logic;
39    signal y_is_even : std_logic;
40    signal y_is_nul  : std_logic;
41    signal s_is_nul  : std_logic;
42
43    signal load_i   : std_logic;
44    signal load_n   : std_logic;
45    signal x_shr    : std_logic;
46    signal y_shr    : std_logic;
47    signal s_clr    : std_logic;
48    signal s_inc    : std_logic;
49    signal s_dec    : std_logic;
50    signal r_load_x : std_logic;
51    signal r_load_y : std_logic;
52    signal r_shl    : std_logic;
53
54    type state_type is (
55      S_IDLE,
56      S_WHILE,
57      S_MUL2,
58      S_READY
59    );
60
61    signal state : state_type;
62
63  begin
```

Het dataverwerkingsgedeelte is een één-op-één vertaling van de tekening uit figuur 7.5. In code D.3 staat het eerste deel van de dataverwerking met de ingangsregisters. Alle registers hebben twee laadfuncties. De registers x_reg en y_reg laden een initiële ingangswaarde of laden de nieuwe waarde die door het proces next_xy berekend is. De signalen x_is_nul, x_is_even, y_is_nul en y_is_even geven de status van x_reg en y_reg.

Code D.3 : De ingangsregisters van het dataverwerkingsdeel.

```
64    register_x : process (clk) is
65    begin
66      if rising_edge(clk) then
67        if load_i = '1' then
68          x_reg  <= unsigned(x);
69        elsif load_n = '1' then
70          x_reg  <= x_reg_n;
71        elsif x_shr = '1' then
72          x_reg  <= '0' & x_reg(n-1 downto 1);
73        end if;
74      end if;
75    end process register_x;
76
77    x_is_nul  <= '1' when x_reg = ZERO else '0';
78    x_is_even <= not x_reg(0);
79
80    register_y : process (clk) is
81    begin
82      if rising_edge(clk) then
83        if load_i = '1' then
84          y_reg  <= unsigned(y);
85        elsif load_n = '1' then
86          y_reg  <= y_reg_n;
87        elsif y_shr = '1' then
88          y_reg  <= '0' & y_reg(n-1 downto 1);
89        end if;
90      end if;
91    end process register_y;
92
93    y_is_nul  <= '1' when x_reg = ZERO else '0';
94    y_is_even <= not y_reg(0);
```

Het combinatorische proces `next_xy` uit code D.4 is de implementatie van de *swap-substract-shift* -bewerking uit figuur 7.5 en komt overeen met de regels 19 tot en met 26 uit code 7.8. Dit proces bepaalt het verschil van `x` en `y`, verwisselt eventueel `x` en `y` en schuift de berekende `y` één positie naar rechts.

Code D.4 : De swap-substract-shift-bewerking uit het dataverwerkingsdeel.

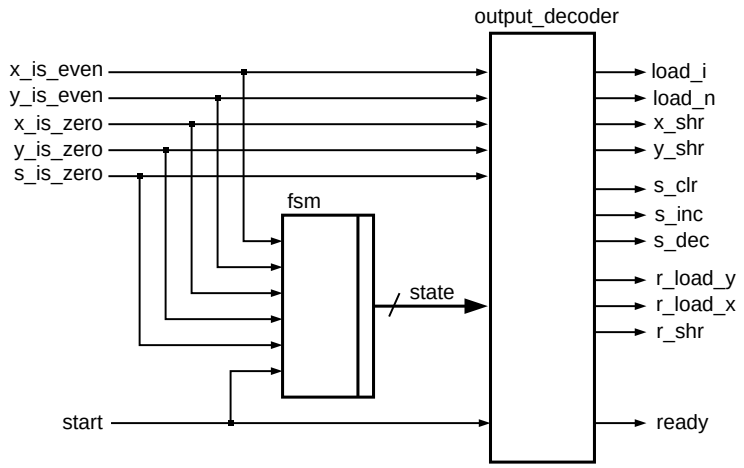
```
96    next_xy : process (x_reg, y_reg) is
97      variable tmp : unsigned(n-1 downto 0);
98    begin
99      if x_reg < y_reg then
100       x_reg_n <= x_reg;
101       tmp     := y_reg - x_reg;
102       y_reg_n <= '0' & tmp(n-1 downto 1);
103     else
104       x_reg_n <= y_reg;
105       tmp     := x_reg - y_reg;
106       y_reg_n <= '0' & tmp(n-1 downto 1);
107     end if;
108   end process next_xy;
```

Code D.5 beschrijft de rest van het dataverwerkingsdeel met het uitgangsregister `r` en de teller `s`.

Code D.5 : De dataverwerking voor het bepalen van de GGD.

```
110   counter_s : process (clk) is
111   begin
112     if rising_edge(clk) then
113       if s_clr = '1' then
114         s <= (others => '0');
115       elsif s_inc = '1' then
116         s  <= s + 1;
117       elsif s_dec = '1' then
118         s  <= s - 1;
119       end if;
120     end if;
121   end process counter_s;
122
123   s_is_nul  <= '1' when s_reg = ZERO else '0';
124
125   register_r : process (clk) is
126   begin
127     if rising_edge(clk) then
128       if r_load_x = '1' then
129         r_reg  <= x_reg;
130       elsif r_load_y = '1' then
131         r_reg  <= y_reg;
132       elsif r_shl = '1' then
133         r_reg  <= r_reg(n-2 downto 0) & '0';
134       end if;
135     end if;
136   end process register_r;
137
138   result <= std_logic_vector(r_reg);
```

De toestandsmachine is de Mealy-machine, waarvan het toestandsdiagram in figuur 7.6 staat en geïmplementeerd is met behulp van twee processen, zoals in figuur D.1 is getekend. Proces `fsm` beschrijft de toestandsdecoder en het toestandsregister. Proces `output_decoder` is de uitgangsdecoder.



Figuur D.1 : De opzet van de Mealy-machine met twee processen.

Code D.6 : Het toestandsregister en de toestandsdecoder van de toestandsmachine.

```
140   fsm : process (clk) is
141   begin
142     if rising_edge(clk) then
143       case state is
144       when S_IDLE =>
145         if start = '1' then
146           state <= S_WHILE;
147         else
148           state <= S_IDLE;
149         end if;
150       when S_WHILE =>
151         if x_is_nul = '1' then
152           state <= S_IDLE;
153         elsif y_is_nul = '1' then
154           state <= S_MUL2;
155         else
156           state <= S_WHILE;
157         end if;
158       when S_MUL2 =>
159         if s_is_nul = '1' then
160           state <= S_READY;
161         else
162           state <= S_MUL2;
163         end if;
164       when S_READY =>
165         state <= S_IDLE;
166       end case;
167     end if;
168   end process fsm;
169
```

Het sequentiële proces `fsm` staat in code D.6 en bepaalt alleen de nieuwe toestand. Het combinatorische proces `output_decoder` staat in code D.7 en berekent uit de nieuwe toestand en de ingangssignalen de waarden van de uitgangssignalen.
In proces `output_decoder` is ervoor gekozen om eerst alle uitgangssignalen de standaardwaarde te geven. Dat is waarde waarvoor deze signalen inactief zijn. Met een case-statement wordt daarna bepaald welke signalen actief zijn.

De complete implementatie voor de bepaling van de grootste gemeenschappelijke deler omvat 218 regels code. Dat is ruim zeven keer zo veel als de C-functie uit code 7.8. Dit komt doordat er voor de implementatie in VHDL veel extra's nodig zijn, zoals een entity en de extra toestanden `S_idle` en `S_ready`. Bovendien is er gekozen voor de methode met een gescheiden dataverwerking en besturing.
Dit is een voorbeeld van een probleem waarbij een implementatie met de FSMD-methode erg gunstig is. Code D.8 geeft de complete body van een FSMD-implementatie. Daarvoor zijn slechts 54 regels nodig. Omdat er ook minder interne signalen nodig zijn, is het totaal aantal regels gelijk aan 103. Dat is ruim de helft minder dan de implementatie met gescheiden dataverwerking en besturing.

Het aantal regels hoeft geen invloed op de simulatie te hebben. Het aantal klokslagen voor een GGD-berekening is voor beide versies exact gelijk. De fysieke simulatieduur is wel verschillend. Deze is getest door een zeer groot aantal keer

de GGD-berekening uit te voeren. De FSMD-implementatie blijkt ongeveer twee keer zo snel te zijn als de methode met gescheiden dataverwerking en besturing.

Code D.7 : Het uitgangsregister van de toestandsmachine van de GGD.

```
170   output_decoder : process (state, start, x_is_nul, y_is_nul,
171                                    x_is_even, y_is_even, s_is_nul) is
172   begin
173     load_i   <= '0';
174     load_n   <= '0';
175     x_shr    <= '0';
176     y_shr    <= '0';
177     s_clr    <= '0';
178     s_inc    <= '0';
179     s_dec    <= '0';
180     r_shl    <= '0';
181     r_load_x <= '0';
182     r_load_y <= '0';
183     ready    <= '0';
184     case state is
185     when S_IDLE =>
186       if start = '1' then
187         load_i <= '1';
188         s_clr  <= '1';
189       end if;
190     when S_WHILE =>
191       if x_is_nul = '1' then
192         r_load_y <= '1';
193       elsif y_is_nul = '1' then
194         r_load_x <= '1';
195       else
196         if (x_is_even = '1') and (y_is_even = '1') then
197           x_shr <= '1';
198           y_shr <= '1';
199           s_inc <= '1';
200         elsif x_is_even = '1' then
201           x_shr <= '1';
202         elsif y_is_even = '1' then
203           y_shr <= '1';
204         else
205           load_n <= '1';
206         end if;
207       end if;
208     when S_MUL2 =>
209       if s_is_nul = '0' then
210         s_dec <= '1';
211         r_shl <= '1';
212       end if;
213     when S_READY =>
214         ready <= '1';
215     end case;
216   end process output_decoder;
217
218 end architecture gedrag;
```

Code D.8 : De body van de architectuur van de FSMD-implementatie.

```
44    result <= std_logic_vector(r_reg);
45
46    fsm : process (clk) is
47      variable tmp : unsigned(n-1 downto 0);
48    begin
49      if rising_edge(clk) then
50        case state is
51        when S_IDLE =>
52          if start = '1' then
53            state <= S_WHILE;
54            x_reg <= unsigned(x);
55            y_reg <= unsigned(y);
56            s     <= (others => '0');
57          else
58            state <= S_IDLE;
59          end if;
60        when S_WHILE =>
61          if x_reg = ZERO then
62            state <= S_IDLE;
63            r_reg <= y_reg;
64          elsif y_reg = ZERO then
65            state <= S_MUL2;
66            r_reg <= x_reg;
67          else
68            state <= S_WHILE;
69            if (x_reg(0) = '0') and (y_reg(0) = '0') then
70              x_reg <= '0' & x_reg(n-1 downto 1);
71              y_reg <= '0' & y_reg(n-1 downto 1);
72              s     <= s + 1;
73            elsif x_reg(0) = '0' then
74              x_reg <= '0' & x_reg(n-1 downto 1);
75            elsif y_reg(0) = '0' then
76              y_reg <= '0' & y_reg(n-1 downto 1);
77            else
78              if x_reg < y_reg then
79                tmp   := y_reg - x_reg;
80                y_reg <= '0' & tmp(n-1 downto 1);
81              else
82                x_reg <= y_reg;
83                tmp   := x_reg - y_reg;
84                y_reg <= '0' & tmp(n-1 downto 1);
85              end if;
86            end if;
87          end if;
88        when S_MUL2 =>
89          if s_reg = ZERO then
90            state <= S_READY;
91          else
92            state <= S_MUL2;
93            s     <= s - 1;
94            r_reg <= r_reg(n-2 downto 0) & '0';
95          end if;
96        when S_READY =>
97          state <= S_IDLE;
98        end case;
99      end if;
100   end process fsm;
101
102   ready <= '1' when state=S_READY else '0';
```

Tabel D.1 : Het syntheseresultaat met Leonardo Spectrum voor Cyclone-II EP2C5Q208C8.

	Gescheiden data-path/control			FSMD		
aantal bits	logische elementen[1]	flipfloppen[2]	frequentie	logische elementen[1]	flipfloppen[2]	frequentie
8	116	34	122 MHz	89	34	122 MHz
16	200	58	122 MHz	144	58	166 MHz
32	374	106	110 MHz	257	106	110 MHz

[1]Leonardo Spectrum noemt het logische element bij de Cyclone-II een LUT
[2]Register `s` heeft bij deze realisatie een standaardgrootte van 8 flipfloppen

Tabel D.2 : Het syntheseresultaat met Leonardo Spectrum voor Spartan-3 3S200pq208-4.

	Gescheiden data-path/control			FSMD		
aantal bits	logische elementen[1]	flipfloppen[2]	frequentie	logische elementen[1]	flipfloppen[2]	frequentie
8	104	34	125 MHz	109	34	143 MHz
16	181	58	113 MHz	187	58	132 MHz
32	337	106	103 MHz	342	106	111 MHz

[1]Leonardo Spectrum noemt het logische element bij de Spartan-3 een *function generator*
[2]Register `s` heeft bij deze realisatie een standaardgrootte van 8 flipfloppen

Tabel D.3 : Het syntheseresultaat met Quartus II voor Cyclone-II EP2C5Q208C8.

	Gescheiden data-path/control			FSMD		
aantal bits	logische elementen	flipfloppen	frequentie	logische elementen	flipfloppen	frequentie
8	91	27	128 MHz	99	27	111 MHz
16	151	56	96 MHz	163	56	96 MHz
32	272	105	69 MHz	276	105	75 MHz

Voor de synthese maakt het niet veel uit voor welke implementatie wordt gekozen. De tabellen D.1, D.2 en D.3 geven de syntheseresultaten voor verschillende synthesizers en FPGA's. Bij de synthese is geen enkele restrictie opgegeven. Bovendien passen de ontwerpen royaal in de FPGA's, zodat de synthesizers weinig optimalisatie nodig hadden voor de realisatie. De resultaten kunnen totaal anders zijn als er wel een restrictie is, zoals een minimale haalbare klokfrequentie of als de FPGA bijna vol is.
Het resultaat voor een Altera Cyclone-II met behulp van Leonardo Spectrum geeft een veel kleinere oplossing voor de FSMD-beschrijving. Hoewel de verschillen kleiner zijn, is de FSMD-oplossing juist groter bij een implementatie met Leonardo Spectrum voor een Xilinx Spartan-3 en een implementatie met Altera Cyclone-II met behulp van Quartus II.
De maximaal haalbare frequentie hangt af van het aantal bits. Het resultaat lijkt voor de FSMD-methode iets gunstiger voor een groot aantal bits en voor een klein aantal bits lijkt de methode met een gescheiden dataverwerking en besturing iets gunstiger.

Uit deze resultaten blijkt dat een kleinere, compacte beschrijving niet per se een kleinere of snellere schakeling oplevert. Het enige evidente voordeel van de FSMD-methode is dat de simulatieduur korter zal zijn.

E

ASCII

ASCII staat voor American Standard Code for Information Interchange en is een standaard om letters, cijfers, leestekens en andere symbolen te representeren. Aan ieder teken is een geheel getal gekoppeld, waarmee het teken wordt aangeduid.

De ASCII-waarden zijn 7-bits breed en representeren 128 verschillende symbolen. In tabel E.1 zijn deze waarden en de bijbehorende betekenissen vermeld. De eerste 32 ASCII-waarden 0 tot en met 31 en de laatste ASCII-waarde 127 zijn besturingscommando's en zijn niet afdrukbaar. Deze ASCII-waarden waren bedoeld als meta-informatie bij bijvoorbeeld de communicatie met een printer of een telex. De andere 95 waarden (32 tot en met 126) zijn wel afdrukbaar. Sommige niet-afdrukbare symbolen hebben nog steeds een functie in de programmeertaal C. Voor deze symbolen, zoals de horizontale tabulator, is ook de *escape sequence* vermeld.

Later zijn er verschillende uitbreidingen van de ASCII-tabel gemaakt tot 256 karakters. Een voorbeeld hiervan is de ISO/IEC 8859-1. Omdat er veel meer karakters dan 256 bestaan, zijn er universele karaktersets ontwikkeld waarmee de karakters van vele talen beschreven worden, zoals Unicode, UTF-8, UTF-16 en UTF-32.

Tabel E.1 : Tabel met ASCII-waarden. De eerste kolom geeft de decimale waarde, de volgende drie kolommen geven de octale, hexadecimale en binaire waarde. De laatste kolom geeft de omschrijving of het afdrukbare karakter. Bij sommige symbolen staat ook de *escape sequence*.

Dec	Oct	Hex	Bin	Symbool
0	000	00	0000 0000	NUL, Null, '\0'
1	001	01	0000 0001	SOH
2	002	02	0000 0010	STX
3	003	03	0000 0011	ETX
4	004	04	0000 0100	EOT
5	005	05	0000 0101	ENQ
6	006	06	0000 0110	ACK
7	007	07	0000 0111	BEL, Bell, '\a'
8	010	08	0000 1000	BS, Backspace, '\b'
9	011	09	0000 1001	HT, Horizontal Tab, '\t'
10	012	0A	0000 1010	LF, Line Feed, '\n'
11	013	0B	0000 1011	VT, Vertical Tab, '\v'
12	014	0C	0000 1100	FF, Form Feed, '\f'
13	015	0D	0000 1101	CR, Carriage Return, '\r'
14	016	0E	0000 1110	SO
15	017	0F	0000 1111	SI
16	020	10	0001 0000	DLE
17	021	11	0001 0001	DC1
18	022	12	0001 0010	DC2
19	023	13	0001 0011	DC3
20	024	14	0001 0100	DC4
21	025	15	0001 0101	NAK
22	026	16	0001 0110	SYN
23	027	17	0001 0111	ETB
24	030	18	0001 1000	CAN
25	031	19	0001 1001	EM
26	032	1A	0001 1010	SUB
27	033	1B	0001 1011	ESC, Escape, '\e'
28	034	1C	0001 1100	FS
29	035	1D	0001 1101	GS
30	036	1E	0001 1110	RS
31	037	1F	0001 1111	US
32	040	20	0010 0000	SP Space, '\ '
33	041	21	0010 0001	!
34	042	22	0010 0010	"
35	043	23	0010 0011	#
36	044	24	0010 0100	$
37	045	25	0010 0101	%
38	046	26	0010 0110	&
39	047	27	0010 0111	'
40	050	28	0010 1000	(
41	051	29	0010 1001	)
42	052	2A	0010 1010	*
43	053	2B	0010 1011	+
44	054	2C	0010 1100	,
45	055	2D	0010 1101	-
46	056	2E	0010 1110	.
47	057	2F	0010 1111	/
48	060	30	0011 0000	0
49	061	31	0011 0001	1
50	062	32	0011 0010	2
51	063	33	0011 0011	3
52	064	34	0011 0100	4
53	065	35	0011 0101	5
54	066	36	0011 0110	6
55	067	37	0011 0111	7
56	070	38	0011 1000	8
57	071	39	0011 1001	9
58	072	3A	0011 1010	:
59	073	3B	0011 1011	;
60	074	3C	0011 1100	<
61	075	3D	0011 1101	=
62	076	3E	0011 1110	>
63	077	3F	0011 1111	?

Dec	Oct	Hex	Bin	Symbool
64	100	40	0100 0000	@
65	101	41	0100 0001	A
66	102	42	0100 0010	B
67	103	43	0100 0011	C
68	104	44	0100 0100	D
69	105	45	0100 0101	E
70	106	46	0100 0110	F
71	107	47	0100 0111	G
72	110	48	0100 1000	H
73	111	49	0100 1001	I
74	112	4A	0100 1010	J
75	113	4B	0100 1011	K
76	114	4C	0100 1100	L
77	115	4D	0100 1101	M
78	116	4E	0100 1110	N
79	117	4F	0100 1111	O
80	120	50	0101 0000	P
81	121	51	0101 0001	Q
82	122	52	0101 0010	R
83	123	53	0101 0011	S
84	124	54	0101 0100	T
85	125	55	0101 0101	U
86	126	56	0101 0110	V
87	127	57	0101 0111	W
88	130	58	0101 1000	X
89	131	59	0101 1001	Y
90	132	5A	0101 1010	Z
91	133	5B	0101 1011	[
92	134	5C	0101 1100	\
93	135	5D	0101 1101	]
94	136	5E	0101 1110	^
95	137	5F	0101 1111	_
96	140	60	0110 0000	`
97	141	61	0110 0001	a
98	142	62	0110 0010	b
99	143	63	0110 0011	c
100	144	64	0110 0100	d
101	145	65	0110 0101	e
102	146	66	0110 0110	f
103	147	67	0110 0111	g
104	150	68	0110 1000	h
105	151	69	0110 1001	i
106	152	6A	0110 1010	j
107	153	6B	0110 1011	k
108	154	6C	0110 1100	l
109	155	6D	0110 1101	m
110	156	6E	0110 1110	n
111	157	6F	0110 1111	o
112	160	70	0111 0000	p
113	161	71	0111 0001	q
114	162	72	0111 0010	r
115	163	73	0111 0011	s
116	164	74	0111 0100	t
117	165	75	0111 0101	u
118	166	76	0111 0110	v
119	167	77	0111 0111	w
120	170	78	0111 1000	x
121	171	79	0111 1001	y
122	172	7A	0111 1010	z
123	173	7B	0111 1011	{
124	174	7C	0111 1100	|
125	175	7D	0111 1101	}
126	176	7E	0111 1110	~
127	177	7F	0111 1111	DEL

Index

B

C

P

Q

R

S

T

U

V

W

X

Z