Digitale Techniek

Een inleiding in het ontwerpen van digitale systemen

Jesse op den Brouw

Deel 3

©2020 Jesse op den Brouw, Den Haag
Versie: 1.1
Datum: 18 augustus 2020

DE HAAGSE HOGESCHOOL

De auteur kan niet aansprakelijk worden gesteld voor enige schade, in welke vorm dan ook, die voortvloeit uit informatie in dit boek. Evenmin kan de auteur aansprakelijk worden gesteld voor enige schade die voortvloeit uit het gebruik, het onvermogen tot gebruik of de resultaten van het gebruik van informatie in dit boek.

De auteur schenkt de royalties aan Stichting KiKa.

Digitale Techniek van Jesse op den Brouw is in licentie gegeven volgens een Creative Commons Naamsvermelding-NietCommercieel-GelijkDelen 3.0 Nederland-licentie.

Suggesties en/of opmerkingen over dit boek kunnen worden gestuurd naar:
J.E.J.opdenBrouw@hhs.nl.

Voorwoord

Dit is een boek over digitale techniek. De reden waarom ik dit boek geschreven heb is dat ik de bestaande boeken niet toereikend vind. Er zijn drie beweegredenen voor de realisatie van dit boek: de onderwerpen, de Nederlandse taal en de prijs.

Een van de eigenschappen van docenten is dat ze het materiaal van anderen nooit perfect vinden voor hun eigen lessen. Slides worden toch nét weer op een andere volgorde gezet dan het boek suggereert en sommige hoofdstukken uit het voorgeschreven boek worden overgeslagen. Daarnaast missen veel boeken nog wel eens stof die wel behandeld zou moeten worden.

Er zijn nauwelijks Nederlandse boeken te vinden op het gebied van digitale techniek. Een eenvoudige zoekopdracht op een bekende boeken-site levert een tiental boeken op waarvan de meest recente in 2008 is verschenen. Er zijn zeker goede Engelse boeken te vinden over digitale techniek. Het nadeel is dat deze boeken over het algemeen vrij duur zijn, dat een drempel vormt voor studenten om het boek aan te schaffen. Ook de taal is hieraan debet.

Dit boek is geschreven voor studenten van de Nederlandstalige hbo-opleiding Elektrotechniek. De populatie wordt gevormd door een mengelmoes van niveaus: havo-ers, mbo-ers, vwo-ers, overstappers van andere hbo-opleidingen en tu-opleidingen. Dat levert nog wel eens problemen op met het niveau van de lessen: sommige studenten vinden de onderwerpen te makkelijk en anderen juist te moeilijk. Dit boek probeert een balans te vinden tussen de twee uitersten. Er zijn extra paragrafen opgenomen met verbredende en verdiepende informatie, zoals vereenvoudigen met de Quine-McCluskey-methode. Voor sommigen is bepaalde stof toch nog moeilijk te begrijpen. Er is getracht om alle principes en voorbeelden goed en uitvoerig te beschrijven. Sommige studenten hebben echter al aan een half woord genoeg.

Wat betreft de Nederlandse taal is dit boek in twee schrijfstijlen geschreven. Enerzijds wordt er informeel geschreven, andere stukken zijn wat formeler opgemaakt. Dat geeft de lezer een prettige afwisseling. Waar nodig zijn Engelse termen gebruikt omdat de industrie die nou eenmaal gebruikt.

Dit boek is opgemaakt in LaTeX [1] (LaTeX-engine = pdfLaTeX 1.40.21). LaTeX leent zich uitstekend voor het opmaken van lopende tekst, tabellen, figuren, programmacode, vergelijkingen en referenties. De gebruikte LaTeX-distributie is TexLive uit 2019 [2]. Als editor is TexStudio [3] gebruikt. Tekst is gezet in Charter [4], een van de standaard

fonts in LaTeX. De keuze hiervoor is dat het een prettig te lezen lettertype is, een *slanted* letterserie heeft en een bijbehorende wiskundige tekenset heeft. Code is opgemaakt in Nimbus Mono [5] met behulp van de *listings*-package [6]. Karnaughdiagrammen zijn opgemaakt met de *askmaps*-package, door de auteur ontwikkeld en beschikbaar gesteld op CTAN [7]. Voor het tekenen van toestandsdiagrammen is *TikZ/PGF* gebruikt [8].

Alle figuren, foto's en broncodes zijn door de auteur zelf ontwikkeld.

Leeswijzer

Hoofdstuk 11 staat geheel in het teken van toestandsmachines. Er wordt begonnen met een algemene introductie op dit gebied waarna de Mealy- en Moore-modellen worden besproken. Een eerste stap van het ontwerpen een toestandsmachine is het opstellen van een toestandsdiagram. Dit is de lastigste stap omdat vanuit een (vaak) geschreven specificatie een machine moet worden ontworpen. Alle volgende stappen kunnen routinematig door de ontwerper of softwaretools worden uitgevoerd. Het is voor een ontwerper goed om te weten hoe de machine in hardware wordt gerealiseerd. We bespreken verder hoe een reset moet worden geïmplementeerd en dat ingangssignalen moeten worden gesynchroniseerd. De taal VHDL wordt gebruikt om toestandsmachines direct vanuit het toestandsdiagram te beschrijven. Voorts bespreken we een tweetal typische toestandsmachines: patroonherkenners die in een seriële ingangsstroom een bepaald patroon herkennen en muntenherkenners die een bepaalde hoeveelheid ingeworpen munten herkennen. We bespreken ook toestandscoderingen en toestandsminimalisatie, hoewel dit laatste niet uitgebreid aan bod komt. Als laatste bepreken we het tijdgedrag van toestandsmachines.

In hoofdstuk 12 komen datapadsystemen aan bod. Dit is een beproefde methode om complexe digitale systemen te verdelen in kleinere onderdelen. Elk onderdeel kan volgens het datapad-besturingsmodel worden gesplitst in een datapad waarlangs de data wordt getransporteerd en gemanipuleerd en een besturing die het datapad aanstuurt zodat het geheel de juiste werking krijgt. De besturing wordt gerealiseerd middels een toestandsmachine. In VHDL is het mogelijk om deze splitsing door te voeren in de code, maar het is ook mogelijk om de acties op het datapad te integreren in de code voor de besturing.

Hoofdstuk 13 gaat over de ontwikkeling van een eenvoudige processor. De processor bestaat uit een datapad, vier registers, een teller en een besturings-ROM. Het datapad kan alleen 8-bits unsigned getallen verwerken. Het is mogelijk om de register te laden met een waarde, rekenkundige en logische bewerkingen uit te voeren en om te springen. Dit laatste kan ook op een conditie afhankelijk van het resultaat van een bewerking. We behandelen zowel hardware- als softwareaspecten. De processor wordt volledig in VHDL beschreven. Daarna laten we zien hoe we software kunnen ontwikkelen. Er wordt een voorbeeld gegeven hoe een looplicht kan worden ontwikkeld. Als laatste behandelen we enkele uitbreidingsmogelijkheden.

Studiewijzer

In onderstaande tabel wordt een overzicht gegeven van de stof die een student bij een eerste introductie onderwezen zou moeten krijgen. Uiteraard is iedereen vrij om zelf de onderwerpen te kiezen.

Deel 3	
Hoofdstuk 11	helemaal
Hoofdstuk 12	helemaal
Hoofdstuk 13	helemaal

Verantwoording inhoud

Kennis en kunde op het gebied van toestandsmachines is essentieel voor de digitale ontwerper. Complexe systemen bestaan namelijk uit meerdere van deze machines. In feite kunnen we zeggen dat elke sequentiële schakeling een toestandsmachine is; flip-flops, registers en schuifregisters vallen hier ook onder. De taal VHDL zorgt ervoor dat de ontwerper zich niet druk hoeft te maken over allerlei routinematige klussen zoals toestandscodering en uitwerken van de schakelfuncties en alleen maar hoeft te zorgen dat de machine correct is beschreven, het liefst met een minimum aan toestanden.

In dit boek worden alleen de kloksynchrone sequentiële machines behandeld. De reden hiervoor is dat asynchrone machines in de industrie nauwelijks ontworpen worden. De overgrote meerderheid is kloksynchroon.

Processoren horen tot het arsenaal aan mogelijkheden van de ontwerper van digitale systemen. Veel systemen zijn uitgerust met een eenvoudige processor (met software) geflankeerd door speciale hardware. Te denken valt aan een systeem met een *processor core* met gespecialiseerde hardware voor het coderen van beeld en geluid. We behandelen slechts een eenvoudig ontwerp, net genoeg om de highlights van een processor te doorgronden.

Website

Op de website https://ds.opdenbrouw.nl zijn slides, practicumopdrachten en aanvullende informatie te vinden. De laatste versie van dit boek wordt hierop gepubliceerd. Er zijn ook voorbeeldprojecten voor de Quartus-software van Altera te vinden. De projecten kunnen vaak zonder aanpassingen op het DE0-bordje van Terasic uitgeprobeerd worden.

Dankbetuigingen

Dit boek had niet tot stand kunnen komen zonder hulp van een aantal mensen. Ik wil collega Harry Broeders van Hogeschool Rotterdam bedanken voor zijn geweldige bijdrage. Niet alleen op technisch gebied, maar ook taalkundig en op de indeling van dit boek. Collega Ben Kuiper heeft een prima bijdrage geleverd op technisch en taalkundig gebied.

Verder worden mijn studenten bedankt, in het bijzonder Marcel Jongkind, Frederick Kramer, Ruben de Graaff, Bob Swinkels, Daniël Vermeulen en Abraar Muhammad, voor

het opsporen van enkele fouten en het geven van suggesties.

Jesse op den Brouw, augustus 2020.

Inhoudsopgave

11

Toestandsmachines

In eerdere hoofdstukken zijn de combinatorische logica en geheugenelementen behandeld. We gebruiken deze digitale componenten om complexere schakelingen te ontwerpen. Twee voorbeelden hebben we al gezien: schuifregisters en tellers. In dit hoofdstuk behandelen we *toestandsmachines*. Dat zijn schakelingen met geheugen die toestanden doorlopen, waarbij de toestand van de machine wordt gevormd door de waarden die liggen opgeslagen in het geheugen.

We geven eerst een algemene inleiding in het concept toestandsmachines. Daarna bespreken we de twee modellen die gebruikt worden te weten de Mealy-machine en de Moore-machine. We laten zien dat de werking van een toestandsmachine grafisch kan worden weergegeven met een toestandsdiagram, voor de ontwerper een belangrijk hulpmiddel. Een toestandsmachine kan ook worden weergegeven met een toestandstabel. Deze variant wordt vaak gebruikt in de literatuur bij het uitvoeren van wiskundige procedures, bijvoorbeeld bij toestandsminimalisatie.

Een belangrijke ontwerpstap is het kiezen van de toestandscodering. Voor complexe machines is het aantal combinaties zeer groot. Een goed gekozen codering leidt tot een minimum aan logica. We zullen ook enkele veel gebruikte coderingen bespreken.

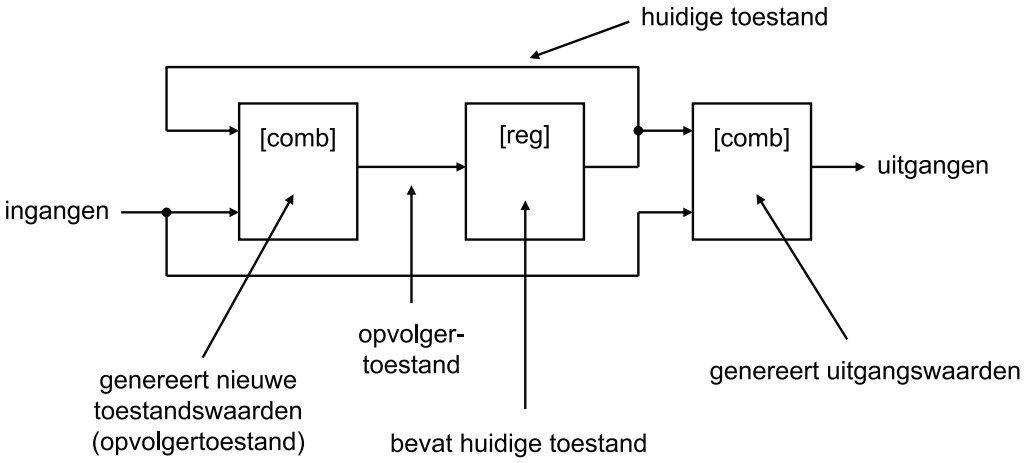
Het beschrijven van toestandsmachine in VHDL vanuit een toestandsdiagram is geen lastige stap. Er zijn zelfs tools voor handen die dit automatisch doen. We laten drie manieren zien hoe machines in VHDL beschreven worden.

Om de theorie wat inzichtelijker te maken bespreken we twee voorbeelden van toestandsmachines: herkenners die een bepaald patroon in een bitreeks herkennen en herkenners die een bepaalde hoeveelheid ingeworpen bedrag herkennen. We bespreken tevens het minimaliseren van het aantal toestanden.

Simuleren is een belangrijk onderdeel van het ontwerptraject. We bespreken de simulatie van een Moore-machine. Het hoofdstuk wordt afgesloten met een bespreking over het tijdgedrag van toestandsmachines.

11.1 Model voor toestandsmachines

Figuur 11.1 laat het algemene model voor een toestandsmachine zien. Het bestaat uit twee stukken combinatorische logica en een geheugen. Het geheugen is in staat om binaire informatie op te slaan. Op elk moment definieert de binaire informatie in het geheugen de *toestand* van de machine. We spreken ook wel over een *toestandsregister* (Engels: state register). Voor nu is het genoeg om te weten dat het aantal geheugenelementen groot genoeg is om elke toestand uniek te benoemen. Stel dan een machine drie toestanden heeft, dan zijn twee geheugenelementen genoeg. De *opvolgertoestand* (Engels: next state) van de machine wordt bepaald door toestand en de ingangswaarden. Een stuk combinatorische logica genereert de nieuwe toestandswaarden. In de figuur is te zien dat er een terugkoppeling (Engels: feedback) plaatsvindt van de toestand naar de nieuwe toestandswaarden. De uitgangen van de machine worden bepaald door de toestand en de ingangswaarden. Ook dit is een stuk combinatorische logica.



Figuur 11.1: *Algemeen model voor toestandsmachines.*

Toestandsmachines worden geclassificeerd in twee typen. Bij *synchrone* machines wordt het gedrag bepaald door de waarden van alle signalen op discrete tijdsintervallen. Dat houdt in dat het geheugen is opgebouwd uit flankgevoelige flipflops. Er is dus een kloksignaal nodig om het machine van de ene toestand naar de andere toestand te laten gaan. Voordat een klokpuls mag worden gegeven, moeten de uitgangssignalen van de combinatorische logica voor de opvolgertoestand een definitieve waarde hebben. Verder zorgt het kloksignaal ervoor dat de uitgangen veranderen nadat de actieve klokflank is gepasseerd. In de praktijk wordt dit type het meeste gebruikt. Het gedrag van *asynchrone* machines wordt bepaald door de waarden van de signalen op een gegeven moment *en* door de volgorde waarin de signalen veranderen. Dit houdt in dat het geheugen is opgebouwd uit (SR-)latches maar het kan ook zijn dat geheugenwerking wordt gerealiseerd door terugkoppellussen. Een van de grootste problemen zijn de zogenoemde race-condities. Dit is als gevolg van het veranderen van de waarden van twee of meer geheugenelementen door verandering van de ingangen. Als de vertraging van de terugkoppellussen niet gelijk zijn kan de machine op een onvoorspelbare wijze van toestand veranderen. Stel dat de toestand van de machine moet veranderen van 00 naar 11, dan kan het zijn dat eerst de (tussen-)toestand 01 wordt gerealiseerd (merk op dat het niet mogelijk is om twee bits tegelijk te laten veranderen), maar het kan ook zijn dat de machine eerst de

(tussen-)toestand 10 aanneemt. Tot op zekere hoogte is hier wel wat aan te doen maar het kost vaak extra logica (denk aan het elimineren van hazards) waardoor de machine uit meer logica bestaat dan volgens de regels van de schakelalgebra strikt noodzakelijk is. Het zorgt er ook voor dat de machine trager wordt.

Al met al zijn er diverse redenen om toestandsmachine kloksynchroon te ontwerpen:

- *Eenvoudig ontwikkeltraject voor het ontwerpen, bouwen en testen van een machine.* Het ontwikkelen van de machine is met behulp van een aantal ontwerpstappen te realiseren. De lastigste stap is het initiële ontwerp vanuit een geschreven specificatie. Alle andere stappen kunnen daarna routinematig worden uitgevoerd. Een belangrijk aspect van het ontwerpen is dat logisch gedrag en tijdgedrag van elkaar gescheiden kunnen worden bestudeerd.
- *Eenvoudige functionele verificatie.* Gegeven een bepaalde toestand waarin de machine verkeert, dan is het eenvoudig om te testen of de machine naar de juiste opvolgertoestand gaat. De ingangen moeten op een juiste waarde worden ingesteld, er wordt gewacht tot alle uitgangen een definitieve waarde hebben aangenomen en daarna wordt een actieve klokflank aangeboden. Het is daarom eenvoudig om extra logica aan te brengen waarmee een machine getest kan worden.
- *Eenvoudige timingverificatie.* Met de gereedschappen die we besproken hebben in een eerder hoofdstuk is eenvoudig aan te tonen dat de machine aan alle tijdseisen voldoet, zoals maximale klokfrequentie, setup- en holdtijden van de ingangen en minimale en maximale vertragingstijden van de uitgangen.
- *Eenvoudige optimalisatie naar de kleinst mogelijke schakeling.* Omdat een machine alleen uit combinatorische logica en flipflops bestaat, is het mogelijke om de minimalisatieprocedures uit een eerder hoofdstuk te gebruiken om de combinatorische logica te minimaliseren. Kritische race-condities komen niet voor.

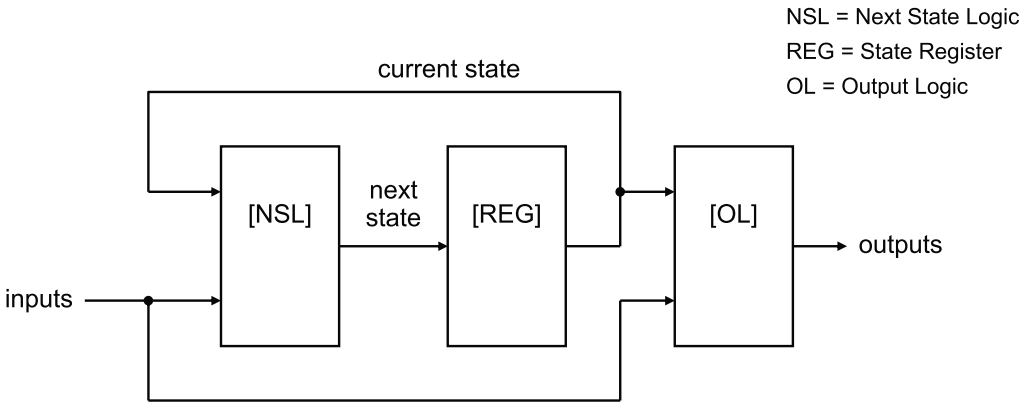
Er zijn ook enkele nadelen:

- *Hulpkloksignaal nodig.* Er is een kloksignaal nodig om de machine van de ene toestand naar de andere toestand te laten gaan. De klok heeft dus geen logische werking.
- *Over het algemeen trager dan asynchrone schakelingen.* In een kloksynchroon systeem moet gewacht worden met de volgende klokflank totdat alle instelsignalen van de flipflops een definitieve waarde hebben. Dat betekent dat de snelheid afhangt van het langzaamste pad (kritieke pad).
- *Verbruikt meer energie dan asynchrone schakelingen.* Zo'n beetje alle moderne ic's worden in de CMOS-technologie gemaakt. Veranderingen van signalen zorgen ervoor dat energie verbruikt wordt. Een kloksynchrone machine verbruikt altijd energie ook als de ingangen en de toestand niet veranderen. Het kloksignaal stopt namelijk niet (we gaan uit van een continu aangevoerd kloksignaal).

Praktische toestandsmachines hebben een eindig aantal toestanden. Een veelgebruikte term is *Finite State Machine* (FSM). Ook de term *automaat* wordt veel gebruikt.

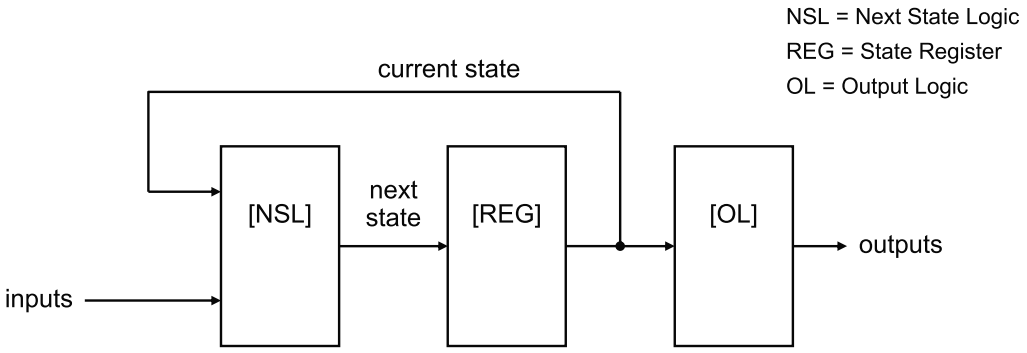
11.2 Mealy- en Moore-machines

Figuur 11.2 laat het blokschema van een *Mealy-machine* [9] zien. In feite komt dit model overeen met het algemene model in figuur 11.1. Kenmerkend voor een Mealy-machine is dat de uitgangen afhankelijk zijn van de huidige toestand (Engels: present state, current state) en van de ingangswaarden. De opvolgertoestand (Engels: next state) wordt gerealiseerd door een combinatorische schakeling (Next State Logic, NSL). De uitgangswaarden worden ook door een combinatorische schakeling gerealiseerd (Output Logic, OL). De toestand ligt opgeslagen in het toestandsregisters (State Register, REG).



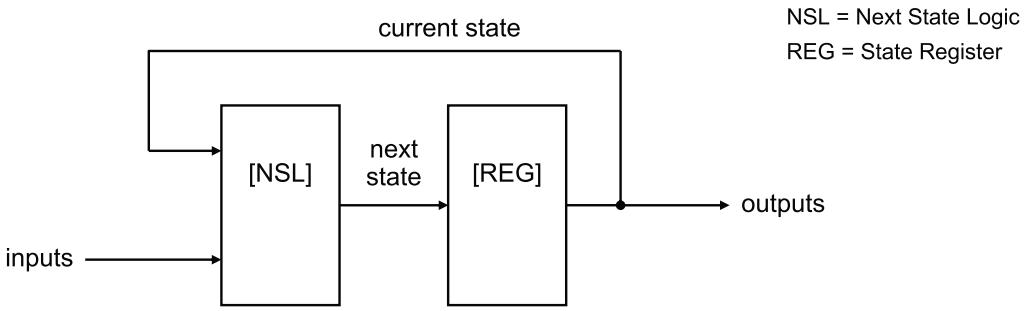
Figuur 11.2: *Model voor een Mealy-machine.*

In figuur 11.3 is het blokschema van een *Moore-machine* [10] te zien. Bij een Moore-machine zijn de uitgangen alleen afhankelijk van de toestand. De Moore-machine is een vereenvoudiging van de Mealy-machine. Te zien is dat de directe verbinding tussen de ingangen en de uitgangen is weggelaten. Een Moore-machine kan altijd worden weergegeven als een Mealy-machine, bijvoorbeeld met een toestandsdiagram (zie paragraaf 11.3).



Figuur 11.3: *Model voor een Moore-machine.*

Een speciaal geval van de Moore-machine ontstaat als de uitgangslogica wordt weggelaten. Een blokschema is gegeven in figuur 11.4. De uitgangswaarden worden dan gevormd door de toestand. Dit wordt een *Medvedev-machine* [11] genoemd. In de literatuur worden ook wel de termen *direct outputs* en *State=output machine* gebruikt. Een voorbeeld van een Medvedev-machine is een teller waarvan de telstand direct op de uitgangen is waar te nemen.



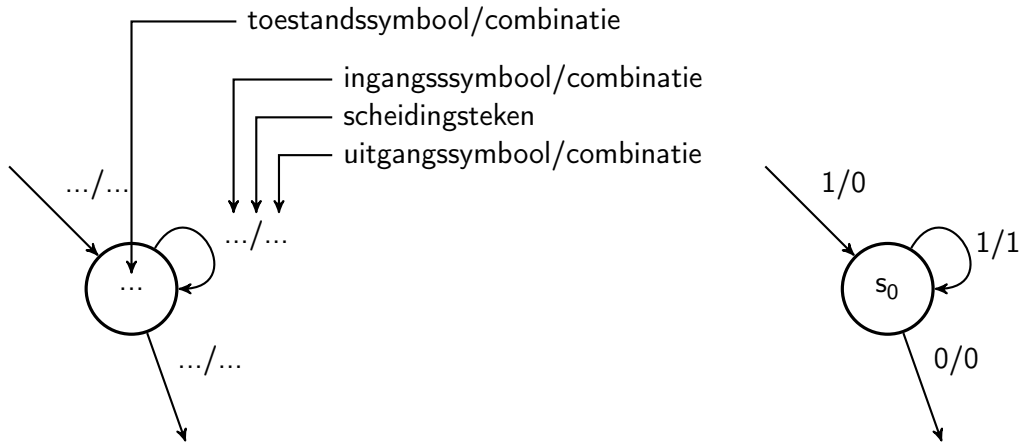
Figuur 11.4: *Model voor een Medvedev-machine.*

Het kan zijn dat bij een bepaalde machine sommige uitgangen alleen afhankelijk zijn van de toestand en de andere van zowel de toestand als de ingangen. We spreken dan van een gecombineerde Mealy/Moore-machine. Dit geeft duidelijk aan dat het verschil tussen beide typen de uitgangen betreft. De overeenkomst tussen de verschillende typen machines is ook duidelijk. Ze zijn allemaal opgebouwd rond een toestandsregister en de logica die de opvolgertoestand bepaalt.

11.3 Toestandsdiagrammen

De werking van een machine kan eenvoudig weergegeven worden met een toestandsdiagram. Het geeft aan wat de machine onder welke conditie gaat (moet gaan) doen. Een toestandsdiagram bestaat uit een aantal cirkels of bollen die de toestanden voorstellen en pijlen die de overgangen (transities) en de overgangscondities van de ene naar de andere toestand aangeven. Er zijn twee beschrijvingswijzen, één voor Mealy-machines en één voor Moore-machines.

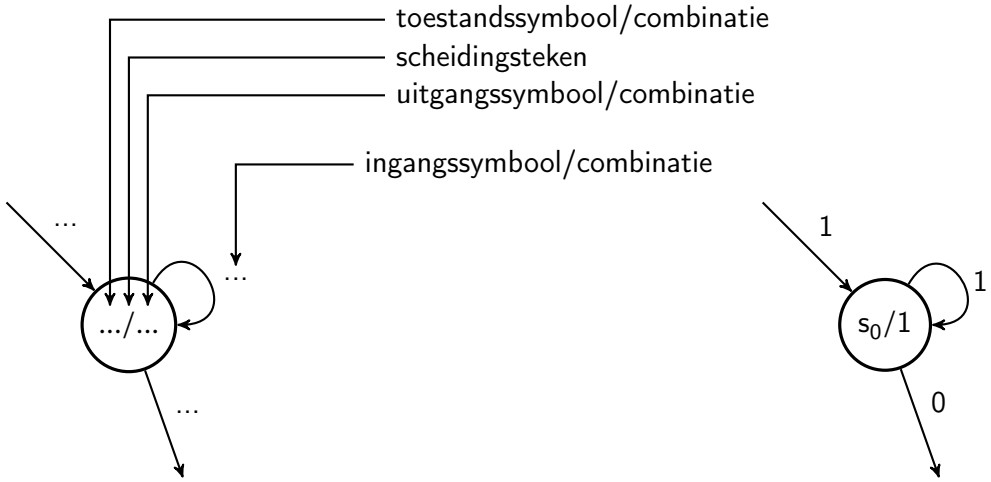
In figuur 11.5 is de beschrijving en een voorbeeld van een Mealy-toestand te zien. Een toestand wordt gekenmerkt door een cirkel. In de toestand wordt het toestandssymbool geschreven, maar er kan ook een bitcombinatie worden geplaatst. De overgangen worden door pijlen aangegeven. Naast de pijlen worden de overgangscondities en de



Figuur 11.5: *Beschrijving en voorbeeld van een Mealy-toestand.*

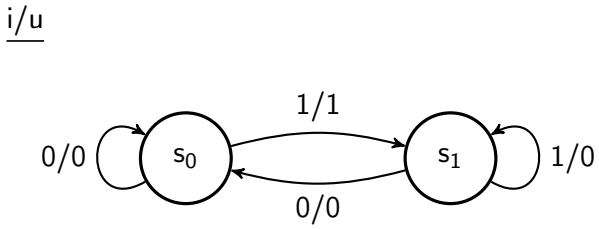
uitgangswaarden geschreven. Dit kunnen symbolen of een bitcombinatie zijn. Een uitgaande pijl kan ook naar de toestand zelf wijzen. Dit geeft aan dat de machine in deze toestand blijft als de overgangsconditie waar is.

De beschrijving en een voorbeeld van een Moore-toestand is te zien in figuur 11.6. In de toestand wordt het toestandssymbool of een bitcombinatie en de uitgangswaarden weergegeven, gescheiden door een schuine streep. Naast een pijl van een overgang wordt de overgangsconditie geplaatst. Dit is een symbool of een bitcombinatie. Een uitgaande pijl kan ook weer naar de toestand zelf wijzen.



Figuur 11.6: *Beschrijving en voorbeeld van een Moore-toestand.*

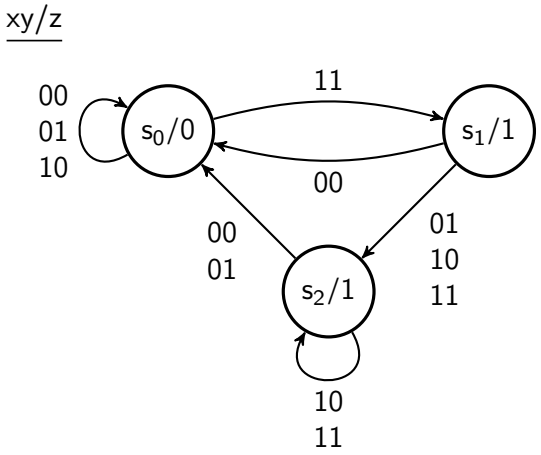
In figuur 11.7 is een toestandsdiagram van een Mealy-machine te zien met twee toestanden die we s_0 en s_1 noemen. Er is één ingang i en één uitgang u. Linksboven het diagram worden deze ingang en uitgang genoteerd zodat de namen niet bij elke overgang hoeven te worden geschreven. Bij toestand s_0 zijn twee overgangen te zien. Als de ingangswaarde 0 is, blijft de machine in toestand s_0. De uitgangswaarde is dan 0. De machine gaat naar toestand s_1 als de ingangswaarde 1 is. De uitgangswaarde is dan ook 1. De machine blijft in toestand s_1 als de ingangswaarde een 1 is (de uitgang is dan 0) en de machine gaat naar toestand s_0 als de ingangswaarde 0 is. Ook dan is de uitgang 0.



Figuur 11.7: *Een voorbeeld van een toestandsdiagram van een Mealy-machine.*

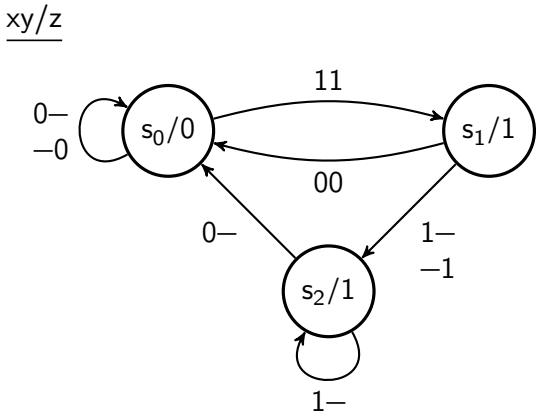
In figuur 11.8 is het toestandsdiagram van een Moore-machine te zien. De machine heeft drie toestanden, s_0, s_1 en s_2. Er zijn twee ingangen genaamd x en y en één uitgang genaamd z. In toestand s_0 is een pijl te zien die naar de toestand zelf wijst. Er zijn drie overgangsvoorwaarden bij geschreven, te weten 00, 01 en 10. Als een van de drie

overgangsvoorwaarden waar is, blijft de machine in deze toestand. We kunnen dit lezen als een logische OR van de voorwaarden. Onder de overgangsvoorwaarde $xy = 11$ gaat de machine naar toestand s_1. Alle andere overgangen kunnen op vergelijkbare wijze gelezen worden. In toestand s_0 is de uitgangswaarde van de machine 0. De machine geeft een 1 af in de toestanden s_1 en s_2.



Figuur 11.8: *Een voorbeeld van een toestandsdiagram van een Moore-machine.*

We kunnen de overgangscondities compacter schrijven door gebruik te maken van don't cares. Dit is te zien in figuur 11.9. De ingangscondities bij de pijl van toestand s_0 naar zichzelf kan gelezen worden als 0— en —0. Alle drie de condities vallen hieronder. Bekijken we nu s_2 dan zien we dat de machine in deze toestand blijft als de ingangsvoorwaarde 1— is. We kunnen ook zeggen dat de machine in s_2 blijft als $x = 1$. De overgang is dus onafhankelijk van de waarde van y.

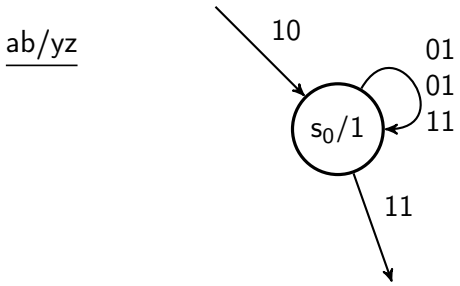


Figuur 11.9: *Een voorbeeld van een toestandsdiagram van een Moore-machine.*

Hoewel dit niet in de figuren is weergegeven, kunnen ook uitgangswaarden als don't care worden gespecificeerd. Algemeen kunnen we zeggen dat een toestandsdiagram *volledig* en *niet tegenstrijdig* moet zijn en dat don't cares kunnen worden gebruikt bij de specificatie. Volledig wil zeggen dat bij een toestand alle overgangscondities moeten worden gegeven. Bij n ingangen betekent dat dus 2^n overgangscondities. Niet tegenstrij-

dig wil zeggen dat bij een toestand geen enkele overgangsconditie twee of meer keer mag voorkomen.

In figuur 11.10 is een voorbeeld te zien van een Moore-toestand die onvolledig en tegenstrijdig is. Zo is de uitgangswaarde van z niet gegeven. Bij de overgangen is niet gegeven wat de machine moet doen bij de overgangsvoorwaarde 10 en het is onduidelijk wat de opvolgertoestand is bij overgangsconditie 11. Er zijn immers twee pijlen met die conditie. Verder moet elke toestand uniek gelabeld worden anders zijn de toestanden niet te onderscheiden. Dat kan een symbolische codering zijn zoals s_0 of een bitcombinatie.



Figuur 11.10: *Een deel van een onvolledig en tegenstrijdig toestandsdiagram.*

Het ontwerpen een van toestandsdiagram (en dus de machine) begint bij het bepalen van het aantal toestanden dat nodig is en welke overgangsvoorwaarden nodig zijn. Dit volgt uit een (meestal) geschreven beschrijving. Hiervoor is geen vaste procedure. De ontwerper moet zelf bedenken wat de machine moet doen op basis van de beschrijving en goed nadenken over eventuele situaties die niet beschreven zijn. Er is altijd een begintoestand waar de machine in terecht komt na een resetopdracht.

11.4 Toestandstabellen

De informatie uit een toestandsdiagram is om te zetten in een *toestandstabel*. In zo'n tabel worden alle mogelijke toestanden, overgangen en uitgangswaarden weergegeven. De toestandstabel van de Mealy-machine is gegeven in tabel 11.1.

Tabel 11.1: *Toestandstabel van de Mealy-machine.*

toestand	opvolger		uitgang	
	0	1	0	1
s_0	s_0	s_1	0	1
s_1	s_0	s_1	0	0

In de kolom toestand worden alle toestanden van de machine opgesomd. De kolom opvolger bestaat uit twee delen. Een deel somt alle opvolgertoestanden op voor de ingangswaarde 0, het andere deel somt alle opvolgertoestanden op voor de ingangswaarde 1. In de kolom uitgang worden alle mogelijke uitgangswaarden van de machine genoteerd. De uitgang is ook afhankelijk van de ingang dus er is een kolom met twee delen.

In tabel 11.2 is de toestandstabel van de Moore-machine te zien. Alleen de opvolgertoestand is afhankelijk van de huidige toestand en de ingangswaarden. De uitgang is alleen afhankelijk van de toestand.

Tabel 11.2: *Toestandstabel van de Moore-machine.*

toestand	opvolger				uitgang
	00	01	10	11	
s_0	s_0	s_0	s_0	s_1	0
s_1	s_0	s_2	s_2	s_2	1
s_2	s_0	s_0	s_2	s_2	1

11.5 Toestandscodering

Om een toestandsmachine met digitale componenten te maken moet aan elke toestand een binaire code worden toegekend. De toestanden moeten uniek gecodeerd worden dus bij n toestanden zijn er tenminste $\lceil {}^2\log n \rceil$ *toestandsbits* nodig. Voor elke toestandsbit is één flipflop nodig. De keuze van de codering is vrij te kiezen zolang deze voor elke toestand uniek is. Er kunnen meer flipflops gebruikt worden dan strikt noodzakelijk is.

De keuze van een toestandscodering is niet gemakkelijk. Zo is het aantal unieke coderingen te berekenen met [12]:

$$N = \frac{(2^k - 1)!}{(2^k - n)!k!} \tag{11.1}$$

Hierin is N het aantal unieke coderingen, k het aantal toestandsbits en n het aantal unieke toestanden. In de noemer komt de factor $k!$ voor. Dit heeft te maken met het feit dat veel coderingen slechts herschikkingen zijn van de codecombinaties. Zo zijn de coderingen {00, 01, 10, 11} en {00, 10, 01, 11} eigenlijk hetzelfde alleen zijn de toestandsbits verwisseld. In bovenstaande formule wordt geen rekening gehouden met eventuele inversen van de toestandsbits. Als we ervan uitgaan dat inversen extra logica met zich meebrengen dan wordt het aantal coderingen:

$$N_D = \frac{2^k!}{(2^k - n)!k!} \tag{11.2}$$

Voor een machine met bijvoorbeeld drie toestandbits en vijf toestanden zijn 1120 verschillende coderingen mogelijk. Gaan we uit van vier toestandsbits met negen toestanden, dan bedraagt het aantal coderingen meer dan 170.000.000.

De keuze van de codering heeft directe consequenties voor de omvang van de combinatorische logica. Een juiste codering zorgt voor een minimum aan logica. Er is echter geen procedure bekend waarmee efficiënt de beste codering kan worden gevonden. Er zit niets anders op dan alle mogelijke coderingen te proberen. Voor een machine met een groot aantal toestanden is dat ondoenlijk.

De praktijk leert echter een aantal “optimale” coderingen voor bepaalde klassen van machines. Zo blijkt dat als twee of meer toestanden dezelfde opvolger hebben, de toe-

standscoderingen in slechts één bit moeten verschillen. Dit geldt ook voor twee opvolgertoestanden van een toestand [13]. Amann toont in [14] aan dat het gebruik van laadbare Johnson-tellers als toestandsgeheugen een besparing oplevert tot wel 30% van de producttermen die de opvolgertoestand en uitgangswaarden genereren. Als blijkt dat in een toestandsmachine een telvolgorde verborgen zit, kan de toestandscodering van een teller worden gebruikt. Ook is de Gray-code dan een geschikte kandidaat.

De one-hot-codering gebruikt voor elke toestand één flipflop. Op elk moment is de stand van slechts één flipflop logisch 1. De achterliggende gedachte is dat de omvang van de producttermen van de logica voor de opvolgertoestand afneemt en hierdoor de maximale propagatietijd van de combinatoriek kleiner wordt. Deze codering wordt veel toegepast bij FPGA's, want die hebben veel flipflops aan boord. Het voordeel is dat er veel don't cares voorkomen en dat minimaliseert de logica. Het nadeel is dat er veel niet-gebruikte codecombinaties zijn. We zullen de one-hot codering in paragraaf 11.7 nader toelichten.

Thijssen heeft onderzocht dat wanneer een groot aantal willekeurig gekozen coderingen wordt geprobeerd, de verhouding tussen de codering met het minst aantal poorten (gate equivalent) en de codering met het meest aantal poorten een factor 1/3 à 1/4 bedraagt [15]. Het advies luidt dan ook: probeer een tiental willekeurige coderingen en selecteer die met de minst aantal poorten. Het gevonden aantal poorten ligt dan dicht tegen het echte minimum aan.

11.6 Toestandsfuncties en uitgangsfuncties

Nadat de toestandscodering bepaald is, worden de functies voor de opvolgertoestand (Engels: next state equations) en de uitgangen (Engels: output equations) afgeleid. Uit de toestandstabel wordt een waarheidstabel opgesteld met van links naar rechts:

- de toestandscoderingen gecombineerd met de ingangscombinaties;
- de opvolgertoestanden;
- de instelsignalen voor de gebruikte flipflops (D, JK, SR, T);
- de uitgangscombinaties.

We zullen dit alleen toelichten met het gebruik van D-flipflops omdat dit type verreweg het meest gebruikt wordt in de huidige chiptechnologie. We bespreken eerst de Mealy-machine. Er zijn slechts twee toestanden dus één flipflop is genoeg om unieke codes toe te kennen. Toestand s_0 wordt gecodeerd met 0 en toestand s_1 met 1. In tabel 11.3 is de waarheidstabel te zien.

Tabel 11.3: *Waarheidstabel met toestandsfuncties en uitgangsfunctie voor de Mealy-machine.*

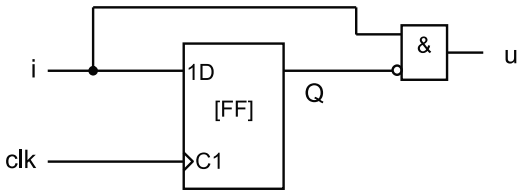
Q^n	i^n	Q^{n+1}	D^n	u^n
0	0	0	0	0
0	1	1	1	1
1	0	0	0	0
1	1	1	1	0

De eerste twee kolommen geven alle mogelijke combinaties aan van toestanden en ingangswaarden. Dat komt omdat de opvolgertoestand en de uitgangswaarde afhankelijk zijn van de huidige toestand en de ingangswaarden. We noemen de flipflop-uitgang Q en de huidige waarde noemen we Q^n. De ingangswaarde wordt i^n genoemd. De nieuwe waarde van de flipflop wordt Q^{n+1} genoemd. Om dat te bereiken moet de D-ingang van de flipflop met de nieuwe waarde worden ingesteld. De D-ingang wordt D^n genoemd omdat de functie van de D-flipflop $Q^{n+1} = D^n$ is. De uitgangswaarde wordt u^n genoemd.

De functies zijn eenvoudig van aard zodat we direct kunnen opschrijven:

$$\begin{aligned} D^n &= i^n \\ u^n &= \overline{Q^n} \cdot i^n = [\overline{Q} \cdot i]^n \end{aligned} \tag{11.3}$$

In figuur 11.11 is het schema van de Mealy-machine te zien. Eén ingang van de AND-poort is geïnverteerd. Dat wordt aangegeven door de kleine cirkel. In de praktijk is echter wel een NOT-poort nodig. Veel flipflops hebben ook een $\overline{Q}$-uitgang beschikbaar zodat de uitgangsschakeling nog vereenvoudigd kan worden.



Figuur 11.11: *Schema van de Mealy-machine.*

De Moore-machine heeft drie toestanden. Er zijn twee flipflops nodig om de toestanden te coderen. Een voor de hand liggende code is: $s_0 = 00$, $s_1 = 01$ en $s_2 = 10$. De combinatie 11 wordt niet gebruikt. De machine heeft dus *ongebruikte toestanden*. Aangezien de machine twee ingangen en twee toestandsbits heeft, bestaat de waarheidstabel uit 16 regels. Deze is gegeven in tabel 11.4.

De eerste vier regels horen bij toestand s_0, de tweede vier regels bij toestand s_1 etc. We lezen de eerste regel als volgt: in toestand $s_0 = 00$ met ingangswaarden $xy = 00$ is de opvolgertoestand $s_0 = 00$, de D-ingangen $D_1D_0 = 00$ en de uitgang $z = 0$. De kolommen voor de opvolgertoestand en de D-ingangen zijn identiek, zodat we de kolommen voor D_1^n en D_0^n kunnen weglaten.

De machine gebruikt twee toestandsbits dus er zijn vier toestanden te benoemen. De laatste toestand wordt niet gebruikt. We kunnen de waarden voor de opvolgertoestand en de uitgang als don't care opgeven. Dat is in de tabel ook gedaan. Het voordeel is dat de logica geminimaliseerd kan worden met gebruik van de don't cares. Het nadeel is dat als de machine in zo'n ongebruikte toestand terecht komt, de werking niet meer correct is. Het zou kunnen zijn dat de machine in de ongebruikte toestand blijft. In de praktijk worden ongebruikte toestanden vaak omgeleid zodat de opvolger altijd naar één toestand overgaat. Meestal is dat de toestand waar de machine in terecht komt bij een reset. Het is ook mogelijk om een gebruikte en ongebruikte toestand als identiek te beschouwen zodat de machine dezelfde werking heeft in beide toestanden. Dit worden

Tabel 11.4: *Waarheidstabel met toestandsfuncties en uitgangsfunctie voor de Moore-machine.*

	Q_1^n	Q_0^n	x^n	y^n	Q_1^{n+1}	Q_0^{n+1}	D_1^n	D_0^n	z^n
s_0	0	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	0	0
	0	0	1	0	0	0	0	0	0
	0	0	1	1	0	1	0	1	0
s_1	0	1	0	0	0	0	0	0	1
	0	1	0	1	1	0	1	0	1
	0	1	1	0	1	0	1	0	1
	0	1	1	1	1	0	1	0	1
s_2	1	0	0	0	0	0	0	0	1
	1	0	0	1	0	0	0	0	1
	1	0	1	0	1	0	1	0	1
	1	0	1	1	1	0	1	0	1
niet gebruikt	1	1	0	0	—	—	—	—	—
	1	1	0	1	—	—	—	—	—
	1	1	1	0	—	—	—	—	—
	1	1	1	1	—	—	—	—	—

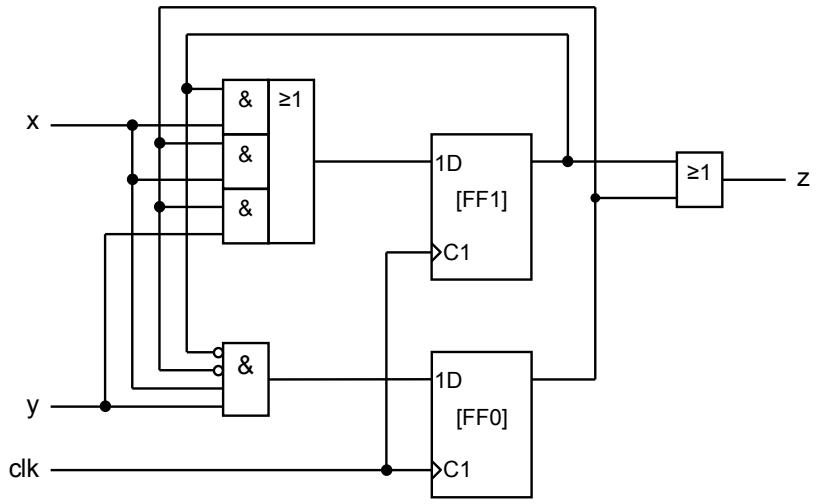
dan *equivalente toestanden* genoemd. De machine heeft dan een gedefinieerde werking in alle toestanden.

De functies voor de D-ingangen en de uitgang kunnen met behulp van Karnaughdiagrammen of Quine-McCluskey worden gevonden (dit wordt aan de lezer overgelaten). De functies zijn als volgt:

$$\begin{aligned} D_1^n &= Q_1^n \cdot x^n + Q_0^n \cdot x^n + Q_0^n \cdot y^n \\ D_0^n &= \overline{Q_1^n} \cdot \overline{Q_0^n} \cdot x^n \cdot y^n \\ z^n &= Q_1^n + Q_0^n \end{aligned} \tag{11.4}$$

Het schema van de Moore-machine is te vinden in figuur 11.12. De logica van de opvolgertoestand bestaat uit een aantal poorten, de uitgang wordt gevormd door een OR-poort. Het is duidelijk te zien dat de uitgang alleen afhankelijk is van de waarde van de flipflops oftewel de toestand van de machine.

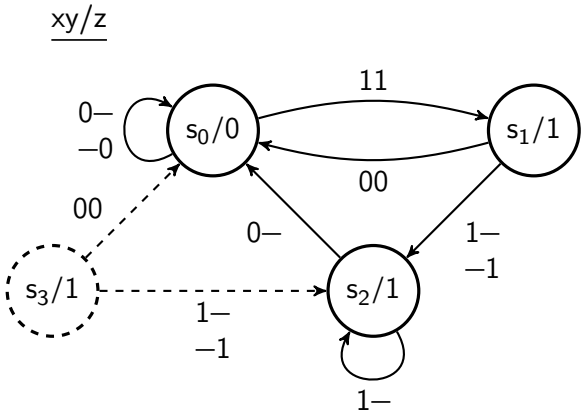
Nu we de toestandsfuncties en uitgangsfunctie hebben uitgewerkt, kunnen we voorspellen wat de machine doet als deze toevallig in de ongebruikte toestand terecht komt. Hiertoe vullen we voor de toestandsbits $Q_1Q_0 = 11$ en voor de ingangen x en y alle mogelijke combinaties in. De gegevens zijn te vinden in tabel 11.5. Het blijkt dat de machine in veel gevallen naar toestand s_2 gaat en dat de uitgangswaarde 1 is. We kunnen nu het originele toestandsdiagram uitbreiden met de ongebruikte toestand. Dit is te zien in figuur 11.13. Voor zowel het toestandsgedrag als de uitgangswaarde is toestand s_3 (we noemen de ongebruikte toestand voor het gemak maar even s_3) identiek aan toestand s_1. De toestanden s_1 en s_3 zijn dus *equivalente toestanden*.



Figuur 11.12: *Schema van de Moore-machine.*

Tabel 11.5: *Waarheidstabel voor de ongebruikte toestand 11.*

x	y	opvolgertoestand	toestandsbits	z
0	0	s_0	00	1
0	1	s_2	10	1
1	0	s_2	10	1
1	1	s_2	10	1

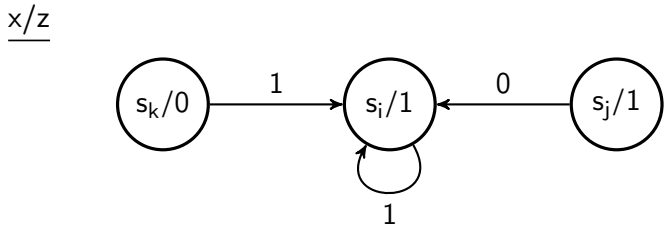


Figuur 11.13: *Het toestandsdiagram van een Moore-machine na invulling van de ongebruikte toestand.*

11.7 De one-hot toestandscodering

Voor het vinden van de toestandsfuncties bij one-hot codering wordt de bovengenoemde procedure onwerkbaar als er meer dan vier toestanden gebruikt worden. Bij een one-hot codering wordt namelijk voor elke toestand één flipflop gereserveerd. Voor tien toestanden zijn dus tien flipflops nodig. Het gevolg is dat de machine veel ongebruikte toestanden heeft. Dat levert wel voordeel op bij het minimaliseren van de toestandsfuncties en uitgangsfuncties.

Gelukkig kunnen de toestandsfuncties en uitgangsfuncties bij one-hot codering gemakkelijk uit het toestandsdiagram gevonden worden. In figuur 11.14 is (een deel) van een toestandsdiagram te zien. Bij het vinden van de functie voor een toestand (dus één flipflop) gaan we uit van alle overgangscondities *naar de toestand toe*. Elke overgang komt overeen met een productterm uit de functie.



Figuur 11.14: *Voorbeeld toestandsdiagram voor one-hot codering.*

Bekijken we toestand s_i dan zien we dat deze toestand wordt bereikt door drie overgangen. Om van toestand s_k naar toestand s_i te gaan, moet toestandsbit $Q_k = 1$ zijn en ingang x ook 1. De bijbehorende productterm is $Q_k^n \cdot x^n$. De overgang van s_j naar s_i wordt gerealiseerd door productterm $Q_j^n \cdot \overline{x^n}$. Er is ook een overgang van de toestand naar zichzelf. De productterm die daar bij hoort is $Q_i^n \cdot x^n$. De gehele functie is dan:

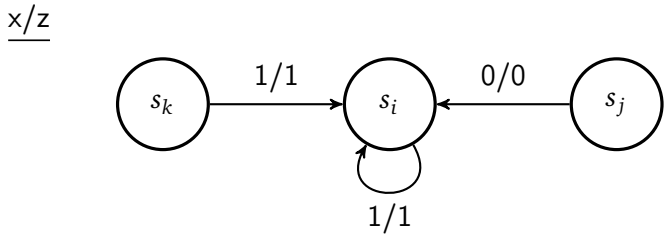
$$D_i^n = Q_k^n \cdot x^n + Q_i^n \cdot x^n + Q_j^n \cdot \overline{x^n} \tag{11.5}$$

Voor de uitgangsfunctie is het noodzakelijk om alle toestanden waar de uitgang 1 is te sommeren met een OR-functie (merk op dat het hier een Moore-machine betreft). Dus:

$$z^n = Q_i^n + Q_j^n \tag{11.6}$$

In figuur 11.15 is een deel van een Mealy-machine te zien. De toestandslogica wordt op dezelfde wijze bepaald als bij het vorige voorbeeld. De uitgangswaarde is echter afhankelijk van de toestand en van de ingangswaarde. Zo is in het voorbeeld te zien dat als de machine in toestand s_k staat, de uitgang 1 wordt als de ingang 1 is. Alleen wanneer de uitgangswaarde 1 is, behoort de productterm tot de uitgangsfunctie. Voor z wordt de functie:

$$z^n = Q_k^n \cdot x^n + Q_i^n \cdot x^n \tag{11.7}$$



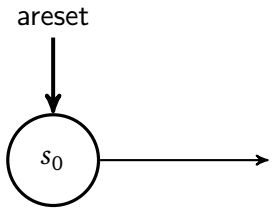
Figuur 11.15: *Voorbeeld toestandsdiagram voor one-hot codering.*

11.8 Reset-implementatie

Een resetingang behoort tot de noodzakelijke faciliteiten van een digitale schakeling. Bij het aanzetten is de toestand van een schakeling meestal niet gedefinieerd. Een power-on-reset zorgt er dan voor dat de schakeling in een van te voren aangewezen toestand terecht komt: de resettoestand. Een reset kan op twee manieren ingebouwd worden:

- *Via combinatoriek*. Het resetsignaal wordt in feite behandeld als een gewoon datasignaal. Dit heet een synchrone reset of *synchrone clear*.
- *Rechtstreeks*. Het resetsignaal wordt direct op de flipflops gezet en werkt buiten de klok om. Dit wordt een *asynchrone reset* genoemd. Een asynchrone reset is sterk storingsgevoelig en moet met zorg gebruikt worden.

Bij machines wordt vaak een *dominante reset* ingebouwd. Vanuit elke toestand wordt direct naar de resettoestand gesprongen indien de reset geactiveerd wordt. Om het aantal pijlen in het toestandsdiagram te beperken, wordt dit meestal aangegeven met een dubbele of dikke pijl. Zie figuur 11.16.

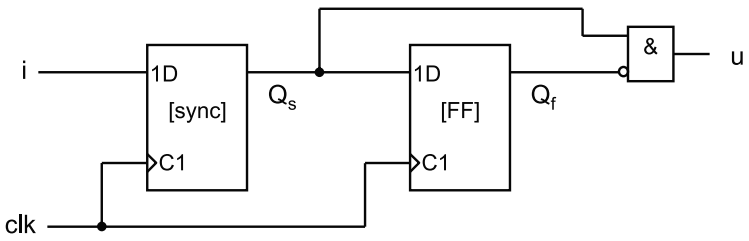


Figuur 11.16: *Voorbeeld van het aangeven van een dominante reset.*

Het heeft de voorkeur om elke schakeling met een *reset synchronizer* uit te rusten.

11.9 Synchronisatie ingangen en uitgangen

Alle asynchrone ingangen van een digitaal systeem moeten gesynchroniseerd worden met de klok. Synchronisatie zorgt ervoor dat de machine Moore-uitgangen heeft. De synchronizer bevat nu een extra toestandsbit van de machine. Zie figuur 11.17. Dit geldt alleen voor machines die hun data van buitenaf aangeboden krijgen.

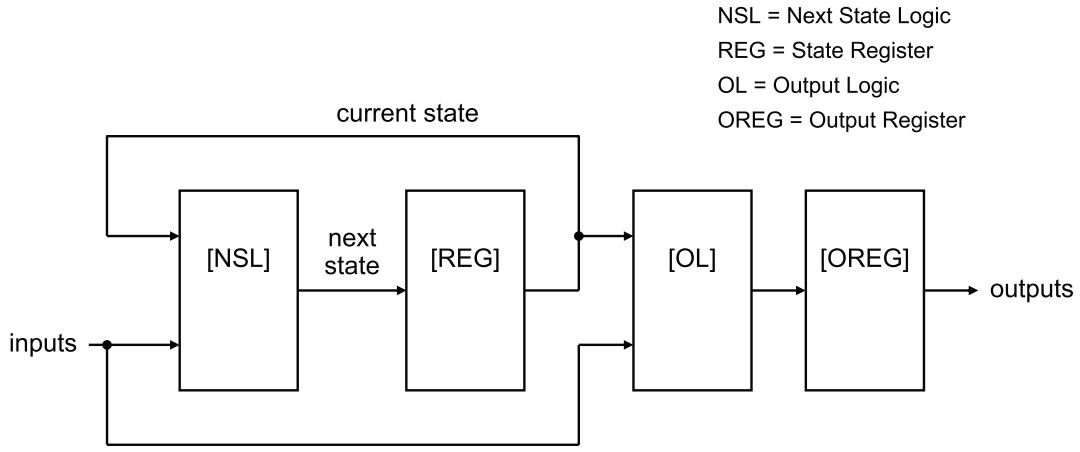


Figuur 11.17: *Mealy-machine met synchronizer zorgt voor Moore-uitgangen.*

De functies van de flipflops en de uitgang zijn:

$$\begin{aligned} D_s^n &= i^n \\ D_f^n &= Q_s^n \\ u^n &= \overline{Q_f^n} \cdot Q_s^n \end{aligned} \tag{11.8}$$

Soms is het handig om achter de uitgangen nog een register te plaatsen. Dit is te zien in figuur 11.18. De uitgangswaarden kunnen namelijk meerdere malen veranderen direct na de actieve klokflank. Bij het aanbieden van deze uitgangen aan een systeem met een ander kloksignaal kan dit problemen geven (o.a. metastabiliteit). Het register moet dan wel *single transition* zijn, de uitgang moet in één keer van waarde veranderen. Alle moderne flipflops doen dat. Andere namen voor gesynchroniseerde uitgangen zijn: registered output, clocked outputs en pipelined outputs. Gesynchroniseerde uitgangen zorgen ervoor dat de machine altijd van het Moore-type is. Het flipflops van het uitgangsregister kunnen namelijk gezien worden als toestandsbits. Het nadeel is dat de juiste uitgangswaarde één klokcyclus later beschikbaar is.



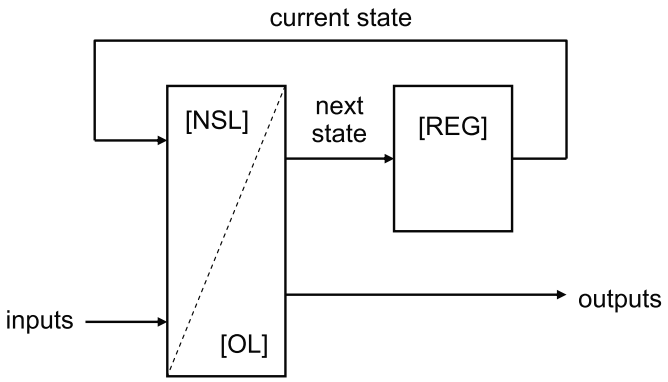
Figuur 11.18: *Algemeen model voor een toestandsmachine met gesynchroniseerde uitgangen.*

11.10 Alternatieve representaties toestandsmachines

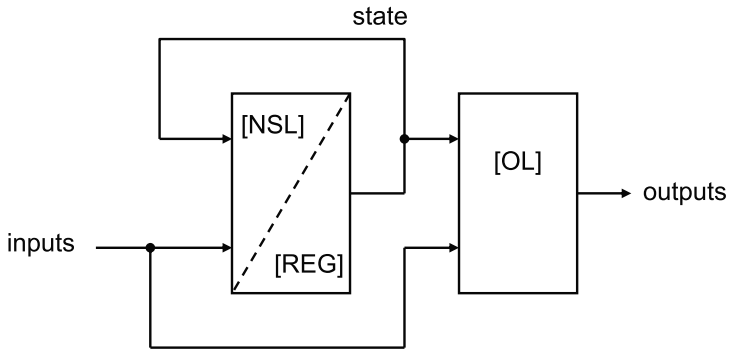
In deze paragraaf behandelen we twee alternatieve representaties van toestandsmachines. De modellen zijn van belang bij het beschrijven van toestandsmachines in VHDL.

Een alternatieve representatie van een Mealy-machine is te zien in figuur 11.19. Hier zijn de twee stukken logica die de opvolgertoestand en de uitgangswaarden bepalen gecombineerd als één stuk combinatoriek. Een Moore-machine kan alleen op deze manier gerepresenteerd worden door expliciet aan te geven dat de uitgangen alleen afhankelijk zijn van de toestand.

In figuur 11.20 zijn de combinatoriek voor de opvolgertoestand en het toestandsregister samengenomen. De huidige toestand en opvolgertoestand zijn nu niet gescheiden te herkennen en er wordt gesproken over de toestand. Dit model kan alleen gebruikt worden bij het beschrijven van een machine in VHDL.



Figuur 11.19: *Alternatief model voor een Mealy-machine.*



Figuur 11.20: *Alternatief model voor een Mealy-machine.*

11.11 Toestandsmachines in VHDL

Voor het beschrijven van toestandsmachines in VHDL wordt uitgegaan van het drie-proces-model voor de modellen in figuren 11.2 en 11.3 en het twee-proces-model in de figuren 11.19 en 11.20:

- *Drie-proces-model.* Er is één proces voor het bepalen van de opvolgertoestand (combinatoriek), één proces voor opslag van de (huidige) toestand (sequentiëel) en één proces voor het bepalen van de uitgangswaarden (combinatoriek).
- *Twee-proces-model.* Er is één proces voor het bepalen van de nieuwe toestand en de uitgangswaarden (combinatoriek) en één proces voor opslag van de (huidige) toestand (sequentiëel).
- *Twee-proces-model (alternatief).* Er is één proces voor het bepalen en opslag van de nieuwe toestand (sequentiëel) en één proces voor het bepalen van de uitgangswaarden (combinatoriek).

In listing 11.1 is de algemene opzet te zien voor een beschrijving van een toestandsmachine volgens het drie-proces-model. Het proces voor het bepalen van de opvolgertoestand en de uitgangswaarden zijn combinatorisch en zijn gevoelig voor de huidige toestand en de ingangen. Het proces voor de opslag van de huidige toestand is gevoelig voor de klok en de asynchrone reset. Voor het beschrijven van de combinatoriek en geheugen kunnen de technieken uit een eerder hoofdstuk worden toegepast.

```
architecture fsm_3proc of machine is
begin

    combstate: process (current_state, inputs) is
        -- beschrijving opvolgertoestand (NSL)
    end process;

    reg: process (clk, areset) is
        -- beschrijving state register (REG)
    end process;

    combout: process (current_state, inputs) is
        -- beschrijving uitgangen (OL)
    end process;

end fsm_3proc;
```

Listing 11.1: *Algemeen opzet voor het drie-proces-model.*

```
architecture fsm_2proc of machine is
begin

    comb: process (current_state, inputs) is
        -- beschrijving opvolgertoestand (NSL)
        -- beschrijving uitgangen (OL)
    end process;

    reg: process (clk, areset) is
        -- beschrijving state register (REG)
    end process;

end fsm_2proc;
```

Listing 11.2: *Algemeen opzet voor het twee-proces-model.*

```
architecture fsm_2proc_alt of machine is
begin

    combreg: process (clk, areset) is
        -- beschrijving opvolgertoestand onder klokflanksturing (NSL/REG)
    end process;

    combout: process (state, inputs) is
        -- beschrijving uitgangen (OL)
    end process;

end fsm_2proc_alt;
```

Listing 11.3: *Algemeen opzet voor het twee-proces-model (alternatief).*

De opzet van het twee-proces-model is te zien in listing 11.2. Hier is het combinatorische proces gevoelig voor de huidige toestand en de ingangswaarden. De beschrijving van

het toestandsregister is hetzelfde als bij het drie-proces-model.

In het alternatieve twee-proces-model in listing 11.3 zijn de beschrijvingen voor de opvolgertoestand en het toestandsregister geïntegreerd. Het proces is alleen gevoelig voor de klok en de asynchrone reset omdat de toekenningen (het bepalen van de opvolgertoestand) onder klokflanksturing valt. We zullen later zien dat dit model uitermate geschikt is voor het integreren van allerlei rekenkundig en logische bewerkingen zoals tellen en schuiven.

Eén van de stappen uit het ontwerpproces is het toekennen van codecombinaties aan de toestanden. VHDL kent de mogelijkheid om een eigen type te definiëren waarmee dit symbolisch kan worden gedaan. Toekenningen en testen kunnen gedaan worden op het type, zie listing 11.4. Te zien is dat met een **`if`**-statement getest en toegekend kan worden, maar het is gebruikelijk om een **`case`**-statement te gebruiken. Er moet toch voor iedere toestand een opvolger en uitgangswaarden worden bepaald en het maakt de code goed leesbaar.

```
type state_type is (s0, s1, s2, got_one, bingo, power_on);
signal current_state, next_state : state_type;

    -- With IF-statements
    if current_state = s0 then
        next_state <= bingo;
    elsif current_state = s1 then
        next_state <= got_one
    else
        next_state <= s0;
    end if;

    -- With CASE-statement
    case current_state is
        when s0 => next_state <= bingo;
        when s1 => next_state <= got_one;
        ...
        when others => next_state <= s0;
    end case;
```

Listing 11.4: *Voorbeeld van toestandsenumeratie.*

Het gebruik van een enumeratie voor de toestanden heeft enkele voordelen. Zo kan bij synthese de synthesizer zelf een goede codering bepalen. Bij (functionele) simulatie worden de elementen uit de enumeratie afgebeeld. Let erop dat bij het initialiseren van de simulatie-omgeving de signalen `current_state` en `next_state` de meeste linkse element krijgen. Het is ook mogelijk om de codering bijvoorbeeld als zuiver binair of one-hot in te stellen. De codering kan ook expliciet door de gebruiker opgegeven worden.

In listing 11.5 is de architecture te zien van de beschrijving van de Mealy-machine. Het begint met de declareren van het type `state_type`. Daarna worden de twee signalen `current_state` en `next_state` gedeclareerd voor de opslag van de huidige toestand en de opvolgertoestand.

```
architecture fsm_3proc of mealy_example is
-- State types
type state_type is (s0, s1);
-- Internal current state and next state
signal current_state, next_state : state_type;
begin
    -- Combinational process generates the next state
    combstate: process (current_state, i) is
    begin
        case current_state is
            when s0 => if i = '0' then
                           next_state <= s0;
                       else
                           next_state <= s1;
                       end if;
            when s1 => if  i = '0' then
                           next_state <= s0;
                       else
                           next_state <= s1;
                       end if;
            when others => next_state <= s0;
        end case;
    end process;

    -- Sequential process generates the state register
    reg: process (clk, areset) is
    begin
        if areset = '1' then
            current_state <= s0;
        elsif rising_edge(clk) then
            current_state <= next_state;
        end if;
    end process;

    -- Combinational process generates the output
    combout: process (current_state, i) is
    begin
        case current_state is
            when s0 => if i = '0' then
                           u <= '0';
                       else
                           u <= '1';
                       end if;
            when s1 => u <= '0';
            when others => u <= 'X';
        end case;
    end process;

end architecture;
```

Listing 11.5: *VHDL-beschrijving van de Mealy-machine met drie processen.*

Het eerste combinatorische proces is voor het bepalen van de opvolgertoestand. Het proces is gevoelig voor de huidige toestand en de ingang. Door middel van een **case**-

statement worden alle toestanden getest. Bij elke toestand wordt met behulp van een **if**-statement de opvolgertoestand bepaald. Het statement **when others** is nodig om een opvolgertoestand te bepalen als de machine in geen van de gedefinieerde toestanden terecht komt. In de praktijk kan hier op diverse wijze mee worden omgegaan. Zo kan de synthesizer alle ongebruikte toestanden als don't care beschouwen of kan het hardware genereren om de resettoestand altijd als opvolger aan te wijzen. Dit is vaak door de ontwerper in te stellen.

Het proces voor het opslaan van de huidige toestand is opgebouwd rond een asynchrone reset en een klokflank. Als de reset geactiveerd is, gaat de machine naar toestand s_0 (dat is in de eerdere toestandsdiagrammen niet aangegeven). Als er een opgaande klokflank wordt gedetecteerd, wordt de nieuwe toestand ingeklokt.

Het derde proces, voor het bepalen van de uitgangswaarde, is net als het eerste proces opgebouwd rond een **case**-statement. Met behulp van een **if**-statement wordt in toestand s_0 de uitgangswaarde bepaald. In toestand s_1 is de uitgangswaarde altijd 0.

In listing 11.6 is de beschrijving te zien van de combinatoriek voor het bepalen van de opvolgertoestand en de uitgangswaarde samen in één proces. Ook hier is weer gebruik gemaakt van een **case**-statement. De beschrijving van het toestandsregister is hetzelfde als in listing 11.5.

```
1   -- Combinational process generates the
2   -- next state and the output
3   comb: process (current_state, i) is
4   begin
5       case current_state is
6           when s0 => if i = '0' then
7                          next_state <= s0;
8                          u <= '0';
9                      else
10                         next_state <= s1;
11                         u <= '1';
12                     end if;
13          when s1 => if  i = '0' then
14                         next_state <= s0;
15                         u <= '0';
16                     else
17                         next_state <= s1;
18                         u <= '0';
19                     end if;
20          when others => next_state <= s0;
21                         u <= 'X';
22      end case;
23  end process;
```

Listing 11.6: *VHDL-beschrijving van de Mealy-machine met twee processen.*

Het alternatieve twee-proces-model demonstreren we aan de hand van de Moore-machine. Zie hiervoor listing 11.6. Er is één proces voor het beschrijven van de opvolgertoestand en het toestandsregister. Merk op dat het proces gevoelig is voor de klok en de reset en niet voor de huidige toestand en de ingangen. De toekenningen vinden immers plaats

onder klokflanksturing (behalve de asynchrone reset natuurlijk). Verder is te zien dat bij het statement **`when others`** het keyword **`null`** is gebruikt. Dit geeft aan dat er geen gedrag is gedefinieerd. Bij simulatie blijft de machine dan in de toestand hangen waarin het gekomen is. Het gedrag bij synthese is afhankelijk van de instellingen. Dat heeft vervelende consequenties voor de werking van de machine. In de praktijk is het beter om de machine naar de reset-toestand te laten gaan. Soms is het mogelijk om **`null`** als don't care te beschouwen. Dit moet dan bij de synthese worden ingesteld.

Het is mogelijk om in de VHDL-beschrijving de toestandscodering op te nemen. Hiervoor wordt een *pragma* gebruikt. Een voorbeeld is te zien in listing 11.8. Er kan een willekeurige codering worden ingesteld zoals de binaire telcode, Gray-code, one-hot of gewoon een willekeurige code. De synthesizer zal deze codering gebruiken bij het realiseren van de schakeling.

11.12 Patroonherkenners

Een typische sequentiële machine is een herkenningsautomaat of *patroonherkenner*. Herkenningsautomaten spelen een belangrijke rol in het vakgebied *berekenbaarheidstheorie* (theory of computation) dat zich bezighoudt met hoe efficiënt problemen kunnen worden opgelost door middel van een algoritme (en of ze eigenlijk wel oplosbaar zijn). De automaten spelen een belangrijke rol bij het opstellen en herkennen van talen (*languages*). Denk hierbij aan het *parsen* van computertalen of zogenoemde *regular expressions*. We zullen in deze paragraaf enkele voorbeelden van patroonherkenners bespreken.

In figuur 11.21 is een herkenner te zien voor het eenmalig herkennen van het patroon 110. Na een (asynchrone) reset bevindt de herkenner zich in toestand s_0. De herkenner blijft in toestand s_0 zolang op de ingang een 0 wordt aangeboden. Als een 1 wordt gedetecteerd, gaat de herkenner naar toestand s_1. Vanuit toestand s_1 wordt naar toestand s_2 gegaan als op de ingang een 1 wordt aangeboden, bijvoorbeeld als ..11.. in het patroon voorkomt. Een 0 zorgt ervoor dat de herkenner teruggaat naar toestand s_0. Als in toestand s_2 een 1 wordt aangeboden, blijft de herkenner in toestand s_2. Denk hierbij aan het patroon ..111.. Wordt een 0 op de ingang gedetecteerd dan gaat de herkenner naar toestand s_3. De herkenner geeft in deze toestand een 1 op de uitgang af en blijft in toestand s_3. In alle andere toestanden geeft de herkenner een 0 af. Linksonder in de figuur is een voorbeeld van een ingangs- en uitgangsreeks gegeven.

De hierboven geschetste herkenner stopt na het eenmaal herkennen van het patroon. We kunnen de herkenner het patroon doorlopend laten herkennen door de overgangscondities van toestand s_3 aan te passen. Dit is te zien in figuur 11.22.

Als in toestand s_3 een 1 wordt gedetecteerd, gaat de herkenner naar toestand s_1. Denk hierbij aan het patroon ..1101.. waarbij de laatste 1 onderdeel is van het opnieuw te herkennen patroon. Een 0 zorgt ervoor dat de herkenner naar toestand s_0, bijvoorbeeld bij het patroon ..1100..

In figuur 11.23 is het toestandsdiagram van de herkenner nogmaals te zien, maar nu ontworpen als Mealy-machine. We merken op dat de herkenner slechts drie toestanden heeft, één minder dan bij de Moore-variant. Daarnaast is de uitgangswaarde 1 ook eerder

```
architecture fsm_2proc_alt of moore_example is
-- State types
type state_type is (s0, s1, s2);
-- Internal (current) state
signal state : state_type;
begin
    -- Sequential process generates the next state and state resgister
    combstate: process (clk, areset) is
    begin
        if areset = '1' then
            state <= s0;
        elsif rising_edge(clk) then
            case state is
                when s0 => if x = '0' or y = '0' then
                               state <= s0;
                           else
                               state <= s1;
                           end if;
                when s1 => if x = '0' and y = '0' then
                               state <= s0;
                           else
                               state <= s2;
                           end if;
                when s2 => if x = '0' then
                               state <= s0;
                           else
                               state <= s2;
                           end if;
                when others => null;
            end case;
    end if;
    end process;

    -- Combinational process generates the output
    combout: process (state) is
    begin
        case state is
            when s0 => z <= '0';
            when s1 => z <= '1';
            when s2 => z <= '1';
            when others => null;
        end case;
    end process;

end architecture;
```

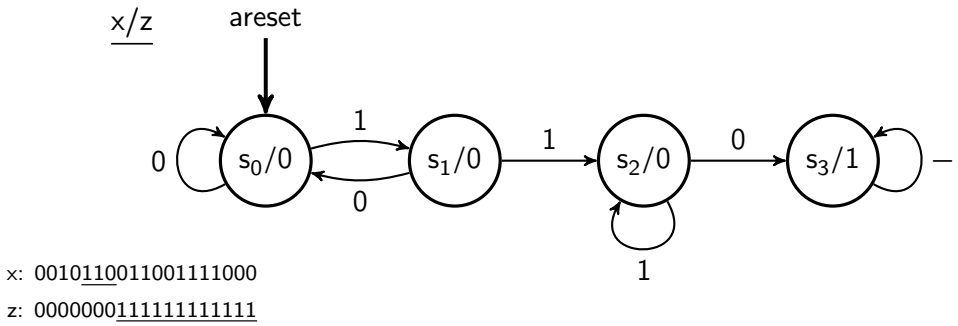
Listing 11.7: *VHDL-beschrijving van de Moore-machine met twee processen (alternatief).*

beschikbaar dan bij de Moore-machine.

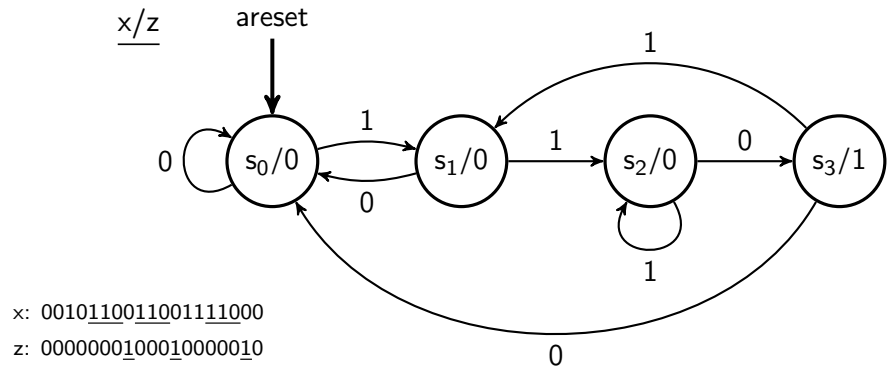
We kunnen het zoekproces iets ingewikkelder maken door bloksgewijs te zoeken naar het patroon 110. Dit is te zien in figuur 11.24. Deze Moore-machine wacht in toestand s_0 totdat een 1 op de ingang wordt gedetecteerd. Daarna worden altijd drie ingangswaarden "afgetikt", ook als het patroon 110 hier niet in voorkomt. De toestanden s_0 t/m s_3 geven de

```
architecture fsm_3proc of moore_example is
-- State types
type state_type is (s0, s1, s2);
-- !!!FORCE STATE ENCODING!!!
attribute ENUM_ENCODING : string;
attribute ENUM_ENCODING of state_type : type is "00 01 10";
--attribute ENUM_ENCODING of state_type : type is "11 10 00";
--attribute ENUM_ENCODING of state_type : type is "100 010 110";
--attribute ENUM_ENCODING of state_type : type is "001 010 100";
-- Internal current state and next state
signal current_state, next_state : state_type;
```

Listing 11.8: *Voorbeeld van een geforceerde toestandscodering.*

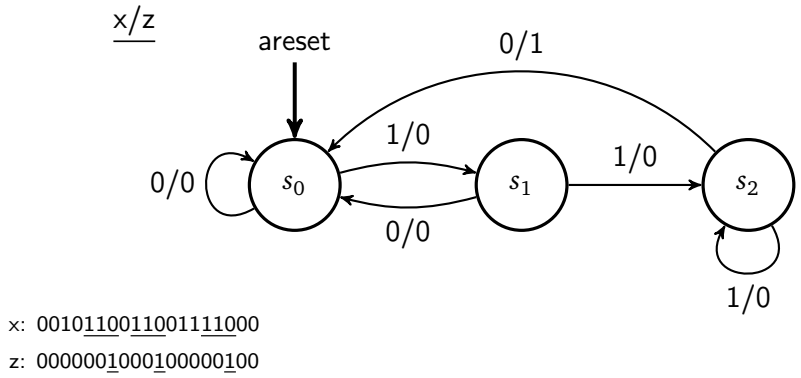


Figuur 11.21: *Herkenner voor het eenmalig herkennen van het patroon 110 (Moore).*



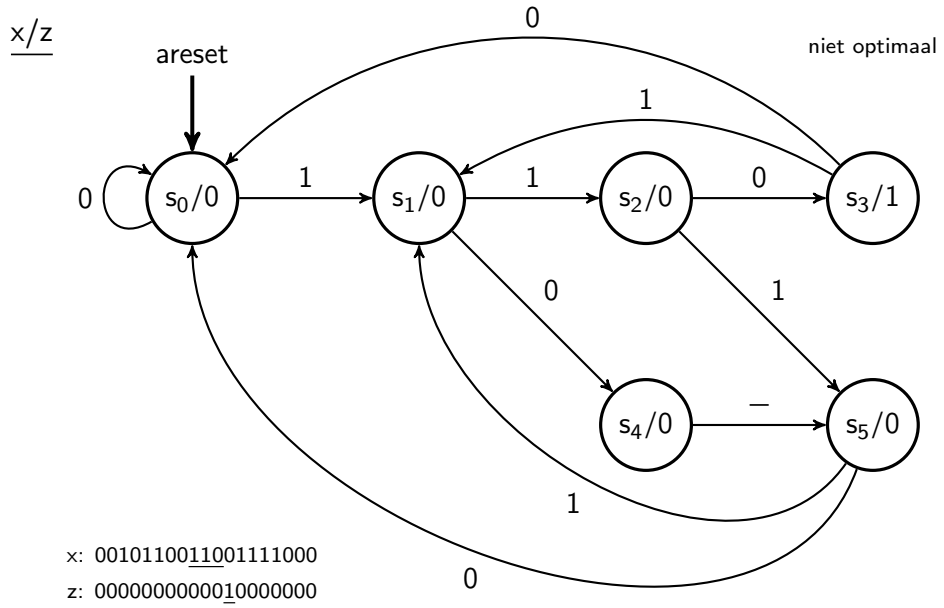
Figuur 11.22: *Herkenner voor het doorlopend herkennen van het patroon 110 (Moore).*

werking van de machine aan als het patroon 110 in de bitstroom voorbij komt. Toestand s_4 wordt bereikt als na de eerste 1 een 0 wordt gedetecteerd. Er zijn nu twee bits bekeken en het patroon kan niet meer voorkomen in de drie bits. Daarom gaat de machine altijd van toestand s_4 naar s_5 (aangegeven door de don't care specificatie). Bij herkenning van het patroon 111 gaat de machine naar toestand s_5. Na het verwerken van de drie ingangswaarden gaat de machine naar toestand s_0 of s_1 als de ingangswaarde een 0 respectievelijk een 1 is. Alleen in toestand s_3 wordt een 1 afgegeven. In alle andere toestanden wordt een 0 afgegeven. Opgemerkt moet worden dat deze machine niet optimaal is voor wat betreft het aantal toestanden. Het is mogelijk om een machine te



Figuur 11.23: *Herkenner voor het doorlopend herkennen van het patroon 110 (Mealy).*

ontwerpen met vijf toestanden die zich identiek gedraagt als de machine in figuur 11.24.

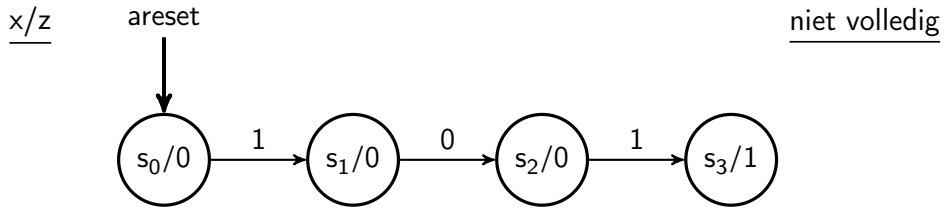


Figuur 11.24: *Herkenner voor het bloksgewijs herkennen van het patroon 110 (Moore).*

Als laatste voorbeeld ontwerpen we een patroonherkenner voor het overlappend herkennen van het patroon 101. Een voorbeeld van overlapping is ..10101.. De middelste 1 hoort zowel bij het eerste patroon als bij het laatste patroon. De herkenner wordt als een Moore-machine ontworpen.

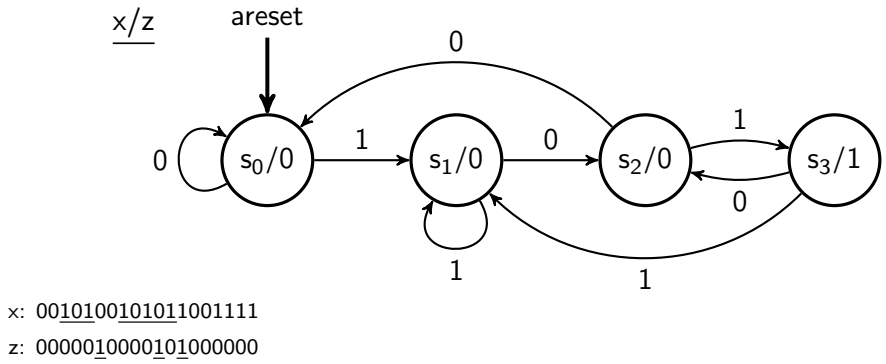
We begin met het opstellen van het toestandsdiagram door eerst de toestanden te definiëren voor het te herkennen patroon. We stellen de toestanden s_0 t/m s_3 op en trekken de pijlen voor het herkennen van 1, 0 en 1. Dit is te zien in figuur 11.25. De machine is in toestand s_0 na een reset. Toestand s_1 wordt bereikt als een 1 gevonden is. Een 0 zorgt ervoor dat de machine naar toestand s_2 gaat. Als de machine in toestand s_2 staat is dus het patroon 10 gevonden. Na nog een 1 in de bitstroom komt de machine in toestand s_3. Het patroon is herkend zodat de machine een 1 afgeeft. In alle andere toestanden geeft

de machine een 0 af.



Figuur 11.25: *Eerste aanzet voor het overlappend herkennen van het patroon 101 (Moore).*

We hebben nu een eerste opzet voor het herkennen van 101. We vullen nu de ontbrekende overgangsvoorwaarden in. Dit is te zien in het volledige toestandsdiagram in figuur 11.26. Als in toestand s_0 een 0 wordt herkend, blijft de machine in toestand s_0. Er is dus een pijl naar de toestand zelf. In toestand s_1 wordt net zolang gewacht totdat er een 0 wordt gedetecteerd. Ook hier is dus een pijl naar de toestand zelf zolang er een 1 wordt gevonden. Een 0 in toestand s_s zorgt ervoor dat de machine weer naar toestand s_0 gaat, bijvoorbeeld bij het patroon ..100.. Als de machine in toestand s_3 staat, is het patroon herkend. De machine blijft dus tussen de toestanden s_2 en s_3 toggelen zolang het (deel-)patroon 10 wordt herkend, bijvoorbeeld bij ..1010101010.. Hier is goed de overlapping te zien. Een 1 in toestand s_3 zorgt ervoor dat de machine naar toestand s_1 gaat, bijvoorbeeld bij het patroon ..1011.. waarbij de eerste drie bits het patroon vormen.



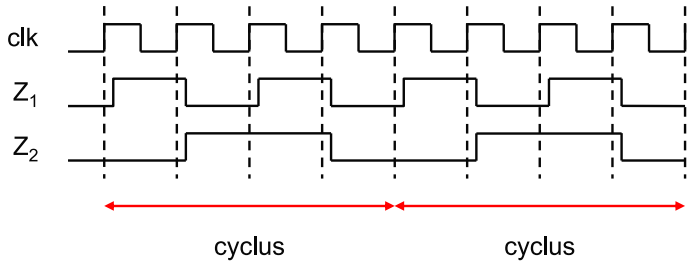
Figuur 11.26: *Herkenner voor het overlappend herkennen van het patroon 101 (Moore).*

11.13 Toestandsmachine op basis van timingdiagrammen

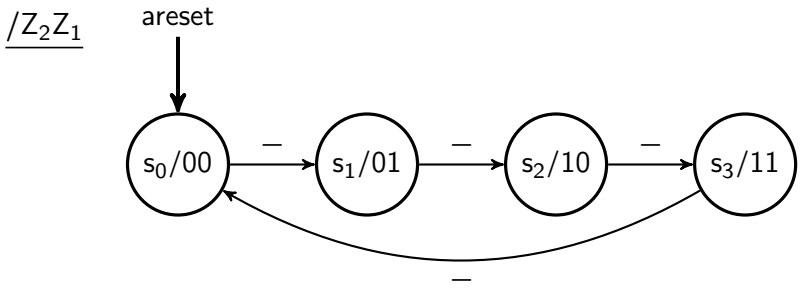
Een toestandsmachine is ook te beschrijven met behulp van een timingdiagram. Dit is te zien in figuur 11.27.

De waarden van de uitgangen herhalen zich elke vier klokcycli. De machine is dus met vier toestanden te beschrijven. Er is geen ingang. De uitgangen doorlopen de cyclus: $Z_2Z_1 = 00$, $Z_2Z_1 = 01$, $Z_2Z_1 = 10$ en $Z_2Z_1 = 11$. Goed beschouwd is dit het gedrag van een teller. Het bijbehorende toestandsdiagram is te zien in figuur 11.28.

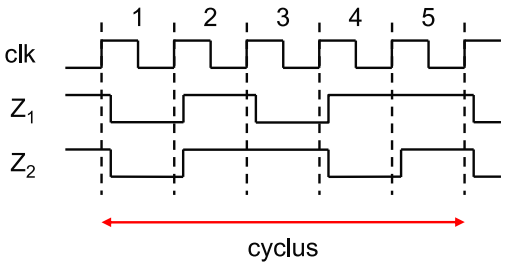
In figuur 11.29 is nog een timingdiagram te zien. Maar nu zijn de uitgangen tijdens twee klokcycli identiek. De uitgangswaarde $Z_2Z_1 = 11$ komt twee keer voor. De machine doorloopt de cyclus $Z_2Z_1 = 00$, $Z_2Z_1 = 11$, $Z_2Z_1 = 01$, $Z_2Z_1 = 10$ en $Z_2Z_1 = 11$.



Figuur 11.27: *Specificatie van de uitgangen van een toestandsmachine d.m.v. een timingdiagram.*

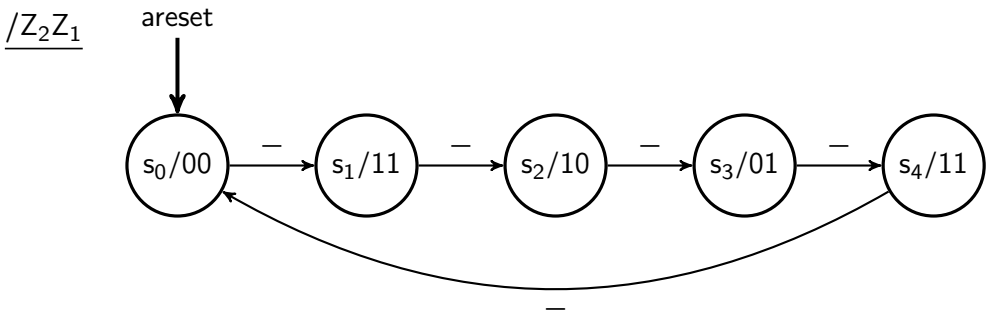


Figuur 11.28: *Toestandsdiagram van de machine op basis van een timingdiagram.*



Figuur 11.29: *Specificatie van de uitgangen van een toestandsmachine d.m.v. een timingdiagram.*

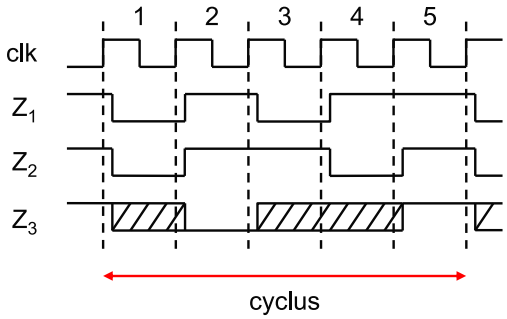
De waarden van de uitgangen herhalen zich elke vijf klokcycli. De machine is dus met vijf toestanden te beschrijven. Er is geen ingang. De machine is te ontwerpen met behulp van een teller en uitgangslogica. Het toestandsdiagram is te zien in figuur 11.30.



Figuur 11.30: *Toestandsdiagram van de machine op basis van een timingdiagram.*

Het is echter ook mogelijk om de machine zonder uitgangslogica te ontwerpen door

toevoeging van slechts één flipflop. Dit is te zien in figuur 11.31. De uitgang van de extra flipflop noemen we Z_3.



Figuur 11.31: *Specificatie van de uitgangen van een toestandsmachine d.m.v. een timingdiagram.*

Bij drie van de vijf standen is de waarde van Z_3 niet belangrijk en kan als don't care opgegeven worden, de waarde wordt namelijk niet uitgevoerd. Dit is in de figuur aangegeven door middel van een arcering.

De toestandscodering ligt gedeeltelijk vast. In de klokcyclus 1 zijn de uitgangen Z_1 en Z_2 beide logisch 0. Omdat deze uitgangscombinatie maar één keer in de cyclus voorkomt, maakt het niet uit wat de waarde van Z_3 is. We kunnen dus stellen dat $Z_3Z_2Z_1 = -00$ is. In de praktijk moet de don't care worden ingevuld als een 0 of een 1. We kunnen echter ook *twee* toestanden voor deze ene toestand gebruiken. We stellen dus dat $Z_3Z_2Z_1 = 000$ en $Z_3Z_2Z_1 = 100$ beide dezelfde toestand voorstellen. De twee toestanden zijn dus *equivalent*. Bij het ontwerpen van de hardware kan deze don't care tot minder logica leiden.

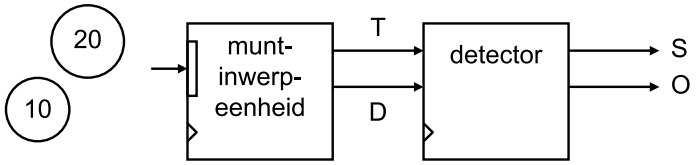
In klokcyclus 2 en 5 zijn de uitgangen Z_1 en Z_2 beide logisch 1. Nu is de waarde van Z_3 van belang om de twee toestanden uniek te coderen. We kiezen voor $Z_3 = 0$ in klokcyclus 2 en $Z_3 = 1$ in klokcyclus 5. De uitgangswaarden $Z_2Z_1 = 10$ en $Z_2Z_1 = 01$ komen slechts één keer voor, dus de waarde van Z_3 kan weer als een don't care worden opgegeven. Ook hier geldt dat er twee toestanden bij deze uitgangswaarden horen.

11.14 Snoepautomaat

In deze paragraaf behandelen we de besturing van een snoepautomaat. De snoepautomaat accepteert alleen muntstukken van 20 cent (T) en 10 cent (D)[1]. Er is 30 cent nodig om snoepgoed vrij te geven. Als er meer dan 30 cent wordt ingeworpen, moet de machine geen snoepgoed vrijgeven maar aanduiden dat er teveel geld is ingeworpen.

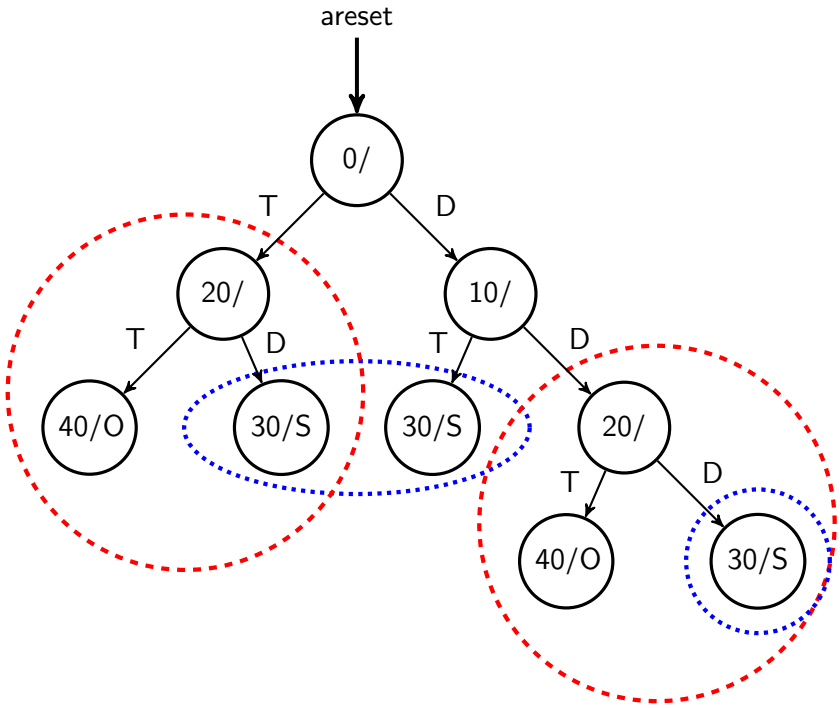
In figuur 11.32 is een schematische weergave van de munt-inwerpeenheid en de detector te zien. De munt-inwerpeenheid geeft twee signalen af, T voor een muntstuk van 20 cent en D voor een muntstuk van 10 cent. De eenheid kan slechts één munt te gelijk accepteren. Na inwerpen van een muntstuk wordt het betreffende signaal één klokcyclus geactiveerd. De signalen T en D kunnen dus nooit tegelijk geactiveerd worden. De detector zorgt voor het verwerken van de signalen T en D. Het signaal S wordt geactiveerd als 30 cent is ingeworpen. Het signaal O wordt geactiveerd als meer dan 30 cent is ingeworpen.

[1] T = twintig, D = dubbeltje



Figuur 11.32: *Schematische weergave van de snoepmachine.*

In figuur 11.33 is het toestandsdiagram getekend van de besturing van de snoepautomaat. We gaan hier uit van een wat gestileerd toestandsdiagram. Er zijn alleen pijlen getekend als er een daadwerkelijke toestandsverandering plaatsvindt. Pijlen naar de toestand zelf zijn niet getekend. Bij de pijlen staat alleen T of D vermeld omdat de machine maar één muntstuk tegelijk kan herkennen. In een toestand staat het tot dan toe ingeworpen bedrag vermeld. Alleen wanneer S of O geactiveerd is staat dat vermeld.



Figuur 11.33: *Toestandsdiagram voor het herkennen van 30 cent.*

Deze eerste opzet heeft negen toestanden. Na een reset komt de machine in toestand 0. Er is nog geen muntstuk ingeworpen. Na het inwerpen van 20 cent (T) gaat de machine naar de toestand 20 aan de linkerkant. Bij inworp van 10 cent (D) gaat de machine naar toestand 10 aan de rechterkant. We bouwen het toestandsdiagram verder uit. Elke keer als er een muntstuk wordt ingeworpen, gaat de machine naar een volgende toestand, totdat óf het juiste bedrag is ingeworpen óf teveel is ingeworpen. We hebben het toestandsdiagram opgebouwd als een *binaire boom*.

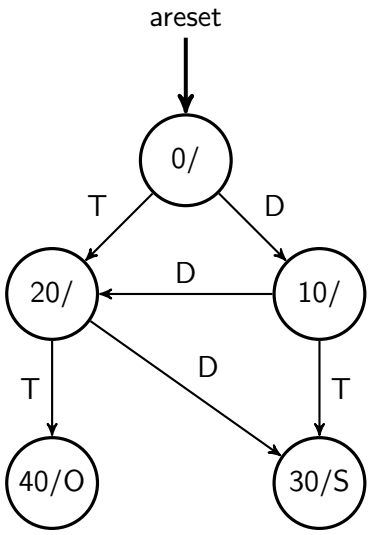
We kunnen echter wat opmerkingen maken over het toestandsdiagram. Zo zijn toestanden die een identiek bedrag voorstellen *equivalent*. Het maakt bijvoorbeeld niet uit of er eerst 20 cent worden ingeworpen en dan 10 cent of andersom. In beide gevallen levert dat 30 cent op. Het is ook mogelijk om drie keer 10 cent in te werpen. Ook dit levert 30

cent op. Verder is het duidelijk dat er nooit meer dan 40 cent kan worden ingeworpen. We hoeven dus niet verder te zoeken naar een hoger bedrag. We kunnen nu een aantal regels opstellen voor ingeworpen bedragen. Deze zijn te zien in vergelijking (11.9):

$$\begin{aligned} T+T &= 40 & T+D &= D+T \\ T+D &= 30 & D+D &= T \\ D+T &= 30 & & \\ D+D+T &= 40 & & \\ D+D+D &= 30 & & \end{aligned} \tag{11.9}$$

In figuur 11.33 zijn alle toestanden die 30 cent voorstellen omcirkeld. Deze kunnen vervangen worden door één toestand. Verder is te zien dat links de hele "boom" onder de begintoestand identiek is aan de boom rechts onder toestand 10 (aangegeven door de gestreepte cirkels). We kunnen dus de twee bomen vervangen door één boom.

Door equivalente toestanden samen te nemen, komen we uit op een toestandsdiagram met vijf toestanden, te zien in figuur 11.34.



Figuur 11.34: *Gereduceerd toestandsdiagram voor het herkennen van 30 cent.*

We hebben nu gezien dat het mogelijk is om *toestandsminimalisatie* uit te voeren. Het idee is om te zoeken naar equivalente toestanden. Twee of meer toestanden zijn equivalent als voor elke mogelijke reeks ingangswaarden dezelfde reeks uitgangswaarden wordt geproduceerd. Dat kunnen exact twee toestanden zijn of twee groepen van toestanden zoals te zien is in figuur 11.33. Een formele procedure is opgesteld door Paull en Unger [16]. Voor een groot aantal toestanden is deze procedure bewerkelijk. In de praktijk wordt toestandsminimalisatie weinig gebruikt, zeker als het aantal toestanden gering is. Een goede ontwerper zorgt meestal voor een minimale of bijna-minimale oplossing.

11.15 VHDL-beschrijving en testbench voor patroonherkenner

In deze paragraaf behandelen we de patroonherkenner uit figuur 11.21. De herkenner is een Moore-type toestandsmachine en herkent doorlopend het patroon 110. We beginnen met het presenteren van de code van de herkenner, te zien in listings 11.9 en 11.10. De herkenner heeft een klokingang (`clk`), een resetingang (`areset`) en een data-ingang (`x`). Er is één uitgang (`z`). In listing 11.9 is de entity-beschrijving te zien. Alle signalen zijn van het type `std_logic`. Tevens is te zien hoe de code becommentarieerd moet zijn. De eerste tien regels geven algemene informatie zoals bestandsnaam en auteur. Daarna volgt een korte beschrijving van wat de code (en dus de hardware) doet.

```
1  -- Filename:      herk110.vhd
2  -- Filetype:      VHDL code (behavior)
3  -- Date:          18 may 2011
4  -- Update:        8 nov 2016
5  -- Description:   A behavioral description of pattern recognizer
6  -- Author:        J. op den Brouw
7  -- State:         demo
8  -- Error:         -
9  -- Version:       1.1.1
10 -- Copyright:     (c)2016, De Haagse Hogeschool
11 --
12 -- This is an example of a behavioral description of a pattern
13 -- recognizer written as a finite state machine (FSM). It
14 -- recognizes the pattern 110 continuously while data bits are
15 -- shifted in sequentially. The output will be logic 1 for one
16 -- clock cycle following the trailing 0 on the input. The
17 -- finite state machine is of type Moore.
18
19 -- Preamble stuff
20 library ieee;
21 use ieee.std_logic_1164.all;
22
23 -- The entity
24 entity herk110 is
25     port (clk    : in std_logic;    -- clock input
26           areset : in std_logic;    -- asynchronous reset
27           x      : in std_logic;    -- the input of the FSM
28           z      : out std_logic); -- the output of the FSM
29 end entity herk110;
```

Listing 11.9: *Beschrijving van herkenner voor het patroon 110 (entity).*

Listing 11.10 laat de architecture van de herkenner zien. We beschrijven de machine volgens het twee-proces-model met één proces voor de opvolgertoestand en de uitgang en één proces voor het toestandsregister. Er zijn vier toestanden die we s_0, s_1, s_2 en s_3 noemen. We doen dit weer met een eigen enumeratietype.

In het combinatorisch proces maken we gebruik van zogenoemde *default assignments*; de opvolgertoestand en de uitgang krijgen eerst een standaard waarde toegekend. Zo zetten we de opvolgertoestand op s_0 en de uitgang op 0. Dat heeft enkele voordelen. Er hoeven alleen veranderingen van de opvolgertoestand en de uitgang aangegeven te worden. Als er geen toekenningen worden gedaan, blijven de standaard waarden gelden.

De code wordt hierdoor ook compacter. We zorgen er zo ook voor dat er geen latches worden gegenereerd als er bijvoorbeeld een **`else`** vergeten is.

```
architecture fsm of herk110 is
-- The FSM is constructed with four self named states.
type state_type is (s0, s1, s2, s3);
-- The current state and next state
signal current_state, next_state : state_type;
begin
    -- The combinational process generates the next state
    -- and output as a function of the current state and
    -- the input.
    comb: process (current_state, x) is
    begin
        -- Default values for next state and output
        z <= '0';
        next_state <= s0;
        -- For every state, describe the next state and the output
        case current_state is
            when s0 => if x = '1' then
                          next_state <= s1;
                       end if;
            when s1 => if x = '1' then
                           next_state <= s2;
                       end if;
            when s2 => if x = '0' then
                           next_state <= s3;
                       else
                           next_state <= s2;
                       end if;
            when s3 => if x = '1' then
                           next_state <= s1;
                       end if;
                       z <= '1';
            when others => null;
        end case;
    end process comb;

    -- The sequential process is used to describe the
    -- state register including the asynchronous reset
    reg: process (clk, areset, next_state) is
    begin
        if areset = '1' then
            current_state <= s0;
        elsif rising_edge(clk) then
            current_state <= next_state;
        end if;
    end process reg;

end fsm;
```

Listing 11.10: *Beschrijving van herkenner voor het patroon 110 (architecture).*

Vaak wordt gedacht dat (bijvoorbeeld) de uitgang kort een 0 vertoond. Dat is echter niet het geval. De toekenning gebeurt namelijk altijd in de (simulatie-)toekomst. Denk

hierbij aan de delta delay. In toestand s_3 wordt de uitgang op 1 gezet. Bij simulatie en synthese is de standaard toekenning van 0 dus niet te zien. Hetzelfde geldt voor de opvolgertoestand. Alleen als `x` 1 is, wordt een toekenning `next_state <= s1` gedaan. Zo niet, dan blijft de standaard waarde gelden.

In listing 11.11 is de entity en een deel van de architecture van de testbench te zien. Opvallend is dat de entity leeg is; er zijn geen ingangen en uitgangen gedefinieerd. Dat komt omdat de testbench zelf niet ingebed is in een hogere hiërachielaag.

Er worden twee tijdconstanten gedefinieerd, één voor de klokperiode met een tijd van 10 ns en een korte vertraging van 2 ns. Die laatste zorgt ervoor dat toekenningen aan de ingang iets na de opgaande klokflank valt. Dan is de verandering goed te zien in de simulator. De signalen `sim_clk`, `sim_areset`, `sim_x` en `sim_z` zijn bedoeld als *monitoring signals*; deze worden in de simulator weergegeven. Om de herkenner te gebruiken in de simulator moet een componentdeclaratie gedaan worden. In feite is dit hetzelfde als de entity-declaratie alleen is het keyword **`entity`** vervangen door **`component`**.

In lising 11.12 is de instantiëring, klokgeneratie en datageneratie te zien. De instantiëring gebeurt op gebruikelijke wijze middels een *port map*. De signalen van de herkenner worden gekoppeld aan de monitoring signals met behulp van een *port association list*.

De klokgenerator is een eeuwig durend proces. Er is geen sensitivity list nodig want er worden **`wait`**-statements gebruikt. De klok begint op 0, daarna wordt er een halve periodetijd gewacht. Vervolgens wordt de klok op 1 gezet en wordt er weer een halve periodetijd gewacht. Dit levert een kloksignaal op met een duty cycle van 50%.

Voor het genereren van de data op de ingang gebruiken we enkele fraaie taalconstructies. Zo wordt er eerst een constante gedeclareerd waarin alle waarden zijn opgenomen die aan de ingang moeten worden aangeboden. De constante is een `std_logic_vector` zonder bereik, een zogenoemde *unconstrained vector*. Daarna wordt een variabele gedeclareerd en wordt de waarde van de constante aan de variabele toegekend. De variabele gebruiken we om de bits een voor een aan de ingang toe te kennen. Als we een ander patroon willen gebruiken, hoeven we alleen de constante te veranderen. Ook de lengte mag wijzigen, alles wordt automatisch uitgerekend.

We beginnen de simulatie met het op 0 zetten van de ingang. Daarna wordt de reset twee klokcycli op 1 gezet en daarna twee klokcycli op 0. De machine is nu gereset en staat in toestand s_0. Er wordt nog even gewacht tot er een opgaande flank wordt gedetecteerd.

Met een **`for`**-statement worden nu een voor een de waarden uit de variabele aan de ingang toegekend. Na elke toekenning wordt gewacht op een klokflank. Daarna na wordt nog even gewacht voordat de volgende toekenning plaatsvindt. Het proces eindigt met een **`wait`**-statement.

Om de simulatie automatisch te laten verlopen gebruiken we een bestand met daarin commando's voor de simulator. Dit bestand is te zien in listing 11.13. We beginnen met het opzetten van de `work`-map. Hierin bewaart de simulator de gecompileerde bestanden. Daarna compileren we de VHDL-beschrijving en de testbench. Vervolgens starten we de simulator en loggen we alle signalen. Dat is prima te doen omdat het aantal signalen klein is. Als het aantal signalen groot is, moeten alleen de signalen die

```
1  -- Filename:      tb_herk110.vhd
2  -- Filetype:      VHDL testbench code
3  -- Date:          18 may 2011
4  -- Update:        8 nov 2016
5  -- Description:   A testbench for the pattern recognizer
6  -- Author:        J. op den Brouw
7  -- State:         demo
8  -- Error:         -
9  -- Version:       1.1.1
10 -- Copyright:     (c)2016, De Haagse Hogeschool
11 --
12 -- This is an example of a testbench for the pattern recognizer.
13 -- The testbench is straight forward: declaration and instantiation
14 -- of the pattern recognizer, a process for generation of the clock
15 -- and a process for generation of de data bitstream. The clock
16 -- process is very simple, it just creates a 50% duty cycle clock.
17 -- The data process is constructed using an variable length array
18 -- of (std_logic) bits. The array indices are read one by one and
19 -- assigned to the input of the FSM. The process waits forever when
20 -- done.
21
22 library ieee;
23 use ieee.std_logic_1164.all;
24
25 -- Empty entity, as usual
26 entity tb_herk110 is
27 end tb_herk110;
28
29 -- The architecture of the testbench
30 architecture testbench of tb_herk110 is
31 -- The constants define the timing parameters for the testbench
32 constant Tperiod : time := 10 ns;
33 constant small_delay : time := 2 ns;
34 -- The signals to be assigned and monitored
35 signal sim_clk : std_logic;
36 signal sim_areset : std_logic;
37 signal sim_x : std_logic;
38 signal sim_z : std_logic;
39
40 -- The Design Under Test (DUT)
41 component herk110
42     port (clk : in std_logic;
43           areset : in std_logic;
44           x : in std_logic;
45           z : out std_logic);
46 end component;
```

Listing 11.11: *Beschrijving van testbench voor herkenner voor het patroon 110 (entity en architecture).*

nodig zijn gelogd worden. Dit versnelt de simulatie. We beelden de signalen in een timingdiagram af middels enkele **`add wave`**-commando's. We openen het timingdiagram (Engels: waveform) en laten daarna de simulatie 260 ns simuleren. Daarna zorgen we ervoor dat het timingdiagram in zijn geheel zichtbaar is op het beeldscherm.

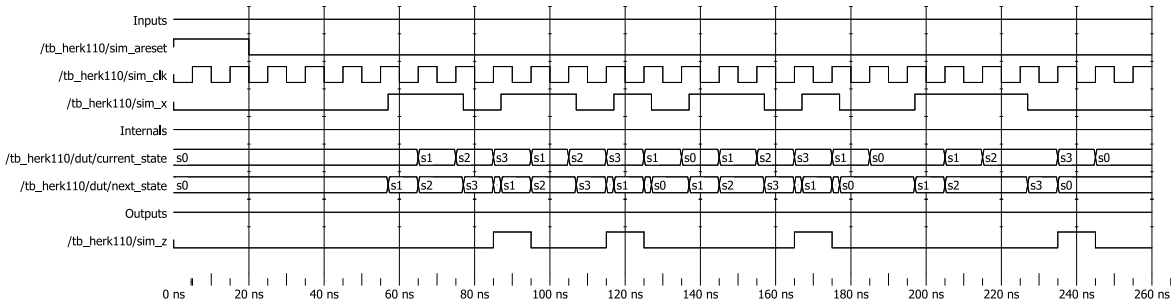
```
begin
    -- Instantiation of the DUT, using a port map
    dut : herk110
    port map (clk => sim_clk, areset => sim_areset, x => sim_x,
              z => sim_z);

    -- The clock generation process
    clockgen: process is
    begin
        sim_clk <= '0';
        wait for Tperiod/2;
        sim_clk <= '1';
        wait for Tperiod/2;
    end process clockgen;

    -- The data generation process
    datagen: process is
    -- All the valuations for the DUT as string. Note the construction:
    -- a constant may be unconstrained, but a variable may not. Thus the
    -- trick is to define an unconstrained constant array and assign it
    -- to a constrained variable using the range of the previously
    -- assigned constant.
    constant x_inputs_setup : std_logic_vector := "0110110101101001110";
    variable x_inputs : std_logic_vector(x_inputs_setup'range) :=
                                                        x_inputs_setup;
    begin
        -- Set input to known value
        sim_x <= '0';

        -- Generation of the asynchronous reset signal.
        -- Set high first and wait two clock cycles
        sim_areset <= '1';
        wait for 2*Tperiod;

        -- Set low and wait two clock cycles
        sim_areset <= '0';
        wait for 2*Tperiod;

        -- Synchronize on the positive edge of the clock, and wait a bit.
        wait until sim_clk = '1';
        wait for small_delay;

        -- Iterate over all array indices
        for index in x_inputs'range loop  -- loop through all input bits
            sim_x <= x_inputs(index);     -- assign to FSM input
            wait until sim_clk = '1';
            wait for small_delay;
        end loop;

        wait;
    end process datagen;

end testbench;
```

Listing 11.12: *Beschrijving van testbench voor herkenner voor het patroon 110 (vervolg architecture).*

```
# Transcript window
transcript on

# Recreate the RTL work directory
if {[file exists rtl_work]} {
    vdel -lib rtl_work -all
}
vlib rtl_work
vmap work rtl_work

# Compile the VHDL description and testbench
vcom -2008 -work work herk110.vhd
vcom -2008 -work work tb_herk110.vhd

# Start simulation with 1 ns time step
vsim -t 1ns -L rtl_work -L work -voptargs="+acc" tb_herk110

# Log all signals in the design, good if the number
# of signals is small.
add log -r *

# Add a number of signals of the simulated design
add wave -divider "Inputs"
add wave sim_areset
add wave sim_clk
add wave sim_x
add wave -divider "Internals"
add wave dut/current_state
add wave dut/next_state
add wave -divider "Outputs"
add wave sim_z

# Open waveform window
view wave

# Run simulation for 260 ns
run 260 ns

# Fill up the waveform in the window
wave zoom full
```

Listing 11.13: *Modelsim command bestand..*

In figuur 11.35 is het simulatieresultaat te zien. Te zien is dat de machine eerst wordt gereset. Daarna worden iets na de opgaande flank de waarden aan `x` toegekend. De opvolgertoestand verandert direct met een verandering van de ingang. Dat komt omdat de logica voor de opvolgertoestand combinatoriek is. De verandering van de huidige toestand gebeurt op opgaande klokflanken wat inhoudt dat het hier om flipflops gaat. Verder is te zien dat de opvolgertoestand ook verandert als de huidige toestand verandert. Dat is logisch want de opvolgertoestand is ook afhankelijk van de huidige toestand. Een aantal keer komt het patroon 110 in de ingangswaarden voor. De machine geeft na het herkennen van 110 een 1 af in de klokcyclus na de 0. Dat betekent dat het hier een Moore-machine betreft.

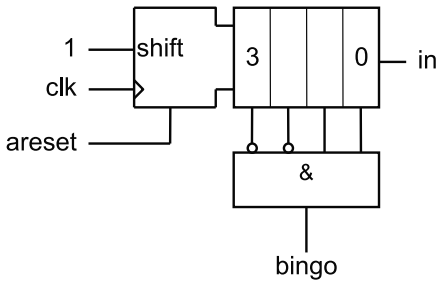


Figuur 11.35: *Simulatieresultaat van de herkenner.*

11.16 Herkenners met schuifregister

Het is mogelijk om een patroonherkenner te maken op basis van een schuifregister en logische poorten. In figuur 11.36 is een schema te zien van een herkenner voor het patroon 0011. Het schuifregister schuift op iedere actieve klokflank een bit van de ingang naar binnen (rechts in de figuur). Als in de bitstroom het patroon 0011 voorkomt, wordt uitgang *bingo* logisch 1. In alle andere gevallen is de uitgang logisch 0.

De reset-implementatie vergt enige aandacht. Het te herkennen patroon is 0011. Het laden van allemaal nullen levert problemen op als na de deactiveren van de reset direct twee enen volgen. De herkenner geeft dan een 1 af terwijl er pas twee bits gedetecteerd zijn. Het juiste reset-patroon moet 1100 zijn, waarbij de voorste 1 de meest significante bit voorstelt.



Figuur 11.36: *Herkenner voor 0011 op basis van een schuifregister.*

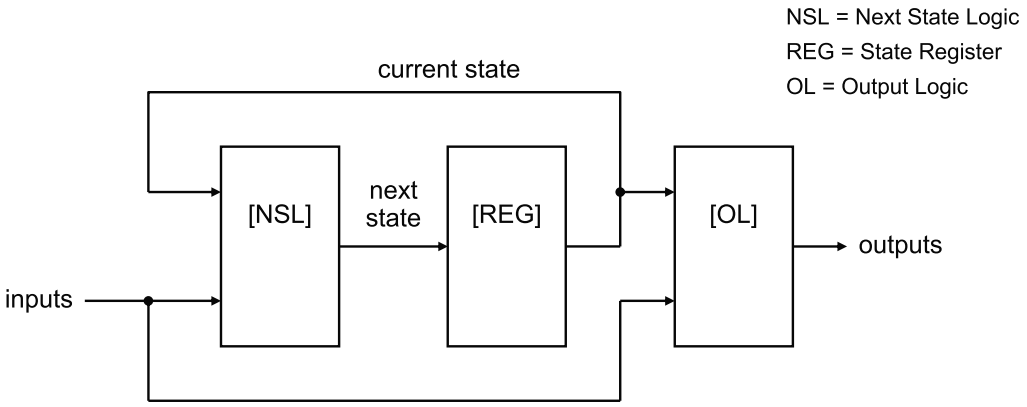
Het voordeel van deze manier van detecteren is dat het schuifregister makkelijk kan worden uitgebreid met synchronisatie-flipflops en een ontdenderschakeling.

11.17 Tijdgedrag van toestandsmachines

Het tijdgedrag van toestandsmachines kan bepaald worden met de procedures uit een eerder hoofdstuk. We nemen als voorbeeld de Mealy-machine. Het blokschema is te zien in figuur 11.37.

Het berekenen van de maximale frequentie gaat als volgt:

$$f_{max} = \frac{1}{t_{P(max)}(\text{REG}) + t_{P(max)}(\text{NSL}) + t_{su}(\text{REG})} \tag{11.10}$$



Figuur 11.37: *Model voor een Mealy-machine.*

Eventuele klokskew is hierin niet meegenomen. De setup- en holdtijden van ingang ten opzichte van de klok kunnen berekend worden met:

$$\begin{aligned} t_{su}(\text{inputs-to-clk}) &= t_{P(max)}(\text{NSL}) + t_{su}(\text{REG}) \\ t_h(\text{inputs-to-clk}) &= t_h(\text{REG}) - t_{P(min)}(\text{NSL}) \end{aligned} \tag{11.11}$$

Het tijdgedrag van de uitgangen is een complexe aangelegenheid. De uitgangen van de Mealy-machine veranderen namelijk op de actieve klokflank én op de ingangen. Dat houdt in dat de uitgangen twee sets vertragingen hebben: ten opzichte van de actieve klokflank (synchroon) en ten opzichte van de ingangen (asynchroon). We vinden voor de uitgangsvertragingen ten opzichte van de klokflank:

$$\begin{aligned} t_{P(min)}(\text{clk-to-outputs}) &= t_{P(min)}(\text{REG}) + t_{P(min)}(\text{OL}) \\ t_{P(max)}(\text{clk-to-outputs}) &= t_{P(max)}(\text{REG}) + t_{P(max)}(\text{OL}) \end{aligned} \tag{11.12}$$

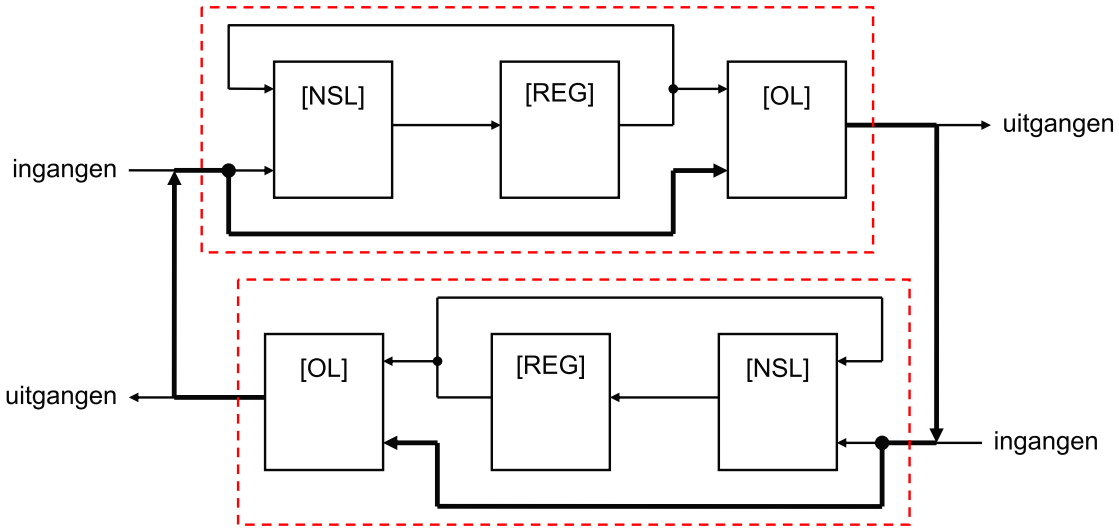
Ten opzichte van de ingangen vinden we:

$$\begin{aligned} t_{P(min)}(\text{inputs-to-outputs}) &= t_{P(min)}(\text{OL}) \\ t_{P(max)}(\text{inputs-to-outputs}) &= t_{P(max)}(\text{OL}) \end{aligned} \tag{11.13}$$

De laatste twee vergelijkingen zijn het gevolg van de combinatorische paden van ingang naar uitgang. Moore-machines hebben dat probleem niet, de uitgangen veranderen altijd synchroon met de klokflank.

Bij Mealy-machines speelt nog een ander probleem. Bekijk figuur 11.38 eens. Hierin zijn twee *rondgekoppelde* Mealy-machines te zien. Van elke machine wordt een aantal uitgangen van de ene machine verbonden met de ingangen van de andere machine, te zien aan de dikke lijnen en pijlen. Het is nu mogelijk dat er *asynchrone* terugkoppeling plaatsvindt. Er is een combinatorisch pad van uitgangen naar ingangen. Oscillatie van signalen is één van de mogelijkheden. De meeste ontwikkeltools waarschuwen bij dit soort terugkoppelingen maar de ontwerper moet dit zelf in de gaten houden.

De oplossing is eenvoudig: één van de machines moet als Moore-machine worden ontworpen.



Figuur 11.38: *Twee rondgekoppelde Mealy-machines met mogelijke asynchrone terugkoppeling.*

11.18 Opgaven

11.1. Teken het toestandsdiagram van een JK-flipflop.

11.2. Teken het toestandsdiagram van een synchrone 6-teller.

11.3. Gegeven is een RG-machine. De machine heeft twee ingangen A en B en twee uitgangen R en G. De machine begint met de uitgangen R en G allebei logisch 0 en wacht op een verandering op A en/of B. Als A logisch 1 wordt, moet de machine achtereenvolgens eerst R logisch 1 maken en daarna zowel R als G logisch 1 maken. Daarna moet de machine weer wachten op een verandering van A en/of B. Als B logisch 1 wordt, moet de machine achtereenvolgens eerst G logisch 1 maken en daarna zowel R als G logisch 1 maken. Daarna moet de machine weer wachten op een verandering van A en/of B. Teken het toestandsdiagram van de RG-machine.

11.4. Stel de toestandstabel van de JK-flipflop op.

11.5. Stel de toestandstabel van de synchrone 6-teller op.

11.6. Stel de toestandstabel van de RG-machine op.

11.7. Is het mogelijk om een Moore-machine als een Mealy-machine te tekenen? En andersom? Motiveer het antwoord.

11.8. Gegeven is een machine met drie toestanden en twee toestandsbits. Bepaal het aantal unieke codecombinaties.

11.9. Bepaal het aantal mogelijke codecombinaties voor een machine met tien toestanden en een minimum aan toestandsbits.

11.10. Geef de toestands- en uitgangsfuncties van de JK-flipflop. Gebruik de binaire telcode voor toestandscodering.

11.11. Geef de toestands- en uitgangsfuncties van de synchrone 6-teller. Gebruik de binaire telcode voor toestandscodering.

11.12. Geef de toestands- en uitgangsfuncties van de RG-machine op. Gebruik de binaire telcode voor toestandscodering.

11.13. Geef de toestands- en uitgangsfuncties van de RG-machine op. Gebruik de one-hot-codering voor toestandscodering.

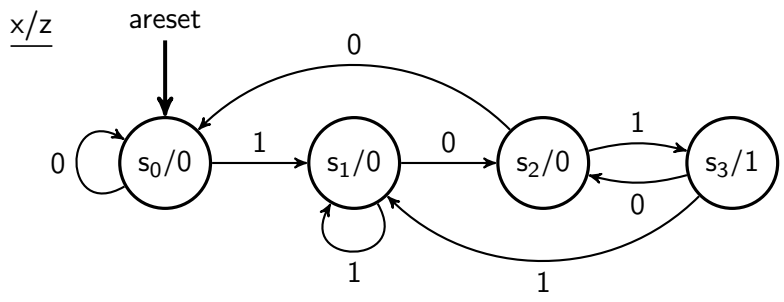
11.14. Ontwerp het toestandsdiagram van een machine voor het herkennen van het patroon (of reeks) 1001. De machine geeft een 1 af en stopt met herkennen.

11.15. Ontwerp het toestandsdiagram van een machine voor het doorlopend herkennen van het patroon (of reeks) 1001. De machine moet bij herkenning een 1 afgeven. Ontwerp zowel een Moore- als een Mealy-machine.

11.16. Ontwerp het toestandsdiagram van een Mealy-machine voor het doorlopend herkennen van het patroon (of reeks) 1001. Overlappingen zijn mogelijk. De machine moet bij herkenning een 1 afgeven. Geef de vergelijkingen voor de opvolgertoestand en de uitgang als gekozen wordt voor de binaire telcode als toestandscodering.

11.17. Ontwerp het toestandsdiagram van een Mealy-machine voor het het doorlopend herkennen van het patroon (of reeks) 100 met een minimaal aantal toestanden.

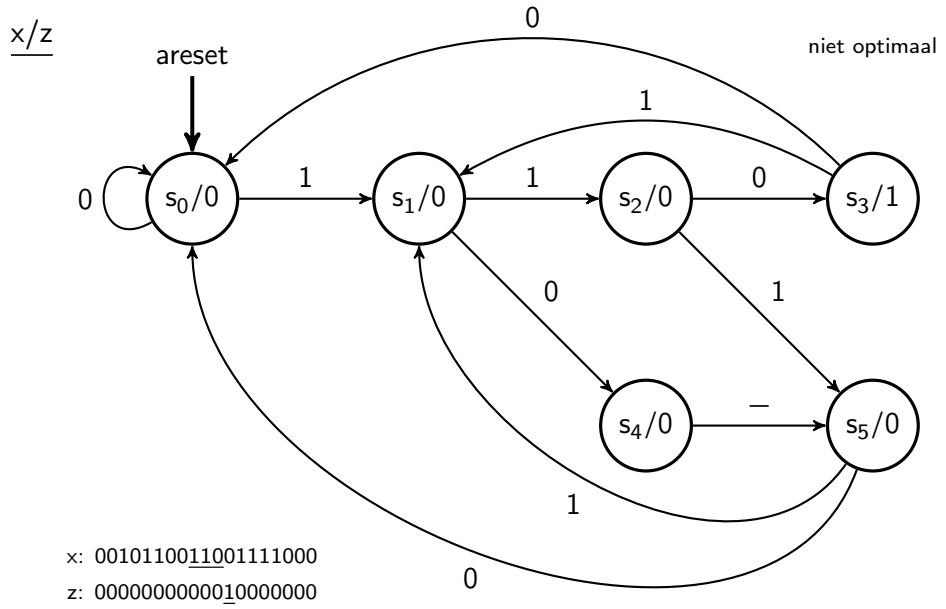
11.18. Gegeven is de onderstaande machine voor het overlappend herkennen van 101 (figuur P11.1). Geef de toestandsfuncties indien one-hot codering wordt gebruikt.



Figuur P11.1: *Herkenner voor het overlappend herkennen van het patroon 101 (Moore).*

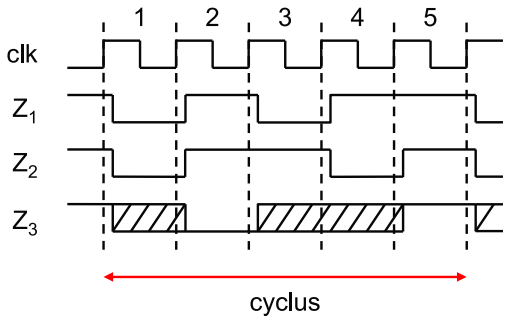
11.19. In figuur P11.2 is het toestandsdiagram van een machine te zien voor het bloksgewijs herkennen van het patroon (of reeks) 110. Een studente beweert dat het toestandsdiagram van deze machine compacter kan worden getekend omdat toestand s_0 en s_5 identiek zijn en samengenomen kunnen worden. Klopt deze bewering? Motiveer het antwoord.

11.20. Ontwerp een Mealy-machine voor het het bloksgewijs herkennen van het patroon (of reeks) 110 met een minimaal aantal toestanden. De machine blijft wachten zolang een 0 wordt aangeboden en gaat pas beginnen met het herkennen van een 1, inclusief de eerste 1.



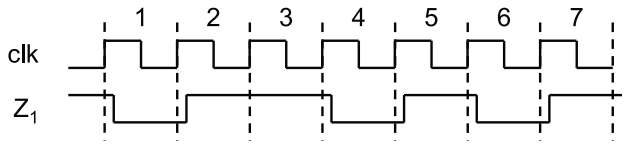
Figuur P11.2: *Herkenner voor het bloksgewijs herkennen van het patroon 110 (Moore).*

11.21. In figuur P11.3 is een machine gespecificeerd d.m.v. een timingdiagram. Geef de functies voor de opvolgertoestandslogica en de uitgangslogica, de toestandscodering ligt gedeeltelijk vast.



Figuur P11.3: *Timingdiagram van een toestandsmachine.*

11.22. Gegeven is het timingdiagram in figuur P11.4. Ontwerp een machine die het timingdiagram uitvoert met gebruikmaking van extra flipflops en zonder uitgangslogica.



Figuur P11.4: *Timingdiagram van een toestandsmachine.*

11.23. Ontwerp een toestandsdiagram voor een snoepmachine die 35 cent moet herkennen. Toegestane munten zijn 5 cent (S) en 10 cent (D). Bij dit bedrag moet

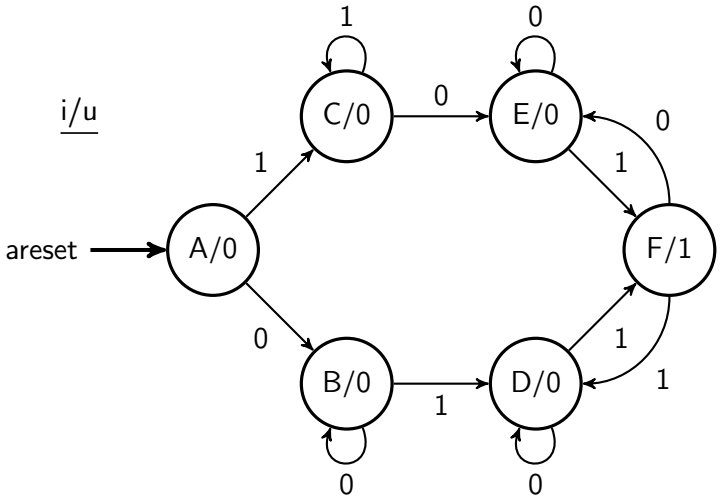
snoepgoed worden geretourneerd. Bij te hoog bedrag moet een signalering geactiveerd worden.

11.24. Wat is het hoogste bedrag dat in de machine in opgave 11.23 kan worden ingeworpen (dat is niet het bedrag dat de machine uiteindelijk moet herkennen)?

11.25. Hoeveel unieke toestanden heeft machine in opgave 11.23?

11.26. Stel dat een snoepautomaat munten van 5 cent (S), 10 cent (D) en 20 cent (T) accepteert en 35 cent moet herkennen. Wat is het hoogste bedrag dat kan worden ingeworpen? Hoeveel unieke toestanden heeft deze snoepautomaat?

11.27. Minimaliseer het aantal toestanden van de machine in figuur P11.5.



Figuur P11.5: *Toestandsdiagram van een Moore-machine.*

11.28. Gegeven een machine die beschreven wordt door onderstaande *next state equations* en *output equation*:

$$\begin{aligned} Q_1^{n+1} &= Q_1^n \cdot X^n + Q_0^n \cdot X^n \\ Q_0^{n+1} &= Q_1^n \cdot X^n + \overline{Q_0^n} \cdot X^n \\ Z^n &= Q_1^n + Q_0^n \end{aligned} \tag{P11.1}$$

(a) Stel de waarheidstabel op voor de functies Q_1^{n+1}, Q_0^{n+1} en Z^n.

(b) Stel het toestandsdiagram op voor deze machine. Gebruik hiervoor de waarheidstabel uit (a).

(c) Het toestandsdiagram kan vereenvoudigd worden, d.w.z. er kunnen toestanden samengenomen worden zonder dat het gedrag van de machine verandert. Zulke toestanden worden *equivalent* genoemd. Laat zien dat het toestandsdiagram uit (b) ook met twee toestanden getekend kan worden zonder dat de werking van de machine verandert.

12

Datapadsystemen

Naar mate de complexiteit van een digitale schakeling toeneemt, wordt het ontwerpen van zo'n systeem lastiger. We zoeken naar een manier waarop we dergelijke complexe systemen kunnen ontwerpen. Dat doen we door een systeem op te delen. We delen de schakeling op in een aantal onderdelen waarbij de functie van elk onderdeel precies gedefinieerd is. Dit opdelen wordt *partitioneren* genoemd. Onderdelen zijn met elkaar verbonden zodat dataoverdracht mogelijk is. Het vergt kennis en kunde om een juiste opdeling van het digitale systeem te maken. Dit is een stukje creativiteit die in de ontwerper schuilt.

Elk onderdeel wordt opgedeeld in een datapad waarlangs de data wordt verwerkt en een besturing die het datapad aanstuurt. We noemen dit het *datapad-besturingsmodel*. Met het datapad-besturingsmodel is het mogelijk om met behulp van digitale schakelingen een *algoritme*[1] te implementeren waarmee de schakeling zijn specifieke functie krijgt.

We beginnen met een introductie waarin het concept datapad-besturingsmodel wordt uitgelegd. We laten het ontwerpen van een datapad en besturing zien aan de hand van een sequentiële vermenigvuldiger. We ontwerpen een eerste versie van het datapad en een besturing. We merken dan op dat het wat eenvoudiger kan en komen zo tot een tweede ontwerp dat we met VHDL gaan beschrijven. In de VHDL-code zien we duidelijk de scheiding tussen datapad en besturing. Dat levert een behoorlijke hoeveelheid code op. Daarna laten we ook zien dat datapadonderdelen geïntegeerd kunnen worden in de besturing. We noemen dit besturing met geïntegreerd datapad (Engels: Finite State Machine with datapad, FSMd). Deze VHDL-code is goed leesbaar en onderhoudbaar.

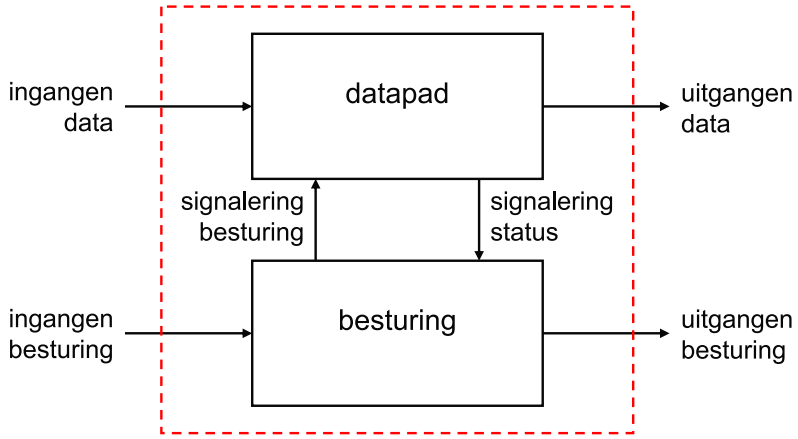
We besluiten dit hoofdstuk met het ontwerpen van een stopwatch. De besturing wordt gerealiseerd met een toestandsmachine en het datapad bestaat uit een teller die de tijd in éénhonderste van een seconde nauwkeurig aangeeft.

[1] Een algoritme (van het Arabische woord algawarizmiat naar de naam van de Perzische wiskundige Al-Chwarizmi) is een eindige reeks instructies – meestal voor berekening of dataverwerking – om vanuit een gegeven begintoestand het daarbij behorende doel te bereiken.

12.1 Het datapad-besturingsmodel

In het datapad-besturingsmodel [17] wordt een digitaal systeem (of een onderdeel daarvan) gescheiden in een datapad en een besturing. Het datapad bevat de onderdelen waarlangs de data getransporteerd en bewerkt wordt. We kunnen hierbij denken aan optellers, aftrekkers, vergelijkers, multiplexers en registers voor opslag. Ook RAM, ROM, schuifregisters en tellers kunnen onderdeel zijn van een datapad. De besturing zorgt ervoor dat het datapad op de juiste manier wordt aangestuurd zodat het geheel de juiste werking realiseert. De besturing wordt meestal uitgevoerd als een toestandsmachine maar in specifieke gevallen zouden ook tellers dienst kunnen doen als besturingseenheid.

In figuur 12.1 is het blokdiagram van een datapadsysteem te zien. Het datapadsysteem bestaat uit twee onderdelen, te weten het datapad en de besturing, die zowel onderling als met de buitenwereld informatie uitwisselen. Het datapad voert bewerkingen uit op aanwijzingen van de besturing (signalering besturing) en geeft informatie terug over de vorderingen (signalering status). De besturing zorgt voor het correcte verloop van de bewerkingen in het datapad.



Figuur 12.1: *Blokdiagram van een datapad met besturing.*

Zowel het datapad als de besturing heeft interactie met de buitenwereld. Zo heeft de besturing ingangen die vanuit de omgeving waarin het datapadsysteem zich bevindt worden geactiveerd. Te denken valt aan een signaal dat aangeeft dat een bepaalde bewerking moet worden gestart. Via de uitgangen kan de besturing informatie aan de buitenwereld geven over het verloop van de bewerking. Hierbij kunnen we denken aan signalering die aangeeft dat de bewerking voltooid is.

Het datapad heeft ook verbinding met de omgeving. Er kan data voor verwerking worden aangeboden en het datapad geeft informatie terug aan de buitenwereld. Als voorbeeld kunnen we een vermenigvuldiging van twee getallen aanvoeren. De twee getallen worden aangeboden waarna een startsignaal wordt gegeven. Als de vermenigvulding klaar is wordt dit gesignaleerd via de besturingsuitgangen en is het antwoord beschikbaar op de data-uitgangen. In de volgende paragraaf behandelen we de sequentiële vermenigvuldiger.

12.2 Sequentiële vermenigvuldiger

In een eerder hoofdstuk is het ontwerp van een vermenigvuldiger besproken. Dat kan heel goed met combinatoriek, maar dat levert veel hardware (lees chipoppervlakte, energieverbruik) op. Een vermenigvuldiger kan echter ook als sequentiële machine worden ontworpen, waardoor de hoeveelheid hardware beperkt wordt. Het nadeel is wel dat het langer duurt voordat het resultaat beschikbaar is. We ontwerpen een vermenigvuldiger voor unsigned getallen.

Eerste versie van de sequentiële vermenigvuldiger

In figuur 12.2 is de vermenigvuldiging van de 4-bits getallen 1101_2 en 1011_2 te zien. Vermenigvuldiging van de twee getallen gaat op vergelijkbare wijze als in het decimale stelsel met de opmerking dat er alleen met 0 en met 1 vermenigvuldigd wordt. Vermenigvuldigen in het binaire talstelsel is eenvoudig. Vermenigvuldigen met 0 levert 0 op en vermenigvuldigen met 1 levert het te vermenigvuldigen getal op. We schuiven nullen aan de rechterkant in om tweetallen, viertallen etc. te vermenigvuldigen. Nadeel van deze opzet is dat een multi-input opteller nodig is om alle deelresultaten op te tellen.

$$
\begin{array}{lrl}
A: & 1101 & \\
B: & \underline{1011} & \times \\
 & 1101 & \\
 & 11010 & \\
 & 000000 & \\
 & \underline{1101000} & + \\
P: & 10001111 &
\end{array}
$$

Figuur 12.2: *Vermenigvuldiging van 1101_2 en 1011_2.*

Over de vermenigvuldiging valt het volgende te zeggen. Het getal A wordt steeds in een verschoven variant opgeteld bij de andere deelresultaten als de bijbehorende bit van B een 1 is. Als de bijbehorende bit van B een 0 is, wordt er 0 bij de andere deelresultaten opgeteld.

We kunnen nu de vermenigvuldiging uitvoeren volgens de schuif-en-optelmethode (Engels: shift and add). Dit is te zien in figuur 12.3. We vermenigvuldigen de twee 4-bits getallen A en B. De vermenigvuldiging voeren we in een aantal slagen, *iteraties* genoemd, uit. We bekijken steeds één bit van B en voeren vervolgens een optelling uit. Afhankelijk van de waarde van de bit van B tellen we (een verschoven versie van) A op of 0. We tellen tussentijds op zodat een standaard full adder kan worden gebruikt. Het resultaat is een 8-bits getal dus we reserveren voor het resultaat 8 bits. In het begin wordt het (tussen-)resultaat PP_0 (Partial Product) op nul gezet.

We beginnen met de minst significante bit van B, dat zijn de eenheden. Deze bit is 1 zodat de waarde van A bij het tussenresultaat PP_0 wordt opgeteld. We gaan verder met de bit in B voor de tweetallen. Ook deze bit is 1 zodat er (een verschoven versie van) A bij wordt opgeteld. We gaan zo door totdat alle bits van B zijn verwerkt. Merk op dat we

	A :	1101	
	B :	1011	$\times$
	PP_0 :	00000000	
iteratie 1:		0000**1101**	$+$
	PP_1 :	00001101	
iteratie 2:		000**1101**0	$+$
	PP_2 :	00100111	
iteratie 3:		00**000**00	$+$
	PP_3 :	00100111	
iteratie 4:		0**1101**000	$+$
	P :	10001111	

Figuur 12.3: *Vermenigvuldiging van twee 4-bits getallen.*

vier tussenresultaten hebben voordat we het uiteindelijke resultaat P (Product)hebben uitgerekend.

We kunnen voor de sequentiële vermenigvuldiger een algoritme opstellen in de vorm van een programma. Dit is te zien in listing 12.1. We gaan hierbij uit van het algemene geval van een $n \times n$-bits vermenigvuldiger.

```
1  A = ...;
2  B = ...;
3  P = 0;                    // Clear intermediate result
4  for i = 0 to n-1 do       // Iterate over the n bits of B
5     if b(i) = 1 then       // If i-th bit of B is 1 then ...
6        P = P + A;          // add A to intermediate result
7     else                   // But if not ...
8        P = P + 0;          // add zero to intermediate result
9     end if;
10    left-shift A;          // Reposition A by shift to left
11 end for;
```

Listing 12.1: *Pseudo-code voor de sequentiële vermenigvuldiger.*

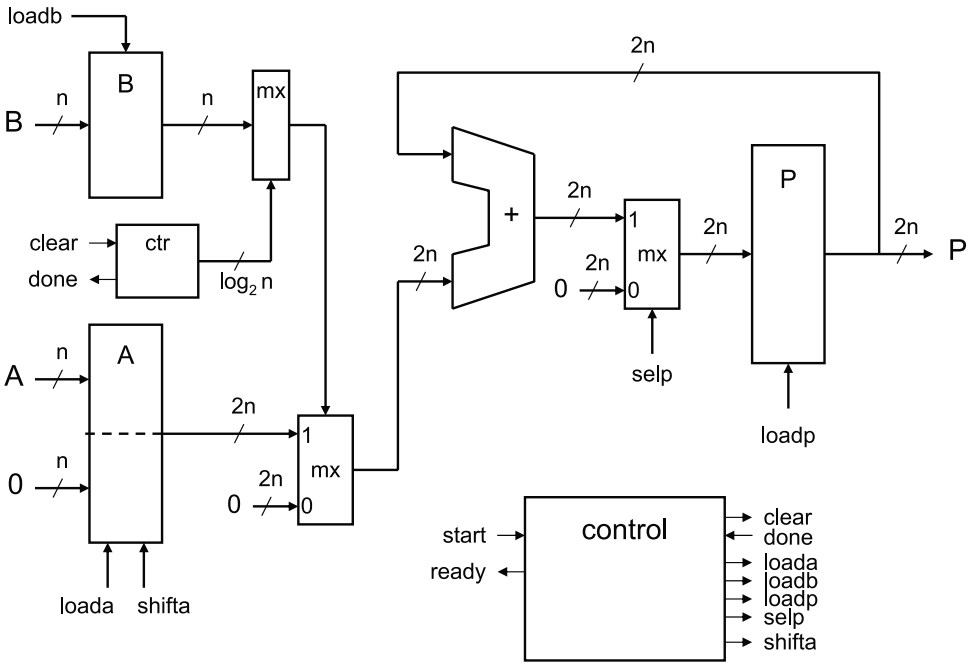
De code is opgebouwd rond een **`for`**-lus. Deze lus geeft het aantal iteraties aan. De lus loopt van 0 t/m $n-1$ om de n bits te verwerken. We houden dit bij met een teller. In de lus wordt steeds één bit van het getal B verwerkt. We selecteren de bit van B met een multiplexer. Als deze bit een 1 is, wordt de waarde van A opgeteld bij het resultaat P. Als de bit een 0 is, wordt 0 bij het resultaat P opgeteld. We selecteren het optellen van A bij P of het optellen van 0 bij P met behulp van een multiplexer. Het optellen wordt gerealiseerd met een $2n$-bits opteller. Daarna wordt A één bit naar links geschoven en aangevuld met een 0. Merk op dat het schuiven van A ervoor zorgt dat A uit $2n$ bits moet bestaan, net als het resultaat P. We implementeren A middels een schuifregister. B en P realiseren we middels twee registers.

We stellen het datapad samen met de volgende onderdelen:

- Er is een $2n$-bits schuifregister nodig voor A. De hoogste n bits worden geladen met nullen.

- Er is een n-bits register nodig voor B.
- Er is een $2n$-bits register nodig voor P.
- Er is een $nx1$-bits multiplexer nodig om de afzonderlijke bits van B te selecteren.
- Er is een teller nodig met $^2\log n$ telbits om de afzonderlijke bits van B aan te wijzen.
- Er is een $2x1$ $2n$-bits multiplexer nodig om A of 0 te selecteren afhankelijk van de waarde van de verwerkte bit van B.
- Er is een $2n$-bits opteller nodig om de waarde van A op te tellen bij het (tussen-) resultaat van P. Er is geen ingaande of uitgaande carry nodig.
- Er is een $2x1$ $2n$-bits multiplexer nodig om register P te laden met de uitkomst van de opteller of met nullen (wissen).
- Alle registers hebben een laad- en onthoudmogelijkheid.

In figuur 12.4 is het datapad en de besturing te zien. Dit is een eerste versie van de vermenigvuldiger. We zien de registers voor B en P en het schuifregister voor A. Verder is de teller voor het bijhouden van de iteraties te zien. Diverse multiplexers zijn aanwezig om de data te routeren. Een opteller is aanwezig om P op te hogen. Duidelijk herkenbaar is dat er een lus in het datapad aanwezig is waarmee (de verschoven versie van) A wordt opgeteld bij P. We spreken dan ook wel van *accumuleren*. P wordt dan ook wel een *accumulator* genoemd. Het datapad wordt aangestuurd door de besturing (control). Er is een $start$-ingang waarmee de vermenigvuldiging gestart wordt en een $ready$-uitgang waarmee wordt aangegeven of de vermenigvuldiging klaar is.

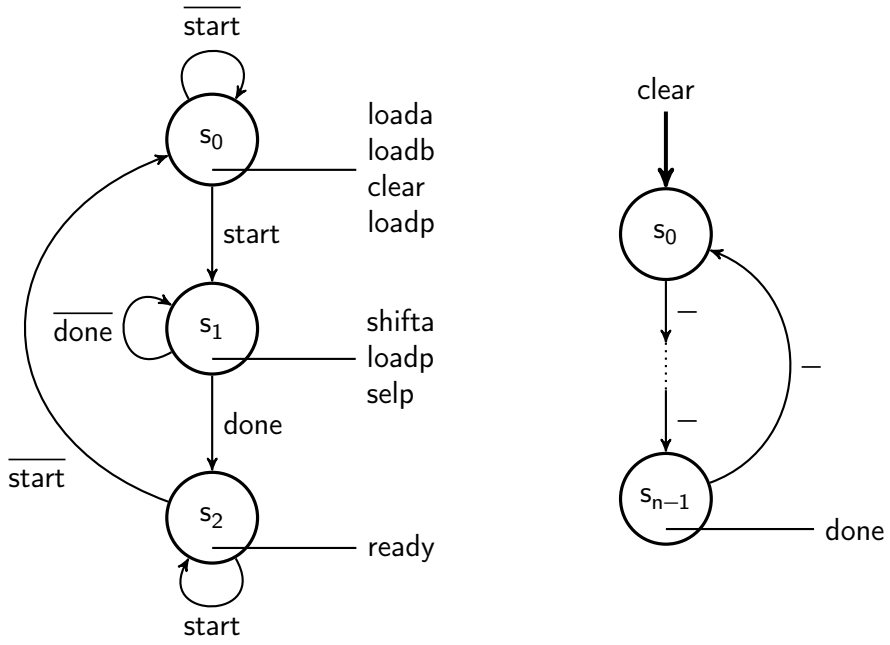


Figuur 12.4: *Datapad en besturing voor de sequentiële vermenigvuldiger (versie 1).*

De diverse datapadcomponenten worden aangestuurd met een aantal besturingssignalen. De signalen $loada$, $loadb$ en $loadp$ zorgen ervoor dat het betreffende register geladen

wordt. Signaal $shifta$ zorgt ervoor dat register A naar links geschoven wordt. Met signaal $selp$ wordt de multiplexer bij P aangestuurd. De signalen $clear$ en $done$ zijn bedoeld voor interactie met de teller. Met $clear$ wordt de teller op 0 gezet. Signaal $done$ geeft aan dat de teller klaar is met tellen.

De besturing wordt uitgevoerd als een toestandsmachine. Deze heeft een $start$-ingang om de bewerking te starten en een $ready$-uitgang om aan te geven dat de bewerking klaar is. Het toestandsdiagram is links te zien in figuur 12.5. Bij de overgangen staan signaalnamen geschreven, al dan niet met een inversebalk erboven. Dit geeft aan of een signaal geactiveerd of gedeactiveerd is. Bij een uitgang staat alleen een signaalnaam vermeld als het signaal geactiveerd is. Het ontbreken van een signaalnaam betekent dat het signaal gedeactiveerd is.



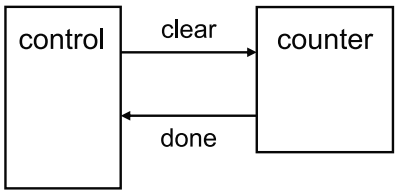
Figuur 12.5: *De toestandsdiagrammen van de besturing (links) en de teller (rechts).*

De machine begint in toestand s_0. De machine blijft in deze toestand zolang $start$ niet geactiveerd is. In deze toestand worden de registers geladen en de teller op 0 gezet. Merk op dat A en B geladen worden met externe data en dat P geladen wordt met 0. Na activatie van $start$ komt de machine in toestand s_1. In deze toestand wordt de feitelijke vermenigvuldiging uitgevoerd. Het schuiven van A wordt geactiveerd net als het laden van register P en het selecteren van de uitkomst van de opteller bij register P. Dit wordt gedaan zolang de teller nog niet klaar is (middels signaal $done$). Als de teller klaar is ($done$ is geactiveerd), gaat de machine naar toestand s_2. In deze toestand wordt $ready$ geactiveerd, de bewerking is immers klaar. Er wordt in deze toestand gewacht zolang $start$ geactiveerd is. Als $start$ gedeactiveerd is, wordt naar toestand s_0 gegaan.

Rechts in figuur 12.5 is het toestandsdiagram van de teller te zien. De teller heeft een dominante $clear$-ingang die de teller in de begintoestand brengt en een $done$-uitgang die geactiveerd is als de teller in de hoogste stand staat.

Gekoppelde toestandmachines

In de eerste versie van de sequentiële vermenigvuldoger zijn de besturing en teller gekoppeld. Aangezien de teller gezien kan worden als een toestandsmachine kunnen we spreken van *gekoppelde toestandsmachines*. Informatie-uitwisseling gaat via de signalen $clear$ en $done$. Merk op dat beide machines (uiteraard) op dezelfde klokflank geklokt worden. Stel dat het signaal $clear$ tijdens een bepaalde klokcyclus geactiveerd wordt dan ziet de teller deze activatie pas op de volgende klokflank. Hetzelfde geldt voor signaal $done$.



Figuur 12.6: *Gekoppelde toestandsmachines.*

Tweede versie van de sequentiële vermenigvuldiger

De eerste vermenigvuldiger kan vereenvoudigd worden. We merken op dat van register B steeds één bit wordt getest of deze bit 0 of 1 is. Hiervoor is een multiplexer aanwezig om de juiste bit te selecteren. We kunnen de multiplexer ook verwijderen en register B als een schuifregister implementeren dat naar rechts schuift en aangevuld wordt met nullen. De te testen bits van B komen zodoende op de minst significante positie in B terecht. Dit leidt echter tot meer logica voor register B. Verder kunnen we ook stellen dat de vermenigvuldiging voltooid is als alle bits van B 0 zijn. Het testen op 0 van register B kan eenvoudig gerealiseerd worden met een vergelijkschakeling. Hiermee sparen we de teller uit. In de eerste versie is ook te zien dat of (de verschoven versie van) A of 0 bij P wordt opgeteld. Het optellen van 0 kunnen we realiseren door op dat moment P niet te laden. Daarnaast moet register P ook gewist kunnen worden zodat we P uitrusten met een clear-ingang.

We kunnen voor de tweede versie van de sequentiële vermenigvuldiger een algoritme opstellen in de vorm van een programma. Dit is te zien in listing 12.2. We gaan weer uit van het algemene geval van een $n \times n$-bits vermenigvuldiger.

```
1  A = ..;
2  B = ..;
3  P = 0;                  // Clear intermediate result
4  while B <> 0 do         // Iterate while B not equal to 0
5      if b(0) = 1 then    // If bit 0 of B is 1 then ...
6          P = P + A;      // add A to intermediate result
7      end if;
8      left-shift A;       // Reposition A by shift to left
9      right-shift B;      // Set new bit available
10 end while;              // Ready
```

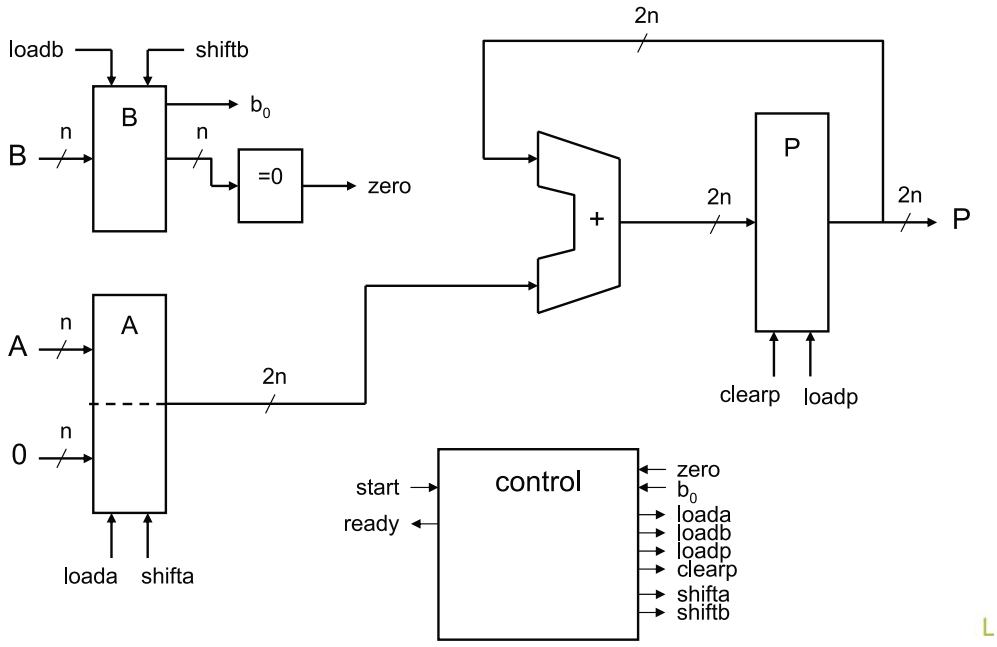
Listing 12.2: *Pseudo-code voor de tweede versie van de sequentiële vermenigvuldiger.*

De code is opgebouwd rond een **`while`**-lus die steeds doorlopen wordt zolang B ongelijk aan 0 is. Binnen de lus wordt steeds bit 0 van B getest. Als deze bit een 1 is, wordt (de verschoven versie van) A bij P opgeteld. Het optellen van 0 bij P is weggelaten want dit kan gerealiseerd worden door op dat moment P niet te laden. Verder wordt A naar links geschoven en B naar rechts geschoven.

Voor de besturing en het datapad van de tweede versie van de sequentiële vermenigvuldiger kunnen we het voorgaande als volgt samenvatten:

- B wordt een schuifregister zodat de $n \times 1$-bits multiplexer overbodig is.
- Als B vóór het schuiven 0 is, dan hoeft er niet meer opgeteld te worden en is het resultaat bekend. Testen op 0 kan eenvoudig waardoor de teller komt te vervallen.
- Het optellen van 0 bij P kan vervangen worden door P op dat moment niet te laden. Hierdoor kan een 2×1 $2n$-bits multiplexer vervallen.
- De 2×1 $2n$-bits multiplexer om P te wissen wordt in P geplaatst. Dat levert iets minder hardware op en minder tekenwerk.
- De besturing heeft nu meer stuuruitgangen en register B is nu een schuifregister geworden. Dat leidt tot meer logica. Verder zijn register A en de opteller nog steeds $2n$-bits breed.

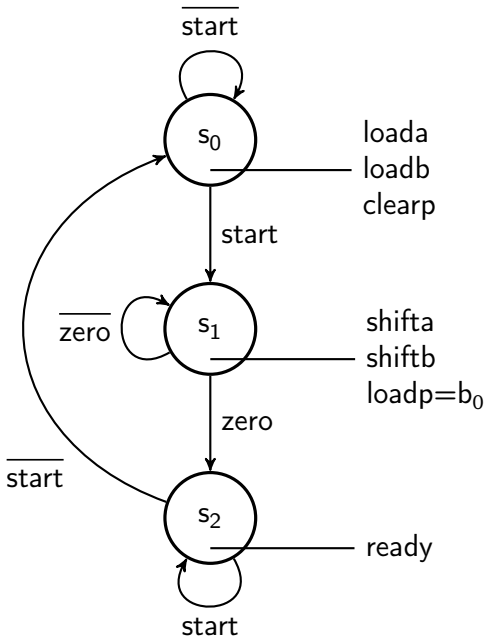
Het blokschema voor het datapad en de besturing is te zien in figuur 12.7. Register A heeft de stuursignalen $loada$ en $shifta$. Register B heeft de stuursignalen $loadb$ en $shiftb$. Register P heeft de stuursignalen $loadp$ en $clearp$. Het testen of de waarde van B 0 is, wordt gerealiseerd met signaal $zero$. Tevens wordt signaal b_0 aangeboden aan de besturing zodat getest kan worden of deze bit een 1 is (zie hiervoor het algoritme in listing 12.2). We hebben hier dus te maken met een overgang van datasignaal naar



Figuur 12.7: *Datapad en besturing voor de sequentiële vermenigvuldiger (versie 2).*

besturingssignaal. Iets dergelijk geldt ook voor signaal $zero$.

In figuur 12.8 is het toestandsdiagram van de tweede versie van de sequentiële vermenigvuldiger te zien. De machine begint in toestand s_0. In deze toestand worden de signalen $loada$, $loadb$ en $clearp$ geactiveerd. In s_0 wordt gewacht totdat signaal $start$ geactiveerd wordt. Dan wordt overgegaan naar toestand s_1. In deze toestand worden de signalen $shifta$ en $shiftb$ geactiveerd. Signaal $loadp$ is afhankelijk van de waarde van b_0. Dit is dus een Mealy-uitgang. In s_1 wordt gewacht zolang signaal $zero$ is gedeactiveerd. Bij activatie van $zero$ wordt overgegaan naar toestand s_2. Daar wordt signaal $ready$ geactiveerd. In s_2 wordt gewacht totdat $start$ is gedeactiveerd. Dan wordt er weer naar toestand s_0 gegaan.



Figuur 12.8: *Het toestandsdiagram van de besturing van de tweede versie van de sequentiële vermenigvuldiger.*

We gaan de tweede versie van de sequentiële vermenigvuldiger in VHDL beschrijven. In het hart van de code zijn drie processen te herkennen: twee voor het beschrijven van de besturing en één voor het datapad.

De eerste twee processen zijn te zien in listings 12.3, samen met de entity. Het systeem heeft de gebruikelijke ingangen voor de klok en de asynchrone reset. Verder zijn er een start-ingang, ingangen voor de data voor A en B en de uitgang voor P en de ready-uitgang. Daarnaast zijn de interne signalen te zien voor de registers A, B en P en de besturingssignalen tussen de toestandsmachine en het datapad. De toestand van de machine wordt in signaal $state$ bijgehouden.

De opbouw van de beschrijving voor de opvolgertoestandlogica en de toestand is rechttoerechtaan. De beschrijving volgt direct uit het toestandsdiagram in figuur 12.8. De beschrijving van de uitgangslogica begint met het toekennen van '0' aan alle uitgangen. Op deze manier hoeven alleen de toekenningen worden gedaan als een uitgang '1' moet worden. Merk op dat signaal $loadp$ afhankelijk is van signaal b_0.

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

entity seq_mult_2 is
    generic (n : integer := 4);
    port (clk : in std_logic;
          areset : in std_logic;
          start : in std_logic;
          dataa : in unsigned(n-1 downto 0);
          datab : in unsigned(n-1 downto 0);
          resultp : out unsigned(2*n-1 downto 0);
          ready : out std_logic
          );
end entity seq_mult_2;

architecture rtl of seq_mult_2 is
constant n_zeros : unsigned(n-1 downto 0) := (others => '0');
signal rega : unsigned(2*n-1 downto 0);
signal regb : unsigned(n-1 downto 0);
signal regp : unsigned(2*n-1 downto 0);
signal shifta, shiftb, loada, loadb, loadp, clearp : std_logic;
signal b0, zero : std_logic;
type state_type is (s0, s1, s2);
signal state : state_type;
begin

    nsl_reg: process (clk, areset) is
    begin
        if areset = '1' then
            state <= s0;
        elsif rising_edge(clk) then
            case state is
                when s0 => if start = '1' then state <= s1; end if;
                when s1 => if zero = '1' then state <= s2; end if;
                when s2 => if start = '0' then state <= s0; end if;
                when others => null;
            end case;
        end if;
    end process;

    ol: process (state, b0) is
    begin
        loada <= '0'; loadb <= '0'; loadp <= '0'; shifta <= '0';
        shiftb <= '0'; clearp <= '0'; ready <= '0';

        case state is
            when s0 => loada <= '1'; loadb <= '1'; clearp <= '1';
            when s1 => shifta <= '1'; shiftb <= '1';
                       if b0 = '1' then loadp <= '1'; end if;
            when s2 => ready <= '1';
            when others => null;
        end case;
    end process;
```

Listing 12.3: *Beschrijving toestandsmachine tweede versie sequentiële vermenigvuldiger.*

De beschrijving van het datapad is te zien in listing 12.4. Het bestaat uit een geklokt stuk en een combinatorisch stuk. De bewerkingen op de registers A, B en P, die vorm hebben gekregen middels de signalen $rega$, $regb$ en $regp$, vallen onder een klokflanksturing. Bij register A wordt getest op de signalen $loada$ en $shifta$ en de juiste toekenningen worden gedaan. Bij register B wordt getest op de signalen $loadb$ en $shiftb$. Bij register P wordt getest op $clearp$ om het register met nullen te laden. Opmerkelijk is dat de operatie $P = P + A$ ook wordt uitgevoerd onder besturing van de klokflank en wel als signaal $loadp$ actief is. De combinatorische logica voor signaal $zero$ en b_0 volgt na het gedeelte met klokflanksturing.

```
    datapad: process (clk, areset, regb) is
    begin
        if areset = '1' then
            rega <= (others => '0');
            regb <= (others => '0');
            regp <= (others => '0');
        elsif rising_edge(clk) then
            if loada = '1' then
                rega <=  n_zeros & dataa;
            elsif shifta = '1' then
                rega <= rega(2*n-2 downto 0) & '0';
            end if;
            if loadb = '1' then
                regb <= datab;
            elsif shiftb = '1' then
                regb <= '0' & regb(n-1 downto 1);
            end if;
            if clearp = '1' then
                regp <= (others => '0');
            elsif loadp = '1' then
                regp <= regp + rega;
            end if;
        end if;
        if regb = n_zeros then zero <= '1'; else zero <= '0'; end if;
        b0 <= regb(0);
    end process;

    resultp <= regp;
end architecture rtl;
```

Listing 12.4: *Beschrijving datapad tweede versie sequentiële vermenigvuldiger.*

12.3 Toestandsmachine met geïntegreerd datapad

We hebben gezien dat het scheiden van de besturing en het datapad helpt om een complex systeem te ontwerpen. Het opstellen van de bijbehorende VHDL-beschrijving is echter omvangrijk. Gelukkig kent VHDL de mogelijkheid om datapadcode te integreren in de besturingscode. De acties op het datapad zijn immers ook afhankelijk van de toestand waarin de besturing zich bevindt. Het voordeel van deze beschrijvingswijze is duidelijk: het geheel wordt een stuk overzichtelijker dan wanneer er via besturingssignalen een actie wordt aangegeven. Tevens sparen we een aantal signalen uit.

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

entity seq_mult_2_fsmd is
    generic (n : integer := 4);
    port (clk : in std_logic;
          areset : in std_logic;
          start : in std_logic;
          dataa : in unsigned(n-1 downto 0);
          datab : in unsigned(n-1 downto 0);
          resultp : out unsigned(2*n-1 downto 0);
          ready : out std_logic
         );
end seq_mult_2_fsmd;

architecture rtl_sig of seq_mult_2_fsmd is
type state_type is (s0, s1, s2);
signal state : state_type;
signal rega : unsigned (2*n-1 downto 0);
signal regb : unsigned (n-1 downto 0);
signal regp : unsigned (2*n-1 downto 0);
constant n_zeros : unsigned (n-1 downto 0) := (others => '0');
begin

    process (clk, areset) is
    begin
        if areset = '1' then
            state <= s0;
            rega <= (others => '0');
            regb <= (others => '0');
            regp <= (others => '0');
        elsif rising_edge(clk) then
            case state is
                when s0 => if start = '1' then state <= s1; end if;
                           rega <= n_zeros & dataa;
                           regb <= datab;
                           regp <= (others => '0');
                when s1 => if regb = n_zeros then state <= s2; end if;
                           if regb(0) = '1' then regp <= regp + rega;
                           end if;
                           rega <= rega(2*n-2 downto 0) & '0';
                           regb <= '0' & regb(n-1 downto 1);
                when s2 => if start = '0' then state <= s0; end if;
                when others => null;
            end case;
        end if;
    end process;

    ready <= '1' when state = s2 else '0';
    resultp <= regp;

end architecture rtl_sig;
```

Listing 12.5: *Besturing met geïntegreerd datapad.*

In listing 12.5 is de beschrijving te zien van de tweede versie van de sequentiële vermenigvuldiger waarbij datapadonderdelen zijn geïntegreerd in de besturing. We gaan hierbij uit van het toestandsdiagram in figuur 12.8. We lopen de beschrijving stap voor stap door.

In de entity zijn de signalen te zien waarmee met de buitenwereld wordt gecommuniceerd. De architecture wordt begonnen met de declaratie van het toestandsregister en de drie registers A, B en P. Merk op dat register A en P $2n$ bits breed zijn en register B is n bits breed. Er wordt ook nog een constante gedeclareerd met in totaal n nullen. Deze constante is nodig bij het laden van register A (de bovenste n bits zijn immers nullen).

Onder besturing van de asynchrone reset wordt het toestandsregister geladen met s_0 en worden de registers A, B en P geladen met nullen. Onder besturing van de klokflank worden de toestanden doorlopen. We herkennen in de toestanden duidelijk de werking van de besturing en de werking van het datapad. In toestand s_0 wordt gewacht totdat signaal $start$ een '1' is. Dan gaat de machine naar toestand s_1. Tevens wordt register A geladen met externe data (en de extra nullen) en wordt register B ook geladen met externe data. Register P wordt geladen met nullen. In toestand s_1 wordt net zolang gewacht totdat register B 0 is. Dan wordt naar toestand s_2 overgegaan. In toestand s_1 wordt tevens register A bij P opgeteld als de minst significante bit van B een '1' is. Verder wordt register A naar links geschoven en register B naar rechts geschoven. De ingeschoven bits zijn '0'. In toestand s_2 is geen datapadactie nodig. Hier wordt alleen gewacht totdat $start$ weer '0' is waarna naar toestand s_0 wordt gegaan. Als laatste is te zien dat signaal $ready$ middels combinatoriek wordt gerealiseerd. Tevens wordt de interne waarde van register P naar buiten gebracht.

Wat opvalt is dat de code compact en helder is beschreven. Het nadeel van deze beschrijvingswijze is dat het nu niet mogelijk is om een blokschema op te stellen omdat er geen duidelijke scheiding is tussen datapad en besturing.

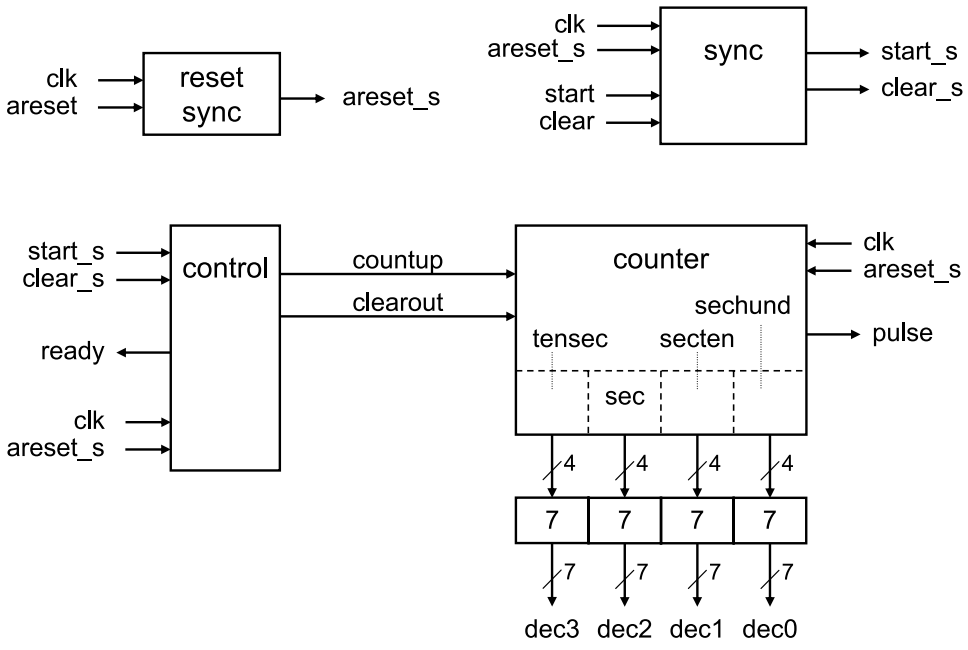
12.4 Voorbeeld: digitale stopwatch

In deze paragraaf bespreken we een eenvoudige digitale stopwatch. De stopwatch heeft de volgende functionele en operationele eisen:

- De te meten tijd ligt tussen 00,00 seconden en 99,99 seconden.
- De resolutie van de te meten tijd is 0,01 seconden.
- De stopwatch heeft een $start$-ingang waarmee de tijdmeting gestart én gestopt wordt.
- De stopwatch heeft een $clear$-ingang waarmee de tijd op 00,00 seconden kan worden gezet.
- De tijdmeting kan alleen op 00,00 seconden worden gezet (middels de $clear$-ingang) als de stopwatch gestopt is.
- De gemeten tijd wordt zichtbaar gemaakt in het decimale talstelsel.

- De stopwatch heeft een *pulse*-uitgang waarmee wordt aangegeven dat de stopwatch tijd aan het meten is. Deze uitgang genereert een puls van 1 Hz als de tijdmeting loopt.
- De stopwatch heeft een *ready*-uitgang waarmee wordt aangegeven of de tijdmeting gestopt is en dus weer gestart kan worden.
- De stopwatch heeft een asynchrone reset waarmee alle onderdelen in de begintoestand worden gedwongen.
- Er is geen mogelijkheid om rondetijden te meten.

Het blokschema is te zien in figuur 12.9. We hebben hier gekozen voor een scheiding tussen datapad en besturing omdat de beschrijving van het datapad nogal complex en groot is. Integratie van de datapadcode in de besturing zal de leesbaarheid niet ten goede komen. De stopwatch bestaat uit een besturing, een teller en vier 7-segment decoders. Verder zijn er twee synchronizer chains om ingangen te synchroniseren. De reset wordt gesynchroniseerd middels een reset synchronizer. De stopwatch heeft twee (gebruikers-)ingangen: *start*, waarmee het telproces kan worden gestart én gestopt en *clear*, waarmee de teller op 0 kan worden gezet. Deze ingangen zijn verbonden met ontdenderde drukknoppen. Verder zijn twee uitgangen beschikbaar te weten *pulse* waarmee een pulssignaal wordt afgegeven als de teller telt en *ready* die aangeeft dat de teller gestopt is met tellen en weer gestart kan worden. Deze uitgangen zijn verbonden met leds. De verstreken tijd wordt weergegeven middels vier 7-segment displays. Verder is te zien dat de besturing met de twee signalen *countup* en *clearout* verbonden is met de teller om aan te geven wat de teller moet doen.



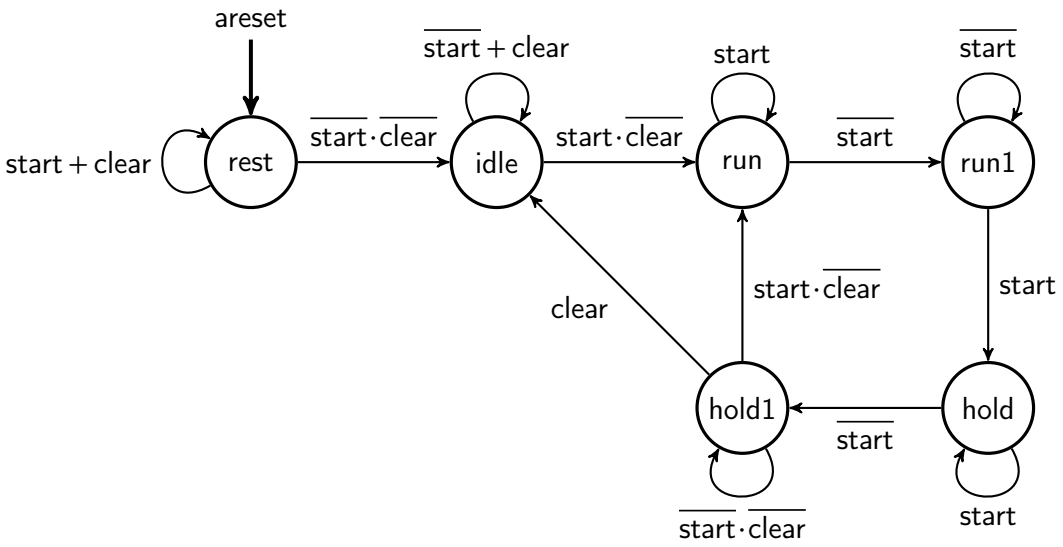
Figuur 12.9: *Blokschema van de stopwatch.*

De werking is al volgt. Na een asynchrone reset wordt gewacht totdat op *start* wordt gedrukt. Als op *start* wordt gedrukt gaat de teller tellen. Signaal *countup* wordt geactiveerd. Als nogmaals op *start* wordt gedrukt, stopt de teller met tellen. Signaal

countup wordt gedeactiveerd. Na nogmaals op *start* te hebben gedrukt, gaat de teller weer verder met tellen. Als de teller gestopt is, kan ook op *clear* worden gedrukt om de teller op 0 te krijgen. Signaal *clearout* wordt dan geactiveerd. Tijdens het tellen werkt de *clear* niet. De verstreken tijd wordt weergegeven in seconden en honderdste seconden. Dat betekent dat de maximale te meten tijd 99,99 seconden bedraagt.

We hebben ervoor gekozen om het *pulse*-signaal door de teller te laten genereren. De reden hiervoor is dat de teller de getelde hoeveelheid tijd bijhoudt en zodoende op de juiste momenten de *pulse*-uitgang kan activeren.

Het toestandsdiagram van de besturing is te zien in figuur 12.10. Na een asynchrone reset komt de machine in toestand *rest*. Daar wordt gewacht tot de ingangen *start* en *clear* niet (meer) geactiveerd zijn. Dan wordt naar toestand *idle* gegaan. Vanuit deze toestand wordt naar toestand *run* gegaan als op *start* wordt gedrukt én niet op *clear* wordt gedrukt. In toestand *run* gaat de teller tellen en wordt gewacht totdat de drukknop van *start* is losgelaten. Reden voor deze opzet is dat een gebruiker de *start*-knop meerdere klokpulsen ingedrukt kan houden. Als de *start*-knop is losgelaten gaat de machine naar toestand *run*1. Ook in deze toestand telt de teller. Na nog een druk op *start* gaat de machine naar toestand *hold* en wordt het tellen gestopt. Ook hier wordt weer gewacht totdat de *start*-knop is losgelaten. In toestand *hold*1 kan de teller op 0 gezet worden door op *clear* te drukken of kan het (verder-)tellen gestart worden door weer op *start* te drukken.



Figuur 12.10: *Het toestandsdiagram van de stopwatch. Hierin zijn de uitgangen niet verwerkt.*

De uitgangswaarden van de machine staan vermeld in tabel 12.1. We hebben ervoor gekozen om de uitgangswaarde niet in het toestandsdiagram op te nemen om de leesbaarheid te bevorderen.

We bespreken alleen de twee belangrijkste onderdelen van de stopwatch: de besturing en de teller. De VHDL-beschrijving van de besturing is te zien in listings 12.6 en 12.7. Het betreft hier de klassieke opbouw van een toestandsmachine. De werking van de toestandsmachine is opgebouwd rond een **case**-statement en in elke tak (herkenbaar aan

Tabel 12.1: *Uitgangswaarden van de toestandsmachine van de stopwatch.*

Toestand	*clearout*	*countup*	*ready*
rest	1	0	0
idle	1	0	1
run	0	1	0
run1	0	1	0
hold	0	0	0
hold1	0	0	1

de **`when`**-statements) wordt met behulp van **`if`**-statements de opvolgertoestand bepaald. Alles gebeurt onder besturing van een opgaande klokflank.

De uitgangen zijn alleen afhankelijk van de toestand en niet van de ingangen. Het betreft hier dus een Moore-machine.

In listings 12.8 en 12.9 is de beschrijving van de teller te zien. Het is gebaseerd op het teller-in-teller-principe uit een eerder hoofdstuk. We gaan ervan uit dat de klokfrequentie van de gehele schakeling hoger is dan de 100 Hz die nodig is om te tellen. Vandaar dat we twee *generics* hebben opgenomen in de entity van de de teller. Met deze twee generics leggen we de systeemfrequentie en de telfrequentie vast. Voor het ontwerp gaan we uit van een frequentie van 50 MHz zoals het geval is op het DE0-experimenteerbordje. De telfrequentie is natuurlijk 100 Hz.

In de code is naast de feitelijke teller ook een *prescaler* opgenomen. Om een resolutie van 100 Hz te krijgen moeten we namelijk een aantal klokpulsen “wegtikken”. Slechts eens in de 500.000 klokpulsen moet de teller met één verhoogd worden. We hoeven dit getal niet zelf uit te rekenen. VHDL heeft de mogelijkheid om tijdens synthese en simulatie het benodigde aantal klokpulsen zelf uit te rekenen. Zodoende kunnen altijd gemakkelijk andere frequenties opgeven worden door de generics te veranderen. Het *pulse*-signaal wordt tijdens het tellen afwisselend 0 en 1. Hiervoor is een tweede prescaler ingebouwd die steeds van 0 tot 49 telt op de frequentie van de teller om een puls te genereren van 1 Hz.

We lopen de code even langs. We beginnen met het declareren van een aantal interne signalen voor de telstand en de prescalers. Voor de telstand gebruiken we vectoren van het type unsigned. Deze signalen worden immers als getallen beschikbaar gesteld aan de buitenwereld. De prescalers zijn interne signalen die niet naar buiten wordt uitgevoerd. We kunnen hiervoor integers gebruiken. De bereiken van de prescalers worden in de code uitgerekend.

Onder besturing van een asynchrone reset worden alle interne signalen op 0 gezet. Onder klokflankbesturing heeft de teller zijn normale werking. Er zijn twee synchrone stuursignalen, één voor het op 0 zetten van de tellers en één voor het tellen van de teller. We testen eerst of de synchrone clear is geactiveerd. Zo ja, dan worden alle tellers op 0 gezet. Dit signaal wordt als eerste getest en heeft dus prioriteit over het andere stuursignaal. Dat maakt niet zoveel uit, de signalen worden vanuit de besturing nooit tegelijk geactiveerd.

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

entity control is
 port (clk : in std_logic;
       areset : in std_logic;
       start : in std_logic;
       clear : in std_logic;
       ready : out std_logic;
       countup : out std_logic;
       clearout : out std_logic
      );
end entity control;

architecture fsm of control is
type state_type is (rest_st, idle_st, run_st, run1_st,
                    hold_st, hold1_st);
signal state : state_type;
begin
    process (clk, areset) is
    begin
        if areset = '1' then
            state <= rest_st;
        elsif rising_edge(clk) then
            case state is
                when rest_st =>  if clear = '0' and start = '0' then
                                     state <= idle_st;
                                 end if;
                when idle_st =>  if clear = '1' then
                                     state <= idle_st;
                                 elsif start = '1' then
                                     state <= run_st;
                                 end if;
                when run_st =>   if start = '0' then
                                     state <= run1_st;
                                 end if;
                when run1_st =>  if start = '1' then
                                     state <= hold_st;
                                 end if;
                when hold_st =>  if start = '0' then
                                     state <= hold1_st;
                                 end if;
                when hold1_st => if clear = '1' then
                                     state <= idle_st;
                                 elsif start = '1' then
                                      state <= run_st;
                                 end if;
                when others => state <= rest_st;
            end case;
        end if;
    end process;
```

Listing 12.6: *Besturing stopwatch (deel 1).*

```
    ol: process (state) is
    begin
        clearout <= '0';
        countup <= '0';
        ready <= '0';
        case state is
            when rest_st =>  clearout <= '1';
            when idle_st =>  clearout <= '1';
                             ready <= '1';
            when run_st =>   countup <= '1';
            when run1_st =>  countup <= '1';
            when hold_st =>  null;
            when hold1_st => ready <= '1';
            when others =>   null;
        end case;
    end process;
end architecture fsm;
```

Listing 12.7: *Besturing stopwatch (deel 2).*

Daarna testen we of het tellen geactiveerd is. Binnen dit deel van de code wordt de prescaler verhoogd. Eens in de 500.000 klokpulsen (bij de gegeven generics) wordt teller met één verhoogd. Tevens wordt de prescaler van de *pulse*-uitgang verhoogd. De *pulse*-uitgang klapt eens in de 50 getelde pulsen om zodat een signaal van 1 Hz wordt gerealiseerd. Verder is de typische teller-in-teller-principe te herkennen. Dit is in feite een 4-decaden BCD-teller. Verder is te zien dat als de teller gestopt is met tellen de *pulse*-uitgang 1 is.

12.5 Opgaven

12.1. Register A is $2n$ bits breed. Ontwerp een derde versie van de vermenigvuldiger waarbij voor A slechts n bits nodig zijn. Hint: P schuiven in plaats van A.

12.2. De opteller is $2n$ bits breed. Ontwerp een vierde versie van de vermenigvuldiger waarbij voor de opteller minder bits nodig zijn.

12.3. Een nadeel van de ontworpen besturing is dat P wordt gewist in de rusttoestand. Pas het toestandsdiagram aan zodat P behouden blijft.

12.4. Herontwerp de toestandsmachine voor de tweede versie van de vermenigvuldiger als een Mealy-machine.

12.5. Toon aan dat een vermenigvuldiging van twee n-bits getallen altijd past in $2n$ bits.

12.6. Een slimme student opperde dat signaal b_0 direct verbonden kan worden aan signaal $loadp$. Werkt de vermenigvuldiger dan nog correct? Motiveer het antwoord.

12.7. Ontwerp een systeem dat van de inhoud van een 8-bits register het aantal enen bepaalt dat in het register is opgeslagen. Ontwerp zowel het datapad als de besturing.

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

entity counter is
 generic (sysfreq : integer := 50000000;
          watchfreq : integer := 100
         );
 port (clk : in std_logic;
       areset : in std_logic;
       clear : in std_logic;
       countup : in std_logic;
       tensec : out unsigned(3 downto 0);
       sec : out unsigned(3 downto 0);
       secten : out unsigned(3 downto 0);
       sechund : out unsigned(3 downto 0);
       pulse : out std_logic
    );
end entity counter;

architecture rtl of counter is
signal tensec_int : unsigned(3 downto 0);
signal sec_int : unsigned(3 downto 0);
signal secten_int : unsigned(3 downto 0);
signal sechund_int : unsigned(3 downto 0);
signal prescaler_int : integer range 0 to (sysfreq/watchfreq-1);
signal pulsecount_int : integer range 0 to watchfreq/2-1;
signal pulse_int : std_logic;
begin
    process (clk, areset, prescaler_int) is
    begin
        if areset = '1' then
            tensec_int <= (others => '0');
            sec_int <= (others => '0');
            secten_int <= (others => '0');
            sechund_int <= (others => '0');
            prescaler_int <= 0;
            pulse_int <= '0';
            pulsecount_int <= 0;
        elsif rising_edge(clk) then
            if clear = '1' then
                tensec_int <= (others => '0');
                sec_int <= (others => '0');
                secten_int <= (others => '0');
                sechund_int <= (others => '0');
                prescaler_int <= 0;
                pulse_int <= '1';
                pulsecount_int <= 0;
```

Listing 12.8: *Teller stopwatch (deel 1).*

```
            elsif countup = '1' then
                if prescaler_int = (sysfreq/watchfreq-1) then
                    if pulsecount_int = (watchfreq/2-1) then
                        pulsecount_int <= 0;
                        pulse_int <= not pulse_int;
                    else
                        pulsecount_int <= pulsecount_int + 1;
                    end if;
                    prescaler_int <= 0;
                    if sechund_int = "1001" then
                        sechund_int <= "0000";
                        if secten_int = "1001" then
                            secten_int <= "0000";
                            if sec_int = "1001" then
                                sec_int <= "0000";
                                if tensec_int = "1001" then
                                    tensec_int <= "0000";
                                else
                                    tensec_int <= tensec_int + 1;
                                end if;
                            else
                                sec_int <= sec_int + 1;
                            end if;
                        else
                            secten_int <= secten_int + 1;
                        end if;
                     else
                         sechund_int <= sechund_int + 1;
                     end if;
                 else
                     prescaler_int <= prescaler_int + 1;
                 end if;
              else
                  pulse_int <= '1';
             end if;
        end if;

    end process;

    tensec <= tensec_int;
    sec <= sec_int;
    secten <= secten_int;
    sechund <= sechund_int;
    pulse <= pulse_int;

end architecture rtl;
```

Listing 12.9: *Teller stopwatch (deel 2).*

13

Eenvoudige microprocessor

De datapadsystemen die geïntroduceerd zijn in hoofdstuk 12 maken het mogelijk om een complex digitaal systeem in behapbare stukken te ontwerpen. De werking van een datapadsysteem is vastgelegd door het datapad waarlangs de data wordt verwerkt en de besturing die ervoor zorgt dat het datapad de juiste bewerkingen uitvoert. Door gebruik te maken van flipflops is het mogelijk om de logische werking en de timing grotendeels van elkaar te scheiden. Een nadeel van een datapadsysteem is dat het een vaste bewerking uitvoert, dat wil zeggen dat het maar voor één doel gebruikt kan worden.

Zou het niet mooi zijn om een digitaal systeem te bouwen dat geschikt is voor het uitvoeren van een willekeurig algoritme? Denk hierbij aan diverse rekenkundige operaties, zoals vermenigvuldiging, deling of het berekenen van het gemiddelde. Zo'n systeem verwerkt data (Engels: to process) en wordt in de volksmond *processor* genoemd. Door de miniaturisering van transistoren is het mogelijk om een complete processor op één ic te plaatsen. Dit wordt een *microprocessor* genoemd. Dankzij VHDL is het relatief eenvoudig een simpele processor te ontwerpen.

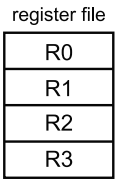
In dit hoofdstuk behandelen we de opbouw en programmering van een eenvoudige microprocessor. De processor is opgebouwd rond een datapad met registers en een rekeneenheid en werkt uitsluitend met unsigned getallen van 8 bits. Het datapad wordt aangestuurd door middel van besturingscodes die worden opgeslagen in een ROM. Steeds wordt een besturingscode uit de ROM aangeboden aan de datapadonderdelen. We gebruiken hiervoor een teller. Het zou zinloos zijn een processor te ontwerpen die niet met de buitenwereld kan communiceren. Daarom wordt de processor uitgerust met een invoer- en uitvoereenheid. Voor de kenners onder ons: de processor heeft geen externe RAM, geen interrupts en geen stack. Voor een eerste indruk van een processor zijn deze onderdelen niet van belang. Het beperkt wel de mogelijkheden van de processor.

Hoewel we nu een globaal beeld van het uiteindelijke ontwerp hebben, bouwen we de processor in stappen op. Eerst behandelen we het datapad en vervolgens komen via een aantal stappen tot het uiteindelijke ontwerp. Bij elke stap wordt een uitbreiding gedaan op de vorige stap. Nadat het uiteindelijke ontwerp is gerealiseerd, gaan we de processor

voorzien van programmatuur. We gaan dus software ontwikkelen. We doen dit door zelf de besturingscodes in de ROM te plaatsen. Aan het eind van het hoofdstuk worden nog enkele uitbreidingsmogelijkheden beschreven.

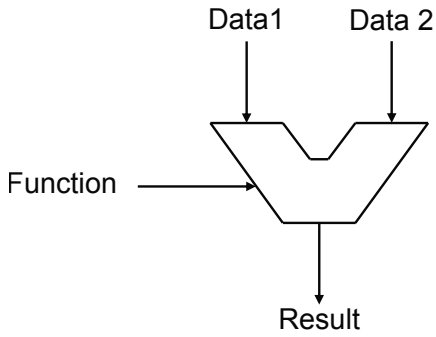
13.1 Opbouw datapad

Een processor verwerkt data. Zo'n systeem moet dan *registers* hebben om (tussen-)resultaten op te slaan en een *rekeneenheid* voor de uitvoeren van de berekeningen. Veel microprocessoren hebben een aantal gelijksoortige registers. Deze worden samengenomen in een *register file*. De registers bestaan uit identieke flipflops en hebben een enable-faciliteit die ervoor zorgt dat een bepaald register geladen kan worden. Alle registers worden geklokt op hetzelfde kloksignaal. In figuur 13.1 is de schematische weergave van de register file te zien.



Figuur 13.1: *De register file.*

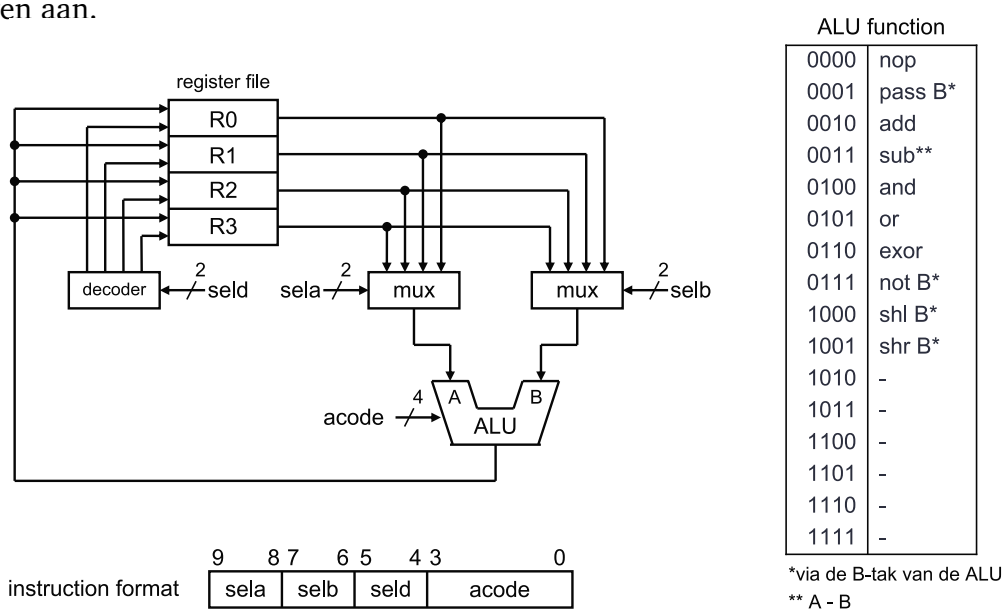
De rekeneenheid wordt *Arithmetic and Logic Unit* genoemd, afgekort tot ALU. Het is een combinatorische schakeling die een aantal rekenkundige en logische bewerkingen kan uitvoeren. Te denken valt aan het optellen en aftrekken van twee getallen, bit-voor-bit AND, OR en EXOR uitvoeren, schuiven, roteren en het ongewijzigd door laten van data. Complexe ALU's kunnen ook vermenigvuldigen en delen. Omdat de ALU verschillende bewerkingen kan uitvoeren, moet met behulp van een stuurcode de juiste bewerking worden aangegeven. In figuur 13.2 is het symbool van de ALU te zien. $Data1$ en $Data2$ stellen de data voor en $Result$ stelt het resultaat voor van de bewerking aangegeven door $Function$.



Figuur 13.2: *De Arithmetic and Logic Unit (ALU).*

De opbouw van het datapad is te zien in figuur 13.3. Het bestaat uit vier registers met de klinkende namen $R0$, $R1$, $R2$ en $R3$ en een ALU. De registernamen zijn zo gekozen zodat uitbreidingen hierop makkelijk kunnen worden ingevoerd. De ALU heeft twee dataingangen die A en B genoemd worden en kan dus een operatie op twee getallen

uitvoeren. We gaan ervan uit dat de ALU in totaal 16 bewerkingen kan uitvoeren zodat de functie van de ALU wordt aangegeven middels een 4-bits code op signaal $acode$. Omdat er vier register zijn, moeten twee multiplexers gebruikt worden om de data uit de juiste registers te selecteren. De multiplexer van de *A*-ingang van de ALU wordt aangestuurd door een 2-bits code $sela$ (select *A*-input), de *B*-ingang wordt aangestuurd door de 2-bits code $selb$ (select *B*-input). Het resultaat van een bewerking moet worden opgeslagen in één van de registers. Dit wordt gerealiseerd door een decoder die wordt aangestuurd door signaal $seld$ (select *D*estination). Kenmerkend voor het datapad is de lus die begint bij de registers en die via de ALU eindigt bij de registers. In de figuur is het kloksignaal voor de registers achterwege gelaten. We nemen het kloksignaal als een gegeven aan.



ALU function	
0000	nop
0001	pass B*
0010	add
0011	sub**
0100	and
0101	or
0110	exor
0111	not B*
1000	shl B*
1001	shr B*
1010	-
1011	-
1100	-
1101	-
1110	-
1111	-

*via de B-tak van de ALU
** A - B

Figuur 13.3: *Opbouw van het datapad met besturingscodes ALU.*

In totaal zijn er tien besturingsingangen. De besturing wordt gerealiseerd met bits. We zouden de besturingsingangen bijvoorbeeld aan schakelaars kunnen koppelen. Er zijn twee bits nodig voor $sela$, twee bits nodig voor $selb$, twee bits nodig voor $seld$ en vier bits nodig voor $acode$. Om het een en ander op eenduidige wijze weer te geven, stellen we het *instructieformaat* op, te zien in de figuur. (In feite maakt het niet uit hoe de besturingsbits aansluiten op schakelaars, maar het geeft ons enige houvast.)

Voor de codering voor $sela$, $selb$ en $seld$ wordt de binaire telcode gebruikt. Dat is ook de meest voor de hand liggende codering. Dus *R*0 wordt geselecteerd met code 00, *R*1 met code 01, *R*2 met code 10 en *R*3 met code 11.

Nu moeten we de besturingscodes voor de ALU bedenken. We kunnen de code zelf opstellen. Er is geen vaste volgorde. De eerste code is `nop` dat *no operation* betekent. Het geeft aan dat de ALU "niets" moet doen. Dit lijkt een onzinnige bewerking want de ALU moet toch immers iets te doen hebben. Het gebruik van `nop` wordt later duidelijk. De tweede operatie is `pass B`. De inhoud van het op de *B*-ingang aangeboden register wordt onveranderd doorgelaten. Deze bewerking is nodig om de inhoud van een register in een ander register op te slaan. De derde en vierde bewerking zijn optellen en aftrekken. Hierbij worden de *A*- en *B*-ingangen gebruikt. Voor aftrekken geldt dat de waarde op

de *B*-ingang wordt afgetrokken van de waarde op de *A*-ingang. Er is geen mogelijkheid om *A* van *B* af te trekken (dat is ook niet nodig). De vijfde, zesde en zevende bewerking zijn respectievelijk de logische AND, OR en EXOR. Hierbij worden waarden op de *A*- en *B*-ingang bit-voor-bit (*bitwise*) bewerkt. De achtste bewerking is de NOT-operatie op de waarde van de *B*-ingang. Ten slotte zijn er nog twee schuifoperaties: `shl B` schuift de waarde op de *B*-ingang één plek naar links en `shr B` schuift de waarde op de *B*-ingang één plek naar rechts. De overige zes combinaties worden voorlopig niet gebruikt.

13.2 Eerste versie van de processor: datapad, teller en ROM

Nu we de codering voor de tien besturingsbits hebben vastgesteld, kunnen we het datapad instellen voor bewerkingen. We gaan het datapad *instructies* geven, vandaar dat we ook wel van instructiebits spreken. Elke reeks van instructiebits wordt een *instructie* genoemd. Als we meerdere instructies achter elkaar uitvoeren, spreken we over een *programma*.

In tabel 13.1 is een klein programma te zien. Links in de tabel is het stapnummer te zien. We beginnen met stap 0. Rechts daarvan is de instructie te zien die we willen uitvoeren. Er is ook een kolom toegevoegd met de binaire codering van het stapnummer. Geheel rechts zijn de waarde van de instructiebits te zien. Een 'x' geeft aan dat de waarde als don't care is gespecificeerd.

Tabel 13.1: *Een klein programma voor besturing van het datapad.*

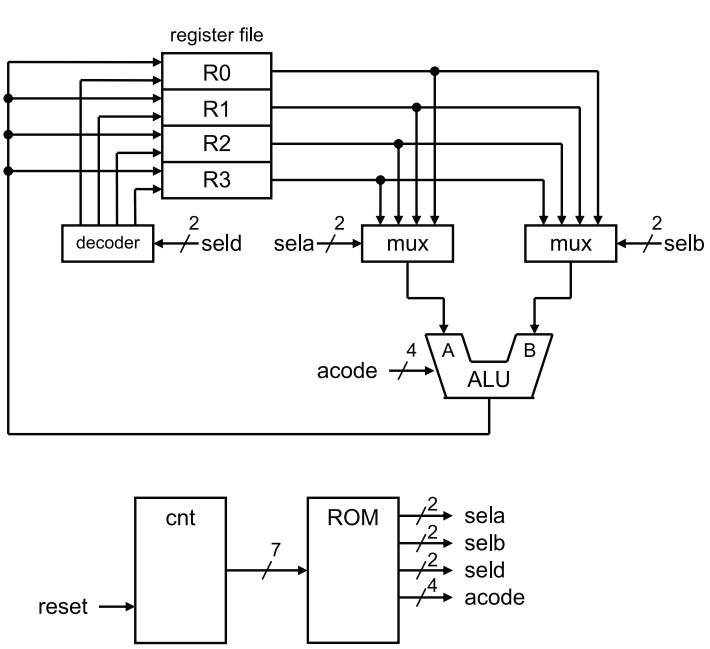
Stap	Instructie	Stap (bits)	Bitpatroon instructie
0	$R0 = R1 + R2$	...0000	01 10 00 0010
1	$R3 = R0 - R3$	...0001	00 11 11 0011
2	$R0 = shl\ R0$	...0010	xx 00 00 1000
3	$R2 = not\ R2$	...0011	xx 10 10 0111
4	$R1 = R3$	...0100	xx 11 01 0001
...	.....	.....	.....

Laten we eens stap 0 bekijken. We voeren hier de bewerking $R0 = R1 + R2$ uit. Dat betekent dat de instructiebits als volgt zijn samengesteld: $sela = 01, selb = 10, seld = 00$ en $acode = 0010$. De gehele instructie wordt dan 0110000010. Voor stap 1 geldt iets dergelijks alleen wordt nu een aftrekking uitgevoerd. Bij de overige stappen is de instelcode voor $sela$ niet van belang omdat alle operaties op de waarde op de *B*-ingang van de ALU worden uitgevoerd.

Aan de tabel kunnen twee zaken worden opgemerkt. De stapnummers worden steeds met één verhoogd. Dit is het gedrag van een *teller*. De instructiebits zijn per stap verschillend. Dit wordt gerealiseerd met combinatorische logica. Het rechter deel van de tabel kan namelijk gezien worden als een waarheidstabel. De besturing kan nu worden gerealiseerd met een teller en een ROM. We rusten de teller uit met zeven telbits zodat we 128 instructies kunnen opslaan in de ROM. Dit is ruim voldoende voor de processor.

Aan het eerder beschreven datapad worden een 7-bits teller en een ROM toegevoegd. De ROM heeft 128 plaatsen die doorgaans *adressen* worden genoemd. Elk adres herbergt een 10-bits instructiecode. In totaal bestaat de ROM dus uit 1280 bits.

In figuur 13.4 is de eerste versie van de processor te zien. Het datapad is nu uitgebreid met een teller en een ROM. De teller wordt geklokt op een (niet getekend) kloksignaal en kan gereset worden zodat het op 0 begint (stap 0). De 7-bits waarde uit de teller wordt aangeboden aan de adresbits van de ROM. Zo worden een voor een de instructies geselecteerd. Goed beschouwd vormen de teller en een ROM een *toestandsmachine* met Moore-uitgangen. De telstand kan namelijk gezien worden als de toestand van de machine. We hanteren in feite het datapad-besturingsmodel uit hoofdstuk 12.



ALU function

0000	nop
0001	pass B
0010	add
0011	sub
0100	and
0101	or
0110	exor
0111	not B
1000	shl B
1001	shr B
1010	-
1011	-
1100	-
1101	-
1110	-
1111	-

Figuur 13.4: *Eerste versie van de eenvoudige processor: datapad, teller en ROM.*

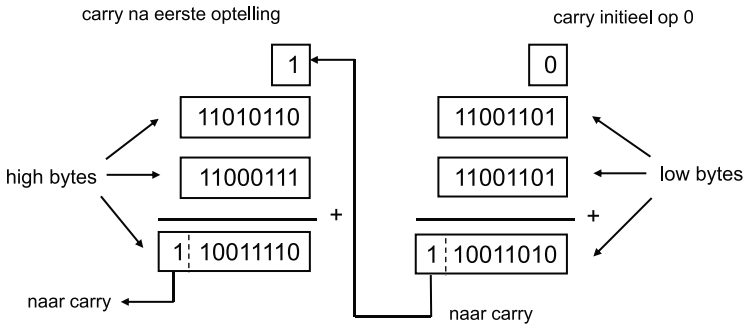
13.3 Tweede versie van de processor: flags, constanten en externe data

We gaan de processor nu uitbreiden met drie nieuwe onderdelen: de status flags, gebruik van een constante waarde en invoer en uitvoer. Daarvoor moeten we echter wel de architectuur van het datapad op kleine punten aanpassen.

Soms is niet het resultaat van een bewerking interessant maar wel het effect ervan. Denk hierbij aan het testen of A gelijk is aan B. Dat kan eenvoudig worden gedaan door de operatie $A-B$. Immers $A=B$ kan ook geschreven worden als $(A-B)=0$. Dit effect wordt opgeslagen in een flipflop, *zero flag* (Z) genoemd. Wanneer het resultaat van een bewerking nul is, wordt de zero flag op 1 gezet, anders wordt de zero flag op 0 gezet. Omdat het resultaat van zo'n test niet moet worden opgeslagen wordt de decoder voor het terugschrijven naar een register uitgebreid met een enable. Hierdoor wordt het mogelijk om geen van de registers te laden met het resultaat uit de ALU.

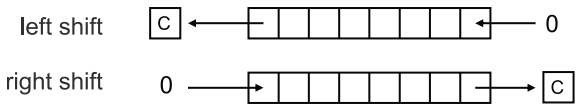
Bij een optelling van twee 8-bits getallen kan het resultaat groter worden dan 8 bits. Het negende bit wordt opgeslagen in de *carry flag* (C). De carry flag geeft overflow of underflow bij unsigned bewerkingen aan. (Merk op dat de processor alleen met unsigned getallen kan werken.) De carry flag wordt onder andere gebruikt bij multi-byte bewerkingen. Zo kunnen 16- en 32-bits optellingen worden gedaan met een 8-bits

processor. Dit is te zien in figuur 13.5. Initieel wordt de carry flag op 0 gezet. Daarna worden de twee minst significante bytes bij elkaar opgeteld. De uitgaande carry wordt opgeslagen in de carry flag. Daarna worden de twee meest significante bytes opgeteld. De carry uit deze optelling wordt weer opgeslagen in de carry flag.



Figuur 13.5: *Optelling van een 16-bits getal middels twee optellingen van 8 bits.*

De carry flag wordt ook gebruikt bij schuifoperaties. Het uitgeschoven bit wordt in de carry flag geplaatst. Het ingeschoven bit is 0. Dit is te zien bij de twee schuifoperaties in figuur 13.6.



Figuur 13.6: *Naar links en naar rechts schuiven met behulp van de carry flag.*

Soms is het noodzakelijk om de flags op een logische 0 of 1 te zetten. Denk hierbij aan (het begin van) een multibyte optelling. De flags kunnen echter alleen via de ALU gemanipuleerd worden. De ALU krijgt twee nieuwe operaties:

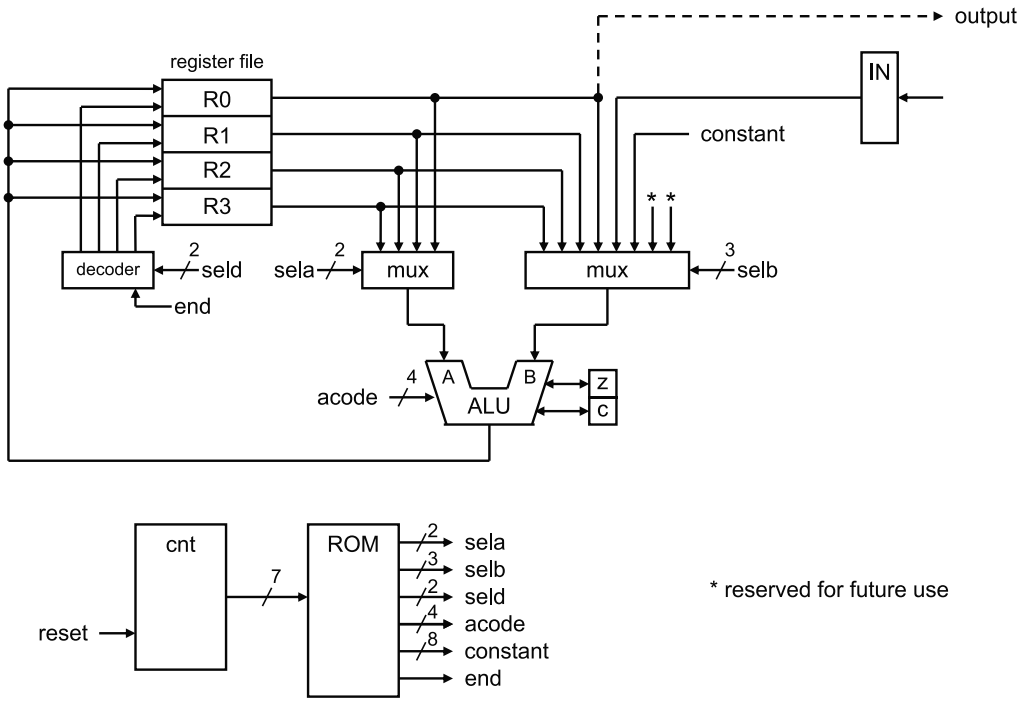
- schrijf flags uit B (`wrf B`)
- lees flags (`rdf`)

In een processor worden de flags gebundeld tot één geheel. Er zijn diverse benamingen in omloop: Processor Status Word (PSW), Status Register (SREG), Condition Code Register (CCR).

In de eerste versie van de processor is het niet mogelijk om data van buiten af in te laden. Daarnaast is het niet mogelijk om een constante te laden. Bedenk dat de constante 0 veel gebruikt wordt (bijvoorbeeld tellers op nul zetten). Het datapad moet aangepast worden zodat de twee invoermogelijkheden kunnen worden gebruikt. De multiplexer aan de *B*-ingang van de ALU wordt uitgebreid van vier naar acht ingangen. Het aantal selectiesignalen wordt dan drie. Verder voeren we de inhoud van register *R*0 naar buiten.

In figuur 13.7 is de tweede versie van de processor te zien. De decoder links is uitgebreid met het signaal *end* (enable *D*estination). Rechts is te zien dat de multiplexer bij de *B*-ingang van de ALU is uitgebreid van vier naar acht ingangen. Naast de al gebruikte ingangen voor de registers wordt ook een ingang verbonden met een extern ingangssignaal en wordt een ingang verbonden met een constante vanuit de ROM. De ROM moet

hiervoor uitgebreid worden van 10 naar 20 bits. Het totaal aantal bits van de ROM komt hiermee op 2560.



Figuur 13.7: *Tweede versie van de eenvoudige processor: uitbreiding met flags, constante en I/O.*

Twee ingangen van de multiplexer aan de *B*-ingang van de ALU worden niet gebruikt. We geven dit aan met “reserved for future use” (gereserveerd voor toekomstig gebruik). Dit betekent dus dat we deze ingangen niet moeten gebruiken (maar het is altijd leuk om te kijken wat er gebeurt).

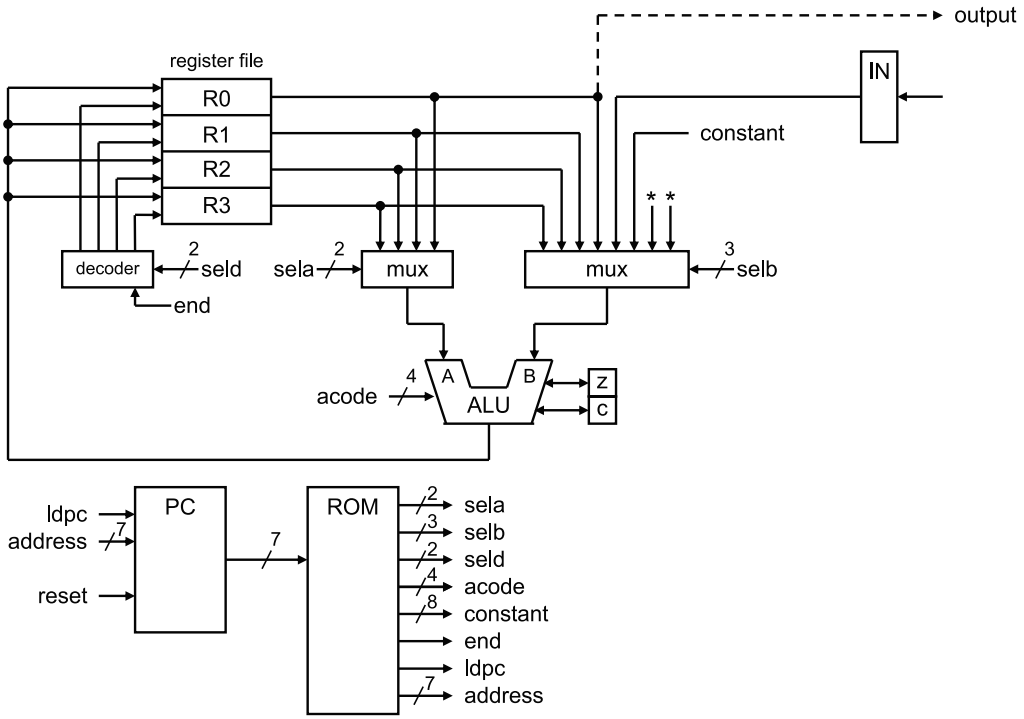
13.4 Derde versie van de processor: springen

De teller die bijhoudt welke instructie wordt uitgevoerd heet *programmateller*. Deze naam is gekozen om de hardware weer te geven: een teller. Beter zou zijn geweest om de naam instructiewijzer te gebruiken want dat is wat de teller doet: aanwijzen welke instructie wordt uitgevoerd. In de computertechniek wordt echter vaak de term Program Counter gebruikt, onder andere bij processoren van Motorola, Atmel en ARM, afgekort tot PC. Intel gebruikt bij de x86- en x64-processoren de term Instruction Pointer, afgekort tot IP. In dit boek gebruiken we de term Program Counter, afgekort tot PC.

Een van de nadelen van de tweede versie van de processor is dat het systeem eeuwig door blijft gaan met het sequentieel uitvoeren van instructies. Wanneer we een aantal instructies steeds moeten herhalen is dit een probleem. We hebben dan (theoretisch) een oneindig grote ROM nodig (en teller) die steeds eenzelfde reeks instructies herhalen.

Slimmer is om de PC opnieuw te laten beginnen op een bepaald punt. We moeten *springen* (Engels: jump) in het programma. Dit kan elegant worden opgelost door de PC te laden met een getal, waar vanaf de PC opnieuw moet beginnen. We noemen zo’n getal een *adres*.

In feite is het inbouwen van een *sprongopdracht* geen moeilijke klus. De ROM krijgt een extra uitgang die naar een stuuringang gaat op de PC, de zogenoemde $ldpc$-ingang. Het benodigde adres wordt ook in de ROM opgeslagen. Dit is te zien in figuur 13.8. De PC is uitgebreid met een $ldpc$-ingang en een $address$-ingang. Deze bits komen van de ROM. De ROM heeft nu 28-bits instructies en de grootte bedraagt 3584 bits.



Figuur 13.8: *Derde versie van de eenvoudige processor: uitbreiding met sprongmogelijkheid.*

De PC geeft een 7-bits adres aan de ROM af en selecteert zo op enig moment één van de 128 adressen. Als de PC op 127 staat en verder moet tellen, dan wordt de waarde van de PC weer 0. De teller telt immers cyclisch. We noemen dit een *roll over*.

13.5 Vierde versie van de processor: conditioneel springen

We kunnen het nog interessanter maken door te springen als er een bepaalde conditie voorkomt, bijvoorbeeld als het resultaat van de laatste berekening nul is. Dan wordt de zero flag op 1 gezet. We kunnen dan beslissingen maken, bijvoorbeeld "als $R0 = R2$, dan ...". Dit wordt in de literatuur *conditioneel springen* (Engels: conditional jump) genoemd. We gebruiken hiervoor de flags en een nieuw stukje logica die de $ldpc$-ingang van de PC aanstuurt. Noot: in de literatuur wordt ook het Engelse woord *branch* veel gebruikt, dit betekent vertakken.

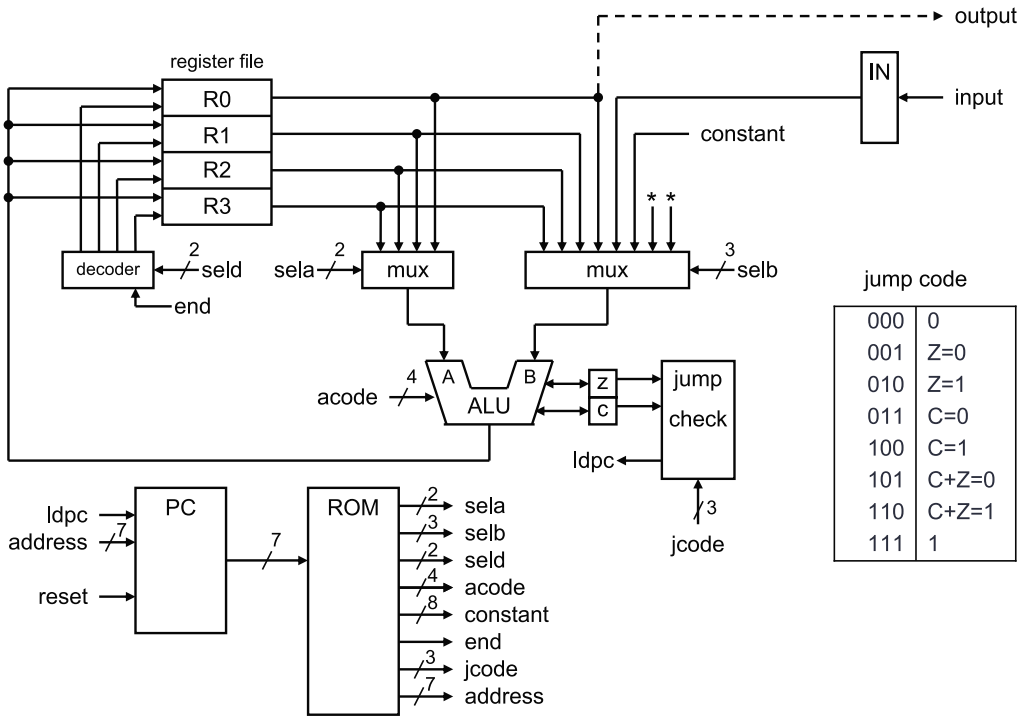
Alle beslissingen zoals $A = B$ en $A > B$ kunnen worden omgewerkt naar een aftrekoperatie en een test op de flags. De zes relationele operatoren en hun invloed op de flags is gegeven in tabel 13.2. Een test wordt omgezet in een aftrekoperatie. Het resultaat van de aftrekoperatie wordt niet opgeslagen. Alleen de flags worden aangepast. Merk op dat het hier unsigned getallen betreft. In de vijfde en zesde regel van de tabel worden C en Z in een logische OR verwerkt.

Tabel 13.2: *De zes relationele operatoren en de invloed op de flags.*

Test	Operatie	Flags
$A \neq B$	$(A-B) \neq 0$	$Z = 0$
$A = B$	$(A-B) = 0$	$Z = 1$
$A \geq B$	$(A-B) \geq 0$	$C = 0$
$A < B$	$(A-B) < 0$	$C = 1$
$A > B$	$(A-B) > 0$	$C + Z = 0$
$A \leq B$	$(A-B) \leq 0$	$C + Z = 1$

Naast de zes conditionele sprongen zijn er ook nog twee niet-conditionele sprongen nodig: niet springen (Engels: never jump) dat effectief betekent dat de $ldpc$-ingang op 0 gezet wordt en altijd springen (Engels: always jump) dat effectief betekent dat de $ldpc$-ingang op 1 gezet wordt. In totaal zijn er acht sprongmogelijkheden. De sprongmogelijkheden kunnen met drie bits gecodeerd worden.

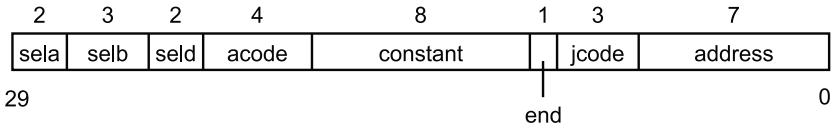
In figuur 13.9 is de vierde versie van de processor te zien. De processor is nu voorzien van een jump checker, een combinatorische schakeling die aangeeft of de PC moet worden geladen met een sprongadres. De $ldpc$-ingang van de ROM is nu vervangen door een 3-bits sprongcode aangegeven door het signaal $jcode$. De acht coderingen zijn te zien in de tabel in de figuur. De eerste regel geeft aan dat er nooit gesprongen moet worden ($ldpc = 0$) en de laatste regel geeft aan dat er altijd gesprongen moet worden ($ldpc = 1$). De overige zes codes komen overeen met de zes relationele operatoren.



Figuur 13.9: *Vierde versie van de eenvoudige processor: uitbreiding met conditionele sprongmogelijkheid.*

De bitpatronen die de instructies vormen zijn 30 bits breed. In totaal zijn er 128 instruc-

ties mogelijk (adres is 7 bits). De complete ROM beslaat dan 3840 bits aan data. In figuur 13.10 is het instructieformaat te zien.



Figuur 13.10: *Het instructieformaat van de vierde versie van de eenvoudige processor.*

13.6 Software-ontwikkeling voor de processor

De microprocessor is nu gereed, dat wil zeggen: de hardware is ontwikkeld. Om het systeem nu te gebruiken moeten de bitpatronen van de instructies in de ROM worden vastgelegd. Deze ROM is eventueel te herprogrammeren (PROM, Flash-ROM, ...) waardoor het systeem een andere functie uitvoert. De hardware-ontwikkeling wordt nu verlaten en gaat over in software-ontwikkeling.

We laten het ontwikkelen van software zien aan de hand van een voorbeeld: het vermenigvuldigen van twee unsigned 8-bits getallen. Dit moet in software worden gerealiseerd want de processor heeft geen *hardware multiplier* aan boord. Gelukkig is vermenigvuldigen ook in software te realiseren.

We laten de werking van de vermenigvuldiging eerst zien in *pseudo-code*. Dat is een niet-bestaande hogere programmeertaal dat veel wegheeft van Pascal en Ada. We doen dit om eerst een algoritme voor de vermenigvuldiging te beschrijven. Daarna gaan we stap voor stap het programma geschikt maken door steeds verder richting de bitpatronen te gaan. Uiteindelijk zijn we zover afgedaald dat we een op een de instructies uit de pseudo-code kunnen omzetten in bitpatronen. Eigenlijk spelen we dan voor *compiler*.

Vermenigvuldigen is niets meer dan herhaald optellen. We gaan de vermenigvuldiging dan ook op deze manier ontwikkelen. In listing 13.1 is de eerste versie van de vermenigvuldiging te zien. Het is opgebouwd rond een **`for`**-lus. Eerst wordt de uitkomst op 0 gezet. Daarna wordt middels de **`for`**-lus 11 keer het getal 13 bij de uitkomst opgeteld.

```
1 uitkomst := 0;
2 for i := 1 to 11 do
3     uitkomst := uitkomst + 13;
4 end for;
```

Listing 13.1: *Pseudo-code voor vermenigvuldiging van 11 met 13 (versie 1).*

De processor is niet in staat om de **`for`**-lus direct uit te voeren. We zetten de **`for`**-lus om naar een **`while`**-lus en gebruiken de variabelen a en b voor de getallen 13 en 11. Dat doen we met in het achterhoofd het feit dat de getallen in de processor in registers worden opgeslagen.

De code is te zien in listing 13.2. Het oplopende bereik van de **`for`**-lus is nu aangepast naar een aflopend bereik bij de **`while`**-lus. Elke keer als de lus wordt doorlopen wordt de waarde van a opgeteld bij de uitkomst. De waarde van b wordt met één verlaagd. Dat

gebeurt zolang b ongelijk is aan 0. Variabele b vertoont het gedrag van een (af-)teller. Na het uitvoeren van de **while**-lus is de waarde van b gelijk aan 0.

```
a := 13;
b := 11;
uitkomst := 0;
while (b <> 0) do
    uitkomst := uitkomst + a;
    b := b - 1;
end while;
```

Listing 13.2: *Pseudo-code voor vermenigvuldiging van 11 met 13 (versie 2).*

De processor is ook niet in staat om de **while**-lus als één instructie uit te voeren. We moeten de lus omzetten in testen (**if**) en springen (**goto**). Bij een spronginstructie wordt naar een ander deel van het programma gesprongen. Om dit elegant weer te geven maken we gebruik van *labels*. De labels zorgen ervoor dat we ons (voorlopig) niet druk hoeven te maken over de sprongadressen (die weten we namelijk nog niet). Dit alles is te zien in listing 13.3. Opmerkelijk is de laatste **goto**-instructie. Deze sprongopdracht is nodig om ervoor te zorgen dat de processor in een eeuwig durende lus blijft wachten.

```
a := 13;
b := 11;
uitkomst := 0;
opnieuw: if (b = 0) goto klaar;
         uitkomst := uitkomst + a;
         b := b - 1;
         goto opnieuw;
klaar:   goto klaar;
```

Listing 13.3: *Pseudo-code voor vermenigvuldiging van 11 met 13 (versie 3).*

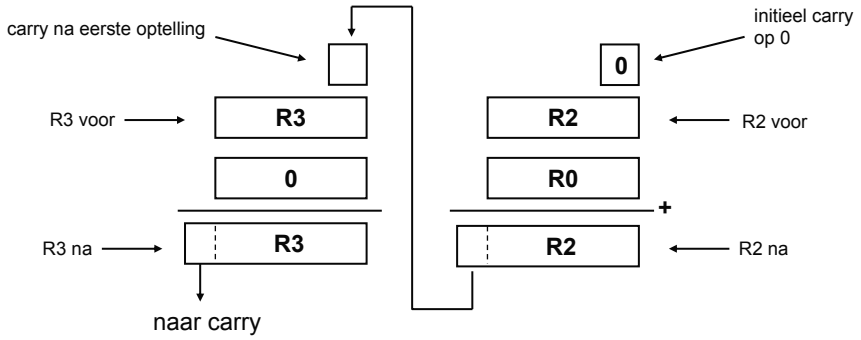
We hebben het **while**-statement vervangen door een aantal statements. Als eerste wordt getest of de waarde van b gelijk is aan 0. Als dat zo is, wordt er *gesprongen* naar een laatste deel van het programma, namelijk naar label $klaar$. Merk op dat we op de *inverse* relationele operator hebben getest. Bij het **while**-statement testen we op ongelijk aan 0, bij het **if**-statement testen we op gelijk aan 0. Daarna wordt a bij de uitkomst opgeteld. Vervolgens wordt gesprongen naar label $opnieuw$ en wordt de lus nogmaals uitgevoerd (als b ongelijk aan 0 is natuurlijk).

De test op 0 moet gesplitst worden in twee instructies: één die de zero flag aanpast aan de waarde van b (de flag wordt op 0 of 1 gezet) en één die conditioneel naar een label springt. Zie listing 13.4.

De volgende stap is het toekennen van registers aan de variabelen (*register mapping*). Voor variabele a gebruiken we register $R0$. Voor variabele b gebruiken we register $R1$. Merk op dat een 8x8-bits vermenigvuldiging een 16-bits resultaat oplevert. Het resultaat moet in twee registers worden opgeslagen en bewerkt. We gebruiken hiervoor de registers $R2$ en $R3$. Register $R2$ wordt gebruikt voor het minst significante byte van de uitkomst, register $R3$ wordt gebruikt voor de meest significante byte. In figuur 13.11 is te zien hoe de 16-bits optelling wordt uitgevoerd met 8-bits registers.

```
a := 13;                              // initialize variables
b := 11;
uitkomst := 0;
opnieuw: zero := (b - 0) = 0;         // set/reset Z flag
         if zero = 1 goto wacht;      // test Z flag
         uitkomst := uitkomst + a;    // update result
         b := b - 1;                  // decrement counter
         goto opnieuw;
wacht:   goto wacht;                  // ready, so stop
```

Listing 13.4: *Pseudo-code voor vermenigvuldiging van 11 met 13 (versie 4).*



Figuur 13.11: *Optelling van een 8-bits register bij een 16-bits registerpaar.*

We moeten de figuur van rechts naar links lezen. Eerst worden *R*2 en *R*0 opgeteld. Bij de optelling wordt de carry meegenomen. Daarom moet de carry flag eerst op 0 gezet worden. Uit de optelling volgt een resultaat dat in *R*2 geplaatst moet worden. De uitgaande carry wordt in de carry flag opgeslagen. Daarna wordt bij register *R*3 het getal 0 en de carry opgeteld, we moeten de optelling immers geschikt maken voor een 16-bits optelling. De carry uit deze optelling wordt in de carry flag opgeslagen. Dit is te zien in listing 13.5. Tevens hebben we een statement toegevoegd dat de inhoud van register *R*0 op de uitgangen zet (merk op dat de processor maar één 8-bits uitgang heeft).

```
R0 := 13;                             // initialize registers
R1 := 11;
R2 := 0;                              // registers R2 and R3
R3 := 0;                              // hold 16-bit result
opnieuw: zero := (R1 - 0) = 0;        // set/reset Z flag
         if zero = 1 goto klaar;      // test Z flag
         carry := 0;                  // clear the carry flag
         R2 := R2 + R0 + carry;       // low byte
         R3 := R3 + 0 + carry;        // high byte
         R1 := R1 - 1 - carry;        // decrement counter
         goto opnieuw;                // and do it again
klaar:   output := R0                 // output low byte
wacht:   goto wacht;                  // ready, so stop
```

Listing 13.5: *Pseudo-code voor vermenigvuldiging van 11 met 13 (versie 5).*

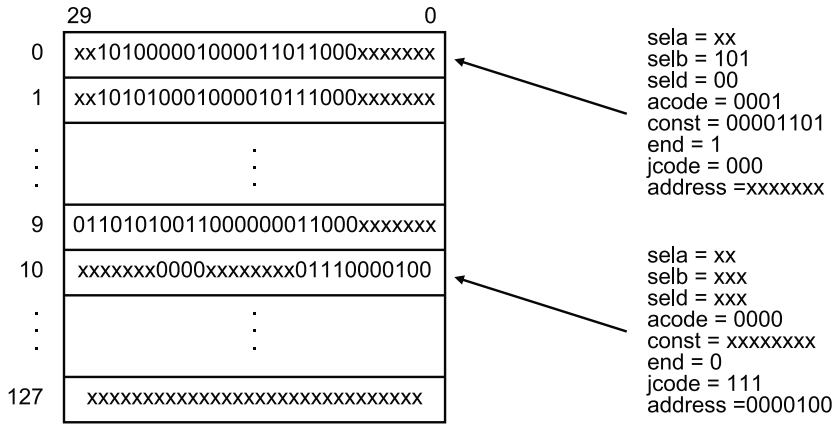
We hebben de pseudo-code nu tot op het laagste niveau uitgewerkt. We kunnen nu een stappenplan (of instructieplan) opstellen voor gebruik in de processor. Het stappenplan is

te zien in tabel 13.3. Dit stappenplan vertaalt de statements uit listing 13.5 in instructies voor de processor. Tevens hebben we de labels vervangen door sprongadressen. We kunnen die nu immers bepalen.

Tabel 13.3: *Stappenplan voor het programma.*

Stap	Instructie
0:	laad register 0 met de constante 13
1:	laad register 1 met de constante 11 (teller)
2:	laad register 2 met de constante 0 (lage byte uitkomst)
3:	laad register 3 met de constante 0 (hoge byte uitkomst)
4:	is register 1 al 0 (zero flag = 1)?
5:	als ja, dan spring naar stap 11 (lus klaar)
6:	zet de carry flag op 0
7:	verwerk lage byte (R2 + R0 + carry)
8:	verwerk hoge byte (R3 + 0 + carry)
9:	verlaag de teller (register 1) met 1
10:	spring naar stap 4
11:	plaats de uitkomst op de uitgang (alleen lage byte)
12:	spring naar stap 12 (wacht voor altijd)

De stappen uit het stappenplan moeten in bitpatronen worden omgezet waarmee het datapad van de processor te besturen is. Deze bitpatronen worden in de ROM geplaatst. In figuur 13.12 is een voorstelling van de ROM te zien met enkele ingevulde instructies. Er wordt ook gebruik gemaakt van don't cares. In werkelijkheid moeten deze don't cares in nullen of enen omgezet worden. De bitpatronen worden in de literatuur *machinecode* genoemd. Merk op dat we kunnen spreken over hardware; de bitpatronen in de ROM vormen immers onderdeel van de processor.



Figuur 13.12: *Invulling van de ROM-tabel (met enkele instructies).*

Het omzetten van de stappen in bitpatronen is een tijdrovende en foutgevoelige klus. Vandaar dat we beter gebruik kunnen maken van een *assembler*. Dit is een programma dat *assembly-taal* omzet in machinecode. In de assembly-taal worden instructies voorgesteld door een begrijpelijke code, *mnemonic* genoemd, en *operands* waarop de mnemonic

werkt. In tabel 13.4 zijn enkele mnemonics en operands te zien. Zo zijn er mnemonics voor het laden van registers, het vergelijken van registers, optellen, aftrekken en spronginstructies. Als operand kunnen dienen: een constante waarde, een register en een sprongadres. Dit worden *addressing modes* genoemd.

Tabel 13.4: *Enkele mnemonics en operands*

Mnemonic	Betekenis
`ld`	Load register with register or constant
`cmp`	Compare register with register or constant (subtract)
`je`	Jump to address if equal (zero flag = 1)
`add`	Add register/constant to register
`sub`	Subtract register/constant from register
`jmp`	Jump to address (always, unconditionally)

Operand	Betekenis
Rx	Register *x* (`ld`, `add`, `sub`, `cmp`)
#yyy	Constant value *yyy* (`ld`, `add`, `sub`, `cmp`)
@aaa	Address *aaa* (`je`, `jmp`)
F	Flags (`ld`)

Middels de tabellen 13.3 en 13.4 kunnen we nu een assembler-programma opstellen. Dit is te zien in listing 13.6. Over de richting van de data moeten we nog een opmerking maken. Zo is bij de instructie **`ld`** `R0,#13` te zien dat register *R*0 geladen wordt met de constante 13. Het transport is (in de assembly-code) van rechts naar links, net als in de toekenning bij de pseudo-code. Met de instructie **`add`** `R2,R2,R0` wordt de inhoud van register *R*2 opgeteld bij de inhoud van register *R*0 en wordt het resultaat in het register *R*2 geplaatst.

```
 0:   ld   R0,#13      ; load multiplicand (constant 13)
 1:   ld   R1,#11      ; load multiplier (constant 11)
 2:   ld   R2,#0       ; clear result registers
 3:   ld   R3,#0       ; or use ld R3,R2
 4:   cmp  R1,#0       ; is multiplier already 0 ?
 5:   je   @11         ; yes it is, so jump!
 6:   ld   F,#0        ; clear the flags (clears carry)
 7:   add  R2,R2,R0    ; add multiplicant (low byte)
 8:   add  R3,R3,#0    ; process carry in high byte
 9:   sub  R1,R1,#1    ; multiplier one off
10:   jmp  @4          ; and again
11:   ld   R0,R2       ; result to output*
12:   jmp  @12         ; hold your horses!
```

Listing 13.6: *Pseudo-code voor vermenigvuldiging van 11 met 13 (assembler).*

13.7 De processor in VHDL

We zullen nu de processor in VHDL beschrijven. Gebruikelijk is om het blokschema in figuur 13.9 te verdelen over een aantal entity's en deze binnen een hiërarchie met

elkaar te verbinden. Dat doen we hier niet. We verdelen het blokschema over een aantal processen die onderling informatie uitwisselen. Daarmee wordt de code beperkt en dat komt de leesbaarheid ten goede. Het eerste deel van de code is te zien in listing 13.7.

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

package microprocessor_common is
    -- The data size and types
    subtype data_type is unsigned (7 downto 0);
    subtype data_type1 is unsigned (8 downto 0);
    constant data_zeros : data_type := (others => '0');

    -- ALU operations (ops) is a 4-bit vector. It instructs
    -- the ALU to perform the desired operation.
    subtype alu_ops_type is std_logic_vector (3 downto 0);
    constant alu_nop   : alu_ops_type := "0000";
    constant alu_passb : alu_ops_type := "0001";
    constant alu_add   : alu_ops_type := "0010";
    constant alu_sub   : alu_ops_type := "0011";
    constant alu_and   : alu_ops_type := "0100";
    constant alu_or    : alu_ops_type := "0101";
    constant alu_exor  : alu_ops_type := "0110";
    constant alu_notb  : alu_ops_type := "0111";
    constant alu_shlb  : alu_ops_type := "1000";
    constant alu_shrb  : alu_ops_type := "1001";
    constant alu_rolb  : alu_ops_type := "1010";
    constant alu_rorb  : alu_ops_type := "1011";
    constant alu_wrfb  : alu_ops_type := "1100";
    constant alu_rdf   : alu_ops_type := "1101";
    -- 1110 and 1111 not used

    -- JUMP operations (ops) is a 3-bit vector. It instructs
    -- the jump checker to perform the desired operation.
    subtype jump_ops_type is std_logic_vector (2 downto 0);
    constant jump_jn   : jump_ops_type := "000";
    constant jump_jnz  : jump_ops_type := "001";
    constant jump_jz   : jump_ops_type := "010";
    constant jump_jnc  : jump_ops_type := "011";
    constant jump_jc   : jump_ops_type := "100";
    constant jump_jncz : jump_ops_type := "101";
    constant jump_jcz  : jump_ops_type := "110";
    constant jump_j    : jump_ops_type := "111";
    -- alternative set
    -- jump_jn is same
    constant jump_jne  : jump_ops_type := "001";
    constant jump_je   : jump_ops_type := "010";
    constant jump_jge  : jump_ops_type := "011";
    constant jump_jl   : jump_ops_type := "100";
    constant jump_jg   : jump_ops_type := "101";
    constant jump_jle  : jump_ops_type := "110";
    constant jump_jmp  : jump_ops_type := "111";
```

Listing 13.7: *Processor: constanten en subtypes (deel 1).*

We beginnen de code met het definiëren van enkele types en constanten. We plaatsen deze types en constanten in een *package*. Zo definiëren we het type `data_type` dat gebruikt wordt voor de data in het datapad. Dit is een unsigned vector van 8 bits. De ALU heeft als resultaat een 9-bits uitkomst, daar definiëren we een apart type voor. Er wordt ook een constante `data_zeros` gedefinieerd.

Voor de operaties van de ALU definiëren we een aantal constanten alsmede voor de operaties van de jump checker. Dat komt de leesbaarheid ten goede. Voor de ALU wordt een 4-bits code gedefinieerd, voor de jump checker een 3-bits code.

In listing 13.8 zijn de definities omtrent de addressen en ROM te zien. Er worden een 7-bits adres en een 30-bits instructie gedefinieerd. Een belangrijk onderdeel is de definities van de inhoud van de ROM. Hierin zijn immers de instructies opgeslagen, of beter gezegd: de ROM herbergt het programma. Instructiebits die niet gebruikt worden, zijn ingevuld als don't care. Hierdoor kan bij synthese de combinatorische logica van de ROM tot een minimum beperkt worden. Merk op dat alle niet-gebruikte instructies volledig als don't

```
    -- Address of the ROM
    subtype  address_type is unsigned (6 downto 0);
    constant address_zeros : address_type := (others => '0');

    -- Full format of an instruction. An instruction is 30 bits wide.
    subtype  instr_type is std_logic_vector (29 downto 0);

     -- The program is held in an 128-instruction word, 30 bit wide ROM.
    type  rom_table is array (0 to 127) of instr_type;

    -- Program ROM starts here
    constant rom : rom_table := (
    -- The program performs a multiplication of the contents of R0
    -- with R1 and the 16-bit result is placed in R3 (high) and R2 (low).
    -- The lower result byte is copied to R0.
    -- It uses the simple successive add algorithm.
        "--101000001000011011000-------", -- Load R0 with constant 13
        "--101010001000010111000-------", -- Load R1 with constant 11
        "--101100001000000001000-------", -- Load R2 with constant 0
        "--101110001000000001000-------", -- Load R3 with constant 0
        "01101--0011000000000000-------", -- Compare R1 with constant 0
        "-------000000000000000100001011", -- Jump if equal (R1 == 0) to
                                           -- address 11
        "00101001100------000000-------", -- Load flags with zeros
        "00010100010--------1000-------", -- Add: R2 := R2 + R0 + carry
        "11101110010000000001000-------", -- Add: R3 := R3 + 0 + carry
        "01101010011000000011000-------", -- Sub: R1 := R1 - 1
        "-------0000--------01110000100", -- Jump to address 4
        "--010000001--------1000-------", -- Load R0 with contents of R2
        "-------0000--------01110001100", -- Jump to address 12 (itself)
        others => "------------------------------"  -- Rest is don't care
    );

end package microprocessor_common;
```

Listing 13.8: *Processor: constanten en subtypes (deel 2).*

care zijn ingevuld. De ROM herbergt het programma uit listing 13.6.

In listing 13.9 is de entity-declaratie van de processor te zien. De port list omvat een klok, een asynchrone reset, een 8-bits ingang en een 8-bits uitgang. Via de ingangen en uitgangen communiceert de processor met de buitenwereld. Merk op dat de definities uit de package ook worden geladen.

```
1  -- The microprocessor has a clock, a asynchronous reset,
2  -- an 8-bit input and an 8-bit output (otherwise the synthesizer has
3  -- nothing to do...).
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7  use work.microprocessor_common.all;
8
9  entity microprocessor is
10     port (clk    : in std_logic;
11           areset : in std_logic;
12           input : in data_type;
13           output: out data_type
14          );
15 end entity microprocessor;
```

Listing 13.9: *Processor: entity.*

De architecture wordt begonnen met het declareren van een groot aantal (interne) signalen, te zien in listing 13.10. De signalen `zflag` en `cflag` bevatten de (huidige) waarde van de zero flag en de carry flag. De signalen `zupdate` en `cupdate` zijn de door de ALU berekende nieuwe waarden van de zero flag en carry flag die na een klokflank in de desbetreffende flag worden geladen.

De overige interne signalen spreken min of meer voor zichzelf. Het signaal `muxaout` bevat de waarde van de uitgangen van de multiplexer op de *A*-ingang van de ALU, `muxbout` bevat de waarde van de uitgangen van de multiplexer op de *B*-ingang van de ALU. Signaal `alu_result` bevat de waarde van het ALU-resultaat. Dit is een 8-bits waarde. (Zoals we later zullen zien, heeft de ALU intern een 9-bits waarde.) De signalen `acode` en `jcode` bevatten de functie van de ALU en de code voor de jump checker. Signaal `ldpc` geeft aan of de Program Counter moet worden geladen.

De signalen `reg0` t/m `reg3` stellen de vier registers *R*0 t/m *R*3 voor. Dit zijn 8-bits waarden. De signalen `enr0` t/m `enr3` geven aan of een bepaald register geladen moet worden. De signalen `sela`, `selb` en `seld` geven aan welk register moet worden geselecteerd voor respectievelijk de *A*-ingang en *B*-ingang van de ALU en het register waarin het ALU-resultaat moet worden teruggeschreven (of ingeklokt).

Het signaal `en_d` (enable *D*estination) geeft aan dat het resultaat uit de ALU naar een van de registers moet worden teruggeschreven. Het signaal `const` (*const*ant) bevat de constante uit de ROM. We hebben hier voor een afwijkende naamgeving gekozen omdat **`end`** en **`constant`** keywords zijn in VHDL.

De register file is beschreven in listing 13.11. De registers bestaan uit flipflops zodat het laden op een klokflank gebeurt. De registers worden allemaal gereset als de asynchrone

```
architecture rtl of microprocessor is
-- All internal signals
signal zflag, cflag : std_logic;                -- zero and carry flags
signal zupdate, cupdate : std_logic;            -- update value flags
signal muxaout, muxbout : data_type;            -- mux outputs
signal alu_result : data_type;                  -- ALU result
signal acode : alu_ops_type;                    -- ALU code
signal jcode : jump_ops_type;                   -- jump code
signal ldpc : std_logic;                        -- load PC
signal reg0, reg1, reg2, reg3 : data_type;      -- registers
signal enr0, enr1, enr2, enr3 : std_logic;      -- enable loading regs
signal sela : std_logic_vector (1 downto 0);    -- mux A data selection
signal selb : std_logic_vector (2 downto 0);    -- mux B data selection
signal seld : std_logic_vector (1 downto 0);    -- decoder selection
signal en_d : std_logic;                        -- enable reg writeback
signal const : data_type;                       -- constant in ROM
signal address : address_type;                  -- address in ROM
signal pc : address_type;                       -- Program Counter
```

Listing 13.10: *Processor: interne signalen.*

reset actief is. De signalen `enr0` t/m `enr3` zorgen ervoor dat een bepaald register geladen wordt als het signaal actief is.

In listing 13.12 is de code van de twee multiplexers en de decoder te zien. De code is rechttoe rechtaan. We maken nog een opmerking over de niet-gebruikte codecombinaties

```
    -- The Register File contains four registers and loads a
    -- specified register if the enable is active.
    regfile : process (clk, areset) is
    begin
        if areset = '1' then
            reg0 <= data_zeros;
            reg1 <= data_zeros;
            reg2 <= data_zeros;
            reg3 <= data_zeros;
        elsif rising_edge(clk) then
            if enr0 = '1' then
                reg0 <= alu_result;
            end if;
            if enr1 = '1' then
                reg1 <= alu_result;
            end if;
            if enr2 = '1' then
                reg2 <= alu_result;
            end if;
            if enr3 = '1' then
                reg3 <= alu_result;
            end if;
        end if;
    end process regfile;
```

Listing 13.11: *Processor: register file.*

bij de multiplexer voor de *B*-ingang van de ALU; bij selectie van de codecombinaties worden de uitgangen op don't care gezet.

```
    -- Selection of the A input of the ALU
    muxa : process (sela, reg0, reg1, reg2, reg3) is
    begin
        case sela is
            when "00" => muxaout <= reg0;
            when "01" => muxaout <= reg1;
            when "10" => muxaout <= reg2;
            when "11" => muxaout <= reg3;
            when others => null;
        end case;
    end process muxa;

    -- Selection of the B input of the ALU
    muxb : process (selb, reg0, reg1, reg2, reg3, input, const) is
    begin
        case selb is
            when "000" => muxbout <= reg0;
            when "001" => muxbout <= reg1;
            when "010" => muxbout <= reg2;
            when "011" => muxbout <= reg3;
            when "100" => muxbout <= input;
            when "101" => muxbout <= const;
            when others => muxbout <= (others => '-');
        end case;
    end process muxb;

    -- Selection of the register to write back the result.
    decoderd : process (seld, en_d) is
    begin
        enr0 <= '0';
        enr1 <= '0';
        enr2 <= '0';
        enr3 <= '0';
        if en_d = '1' then
            case seld is
                when "00" => enr0 <= '1';
                when "01" => enr1 <= '1';
                when "10" => enr2 <= '1';
                when "11" => enr3 <= '1';
                when others => null;
            end case;
        end if;
    end process decoderd;
```

Listing 13.12: *Processor: multiplexers en decoder.*

Listing 13.13 laat het eerste deel van de ALU-code zien. De code is gevormd rond een **case**-statement waarbij bij elke tak een ALU-operatie wordt beschreven. In het begin wordt het ALU-resultaat op 0 gezet en worden de huidige waarden van de flags geladen. Merk op dat sommige ALU-operaties de flags niet aanpassen. Te denken valt aan de `nop`-operatie. De operatie `pass b` past wel de zero-flag aan maar niet de carry flag; de

carry flag is namelijk niet van toepassing bij deze operatie. Optellen wordt gerealiseerd met een veel gebruikte oplossing in VHDL. De op te tellen getallen wordt eerst naar 9 bits omgezet en vervolgens met de ingaande carry opgeteld. De ingaande carry wordt ook uitgebreid tot 9 bits. Het hoogste bit van het resultaat wordt aan de carry flag toegekend.

Verder is te zien dat de aftrekking wordt gerealiseerd middels een two's complement optelling alhoewel de data uit *unsigned* getallen bestaat. Toch kan de two's complement optelling gebruikt worden.

```
1   -- The ALU. This is a pure combinational circuit that can
2   -- add, subtract, do logic, shifts and rotates and more.
3   -- Some ALU ops use the flags, some don't. Note that the
4   -- behavior of the flags depends on the ALU op.
5   alu : process (acode, muxaout, muxbout, zflag, cflag) is
6   variable alu_result_int : data_type1;
7   variable z_int, c_int : std_logic;
8   begin
9       -- Set result to 0 and get the current flag data
10      alu_result_int := (others => '0');
11      z_int := zflag;
12      c_int := cflag;
13      -- Do the operation
14      case acode is
15          when alu_nop =>
16              alu_result_int := '0' & muxbout;
17          when alu_passb =>
18              alu_result_int := '0' & muxbout;
19          when alu_add =>
20              alu_result_int := ('0' & muxaout) + ('0' & muxbout)
21                                              + ("" & cflag);
22              c_int := alu_result_int(8);
23          when alu_sub =>
24              -- Subtraction can be performed using two's complement
25              -- even if the numbers are all unsigned.
26              -- Two's complement subtraction: A, B: 8 bit unsigned,
27              -- Cin : 1 bit Carry In
28              --     A - B - Cin  ==  A + (not(B) + 1) - Cin
29              --                  ==  A + not(B) + (1-Cin)
30              --                  ==  A + not(B) + not(Cin)
31              --     Cout = not(Carry out of the adder hardware)
32              alu_result_int := ('0' & muxaout) + ('0' & not muxbout)
33                                        + ("" & not cflag);
34              c_int := not alu_result_int(8);
35          when alu_and =>
36              alu_result_int := '0' & (muxaout and muxbout);
37          when alu_or =>
38              alu_result_int := '0' & (muxaout or muxbout);
39          when alu_exor =>
40              alu_result_int := '0' & (muxaout xor muxbout);
41          when alu_notb =>
42              alu_result_int := '0' & (not muxbout);
```

Listing 13.13: *Processor: ALU (deel 1).*

Het gebruik van de two's complement optelling is toegestaan omdat geldt:

$$\begin{aligned} A-B-c_{in} &= A-B-c_{in}+2^8 \\ &= A+(2^8-B)-c_{in} \\ &= A+(\text{two's complement of } B)-c_{in} \\ &= A+\overline{B}+1-c_{in} \\ &= A+\overline{B}+(1-c_{in}) \\ &= A+\overline{B}+\overline{c_{in}} \end{aligned} \tag{13.1}$$

De inkomende en uitgaande carry's moet wel geïnverteerd worden omdat de borrow (leen) de inverse is van de carry. In listing 13.14 is het tweede deel van de ALU te zien.

```
            when alu_shlb =>
                alu_result_int := muxbout & '0';
                c_int := alu_result_int(8);
            when alu_shrb =>
                -- we need a little trick here...
                alu_result_int := "00" & muxbout(7 downto 1);
                c_int := muxbout(0);
            when alu_rolb =>
                alu_result_int := muxbout & cflag;
                c_int := alu_result_int(8);
            when alu_rorb =>
                -- we need a little trick here...
                alu_result_int := '0' & cflag & muxbout(7 downto 1);
                c_int := muxbout(0);
            when alu_wrfb =>
                alu_result_int := ('0' & muxbout);
                c_int := alu_result_int(0);
                z_int := alu_result_int(1);
            when alu_rdf =>
                alu_result_int := (0=>cflag, 1=>zflag, others=>'0');
            when others =>
                alu_result_int := ('0' & muxbout);
        end case;
        -- The read/write flags and nop ALU ops do not touch the Z flag.
        if acode /= alu_rdf and acode /= alu_wrfb and acode /= alu_nop
           then
            if alu_result_int(7 downto 0) = data_zeros then
                z_int := '1';
            else
                z_int := '0';
            end if;
         end if;
         -- Output the ALU result and the new results of
         -- the Zero and Carry flag.
         alu_result <= alu_result_int(7 downto 0);
         zupdate <= z_int;
         cupdate <= c_int;
    end process alu;
```

Listing 13.14: *Processor: ALU (deel 2).*

Te zien zijn de schuif- en rotatieoperaties. Verder zijn de directe bewerkingen op de flags te zien. In het laatste stuk van de code is te zien hoe de flags worden aangepast. De waarde van de zero flag is afhankelijk van het resultaat van de ALU (die moet namelijk 0 zijn om de zero flag op 1 te zetten). Bij drie operaties wordt de zero flag niet aangepast. Het aanpassen van de carry flag wordt bij de operaties zelf beschreven. Een aantal operaties past de carry flag niet aan. Aan het eind van de code wordt het resultaat via signaal `alu_result` naar buiten gebracht. Ook de nieuwe waarden van de zero flag (`zupdate`) en de carry flag (`cupdate`) worden toegekend.

In listing 13.15 is de code te zien voor het aanpassen van de flags met de nieuwe waarden. Merk op dat de flags op 0 gezet worden bij een asynchrone reset.

```
1   -- The Zero and Carry flags. They are really flip-flops and
2   -- depend on the outcome of the ALU result.
3   flags : process (clk, areset) is
4   begin
5       if areset = '1' then
6           cflag <= '0';
7           zflag <= '0';
8       elsif rising_edge(clk) then
9           cflag <= cupdate;
10          zflag <= zupdate;
11      end if;
12  end process flags;
```

Listing 13.15: *Processor: zero flag en carry flag.*

De Program Counter wordt beschreven in listing 13.16. Onder een asynchrone reset wordt de PC geladen met allemaal nullen zodat wordt begonnen met het uitvoeren van de instructie op ROM-adres 0. Op elke klokflank wordt de PC afhankelijk van de waarde van signaal `ldpc` óf geladen met een nieuwe waarde (sprongadres) óf met één verhoogd.

```
1   -- The Program Counter. It can be loaded which
2   -- effectively jumps to another program location.
3   -- Program starts a address 0 on a reset.
4   regpc : process (clk, areset) is
5   begin
6       if areset = '1' then
7           pc <= address_zeros;
8       elsif rising_edge(clk) then
9           if ldpc = '1' then
10              pc <= address;
11          else
12              pc <= pc + 1;
13          end if;
14      end if;
15  end process regpc;
```

Listing 13.16: *Processor: Program Counter.*

In listing 13.17 is de code van de jump checker te zien. Onder besturing van signaal `jcode` wordt één van de acht toekenningen uitgevoerd zoals is aangegeven in tabel 13.2 en figuur 13.9.

```
    -- The Conditional Jump Checker. This process determines if
    -- a certain condition is met and sets the PC Load signal
    -- accordingly.
    jumpcheck : process (jcode, zflag, cflag) is
    begin
        case jcode is
            when jump_jn =>   ldpc <= '0';
            when jump_jnz =>  ldpc <= not zflag;
            when jump_jz =>   ldpc <= zflag;
            when jump_jnc =>  ldpc <= not cflag;
            when jump_jc =>   ldpc <= cflag;
            when jump_jncz => ldpc <= not (cflag or zflag);
            when jump_jcz =>  ldpc <= cflag or zflag;
            when jump_j =>    ldpc <= '1';
            when others =>    null;
        end case;
    end process jumpcheck;
```

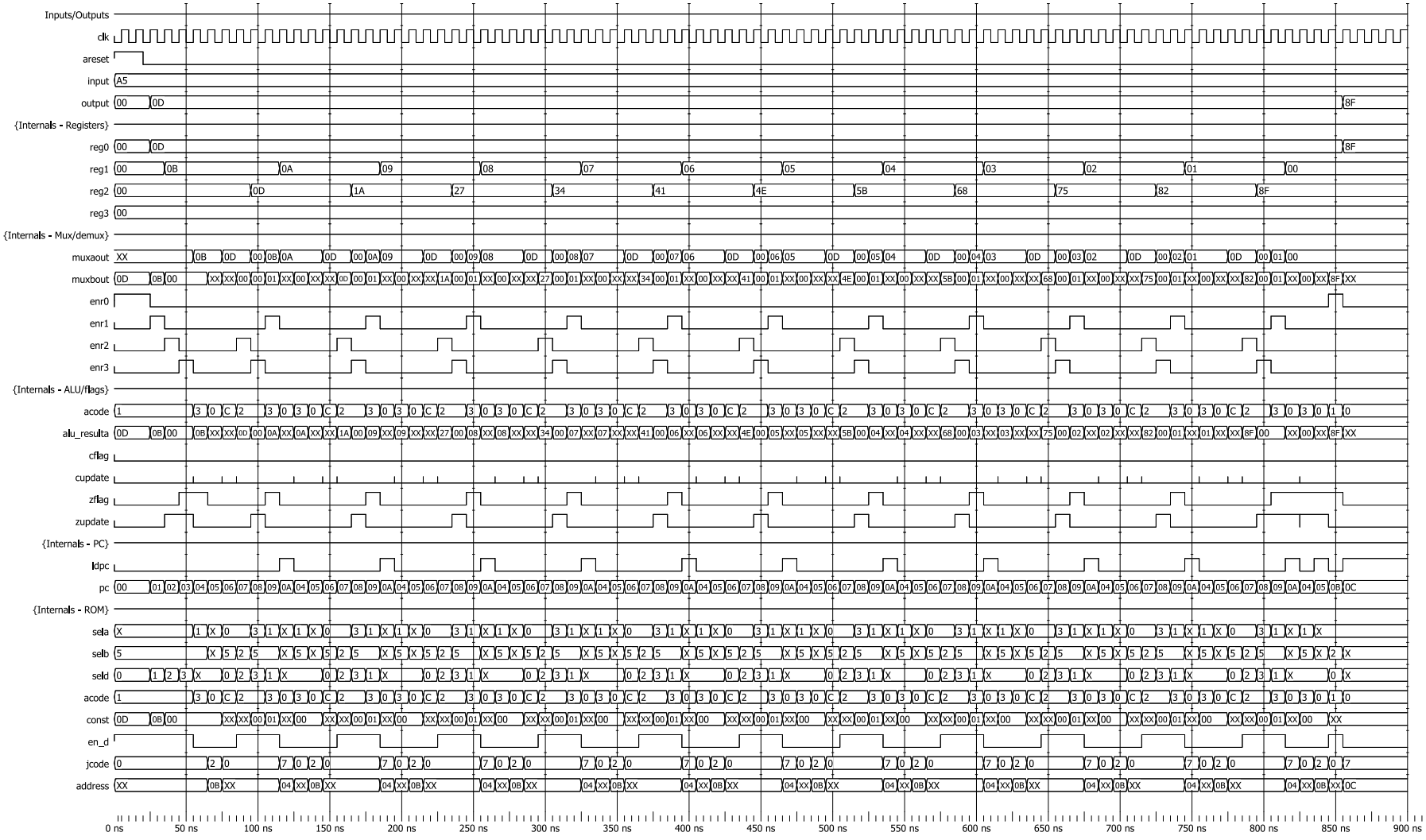
Listing 13.17: *Processor: jump checker.*

Het toekennen van de instructiebits aan de stuursignalen is te zien in listing 13.18. We noemen dit ook wel instructiedecodering. In feite is het niets anders dan een reeks simpele toekenningen, maar omdat er verschillende datatypen worden gebruikt moeten diverse conversieroutines worden aangeroepen. Verder is te zien dat de inhoud van register *R*0 aan de uitgangen wordt toegekend. Als we dat niet zouden doen, dan zal bij synthese alle hardware worden verwijderd want, zo is de gedachte, de processor voert "niets" uit.

```
    -- Instruction ROM decoding is nothing more then rewiring
    -- the instruction bits to the control inputs. Only VHDL
    -- makes such a trouble to do this the simple way...
    instr_rom : process (pc) is
    begin
        sela    <= rom(to_integer(pc))(29 downto 28);
        selb    <= rom(to_integer(pc))(27 downto 25);
        seld    <= rom(to_integer(pc))(24 downto 23);
        acode   <= rom(to_integer(pc))(22 downto 19);
        const   <= unsigned(rom(to_integer(pc))(18 downto 11));
        en_d    <= rom(to_integer(pc))(10);
        jcode   <= rom(to_integer(pc))(9 downto 7);
        address <= unsigned(rom(to_integer(pc))(6 downto 0));
    end process instr_rom;

    -- The output is only here so that the synthesizer has
    -- something to synthesize.
    output <= reg0;

end architecture rtl;
```

Listing 13.18: *Processor: instructiedecodering en uitgang.*

In figuur 13.13 is het simulatieresultaat te zien van de vermenigvuldiging van 11 met 13. Merk op dat deze simulatie kan afwijken van een fysieke implementatie omdat we bij



Figuur 13.13: *Simulatie van de vermenigvuldiging van 11 met 13.*

simulatie gebruik maken van don't cares. Bij een praktische uitvoering worden deze don't cares omgezet in nullen en enen.

Bovenaan de figuur zijn de ingangen en de uitgangen van de processor te zien: de klok, de asynchrone reset, de ingangen en de uitgangen. Daarna zijn de inhouden (of waarden) van de registers te zien, gevolgd door de uitgangen van de multiplexers en de decoder. Vervolgens worden de signalen die met de dataverwerking van de ALU te maken hebben afgebeeld. Daarna worden de signalen getoond die te maken hebben met de adresverwerking van de Program Counter.

Als laatste is de decodering van de diverse stuur- en datasignalen te zien. Hiermee wordt het datapad ingesteld. Merk op dat bij sommige signalen een 'X' of 'XX' staat. Hier gaat hierbij niet om *forcing unknown* signalen maar om don't cares. De simulator beeldt die echter als 'X'-en af.

Vanuit de simulatie kunnen we een uitspraak doen over de snelheid waarmee de vermenigvuldiging wordt uitgevoerd. Elke doorloop van de lus wordt in zeven klokpulsen uitgevoerd, behalve als register $R1$ gelijk aan 0 is: dan wordt na twee klokpulsen al gesprongen naar het einde van het programma. In totaal duurt het uitvoeren van de lus dus 79 klokpulsen: 11 keer 7 klokpulsen voor een doorloop van de lus als $R1$ groter is dan 0 en 2 klokpulsen voor het uitvoeren van instructies als $R1$ gelijk is aan 0. In formulevorm:

$$\text{aantal klokpulsen} = 7 \times R1 + 2 \tag{13.2}$$

De maximale executietijd van de vermenigvuldiging bij een waarde $R1 = 255$ is dus 1787 klokpulsen.

13.8 Voorbeeldprogramma: looplicht

Een eerste opzet van de code is te zien in listing 13.19. Eerst wordt een 1 geladen in het meest significante bit van register $R0$. De overige bits zijn alle 0. Daarna wordt register $R0$ één plaats naar rechts geschoven waarbij de minst significante bit in de carry flag wordt geplaatst. Zolang de carry 0 is wordt steeds de schuifoperatie uitgevoerd. Als de carry 1 is, betekent dit dat de 1 uit register $R0$ eruit geschoven is. Dan wordt weer naar het begin van het programma gesprongen.

```
1 0:     ld   R0,128     ; load start value
2 1:     shr  R0,R0      ; shift right into carry
3 2:     jnc  @1         ; jump if carry = 0
4 3:     jmp  @0         ; and do it again
```

Listing 13.19: *Pseudo-code voor looplicht (assembler).*

We kunnen twee opmerkingen maken over de eerste opzet. Het is nu alleen mogelijk om één 1 op de uitgangen te zetten omdat register $R0$ steeds met 128 (80_{16}) geladen wordt. Verder kunnen we aannemen dat de klokfrequentie van de processor in de megahertzen loopt. Het schuiven van de uitgangen gaat dan zo snel dat we het niet goed kunnen zien. We moeten de code aanpassen zodat de twee problemen opgelost worden.

In listing 13.20 is een aanpassing van het programma te zien. Als eerste wordt de waarde van de ingangen in register *R*0 geladen. Daarna wordt naar rechts geschoven. Bij deze operatie wordt een 0 in de meest significante bit van *R*0 geschoven. De minst significante bit wordt in de carry geschoven. Vervolgens testen we de carry flag. Als de carry flag 0 is wordt weer gesprongen naar de schuifoperatie. Zo niet, dat wordt de volgende instructie uitgevoerd. Deze instructie betreft een logische OR van register *R*0 met het getal 128 (80_{16}). Hiermee wordt de meest significante bit van *R*0 op 1 gezet.

```
0:      ld   R0,IN       ; load start value from inputs
1:      shr  R0,R0       ; shift right into carry
2:      jnc  @1          ; jump to shr if carry = 0
3:      or   R0,R0,#128  ; carry is 1 so set high bit R0
4:      jmp  @1          ; and do it again
```

Listing 13.20: *Pseudo-code voor looplicht (assembler).*

We hebben nu een programma dat een willekeurig op de ingangen aangeboden patroon rechtsom roteert. De waarde daarvan is op de uitgangen te zien. Maar zoals gezegd, gebeurt het roteren op hoge snelheid. Het patroon is dan niet goed te zien. Wat we moeten doen is duidelijk: na elke rotatie even wachten. Dit kunnen we oplossen door het inbouwen van een *wachtlus* (Engels: delay loop).

Een wachtlus is niets anders dan instructies die klokpulsen "wegtikken". Stel dat we 0,25 seconden willen wachten. Dan moeten we 0,25 keer het aantal klokpulsen per seconden wegtikken (we spreken ook wel van verstoken). Het wegtikken van de klokpulsen kunnen we doen door een register te laden met een beginwaarde en af te tellen naar 0.

We gaan voor het beschrijven van de wachtlus uit van een klokfrequentie van 50 MHz. Dit is het geval op het DE0-bordje van Altera. Voor de wachtlus in het programma moeten we dus $50.000.000 \cdot 0{,}25 = 12.500.000$ klokpulsen wachten. Dat is met één register niet te realiseren. Daarom gebruiken we *drie* registers. We moeten de drie registers zien als een 24-bits waarde die geladen wordt met een beginwaarde en waar steeds één van afgetrokken wordt. Een opzet van de wachtlus is te zien in listing 13.21. We hebben hier geen instructie-adressen gegeven en gebruiken een *label* voor het springen.

```
        ld   R1,#<value1>   ; load value 1
        ld   R2,#<value2>   ; load value 2
        ld   R3,#<value3>   ; load value 3
        ld   F,#0           ; clear flags
delay:  sub  R1,R1,#1       ; subtract 1 from low byte
        sub  R2,R2,#0       ; subtract carry from middle byte
        sub  R3,R3,#0       ; subtract carry from high byte
        jnc  delay          ; jump if no carry
```

Listing 13.21: *Wachtlus op basis van testen van de carry flag.*

De registers *R*3, *R*2 en *R*1 vormen samen het 24-bits getal waarbij *R*3 als meest significante byte dient en *R*1 als minst significante byte dient. Eerst worden de registers geladen (met een nog te berekenen) beginwaarde. Het aftrekken wordt gerealiseerd door de drie **sub**-instructies en de **jnc**-instructie. Dit werkt als volgt. Eerst wordt van de minst significante

byte één afgetrokken. De waarde van de carry volgt uit de inhoud van register $R1$ na aftrekking. Als $R1$ op 0 stond, wordt de inhoud 255 en wordt de carry flag op 1 gezet. In alle andere gevallen wordt het register met één verlaagd en wordt de carry flag op 0 gezet. Een overgang van 0 naar 255 betekent dat het register een *roll over* heeft ondergaan en dat is een teken dat register $R2$ met één verlaagd moet worden. Dat wordt automatisch geregeld door de tweede **sub**-instructie die register $R2$ met één verlaagd als de carry flag 1 is. Hetzelfde principe geldt voor een roll over van register $R2$. Dan wordt register $R3$ automatisch met één verlaagd. Zolang $R3$ geen roll over ondergaat zal de carry flag 0 zijn. Er wordt dan gesprongen naar het begin van de lus. Uiteindelijk zijn alle inhouden van de registers 0. Een aftrekking van één zorgt ervoor dat alle register de inhoud 255 krijgen en wordt de carry flag op 1 gezet. De wachttijd is verstreken en de sprong wordt niet genomen. Merk op dat de lus één keer meer wordt uitgevoerd vanwege de roll overs van de registers.

We kunnen de beginwaarde van de teller berekenen middels de formule:

$$\text{aantal keer lus doorlopen} = \frac{f_{proc} \cdot \text{wachttijd}}{\text{klokpulsen per lusdoorloop}} \tag{13.3}$$

De klokfrequentie is 50 MHz, de wachttijd bedraagt 0,25 seconden en een lusdoorgang duurt vier klokpulsen. In dat geval moet de teller geladen worden met 3.125.000, verdeeld over drie registers. We moeten van dit getal één aftrekken vanwege de roll overs van alle registers. Verder duurt het laden van de registers en het op 0 zetten van de flags precies vier klokpulsen, evenveel als één lusdoorgang. De te laden waarde is dus 3.124.998. In hexadecimale notatie is dit $2FAF06_{16}$. Hierin is goed te zien met welke waarden de registers geladen moeten worden. Register $R3$ moet geladen worden met 47 ($2F_{16}$), register $R2$ moet geladen worden met 175 (AF_{16}) en register $R1$ moet geladen worden met 6 (06_{16}).

Het volledige programma is te zien in listing 13.22. De sprongadressen zijn nu ingevuld. Merk op dat de ingangen slechts één keer worden ingelezen, namelijk aan het begin.

```
0:    ld   R0,IN          ; load start value from inputs
1:    shr  R0,R0          ; shift right into carry
2:    jnc  @4             ; jump to delay if carry = 0
3:    or   R0,R0,#128     ; carry is 1 set high bit R0
4:    ld   R1,#6          ; load value 06 hex
5:    ld   R2,#175        ; load value af hex
6:    ld   R3,#47         ; load value 2f hex
7:    ld   F,#0           ; clear flags
8:    sub  R1,R1,#1       ; subtract 1 from low byte
9:    sub  R2,R2,#0       ; subtract carry from middle byte
10:   sub  R3,R3,#0       ; subtract carry from high byte
11:   jnc  @8             ; jump if no carry
12:   jmp  @1             ; and again
```

Listing 13.22: *Roteren van bitpatroon met wachtlus.*

Om dit programma door de processor te laten uitvoeren moet de programma-ROM worden aangepast. De nieuwe inhoud van de ROM is te zien in listing 13.23. Deze code moet de bestaande code vervangen.

```
    -- Program ROM starts here
    constant rom : rom_table := (
        "--100000001--------1000-------",  -- Load input in R0
        "--000001001--------1000-------",  -- Shift right R0
        "-------0000--------00110000100",  -- Jump if C=0 skip next
        "00101000101100000001000-------",  -- OR R0 with 128
        "--10101000100000110100-------",  -- Load R1 with 6
        "--10110000110101111000-------",  -- Load R2 with 175
        "--10111000100101111000-------",  -- Load R3 with 47
        "--101--1100000000000000-------",  -- Clear the flags
        "01101010011000000011000-------",  -- Subtract 1 from R1
        "10101100011000000001000-------",  -- Subtract carry from R2
        "11101110011000000001000-------",  -- Subtract carry from R3
        "-------0000--------00110001000",  -- Jump if no carry out
        "-------0000--------01110000001",  -- Jump to start of rotate
        others => "------------------------------"  -- Rest is don't care
    );
```

Listing 13.23: *ROM-code voor het looplicht.*

13.9 Mogelijke uitbreidingen van de processor

In deze paragraaf bespreken we enkele mogelijke uitbreidingen op het bestaande ontwerp van de processor.

Een eerste uitbreiding is onmiskenbaar het geschikt maken van de processor voor het berekenen van two's complement getallen. De flags moeten dan worden uitgebreid met een overflow flag (V) en een negative flag (N). Het inbouwen van de extra flags en het aanpassen van de flags in de ALU is geen lastige klus. We moeten echter ook het aantal conditionele spronginstructies uitbreiden om gebruik te maken van de two's complement getallen. Te denken valt aan de zes relationele operatoren die nu alleen voor unsigned getallen gebruikt kunnen worden. De ROM moet met één bit (voor signaal $jcode$) worden uitgebreid zodat er 16 mogelijke sprongopdrachten kunnen worden gerealiseerd.

De processor heeft slechts vier registers en kan dus een beperkte hoeveelheid data tegelijk verwerken. Het is zinvol om de processor uit te breiden met een RAM-geheugen. Te denken valt aan een RAM van 256 plaatsen van 8 bits per plek. De ROM moet hiervoor uitgebreid worden met een 8-bits RAM-adres en een signaal dat aangeeft of er naar de RAM geschreven moet worden. In totaal wordt de ROM dus uitgebreid met 9 signalen. We kunnen ervoor kiezen om de data op de B-ingang van de ALU naar de RAM te schrijven. Dat betekent dat naast de vier registers ook een constante direct in de RAM geplaatst kan worden. Voor het lezen van een RAM-adres kan één van de niet-gebruikte ingangen van de multiplexer bij de B-ingang van de ALU gebruikt worden, er waren immers nog twee ingangen niet gebruikt. Een interessante mogelijkheid is om in plaats van een "hard" adres een register te gebruiken om een geheugenplaats aan te wijzen, bijvoorbeeld register $R3$. Op deze manier is het mogelijk om *pointers* te realiseren. De ROM hoeft dan niet met 8 adresbits te worden uitgebreid.

Een derde uitbreiding zou een apart register kunnen zijn voor de uitgangen. Nu wordt continu de inhoud van register $R0$ aan de uitgangen doorgegeven. Er is één signaal nodig om aan te geven dat het uitgangsregister geladen moet worden. Als data zou de uitgang

van de multiplexer aan de *B*-ingang van de ALU kunnen dienen.

Een vierde uitbreiding betreft het voorzien van de externe data-ingangen van synchronisatieflipflops. Over het algemeen wordt dubbele synchronisatie toegepast. De uitbreiding kan zonder veel problemen worden ingepast in het bestaande ontwerp. Er zijn geen extra stuursignalen vanuit de ROM nodig. Nadeel is wel dat de data pas na twee klokpulsen echt wordt ingelezen en verwerkt. De data moet immers nog door de synchronisatieflipflops geloodst worden.

Als vijfde uitbreiding zouden we een *stack* (stapel) kunnen implementeren. Een stack is te vergelijken met een stapel A4'tjes. We kunnen op de bovenkant van de stapel steeds één velletje erop leggen of eraf halen. Zo werkt dat ook met een stack. De stack is te implementeren met een stukje RAM en een *stack pointer* die aangeeft welk adres de bovenkant van de stack is. Er zijn wel wat uitbreidingen op het bestaande ontwerp nodig. Ten eerste moet er een stuk RAM worden toegevoegd. Ten tweede moet een register worden toegevoegd die als stack pointer dient. Data moet op de stack kunnen worden gezet, dat zou kunnen vanuit de multiplexer op de *B*-ingang van de ALU. Daarnaast moet ook data van de stack kunnen worden gehaald. Daar zou een ingang op de multiplexer aan de *B*-ingang van de ALU voor kunnen worden gebruikt. Het zou ook interessant zijn om de inhoud van de PC op de stack te kunnen zetten of af te kunnen halen. We kunnen dan *subroutines* implementeren. Nadat de PC op de stack is gezet, kan gesprongen worden naar een stukje code in het programma. Aan het einde van dat stukje code wordt dan de PC weer vanuit de stack gelezen. Het gebruik van subroutines sluit aan bij het implementeren van functies in hogere programmeertalen zoals 'C'.

Een zesde uitbreiding behelst het onderbreken van het lopende programma door een extern *interruptsignaal*. Een interrupt zorgt ervoor dat de processor op een van te voren bepaald adres instructies gaat uitvoeren. Het inbouwen van deze mogelijkheid is niet zo lastig als we gebruik kunnen maken van de stack. De stack voorziet ons immers van het opslaan van de PC. Nadat het interruptsignaal is gedetecteerd, wordt de PC op de stack gezet (eigenlijk moet de PC plus één worden opgeslagen). Daarna wordt de PC geladen met een hard-gecodeerd adres waar het interruptprogramma begint. Als de interruptroutine is afgelopen, wordt de PC weer van de stack gehaald en kan het onderbroken programma weer verder gaan. Er is een keuze tussen een flankgestuurde of niveaugestuurde interrupt. Bij een flankgestuurde interrupt zorgt een signaalwisseling op de interruptingang voor het starten van de interruptroutine, bij een niveaugestuurde interrupt zorgt een signaalniveau (hoog of laag) voor het starten van de interruptroutine. Het is zinvol om interrupts uit te zetten tijdens het uitvoeren van de interruptroutine om *stack flooding* (het vollopen van de stack) te voorkomen.

Een zevende uitbreiding zou het aantal register kunnen zijn. De processor heeft maar vier registers en kan dus maar een beperkt aantal resultaten tegelijk onthouden. We zouden het aantal registers naar acht kunnen uitbreiden. Daarvoor zouden de signalen $sela$, $selb$ en $seld$ met één bit moeten worden uitgebreid.

Een achtste uitbreiding zou het aanpassen van de bitbreedte van het datapad en de Program Counter kunnen zijn. Op dit moment is de bitbreedte van de data 8 bits. Het is vrij eenvoudig om dit op te schalen naar 16 bits of 32 bits. Hiervoor moet wel het aantal

databits voor constanten worden uitgebreid. Een nadeel is dat de maximale frequentie van de processor lager wordt. Dit komt onder andere door de ALU: de opteller moet nu meer bits optellen en wordt daardoor langzamer. De Program Counter is nu 7 bits breed. Het is eenvoudig om dit aantal uit te breiden zodat meer instructies kunnen worden opgeslagen. De ROM wordt hierdoor wel groter.

Observatie van de ontwikkelde assembler-programma's en bijbehorende bitpatronen laat zien dat veel ROM-bits als don't care worden gespecificeerd. Te zien is dat de bits voor de signalen *constant* en *address* vaak niet tegelijk gebruikt worden. Laden van een register en springen naar een adres wordt namelijk niet tegelijkertijd uitgevoerd (dat kan wel, maar we doen het niet). We zouden deze bits kunnen samennemen tot een 8-bits codering waarbij moet worden opgemerkt dat de meest significante bit niet gebruikt wordt voor adressering. Dit levert een besparing op van 7 bits. De ROM wordt dus kleiner. Wel zorgt dit ervoor dat signaal *jcode* met 000 gecodeerd moet worden als het gecombineerde constante/adres en constante vertegenwoordigd. Als de constante/adres een adres voorstelt dan moet signaal *end* 0 zijn (doelregister wordt niet geladen) en moet signaal *acode* 0000 zijn (ALU nop-operatie). We kunnen ook de signalen *sela* en *seld* samennemen want, zo is in de programma's te zien, het doelregister is vaak hetzelfde als één van de bronregisters. Dat levert een besparing van nog eens twee bits op. De mogelijkheden van de processor worden natuurlijk wel beperkt maar dat is in de regel geen probleem.

Al met al zijn er nogal wat uitbreidingsmogelijkheden. Veel van deze mogelijkheden zijn geïmplementeerd in moderne processoren. Het gaat hier echter te ver om zo'n uitgebreide processor te ontwerpen.

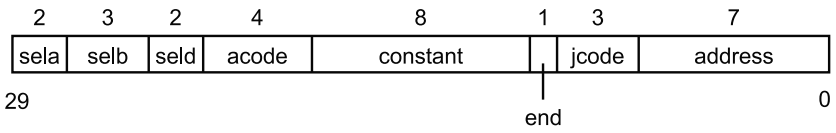
13.10 Opgaven

13.1. Beschrijf waarom de ALU-operatie `nop` handig is.

13.2. Beschrijf waarom de ALU-operatie `pass` handig is.

13.3. Zijn er nog meer (eenvoudige) logische operaties te bedenken? Zijn deze operaties ook met de huidige verzameling van logische operaties te realiseren (want dan zijn de nieuwe operaties overbodig ...)?

13.4. Geef het bitpatroon dat ervoor zorgt dat de EXOR van R2 en R3 in R1 terecht komt (snelschrift: **`exor`** `R1,R2,R3`). Het instructieformaat is te zien in figuur P13.1.

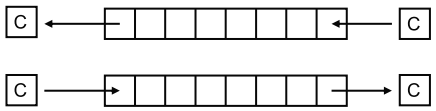


Figuur P13.1: *Het instructieformaat van de vierde versie van de eenvoudige processor.*

13.5. Beschrijf de werkingen van **`and`** `R1,R2,R3` en **`and`** `R1,R3,R2`.

13.6. Eerder zijn de schuifoperaties aan bod geweest. Hierbij werd de meest significante of minst significante bit in de carry flag geplaatst en werd een logische 0

ingeschoven. In plaats van deze 0 zou ook de carry flag kunnen worden ingeschoven. Beschrijf het effect van deze nieuwe operatie. Merk op dat alle registers en flags flankgestuurd zijn, dus de huidige waarde van de carry flag wordt ingeschoven en de huidige meest significante of minst significante bit van het register wordt uitgeschoven. Zie ook figuur P13.2. Bedenk een toepassing van deze nieuwe operaties.



Figuur P13.2: *Naar links en naar rechts roteren met behulp van de carry flag.*

13.7. Stel dat het voorbeeldprogramma in listing P13.1 niet vanaf adres 0 geplaatst is, maar vanaf adres 5 (het programma zelf blijft ongewijzigd). Kan het programma daar zomaar "neergezet" worden? Motiveer het antwoord.

```
 0:   ld   R0,#13      ; load multiplicand (constant 13)
 1:   ld   R1,#11      ; load multiplier (constant 11)
 2:   ld   R2,#0       ; clear result registers
 3:   ld   R3,#0       ; or use ld R3,R2
 4:   cmp  R1,#0       ; is multiplier already 0 ?
 5:   je   @11         ; yes it is, so jump!
 6:   ld   F,#0        ; clear the flags (clears carry)
 7:   add  R2,R2,R0    ; add multiplicant (low byte)
 8:   add  R3,R3,#0    ; process carry in high byte
 9:   sub  R1,R1,#1    ; multiplier one off
10:   jmp  @4          ; and again
11:   ld   R0,R2       ; result to output*
12:   jmp  @12         ; hold your horses!
```

Listing P13.1: *Code voor het vermenigvuldigen van 11 met 13.*

13.8. In het programma in listing P13.1 wordt direct na de twee **add**-instructies een **sub**-instructie uitgevoerd. Maar de **add**- en **sub**-instructies verwerken ook de carry-flag. Moet dan vóór de **sub**-instructie de carry flag niet gewist worden?

13.9. De instructie **not** `Rx,#const` "bestaat" niet. Is deze instructie wel te maken? Motiveer het antwoord.

13.10. Stel dat er helemaal geen **not**-instructie zou bestaan. Kan dan toch de NOT-operatie worden uitgevoerd/verwerkt? (ja dat kan). Motiveer het antwoord.

13.11. Ontwerp een instructie die twee registers met elkaar vermenigvuldigd. Het antwoord bestaat uit twee keer zoveel bits als de bitbreedte van de registers. Zorg ervoor dat alleen de minst significante bits worden opgeslagen. Geef de benodigde VHDL-code die in de processorbeschrijving moet worden geplaatst.

13.12. Ontwerp een programma dat de input eenmaal inleest en het aantal enen bepaalt in de ingelezen waarde. De pseudo-code is gegeven in listing P13.2.

```
R3 := 0;                        // result
R1 := 8;                        // counter
R2 := input;                    // read input once
do                              // loop
    R2 := shr(R2);              // shift right R2, low bit in carry
    R3 := R3 + 0 + carry;       // add carry
    carry := 0                  // clear the carry
    R1 := R1 - 1 - carry;       // decrement counter
while (R1 <> 0);                // as long as R1 not equal to 0
```

Listing P13.2: *Code voor het tellen van het aantal enen in de input.*

13.13. Het ontwikkelde programma uit opgave 13.12 kent een lus die een aantal klokpulsen duurt. Hoeveel klokpulsen duurt het uitvoeren van de lus?

13.14. Gegeven is de pseudo-code in listing P13.3. Vertaal deze code naar assembler. Gebruik hierbij vergelijk- en spronginstructies.

```
if R1 = 5 then
    R2 := 8;
else
    R2 := 10;
end if;
```

Listing P13.3: *Code voor een beslissing.*

13.15. Een ontwerper wil een wachtlus beschrijven waarbij op basis van de zero flag wordt getest of de registers allemaal op 0 staan. Zie listing P13.4. Werkt deze opzet correct?

```
        ld   R1,#<value1>   ; load value 1
        ld   R2,#<value2>   ; load value 2
        ld   R3,#<value3>   ; load value 3
        ld   F,#0           ; clear flags
delay:  sub  R1,R1,#1       ; subtract 1 from low byte
        sub  R2,R2,#0       ; subtract carry from middle byte
        sub  R3,R3,#0       ; subtract carry from high byte
        jne  delay          ; jump if not equal
```

Listing P13.4: *Code voor wachtlus gebaseerd op de zero flag.*

13.16. Gegeven is de wachtlus gebaseerd op de carry flag, te zien in listing P13.5. Verder is gegeven dat de processor loopt op 50 MHz en dat de wachttijd 2/3 seconden bedraagt. Bereken de waarden waarmee de registers geladen moeten worden.

```
        ld   R1,#<value1>   ; load value 1
        ld   R2,#<value2>   ; load value 2
        ld   R3,#<value3>   ; load value 3
        ld   F,#0           ; clear flags
delay:  sub  R1,R1,#1       ; subtract 1 from low byte
        sub  R2,R2,#0       ; subtract carry from middle byte
        sub  R3,R3,#0       ; subtract carry from high byte
        jnc  delay          ; jump if no carry
```

Listing P13.5: *Code voor wachtlus gebaseerd op de carry flag.*

13.17. Gegeven is de code in listing P13.6. Wat zijn de waarden (of inhouden) van $R1$ en $R2$ na uitvoeren van de code?

```
    ld   R1,#13
    ld   R2,#45
    exor R1,R1,R2
    exor R2,R2,R1
    exor R1,R1,R2
```

Listing P13.6: *Code.*

13.18. In listing P13.7 is de pseudo-code te zien voor het berekenen van de deling van twee getallen. De code voert de deling A/B uit. Na de deling is het quotiënt te vinden in variabele `q` en de rest is te vinden in variabele `a`. Ontwerp een assemblerprogramma voor deze deelroutine.

```
a := ..;            // A is dividend
b := ..;            // B is divisor
q := 0;             // Reset quotient
while (a>=b) do     // If A >= B then ...
    a := a-b;       // the quotient is ...
    q := q+1;       // one more ...
end while;          // At the end, A holds the remainder
```

Listing P13.7: *Code voor een deling.*

Bibliografie

Veel materiaal is tegenwoordig (alleen) via Internet beschikbaar. Voorbeelden hiervan zijn de datasheets van ic's die alleen nog maar via de website van de fabrikant beschikbaar worden gesteld. Dat is veel sneller toegankelijk dan boeken en tijdschriften. De keerzijde is dat websites van tijd tot tijd veranderen of verdwijnen. De geciteerde weblinks werken dan niet meer. Helaas is daar niet veel aan te doen. Er is geen garantie te geven dat een weblink in de toekomst beschikbaar blijft.

[1] LaTeX Project Team. *LaTeX – A document preparation system*. URL: https://www.latex-project.org/ (bezocht op 30-12-2018) (blz. iii).

[2] Tex User Groups. *TeX Live Website*. URL: https://www.tug.org/texlive/ (bezocht op 30-12-2018) (blz. iii).

[3] Benito van der Zander. *TeXstudio, LaTeX made comfortable*. URL: https://www.texstudio.org/ (bezocht op 04-05-2019) (blz. iii).

[4] Bitstream Inc. *Charter Fonts*. URL: https://www.ctan.org/pkg/charter (bezocht op 30-12-2018) (blz. iii).

[5] Artifex. *Nimbus 15 Mono*. Okt 2015. URL: https://www.ctan.org/tex-archive/fonts/nimbus15 (bezocht op 30-12-2018) (blz. iv).

[6] J. Hoffman. *listings – Typeset source code listings using LaTeX*. URL: https://www.ctan.org/pkg/listings (bezocht op 30-12-2018) (blz. iv).

[7] J.E.J. op den Brouw. *askmaps -– Typeset American style Karnaugh maps*. Dec 2013. URL: https://www.ctan.org/pkg/askmaps (bezocht op 20-12-2018) (blz. iv).

[8] T. Tantau. *pgf – Create PostScript and PDF graphics in TeX*. Aug 2015. URL: https://www.ctan.org/pkg/pgf (bezocht op 30-12-2018) (blz. iv).

[9] G.H. Mealy. "A method for synthesizing sequential circuits". In: *The Bell System Technical Journal* 34.5 (sep 1955), p. 1045–1079 (blz. 4).

[10] E.F. Moore. "Gedanken-experiments on Sequential Machines". In: *Annals of Mathematical Studies* 34 (1956), p. 129–153 (blz. 4).

[11] H. Kaeslin. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, 2008, p. 787. ISBN: 978-0-521-88267-5 (blz. 4).

[12] E.J. McCluskey en S.H. Unger. "A Note on the Number of Internal Variable Assignments for Sequential Switching Circuits". In: *IRE Transactions on Electronic Computers* EC-8.4 (dec 1959), p. 439–440 (blz. 9).

[13] D.B. Armstrong. "On the Efficient Assignment of Internal Codes to Sequential Machines". In: *IRE Transactions on Electronic Computers* EC-11.5 (okt 1962), p. 611–622 (blz. 10).

[14] R. Amann, B. Eschermann en U.G. Baitinger. “PLA based finite state machines using Johnson counters as state memories”. In: *Computer Design: VLSI in Computers and Processors, 1988. ICCD ’88., Proceedings of the 1988 IEEE International Conference on*. Okt 1988, p. 267–270 (blz. 10).

[15] A.P. Thijssen en H.A. Vink. *Digitale Techniek, van probleemstelling tot realisatie deel 2*. 5de ed. Delft University Press, 2000, p. 210. ISBN: 90-407-1838-5 (blz. 10).

[16] M.C. Paull en S.H. Unger. “Minimizing the Number of States in Incompletely Specified Sequential Switching Functions”. In: *IRE Transactions on Electronic Computers* EC-8.3 (sep 1959), p. 356–367 (blz. 30).

[17] G.A. Blaauw. *Digital system implementation*. Prentice-Hall Series in Automatic Computation. Prentice Hall PTR, 1976. ISBN: 9780132122412 (blz. 44).

[18] J. Rose e.a. “Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency”. In: *IEEE Journal of Solid-State Circuits* 25.5 (okt 1990), p. 1217–1225 (blz. 101).

A

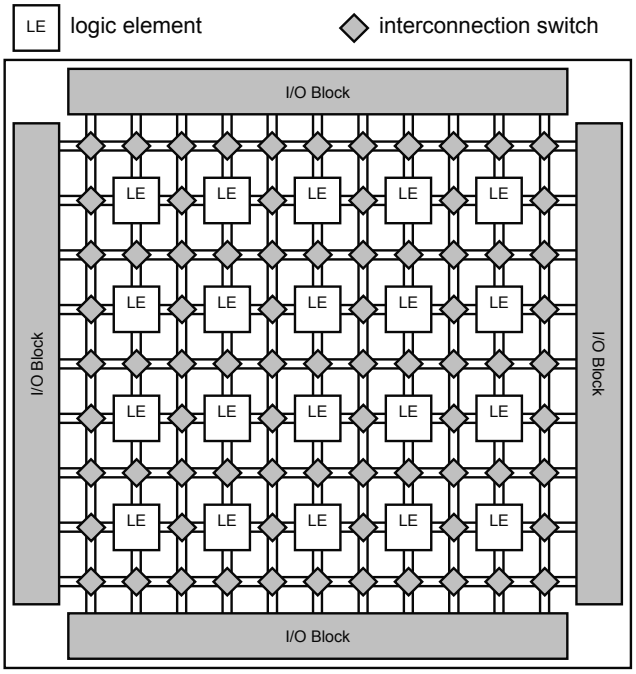
De FPGA

De ontwerper heeft de keuze om een digitaal systeem te realiseren met configureerbare ic zoals een FPGA (Field Programmable Gate Array) of met een ASIC (Application Specific Integrated Circuit). Een FPGA is een configureerbaar ic waarin de de schakeling wordt opgebouwd met behulp van *cellen*. De cellen vervullen elk (een deel van) een logische functie zodat het geheel de juiste werking vertoont. Bij een ASIC worden in de fabriek de transistoren (daar bestaat een ic namelijk uit) direct om te juiste plaats gelegd. De inhoud van de ASIC is dus na productie niet te veranderen terwijl in een FPGA een nieuwe configuratie kan worden geladen als dat nodig mocht zijn.

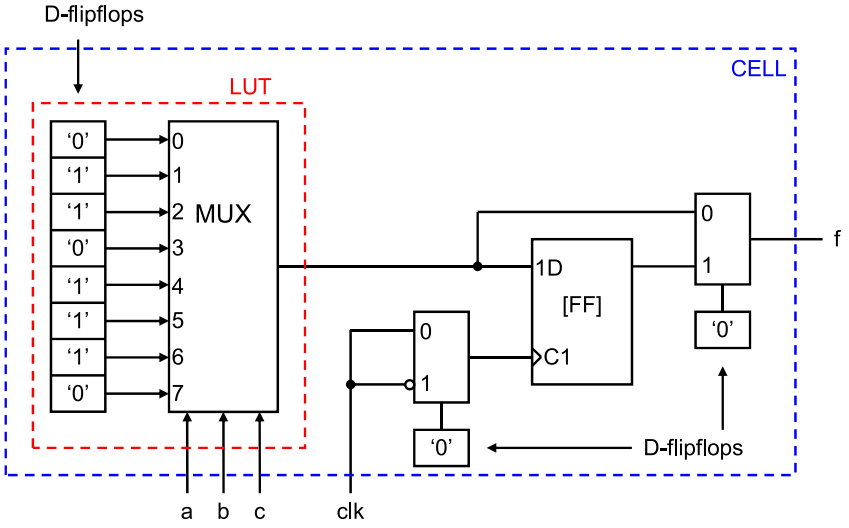
FPGA's hebben een behoorlijk verschillende architectuur ten opzichte van PAL's, PLA's en ROM's omdat FPGA's niet bestaan uit AND- en OR-matrixen maar uit cellen. In figuur A.1 is de globale opbouw van een FPGA te zien. Er zijn drie typen componenten te onderscheiden. Aan de randen van het ic zijn de zogenoemde *I/O Blocks* geplaatst. Deze componenten verzorgen de communicatie met de kern van de FPGA en de buitenwereld. De I/O Blocks bevatten vaak speciale voorzieningen zoals Schmitt-trigger ingangen, flipflops die metastabiel gehard zijn, tri-state en open collecter uitgangen, *line drivers* voor hoge snelheidsbussen en voorzieningen om een keur van spanningsniveaus aan te kunnen (denk hierbij aan TTL en CMOS).

De kern bestaat uit een matrix van programmeerbare kruispunten (*Interconnection Switch*) en logische elementen (*Logic Elements*). Die laatste worden ook wel *cellen* genoemd. Elke cel vervult een logische functie zodanig dat het geheel de juiste werking vertoont. Met behulp van de programmeerbare kruispunten kunnen de diverse cellen via verbindingen (*Interconnect*) met elkaar verbonden worden.

Elke cel bestaat uit een programmeerbare multiplexer en een flipflop. In figuur A.2 in een vereenvoudigd voorbeeld van een cel te zien. De multiplexer realiseert een combinatorische functie en wordt ook wel een Look Up Table (LUT) genoemd. Met behulp van configuratiebits wordt de functie vastgelegd. De stuuringangen van de multiplexer zijn verbonden met de ingangen van de functie.



Figuur A.1: *Globale opbouw FPGA.*



Figuur A.2: *Schematische weergave van een cel in een FPGA.*

Twee andere configuratiebits zorgen ervoor dat de cel te gebruiken is als een stukje combinatorische logica (zoals in de figuur te zien is) of als een stukje sequentiële logica. Daarnaast is de flank waarop de flipflop reageert met een configuratiebit in te stellen.

De configuratiebits zijn uitgevoerd als RAM-cellen, dat wil zeggen dat als de spanning uitvalt de inhoud verdwijnt. Bij het aanzetten van de spanning moeten de configuratiebits dus opnieuw geladen worden. Dat gebeurt in de regel met een kleine seriële EEPROM (Electrical Erasable Programmable ROM) die met de FPGA verbonden is. De FPGA leest de inhoud van de EEPROM in de configuratiebits in waarna de FPGA zijn taak kan uitvoeren. Dit inlezen gaat zeer snel, meestal niet meer dan enkele milliseconden.

FPGA's worden gebruikt als een systeem complex is, time-to-market kort is, de prestatie (bijvoorbeeld snelheid) het toelaat om een FPGA te gebruiken en het aantal te bouwen systemen beperkt is. Schakelingen kunnen snel worden ontworpen met een HDL (Hardware Description Language). FPGA-leveranciers leveren software tools om de HDL-beschrijving van een schakeling te synthetiseren naar een een schakeling die volledig bestaat uit *primitieven*. Voorbeelden van primitieven zijn poorten, multiplexers, optellers, aftrekkers, vergelijkers en flipflops. Deze beschrijving van primitieven moeten dan in de cellen geplaatst worden (fitting). Daarna wordt een configuratiebestand gegenereerd (assembling) die in de FPGA geladen wordt (programming).

Fabrikanten brengen op de FPGA's vaak speciale componenten aan. Te denken van aan vermenigvuldigers, DSP's (Digital Signal Processors), Ethernet-modules en RAM. Met behulp van de ontwikkelsoftware kan een ontwerper deze componenten gebruiken. Het voordeel van de speciale componenten is dat er geen logische cellen gebruikt worden om een component te implementeren en dat de componenten sneller zijn dan wanneer ze met cellen worden gerealiseerd.

In de praktijk komen LUT's met vier en zes (stuur-)ingangen voor. Daarmee zijn logische functies met vier respectievelijk zes variabelen te realiseren. Rose toont in [18] aan dat een aantal in de industrie gebruikte ontwerpen het best kunnen worden gerealiseerd met LUT's van drie of vier variabelen en dat het efficiënt is om een cel met een flipflop uit te rusten.

FPGA's zijn er in alle soorten en maten. De kleinste FPGA's hebben zo'n 10.000 cellen en de grootste zo'n 500.000[1]. Veelal zijn snelle en langzame versies te krijgen (de zogenoemde *speed grade*). Ook qua verpakkingen is er nogal een verschil te vinden. Bij de zogenoemde Ball Grid Array (BGA) zijn aan de onderkant van het ic halve bollen te vinden. Bij de Pin Grid Array (PGA) is de onderkant voorzien van pennen die door de print heen steken. Een andere verpakking is Thin Quad Flat Package (TQFP). Hierbij is het ic voorzien van aansluitingen aan de zijkanten van de behuizing van het ic.

[1] Fabrikant Xilinx heeft op het moment van schrijven een FPGA met een kleine 9 miljoen cellen beschikbaar.

Index