Ontwerptraject digitale IC's C4

DHR. DR. IR. S.H. GEREZ

1 Inleiding

De bekende *Wet van Moore* (genoemd naar Gordon Moore, een van de oprichters van Intel) stelt dat het aantal transistoren op een IC elke achttien maanden verdubbelt. De wet werd in 1965 geformuleerd en blijkt nog altijd bruikbaar te zijn. Lag het aantal transistoren van een complexe IC in het begin van de jaren zeventig rond de duizend; via een miljoen aan het begin van de jaren negentig groeide dit aantal naar circa een miljard in 2005.

Een dergelijke mate van complexiteit kan alleen bereikt en beheerst worden:

- Door gebruik te maken van geavanceerde *synthesetechnieken*. Deze stellen een ontwerper in staat vanuit een compacte beschrijving een grote schakeling te generen. Men gebruikt hierbij een *hardware beschrijvingstaal* om een schakeling op een bepaald abstractieniveau te beschrijven. Een of meer *synthesegereedschappen* zetten vervolgens zo'n beschrijving om in een *lay-out* die geschikt is voor fabricatie.
- Door hergebruik van blokken van een ontwerp met een duidelijk gedefinieerde functionaliteit. In deze context wordt vaak de term *IP-blok* gebruikt, waarbij IP een afkorting is van *Intellectual Property* (intellectueel eigendom). De gedachte achter deze term is dat de ontwerper(s) van het blok tijd en moeite hebben geïnvesteerd in het ontwerp en dat het ontwerp daarom economische waarde vertegenwoordigt.

Dit hoofdstuk gaat voornamelijk over het ontwerpen van digitale blokken op een IC door middel van *standaardcelsynthese*, de meest gangbare vorm van digitaal ontwerpen. Een *standaardcel* is een relatief kleine digitale schakeling (bijv. een *NAND-poort*, een *full-adder* of een *flipflop*) waarvan de lay-out en dus ook het transistorschema vastligt. Een standaardcel maakt deel uit van een *standaardcelbibliotheek*, een verzameling van elementaire schakelingen waaruit een synthesegereedschap kan kiezen bij de samenstelling van een ontwerp. Zo'n bibliotheek wordt meestal betrokken van een extern bedrijf (of van een gespecialiseerde afdeling van het eigen bedrijf).

Andere vormen van ontwerpen van digitale IC's, waarbij de ontwerper zelf functies uit transistoren samenstelt en eventueel ook de lay-out (deels) handmatig bepaalt, blijven buiten beschouwing. Deze vormen van ontwerpen zijn arbeidsintensiever en worden alleen gebruikt als er bijzondere eisen worden gesteld aan de snelheid, de oppervlakte of het vermogensgebruik van een IC.

Hoewel het onderscheid niet altijd scherp aan te geven is, worden IC's over het algemeen ingedeeld in twee klassen:

1. De eerste is die van IC's die breed inzetbaar zijn en een veelheid aan taken aankunnen, de *general-purpose integrated circuits*. Voorbeelden van IC's in deze klasse zijn *microprocessoren* en *digitale signaalprocessoren* (*DSP's*).
2. De tweede klasse bevat IC's die voor een beperkt toepassingsgebied zijn ontworpen: de *application-specific integrated circuits* (*ASIC's*).

Een aan een IC verwant implementatiemedium voor digitale schakelingen is de *field-programmable gate array* (*FPGA*). FPGA's zijn IC's waarop elementaire logische functies en geheugenelementen op een regelmatige manier zijn gerangschikt en waarvan de onderlinge verbindingen programmeerbaar zijn. Op deze manier kan 'in het veld' de functie van een FPGA ingesteld worden. Deze flexibiliteit gaat ten koste van oppervlakte en snelheid. Het ontwerptraject van FPGA's wijkt niet veel af van het traject voor digitale IC's.

2 Ontwerpmethodologie

2.1 Optimalisatiecriteria

Tijdens het ontwerpen moeten er keuzes gemaakt worden. Bij die keuzes zal men proberen het ontwerp volgens een of meer criteria te optimaliseren. Vaak zijn deze criteria tegenstrijdig, wat het ontwerpen moeilijk maakt. De meest gangbare optimalisatiecriteria in IC-ontwerp zijn:

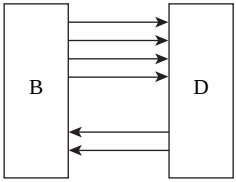
- *Oppervlakte*. Ondanks de steeds verder gaande miniaturisatie is de gebruikte siliciumoppervlakte nog altijd een dominante component van de kostprijs van een IC. Minimalisatie van de oppervlakte is daarom een primair doel. Een IC met kleinere oppervlakte zal ook een hogere *yield* (opbrengst) hebben omdat de kans op een defect tijdens productie (bijv. door toedoen van een stofdeeltje) kleiner wordt.
- *Snelheid*. Verhoging van de snelheid gaat vaak ten koste van de oppervlakte. Om een berekening te versnellen zal men meer parallel (tegelijkertijd) moeten doen en dus meer hardware beschikbaar moeten hebben. Omdat veel standaarden de snelheid vastleggen (bijv. het aantal keren per seconde dat een audiosignaal bemonsterd is) heeft het geen zin snellere schakelingen te ontwerpen dan nodig. Er zijn ook situaties denkbaar waarbij vooraf niet bekend is wat de eisen aan de snelheid zijn, maar wel dat een snellere schakeling voor een klant aantrekkelijker zal zijn. Een voorbeeld is het ontwerp van een processor die meer berekeningen binnen een bepaalde tijd kan uitvoeren als zijn *kloksnelheid* hoger is.
- *Vermogensgebruik*. Dit criterium is vooral van belang in draagbare, door een batterij gevoede, apparatuur. Ook als een apparaat door het net wordt gevoed, is een hoge dissipatie ongewenst (denk bijv. aan microprocessoren die tientallen Watts dissiperen en die soms zelfs door water gekoeld moeten worden). Aanpassingen in het ontwerp om vermogensgebruik te verkleinen kunnen ten koste gaan van oppervlakte of snelheid.
- *Testbaarheid*. Vanwege het optreden van defecten tijdens fabricatie, is het van cruciaal belang dat elk gefabriceerd IC uitgebreid getest wordt voor het aan de klant wordt geleverd. Testtijd is daarom naast siliciumoppervlakte een component van de kostprijs van een IC. *Testbaar ontwerpen* leidt tot hardwarestructuren die aan een ontwerp worden toegevoegd met het doel de testbaarheid te vergroten (en de testtijd te verkorten). Testbaar ontwerpen kost oppervlakte.
- *Ontwerptijd*. Meestal is er geen tijd om een IC optimaal te ontwerpen. In verband met het veroveren of handhaven van een bepaald marktaandeel en het tevreden houden van een klant is het van belang dat een IC snel ontworpen wordt.

2.2 *Structuur en gedrag, hiërarchie en abstractie*

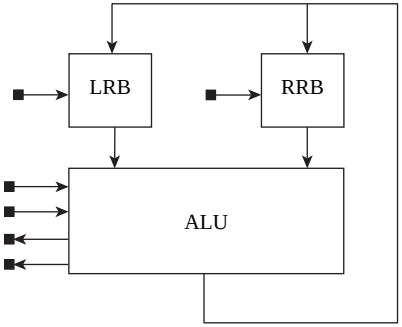
De complexiteit van een schakeling is alleen te beheersen door middel van structuur. In de context van ontwerpmethodologie zegt men dat een schakeling structuur heeft als deze bestaat uit *deelschakelingen*. De deelschakelingen en de *interconnectie*, de manier waarop de deelschakelingen met elkaar zijn verbonden, leggen de structuur vast.

Een goede structuur beperkt het aantal deelschakelingen tot een overzichtelijk aantal. Elke deelschakeling zal op zijn beurt zelf structuur hebben. Men kan doorgaan met het steeds verder opdelen van schakelingen totdat een elementair niveau wordt bereikt, bijv. het niveau van de eerder genoemde standaardcellen.

Beschouw het voorbeeld van een processor P die opgebouwd is uit een *datapad* D en een *besturing* B zoals weergegeven in figuur 2.1. De bovenste signalen stellen *stuursignalen* voor, zoals geheugenadressen, selectiesignalen van *multiplexers*, enz. De onderste signalen zijn *statussignalen*, uitkomsten van logische bewerkingen, bijvoorbeeld als een test of twee getallen gelijk zijn. Het datapad zou kunnen bestaan uit een *ALU* en twee registerbanken, de linker registerbank LRB en de rechter registerbank RRB, zoals te zien in figuur 2.2.



Figuur 2.1 Schema van een processor bestaande uit een besturing B en een datapad D

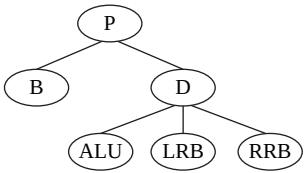


Figuur 2.2 Opbouw van datapad D uit een ALU en twee registerbanken

Merk op dat datapad D de twee genoemde schema's aan elkaar koppelt. Terwijl in figuur 2.1 D een deel van P is, vormt figuur 2.2 het geheel van D waarin zijn onderdelen zichtbaar zijn gemaakt. Dit concept vindt men terug in *schema-editors* waarmee een ontwerper de geneste structuur van een schakeling vastlegt. In termen van schema-editors zegt men dat figuur 2.1 het schema van P weergeeft waarin het symbool van D zichtbaar is. Figuur 2.2 bevat het schema van D. In plaats van grafisch kan structuur ook tekstueel worden vastgelegd zoals te lezen in paragraaf 4.

Een geneste structurele beschrijving wordt ook wel een *hiërarchie* genoemd. De hiërarchie kan gevisualiseerd worden d.m.v. een *decompositieboom*. In zo'n boom wordt een schakeling gerepresenteerd door een knooppunt (ovaal). Op een lager niveau staan de knooppunten die de deelschakelingen representeren. Verbindingslijnen wijzen de deelschakelingen aan waaruit een schakeling is samengesteld. De decompositieboom voor het boven gepresenteerde voorbeeld is getekend in figuur 2.3.

Het gebruik van een deelschakeling in een omvattende schakeling wordt de *instantiatie* van de deelschakeling genoemd. In figuur 2.3



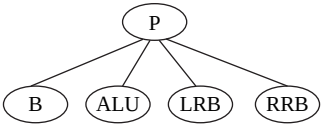
Figuur 2.3 De decompositieboom van het processorvoorbeeld

instantieert P de blokken B en D en D instantieert op zijn beurt de blokken ALU, LRB en RRB. Een deelschakeling kan meervoudig geïnstantieerd worden. In het voorbeeld kunnen de registerbanken identiek zijn en beschreven worden door een schema RB. De twee instanties van RB krijgen dan aparte namen LRB en RRB om ze te kunnen onderscheiden.

Het beschouwen van een bepaalde laag in de hiërarchie met weglating van details over de onderliggende structuur heet *abstractie*. In een ontwerpomgeving waar digitale schakelingen worden opgebouwd uit standaardcellen, is het laagste abstractieniveau het *poortniveau* (Eng. *gate level*). Een belangrijk hoger abstractieniveau is het *register-transfer niveau* (Eng. *register-transfer level,* afgekort als *RTL* of RT-niveau). Op dit niveau redeneert men in termen van geklokte geheugenelementen en combinatorische logica die bepaalt hoe de inhoud van de geheugenelementen per klokslag verandert. In paragraaf 3 wordt hier dieper op ingegaan.

Een functionele beschrijving van een deelschakeling die aangeeft hoe de uitgangen uit de ingangen berekend worden zonder te verwijzen naar een inwendige structuur wordt een *gedragsbeschrijving* genoemd. De top-down stijl van ontwerpen bestaat uit het herhaaldelijk vervangen van gedrag door structuur totdat voor elk onderdeel een voldoende mate van detaillering is bereikt.

Het vervangen van gedrag door structuur heet *synthese* (ongeacht of dit 'handmatig' of met ontwerpgereedschap wordt gedaan). Men onderscheidt verschillende vormen van synthese afhankelijk van het abstractieniveau waarin het gedrag is beschreven. Zo heeft *logische*



Figuur 2.4 De platgeslagen decompositieboom van figuur 2.3

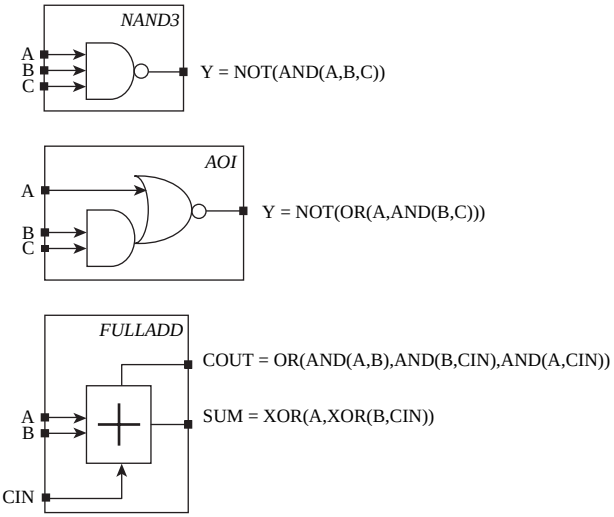
synthese betrekking op het omzetten van gedrag gegeven in RTL-beschrijvingen in schakelingen opgebouwd uit standaardcellen. Op een hoger abstractieniveau zullen gedragsbeschrijvingen niet langer verwijzen naar een klok en zal het opdelen van een berekening over een aantal klokslagen een taak van de synthese zijn. Dit soort synthese wordt *architectuursynthese* of *hoog-niveausynthese* genoemd.

Een hiërarchische opdeling met een overzichtelijk aantal deelschakelingen bij een overgang naar een lager niveau is vooral nuttig voor een menselijke ontwerper. Op het moment dat een schakeling door een computerprogramma moet worden verwerkt kan het handiger zijn om geen hiërarchie te hebben, bijvoorbeeld om optimalisatie over modulegrenzen heen mogelijk te maken. Het verwijderen van hiërarchische tussenlagen noemt men het *platslaan* van de hiërarchie (Eng. *flattening*). Figuur 2.4 toont de platgeslagen decompositieboom van het voorbeeld uit figuur 2.3.

3 Implementatie van RTL-beschrijvingen

3.1 Bouwblokken

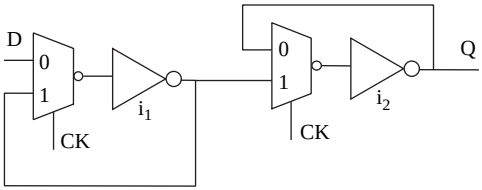
Een logische schakeling die geen intern geheugen heeft heet *combinatorisch*. Zij heeft de eigenschap dat de uitgangswaarden eenduidig te bepalen zijn uit de momentane ingangswaarden, na inachtneming van de *propagatievertraging* van ingang naar uitgang. Als daarentegen de uitgangen niet alleen van de huidige ingangen



Figuur 3.1 Drie voorbeelden van combinatorische standaardcellen: een 3-input NAND, een and-or-invert functie en een full adder

maar van de voorgeschiedenis aan ingangswaarden afhangen, is de schakeling *sequentieel*. Zo'n schakeling heeft intern geheugen en de toestand van het geheugen bepaalt mede de uitgangswaarden. In figuur 3.1 zijn een drietal combinatorische standaardcellen afgebeeld; bij elk uitgangssignaal is de berekende logische functie vermeld.

Samenstellingen van combinatorische schakelingen blijven combinatorisch zolang er geen signalen van een uitgang naar een ingang worden teruggekoppeld. Een *terugkoppeling* kan leiden tot het ontstaan van geheugen of tot oscillaties. Deze effecten maken schakelingen met terugkoppelingen lastig te analyseren. Het ontwerpen gaat veel gemakkelijker indien geheugens alleen worden



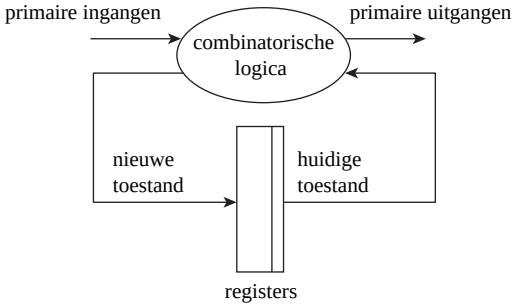
Figuur 3.2 Schema van een positive edge-triggered flipflop

samengesteld uit daarvoor bedoelde sequentiële standaardcellen. De meest gebruikelijke cel is de *positive edge-triggered D-flipflop*. Een mogelijke implementatie met behulp van twee inverterende multiplexers en twee inverters is te zien is in figuur 3.2. De waarde van de data-ingang D wordt op de opgaande flank van het *kloksignaal* CK overgenomen op uitgang Q (men zegt ook wel dat het signaal op D wordt ingeklokt). In de rest van de klokperiode behoudt de uitgang zijn waarde. Een praktische implementatie van een *D-flipflop* zal ook set- en resetsingalen hebben die synchroon (op de opgaande flank van de klok) of asynchroon (onmiddellijk) het opgeslagen signaal op een gespecificeerde waarde forceren.

3.2 RTL-structuur

Hardware die volgens de zojuist beschreven principes wordt opgebouwd, heeft een structuur zoals geschetst in figuur 3.3. In de figuur zijn alle flipflops van de schakeling samengenomen tot één symbool aangeduid met het trefwoord registers. De inhoud van de registers vormt de huidige toestand van de schakeling. Uit de huidige toestand en de waarde van de *primaire ingangen* worden de nieuwe toestand en de *primaire uitgangen* berekend. Dit gebeurt door *combinatorische logica*.

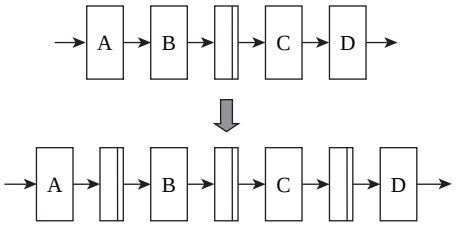
Men kan de hardware ook zien als een verzameling signalen die door combinatorische logica in nieuwe signalen worden omgezet. Sommige van deze signalen zijn in registers opgeslagen. Dit zijn de zogenaamde *geklokte signalen*.



Figuur 3.3 Algemene structuur van sequentiële logica

Het feit dat de flipflops flankgevoelig zijn betekent dat de hele klokperiode beschikbaar is voor het berekenen van de nieuwe ingangswaarde. Dat wil ook zeggen dat de maximale *propagatietijd* door de combinatorische logica (het langste pad) een ondergrens vormt voor de klokperiode. Als men iets preciezer kijkt, moet men de ondergrens verhogen met de *setuptijd* van de registers. Dit is de tijd dat het *D-signaal* van een flipflop stabiel moet zijn voordat de klokflank verschijnt. Om het D-signaal betrouwbaar te kunnen inklokken, moet het ook na de opgaande flank van de klok een tijd stabiel blijven. Deze tijd wordt de *holdtijd* genoemd.

Figuur 3.3 beschrijft de kijk op hardware op register-transfer niveau (RTL). Een kenmerk van het ontwerpen op dit niveau is dat vastligt welke de geklokte signalen zijn, hoe de berekening zich in opeenvolgende klokperiodes ontwikkelt en welke combinatorische functies nodig zijn. De ontwerpgereedschappen voor logische synthese die in het vervolg worden besproken, zullen de verzameling geklokte signalen intact laten. De keuze voor de plaatsing van registers in de combinatoriek is daarom bepalend voor de maximaal haalbare kloksnelheid. Er zijn vele technieken om de *RTL-structuur* te veranderen ten behoeve van een hogere kloksnelheid. Twee daarvan zullen hier kort worden toegelicht.

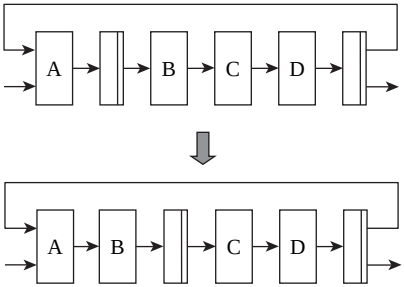


Figuur 3.4 Het verkorten van de klokperiode door middel van pipelining

Pipelining
Pipelining houdt in dat er extra registers in het signaalpad van ingang naar uitgang worden geplaatst, waardoor er minder combinatorische logica in een klokslag hoeft te worden doorlopen. Dit is geïllustreerd in figuur 3.4, waar de blokken A, B, C en D combinatorische functies zijn. Als er geen terugkoppelingen in het signaalpad zijn zal pipelining de uitgevoerde berekening niet verstoren, afgezien van het feit dat het aantal klokperiodes tussen consumptie van een ingangssignaal en productie van een bijbehorend uitgangssignaal, de zogenaamde *latency*, toeneemt.

Retiming
Retiming komt neer op het omwisselen van de volgorde tussen registers en combinatorische logica met als doel het balanceren van de lengte van de signaalpaden tussen registers. Een voorbeeld is te zien in figuur 3.5. Aannemende dat de vertraging in de combinatorische blokken A, B, C en D ongeveer even groot is, is er voor de transformatie een lang *combinatorisch pad* door B, C en D en een kort pad door A. Na de transformatie zijn er twee ongeveer even lange paden. Retiming laat de relatie tussen in- en uitgangssignalen geheel intact en kan ook worden toegepast in situaties met terugkoppeling.

Niet alle hardware kan volgens de structuur van figuur 3.3 gebouwd worden. Een IC van enige complexiteit zal bijv. meerdere kloksignalen hebben. Registers die door hetzelfde kloksignaal worden



Figuur 3.5 Het verkorten van de klokperiode door middel van retiming

bestuurd behoren tot hetzelfde *klokdomein*. De structuur van figuur 3.3 geldt dan voor elk klokdomein afzonderlijk. Verder kan het voorkomen dat een kloksignaal als datasignaal wordt gebruikt om een afgeleide klok te maken of dat sommige signalen op de neergaande klokflank moeten worden ingeklokt, enz. In dergelijke situaties is het aan te raden een ontwerp zodanig op te delen dat de meeste delen voldoen aan de structuur van figuur 3.3 en alle afwijkende zaken onder te brengen in geïsoleerde delen van het ontwerp.

3.3 Logische synthese en lay-out

Vanwege de beschikbaarheid van *computer aided design* (*CAD*) gereedschappen voor logische synthese kan de ontwerper volstaan met een abstracte beschrijving van de combinatorische logica in een hardware beschrijvingstaal (zie paragraaf 4). Het gereedschap zal het zo optimaal mogelijk implementeren van de beschrijving d.m.v. cellen uit de standaardcelbibliotheek voor zijn rekening nemen.

Daartoe zal het eerst de tekstuele beschrijving van de schakeling analyseren en daaruit enerzijds de *registers* en anderzijds de *Booleaanse formules* van de combinatorische logica extraheren. De registers worden afgebeeld op de flipflops uit de bibliotheek. De

logica wordt eerst vereenvoudigd (constante ingangen kunnen bijv. tot gevolg hebben dat NAND-, *NOR-poorten*, enz. ook verdwijnen) en vervolgens gerealiseerd met combinatorische cellen uit de bibliotheek.

Opeenvolgende combinatorische cellen die met elkaar zijn verbonden vormen een *combinatorisch pad*. Het aantal cellen in het langste pad door de combinatorische logica wordt de diepte van de *logica* genoemd. Vaak geldt dat een grotere diepte tot een kleinere oppervlakte leidt, maar ook tot meer vertraging. De klokperiode is een belangrijke randvoorwaarde voor logische synthese. Gegeven de opgegeven waarde voor de klokperiode, zal het synthesegereedschap proberen de kleinst mogelijke schakeling te maken. Een andere parameter voor synthese is de te verwachten *capacitieve belasting* van de uitgangen. Meestal zijn van de standaardcellen verschillende versies in de bibliotheek beschikbaar met variërende uitgangbuffers. Bij een zwaardere belasting zal het synthesegereedschap voor versies met geschikte bufferafmeting kiezen.

Wat betreft de vertragingen in een schakeling opgebouwd uit standaardcellen kan men onderscheid maken tussen interne en externe vertraging. De *interne vertraging* is het gevolg van het schakelen in de cel. De *externe vertraging* ontstaat doordat dat verbindingsdraden tussen de cellen en ingangen van een cel een capaciteit hebben die tijdens een signaalovergang opgeladen of ontladen moet worden. Deze vertraging is dus afhankelijk van de draadlengte.

De klassieke vorm van logische synthese genereert een netwerk van standaardcellen. Dat wordt vervolgens omgezet in een lay-out d.m.v. een CAD-gereedschap voor *placement and routing* dat elke cel een plaats toekent en vervolgens bepaalt hoe de bedrading tussen en over de cellen gelegd moet worden.

Een gevolg van de steeds kleiner wordende afmetingen op IC's is dat de relatieve bijdrage van de externe vertraging aan de totale vertraging groter wordt. Dat maakt het nodig om informatie over draadlengtes (eigenlijk parasitaire capaciteiten en weerstanden) zoals door de lay-outgereedschappen bepaald, terug te koppelen naar het netwerk om te controleren of er ook na lay-out aan alle timingeisen wordt voldaan. Deze terugkoppeling heet *back annotation*.

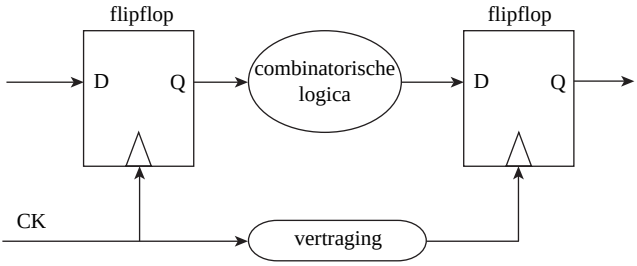
Als de controle negatief uitvalt, is men gedwongen de lay-out opnieuw te genereren of zelfs de logische synthese opnieuw te doen. Bij een nieuwe poging voor lay-outgeneratie of logische synthese zal men dan het CAD-gereedschap zo proberen te sturen dat de gevonden problemen in de nieuwe oplossing vermeden zullen worden. In het algemeen is er geen garantie dat deze stappen tot het gewenste resultaat zullen leiden en dreigt men de lus van lay-outgeneratie gevolgd door controle op timing eindeloos te doorlopen. Het probleem van het sluitend krijgen van de randvoorwaarden voor timing heet *timing closure*.

In de moderne ontwerpgereedschappen verdwijnt steeds meer het onderscheid tussen logische synthese en lay-outontwerp: CAD-gereedschappen voor lay-out zijn in staat zelf de controle op timing uit te voeren en waar nodig het netwerk te modificeren.

Een bijzondere modificatie van het netwerk betreft het toevoegen van een *klokboom* (Eng. *clock tree*). Beschouw nogmaals de hardwarestructuur van figuur 3.3. Als de toestand op tijdstip n gegeven is door $T(n)$, is het de bedoeling dat de bits die de 'nieuwe toestand' $T(n+1)$ representeren, op het moment van een opgaande klokflank stabiel zijn en in de registers worden opgeslagen. Zodra de bits aan de uitgang van de registers verschijnen, beginnen de signalen die na het inklokken veranderd zijn, door de combinatorische logica te propageren ter berekening van toestand $T(n+2)$.

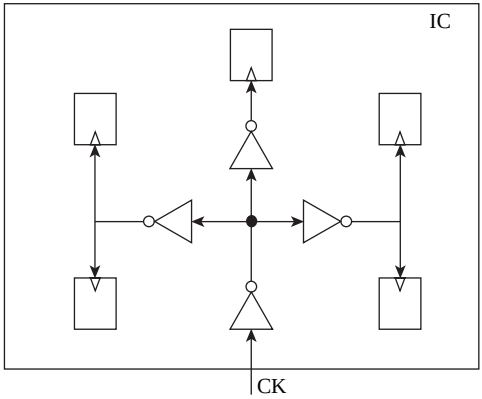
Twee van de flipflops uit de registers zijn weergegeven in figuur 3.6. In de figuur is de mogelijke vertraging in het kloksignaal expliciet zichtbaar gemaakt. Stel nu dat de linker flipflop een bit behorende bij $T(n+1)$ heeft ingeklokt op een opgaande klokflank en dat deze opgaande flank nog niet bij de rechter flipflop is aangekomen. De combinatorische logica aan de uitgang van de linker flipflop is inmiddels bezig toestand $T(n+2)$ uit te rekenen. Bij een te grote vertraging van de klok ten opzichte van de combinatoriek zal de rechter flipflop een bit van $T(n+2)$ in plaats van $T(n+1)$ inklokken en zal er een verkeerde berekening worden uitgevoerd.

Om dit effect te voorkomen wordt de lay-out zodanig ontworpen dat het verschil in aankomsttijd van het kloksignaal bij de flipflops geminimaliseerd wordt. Dit wordt bereikt door een klokboom, een



Figuur 3.6 Situatie waarin vertraging in het kloksignaal leidt tot een foutieve berekening

zich vertakkend gebufferd netwerk waarin de vertraging van bron naar elk uiteinde ongeveer even lang is. Dit idee wordt op symbolische wijze geïllustreerd in figuur 3.7. Waar in de figuur het externe kloksignaal CK op een gebalanceerde manier naar de vijf getekende



Figuur 3.7 Symbolische voorstelling van een klokboom

flipflops verspreid over de hele IC wordt gebracht, vertakt een klokboom zich in de praktijk naar duizenden flipflops.

Het verschil in aankomsttijd van het kloksignaal wordt de *clock skew* genoemd. Voor elk paar flipflops moet de skew kleiner zijn dan het snelste combinatorische pad tussen die flipflops.

4 Hardware beschrijvingstalen

4.1 Overzicht

Met de opkomst van de computertechnologie in de jaren vijftig en zestig van de vorige eeuw zijn de eerste *programmeertalen* ontstaan. Programmeertalen zijn bijzonder nuttig bij het ontwerpen van software: ze leggen een berekening eenduidig vast, ze helpen bij het structureren van complexe berekeningen en de *compiler* schermt de hardware af van de gebruiker waardoor dezelfde software op diverse platforms uitgevoerd kan worden.

De eigenschappen van programmeertalen zijn ook nuttig bij het ontwerpen van hardware. Daarom zijn in het verleden klassieke programmeertalen gebruikt voor het specificeren en modelleren van hardware. Soms gebeurt dat nog steeds (bijv. in de taal *C*). De klassieke programmeertalen missen echter ondersteuning voor zaken die kenmerkend zijn voor hardware, met name voor het modelleren van het feit dat alle elementen in hardware tegelijkertijd actief zijn (parallellisme). Om deze reden zijn er al vanaf de jaren zestig zogenaamde hardwarebeschrijvingstalen bedacht (Eng. *hardware description language*, afgekort *HDL*).

De vele initiatieven die oorspronkelijk zijn ontplooid om HDL, afgekort *HDL*'s te ontwikkelen, hebben lange tijd niet geresulteerd in de algemene acceptatie van een specifieke taal. Dit veranderde met de komst van *VHDL*, een taal die begin jaren tachtig bedacht is door een groep van experts, bij elkaar gebracht in het kader van het VHSIC (Very High Speed Integrated Circuits) programma van het Amerikaanse ministerie van defensie. VHDL is dan ook een afkorting van *VHSIC Hardware Description Language*. VHDL is voor het

eerst in 1987 door de *IEEE* gestandaardiseerd (en later nog weer in 1993, 2000 en 2002).

Een andere populaire HDL, afgekort *HDL* is *Verilog*. Verilog was oorspronkelijk een commercieel product dat in een later stadium vrijgegeven is voor gebruik door meerdere partijen. Vergeleken met VHDL is Verilog vrij beperkt. De syntaxis van Verilog is compacter dan VHDL, maar staat door zijn zwakke typering ook meer toe, wat aanleiding kan geven tot fouten. Verilog is in 1995 door de IEEE gestandaardiseerd (en later nog eens in 2001). De in 2005 gestandaardiseerde taal *SystemVerilog* voorziet in veel uitbreidingen op Verilog, vooral op het gebied van *verificatie*.

Een derde taal die genoemd moet worden, is *SystemC*. De taal is het resultaat van het initiatief van een aantal bedrijven in de ontwerpautomatisering om een open-source simulator te ontwikkelen gebaseerd op *C++*. De taal is vooral bedoeld voor modellering op abstractieniveaus boven RTL. Standaardisatie door de IEEE vond plaats in 2005.

Later in deze paragraaf zal nader worden ingegaan op de drie talen, waarbij VHDL uitgebreide aandacht zal krijgen. Eerst wordt hieronder in algemene zin ingegaan op simulatie en synthese.

4.2 Simulatie en synthese

HDL's leggen op een formele en dus eenduidige manier het gedrag van hardware vast. Deze vorm van specificeren is lange tijd vooral van belang geweest om de in een HDL vastgelegde hardwaremodellen met behulp van simulatie te kunnen verifiëren (zie ook paragraaf 5). Pas later zijn er CAD-gereedschappen beschikbaar gekomen om de in een HDL op RT-niveau beschreven hardware automatisch te synthetiseren.

Zowel VHDL als Verilog zijn ontworpen in een tijd waarin logische synthese nog geen gemeengoed was. In beide talen treft men constructies aan voor het beschrijven van typische simulatieconcepten. Het toekennen van een nieuwe waarde aan een signaal na een bepaalde vertraging is een voorbeeld van zo'n constructie. Voor synthese is zo'n vertraging niet relevant aangezien de vertragingen

het gevolg zijn van de implementatie en het niet zo is dat de opgegeven vertraging precies in de implementatie gerealiseerd hoeft te worden.

Op het moment dat commerciële producten voor RTL-synthese op de markt kwamen, lag het voor de hand de al bekende talen VHDL of Verilog als invoertaal te kiezen. Dat deed men door deelverzamelingen van de mogelijke taalconstructies aan te wijzen die in hardware konden worden omgezet, de zogenaamde *synthetiseerbare subset* van de syntaxis. Zowel voor VHDL als voor Verilog zijn synthetiseerbare subsets door de IEEE gestandaardiseerd.

De voorbeelden van hardwarebeschrijvingen die verderop in deze tekst worden gegeven zijn synthetiseerbaar. De niet-synthetiseerbare taalconstructies worden in de huidige praktijk bijvoorbeeld toegepast voor simulatie na synthese *(postsynthese simulatie)*. Na synthese bestaat de beschrijving uit instanties van standaardcellen met hun onderlinge verbindingen. In deze situatie is het juist van groot belang dat vertragingen in signaalpropagatie nauwkeurig gemodelleerd worden.

Een tweede belangrijke toepassing van niet-synthetiseerbare taalconstructies is de modellering van de omgeving van de te simuleren hardware, de zogenaamde *testbench*. In paragraaf 5 wordt hier dieper op ingegaan.

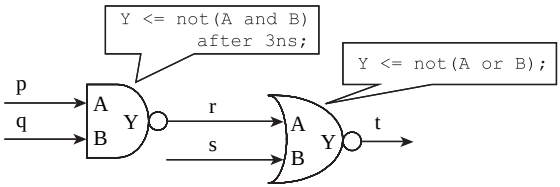
4.3 De werking van de simulator

In hoofdlijnen verschillen de simulatoren van VHDL, Verilog of SystemC niet van elkaar. De onderliggende techniek is *event-driven simulatie*. Hierbij wordt de parallelle natuur van hardware gemodelleerd door een veelvoud aan processen die elk sequentieel worden uitgevoerd. Afhankelijk van het abstractieniveau kan een proces een enkele NAND-poort of een complete ALU modelleren. Signalen maken het mogelijk dat processen met elkaar communiceren. Een signaal kan een ingang voor een proces zijn; men zegt dan dat het proces het signaal leest. Op basis van de waarden van zijn ingangen berekent een proces zijn uitgangssignalen. Als een proces een waarde aan zijn uitgangssignaal toekent, zegt men dat het zijn uitgangsignaal schrijft.

Hoewel hardware van nature parallel is, wordt hij doorgaans op een sequentiële machine gesimuleerd. Op de een of andere manier moeten dus processen die in werkelijkheid tegelijkertijd actief zijn, en signalen die tegelijkertijd van waarde veranderen, zodanig worden afgehandeld dat het verschil met de werkelijkheid zo klein mogelijk is. Dit wordt bereikt door een simulatie gebaseerd op *events*. Een event is gedefinieerd als de combinatie van een signaalverandering en een tijdstip waarop de verandering optreedt. De simulator houdt een lijst van events bij, gesorteerd naar oplopende tijd: de *event queue*. Simuleren komt dan neer op het afhandelen van events uit de event queue en het zonodig toevoegen van nieuwe events.

Een proces heeft een *sensitivity list*, een lijst van signalen waar het gevoelig voor is. Als een simulator een event voor een bepaald signaal afhandelt, zal het de processen die het signaal in hun sensitivity list hebben activeren. Als deze activering leidt tot een signaalverandering van de uitgang van het proces, worden er nieuwe events aangemaakt en op de juiste plaats in de *event list* geplaatst. Om de simulatie uitvoerbaar te maken moeten zulke events altijd op een later tijdstip worden afgehandeld dan het event dat ze veroorzaakt heeft. Het tijdverschil zal de vertraging zijn die in het model gegeven is. Als zo'n vertraging niet is gespecificeerd, wordt een oneindig kleine vertraging verondersteld, een zogenaamde *delta delay.*

Het idee van event-driven simulatie wordt geïllustreerd in figuur 4.1. De schakeling bestaat uit een NAND-poort met een inwendige vertraging van 3 ns en een NOR-poort waarvoor geen inwendige vertraging is gespecificeerd. De simulator zal deze poort met een delta delay simuleren. In de onderste helft van de figuur zijn de waarden van de signalen en de inhoud van de event queue Q voor opvolgende waarden van de simulatortijd τ (in ns) weergegeven. De beschrijving van elk element van Q bestaat uit achtereenvolgens de signaalnaam, de signaalverandering en het tijdstip van verandering. Men kan onder andere zien dat de verandering van p van een 0 naar een 1 op tijd 1, leidt tot een verandering van r van 1 naar 0 op tijd 4. Het event op s op tijd 2 heeft geen verandering van de uitgang tot gevolg en genereert geen nieuw event. Signaal t verandert een delta delay later dan signaal r.



τ = 0: p = 0; q = 1; r = 1; s = 1; t = 0; Q = { (p: 0→1@1), (s: 1→0@2) }

τ = 1: p = 1; q = 1; r = 1; s = 1; t = 0; Q = { (s: 1→0@2), (r: 1→0@4) }

τ = 2: p = 1; q = 1; r = 1; s = 0; t = 0; Q = { (r: 1→0@4) }

τ = 4: p = 1; q = 1; r = 0; s = 0; t = 0; Q = { (t: 0→1@4+Δ) }

τ = 4+Δ: p = 1; q = 1; r = 0; s = 0; t = 1; Q = {}

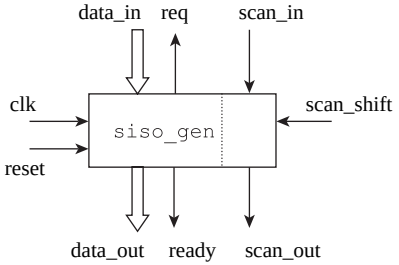
Figuur 4.1 Event-driven simulatie van een kleine schakeling

4.4 VHDL

VHDL is zowel syntactisch als semantisch een uitgebreide taal. Hieronder worden de belangrijkste concepten van VHDL gepresenteerd zonder te streven naar volledigheid.

Top-down ontwerpen kenmerkt zich door het stapsgewijs toevoegen van meer detail aan hetzelfde ontwerp (zie ook paragraaf 2.2). Gedurende deze ontwerpstappen zal het interface (in- en uitgangssignalen) van een schakeling met de buitenwereld doorgaans niet veranderen. Een goed voorbeeld van een ontwerpstap waarbij het interface niet verandert is *RTL-synthese*. De twee versies van een schakeling voor en na synthese zijn beide in VHDL te beschrijven.

Om deze reden scheidt VHDL de declaratie van het interface van een schakeling van zijn inhoud. Een schakeling heet in VHDL een *entiteit* (Eng. *entity*). Een *entiteitdeclaratie* bevat informatie over de naam van de schakeling, de in- en uitgangssignalen en eventuele generieke parameters maar zegt niets over de invulling van de schakeling.



Figuur 4.2 De in- en uitgangssignalen van de schakeling `siso_gen`

Een voorbeeldschakeling met de naam `siso_gen` is uitgekozen om de diverse concepten van VHDL in deze paragraaf te illustreren (de naam is afgeleid van *serial in serial out generic*, omdat data serieel in- en uitgaat en de beschrijving een generieke woordlengte gebruikt). Zijn in- en uitgangssignalen zijn grafisch weergegeven in figuur 4.2. De geheugenelementen van de schakeling klokken hun data in op de opgaande flank van het kloksignaal `clk`. Deze geheugenelementen worden gereset door het signaal `reset`. De schakeling haalt zijn gegevens serieel binnen via het ingangssignaal `data_in`. Als zij nieuwe data nodig heeft maakt zij op een bepaalde opgaande klokflank het signaal `req` hoog en gaat ervan uit dat de omgeving snel genoeg is om voor de volgende opgaande klokflank een nieuwe waarde voor `data_in` klaar te zetten die ingeklokt kan worden. De resultaten `siso_gen` worden serieel naar buiten gebracht via uitgangssignaal `data_out`. Als de waarde van `data_out` geldig is, wordt het signaal `ready` hooggemaakt en gaat de schakeling ervan uit dat de omgeving de data binnen een klokperiode overneemt. De signalen beginnend met 'scan' die aan de rechterkant zijn getekend, hebben te maken met het testen van de schakeling. Zij zullen in paragraaf 6 nader worden toegelicht.

De entiteitdeclaratie van `siso_gen` is te zien in figuur 4.3. Alle informatie die in VHDL aan een CAD-systeem wordt aangeboden, wordt na compilatie in een *library* opgeslagen. Het concept van libraries stelt ontwerpers in staat hun ontwerpgegevens te ordenen,

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

entity siso_gen is
   generic (word_length: natural);
   port (data_in: in std_logic_vector(word_length-1 downto 0);
         clk: in std_logic;
         reset: in std_logic;

         req: out std_logic;
         data_out: out std_logic_vector(word_length-1 downto 0);
         ready: out std_logic;

         -- scan-chain interface
         scan_in, scan_shift: in std_logic;
         scan_out: out std_logic);
end siso_gen;
```

Figuur 4.3 Entiteitdeclaratie van de schakeling `siso_gen`

op een overzichtelijke manier van elkaars gegevens gebruik te maken en (deel)ontwerpen op te slaan voor later hergebruik. De ontwerper kan in zijn VHDL-code aangeven dat hij gegevens uit andere libraries wil gebruiken. In dit voorbeeld wordt de typedefinitie `std_logic`, die in het *package* `std_logic_1164` van de library `ieee` is gedefinieerd, gebruikt.

In het algemeen bevat een package definities van datatypes, procedures en functies die uit bepaalde overwegingen samengenomen zijn. Het package `std_logic_1164` bevat een door de IEEE gestandaardiseerde negenwaardig datatype voor de modellering van logische signalen en functies die op deze signalen opereren. Een signaal van dit type kan naast de 'normale' waarden `'0'` en `'1'` (voor 'sterke' binaire signalen), o.a. `'X'` voor een onbekend signaal en `'U'` voor een ongeïnitialiseerd signaal als waarde aannemen. Het feit dat `std_logic` in een package gedefinieerd wordt betekent dat

het geen ingebouwd datatype van VHDL is. De taal VHDL heeft een beperkt aantal ingebouwde datatypes (bijv. voor gehele getallen, drijvende-komma getallen en tijd). Daar staat tegenover dat VHDL een krachtig mechanisme heeft om nieuwe datatypes te definiëren. Men kan onder andere door enumeratie en arrayvorming van bestaande datatypes nieuwe datatypes vormen.

Na declaratie van de gebruikte library en package volgt de eigenlijke declaratie van de entiteit. De body van een entiteit declareert eerst de zogenaamde generic parameters. *Generics* stellen de gebruiker in staat met één declaratie een hele klasse van modellen te definiëren. In een latere fase van het ontwerp wordt een specifiek element uit de klasse gekozen door generics een concrete waarde toe te kennen. In dit voorbeeld is de woordbreedte van de datasignalen een generic. Een veelvoorkomende toepassing van generics zijn vertragingstijden in modellen van standaardcellen. Vertragingstijden van verschillende instanties van dezelfde cel zullen afhangen van de belasting van de uitgangen. Ze worden bepaald na back annotation (zie paragraaf 3.3).

Na de generics volgen de *poorten* van de schakeling, de in- en uitgangssignalen gegroepeerd met het trefwoord 'port' . Al deze signalen behalve `data_in` en `data_out` zijn één bit breed en hebben het eerder genoemde datatype `std_logic`. `data_in` en `data_out` hebben een breedte gegeven door de generic `word_length` en zijn van het type `std_logic_vector` (dit datatype wordt ook in het pakket `std_logic_1164` gedefinieerd).

VHDL gebruikt de term *architectuur* (Eng. *architecture*) voor de beschrijving van de inhoud van een entiteit. Aan een entiteit kunnen meerdere architecturen die gedrag of structuur beschrijven gekoppeld worden, ieder met een eigen naam. Figuur 4.4 geeft een gedragsbeschrijving genaamd `copy` voor de entiteit `siso_gen`. Met deze invulling zal `siso_gen` na reset op elke opgaande klokflank de via `data_in` komende data inklokken en via de `data_out` uitgang naar buiten brengen.

Gedrag wordt in VHDL aangegeven door middel van een proces (zie paragraaf 4.3), waarvan een architectuur er meer dan een kan bevatten. In dit voorbeeld is er het proces `seq` die de sequentiële logica beschrijft. De sensitivity list bevat de signalen `clk` en `reset`,

```
architecture copy of siso_gen is
begin

  seq: process(clk, reset)
  begin
    if (reset = '1')
    then
      data_out < = (others => '0');
      ready < = '0';
    elsif rising_edge(clk)
    then
      data_out < = data_in;
      ready < = '1';
    end if;
  end process seq;

  req < = '1';
end copy;
```

Figuur 4.4 Declaratie van de architectuur `copy` van de entiteit `siso_gen`

wat wil zeggen dat de signalen die door dit proces worden geschreven, namelijk `data_out` en `ready`, alleen kunnen veranderen als `clk` of `reset` verandert. In het if-statement wordt eerst getest op de waarde van `reset`; als deze `'1'` is, worden alle uitgangsbits op `'0'` gezet. Dit modelleert een asynchrone reset (de reset heeft effect ongeacht de klok). Als `reset` niet actief is, wordt data op de opgaande klokflank gekopieerd van `data_in` naar `data_out`. Dit modelleert een *positive edge-triggered flipflop* (zie paragraaf 3.1). Programmatuur voor RTL-synthese zal dan ook uit deze syntaxis opmaken dat de signalen `data_out` en `ready` uitgangen van flipflops moeten zijn. Merk op dat `data_in` niet in de sensitivity list van het proces voorkomt, overeenkomstig het feit dat een flipflop niet reageert op veranderingen van ingangsdata zolang er geen opgaande klokflank is.

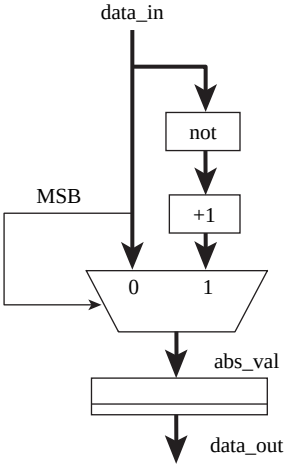
Het laatste statement in de architectuur maakt geen deel uit van een proces. Zo'n los statement heet een *concurrent assignment* en wordt behandeld als een impliciet proces met de signalen in het rechterlid in een impliciete sensitivity list (in dit geval zijn er geen signalen in het rechterlid).

De combinatie van de entiteit `siso_gen` met zijn architectuur `copy` is synthetiseerbaar mits er een waarde voor de woordlengte wordt gespecificeerd. Voor de herbruikbaarheid van de code is het in het algemeen verstandig parameters als generics te specificeren. Zo zijn generics zeer geschikt voor het beschrijven van IP-blokken in VHDL (zie paragraaf 1).

Een tweede voorbeeldarchitectuur voor de entiteit `siso_gen`, met de naam `absolute`, voegt de berekening van de absolute waarde van de ingang toe aan de functionaliteit van `copy`. Een datapad die deze berekening kan uitvoeren, is te zien in figuur 4.5. Uitgangspunt is dat getallen in de datastroom als 2's complement gecodeerd zijn. Bij die codering zal het meest significante bit (MSB) `'0'` zijn voor een positief getal en `'1'` voor een negatief getal. Het MSB wordt daarom gebruikt als stuursignaal van een multiplexer. Het vermenigvuldigen met -1 (nodig om van een negatief getal een positief getal te maken) is geïmplementeerd door het getal bitsgewijs te inverteren en er vervolgens 1 bij op te tellen. Omdat de absolute waarde van het grootste negatieve getal in de 2's complement codering niet in hetzelfde aantal bits past, werkt de schakeling niet voor dat getal.

De VHDL-code voor de architectuur `absolute` is te vinden in figuur 4.6. Deze architectuur heeft het package `numeric_std` uit de library `ieee` nodig. Hierin zijn onder andere de datatypes `signed` en `unsigned` gedeclareerd samen met functies die op deze datatypes opereren. Deze datatypes maken het mogelijk om bitpatronen uit een `std_logic_vector` te interpreteren als een getal met of zonder teken.

Het proces `combi` is een combinatorisch proces dat de absolute waarde van de ingang `data_in` berekent. `data_in` komt in zijn sensitivity list voor, wat inhoudt dat het proces bij elke verandering van dit signaal steeds opnieuw doorlopen wordt. Het datapad van figuur 4.5 is direct terug te vinden in de code. Het if-statement komt



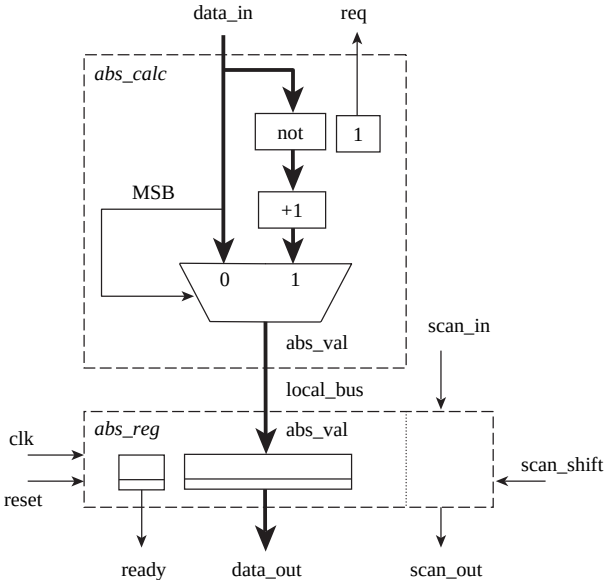
Figuur 4.5 Datapad voor het berekenen van absolute waardes in de entiteit `siso_gen`

overeen met de multiplexer. In het algemeen zal een synthesegereedschap aparte hardware genereren voor de then- en else-tak van het if-statement (in dit voorbeeld vertaalt de else-tak zich in slechts een draad omdat er geen berekening in voorkomt). Het register dat in het datapad op de multiplexer volgt is beschreven in proces `seq`, op een identieke wijze als in architectuur `copy`.

Niet zozeer omdat het een goede ontwerpbeslissing is, maar om te laten zien hoe structuur in VHDL beschreven wordt, is het zojuist besproken voorbeeld opgedeeld in twee deelschakelingen volgens het schema van figuur 4.7. De twee processen uit figuur 4.6 zijn nu min of meer ondergebracht in twee aparte entiteiten. De declaraties van deze entiteiten zijn te vinden in figuur 4.8. De architecturen zijn niet in deze tekst opgenomen.

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

architecture absolute of siso_gen is
  signal abs_val: unsigned(word_length-1 downto 0);
begin

  combi: process(data_in)
  begin
    if (data_in(word_length-1) = '1')
    then
       abs_val < = unsigned(not data_in) + 1;
    else
      abs_val < = unsigned(data_in);
    end if;
  end process combi;

  seq: process(clk, reset)
  begin
    if (reset = '1')
    then
      data_out < = (others => '0');
      ready < = '0';
    elsif rising_edge(clk)
    then
      data_out < = std_logic_vector(abs_val);
      ready < = '1';
    end if;
  end process seq;

  req < = '1';
end absolute;
```

Figuur 4.6 Declaratie van de architectuur `absolute` *van de entiteit* `siso_gen`



Figuur 4.7 De uit twee deelschakelingen samengestelde `siso_gen` *architectuur voor het berekenen van absolute waardes*

De VHDL-syntaxis die nodig is voor het beschrijven van structuur, wordt geïllustreerd in figuur 4.9. Net als een gedragsbeschrijving wordt structuur beschreven door een architectuur, in dit geval de architectuur `struct_abs` van de entiteit `siso_gen`. De te instantiëren entiteiten worden lokaal gedeclareerd als componenten. Doordat de interfaces op deze manier bekend zijn, kan de compiler controleren of de aansluitingen op de instanties verderop consistent zijn. Bovendien kan de architectuur los gecompileerd worden, d.w.z. zonder dat de entiteiten van de deelschakelingen vooraf gecompileerd hoeven te zijn.

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

entity abs_calc is
   generic (word_length: integer := 16);
   port (data_in: in std_logic_vector(word_length-1 downto 0);
         req: out std_logic;
         abs_val: out std_logic_vector(word_length-1 downto 0));
end abs_calc;

library ieee;
use ieee.std_logic_1164.all;

entity abs_reg is
   generic (word_length: integer := 16);
   port (abs_val: in std_logic_vector(word_length-1 downto 0);
         clk: in std_logic;
         reset: in std_logic;

         data_out: out std_logic_vector(word_length-1 downto 0);
         ready: out std_logic;

         scan_in, scan_shift: in std_logic;
         scan_out: out std_logic);
end abs_reg;
```

Figuur 4.8 VHDL entiteitdeclaraties voor de twee deelschakelingen van figuur 4.7

Bij het instantiëren (tussen de trefwoorden `begin` en `end` van de architectuurdeclaratie), krijgt elke instantie een naam; in het voorbeeld: `ac` en `ar`. Daarna wordt met een `port map` elk lokaal signaal van de instantie gekoppeld aan een signaal van de instantiërende architectuur. In het voorbeeld zijn de lokale signaalnamen in de meeste gevallen gelijk aan de signaalnamen die erop aansluiten. De

```
architecture struct_abs of siso_gen is
  component abs_calc
    generic (word_length: integer := 16);
    port (data_in: in std_logic_vector(word_length-1 downto 0);
          req: out std_logic;
          abs_val: out std_logic_vector(word_length-1 downto 0));
  end component;
  component abs_reg
    generic (word_length: integer := 16);
    port (abs_val: in std_logic_vector(word_length-1 downto 0);
          clk: in std_logic;
          reset: in std_logic;

          data_out: out std_logic_vector(word_length-1 downto 0);
          ready: out std_logic;

          scan_in, scan_shift: in std_logic;
          scan_out: out std_logic);
  end component;
  signal local_bus: std_logic_vector(word_length-1 downto 0);
begin
  ac: abs_calc
      generic map (word_length => word_length)
      port map (data_in => data_in, req => req,
                abs_val => local_bus);
  ar: abs_reg
      generic map (word_length => word_length)
      port map (abs_val => local_bus, clk => clk, reset => reset,
                data_out => data_out, ready => ready,
                scan_in => scan_in, scan_shift => scan_shift,
                scan_out => scan_out);
end struct_abs;
```

Figuur 4.9 Structurele beschrijving in VHDL van de samenstelling van twee deelschakelingen uit figuur 4.7

uitzondering is het signaal `local_bus` dat van de ene deelschakeling naar de andere deelschakeling loopt en in de deelschakelingen zelf `abs_val` heet. Tijdens instantiatie kan m.b.v. een `generic map` een generic van een instantie een waarde krijgen. In het voorbeeld wordt de generic `word_length` van `siso_gen` doorgegeven aan de instanties.

4.5 (System)Verilog

De oorspronkelijke versie van Verilog (gestandaardiseerd in 1995) heeft grote beperkingen vergeleken met VHDL. De belangrijkste beperkingen zijn:

– Verilog kent geen scheiding tussen een entiteit en een architectuur. De interface en inhoud van een schakeling worden tegelijk beschreven als een module. Het concept van een configuratie die in VHDL een koppeling aanbrengt tussen de entiteiten in een hiërarchisch model en de gekozen architecturen ten behoeve van een simulatie (te bespreken in paragraaf 5.1) ontbreekt ook. In Verilog is voor elke combinatie van entiteit en architectuur een unieke modulenaam nodig. Het beheer van modules die betrekking hebben op dezelfde schakeling (bijv. voor en na synthese) moet buiten de taal om gebeuren.
– Verilog heeft geen sterke datatypering. Dat wil zeggen dat de compiler minder streng is bij toekenningen (*assignments*) tussen signalen van verschillende datatypes. Bepaalde soorten signalen (*wires*) hoeven niet voor gebruik gedeclareerd te worden, wat aanleiding kan geven tot fouten (gelijke signalen worden verschillend als gevolg van een typefout).
– Verilog heeft niet de mogelijkheid nieuwe datatypes te definiëren. Waar in VHDL datatypes en erop opererende functies los van de taal kunnen worden ontwikkeld en gestandaardiseerd en vervolgens via een package beschikbaar worden gesteld (`std_logic_1164` en `numeric_std` zijn hier voorbeelden van), heeft Verilog een uitbreiding van de taal zelf nodig bij de introductie van nieuwe datatypes. Zo is er bij de herziening van de Verilog-standaard in 2001 het trefwoord `signed` toegevoegd om in bepaalde situaties duidelijk te maken dat bitpatronen als 2's complement moesten worden geïnterpreteerd.

Ondanks zijn nadelen, is Verilog altijd een populaire taal geweest. De taal kwam ongeveer tegelijk met VHDL beschikbaar, werd ondersteund door de grote producenten van CAD-software en veroverde zo een aanzienlijk marktaandeel. Daarnaast had Verilog het voordeel dat het minder complex was dan VHDL en dus laagdrempeliger was. Zo ontstond een grote hoeveelheid IP (zie paragraaf 1) in Verilog. Dit creëerde op zijn beurt de behoefte om de taal Verilog verder te ontwikkelen.

Deze behoefte resulteerde in de taal SystemVerilog. SystemVerilog is compatibel met Verilog, opdat ontwerpgereedschappen voor SystemVerilog ook overweg kunnen met bestaande Verilog IP. De uitbreidingen zijn tweeledig:

1. De taal als ontwerptaal, dus als invoertaal voor synthese, is significant uitgebreid. Zo is het in SystemVerilog mogelijk nieuwe datatypes te definiëren.
2. Er zijn taalconstructies toegevoegd die rechtstreeks betrekking hebben op het ontwerpen op RT-niveau. Dit laatste wordt geïllustreerd in figuur 4.10. De getoonde code beschrijft de reeds in VHDL gepresenteerde schakeling `siso_gen` met een gedrag waarin de absolute waarde van de ingang naar de uitgang wordt gekopieerd.

Hoewel SystemVerilog een soortgelijk concept heeft als de generic in VHDL, is in dit voorbeeld een vaste woordlengte van 5 bits gekozen. Processen in Verilog worden beschreven met behulp van een `always` constructie waarbij ook een sensitivity list opgegeven kan worden. SystemVerilog voegt hier een aantal gespecialiseerde varianten aan toe.

– Met `always_comb` wordt combinatorische logica beschreven. Omdat combinatorische logica direct moet reageren op elke verandering van een ingangssignaal, houdt dit in dat elk signaal dat in het proces wordt gelezen ook in de sensitivity list moet zitten.
– In een `always_comb` constructie hoeft de gebruiker de sensitivity list niet zelf op te geven, maar wordt deze automatisch door de compiler geconstrueerd. Omdat de gebruiker bovendien expliciet aangeeft combinatorische logica te beschrijven, kan de compiler controleren of het proces echt combinatorisch is (elk

```
module siso_gen (input [4:0] data_in, input clk, input reset,
                 output logic req, output logic [4:0] data_out,
                 output logic ready,
                 input scan_in, input scan_shift, output logic scan_out);

  logic[4:0] abs_val;

  always_comb begin
    if (data_in[4] == 1'b1)
      abs_val < = ~data_in + 1;
    else
      abs_val < = data_in;
  end

  assign req = 1'b1;

  always_ff @ (posedge clk, posedge reset)
    if (reset == 1'b1) begin
      data_out < = 5'b0;
      ready < = 1'b0;
    end
    else begin
      data_out < = abs_val;
      ready < = 1'b1;
    end
endmodule
```

Figuur 4.10 Beschrijving in SystemVerilog van de berekening van de absolute waarde conform het interface van de VHDL-entiteit `siso_gen`

signaal moet onder alle voorwaarden een waarde krijgen en er mag geen terugkoppeling van uitgang naar ingang zijn).

– Met `always_ff` geeft de gebruiker juist aan dat de signalen waaraan een toekenning plaatsvindt, in een flipflop moeten worden

opgeslagen. In de sensitivity list mogen alleen de klok- en reset-signalen staan; met `posedge` respectievelijk `neg-edge` wordt gevoeligheid voor opgaande dan wel neergaande flank opgegeven.

De tweede uitbreiding van SystemVerilog ten opzichte van Verilog betreft het gebruik van Verilog als *verificatietaal*. Zoals in paragraaf 5 zal worden aangegeven is er in recente jaren grote vooruitgang geboekt op het gebied van verificatie. Vele aspecten van verificatie (random constrained patterns, code coverage, assertions) worden rechtstreeks vanuit de taal ondersteund.

4.6 SystemC en ontwerpen op systeemniveau

De schakelingen die op een IC werden gerealiseerd zijn in de loop der jaren steeds complexer geworden. Steeds vaker is op deze schakelingen de term *system on chip* (*SoC*) van toepassing waarmee wordt bedoeld dat het IC een volwaardig elektronisch systeem is geworden met een omzetter van analoge externe signalen naar het digitale domein, signaalbewerking in dat digitale domein en, ten slotte, omzetting van het bewerkte digitale signaal in een analoog signaal. Een SoC zal doorgaans ook een of meer processors bevatten die door software worden aangestuurd.

Een belangrijke ontwerpoverweging is of een bepaalde functionaliteit in hardware dan wel software zal worden gerealiseerd. Functies die door hun realisatie in software het systeem te traag maken moeten worden geïdentificeerd en vervolgens een hardwarerealisatie krijgen. Het ontwerpprobleem waarbij over de opdeling van het systeem in hardware en software moet worden beslist heet *hardware-software co-design* of kortweg '*co-design*'. De term *ESL* (*electronic system level*) is recentelijk populair geworden bij verwijzingen naar het ontwerpen op systeemniveau.

Een probleem bij co-design is dat van elke potentiële oplossing de prestaties geëvalueerd moeten worden. Bij de analyse van de prestatie gaat het niet alleen om het aantal klokcycli dat voor een bepaalde taak nodig is, maar ook om de vraag of de communicatie tussen deelschakelingen adequaat is en of bijvoorbeeld een bepaalde bus niet

overbelast is. *Systeemsimulaties* zijn een voor de hand liggend middel om de prestatie te evalueren. Daartoe kan men het hele systeem, inclusief de processors, in de traditionele HDL's VHDL en/of Verilog modelleren. De gecompileerde software is dan in het systeem beschikbaar via de eveneens gemodelleerde geheugens.

Een nadeel van de zojuist beschreven aanpak voor systeemsimulaties is dat er te veel detail meegenomen wordt, dat het abstractieniveau van modellering te laag is waardoor er maar een beperkt aantal klokcycli in een redelijke tijd gesimuleerd kan worden. Bovendien moet van elke deelschakeling een beschrijving op RT-niveau beschikbaar zijn voordat het hele systeem gesimuleerd kan worden.

Veel snellere simulatie is mogelijk door de hardware op een hoog niveau in de softwareomgeving te modelleren in plaats van de gecompileerde software in een hardwaresimulatie te executeren. De behoefte aan hoog-niveau hardwaremodellering en het feit dat C/C++ de meest gebruikte programmeertaal is, heeft geleid tot de taal SystemC. Het initiatief kwam rond het jaar 2000 van een aantal producenten van ontwerpsoftware, normaliter elkaars concurrenten, en had tot doel een standaard te creëren voor systeemsimulaties gebaseerd op een bibliotheek van open-source C++ code. In 2005 was goedkeuring van de standaard door de IEEE een feit.

De open-source distributie van SystemC legt de taal vast door de definitie van datatypes voor modules, signalen, enz. Ook een event-driven simulator maakt deel uit van de distributie. Iedereen met een C++-compiler kan dan ook een systeem in SystemC modelleren en simuleren. Het betreft dan wel een kale, op tekstbestanden gebaseerde simulatie. Gereedschappen met grafische gebruikers-interfaces zijn commercieel verkrijgbaar bij producenten van ontwerpsoftware. *Automatische synthese* vanuit SystemC is geen primair doel. Er bestaan echter wel commerciële en niet-commerciële partijen die synthesegereedschappen voor SystemC hebben ontwikkeld.

Omdat SystemC net als VHDL en Verilog gebaseerd is op een event-driven simulator, zou SystemC in principe de traditionele HDL's geheel kunnen vervangen. Dat is echter niet de bedoeling. SystemC richt zich op het RT-niveau en hogere abstractieniveaus. Van hard-

ware die op het RT-niveau is beschreven is per klokcyclus bekend wat erin wordt berekend. Men spreekt van een *clock-cycle true modellering*. Op een iets hoger abstractieniveau kan men modelleren dat een berekening een bepaalde tijd kost zonder de berekening over klokcycli te verdelen. Dit heet *timed modellering*. Bij een *untimed* of *data-flow modellering* is de tijd uit de beschrijving verdwenen en wordt alleen aangegeven dat een bepaalde berekening het resultaat van een andere berekening nodig heeft en dus moet wachten tot dat resultaat beschikbaar is. Bij een dergelijke modellering kan men denken aan het gebruik van *first-in first-out (FIFO) buffers* voor de communicatie tussen deelschakelingen.

Om een indruk te geven van een hardwarebeschrijving in SystemC, wordt hierna ingegaan op de SystemC-versie van de reeds voor VHDL en SystemVerilog gepresenteerde schakeling `siso_gen` waarin de absolute waarde van een datastroom wordt berekend. De beschrijving is verdeeld over figuur 4.11 en figuur 4.12.

Een module in SystemC is een klasse afgeleid van de interne, voor de gebruikers minder relevante, klasse `sc_module`. Het trefwoord `SC_MODULE` in figuur 4.11 is dan ook een C++ macro die de klasse `siso_gen` declareert. De rest van de figuur somt de onderdelen op waaruit de klasse bestaat. Als eerste worden het kloksignaal (`sc_in_clk`), de overige ingangen (`sc_in`), uitgangen (`sc_out`) en lokale signalen (`sc_signal`) met hun datatypes gedeclareerd. De in dit voorbeeld voorkomende datatypes zijn `sc_logic` voor enkelbits logische signalen, vergelijkbaar met `std_logic` in VHDL, `sc_lv` voor vectoren van het type `sc_logic`, en `sc_int` voor vectoren met een 2's complement interpretatie.

Vervolgens worden de lidfuncties `process_clock` en `process_inputs` gedeclareerd. Zij zullen respectievelijk worden gebruikt voor de sequentiële en combinatorische processen die samen het gedrag van de schakeling beschrijven. Met de macro `SC_CTOR` wordt de *constructor* voor de module gedeclareerd. De constructor moet de lidfuncties als processen aanmelden bij de simulator en daarbij ook hun sensitivity list registreren. `SC_METHOD` is een van de drie soorten processen die SystemC kent, namelijk een proces die van begin tot eind wordt doorlopen zodra een van de signalen in zijn sensitivity list is veranderd. Gevoeligheid voor de opgaande flank

```
SC_MODULE (siso_gen) {
    sc_in_clk clk;
    sc_in< sc_logic>  reset;
    sc_in< sc_lv< 5> >  data_in;
    sc_out< sc_logic>  req;
    sc_out< sc_lv< 5> >  data_out;
    sc_out< sc_logic>  ready;

    sc_in< sc_logic>  scan_in;
    sc_in< sc_logic>  scan_shift;
    sc_out< sc_logic>  scan_out;

    sc_signal< sc_int< 5> >  abs_val;

    void process_clock();
    void process_input();

    SC_CTOR (siso_gen) {
        SC_METHOD (process_clock);
            sensitive_pos <<  clk;
            sensitive_neg <<  reset;

        SC_METHOD (process_input);
            sensitive <<  data_in;
    }
};
```

Figuur 4.11 Declaratie van de module `siso_gen` in SystemC

van het signaal wordt aangegeven met `sensitive_pos` en voor de neergaande met `sensitive_neg`. Gevoeligheid voor een willekeurige verandering wordt aangegeven met `sensitive`.

De inhoud van de twee processen uit het voorbeeld is te vinden in figuur 4.12. De code heeft nauwelijks enige toelichting nodig.

```
void siso_gen::process_input() {
    sc_lv< 5>  local_in = data_in.read();
    sc_int< 5>  local_out;
    sc_logic sign = local_in[4];
    if (sign == '1')
      local_out = (sc_int< 5> )(~local_in) + 1;
    else
       local_out = local_in;
    abs_val.write(local_out);
}

void siso_gen::process_clock() {
   sc_logic reset_val = reset.read();
   sc_logic ready_val, req_val;
   sc_lv< 5>  data_out_val;
   if (reset_val == '1') {
     data_out_val = 0;
     ready_val = '0';
   }
   else {
     data_out_val = abs_val.read();
     ready_val = '1';
   }
   req_val = '1';

   data_out.write(data_out_val);
   ready.write(ready_val);
   req.write(req_val);
}
```

Figuur 4.12 De lidfuncties behorende bij de System-module `siso_gen`

Hoewel het in dit geval niet nodig is, is in de code ervoor gezorgd dat elk ingangssignaal in een proces maar één keer wordt gelezen met de functie `read`. Elk uitgangssignaal wordt één keer geschreven met de functie `write`.

Een signaal in SystemC is een bepaalde vorm van een communicatiekanaal. Een ander voorbeeld van een communicatiekanaal is een *FIFO-buffer*. Bij een FIFO-buffer zal elke aanroep van `read` een volgend element uit de buffer ophalen en is het dus van belang de functie niet meervoudig aan te roepen als slechts één element moet worden opgehaald.

Naast het modelleren van hardware biedt SystemC ook middelen voor geavanceerde verificatie. Zo bestaat er een open-source distributie genaamd *SystemC Verification Library* (SCV) waarmee bijvoorbeeld *constrained random patterns* gegenereerd kunnen worden (zie ook paragraaf 5.7).

5 Verificatie

Gezien de hoge kosten die gemoeid zijn met de productie van een IC (een verzameling belichtingmaskers voor het maken van een IC kost tienduizenden euro's; een moderne IC-fabriek kost miljarden), is het van groot belang dat zoveel mogelijk ontwerpfouten nog *voor* productie worden gedetecteerd. Dit proces heet *verificatie* en is het onderwerp van deze paragraaf. Verificatie komt in de praktijk vooral neer op simulatie. Vanwege de beperkingen van simulatie zijn er talloze aanvullende technieken om het verificatieproces te complementeren. Zowel simulatie als de meeste andere technieken krijgen verderop enige aandacht.

Verificatie moet niet worden verward met activiteiten die plaatsvinden na productie:

– *Evaluatie*: het nagaan van de functionele correctheid van een IC; doet het IC wat het moet doen? Evaluatie is vooral belangrijk omdat verificatie vrijwel altijd onvolledig is. Het kost te veel tijd om alles wat men tijdens verificatie zou willen doen ook daadwerkelijk uit te voeren. Omdat simulatie meerdere ordes

van grootte trager is dan het werkelijke IC, kan men, als men het IC eenmaal heeft, veel meer zaken nalopen dan in simulatie haalbaar is. Een andere reden voor de onvolledigheid van verificatie is dat de modellering niet altijd voldoende nauwkeurig is waardoor simulatie niet overeenkomt met de werkelijkheid.

– *Karakterisatie*: het bepalen van de operationele grenzen van een IC. Het gaat daarbij om zaken als beperkingen aan de klokfrequentie, timingeisen aan opeenvolgende schakelende signalen, spanningniveaus en temperatuurbereik. De uitgebreide metingen, die meerdere dagen kunnen duren, worden verricht op een klein aantal IC's. Vervolgens worden de metingen op basis van statistiek vertaald naar eigenschappen van de gehele productie.
– *Testen*: het voor elk individueel IC nalopen of het aan de specificaties voldoet. Het productieproces van IC's is niet foutvrij. Ondanks het feit dat IC's worden geproduceerd in cleanrooms waaraan heel hoge eisen worden gesteld (tot minder dan een stofdeeltje per kubieke meter) kan het gebeuren dat er een stofdeeltje op een IC belandt en een kortsluiting of onderbreking veroorzaakt. Ook kan een verkeerde concentratie van chemicaliën tijdens een van de productiestappen de betrouwbaarheid van verbindingen aantasten. Het doel van testen is om in zo weinig mogelijk tijd (seconden) na te gaan of het IC vrij is van productiefouten. In paragraaf 6 wordt uitgebreider op dit onderwerp ingegaan. Merk op dat bij het ontwikkelen van software de term 'testen' wordt gebruikt voor het achterhalen van ontwerpfouten, dus voor het equivalent van 'verificatie' bij hardware.

5.1 Klassieke simulatie

Een schakeling die door een HDL is gemodelleerd kan gesimuleerd worden door een geschikte simulator (zie paragraaf 4.3). De beschrijving van de hardware alleen is normaliter niet voldoende voor het kunnen simuleren. De hardware heeft immers interactie met zijn omgeving.

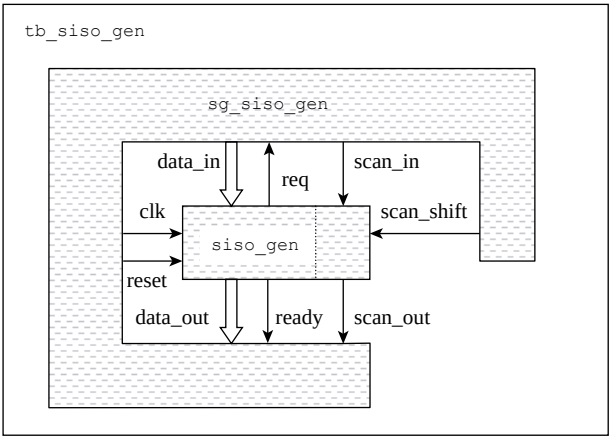
Ingangssignalen moeten op een juiste manier moeten worden aangestuurd. Signaalpatronen waarmee een hardwaremodel wordt aangestuurd worden *stimuli* genoemd. Soms zijn ingangssignalen zelfs afhankelijk van uitgangssignalen van de hardware en moeten de

uitgangssignalen worden verwerkt voordat de nieuwe waarden voor ingangssignalen bepaald kunnen worden. Een voorbeeld van zo'n situatie treedt op in de schakeling `siso_gen` uit figuur 4.2 waar signaal `req` gebruikt wordt om een nieuwe waarde voor signaal `data_in` te verzoeken.

In de huidige praktijk is het gebruikelijk naast de te verifiëren hardware, het zogenaamde *design under verification* (*DUV*), ook zijn omgeving in een HDL te modelleren. Het geheel van omgeving en DUV heet de *testbench* (de naam is enigszins ongelukkig gekozen omdat de naam aan testen doet denken terwijl het om verificatie gaat). Het voordeel van het gebruik van een gestandaardiseerde HDL voor de testbench is portabiliteit, onafhankelijkheid van een specifieke simulator.

Een testbench heeft per definitie geen in- en uitgangen. Het is vrij gebruikelijk de testbench samen te stellen uit het DUV en een complementaire stimulusgenerator. De stimulusgenerator heeft een uitgang voor elke ingang van het DUV en een ingang voor elke uitgang van het DUV. De toepassing van deze opzet op de voorbeeldschakeling `siso_gen` is in figuur 5.1 geïllustreerd. In de figuur heet de stimulusgenerator `sg_siso_gen` en de testbench `tb_siso_gen`. De stimulusgenerator zou bijvoorbeeld kunnen beginnen met een resetpuls en vervolgens op elke neergaande klokflank dat `req` hoog is, een datawoord uit een bestand kunnen lezen om het woord aan te bieden op `data_in` (als de omgeving de neergaande en het DUV op de opgaande klokflank opereren, geldt voor beide dat data stabiel is op de eigen klokflank). Elke neergaande klokflank dat `ready` hoog is, zou hij de waarde op `data_out` in een bestand kunnen schrijven, of kunnen vergelijken met een waarde uit een referentiebestand.

Vaak zal de stimulusgenerator net als het DUV hiërarchisch zijn opgebouwd; hij zou bijvoorbeeld modellen van externe geheugens kunnen bevatten. In HDL's zoals VHDL en Verilog is het niet noodzakelijk dat een beschrijving zuiver uit gedrag of zuiver uit structuur bestaat. Het is ook toegestaan om in een gedragsbeschrijving een subschakeling te instantiëren. Als er een dergelijke stijl wordt gehanteerd vallen testbench en stimulusgenerator samen en wordt alleen het DUV in de testbench geïnstantieerd.



Figuur 5.1 Opbouw van een testbench uit een DUV en een stimulusgenerator

Bij een *zelfcheckende testbench* verwerkt de stimulusgenerator de uitgangen van het DUV en is op basis daarvan in staat te beslissen of de simulatie succesvol is. Het bovengenoemde voorbeeld waarin de uitgangen van het DUV automatisch vergeleken worden met waarden in een referentie datastroom is een voorbeeld van een *zelfcheckende simulatie*. Zelfcheckende testbenches lenen zich goed voor *batchverwerking*. Bij deze vorm van verwerking wordt de simulatie uitgevoerd op een rekenserver, zonder tussentijdse ingreep van een verificatie-ingenieur.

Voor het lokaliseren van fouten, maar ook voor die gevallen waarin interne signalen van het DUV die niet zichtbaar zijn voor de stimulusgenerator geobserveerd moeten worden, biedt een simulator de mogelijkheid het verloop van de signalen in de tijd te registeren en vervolgens de zo verkregen golfvormen grafisch zichtbaar te maken. De grafische interface van een simulator komt vooral van pas bij

interactieve simulaties, waarbij de analyse van golfvormen samengaat met het herhaaldelijk stopzetten en doorstarten dan wel opnieuw opstarten van de simulatie.

VHDL, zoals behandeld in paragraaf 4.4, heeft de bijzondere eigenschap dat een entiteit meerdere architecturen kan hebben. Deze eigenschap is nuttig in de context van top-down ontwerpen en het stapsgewijs verfijnen van een ontwerp. Tegelijkertijd is er het verlies aan eenduidigheid op het moment dat er meerdere architecturen van een entiteit door een simulator zijn gecompileerd.

De gewenste eenduidigheid wordt in VHDL geleverd door de taalconstructie voor configuraties. Een *configuratie* geeft voor een schakeling op elk niveau van de hiërarchie de combinatie van entiteit en architectuur die gesimuleerd moet worden. Van de entiteit `siso_gen` uit figuur 4.3 is in figuur 4.4 de `copy` architectuur en in figuur 4.6 de `absoulute` architectuur getoond. Voor het simuleren is `siso_gen` ondergebracht in een testbench `tb_siso_gen` en gekoppeld aan een stimulusgenerator `sg_siso_gen`, zoals aangegeven in figuur 5.1. Bovendien is er een extra hiërarchisch niveau `tb_siso gen_top` aangebracht om de woordlengte tot in de testbench generiek te houden. De entiteit en architectuur van dit topniveau zijn weergegeven in figuur 5.2. Daarin is te zien hoe het

```
entity tb_siso_gen_top is
end tb_siso_gen_top;

architecture top of tb_siso_gen_top is
  component tb_siso_gen
    generic(word_length: natural := 8);
  end component;
begin
  tg: tb_siso_gen;
end top;
```

Figuur 5.2 Entiteit- en architectuurdeclaratie van een testbench topmodel

```
configuration conf_tb_siso_gen_abs of tb_siso_gen_top is
  for top
    for tg: tb_siso_gen use entity work.tb_siso_gen(structure)
          generic map (word_length => 5);
      for structure
        for duv: siso_gen use entity work.siso_gen(absolute)
            generic map (word_length => 5);
        end for;
        for sg: sg_siso_gen use entity work.sg_siso_gen(file_io)
            generic map (word_length => 5,
                         in_file_name => "abs5_in.txt",
                         out_file_name => "abs5_out.txt");
        end for;
      end for;
    end for;
  end for;
end conf_tb_siso_gen_abs;
```

Figuur 5.3 VHDL-configuratie voor de simulatie van de architectuur `absolute` *van de entiteit* `siso_gen` *met woordlengte 5*

model tb_siso_gen met een *generic* `word_length` geïnstantieerd wordt in het model `tb_siso_gen_top` en daar instantienaam `tg` krijgt.

Een configuratie voor de simulatie van de architectuur `absolute` in de zojuist beschreven opzet is te vinden in figuur 5.3. Zoals in de eerste regel te zien is krijgt een configuratie een naam (`conf_tb_siso_gen_abs`). In de body van de declaratie wordt de hiërarchie van het model doorlopen en bij elke instantiatie wordt verteld welke combinatie van entiteit en architectuur uit welke library (hier: `work`) gebruikt moet worden.

Bij een *instantiatie* kan men een waarde toekennen aan generics. In het voorbeeld gebeurt dat meerdere malen voor de woordlengte en bovendien voor de tekstbestanden waar de stimulusgenerator

gebruik van maakt. Men kan in een configuratie ook de *port map* opgeven, wat nodig is als de geïnstantieerde component andere poortnamen heeft dan in de architectuurbeschrijving gebruikt. Een configuratie hoeft niet de hele hiërarchie te doorlopen zoals in dit voorbeeld: in plaats van een combinatie van entiteit en architectuur kan men ook naar een configuratie van een submodel verwijzen die zelf vervolgens een deel van de hiërarchie specificeert.

Een veelgebruikte toepassing van configuraties is bij de simulatie van een schakeling na logische synthese en lay-out. Zoals in paragraaf 3.3 aangegeven kan een schakeling na logische synthese beschreven worden als een netwerk van standaardcellen en kan men na lay-out vrij nauwkeurig de interne en externe signaalvertragingen schatten. Deze schakeling kan in dezelfde testbench gesimuleerd worden als de schakeling voor synthese. Het volstaat een nieuwe configuratie te definiëren waarin het DUV is vervangen, maar de stimulusgenerator en verdere structuur onveranderd blijven.

Hoewel het mogelijk is een schakeling met alle informatie over de uit de lay-out bepaalde vertragingen geheel in VHDL te beschrijven, worden vertragingen in de praktijk in externe bestanden gespecificeerd. Pas tijdens simulatie wordt dan het netwerk van standaardcellen die in VHDL is beschreven gecombineerd met de vertragingen. Dit combineren komt voor VHDL neer op het toekennen van waarden aan de generics behorende bij de beschrijvingen van de standaardcellen. Gangbare, gestandaardiseerde, formaten voor de beschrijving van vertragingen zijn *SDF* (*Standard Delay Format*) en *SPEF* (*Standard Parasitic Extended Format*).

Simulatie als middel om ontwerpfouten te detecteren heeft zijn beperkingen. Dit heeft te maken met het feit dat het aantal mogelijke combinaties van waarden voor ingangssignalen en interne toestanden enorm groot is (het aantal mogelijkheden verdubbelt voor elk nieuw signaal). Het aflopen van alle mogelijkheden wordt al bij schakelingen van bescheiden omvang onhaalbaar. Door veel te simuleren en daarbij de te simuleren situaties slim te kiezen groeit wel het vertrouwen in de correctheid van een schakeling.

Een manier voor het versnellen van simulaties is het gebruiken van meer rekencapaciteit door het inzetten van meerdere computers. Een

stap verder is het gebruik van FPGA's waarop een IC niet gesimuleerd maar *geëmuleerd* wordt. Het ontwerptraject voor FPGA's verschilt weinig van dat van een digitale IC en kent onder andere synthese vanuit op RT-niveau beschreven HDL. Dat betekent dat er met een relatief kleine inspanning een FPGA-prototype is te maken dat vervolgens ingezet kan worden voor verificatie.

Vanwege de beperkingen van de 'klassieke simulatie' is er veel aandacht voor meer geavanceerde simulatietechnieken en verificatiemethoden waarin geen simulatie wordt gebruikt. Een overzicht wordt hieronder gepresenteerd.

5.2 Codedekking g

Een gereedschap voor *codedekking* (Eng. *code coverage*) houdt bij welke regels van de HDL-broncode zijn gesimuleerd. Regels die niet zijn gesimuleerd worden gerapporteerd. Die informatie kan dan gebruikt worden om de stimuli zodanig uit te breiden dat de niet gesimuleerde code tijdens simulatie geactiveerd wordt.

Codedekking op basis van alleen tekstregels is de meest eenvoudige vorm. Verdergaande vormen houden bijvoorbeeld niet alleen bij of alle toestanden van een toestandsmachine tijdens de simulatie bezocht zijn, maar ook of alle mogelijke toestandsovergangen doorlopen zijn. Of ze houden niet alleen bij of een conditie `(a OR b)` van een if-statement waar of onwaar is geweest, maar ook of de signalen `a` en `b` afzonderlijk waar en onwaar zijn geweest (als beide signalen steeds tegelijk waar of onwaar zijn, hoeft er in de conditie alleen naar een van de twee verwezen te worden en is er in zekere zin sprake van een ontwerpfout).

5.3 Analyse van HDL-code

Gereedschappen voor controle van *codeer- en ontwerpstijl* (Eng. *design and coding guideline checkers*) worden ingezet voorafgaand aan synthese. Zij analyseren de beschrijving van hardware op RT-niveau in een HDL. Zij kunnen controleren of overal gebruik is gemaakt van de synthetiseerbare subset (zie paragraaf 4.2) van de taal. Zij kunnen ook nagaan of de ontwerper zich gehouden heeft

aan afgesproken ontwerpregels. Voorbeelden van zulke regels zijn dat de reset altijd asynchroon of juist altijd synchroon is, dat alle registers op dezelfde klokflank werken, enz. Weer een andere toepassing is het controleren of documentatie in de code (commentaar) voldoet aan standaarden die in een bedrijf gehanteerd worden.

5.4 Statische timinganalyse

In paragraf 3.2 en 3.3 zijn timing gerelateerde begrippen als setup- en holdtijd, clock skew, enz. nader toegelicht. Als niet is voldaan aan timingeisen zal dat kunnen leiden tot foutieve werking van het IC. Tijdens logische synthese en steeds vaker ook bij het maken van de lay-out wordt er rekening met de timing gehouden. Na de lay-out zijn schattingen van alle vertragingen bekend. Ze kunnen, zoals hierboven gezegd, in postlay-out simulaties gebruikt worden, maar ze kunnen ook de invoer zijn van een gereedschap voor *statische timinganalyse*. Daar waar een simulatie een dynamisch gebeuren is waar events aanleiding geven tot nieuwe events (zie paragraaf 4.3), zal statische timinganalyse op basis van langste en kortste paden door de logica nagaan of aan alle timingeisen is voldaan.

Uitgangspunt voor statische timinganalyse is de structuur van een sequentiële schakeling zoals getoond in figuur 3.3. Een signaalpad begint bij een primaire ingang of een registeruitgang en eindigt in een primaire uitgang of een registeringang. Bepaling van langste paden is bijvoorbeeld noodzakelijk voor de analyse van de haalbaarheid van een gegeven klokfrequentie. Analyse van clock skew is juist een voorbeeld van een situatie waarbij het korst mogelijke pad van belang is (zie figuur 3.6).

5.5 Formele verificatie

Om het exponentieel groeiende aantal combinaties in simulatie het hoofd te bieden heeft men in de loop der jaren ook naar de wiskunde gekeken. Daar is het immers gebruikelijk stellingen te bewijzen over verzamelingen met zelfs een oneindig aantal elementen (bijv. de verzameling van natuurlijke getallen) zonder elk element afzonderlijk te beschouwen. Verificatiemethoden gebaseerd op wiskundige bewijsvoering vallen in de categorie *formele verificatie*.

Een formele verificatiemethode die zijn toepassing in de praktijk heeft gevonden, is de RT-niveau *equivalentiecontrole* (Eng. *equivalence check*). Het doel bij deze controle is aan te tonen dat twee beschrijvingen van dezelfde schakeling functioneel volledig identiek zijn. Er zijn diverse situaties waarbij zo'n controle van pas komt. Men kan bijvoorbeeld de beschrijvingen van een schakeling voor en na synthese met elkaar vergelijken en zo aantonen dat de synthese gelukt is. Een andere situatie doet zich voor als de hiërarchische opdeling van schakeling wordt aangepast en een deelschakeling naar een andere tak van de decompositieboom (zie figuur 2.3) wordt verplaatst. Door de schakelingen voor en na de verplaatsing met elkaar te vergelijken kan men aantonen dat de verplaatsing de functionaliteit intact heeft gelaten.

De equivalentiecontrole is gebaseerd op het feit dat elke primaire uitgang of registeringang in het RT-model van figuur 3.3 te beschouwen is als een Booleaanse functie. Van alle combinatorische logica in de schakeling wordt voor elk uitgangsbit de Booleaanse functie bepaald en wordt de functie opgeslagen in een canonieke vorm. *Canoniek* wil zeggen dat de functie ongeacht de manier waarop hij extern, dus buiten de checker, is gespecificeerd op een eenduidige manier door een interne datastructuur gerepresenteerd kan worden. Een waarheidstabel die voor elke combinatie van ingangssignalen de uitgangswaarde van een Booleaanse functie geeft is een voorbeeld van een canonieke representatie. Het heeft het grote nadeel dat hij exponentieel groeit als functie van het aantal ingangssignalen. In de praktijk wordt gebruik gemaakt van compactere canonieke representaties.

Equivalentiecontrole verloopt in twee stappen:

1. In de eerste stap worden van de twee te vergelijken schakelingen de primaire uitgangen en registeringangen paarsgewijs aan elkaar gekoppeld (Eng. *matching*). Dit gebeurt op basis van signaalnamen en indien nodig handmatige specificatie door de gebruiker.
2. Vervolgens wordt voor elk paar van functies aangetoond dat ze equivalent zijn. Als dat niet lukt wordt gerapporteerd voor welke combinaties van ingangssignalen er geen equivalentie is.

Naast de equivalentiecontrole op RT-niveau, zijn tal van andere formele verificatiemethoden onderwerp van onderzoek en producttontwikkeling. Ze blijven echter onbesproken omdat ze nog niet breed worden toegepast.

5.6 Assertions

Assertions (beweringen, eigenschappen) zijn toevoegingen aan een hardwarebeschrijving. Ze bestaan uit twee delen:

1. Het eerste deel is een combinatie van signaalwaarden die bijzonder is. Meestal gaat het om een combinatie die niet zou mogen voorkomen. De waarden kunnen betrekking hebben op één moment in de simulatie of op een sequentie van waarden in het verloop van de simulatie.
2. Het tweede deel bevat een actie die uitgevoerd moet worden op het moment dat de conditie in het eerste deel waar wordt. Bij zo'n actie kan men denken aan het afdrukken van een waarschuwingsboodschap of het stilzetten van de simulatie.

Bij een bus waar meerdere hardware blokken op schrijven zou men bijvoorbeeld een assertion kunnen formuleren die controleert of er hooguit één blok tegelijk op schrijft. Een ander voorbeeld is '*one-hot codering*', een codering van een woord waarin hooguit een bit `'1'` is en alle andere `'0'` zijn. Als een blok voor een van zijn ingangen zo'n codering verwacht kan een assertion geformuleerd worden die dit controleert.

VHDL heeft sinds zijn ontstaan assertions gekend, al zijn deze vrij beperkt. System-Verilog heeft uitgebreide mogelijkheden voor het beschrijven van assertions. Er bestaan ook talen, zoals *PSL* (*Property Specfication Language*), die uitsluitend voor assertions zijn bedoeld. Simulatoren worden dan geacht een hardwarebeschrijving in bijvoorbeeld VHDL en assertions in SystemVerilog of PSL met elkaar te combineren.

Het is de bedoeling assertions in alle simulaties mee te nemen. Ze vertragen weliswaar de simulatie, maar bieden ook meer zekerheid over de correctheid van een ontwerp.

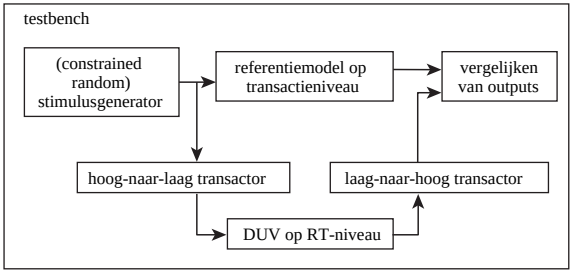
5.7 Geavanceerde simulatie

Bij het maken van een systeemmodel van te ontwerpen hardware kan men abstraheren van de precieze natuur van interfaces in zowel de ruimte (het aantal signalen dat betrokken is bij het interface, woordbreedtes van bussignalen, enz.) als de tijd (het schakelen van signalen in een bepaalde volgorde). Stel bijvoorbeeld dat een datawoord vanuit een processor over een bus naar een geheugen moet worden getransporteerd. Zo'n transport kan in abstracte zin bestaan uit een enkele functieaanroep van het busmodel. De concrete implementatie vereist het doorlopen van een busprotocol waarin bijvoorbeeld adres, data en andere signalen in de juiste volgorde verdeeld over meer klokperiodes geschakeld moeten worden.

Communicatie over een abstracte interface wordt een *transactie* genoemd. Een model van de hardware gebaseerd op de uitwisseling van transacties wordt een model op *transactieniveau* (Eng. *transaction level*) genoemd. Beschouw bijvoorbeeld een telefoon: een toets die wordt ingedrukt van zou een ingangstransactie kunnen zijn, terwijl tekst die op het display zichtbaar moet worden gemaakt een mogelijke uitgangstransactie is.

Terwijl aan de hardware wordt ontworpen, kan het model op transactieniveau dienen voor het tegelijkertijd ontwikkelen van de software. Om te verifiëren dat de software correct is kan men simuleren met zorgvuldig uitgezochte sequenties van transacties. Men kan echter ook gebruik maken van een generator van random patronen om de robuustheid van het systeem te testen. In het voorbeeld van de telefoon geldt weliswaar dat niet elke reeks toetsaanslagen zinvol is, maar het is tevens wenselijk dat de software niet vastloopt na een onzinnige reeks toetsaanslagen.

Generatoren van random patronen maken tegenwoordig deel uit van verificatieomgevingen (SystemVerilog, SystemC *Verification Library*). Ze kunnen constrained patronen genereren, d.w.z. dat bepaalde delen van de patroonruimte uitgesloten kunnen worden (omdat de hardware niet ontworpen is voor die bepaalde patronen) en dat patronen die wel gegenereerd mogen worden moeten voldoen aan bepaalde kansverdelingen.



Figuur 5.4 Opzet van een testbench die gebruik maakt van modellering op transactieniveau

Als de hardware tot op RTL-niveau is uitgewerkt, kan deze samen gesimuleerd worden met het abstracte model op transactieniveau (zie figuur 5.4). Dat vereist een *hoog-naar-laag transactor* die ingangstransacties vertaalt naar concrete stimuli. In het geval van de telefoon zal zo'n transactor het indrukken van een toets moeten vertalen naar bijvoorbeeld een kortsluiting in een matrix van draden. De uitgangen van het DUV op RT-niveau moeten weer terugvertaald worden naar het hoge transactieniveau om te kunnen worden vergeleken met de uitgangen van het abstracte model. Dit wordt gedaan door een *laag-naar-hoog transactor.* In het voorbeeld zal zo'n transactor het dataverkeer op een display moeten registreren en terugvertalen naar de teksten die erop vertoond worden.

Deze manier van geavanceerd simuleren kan ook worden toegepast op tussenstadia van ontwerpen. Bij elk tussenstadium kan het nodig zijn de twee transactoren aan te passen aan het abstractieniveau van het ontwerp op dat moment. Het referentiemodel op transactieniveau kan keer op keer hergebruikt worden.

6 Testbaar ontwerpen en testen

Zoals in het begin van paragraaf 5 is gezegd, is het doel van testen het detecteren van defecten aan een IC als gevolg van onvolkomenheden

in de productie. Daartoe worden reeksen signalen en testpatronen aan de ingangen van een IC aangeboden en worden de uitgangen geobserveerd. Als de observatie afwijkt van wat er verwacht was is het IC defect.

Er kan in de productie veel mis gaan. Om systematisch testpatronen te genereren moet men zich beperken tot bepaalde klassen van fouten, beschreven door zogenaamde *foutmodellen*. De foutmodellen die vaak gehanteerd worden zijn:

- *Stuck-at*. Hierbij neemt men aan dat een knooppunt in het netwerk van logische poorten kortgesloten is met de positieve (stuck-at-1) dan wel negatieve (stuck-at-0) voedingsspanning.
- *Stuck-open/stuck-closed*. Dit foutmodel neemt aan dat een transistor die zich normaal als een schakelaar gedraagt, niet meer kan schakelen en permanent een open verbinding respectievelijk een kortsluiting vormt.
- *Bridging*. Dit model neemt aan dat twee verbindingen met elkaar zijn kortgesloten.
- *Delay.* Er is sprake van een *delayfout* als er degradatie optreedt in de propagatietijd van een signaal. Een verbinding die normaliter snel genoeg is wordt traag, waardoor bijvoorbeeld een berekening niet meer binnen een klokperiode afgerond wordt.

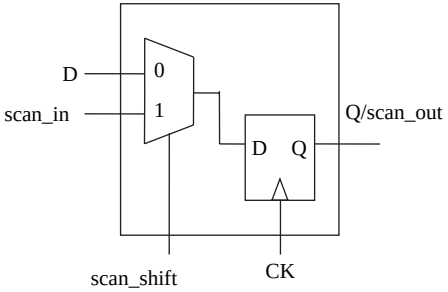
Testpatronen worden automatisch gegenereerd door een CAD-gereedschap voor *automatische testpatroongeneratie* (*ATPG*). Zo'n gereedschap zal met zo weinig mogelijk patronen zoveel mogelijk fouten binnen een foutmodel proberen te detecteren. Het percentage van fouten dat een verzameling testpatronen kan detecteren, wordt de *dekkingsgraad* van de testpatronen genoemd.

Combinatorische schakelingen zijn relatief eenvoudig te testen. Vanwege het ontbreken van interne geheugenelementen is uit elk ingangspatroon op een eenduidige manier de signaalwaarde van elk intern knooppunt en elke uitgang te bepalen. Programmatuur voor ATPG kan ieder gewenst ingangspatroon aan de schakeling aanbieden en daarmee de schakeling zo sturen dat zoveel mogelijk fouten aan de uitgang observeerbaar worden.

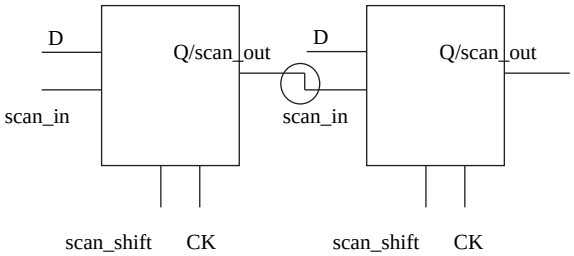
Bij sequentiële schakelingen is een groot deel van de ingangssignalen van de combinatorische logica de uitgang van een register (zie figuur 3.3). Het is zeer gewenst dat programmatuur voor ATPG ook de inhoud van de registers op een eenvoudige manier kan instellen. Een manier om dit mogelijk te maken is het onderbrengen van de registers in een zogenaamd *scanpad*.

De eerste stap bij het bouwen van een scanpad is het vervangen van de flipflops door *scanflipflops*. Een scanflipflop is een 'gewone' flipflop, bijvoorbeeld de positive edge-triggered flipflop uit figuur 3.2, met aan zijn ingang een multiplexer. Dit is geïllustreerd in figuur 6.1. Zolang het signaal `scan_shift` waarde `'0'` heeft gedraagt de scanflipflop zich als een normale flipflop. Als echter `scan_shift` waarde `'1'` heeft wordt het signaal `scan_in` ingeklokt.

Een *scanketen* (Eng. *scan chain*) wordt opgebouwd door scanflipflops zodanig te schakelen dat de uitgang `scan_out` van de ene scanflipflop aangesloten wordt op de ingang `scan_in` van een volgende scanflipflop. Dit is aangegeven in figuur 6.2. Als nu `scan_shift` waarde `'1'` heeft, gedraagt de scanketen zich als een *schuifregister*.

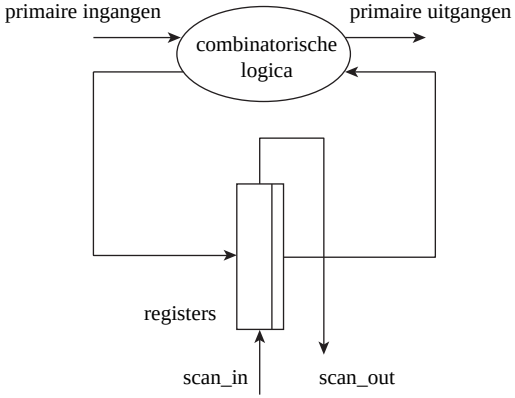


Figuur 6.1 De samenstelling van een scanflipflop uit een flipflop en een multiplexer



Figuur 6.2 Het koppelen van twee scanflipflops t.b.v. een scanketen

Door alle flipflops te vervangen door scanflipflops en ze te schakelen in één lang schuifregister wordt het mogelijk alle ingangen van de combinatorische logica te benaderen ten behoeve van testen. Dit is geïllustreerd in figuur 6.3. Het testen gaat als volgt:



Figuur 6.3 Algemene structuur van synchrone hardware met een testketen

- Terwijl het signaal `scan_shift` de waarde `'1'` heeft, wordt het testpatroon serieel in de scanflipflops geladen.
- Als het hele patroon geladen is wordt `scan_shift` gedurende één klokperiode `'0'`, terwijl er ook een testpatroon aan de primaire ingangen wordt aangeboden. Op dat moment hebben alle ingangen van de combinatorische logica een bekende waarde. De primaire uitgangen worden meteen geobserveerd en de flipflops slaan de waarde van de uitgangen van de combinatorische logica op.
- Het signaal `scan_shift` krijgt opnieuw de waarde `'1'` en de uitgangen van de combinatorische logica die in de flipflops waren opgeslagen worden serieel naar buiten geschoven en vergeleken met het verwachte patroon. Tegelijkertijd kan een nieuw patroon in de flipflops geschoven worden.

De seriële toegang tot de scanketen is een nadeel van deze methode van testen. Om het testen te versnellen kan men meerdere scanketens gebruiken. Dit kost dan wel extra pinnen aan het IC; per scanketen is een aparte pin voor `scan_in` en `scan_out` nodig.

Een ander nadeel van scanketens is de toename van oppervlakte. Het systematisch vervangen van elke flipflop door een scanflipflop doet de oppervlakte van het IC met zo'n 10 tot 20 % toenemen. Als bepaalde flipflops en de combinatorische logica aan hun in- en uitgangen op een functionele manier zijn te testen, kan men deze flipflops uit de scanketen weglaten.

Een *functionele test* verwijst naar een reeks van testpatronen die met kennis van de functie van het IC zijn samengesteld. Om bijvoorbeeld de opteller in het datapad van een processor te testen kan men de processor een aantal optellingen laten doen. Bij testpatronen die uitsluitend door toepassing van een foutmodel op een netlist van standaardcellen tot stand zijn gekomen, spreekt men van *structurele testen*.

Naast het aanbrengen van scanketens zijn er allerlei andere methoden voor het testbaar maken van een IC. Een interessante categorie is de *built-in self test* (*BIST*). Hierbij wordt de testpatroongenerator in hardware op het IC gerealiseerd, net als de hardware die de uitgan-

gen van de testen blokken evalueert. Alleen het resultaat van de evaluatie hoeft dan naar buiten te worden gecommuniceerd.

7 Samenvatting

In de tekst hierboven is veel informatie gepresenteerd dat in verband staat met het ontwerptraject van digitale IC's. Ter afsluiting van dit hoofdstuk wordt deze informatie samengevat door de chronologie van het ontwerptraject te volgen.

Het ontwerpen van een IC van aanzienlijke omvang, van een system-on-chip, begint met systeemstudies. Hardware-software co-design dat onder andere gebruik maakt van systeemsimulaties met modellering op transactieniveau, leidt tot een architectuur. Hardwarebeschrijving geschiedt in een taal die geschikt is voor dit abstractieniveau zoals SystemC.

Als de architectuur vast ligt moeten de blokken uitgewerkt worden tot het RT-niveau. Mogelijk kan men bestaande IP-blokken hergebruiken. Voor modellering op RT-niveau zijn de hardwarebeschrijvingstalen VHDL en Verilog beschikbaar. Ontwerpfouten worden voornamelijk opgespoord door simulaties.

Logische synthese zet beschrijvingen op RT-niveau om naar een netlist van standaardcellen. Lay-outsynthese ten slotte verdeelt de cellen over de oppervlakte en bedraadt ze onderling. Simulatie op poortniveau en timinganalyse, waarbij vertragingen uit de lay-out m.b.v. back annotation gekoppeld zijn aan de netlist, zijn nodig om te verifiëren dat het eindproduct aan de specificaties voldoet.