# Digitale Techniek

*Een inleiding in het ontwerpen van digitale systemen*

Jesse op den Brouw

Deel 2

Versie: 1.1
Datum: 18 augustus 2020

DE HAAGSE HOGESCHOOL



De auteur schenkt de royalties aan Stichting KiKa.




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

# Voorwoord

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

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

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

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

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

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

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

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

**Leeswijzer**

In hoofdstuk 7 wordt de beschrijvingstaal VHDL geïntroduceerd. We beginnen met de concepten van VHDL, hoe de taal is opgebouwd en hoe het gebruikt moet worden. VHDL is nauw verwant met simulatie zodat een flink deel is ingeruimd om dit te behandelen. VHDL kan gebruikt worden voor synthese, het automatisch genereren van hardware. We vertellen wat wel en wat niet kan worden gesynthetiseerd. VHDL is een omvangrijke taal met veel taalconstructies en mogelijkheden. Daarom behandelen we de taal niet uitgebreid. Referenties naar diverse literatuur worden gegeven.

Hoofdstuk 8 staat in het teken van schuifregisters en tellers. Dat zijn twee veel voorkomende digitale bouwstenen. Er wordt uitgelegd wat een schuifregister is en wat de mogelijkheden zijn zoals linksom en rechtsom schuiven, laden en data vasthouden. Schuiven heeft ook een rekenkundige werking, namelijk vermenigvuldigen van en delen door 2. Dat is voor unsigned en signed getallen verschillend. De opbouw en VHDL-beschrijving van schuifregisters worden uitgelegd en er wordt een voorbeeld gegeven van een toepassing: de RS-232 interface. Een andere veel gebruikte component is de teller. De basis van tellen in het binaire talstelsel wordt behandeld en leidt tot de constructie van tellers op volgens het binaire telproces. In eerste instantie worden tellers die tellen met een macht van 2 behandeld. Later passeren ook andere telcycli de revue. We laten twee voorbeelden zien waarbij tellers gebruikt worden: een digitale signaalvormgenerator en een PWM-generator.

In hoofdstuk 9 gaan we dieper in op timing in kloksynchrone systemen. We bekijken directe en indirecte dataoverdracht tussen flipflops en laten zien dat het mogelijk is om de maximale toelaatbare klokfrequentie te berekenen. De timing van externe ingangen en uitgangen worden ook besproken, evenals de timing bij dataoverdracht tussen flipflops met verschillende timingparameters. Een flink deel van dit hoofdstuk is ingevuld met voorbeelden.

Hoofdstuk 10 behandelt drie onderwerpen waar we met praktische systemen mee te maken krijgen: synchronisatie, metastabiliteit en reset. We laten eerst zien dat synchronisatie van externe ingangssignalen noodzakelijk is om de betrouwbaarheid van een digitaal system te waarborgen. Synchronisatie van uitgangssignalen kan noodzakelijk zijn om verschijnselen als hazards te voorkomen. Een reset behoort tot een vast onderdeel van elk digitaal systeem met geheugen. Het dwingt de geheugenelementen in een bekende begintoestand zodat de een correcte werking verzekerd is.

### Studiewijzer

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

| **Deel 2** | |
|---|---|
| Hoofdstuk 7 | helemaal |
| Hoofdstuk 8 | 8.1, 8.4, 8.6 (aanbevolen 8.2, 8.3 en 8.5) |
| Hoofdstuk 9 | helemaal |
| Hoofdstuk 10 | helemaal |

### Verantwoording inhoud

De taal VHDL wordt pas na ongeveer de helft van het boek geïntroduceerd. Dat heeft als reden de student in het eerste gedeelte niet te vermoeien met allerlei zaken als opbouw van de taal, simulatie en synthese. Een te snelle introductie leidt tot onvoldoende kennis van de synthese van digitale schakelingen. Er wordt slechts een deel van de taal besproken en er wordt ingegaan op simulatie en synthese. Onderwerpen als files, pointer en records worden niet besproken.

Twee belangrijke bouwstenen in digitale systemen zijn schuifregisters en tellers. In feite zijn het toestandsmachines maar dan met een specifieke, generieke functie. We laten zien dat het eenvoudig is om schuifregisters en tellers met VHDL te beschrijven.

Timing bij dataoverdracht wordt in de meeste boeken over digitale techniek niet uitgebreid behandeld terwijl het een belangrijk onderwerp is. Een schakeling kan misschien op logisch gedrag correct werken maar niet qua timing. Alleen de timing bij kloksynchrone systemen wordt behandeld omdat verreweg de meeste systemen kloksynchroon ontworpen worden.

Bij praktische systemen krijgen de studenten te maken met begrippen als synchronisatie, metastabiliteit en reset-implementatie. Het is dus vooral een praktische exercitie waarbij de begrippen worden uitgelegd.

### Website

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

### Dankbetuigingen

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

Verder worden mijn studenten bedankt, in het bijzonder Marcel Jongkind, Frederick Kramer, Ruben de Graaff, Bob Swinkels, Daniël Vermeulen en Abraar Muhammad, voor het opsporen van enkele fouten en het geven van suggesties.

Jesse op den Brouw, augustus 2020.

# Inhoudsopgave


**Voorwoord** iii

**7 Introductie VHDL** 1
- 7.1 Een eerste kennismaking . . . 4
  - 7.1.1 Design unit . . . 4
  - 7.1.2 Entity en architecture . . . 5
  - 7.1.3 Signalen . . . 6
  - 7.1.4 Commentaar . . . 6
  - 7.1.5 Identifiers . . . 6
  - 7.1.6 Standaard datatypes . . . 7
  - 7.1.7 De types std_ulogic en std_logic . . . 8
  - 7.1.8 De types unsigned en signed . . . 10
  - 7.1.9 Declaratie van types en subtypes . . . 11
  - 7.1.10 Vectoren . . . 12
  - 7.1.11 Attributes . . . 14
- 7.2 Concurrent VHDL . . . 15
  - 7.2.1 Conditional Signal Assignment . . . 15
  - 7.2.2 Selected Signal Assignment . . . 16
  - 7.2.3 VHDL delays . . . 18
  - 7.2.4 Voorbeeld: tri-state buffers . . . 19
- 7.3 Sequential VHDL . . . 20
  - 7.3.1 Proces . . . 21
  - 7.3.2 If . . . 22
  - 7.3.3 Case . . . 22
  - 7.3.4 Variabele . . . 23
  - 7.3.5 For . . . 23
  - 7.3.6 While . . . 24
  - 7.3.7 Gated D-latch . . . 24
  - 7.3.8 D-flipflop . . . 25
  - 7.3.9 Wait . . . 26
- 7.4 Structural VHDL . . . 28
  - 7.4.1 Declaratie en instantiëring van componenten . . . 28
  - 7.4.2 Generics . . . 29
- 7.5 Simulatie . . . 31
  - 7.5.1 Delta delay . . . 34

7.5.2 Initialisatie van signalen en variabelen . . . 35
7.5.3 Testbench . . . 35
7.5.4 Kloksignaal . . . 36
7.5.5 Datasignalen . . . 36
7.5.6 Reset . . . 37
7.5.7 Simulatie van een T-flipflop . . . 37
7.6 Synthese . . . 44
7.6.1 Combinatorische logica . . . 45
7.6.2 Geheugenelementen . . . 47
7.6.3 Variabelen en signalen . . . 49
7.6.4 Synthese van datatypes . . . 50
7.7 Rekenen met de types unsigned en signed . . . 50
7.8 RAM en ROM . . . 53
7.9 Hints bij simulatie en synthese . . . 55
7.10 Opgaven . . . 57

**8 Schuifregisters en tellers** **61**
8.1 Schuifregisters . . . 61
8.1.1 Bidirectionele schuifregisters . . . 63
8.1.2 Universeel schuifregister . . . 64
8.1.3 Seriële transmissie . . . 65
8.1.4 Vermenigvuldigen met en delen door 2 . . . 67
8.2 Voorbeeld: ontdenderen schakelaar . . . 68
8.3 Voorbeeld: RS232-communicatie . . . 69
8.4 Tellers . . . 71
8.4.1 Grondbeginselen van tellers . . . 72
8.4.2 Ontwerp van een 4-bits teller met structural VHDL . . . 76
8.4.3 Ontwerp van een 4-bits teller met een opteller . . . 78
8.4.4 Ontwerp van een BCD-teller . . . 80
8.4.5 Tellers met integers . . . 83
8.5 Voorbeeld: digitale signaalvormgenerator . . . 85
8.6 Voorbeeld: digitale PWM-generator . . . 89
8.7 Ringtellers . . . 95
8.8 Opgaven . . . 97

**9 Timing bij dataoverdracht** **101**
9.1 Timing D-flipflop . . . 102
9.2 Directe dataoverdracht tussen flipflops . . . 103
9.3 Klokskew . . . 105
9.4 Indirecte dataoverdracht tussen flipflops . . . 108
9.5 Timing van externe ingangen en uitgangen . . . 109
9.6 De maximale systeemfrequentie . . . 111
9.7 Vertraging van signaallijnen . . . 113
9.8 Timing met verschillende parameters . . . 113
9.9 Voorbeelden . . . 114
9.10 Opgaven . . . 120

**10 Synchronisatie, metastabiliteit en reset** **125**
10.1 Synchronisatie van ingangssignalen . . . . . 126
10.2 Synchronisatie van uitgangssignalen . . . . . 129
10.3 Synchronisatie tussen clock domains . . . . . 129
10.4 Metastabiliteit . . . . . 130
10.5 Reset-implementatie . . . . . 133
10.6 Opgaven . . . . . 135

**Bibliografie** **139**

**A De FPGA** **141**

**Index** **145**

# 7

# Introductie VHDL

In dit hoofdstuk wordt de taal VHDL behandeld. VHDL staat voor *VHSIC Hardware Description Language* en in de naam staat al een eerste aanwijzing wat de taal inhoudt. Het is een namelijk *beschrijvingstaal*. Het is een taal om hardware te beschrijven en modelleren. Aangezien hardware van nature parallel opereert, kunnen we ook spreken van een beschrijvingstaal voor *concurrent* systemen. Concurrent wil zeggen gelijktijdig. VHSIC staat trouwens voor *Very High Speed Integrated Circuit*.

Noot: VHDL is een omvangrijke taal met veel taalconstructies en mogelijkheden. We zullen slechts een introductie geven, zowel over de taal als over simulatie en synthese. Zie voor meer informatie [9, 10, 11].

De eerste ontwikkelingen vonden plaats rond 1980. Het Amerikaanse leger had behoefte aan een gestandaardiseerde methode voor het beschrijven van elektronische systemen. VHDL is voor het eerst vastgelegd in 1987 door IEEE (standaard IEEE 1076-1987). De standaard is herzien in 1993 (VHDL-93), in 2000 (VHDL-2000), in 2001 (VHDL-2001) en in 2008 (VHDL-2008). Er is ook een standaard die specifiek voor synthese is bedoeld: IEEE 1076-6. Alle verkrijgbare VHDL-tools kunnen in ieder geval overweg met VHDL-1993. De meeste kunnen ook overweg met VHDL-2008.

Oorspronkelijk is de taal bedoeld voor het beschrijven, simuleren en documenteren van digitale systemen. VHDL is dan ook nauw gekoppeld aan simulatie. Er zijn veel taalconstructies die alleen gebruikt kunnen worden bij simulatie. Tegenwoordig kan een deel van de taal gebruikt worden om hardware te synthetiseren, d.w.z. dat er hardware gegenereerd wordt.

Er zijn veel redenen om digitale schakelingen te beschrijven met VHDL. De belangrijkste reden is dat de taal technologie- en platformonafhankelijk is. Overzetten van het ene naar het andere platform gaat min of meer probleemloos[1]. Daarnaast ondersteunt VHDL

[1] Natuurlijk is dat wel betrekkelijk. Voor ontwerpen die niet “on the bleeding edge” van de technologie liggen is dit waar. Het gaat ook niet op voor ontwerpen die gebruik maken van fabrikant-specifieke ingebouwde schakelingen zoals vermenigvuldigers en Ethernet-controllers.

hiërarchisch ontwerpen dat belangrijk is bij het ontwikkelen van grote systemen. Ontwikkelde beschrijvingen kunnen in een *library* (bibliotheek) worden geplaatst zodat ze kunnen worden hergebruikt in latere ontwerpen. Bij het ontwikkelen van een schakeling met poorten moet een ontwerper rekening houden met de logische werking, elektrische eigenschappen, ic-oppervlak en tijdgedrag. VHDL maakt het mogelijk om te concentreren op de logische werking en tijdgedrag. De rest wordt aan softwaretools overgelaten. Ook het ontwerpen van *generieke* beschrijvingen is een voordeel. Zo kan met een simpele wijziging een 8-bits opteller worden omgezet naar een 16-bits opteller. VHDL wordt gebruik voor:

| | |
|---|---|
| Specificatie van het ontwerp: | Ondubbelzinnige definitie van functionaliteit, componenten en interfaces in een groot ontwerp. |
| Simulatie van het ontwerp: | Verificatie van het systeem in termen van prestatie, functionaliteit etc. voordat deze wordt geïmplementeerd. |
| Synthese van het ontwerp: | Automatisch genereren van hardware. |

VHDL is **geen** programmeertaal. Het is een beschrijvingstaal voor fysieke systemen zoals een digitale schakeling. Digitale systemen hebben een hoge mate van concurrency. Dit levert problemen op voor softwareprogrammeurs. Zij denken in sequentiële constructies. In software worden instructies na elkaar uitgevoerd. Programmeurs moeten anders leren denken: concurrent i.p.v. sequentieel. VHDL is geen Verilog of SystemC. Verilog is een beschrijvingstaal die veel weg heeft van 'C'. De taal wordt veel in Amerika gebruikt. SystemC is een verzameling van 'C++'-klassen en macro's voor het simuleren van concurrent processen, elk beschreven met behulp in de gewone 'C++'-syntax.

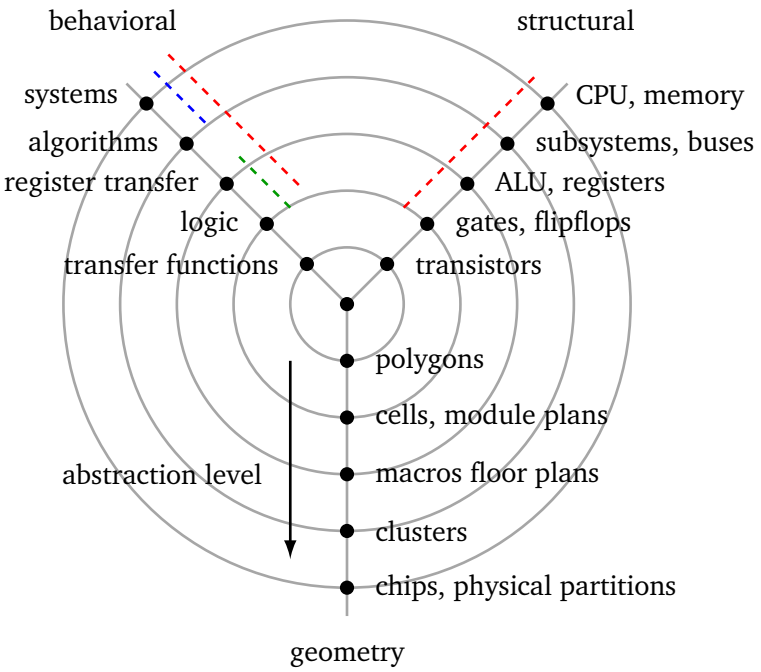
### Ontwerpdomeinen

Ontwerpen kunnen met drie verschillende *domeinen* beschreven worden:

| | |
|---|---|
| Behavioral | het gedrag van (een deel van) het ontwerp wordt beschreven. |
| Structural | het ontwerp wordt beschreven in termen van deelontwerpen. |
| Geometry | het ontwerp wordt beschreven in termen van fysieke plaatsing van structuren (transistoren, weerstanden, condensatoren) op een ic. |

De domeinen kunnen worden weergegeven met de *Gajski-Kuhn Y-chart* [12], te zien in figuur 7.1. In de figuur is aangegeven welke domeinen en abstractieniveaus in VHDL kunnen worden gebruikt. Hieronder volgt een uiteenzetting waarbij we de lijn van het gedrag (behavioral) volgen.

**Systems** Het ontwerp wordt beschreven in termen van complete systemen die interactie met elkaar hebben. Te denken valt aan een microprocessor met geheugen. Er wordt een beschrijving gemaakt van elk systeem en die worden aan elkaar gekoppeld. Het gaat hierbij vooral om het functionele gedrag en het globale tijdgedrag in kaart te brengen. Op dit niveau is niet bekend hoe de hardware eruit komt te zien. Synthese is niet altijd mogelijk, simulatie wel. De functionaliteit van het geheel is goed te volgen.




**Figuur 7.1:** *Gajski-Kuhn Y-chart.*

**Algorithms** Op dit niveau wordt het ontwerp beschreven in algoritmes. Denk hierbij aan het bepalen van het grootste getal uit een rij getallen of het beschrijven van een deelroutine voor two's complement getallen. Er wordt gebruik gemaakt van lusstructuren en variabelen. Ook hierbij gaat het vooral om de functionaliteit. Het exacte tijdgedrag is wat minder van belang. Er kan volstaan worden met een globaal timingmodel. De functionaliteit van het ontwerp is nog te volgen al kost het enige moeite. Synthese is niet altijd mogelijk, simulatie wel.

**Register Transfer (RT)** Hierbij wordt het ontwerp verdeeld in afzonderlijke blokken die eenvoudig beschreven kunnen worden zoals registers, tellers, optellers, ROM-tabellen en toestandsmachine. De functionaliteit van het systeem is niet meer in één oogopslag te zien. Simulatie en synthese met VHDL zijn mogelijk.

**Logic** Hierbij wordt het ontwerp volledig beschreven in logische functies zoals AND, OR en NOT. De functionaliteit van het systeem is niet meer te zien. Op dit niveau is het mogelijk om een gedetailleerd tijdgedrag te bepalen. Simulatie en synthese met VHDL zijn mogelijk.

Het is met VHDL niet mogelijk om direct structuren (geometry) op een ic te beschrijven. Dat willen we ook niet, we laten dat graag over aan de softwaretools. VHDL heeft geen taalconstructies voor simulatie in dit domein. Wat betreft de andere twee domeinen is simulatie en synthese (deels) mogelijk. De stippellijnen geven de mogelijkheden aan. In het structural domein is simulatie en synthese mogelijk met uitzondering van het

transistorniveau. Het behavior domein is te splitsen in twee delen: high level synthese (systems, algoritms) en low level synthese (register transfer, logic). Het is niet altijd mogelijk om high level synthese toe te passen. Dat heeft te maken met de manier waarop een (deel van het) ontwerp is beschreven. Er kunnen bijvoorbeeld lusconstructie gebruikt worden waar niet direct hardware voor te synthetiseren is. Low level synthese is wel mogelijk. Het laagste niveau is niet in VHDL te gebruiken. Simulatie van de niveaus is met VHDL mogelijk.

Noot: wij zullen ons alleen bezig houden met het Register Tranfer niveau en het Logic niveau.

**Simulatie en synthese**

Nadat een digitaal ontwerp beschreven is in VHDL dient het *gesimuleerd* te worden. Simulatie wordt gebruikt voor verificatie van het ontwerp; doet het ontwerp wat het moet doen? Wees ervan bewust dat simulatie is een model van de werkelijkheid en kan dus afwijken van die werkelijkheid.

Als blijkt dat de simulatie goed is verlopen dan kan het ontwerp *gesynthetiseerd* worden. Synthese is het automatisch vertalen van VHDL-code naar primitieven (poorten, flipflops, optellers, multiplexers) op een chip. Dit kan een configureerbaar ic (FPGA, CPLD) zijn of een ASIC. Synthese is een lastig proces. De synthesesoftware is nog volop in ontwikkeling. Het is mogelijk VHDL-code te schrijven die wel simuleerbaar is, maar niet synthetiseerbaar.

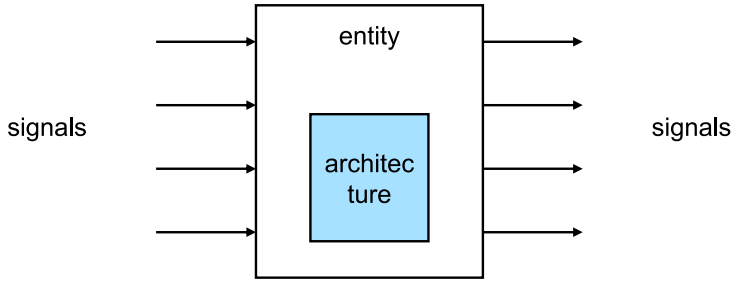
## 7.1 Een eerste kennismaking

We zullen enkele basisconcepten uitleggen aan de hand van voorbeelden. We leggen uit wat een design unit is en hoe deze is opgebouwd. Interactie tussen design units gaat via *signals*, een belangrijk concept binnen hardware-beschrijvingstalen. Het is ook mogelijk om array's te gebruiken. Operaties op array's gaat verder dan in een programmeertaal als 'C'.

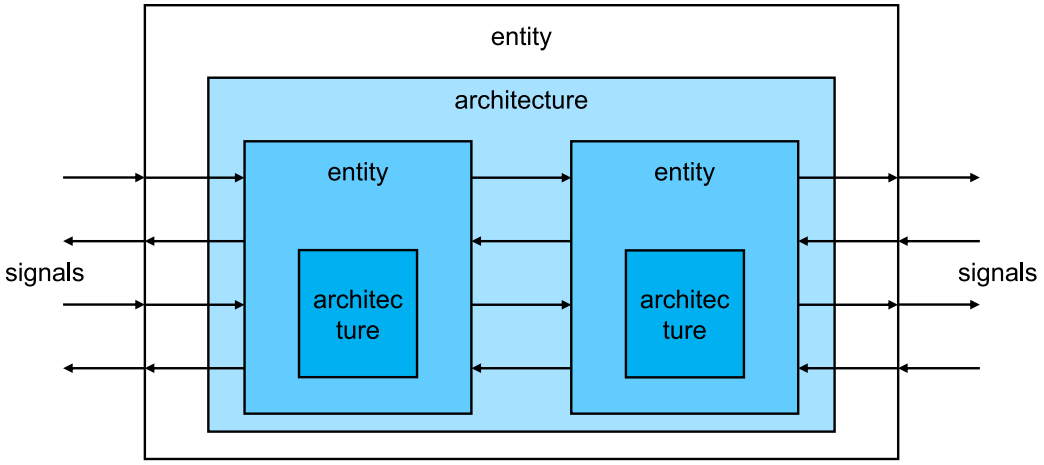
### 7.1.1 Design unit

Een VHDL-beschrijving is opgebouwd uit *design units*. Een design unit kan worden voorgesteld als een black box met ingangen, uitgangen, een gedrag en bestaat uit een entity en een architecture. Dit is te zien in figuur 7.2. Merk op dat voor eenvoudige ontwerpen de gehele VHDL-beschrijving uit één design unit bestaat. Design units wisselen informatie uit via *signals*.

Design units kunnen gekoppeld en ingebed worden in een hogere hiërarchische laag. Dit wordt *structural VHDL* genoemd. Een voorbeeld is te zien in figuur 7.3. Op het hoogste niveau, de *top level* genoemd, is een design unit weergegeven die twee andere design units heeft ingebed. De top level heeft een koppeling met de buitenwereld. De design units die ingebed zijn, wisselen informatie via signalen met elkaar uit en met de top level design unit.




**Figuur 7.2:** *Schematische weergave van een design unit.*




**Figuur 7.3:** *Schematische weergave van design units in een hiërarchie.*

### 7.1.2 Entity en architecture

In listing 7.1 is de volledige beschrijving gegeven van een half adder. De eerste vier regels beschrijven de *entity*. Een entity beschrijft de koppeling (Engels: interface) tussen de design unit en de omgeving waarin het wordt gebruikt. Het kan zijn dat het volledige systeem in één design unit wordt beschreven. In dit geval beschrijft de entity de koppeling met de fysieke buitenwereld. In de *port list* zijn de ingangen en uitgangen beschreven. De signalen `a` en `b` zijn ingangen (keyword **`in`**) van de design unit, `sum` en `carry` zijn uitgangen (keyword **`out`**). Alle signalen zijn van het type `bit`.

```
1  -- VHDL code of a half adder
2  entity half_adder is
3      port( a, b :       in bit;
4            sum, carry : out bit);
5  end entity half_adder;
6
7  architecture logic of half_adder is
8  begin
9      sum   <= a xor b after 5 ns;
10     carry <= a and b after 5 ns;
11 end architecture logic;
```

**Listing 7.1:** *VHDL-beschrijving van een half adder.*

De architecture beschrijft de *werking* (of anders: gedrag) van de design unit. Een architecture moet een naam hebben, in dit geval is dat `logic`. Te zien is dat er twee *concurrent signal assignment statements*. De volgorde van de statements is niet van belang. Hardware werkt immers van nature parallel. Naast de logische functie is ook een vertraging opgegeven. Deze vorm van beschrijven wordt *dataflow VHDL* genoemd.

Bij een entity kunnen meerdere architectures horen. Zo wordt tegemoet gekomen aan de indeling in de abstractieniveaus in de Gajski-Kuhn Y-chart. Een ontwerp kan bijvoorbeeld op zowel register tranfer als op logic niveau worden beschreven. Tijdens simulatie of synthese kan dan voor een bepaalde architecture gekozen worden.

### 7.1.3 Signalen

Signalen (Engels: signals) worden gebruikt om verbindingen (Engels: wires) te modelleren. Signalen hebben een waarde- en een tijdcomponent (bij simulatie), dus van signalen is een timingdiagram te maken. Waarden worden toegekend met de *signal assignment operator* <= (kleiner-dan- en is-gelijk teken). In listing 7.2 zijn drie signaaltoekenningen te zien. De eerste is een eenvoudige toekenning. De tweede en derde toekenningen hebben een **`after`**-clausule. De toekenning gebeurt niet gelijk maar na een vertraging van de opgegeven tijd. De **`after`**-clausule heeft alleen effect bij simulatie. Bij synthese worden ze genegeerd. Het is immers niet te verwachten dat de gegenereerde hardware ook daadwerkelijk deze vertragingen heeft. Noot: een programmeertaal als 'C' kent geen equivalent van een signaal. Opmerking: de namen die bij de port list worden opgegeven zijn per definitie signalen.

```
1 ...
2 r <= a;  -- no after
3 s <= b after 5 ns;
4 t <= c and d after 10 ns;
5 ...
```

**Listing 7.2:** *Voorbeeld signal assignment.*

### 7.1.4 Commentaar

Een commentaarregel in VHDL begint met -- (twee mintekens) en loopt tot het einde van de huidige regel. Blokcommentaar zoals in 'C' is alleen in VHDL-2008 mogelijk. Het is een goede gewoonte om elk VHDL-bestand te beginnen met enkele commentaarregels waarin de naam van het bestand, de functie en de auteur worden vermeld. Zie listing 7.3.

### 7.1.5 Identifiers

Identifiers worden gebruikt om namen te onderscheiden. Zo worden onder andere signalen, entity's en architectures een identifier gegeven. Het is een goede gewoonte om namen te gebruiken die het doel aangeven. Een identifier als `a` zegt weinig over waar deze identifier voor dient. Een identifier als `count_value` zegt veel meer. VHDL stelt geen restricties aan de lengte van een identifier dus er is geen noodzaak om te korte identifiers te gebruiken. VHDL is kast-ongevoelig dus `Count` is hetzelfde als `count`.

```
1  -- Filename:      my_first_description.vhd
2  -- Filetype:      VHDL code (behavior)
3  -- Date:          10 jan 2016
4  -- Update:        -
5  -- Description:   A behavioral description of my description
6  -- Author:        J.E.J op den Brouw
7  -- State:         demo
8  -- Error:         -
9  -- Version:       1.0
10 -- Copyright:     (c)2016, My Company Name
11 --
12 -- This is an example of a introductory comment
```

**Listing 7.3:** *Voorbeeld commentaar.*

Voor een identifier gelden de volgende regels:

- mag alleen bestaan uit letters ('A' t/m 'Z' en 'a' t/m 'z'), cijfers ('0' t/m '9') en de *underscore* ('_');
- moet beginnen met een letter;
- mag niet eindigen met een underscore;
- mag niet meerdere aaneengesloten underscores bevatten;
- mag niet hetzelfde zijn als een keyword.

Geldige identifiers zijn:

```
count_val  clk  xyz_03  input_data
```

Ongeldig als identifier zijn:

```
50cent  _internal_val  start__stop  in  entity
```

### 7.1.6 Standaard datatypes

Net als in een "gewone" programmeertaal kent VHDL een aantal datatypes. VHDL is een *strong typed* taal wat betekent dat elk object (zoals signalen) alleen waardes uit het type mogen aannemen dat bij declaratie is opgegeven. De reden hiervoor is om fouten te voorkomen tijdens toekenningen en beslissingen. Er zijn conversieroutines om van het ene naar het andere type om te zetten. De meest belangrijke types zijn:

| | |
|---|---|
| `boolean` | kan de waarde `true` of `false` hebben; |
| `bit` | kan de waarde '0' of '1' hebben; |
| `integer` | gehele getallen tussen −2147483647 en +2147483647 (32 bits, two's complement); |
| `character` | kan een karakter bevatten. |

Deze datatypes kunnen allemaal gesynthetiseerd worden. In listing 7.4 zijn enkele voorbeelden gegeven van de vier types. Het type `boolean` wordt gebruikt bij logische signalen. Het type kan `true` of `false` zijn. Het type `bit` is het standaard datatype voor binaire

signalen. Een bit kan '0' of '1' zijn. Let hierbij op het gebruik van de apostrofes (*quotes*). Een integer kan elke waarde aannemen tussen −2147483647 en +2147483647. Let op de symmetrie in de integers. De meeste systemen kunnen ook −2147483648 weergeven. Dat komt overeen met de 32-bits two's complement representatie. Denk hieraan tijdens synthese. Als er geen bereik is opgegeven, wordt hardware voor 32-bits getallen gesynthetiseerd. Een bereik kan worden aangegeven met de **range**-clausule. Zie de listing voor een voorbeeld. Een `character` kan één karakter bevatten. De standaard verzameling bestaat uit 128 karakters die overeen komen met de ASCII-code. VHDL-1993 definieert nog 128 additionele karakters.

```
signal b : boolean;
signal t : bit;
signal i : integer range -10000 to 10000;
signal c : character;

b <= true;
b <= false;

t <= '0';  -- mark the quotes
t <= '1' after 5 ns;

i <= -3456;

c <= 'A';
```

**Listing 7.4:** *Voorbeeld types bit, boolean, integer en character.*

Er zijn nog twee andere belangrijke datatypes:

| | |
|---|---|
| `time` | kan een tijdwaarde bevatten (denk aan simulatie); |
| `real` | kan een drijvende kommagetal (floating point) bevatten; |

Time kan niet gesynthetiseerd worden en heeft alleen betekenis bij simulaties. Reals kunnen niet gesynthetiseerd worden. Zie listing 7.5. Er is een nieuw type, `float`, dat wel gesynthetiseerd kan worden. Dat levert echter veel hardware op. We gaan hier niet verder op in.

```
signal t : time;
signal r : real;

t <= 10 ns;    -- 10 nanoseconds, mark the space between 10 and ns
t <= 104.7 us; -- 107.4 microseconds

r <= 1.34e+10;
r <= 3.14159;
```

**Listing 7.5:** *Voorbeeld types time en real.*

### 7.1.7 De types std_ulogic en std_logic

Het type `bit` voldoet niet in alle gevallen (simulatie en synthese). In de praktijk zijn er andere uitgangswaarden dan 0 en 1 mogelijk. Te denken valt aan:

- de waarde is onbekend;
- de waarde is don't care;
- de waarde wordt met pull-up/pull-down signalen beschreven;
- de waarde wordt met tri-state signalen beschreven.

Hiervoor is een apart datatype ontwikkeld: `std_ulogic`. Het is gedefinieerd door IEEE (IEEE 1164) en het moet in een VHDL-beschrijving geladen worden uit een bibliotheek. Het datatype heeft negen verschillende waarden, zie listing 7.6. Let erop dat de waardes met hoofdletters zijn geschreven.

```
type std_ulogic is
    ( 'U',    -- Uninitialized
      'X',    -- Forcing Unknown
      '0',    -- Forcing 0
      '1',    -- Forcing 1
      'Z',    -- High Impedance
      'W',    -- Weak Unknown
      'L',    -- Weak 0
      'H',    -- Weak 1
      '-'     -- Don't care
);
```

**Listing 7.6:** *Het type std_ulogic.*

De waarde 'U' geeft aan dat een signaal (nog) niet geïnitialiseerd is. Dit is vooral van belang bij simulatie. We weten dan immers dat geen enkele *driver*[2] het signaal heeft geactiveerd. De waarde 'X' geeft aan dat twee of meer drivers verschillende waardes op een signaal willen zetten. In de praktijk betekent dit vaak kortsluiting. De waardes '0' en '1' zijn "harde" waardes. Denk hierbij aan een uitgang die intern met de voedingsspanning of met de referentie verbonden is. De waarde 'Z' geeft aan dat een signaal hoog-impedant is. Dat wordt gebruikt om tri-state uitgangen te realiseren. De volgende drie waardes hebben te maken met pull-up en pull-down weerstanden i.c.m. open drain uitgangen. Waarde "W" geeft aan dat een signaal een onbekende waarde heeft. Waardes 'L' en 'H' zijn zachte waardes en worden gebruikt bij pull-down resp. pull-up weerstanden. De laatste waarde '—' betekent dat een signaal een don't care waarde heeft. Noot: let op het gebruik van '—' in vergelijkingen. Zie paragraaf 7.9.

In listing 7.7 is een voorbeeld gegeven van een 2-input NAND (logisch gedrag). De eerste twee regels laden de bibliotheek met de definities van `std_ulogic`. Daarna kan het type gebruikt worden. Er wordt gebruik gemaakt van het interne signaal `int`. Merk op dat `int` geen keyword is in VHDL. Vanwege de concurrency mogen de regels ook verwisseld worden. Het logische gedrag blijft hetzelfde.

Als een `std_ulogic` signaal wordt aangestuurd door meerdere drivers resulteert dat in een simulatiefout. Om toch te kunnen simuleren is het zogenoemde *resolved type* `std_logic` ontwikkeld. Resolved wil zeggen dat als een signaal door twee of meer drivers wordt aangestuurd, een *resolution function* wordt aangeroepen om zo het conflict op te

[2] Een driver is een signaaltoekenning voor een bepaald signaal.

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

entity nand_2 is
    port (a, b: in  std_ulogic;
          c:    out std_ulogic);
end entity nand_2;

architecture logic of nand_2 is
signal int: std_ulogic;   -- intern signaal
begin
    int <= a and b;
    c   <= not int;
end architecture logic;
```

**Listing 7.7:** *VHDL-code van een 2-input NAND (logisch gedrag).*

lossen en de nieuwe waarde van het signaal te bepalen. Voor alle VHDL-beschrijvingen is `std_logic` het aanbevolen type.

In listing 7.8 is de resolution-tabel gegeven. Stel dat twee drivers een '0 en een '1' aan een signaal willen toekennen, dan zien we in de tabel dat het signaal de waarde 'X' krijgt. Als minstens één driver een 'U' heeft, is het resultaat ook 'U'.

```
------------------------------------------------------------------
-- resolution function
------------------------------------------------------------------
CONSTANT resolution_table : stdlogic_table := (
--      ---------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -      |   |
--      ---------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
        ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
        ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
        ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
        ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
        ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
        ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
    );
```

**Listing 7.8:** *VHDL-code resolution tabel.*

Een voorbeeld van een 3-input NOR (logisch gedrag) met `std_logic` is gegeven in listing 7.9. Ook hier is weer gebruik gemaakt van een intern signaal.

### 7.1.8 De types unsigned en signed

Specifiek voor het rekenen met bits zijn de types `unsigned` en `signed` ontwikkeld. Het type `unsigned` wordt gebruikt voor unsigned getallen en het type `signed` wordt gebruikt voor two's complement getallen. Het rekenen met deze types wordt besproken in paragraaf 7.7.

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

entity nor_3 is
    port (a, b, c: in  std_logic;
          d:       out std_logic);
end entity nor_3;

architecture logic of nor_3 is
signal int: std_logic;
begin
    int <= a or b or c;
    d   <= not int;
end architecture logic;
```

**Listing 7.9:** *VHDL-code van een 3-input NOR (logisch gedrag).*

### 7.1.9 Declaratie van types en subtypes

Het is in VHDL mogelijk om een eigen type of subtype de declareren. Dat kan door middel van de keywords **`type`** en **`subtype`**. Zo kunnen we het type `trilogic` en het signaal `tri` declareren en een waarde aan het signaal toekennen. Zie listing 7.10.

```vhdl
type trilogic is ('Z', '0', '1');
signal tri : trilogic;
...
tri <= '0';
```

**Listing 7.10:** *Declaratie van het type trilogic.*

Als een nieuw type gebaseerd is op een bestaand type dan gebruiken we het keyword **`subtype`**. In listing 7.11 is te zien hoe we een de subtypes `byte` en `short` als een integer declareren die een bereik hebben van 0 t/m 255 respectievelijk −32768 t/m 32767.

```vhdl
subtype byte is integer range 0 to 255;
subtype short is integer range -32768 to 32767;
signal datab : byte;
signal datas : short;
...
datab <= 151;
datas <= -9384;
```

**Listing 7.11:** *Declaratie van de subtypes byte en short.*

Noot: in VHDL zijn alle types gebaseerd op drie mogelijkheden: getallen, karakters of woorden. Zo zijn de types `boolean`, `bit` en `integer` als volgt gedeclareerd.

```vhdl
type BOOLEAN is (FALSE,TRUE);
type BIT is ('0', '1');
type INTEGER is range -2147483647 to 2147483647;
```

**Listing 7.12:** *Declaratie van enkele standaard types.*

Het is in VHDL ook mogelijk om een eigen enumeratietype te declareren. In listing 7.13 zijn de enumeratietypes `color_type` en `level_type` te zien. Enumeratietypes worden veel gebruikt bij toestandsmachines. Signalen van een type kunnen niet een waarde krijgen van een ander type. In de listing is daar een voorbeeld van gegeven.

```
type color_type is (red, blue, green, yellow);
type level_type is (low, high, unknown);
...
signal mycolor : color_type;
signal mylevel : level_type
    ...
    mycolor <= yellow;
    mylevel <= high;
    ...
    mycolor <= green;
    mylevel <= unknown;
    ...
    mycolor <= low;   -- ERROR!
    mylevel <= red;   -- ERROR!
```

**Listing 7.13:** *Voorbeelden van enumeratietypes.*

### 7.1.10 Vectoren

Het is mogelijk om een *array* van een bepaalde datatype te declareren. Een array is een manier om bij elkaar horende gegevens te groeperen en gemakkelijk te bewerken. Een andere, in de computerindustrie gebruikte term, is *bus*. Een naam die in VHDL wordt gebruikt is *vector*.

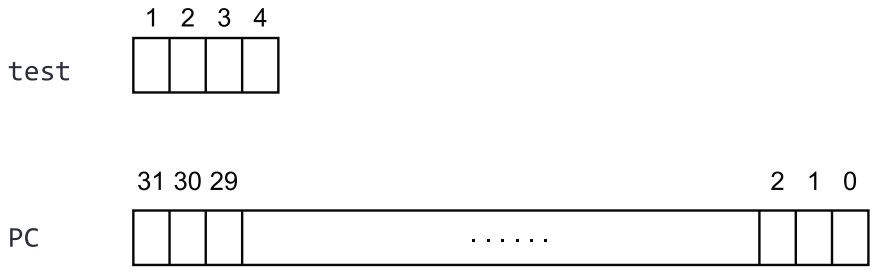
In listing 7.14 zijn twee types gedeclareerd. Het type `test_type` is een vector met vier *elementen* van het type `bit`. Binnen de haakjes staat het bereik (Engels: range) opgegeven. Zo heeft het type `test_type` het bereik van 1 tot en met 4. Het keyword **`to`** geeft aan dat de vector een oplopend bereik heeft. Het type `longword` heeft 32 elementen van het type `std_logic` en heeft een aflopend bereik (keyword **`downto`**). Het gebruik van **`downto`** sluit heel goed aan bij manier waarop binaire getallen worden weergegeven. Het meest linker bit is het meest significante bit. Dit is weergegeven in figuur 7.4.

```
type test_type is array (1 to 4) of bit;
type longword is array (31 downto 0) of std_logic;
signal test: test_type;
signal PC: longword;
```

**Listing 7.14:** *Twee voorbeelden van vectoren.*

Vectoren van de types `bit` en `std_logic` worden zo vaak gebruikt dat er twee nieuwe datatypes zijn gedeclareerd, de types `bit_vector` en `std_logic_vector`, zodat een en ander eenvoudiger geschreven kan worden. Toekennen gaat eenvoudig middels een *bitstring*. In listing 7.15 is een tweetal voorbeelden gegeven.

In listing 7.16 zijn enkele voorbeelden van vectoren te zien. In de eerste twee toekenningen wordt steeds één enkel element gebruikt. In de twee laatste toekenningen wordt




**Figuur 7.4:** *Schematische weergave van vectoren.*

```
signal maximum: bit_vector (3 downto 0);
signal counter: std_logic_vector (15 downto 0);

    ...
    maximum <= "1011";                  -- the decimal number 11
    counter <= "1000000110010011";      -- the hex number 8193
    ...
```

**Listing 7.15:** *Twee voorbeelden van vectoren.*

een deel van de vector als één geheel gebruikt. Dit wordt een *vector slice* genoemd. Het aantal elementen aan de linkerkant van het toekenningssymbool moet even groot zijn als aan de rechterkant.

```
signal test: bit_vector (0 to 3);
signal PC: std_logic_vector (31 downto 0);

    ...
    test(3) <= '1';                     -- index 3 of test is '1'
    test(2) <= test(1);                 -- copy index 1 to index 2
    ...
    PC(31 downto 24) <= "00000000";     -- bits 24 through 31 are 0
    PC(15 downto 12) <= PC(3 downto 0); -- copy 4 bits
    ...
```

**Listing 7.16:** *Vier voorbeelden van vectoren.*

Het is mogelijk om een vector samen te stellen uit (delen van) andere vectoren en constanten. Hiervoor wordt de *concatenation operator* (`&`) gebruikt. Zie listing 7.17. Vector `int_vec` wordt samengesteld uit een deel van `PC`, alle bits van `int_no` en de bitstring `"000"`. Het beschrijven van een schuifoperatie is zo eenvoudig te realiseren. Vector `shiftreg` wordt één plaats naar links geschoven en aangevuld met een '0'. Het meest significante bit verdwijnt. We noemen dit "eruit geschoven". Schuiven wordt uitgebreid behandeld in hoofdstuk 8.

Soms moeten in een vector meerdere elementen op een bepaalde waarde worden gezet. Dat kan elegant met een *aggregate*. In listing 7.18 wordt aan signaal `register_a` drie keer een waarde toegekend. Bij de eerste toekenning wordten alle bits op '0' gezet. Dat is veel handiger en sneller dan een bitstring met 64 nullen. In de tweede toekenning worden bits 63, 62 en 61 op een '1' gezet en de rest op '0'. Het laatste voorbeeld is

```
signal PC, int_vec: std_logic_vector (31 downto 0);
signal int_no: std_logic_vector (2 downto 0);
signal shiftreg: std_logic_vector (7 downto 0);

    ...
    int_vec  <= PC(31 downto 6) & int_no & "000";
    ...
    shiftreg <= shiftreg(6 downto 0) & '0'; -- shift left
    ...
```

**Listing 7.17:** *Twee voorbeelden van vectoren.*

identiek aan het tweede voorbeeld.

```
signal register_a : std_logic_vector (63 downto 0);

    ...
    -- alle bits op '0'
    register_a <= (others => '0');
    ...
    -- bits 63-61 op '1', de rest op '0'
    register_a <= (63 => '1', 62 => '1', 61 => '1', others => '0');
    ...
    -- hetzelfde als de vorige
    register_a <= (63|62|61 => '1', others => '0');
```

**Listing 7.18:** *Voorbeelden van aggregates.*

### 7.1.11 Attributes

Met een attribute kan een bepaalde specificatie van een data-object (signaal, vector) worden opgevraagd. Er zijn er zo'n 30. Attributes worden geschreven na de signaalnaam of vectornaam en begint altijd met een apostrof (Engels: quote) gevolgd door de attributenaam. We zullen alleen de belangrijkste bespreken. Zie tabel 7.1.

**Tabel 7.1:** *Enkele attributes van types, vectoren en signalen.*

| | |
|---|---|
| `typenaam'left` | Geeft het meest linker elementwaarde terug. |
| `typenaam'right` | Geeft het meest rechter elementwaarde terug. |
| `typenaam'high` | Geeft de waarde van het grootste element terug. |
| `typenaam'low` | Geeft de waarde van het kleinste element terug. |
| `vectornaam'left` | Geeft het linker elementnummer terug. |
| `vectornaam'right` | Geeft het rechter elementnummer terug. |
| `vectornaam'high` | Geeft het hoogste elementnummer terug. |
| `vectornaam'low` | Geeft het laagste elementnummer terug. |
| `vectornaam'length` | Geeft het aantal elementen van een vector terug. |
| `vectornaam'range` | Geeft het bereik van de elementen terug. |
| `signaalnaam'event` | Geeft een `true` als er een verandering op het signaal is geweest. |

In listing 7.19 is een aantal voorbeelden gegeven. Merk op dat in een aantal gevallen `'low` en `'high` equivalent zijn aan `'left` en `'right`.

```vhdl
type fruit_type is (apple, banana, grapes, pear);
subtype level_type is integer range -5 to 5;
type test_type is array (1 to 4) of std_logic;
type longword_type is array (31 downto 0) of std_logic;

signal level : level_type;
signal fruit : fruit_type;
signal test : test_type;
signal PC : longword_type;
signal clk : std_logic;

    level_type'low      -- returns -5
    level_type'high     -- returns 5
    level_type'left     -- return -5
    level_type'right    -- returns 5

    fruit_type'left     -- returns apple
    fruit_type'right    -- returns pear
    fruit_type'low      -- returns apple
    fruit_type'high     -- returns pear

    test'length         -- returns 4
    test'range          -- returns 1 to 4
    test'left           -- returns 1
    test'right          -- returns 4

    PC'length           -- returns 32
    PC'range            -- returns 31 downto 0
    PC'left             -- returns 31
    PC'right            -- returns 0
    PC'low              -- returns 0
    PC'high             -- returns 31

    clk'event           -- returns true if an event on clk
    std_logic'left      -- returns 'U'
    std_logic'right     -- returns '-'
```

**Listing 7.19:** *Een aantal voorbeelden van attributes.*

## 7.2 Concurrent VHDL

Hardware werkt van nature parallel en het is dan ook niet verwonderlijk dat VHDL parallellisme (of concurrency) ondersteunt. Alle statements binnen een architecture worden parallel uitgevoerd (denk hierbij aan simulatie). We hebben al de eenvoudige toekenning gezien. We bespreken nog twee andere toekenningen.

### 7.2.1 Conditional Signal Assignment

Met het CSA-statement kunnen waardes aan een signaal worden toegekend op basis van *voorwaarden*. In listing 7.20 is een voorbeeld van een CSA-statement te zien. De syntax bestaat uit het keyword **`when`** en kan meerdere **`else`**...**`when`** keywords bevatten. De volgorde van de regels is van belang. In het voorbeeld wordt eerst de regel met signaal

`a` verwerkt. Als blijkt dan signaal `en` '1' is, wordt de toekenning `s <= a` gedaan. Zo niet, dan wordt de tweede regel verwerkt. Als alle voorwaarden niet waar zijn wordt de toekenning `s <= d` gemaakt. De laatste regel moet niet vergeten worden anders worden er latches gegenereerd. Merk op dat in de voorwaarden steeds andere signalen worden gebruikt. Dat is niet strikt noodzakelijk.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity prior is
5      port (a,b,c,d,en,data,reset: in std_logic;
6            s : out std_logic);
7  end entity prior;
8
9  architecture csa of prior is
10 begin
11     s <= a  when en = '1' else
12          b  when data = '0' else
13          c  when reset = '1' else
14          d;
15 end architecture csa;
```

**Listing 7.20:** *Voorbeeld van een Conditional Signal Assignment.*

Met het CSA-statement is het mogelijk om heel eenvoudig een "groter dan"-schakeling te beschrijven. Zie listing 7.21. Als blijkt dat `a` groter is dan `b`, wordt de toekenning `s <= '1'` uitgevoerd, anders wordt de toekenning `s <= '0'` uitgevoerd.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity agtb is
5      port (a, b : in std_logic_vector (3 downto 0);
6            s    : out std_logic);
7  end entity agtb;
8
9  architecture csa of agtb is
10 begin
11     s <= '1' when a > b else '0';
12 end architecture csa;
```

**Listing 7.21:** *Voorbeeld van groter dan met een CSA-statement.*

### 7.2.2 Selected Signal Assignment

Met het SSA-statement kunnen toekenning aan signalen worden gedaan op basis van voorwaarden van één enkele signaal of een vector. In listing 7.22 is de beschrijving te zien van een 4x1-multiplexer. De volgorde van de regels is niet van belang omdat alle voorwaarden elkaar uitsluiten. Er zijn nooit twee voorwaarden tegelijk waar. De regel met **`when others`** is noodzakelijk om een signaal een waarde toe te kennen als geen van de voorwaarden waar is. In dit geval lijkt het overbodig omdat alle vier de combinaties

van twee bits worden beschreven maar bedenk dat tijdens simulatie een signaal ook andere waardes kan hebben.

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

entity mux4 is
    port (in0,in1,in2,in3: in std_logic;
          sel: in std_logic_vector (1 downto 0);
          z : out std_logic);
end entity mux4;

architecture ssa of mux4 is
begin
    with sel select
    z <= in0  when "00",
         in1  when "01",
         in2  when "10",
         in3  when "11",
         'X'  when others;
end architecture ssa;
```

**Listing 7.22:** *Voorbeeld van een SSA-statement.*

Met het SSA-statement kan heel eenvoudig een schakeling worden beschreven met een waarheidstabel. Listing 7.23 beschrijft een full adder op basis van een waarheidstabel. Het is te zien dat ook vectoren mogen worden gebruikt in de voorwaarden en toekenningen.

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

entity full_adder_ssa is
    port (a_b_cin: in std_logic_vector (2 downto 0);
          cout_s : out std_logic_vector (1 downto 0));
end entity full_adder_ssa;

architecture truth_table of full_adder_ssa is
begin
    with a_b_cin select
    cout_s <= "00" when "000",
              "01" when "001",
              "01" when "010",
              "10" when "011",
              "01" when "100",
              "10" when "101",
              "10" when "110",
              "11" when "111",
              "XX" when others;
end architecture truth_table;
```

**Listing 7.23:** *Voorbeeld van een waarheidstabel met een SSA-statement.*

Er is nog een derde concurrent statement, **process**. Zie hiervoor paragraaf 7.3.

### 7.2.3 VHDL delays

Het beschrijven van vertragingen en minimale pulsbreedte wordt gedaan door middel van de *transport delay model* en de r*eject inertial delay model*. De transport delay model wordt gebruikt als een signaal alleen vertraagd moet worden zoals bij bedrading op een chip (wire delay). De reject inertial delay model wordt gebruikt om vertragingen én minimale pulsbreedte van logica (poorten etc.) te modelleren. Informatie over (de werking van) het vertragingsmodel kan gevonden worden in de VHDL Language Manual 2008, par.10.5. [10]. Merk op dat delays alleen van toepassing zijn tijdens simulatie.

In listing 7.24 zijn twee voorbeelden van signal delays te zien. Het eerste voorbeeld laat een exacte vertraging van 10 ns zien. Een puls, hoe kort dan ook, wordt doorgegeven. In het tweede voorbeeld is te zien dat er meerdere toekenningen plaatsvinden. Hiermee is het mogelijk om een pulstrein te realiseren, bijvoorbeeld voor gebruik tijdens een simulatie. De eerste toekenning vindt plaats na 2 ns. Daarvóór is er nog geen toekenning gedaan. Een signaal van het type `std_logic` heeft dan nog de waarde 'U'. Na 7 ns wordt het signaal op '0' gezet zodat een puls van 5 ns wordt gerealiseerd. Daarna worden nog twee pulsen gerealiseerd. Na 21 ns wordt weer een '0' aan het signaal toegekend. Tijdens de rest van de simulatie blijft het signaal op deze waarde.

```
1  -- exact delay of 10 ns for sig
2  dly <= transport sig after 10 ns;
3
4  -- create a signal delay for simulation
5  x <= transport '1' after  2 ns, '0' after  7 ns,   -- pulse 5 ns
6                 '1' after 11 ns, '0' after 15 ns,   -- pulse 4 ns
7                 '1' after 18 ns, '0' after 21 ns;   -- pulse 3 ns
```

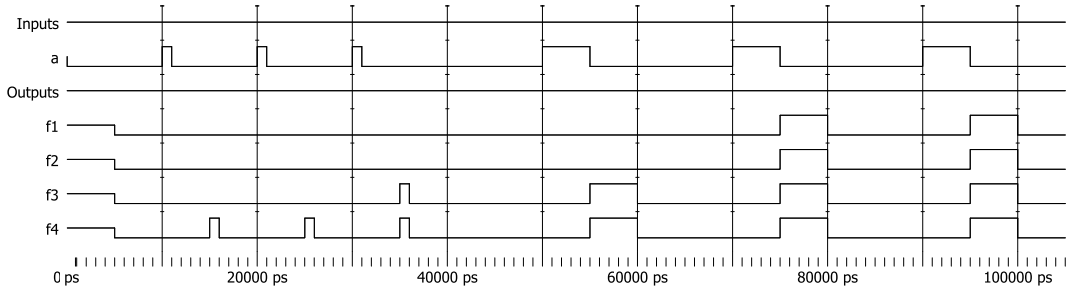
**Listing 7.24:** *Voorbeelden van signal delays.*

Vertragingen en minimale pulsbreedte voor logica worden gemodelleerd met de keywords **`reject`**, **`inertial`** en **`after`**. Een aantal voorbeelden is gegeven in listing 7.25.

```
1  -- Two equal statements, keyword inertial is optional
2  f1 <= inertial x  after  5 ns;   -- delay 5 ns
3  f1 <=          x  after  5 ns;   -- same
4
5  -- Pulse width > 1 ns
6  f2 <= reject 1 ns inertial x  after  5 ns;
7
8  -- Pulse width >= 5 ns
9  f3 <= reject 5 ns inertial x  after  5 ns;
10
11 -- Pulse width 0 ns is same as transport
12 f4 <= reject 0 ns inertial a after 10 ns;
13 f4 <= transport a after 10 ns;
14
15 -- Assignments can have logic constructions
16 f5 <= reject 2 ns inertial a or b after 10 ns;
```

**Listing 7.25:** *Voorbeelden van signal delays.*

In de eerste twee regels wordt signaal `f1` 5 ns vertraagd. De statements zijn identiek; het keyword **`inertial`** is optioneel. Het nadeel hiervan is dat (uitgangs-)pulsen korter dan 5 ns niet worden doorgegeven (minimale pulsbreedte is gelijk aan vertraging). Om minimale pulsbreedte te modelleren moeten de keywords **`reject`** en **`inertial`** gebruikt worden. Dit is te zien in de toekenning voor signaal `f2`. De vertraging van signaal `f2` is 5 ns en de minimale pulsbreedte van signaal `x` moet *groter* dan 1 ns zijn. Als de minimale pulsbreedte even lang is als de vertragingstijd, dan wordt de puls doorgegeven. Dat is te zien in de toekenning van signaal `f3`. Een puls van precies 5 ns wordt dus doorgegeven. In figuur 7.5 is de simulatie van de uitgangswaarden te zien.




**Figuur 7.5:** *Enkele pulsbreedtes.*

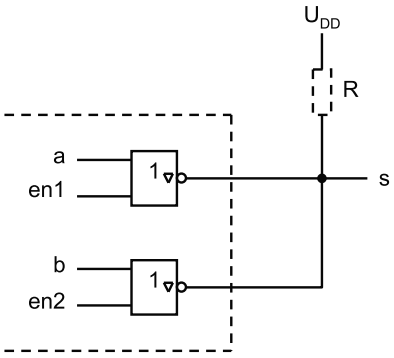
Een minimale pulsbreedte van 0 ns gedraagt zich identiek met het gebruik van het keyword **`transport`**. De toekenningen aan signaal `f4` zijn dus hetzelfde. Let op dat de minimale pulsbreedte over de *uitgang* gaat (het betreft een toekenning). Als de OR-operatie voor signaal `f5` een puls produceert korter dan 2 ns, is deze niet te zien in de simulatie.

VHDL kent nog een derde delay-model: de *delta delay*. Deze delay wordt gebruikt als er geen expliciete **`after`** wordt opgegeven en staat gelijk aan **`after`** `0 ns`. Hierdoor wordt iets duidelijk: toekenningen aan een signaal kunnen dus alleen in de (simulatie-)toekomst worden gegeven. Zie als voorbeeld listing 7.2. Delta delay is nauw gekoppeld aan simulatie. We behandelen de werking van delta delays in paragraaf 7.5. Voor nu is het genoeg om te weten dat de simulator een techniek heeft om concurrency na te bootsen.

### 7.2.4 Voorbeeld: tri-state buffers

Eerder zijn de tri-state buffers al behandeld. De tri-state uitgang kan naast een logische 0 of 1 ook een hoog-impedante uitgangswaarde aannemen. Dit wordt ook wel *High-Z* genoemd. In figuur 7.6 is een mogelijke schakeling te zien. De signalen `en1` en `en2` besturen de uitgangen. Een logische 1 zorgt voor de normale bufferwerking, een logische 0 zorgt ervoor dat de uitgang hoog-impedant wordt. De uitgang is dan als het ware van de gemeenschappelijke verbinding losgekoppeld, De signalen `en1` en `en2` mogen niet tegelijk logisch 1 zijn. Ze mogen wel tegelijk logisch 0 zijn. In dat geval zorgt de pull-up weerstand voor een logische 1 voor signaal `s`.

Het is mogelijk om aan een signaal de waarde 'Z' toe te kennen met behulp van een Conditional Signal Assignment. Zie listing 7.26. Het is duidelijk te zien dat signaal `s` wordt aangestuurd door twee drivers. Dat levert geen problemen op zolang `en1` en `en2` niet tegelijk logisch 1 zijn. Als dat wel gebeurt dan hangt het af van de waarden van `a` en




**Figuur 7.6:** *Aansluiting van tri-state NOT-poorten aan een gemeenschappelijke buslijn.*

`b` wat de uiteindelijke waarde van `s` wordt. Als `a` en `b` dezelfde waarden hebben, zal `s` die waarde ook hebben. Bij verschillende waarden zal `s` een 'X' krijgen toegewezen (tenzij `a` en/of `b` de waarde 'U' hebben). Zie hiervoor de tabel in listing 7.8. In een simulator is dit goed te zien en duidt vaak op een ontwerpfout.

```
signal a, b, en1, en2: std_logic;

begin
    s <= a when en1 = '1' else 'Z';
    s <= b when en2 = '1' else 'Z';
end;
```

**Listing 7.26:** *VHDL-beschrijving tri-state buffers.*

## 7.3 Sequential VHDL

Concurrent signal assignments statements worden gebruikt om digitale systemen te specificeren op logisch niveau. De uitgangswaarden worden direct berekend uit de ingangswaarden, eventueel met een vertragingstijd. Als we complexe systemen willen beschrijven zoals microprocessors en geheugens dan zijn deze taalconstructies niet toereikend. Bij veel digitale systemen wordt gebruik gemaakt van intern geheugen. De inhoud is dan intern beschikbaar maar is voor de buitenwereld niet te zien. De geheugens willen we bijvoorbeeld door middel van een vector realiseren. Dit laat zich niet makkelijk in concurrent signal assignments statements beschrijven.

Verder ondersteunen de concurrent signal assignments geen hogere abstractieniveaus. We willen deze complexe systemen niet op logisch niveau beschrijven. Het tijdgedrag en logisch gedrag van de systemen zijn op deze manier lastig te realiseren. Zo is het invoer/uitvoer-gedrag niet makkelijk te vertalen naar concurrent signal assignments, willen we modellen gebruiken waarmee toestanden te beschrijven zijn en willen we gebruik maken van complexe datastructuren. Er zijn dus krachtige taalconstructies nodig. VHDL ondersteunt dat middels de **`process`**-constructie. Noot: we beschouwen eerst simulatie, later synthese.

### 7.3.1 Proces

Hardware is van nature concurrent, maar het *gedrag in tijd* is vaak sequentieel. VHDL kent de mogelijkheid om het gedrag van hardware sequentieel te beschrijven. Dit kan door de code te plaatsen binnen een *proces*. Een proces kan worden gezien als één complexe signal assignment statement. Processen communiceren met elkaar via signalen. Statements binnen een proces worden sequentieel uitgevoerd. De signal assignment statements worden dus *sequentiële signal assignment statements* genoemd. De code binnen een proces kent diverse taalconstructies en lijkt op Pascal en Ada. Het is bijvoorbeeld mogelijk om gebruik te maken variabelen (zie verderop). Een proces wordt concurrent uitgevoerd met andere concurrent signal assignment statements.

In het proces in listing 7.27 zijn drie toekenningen te zien. De statements worden sequentieel uitgevoerd en dat betekent dat als het proces doorlopen is (denk aan simulatie) dat signaal `b` de waarde '0' heeft en `a` de waarde '0' heeft. De tussentijdse toekenning van '1' aan `a` wordt dus overschreven en is niet te zien. Merk op dat de uiteindelijke toekenningen één delta later plaatsvinden. Er is immers geen **`after`** gebruikt.

```
1  architecture example of example_process is
2  begin                        -- architecture
3      process is               -- process statement
4      begin                    -- process body begins
5          ...
6          A <= '1';
7          B <= '0';
8          A <= '0';
9          ...
10     end process;             -- process body ends
11 end architecture example;    -- architecture ends
```

**Listing 7.27:** *Process.*

In de listings 7.28 en 7.29 zijn twee processen te zien. Beide beginnen met een label voor het keyword **`process`**. De labels zijn optioneel. De opsomming `(A, B)` is een *sensitivity list*. Het proces is alleen gevoelig voor de signalen `A` en `B`. Als er een verandering op A en/of B komt, wordt het proces doorlopen. Dit kan zijn bij simulatie en synthese.

```
1  begin
2      NAAM: process (A,B) is
3      begin
4          ...
5      end process;
6  end;
```

**Listing 7.28:** *Process.*

```
1  begin
2      NAAM2: process (B,C) is
3      begin
4          ...
5      end process;
6  end;
```

**Listing 7.29:** *Process.*

Processen werken onderling concurrent. Als signaal `B` verandert, worden beide processen eenmaal doorlopen. Het doorlopen van een proces gebeurt in 0 ns simulatietijd en kan veranderingen in de toekomst genereren.

### 7.3.2 If

Met de keyword **if** wordt een beslissing beschreven. Gebruik van **else** en **elsif** is mogelijk. Zie listing 7.30. Hoewel dit een sequentiële stijl van beschrijven is, representeert de code combinatorische logica.

```
IF_EX1: process (A, B) is
begin
    if A = '1' and B = '0' then
        P <= '0';
    elsif A = '0' and B = '1' then
        P <= '0';
    else
        P <= '1';       -- don't forget
    end if;
end process;
```

**Listing 7.30:** *Voorbeeld van een if-statement.*

Bij het beschrijven van combinatoriek mag de **else** mag niet vergeten worden. Stel dat de **else** niet in de code voorkomt dan wordt er geen nieuwe waarde aan P toegekend bij $AB = 00$ of $AB = 11$. Dat zorgt ervoor dat er latches worden gesynthetiseerd om de laatst ingebrachte waarde voor P te onthouden. Synthesesoftware geeft hiervoor meestal een waarschuwing.

De code in listing 7.31 is ook volgens het boekje. Hier worden eerst de signalen R en S op een *default* waarde gezet en daarna worden ze eventueel gewijzigd als gevolg van de **if**-statements. Deze vorm is zeer geschikt om signalen die maar weinig veranderen te beschrijven en het zorgt ervoor dat er nooit latches worden gesynthetiseerd.

```
IF_EX2: process (A,B) is
begin
    R <= '0';     -- default value
    S <= '1';     -- default value
    if A = '1' and B = '1' then
        R <= '1';
        S <= '0';
    end if;       -- no else!
    if A = '1' and B = '0' then
        R <= '1';
    end if;       -- no else!
end process;
```

**Listing 7.31:** *Voorbeeld van een if-statement.*

### 7.3.3 Case

Met **case** kan het testen van één signaal worden beschreven. Een voorbeeld is gegeven in listing 7.32. Deze wijze sluit goed aan bij het beschrijven van waarheidstabellen. De keywords **when others** is een *catch-all* als eerdere condities niet waar zijn.

```
CASE_EX: process (SEL) is
begin
    case SEL is                         -- SEL is a 2-bit vector
        when "00" =>   F <= "000";      -- F is a 3-bit vector
        when "01" =>   F <= "110";
        when "10" =>   F <= "110";
        when "11" =>   F <= "011";
        when others => F <= "000";      -- catch-all
    end case;
end process;
```

**Listing 7.32:** *Voorbeeld van een case-statement.*

### 7.3.4 Variabele

Binnen een proces is het mogelijk om lusconstructies te gebruiken. Zo is het mogelijk om **for**-loops te gebruiken. Bij deze programmeerconstructie wordt gebruik gemaakt van een lusvariabele. Dat kan geen signaal zijn, want de toekenning aan signalen gebeurt altijd in de simulatietoekomst. Er is een nieuw data-object nodig: *variable*. Variabelen hebben alleen waarden en bestaan alleen binnen een proces. Toekenning van een waarde gebeurt gelijk op het moment van het uitvoeren van de toekenningsopdracht := (dubbele-punt-is-gelijkteken). Variabelen kunnen dezelfde datatypes aannemen als signalen. Vectoren zijn ook mogelijk. In listing 7.33 is een voorbeeld van het gebruik van variabelen te zien.

```
process (a, b) is
variable test : boolean;       -- declaration
variable temp : std_logic      -- declaration
begin

    ...
    test := true;              -- assignment with :=
    temp := a;

    ...
    if test = true then        -- use in expression
        temp := b;
        ...

    if temp = '1' then
    ...
```

**Listing 7.33:** *Voorbeeld van het gebruik van variabelen.*

### 7.3.5 For

Met een **for**-statement kan een exact aantal keer een lus worden doorlopen. For wordt onder andere gebruikt om langs een vector te lopen. Er kan oplopend (**to**) of aflopend (**downto**) geïtereerd worden. De lusvariabele is tijdens het uitvoeren van de *body* van de lus een constante en kan dus niet aangepast worden. Daarnaast hoeft de lusvariabele niet gedeclareerd te worden.

In listing 7.34 staat de beschrijving van een 8-input NOR. Van de vector `x` moet de NOR-functie bepaald worden. We gebruiken een **`for`**-statement om langs alle elementen van `x` te lopen. Als lusvariabele gebruiken we `i`. Die hoeft niet gedeclareerd te worden. In het begin zetten we variabele `p` op '0'. Bij elke doorloop van de **`for`**-lus wordt de OR bepaald van `p` en een element van `x`. Nadat de lus is doorlopen heeft `p` de waarde van de 8-input OR van `x`. Aan signaal `q` wordt de inverse van `p` toegekend. Deze beschrijving is eenvoudig te wijzigen in een ander aantal ingangen.

```
signal x : bit_vector (7 downto 0);
signal q : bit;
    ...
    process (x) is                  -- sensitive for x
    variable p : bit;               -- only p, no need to declare i
    begin
        p := '0';                   -- initialize to 0
        for i in 7 downto 0 loop    -- i = 7,6,5,4,3,2,1,0
            p := p or x(i);         -- OR p with one of the elements
        end loop;
        q <= not p;                 -- signal assignment
    end process;
```

**Listing 7.34:** *VHDL-beschrijving van een 8-input NOR.*

Al eerder is de attribute **`'range`** besproken. Deze attribute geeft een bereik van een bepaald type (signaal, variabele) of vector terug. We kunnen de **`'range`**-attribute gebruiken in een **`for`**-lus ter vervanging van de **`for`**-range. In listing 7.35 is de **`for`**-lus uit listing 7.34 herschreven met een **`'range`**-attribute.

```
        -- iterate over all elements of x
        for i in x'range loop
            p := p or x(i);
        end loop;
```

**Listing 7.35:** *VHDL-beschrijving van een 8-input NOR.*

### 7.3.6 While

Vaak is het nodig om een stuk code te herhalen als een bepaalde conditie waar is. Dit houdt in dat het aantal herhalingen niet van te voren vastligt zoals bij **`for`** het geval is. Hiervoor wordt de **`while`**-lusconstructie gebruikt. Merk op dat in de **`while`**-lus variabelen mogen worden aangepast (dat mag bij **`for`** ook alleen de lusvariabele mag niet worden aangepast). De **`while`**-lus is een krachtig instrument bij simulatie. In listing 7.36 is een voorbeeld van een **`while`**-lus te zien.

### 7.3.7 Gated D-latch

In listing 7.37 is de code van een gated D-latch te zien. Het proces is gevoelig voor de signalen `en` en `d`. In de code staat één enkele **`if`**-statement zonder **`else`**. Het geheel werkt als volgt. Als er een verandering plaatsvindt op `en` en/of `d` wordt het proces doorlopen.

```
j := 1;
while j<10 loop
    ...
    if a = '1' then
        j := j+2;
    else
        j := j+1;
    end if;
    ...
end loop;
```

**Listing 7.36:** *VHDL-beschrijving van while-lus.*

Het **if**-statement zorgt ervoor dat `d` aan `z` wordt toegekend als `en` '1' is. Als `en` '0' is, blijft de waarde van `z` onveranderd. Er is immers geen alternatief opgegeven. Noot: over het algemeen worden geheugens niet opgebouwd met latches onder meer door timingproblemen, (zie hoofdstuk 9).

```
LATCH: process (en, d) is
begin
    if en = '1' then
        z <= d;
    end if;  -- no else
end process;
```

**Listing 7.37:** *VHDL-beschrijving van een gated D-latch.*

### 7.3.8 D-flipflop

Met een **if**-statement kan een flipflop beschreven worden. In listing 7.38 is de beschrijving een positive edge triggered D-flipflop met behulp van de *signal attribute* `'event` te zien. De toekenningsopdracht `Q <= D` wordt uitgevoerd als er een verandering is op signaal `clk` én `clk = 1`. Dat is dus nadat de verandering heeft plaatsgevonden.

Merk op dat signaal `d` niet is opgenomen in de sensitivity list. Dat is ook niet nodig (maar niet fout). Een verandering op (alleen) `d` zal de toekenning `Q <=` niet activeren want er is geen verandering op `clk`.

```
DFF: process (clk) is  -- sensitive to clk only!
begin
    if clk'event and clk = '1' then
        Q <= D;
    end if;  -- no else
end process;
```

**Listing 7.38:** *VHDL-beschrijving van een positive edge triggered D-flipflop.*

De programmeerconstructies `clk'event and clk = '1'` en `clk'event and clk = '0'` worden vaak gebruikt zodat er twee functies voor zijn gemaakt. Deze werken ook beter in een simulatieomgeving waar een signaal andere waarden dan '0' en '1' kan hebben.

`rising_edge(clk)` levert true als er een opgaande flank is geconstateerd.

`falling_edge(clk)` levert true als er een neergaande flank is geconstateerd.

De beschrijving van de D-flipflop kan worden uitgebreid met een asynchrone reset. Dit is te zien in listing 7.39. Het proces is gevoelig voor de signalen `clk` en `areset`. Als eerste wordt gekeken of de reset geactiveerd is. Het betreft hier een actief hoge reset. De uitgang `Q` wordt op '0' gezet bij actieve reset (dit mag ook '1' zijn afhankelijk van de toepassing). Als de reset niet geactiveerd is, wordt gekeken of er een opgaande flank is geweest. Dan volgt eigenlijk de beschrijving van de flipflop.

```
DFF: process (clk,areset) is  -- sensitive to clk and reset
begin
    if areset = '1' then
        Q <= '0';
    elsif rising_edge(clk) then
        Q <= D;
    end if;  -- no else
end process;
```

**Listing 7.39:** *VHDL-beschrijving van een positive edge triggered D-flipflop met asynchrone reset.*

Met behulp van de klokflankbeschrijving kan een schuifregister worden beschreven. Code is te vinden in listing 7.40. De toekenning gaat als volgt. Op de opgaande flank wordt de huidige waarde van `DATA_IN` toegekend aan `A`, de huidige waarde van `A` toegekend aan `B`, de huidige waarde van `B` toegekend aan `C` en de huidige waarde van `C` toegekend aan `DATA_OUT`. Merk up dat de huidige waarde van `DATA_OUT` verloren gaat.

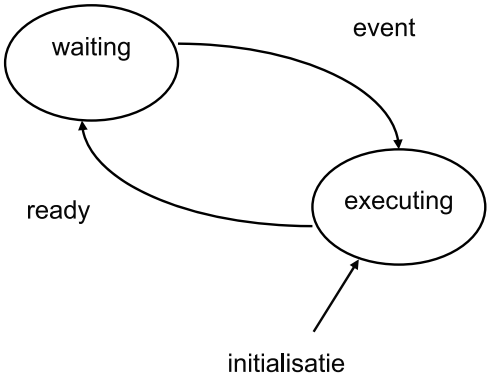
```
SHFT: process (clk) is
begin
    if rising_edge(clk) then
        A        <= DATA_IN;
        B        <= A;
        C        <= B;
        DATA_OUT <= C;
    end if;
end process
```

**Listing 7.40:** *VHDL-beschrijving van een 4-bits schuifregister.*

### 7.3.9 Wait

Een proces kent twee toestanden tijdens simulatie: waiting en executing. Dit is te zien in figuur 7.7. Een proces staat eerst in de waiting toestand. Als er een verandering op een ingangssignaal binnenkomt, wordt het proces gestart. Het komt in de executing toestand. Nadat alle statements binnen het proces zijn uitgevoerd en de waarden van de uitgangen bekend zijn, gaat het weer in de waiting toestand. Deze veranderingen op de uitgangen kunnen tot gevolg hebben dat weer nieuwe veranderingen optreden. Noot: bij initialisatie van de simulatie-omgeving wordt elk proces automatisch één keer gestart.

Een proces kan in de waiting-toestand gebracht worden door een **`wait`**-statement. Er zijn vier vormen:




**Figuur 7.7:** *De toestanden van een proces.*

| | |
|---|---|
| `wait until` | wacht tot een signaal een bepaalde waarde heeft; |
| `wait on` | wacht op een verandering van één of meerdere signalen; |
| `wait for` | wacht een bepaalde (simulatie-)tijd; |
| `wait` | wacht voor altijd. |

Als minstens één **`wait`**-statement wordt gebruikt, mag er geen sensitivity list worden opgegeven. Met **`wait until`** wordt een proces in de waiting-toestand gebracht tot een signaal een bepaalde waarde heeft gekregen. Dan moet er dus ook een verandering zijn geweest. In listing 7.41 is een alternatieve beschrijving van een D-flipflop te zien.

```
1 dff_alt: process is  -- no sensitivity list
2 begin
3     wait until clk = '1';
4     q <= d;
5 end process;
```

**Listing 7.41:** *Alternatieve VHDL-beschrijving van een D-flipflop.*

Merk op dat als er meerdere toekenningen in bovenstaande listing staan dit zal leiden tot een flipflop voor elke toekenning. Met bovenstaande listing is het dus niet mogelijk om combinatorische logica te beschrijven (bijvoorbeeld een asynchrone reset).

De **`wait on`** wordt gebruikt om op veranderingen van signalen te wachten. In listings 7.42 en 7.43 zijn twee processen te zien. Beide processen worden identiek verwerkt, dus een

```
1 begin
2     proc1: process (A,B) is
3     begin
4         ...
5
6     end process;
7 end;
```

**Listing 7.42:** *Process.*

```
1 begin
2     proc2: process is
3     begin
4         ...
5         wait on A,B;
6     end process;
7 end;
```

**Listing 7.43:** *Process.*

proces met een sensitivity list is identiek aan een proces zonder sensitivity list maar met een **wait on** statement als laatste statement.

Met **wait for** kan een proces voor een bepaalde (simulatie-)tijd in de waiting-toestand gezet worden. Dit wordt vooral gebruikt bij high-level beschrijvingen en testbenches. Zie listing 7.44.

```
1  testbench: process is      -- no sensitivity list
2  begin
3      data <= '0';           -- data is 0
4      reset <= '1';          -- activate reset
5      wait for 100 ns;       -- wait a little bit
6      data <= '0';           -- data is still 0
7      reset <= '0';          -- deactivate reset
8      wait for 200 ns;       -- wait a bit more
9      data <= '1'';          -- data is 1, reset is still 0
10     ...
11     wait;
12 end process;
```

**Listing 7.44:** *Voorbeeld van wait for.*

Merk op dat de toekenningen in regels 3 en 4 tegelijk plaatsvinden. Er zit namelijk 0 ns (simulatie-)tijd tussen de twee toekenningen.

Met alleen het keyword **wait** wordt een proces voor eeuwig in de waiting-toestand geplaatst. Dit wordt vaak gebruikt aan het eind van een *testbench*. De verschillende **wait**-constucties kunnen door elkaar heen gebruikt worden. De constructies **wait until** en **wait on** kunnen leiden tot synthese, **wait for** en **wait** kunnen niet gesynthetiseerd worden. Uitvoeren van een **wait** leidt tot updaten van signalen.

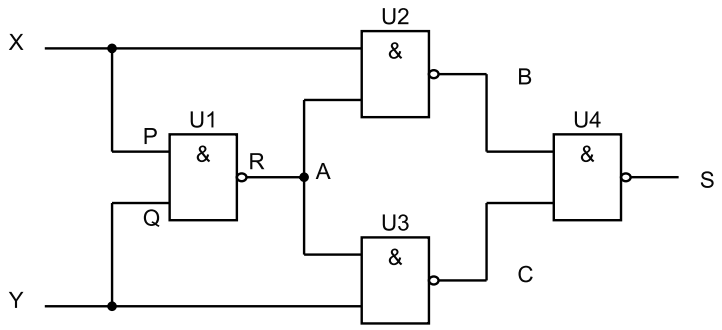
## 7.4 Structural VHDL

Grote projecten worden opgedeeld in meerdere design units. Deze design units vormen samen het hele systeem. Als zo'n design unit nog te groot is om in één keer ontwikkeld te worden, moet het weer onderverdeeld worden in een aantal design units. Er worden dus meerdere lagen in het ontwerp gemaakt: er wordt hiërarchisch beschreven. Deze manier van beschrijven wordt *structural VHDL* genoemd.

### 7.4.1 Declaratie en instantiëring van componenten

De werkwijze in VHDL is als volgt. Eerst wordt een design unit beschreven, zowel de entity als de architecture. Deze design unit wordt gesimuleerd en geverifieerd op functionaliteit en timing. Daarna wordt de design unit als *component* gebruikt in een hogere laag in de hiërarchie. In deze hogere laag wordt de component gedeclareerd en *geïnstantieerd*.

Als voorbeeld nemen we een EXOR die opgebouwd is uit vier NAND-poorten. Zie figuur 7.8. Het geheel heeft de ingangen $X$ en $Y$ en uitgang $S$. De interne signalen zijn $A$, $B$ en $C$. De NAND-poorten hebben de ingangen $P$ en $Q$ en uitgang $R$.




**Figuur 7.8:** *Een EXOR-poort met vier NAND-poorten.*

In listing 7.45 is de code van de NAND-poort te zien. De code is recht-toe recht-aan.

```
entity Nand_2 is
    port (P: in std_logic;
          Q: in std_logic;
          R: out std_logic);
end Nand_2;

architecture logical of Nand_2 is
begin
   R <= not (P and Q); -- the nand
end logical;
```

**Listing 7.45:** *VHDL-beschrijving van een 2-input NAND.*

In listing 7.46 is de code van de EXOR-poort te zien. Het begint met de declaratie van de ingangen en uitgangen. In de architecture volgt de declaratie van de component `Nand_2`. In feite is de declaratie niets anders dan de entity-beschrijving van de NAND. Er worden drie interne signalen gedeclareerd, te weten `A`, `B` en `C`. Dan volgt de daadwerkelijke beschrijving van de structuur. Er worden vier `Nand2`-componenten geïnstantieerd middels een *port map*. Elke instantiëring wordt vooraf gegaan met een verplichte label. Anders zijn de vier NAND-poorten niet van elkaar te onderscheiden. De port map geeft aan hoe een NAND met zijn buitenwereld verbonden is door middel van de *port association list*. In het schema in figuur 7.8 is te zien dat van component `U1` ingang `P` verbonden is met ingang `X` van de EXOR en dat ingang `Q` verbonden is met ingang `Y` van de EXOR. Uitgang `R` is verbonden met intern signaal `A`.

De overige NAND-poorten worden op vergelijkbare wijze gekoppeld. De hele architecture bestaat alleen maar uit geïnstantieerde componenten. Er is geen “echte” hardware beschreven. Goed om te weten is dat met een simulator de interne signalen te volgen zijn. Zo kunnen fouten worden opgespoord. Het gebruik van een simulator is onontbeerlijk tijdens het ontwerptraject.

### 7.4.2 Generics

Het is mogelijk om een VHDL-component *parametriseerbaar* maken. Als voorbeeld kunnen we denken aan de bitbreedte van een opteller of vertragingstijd van een poort. Dit kan elegant worden opgelost door middel van een *generic constant*. Bij gebruik van de

```
entity xor4nand is          -- entity of our XOR
    port (X: in std_logic;
          Y: in std_logic;
          S: out std_logic);
end xor4nand;

architecture structural of xor4nand is
component Nand_2 is       -- component declaration
    port (P: in std_logic;
          Q: in std_logic;
          R: out std_logic);
end component;

signal A,B,C : std_logic;
begin
    U1: Nand_2        -- component instantiation
    port map (P => X, Q => Y, R => A);
    U2: Nand_2         -- component instantiation
    port map (P => X, Q => A, R => B);
    U3: Nand_2         -- component instantiation
    port map (P => A, Q => Y, R => C);
    U4: Nand_2         -- component instantiation
    port map (P => B, Q => C, R => S);
end structural;
```

**Listing 7.46:** *Structural VHDL-beschrijving van een EXOR-poort met vier NAND-poorten.*

component in een hogere hiërachie (structural VHDL) kan dan de werkelijke waarde worden opgegeven. Bij gebrek aan een parameter wordt de standaard waarde gebruikt.

Als voorbeeld is de beschrijving gegeven van een multi-input NOR-poort, zie listing 7.47. In de entity-beschrijving is een *generic list* opgenomen met daarin twee parameters. De constante `tpd` is van het type time. De default-waarde is 2 ns. De constante `n` is van het type natural (geheel getal groter dan of gelijk aan 0). De default-waarde is 8. In de port list is nu een vector `x` gedefinieerd die loopt van $n-1$ naar 0. Het aantal elementen (of lengte) is dus $n$.

De multi-input NOR-poort wordt geïnstantieerd in een hoger niveau. Dit is te zien in listing 7.48. Het begint met de declaratie van de ingangen en uitgangen van het hogere niveau. Deze heeft een bitbreedte van 12 bits. In de archtecture wordt de componentbeschrijving gegeven van de multi-input NOR-poort. In feite is dat de entity-beschrijving. In de daadwerkelijke instantiëring wordt de vertragingstijd van de NOR-poort van 8 ns opgegeven, samen met de bitbreetde van 12 bits. Merk op dat de instantiëring over drie regels geschreven is maar dit als één statement gelezen moet worden.

Door middel van de generic constants is het eenvoudig om allerlei parameters te berekenen. In listing 7.49 is een deel van de beschrijving van een RS232-zender (ook bekend als de COM-poort op een PC) te zien. De systeemfrequentie van de schakeling is 50 MHz en de transmissiesnelheid is 9600 bits per seconde. De schakeling moet dan een aantal klokpulsen “wegtikken” om de te verzenden bits op de juiste momenten “op de lijn te zetten”. De constante `bittime` wordt automatisch berekend. Het signaal `timer` kan dan

```vhdl
entity multinor is
    generic (tpd : time := 2 ns;
             n : natural := 8);
    port (x : in std_logic_vector (n-1 downto 0);
          q : out std_logic);
end entity multinor;

architecture cascode of multinor is
begin
    process (x) is
    variable p : std_logic;
    begin
        p := '0';                    -- Initialize to '0'
        for i in n-1 downto 0 loop   -- Iterate the loop
            p := p or x(i);          -- OR-ing the lot
        end loop;
        q <= not p after tpd;        -- Make it a NOR
    end process;
end architecture cascode;
```

**Listing 7.47:** *VHDL-beschrijving van een multi-input NOR.*

```vhdl
entity big_multinor is      -- the higher level
    port (x : in std_logic_vector(11 downto 0);
          q : out std_logic);
end big_multinor;

architecture structural of big_multinor is

component multinor          -- declaration of component multinor
    generic (tpd : time := 2 ns;
             n : natural := 8);
    port (x : in std_logic_vector(n-1 downto 0);
          q : out std_logic);
end component;

begin
    instnor : multinor   -- instantiation of multinor
    generic map (tpd => 8 ns, n => 12)
    port map (q => q, x => x);
end structural;
```

**Listing 7.48:** *VHDL-beschrijving van een multi-input NOR.*

een bereik worden gegeven met deze constante. Zo is het eenvoudig om de systeemfrequentie of de transmissiesnelheid aan te passen zonder dat alle parameters met de hand moeten worden uitgerekend.

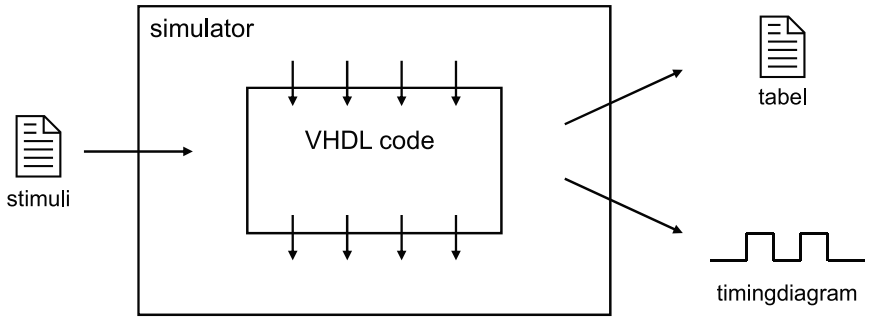
## 7.5 Simulatie

VHDL is oorspronkelijk bedoeld voor simulatie van digitale systemen. Bij simulatie wordt de VHDL-code stap voor stap doorlopen (geïnterpreteerd), alsof het een “gewone” pro-

```
entity rs232_module is
    generic (sysfreq : integer := 50000000;
             baudrate : integer := 9600);
    port (clk : in std_logic;
          ...       -- omit rest for now
         );
end entity rs232_module;

architecture rtl of rs232_module is
constant bittime : integer := sysfreq / baudrate;
signal timer : integer range 0 to bittime-1;
...
```

**Listing 7.49:** *Deel van een VHDL-beschrijving van een RS232-interface.*

grammeertaal is. Op de ingangssignalen worden (via de port list) stimuli (prikkel, aansporing) aangeboden. De stimuli worden aangeboden door middel van een *testbench*. Een testbench is VHDL-code die signalen genereert, maar het kan ook direct door de simulator gegeven worden. In de simulator kunnen interactieve commando's gegeven worden maar het is ook mogelijk om de commando's onder te brengen in een *script*. De simulator berekent de uitgangssignalen bij gegeven ingangssignalen en geeft dit weer in een tabel of grafiek (tijddiagram). Een schematische weergave is te zien in figuur 7.9.




**Figuur 7.9:** *Schematische weergave van de simulatie-omgeving.*

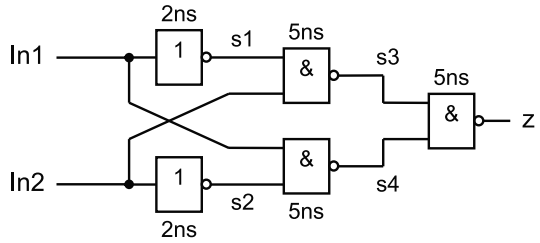
Simulatoren zijn *event-driven*. Code wordt alleen maar uitgevoerd als daar noodzaak voor is. Deze noodzaak wordt veroorzaakt als de waarde van een signaal verandert (*event*). In listing 7.50 is de beschrijving van een half adder te zien. Als tijdens simulatie signaal `a` en/of `b` verandert (event), worden beide statements doorgerekend. Te zien is dat de statements zelf ook weer events genereren en wel in de toekomst door middel van een **`after`**-clausule.

```
-- Only the achitecture
architecture logic of half_adder is
    begin
        sum   <= a xor b after 5 ns;
        carry <= a and b after 5 ns;
end architecture logic;
```

**Listing 7.50:** *VHDL-beschrijving van een half adder.*

Een event is een nieuwe waarde van een signaal samen met een tijdstip waarop het signaal verandert. De simulator moet dus een *geordende lijst van events* bijhouden van wanneer nieuwe toekenningen moeten worden uitgevoerd. Concurrency wordt gesimuleerd door de eerstvolgende gelijktijdige events uit te voeren.

We zullen het bovenstaande nader toelichten met een voorbeeld. In figuur 7.10 is de simulatie te zien van een EXOR-poort. De poort heeft twee ingangen `In1` en `In2` en een uitgang `z`. Verder zijn er vier interne signalen te zien. Naast het schema is de beginsituatie geschetst.




beginsituatie:

in1 = '0', in2 = '1'

s1 = '1', s2 = '0',

s3 = '0', s4 = '1', z = '1'

| tijdstip 0 ns | tijdstip 2 ns | tijdstip 5 ns | tijdstip 7 ns | tijdstip 10 ns | tijdstip 12 ns |
|---|---|---|---|---|---|
| In1 → 1 | s1 → 0 | s3 → 1 | s3 → 1 | z → 0 | z → 1 |
| In2 → 0 | s2 → 1 | s4 → 1 | s4 → 0 | z (1, 12ns) | ↓ |
| ↓ | s3 (1, 5ns) | s3 (1, 7ns) | z (0, 10 ns) | ↓ | stop |
| s1 (0, 2ns) | s4 (1, 5ns) | s4 (0, 7ns) | ↓ | - | |
| s2 (1, 2ns) | ↓ | ↓ | z (1, 12ns) | | |
| s3 (1, 5ns) | s3 (1, 7ns) | z (0, 10ns) | | | |
| s4 (1, 5ns) | s4 (0, 7ns) | | | | |

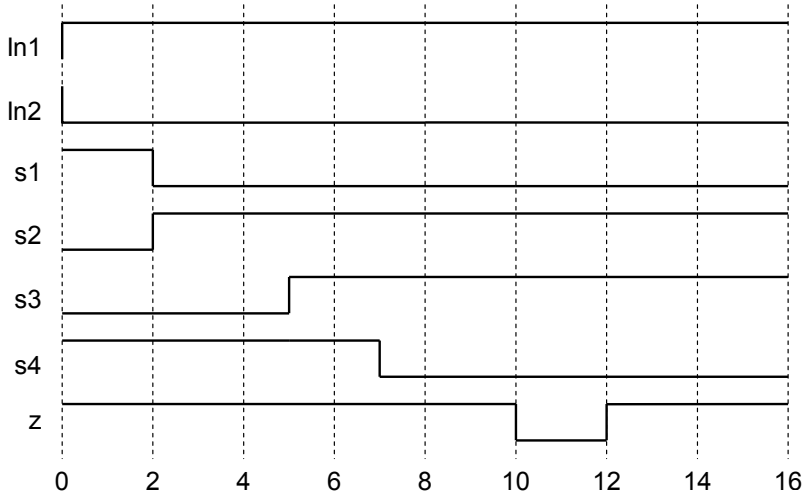
**Figuur 7.10:** *Voorbeeld van het doorrekenen van een simulatie.*

Op tijdstip 0 ns worden de twee toekenningen uitgevoerd, `in1`→ 1 en `in2`→ 0. Dat heeft tot gevolg dat er nieuwe waarden worden toegekend aan `s1`, `s2`, `s3` en `s4`. Dat gebeurt niet gelijk maar in de toekomst. We spreken dan ook wel van het inplannen (Engels: schedule) van toekenningen. De signalen `s1` en `s2` worden na 2 ns '0' respectievelijk '1'. De signalen `s3` en `s4` worden na 5 ns '1'. Alle events van tijdstip 0 ns zijn uitgevoerd, dus de simulator maakt nu een stap naar het eerstvolgende tijdstip. Dat is 2 ns.

Op tijdstip 2 ns worden de toekenningen `s1`→ 0 en `s2`→ 1 uitgevoerd. Er staan nog toekenningen gepland op tijdstip 5 ns. Die blijven rustig staan. Het feit dan `s1` en `s2` veranderen zorgt voor nieuwe toekenningen aan de signalen `s3` en `s4`. De vertraging van de NAND-poort is 5 ns dus worden er twee toekenningen ingepland op tijdstip 7 ns: `s3` wordt '1' en `s4` wordt '0'. Alle toekenningen voor tijdstip 2 ns zijn verwerkt en de simulator maakt een stap naar tijdstip 5 ns.

Op tijdstip 5 ns worden de toekenningen aan `s3` en `s4` uitgevoerd. Deze toekenningen zorgen voor een nieuwe toekenning aan `z` op tijdstip 10 ns. De toekenningen op tijdstip 7 ns blijven staan tot een later tijdstip. Daarna maakt de simulator een stap naar tijdstip 7 ns. Op dat tijdstip worden de toekenningen aan `s3` en `s4` uitgevoerd. Er staat nog een toekenning aan `z`, die blijft staan. Er volgt een nieuwe toekenning aan `z` op tijdstip 12 ns. De simulator gaat nu verder met tijdstip 10 ns. Signaal `z` krijgt een nieuwe

waarde. Dit genereert geen nieuwe toekenningen. Er staat nog één toekenning in de lijst. De simulator gaat nu naar tijdstip 12 ns en voert de laatste toekenning uit. Daar na stopt de simulator. In figuur 7.11 is het volledige timingdiagram te zien.




**Figuur 7.11:** *Timingdiagram van het doorrekenen van een simulatie.*

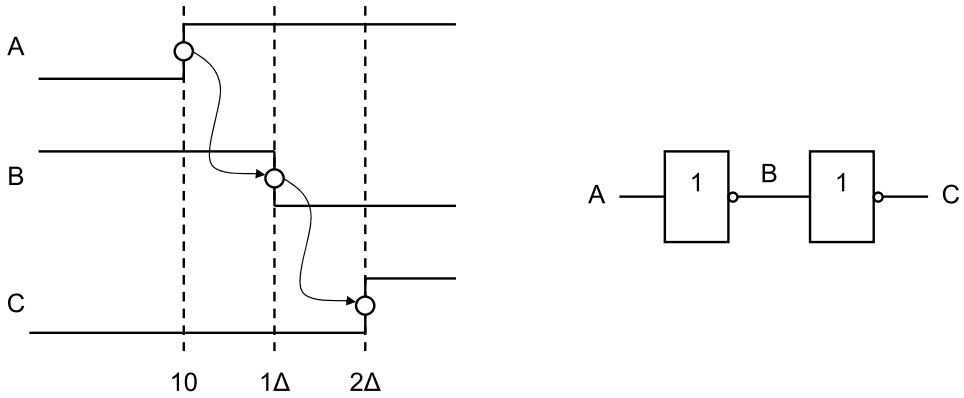
### 7.5.1 Delta delay

Het is mogelijk om een toekenning zonder vertraging te beschrijven, dus een event te genereren op dezelfde (simulatie-)tijd als waar de simulator events aan het evalueren is. Het probleem is dat bij het simuleren alleen events is de toekomst kunnen worden gegenereerd. De simulator lost dat op door een *delta delay* ($\Delta$) in te voegen. Delta delay wordt gebruikt om herhaald doorrekenen van events zonder vertraging te ordenen. Het wordt ook wel gezien als een oneindig kleine tijdstap. Er bestaat geen wereldlijke equivalent van delta delay, alleen bij simulatie.

Als voorbeeld nemen we een buffer die is opgebouwd uit twee inverters, zie listing 7.51. Als `A` van waarde verandert, wordt alleen de tweede toekenning uitgevoerd. Het is direct te zien dat als `A` verandert, `B` ook verandert en dat de eerste toekenning dus ook moet worden uitgevoerd. In figuur 7.12 zijn de toekenningen te zien met delta delays en het schema van de schakeling. Op tijdstip 10 ns wordt signaal `A` '1'. Als gevolg daarvan worden signaal `B` één delta delay later '0'. Dat heeft weer tot gevolg dat signaal `C` op tijdstip 10+2$\Delta$ ns '1' wordt.

```
1 architecture double_inv of buf is
2 begin
3     C <= not B;     -- no delay
4     B <= not A;     -- no delay
5 end double_inv;
```

**Listing 7.51:** *Codefragment van een buffer met twee inverters.*




**Figuur 7.12:** *Voorbeeld van het doorrekenen van een simulatie.*

### 7.5.2 Initialisatie van signalen en variabelen

Aan het begin van een simulatie krijgen alle signalen en variabelen een beginwaarde toegekend. Dit is het meest linkse element (bij enumeratietypen en karaktertypen) en het eerste getal in een bereik (range). Bij een `bit` is dit dus '0' en bij een `std_logic` is dit 'U'. Bij een integer zonder bereik is dit −2147483647. Het is mogelijk om de beginwaarde zelf in te stellen door bij de declaratie van een signaal of variabele een waarde toe te kennen. Zie listing 7.52.

```
signal a : std_logic := '0';
signal b : bit := '1';
signal c : std_logic_vector(7 downto 0) := "--------";
type d_type is (idle, start, stop, bingo);
signal d : d_type := bingo;
```

**Listing 7.52:** *Enkele initialisaties van signalen.*

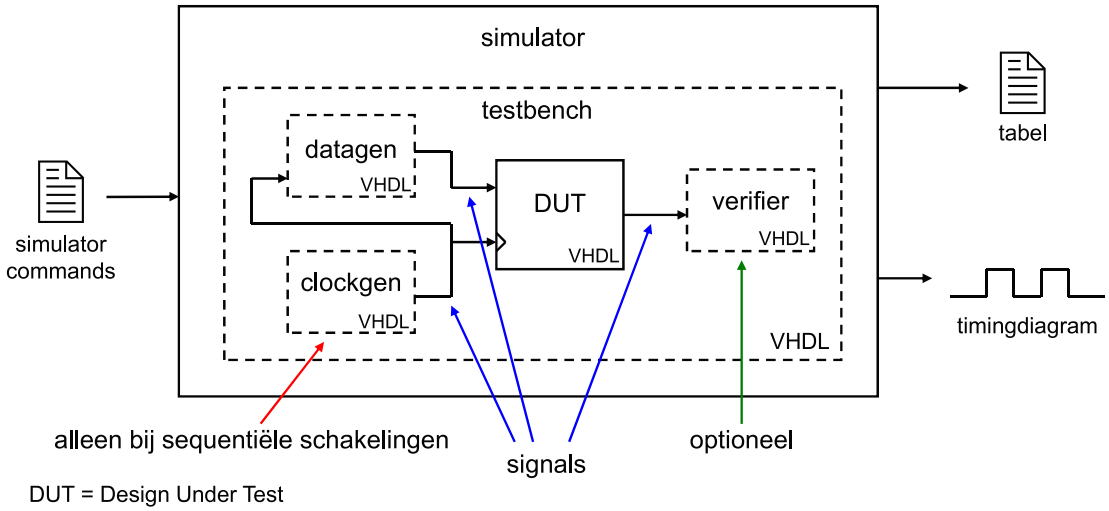
Let erop dat een initialisatie alleen betekenis heeft bij simulatie. Bij synthese worden ze genegeerd. Beginwaarden van flipflops en latches moeten altijd onder besturing van een (asynchrone) reset gebeuren.

### 7.5.3 Testbench

VHDL is oorspronkelijk bedoeld als simulatietaal. Dat is te merken aan de vele taalconstructies die niet voor synthese bedoeld zijn (**`wait`**, **`after`**, **`transport`**, pointers, files). Het simuleren van een VHDL-beschrijving gebeurt met behulp van een simulator. Tijdens het simuleren is het mogelijk een VHDL-beschrijving automatisch te *testen*.

De simulatie-omgeving biedt alle faciliteiten om te simuleren en te testen zoals het aanbieden van testsignalen (stimuli), afbeelden van de resultaten van de simulatie en het automatisch testen van de resultaten ten opzichte van een referentiemodel. Het simuleren en testen gebeurt met een *testbench*: een VHDL-beschrijving die signalen genereert om een (andere) VHDL-beschrijving te simuleren. De simulator wordt aangestuurd door commando's.

In figuur 7.13 is de simulatie- en testomgeving schematisch weergegeven. Het bestaat uit de te testen beschrijving (DUT = design under test) en een aantal processen. Het geheel is ingebed in een hogere hiërachielaag (structural VHDL).




**Figuur 7.13:** *Schematische weergave van de simulatie-omgeving.*

Het proces `clockgen` genereert het kloksignaal dat zowel voor de DUT als proces `datagen` gebruikt wordt. Het proces `datagen` genereert data-stimuli voor de DUT, inclusief het resetsignaal. De `verifier` verifieert de uitkomst van de DUT met een bekend en werkend model. Deze beschrijving is optioneel. Alle onderdelen binnen de testbench zijn geschreven in VHDL. De processen en DUT wisselen gegevens uit via signalen.

### 7.5.4 Kloksignaal

Bij een sequentiële VHDL-beschrijving is een kloksignaal nodig. Meestal is dit een symmetrisch signaal, maar dat is niet verplicht. Een kloksignaal kan beschreven worden door toekenningen en **`wait`**-statements in een eeuwig durende lus, zie listing 7.53. De klok is eerst 50 ns laag en daarna 50 ns hoog. In totaal is de periodetijd 100 ns en de frequentie is 10 MHz. In de beschrijving ontbreekt het losse **`wait`**-statement. Dit zorgt ervoor dat het proces zich steeds blijft herhalen.

```
1 clockgen: process is     -- no sensitivity list
2 begin
3     clk <= '0';          -- clock 50 ns low
4     wait for 50 ns;
5     clk <= '1';          -- clock 50 ns high
6     wait for 50 ns;
7 end process clockgen;
```

**Listing 7.53:** *VHDL-beschrijving van een kloksignaal.*

### 7.5.5 Datasignalen

Met de datasignalen worden waarden aan de ingangen anders dan het kloksignaal aan de DUT toegekend. Dat kan door middel van toekenningen en **`wait`**-statements.

Het statement **wait for** kan gebruikt worden om een proces te laten wachten (suspend, waiting). De simulatietijd loopt wel door. In listing 7.54 zijn enkele voorbeelden te zien. Er wordt een klokperiode van 50 ns gedefinieerd en een korte vertraging van een tiende van de periodetijd. Daarna volgen drie vormen van **wait for**.

```
constant Tperiod : time := 50 ns;
constant small_delay : time := Tperiod/10;
...
wait for 10 ns;
wait for Tperiod;
wait for small_delay;
```

**Listing 7.54:** *Gebruik van constanten in een testbench.*

Het statement **wait until** kan gebruikt worden om te *synchroniseren* op het signaal, bijvoorbeeld een kloksignaal:

```
wait until clk = '1';  -- wait for positive edge
wait until clk = '0';  -- wait for negative edge
```

**Listing 7.55:** *Gebruik van wait for om te synchroniseren op een klokflank.*

In listing 7.56 is het gebruik van **wait until** en **wait for** te zien. De kleine vertraging zorgt ervoor dat de nieuwe stimuli iets later aangeboden worden zodat ze goed zichtbaar zijn in het timingdiagram van de simulator

```
x <= '0';                -- set data
wait until clk = '1';    -- synchronize on positive edge
wait for 10 ns;          -- slightly after edge

x <= '1';                -- set data
wait until clk = '1';    -- synchronize on positive edge
wait for 10 ns;          -- slightly after edge
```

**Listing 7.56:** *Gebruik van wait for om te synchroniseren op een klokflank.*

### 7.5.6 Reset

Het resetsignaal wordt als eerste beschreven zodat de schakeling naar een gedefinieerde toestand gaat. Dit is te zien in listing 7.57. Let erop dat voordat de reset geactiveerd wordt alle ingangen een gedefinieerde waarde toegekend krijgen anders volgen er ongewenste bijkomstigheden bij simulatie.

### 7.5.7 Simulatie van een T-flipflop

In deze paragraaf bespreken we een testbench voor een T-flipflop. De testbench is als volgt opgebouwd:

- lege entity-beschrijving (geen port list);

```
datagen: process is            -- no sensitivity list
begin
    x <= '0';                  -- data 0
    ...                        -- initialize other signals

    areset <= '1';             -- activate reset
    wait for 100 ns;           -- reset 100 ns active

    areset <= '0';             -- deactivate reset
    wait for 50 ns;            -- wait a bit ...
    ...
```

**Listing 7.57:** *Gebruik van het resetsignaal.*

- declaratie van signalen die gestimuleerd en gemonitord moeten worden door de simulator;
- component-declaratie (de DUT);
- component-instantiëring;
- de feitelijke testbench (code met stimulus-opdrachten).

Om het simulatieproces goed te laten verlopen, worden commando's aangeboden aan de simulator door middel van een script. In het script staan commando's voor:

- het compileren van de DUT en testbench;
- het starten van de simulatie van de testbench en de DUT;
- het afbeelden van signalen op een grafische interface.

Een T-flipflop is een flipflop die van stand verandert als de stuuringang (T) logisch '1' is. Als de stuuringang logisch '0' is, blijft de T-flipflop de huidige stand vasthouden.

In listing 7.58 is de beschrijving van de T-flipflop gegeven. Eerst wordt de `std_logic`-bibliotheek geladen. Daarna volgen de declaratie van de ingangen en uitgangen.

Het gedrag van de T-flipflop wordt beschreven in de architecture. Er wordt gebruik gemaakt van een intern signaal `q_int` want in VHDL is het niet toegestaan om te lezen van een uitgang (vanaf VHDL-2008 mag dat wel). De T-flipflop beschikt over een asynchrone, actieve hoge reset. Die zorgt ervoor dat de stand van de flipflop '0' wordt. Onder de klokflankbeschrijving wordt nagegaan of signaal `T` een '1' is. Als dat zo is, wordt de stand van de flipflop geïnverteerd. Als `T` '0' is, gebeurt er dus niets en blijft de flipflop zijn stand onthouden. Het steeds maar weer inverteren wordt *toggelen*[3] genoemd. Vandaar ook de naam T-flipflop.

De testbench ziet eruit als gewone VHDL-code. Er wordt echter een lege entity opgegeven omdat de testbench niet ingebed wordt in een hogere hiërarchie en ook geen verbindingen heeft met de buitenwereld. In listing 7.59 is de entity-declaratie en een deel van de architecture te zien. De naam van de entity is `tb_flipflop`. In de architecture declareren we een aantal constanten die handig zijn tijdens simulatie. Verder zijn de vier signalen

[3] Er is niet echt een Nederlands woord voor. Schakelen komt nog het meest dichtbij.

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

entity tflipflop is
    port (clk    : in std_logic;
          areset : in std_logic;
          t      : in std_logic;
          q      : out std_logic
         );
end entity tflipflop;

architecture behavioral of tflipflop is
signal q_int : std_logic;              -- internal signal
begin

    process (clk, areset) is
    begin
        if areset = '1' then           -- reset is active high
            q_int <= '0';
        elsif rising_edge(clk) then    -- on posedge of clock
            if t = '1' then            -- if T = 1 ...
                q_int <= not q_int;    -- ... then toggle ...
            end if;
        end if;
    end process;

    q <= q_int;                        -- copy internal to external

end architecture behavioral;
```

**Listing 7.58:** *VHDL-beschrijving van een T-flipflop (entity en architecture).*

te zien die worden bekeken tijdens simulatie. Het laatste stuk betreft de declaratie van de component `tflipflop`. In feite is dat de entity-beschrijving van de component.

De code in listing 7.60 begint met de instantiëring van de T-flipflop. In de port association list zijn beschrijvingen te zien als `clk => clk`. Hiermee worden verbindingen gemaakt tussen het signaal `clk` van testbench en het signaal `clk` van de DUT. De eerste in de verbinding is het signaal van de DUT, de tweede is het signaal van de testbench. Daarna volgt de beschrijving van het kloksignaal. De frequentie van het kloksignaal is 50 MHz.

De beschrijving van de datagen is te zien in listing 7.61. De ingang `t` wordt op '0' gezet en de reset op '1'. Daarna wordt er twee klokperiodes gewacht. Vervolgens wordt de reset op '0' gezet en wordt er weer twee klokperiodes gewacht. Tevens wordt er gesynchroniseerd op de opgaande flank van de klok. Voordat aan `t` een '1' wordt toegekend wordt eerst nog kort gewacht. Zo is de signaalverandering op `t` goed te zien in een timingdiagram. Na de toekenning wordt drie klokpulsen gewacht. Vervolgens wordt na een korte wachttijd `t` weer op '0' gezet. Het laatste statement zorgt ervoor dat het proces stopt met uitvoeren.

In listing 7.62 op pagina 42 zijn de commando's voor de simulator te zien. Het commando **`transcript`** `on` zorgt ervoor dat alle meldingen op het scherm worden gegeven. De volgende drie coderegels verwijderen de eerder aangemaakte bestanden uit de *work*

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

entity tb_tflipflop is                          -- empty entity
end entity tb_tflipflop;

architecture sim of tb_tflipflop is
constant Tperiod : time := 20 ns;               -- 50 MHz
constant small_delay : time := Tperiod / 10;    -- 2 ns
signal clk    : std_logic;                      -- monitoring signals
signal areset : std_logic;
signal t      : std_logic;
signal q      : std_logic;

component tflipflop is
   port (clk    : in std_logic;
         areset : in std_logic;
         t      : in std_logic;
         q      : out std_logic);
end component tflipflop;
...
```

**Listing 7.59:** *VHDL-testbench van een T-flipflop (entity, signalen en component).*

```
...
begin

    -- Instantiate a T flipflop and map the port signals
    -- to the monitoring signals
    dut : tflipflop
    port map (clk => clk, areset => areset, t => t, q => q);

    clockgen: process is   -- no sensitivity list
    begin
        clk <= '0';         -- clock has 50% Duty Cyle
        wait for Tperiod/2;
        clk <= '1';
        wait for Tperiod/2;
    end process clockgen;
...
```

**Listing 7.60:** *VHDL-beschrijving van een T-flipflop (port map en clockgen).*

*library*. Vervolgens wordt de work library opnieuw aangemaakt.

Met het commando **vcom** worden de VHDL-bestanden van de DUT en de testbench in gecompileerde vorm opgeslagen in de work library. De optie `-93` zorgt ervoor dat de compiler VHDL-1993-code accepteert. Let op de naamgeving van de bestanden: de T-flipflop is opgeslagen in het bestand `tflipflop.vhd` en de testbench in het bestand `tb_tflipflop.vhd`. Merk op dat we de bestandsnamen genoemd hebben naar de entity-namen. Dat hoeft niet, maar maakt het een en ander wel makkelijk.

We vervolgen ons script met het starten van de simulatie met het commando **vsim**. Als

```
...
    datagen: process is            -- no sensitivity list
    begin
        t <= '0';
        areset <= '1';
        wait for 2*Tperiod;      -- wait for two clock cycles

        areset <= '0';
        wait for 2*Tperiod;
        wait until clk = '1';    -- synchronize on positive edge
        wait for small_delay;    -- and wait a bit

        t <= '1';                -- set value
        wait until clk = '1';    -- wait for three clocks
        wait until clk = '1';
        wait until clk = '1';
        wait for small_delay;

        t <= '0';
        wait;                    -- wait forever

    end process;

end architecture sim;
```

**Listing 7.61:** *VHDL-beschrijving van een T-flipflop (datagen).*

simulatietijdstap wordt 1 ns opgegeven. Dat betekent dat alle tijden in een resolutie van 1 ns worden verwerkt. Verder wordt als bibliotheek de eerder aangemaakt work library opgegeven. De simulatie wordt gestart met `tb_tflipflop` als *top level entity*.

Met het commando **`add log`** `-r *` worden alle signalen in de simulatie-omgeving gelogd, ook de signalen in de DUT. Dit is een prima keuze als het aantal signalen klein is. Bij grote systemen met veel signalen moet er een keuze worden gemaakt welke signalen gelogd worden en welke niet.

De vijf commando's **`add list`** zorgen ervoor dat de genoemde signalen in een lijst worden opgeslagen. Deze lijst is zichtbaar te maken op het scherm. We maken ook het interne signaal `q_int` van de T-flipflop zichtbaar. De commando's **`add wave`** doen hetzelfde maar dan worden signalen in een timingdiagram zichtbaar. De regels met de optie `-divider` zorgen ervoor dat de signalen in groepen worden afgebeeld.

Met de commando's **`view list`** en **`view wave`** worden de lijst respectievelijk het timing-diagram met signalen zichtbaar gemaakt. De laatste stap is het daadwerkelijk starten van de simulatie met het commando **`run`** `500 ns`.

Een deel van de uitvoer is in tabelvorm te zien in listing 7.63. Duidelijk is te zien dat de simulator in eerste instantie alle signalen op 'U' heeft geïnitialiseerd. Na enkele delta delays hebben de signalen de toegekende waarden gekregen. Het kloksignaal toggelt om de 10 ns en op tijdstip 92 ns (plus één delta) wordt '1' toegekend aan de T-ingang. Op tijdstip 110 + 3 delta ns toggelt de uitgang `q`. De namen van de signalen worden vooraf gegaan door `/tb_tflipflop/`. Dat houdt in dat de signalen bij de testbench horen.

```
# Turn on transcript
transcript on

# Set up RTL work library
if {[file exists rtl_work]} {
    vdel -lib rtl_work -all
}

# Create RTL work library and use it as work
vlib rtl_work
vmap work rtl_work

# Compile VHDL description of T flip-flop
vcom -93 -work work tfliflop.vhd

# Compile VHDL testbench of T flip-flop
vcom -93 -work work tb_tflipflop.vhd

# Set up simulation with the testbench as top level
#   -t 1ns  =  1 ns time step
#   -L ...  =  use library ...
vsim -t 1ns -L rtl_work -L work tb_tflipflop

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

# Add a number of signals of the simulated design
# to the tabular list (text oriented)
add list clk
add list areset
add list t
add list dut/q_int
add list q

# Add a number of signals of the simulated design
# to the waveform list (GUI oriented)
add wave -divider "System"
add wave clk
add wave areset
add wave -divider "Inputs"
add wave t
add wave -divider "Internals"
add wave dut/q_int
add wave -divider "Outputs"
add wave q

# Open List and Waveform window
view list
view wave

# Run simulation for 500 ns
run 500 ns
```

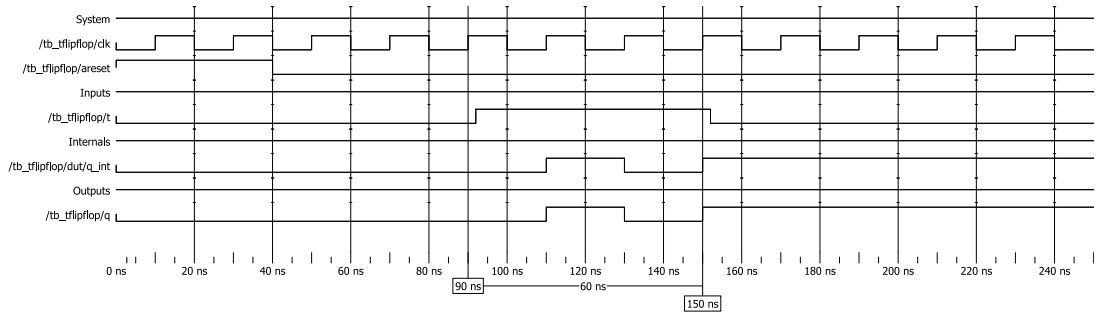
**Listing 7.62:** *Modelsim commando-script voor simulatie van de T-flipflop.*

Signaal `q_int` heeft ook nog `dut` in de naam want dit signaal hoort bij de DUT.

```
          ns       /tb_tflipflop/clk
           delta  /tb_tflipflop/areset
                         /tb_tflipflop/t
                   /tb_tflipflop/dut/q_int
                             /tb_tflipflop/q
           0   +0                  U U U U U
           0   +1                  0 1 0 U U
           0   +2                  0 1 0 0 U
           0   +3                  0 1 0 0 0
          10   +1                  1 1 0 0 0
          20   +1                  0 1 0 0 0
          30   +1                  1 1 0 0 0
          40   +1                  0 0 0 0 0
          50   +1                  1 0 0 0 0
          60   +1                  0 0 0 0 0
          70   +1                  1 0 0 0 0
          80   +1                  0 0 0 0 0
          90   +1                  1 0 0 0 0
          92   +1                  1 0 1 0 0
         100   +1                  0 0 1 0 0
         110   +1                  1 0 1 0 0
         110   +2                  1 0 1 1 0
         110   +3                  1 0 1 1 1
         120   +1                  0 0 1 1 1
```

**Listing 7.63:** *Deel van de uitvoer van de simulatie van de T-flipflop in lijstvorm.*

In figuur 7.14 is het timingdiagram te zien. Hier zien we de delta delays niet (dat kan wel worden ingesteld). We hebben twee *markers* geplaatst op de tijdstippen 90 ns en 150 ns. Goed is te zien dat signaal `T` op tijdstip 92 ns op '1' wordt gezet. Merk op dat de signalen `q_int` en `q` identiek zijn. Ook hier worden de signalen voorafgegaan door `/tb_tflipflop/`. Signaal `q_int` wordt voorafgegaan door `/tb_tflipflop/dut/`.




**Figuur 7.14:** *Timingdiagram van de simulatie van de T-flipflop.*

### Alternatieve datageneratie voor T-flipflop

VHDL is een rijke taal met veel taalconstructies. Het is mogelijk om vectoren te gebruiken en lussen te construeren. Deze mogelijkheden kunnen goed ingezet worden bij testbenches. In listing 7.64 is een voorbeeld te zien van een alternatieve datagen. Een *string*

(vector van characters) wordt geladen met een bitpatroon. Er wordt gebruik gemaakt van een constante *unconstraint vector*. In deze vector worden de te testen bitwaarden opgeslagen. De vector wordt dan toegekend aan een variabele. In de **for**-lus worden de waarden een voor een toegekend aan de ingang van de T-flipflop. Op deze manier is het testpatroon eenvoudig te wijzigen.

```
1  datagen: process is
2
3  -- the valuations for the T-flipflop as a string
4  constant t_i_s : std_logic_vector := "1111110101101001110";
5  -- range is from 0 to 18 (19 elements)
6  variable t_i : std_logic_vector(t_i_s'range) := t_i_s;
7  begin
8
9      -- reset omitted for clarity
10
11     for index in t_i'range loop    -- loop through all input bits
12         t <= t_i(index);           -- assign to flipflop input
13         wait until clk = '1';
14         wait for small_delay;
15     end loop;
16
17     wait;
18 end process;
```

**Listing 7.64:** *Alternatieve datageneratie.*

## 7.6 Synthese

Synthese is het proces van het automatisch genereren van hardware uit een beschrijving. Aan het syntheseproces kunnen zogenoemde *constraints* (beperkingen) worden opgegeven. Voorbeelden van constraints zijn: minimale werkfrequentie, maximaal logische niveaus, maximale chipoppervlakte (bij ASIC), maximaal aantal cellen (bij FPGA), maximale vertraging (bijvoorbeeld van klokflank naar uitgang) en maximale dissipatie.

De beschrijving moet natuurlijk echte hardware kunnen opleveren (denk weer aan simulatie). De *synthesizer* vertaalt de VHDL-beschrijving eerst naar bekende *primitieven* zoals poorten, multiplexers, optellers, vergelijkers, latches en flipflops. In wezen is de synthesizer vergelijkbaar met een compiler voor programmeertalen als 'C' alleen wordt nu hardware gegenereerd. Daarna zorgt de synthesizer ervoor dat deze primitieven worden ingepast op de beschikbare technologie. Voorbeelden van primitieven en technologie bij een FPGA zijn te vinden in tabel 7.2.

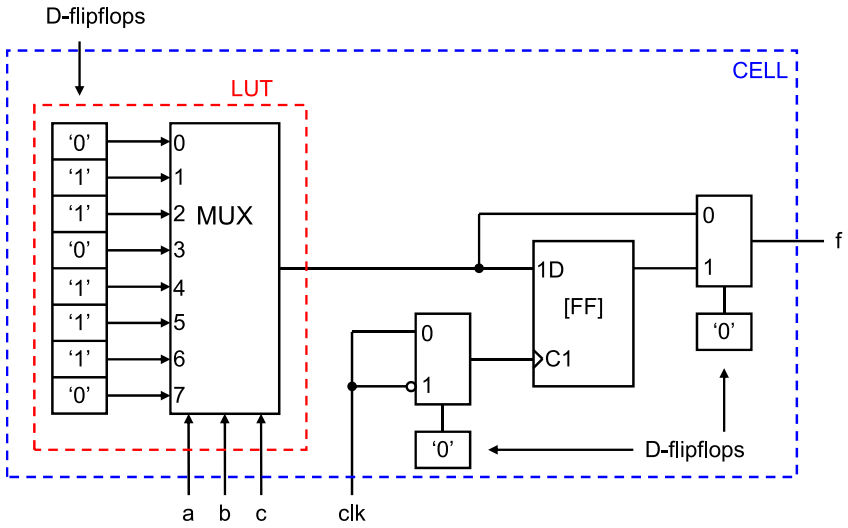
De synthesizer probeert aan alle constraints te voldoen. Als dat lukt is het syntheseproces geslaagd. Lukt het niet, dan moeten de beschrijving en/of de constraints worden aangepast. Bij het aanpassen van de beschrijving kunnen we denken aan pipelining, retiming en *resource sharing*.

Opmerking: de meeste synthesizers negeren de sensitivity list volledig. Als er signalen in de sensitivity list vergeten zijn dan komen simulatie en synthese niet overeen.

**Tabel 7.2:** *Voorbeelden van primitieven en technologie bij een FPGA.*

| Primitieven | | Technologie |
|---|---|---|
| case, with | multiplexers | Lookup Tables (LUT) |
| if, when else | priority encoders | D-flipflops |
| ... 'event and ... | flipflops, registers | on-chip multipliers |
| + − * / | adders, subtractors, ... | on-chip memory |

In een FPGA worden *cellen* (Engels: cells) gebruikt om functies te realiseren. Een vereenvoudigde weergave is gegeven in figuur 7.15. Een cel bestaat uit een *Lookup Table* (LUT) en een D-flipflop en kan een combinatorische of sequentiële functie realiseren. Hiervoor worden multiplexers en configuratieflipflops gebruikt. De 2x1-multiplexer in de kloklijn selecteert de opgaande of neergaande flank voor de klokingang van de flipflop. De andere 2x1-multiplexer selecteert de uitgang van de flipflop of van de LUT. In de figuur zijn de instellingen te zien voor een combinatorische functie.




**Figuur 7.15:** *Schematische weergaven van een cel in een FPGA.*

Een LUT is niets anders dan een multiplexer waarvan de dataingangen verbonden zijn met de constanten 0 en 1. De selectie-ingangen selecteren op enig moment een van de dataingangen. Voor de dataingangen zijn D-flipflops geplaatst. In de figuur is een LUT met drie stuuringangen te zien. De gerealiseerde functie is $f_{a,b,c} = \sum m(1,2,4,5,6)$. Gewoonlijk worden LUT's uitgevoerd met vier stuuringangen. Een kleine FPGA heeft enkele duizenden cellen. Een grote FPGA heeft enkele honderdduizenden cellen. Zie voor meer informatie over de FPGA bijlage A.

### 7.6.1 Combinatorische logica

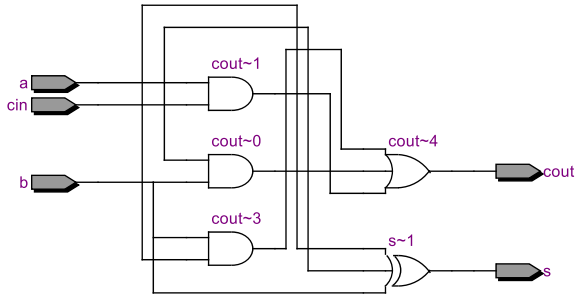
We zullen nu een aantal voorbeelden van synthese geven. We beginnen met een full adder die is opgebouwd uit losse poorten. In listing 7.65 is de code gegeven.

De synthesizer genereert het schema dat te zien in in figuur 7.16. De symbolen, hier poorten, worden primitieven (Engels: primitives genoemd. Te zien is dat er een aantal

```vhdl
architecture logic of full_adder is
begin
    s    <= a xor b xor cin;
    cout <= (a and b) or (a and cin) or (b and cin);
end architecture logic;
```

**Listing 7.65:** *VHDL-beschrijving van een full adder met poorten.*

interne namen is gedefinieerd.



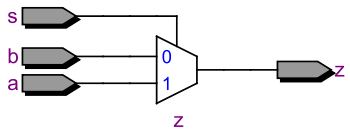

**Figuur 7.16:** *Synthese van een full adder.*

Het tweede voorbeeld betreft een 2x1 multiplexer. De code is te zien in listing 7.66. De beschrijving is met een `if`-statement uitgevoerd. Deze manier van beschrijven wordt ook wel behavioral VHDL genoemd.

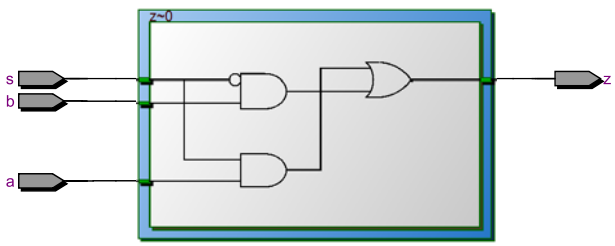
```vhdl
process (a,b,s) is
begin
    if s = '1' then
        z <= a;
    else
        z <= b;
    end if;
end process;
```

**Listing 7.66:** *VHDL-beschrijving van een 2x1 multiplexer.*

In figuur 7.17 is de synthese van de multiplexer te zien in primitieven (een multiplexer) en in een LUT. Bij de LUT wordt een schema met poorten gegeven. In werkelijkheid bestaat de LUT niet uit poorten maar uit een multiplexer met geheugencellen.




**(a)** *Synthese van een multiplexer met primitieven.*




**(b)** *Synthese van een multiplexer met een LUT.*

**Figuur 7.17:** *Synthese van een 2x1 multiplexer met primitieven en een LUT.*

In listing 7.67 en 7.68 zijn de beschrijvingen gegeven van een stuk combinatorische logica met een Conditional Signal Assignment (CSA) respectievelijk een **if**-statement.

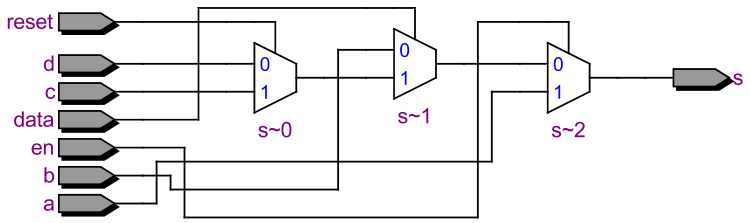
```
architecture behav of prior is
begin
    s <= a  when en = '1' else
         b  when data = '0' else
         c  when reset = '1' else
         d;
end architecture behav;
```

**Listing 7.67:** *VHDL-beschrijving van een CSA.*

```
process(en, data, reset) is
begin
    if en = '1' then
        s <= a;
    elsif data = '0' then
        s <= b:
    elsif reset = '1' then
        s <= c;
    else
        s <= d;
    end if;
end process;
```

**Listing 7.68:** *Beschrijving met if.*

De volgorde van de regels is belangrijk. Er is een prioriteitenvolgorde. Dat is goed te zien in de synthese van de code, zie figuur 7.18. In de code wordt eerst signaal `en` geëvalueerd. Als `en` '1' is, wordt signaal `a` aan `s` toegekend. In het schema is dat de multiplexer aan de rechterkant. Als `en` niet '1' is, wordt de tweede regel geëvalueerd. Daar wordt bekeken of signaal `data` '0' is. Dan volgt toekenning van `b` aan `s`. Dit is te zien aan de multiplexer in het midden. Als laatste wordt gekeken of `reset` gelijk is aan '1'. Zo ja, dan wordt `c` toegekend aan `s`, anders wordt `d` toegekend. Dit is te zien aan de multiplexer aan de linkerkant. Deze schakeling wordt ook wel een *priority encoder* genoemd.




**Figuur 7.18:** *Synthese van een priority encoder.*

Het **case**-statement wordt gesynthetiseerd met een multiplexer. De code is te zien in listing 7.69. De synthese is te zien in figuur 7.19.

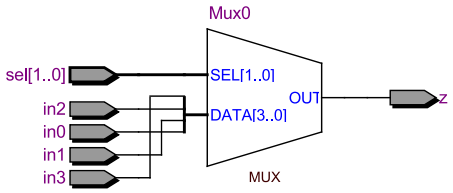
### 7.6.2 Geheugenelementen

In figuur 7.20 is de synthese te zien van de gated D-latch uit paragraaf 7.3.7. Links is de primitieve te zien en rechts de opbouw met een LUT. De latch wordt gebouwd door logica met een *asynchrone feedback*. Asynchroon wil zeggen dat er geen kloksignaal gebruikt wordt. Merk op dat de latch gebaseerd is op een multiplexer.

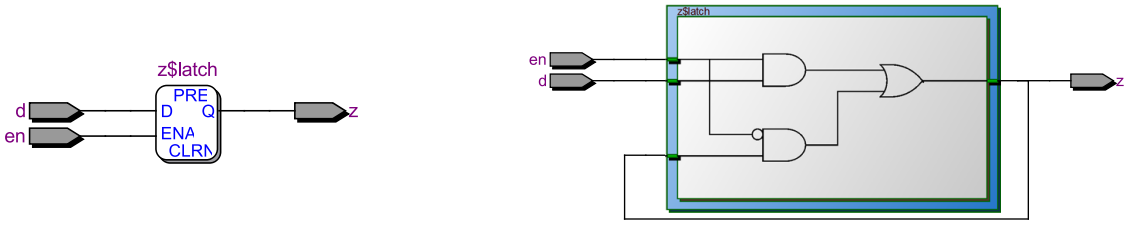
In listing 7.70 is de beschrijving te zien van een D-flipflop met enable en reset. De verwerking van de reset is buiten de klokflankbeschrijving geplaatst en dus asynchroon. De enable is binnen de klokflankbeschrijving geplaatst en dus synchroon. De toekenning `q <= d` vindt alleen plaats als er een opgaande flank is geweest én `en` is '1'.

```vhdl
process (sel,in0,in1,in2,in3) is
begin
    case sel is
        when "00" => z <= in0;
        when "01" => z <= in1;
        when "10" => z <= in2;
        when "11" => z <= in3;
        when others => z <= 'X';
    end case;
end process;
```

**Listing 7.69:** *Een* ***case****-statement.*




**Figuur 7.19:** *Synthese van* ***case****.*

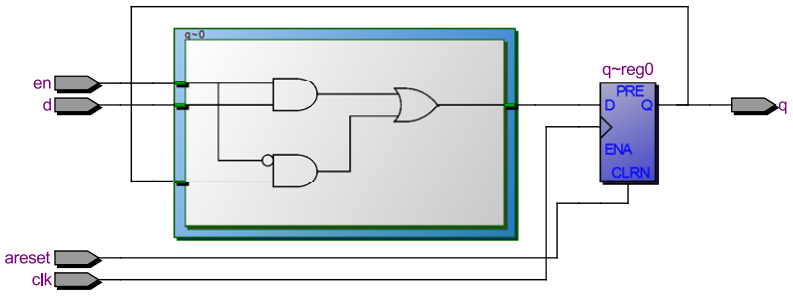



**Figuur 7.20:** *Gated D-latch met primitieve en LUT.*

```vhdl
process (clk, areset) is
begin
    if areset = '0' then
        q <= '0';
    elsif rising_edge(clk) then
        if en = '1' then
            q <= d;
        end if;
    end if;
end process;
```

**Listing 7.70:** *VHDL-beschrijving van een D-flipflop met enable en asynchrone reset.*

De synthese is te zien in figuur 7.21. Op een FPGA zijn al compleet functionerende flipflops geplaatst. Die worden dus niet met poorten gerealiseerd (vandaar het symbool van de flipflop in de figuur). De enable-faciliteit wordt gerealiseerd met een multiplexer.




**Figuur 7.21:** *Synthese van D-flipflop met enable en asynchrone reset.*

### 7.6.3 Variabelen en signalen

Let goed op met synthese van variabelen en signalen. Signalen onder een klokflank-beschrijving leveren altijd een flipflop op. Variabelen *kunnen* flipflops opleveren. In listings 7.71 en 7.72 is het gebruik van variabelen en signalen te zien.

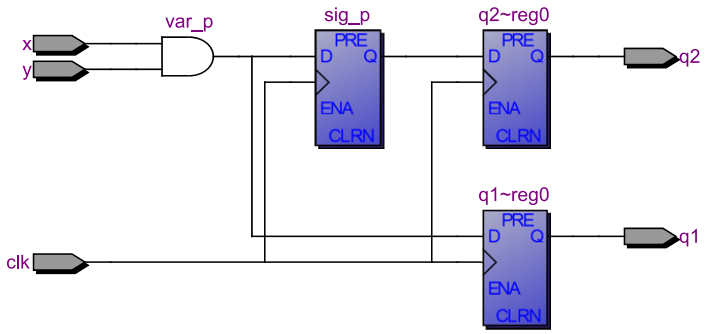
```
1 proc3: process (clk) is
2 variable var_p : std_logic;
3 begin
4     if rising_edge(clk) then
5         var_p := x and y;
6         q1 <= var_p;
7     end if;
8 end process;
```

**Listing 7.71:** *Gebruik van variabele en signaal.*

```
1 signal sig_p : std_logic;
2 proc4: process (clk) is
3 begin
4     if rising_edge(clk) then
5         sig_p <= x and y;
6         q2 <= sig_p;
7     end if;
8 end process;
```

**Listing 7.72:** *Gebruik van twee signalen.*

In figuur 7.22 is de schakeling te zien. Te zien is dat voor zowel `sig_p` als `var_p` slechts één AND-poort wordt gebruikt. Dit noemen we *resource sharing*. Voor `var_p` wordt geen flipflop gesynthetiseerd ondanks dat de toekenning onder een klokflankbeschrijving staat. Voor `sig_p` wordt wel een flipflop gesynthetiseerd.

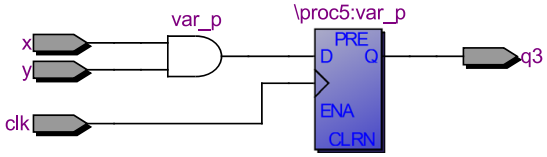



**Figuur 7.22:** *Synthese van variabelen en signalen.*

De code in listing 7.73 werkt identiek aan die van listing 7.71. De toekenning `q3 <= var_p` mag dus ook buiten de klokflankbeschrijving staan. Het resultaat is een flipflop voor `q3`. Zie figuur 7.23.

```
1 proc5: process (clk) is
2 variable var_p : std_logic;
3 begin
4     if rising_edge(clk) then
5         var_p := x and y;
6     end if;
7     q3 <= var_p;
8 end process;
```

**Listing 7.73:** *Gebruik van variabele en signaal.*




**Figuur 7.23:** *Synthese van variabele en signaal.*

### 7.6.4 Synthese van datatypes

De synthese van de meest gebruikte datatypes is gedefinieerd in de IEEE standaard 1076.3 [13]. Bij een `boolean` wordt `false` gelijk gesteld aan een logische 0 en `true` gelijk gesteld aan een logisch 1. Bij een `bit` wordt '0' gelijk gesteld aan een logische 0 en '1' gelijk gesteld aan een logische 1.

Bij het type `std_logic` worden '0' en 'L' geïnterpreteerd als een logische 0. De waarden '1' en 'H' worden geïnterpreteerd als een logische 1. De waarden 'H' en 'L' worden dus niet vertaald naar een pull-up en pull-down weerstand. De waarde 'Z' wordt vertaald naar een tri-state buffer (zie listing 7.26). De waarden 'U', 'X', 'W' en '—' kunnen niet vertaald worden naar een logische waarde. Een test voor gelijkheid met een **`if`**-statement met deze waarden levert altijd een niet waar op en een test op ongelijkheid levert altijd een waar op. Het kan zijn dat de synthesesoftware deze waarden niet accepteert en een foutmelding geeft. Denk hierbij aan rekenkundige en logische operaties zoals optellen, aftrekken en schuiven.

Een `integer` wordt gesynthetiseerd als een two's complement getal. Het aantal bits hangt af van het kleinste en grootste getal die de integer kan aannemen. Als er geen bereik met de **`range`**-clausule is opgegeven, worden 32-bits getallen gesynthetiseerd. Het getal 0 wordt altijd meegenomen ook al komt het niet in het bereik voor. Als het bereik geen negatieve getallen heeft, kan het zijn dat de tekenbit niet gesynthetiseerd wordt. Stel dat we een bereik van 1000 tot 1023 opgeven dan worden 10-bits getallen gesynthetiseerd. Een bereik van $-1$ tot 255 levert 9-bits getallen op.

Het type `character` wordt gesynthetiseerd als een 8-bits ASCII-karakter.

Enumeratietypen kunnen door de synthesizer vrijelijk worden gesynthetiseerd. Gewoonlijk wordt de binaire telcode oplopend gebruikt, maar andere coderingen zijn mogelijk zoals de Gray-code en *one-hot*-code. Voor one-hot wordt voor elke element van de enumeratie één bit gereserveerd. Op enig moment is slechts een van de bits logisch 1 (one-hot), de overige bits zijn dan 0.

Vectoren van de genoemde typen worden gesynthetiseerd naar het aantal elementen van de vector met dezelfde eigenschappen voor elk element.

Gewoonlijk kunnen twee-dimensionale vectoren worden gesynthetiseerd. Niet elke tool ondersteunt dat.

## 7.7 Rekenen met de types unsigned en signed

VHDL kent de bibliotheek `numeric_std` om rekenschakelingen te beschrijven. Het definieert twee nieuwe types: `unsigned` en `signed`. Signed getallen worden in de two's complement representatie weergegeven. In feite zijn dit vectoren van `std_logic`, net als `std_logic_vector`. Op deze nieuwe types zijn diverse operatoren gedefinieerd zoals optellen, aftrekken, vermenigvuldigen en delen. Er zijn ook conversieroutines, bijvoorbeeld tussen signed/unsigned en integer. Door de strikte hantering van types zijn signed/unsigned niet zonder meer te bewerken als `std_logic_vector`.

In listing 7.74 is een voorbeeld gegeven. Om de types `unsigned` en `signed` te kunnen gebruiken moet de bibliotheek `numeric_std` geladen worden. Dat is in de derde regel te zien. Verder is te zien dat '+' en '−' gebruikt kunnen worden voor optellen en aftrekken. Bij deze oplossing is er geen overflowdetectie.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 ...
5 signal sum, a, b : unsigned(3 downto 0);
6 signal dif, c, d : signed(3 downto 0);
7 ...
8 sum <= a + b;
9 dif <= c - d;
```

**Listing 7.74:** *Voorbeelden van optellen en aftrekken met signed en unsigned.*

Voor het detecteren van een overflow bij unsigned getallen is een extra sombit nodig. Deze meest significante bit geeft aan of er unsigned overflow is. Dit is te zien in listing 7.75. Er wordt gebruik gemaakt van een tijdelijk signaal `tmp` dat één bit groter is dan de signalen `a` en `b`. Om deze signalen uit net zoveel bits te laten bestaan, wordt een '0' voor de signalen geplaatst. De uitkomst wordt in tweeën gesplitst: de vier minst significantie bits worden aan `sum` toegekend, de meest significante bit wordt aan `cout` toegekend. Er is overflow geconstateerd als `cout` '1' is.

```
1 signal sum, a, b : unsigned(3 downto 0);
2 signal tmp : unsigned(4 downto 0);  -- one bit more
3 signal cout : std_logic;
4 ...
5 tmp  <= ('0' & a) + ('0' & b);
6 sum  <= tmp(3 downto 0);
7 cout <= tmp(4);
```

**Listing 7.75:** *Overflowdetectie bij optellen met unsigned getallen.*

Het detecteren van een overflow bij signed getallen volgt direct uit de twee op te tellen getallen en het antwoord. Er is sprake van signed overflow als de twee op te tellen getallen een gelijk teken hebben en het antwoord een ander teken. In listing 7.76 is de functie voor de overflow op logisch niveau en of RTL-niveau beschreven. Voor aftrekken moet de functie worden aangepast.

Een inkomende carry kan direct met een extra '+' aan een optelling worden toegevoegd. De inkomende carry is van het type `std_logic` en er is geen optelfunctie van een `unsigned` en `signed` vector en een bit van het type `std_logic`. Er zijn twee mogelijkheden om een inkomende carry te verwerken. Zie listing 7.77.

In de eerste situatie wordt de inkomende carry uitgebreid naar een vector met evenveel bits als `a` en `b`. Merk op dat er twee optellingen wordt beschreven. In de tweede situatie worden de vectoren `a` en `b` aan de rechterkant met één bit uitgebreid. Merk op dat in deze situatie er slecht één keer wordt opgeteld. Een synthesizer zal dus ook één opteller synthetiseren.

```
signal sum, a, b : signed(3 downto 0);
signal over : std_logic;

...
sum  <= a + b;
over <= (not a(3) and not b(3) and sum(3)) or
           (a(3) and b(3) and not sum(3));
-- or like this
over <= '1' when (a(3) = b(3)) and (a(3) /= sum(3)) else '0';
```

**Listing 7.76:** *Overflowdetectie bij optellen met signed getallen.*

```
   sum <= a + b + ("0" + cin);
   sum <= a + b + ("" + cin);      -- works also
   ...
   tmp <= (a & '1') + (b & cin);  -- tmp is a 5-bits vector
   sum <= tmp(4 downto 1);
```

**Listing 7.77:** *Verwerken van de inkomende carry.*

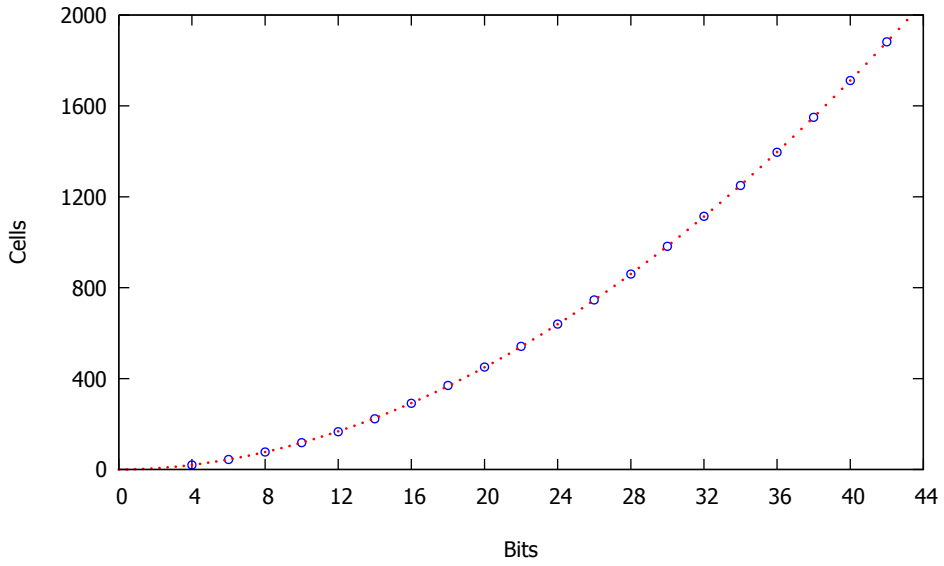
Vermenigvuldigen en delen kan ook. Let erop dat combinatorische schakelingen hiervoor veel poorten of cellen kost. De beschrijving in listing 7.78 van een 32-bits unsigned deler kost 1114 cellen (synthese met Altera Quartus II voor een Cyclone III FPGA). Voor een 32-bits signed deler kost de deler 1245 cellen.

```
entity divider_comb is
    generic (n: positive := 32);
    port (a : in unsigned(n-1 downto 0);
          b : in unsigned(n-1 downto 0);
          q : out unsigned(n-1 downto 0)
         );
end entity divider_comb;

architecture comb of divider_comb is
begin
    q <= a/b;
end architecture comb;
```

**Listing 7.78:** *Voorbeeld van een combinatorische 32-bits unsigned deler.*

In figuur 7.24 is te zien dat het aantal cellen voor een unsigned $n \times n$ deler bijna kwadratisch oploopt.

De bibliotheek `numeric_std` bevat nog veel meer routines, onder andere voor conversie, resizing van signed en unsigned, relationele bewerkingen en schuifoperaties. In listing 7.79 is het een en ander te zien. In de laatste vier regels is het gebruik van de `length`- attribute te zien. De functie `resize` wordt gebruikt om vectoren groter of kleiner te maken. Merk op dat functie voor zowel signed als unsigned gebruikt wordt. Bij het vergroten van een unsigned vector worden de nieuwe bits gevuld met nullen. Bij een signed vector wordt de tekenbit gekopieerd. Dit laatste wordt *sign extension* genoemd.




**Figuur 7.24:** *Aantal cellen voor een unsigned n × n deler.*

```
constant n : integer := 8;
signal sv1, sv2, sv3, sv4 : std_logic_vector (n-1 downto 0);
signal u1, u2, u3, u3 : unsigned (n-1 downto 0);
signal s1, s2, s3, s4 : signed (n-1 downto 0);A
signal n1, n2 : natural;
signal i1 : integer;

signal ls1 : signed(2*n-1 downto 0);     -- twice as long
signal lu1 : unsigned(2*n-1 downto 0);   -- ditto

sv1 <= std_logic_vector(u1);          -- unsigned to std_logic_vector
sv2 <= std_logic_vector(s1);          -- signed to std_logic_vector
u2  <= unsigned(sv3);                 -- std_logic_vector to unsigned
s2  <= signed(sv4);                   -- std_logic_vector to signed
n1  <= to_integer(u3);                -- unsigned to natural
i1  <= to_integer(s3);                -- signed to integer
u4  <= to_unsigned(n2, u4'length);    -- natural to unsigned
s4  <= to_signed(i2, s4'length);      -- integer to signed

lu1 <= resize(u1,lu1'length);         -- resize unsigned
ls1 <= resize(s1,ls1'length);         -- resize signed
```

**Listing 7.79:** *Voorbeeld van conversies tussen de verschillende types.*

## 7.8 RAM en ROM

Twee veel gebruikte componenten in digitale systemen zijn RAM en ROM. RAM staat voor Random Access Memory en ROM staat voor Read Only Memory. In een RAM kan data worden opgeslagen en weer terug worden gelezen. Een ROM heeft een vaste inhoud. De inhoud kan niet tussentijds wijzigen. Een ROM kan gezien worden als een elektronische waarheidstabel. RAM en ROM bestaan uit een aantal *geheugenplaatsen*.

Een geheugenplaats wordt aangegeven met een *adres*. Aangezien de adressen in het binaire talstelsel worden aangeboden, is het aantal geheugenplaatsen altijd een macht van 2. De signalen die het adres vormen, worden aangeduid met *adresbus*. De signalen waarlangs de data beschikbaar is, wordt *databus* genoemd.

We kunnen de ROM synthetiseren door middel van een twee-dimensionale vector. We declareren eerst het type `rom_type` en daarna declareren we een constante `rom` die gelijk ook waarden krijgt toegewezen. Zie listing 7.80.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity rom is
6      port (address : in std_logic_vector (2 downto 0);
7            oe : in std_logic;
8            data_out : out std_logic_vector (7 downto 0)
9           );
10 end entity rom;
11
12 architecture rtl of rom is
13 type rom_type is array (0 to 7) of std_logic_vector (7 downto 0);
14 constant rom : rom_type := ("11110000", "01011010", "11000110",
15                            "11111111", "00000000", "01001110",
16                            "00001111", "10101010");
17 begin
18     data_out <= rom(to_integer(unsigned(address))) when oe = '1' else
19                 (others => 'Z');
20 end architecture rtl;
```

**Listing 7.80:** *Een voorbeeld van een 8-bytes ROM met output enable.*

Aangezien de ROM uit acht adressen bestaat, declareren we een vector van acht elementen. Elk element is een vector van acht bits. In totaal zijn er dus 64 databits. De bits worden geïnitialiseerd met de waarden zoals te zien is in de listing. Het adres wordt aangeboden als een 3-bits vector. Het adres moet omgezet worden naar een integer. Daarvoor zijn twee conversieroutines nodig. Eerst moet de `std_logic_vector` worden omgezet naar een `unsigned` en die moet naar een `integer` worden omgezet. Het gebruik van de twee conversieroutines levert geen extra hardware op. Tevens is te zien dat de uitgangen in High-Z kunnen worden gezet als signaal `oe` (output enable) logisch 0 is. Op deze manier is het mogelijk om de uitgangen van meerdere ROMs aan elkaar te koppelen.

In listing 7.81 is de code van een 8-bytes RAM te zien. Er zijn in feite twee databussen nodig: één om de data in het geheugen te schrijven en één om de data te lezen. Er is een signaal `we` (write enable) dat aangeeft dat er data opgeslagen moet worden. Dit alles gebeurt onder besturing van een klokflank. Ook hier zijn weer conversieroutines nodig om het adres om te zetten naar een integer. Merk op dat de uitgaande data niet onder de besturing van een klokflank staat. Het adres geeft zowel de plek voor schrijven als lezen aan. Er zijn ook RAMs die maar één databus hebben. Dat spaart aansluitingen uit. De databus moet dan data twee kanten op kunnen sturen. We spreken dan van een *bidirectionele databus*.

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

entity ram is
    port (clk : in std_logic;
          areset : in std_logic;
          we : in std_logic;
          address : in std_logic_vector (2 downto 0);
          data_in : in std_logic_vector (7 downto 0);
          data_out : out std_logic_vector (7 downto 0)
         );
end entity ram;

architecture rtl of ram is
type ram_type is array (0 to 7) of std_logic_vector (7 downto 0);
signal ram : ram_type;
begin

    process (clk, areset, address, ram) is
    begin
        if areset = '1' then
            ram <= (others => "00000000");
        elsif rising_edge(clk) then
            if we = '1' then
                ram(to_integer(unsigned(address))) <= data_in;
            end if;
        end if;
        data_out <= ram(to_integer(unsigned(address)));
    end process;

end architecture rtl;
```

**Listing 7.81:** *Een voorbeeld van een 8-bytes RAM met write enable.*

## 7.9 Hints bij simulatie en synthese

In de onderstaande opsomming zijn enige hints te vinden:

- Bedenk dat de code hardware genereert. Vermijd constructies die bij programmeertalen gebruikt worden. Een typische *programmeerconstructie* is te zien in listing 7.82. De bedoeling is dat signaal `count` 0 wordt als deze 9 is, bijvoorbeeld als onderdeel van een teller. Hier wordt ten onrechte van uitgegaan dat de ophoging van signaal `count` gelijk uitgevoerd wordt en dat kort de waarde 10 aan `count` toegekend wordt. We vergeten alleen even dat de toekenning pas één delta later plaatsvindt. Tijdens het testen met **`if`** is de ophoging nog niet uitgevoerd. De telstand 10 blijft dus nog langer zichtbaar voor de rest van de schakeling.

  De juiste manier om tussen 0 en 9 te blijven, is te zien in listing 7.83. Eerst wordt getest of `count` gelijk is aan 9 waarna het signaal op 0 gezet wordt, anders wordt `count` met één verhoogd.

```
...
count <= count+1;
if count = 10 then
    count <= 0;
end if;
...
```

**Listing 7.82:** *Incorrecte toekenning.*

```
...
if count = 9 then
    count <= 0;
else
    count <= count+1;
end if;
...
```

**Listing 7.83:** *Correcte toekenning.*

- Het **`while`**-lus statement kan veelal niet gesynthetiseerd worden omdat het bereik (range) tijdens het compileren bekend moet zijn.
- Het **`for`**-lus statement kan gesynthetiseerd worden als het bereik (range) bekend is tijdens compileren.
- Zorg ervoor dat signalen bekende beginwaarden hebben voordat een blok conditionele code wordt beschreven, bijvoorbeeld bij het gebruik van **`if`** en **`case`**.
- Maak geen gebruik van initiële waarden voor signalen en variabelen. Dit heeft alleen effect bij simulatie. De meeste synthesizers negeren ze. Initialisatie moet altijd gebeuren onder besturing van een resetsignaal (bij sequentiële schakelingen).
- Specificeer expliciet bereiken in de declaratie van signalen en variabelen. Dit zorgt ervoor dat er niet teveel bits worden gesynthetiseerd.
- Gebruik geen don't care symbolen ('—') in vergelijkingen. In simulatie wordt een signaal ook echt met don't care vergeleken en dat is vaak niet de bedoeling. Er bestaat geen hardware-equivalent van een don't care en deze vergelijkingen leveren altijd niet waar (false) op. Een uitzondering hierop is de functie `std_match()` gedefinieerd in `numeric_std`. Met deze functie is het mogelijk om een zogenoemde *wild card matching* uit te voeren. In listing 7.84 is te zien hoe twee bits van een 4-bits vector kunnen worden getest.

```
...

if std_match(q,"1--0") then    -- test if q(3)='1' and q(0)='0'
    ...
end if;
...
```

**Listing 7.84:** *Een test met std_match.*

- Gebruik altijd flankgevoelige geheugenelementen (flipflops) in sequentiële schakelingen d.m.v. de functies `rising_edge()` of `falling_edge()`.
- Zorg ervoor dat een design unit óf alleen structuur bevat (structural VHDL) óf gedragsbeschrijving (dataflow en/of sequential VHDL). Uitzondering hierop zijn testbenches.

- Let erop dat alle signalen waar een proces op moet reageren in de sensitivity list staan, anders is er een verschil tussen simulatie en synthese. Synthesizers negeren over het algemeen de sensitivity list.
- Zorg voor een balans tussen het aantal hiërarchische lagen en de grootte van de design units (zowel in code als in hardware). Probeer niet meer dat vier niveaus te beschrijven.
- Vergeet bij het **`if`**-statement niet om af te sluiten met een **`else`** als combinatorische logica moet worden beschreven. Anders worden er latches gesynthetiseerd.
- Een proces heeft óf een sensitivity list óf en of meerdere **`wait`**-statements.
- Een **`wait`**-statement in een proces zorgt voor het updaten van signalen.
- De uitvoering van de toekenning aan een variabele gebeurt direct. Uitvoering van de toekenning aan een signaal gebeurt één delta later (als er geen **`after`** is gebruikt).
- Gebruik voor het representeren van binaire getallen de types `signed` en `unsigned`. Er zijn wel mogelijkheden om het type `std_logic_vector` te gebruiken maar dat wordt afgeraden.
- Bij gebruik van de types `signed` en `unsigned` wordt aanbevolen om het bereik als **`downto`** te specificeren. Deze vectoren stellen over het algemeen binaire getallen voor.

## 7.10 Opgaven

**7.1.** Geef de logische functie van onderstaande CSA:

```
1 s <= '1' when a = '0' else
2      '1' when b = '1' else
3      '1' when c = '0' else
4      '0';
```

**Listing P7.1:** *VHDL-beschrijving Conditional Signal Assignment.*

**7.2.** Geef de logische functie van onderstaande CSA:

```
1 s <= a when data = '0' else
2      b when data = '1' and en = '0' else
3      c when en = '0' else
4      d;
```

**Listing P7.2:** *VHDL-beschrijving Conditional Signal Assignment.*

**7.3.** Geef een VHDL-beschrijving van een EXOR d.m.v. een Conditional Signal Assignment en een Selected Signal Assignment.

**7.4.** Geef de VHDL-statement(s) om een 8-bits vector één plek naar rechts te schuiven.

**7.5.** Geef de VHDL-statement(s) om een 8-bits vector één plek naar links te roteren, waarbij de meest significante bit in de minst significante bit wordt geplaatst (dit wordt roteren genoemd).

**7.6.** Geef de volledige VHDL-beschrijving van een SR-latch met overheersende set.

**7.7.** Geef een VHDL-beschrijving voor een EXOR-poort door middel van sequentiële VHDL-statements. Geef zoveel mogelijke oplossingen.

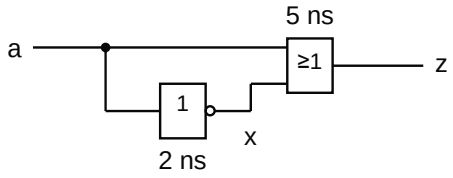
**7.8.** Geef een VHDL-beschrijving voor een negative edge triggered D-flipflop.

**7.9.** Geef een VHDL-beschrijving voor een 8-bits schuifregister dat schuift op de opgaande flank.

**7.10.** Geef de VHDL-beschrijving van één sectie van een BCD-opteller. De BCD-opteller heeft een extra uitgang die aangeeft of de op te tellen cijfers BCD-cijfers zijn. De BCD-opteller heeft een inkomende carry.

**7.11.** Eerder is code van een 8-input NOR gegeven. Geef nog twee andere mogelijkheden voor het beschrijven van de NOR. Gebruik de eerder gegeven code als leidraad.

**7.12.** Gegeven de schakeling in figuur P7.1. Alle signalen zijn van het type `bit` en op '0' geïnitialiseerd. Op tijdstip t = 3 ns wordt `a` logisch 1. Reken het schema door en stel de event-lijsten op voor elk event-tijdstip.




**Figuur P7.1:** *Schema voor doorrekenen tijdgedrag.*

**7.13.** Als opgave 7.12, maar nu wordt signaal `a` na 10 ns weer logisch 0.

**7.14.** Als opgave 7.12, maar nu als geen vertragingen zijn opgegeven.

**7.15.** Geef het schema van onderstaande VHDL-codes. Maak gebruik van multiplexers en poorten.

```
1  if sel = '1' then
2     y <= a and b;
3  else
4     y <= a or b;
5  end if;
```

**Listing P7.3:** *VHDL-code voor synthese.*

```
1  if x = y then
2      z <= '1';
3  else
4      z <= '0';
5  end if;
```

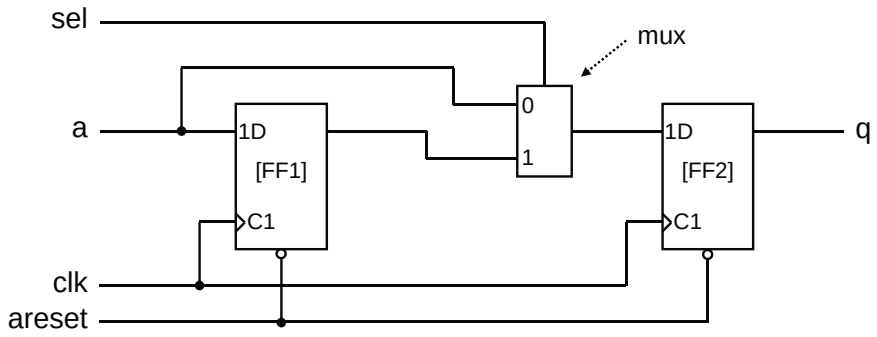
**Listing P7.4:** *VHDL-code voor synthese.*

**7.16.** Geef het schema van de VHDL-code in listing P7.5.

```vhdl
if rising_edge(clk) then
    ff1 <= x;
    if ff1 = '1' and x = '0' then
        ff2 <= '1';
    else
        ff2 <= '0';
    end if;
end if;
```

**Listing P7.5:** *VHDL-code voor synthese.*

**7.17.** Geef de complete VHDL-beschrijving van het schema in figuur P7.2.




**Figuur P7.2:** *Schema voor ontwerp VHDL-code.*

**7.18.** Geef het schema van de onderstaande code. Probeer zo dicht mogelijk de originele code te volgen.

```vhdl
signal x : bit_vector (7 downto 0);
signal q : bit;
    ...
    process (x) is                  -- sensitive for x
    variable p : bit;               -- only p, no need to declare i
    begin
        p := '0';                   -- initialize to 0
        for i in 7 downto 0 loop    -- i = 7,6,5,4,3,2,1,0
            p := p or x(i);         -- OR p with each element
        end loop;
        q <= not p;                 -- signal assignment
    end process;
```

**Listing P7.6:** *VHDL-beschrijving van een 8-input NOR voor synthese.*

**7.19.** Geef de VHDL-beschrijving van een 32x32-bits parallelle vermenigvuldiger. Bedenk dat het product een 64-bits getal is.

# 8

# Schuifregisters en tellers

In dit hoofdstuk behandelen we twee veel gebruikte digitale componenten: het schuifregister en de teller. Schuiven is een veel voorkomende bewerking in de digitale techniek. Denk alleen maar aan seriële transmissie: zender en ontvanger schuiven de bits een voor een over de lijn. Daarnaast heeft schuiven ook een rekenkundige betekenis. Vermenigvuldigen met en delen door 2 is niets anders dan de bits één plaats verschuiven. De andere veel voorkomende bewerking is tellen. Een teller kan bijvoorbeeld bijhouden hoe vaak en bepaalde gebeurtenis is voorgekomen, maar het is ook mogelijk om tijd de tellen. Dan kan een teller gebruikt worden tijdsintervallen te genereren.

We beginnen het hoofdstuk met de introductie over schuifregisters. We laten de opbouw zien en bespreken hoe schuiven in VHDL geïmplementeerd wordt. We geven een voorbeeld van seriële communicatie: een RS-232-zender.

Het tweede deel behandelt tellers. We laten zien hoe het telproces in z'n werk gaat en bouwen tellers op rond dit telproces. Daarna bespreken we hoe tellers met VHDL beschreven kunnen worden. We geven twee voorbeelden van het gebruik van tellers. We ontwerpen een digitale sinusgenerator die kan dienen als testsignaal en we behandelen het ontwerp van een PWM-generator. PWM staat voor Pulse Width Modulation. Hierbij wordt een (digitale) waarde voorgesteld met een bepaalde pulsduur of pulsbreedte.

Als laatste bespreken we twee speciale tellers: de gewone ringteller en de Johnson-teller. Dit zijn eigenlijk schuifregisters die als teller gebruikt worden.

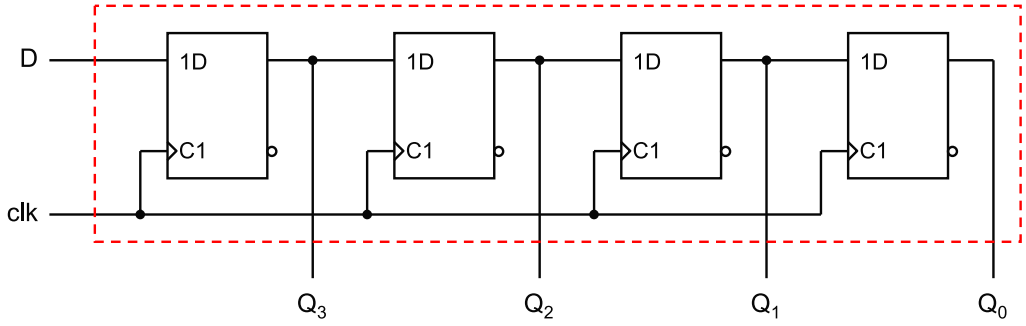
## 8.1 Schuifregisters

In de digitale techniek en met name in de digitale communicatie wordt veel gebruik gemaakt van seriële transmissie. Het voordeel ten opzichte van parallelle transmissie is overduidelijk: er is slechts een beperkt aantal verbindingen nodig om informatie (bits) te verzenden of te ontvangen. Het nadeel is dat het versturen van een byte of word veel langer duurt dan bij parallelle transmissie. Voorbeelden waarbij seriële transmissie wordt

gebruikt zijn:

- RS232 (beter bekend als de COM-poort in de PC)
- Ethernet
- $I^2C$ (wordt ook wel Two Wire Interface genoemd)
- SPI
- Infrarood afstandsbediening
- USB
- SATA
- CD/DVD/BD

Bij seriële transmissie worden *schuifregisters* gebruikt. Een schuifregister is een groep (D-)flipflops waarvan de data-uitgang van een flipflop is verbonden met de data-ingang van de naastgelegen flipflop. In figuur 8.1 is een een 4-bits SIPO-schuifregister te zien. SIPO staat voor Serial In Parallel Out. De data op ingang *D* wordt op elke opgaande flank ingeklokt. De overige flipflops nemen hun data over van de uitgang van links gelegen flipflop. Er zijn maar vier flipflops beschikbaar dus na verloop van tijd verdwijnen bits aan de rechterkant. We noemen dat “eruit geschoven”.




**Figuur 8.1:** *Schema van een 4-bits schuifregister.*

Bij het bespreken van schuifregisters wordt gesproken van linksom schuiven (left shift) en rechtsom schuiven (right shift). Linksom is letterlijk zoals op papier zou worden getekend van rechts naar links. Rechtsom is letterlijk zoals op papier zou worden getekend van links naar rechts. De schuifregisters worden getekend met de meest significante bit aan de linkerkant.

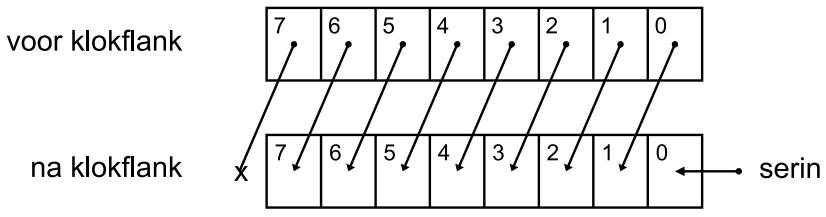
Schuiven kan in VHDL eenvoudig gerealiseerd worden met behulp van de concatenation operator (`&`). In listing 8.1 is de code te zien van een 8-bits schuifregister dat naar links schuift. De schakeling heeft naast een klok en een reset ook een seriële data-ingang. Als uitgang heeft het schuifregister een 8-bits waarde.

De code is opgebouwd rond een asynchrone reset en een opgaande klokflank. Als de reset geactiveerd is, worden alle flipflops op logisch 0 gezet middels een *aggregate*. Regel 21 beschrijft het schuiven van de vector. De zeven minst significante bits worden gebruikt en aangevuld met een ‘0’. Dit levert een 8-bits vector die aan `q_int` wordt toegekend.

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

entity shiftreg_simple is
    port (clk : in std_logic;
          areset : in std_logic;
          serin : in std_logic;
          q : out std_logic_vector (7 downto 0)
         );
end entity shiftreg_simple;

architecture rtl of shiftreg_simple is
signal q_int : std_logic_vector (7 downto 0);
begin

    process (clk, areset) is
     begin
        if areset = '1' then                        -- reset
            q_int <= (others => '0');
        elsif rising_edge(clk) then                 -- positive edge
            q_int <= q_int(6 downto 0) & serin;     -- shift left
        end if;
    end process;

    q <= q_int;                                     -- copy data

end architecture rtl;
```

**Listing 8.1:** *VHDL-beschrijving van een 8-bits schuifregister.*

Hoe de constructie werkt is te zien in figuur 8.2. Voor de klokflank staan alle bits op de juiste plaats in het register. Na de klokflank is te zien dat bits 6 t/m 0 (let op dat de vector als **downto** is gespecificeerd) één plaats naar links zijn geschoven en dat de waarde van bit 0 van de seriële ingang afkomstig is. Bit 7 wordt, zoals eerder is gezegd, "eruit geschoven" en verdwijnt. Merk op dat een dergelijke constructie alleen onder een klokflankbesturing mag worden beschreven anders wordt er asynchrone logica gesynthetiseerd.

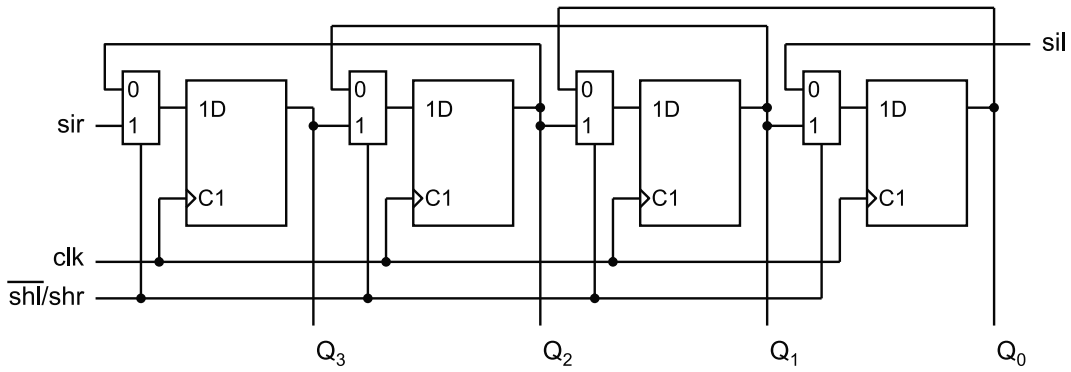



**Figuur 8.2:** *Uitleg linksom schuiven.*

### 8.1.1 Bidirectionele schuifregisters

Er bestaan ook bidirectionele schuifregisters. Dat zijn schuifregister die twee kanten kunnen opschuiven. In figuur 8.3 is het schema van het bidirectionele schuifregister te zien. Er wordt gebruik gemaakt van D-flipflops en 2x1 multiplexers. Het signaal $\overline{\text{shl}}$/shr

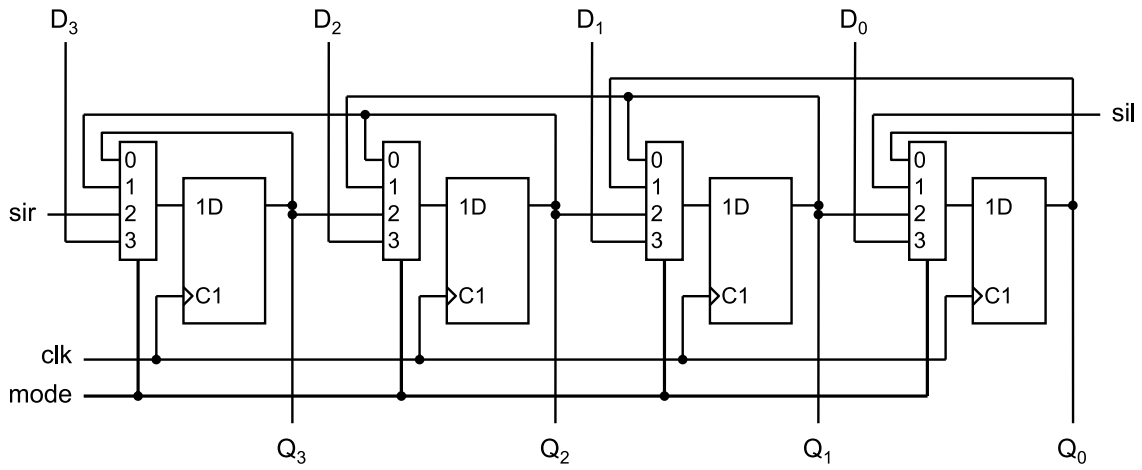
zorgt voor de schuifrichting. Als het signaal logisch 0 is, wordt er linksom geschoven, bij een logische 1 wordt er rechtsom geschoven.




**Figuur 8.3:** *Schema van een 4-bits bidirectioneel schuifregister.*

### 8.1.2 Universeel schuifregister

Een *universeel schuifregister* [14] heeft vier werkmodi. Het register heeft de mogelijkheid om linksom en rechtsom te schuiven. Daarnaast kan het ook parallelle data *inladen* en heeft het de mogelijkheid om data onveranderd in het register te laten staan, ook al passeert er een actieve klokflank. Deze laatste mogelijkheid wordt *hold* genoemd. In figuur 8.4 is het schema van het universele schuifregister te zien. Er is gebruik gemaakt van D-flipflops voor de opslag van de bits en van 4x1 multiplexers om de juiste data te selecteren.



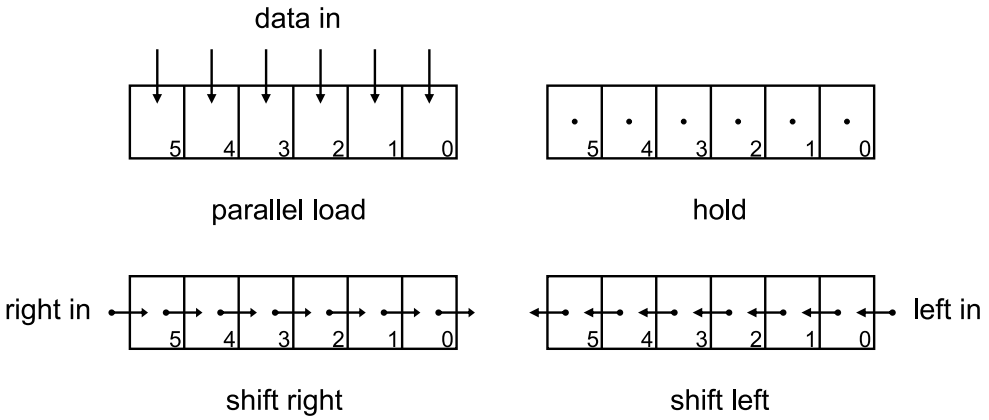

**Figuur 8.4:** *Schema van een 4-bits universeel schuifregister.*

De toekenning aan de functies van signaal `mode` is arbitrair maar een veel voorkomende toekenning is in tabel 8.1 gegeven.

De vier werkmodi zijn weergegeven in figuur 8.5. Het betreft in dit geval een 6-bits register. De uitgangen zijn niet getekend. Bij schuiven verdwijnt een bit aan de linker- of rechterkant.

**Tabel 8.1:** *Toekenning van de vier werkmodi.*

| mode | operatie |
|---|---|
| 00 | hold |
| 01 | shift left |
| 10 | shift right |
| 11 | load |




**Figuur 8.5:** *De vier werkmodi van het universeel schuifregister (uitgangen niet getekend).*

In listing 8.2 is de code van het universele schuifregister te zien. De reset is achterwege gelaten maar moet normaliter wel geïmplementeerd worden. De werkmodi worden door middel van de 2-bits vector opgegeven. Er is gebruik gemaakt van een *generic constant* waarmee het aantal bits van het schuifregister wordt aangeven. Zo is de code makkelijk te hergebruiken. Voor de verandering is een variabele gebruikt om het interne schuifregister te beschrijven.

De werkmodi worden met een `case`-statement beschreven zoals aangegeven in tabel 8.1. Merk het keyword `null` op. Dit geeft aan dat onder die mogelijkheid niets moet worden gedaan.

### 8.1.3 Seriële transmissie

Schuifregisters worden gebruikt bij seriële transmissie waarbij bits een voor een over een verbinding worden gestuurd. Het pricipeschema is te zien in figuur 8.6. Dit wordt een tweedraadsverbinding genoemd maar dat is niet geheel juist. Bij elektrische systemen is er ook nog een referentie nodig.

De zender heeft een uitgaande datalijn en een uitgaande kloklijn. De ontvanger heeft een inkomende datalijn en een inkomende kloklijn. Hiermee kan de ontvanger de juiste momenten van dataoverdracht van de bits bepalen.

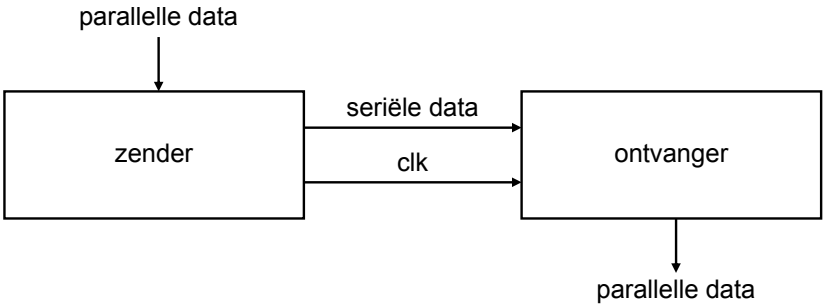
Bij verzenden van data gaat het omzetten van parallel naar serieel als volgt:

Eerst wordt de (parallelle) data geladen.

Daarna wordt de data serieel naar buiten geschoven.

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

entity shiftreg is
    generic (n : positive := 6);
    port (clk  : in std_logic;
          mode : in std_logic_vector(1 downto 0);
          data : in std_logic_vector(n-1 downto 0);
          sir  : in std_logic;
          sil  : in std_logic;
          q    : out std_logic_vector(n-1 downto 0)
         );
end entity shiftreg;

architecture rtl of shiftreg is
begin

    process (clk) is
    variable qint : std_logic_vector(n-1 downto 0);
    begin
        if rising_edge(clk) then
            case mode is
                -- hold
                when "00" => null;
                -- shift left
                when "01" => qint := qint((n-1)-1 downto 0) & sil;
                -- shift right
                when "10" => qint := sir & qint(n-1 downto 1);
                -- load
                when "11" => qint := data;
                -- do nothing
                when others => null;
            end case;
        end if;
        q <= qint;
    end process;

end architecture rtl;
```

**Listing 8.2:** *VHDL-beschrijving van een universeel schuifregister.*




**Figuur 8.6:** *Principeschema seriële transmissie.*

Deze vorm van verwerken wordt Parallel In Serial Out (PISO) genoemd. Bij ontvangen van data gaat het omzetten van serieel naar parallel als volgt:

Eerst wordt de data serieel naar binnen geschoven.

Daarna wordt de (parallelle) data beschikbaar gesteld.

Dit wordt Serial In Parallel Out (SIPO) genoemd. Het schuifregister in figuur 8.1 is een SIPO-schuifregister. Een praktisch voorbeeld van een SIPO-schuifregister is het bekende ic 74HC595 [15].

Nu rijst de vraag: welke bit wordt als eerste naar buiten (of binnen) geschoven? Dat is systeemafhankelijk. Bij RS232 wordt de minst significante bit eerst verwerkt, bij Ethernet de meest significante bit eerst. De ontwerper moet dus goed opletten.

### 8.1.4 Vermenigvuldigen met en delen door 2

Schuiven heeft ook nog een rekenkundige functie. Naar links schuiven en aanvullen met nullen is hetzelfde als vermenigvuldigen met 2. Dit is te zien in figuur 8.7. Elke keer als er één plek naar links wordt geschoven en aangevuld wordt met een 0, wordt er vermenigvuldigd met 2.

1011 – het getal (+11 unsigned)
10110 – eerste keer schuiven (×2)
101100 – tweede keer schuiven (×4)
1011000 – derde keer schuiven (×8 = +88 unsigned)

**Figuur 8.7:** *Vermenigvuldigen met 2 is schuiven naar links (unsigned).*

Naar rechts schuiven en aanvullen met 0 is hetzelfde als delen door 2. Dit is te zien in figuur 8.8. De minst significante bits gaan wel verloren.

1011011 – het getal (+91 unsigned)
0101101 – eerste keer schuiven (÷2)
0010110 – tweede keer schuiven (÷4)
0001011 – derde keer schuiven (÷8 = +11 unsigned)

**Figuur 8.8:** *Delen door 2 is schuiven naar rechts (unsigned).*

Het is ook mogelijk om signed naar links te schuiven. In figuur 8.9 is dit te zien. In feite is dit hetzelfde als unsigned naar links schuiven. Het resultaat moet wel passen.

1111011 – het getal (−5 signed)
1110110 – eerste keer schuiven (×2)
1101100 – tweede keer schuiven (×4)
1011000 – derde keer schuiven (×8 = −40 signed)

**Figuur 8.9:** *Vermenigvuldigen met 2 is schuiven naar links (signed).*

Signed naar rechts schuiven gaat ook, maar er is wel een voorziening nodig om de tekenbit te behouden, anders zou een getal ineens positief worden. Zie figuur 8.10.

De minst significante bits gaan wel verloren. Negatieve getallen worden naar beneden afgerond.

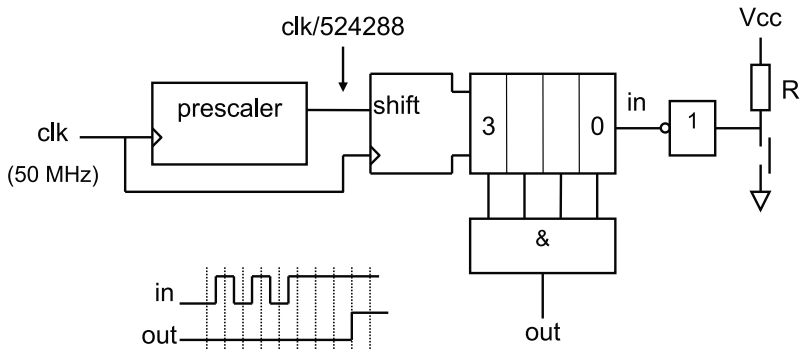
1011011 – het getal (−37 signed)
1101101 – eerste keer schuiven (÷2)
1110110 – tweede keer schuiven (÷4)
1111011 – derde keer schuiven (÷8 = −5 signed)

**Figuur 8.10:** *Delen door 2 is schuiven naar rechts (signed).*

Naar links schuiven is identiek voor unsigned en signed. In de computertechniek wordt dit *logical shift left* (LSL) respectievelijk *arithmetic shift left* (ASL) genoemd. Naar rechts schuiven is voor unsigned en signed verschillend. Unsigned naar rechts schuiven wordt ook wel *logical shift right* (LSR) genoemd en signed naar rechts schuiven wordt *arithmetic shift right* (ASR) genoemd.

## 8.2 Voorbeeld: ontdenderen schakelaar

Een mechanische drukknop of schakelaar heeft last van *denderen*. Bij het overgaan van de ene naar de andere stand stuitert de metalen kontaktveer. Dit levert meerdere pulsen op aan een ingang. Een schuifregister biedt uitkomst. Het principeschema is te zien in figuur 8.11.




**Figuur 8.11:** *Principeschema ontdenderen.*

We gaan uit van het *bemonsteren* van het ingangssignaal. Een schuifregister klokt op zekere momenten de ingangswaarde in. Als de schakelaar wordt omgezet, zullen er vanwege het denderen signaalwisselingen plaatsvinden op de ingang. Het schuifregister klokt zodoende nullen en enen in. Als gedurende een langere tijd de ingangswaarde constant is, is de schakelaar uitgedenderd. In de figuur moet vier keer een 1 achter elkaar worden ingeklokt voordat de uitgang 1 wordt.

Er moet worden vastgesteld hoe lang het denderen duurt. Dat kan proefondervindelijk maar het wordt ook vaak in de datasheets vermeld. Voor kleine schakelaars geldt een dendertijd van 0,2 ms tot 4 ms, maar grote schakelaars kunnen wel een dendertijd van 40 ms hebben. Dat is vele malen sneller dan de periodetijd van de systeemklok. Het gebruik van een *prescaler* biedt uitkomst. De prescaler zorgt ervoor dat eens in de zoveel klokpulsen een schuifopdracht wordt gegeven. In de figuur is dit eens in de 524.288 ($2^{19}$)

klokpulsen. De bemonsteringsfrequentie ligt dan in de buurt van de 100 Hz (periodetijd van 10 ms). Het ingangssignaal moet dus 40 ms stabiel zijn voor een positief resultaat.

## 8.3 Voorbeeld: RS232-communicatie

In deze paragraaf behandelen we een van de oudste seriële communicatiesystemen: RS-232. In tegenstelling tot veel andere seriële systemen beschikt RS-232 niet over een gemeenschappelijk kloksignaal. Dit wordt dan ook asynchrone communicatie genoemd. RS-232 is in zijn eenvoud toch aardig complex. Daarom behandelen we slechts een van de mogelijkheden. Merk op dat de RS-232 interface (ook wel de COM-poort genoemd) tegenwoordig steeds minder aanwezig is op een PC. Daarentegen zijn microcontrollers nog steeds uitgerust met een RS-232 interface.

### Inleiding

Een zeer eenvoudige vorm van seriële communicatie is in 1962 ontwikkeld onder de officiële naam EIA/TIA-232-F [16], beter bekend als RS-232 (RS staat voor "Recommended Standard"). De specificatie beschrijft tal van zaken zoals de opbouw van de connectoren, de signaalnamen en de betekenis van de signalen, logica- en spanningsniveaus en frameopbouw. Het beschrijft niet wat de betekenis van de verstuurde data is, dat wordt aan de applicatie overgelaten.

De RS-232-standaard is zeer uitgebreid. De standaard beschrijft een behoorlijk aantal configuraties. Veel van deze configuraties zijn terug te voeren naar het gebruik van modems en verbindingen via de ouderwetse telefoonlijnen. Wij zullen ons houden aan een eenvoudige variant die veel gebruikt wordt bij directe verbindingen, dus zonder modems.

### Signalen

De meest eenvoudige vorm van RS-232-communicatie bestaat uit slechts drie signalen en dus drie draden: TxD (transmit data), RxD (receive data) en GND (ground). Hardwarematige *handshaking* (signalering die zorgt voor correcte dataoverdracht) is niet mogelijk en dat houdt in dat zender en ontvanger voldoende snel moeten zijn om de datastroom te kunnen verwerken. Merk op dat er geen kloksignaal is. Een mogelijke connector is de zogenoemde 9-pins Sub-D connector. Pin 2 op de connector wordt gebruikt voor RxD en pin 3 wordt gebruikt voor TxD, tenminste als het over een PC gaat. Bij een modem is het andersom. Daarom staat RS-232 ook wel voor "Regelmatig Solderen van pin 2 naar pin 3 naar pin 2".

### Spanningsniveau's

De fysieke spanningen voor de datalijnen van RS-232 zijn als volgt gedefinieerd:

| | | |
|---|---|---|
| logisch '0' | +3 V tot +25 V | 'space' |
| logisch '1' | −3 V tot −25 V | 'mark' |

Het spanningsbereik tussen −3 V en +3 V is dus ongeldig. Wat opvalt is dat een negatieve spanning een logische 1 representeert en een positieve spanning een logische 0. Hoewel de spanningen in het gebied van −25 V tot +25 V liggen zijn de *nominale* spanningen op −12 V en +12 V. Merk op dat de zender een iets hogere minimale spanning moet leveren in verband met spanningsval op de signaallijnen.

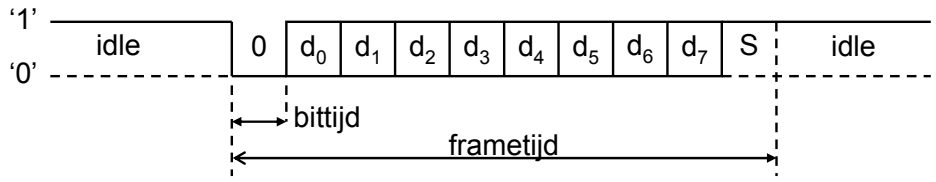
Omdat er veel gebruik wordt gemaakt van microcontrollers en digitale systemen bestaat er ook zoiets als TTL-level RS-232. Daarin wordt een logische 0 voorgesteld met 0 V (ground) en een logische 1 met de voedingsspanning bijvoorbeeld +3,3 V of +5 V. Om signalen uit te wisselen tussen echte RS-232-systemen en TTL-level wordt gebruik gemaakt van *level shifters* zoals de MAX232 [17].

### Seinsnelheid

RS-232 is eigenlijk bedoeld voor seinsnelheden tot 20000 bps (bits/sec). In RS-232-terminologie wordt dit de *baud rate* genoemd. Deze term heeft echter meer met signalering tussen modems te maken dan met zuivere bits. In de praktijk worden snelheden van 115200 bps en hoger gebruikt.

### Frameformaat

Het frameformaat bestaat uit een startbit, gevolgd door een aantal databits, een optionele *pariteitsbit* en één of meer stopbits. De pariteitsbit wordt gebruikt voor foutdetectie. Het geeft aan of het aantal bits in de te verzenden data even (Engels: even) of oneven (Engels: odd) is. In de praktijk wordt de pariteitsbit nauwelijks gebruikt vanwege de beperkte foutdetectiemogelijkheden. Het aantal databits varieert tussen vijf en negen. De meest gebruikte combinatie is acht databits, geen pariteit en één stopbit. Dit wordt aangegeven met de afkorting 8N1. In figuur 8.12 is het frameformaat geschetst.



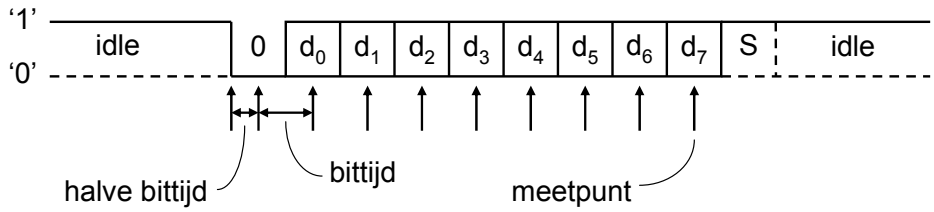

**Figuur 8.12:** *Het 8N1-formaat van een RS-232 frame*

In rust (idle mode) is de zendlijn (TxD) logisch 1. De startbit heeft de logische waarde 0 (zie uitleg verderop). Daarna volgen de acht databits beginnend met de minst significante bit. Daarna volgt een stopbit die logisch 1 is. We zouden kunnen denken dat deze bit weggelaten kan worden omdat de lijn in rust toch al logisch 1 is. Maar dan is het verschil niet zien tussen een laatste databit die 0 is en een startbit. De stopbit is dus nodig voor synchronisatiedoeleinden tussen zender en ontvanger.

### Zenden en ontvangen

Het is mogelijk om tegelijk te zenden en te ontvangen. Er zijn immers aparte zend- en ontvangstlijnen. Dat wordt *full duplex* genoemd. Dat houdt in dat het systeem een aparte zend- en ontvangstmodule bevat.

Het zenden gaat als volgt. Eerst wordt de startbit “op de lijn gezet”. Daarna volgen de acht databits en vervolgens volgt de stopbit. Daarmee is de transmissie klaar. Bij het ontvangen komt wat meer kijken. De ontvanger wacht tot er op de ontvangstlijn een hoog-laag wisseling plaatsvindt. Dat is het teken dat er een startbit is verzonden. Om er zeker van te zijn dat het echt om een startbit gaat en niet om een *spike* of *glitch* wordt na een *halve* bittijd de ontvangstlijn nog eens bekeken. Blijkt het echt om een startbit te gaan, dan wordt de ontvangst doorgezet. Daarbij is het verstandig om niet te dicht bij de signaalwisselingen van de bits te bemonsteren[1]. De exacte momenten liggen namelijk niet geheel vast (jitter van de klok aan de zendkant) en er is sprake van stijg- en daaltijden van de signalen. Merk ook op dat er geen kloksignaal wordt meegestuurd. Het is dus van belang dat de klokken aan de zend- en ontvangstkant op dezelfde frequentie lopen. De maximale afwijking is ongeveer 5%[2]. Het is dus verstandig om in het midden tussen twee signaalwisselingen te bemonsteren. Zie figuur 8.13.




**Figuur 8.13:** *Timing binnen een RS-232 frame.*

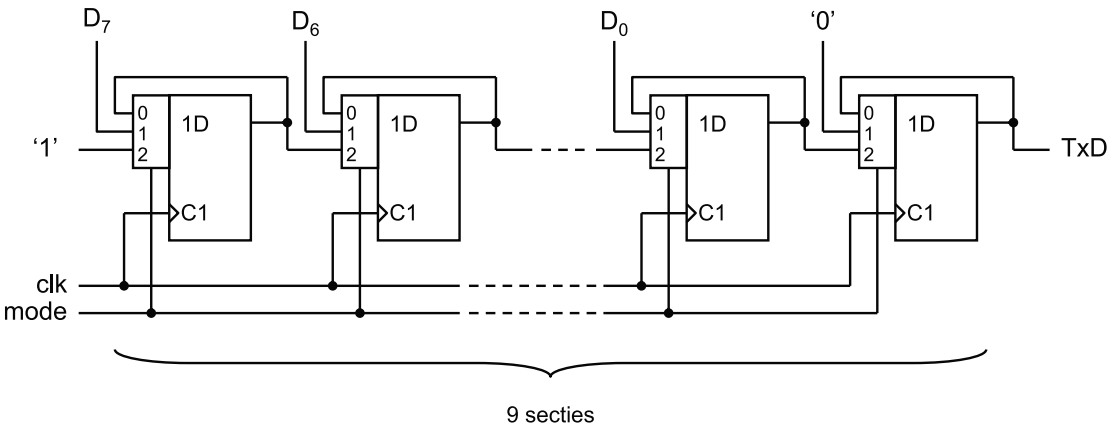
In figuur 8.14 is de opbouw van het zendregister te zien. Het register moet drie dingen kunnen: parallelle data laden, data schuiven en data vasthouden. Voordat er daadwerkelijk gezonden kan worden, moet de parallelle data eerst geladen worden. Te zien is dat ook de startbit geladen wordt. Dit zorgt ervoor dat tijdens het laden de startbit als eerste op de lijn komt. Aan de linkerkant is te zien dat de seriële data-ingang verbonden is met een logische 1. Dit zorgt ervoor dat na 8 keer schuiven de zendlijn op logisch 1 wordt gezet zodat automatisch de stopbit op de lijn wordt gezet. De frequentie van de systeemklok van de zendmodule is meestal vele malen groter dan de zendfrequentie. Dat houdt in dat slechts af en toe data geschoven moet worden. Een groot deel van de (zend-)tijd moet het register de data dus vasthouden. Het zendregister heeft dus ook een hold-mogelijkheid. Om voor een goede verzending te zorgen is er een *besturing* nodig. We laten dit buiten beschouwing.

## 8.4 Tellers

Tellers worden in digitale systemen voor veel doeleinden gebruikt. Een teller kan bijvoorbeeld bepaalde gebeurtenissen tellen. In Engelstalige literatuur wordt dan gesproken van een *counter*. Een teller kan ook tijd “tellen” en tijdintervallen genereren. Zo’n teller wordt een *timer* genoemd. Tellers hebben de eigenschap een getelde hoeveelheid te onthouden. Dat betekent dat tellers geheugen bezitten.

[1] Bemonsteren = samplen

[2] Dat is eenvoudig uit te rekenen. Een 8N1 RS-232-frame bestaat uit 10 bits. Het bemonsteren begint halverwege de startbit. Dan kan de ontvanger een afwijking hebben van een halve bittijd op 10 bittijden = 1/20 = 5%.




**Figuur 8.14:** *Opbouw van het zendregister voor RS-232-communicatie.*

### 8.4.1 Grondbeginselen van tellers

Een teller telt vrijwel altijd *cyclisch*. De lengte van de telcyclus hangt af van het aantal bits van de teller. Voor een $n$-bits teller is de telcyclus *maximaal* $2^n$ telstanden. Dit kan minder zijn afhankelijk van de toepassing van de teller. Hieronder een voorbeeld van een 4-bits binaire teller met 16 telstanden:

$0000 \rightarrow 0001 \rightarrow 0010 \rightarrow 0011 \rightarrow 0100 \rightarrow 0101 \rightarrow 0110 \rightarrow 0111 \rightarrow$

$1000 \rightarrow 1001 \rightarrow 1010 \rightarrow 1011 \rightarrow 1100 \rightarrow 1101 \rightarrow 1110 \rightarrow 1111 \rightarrow \underline{0000}$

Duidelijk is dat bij de huidige telstand steeds 1 wordt opgeteld om de nieuwe telstand te krijgen. Dit is het gedrag van een *incrementer*.

Tellen wordt meestal gedaan in het binaire stelsel, maar het is heel goed mogelijk in het decimale systeem te tellen. In dit geval worden de decimale cijfers in de BCD-code voorgesteld. Deze tellers worden vrijwel altijd modulair opgebouwd. In de BCD-code is dat van nature vier bits. De cyclus van een 3-decaden-teller loopt van 0 tot 999, waarna de teller weer in 000 start.

$000 \rightarrow 001 \rightarrow 002 \rightarrow 003 \rightarrow \ldots \rightarrow 099 \rightarrow 100 \rightarrow 101 \rightarrow 102 \rightarrow \ldots \rightarrow 999 \rightarrow$

$\underline{000} \rightarrow 001 \rightarrow 002 \rightarrow \ldots$

Specifieke eigenschappen van tellers zijn:

- Omhoog tellen (optellen, up counter)
- Omlaag tellen (aftellen, down counter)
- Omhoog/omlaag tellen (optellen/aftellen, up-down counter)
- Laden van een beginwaarde (load)
- Tellen activering met een enable
- Wissen, zowel asynchroon (reset) als synchroon (clear)
- Telbereik/telcodering (binair, BCD, Gray, One-hot, Johnson, ...)

Als een teller alleen maar een vast aantal gebeurtenissen moet (af-)tellen, kan de interne telstand vrij gekozen worden.

Tellers moeten kloksynchroon te worden ontworpen. Alle klokingangen van de flipflops zijn op hetzelfde kloksignaal aangesloten.

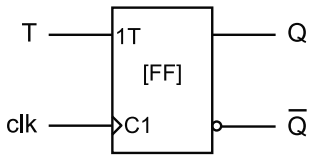
Het principe van tellen wordt weergegeven in figuur 8.15. Links is de telcyclus van een 1-bit teller te zien. We zien dat de telbit steeds van stand verandert. Dat wordt *toggelen*[3] genoemd.

| | | |
|---|---|---|
| 0 | 00 | 000 |
| $\underline{1}$ | $0\underline{1}$ | $00\underline{1}$ |
| 0 | 10 | 010 |
| $\underline{1}$ | $\underline{11}$ | $0\underline{11}$ |
| 0 | 00 | 100 |
| $\underline{1}$ | $0\underline{1}$ | $10\underline{1}$ |
| 0 | 10 | 110 |
| $\underline{1}$ | $\underline{11}$ | $\underline{111}$ |
| 0 | 00 | 000 |
| ⋮ | ⋮ | ⋮ |

**Figuur 8.15:** *Telcycli van 1-bit, 2-bits en 3-bits tellers.*

Bij de 2-bits telcyclus in het midden wordt al wat duidelijk. Als een telbit 1 is moet de links aangrenzende telbit toggelen en de telbit zelf wordt 0. Dit is duidelijk te zien bij de telcyclus van de 3-bits teller geheel rechts. Als de telstand 011 is, moet de linker bit toggelen en de minder significante bits worden 0. De onderstrepingen geven aan wanneer er getoggeld moet worden.

Het toggelen kan gedaan worden met een *T-flipflop*. In figuur 8.16 is het symbool en de functie te zien. Als de stuuringang $T$ logisch 1 is, verandert de flipflop van stand als een actieve klokflank passeert. Is de stuuringang $T$ logisch 0, dan behoudt de flipflop zijn stand. Later zullen we de D-flipflop gaan gebruiken.




$$Q^{n+1} = \begin{cases} Q^n & \text{als } T^n = 0 \\ \overline{Q^n} & \text{als } T^n = 1 \end{cases} \tag{8.1}$$

**Figuur 8.16:** *Symbool en functie van de T-flipflop.*

Het tellen gebeurt heel systematisch. Om een telbit te laten toggelen moeten alle voor-

[3] To toggle: tokkelen. In het geval van een logisch signaal spreken we liever van omklappen of van stand veranderen.

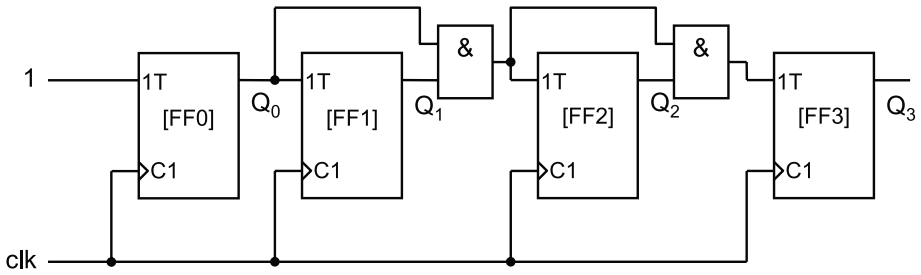
gaande telbits 1 zijn. We kunnen dit in formulevorm beschrijven:

$$\begin{aligned} T_0^n &= 1 \\ T_1^n &= Q_0^n \\ T_2^n &= Q_1^n \cdot Q_0^n \\ T_3^n &= Q_2^n \cdot Q_1^n \cdot Q_0^n \\ &\vdots \\ T_k^n &= Q_{k-1}^n \cdot Q_{k-2}^n \cdots Q_2^n \cdot Q_1^n \cdot Q_0^n \end{aligned} \tag{8.2}$$

(De superscript $\cdots^n$ is algemeen gebruik bij kloksynchrone logica). Ingang $T_0$ moet altijd logisch 1 zijn, want deze flipflop moet altijd toggelen. Ingang $T_1$ moet alleen toggelen als $Q_0$ logisch 1 is. Voor ingang $T_k$ geldt dat alle minder significante $Q$-uitgangen logisch 1 zijn als er getoggeld moet worden. Wat opvalt is dat de $T$-ingangen, met uitzondering van $T_0$ en $T_1$, worden aangestuurd door een AND-functie van alle voorafgaande $Q$-uitgangen. Het principe van het toggelen is dus eenvoudig. Een nadeel is dat er steeds groter wordende AND-poorten moeten worden gebruikt. We kunnen de functie echter herschrijven als:

$$\begin{aligned} T_0^n &= 1 \\ T_1^n &= Q_0^n \\ T_2^n &= Q_1^n \cdot Q_0^n \\ T_3^n &= Q_2^n \cdot (Q_1^n \cdot Q_0^n) \\ &\vdots \\ T_k^n &= Q_{k-1}^n \cdot (Q_{k-2}^n \cdots (Q_2^n \cdot (Q_1^n \cdot Q_0^n))) \end{aligned} \tag{8.3}$$

waardoor de functies zijn op te bouwen met AND-poorten met twee ingangen. Het principeschema voor vier bits is te zien in figuur 8.17.

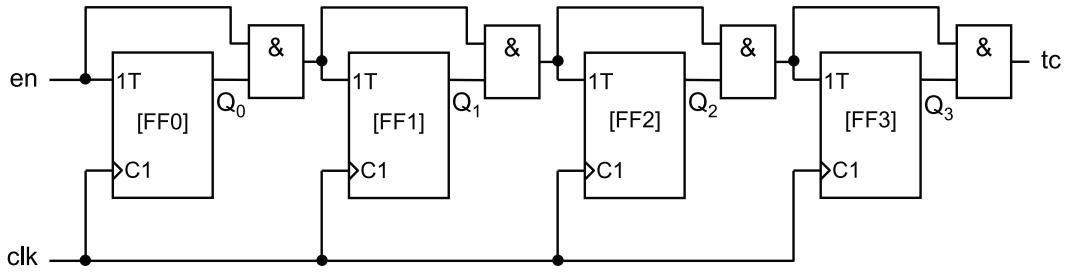



**Figuur 8.17:** *Schema van een 4-bits teller op basis van T-flipflops.*

We breiden de teller uit met een *enable* (`en`) en een *terminal count* (`tc`). Als `en` logisch 1 is, telt de teller op een klokflank. Als de teller in de hoogste telstand staat en de enable is actief dan is `tc` logisch 1, anders is `tc` logisch 0. Het schema is te zien in figuur 8.18.

De functies van de $T$-ingangen zijn:

$$\begin{aligned} T_0^n &= en^n \\ T_1^n &= Q_0^n \cdot en^n \\ T_2^n &= Q_1^n \cdot (Q_0^n \cdot en^n) \\ T_3^n &= Q_2^n \cdot (Q_1^n \cdot (Q_0^n \cdot en^n)) \end{aligned} \tag{8.4}$$




**Figuur 8.18:** *Schema van een 4-bits teller met enable en terminal count.*

De functie van de terminal count is:

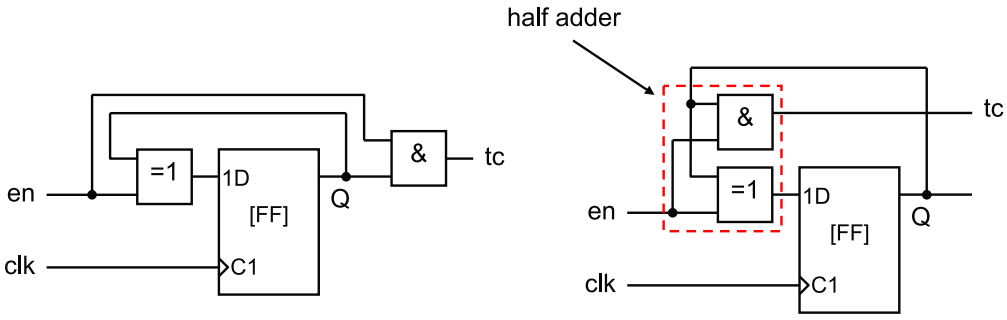
$$\begin{aligned} tc^n &= Q_3^n \cdot (Q_2^n \cdot (Q_1^n \cdot (Q_0^n \cdot en^n))) \\ &= Q_3^n \cdot Q_2^n \cdot Q_1^n \cdot Q_0^n \cdot en^n \end{aligned} \tag{8.5}$$

Te zien is dat de teller een zeer gelijkmatige opbouw heeft. Elke telbit bestaat uit een T-flipflop en een AND-poort. De teller is nu eenvoudig te uit te breiden tot een grotere teller (denk aan structural VHDL).

We kunnen functie (8.1) herschrijven als:

$$\begin{aligned} Q^{n+1} &= Q^n \cdot \overline{T^n} + \overline{Q^n} \cdot T^n \\ &= T^n \oplus Q^n \end{aligned} \tag{8.6}$$

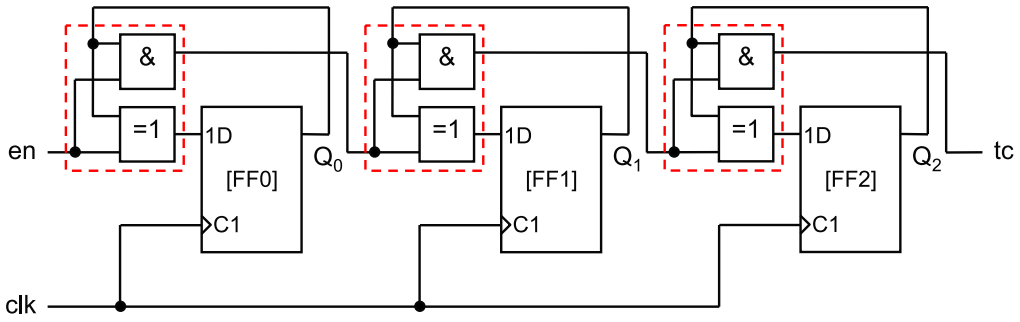
De T-flipflop is dus te realiseren met een D-flipflop en een EXOR-poort. Samen met de AND-poort die nodig is voor het instellen van de $T$-ingangen krijgen we het schema voor één telbit zoals te zien is links in figuur 8.19. Door het schema iets anders te tekenen ontdekken we dat een telbit bestaat uit een D-flipflop en een half adder. Dit is te zien rechts in de figuur.



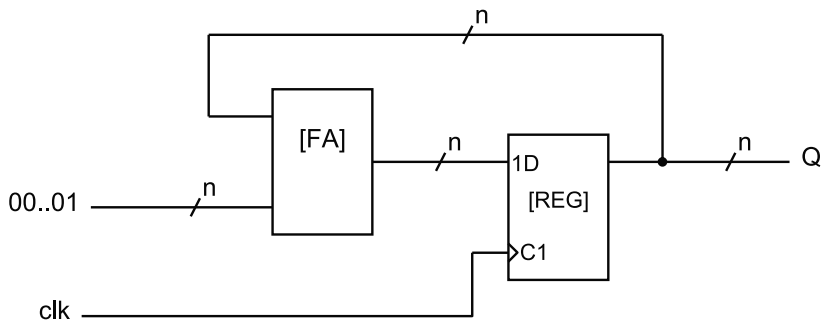

**Figuur 8.19:** *Schema van een telbit op basis van een D-flipflop en een half adder.*

Een teller is dus te realiseren met D-flipflops en half adders. Voor drie bits is dit te zien in figuur 8.20.

In feite is een teller te realiseren met een full adder en een register. De huidige stand van het register (de tellerwaarde) wordt teruggevoerd naar de opteller. Op de andere ingang van de teller wordt de constante 00..01 aangeboden. De nieuwe berekende waarde wordt (op een klokflank) in het register geklokt. Zie figuur 8.21.




**Figuur 8.20:** *Schema van een 3-bits teller op basis van D-flipflops en half adders.*




**Figuur 8.21:** *Schema van een n-bits teller op basis van een register en een full adder.*

### 8.4.2 Ontwerp van een 4-bits teller met structural VHDL

Laten we eens een 4-bits teller beschrijven. De teller is op te bouwen uit vier telsecties voor één bit. Eerst beschrijven we de werking van één telsectie. Daarna instantiëren we vier van deze secties. In listing 8.3 is de code van zo'n telsectie te zien. De T-flipflop is gerealiseerd volgens functie (8.6). Er is een resetfunctionaliteit aangebracht zodat de flipflop in een bekende stand gedwongen kan worden. De flipflop wordt beschreven middels een proces. Er is een intern signaal nodig omdat de waarde van de telbit ook gelezen wordt, d.w.z. dat het aan de rechterkant van het toekenningsteken staat. De terminal count wordt gerealiseerd met een AND-functie. De interne waarde wordt toegekend aan de uitgang `q`. Merk op dat hier dus drie concurrent assignment staan: één proces en twee eenvoudige toekenningen.

We gebruiken de telsectie in een hogere hiërarchie. Dit is te zien in listing 8.4. De entity is rechttoe-rechtaan. De telstand wordt door middel van een 4-bits vector beschikbaar gesteld aan de buitenwereld. In de architecture worden drie interne signalen gedeclareerd: `tc0`, `tc1` en `tc2`. Deze dienen als enable-signaal voor de volgende secties (vergelijkbaar met de carry-structuur in een ripple carry opteller). Daarna wordt de telsectie als component gedeclareerd. In feite is dit de entity-beschrijving van de telsectie. Na de componentdeclaratie volgt de daadwerkelijke instantiëring. Er worden vier telsecties geïnstantieerd door middel van een port map.

Elke instantiëring begint met een verplichte naam (`count0` t/m `count3`). In de *port association list* staat hoe de signalen moeten worden aangesloten. Alle telsecties krijgen hetzelfde klok- en resetsignaal aangeboden. Instantie `count0` is de telsectie voor de

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

entity counter_bit_struct is
    port (clk : in std_logic;
          areset : in std_logic;
          en : in std_logic;
          q : out std_logic;
          tc : out std_logic
         );
end entity counter_bit_struct;

architecture section of counter_bit_struct is
signal q_int : std_logic;
begin

    process (clk, areset) is
    begin
        if areset = '1' then
            q_int <= '0';
        elsif rising_edge(clk) then
            q_int <= en xor q_int;
        end if;
    end process;

    tc <= en and q_int;
    q <= q_int;

end architecture section;
```

**Listing 8.3:** *VHDL-beschrijving van één telsectie.*

eenheden. Deze instantie krijgt het enable-signaal van de hogere hiërarchie. De uitgang `q` wordt gekoppeld aan het minst significante telbit `q(0)` van de hogere hiërarchie. Uitgang `tc` wordt gekoppeld aan intern signaal `tc0`.

Bij de instantiëring van `count1` wordt de enable verbonden met intern signaal `tc0`. Deze telsectie voor de tweetallen zal dus toggelen als `tc0` logisch 1 is. Uitgang `q` is verbonden met `q(1)` van de hogere hiërarchie. De terminal count is verbonden met signaal `tc1`. Voor de overige twee telsecties geldt dezelfde opzet. Te zien is dat de terminal counts een serieketen vormen van `count0` naar `count3`. Zo'n serieketen wordt een *cascadeschakeling* genoemd. De interne signalen `tc0`, `tc1` en `tc2` hebben geen externe functie en zijn alleen bedoeld als signaaldragers. We noemen dit ook wel *glue signals*.

De hierboven gepresenteerde manier van het ontwerpen van een teller heeft een aantal nadelen. Ten eerste beschrijven we de teller op logisch niveau, namelijk een T-flipflop en een AND-poort. Ten tweede is het niet eenvoudig om het telbereik aan te passen. Op deze manier is het telbereik altijd een macht van 2. VHDL kent echter de rekenkundige optelling ('+'). Hiermee zijn eenvoudig optellers te beschrijven. Het voordeel van de rekenkundige optelling is dat de synthesizer slimme en snelle optellers kan realiseren en dat het telbereik eenvoudig is aan te passen.

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

entity counter_struct is
    port (clk : in std_logic;
          areset : in std_logic;
          en : in std_logic;
          q : out std_logic_vector (3 downto 0);
          tc : out std_logic
         );
end entity counter_struct;

architecture structural of counter_struct is
signal tc0, tc1, tc2: std_logic;

component counter_bit_struct is
    port (clk : in std_logic;
          areset : in std_logic;
          en : in std_logic;
          q : out std_logic;
          tc : out std_logic
         );
end component counter_bit_struct;

begin

    count0: counter_bit_struct
    port map (clk => clk, areset => areset, en => en, q => q(0),
              tc => tc0);

    count1: counter_bit_struct
    port map (clk => clk, areset => areset, en => tc0, q => q(1),
              tc => tc1);

    count2: counter_bit_struct
    port map (clk => clk, areset => areset, en => tc1, q => q(2),
              tc => tc2);

    count3: counter_bit_struct
    port map (clk => clk, areset => areset, en => tc2, q => q(3),
              tc => tc);

end architecture structural;
```

**Listing 8.4:** *VHDL-beschrijving van de 4-bits teller met vier telsecties.*

### 8.4.3 Ontwerp van een 4-bits teller met een opteller

We zullen een 4-bits teller ontwerpen op basis van een opteller. In listing 8.5 is de beschrijving van de entity te zien. We gaan bij deze teller naast een asynchrone reset gebruik maken van een synchrone clear. De tellerstand wordt naar buiten gebruik middels een 4-bits vector `q` van het type `unsigned`.

In de architecture, te zien in listing 8.6, wordt zoals gebruikelijk eerst de asynchrone

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;                       -- needed for '+'

entity counter_4bit is
    port (clk     in std_logic;                 -- the clock
          areset  in std_logic;                 -- asynchronous reset
          en      in std_logic;                 -- enable
          clear   in std_logic;                 -- synchronous clear
          q       out unsigned(3 downto 0);     -- counter value
          tc      out std_logic                 -- termininal count
         );
end entity counter_4bit;
```

**Listing 8.5:** *De entity-beschrijving van de 4-bits teller.*

reset beschreven. Onder de klokflankbeschrijving is te zien dat als signaal `clear` logisch 1 is, de teller in de stand $0000_2$ wordt gezet. Dit wordt een synchrone clear genoemd. Dit signaal wordt als eerste getest een heeft dus de hoogste prioriteit. Als `clear` logisch 0 is, wordt gekeken op signaal `en` logisch 1 is. Als dat zo is, wordt de telstand wordt met één verhoogd door de regel `count <= count + 1`. Mocht de telstand op de maximale waarde van $1111_2$ staan dan wordt de telstand $0000_2$. Dit wordt automatisch gerealiseerd omdat `count` gedeclareerd is als een 4-bits vector. Ook deze teller telt dus cyclisch.

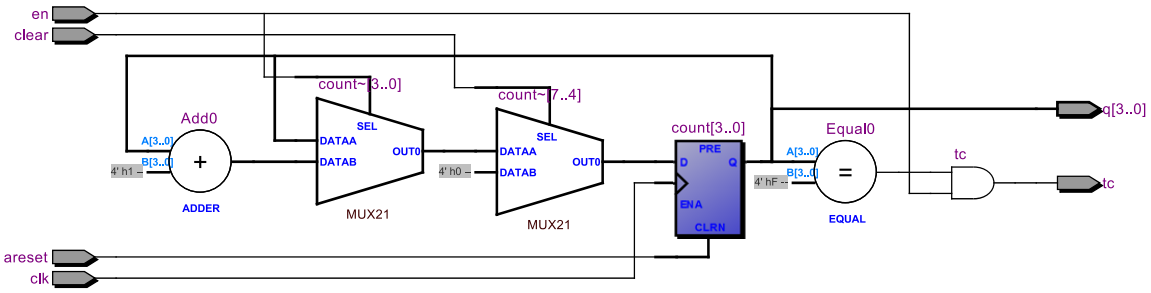
```
architecture rtl of counter_4bit is
signal count : unsigned(3 downto 0);    -- internal counter value
begin

    process (clk, areset) is
    begin
        if areset = '1' then            -- asynchonous reset
            count <= "0000";
        elsif rising_edge(clk) then     -- clock edge
            if clear = '1' then         -- synchronous clear
                count <= "0000";
            elsif en = '1' then         -- enable
                count <= count + 1;
            end if;
        end if;
     end process;

     q <= count;                        -- copy value
     tc <= '1' when count = "1111" and en = '1' else '0';

end architecture rtl;
```

**Listing 8.6:** *De architecture-beschrijving van de 4-bits teller.*

Het voordeel van deze beschrijving is duidelijk. De beschrijving presenteert een duidelijke structuur en er hoeft geen rekening gehouden met het ontwerp op het logisch niveau. We laten het aan de synthesizer over om de teller in hardware om te zetten. Het is eveneens mogelijk om het telbereik eenvoudig aan te passen. Dat zullen we later zien bij de BCD-teller.

Ook hier is weer sprake van drie concurrent signal assignments: een proces, een simpele toekenning en een Conditional Signal Assignment (CSA). Met deze laatste toekenning wordt de terminal count beschreven. Die is logisch 1 als de teller in de hoogste telstand staan én de enable is logisch 1, anders is de terminal count logisch 0. Doordat de enable meegenomen is in de toekenning kunnen makkelijk grotere tellers gerealiseerd worden door `tc` van een telsectie te verbinden met `en` van een andere telsectie.

In figuur 8.22 is het gesynthetiseerde schema te zien. Er worden flipflops gerealiseerd voor de interne telstand `count` en `count` wordt aan `q` toegekend. De optelling wordt gesynthetiseerd met een full adder, de test op de hoogste stand met een comparator. De **`if`**-statements worden gesynthetiseerd met multiplexers.

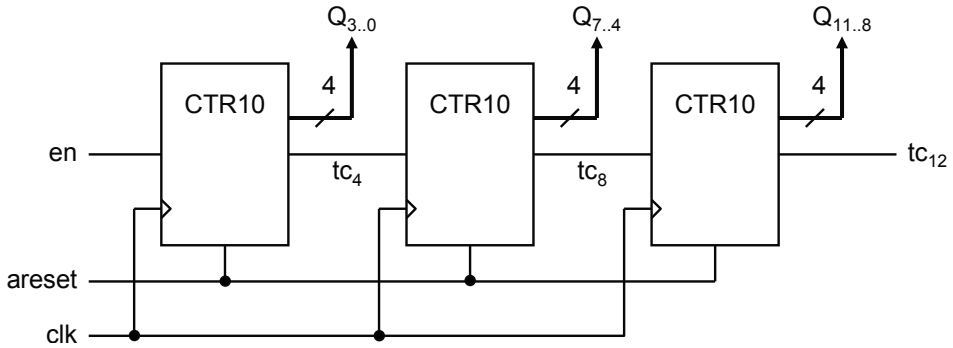



**Figuur 8.22:** *Gesynthetiseerd schema van een 4-bits teller op basis van een register en een opteller.*

### 8.4.4 Ontwerp van een BCD-teller

Tellen in het binaire systeem is erg simpel. Het afbeelden van het binaire getal in decimale vorm echter niet. Het is dan handiger om Binary Coded Decimal-tellers (BCD-tellers) te gebruiken. De telstand is dan eenvoudig af te beelden op bijvoorbeeld 7-segment displays. De telstanden van een BCD-teller lopen van 0 t/m 9. Daarna wordt de telstand weer 0. De telstand 9 is dus de hoogste stand van een BCD-teller en moet uitgecodeerd worden als terminal count zodat grotere gecascadeerde (in serie geschakelde) BCD-tellers te ontwerpen zijn.

In figuur 8.23 is het principeschema van een 3-decaden BCD-teller te zien. De uitgaande terminal counts zijn verbonden met ingaande enables via de signalen $tc_4$ en $tc_8$.




**Figuur 8.23:** *Blokschema van drie gecascadeerde BCD-tellers.*

We laten de code zien van één BCD-telsectie. Deze is te zien in listing 8.7. De opzet is vergelijkbaar met de 4-bits teller in listings 8.5 en 8.6 alleen heeft de BCD-teller geen synchrone clear. Onder de klokflankbeschrijving wordt getest of de enable actief is. Als de tellerstand op 9 staat (de hoogste telstand) dan wordt de tellerstand op 0 gezet, anders wordt de tellerstand met één verhoogd.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;                          -- needed for '+'
4
5  entity counter_bcd is
6      port (clk    : in std_logic;                   -- clock
7            areset : in std_logic;                   -- reset
8            en     : in std_logic;                   -- enable
9            q      : out unsigned(3 downto 0);       -- counter value
10           tc     : out std_logic                   -- terminal count
11          );
12 end counter_bcd;
13
14 architecture rtl of counter_bcd is
15 signal count : unsigned (3 downto 0);
16 begin
17
18     process (clk, areset) is
19     begin
20         if areset = '1' then                       -- asynchronous reset
21             count <= "0000";
22         elsif rising_edge(clk) then
23             if en = '1' then                       -- enable active
24                 if count = "1001" then             -- counter at 9 then
25                     count <= "0000";               -- ... counter set to 0
26                 else
27                     count <= count + 1;            -- otherwise count
28                 end if;
29             end if;
30         end if;
31     end process;
32
33     tc <= '1' when count = "1001" and en = '1' else '0';
34     q <= count;
35
36 end architecture rtl;
```

**Listing 8.7:** *De VHDL-beschrijving van een 1-decade BCD-teller.*

De terminal count wordt logisch 1 als de teller op stand 9 staat én de enable is actief, anders is terminal count logisch 0.

Om een 3-decaden teller uit figuur 8.23 te beschrijven kunnen we gebruik maken van gecascadeerde tellers volgens het terminal count/enable principe, maar met VHDL is een veel krachtigere methode mogelijk: een teller binnen een teller. We doen dat aan de hand van de 3-decaden BCD-teller.

Laten we eerst eens kijken naar een teller voor één BCD-cijfer. Dit is te zien in listing 8.8. De telstand wordt op $0000_{\text{BCD}}$ gezet als de teller op $1001_{\text{BCD}}$ staat. Zo niet, dan wordt de telstand met 1 verhoogd. Normaliter staat er op het punt gemarkeerd met een $\star$ geen code.

```
if rising_edge(clk) then
    if count = "1001" then
        count <= "0000";
        -- *
    else
        count <= count + 1;
    end if;
end if;
```

**Listing 8.8:** *Een 1-decade BCD-teller.*

Het is echter ook mogelijk om op het punt gemarkeerd met een $\star$ de beschrijving van een tweede teller te plaatsen.

In listing 8.9 is een voorbeeld te zien van een drie-decaden BCD-teller. Deze telt van 0000 0000 $0000_{\text{BCD}}$ tot en met 1001 1001 $1001_{\text{BCD}}$. Het principe is eenvoudig. Het bestaat uit drie BCD-tellers, één voor de eenheden, één voor de tientallen en één voor de honderdtallen. Het zijn in feite identieke tellers die in elkaar verweven zitten.

```
if rising_edge(clk) then
    -- Start counter for units (ones)
    if countones = "1001" then
        countones <= "0000";
        -- Start counter for tens
        if counttens = "1001" then
            counttens <= "0000";
            -- Start counter for hundreds
            if counthundreds = "1001" then
                -- no thousands, so do nothing
                null;
            else
                counthundreds <= counthundreds + 1;
            end if;
            -- End counter for hundreds
        else
            counttens <= counttens + 1;
        end if;
        -- End counter for tens
    else
        countones <= countones + 1;
    end if;
    -- End counter for units
end if;
```

**Listing 8.9:** *Voorbeeld van een 3-decaden BCD-teller.*

Er wordt eerst getest of de eenheden op telstand $1001_{\text{BCD}}$ staan (uiteraard maakt het niet uit of we binair of BCD testen, het bitpatroon is identiek). Zo ja, dan zetten we de

eenheden op $0000_{BCD}$ en starten we de test voor de tientallen. Staan de tientallen op $1001_{BCD}$, dan zetten we die weer op $0000_{BCD}$ en worden de honderdtallen bekeken. De tellers worden met één verhoogd als de telstand niet op $1001_{BCD}$ staat.

Deze beschrijving is eenvoudig uit te breiden naar grotere BCD-tellers. Als nadeel kan worden genoemd dat de gehele teller moet worden gesimuleerd, er kan niet volstaan worden met het (eerst) testen van één decade.

### 8.4.5 Tellers met integers

Het gebruik van integers sluit heel natuurlijk aan bij tellen. In VHDL zijn integers standaard 32 bits breed (denk aan synthese!). Door het gebruik van de **range**-clausule kan heel makkelijk een bepaald telbereik gerealiseerd worden. Zie listing 8.10

```
constant n : integer := 8;
signal count_i : integer range 0 to 2**n-1;
```

**Listing 8.10:** *Voorbeeld van een integer met bereik.*

Hierin staat ** voor machtsverheffen. Conversie tussen unsigned en integer gebeurt door middel van conversieroutines die in VHDL beschikbaar zijn. De lengte (het aantal bits) van een unsigned kan worden opgevraagd door de `length`-attribute. Zie listing 8.11.

```
constant n : integer := 8;
signal count_i : integer range 0 to 2**n-1;  -- 2**n-1 == 255
signal count_u : unsigned(n-1 downto 0);

count_i <= to_integer(count_u);
count_u <= to_unsigned(count_i, count_u'length);
-- or: count_u <= to_unsigned(count_i, n)
```

**Listing 8.11:** *Voorbeeld van een integer met bereik.*

Door het gebruik van integers kan heel makkelijk een teller worden beschreven met een bepaald telbereik middels de **range**-clausule. Het testen op de maximale waarde moet expliciet beschreven worden. Dit is te zien in listing 8.12.

```
signal count_val : integer range 0 to 199;

if count_val = 199 then   -- or count_val = count_val'high
    count_val <= 0;
else
    count_val <= count_val + 1;
end if;
```

**Listing 8.12:** *VHDL-beschrijving van een teller met integers.*

Bij synthese wordt het minimale aantal telbits uitgerekend, volgens de bekende formule $n = \lceil {}^2\log(199+1) \rceil$. Als de teller buiten het bereik komt, volgt (uiteraard) geen foutmelding; de teller telt gewoon door tot alle telbits op 1 staan. Bij simulatie volgt een foutmelding als de telstand buiten het bereik komt, ook als het telbereik exact een macht

van 2 is. De simulatie stopt dan. Let erop dat bij synthese altijd het getal 0 wordt meegenomen, ook als deze niet in het bereik ligt.

In listing 8.13 is de volledige beschrijving van de teller te zien. In de entity worden alleen maar de types `std_logic` en `unsigned` gebruikt. Het type `integer` komt niet voor. Dat komt omdat een aantal softwaretools daar niet mee overweg kan. Dat zou betekenen dat de code niet te porteren is tussen tools. In de beschrijving hebben we een mogelijkheid

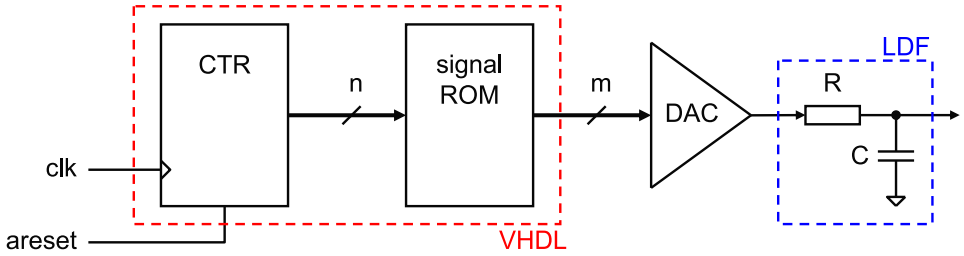
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5     entity counter_integer is
6        generic (n : positive := 8);
7        port (clk    : in std_logic;
8              areset : in std_logic;
9              load   : in std_logic;
10             data   : in unsigned(n-1 downto 0);
11             q      : out unsigned(n-1 downto 0);
12             tc     : out std_logic);
13    end entity;
14
15    architecture rtl of counter_integer is
16    -- Range is 0 to 2 to-the-power-of-n minus 1
17    signal count_val : integer range 0 to 2**n-1;
18    begin
19       process (clk, areset) is
20       begin
21          if areset = '1' then
22             count_val <= 0;
23          elsif rising_edge(clk) then
24             if load = '1' then
25                -- conversion from unsigned to integer
26                count_val <= to_integer(data);
27             else
28                if count_val = 2**n-1 then    -- explicit test!
29                   count_val <= '0';
30                else
31                   count_val <= count_val + 1;
32                end if;
33             end if;
34          end if;
35       end process;
36
37    -- Conversion from integer to unsigned
38    -- Note the use of the length attribute
39    q <= to_unsigned(count_val, q'length);
40
41    -- Concurrent statement for terminal count
42    tc <= '1' when count_val = 2**n-1 else '0';
43
44 end architecture rtl;
```

**Listing 8.13:** *VHDL-beschrijving van een teller met integers.*

beschreven om de tellerstand te laden met een externe waarde. Als signaal `load` logisch 1 is, laadt de teller de externe stand in. Omdat we intern met integers werken, moet de externe data met behulp van een conversieroutine omgezet worden van een integer naar een unsigned. Merk op dat we expliciet testen op de maximale telstand.

## 8.5 Voorbeeld: digitale signaalvormgenerator

Een mooi voorbeeld waarin een teller wordt gebruikt, is een digitale signaalvormgenerator. Het principeschema van de schakeling is te zien in figuur 8.24. Het te genereren signaal is sinusvormig (als voorbeeld). De teller telt cyclisch en begint weer bij 0 nadat de hoogste telstand is bereikt. De teller stuurt een ROM-tabel aan. De tabel bevat een gecodeerde sinus. Een ROM is in feite een elektronische waarheidstabel. De ROM-tabel stuurt een digitaal-analoog converter (DAC) aan. Deze genereert het analoge signaal. Aan de uitgang van de digitaal-analoog converter is een laagdoorlaatfilter (LDF) gekoppeld die het signaal reconstrueert. We beschrijven alleen de teller en de signaal-ROM.




**Figuur 8.24:** *Blokschema van de digitale signaalvormgenerator.*

We hebben in de figuur gekozen voor een algemene opzet. De teller heeft in de figuur $n$ telbits. De telstand loopt van 0 t/m $2^n-1$. De signal ROM heeft $n$ *adresbits* die verbonden zijn met de uitgangen van de teller. Zo selecteert de teller op enig moment een adres in de ROM. De ROM heeft $m$ *databits* dus op een adres kan een van $2^m$ verschillende waarden geplaatst worden.

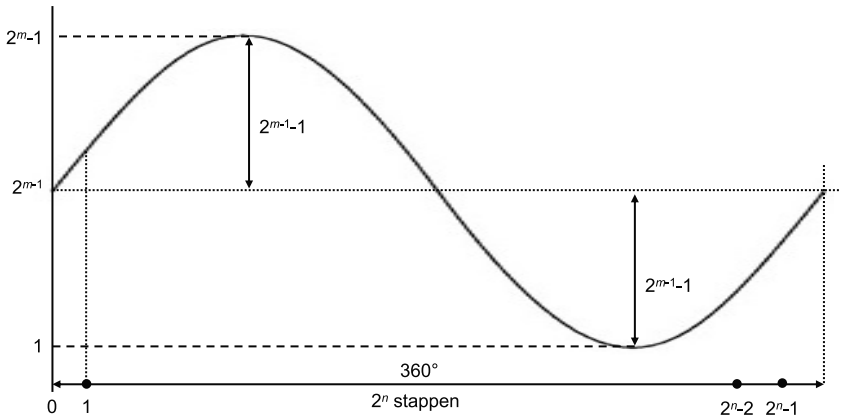
We moeten de ROM vullen met de gecodeerde sinuswaarden. Dat kan met de volgende formule:

$$\text{tabelwaarde}_i = \sin\left(i \cdot \frac{360^\circ}{2^n}\right) \cdot (2^{m-1}-1) + 2^{m-1} \qquad \text{(voor } 0 \le i \le 2^n-1\text{)} \qquad (8.7)$$

In figuur 8.25 zijn de relaties tussen de diverse parameters te zien. In één periode van de sinus worden $2^n$ waarden berekend. Elke sinuswaarde wordt geschaald naar helft-1 van het bereik $(2^{m-1}-1)$ met een offset van de helft van het bereik $(2^{m-1})$. De minimale waarde is 1 en de maximale waarde is $2^m-1$. Uiteraard moeten de getallen worden afgerond naar het dichtsbijzijnde gehele getal.

We zullen dit demonstreren aan de hand van een voorbeeld. We ontwerpen de signaalvormgenerator voor $n = m = 4$. Er zijn dus 16 stappen en de sinus wordt gecodeerd met 16 verschillende waarden. Met behulp van formule (8.8) kunnen we de waarden uitrekenen. Deze zijn te zien in tabel 8.2.

$$\text{tabelwaarde}_i\Big|_{n=m=4} = 7 \cdot \sin(i \cdot 22{,}5^\circ) + 8 \qquad \text{(voor } 0 \le i \le 15\text{)} \qquad (8.8)$$




**Figuur 8.25:** *Relaties tussen een periode van een sinusvormig signaal en bitbreedtes.*

**Tabel 8.2:** *Berekende sinuswaarden van de signaalvormgenerator.*

| i | hoek | sinus | i | hoek | sinus |
|---|---|---|---|---|---|
| 0 | 0,0 | 1000 | 8 | 180,0 | 1000 |
| 1 | 22,5 | 1011 | 9 | 202,5 | 0101 |
| 2 | 45,0 | 1101 | 10 | 225,0 | 0011 |
| 3 | 67,5 | 1110 | 11 | 247,5 | 0010 |
| 4 | 90,0 | 1111 | 12 | 270,0 | 0001 |
| 5 | 112,5 | 1110 | 13 | 292,5 | 0010 |
| 6 | 135,0 | 1101 | 14 | 315,0 | 0011 |
| 7 | 157,5 | 1011 | 15 | 337,5 | 0101 |

De code van de signaalvormgenerator is te zien in listing 8.14. De opzet is rechttoe-rechtaan. Er zijn twee ingangen voor de klok en de reset en er is één uitgang voor het gegenereerde signaal. Dat is een 4-bits vector. De teller is eenvoudig van aard. Het is een 4-bits teller die continu cyclisch telt. Er is geen enable in verwerkt. De teller levert een 4-bits telstand aan de ROM-tabel. De ROM-tabel is opgebouwd rond een **`case`**-statement en levert de signaalwaarde voor de uitgang.

In figuur 8.26 is de uitvoer van de simulatie van één periode te zien. Naast de binaire waarden van het sinussignaal is ook de decimale waarde en een analoge versie te zien. Duidelijk is te zien dat signaal `sinq` de vorm van een sinus volgt.

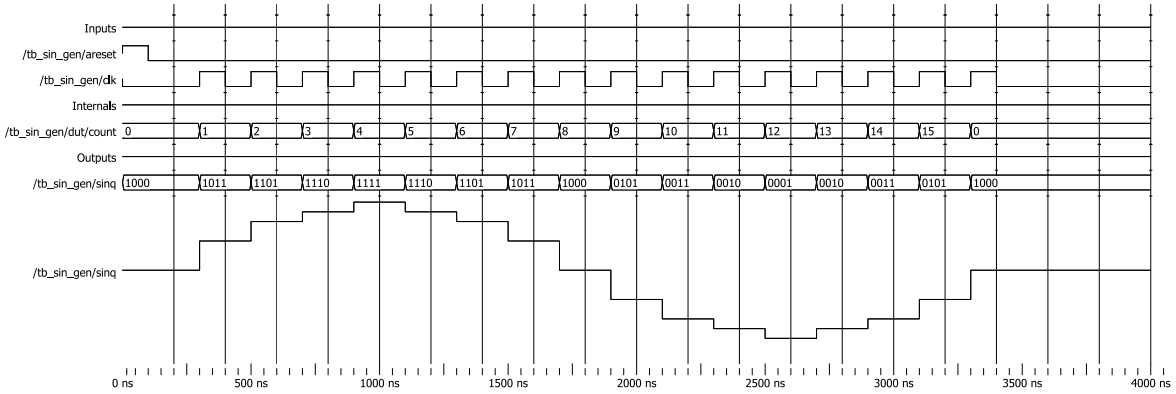
Het systeem is in de praktijk gerealiseerd met een klokfrequentie van 16 kHz zodat het sinusvormige signaal een frequentie van 1 kHz heeft. Een oscilloscoopbeeld is te zien in figuur 8.27. In de figuur zijn twee signalen te zien. Het geblokte signaal is het signaal direct uit de ADC, dus voor het laagdoorlaatfilter. Het signaal heeft al de vorm van een sinus. Het andere signaal komt van de uitgang van het filter en is duidelijk een sinusvormig signaal. Merk op dat de instellingen van de y-as verschillend zijn. De verzwakking van het uitgangssignaal en de fasedraaiing tussen de twee signalen worden veroorzaakt door het filter. Overigens is het tweede signaal ook enigszins geblokt, maar dat komt door de werking van de oscilloscoop. Dit is een *digitizing* oscilloscoop.

De steile flanken in het niet-gefilterde signaal zorgen voor hogere harmonischen in het spectrum van het signaal. Dit spectrum is te zien in figuur 8.28(a). Te zien is dat de

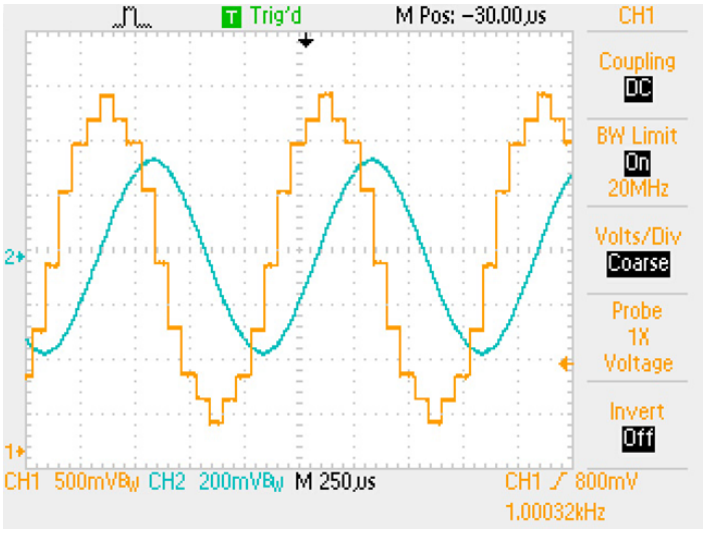
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sin_gen is
    port (clk   : in std_logic;
          areset : in std_logic;
          sinq  : out std_logic_vector(3 downto 0)
         );
end sin_gen;

architecture rtl of sin_gen is
signal count : unsigned(3 downto 0);
begin

    cntr: process (clk, areset) is
    begin
        if areset = '1' then
            count <= "0000";
        elsif rising_edge(clk) then
            count <= count + 1;
        end if;
    end process;

    sintbl: process (count) is
    begin
        case count is
            when "0000" => sinq <= "1000";
            when "0001" => sinq <= "1011";
            when "0010" => sinq <= "1101";
            when "0011" => sinq <= "1110";
            when "0100" => sinq <= "1111";
            when "0101" => sinq <= "1110";
            when "0110" => sinq <= "1101";
            when "0111" => sinq <= "1011";
            when "1000" => sinq <= "1000";
            when "1001" => sinq <= "0101";
            when "1010" => sinq <= "0011";
            when "1011" => sinq <= "0010";
            when "1100" => sinq <= "0001";
            when "1101" => sinq <= "0010";
            when "1110" => sinq <= "0011";
            when "1111" => sinq <= "0101";
            when others => sinq <= "XXXX";
        end case;
    end process;

end rtl;
```

**Listing 8.14:** *Volledige VHDL-beschrijving van de signaalvormgenerator.*

frequentie van 1 kHz links prominent aanwezig is. We spreken van een *paal* op 1 kHz. Maar er zijn ook palen te zien op 15 kHz, 17 kHz, 31 kHz, 33 kHz etc. Het midden van 15 kHz en 17 kHz is 16 kHz, de frequentie van de klok. Er is geen paal te zien op 16 kHz




**Figuur 8.26:** *Uitvoer van de simulatie van de signaalvormgenerator.*

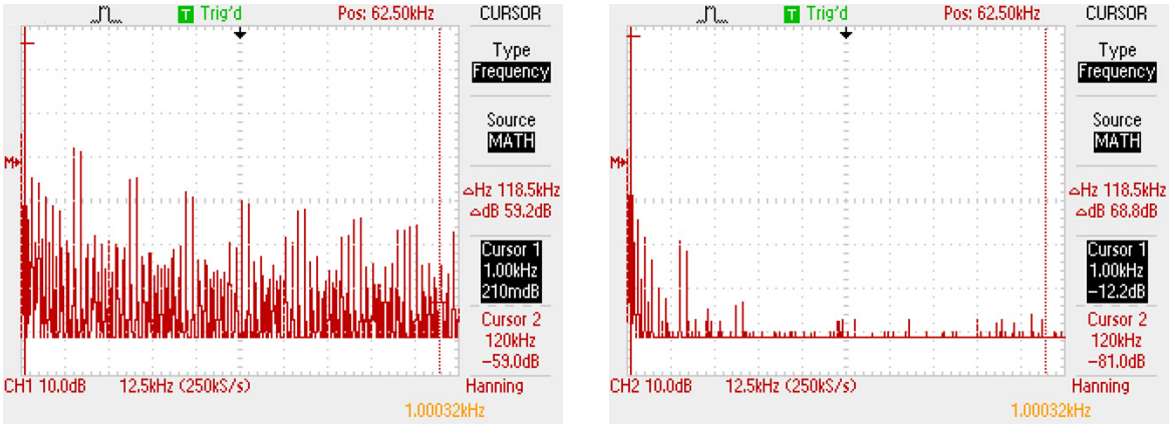



**Figuur 8.27:** *Oscilloscoopbeeld van de signalen.*

maar wel twee *zijbanden*. Die zijbanden zijn ook te zien rond een veelvoud van 16 kHz, dus bij 32 kHz, 48 kHz etc.

In figuur 8.28(b) is het spectrum van het gefilterde signaal te zien. Idealiter komt hierin alleen de frequentie van 1 kHz voor. Te zien is dat er nog palen op 15 kHz en 17 kHz staan, maar de rest is goed weggefilterd. Een beter filter kan uitkomst bieden.

Het is ook mogelijk om de signaalvormgenerator met behulp van integers te beschrijven. Deze code is te zien in listing 8.15. Alleen de architecture is gegeven. We definiëren een nieuw type `sin_array_type` als een vector van 16 elementen van integers. Vervolgens vullen we de vector met de gecodeerde sinuswaarden. De teller telt cyclisch van 0 t/m 15. De waarde van `sinq` komt van één van de elementen van de vector, aangegeven door de waarde van de teller. Let op de conversie tussen de tabelwaarden en het type van `sinq`. Er zijn twee conversieroutines nodig voor de toekenning. Eerst moet de integerwaarde uit de tabel omgezet worden naar een `unsigned`. Daarna moet de `unsigned` omgezet worden naar een `std_logic_vector`. Voor wat betreft synthese maakt het geen verschil welke beschrijving gebruikt wordt. Beide beschrijvingen leveren dezelfde schakeling op.




**(a)** *Spectrum voor filtering.*

**(b)** *Spectrum na filtering.*

**Figuur 8.28:** *Spectrum van het uitgangssignaal voor en na het laagdoorlaatfilter.*

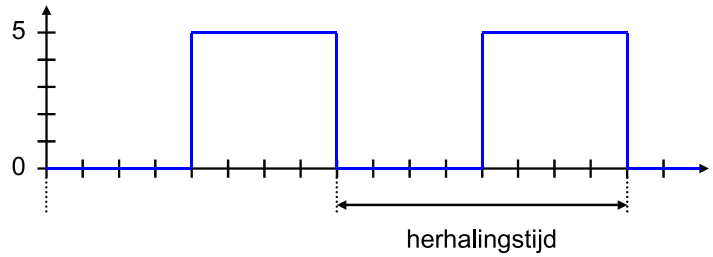
```
architecture rtl_integers of sin_gen is
-- Create array with the sine wave tabulated
type sin_array_type is array (0 to 15) of integer;
constant sin_array : sin_array_type :=
                            (8,11,13,14,15,14,13,11,8,5,3,2,1,2,3,5);
-- Signal count is of type integer
signal count : integer range 0 to 15;
begin
    cntr: process (clk, areset) is
    begin
        if areset = '1' then
            count <= 0;
        elsif rising_edge(clk) then
            if count = 15 then
                count <= 0;
            else
                count <= count + 1;
            end if;
    end if;
    end process;

    sinq <= std_logic_vector(to_unsigned(sin_array(count),4));
end architecture rtl_integers;
```

**Listing 8.15:** *Architecture van de signaalvormgenerator met integers.*

## 8.6 Voorbeeld: digitale PWM-generator

Het regelen van het toerental van een elektromotor kan eenvoudig worden gerealiseerd door gebruik te maken van *Pulse Width Modulation* (PWM). Een elektromotor heeft een zekere massatraagheid. De motor reageert met een zekere traagheid als het stuursignaal (spanning of stroom) sterk verandert. Daar maakt PWM gebruik van. Door de motor met een herhalende puls (pulstrein) aan te sturen zal deze sneller of langzamer willen draaien en na verloop van tijd heeft de motor zijn nieuwe toerental bereikt. De pulstrein wordt door de motor uitgemiddeld.

De puls heeft een zekere maximum en minimum en kan gezien worden als een binair signaal. In de digitale techniek kan dit bijvoorbeeld tussen 0 V en 5 V zijn. Het minimum kan ook negatief zijn. In figuur 8.29 is een PWM-signaal te zien. De *herhalingstijd* van de puls is de tijdsduur tussen het begin en eind van de pulscyclus. Deze herhalingstijd moet veel kleiner zijn dan de reactiesnelheid van de motor.



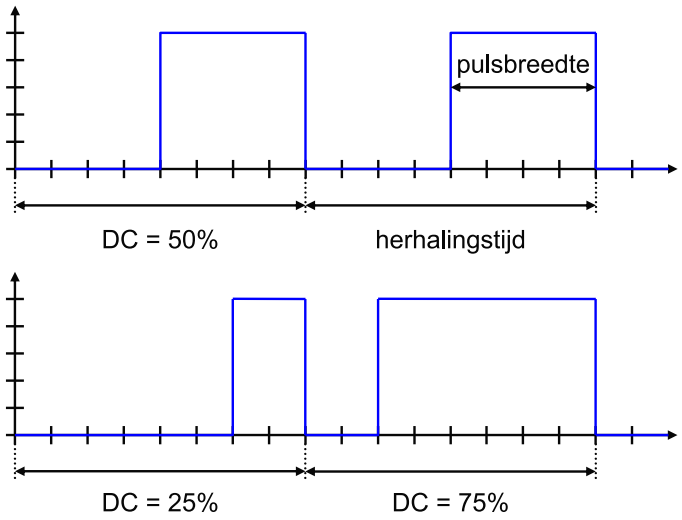

**Figuur 8.29:** *Voorbeeld van een PWM-signaal.*

De inverse van de herhalingstijd is de PWM-schakelfrequentie. Voor een elektromotor ligt de schakelfrequentie tussen enkele kilohertz (kHz) en enkele tientallen kilohertz. Voor een dimmer ligt dit op 100 Hz. Voor een oven ligt dit rond 30 mHz (5x per minuut). Recente ontwikkelingen zijn klasse-D audioversterkers [18].

De *pulsduur* of *pulsbreedte* van de puls is de tijdsduur dat de puls "aan"[4] is. De pulsbreedte ligt tussen 0 en de herhalingstijd. De *Duty Cycle* (DC) is de verhouding tussen de pulsbreedte en de herhalingstijd en wordt uitgedrukt in procenten:

$$DC = \frac{\text{pulsduur}}{\text{herhalingstijd}} \times 100\% \tag{8.9}$$

In figuur 8.30 is een aantal PWM-signalen te zien met verschillende Duty Cylcles.




**Figuur 8.30:** *Voorbeelden van Duty Cycle van een PWM-signaal.*

[4] Het hangt af van de definitie wat "aan" is. Hier is gekozen voor een spanning groter dan 0.

De gemiddelde waarde van een PWM-signaal is:

$$\begin{aligned} U_{gem} &= DC \cdot (U_{max} - U_{min}) + U_{min} \\ &= DC \cdot U_{max} + (1 - DC) \cdot U_{min} \end{aligned} \tag{8.10}$$

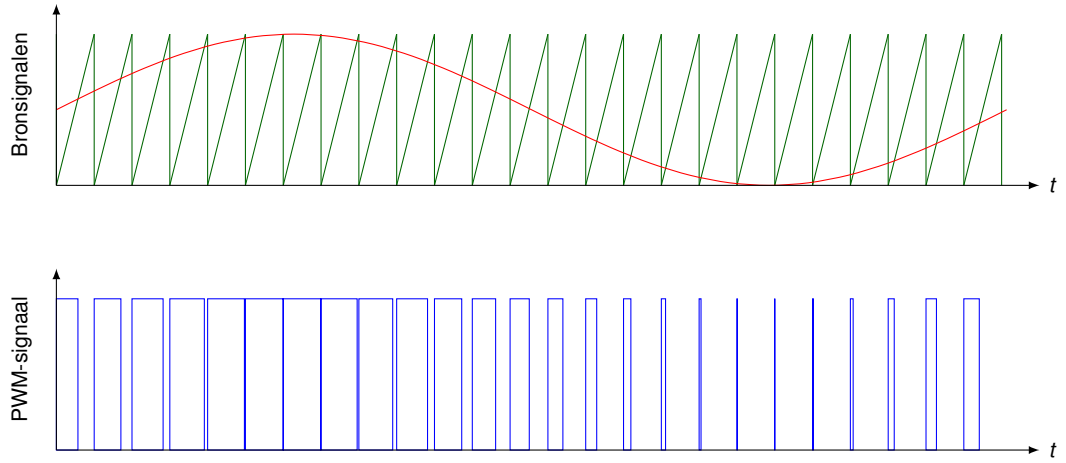
Bij $U_{min} = 0$ V wordt dit:

$$U_{gem} = DC \cdot U_{max} \tag{8.11}$$

**Voorbeelden:**

$U_{PWM}$ is 0 V – 5 V, DC = 50%: $U_{gem} = 2{,}5$ V
$U_{PWM}$ is 0 V – 12 V, DC = 25%: $U_{gem} = 3{,}0$ V

In figuur 8.31 is een voorstelling van een PWM-signaal te zien. De eenvoudigste manier om PWM-signalen te genereren is met behulp van een zaagtandsignaal (boven in de figuur) en een vergelijkschakeling (comparator). Als de waarde van het referentiesignaal (de sinus boven in de figuur) groter is dan van het zaagtandsignaal dan is het PWM-signaal (onder in de figuur) maximaal.

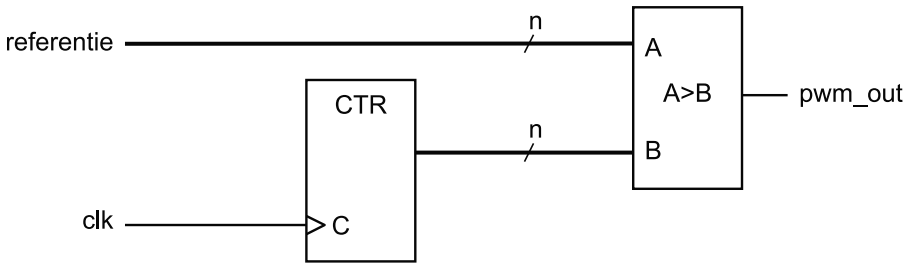



**Figuur 8.31:** *Voorbeeld van het genereren van een PWM-signaal.*

Deze techniek kan eenvoudig worden uitgewerkt in een digitale schakeling. De signalen worden gedigitaliseerd (omgezet naar unsigned getallen). Het zaagtandsignaal wordt gemaakt door een teller cyclisch te laten tellen van 0 tot een maximum. Het referentiesignaal is gewoon een binair getal. Beide worden aangeboden aan een vergelijkschakeling. In figuur 8.32 het principeschema van PWM-generatie te zien. Als de referentie groter is dan de tellerwaarde, is de PWM-uitgang hoog.

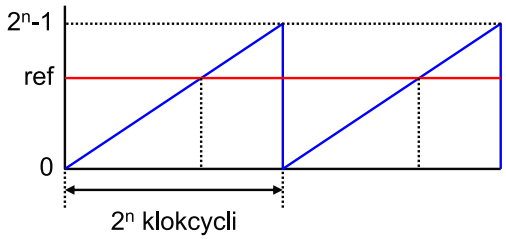
We kunnen wat algemene opmerkingen maken over het PWM-systeem. Dit wordt ondersteund met figuur 8.33.

- Eén PWM-cyclus duurt $2^n$ klokpulsen.
- De waarden van de teller en de referentiewaarde liggen tussen 0 en $2^n - 1$.
- De schakelfrequentie is $f_{pwm} = \dfrac{f_{clk}}{2^n}$. Hierin is $f_{clk}$ de frequentie van de systeemklok.




**Figuur 8.32:** *Principeschema PWM-signaalgeneratie.*

- De Duty Cycle is $DC = \frac{ref}{2^n} \times 100\%$.
- De Duty Cycle is maximaal $\frac{2^n - 1}{2^n} \times 100\%$.



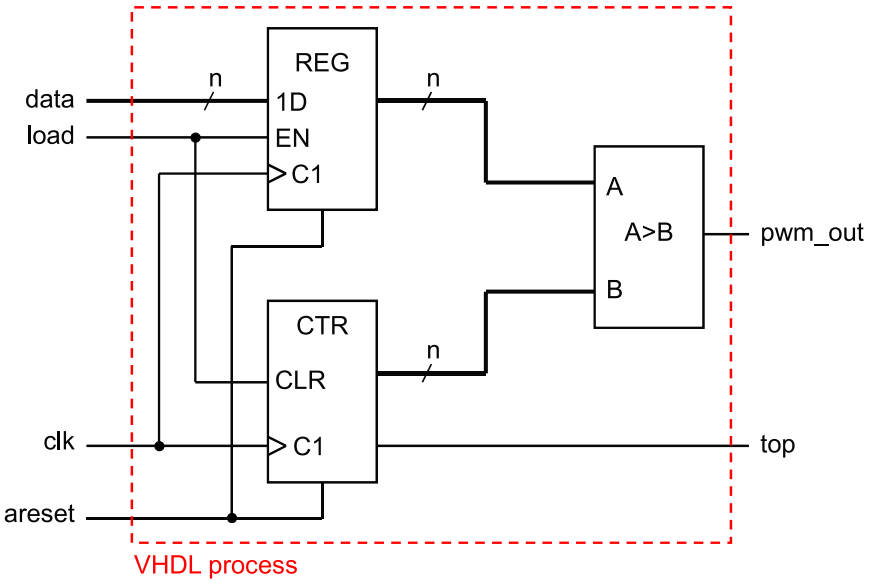

**Figuur 8.33:** *Relatie tussen de diverse PWM-parameters.*

We zullen nu een PWM-schakeling beschrijven voor een 8-bits systeem. De functionele beschrijving van de PWM-schakeling is als volgt:

- Het systeem moet een nieuw referentiegetal kunnen laden. De PWM-generatie wordt dan opnieuw gestart (de cyclus start opnieuw).
- De teller telt van 0 t/m 255 en begint dan weer op 0.
- Op enig moment is de PWM-uitgang hoog als de waarde van het referentiegetal groter is dan de waarde van de teller.
- Als de teller op de maximale waarde staat, moet een puls worden afgegeven (handig voor synchronisatie met externe componenten).
- Het systeem heeft een asynchrone reset.

Natuurlijk kan de beschrijving helemaal structural worden opgesteld met op de onderste laag de gedragsbeschrijvingen voor de diverse componenten. We hebben echter gekozen voor een gedragsbeschrijving van het gehele systeem. Het toont goed de kracht van VHDL. In figuur 8.34 is het blokschema van het PWM-systeem te zien. Er is een register dat de ingebracht referentiewaarde opslaat en een teller die het zaagtandsignaal nabootst. Er is een vergelijkschakeling die aangeeft of de referentiewaarde groter is dan de tellerwaarde. Alles wordt in één VHDL-proces beschreven.

Op signaal `data` wordt de referentiewaarde aangeboden. Als signaal `load` logisch 1 is, wordt de nieuwe waarde geladen. Het signaal `pwm_out` is logisch 1 als de referen-




**Figuur 8.34:** *Blokschema van de PWM-generator.*

tiewaarde groter is dan de tellerwaarde. Signaal `top` is logisch 1 als de teller op de maximale telstand staat, in dit geval dus op 255.

De code is te vinden in listing 8.16. Zowel het register als de teller krijgen nieuwe waarden onder besturing van de asynchrone reset en de opgaande klokflank. Bij een reset worden beide signalen op 0 gezet. Als tijdens een klokflank signaal `load` logisch 1 is, wordt het register `reg` geladen met de externe waarde op signaal `data`. De teller wordt dan op 0 gezet zodat er een nieuwe PWM-cyclus wordt begonnen. Als `load` logisch 0 is, wordt de teller met één verhoogd. Merk op dat de teller cyclisch telt van 0 t/m 255.

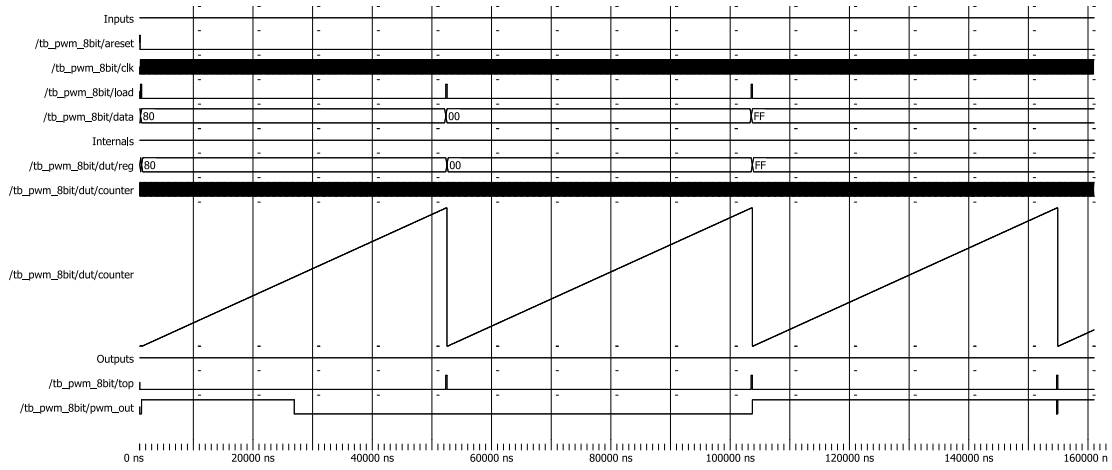
De toekenning aan de signalen `top` en `pwm_out` zijn in het proces verwerkt door middel van een **`if`**-statement. Merk op dat `counter` en `reg` in de sensitivity list staan omdat het proces gestart moet worden (denk aan simulatie) als de waarden van de signalen veranderen.

In figuur 8.35 is de simulatie van drie PWM-cycli te zien, telkens met een andere referentiewaarde. De tellerwaarde is ook “analoog” afgebeeld zodat de vorm van het zaagtandsignaal zichtbaar wordt. In de eerste cyclus is de referentie waarde 128 ($80_{16}$). Het PWM-signaal is dan gedurende een halve cyclus logisch 1. In de tweede cyclus is de referentiewaarde 0 zodat het PWM-signaal de gehele cyclus logisch 0 is. In de laatste cyclus is de referentiewaarde maximaal namelijk 255 ($FF_{16}$). Hier is goed te zien dat het PWM-signaal bijna de gehele cyclus logisch 1 is. Alleen op de maximale telstand is het PWM-signaal even logisch 0. Een Duty Cycle van 100% is dus niet te realiseren.

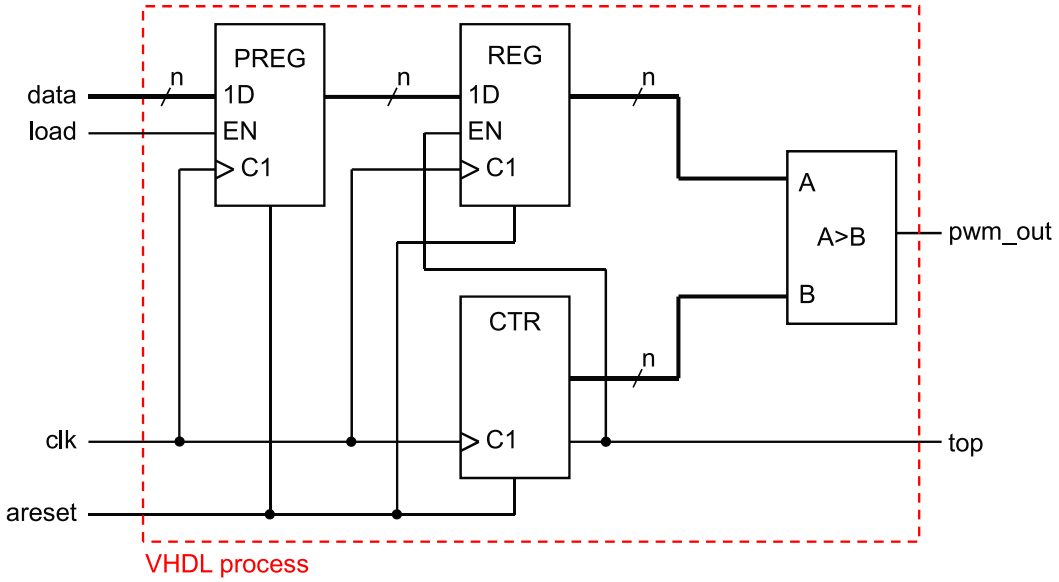
Een nadeel van de hierboven beschreven PWM-generator is dat de PWM-cyclus wordt afgebroken als de referentiewaarde wordt geladen. Veel PWM-systemen gebruiken daarom een *preload-register* waar een nieuwe waarde even tijdelijk wordt opgeslagen zolang de PWM-cyctus nog bezig is. Een nieuwe waarde wordt eerst in het preload-register opgeslagen. Als de PWM-cylcus aan het einde is, wordt de nieuwe waarde van het preload-register naar het referentieregister gekopieerd. Zie figuur 8.36.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;                -- for '+' and '>'

entity pwm_8bit is
    port (clk     : in std_logic;
          areset  : in std_logic;
          load    : in std_logic;        -- load new data
          data    : in unsigned (7 downto 0);
          top     : out std_logic;       -- counter at top
          pwm_out : out std_logic        -- PWM output
         );
end pwm_8bit;

architecture rtl of pwm_8bit is
signal counter : unsigned (7 downto 0);  -- the counter
signal reg : unsigned (7 downto 0);      -- the PWM data register
begin

    process (areset, clk, reg, counter) is
    begin
        if (areset = '1') then
            counter <= "00000000";       -- reset the PWM counter
            reg <= "00000000";           -- reset PWM data register
        elsif rising_edge(clk) then      -- on positive edge
            if load = '1' then           -- load active
                reg <= data;             -- load new PWM data
                counter <= "00000000";   -- restart PWM cycle
            else
                counter <= counter + 1;  -- update counter
            end if;
        end if;

        if counter = "11111111" then     -- counter at top
            top <= '1';
        else
            top <= '0';
        end if;

        if reg > counter then            -- PWM data > counter
            pwm_out <= '1';
        else
            pwm_out <= '0';
        end if;
    end process;

end rtl;
```

**Listing 8.16:** *VHDL-beschrijving van een 8-bits PWM-systeem.*




**Figuur 8.35:** *Simulatie van drie PWM-cycli.*

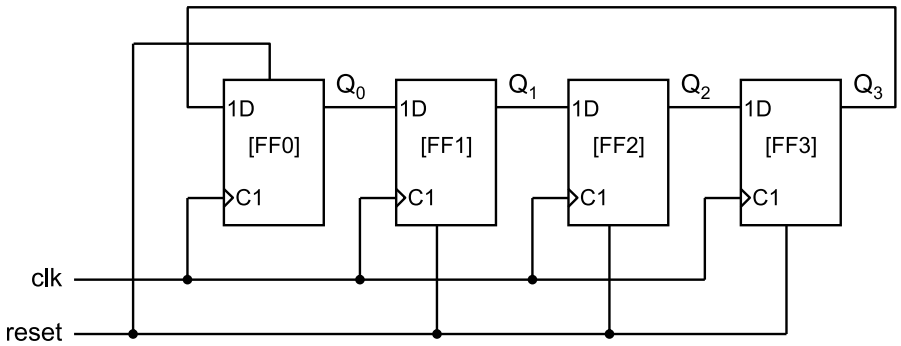



**Figuur 8.36:** *Blokschema van de PWM-generator met preload-register.*

## 8.7 Ringtellers

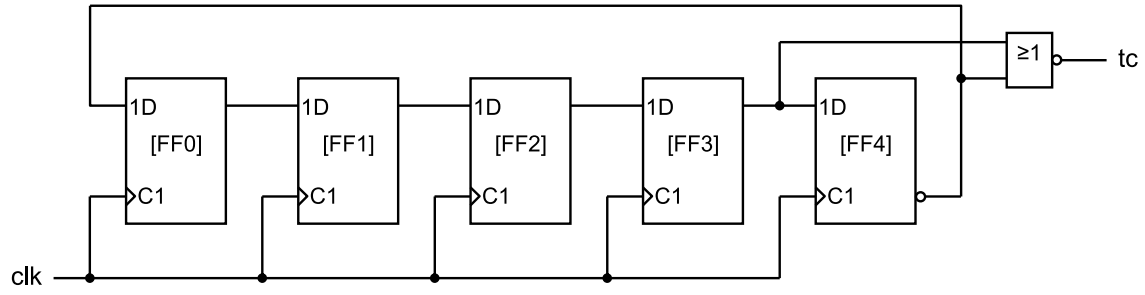
Een speciale klasse van tellers is de *ringteller*. We bespreken twee varianten: de gewone ringteller en de Johnson-teller. Een gewone ringteller is gebaseerd op een schuifregister waarvan de uitgang van de laatste flipflop is gekoppeld aan de ingang van de eerste flipflop. Een principeschema van een 4-bits ringteller is te zien in figuur 8.37.

De asynchrone reset zorgt ervoor dat één 1 in het register wordt geladen, in dit geval in de flipflop aan de linkerkant. De ringteller telt dan de volgende telcyclus: 1000–0100–0010–0001. Aangezien er altijd maar één uitgang logisch 1 is, wordt dit ook wel een one-hot-teller genoemd [19]. Er moet voor gewaakt worden dat de teller niet in een parasitaire telcyclus terecht komt. Dat is een telcyclus anders dan hierboven is beschreven. Dit type ringteller wordt niet veel gebruikt.




**Figuur 8.37:** *Principeschema 4-bits ringteller.*

Bij een Johnson-teller is de inverse uitgangswaarde van de laatste flipflop gekoppeld aan de ingang van de eerste flipflop. Andere benamingen zijn getwiste ringteller en möbuis-teller. Een principeschema is te vinden in figuur 8.38.




**Figuur 8.38:** *Principeschema 5-bits Johnson-teller met terminal count.*

De asynchrone reset is weggelaten, die moet natuurlijk wel geïmplementeerd worden. De 5-bits teller heeft precies 10 telstanden, te weten 00000–10000–11000–11100– 11110–11111–01111–00111–00011–00001. Ook is het mogelijk om en telstand (hier 00001) uit te coderen met één enkele poort. De functie voor de terminal count is:

$$tc^n = Q_4^n \cdot \overline{Q_3^n} = \overline{\overline{Q_4^n} + Q_3^n} \tag{8.12}$$

Een $n$-bits Johnson-teller heeft precies $2n$ telstanden. Johnson-tellers zijn zeer snel want er zit geen logica tussen de flipflops (de signaalvertraging van de inverse stand kan meestaal verwaarloosd worden). Bij een gewone binaire teller moet de nieuwe telstand gegenereerd worden door een opteller of poorten. Er zijn tal van toepassingen van Johnson-tellers. Zo kan een 5-bits Johnson-teller gebruikt worden als decade frequentie-deler (een schakeling die een kloksignaal door 10 deelt) [20]. Johnson-tellers worden gebruikt als meerfasige kloksignaalgeneratoren. Dergelijke generatoren worden gebruikt in grote digitale systemen waarbij timingsignalen nodig zijn met een hoge nauwkeurigheid ten opzichte van de centrale klok [21]. Johnson-tellers produceren storingsvrije symmetrische (50% Duty Cycle) pulsen per telcyclus [22].

In listing 8.17 is een beschrijving van een $n$-bits Johnson-teller te zien. De schakeling wordt gerealiseerd door de code in regel 22. Hierin is duidelijk een schuifregister te herkennen. De terminal count is anders uitgevoerd dan in het schema in figuur 8.38.

Hier wordt gekeken of de twee bits aan de rechterkant gelijk is aan `"01"`. Er is maar één telstand waarbij dit gebeurt, namelijk 00001. Het is dus niet nodig om elke telbit te testen.

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

entity johnson_counter is
    generic (n : integer := 5);
    port (clk :    in std_logic;
          areset : in std_logic;
          count :  out std_logic_vector (0 to n-1);
          tc :     out std_logic
         );
end entity johnson_counter;

architecture rtl of johnson_counter is
signal count_int : std_logic_vector (0 to n-1);
begin

    process (clk, areset) is
    begin
        if areset = '1' then
            count_int <= (others => '0');
        elsif rising_edge(clk) then
            count_int <= not count_int(n-1) & count_int(0 to n-2);
        end if;
    end process;

    count <= count_int;
    tc <= '1' when count_int(n-2 to n-1) = "01" else '0';

end architecture rtl;
```

**Listing 8.17:** *Een VHDL-beschrijving van een n-bits Johnson-teller.*

## 8.8 Opgaven

**8.1.** Ontwerp een VHDL-beschrijving van een 4-bits omhoogteller die ook een begintelstand kan laden.

**8.2.** Ontwerp een 4-bits omhoogteller met JK-flipflops en poorten.

**8.3.** Ontwerp een 4-bits omlaagteller met T-flipflops en poorten. Ga uit van een 4-bits omhoogteller.

**8.4.** Ontwerp een 6-teller, een omhoogteller die telt van 0 t/m 5, met losse poorten en T-flipflops.

**8.5.** Een teller is te ontwerpen met een register en een full-adder. Op één ingang van de full-adder wordt de telstand aangeboden, op de andere ingang wordt de constante 00..01 aangeboden. Implementeer een enable in deze teller zodat de telstand behouden blijft ook al passeert er een klokflank.

**8.6.** Gegeven een deel van de VHDL-beschrijving van de BCD-teller in listing P8.1. Wat doet de teller als deze per ongeluk in stand 12 terecht is gekomen?

```
1   process (clk, areset) is
2   begin
3       if areset = '1' then
4           count <= "0000";
5       elsif rising_edge(clk) then
6           if en = '1' then
7               if count = "1001" then
8                   count <= "0000";
9               else
10                  count <= count + 1;
11              end if;
12          end if;
13      end if;
14  end process;
15
16  tc <= '1' when count = "1001" and en = '1' else '0';
17  q <= count;
```

**Listing P8.1:** *De VHDL-beschrijving van een 1-decade BCD-teller.*

**8.7.** Pas de VHDL-code in listing P8.1 zo aan dat de teller alleen (verder) telt als de teller zich in telstand 0 t/m 9 bevindt. Denk ook aan de terminal count.

**8.8.** Pas de code in listing P8.1 aan zodat de teller een 13-teller wordt.

**8.9.** Voor het detecteren van telstand 9 zijn in principe alleen telbit 3 en 0 nodig ($Q_3$ en $Q_0$). De andere twee zijn don't care. Helaas werkt de constructie

```
when count = "1--1"
```

niet. Pas de code aan zodat voor de test alleen naar de waarden $Q_3$ en $Q_0$ gekeken wordt.

**8.10.** Ontwerp een urenteller in VHDL. De urenteller wordt voor zowel de Amerikaanse als Europese markt ontwikkeld. De urenteller moet aan de volgende eisen voldoen:

- De urenteller heeft een asynchrone reset, actief hoog;
- De urenteller kan tellen in 12-uur-modus of 24-uur-modus middels het signaal `ampm`.
- Als signaal `ampm` logisch '1' is, telt de teller cyclisch van 1 t/m 12;
- Als signaal `ampm` logisch '0' is, telt de teller cyclisch van 0 t/m 23;
- De teller gaat naar de volgende telstand als signaal `up` logisch '1' is;
- Het uur (de tellerwaarde) is beschikbaar als 5-bits unsigned vector.

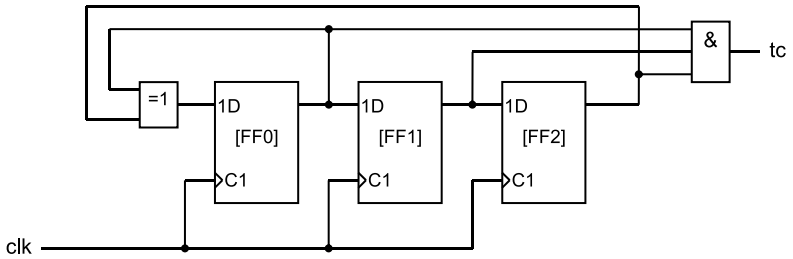
In de onderstaande code zijn de entity met port-beschrijving en de (interne) teller al ingevuld.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity opgave5 is
6      port (clk : in std_logic;                -- The clock
7            areset : in std_logic;             -- The reset
8            up : in std_logic;                 -- Count up
9            ampm : in std_logic;               -- 24 or 12 hour clock
10           hour : out unsigned (4 downto 0)   -- Hour counter
11     );
12 end entity opgave5;
13
14 architecture rtl of opgave5 is
15 signal counter : unsigned(4 downto 0);     -- Internal hour counter
16 begin
17
18     -- Deze code moet ontwikkeld worden
19
20 end architecture rtl;
```

**Listing P8.2:** *De entity-beschrijving van een 12/24-uursklok.*

Geef de VHDL-code van de architecture van deze urenteller.

**8.11.** In figuur P8.1 is een schakeling te zijn. De schakeling wordt een *Linear Feedback Shift Register* genoemd. De schakeling lijkt op een schuifregister maar kan ook als een teller worden beschouwd. De reset is achterwege gelaten maar als werkt als volgt: als de reset geactiveerd is worden de flipflops geladen met een logische 1. De "teller" doorloopt 7 telstanden. Bepaal de telcyclus.




**Figuur P8.1:** *Een 3-bits Linear Feedback Shift Register.*

**8.12.** Bepaal de resolutie van de Duty Cycle voor een $n$-bits digitaal PWM-systeem.

**8.13.** Het PWM-signaal is hoog als de referentiewaarde groter is dan de tellerwaarde (op enig moment). Daardoor is een DC van 100% niet haalbaar. Een student past de schakeling aan zodat het PWM-signaal is hoog als de referentiewaarde groter is dan of gelijk is aan de tellerwaarde (op enig moment). Is dit een slimme keuze? Zijn er andere problemen te verwachten? Motiveer het antwoord.

**8.14.** Bedenk een manier om tot een DC van 100% te komen zonder de vergelijkschakeling aan te passen.

**8.15.** Een ontwerper heeft de onderstaande code geschreven, zie listing P8.3. Het betreft hier een 1-decade BCD-teller. Merk op dat de beschrijving van de terminal count verweven is met de teller. Leg kort uit waarom de terminal count niet correct functioneert.

```
process (clk, areset) is
begin
    if areset = '1' then
        count <= "0000";
    elsif rising_edge(clk) then
        if en = '1' then
            if count = "1001" then
                count <= "0000";
                tc <= '1';
            else
                count <= count + 1;
                tc <= '0';
            end if;
        end if;
    end if;
end process;

q <= count;
```

**Listing P8.3:** *Een deel van de beschrijving van de 1-decade BCD-teller.*

**8.16.** Ontwerp een PWM-generator in VHDL. De PWM-generator moet aan de volgende eisen voldoen:

- De PWM-generator heeft een asynchrone reset, actief hoog.
- De Duty Cycle (DC) kan in stappen van 0,5% worden ingesteld.
- De PWM-generator heeft een referentie-ingang genaamd `ref` waarmee een 8-bits waarde geladen kan worden in het interne referentieregister.
- De referentiewaarde moet steeds aan het begin van een PWM-cyclus worden geladen.
- De PWM-generator heeft een uitgang `pwm_out` die '1' is als de interne referentiewaarde hoger is dat de interne telstand, anders is deze uitgang '0'.
- De PWM-generator heeft een uitgang `top` die '1' is als de interne telstand maximaal is, anders is deze uitgang '0'.
- De interne tellerstand en referentiewaarde moeten als integers gedeclareerd worden.

# 9

# Timing bij dataoverdracht

We hebben ons tot nu toe vooral bezig gehouden met de logische werking van digitale schakelingen. Hier en daar zijn ook de elektrische eigenschappen besproken. Verder is in hoofdstuk 6 de opbouw en timing van latches en flipflops behandeld.

In dit hoofdstuk richten we ons op de timingseigenschappen van digitale schakelingen. We doen dit aan de hand van dataoverdracht tussen flipflops, al dan niet met combinatorische logica tussen de flipflops. We laten zien dat dataoverdracht correct verloopt als aan een aantal timingvoorwaarden wordt voldaan. We bespreken verder het onderwerp klokskew, het vertragen van het kloksignaal, dat invloed heeft op de timing van een digitaal systeem.

Digitale systemen hebben interactie met de buitenwereld. Flipflops krijgen hun data van externe ingangen. Aan de timing van deze ingangen worden eisen gesteld om de dataoverdracht correct te laten verlopen. De uitgangen van een digitaal systeem worden verbonden met andere systemen, bijvoorbeeld een ander ic. We kunnen van de uitgangen ook de nodige timinginformatie bepalen.

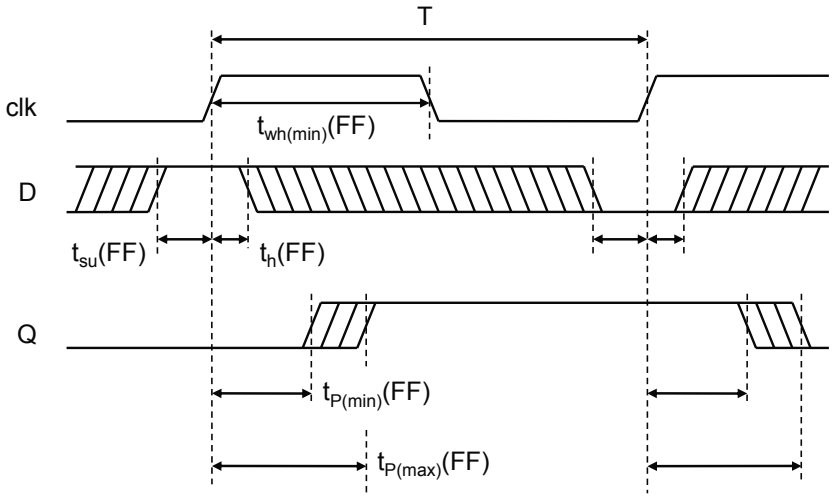
We beginnen dit hoofdstuk met een herhaling van de timing van een D-flipflop. Kennis hiervan is nodig om de rest van dit hoofdstuk te begrijpen. Vervolgens bespreken de timing bij directe dataoverdracht waarbij de uitgang van een flipflop direct wordt verbonden met de ingang van een andere flipflop. We bespreken het verschijnsel klokskew en de invloed daarvan op de timingseigenschappen bij dataoverdracht. In veel gevallen is tussen twee flipflops nog combinatorische logica geplaatst. De tijden van de logica moeten worden meegenomen in de berekeningen. We laten zien dat het timingmodel hiervoor kan worden uitgebreid.

Een belangrijke parameter van een digitaal systeem is de maximale frequentie waarop de schakeling nog betrouwbaar werkt. We laten zien dat het berekenen van de maximale frequentie niet ingewikkeld is. Dit is echter niet de enige parameter waarmee de betrouwbaarheid kan worden aangetoond. Een andere belangrijke parameter is de hold slack. We laten zien dat ook deze parameter kan worden bepaald.

## 9.1 Timing D-flipflop

De digitale component die verreweg het meest wordt gebruikt voor het opslaan van logische waarden is de edge triggered D-flipflop. Deze component neemt de data op de $D$-ingang over op een flank van het kloksignaal. Het kloksignaal is een hulpsignaal en heeft geen logische waarde. Dit overnemen wordt ook wel inklokken genoemd.

In figuur 9.1 is het timingdiagram van een positive edge triggered D-flipflop te zien. We zullen een voor een de timingparameters bespreken.




**Figuur 9.1:** *Timingsdiagram positive edge-triggered D-flipflop.*

Het kloksignaal heeft een periodetijd, weergegeven met de parameter $T$. Verder geldt dat de ingang een gedefinieerde waarde moet hebben rond de opgaande flank. Gedefinieerd wil zeggen een van de twee logische waarden 0 of 1. De setuptijd $t_{su}$(FF) is de tijd waarbij de waarde van de $D$-ingang gedefinieerd moet zijn vóórdat de klok van 0 naar 1 gaat. De holdtijd $t_h$(FF) is de tijd waarbij de waarde van de $D$-ingang gedefinieerd moet blijven nádat de klok van 0 naar 1 gaat. In het gebied $t_{su}$(FF)$-t_h$(FF) moet $D$ dus stabiel blijven en mag niet veranderen. Verandering op de $D$-ingang tijdens $t_{su}$(FF)$-t_h$(FF) kan leiden tot het niet correct overnemen van de data, oscilleren of tot *metastabiliteit* (zie hoofdstuk 10), maar het kan ook zijn dat de flipflop de waarde wel correct overneemt.

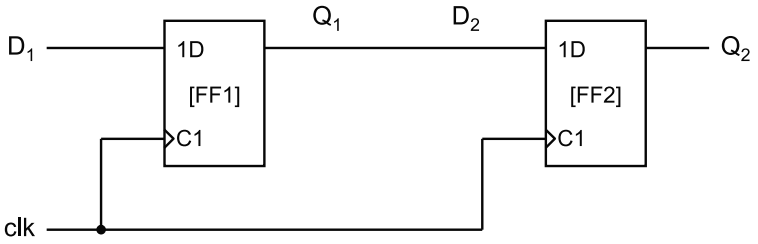
Als aan de voorwaarden voor de setuptijd en holdtijd wordt voldaan dan is de uitgangsreactie niet eerder zichtbaar dan na de minimale vertragingstijd $t_{P(min)}$(FF). Tot die tijd is de oude waarde van de uitgang nog beschikbaar. De nieuwe waarde van de uitgang is beschikbaar na de maximale vertragingstijd $t_{P(max)}$(FF). In het gebied $t_{P(min)}$(FF)$-t_{P(max)}$(FF) is de uitgangswaarde niet gedefinieerd. We weten niet precies hoe het uitgangssignaal verloopt. Het kan zijn dat de uitgangswaarde in één keer omgaat (single transition) of dat de uitgangswaarde een aantal keer verandert voordat een stabiele waarde wordt gerealiseerd.

Aan het kloksignaal worden ook eisen gesteld. Zo moet de tijd dat het kloksignaal hoog is een zekere minimale pulsbreedte hebben. Deze tijd wordt $t_{wh(min)}$(FF) genoemd. Als de pulsbreedte te kort is, kan het zijn dat de flipflop de data niet correct overneemt. Hetzelfde geldt voor de periode dat het kloksignaal laag is. Dit is niet in het timingdiagram

opgenomen. Er zijn ook timingeisen voor asynchrone reset en preset. Die worden niet besproken.

## 9.2 Directe dataoverdracht tussen flipflops

Bij directe dataoverdracht tussen twee flipflops is de uitgang van de eerste flipflop direct verbonden met de ingang van de tweede flipflop. Figuur 9.2 laat het principeschema zien. Ook is te zien dat beide flipflops op hetzelfde kloksignaal zijn aangesloten. Dat betekent dat beide flipflops op hetzelfde moment hun data inklokken[1].




**Figuur 9.2:** *Directe dataoverdracht tussen twee D-flipflop.*

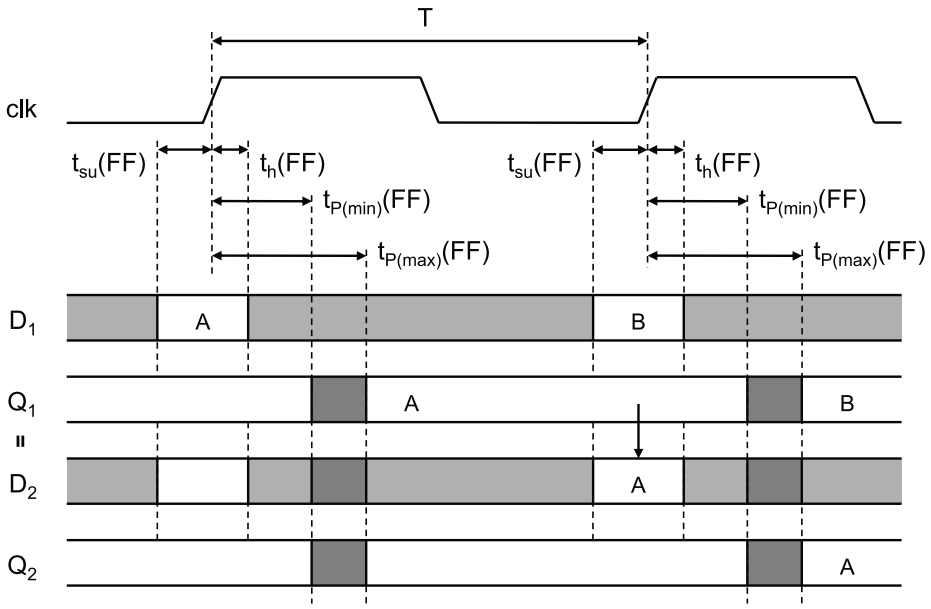
We gaan in eerste instantie ervan uit dat de flipflops dezelfde timingeigenschappen hebben, bijvoorbeeld op hetzelfde ic. Er is directe dataoverdracht, dus geldt $D_2 = Q_1$. Ingang $D_1$ komt van een externe bron en uitgang $Q_2$ gaan naar een externe bron.

Merk op dat alle timingparameters worden gegeven ten opzichte van de actieve klokflank. Voor flipflop 1 geldt dat $D_1$ stabiel moet zijn rond de actieve flank, dus tijdens het gebied $t_{su}(\text{FF}) - t_h(\text{FF})$. Uitgang $Q_1$ van flipflop 1 verandert in het gebied $t_{P(min)}(\text{FF}) - t_{P(max)}(\text{FF})$. Voor flipflop 2 geldt dat $D_2$ stabiel moet zijn rond de actieve klokflank, dus tijdens het gebied $t_{su}(\text{FF}) - t_h(\text{FF})$. Uitgang $Q_2$ van flipflop 2 verandert in het gebied $t_{P(min)}(\text{FF}) - t_{P(max)}(\text{FF})$.

Het geheel kan worden getekend in een timingdiagram. Zie hiervoor figuur 9.3. Alle timingparameters zijn gegeven ten opzichte van de opgaande flank. Ingang $D_1$ moet stabiel zijn in het setuptijd-holdtijd gebied. Dit is aangegeven met de letters A en B. In de lichtgrijze gebieden mag de waarde van $D_1$ veranderen zonder dat de werking van de flipflop wordt beïnvloed. Als gevolg hiervan zal uitgang $Q_1$ van waarde veranderen tussen de minimale en maximale vertragingstijd van de flipflop. Dit is aangegeven met het donkergrijze gebied bij uitgang $Q_1$. Na de maximale vertragingstijd is de waarde A beschikbaar op de uitgang (geldt ook voor waarde B). We zien dat de waarde A ongeveer één klokperiode een stabiele waarde heeft.

De setuptijd en holdtijd gelden ook voor ingang $D_2$, want de flipflops hebben identieke timingparameters. In het witte gebied moet de waarde van $D_2$ stabiel zijn. In het lichtgrijze gebied mag de waarde op ingang $D_2$ veranderen. Aangezien uitgang $Q_1$ is aangesloten op ingang $D_2$ zien we dat de waarde op $D_2$ pas echt veranderd in het donkergrijze gebied, namelijk tussen $t_{P(min)}(\text{FF})$ en $t_{P(max)}(\text{FF})$. De waarde is stabiel ruim voordat de setuptijd ingaat. Dit is weergegeven met een pijl vanuit $Q_1$ naar $D_2$.

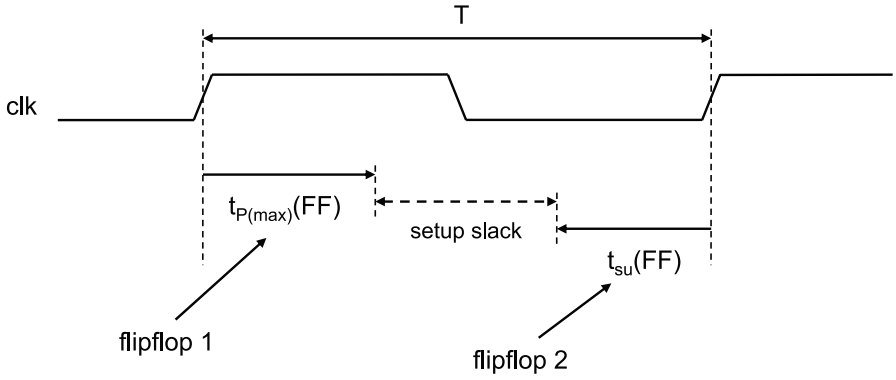
[1] In de praktijk is dat niet het geval, zie paragraaf 9.3.




**Figuur 9.3:** *Timing directe dataoverdracht tussen twee D-flipflop.*

We willen graag betrouwbare dataoverdracht tussen de twee flipflops hebben. Voor flipflop 2 geldt dat $D_2$ stabiel moet zijn tijdens het gebied $t_{su}(\text{FF}) - t_h(\text{FF})$. Dat betekent dat de uitgang $Q_1$ van flipflop 1 dus stabiel moet zijn tijdens het gebied $t_{su}(\text{FF}) - t_h(\text{FF})$. Hieruit volgen twee voorwaarden.

De eerste voorwaarde is dat uitgang $Q_1$ van flipflop 1 stabiel moet zijn *voordat* de setuptijd $t_{su}(\text{FF})$ van flipflop 2 ingaat. Dat houdt in dat de setuptijd $t_{su}(\text{FF})$ van flipflop 2 pas mag ingaan na de maximale propagatietijd $t_{P(max)}(\text{FF})$ van flipflop 1. Deze voorwaarde is weergegeven in figuur 9.4. Dit is een gestileerd timingdiagram waarin alleen de klok is getekend en de tijden die nodig zijn voor het uitdrukken van de eerste voorwaarde. Bij een voldoende grote periodeduur van de klok is tussen het verstrijken van de maximale vertragingstijd van de eerste flipflop het het ingaan van de setuptijd van de tweede flipflop nog enige tijd over. Deze tijd wordt de *setup slack* genoemd. Voor betrouwbare dataoverdacht moet de setup slack groter zijn dan of gelijk zijn aan 0. Deze voorwaarde is *frequentie-afhankelijk*.



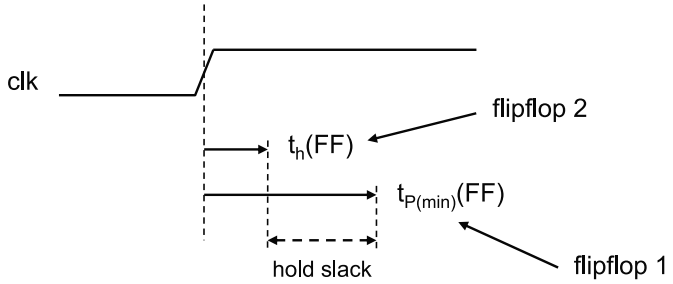

**Figuur 9.4:** *De setup slack bij directe dataoverdracht.*

We kunnen voor de tijden in de figuur de volgende vergelijking opstellen:

$$T = t_{P(max)}(\text{FF}) + t_{slack,su} + t_{su}(\text{FF}) \tag{9.1}$$

Hierin is $t_{slack,su}$ de setup slack.

De tweede voorwaarde is dat uitgang $Q_1$ van flipflop 1 pas weer mag veranderen (niet stabiel zijn) nadat de holdtijd $t_h$(FF) van flipflop 2 is verstreken. Daaruit volgt dat de minimale propagatietijd $t_{P(min)}$(FF) van flipflop 1 groter moet zijn dan of gelijk zijn aan de holdtijd $t_h$(FF) van flipflop 2. Dit is te zien in figuur 9.5. Bij goed ontworpen flipflops is er enige tijd over tussen het verstrijken van de holdtijd en de minimale propagatietijd. Deze tijd wordt *hold slack* genoemd. Voor betrouwbare dataoverdracht moet de hold slack groter zijn dan of gelijk zijn aan 0. Deze voorwaarde is *frequentie-onafhankelijk*.




**Figuur 9.5:** *De hold slack bij directe dataoverdracht.*

We kunnen voor de tijden in de figuur de volgende vergelijking opstellen:

$$t_{P(min)}(\text{FF}) = t_h(\text{FF}) + t_{slack,h} \tag{9.2}$$

Hierin is $t_{slack,h}$ de hold slack.

Aan de voorwaarden voor betrouwbare dataoverdracht wordt voldaan als de tijden van de setup slack en hold slack groter zijn dan of gelijk zijn aan 0:

$$\begin{aligned} t_{slack,su} &= T - t_{P(max)}(\text{FF}) - t_{su}(\text{FF}) \qquad \text{en } t_{slack,su} \geq 0 \\ t_{slack,h} &= t_{P(min)}(\text{FF}) - t_h(\text{FF}) \qquad \text{en } t_{slack,h} \geq 0 \end{aligned} \tag{9.3}$$

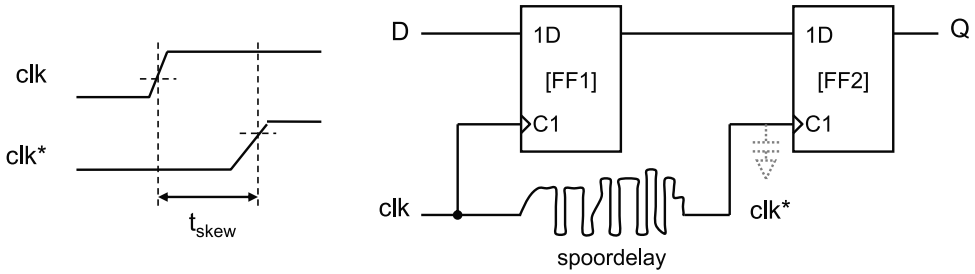
## 9.3 Klokskew

Er zijn nog aanvullende eisen met betrekking op de vorm van het kloksignaal en de interne werking van de flipflop. Het kloksignaal moet *zonder vertraging* worden aangeboden aan beide flipflops. De klokflanken die aan de flipflops worden aangeboden moeten *identieke vorm* hebben (niet vervormd). Het vervormen gebeurt onder andere door *capacitieve belasting*. Beide flipflops moeten op *hetzelfde (spannings-)niveau* van de actieve klokflank data inklokken.

In de praktijk wordt hier nooit aan voldaan. Vooral op grote ic's is vertraging in de klok-lijn een lastig probleem. Op grote ic's wordt daarom een speciaal klokdistributiesysteem aangelegd om vertragingstijden door belasting te minimaliseren. Het geheel van vertraging, vervorming door capacitieve belasting en triggerpunt wordt *klokskew* (Engels: clock skew) genoemd.

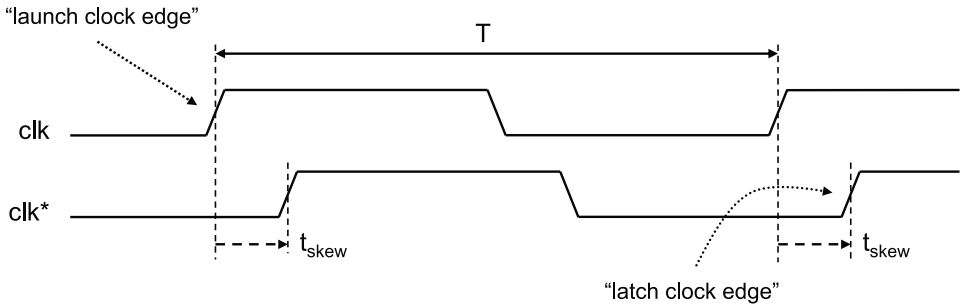
*In elk digitaal systeem is dus sprake van klokskew.*

Het timingmodel moet worden uitgebreid met een nieuwe parameter $t_{skew}$. Zie figuur 9.6.



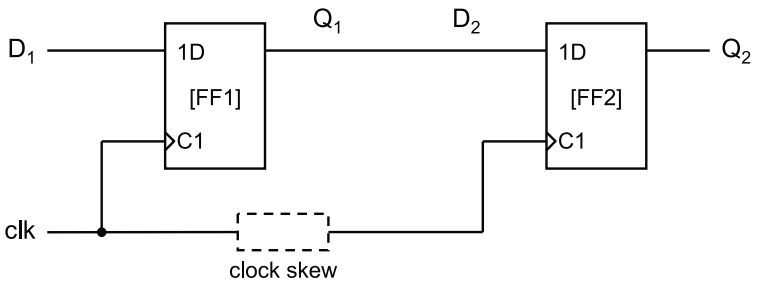

**Figuur 9.6:** *Klokskew in een digitaal systeem.*

De klokskew kunnen we in een timingdiagram tekenen als twee kloksignalen die ten opzichte van elkaar iets verschoven zijn. Om de twee actieve klokflanken uit elkaar te houden, wordt gesproken van een *launch clock edge* en een *latch clock edge*. Figuur 9.7 verduidelijkt de twee begrippen. Merk op dat er geen exacte timinginformatie is gegeven.




**Figuur 9.7:** *De launch clock edge en de latch clock edge.*

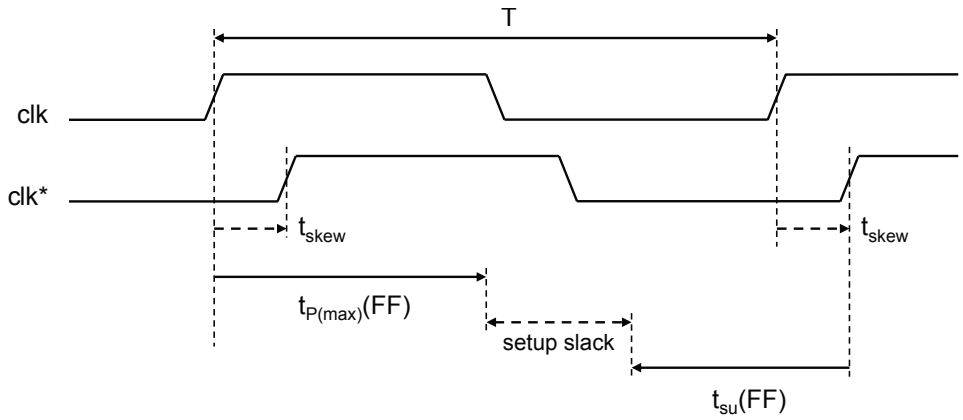
Voor het berekenen van de timing gaan we uit van het schema in figuur 9.8. In de signaalweg van de klok is nu een vertraging opgenomen die de klokskew voorstelt.




**Figuur 9.8:** *Directe dataoverdracht met klokskew.*

We tekenen een timingdiagram en leiden de formule voor $T$ af. Merk op dat de periodeduur voor de dataoverdracht wordt verlengd met $t_{skew}$. Zie figuur 9.9. We krijgen dus de volgende vergelijking:

$$T + t_{skew} = t_{P(max)}(\text{FF}) + t_{slack,su} + t_{su}(\text{FF}) \tag{9.4}$$



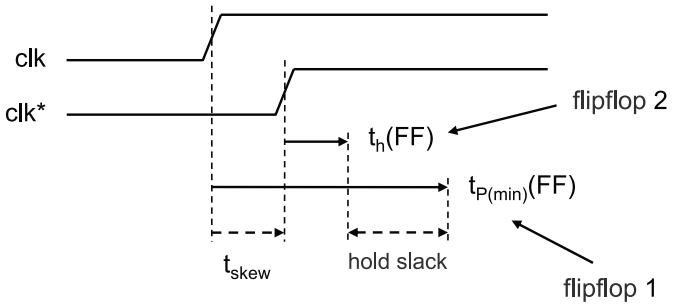

**Figuur 9.9:** *Timing directe dataoverdracht met klokskew.*

We herschrijven de vergelijking door $T$ expliciet te maken:

$$T = t_{P(max)}(\text{FF}) + t_{slack,su} + t_{su}(\text{FF}) - t_{skew} \tag{9.5}$$

Klokskew heeft (in dit geval) *positieve* invloed op de setup slack. De setup slack wordt groter bij een positieve waarde van de klokskew.

We moeten echter ook nog naar de hold slack kijken. Het timingdiagram is getekend in figuur 9.10.




**Figuur 9.10:** *Timing directe dataoverdracht met klokskew.*

Klokskew heeft (in dit geval) *negatieve* invloed op de hold slack. De uitgang van de eerste flipflop mag pas veranderen na de holdtijd van de tweede flipflop. Deze holdtijd wordt "verlengd" door de klokskew. In formulevorm:

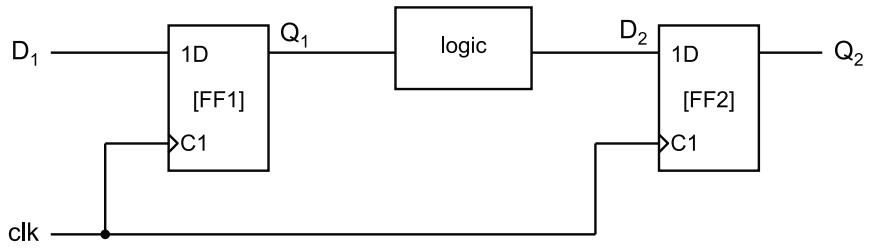
$$t_{P(min)}(\text{FF}) = t_{skew} + t_h(\text{FF}) + t_{slack,h} \tag{9.6}$$

De hold slack wordt kleiner bij een positieve waarde van de klokskew.

In bovengenoemde gevallen is waarde van de klokskew positief. Het is echter ook mogelijk om een negatieve waarde van de klokskew te hebben. Dat is het geval als de sturende flipflop de klokflank later krijgt aangeboden dan de ontvangende flipflop. De eerder gegeven formules moeten hiervoor worden aangepast. Zie ook paragraaf 9.9.

## 9.4 Indirecte dataoverdracht tussen flipflops

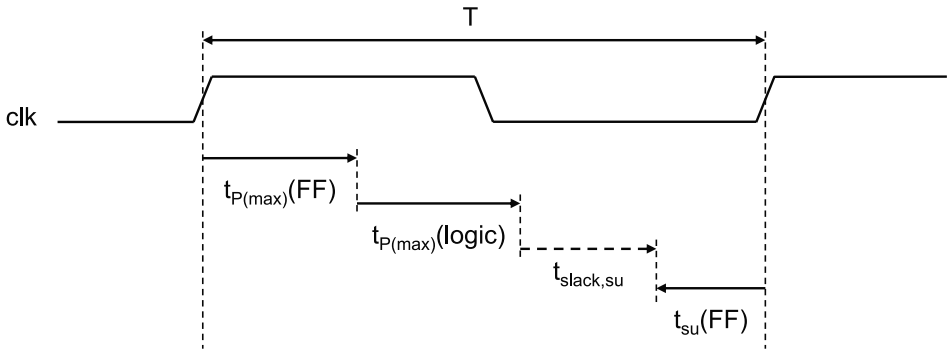
Bij *indirecte dataoverdracht* ligt tussen de twee flipflops nog een stuk combinatorische logica. De uitgang van de eerste flipflop is verbonden met de ingang van de logica en de uitgang van de logica is verbonden met de ingang van de tweede flipflop. Een principeschema is te zien in figuur 9.11.




**Figuur 9.11:** *Indirecte dataoverdracht tussen twee D-flipflop.*

Noot: in de figuur en de formules is de klokskew niet opgenomen.

Voor het berekenen van de periodeduur $T$ moet nu de maximale vertragingstijd van de logica $t_{P(max)}$(logic) meegenomen worden. Het kost namelijk enige tijd voordat een signaal door de logica is gepropageerd. Dit is te zien in het timingdiagram in figuur 9.12.




**Figuur 9.12:** *Timing indirecte dataoverdracht tussen twee D-flipflop.*

We kunnen de vergelijking voor $T$ opstellen:

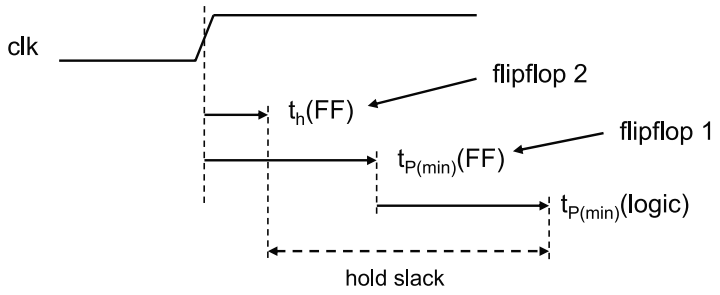
$$T = t_{P(max)}(\text{FF}) + t_{P(max)}(\text{logic}) + t_{slack,su} + t_{su}(\text{FF}) \tag{9.7}$$

Hieruit volgt dat logica tussen de flipflops een negatieve invloed heeft op de setup slack.

Over de hold slack kunnen we het volgende zeggen. De waarde van de uitgang van de eerste flipflop verandert op z'n vroegst na de minimale vertragingstijd van de flipflop. Daarna gaat het signaal door de logica. Op z'n vroegst verandert de uitgang van de logica na de minimale vertragingstijd van de logica. Deze twee tijden opgeteld moet groter zijn dan of gelijk zijn aan de holdtijd. Dit is te zien in figuur 9.13. In formulevorm:

$$t_{P(min)}(\text{FF}) + t_{P(min)}(\text{logic}) = t_h(\text{FF}) + t_{slack,h} \tag{9.8}$$

Logica tussen de flipflops heeft dus een positieve invloed op de hold slack.




**Figuur 9.13:** *Timing indirecte dataoverdracht tussen twee D-flipflop.*

Voor betrouwbare dataoverdracht moet gelden dat de setup slack en de hold slack groter zijn dan of gelijk zijn aan 0. We kunnen dus voor de setup slack en de hold slack het volgende opstellen:

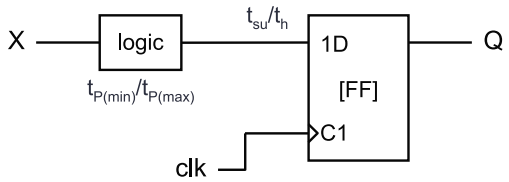
$$\begin{aligned} t_{slack,su} &\geq T - t_{P(max)}(\text{FF}) - t_{P(max)}(\text{logic}) - t_{su}(\text{FF}) \qquad \text{en } t_{slack,su} \geq 0 \\ t_{slack,h} &\geq t_{P(min)}(\text{FF}) + t_{P(min)}(\text{logic}) - t_h(\text{FF}) \qquad \text{en } t_{slack,h} \geq 0 \end{aligned} \tag{9.9}$$

Noot: klokskew is niet in de formule verwerkt.

## 9.5 Timing van externe ingangen en uitgangen

We hebben ons tot nu toe bezig gehouden met timing tussen flipflops onderling. Systemen hebben echter ook interactie met de buitenwereld. We kunnen voor de ingangen en uitgangen timingseisen opstellen zodat dataoverdracht correct verloopt.

Als er logica geplaatst wordt voor de ingang van de flipflop, heeft dit gevolgen voor de timingparameters van de sturende ingang. In figuur 9.14 is dit weergegeven.




**Figuur 9.14:** *Aansturing van een D-flipflop met een extern signaal.*

Signaal $X$ wordt aangesloten op een stuk combinatorische logica en die is weer aangesloten op de $D$-ingang van de flipflop. Voor de combinatorische logica gelden de minimale en maximale vertragingstijden $t_{P(min)}$(logic) en $t_{P(max)}$(logic). Voor signaal $X$ gelden dan andere setup- en holdtijden dan voor de $D$-ingang van de flipflop. Voorwaarde is natuurlijk dat de setup- en holdtijden van de $D$-ingang gerespecteerd worden. Een verandering op signaal $X$ heeft maximaal $t_{P(max)}$(logic) nodig om door de logica te propageren. De laatst mogelijke verandering op $X$ moet dus $t_{P(max)}$(logic) vóór $t_{su}$(FF) gebeuren. Dus geldt:

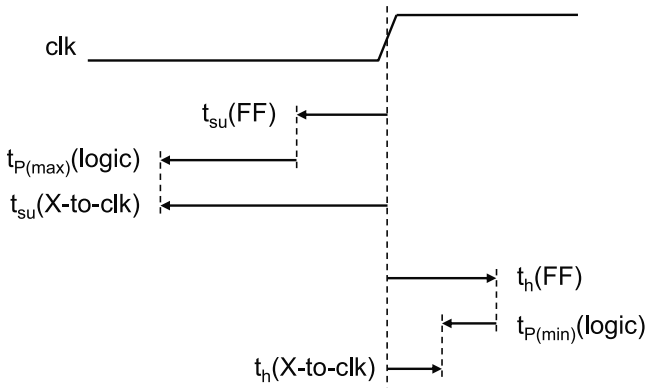
$$t_{su}(\text{X-to-clk}) = t_{su}(\text{FF}) + t_{P(max)}(\text{logic}) \tag{9.10}$$

De waarde op de $D$-ingang van de flipflop moet na de actieve flank nog even stabiel blijven, namelijk voor de duur van holdtijd $t_h$(FF). Een verandering op signaal $X$ heeft

minimaal $t_{P(min)}$(logic) nodig om door de logica te propageren. Dan pas mag de holdtijd $t_h$(FF) van de flipflop verstreken zijn. De eerst mogelijke (nieuwe) verandering op $X$ mag dus $t_{P(min)}$(logic) vóór $t_h$(FF) gebeuren. Dus geldt:

$$t_h(\text{X-to-clk}) = t_h(\text{FF}) - t_{P(min)}(\text{logic}) \tag{9.11}$$

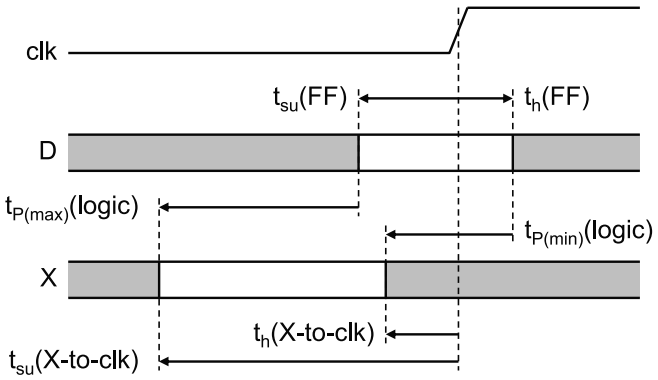
Let erop dat er een minteken in de vergelijking staat. Dit is gevolg van de definities van minimale vertragingstijd en de holdtijd. Beide vergelijkingen zijn uitgebeeld in figuur 9.15.




**Figuur 9.15:** *Timing van de setup- en holdtijden van externe ingangen.*

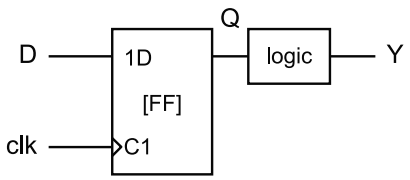
Voor de holdtijd kan nog een opmerking worden gemaakt. In veel gevallen is de minimale vertragingstijd $t_{P(min)}$(logic) *groter* dan de holdtijd $t_h$(FF). De berekening van de holdtijd $t_h$(X-to-clk) levert dan een *negatief* antwoord op. Dit houdt in dat het signaal op de ingang van de logica *voor* de klokflank al weer mag veranderen.

In figuur 9.16 is nogmaals te zien hoe de timingparameters van de externe ingang kunnen worden berekend. In het lichtgrijze gedeelte mag het ingangssignaal veranderen. In het witte gedeelte moet het ingangssignaal stabiel blijven. Te zien is dat het gebied waar ingangssignaal $X$ stabiel moet blijven in z'n geheel voor de opgaande flank ligt.




**Figuur 9.16:** *Timing van de setup- en holdtijden van externe ingangen.*

Aan de uitgangen van de flipflops kan logica worden gekoppeld. In figuur 9.17 is dit geschetst.

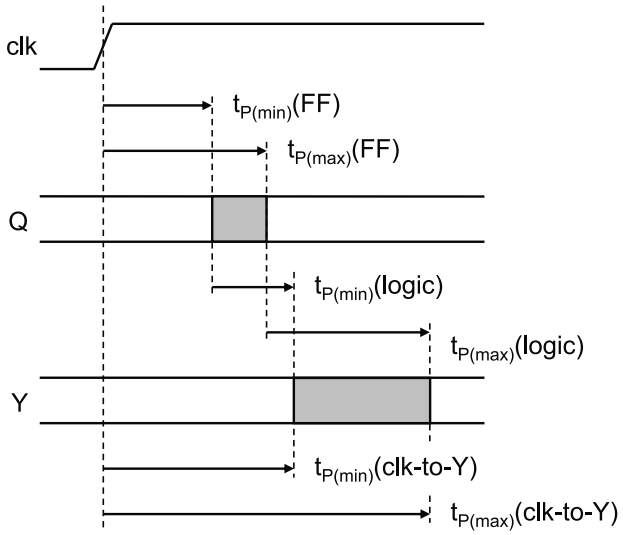



**Figuur 9.17:** *Logica geplaatst na de flipflops.*

Dit levert een vertraging op van de uitgang ten opzichte van de actieve klokflank. De minimale vertragingstijd van de uitgang is de som van de minimale vertragingstijden van de flipflop en de logica. De maximale vertragingstijd van de uitgang is de som van de maximale vertragingstijden van de flipflop en de logica. Deze zijn weergegeven in onderstaande vergelijkingen:

$$\begin{aligned} t_{P(min)}(\text{clk-to-Y}) &= t_{P(min)}(\text{FF}) + t_{P(min)}(\text{logic}) \\ t_{P(max)}(\text{clk-to-Y}) &= t_{P(max)}(\text{FF}) + t_{P(max)}(\text{logic}) \end{aligned} \tag{9.12}$$

De vergelijkingen zijn weergegeven in het timingdiagram in figuur 9.18.

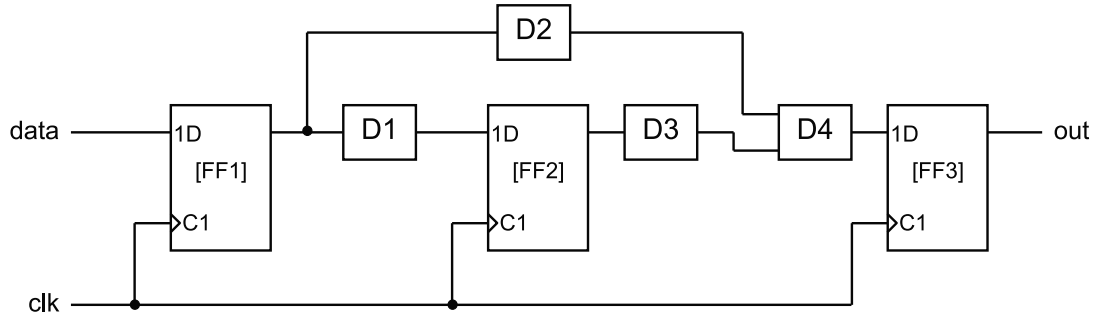



**Figuur 9.18:** *Timing van de uitgangen met logica geplaatst na de flipflops.*

## 9.6 De maximale systeemfrequentie

Een belangrijke parameter van een digitaal systeem is de maximale frequentie waarop het systeem nog betrouwbaar werkt. In een digitaal systeem zijn er meerdere paden tussen de flipflops. Voor het bepalen van de maximale frequentie (oftewel de minimale periodetijd) moeten alle paden doorgerekend worden. We gaan ervan uit dat alle flipflops identieke timing hebben. Binnen een ic is dat bijna altijd het geval.

We zullen dit toelichten aan de hand van het voorbeeld in figuur 9.19. In de schakeling zijn drie paden te ontdekken waarlangs data van flipflop naar flipflop wordt getransporteerd. De paden zijn:




**Figuur 9.19:** *Schakeling voor het bepalen van de maximale systeemfrequentie.*

FF1 → D1 → FF2
FF1 → D2 → D4 → FF3
FF1 → D3 → D4 → FF3

Elk pad levert een minimale periodetijd $T_{min}$ behorende bij het pad. De *grootste* minimale periodetijd bepaalt de maximale frequentie van het hele systeem. We kunnen dus noteren:

$$\begin{aligned} T_{min,1} &= t_{P(max)}(\text{FF}) + t_{P(max)}(\text{D1}) + t_{su}(\text{FF}) \\ T_{min,2} &= t_{P(max)}(\text{FF}) + t_{P(max)}(\text{D2}) + t_{P(max)}(\text{D4}) + t_{su}(\text{FF}) \\ T_{min,3} &= t_{P(max)}(\text{FF}) + t_{P(max)}(\text{D3}) + t_{P(max)}(\text{D4}) + t_{su}(\text{FF}) \end{aligned} \tag{9.13}$$

Klokskew is in de vergelijkingen niet meegenomen. De minimale periodetijd van het gehele systeem kan berekend worden met:

$$T_{min,systeem} = \max(T_{min,1}, T_{min,2}, T_{min,3}) \tag{9.14}$$

Het langzaamste pad bepaalt de maximale frequentie. Dit wordt het *kritieke pad* genoemd. De setup slack van dit pad is 0. De reciproke van de minimale periodetijd bepaalt de maximale frequentie:

$$f_{max,systeem} = \frac{1}{T_{min,systeem}} \tag{9.15}$$

Te zien is dat het berekenen van de maximale frequentie geen ingewikkelde zaak is. Het enige is dat een groot systeem misschien we honderden paden heeft. De rekenklus duurt dan erg lang. Gelukkig kan de synthesesoftware dit allemaal uitrekenen.

Verder moet gelden dat geen enkel pad een *hold time violation* mag hebben. De hold slack moet van alle paden moet groter zijn dan of gelijk zijn aan 0. Dit kan eventueel opgelost worden door een vertraging in het datasignaal te plaatsen. De hold slacks van het systeem in figuur 9.19 kunnen berekend worden met:

$$\begin{aligned} t_{slack,h}(\text{FF1} \rightarrow \text{FF2}) &= t_{P(min)}(\text{D1}) - t_h(\text{FF}) \\ t_{slack,h}(\text{FF1} \rightarrow \text{FF3}) &= t_{P(min)}(\text{D2}) + t_{P(min)}(\text{D4}) - t_h(\text{FF}) \\ t_{slack,h}(\text{FF2} \rightarrow \text{FF3}) &= t_{P(min)}(\text{D3}) + t_{P(min)}(\text{D4}) - t_h(\text{FF}) \end{aligned} \tag{9.16}$$

Ook dit is geen lastige klus maar levert wel veel rekenwerk op.

Er zijn technieken om de timing van de paden te optimaliseren. Zo kan er een herontwerp van het kritieke pad gedaan worden. Eerst wordt uitgezocht welk pad het langzaamste pad is en wordt de schakeling daar op aangepast. Dat is niet altijd gemakkelijk. In veel gevallen moet de gehele schakeling opnieuw ontworpen worden. Een andere mogelijkheid is *retiming*. Hierbij wordt met (interne) flipflops in de schakeling geschoven zodat de timing van de paden verandert. Er is software op de markt die dit automatisch doet. Er kan ook gebruikt gemaakt worden van *pipelining*. Hierbij wordt een langdurende operatie opgeknipt in een aantal kortdurende operatie. Denk hierbij aan de parallelvermenigvuldiger waarin een aantal optellers is geplaatst. Door tussen de optellers flipflops te plaatsen, wordt de tijdsduur van de paden ingekort. Het nadeel is wel dat er een aantal klokslagen moet worden gewacht voordat de uitkomst bekend is. Het is ook mogelijk om een *multi-cycle pad* te ontwerpen. Hierbij wordt het overnemen van de data door een flipflop een aantal klokflanken uitgeschakeld. Dit wordt gerealiseerd met een enable-faciliteit op de flipflop. De kloklijn wordt natuurlijk niet afgeschakeld. Het nadeel is dat de timingsoftware hiervan wel op de hoogte moet zijn.

## 9.7 Vertraging van signaallijnen

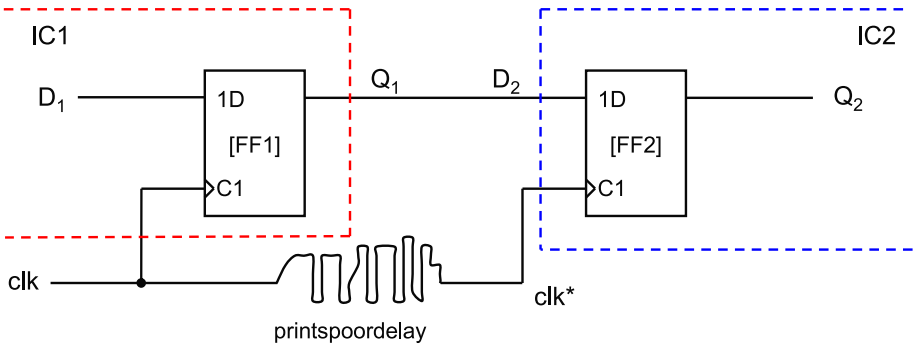
Al eerder hebben we het verschijnsel klokskew behandeld. Dat is vertraging van het kloksignaal door allerlei factoren. Ook datalijnen hebben enige vertraging. We noemen dit *dataskew*. Bedenk dat bij FPGA's de vertragingstijden van verbindingen tussen cellen tot de helft van de totale vertragingstijd tussen twee flipflops kunnen bedragen. Ook externe verbindingen leveren vertraging op. Een printspoor heeft een vertraging van ongeveer 1 ns per 15 cm. Stel dat een geheugen op 15 cm afstand van een microprocessor is geplaatst dan bedraagt de vertraging over de print al 2 ns. Bij langzame logica is dat niet zo'n probleem maar veel processoren draaien al op enkele gigahertz. Een processor die draait op 1 GHz krijgt dan elke nanoseconde een klokpuls. De processor moet dan sowieso 2 ns wachten (2 klokpulsen) voor het versturen en ontvangen van de data (de processor moet ook nog wachten tot het geheugen de data beschikbaar heeft, de zogenoemde *access time*). In de meeste gevallen kunnen we de dataskew verrekenen met de vertraging van de logica.

## 9.8 Timing met verschillende parameters

Binnen een ic zijn de timingparameters van de flipflops identiek. Bij verbindingen tussen verschillende ic's is de kans groot dat de timingparameters verschillend zijn. In figuur 9.20 is een verbinding tussen twee verschillende ic's te zien. Flipflop 1 is de sturende flipflop en flipflop 2 is de ontvangende flipflop. Tevens is er sprake van klokskew. We kunnen de setup slack en hold slack als volgt berekenen:

$$\begin{aligned} t_{slack,su} &= T + t_{skew} - t_{P(max)}(\text{FF1}) - t_{su}(\text{FF2}) \\ t_{slack,h} &= t_{P(min)}(\text{FF1}) - t_{skew} - t_h(\text{FF2}) \end{aligned} \tag{9.17}$$

Er kunnen problemen ontstaan bij overdracht van een snelle naar een langzame technologie, met name bij de hold slack. Bij een snel ic is de minimale vertragingstijd klein.

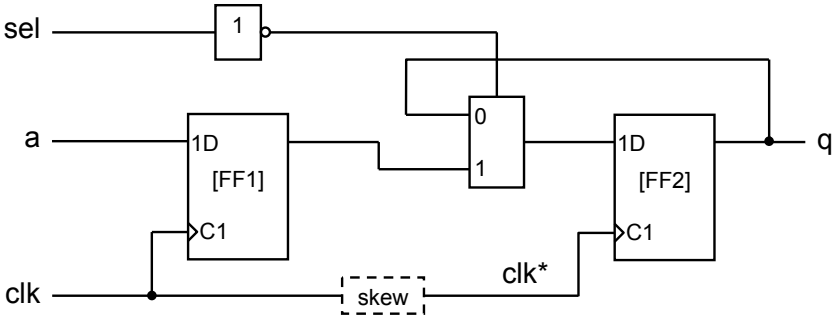



**Figuur 9.20:** *Timing met verschillende ic's.*

Bij een langzaam ic is de holdtijd groot. Er is dan een gerede kans dat de hold slack negatief is. Er is dan sprake van een hold time violation. De setup slack vormt meestal geen probleem omdat de klokfrequentie aangepast kan worden aan het langzaamste ic.

## 9.9 Voorbeelden

Gegeven is de schakeling in figuur 9.21. Hierin zijn twee flipflops opgenomen (FF1 en FF2) en twee stukken combinatorische logica (multiplexer en NOT-poort). De kloklijn heeft last van skew, de klok wordt vertraagd aangeboden aan FF2.




**Figuur 9.21:** *Synchrone flipflop-schakeling*

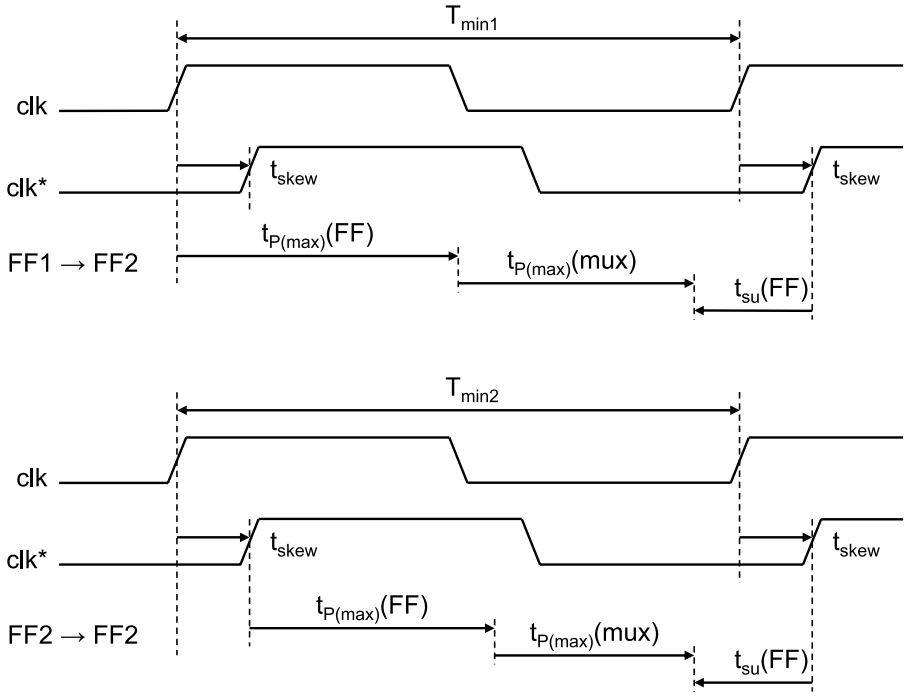
Van de bouwstenen zijn de volgende timingparameters gegeven:

D-flipflops: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 3/1/5/7$ ns
NOT-poort: $t_{P(min)}/t_{P(max)} = 1/3$ ns
multiplexer: $t_{P(min)}/t_{P(max)} = 5/8$ ns
skew: $t_{skew} = 2$ ns

Van de skew is maar één vertragingstijd gegeven, er wordt geen onderscheid gemaakt tussen de minimale en maximale vertragingstijd. Het betreft hier namelijk klokskew.

In het systeem zijn twee paden tussen de flipflops te ontdekken: FF1 → FF2 en FF2 → FF2. Voor het bepalen van de minimale periodetijd van het hele systeem tekenen we twee timingdiagrammen. Zie figuur 9.22. Bij dataoverdracht van FF1 → FF2 gaat de data weg op de opgaande flank van $clk$ en komt aan op de opgaande flank van $clk*$. Bij deze dataoverdracht is er sprake van klokskew. Bij dataoverdracht van FF2 → FF2 gaat

de data weg op de opgaande flank van $clk*$ en komt de data aan op de opgaande flank van $clk*$. De klokskew heeft hier geen invloed op deze timing.




**Figuur 9.22:** *Bepalen minimale periodetijd.*

We stellen de formules op om de minimale periodetijd te berekenen:

$$\begin{aligned} \text{FF1} \rightarrow \text{FF2:}\ & T_{min1} = t_{P(max)}(\text{FF}) + t_{P(max)}(\text{mux}) + t_{su}(\text{FF}) - t_{skew} \\ \text{FF2} \rightarrow \text{FF2:}\ & T_{min2} = t_{P(max)}(\text{FF}) + t_{P(max)}(\text{mux}) + t_{su}(\text{FF}) \end{aligned} \tag{9.18}$$

De grootste minimale periodetijd wordt gegeven door de dataoverdracht FF2 → FF2 en bedraagt 18 ns. De maximale frequentie bedraagt ongeveer 55 MHz.

De setuptijd en holdtijd van signaal $a$ ten opzichte van de klok zijn eenvoudigweg de setuptijd en holdtijd van de flipflop. Voor signaal $sel$ ligt dat anders. Dit is te zien in figuur 9.23. Het inklokken door flipflop 2 is door de klokskew naar "achter" geschoven, het gebeurt immers op de opgaande flank van $clk*$.

We kunnen voor de setuptijd van signaal $sel$ het volgende opstellen:

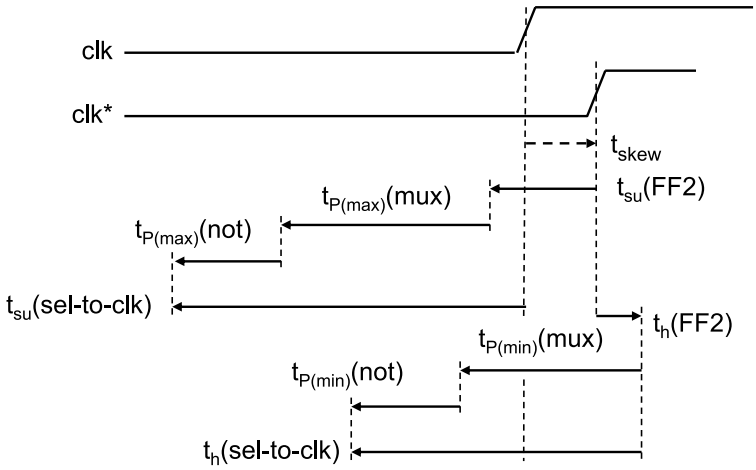
$$t_{su}(\text{sel-to-clk}) = -t_{skew} + t_{su}(\text{FF2}) + t_{P(max)}(\text{mux}) + t_{P(max)}(\text{NOT}) = 12 \text{ ns} \tag{9.19}$$

Merk op dat de skewtijd in de formule van het geheel moet worden afgetrokken als gevolg van de richting van de tijden. Voor de holdtijd kunnen we opstellen:

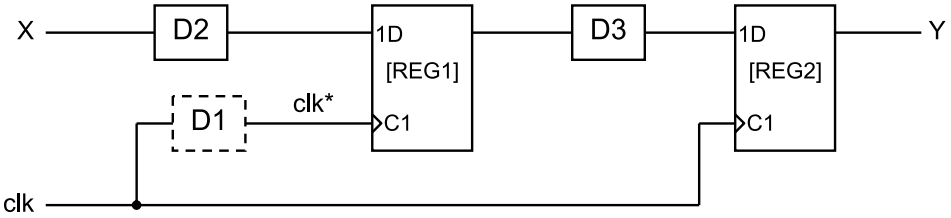
$$t_h(\text{sel-to-clk}) = t_{skew} + t_h(\text{FF2}) - t_{P(min)}(\text{mux}) - t_{P(min)}(\text{NOT}) = -3 \text{ ns} \tag{9.20}$$

De holdtijd is negatief. Dat houdt in dat de data nog voor de klokflank mag veranderen.

Gegeven is de schakeling in figuur 9.24. Hierin zijn twee registers opgenomen (REG1 en REG2) en drie stukken combinatoriek (D1, D2 en D3). D1 vertraagt het kloksignaal




**Figuur 9.23:** *Bepalen setuptijd en holdtijd van signaal* `sel`.




**Figuur 9.24:** *Synchrone flipflop-schakeling*

voor REG1. D2 is combinatoriek voor de ingang van REG1 en D3 is combinatoriek tussen REG1 en REG2.

Van de bouwstenen zijn de volgende timingparameters gegeven:

Registers: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 2/1/5/7$ ns

D2: $t_{P(min)}/t_{P(max)} = 1/3$ ns

D3: $t_{P(min)}/t_{P(max)} = 5/8$ ns

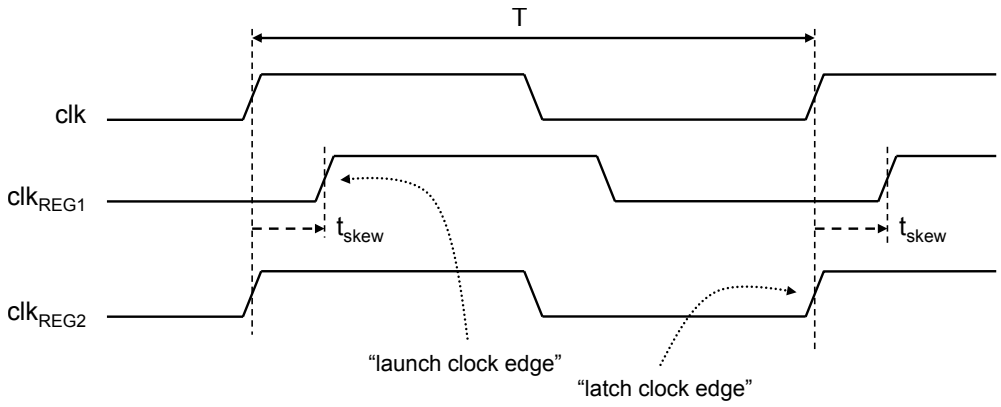
D1: $t_P = 4$ ns

Van D1 is maar één vertragingstijd gegeven, er wordt geen onderscheid gemaakt tussen de minimale en maximale vertragingstijd. Het betreft hier namelijk klokskew.

Als eerste tekenen we de kloksignalen ten opzichte van de (externe) systeemklok. Zie figuur 9.25. Let op de skew in de klok van register 1. Deze situatie doet zich voor wanneer het sturende register zijn klok later krijgt aangeboden dan het ontvangende register. Merk op dat $clk$ en $clk_{\mathrm{REG2}}$ identiek zijn.

We kunnen de maximale systeemfrequentie als volgt uitrekenen. De data gaat weg op de launch clock edge. Dat is de skewtijd na de actieve klokflank van $clk$. De data wordt ontvangen op de latch clock edge. Deze klok heeft geen last van skew. De periodetijd wordt dus verkort met de skewtijd. In formulevorm:

$$T_{min} - t_{skew} = t_{P(max)}(\mathrm{REG1}) + t_{P(max)}(\mathrm{D3}) + t_{su}(\mathrm{REG2}) \tag{9.21}$$

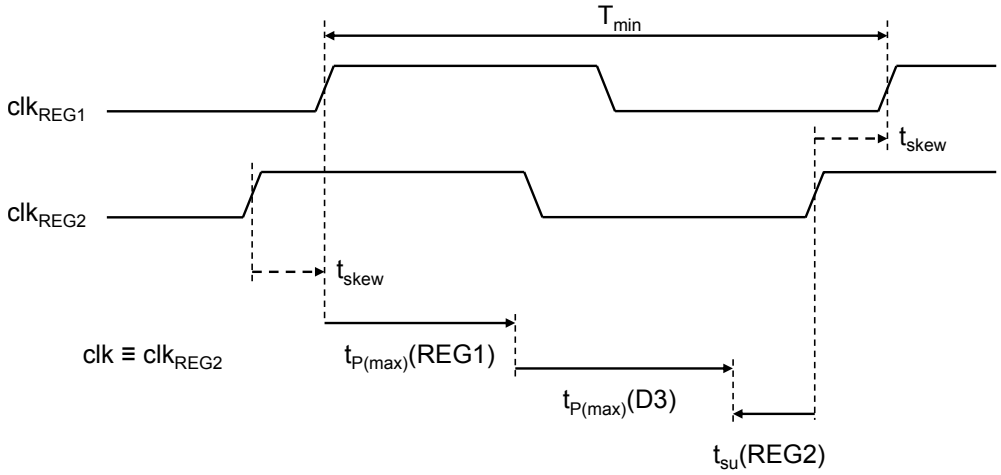



**Figuur 9.25:** *Timingdiagram kloksignalen.*

We kunnen nu $T$ expliciet maken:

$$\begin{aligned} T_{min} &= t_{P(max)}(\text{REG1}) + t_{P(max)}(\text{D3}) + t_{su}(\text{REG2}) + t_{skew} \\ &= 7 + 8 + 2 + 4 \\ &= 21 \text{ ns} \end{aligned} \tag{9.22}$$

De klokskew heeft nu een negatieve invloed op de minimale periodetijd. Zie ook figuur 9.26.



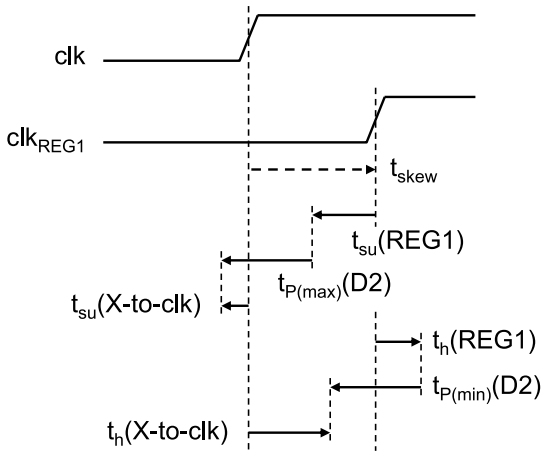

**Figuur 9.26:** *Timingdiagram voor bepalen minimale periodetijd.*

In figuur 9.27 staat het timingdiagram waarmee de setuptijd en holdtijk van $X$ t.o.v. de klok kan worden berekend. Let er op dat $clk_{\text{REG1}}$ skew ondervindt en dat moet verrekend worden.

We rekenen de setuptijd uit:

$$\begin{aligned} t_{su}(\text{X-to-clk}) &= -t_{skew} + t_{su}(\text{REG2}) + t_{P(max)}(\text{D2}) \\ &= -4 + 2 + 3 \\ &= 1 \text{ ns} \end{aligned} \tag{9.23}$$

Merk op dat de skewtijd in de formule van een minteken is voorzien. Dat is het gevolg van de definities van de richtingen van de diverse tijden. Merk ook op dat de setuptijd




**Figuur 9.27:** *Timingdiagram voor bepalen setuptijd en holdtijd.*

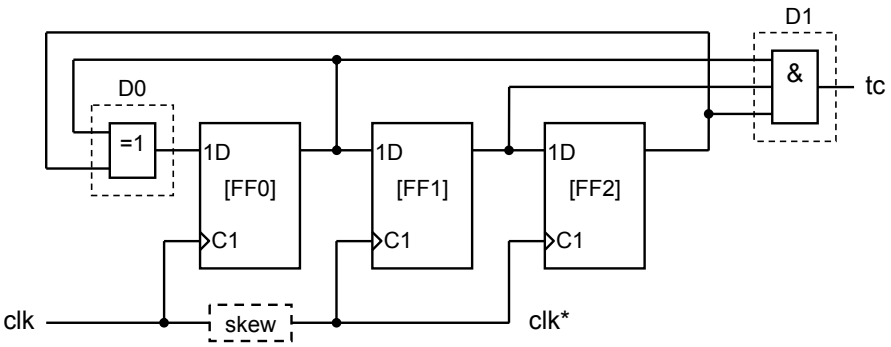
door de skew verkort wordt. Theoretisch zou de setuptijd negatief kunnen worden als de skew erg groot is.

Voor de holdtijd stellen we de formule op:

$$\begin{aligned} t_h(\text{X-to-clk}) &= t_{skew} + t_h(\text{FF}) - t_{P(min)}(\text{D2}) \\ &= 4 + 1 - 1 \\ &= 4 \text{ ns} \end{aligned} \tag{9.24}$$

Merk op dat de skewtijd een positieve bijdrage heeft aan de holdtijd als gevolg van de definities van de richtingen van de diverse tijden.

Gegeven is de schakeling in figuur 9.28. Hierin zijn drie D-flipflops opgenomen (FF0, FF1 en FF2), twee stukken combinatoriek (D0, D1) en één klokvertraging (skew). D0 is combinatoriek voor de ingang van FF0 en D1 is combinatoriek na de uitgangen van FF0, FF1 en FF2. FF1 en FF2 krijgen een vertraagd kloksignaal aangeboden (skew).




**Figuur 9.28:** *Synchrone flipflopschakeling.*

Van de bouwstenen zijn de volgende timingparameters gegeven:

D-flipflops: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 2/1/6/8$ ns
AND: $t_{P(min)}/t_{P(max)} = 2/3$ ns
EXOR: $t_{P(min)}/t_{P(max)} = 3/5$ ns

Skew: $t_p = 3$ ns

Van de skew is maar één vertragingstijd gegeven, er wordt geen onderscheid gemaakt tussen de minimale en maximale vertragingstijd. Het betreft hier namelijk klokskew.

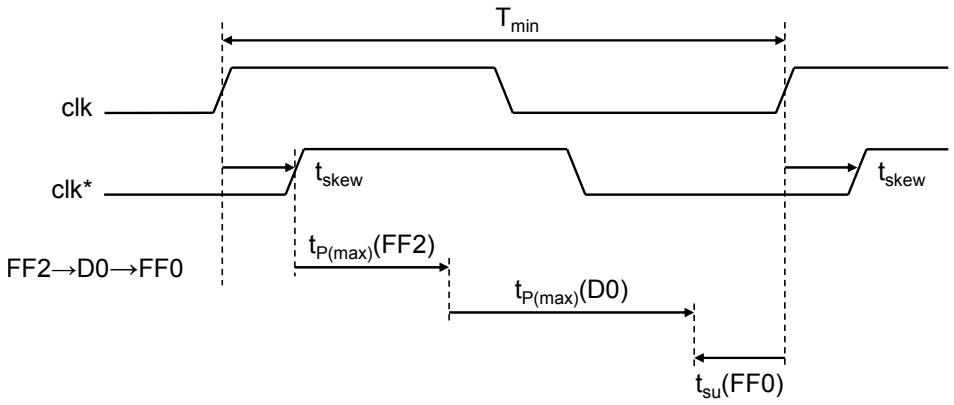
In dit systeem zijn er vier paden tussen de flipflops waarlangs data wordt getransporteerd. De vier paden zijn:

FF0 → D0 → FF0
FF0 → FF1
FF1 → FF2
FF2 → D0 → FF0

Merk op dat D1 in geen van de paden voorkomt want dat een logica voor een uitgang. Na onderzoek blijkt het pad FF2 → D0 → FF0 de grootste periodetijd op te leveren.

In figuur 9.29 is het timingdiagram getekend. Het langste pad in tijd is van FF2 via D0 naar FF0. De klokingang van FF2 heeft last van skew, dus pas na $t_{skew}$ krijgt FF2 een actieve flank. Dat is signaal $clk*$. Dan moet nog de maximale propagatietijd van de EXOR-poort gewacht worden (D0) voordat de data beschikbaar is op de ingang van FF0. Dan moet nog één setuptijd $t_{su}$(FF) gewacht worden voordat een actieve flank voorbij mag komen. Let op dat FF0 geen last heeft van klokskew, dus FF0 klokt data in op signaal $clk$.

Noot: de andere drie paden leveren na uitwerking een kleinere minimale periodetijd dan het hier boven genoemde pad.




**Figuur 9.29:** *Timingdiagram voor bepalen maximale frequentie.*

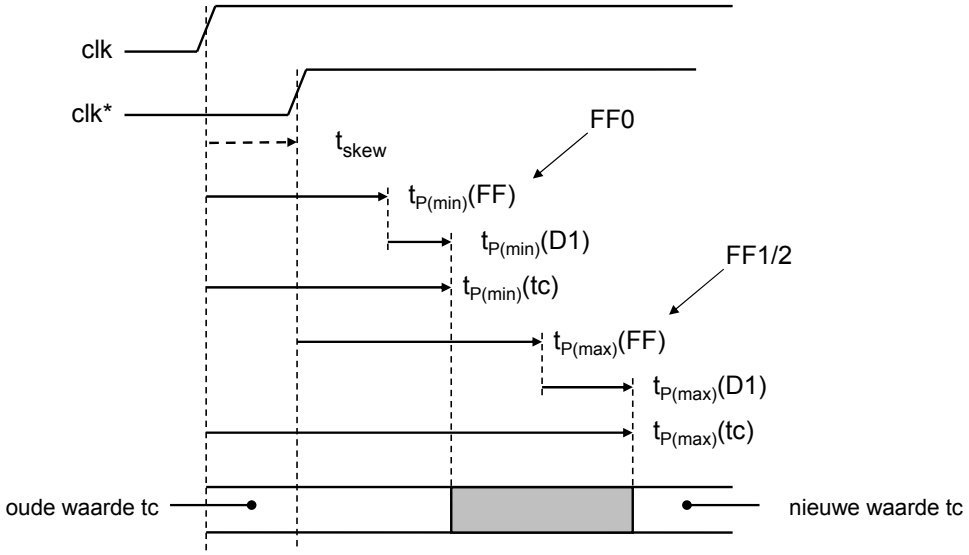
We stellen de formule op voor het berekenen van de minimale periodeduur van de klok:

$$T_{min} - t_{skew} = t_{P(max)}(\text{FF2}) + t_{P(max)}(\text{D0}) + t_{su}(\text{FF0}) \tag{9.25}$$

We herschikken de skewtijd zodat de minimale periodeduur kan worden berekend:

$$\begin{aligned} T_{min} &= t_{P(max)}(\text{FF2}) + t_{P(max)}(\text{D0}) + t_{su}(\text{FF0})) + t_{skew} \\ &= 8 + 5 + 2 + 3 \\ &= 18 \text{ ns} \end{aligned} \tag{9.26}$$

Bij de berekeningen van de minimale en maximale vertragingstijden gaan we uit van de globale tijden, dus van de gehele schakeling. Let op dat FF0 vóór en FF1/FF2 ná de skew geplaatst zijn. Bekijken we de flipflops als geheel, dan zien we dat de uitgang van FF0 op zijn vroegst $t_{P(min)}$(FF) na de flank van $clk$ verandert. Vervolgens moet nog de minimale vertragingstijd van de AND-poort gewacht worden. In vergelijking daarmee veranderen de uitgangen van FF1 en FF2 pas later, er is immers sprake van klokskew. Dus pas na één skewtijd én een $t_{P(max)}$(FF) hebben deze uitgangen zeker een nieuwe waarde. Daar moet dan nog de maximale vertragingstijd van de AND-poort bij opgeteld worden. In figuur 9.30 is het geheel getekend.




**Figuur 9.30:** *Timingdiagram voor het bepalen van de vertragingstijden van uitgang* $tc$.

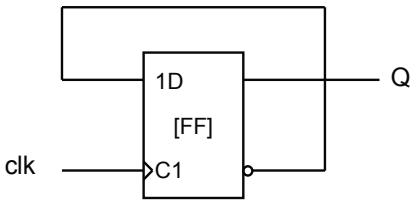
We kunnen nu twee formules opstellen voor het uitrekenen van de minimale en maximale vertragingstijd van uitgang $tc$:

$$t_{P(min)}(\text{tc-to-clk}) = t_{P(min)}(\text{FF}) + t_{P(min)}(\text{D1}) = 6 + 2 = 8 \text{ ns}$$
$$t_{P(max)}(\text{tc-to-clk}) = t_{skew} + t_{P(max)}(\text{FF}) + t_{P(max)}(\text{D1}) = 3 + 8 + 3 = 14 \text{ ns}$$

## 9.10 Opgaven

**9.1.** Van de tweedeler in figuur P9.1 zijn de volgende timingparameters gegeven.

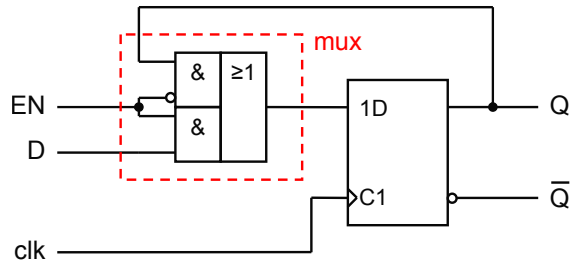
$$t_{su}(\text{FF}) = 15 \text{ ns}$$
$$t_h(\text{FF}) = 5 \text{ ns}$$
$$t_{P(min)}(\text{FF}) = 10 \text{ ns}$$
$$t_{P(max)}(\text{FF}) = 35 \text{ ns}$$




**Figuur P9.1:** *Timingparameters en schema van een tweedeler.*

Bepaal de maximale frequentie waarop dit systeem kan werken. Is betrouwbare dataoverdracht mogelijk?

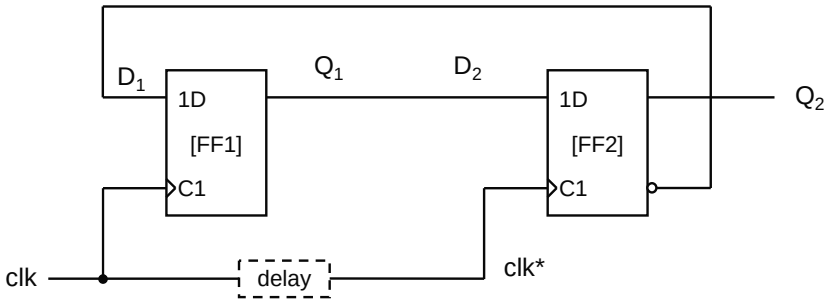
**9.2.** Gegeven is een flipflop met enable in figuur P9.2. Bepaal de setup- en holdtijd voor signaal $D$ t.o.v. van de actieve klokflank. Bepaal de maximale frequentie waarop dit systeem betrouwbare kan werken. Teken tijddiagrammen waarmee de berekeningen kunnen worden gemaakt.




$$t_{su}(\text{FF}) = 14 \text{ ns}$$
$$t_h(\text{FF}) = 4 \text{ ns}$$
$$t_{P(min)}(\text{FF}) = 7 \text{ ns}$$
$$t_{P(max)}(\text{FF}) = 17 \text{ ns}$$
$$t_{P(min)}(\text{logic}) = 8 \text{ ns}$$
$$t_{P(max)}(\text{logic}) = 19 \text{ ns}$$

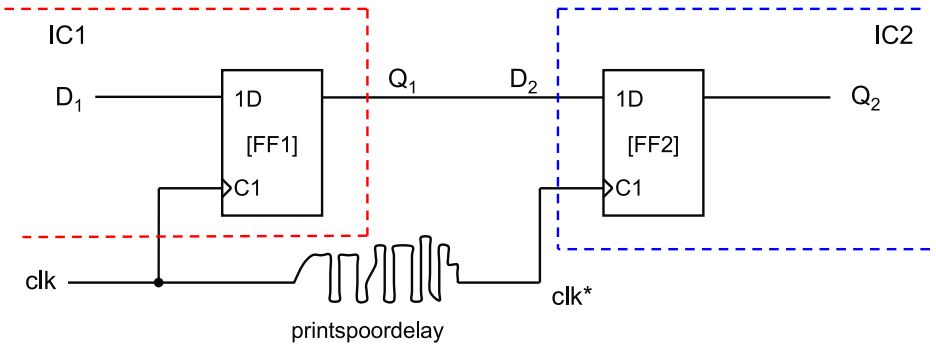
**Figuur P9.2:** *Schema en timing van een D-flipflop met enable.*

**9.3.** Gegeven is het schema in figuur P9.3. Het kloksignaal van FF2 is iets vertraagd t.o.v. de klok van FF1. Druk de maximale frequentie uit in de timingparameters van de flipflops en de clock skew (delay). Analyseer de logische werking van dit systeem.




**Figuur P9.3:** *Twee flipflops met verbindingen en klokskew.*

**9.4.** Gegeven is het schema in figuur P9.4. Het laat dataoverdracht tussen twee flipflops van verschillende ic's zien.




**Figuur P9.4:** *Verbinding tussen verschillende ic's.*

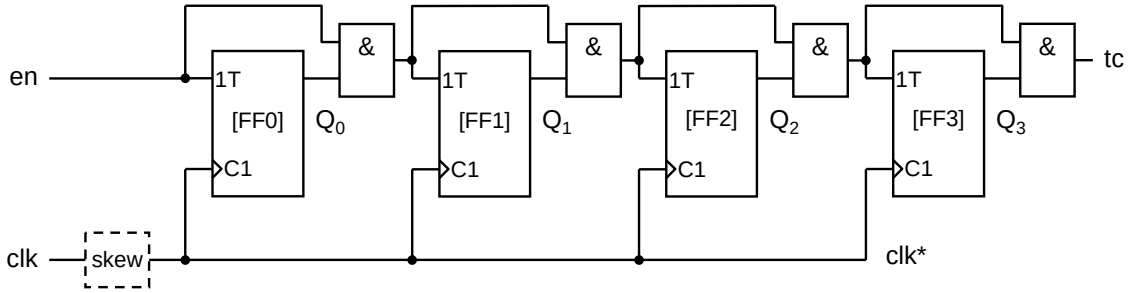
De timing van de flipflops is verschillend:

FF1: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 10/3/7/12$ ns

FF2: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 20/6/10/25$ ns

De skewtijd bedraagt 2 ns. Is betrouwbare dataoverdracht mogelijk?

**9.5.** Gegeven is de onderstaande 4-bits teller.




**Figuur P9.5:** *4-bits teller opgebouwd uit 1-bit tellersecties.*

Van de componenten zijn de volgende timingparameters gegeven:

Flipflops: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 2/1/4/7$ ns

AND: $t_{P(min)}/t_{P(max)} = 3/5$ ns

Skew: $t_P = 4$ ns

In dit systeem zijn meerdere paden te ontdekken waarlangs data wordt getransporteerd.

a) Geef alle paden waarlangs data van flipflop naar flipflop wordt getransporteerd.

Van dit systeem moet de maximale frequentie worden berekend.

b) Teken een tijddiagram met alleen relevante timingparameters waarmee de maximale frequentie berekend moet worden.

c) Bereken de maximale frequentie waarop dit systeem nog betrouwbaar werkt.

Het signaal *en* (enable) wordt vertraagd aangeboden aan de ingang van de flipflops. Dat heeft invloed op de setup- en holdtijden van signaal *en* t.o.v. het kloksignaal $clk$.

d) Teken een tijddiagram met alleen relevante timingparameters waarmee de setuptijd en holdtijd van *en* (enable) t.o.v. de actieve flank van $clk$ berekend kan worden.

e) Bereken de setuptijd en holdtijd van *en* (enable) t.o.v. de actieve flank van $clk$.

De uitgang $tc$ geeft aan of de teller op de maximale stand staat. Na het passeren van de actieve klokflank duurt het even voordat de nieuwe waarde op de uitgang beschikbaar is.

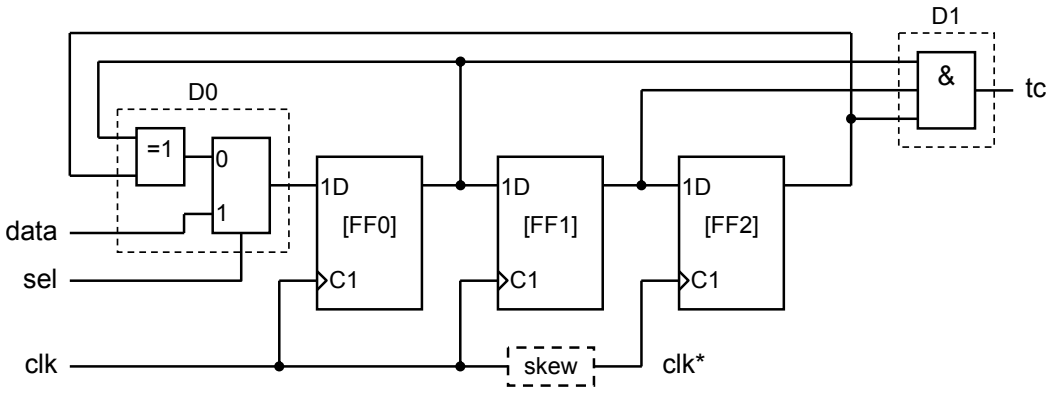
f) Teken een tijddiagram met alleen relevante timingparameters waarmee de minimale en maximale vertragingstijden van uitgang $tc$ t.o.v. de actieve flank van $clk$ berekend kan worden.

g) Bepaal de minimale en maximale vertragingstijden van de uitgang $tc$ t.o.v. de actieve klokflank.

De opbouw van de teller is zeer elegant en makkelijk uitbreidbaar maar hoe zit het nou met de timing.

h) Beschrijf kort wat de invloed is van het aantal telsecties op de timing, bv. als het aantal secties wordt uitgebreid.

**9.6.** Gegeven is de schakeling in figuur P9.6. Hierin zijn drie D-flipflops opgenomen (FF0, FF1 en FF2), twee stukken combinatoriek (D0, D1) en één klokvertraging (skew). D0 is combinatoriek voor de ingang van FF0 en D1 is combinatoriek na de uitgangen van FF0, FF1 en FF2. FF2 krijgt een vertraagd kloksignaal aangeboden (skew).




**Figuur P9.6:** *Synchrone flipflop-schakeling*

Van de bouwstenen zijn de volgende timingparameters gegeven:

D-flipflops: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 2/1/6/8$ ns

AND: $t_{P(min)}/t_{P(max)} = 2/3$ ns

EXOR: $t_{P(min)}/t_{P(max)} = 3/5$ ns

Mux: $t_{P(min)}/t_{P(max)} = 4/6$ ns

Skew: $t_P = 3$ ns

Van de skew is maar één vertragingstijd gegeven, er wordt geen onderscheid gemaakt tussen de minimale en maximale vertragingstijd. Het betreft hier namelijk clock skew.

Merk op dat er in dit systeem **vier** paden zijn tussen de flipflops waarlangs data wordt getransporteerd.

a) Geef aan welke paden er tussen de flipflops zijn waarlangs data wordt getransporteerd. Geef hierbij duidelijk aan welke stukken combinatorische

logica wordt gepasseerd.

Van het systeem moet de maximale frequentie worden berekend. Na onderzoek blijkt het pad FF2 → D0 → FF0 de grootste minimale periodetijd op te leveren.

b) Teken een tijddiagram met alleen relevante timingparameters waarmee de maximale frequentie berekend moet worden.

c) Bereken de maximale frequentie waarop dit systeem nog betrouwbaar werkt.

FF2 heeft last van klokskew, het kloksignaal komt vertraagd aan op de klokingang van FF2. Dat heeft invloed op de holdtijd van de dataingang van FF2.

d) Teken een tijddiagram met alleen relevante timingparameters waarmee de *hold slack* van de dataingang van FF2 berekend kan worden.

e) Bereken de *hold slack* van de dataingang van FF2.

# 10

# Synchronisatie, metastabiliteit en reset

In dit hoofdstuk behandelen we drie onderwerpen waar we in een praktisch digitaal systeem te maken hebben: synchronisatie, metastabiliteit en reset-implementatie.

Synchronisatie houdt in dat elk extern aangeboden datasignaal eerst wordt gesynchroniseerd met de systeemklok alvorens het wordt verwerkt in het systeem. Dit is een noodzakelijke actie om ervoor te zorgen dat het systeem betrouwbaar functioneert. We laten zien dat een flipflop voor dit doel gebruikt kan worden. In veel gevallen wordt dubbele synchronisatie toegepast om de betrouwbaarheid te verhogen. Synchronisatie dient nog een doel: externe signalen die eerst door combinatorische logica gaan met verschil in vertragingstijden kunnen de werking van een systeem negatief beïnvloeden. Voorts zijn er systemen waarin meerdere kloksignalen worden gebruikt (de zogenoemde clock domains). Hier is synchronisatie tussen de domeinen een vereiste anders komt de betrouwbaarheid van de gehele schakeling in het geding.

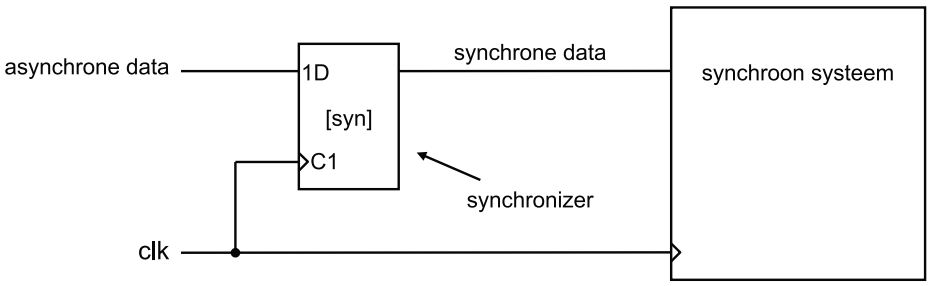
Voor het synchroon maken van een extern datasignaal wordt een flipflop gebruikt. Het kan gebeuren dat het externe signaal verandert tijdens het setuptijd-holdtijd interval. De uitgang van de flipflop kan dan tijdelijk metastabiel worden. Metastabiliteit houdt in dat de (uitgang van een) flipflop tijdelijk een waarde heeft tussen een logische 0 en logische 1 als gevolg van signaalwisselingen tijdens het setuptijd-holdtijd interval. We bespreken eerst wat metastabiliteit is en hoe het wordt veroorzaakt. We laten zien dat we een uitspraak kunnen doen over de robuustheid van een systeem om zich te wapenen tegen metastabiliteit. We trekken de conclusie dat metastabiliteit niet te voorkomen is maar dat de kans dat een systeem de fout in gaat minimaal kan worden gehouden.

Elk synchroon digitaal systeem moet uitgerust zijn met een resetfaciliteit die ervoor zorgt dat de geheugenelementen in een bekende stand worden gedwongen. Er zijn twee varianten: asynchroon en synchroon met het kloksignaal. We laten zien dat een beide varianten voordelen en nadelen hebben. Voorts bespreken we een variant die gestoeld is op de asynchrone en synchrone resetfunctionaliteit: de reset-synchronizer. In een systeem met meerdere kloksignalen moeten verschillende reset-synchronizers worden gebruikt.

## 10.1 Synchronisatie van ingangssignalen

Een complex digitaal systeem bestaat uit combinatorische en sequentiële logica (poorten en flipflops). Binnen het systeem worden alle acties uitgevoerd op de klokflank. Het systeem heet dan *kloksynchroon*. Het digitale systeem heeft ingangen en uitgangen die gekoppeld zijn aan andere systemen, anders kunnen we er niet veel mee doen. De ingangen zijn echter asynchroon van aard; veranderingen op de ingangen zijn niet synchroon met de klokflank. Dat levert allerlei problemen als de ingangssignalen veranderen rond de actieve klokflank van het ontvangende systeem. De ingangen zijn eenvoudig kloksynchroon te maken met behulp van een *synchronizer*.

In figuur 10.1 is een synchroon systeem te zien dat voorzien is van een synchronizer. De synchronizer bemonstert op de opgaande flank van de klok de asynchroon binnenkomende data en zorgt voor een geldig synchroon signaal in de volgende klokcyclus.




**Figuur 10.1:** *Digitaal systeem met een synchronizer.*

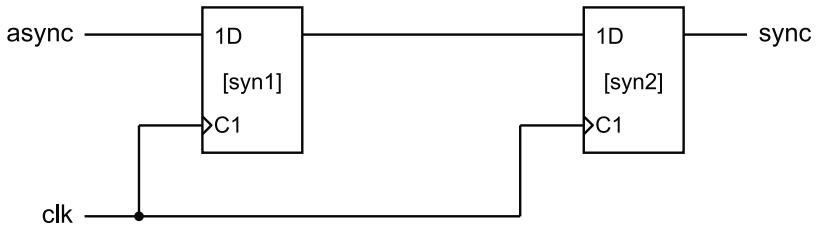
We zien dat de synchronizer in feite een gewone D-flipflop is. De D-flipflop doet zijn werk goed als de data op de *D*-ingang tijdens de setuptijd-holdtijd interval niet veranderd. Maar de asynchrone signalen veranderen niet synchroon met de klokflank. Dat houdt in dat een verandering op de *D*-ingang binnen het setuptijd-holdtijdgebied kan vallen. De synchronizer kan dan niet betrouwbaar data overnemen (inklokken).

De uitgang van de synchronizer kan verschillende waarden aannemen:

- De uitgang neemt een nieuwe waarde aan. Dit kan een logische 0 of 1 zijn. Eventueel is er eerst een (kleine) verandering van het uitgangssignaal waarneembaar maar na de maximale vertragingstijd is de waarde weer beschikbaar.
- De uitgang gaat tijdelijk oscilleren. Dit is het gevolg van de interne werking van de flipflop. Uiteindelijk zal de uitgang een van de twee logische waarden aannemen.
- De uitgangswaarde ligt tijdelijk tussen logisch 0 en 1, dus een ongeldige logische waarde. De uitgang is dan tijdelijk *metastabiel*. Na een tijd zal de uitgang een van de twee logische waarden aannemen.

Het begrip metastabiliteit zullen we in de paragraaf 10.4 behandelen. Een beproefde methode om een ingangssignaal te synchroniseren is het gebruik van een *synchronizer chain*, zie figuur 10.2. Hierbij worden meerdere synchronizers in serie geschakeld. Het resultaat is in feite een schuifregister. Meestal zijn twee in serie geschakelde synchronizers genoeg om betrouwbaarheid van het systeem voldoende te garanderen. De eerste synchronizer

kan metastabiel worden maar de kans dat de tweede synchronizer metastabiel wordt is zeer klein (maar het kan wel!).




**Figuur 10.2:** *Dubbele synchronisatie.*

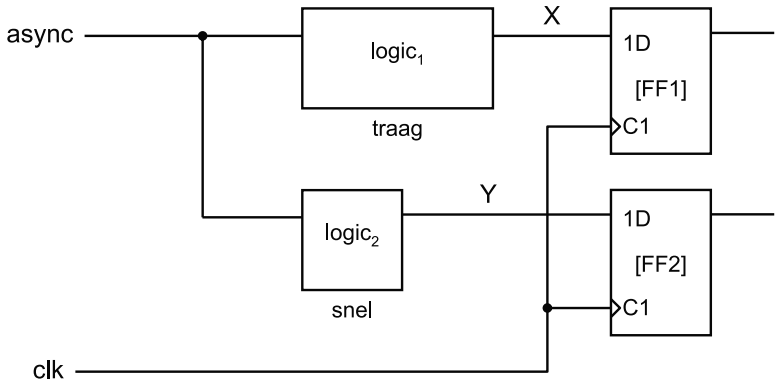
Bij het toepassen van een synchronizer chain moet erop gelet worden dat de synchronizers bij elkaar op het ic worden geplaatst, anders heeft het last van klokskew en dataskew. Ontwikkelsoftware kan een synchronizer chain herkennen en maatregelen treffen voor correct gebruik.

In listing 10.1 is een beschrijving gegeven van een synchronizer chain met een generieke parameter *n* die aangeeft uit hoeveel flipflops de synchronizer chain bestaat. In de code zijn dat er twee, maar het kan eenvoudig worden uitgebreid middels een **`generic map`** statement in de hogere hiërachie.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity synchronizer_chain is
5      generic (n    : positive := 2);
6      port (clk    : in std_logic;
7            areset : in std_logic;
8            d      : in std_logic;
9            q      : out std_logic
10          );
11 end entity synchronizer_chain;
12
13 architecture rtl of synchronizer_chain is
14 signal sync_chain : std_logic_vector (1 to n);
15 begin
16
17     process (clk, areset) is
18     begin
19         if areset = '1' then
20             sync_chain <= (others => '0');
21         elsif rising_edge(clk) then
22             sync_chain <= d & sync_chain(1 to n-1);
23         end if;
24     end process;
25
26     q <= sync_chain(n);
27
28 end architecture rtl;
```

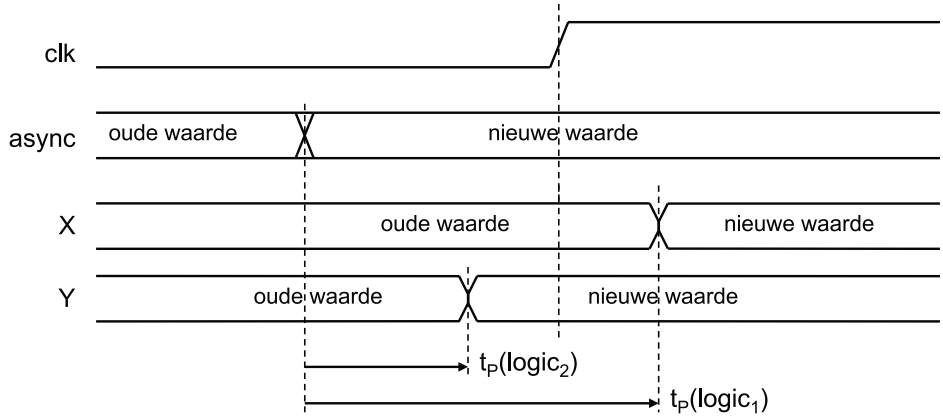
**Listing 10.1:** *VHDL-code van een synchronizer chain.*

Synchronisatie van ingangssignalen dient nog een ander doel: het zorgt ervoor dat de flipflops in een schakeling allemaal dezelfde ingangswaarde zien. We demonstreren dit aan de hand van de schakeling in figuur 10.3.




**Figuur 10.3:** *Meerdere flipflops met voorzetlogica.*

In de schakeling zijn twee flipflops afhankelijk van de waarde van het signaal $async$. Voor de flipflops is combinatorische logica geplaatst waarvan de vertragingstijden niet gelijk zijn. De logica bij $logic_1$ is traag en de logica bij $logic_2$ is snel. We maken uit de figuur dus op dat $t_{P(max)}(\text{logic}_1) > t_{P(max)}(\text{logic}_2)$. Stel dat het ingangssignaal $async$ op een bepaald moment een waarde aanneemt, dan kost het nog enige tijd voordat de juiste waarden op signalen $X$ en $Y$ stabiel zijn. Als logica $logic_1$ nu ná de klokflank en logica $logic_2$ vóór de klokflank de juiste waarde afgeven, klokken de flipflops als geheel dus niet de bedoelde waarden in. Dit wordt weergegeven in het timingdiagram in figuur 10.4. We gaan er trouwens vanuit dat de setuptijden en holdtijden gerespecteerd worden.

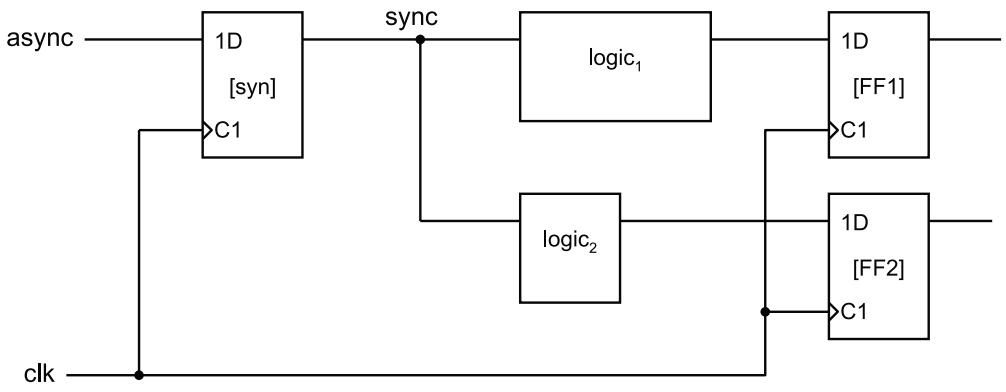



**Figuur 10.4:** *Timing van meerdere flipflops met voorzetlogica.*

De oplossing is simpel: gebruik een synchronizer. Dit is te zien in figuur 10.5. De synchronizer zorgt ervoor dat de nieuw ingeklokte waarde na $t_{P(max)}(\text{synch})$ beschikbaar is. We kunnen over de timing dan het volgende opstellen:

$$T_{min} = t_{P(max)}(\text{sync}) + t_{P(max)}(\text{logic}_1) + t_{su}(\text{FF}) \tag{10.1}$$

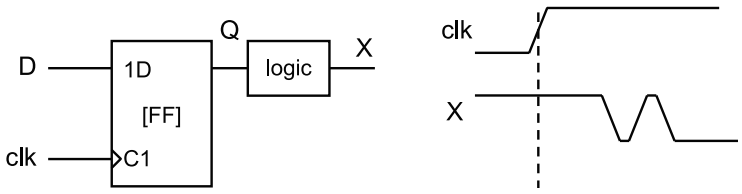
De stelregel is dus: externe asynchrone ingangen moeten altijd gesynchroniseerd worden met de systeemklok.




**Figuur 10.5:** *Synchronisatie bij meerdere flipflops met voorzetlogica.*

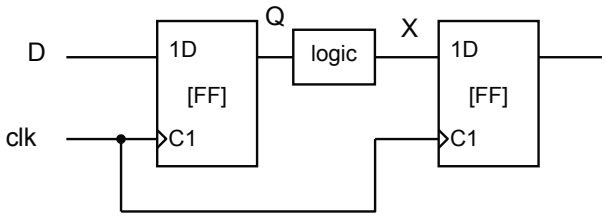
## 10.2 Synchronisatie van uitgangssignalen

De uitgangen van een synchroon systeem veranderen na de klokflank en zijn synchroon met de klok. De uitgangslogica kan echter wel *multi transition* zijn, dat inhoudt dat er meerdere signaalwisselingen volgen voordat de definitieve waarde wordt aangenomen. Zie figuur 10.6. Dit kan problemen opleveren voor het achterliggende systeem.




**Figuur 10.6:** *Uitgangen kunnen meerder transities hebben.*

Dit kan eenvoudig worden opgelost door een uitgangsflipflop te gebruiken. We noemen deze uitgangen *registered outputs* (ook wel *clocked outputs* of *synchronous outputs* genoemd). Voorwaarde is wel dat de uitgangsflipflop single transition moet zijn. Bijna elke moderne flipflop is single transition. Een nadeel van deze methode is dat de uitgangswaarde één klokpuls later beschikbaar is. Zie figuur 10.7.

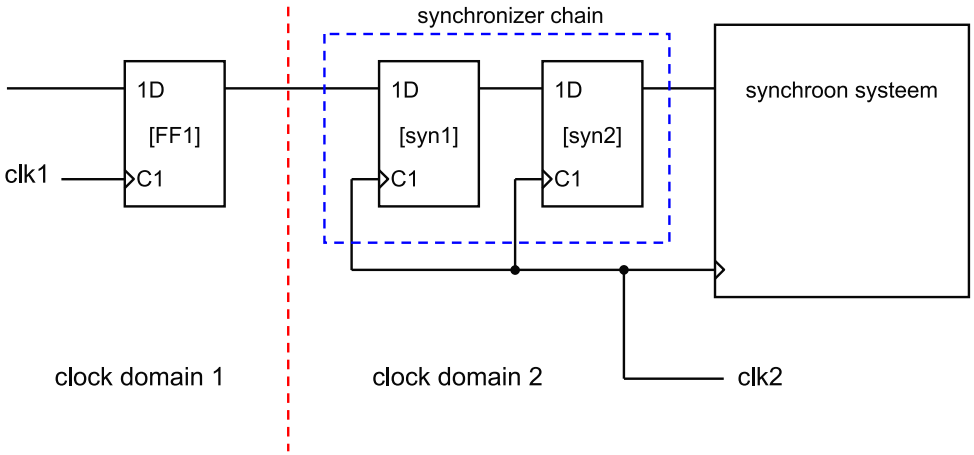



**Figuur 10.7:** *Een schakeling met een uitgangsflipflop.*

## 10.3 Synchronisatie tussen clock domains

In grote systemen zijn soms meerdere kloksignalen aanwezig die geen relatie met elkaar hebben. De kloksignalen zijn asynchroon ten opzichte van elkaar. We spreken dan over een digitaal systeem met meerdere *clock domains*. Synchronisatie is dan noodzakelijk om

data betrouwbaar van het ene naar het andere clock domain te transporteren. Synchronisatie wordt gerealiseerd met behulp van een synchronizer chain. Vanwege de veelal hoge frequenties is dubbele synchronisatie een vereiste. Dit is te zien in figuur 10.8. Een enkele gevallen is zelfs drievoudige synchronisatie nodig.




**Figuur 10.8:** *Synchronisatie tussen clock domains.*

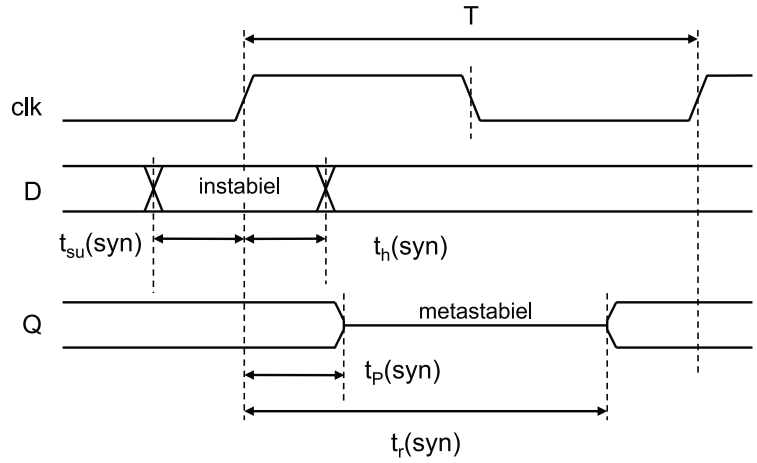
*Clock domain crossing* (CDC) wordt veel gebruikt op grote ic's. Een voorbeeld hiervan is een multi-core processor waarbij elke kern een eigen kloksignaal heeft. De snelheid van de klokken kan worden gevarieerd om het energieverbruik te minimaliseren. Als een kern niet wordt gebruikt, wordt de klok zelfs helemaal afgeschakeld. Een ander voorbeeld is een microcontroller waar naast een processorkern ook nog allerlei randapparatuur is geplaatst. Te denken valt aan een Ethernet-controller, een seriële controller, timers, counters en een analoog-digitaal converter. De randapparatuur wordt meestal gevoed met een eigen klok.

## 10.4 Metastabiliteit

We zijn er tot nu toe altijd vanuit gegaan dat de ingangssignalen stabiel zijn tijdens het setuptijd-holdtijd interval. We kunnen echter aannemen dat dat niet altijd het geval is. Neem als voorbeeld een drukknop die door een gebruiker bediend wordt. Het is niet te verwachten dat de gebruiker op de juiste momenten de drukknop indrukt en zorgt voor een stabiele waarde in het setuptijd-holdtijd interval. Mocht het ingangssignaal tijdens dit interval veranderen dan kan de uitgang van de synchronizer (in feite elke flipflop) metastabiel worden; de uitgangwaarde zweeft een tijdje tussen een logische 0 en 1. De kans dat een uitgang metastabiel is, wordt kleiner met de voortschrijdende tijd. Als we maar voldoende lang wachten, dan zal de uitgang wel een stabiele waarde hebben aangenomen. De vraag is alleen hoelang we moeten wachten. Zeker is wel dat de maximale vertragingstijd door het metastabiel zijn wordt overschreden. Het hangt helemaal van het ontwerp af hoeveel invloed de metastabiliteit op het systeem heeft.

In figuur 10.9 is de timing te zien van een flipflop die metastabiel is. In het setuptijd-holdtijd interval (ook wel het beslissingsvenster genoemd) verandert het signaal op de *D*-ingang. Als gevolg daarvan wordt de uitgang na enige tijd metastabiel. Na een zekere

tijd $t_r$, de *resolution time* genoemd, neemt de uitgang een van de twee stabiele waarden aan.




**Figuur 10.9:** *Timing bij metastabiliteit.*

Metastabiliteit wordt gekarakteriseerd door de *Mean Time Between Failures* (MTBF). MTBF is een statistische meting van betrouwbaarheid. Het geeft de gemiddelde tijd tussen twee fouten aan. Merk op dat het een *gemiddelde* tijd tussen twee fouten is. Het kan zijn dat er direct een fout optreedt of dat twee fouten elkaar direct opvolgen.

Theoretisch en experimenteel onderzoek wijst uit dat de MTBF van een synchronizer wordt beschreven door de formule:

$$\text{MTBF}(t_r) = \frac{1}{f_{clk} \cdot f_{data} \cdot T_0} \cdot e^{t_r/\tau} \tag{10.2}$$

Hier geeft MTBF($t_r$) de gemiddelde tijd aan tussen twee synchronizer-fouten aan, wanneer de fout optreedt als metastabiliteit de tijd $t_r$ (*resolution time*) aanhoudt na de klokflank waarbij geldt dat $t_r \geq t_{P(max)}$(FF). De MTBF is afhankelijk van de klokfrequentie van het systeem ($f_{clk}$), de frequentie van het asynchroon binnenkomende signaal ($f_{data}$) en twee constanten $T_0$ en $\tau$ die afhangen van de elektrische eigenschappen van de flipflop. Opgemerkt moet worden dat sommige fabrikanten niet uitgaan van de frequentie van het asynchroon binnenkomende signaal maar van het aantal signaalwisselingen. Dat is dus twee keer de frequentie.

In tabel 10.1 is van een aantal componenten de metastabiliteitparameters te zien. Deze zijn te vinden in [23, 24, 25, 26, 27]. De parameters zijn gemiddelde waarden. De waarden variëren namelijk met de voedingsspanning, de omgevingstemperatuur, de belasting en het productieproces. Zelfs tussen verschillende fabrikanten die hetzelfde ic produceren, is er verschil in parameterwaarden. Het kan dus zijn dat de werkelijk MTBF beter of slechter is dan de berekende waarde.

Stel dat we een 74LS74 D-flipflop van fabrikant TI als synchronizer gebruiken. Hiervan zijn $\tau = 1{,}35$ ns en $T_0 = 4{,}8 \cdot 10^{-3}$ s gegeven. De systeemfrequentie is 20 MHz (periodetijd is 50 ns) en de frequentie van het binnenkomende signaal is 100 kHz. We kunnen

**Tabel 10.1:** *Metastabiliteitparameters van enkele componenten.*

| Auteur | Component | $\tau$ (ns) | $T_0$ (s) |
|---|---|---|---|
| Chaney | 74LS74 | 1,5 | 0,4 |
| TI | 74LS74 | 1,35 | $4{,}8 \cdot 10^{-3}$ |
| Chaney | 7474 | 1,6 | $3 \cdot 10^{-3}$ |
| Xilinx | 7300 | 0,29 | $1{,}0 \cdot 10^{-15}$ |
| Xilinx | 9500 | 0,63 | $9{,}5 \cdot 10^{-18}$ |
| NXP | 74F50729 | 0,115 | $1{,}3 \cdot 10^{10}$ |
| TI | 74ABT7819 | 0,30 | $7 \cdot 10^{-12}$ |

nu berekenen wat de gemiddelde tijd tussen twee fouten is als de uitgang van de D-flipflop 30 ns tijd heeft om uit zijn metastabiele toestand te komen:

$$\text{MTBF(30 ns)} = \frac{e^{30/1,35}}{20 \cdot 10^6 \cdot 10^5 \cdot 4{,}8 \cdot 10^{-3}} = 0{,}47 \text{ s} \tag{10.3}$$

De gemiddelde tijd tussen twee fouten is 0,47 seconden. Gebruiken we voor hetzelfde systeem echter een 74F50729 D-flipflop dan wordt de gemiddelde tijd tussen twee fouten:

$$\text{MTBF(30 ns)} = \frac{e^{30/0,115}}{20 \cdot 10^6 \cdot 10^5 \cdot 1{,}3 \cdot 10^{10}} = 7{,}6 \cdot 10^{90} \text{ s} \tag{10.4}$$

Dat is nog eens een lange tijd. Een goede benadering van een MTBF van de dubbele synchronisatie wordt gegeven door:

$$\begin{aligned} \text{MTBF(double)} &= \frac{e^{t_{r1}/\tau}}{f_{clk} \cdot f_{data} \cdot T_0} \cdot \frac{e^{t_{r2}/\tau}}{f_{clk} \cdot T_0} \\ &= \text{MTBF}(t_{r1}) \cdot \frac{e^{t_{r2}/\tau}}{f_{clk} \cdot T_0} \\ &= \frac{e^{t_{r2}/\tau}}{\frac{1}{\text{MTBF}(t_{r1})} \cdot f_{clk} \cdot T_0} \end{aligned} \tag{10.5}$$

waarbij $t_{r1}$ en $t_{r2}$ de resolution times zijn van de eerste en tweede synchronizer. We gaan hierbij uit van identieke synchronizers dat op één ic vaak het geval is. We kunnen dus stellen dat de frequentie van het inkomende datasignaal van de tweede synchronizer de reciproke is van de MTBF van de eerste synchronizer.

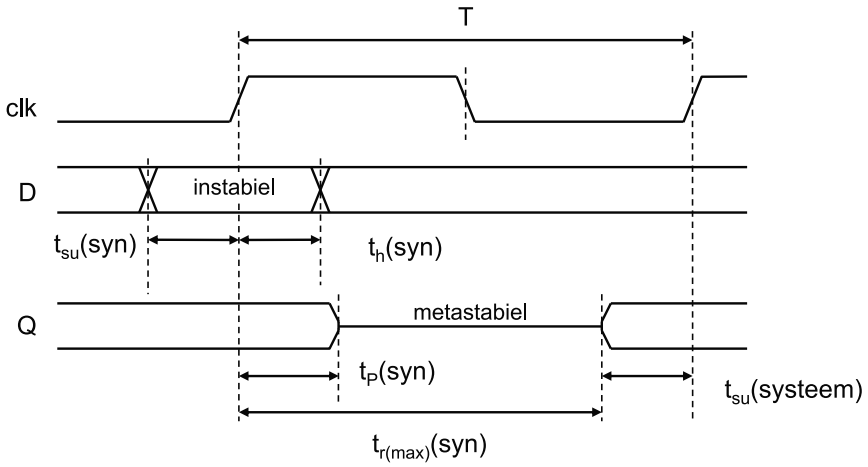
In het geval dat er meerdere synchronizer chains in een ontwerp voorkomen, is de kans dat de achterliggende schakeling te maken krijgt met metastabiliteit groter dan bij één chain. We kunnen het geheel zien als een *parallelschakeling* van de afzonderlijke sychronizer chains. De MTBF van het gehele systeem wordt berekend met:

$$\frac{1}{\text{MTBF}_{systeem}} = \frac{1}{\text{MTBF}_1} + \ldots + \frac{1}{\text{MTBF}_k} \tag{10.6}$$

Hierbij is $k$ het aantal synchronizer chains van het systeem [28]. Op één ic zijn de timingparameters van de flipflops identiek zodat we kunnen stellen dat:

$$\text{MTBF}_{systeem} = \frac{\text{MTBF}_{chain}}{k} \tag{10.7}$$

We kunnen nog iets zeggen over de resolution time van de synchronizer. Als we uitgaan van de systeemfrequentie en de setuptijd van het achterliggende synchrone systeem, dan kunnen we de maximale resolution time uitrekenen. Zie hiervoor het timingdiagram in figuur 10.10.




**Figuur 10.10:** *Timing van een systeem met synchronizer.*

Uit de figuur is eenvoudig op te maken dat geldt:

$$t_{r(max)}(\text{syn}) = T - t_{su}(\text{systeem}) \tag{10.8}$$

Het wil niet zeggen dat de synchronizer na deze tijd niet meer metastabiel is. Het geeft alleen aan hoeveel tijd de synchronizer heeft om uit zijn metastabiele toestand te komen zodat de achterliggende schakeling correct functioneert.

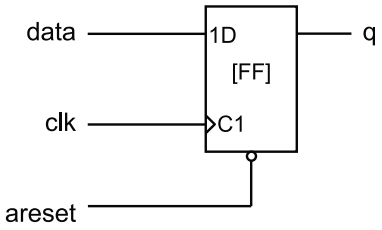
Als laatste nog een opmerking: metastabiliteit kan nooit helemaal vermeden worden. Er is altijd een kans dat een flipflop metastabiel wordt. We kunnen er alleen voor zorgen dat die kans heel klein wordt.

Er is veel geschreven over de problematiek rond metastabiliteit. Meer informatie kan gevonden worden in [29]. Een goede afleiding voor de formule voor de MTBF wordt beschreven in [30, 31, 32].

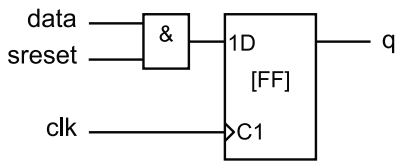
## 10.5 Reset-implementatie

Een reset is een noodzakelijke faciliteit van elk digitaal systeem met geheugenwerking. Het zorgt ervoor dat de flipflops in een bepaalde (toe-)stand worden gedwongen. Bij het starten van een systeem (het opkomen van de voedingsspanning) is namelijk niet bekend wat de toestanden van de flipflops zijn.

Er zijn twee soorten resets: asynchroon en synchroon. Bij een asynchrone reset komt het resetsignaal direct op de flipflops binnen als een apart stuursignaal. Deze vorm werkt buiten de klok om. Bij een synchrone reset komt het resetsignaal binnen als een “gewoon” datasignaal en kan verwerkt worden in de voorzetlogica van de flipflops. Deze vorm is synchroon met de klok. Beide vormen zijn te zien in figuur 10.11.




**(a)** *Asynchrone reset op een flipflop.*

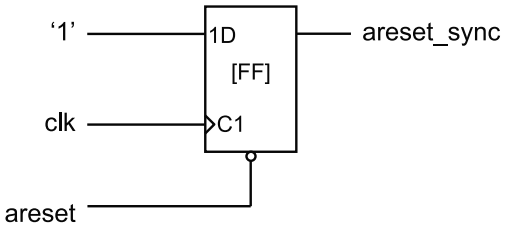



**(b)** *Synchrone reset op een flipflop.*

**Figuur 10.11:** *Asynchrone en synchrone reset.*

Het ligt voor de hand om een asynchrone reset te gebruiken. De flipflops worden bij activatie direct in een bekende beginstand gezet (het kost natuurlijk wel enige tijd voordat de flipflop de activatie “ziet”). Maar als het resetsignaal gedeactiveerd wordt tijdens de klokflank kunnen er problemen ontstaan. De flipflop kan tijdelijk metastabiel worden. Het recept is dus eenvoudig: asynchrone activatie en synchrone deactivatie.

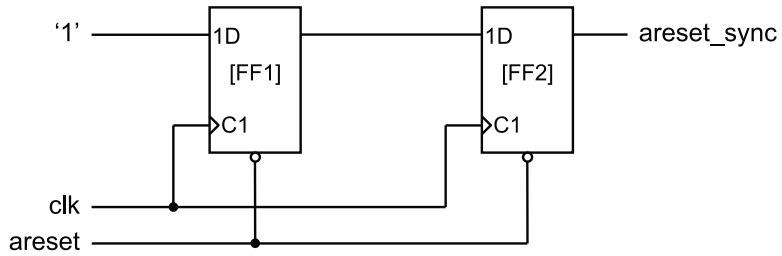
Een eerste poging is te zien in figuur 10.12. Het resetsignaal $areset$ is verbonden met de asynchrone resetingang van de flipflop. De dataingang van de flipflop is verbonden met een logische 1. Bij activatie wordt de flipflop asynchroon gereset, de uitgangswaarde van de flipflop (signaal $areset\_sync$) is dan logisch 0. Bij deactivatie wordt de flipflop op de klokflank geladen met een logische 1. We gaan ervan uit dat het achterliggende systeem een actief lage reset heeft. Er kleeft wel een nadeel aan deze schakeling. De flipflop kan metastabiel worden als de reset gedeactiveerd wordt binnen het setuptijd-holdtijd interval.




**Figuur 10.12:** *Poging tot synchroniseren van het resetsignaal.*

Een tweede poging is te zien in figuur 10.13. Direct na activatie worden beide flipflops asynchroon geladen met logische 0. Na deactivatie wordt de eerste flipflop op de klokflank geladen met logische 1, de tweede flipflop wordt geladen met een logische 0 van de uitgang van de eerste flipflop is 0. De eerste flipflop kan metastabiel worden maar de tweede niet, want zowel de ingang als de uitgang van de tweede flipflop zijn logisch 0 (merk op dat metastabiliteit pas ná de klokflank optreedt). Na nog een klokflank is ook de tweede flipflop geladen met een logische 1. Een diepgaande uitleg wordt gegeven in [33]. Helaas kan de tweede flipflop metastabiel worden als gevolg van metastabiliteit van de

eerste flipflop [34]. De kans daarop is echter zeer klein. Verder moet er minstens twee klokflanken gewacht worden voordat de achterliggende schakeling uit de resettoestand komt. De schakeling in figuur 10.13 wordt een *reset synchronizer* genoemd.




**Figuur 10.13:** *Correcte uitvoering van synchronisatie van het resetsignaal.*

In listing 10.2 is de VHDL-code gegeven van de reset synchronizer uit figuur 10.13.

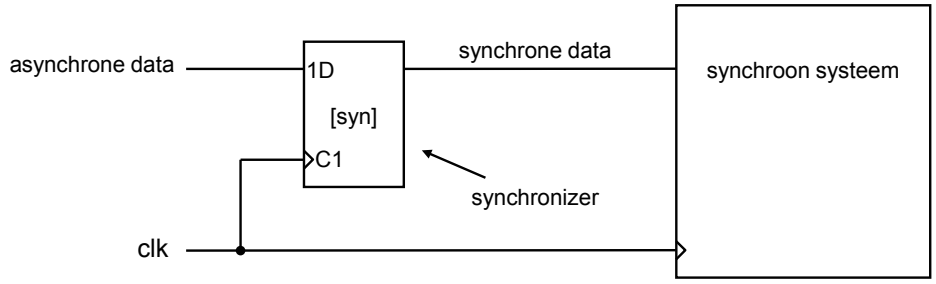
```
library ieee;
use ieee.std_logic_1164.all;

entity reset_synchronizer is
    port (clk: in std_logic;
          areset: in std_logic;
          areset_sync: out std_logic
         );
end entity reset_synchronizer;

architecture rtl of reset_synchronizer is
signal ff1, ff2 : std_logic;
begin

    process (clk, areset) is
    begin
        if areset = '0' then
            ff1 <= '0';
            ff2 <= '0';
        elsif rising_edge(clk) then
            ff1 <= '1';
            ff2 <= ff1;
        end if;
    end process;
    areset_sync <= ff2;

end architecture rtl;
```

**Listing 10.2:** *VHDL-code van de reset-synchronizer.*

## 10.6 Opgaven

**10.1.** Gegeven is het synchrone systeem in figuur P10.1 met een frequentie van 20 MHz. Data komt asynchroon binnen met 100 kHz.

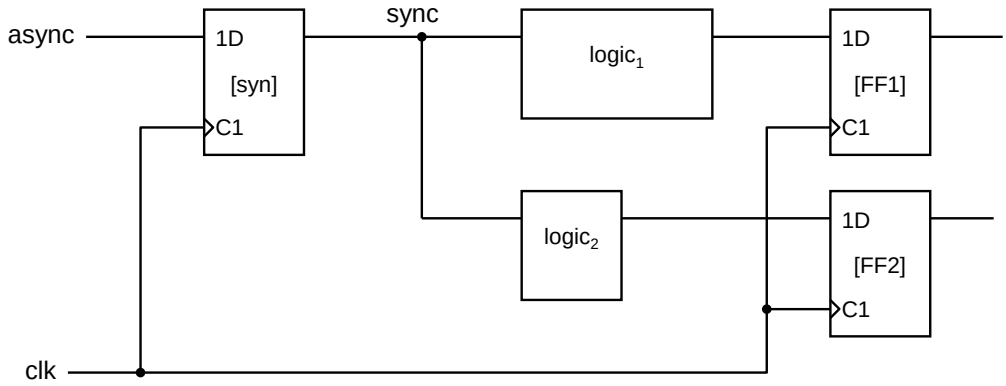
Van de synchronizer 74F50109 is gegeven: $\tau = 0{,}135$ ns, $T_0 = 9{,}8 \cdot 10^6$ s




**Figuur P10.1:** *Een synchronisatiesysteem*

De setuptijd van het achterliggende systeem is 20 ns. Bereken de MTBF van dit systeem.

**10.2.** Gegeven is het systeem van in figuur P10.2.




**Figuur P10.2:** *Een synchronisatiesysteem*

Van flipflops FF1 en FF2 is gegeven $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 5/2/4/10$ ns

Van de synchronizer is gegeven $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 2/0/3/5$ ns, $\tau = 0{,}135$ ns, $T_0 = 9{,}8 \cdot 10^6$ s.

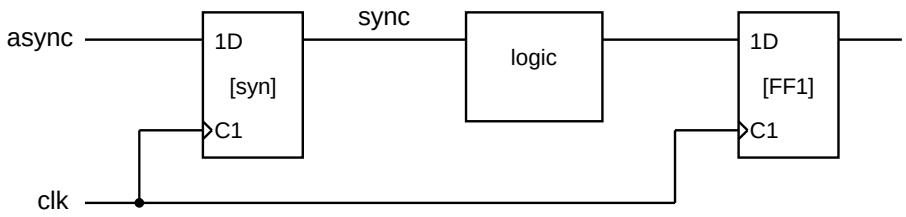
Van logica$_1$ is gegeven $t_{P(min)}/t_{P(max)} = 10/20$ ns.

Van logica$_2$ is gegeven $t_{P(min)}/t_{P(max)} = 5/12$ ns.

De systeemfrequentie is 25 MHz. De frequentie van de asynchroon binnenkomende data bedraagt 2 kHz.

Bepaal de MTBF van dit systeem.

**10.3.** Gegeven is de schakeling in figuur P10.3.




**Figuur P10.3:** *Synchrone schakeling met synchronizer.*

Van flipflop FF1 is gegeven: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 5/2/4/10$ ns
Van de synchronizer is gegeven: $\tau = 0{,}15$ ns, $T_0 = 9{,}8 \cdot 10^6$ s
Van logica is gegeven: $t_{P(min)}/t_{P(max)} = 7/10$ ns

De systeemfrequentie is 40 MHz. Een ontwerper wil dat bij dit systeem de gemiddelde tijd tussen twee fouten 1 jaar (1 jaar = 365 dagen) bedraagt. Bereken de maximaal toegestane frequentie van het asynchroon binnenkomende signaal om aan de eis van de ontwerper te kunnen voldoen.

# Bibliografie

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


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

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

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

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

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

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

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

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

[9] P.J. Ashenden. *The Student's Guide to VHDL*. Morgan Kaufmann. Elsevier Science Limited, 2008. ISBN: 978-1-55860-865-8 (blz. 1).

[10] "IEEE Standard VHDL Language Reference Manual - Redline". In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) - Redline* (jan 2009) (blz. 1, 18).

[11] S. Sjoholm en L. Lindh. *VHDL for Designers*. Prentice Hall, 1997. ISBN: 9780134734149 (blz. 1).

[12] D.D. Gajski en R. Kuhn. "Guest Editors' Introduction: New VLSI Tools". In: *Computer* 16.12 (nov 1983), p. 11–14 (blz. 2).

[13] "IEEE Standard VHDL Synthesis Packages". In: *IEEE Std 1076.3-1997* (jun 1997) (blz. 50).

[14] A.P. Godse en D.A. Godse. *Digital Electronics And Logic Design*. Technical Publications, 2009, p. 8-12 –8-13. ISBN: 9788184316698 (blz. 64).

[15] Texas Instruments. *SNx4HC595 8-Bit Shift Registers With 3-State Output Registers*. Sep 2015. URL: https://www.ti.com/lit/ds/symlink/sn74hc595.pdf (bezocht op 30-12-2018) (blz. 67).

[16] TIA. *TIA-232-F, Interface Between Data Terminal Equipment and Data Circuit- Terminating Equipment Employing Serial Binary Data Interchange*. Telecommunications Industry Association, dec 2012 (blz. 69).

[17] Maxim. *MAX220–MAX249: +5V-Powered, Multichannel RS-232 Drivers/Receivers*. Jan 2015. URL: https://datasheets.maximintegrated.com/en/ds/MAX220-MAX249.pdf (bezocht op 20-12-2018) (blz. 70).

[18] D. Self. *Audio Power Amplifier Design*. Taylor & Francis, 2013. ISBN: 9781136123825 (blz. 90).

[19] R.S. Sandige en M.L. Sandige. *Fundamentals of Digital and Computer Design with VHDL*. McGraw-Hill Education, 2012, p. 164–165. ISBN: 978-1-259-00755-2 (blz. 95).

[20] R.M. Marston. *Modern CMOS Circuits Manual*. Newnes, 1996, p. 178–179. ISBN: 0-7506-2565-1 (blz. 96).

[21] A.P. Thijssen en H.A. Vink. *Digitale Techniek, van probleemstelling tot realisatie deel 2*. 5de ed. Delft University Press, 2000, p. 251. ISBN: 90-407-1838-5 (blz. 96).

[22] B. Cohen. *Real Chip Design and Verification Using Verilog and VHDL*. VhdlCohen Publishing, 2002. ISBN: 0-9705394-2-8 (blz. 96).

[23] C. Wellheuser. *Metastability Performance of Clocked FIFOs*. 1996. URL: https://www.ti.com/lit/an/scza004a/scza004a.pdf (bezocht op 30-12-2018) (blz. 131).

[24] T.J. Chaney. "Measured Flip-Flop Responses to Marginal Triggering". In: *IEEE Transactions on Computers* C-32.12 (dec 1983), p. 1207–1209 (blz. 131).

[25] NXP. *74F50729 Synchronizing dual D-type flip-flop with edge-triggered set and reset and metastable immune characteristics*. Originele datasheet niet beschikbaar bij fabrikant. 2003. URL: https://www.digchip.com/datasheets/download_datasheet.php?id=394707&part-number=I74F50729D (bezocht op 30-12-2018) (blz. 131).

[26] Xilinx. *XAPP077 Metastability Considerations*. Originele application note niet beschikbaar bij fabrikant. Jan 1997. URL: http://userweb.eng.gla.ac.uk/scott.roy/DCD3/technotes.pdf (bezocht op 30-12-2018) (blz. 131).

[27] Texas Instruments. *Design Considerations for Logic Products, Application Book*. 1997. URL: https://www.ti.com/lit/an/sdya002/sdya002.pdf (bezocht op 30-12-2018) (blz. 131).

[28] Altera. *Understanding Metastability in FPGAs*. Altera is overgenomen door Intel. Jul 2009. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01082-quartus-ii-metastability.pdf (bezocht op 30-12-2018) (blz. 133).

[29] D.J. Kinniment. *Synchronization and Arbitration in Digital Systems*. Wiley, 2008. ISBN: 978-0-470-51713-0 (blz. 133).

[30] S. Harris en D. Harris. *Digital Design and Computer Architecture: ARM Edition*. Elsevier Science, 2015. ISBN: 9780128009116 (blz. 133).

[31] J. Sparsø en S. Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective*. European low-power initiative for electronic system design. Springer, 2001, p. 79–80. ISBN: 9780792376132 (blz. 133).

[32] R. Ginosar. "Metastability and Synchronizers: A Tutorial". In: *IEEE Design Test of Computers* 28.5 (sep 2011), p. 23–35 (blz. 133).

[33] C.E. Cummings, D. Mills en S. Golson. *Asynchronous & Synchronous Reset Design Techniques - Part Deux*. 2003. URL: http://www.sunburst-design.com/papers/CummingsSNUG2003Boston_Resets.pdf (bezocht op 30-12-2018) (blz. 134).

[34] Develeast SA. *Reset Synchronization*. Bedrijf bestaat niet meer. 2010. URL: https://www.scribd.com/document/87138725/TN005-Reset-Synchronization-v00 (bezocht op 30-12-2018) (blz. 135).

[35] J. Rose e.a. "Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency". In: *IEEE Journal of Solid-State Circuits* 25.5 (okt 1990), p. 1217–1225 (blz. 143).

# A

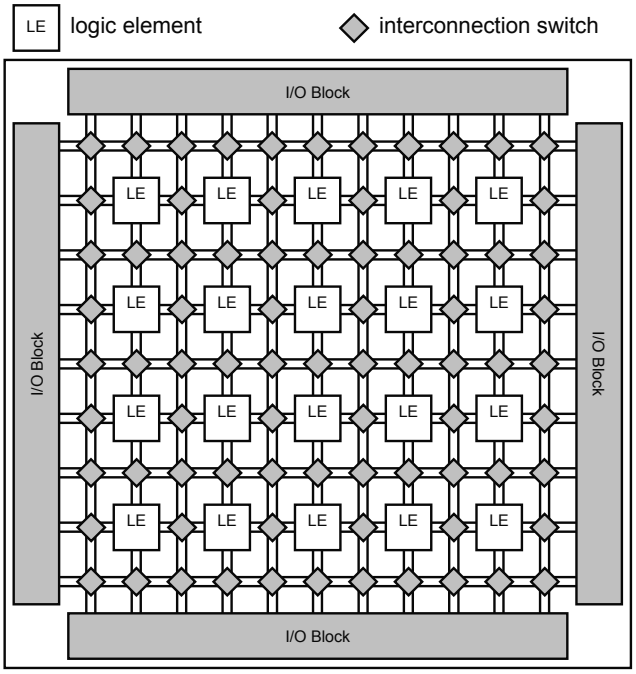
# De FPGA

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

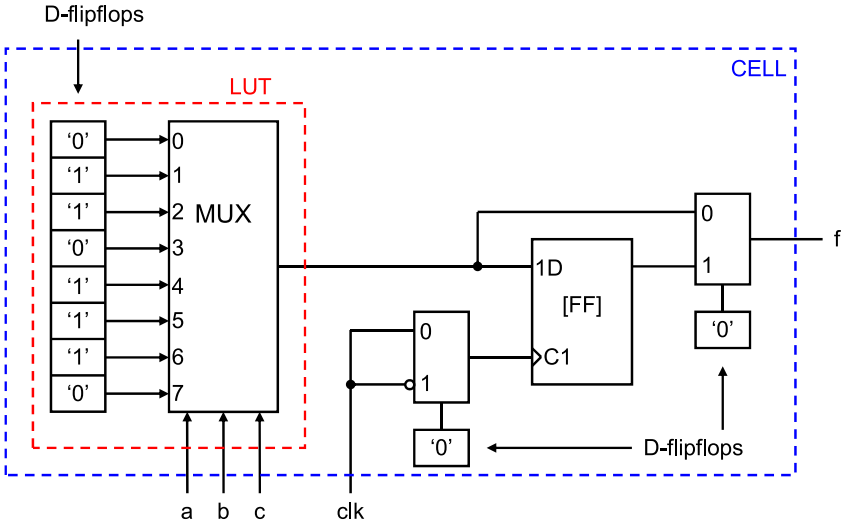
FPGA's hebben een behoorlijk verschillende architectuur ten opzichte van PAL's, PLA's en ROM's omdat FPGA's niet bestaan uit AND- en OR-matrixen maar uit cellen. In figuur A.1 is de globale opbouw van een FPGA te zien. Er zijn drie typen componenten te onderscheiden. Aan de randen van het ic zijn de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

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

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




**Figuur A.1:** *Globale opbouw FPGA.*




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

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

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

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

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

In de praktijk komen LUT's met vier en zes (stuur-)ingangen voor. Daarmee zijn logische functies met vier respectievelijk zes variabelen te realiseren. Rose toont in [35] aan dat een aantal in de industrie gebruikte ontwerpen het best kunnen worden gerealiseerd met LUT's van drie of vier variabelen en dat het efficiënt is om een cel met een flipflop uit te rusten.

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

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

# Index


**A**
adres, 85
adresbits, 85
adresbus, 54
after, 6
aggregate, 13, 62
architecture, 6
arithmetic shift, 68
array (VHDL), 12
asynchrone reset, 133

**B**
BCD-teller, 80
bidirectioneel schuifregister, 63
bit (VHDL), 5, 7
bit_vector, 12
bitstring (VHDL), 12
boolean (VHDL), 7

**C**
character (VHDL), 8
clock domain, 129
commentaar (VHDL), 6
component (VHDL), 28
concatenation (VHDL), 13
concatenation operator, 62
concurrent VHDL, 15
Conditional Signal Assignment, 15
constraints, 44

**D**
D-flipflop (VHDL), 25
DAC, 85
databits, 85
databus, 54
dataflow VHDL, 6
dataskew, 113
delay (VHDL), 18
delta delay (VHDL), 34
Design Unit (VHDL), 4
digitaal-analoog converter, 85
directe dataoverdracht, 103
don't care (VHDL), 9
downto (VHDL), 12
driver, 9
Duty Cycle, 90

**E**
element (array), 12
entity, 5
enumeratie, 12
event, 32

**F**
for (VHDL), 23
full adder (VHDL), 45
full duplex, 70

**G**
Gajski-Kuhn Y-chart, 2
gated D-latch (VHDL), 24
generic constant, 29
generic list, 30
gewone ringteller, 95
glue signals, 77

**H**
half adder, 75
hiërarchie (VHDL), 28
High-Z, 19, 54
hold slack, 105
hold time violation, 112
holdtijd, 102

**I**
Identifier (VHDL), 6
in (VHDL), 5
indirecte dataoverdracht, 108
instantiëring, 29
integer (VHDL), 8

**J**
Johnson-teller, 96

**K**
kloksignaal (VHDL), 36
klokskew, 105
kloksynchroon, 126
kritieke pad, 112

**L**
laagdoorlaatfilter, 85
latch clock edge, 106
launch clock edge, 106
length attribute, 52
logical shift, 68

**M**
maximale frequentie, **111**
maximale vertragingstijd, 102
metastabiliteit, 130
minimale pulsbreedte klok, 102
minimale vertragingstijd, 102
MTBF, 131
multiplexer (VHDL), 46

**N**
NAND (VHDL), 9
negatieve holdtijd, 110
NOR (VHDL), 10, 24
numeric_std bibliotheek, 50

**O**
one hot, 50
ontwerpdomeinen, 2
out (VHDL), 5
output enable, 54

**P**
pariteitsbit, 70
periodetijd, 102
port association list, 29, 76
port map, 29
prescaler, 68
primitieven, 44, 45
priority encoder, 47
process (VHDL), 21
pull-down (VHDL), 9
pull-up (VHDL), 9
Puls-breedte modulatie, 89
Pulse Width Modulation, 89
PWM, 89

**R**
RAM, 53
range (VHDL), 8
real (VHDL), 8
reset, 133
reset (VHDL), 37
reset synchronizer, 135
resize (VHDL), 52
resolution functie, 9
resolution tabel, 10
resolution time, 131
resolved type, 9
ringteller, 95
ROM, 53, 85
RS-232, 69

**S**
schuifregisters, 61
Selected Signal Assignment, 16
sequential VHDL, 20
setup slack, 104
setuptijd, 102
sign extension, 52
signal attribute, 25
signals (VHDL), 6
signed (VHDL), 50
simulatie, 31
std_logic, 8
std_logic_vector, 12
std_ulogic, 8
stopbit, 70
strong typed, 7
structural VHDL, 28
subtype, 11
synchrone clear, 79
synchrone reset, 133
synchronisatie, 126
synchronizer, 126
synchronizer chain, 126
synthese (VHDL), 44
synthesizer, 44

**T**
T-flipflop, 73
teller, 71
    cyclisch, 72
    eigenschappen, 72
testbench, 32, **35**
time (VHDL), 8
timing externe ingangen, 109
timing externe uitgangen, 109
to (VHDL), 12
top level (VHDL), 4
tri-state (VHDL), 9
type, 11

**U**
universeel schuifregister, 64
unsigned (VHDL), 50

**V**
variabele (VHDL), 23
vector (VHDL), 12
vector slice, 13

vertragingstijden (VHDL), 18

voorwaarden dataoverdracht, 104

**W**

wait (VHDL), 26
while (VHDL), 24