KU Leuven

Ma Ingenieurswetenschappen:
Computerwetenschappen

Digitale Elektronica en Processoren

Lesnota's

Author:
Helena Brekalo

2015-2016

Contents

Chapter 1

Les 1

1.1 Slides: 1_Intro

Slide 6: Die meerdere cores zorgen voor een grotere verwerkingskracht. Voorbeeld van de smartphone: het blokschema is hier van belang. Je hebt een aantal dingen die contact maken met de buitenwereld. In het blauw: de telecommunicatie, het echte gsm-gebeuren. Je hebt ook ergens iets draadloos (WiFi) (bruin). Het is dus heel specifieke hardware die een bepaalde taak vervult, die is geoptimaliseerd om één ding heel goedkoop, eenvoudig en compact te doen. Je hebt ook geheugen en powermanagement. De donkere blokjes in het midden zijn een algemene processor (rechts) die redelijk flexibel is en een specifieke processor (links) die alle signaalverwerking doet.

Slide 7: SoC: alles op 1 enkele chip. Heel wat verschillende dingen zijn weer aanwezig. 3 grote stukken: opgelijst met tweekleurige pijltjes:

- Programmeerbare processor: die kan vanalles en nog wat doen.
- Niet-programmeerbare processoren: specifieke processoren voor beeldverwerking bv. Ze zijn meestal beperkt programmeerbaar, om één specifieke familie van problemen op te lossen.
- Controle-eenheden.

Waarom die tweede groep? → Veel goedkoper! Je hebt ongeveer 1000 keer zoveel energie nodig bij een programmeerbare processor dan bij een niet-programmeerbare. Bv. in plaats van 1W 1kW.

Slide 8: We vertrekken van een doel: we hebben een algoritme of beschrijving op gedragsniveau. Dat is een combinatie van software en hardware. Sommige dingen doe je via software omwille van flexibiliteit, andere dingen doen we via hardware. Dat is allemaal afhankelijk van elkaar. De keuze van welk programma je gaat gebruiken zal impact zal hebben op de hardware en de hardware die je hebt zal bepalen hoe programmeerbaar het is. Men spreekt daarom van hardware-software codesign. Hardware gaat ook interrageren met wat als algoritme gebruikt kan worden: sommige dingen zijn makkelijk te implementeren, andere moeilijker. Het kan zinvol zijn om de algoritmen eventueel wat aan te

passen.
Bij de hardware heb je een digitaal en een analoog gedeelte. Bij een smartphone: alle telecommunicatie is analoog: continue signalen. Wat in het midden stond was meer digitaal: meer berekeningen. Beiden moeten geïmplementeerd worden, gewoonlijk op dezelfde componenten, momenteel gebeurt dat met transistoren en dat moet vertaald worden naar een chip (geïntegreerde schakeling). Wij gaan het hier enkel hebben over het linkerdeel. Dit kan nog opgesplitst worden in 3 lagen. Poorten liggen het dichtst aan bij transistoren, die kunnen gebruikt worden om basiscomponenten te maken (bv. basisprocessoren) die samengezet kunnen worden om systemen te bouwen.

Slide 9: Het gaat in eerste instantie over het ontwerp van elektronische schakelingen. We gaan daar ook voorbeelden van zien. Hardware kan je beschrijven via een schema, maar ook via een taal: hardware-beschrijvingstalen. We gaan die ook bekijken. We gaan dit ook toepassen in oefeningen en labo's.

Slide 10: Drie grote blokken:

1. Basis: principes van digitaal ontwerp en hoe je dat doet met elektronica. Veel van de dingen die we gaan zien kan ook met andere dingen dan elektronica.

2. Dan gaan we zien hoe we daar schakelingen mee kunnen maken. Eerst zonder geheugen en dan met geheugen.

3. Dan gaan ingewikkeldere algoritmen implementeren en kijken hoe dat geïmplementeerd kan worden.

Slide 11: Vereiste voorkennis.

Examen: FSM is schriftelijk, de rest is mondeling met schriftelijke voorbereiding. Eerst wordt de theorie ondervraagd, dan de vertaling van een algoritme (vraag 1).

1.2 Slides: 2_Digitaal_Ontwerp

Slide 3: Documentatie is heel belangrijk: als handleiding en om bij te houden wat geprobeerd is (en wat dus wel en niet werkt).

Slide 4: Specificatie: beschrijving, heel dikwijls in natuurlijke taal, van welke berekeningen je wilt/hebt. Specificaties zijn in de meeste gevallen te algemeen en te onvolledig om er een duidelijke implementatie van te maken omdat het in veel gevallen te weinig specifiëert. Er kunnen heel veel verschillende algoritmes gebruikt worden om een probleem op te lossen. Langs de andere kant gaat het soms al zaken vastleggen waarvan je niet de bedoeling had dat het vastgelegd werd (bv. vaste of vlottende komma).

Slide 5: Het geheel is dikwijls iteratief (vandaar de lus). Synthese: schakeling maken. Het is niet het bouwen van de schakeling maar een iets algemenere beschrijving maken. Normaal doe je geen synthese van een algemene beschrijving om onmiddelijk op een geïntegreerde schakeling uit te komen. Je gaat daarom in niveau's werken: in stapjes oplossen: eerst op het hoogste niveau kijken en een heel algemene beschrijving doen: ik wil een FFT implementeren, wat houdt dat in, wat voor verwerking heb ik nodig?
Vervolgens ga je een niveau lager: welke blokken heb je allemaal nodig? Die ga je verder verfijnen tot op een niveau waarop je blokken krijgt waarvan je ongeveer weet hoe die geïmplementeerd moeten worden. Je gaat dat dan nog verder opsplisten (in poorten en flipflops). Dan komt er nog een niveau onder: hoe maak ik een poort (met transistoren)?
Door dat in stukjes te doen en elk stukje per keer op te lossen is het doenbaar: je maak ter hapklare brokken van die je kan verwerken. Wij gaan de twee middelste niveau's vooral bekijken.

Slide 6: Belangrijk is het gebruik van bibliotheken: de kennis van anderen, dingen die je zelf niet meer moet doen. We gaan componenten hergebruiken. Als iemand iets nog niet gedaan heeft ga je het zelf maken en het in de bibliotheek steken zodat het hergebruikt kan worden. Die bibliotheken komen op ieder van de niveau's terug. Waarom zo belangrijk? → Wet van Moore.
We krijgen dus heel wat mogelijkheden jaarlijks erbij. Uiteindelijk zal dat wel gaan vastlopen. Wat ontstaat er nu? We krijgen elk jaar meer en meer mogelijkheden. We kunnen dit jaar een twee keer zo complexe schakeling maken dan 2 jaar geleden. Een meer complexe schakeling ontwerpen duurt meer dan twee keer zo lang want je raakt volledig het overzicht kwijt. De processoren aan de rechterkant teken je niet zomaar op 1 blad, dat duurt eindeloos lang. De enige manier om dat doenbaar te krijgen is door zoveel mogelijk te hergebruiken. Dit betekent dat het ontwerpen altijd achterloopt op de mogelijkheden die we hebben.

Slide 7: Blauw: wat we als verwerkingskracht kunnen doen, dit is wat ons interesseert. Meer cores op 1 chip: kost qua ontwerpinspanning niet zoveel moeite. In de toekomst gaan we dus veel meer in parallel moeten laten gebeuren. Vermogen (Typical Power): is belangrijk want je kan dat niet eindeloos laten toenemen. Het vermogengebruik hangt af van de spanning in het kwadraat (V), de grootte van de chip (C) en de frequentie (f). Hoe hoger de frequentie, hoe meer verwerkingskracht je hebt. Oorspronkelijk zaten we aan minder dan 1 Watt voor een geïntegreerde schakeling zitten we nu aan 100 Watt.
We willen het vermogen dus constant houden maar dat heeft zijn impact op de frequentie en de verwerkingskracht.

Slide 8: Analyse: nagaan of het voldoet aan de specificaties. Dat doe je niet helemaal op het einde: het ontwerp gebeurt in stapjes en je gaat telkens testen of dat het wel voldoet aan zijn specificaties op dat niveau. Dat betekent meer dan alleen maar testen of het doet wat het moet doen. Het functionele is eigenlijk maar één aspect ervan, er hoort veel meer bij: het vermogenverbruik bv. mag niet eindeloos oplopen. De snelheid waaraan dat kan werken, de kostprijs,... dat

soort zaken moet ook nagekeken worden. De testbaarheid is ook een belangrijke factor.

Slide 10: Schakelingen gebaseerd op logische werking. We werken met discrete signalen en in de praktijk met bits: de waarden worden enkel 0 of 1.

Slide 11: Er is een verband tussen het al dan niet ingedrukt zijn van de knop en de uitgang. Je kan ook andere functies hebben: een complementaire die in rusttoestand een verbinding maakt: in rusttoestand is er een verbinding en brandt de lamp. Druk je de knop in, gaat de lamp uit. Er zijn heel wat verschillende manieren om een logische functie aan te duiden, afhankelijk van welk boek/welke beschrijving je neemt: onderaan: verschillende manieren om NOT te schrijven. Wij gebruiken normaal altijd een accent (').

Slide 12: Andere logische functies:

- AND: schakelaars in serie zetten en dan brandt de lamp alleen als beide knoppen ingedrukt zijn.
- OR: schakelaars in parallel zetten: de lamp brandt als een van de twee schakelaars ingedrukt is of ze alletwee ingedrukt zijn.

We kunnen complexere schakelaars maken: je kan een lamp bedienen vanop twee plaatsen, dat is ook een logische functie: XOR. De lamp zal branden als de ene of de andere knop ingedrukt is, maar niet als ze alletwee ingedrukt zijn of alletwee niet ingedrukt zijn. Voor andere logische functies kan je ze herleiden tot de twee eersten (AND en OR): XOR is een parallelschakeling van een serieschakeling: je hebt alleen maar een parallel- en een serieschakeling nodig en de inverter. We hebben dus maar 3 logische schakelingen nodig: het inverse (NOT), een AND en een OR, daarmee kunnen we alle logische functies bepalen.

Slide 13: Hoe beschrijven we de functionaliteit van zo'n logische schakeling? $\rightarrow$ Met een waarheidstabel: een opsomming van alle mogelijke combinaties aan de ingang en daarbij wordt aangegeven wat de uitgang is. Vermits een waarheidstabel eigenlijk de functionaliteit beschrijft zijn twee implementaties met eenzelfde waarheidstabel volledig equivalent, die doen exact hetzelfde. Je kan dus ook zeggen dat als je een waarheidstabel hebt die je op verschillende manieren kan maken, je aan de hand van die waarheidstabellen verschillende schakelingen kan maken. Het is dus niet zo dat bij een functionaliteit altijd maar 1 implementatie mogelijk is, er zijn meerdere mogelijkheden.

Slide 14: Logische poorten: meer dan de inverter, de AND en de OR-poort heb je in principe niet nodig, hoewel men dikwijls gebruik maakt van complexe poorten. In CMOS is het bv. goedkoper om met complexe poorten te werken dan met de basispoorten. Een logische schakeling is dan een combinatie van die poorten. Je kan dit met een schema doen of een programma.

Slide 15: Als je zo'n schema hebt kan je ook makkelijk de waarheidstabel opstellen. Je gaat gewoon op alle tussenliggende draden kijken wat er gebeurt voor elke combinatie van ingangen aan de uitgangen: dus aan a en b en daarna f en op die manier kan je de functionaliteit gaan beschrijven van wat die schakeling doet.
Dat is een ding: je hebt een schakeling en je weet wat die doet.
Meestal wil men het omgekeerde: wat is de schakeling voor de gegeven waarheidstabel? Maar zelfs dit is een onvolledige beschrijving van wat die schakeling doet want het beschrijft alleen de logische functie en in de praktijk is een schakeling meer dan alleen zijn logische functie.
Een schakeling wordt altijd nog aangevuld met een tijdsgedrag. Het gaat er hier niet over hoe traag die poorten werken: het is niet dat als er bij x iets verandert, er onmiddelijk iets bij b verandert, dat gebeurt nooit ogenblikkelijk. Ogenblikkelijk bestaat niet: het duurt altijd een zekere tijd. Dat betekent dat als we iets veranderen bij x bv., dat een tijdje later a gaat veranderen. y veranderen gaat een tijdje later b veranderen. Als a en b veranderd zijn, zal f een tijdje later veranderen. Wat je dan ziet verschijnen zijn vertragingen: het ogenblik dat de ingang verandert en dat de uitgang verandert, in elke reële schakeling is dat zo. Die vertraging gaat meespelen in hoe efficiënt de schakeling werkt. Het is dus belangrijk om die te kennen, je gaat dat nooit terugvinden in een logische functie, dat is bijkomende informatie.
Het tijdsgedrag nakijken is ook heel belangrijk om problemen te detecteren, problemen die je niet gaat zien als je alleen maar naar de functionaliteit gaat kijken. Bv. a en b veranderen "tegelijkertijd" (dat bestaat niet! De ene gebeurtenis zal altijd iets voor de andere plaatsvinden.) $\rightarrow$ dat kan zijn gevolgen hebben: als b iets vroeger verandert dan a, dan krijgen we heel kortstondig een waarde 0, dan zie je aan de uitgang plots een piekje verschijnen, wat er niet zou mogen zijn. Als je gewoon naar de logica kijkt, zou die er niet mogen zijn, maar in praktijk is dat er wel en dit kan voor problemen zorgen. Dat piekje dat er is, dat we niet verwachten, kan opeens gevolgen hebben in de verdere verwerking van de schakeling, dus we moeten ons daar bewust van zijn. We moeten dus niet alleen de logische functies controleren, maar het tijdsgedrag is even belangrijk.

Slide 16: Kijken we dan naar het omgekeerde: we hebben een functionaliteit, hoe kunnen we dat implementeren? Er staan twee implementaties, als je die gaat uitrekenen, dan zie je dat die identiek zijn, die doen juist hetzelfde.
Als we dit gedaan hebben kunnen we kiezen voor een van de implementaties en vanuit logisch standpunt maakt het geen enkel verschil. Het is dan niet zo dat je eender welke realisatie zomaar mag kiezen: de linkse heeft meer poorten dan de rechtse: het gaat meer kosten (meer transistoren, een grotere chip,...). De linkse gaat ook trager werken want als je van x naar de uitgang moet ga je door twee ingangen telkens moeten gaan waar dat rechts niet het geval is.
Als we dan toch kunnen kiezen willen we het zo goedkoop mogelijk en zo snel mogelijk hebben. Hoe sneller die schakeling is, hoe hoger de verwerkingskracht: we kunnen maar aan de volgende bewerking beginnen als de vorige gedaan is. Als die schakeling dus heel snel een resultaat levert kan die heel snel aan een volgende bewerking beginnen. Vandaar dat hoe sneller de schakeling werkt, hoe hoger de verwerkingskracht. Als we dan toch kunnen kiezen willen we die schakeling met de minimale kostprijs en die schakeling met de hoogste verwerk-

ingskracht (die zo snel mogelijk het resultaat geeft). We gaan daarvoor gebruik maken van de formules die erbij staan.
De snelheid is evenredig met het aantal ingangen, dus de linkse zal trager werken dan de rechtse. Op die manier kan men tussen de twee schakelingen kiezen en de beste kiezen.

Slide 18: Boole algebra kan gebruikt worden omdat we ook hier met twee (logische) waarden werken. In de Boole algebra vertrekt men van een aantal axioma's. De duale uitdrukkingen zijn hier ook steeds geldig: je mag 0 door 1 vervangen en omgekeerd en + door . en omgekeerd.

Slide 20: Wij gaan vooral gebruik maken van de wet van De Morgan: de inverse van het product is de som van de inversen en het inverse van de som is het product van de inversen.
In dit vak zal een distributiviteit voor de vermenigvuldiging ten opzichte van de optelling en distributiviteit van de optelling ten opzichte van de vermenigvuldiging gebruikt worden.

Chapter 2

Les 2

2.1 Slides: 2_Digitaal_Ontwerp

Slide 15: Je wil weten wat de vertragingen zijn: het duurt een poosje als je iets aan de ingang verandert eer je het resultaat aan de uitgang ziet. Het tijdsgedrag toont ook problemen met de implementatie. Het probleem op de tekening toont dat a en b quasi gelijktijdig veranderen. Als a van 0 naar 1 gaat en b van 1 naar 0 zal de uitgang ook 1 worden, dus f zou 1 moeten blijven. Het probleem ontstaat wanneer b iets vroeger verandert dan a: het uiteinde zal even 0 zijn, hoewel dat logisch gezien niet zou mogen. Op het moment dat je het ontwerp maakt is het belangrijk te weten wat zou kunnen optreden als probleem en je moet dat testen aan de hand van de reële implementatie.

Slide 22: Wij willen iets doen met de schakeling, hoe maak je de schakeling zodat die daaraan voldoet? Er zijn 2 manieren om dat te benaderen: een schema tekenen/zoeken dat eraan voldoet. Anderzijds kan je ook een beschrijving in een taal doen.
Vandaag gaan we beiden ingeleid zien.

Slide 23: Hoe best het gedrag beschrijven? → Aan de hand van een waarheidstabel. Het is nu de bedoeling om aan de hand van de waarheidstabel een implementatie te maken. De logische functie kan je ook altijd omzetten naar een waarheidstabel. Verschilllende logische functies kunnen eenzelfde waarheidstabel geven. De basisfunctionaliteit voor combinatorische schakelingen is dus de waarheidstabel.
In die waarheidstabel staat dat de uitgang 1 is als x en y 0 zijn,...
Je kan dit rechtstreeks implementeren met poorten: zie tekening rechtsonder.
Ofwel zijn beiden nul, ofwel is x 0 en y 1 ofwel zijn ze beiden 1 geeft f = 1. We kunnen van een beschrijving altijd rechtstreeks overgaan naar een implementatie. Probleem: we zijn niet geïnteresseerd in *een* implementatie maar in de beste, afhankelijk van de factoren die we belangrijk vinden.
Er zijn natuurlijk verschillende oplossingsmogelijkheden. Het komt er dus op neer om de beste te zoeken.

Slide 24: Je kan daar naartoe werken door de bovenste uit te schrijven en dat eventueel te vereenvoudigen en daar wiskundig op te beginnen rekenen met de Boole-algebra. Het resultaat op deze slide getoond is veel eenvoudiger dan het voorgaande. Het gaat dus goedkoper zijn en een kleinere vertraging hebben en een kleiner vermogen verbruiken.
Hoe weet je nu dat je die regels moet toepassen? Niet: de prof wist wat hij moest vinden en heeft dan regels gebruikt om ertoe te komen. Je kan dus eindeloos vanalles proberen en als je dat op de zuiver wiskundige manier wil doen moet je heel veel proberen en dan de oplossing behouden tot je het beste resultaat bekomt. Je zou ook een regel kunnen toepassen die op zich een minder goede oplossing geeft. Het is dus heel moeilijk om te zien wanneer je moet stoppen met regels toe te passen. In CMOS is het tweede goedkoper dan het eerste: de technologie komt erbovenop om te weten wat het beste is.
Je hebt dus geen stappenplan/algoritme waarmee je er gegarandeerd komt. Hoe komen we dan tot een beter resultaat? → Gaan we later zien, in het volgende hoofdstuk: hoe kunnen we met een minimum aan uitproberen tot een goed resultaat komen?

Slide 25: Je kan dat formuleren in functie van min- en maxtermen. Een minterm is een functie zoals beschreven op de slide die alleen waar is voor één van de rijen van de waarheidstabel. Je hebt hier 3 variabelen dus 2^3 mogelijkheden. Hier staat geen functie of uitgang bij. Als je de functie erbij betrekt spreekt men over 1-minterm: de functie waarvoor de uitgang van de schakeling 1 is. In dit geval zijn dat er 4.

Slide 26: Je kan dat als volgt omschrijven: je kan altijd een implementatie van een schakeling doen door de som van zijn 1-mintermen te nemen. Dus voor elke rij waarvoor de functie 1 is hebben we een 1-minterm. Men noemt dit de canonieke som van producten oplossing → de voor de hand liggende oplossing, waar je niet over moet nadenken en je zo kan realiseren.
Het mooie aan Boole is dat de duale uitdrukking ook altijd geldig is. Dit kunnen we ook hier gaan toepassen.

Slide 27: Maxterm = logische functie die 0 is voor 1 bepaalde rij van de kolom. Weerom kun je de functie er dan bij betrekken. Je krijgt hier geen 1-maxterm maar een 0-maxterm: maxterm waarvoor de functie 0 is en dus de rijen die er bij de 1-minterm niet bijzaten.

Slide 28: Als je dit gaat combineren krijg je een product van sommen. Voor elke beschrijving die we hebben weet je dat er 2 implementaties zijn. Dat zijn 2 verschillende implementaties en we gaan sowieso moeten kiezen wat het beste is.

Slide 29: Het probleem met de canonieke: de duurste oplossing die er is (maar wel makkelijk op te schrijven): elke AND-poort heeft zoveel ingangen als er variabelen zijn. Een OR-poort heeft zoveel ingangen als dat er rijen zijn waar een 1 in voorkwam bij f. Dat is dus niet de compactste oplossing. We zijn op zoek naar eenzelfde manier van werken (SOP of POS) maar met poorten met minder ingangen die zo ook minder vermogen verbruiken. We gaan niet

meer voor elke AND-poort alle ingangen proberen gebruiken en niet evenveel ingangen voor de OR-poort als in de waarheidstabellen.

Slide 30: De twee canonieke vormen bovenaan: als je daar een paar regels op toepast kan je beiden vereenvoudigen zoals onderaan getoond. Dit zijn de standaardvormen: die vorm die het minimum aantal poorten en de poorten met de minste ingangen gaat proberen te gebruiken. Als je de bovenste rijen daarmee vergelijkt bevatten de poorten van de bovenste term 3 ingangen vs. 2 bij de termen onderaan. Dat is al een winst in alle randvoorwaarden die we willen. Daarenboven hebben we vanboven 4 OR-poorten en vanonder maar 2. Ook dit is dus een duidelijke reductie in hardware.
We weten nu wel nog niet hoe we overgaan naar standaardvorm.

Slide 31: We gaan dikwijls naar de SOP en POS-implementatie kijken omdat het zo is dat die standaardvorm eigenlijk de goedkoopste implementatie en dikwijls ook de snelste is, zeker wanneer je je beperkt tot 2 lagen. Een laag: bovenste tekening: een deel/laag AND-poorten en een laag OR-poorten (1 poort in dit geval). Die bepaalt de vertraging: je moet door 2 lagen om tot het einde te geraken. Hoe meer lagen, hoe trager het geheel gaat werken. We gaan dus proberen het aantal lagen te beperken. Het minimum dat we in normale omstandigheden nodig hebben zijn 2 lagen. Vandaar dat de getoonde tekening de snelste oplossing is.
We kunnen ook naar meer lagen gaan: onderste tekening. Hier gaat het duidelijk trager zijn: 3 lagen (dit was een examenvraag vorig jaar!). Soms is snelheid niet het belangrijkste: soms wil je een zo goedkoop/compact mogelijke implementatie en dan kan dit wel interessant zijn. Het onderste kan goedkoper zijn dan het bovenste: het verschil zit 'm in het feit dat er bovenaan een poort is met 3 ingangen en je die vanonder niet hebt. Als je dus niet in de snelheid geïnteresseerd bent, kan de onderste dus beter zijn.
Wij gaan meestal met 2 lagen werken omdat dat evidenter is om toe te komen, 3 lagen is minder evident.

Slide 32: We gaan nu ook nog kijken naar NAND en NOR.

Slide 33: We maken gebruik van 1 enkele soort poort. Dat is mogelijk door inverterende poorten zoals NAND of NOR-poorten te volgen. Het wordt als 1 poort beschouwd (de combinatie van AND en inverse of OR en inverse). In de CMOS-technologie is het makkelijker om een NAND-poort te maken dan een AND-poort:

- Je moet een AND maken door een NAND te gebruiken en er nog een inverter achter te zetten.
- Wanneer je NAND of NOR gebruikt moet je geen 3 basisschakelingen hebben, je hebt er maar 2 nodig. Schema onderaan: illustratie hoe je poorten kan maken met alleen NAND of alleen NOR. Denk tijdens het studeren na welk van de mogelijkheden het interessantst is om te gebruiken.

Ook een NOR-poort kan je maken met een NAND-poort: de wetten van De Morgan: product van inversen is hetzelde als inverse van een som. Dat zie je in de omkaderde vakjes. Met een enkele basispoort kan je dus alles maken.
Het is nu wel zo dat de technieken die we later gaan zien altijd van deze basispoorten gebruik maken. Die gaan iets leveren dat bestaat uit AND en OR en invertoren. Er zijn geen specifieke technieken om alleen NAND of NOR te gebruiken. Op zich is dat geen probleem omdat een implementatie zoals op **Slide 34** kan omzetten naar alleen NAND of alleen NOR.

Slide 34: Maak gebruik van invertoren aan de uitgang en ingang. De drie schakelingen onder elkaar zijn dezelfde logische functies. Hetzelfde met iets alleen NOR-poorten.
Houd er rekening mee dat als je tot een implementatie met NAND-poorten wil komen je eerst een som van producten moet implementeren.

Slide 35: Als je gaat kijken naar de ontwerpmiddelen die bestaan, dan gaan die een ontwerp maken en die gaan je erbij helpen. Het grootste werk dat je zal hebben is het ingeven van het ontwerp, hetzij via een schema, hetzij via VHDL. Daarna moet je doen wat we in de volgende hoofdstukken gaan bekijken. Die programma's zijn slim genoeg om alles te doen wat wij gaan zien, dus ze gaan in veel gevallen een automatische synthese doen. Op dat moment heb je een logische implementatie en kan je een functionele simulatie doen: testen of het doet wat het moet doen: als je een bepaalde combinatie van inputs geeft gaat het geven aan de uitgang wat het moet geven.
Het tijdsgedrag is even belangrijk, maar dat kan je op dit moment nog niet testen: je hebt alleen een logische beschrijving. Daarom komt er een stap achter: je moet ook het fysische ontwerp maken en nagaan wat het tijdsgedrag en de elektrische eigenschappen etc. zijn. Je moet een prototype bouwen en daarop metingen uitvoeren. Wanneer dat tot een geïntegreerde schakeling moet leiden is dat een verspilling van tijd en kosten, dus normaal gaat de fabrikant van die geïntegreerde schakeling weten wat de eigenschappen gaan zijn van die componenten. → Al die details zijn gekend. Die zal een module aanbieden waarin alle gegevens verwerkt zijn en het fysische ontwerp nabootst. Op die manier kan je metingen doen qua tijd etc. Je kan ook naar alle problemen zoals glitches gaan zoeken. Pas dan ga je de finale stappen uitvoeren: implementeren of de chip laten maken.

Slide 37: Andere manier om hardware te omschrijven: hardware-omschrijvingstaal. Wij gaan kijken naar VHDL. Dat is ontstaan uit subsidies van het Amerikaanse ministerie van defensie. De bedoeling was dat men een methode wou hebben om heel complexe ontwerpen te beschrijven en alle gegevens erbij te zetten, alle informatie om te simuleren en beschrijven en dat systeem op een voldoende hoog niveau te beschrijven.
De taal moet in staat zijn om gelijk welk digtaal systeem te beschrijven en alle aspecten ervan. Het moet het dus volledig kunnen specifiëren. Dus niet alleen op poortniveau maar ook op hogere niveau's. Je begint dus op een hoog niveau en je daalt af. De bedoeling is hier dat je het systeem op alle niveau's kan beschrijven:

- Op het hoogste niveau heb je een beschrijving van het gedrag: "Dit blokje moet dat doen".
- Je moet ook kunnen testen dus tijdsparameters erin beschrijven. Simulatie moet hier dus ook mogelijk zijn.
- Men zou ook graag hebben dat er een automatische synthese is: dat je je geen zorgen moet maken om de tussenstappen.
- Ook documentatie is belangrijk.

Het is ook belangrijk dat het gestandaardiseerd is. Dat zorgt ervoor dat verschillende fabrikanten die ieder hun eigen implementaties doen mekaar altijd verstaan: er kan geen verkeerde implementatie mogelijk is want in de taal is de implementatie exact beschreven. Nadeel: als er nieuwe ideeën komen/er zijn tekorten, dan kan men het niet meer veranderen → nieuwe versie van de taal/software nodig.

Slide 38: VHDL is voor digitale systemen, maar een systeem bestaat meestal uit digitale en analoge componenten. Men heeft het systeem dus uitgebreid naar het combineren met analoge signalen. Om een analoge schakeling op te lossen heb je 200-300 gekoppelde differentiaalvergelijkingen nodig om tot uw oplossing te komen.

Slide 39: Andere kandidaat die minstens evenwaardig is: Verilog. Wordt meer gebruikt in de VS. VHDL is populairder in Europa. De concepten die erachter zitten zijn speciaal: hardware beschrijven is niet hetzelfde als een programma schrijven. Er zitten dus speciale concepten achter die je moet begrijpen voor beide talen, alleen de syntax verschilt.

Slide 40: Wat is nu het interessantste: schema of hardwaretaal? → Hangt af wat je ermee moet doen. Schema's zijn interssant voor kleine ontwerpen, VHDL is gewoonlijk interessanter wanneer het over grotere ontwerpen gaat. Nadelen van VHDL ten opzichte van schema's:

- De concepten achter VHDL zijn niet voor de hand liggend en je hebt dus enige tijd nodig om dat te leren beheersen.
- Het is nogal langdradig: om de kleinste component te beschrijven heb je onmiddelijk veel lijnen code nodig. Dit maakt dat het voor mensen moeilijk is om te begrijpen wat er allemaal gebeurt. Een schema is visueler en daarin zijn mensen meer getraind.
- VHDL bevat alle mogelijkheden: alle mogelijkheden voor simulaties, maar om te begrijpen wat het doet, om alleen de functionaliteit te kennen heb je dat allemaal niet nodig.

Slide 41: Voordelen van VHDL ten opzichte van schema's:

- Het is een standaard: je kan het gebruiken om informatie uit te wisselen tussen programma's/tools van verschillende fabrikanten. Het is dus makkelijk overdraagbaar.

- VHDL is goed in alles waar programma's goed in zijn. Wanneer je repetitieve structuren hebt (een aantal cores die allemaal hetzelfde zijn), dat is 1 instructie in VHDL. Als je een schema tekent, moet je dat zoveel keer tekenen. In een programma kan je ook parameters gebruiken. Als je een schema tekent moet je het helemaal uitgommen en opnieuw beginnen indien een parameter verandert. In een programma, als uw parameter bv. niet voldoende groot is, pas je dat gewoon aan.

- In een schema moet je onmiddelijk componenten kiezen om erop te zetten. VHDL heeft het voordeel dat je het gedrag kunt beschrijven, je moet dus niet per sé direct een implementatie kiezen.

Slide 42: Beperkingen:

- VHDL heeft zoveel mogelijkheden dat het enorm veel inspanning vraagt van de fabrikanten om alle aspecten uit de taal om te zetten naar hardware. Men heeft zich dus beperkt tot een subset die makkelijker om te zetten is in hardware. Wanneer je gaat synthetiseren ga je je dus moeten beperken.

- Van een hardwarecompiler verwachten we een enorme intelligentie. In vele gevallen zijn er een tiental mogelijkheden om een implementatie te realiseren. De compiler moet dus raden wat je bedoelt met wat je neerschrijft. In de meeste gevallen raadt hij dat zo strict mogelijk. Je moet in de meeste gevallen de compiler dus wat helpen. Je moet ongeveer weten hoe hij interpreteert wat jij hebt neergeschreven, dus je moet je aanpassen naar de mogelijkheden van de compiler. De hardware die uit een ontwerp van een beginneling komt zal minder optimaal zijn dan die an een meer ervaren ontwerper.

Slide 43: Schema tekenen: kan in een keer, maar in de praktijk ga je op de hiërarchische manier werken. Je gebruikt blokjes die je eventueel nog niet definiëert en waarvan je later de ontbrekende blokjes gaat specifiëren. Je kan het als een doos bekijken met een aantal in- en uitgangen. Die hebben een aantal karakteristieken en wat er in die doos inzit. Wat zit er in die doos? De bewerkingen die we moeten doen. Stel dat we een component hebben die test of 2 getallen aan elkaar gelijk zijn. In die doos zitten dus componenten en die leveren resultaten af. We hebben ook verbindingen nodig.
We kunnen dit nog niet implementeren omdat we nog niet hebben aangegeven wat de component is. We hebben wel al aangegeven dat het twee keer hetzelfde is. Die component moeten we nu maar 1 keer ontwerpen en die kunnen we 2 keer gebruiken.
Nu de rechtse doos: op een lager niveau specifiëren. We kiezen een mogelijke implementatie. Die twee stukken kunnen door 2 verschillende personen gemaakt worden. Men kan eerst het rechtse maken en dan het linkse. In veel gevallen is het rechtse beschikbaar in een bibliotheek.
Wanneer je met VHDL werkt, gaat dat eignelijk juist hetzelfde zijn.

Slide 44: 2 stukken: entiteit die de doos beschrijft. We hebben een comparator Comp die twee ingangen heeft: 2 ingangen en 1 uitgang en de ingangen zijn 8 bit en de uitgang is 1 bit.

Daaronder staat de architectuur: wat er in de doos zit, wat we daarnet in de doos tekenden.
Het verschil met het schema is dat we nu ook het gedrag kunnen beschrijven zoals gedaan is op de slide: we kunnen schrijven wat de comparator doet. Er staat niet geschreven hoe dit geïmplementeeerd moet worden, er is gewoon beschreven wat het doet.
Het is ook zo dat voor een bepaalde component er verschillende implementaties kunnen zijn: gedrag beschrijven, implementatie beschrijven, voor sommige theorieën zal iets goed zijn, in anderen niet. De architecturen krijgen daarom een naam omdat je zo kan kiezen voor een bepaalde implementatie.

Slide 45: Kijken we naar het topschema beschreven via de structuur. Een structurele beschrijving is juist hetzelfde als het schema tekenen. Entity Test is weer de doos en de architecture is weer de beschrijving hoe het geïmplementeerd is. We hebben een component Comparator nodig, dat is een virtuele component en we moeten die nog niet meteen beschrijven, gewoon de in- en uitgangen.
Bij begin staat er dat je die twee keer gebruikt. Je kan die verschillende namen geven (Comp1 en Comp2) om een onderscheid te kunnen maken. Rechts worden de draden beschreven.
Dit is in woorden neergeschreven wat rechts vanboven met een schema staat.
Belangrijk is om in gedachten te houden dat het een beschrijving van hardware is. De volgorde van instructies is hier niet belangrijk omdat het over hardware gaat. Het in een andere volgorde zetten gaat hetzelfde eindresultaat geven: de uiteindelijke manier van werken blijft hetzelfde. Ook de volgorde van de uitdrukkingen doet er dus niet toe: parallele uitdrukkingen. De volgorde is totaal onbelangrijk.

Slide 46: Hoe leg je het verband tussen de twee schema's/beschrijvingen? Dat gebeurt heel expliciet: VHDL is een heel strikte taal dus normaal moet je alles heel uitgebreid beschrijven, maar als je uw namen goed schrijft kan hij raden wat je bedoelt. Als de specificaties overeenkomen (naam Comparator met x ingangen en y uitgangen) dan zal die dat kunnen raden. Je kan ook een specifieke mapping maken: we gebruiken de component Comp met architectuur Behav1: we kiezen meteen een specifieke implementatie. Mapping: in het ene geval hebben we X,Y,Z gekozen, in het andere A,B,EQ. Je kan dus heel specifiek aangeven hoe je de schema's met elkaar verbindt.

Slide 47: VHDL verschilt vrij hard van een gewone programmeertaal, het ziet er alleen uit als een programmeertaal. Vanboven: "gewone" programmeertaal: functie die je definiëert. In VHDL beschrijf je gedrag. Het topniveau is het hoofdprogramma: main() die de functie 2 keer gaat gebruiken. Belangrijk verschil: bij het vorige waren de uitdrukkingen parallel, hier sequentiëel: het ene wordt uitgevoerd na het andere, er is een strikte volgorde en dat kan zijn impact hebben. Bij hardware is dat niet zo: het is niet zo dat de bovenste component eerst werkt en dan de onderste.
Nog een verschil: een softwareprogramma start je op, die voert het programma uit en aan het einde van het programma stopt het. Bij hardware is het juist omgekeerd: hardware stopt niet. Hardware begint te werken zodra je de spanning opzet en die blijft eindeloos altijd maar hetzelfde doen tot je de spanning

afzet. Dat is de enige manier om die tot stilstand te brengen. Software: er is een bug aanwezig als het blijft runnen, hardware: er is een bug aanwezig als het stopt.

Slide 48:

- Datatypes: als je een computer gebruikt met bv. 32 bits of 64 en je wilt 2 8-bits optellen, zal dat toch een 32 of 64-bitoptelling zijn. Bij hardware is dat niet zo. Bij hardware ga je niet nodeloos bits gebruiken: als je het met 4 bits kan doen ga je dat doen. Je gaat altijd het minimaal aantal bits gebruiken omdat dat het goedkoopste is en de compactste oplossing is. In VHDL moet je dat daarom ook specifiëren: gaat erin als 5 bit en komt eruit als 7 bits.

- Gelijktijdigheid: hardware werkt altijd gelijktijdig, niet de ene na de andere.

- Tijdsconcept: componenten werken continu, hardware stopt nooit. Qua tijdsconcept er ook rekening mee houden: de simulatie is niet de reële tijd. Het kan zijn dat je een minuut moet rekenen om de simulatie 1 nanoseconde vooruit te laten gaan.

2.2 Sides: 3_Technologie

Slide 3: Als we elektronische schakelingen willen implementeren, hoe gebeurt dat in de praktijk? Het eerste wat we moeten beslissen is welke fysische waarden we hebben. Als je een logische schakeling maakt, moet je een fysische waarde hebben. We gaan dat doen met fysische bereiken: als de blauwe pijl de spanning voorstelt: L is een lage waarde, H is een hoge waarde. Die twee bereiken raken elkaar niet: daartussen is een stuk waar het noch hoog, noch laag is: ongedefiniëerd.
Waarom bereiken? We willen onze schakeling zo robuust mogelijk maken. We willen niet dat we de schakeling zo nauwkeurig moeten maken zodanig dat er schakelingen nodig zijn waar exact 3V uitkomt en niet 3.1V bv. Als je toelaat dat het 3V, 3.5V of 2.5V kan zijn, laat je veel meer variatie toe. Verschillende componenten hebben allemaal verschillende karakteristieken, er zijn fluctuaties. Het gebruik van een bereik laat fluctuaties toe. Op die manier kunnen we dat oplossen en kunnen we veel makklijker ingewikkelde zaken maken, dat hebben we ook als we verschillende transistoren op 1 component willen zetten.
Ook het gebruik beïnvloedt dit: elektronica is heel gevoelig aan temperatuur: enorm groot verschil of de schakeling werkt aan 20 graden Celcius of 40 graden Celcius. Hou er rekening mee dat de schakeling zelf ook opwarmt. Je wil dat uw schakeling blijft werken.
Ook de spanning is niet altijd juist dezelfde.
Waarom 2 bits: om het systeem minder gevoelig te maken. Hoe meer mogelijke waarden, hoe kleiner de bereiken zijn.
We hebben het nog niet over 0 of 1 gehad. Het hoeft niet per sé zo te zijn dat 0 laag is en 1 hoog $\rightarrow$ positieve logica, maar je kan ook negatieve logica hebben waarbij men het omgekeerd doet (0 is hoog en 1 is laag).

Slide 4: Hoe die poorten maken? Los van de elektronica: we kunnen dat maken als we beschikken over schakelaars die we kunnen aansturen. We hebben een schakelaar die 2 standen heeft: open of gesloten. We moeten die kunnen aansturen. → 2 soorten schakelaars.

Slide 5: NMOS-transistor: poort geïsoleerd van het blokje p. Als de spanning laag is zal er niks gebeuren en zal de halfgeleider zich gedragen als een isolator. Wordt de spanning voldoende groot, zal die elektronen aantrekken om een geleidend pad te creëren en wordt er een verbinding gemaakt. Op die manier kunnen we die gebruiken als schakelaar.
PMOS: hetzelfde prentje waar de p en de n verwisseld zijn en de source en de drain verplaatst zijn. Dat zal ervoor zorgen dat de gesloten schakelaar lage spanning betekent: de negatieve spanning is heel groot.
Op die manier kunnen we die 2 soorten schakelaars maken. We krijgen dit op dezelfde geïntegreerde schakeling.

Slide 7: Er poorten mee maken.

1. Enkel gebruik maken van NMOS. Dan krijg je een schakeling zoals op de afbeelding. Stel dat er aan x een lage spanning is, dan zal de transistor (rood) niet geleiden. De uitgang zal via de weerstand verbonden zijn met de voedingsspanning. Als er geen (grote) stromen staan, dan zal de spanning ongeveer gelijk zijn aan de bronspanning. Als er een hoge spanning op zit zal de transistor een lage weerstand hebben (veel lager dan eerst) met als gevolg dat die de uitgangsspanning naar beneden trekt: de weerstand is niet krachtig genoeg om het tegen te houden. Daarom komt er een laag niveau op te staan. Als je dit met positieve logica doet, dan hebben we een invertor gebouwd. Dit is tegenwoordig niet de manier die meestal gebruikt wordt omdat er belangrijke nadelen zijn:

   (a) De weerstand zal nooit helemaal nul zijn. Je krijgt een spanningsdeling over de twee weerstanden. Dat is geen probleem als het maar laag genoeg is, maar het gaat niet echt nul zijn. Dat betekent dat de bovenste weerstand beduidend groter moet zijn want anders krijg je het omgekeerde.

   (b) Dit soort schakeling heeft een statisch vermogenverbruik: we verbruiken ook vermogen op het moment dat er geen veranderingen optreden: alles is constant en toch blijven we vermogen verbruiken. Als er een geleiding is, loopt er een stroom van boven naar beneden. Een stroom door een weerstand betekent vermogenverlies. We kunnen dat alleen maar klein houden als die R zeer hoog is. In het realistische voorbeeld zal het toch 1mW verbruiken, maar een schakeling bestaat niet uit maar 1 inverter/transistor, er zal dus een enorm hoog vermogenverbruik zijn.

   Het wordt niet meer gebruikt behalve bij open-drain.

Slide 8: De weerstand wordt buiten de verpakking weergegeven (stippellijn). Als de schakeling nog niet gemaakt is, hangt die draad los dus vandaar een open drain. Wat is er hier specifiek aan? Als je 1 inverter hebt: niks. Als

je er meerdere hebt en je hangt de uitgangen aan elkaar, dan krijg je bv. de gegeven waarheidstabel. We hebben zo een NOR-poort gemaakt. We noemen dit een wired-AND functionaliteit: we hadden oorspronkelijk 2 op zichzelf staande invertoren. We hebben die uitgangen aan elkaar gehangen en zo een NOR gemaakt. Die NOR kan ook op rechtsonderstaande manier omschreven worden. In het logische schema zit ook een AND-poort. Dit komt omdat de twee uitgangen aan elkaar hangen. Door een bedrading aan te leggen is er een extra functionaliteit. Het nadeel is dat je met NMOS blijft werken en het statisch vermogenverbruik heeft.

Slide 9: Lichtschakelaars in serie of parallel zetten. Als we hier 2 transistoren in serie zetten krijgen we wat op de slide staat. We hebben hier een NAND-poort gemaakt. We kunnen dat ook in parallel zetten, we krijgen dan een NOR: zodra 1 van de twee 1 wordt, wordt de stroom weggeleid.

Slide 10: Stel dat we het statisch vermogenverbruik willen vermijden. Dit kan door te zorgen dat er nooit een geleidend pad is. Dit kan door 2 transistoren te voorzien: 1 boven- en 1 onderaan. Vanboven een PMOS en het ander een NMOS. Het één is een pull up, het ander een pull-down. Hier hebben we geen statisch vermogengebruik want in geen van de twee gevallen is er een geleidend pad van de bron naar onder want in elk geval zal een van de twee open zijn.

Slide 11: Je mag dit nooit met CMOS doen: die werkt alleen als PUN en PDN complementair zijn: als ze alletwee beginnen geleiden, dan loopt er een stroom die bepaald wordt door de weerstand van een transistor die geleidt, wat een zeer lage weerstand is, er zal dus een enorm grote stroom doorlopen. Gevolg: uw schakeling ontploft (wordt getoond in de slides normaal gezien). Je moet dus altijd zorgen dat de twee nooit gelijktijdig geleiden.
Als je de uitgangen aan elkaar hangt kan je dit ook krijgen: ook een grote stroom en minstens één van de twee zal kapot gaan.
CMOS-uitgangen mag je dus nooit zomaar aan elkaar hangen!

Slide 12: Als we ze toch aan elkaar moeten hangen gaan we een speciale oplossing gebruiken. We hebben een inverter gemaakt en een serie- en parallelschakeling. Op deze manier kunen ze nooit gelijktijdig geleiden. Het bovenste PUN is altijd complementair aan het PDN. Ook hier is een NAND-poort gemaakt. Je kan dit ook omkeren: boven serie en vanonder de parallel. Dat leidt tot een NOR-poort. Ook hier zijn de basiscomponenten een NAND en NOR poort.
Interessant hier is dat we daarnet 2 ingangen hadden en 2 transistoren. Hier 2 ingangen en 4 transistoren: per ingang 2 transistoren erbij, dit blijft gelden: per ingang die erbij komt hebben we 2 transistoren nodig. Daaruit volgt dat de kostprijs van een poort bij CMOS evenredig is met het aantal ingangen.

Slide 13: Negatieve logica en actief lage signalen.

Chapter 3

Les 3

3.1 Sides: 3_Technologie

Slide 13: Negatieve logica en actief lage signalen hebben strikt gezien niets met elkaar te maken. Het lage bereik komt overeen met een 1 (negatieve logica).

Slide 14: Impact op de implementatie: die implementatie geeft een NAND-poort. De onderste waarheidstabel hoort daar dus bij en niet de rechtsbovenste. De vertaling van een L naar een nul is positieve logica. Werk je met negatieve logica is het natuurlijk omgekeerd. Je krijgt dan een NOR-poort. In de praktijk werken we omgekeerd: we maken een schema en afhankelijk van de technologie kiezen we voor positieve of negatieve logica. Kiezen we voor positieve logica, dan zal het de afgebeelde afbeelding zijn, anders zal het een andere implementatie zijn.
In sommige schema's zie je een klein driehoekje staan: geeft aan dat die draad met negatieve logica werkt. In de praktijk ga je dat vrijwel nooit terugvinden: ofwel werk je met positieve ofwel met negatieve logica. Het is pas wanneer je gaat mengen dat je die driehoekjes nodig hebt.

Slide 15: Actief lage signalen: heeft niks met het lage en het hoge te maken, het actief zijn heeft enkel te maken met de interpretatie, niet met het binair signaal dat erop staat.
Voorbeeld: wanneer je een reset-signaal hebt (om een systeem in een initiële toestand te brengen: als je de spanning opzet, komt uw systeem in een random toestand. In praktijk wordt de spanning opgezet en onmiddelijk een reset aangebracht om het toestel in een gekende toestand te brengen). Het resetten van een toestand is een activiteit, je moet dat doen. Iets doen kan met een signaal dat 0 of 1 is. Het is niet de logische spanning die op de lijnen staat, maar wanneer iets wordt gedaan: wanneer het een 0 is wat erop staat of wanneer het een 1 is wat erop staat. Dat heeft dezelfde notatie als het inverse, maar daar heeft het dus niets mee te maken. Wij gaan inverse aangeven met ' en actief signaal met *. Als het signaal 0 is, zal de schakeling op de slide gereset worden.
Als je poorten/logische functies gaat bekijken treedt hier voor het eerst het verschil op tussen de poort en de functionaliteit. De functionaliteit beschrijft wat er moet gebeuren in welke omstandigheden (bv. uitgang zal actief zijn bij AND

als beide signalen actief zijn). Bekijken we nu de waarheidstabel op de slide verwachten we normaal een AND-functionaliteit, dat is er een voor actief hoge signalen. Als “er gebeurt iets” overeenkomt met een 0, dan is de waarheidstabel een OR-functionaliteit. Dat is en blijft een AND-poort en de positieve of negatieve logica staat hier totaal los van. De interpretatie van “er gebeurt iets” verandert afhankelijk van of het actief hoge of actief lage signalen zijn. Meestal wordt er gekozen om met actief hoge signalen te werken, dat vereenvoudigt de interpretatie.
Toch zijn er veel plaatsen met actief lage signalen. Er zijn typisch 2 plaatsen waar je het frequent tegenkomt: reset en bij afgesloten verbindingen. Wired OR = wired AND (want je kan geen wired OR maken op zich) met actief lage signalen waardoor die AND poort voor een OR-functionaliteit gebruikt kan worden.

Slide 16: Bedoeling: OR-functionaliteit nodig en die met wired AND organiseren, of ook: met een schakeling vanonder getoond: met RC-schakeling. De schakeling wordt opgezet en de spanning gaat blijven stijgen. In dat eerste deel kan je niks zeggen over de schakeling, daar gebeurt de rommel, je weet niet wat er gebeurt. Een keer die in werkende toestand is, kan je die resetten. De capaciteit gaat proberen de spanning laag te houden en via de weerstand gaat het opgebouwd worden. Die ingang van de schakeling RST* (groene lijn op de grafiek) gaat langzaamaan opbouwen. De spanning bij RST* gaat dus vrij lang laag gehouden worden. Om dat soort zaken te doen moet het een actief laag signaal kunnen zijn.
Je wil een reset-knop hebben waar je op drukt en dan moet dat in orde zijn. Het is makkelijker om het met een wired-AND te doen dan met poorten. Het grote voordeel daarvan is dat je dat eindeloos kan uitbreiden: gewoon een draad erbij leggen, geen poort vervangen ofzo.
In de meeste schakelingen is de RST een actief laag signaal.

Slide 17: Tweede reden waarom er actief lage signalen gebruikt worden: wanneer dingen afgesloten zijn, zit je meestal in een hogere toestand. Op de slide getoond: langere lijnen, maar het hangt af van de frequenties waar je mee werkt. Verbinding: spanning aan de ene kant en verwachten die onmiddelijk aan de andere kant te zien. Bij een transmissielijn plant die overgang zich voort over de lijn en dat duurt een zekere tijd eer die de andere kant bereikt. Dan gaat niet alleen de spanning verhogen aan de andere kant maar eventueel slechts een deel van de energie opgenomen worden door degene die de spanning al dan niet meet en een deel gaat teruggekaatst worden, het zal daar ook gedeeltelijk geabsorbeerd worden en gedeeltelijk teruggekaatst worden.
Wat als je dat op een oscilloscoop bekijkt? → Het rechtse signaal. Hoe meer reflecties er heen en weer gaan, hoe groter de overshoots zijn. Wat we eigenlijk willen is het linkse: je gaat naar een ander niveau en blijft daar. Dat kan je bereiken bij een transmissielijn door ervoor te zorgen dat degene die het signaal moet opnemen dezelfde impedantie heeft als de draden. Het komt erop neer dat je die draden moet afsluiten op de karakteristieke impedantie. Die kan soms redelijk laag zijn.
Hoe sluit je dat af? → Zoals linksonder getoond: aan de ontvangstzijde 2 weerstanden zetten die ervoor zorgen dat de karakteristieke impedantie gelijk is aan

de parallelschakeling van die 2 weerstanden. Welke spanning er op de draad hangt wordt bepaald door de twee weerstanden. Als je uw spanning aanzet zal er 5V opstaan. Als die draad niet aangestuurd wordt, als daar niets mee gebeurt, dan staat daar een 1 op. Dat betekent dat we met actief lage signalen moeten werken: we doen niks en er staat een 1 op.
Vandaar, bij plusverbindingen (die tekening) ga je het regelmatig tegenkomen dat er ook daar met actief lage signalen gewerkt wordt.

Slide 19: Meer complexe schakelingen maken: bij CMOS moet je ervoor zorgen dat als de bovenste geleidt, de onderste niet geleidt en omgekeerd. Je kan ook heel wat andere combinaties van transistoren maken.
Het makkelijkste is te kijken wanneer de onderste tak gaat geleiden: wanneer a & b 1 zijn of wanneeer x, y, z 1 zijn, dan zal het signaal nul zijn.
Dit is 1 enkele poort die iets meer doet dan alleen maar AND of OR implementeren: AOI.
Vanonder altijd NMOS, vanboven altijd PMOS.
Het aantal transistoren dat je nodig hebt is twee keer zo groot als het aantal ingangen. De kostprijs van de schakeling is evenredig met het aantal ingangen. Let wel: ook hier gaat het over inverterende poorten. De uitgangen van de twee ANDs of de twee ORs tellen niet mee als ingang!

Slide 20: Er zijn een aantal dingen die je niet kan maken met 1 enkele poort: je moet een NAND-poort maken gevolgd door een invertor → impact op kostprijs en vertraging.
Buffer: doet op zichzelf niks, maar je kan meer aansturen.
Als je naar een XOR-poort gaat kijken: die gaat 1 zijn als ingang 1 of 2 1 is maar niet als ze alletwee 1 zijn. XNOR zal 1 zijn als beide ingangen 1 of 0 zijn. Hoe maak je dat? Met AND en OR-poorten, maar het kan efficiënter zoals in het blauw: dat is een OR-AND-inverter. Voordeel: bij gewone poorten heb je er 3 nodig met ieder en kostprijs, bij OAI heb je 1 poort nodig en 2 invertoren.

Slide 21: Mogelijkheid: uitgang is zwevend/open/nergens aan verbonden. Zwevende draden zorgen voor problemen dus wil je niet. Omdat ze door niets aangestuurd worden, is er niets dat ervoor zorgt dat die 0 of 1 blijven. Normaal staat daar geen spanning op, maar die reageren als antenne: je steekt de verlichting bv. aan en door inductie verandert de spanning op die draad. Loshangende draden wil je dus niet.
Waarom wil je dat toch kunnen doen (die aan niks laten hangen)? We hebben het al over wired-AND gehad, bij CMOS mag je de uitgangen nooit aan elkaar hangen, maar wat als je de ingangen aan elkaar wilt hangen? De enige oplossing om uitgangen bij CMOS aan elkaar te hangen is door gebruik te maken van een tristate buffer: geeft ingang door aan uitgang zonder functionaliteit, maar die kan de uitgang loskoppelen.
Hoe kan je dat implementeren? Met de getoonde schakeling: geen van beide zal geleidend zijn en dus losgekoppeld van de rest van de schakeling.

Slide 22: Bij een normale aansturing heb je 1 element dat aanstuurt en je op alle plaatsen gebruikt (bovenste tekening). Als je een bus-verbinding hebt (kan

vanop verschillende plaatsen aangestuurd worden, wel van 1 plaats tegelijkertijd), moet je die vanaf 2 kanten kunnen aansturen, nooit tegelijkertijd (want dan heb je uitgangen die aan elkaar hangen!). Als je nu een intelligente sturing hebt die ervoor zorgt dat maximaal 1 of 2 verbonden kan zijn met de bus, dan kan dat wel op die manier. Je kan uitgangen van CMOS aan elkaar hangen als je gebruik maakt van een tristate buffer en die op een intelligente manier aanstuurt.

Slide 23: Praktische aspecten: wat voor soort transistoren we gebruiken,... → heeft impact op de karakteristieken van de schakeling (vooral op het tijdsgedrag, niet zozeer op de logische functie).

Slide 24: Gevoeligheid voor storingen: men zorgt er altijd voor dat er een ruismarge is: we hebben een hoog bereik en een laag bereik. Normaal verwacht je dat als een poort iets aanstuurt, de uitgang van de poort die aanstuurt hetzelfde bereik gebruikt als de poort die zijn uitgang moet bekijken. Die kan dus geen groter bereik hebben bij het produceren van de spanning dan wat gespecifiëerd is. Je kan wel een kleiner bereik hebben. Eigenlijk heb je dus een stuk niet nodig: het groene stuk. Het is toch nuttig om het verschil te maken want dat is de ruismarge. Waarvoor gaan we de gebruiken? Stel dat je een spanning genereert aan de uitgang die verbonden wordt aan de ingang van de volgende. Op die draad ontstaan storingen. De spanning die de andere poort aan de ingang ziet kan hoger/lager zijn dan die gegenereerd door de eerste poort. Wanneer dat naar boven gaat is dat niet erg, maar als dat te laag gaat kan de ontvangende poort dat niet meer correct interpreteren. Die ruismarge dient dus om storing op te vangen. Daardoor wordt uw schakeling minder gevoelig voor storingen. Bij CMOS: hoog bereik: tussen 3.5 en 5V. Je hebt 0.9V ruismarge want de meesten gaan tussen 4.4 en 5V genereren. Bij CMOS is dat mooi symmetrisch maar dat hoeft niet zo te zijn. Bij TTL (transistor transistor logic) is dat niet symmetrisch.

Slide 25: Niet-ideale gedrag van de schakeling: als je een inverter bekijkt en je gaat naar de spanningen kijken. Horizontaal: ingangsspanning, verticaal: uitgangsspanning. L = laag bereik aan in- en uitgang, H = hoog bereik aan in- en uitgang. Wat in het groen aangegeven is is de normale werking: als er een hoog signaal aan de ingang is, zal er een laag signaal aan de uitgang zijn. Als we schakelen (overgaan van 0 naar 1 of omgekeerd) moeten we overgaan van het ene groene stuk naar het andere. We noemden het stuk ertussen ongedefiniëerd, maar die zijn niet ongedefiniëerd, die spanningen. De spanning zal niet met een 0 overeenkomen, ook niet met 1, ook niet met tristate, dat heeft logisch gezien geen enkele betekenis. Als je V_T aanlegt weet je totaal niet wat uw spanning gaat doen, ze gaat in elk geval niet meer digitaal werken, het zal zich analoog gedragen: als een versterker: kleine veranderingen aan de ingang hebben een grote uitgang. Het ideale zou zijn als we geen curve hebben zoals getoond, liever zouden we iets ideaal hebben: hoog tot aan de drempelspanning (V_T, wanneer schakelen we over van laag naar hoog). Zodra we over die drempelspanning zijn, zullen we van laag naar hoog gaan. In het tussengebied is de uitgang ook niet gedefiniëerd. Het probleem van in het ongedefiniëerde gebied te zitten: de schakeling werkt daar niet op een digitale manier en we kunnen niet voorspellen

wat de schakeling daar doet. We moeten daar aan heel hoge snelheid doorgaan: we hebben even een probleem, maar we zijn er onmiddelijk voorbij. We weten dat er bij elke overgang heel even in het gebied gezeten zal worden.
Een ander probleem is dat als je kijkt naar CMOS: dynamisch vermogenverbruik: bij tussenin zal niet 1 van de schakelaars open/gesloten zal zijn, ze zullen alletwee een beetje openstaan en er beiden een beetje tussenin zitten: gaat door alle tussenliggende waarden. Dat is het moment dat zowel de bovenste als onderste tak kunnen geleiden, waarbij je vermogenverbruik krijgt. Dus in het rode gedeelte ga je ook bij CMOS vermogenverbuik hebben. Je hebt dit vermogenverbruik ook alleen wanneer je een overgang maakt van de ene naar de andere toestand (vandaar dynamisch). Die curves zijn ook niet constant, die kunnen variëren omwille van het productieproces maar ook door temperatuursvariaties.

Slide 26: Het is belangrijk om zo snel mogelijk door dat gebied te gaan. Normaal zorgen we ervoor dat aan de ingang die signalen redelijk snel veranderen, dus we gaan daar maar korte tijd inzitten, in die overgang. Jammer genoeg hebben we dat niet altijd in de hand. Als je het signaal opmeet dat van de buitenwereld komt (je hebt een sensor die iets meet en de snelheid waaraan dat verandert wordt door de omgeving bepaald), dan bepaal je dat niet zelf. Het kan dus zeker zijn dat je traag variërende signalen hebt aan de ingang. Dat soort signalen heeft ook typisch wat ruis/verstoringen. Dat gaat niet mooi van laag daar hoog, daar zit variatie op. Als je het reële signaal in een digitale schakeling binnenbrengt en je steekt dat in een inverter, dan krijg je de tweede bovenste grafiek (de middelste). Rond die drempelspanning zal dat werken als een versterker en kleine veranderingen aan de ingang worden grote veranderingen aan de uitgang. Als je dat signaal verder gaat verwerken, dan gaan bepaalde van die piekjes misschien verdwijnen. Dus in plaats van 1 enkele overgang, wat je normaal wilt, ga je heel wat verschillende overgangen zien. Dat is niet wat we willen: we willen een signaal dat op een bepaald moment van laag naar hoog gaat.
Oplossing: Schmitt-trigger-ingangen: speciale implementaties van een ingang: we hebben niet 1 overgang, maar als we van hoog naar laag gaan volgen we de rechtse curve op de rechtsonderfiguur en als we van laag naar hoog gaan volgen we de linkse curve op de rechtsonder figuur.

Slide 27: Het dynamisch gedrag is heel belangrijk want het bepaalt het tijdsgedrag.

Slide 28: Wat hier getekend is, is onmiddelijk nadat er overgeschakeld is: de rode geeft geen doorverbinding meer en de bovenste wel. Je zou verwachten dat de spanning onmiddelijk verandert, maar dat gebeurt niet omdat er vertragende elementen in de schakeling zitten. Als je het elektrische equivalent tekent dan is dat het rechtse: je hebt 2 weerstanden (de rechtse: de ingangsimpedantie, wat je aan de ingang ziet en dus niet de weerstand van de rechtsondergroene transistor op de bovenste tekening). Het is de weerstand tussen de gate en de source. Vermits die van elkaar geïsoleerd zijn, verwacht je dat die zeer hoog/oneindig is. Er is ergens een weerstand tussen de gate en de source. Capaciteit: je hebt er een wanneer je 2 geleiders naast mekaar hebt, dus 2 draden boven elkaar (meestal geen grote capaciteit). Dus overal in de schakeling heb je eigenlijk een

capaciteit.
Overal in de schakeling heb je een grond en een verbinding.
Hoe langer uw draad is, hoe groter uw capaciteit is: bij lange draden is die groot, bij korte draden is die capaciteit klein. Bij de onderste overgang heb je iets verlijkbaar. De ingangsimpedantie blijft altijd.

Slide 29: We hebben dit schema, wat we nu gaan doen is een plotse overgang maken van laag naar hoog. Dat heeft tot gevolg dat waar er oorsponkelijk geen spanning op stond, een spanning wordt aangebracht en die schakeling gaat beginnen reageren. Dit is een RC-netwerk. De spanning gaat exponentiëel oplopen. De spanning tussen de uitgang van de ene poort naar de ingang van de andere gaat geleidelijk overgaan van een laag niveau naar een hoog niveau. De uiteindelijke spanning is niet de blauwe.
De eindspanning zal dan ongeveer gelijk zijn aan de voedingsspanning uiteindelijk.
Voor de rest zie je nog dat die overgang niet ogenblikkelijk gebeurt: het duurt een poosje voor je van het laag bereik in het hoog bereik terechtgekomen bent. Dat is heel belangrijk: de snelheid waaraan je kan werken hangt af van hoe snel je uw bewerkingen afwerkt. Je kan uw spanning niet laten zakken als ze nog nooit hoog gworden is. Hoe trager die curve dus omhoog gaat, hoe langer je moet wachten. We streven er dus naar dat de groene curve zo snel mogelijk stijgt. Dat hangt af van die tijdsconstante τ. Die wordt bepaald door de capaciteit en de twee weerstanden: de parallelschakeling ervan.
Hetzelfde gebeurt wanneer je een H $\rightarrow$ L overgang hebt.

Slide 30: Nu we dat weten, hoe kunnen we er nu voor zorgen dat die overgang zo snel mogelijk gebeurt? Die ingangsimpedantie moet zo hoog mogelijk zijn (groter dan de uitgangsimpedantie). Als uw ingangsimpedantie zeer groot is, dan is het statisch vermogenverbruik verwaarloosbaar klein.
Hoe kiezen we onze uitgangsimpedantie? Hier zitten we met een conflict: stel dat de ingangsimpedantie zeer hoog is, dan zal de tijdsconstante zich verhouden tot R_{OH} * C. We willen R_{OH} dus zo klein mogelijk hebben.
Nadeel aan te kleine weerstand: hoe kleiner die is, hoe groter het ogenblikkelijk vermogenverbuik: ogenblikkelijk grote stromen (P(0)). Je hebt dan over de uitgangsimpedantie de volledige voedinsspanning. Dat zorgt voor een groot vermogen. Het is maar ogenblikkelijk, maar als je plots een heel hoge stroom nodig hebt, kan uw voeding in de problemen geraken want die moet die stroom kunnen leveren (voor 1 is dat geen probleem, wel als je er meerdere hebt). Je moet dus dikwijls een compromisoplossing hebben.
Je kan ook proberen van uw C zo klein mogelijk te maken door korte draden te gebruiken: lange draden zorgen ervoor dat je traag moet beginnen werken.
Uitgangsimpedantie die je zo klein mogelijk wil (R_0): de NMOS-poort is niet erg populair want die heeft statisch vermogenverbruik, maar die is ook qua tijdsgedrag enorm slecht als je dat vergelijkt met CMOS. Waarom is die traag? Je kan geen te kleine weerstand zetten want dan heb je een enorm groot statisch vermogenverbuik, je moet dus een grote weerstand hebben, maar dan heb je dus een slechte tijdsconstante.
Je gebruikt NMOS dus niet tenzij je die echt nodig hebt om die open drain te implementeren.

Slide 31: Dynamisch gedrag van de poort: hoe snel kan je stijgen en dalen: stijgtijd: tijd om van 0 naar 1 te gaan, daaltijd: tijd nodig om van 1 naar 0 te gaan, daar is een zekere tijd voor nodig. We hebben niet een lage spanning en een hoge spanning maar een laag bereik en een hoog bereik, je kan dus meerdere dingen meten: hoe lang duurt het om van het einde van het laag bereik naar het begin van het hoogbereik te gaan,...
De stijg- en daaltijd is iets wat niet zo specifiek is aan de poort. De poort speelt daar ook in mee omdat die capaciteit deels door de eigenschappen van de poort bepaald wordt, maar dat heeft weinig/niks met de complexiteit van de poort te maken. Dat is niet het enige wat het tijdsgedrag bepaalt, het bepaalt hoe snel je kan overschakelen van laag naar hoog of omgekeerd, het bepaalt de maximale snelheid waaraan we kunnen werken. Je ziet op die tekening ook dat die stijg- en daaltijd nogal verschillend kunnen zijn, ze wordt bepaald door de weerstand bovenaan (stijgtijd) terwijl de daaltijd bepaald wordt door de transistor (denk ik??). Vertragingstijd: ik zet een signaal op de ingang, hoe lang heeft die poort ervoor nodig om die berekening te doen en dat op de uitgang te zetten? Dat wordt gemeten als wat er op de ingang verschijnt en wat er aan de uitgang uitkomt.
Het probleem is, hoe meet je wanneer de ingang gebruikt wordt en wanneer het resultaat op de uitgang beschikbaar is? Je hebt ook weer 2 soorten vertragingstijd. In vele gevallen is dit ongeveer hetzelfde, in andere gevallen kan dat verschillen. Als je over *de* vertragingstijd spreekt, spreekt men over het gemiddelde.
Strikt gezien zou je kunnen zeggen dat de vertragingstijd gekoppeld is aan het interne van de poort. De stijg- en daaltijd heeft te maken met hoe poorten met elkaar verbonden zijn. Stijg- en daaltijd bepalen hoe snel je kan werken. Wanneer je dat op deze manier gaat meten, in de gemeten vertragingstijd zit de echte vertraging van de poort in plus hoe snel die verandert.

Slide 32: Korte stijgtijden links: geeft de tijd aan die nodig is om het resultaat te berekenen. Als je een slechte stijgtijd hebt (rechts), dan komt er nog de helft van de stijgtijd bij.
In de gemeten vertragingstijd zitten dus beide effecten in: complexiteit van de poorten en hoe ze met elkaar verbinden. Vandaar dat ook, als je wil gaan rekenen, kan je de vertragingstijd meten en met beide aspecten gelijktijdig rekening houden. In de gemeten vertragingstijd ga je dus altijd een combinatie van de twee hebben. $\Rightarrow$ Heel belangrijk!

Slide 34: Vermogenverbruik: je moet het vermogen kunnen leveren (dat kost aan energie, batterijen,...) en dat vermogen wordt voor iets gebruikt: wordt gewoonlijk omgezet in andere energie, die gaat niet echt verloren. In ons geval betekent dat dat het omgezet wordt in warmte en die moet afgevoerd worden. Als je die niet snel afvoert, gaat uw schakeling enorm snel opwarmen en smelten. Je moet energie dus niet alleen kunnen leveren, ook wegnemen. Voor grote schakelingen is dit ondertussen een van de belangrijkste problemen geworden. Als je weet dat iets op een bepaald moment niet gebruikt wordt, ga je er gewoon de voeding vanaf nemen, maar als je het dan nodig hebt, heb je tijd nodig om in gang te geraken. Voor de rest zijn er heel wat technieken om schakelingen te maken die heel wat minder vermogen verbruiken.

Je hebt dus statisch vermogen: vermogen dat je constant hebt en dat je zeker wil vermijden (dat heb je bij NMOS), bij CMOS is dat verwaarloosbaar klein (de belangrijkste reden om CMOS te nemen). Ze hebben beiden dynamisch vermogenverbruik: als ze schakelen: die capaciteiten moeten opgeladen worden en ontladen worden.
Je kan nagaan hoeveel energie je nodig hebt om een capaciteit van de ene spanning naar de andere te brengen. Dat is $CV^2/2$. Of je dat nu snel doet of traag maakt niet uit, het verschil zit 'm wel in dat als je het snel doet je ogenblikkelijk veel energie nodig hebt op korte tijd. Om die van de ene spanning naar de andere te brengen is $CV^2/2$.
De meeste systemen werken op een klok, hoe dikwijls per seconde kunnen we het signaal veranderen? $\rightarrow$ 2 overgangen (van nul naar 1 en omgekeerd): CV^2 aan energie nodig. Voor praktische gevallen kan het dynamisch vermogen beperkt gehouden worden. Hier moet je er ook rekening mee houden dat niet alle poorten tegelijkertijd schakelen, er is geen enkele schakeling waarbij alle poorten tegelijkertijd schakelen.

Slide 35: In alle schakelingen moet de PMOS vanboven staan en NMOS vanonder, waarom is dat? Als we die zouden omwisselen, kunnen we niet alleen inverterende schakelingen maken, maar ook een AND poort.

Slide 36: Waarom mag dat niet? NMOS-transistoren zijn zeer slechte pull-ups. Een NMOS-transistor is goed om een spanning naar beneden te trekken, PMOS om naar boven te trekken, daarom staan ze ook in die takken.
Waarom is dat zo? Er zijn verschillende redenen. De meest evidente: stel dat we NMOS bovenaan zouden zetten: wanneer gaat de transistor geleiden bij de middelste tekening? $\rightarrow$ Als de spanning voldoende groot is. Om ervoor te zorgen dat we een hoger niveau krijgen moet er een verschil zijn van de drempelspanning van de transistor tussen de gate en de source. Als je bij V een hoge spanning opzet, zal die nooit boven $-V_T$ kunnen geraken, moest die hoger geraken zou de transistor niet meer geleiden. In regimetoestand gaat de uitgang van de linkse poort de hoge spanning $-V_T$ hebben. Wanneer we een bepaalde spanning hierop hebben, wat die ook is, op de uitgang zal altijd een lagere spanning staan. Als je 2 van die poorten na elkaar hebt, dan gaat er hier eerst V_T af en daarna daar V_T erbij. Na een tijd ga je een spanning hebben die lager is dan het lage niveau. Is dat hetzelfde probleem als je die beneden zet? Nee, want daar is het niet de spanning tussen de gate en de uitgang maar tussen de gate en de source en de source ligt daar altijd op nul.
Nog een verklaring: die gaat zich zo instellen dat die altijd een beetje blijft geleiden dus gaat altijd ook statisch vermogenverbruik hebben. De boodschap van de dia is dat je nooit NMOS transistoren bovenaan mag zetten, je moet die altijd onderaan zetten. Hetzelde met PMOS: nooit onderaan.
Daarom kunnen we alleen maar inverterende basispoorten maken.

Slide 38: Fan-in: aantal ingangen. Waarom komt dat van pas? Je kan niet een eindeloos aantal ingangen hebben, je kan geen poort maken met 100 ingangen omdat hoe meer ingangen je hebt, hoe slechter de poort gaat werken: de vertragingstijd gaat stijgen: hoe meer ingangen je hebt, hoe meer transistoren je hebt (ingangen $\sim$transistoren), ieder van die transistoren heeft een interne

capaciteit,...
Ander probleem: als je kijkt naar de tekening rechts boven: een groot aantal ingangen betekent een groot aantal transistoren in serie. Ieder van die transistoren is niet-ideaal: zelfs als die geleidt zal de spanning over die transistor niet exact 0V zijn. Als je er verschillende boven elkaar zet gaat dat allemaal optellen en kan je in het hoog bereik terechtkomen hoewel dat niet de bedoeling is. We gaan daar rekening mee moeten houden, een oplossing voor mogen zoeken.
Fan-out (!!!): $\neq$ aantal uitgangen. Fanout = # ingangen dat aan die uitgang verbonden is. Waarom zijn we daarin geïnteresseerd? Ook dat zal onze snelheid gaan bepalen: hoe meer ingangen er aan de uitgang hangen, hoe groter de capaciteit, hoe trager de schakeling werkt. Er is maar een maximale stroom die geleverd kan worden. Dus hoe groter de fanout, hoe trager de schakeling gaat werken. Soms kunnen we daar niet aan ontsnappen: zolang je op een geïntegreerde schakeling werkt, heb je niet veel energie nodig om het te laten omschakelen. Een keer je vanaf een geïntegreerde schakeling naar een andere gaat, heb je een lange verbinding en heb je een veel grotere capaciteit. Wil je dus een hogere snelheid halen op een PCB moet je een andere methode zoeken. Het voordeel van op de chip te werken is dat je aan een grote snelheid kan werken. Gegevens uitwisselen met de buitenwereld (dus buiten de chip) gebeurt niet aan een hoog tempo om deze reden.
Je kan dat wat verhelpen door ervoor te zorgen dat je een schakeling hebt die wel grote stromen kan leveren: een poort die een enorm kleine uitgangsimpedantie heeft. Een driver/buffer dient om veel stroom te kunnen geven. Die stroom heb je nodig om dingen met een hoge capaciteit toch snel te kunnen laten veranderen. Een keer je vanaf uw schakeling naar uw PCB gaat of van PCB naar randapparatuur heb je buffers nodig die voldoende stroom kunnen leveren om een redelijke snelheid te halen.

Chapter 4

Les 4

4.1 Sides: 3_Technologie

Slide 39: We hebben besproken hoe poorten gemaakt worden in NMOS en CMOS en welke impact de parameters hebben. Buiten de praktische kanten is er nog een ander aspect: de technologieën: hoe wordt dat geïmplementeerd? Er zijn ook heel wat andere mogelijkheden.

Slide 40: Als je dat gaan bekijken zijn er 3 grote categorieën die gebruikt worden als het over implementatie gaat: standaard chips, programmeerbare logica en specifieke chips.
Standaardchips zijn de oudste. De bedoeling was dat je de componenten kocht en die zelf samenzette op een PCB om op die manier een schakeling te maken. Ondertussen is dat wat in onbruik geraakt (die standaardcomponenten worden wel nog gebruikt, voornamelijk als lijm tussen de andere: glue logic). Als je naar een PCB kijkt, dan staan daar een paar belangrijke componenten op die zorgen dat de chip iets specifiek doet. Je hebt een component met verwerkingskracht, uitbreiding,... Ook een klok nodig die gegenereerd moet worden, buffers om meer stroom te leveren wanneer je naar andere componenten gaat of wanneer je van het bord afgaat. Dat is typisch die glue logic, de rommel die erbij staat die functioneel niet erg belangrijk is maar toch nodig is om het te doen werken. Daar zit niet veel evolutie meer in en we gaan dat verder ook niet meer bespreken.
Specifieke chips: andere kant van het spectrum. Dat is een enkele component die gans het systeem omvat. Het nadeel ervan is dat het erg duur is om te maken. Het ontwerpproces is redelijk lang, vereist heel wat kennis en het maken vergt ook veel tijd en energie. Daarom ga je dat alleen doen als je die ontwerpkost kan afschrijven op een heel groot aantal componenten. Bv. als je een microprocessor maakt kan je die voor vanalles en nog wat verkopen.
Je hebt ook een tussencategorie voor de kleine aantallen of voor een beperkt aantal ((tien)duizenden componenten), daarvoor zijn de specifieke chips te duur en de standaardchips zijn te eenvoudig. Daarvoor hebben we programmeerbare logica (is geen microprocessor of waarop je kan programmeren), het is een logica waarop je de functionaliteit kan programmeren. Je kan de functionaliteit van de component wijzigen. Dat heeft het voordeel dat je de component voor heel wat toepassingen kan gebruiken. Door de stijging van het integratieniveau kunnen

we daar heel hoge complexiteiten mee halen.

Slide 41: Als we iets specifieker gaan kijken, de specifieke chips, daarbij is maatwerk het meest efficiënte: zo maken dat die optimaal werkt qua snelheid,...

Slide 42: Je gaat zelf bepalen waar de transistoren allemaal staan, hoe die verbonden zijn, hoe die poorten gaan vormen,... Dat is heel complex (je gaat de ganse schakeling in één keer maken), maar het is het meest efficiënte. Door het feit dat je geen overzicht meer hebt, is dat niet goed voor grote schakeling. Het kan wel gebruikt worden als onderste laag die de fabrikant meelevert.
Uit de wet van Moore volgt dat er om de 1.5-2 jaar een verdubbeling is van integratie. Als je op het laagste niveau van transistoren gaat werken, ben je verplicht om uw ontwerp om de 2 jaar helemaal opnieuw te maken, anders ben je niet meer mee.

Slide 43: Je kan gebruik maken van standaardcellen (het niveau juist erboven, boven de poorten. De poorten kan je uit een bibliotheek nemen). Om het op een structurele manier te regelen wordt het gebruikt in cellen. De hoogte is hetzelfde, de breedte variëert wat en tussen de rijen zitten gaten, daar worden geen transistoren gebruikt, die ruimte dient om de draden te leggen. Je hebt maar een beperkt aantal metalisatielagen die je boven elkaar gaan leggen, je kan draden niet zomaar boven elkaar leggen/elkaar laten kruisen (zou het veel complexer maken), daarom voorzie je plaats tussenin, dan moet je niet in de hoogte werken.
Hoe geberut het ontwerp hier? Je moet eerst een logisch ontwerp maken met alle poorten etc. De volgende stappen zijn:

1. Bepalen waar welke poort ligt: het is niet omdat je 3 AND-poorten hebt en een OR-poort dat je eerst de ANDs moet leggen en dan de OR. De vraag is wat het beste is: hoe kunnen die verbonden worden met elkaar? We hebben bij voorkeur korte draden zodanig dat de schakeling zo snel mogelijk werkt.

2. Het plaatsen van die componenten hangt samen met hoe die draden kunnen lopen. $\rightarrow$ Bepalen waar de draden liggen.

Die twee dingen liggen samen, ontwerp je samen. Dit is een niveau dat al veel makkelijker is om ontwerpen te maken. Dit wordt veel toegepast op een normaal ontwerp.

Slide 44: Je kan nog een stap verder en dat is de gate array. Het gaat er altijd om hoe je het goedkoper en sneller kan laten gebeuren. Je kan een beperking opleggen aan het aantal componenten.
Waarom geen component maken met alleen maar NAND-poorten op? Het voordeel daarvan is dat je dat één keer op voorhand kan maken. Welke schakeling je ook wil hebben, de poorten liggen al vast (vandaar een gate array). Wat moet er dan nog gebeuren? De verbindingen moeten nog gelegd worden. Het grote voordeel is dat de meest dure en complexe stappen al gedaan zijn. De laatste stappen zijn die draadjes leggen bovenaan, dat is goedkoop en eenvoudig. De fabrikant kan een ganse reeks chips maken waar al die poorten opstaan en per

gebruiker zal er een andere metalisatie op gelegd worden.
Het legt beperkingen op aan wat je kan gebruiken.

Slide 45: Een andere manier van implementeren die redelijk veel gebruikt wordt (en we gaan toepassen in het labo) is die van programmeerbare chips. We gaan hier geen transistoren verschuiven.

Slide 46: Wat je wel kan doen is: stel de poorten liggen vast, je kan de verbindingen wijzigen: je kan zo andere schakelingen maken. Je kan dit ook doen bij een component die helemaal klaar is. Je kan ervoor zorgen dat verbindingen wel of niet onderbroken zijn (dat is hetzelfde als een verbinding leggen, maar het is soms makkelijker om te onderbreken).

- Zekeringen: voorkomen dat er teveel warmte onstaat en daarom schade. Een goede zekering is het eerste wat kapot gaat in uw toestel. Als er een te grote stroom door de zekering loopt, dan brandt die door. Je kan een zekering zetten op een draad en als we niet willen dat daar een verbinding is, dan zetten we daar even een grote stroom op zodat die zekering doorbrandt. Het nadeel is dat dit onomkeerbaar is. Als je later vaststelt dat je een fout gemaakt hebt kan je die in principe niet meer wijzigen, tenzij het enige wat moet gebeuren het onderbreken van extra verbindingen is. We veranderen hier dus niet de poorten, enkel de verbindingen.

- Op een reversibele manier: we kunnen een transistor laten geleiden of isoleren. Dat komt op hetzelfde neer als de zekering die wel of niet doorgebrand is. Als je zo een transistor op een draad zet en je stuurt die op een juiste manier aan, kan je ervoor zorgen dat die verbinding er wel of niet is. Je moet alleen de spanning op de gate veranderen om te zorgen dat de verbinding al dan niet gemaakt is. In de meeste gevallen blijf je wel bij uw keuze, dus ook als je de spanning afzet en terug opzet, wil je dat dat behouden blijft, je wil dat op een niet-vluchtige manier doen. Je kan dezelfde techniek als in flashgeheugen hier toepassen: 2 gates: de bovenste is verbonden met de buitenwereld, de middelste is totaal geïsoleerd (floating gate). Hoe kunnen we die nu gebruiken? Dankzij quantummechanica weten we dat het kan gebeuren dat een elektron op een bepaald moment op een plaats is en op een ander ogeblik ergens anders zonder dat het ergens tussenin geweest is. We kunnen daarvan gebruik maken. Door de juiste spanning op te leggen kunnen we ervoor zorgen dat elektronen die zich op de bovenste poort bevinden zich plots op de onderste poort bevinden. We gaan elektronen overpompen van boven naar die middenste poort. Als je de spanning afzet, kunnen de elektronen niet meer wegtunnelen (dat kan alleen als je de juiste spanning aanlegt). Als je dan de spanning opnieuw aanlegt, dan zullen die elektronen weer geleiden. Je kan dit ook omkeren: die elektronen terug naar de bovenste poort laten gaan. Op die manier kan je een transistor gebruiken om verbindingen te maken en onderbreken. Je kan dit omkeren, maar dat gaat niet zo simpel (vrij traag). Je kan dat ook niet blijven doen, uiteindelijk krijg je daar toch neveneffecten van en dan werkt dat niet zo goed meer, maar normaal wil je dat ook niet.

- Tegenwoordig vind je SRAM het meeste terug: aansturen vanuit een geheugen. We hebben een transistor die voor de verbinding zorgt, een

geheugen dat aangeeft of de transistor moet geleiden of niet. Als dit een statisch RAM-geheugen is, gaat de informatie verloren wanneer je de spanning afzet. Je kan ergens bijhouden (in een niet-vluchtig geheugen) wat wel en niet moet doorlaten. Tijdens het resetten wotdt die informatie gewoon overgepompt naar uw SRAM. Het voordeel daarvan is dat het herprogrammeren heel vlot verloopt: gewoon nieuwe informatie in het RAM-geheugen inladen. Dat kan aan hoge snelheid en je kan het vaak veranderen. In extremis kan je die component dat zelf laten doen, maar dat is erg moeilijk want het is moeilijk bij te houden wat er nu juist aan het gebeuren is.

Slide 47: Hoe ziet dat eruit? Een laag AND-poorten gevolgd door een laag OR-poorten, daarmee kan je elke mogelijke schakeling maken. Het enige wat je hierbij moet doen is programmeren hoe de ingangen met elkaar verbonden zijn (via de uitgangen). Dat is dan het programmeren en daarmee kan je een schakeling maken.
Als we gaan kijken naar de kostprijs: de poorten gaan niet meer de kostprijs bepalen, wel de plaatsen waar je al dan niet verbindingen maakt. Het wordt dikwijls zo getekend, hoe interpreteer je dat: een beknopte schrijfwijze om de rechtse tekening weer te geven. Tussen ieder van die dingen (bolletjes) zit een zekering. Je kan er sommige dan opblazen en dan krijg je kruisjes en dan weet je wat al dan niet gaat geleiden.
Het is een XNOR-poort op de slide. Kruisje: zekering die is doorgebrand.
Problemen: snelheid en betrouwbaarheid. Hoe groter die matrices worden, hoe langer de afstanden worden en hoe trager het systeem wordt. De snelheid is dus veel lager dan de rechtstreekse implementatie.
Betrouwbaarheid: hoe test je een zekering: opblazen en testen of het goed gebeurd is. Een keer je dat gedaan hebt moet je die eigenlijk weggooien. Je kan dat dus nooit helemaal testen: bij een paar testen en aannemen dat de rest ongeveer ook zo zal zijn.
Die lagere snelheid dan rechtstreekse verbinding en onbetrouwbaarheid zorgt ervoor dat men die volledige flexibilteit niet altijd zal gebruiken, sommige dingen zal men vastleggen: de OR-poorten onmiddelijk aan de AND-poorten hangen bv.
Dan krijgt dat andere namen: PAL: vaste OR-matrix.

Slide 48: Naarmate dat we naar een hogere integratie konden gaan, is men gaan beseffen dat je alles kan implementeren met tweelagen-logica, maar als je bv. een flipflop wil gaan maken, heb je daar 8-10 poorten voor nodig. Als je dat via de trage AND-OR verbindingen gaat doen, werkt dat heel traag. Als je weet dat je toch in 80% van de gevallen een geheugenelement nodig hebt, zet dat er dan gewoon bij. In die overige 20% ga je dat niet gebruiken en staat het wat in de weg, maar voor die 80% verantwoordt het wel die extra meerkost.
Men gaat de dingen waarvan men verwacht dat die regelmatig gaan gebruikt worden erbij zetten. Op de slide: 1-bitgeheugen erbij gezet en een selector die zegt of het geheugenelement gebruikt wordt of niet. Men spreekt dan van macrocellen: de uitgang wordt complexer en er zitten verschillende functionaliteiten in.
We kunnen alsmaar meer op een chip zetten. Als we die matrix gaan uitbreiden,

gaat de chip eerder met de matrix volstaan, niet zozeer met de schakelingen. Je hebt altijd een paar componenten die met elkaar verbonden moeten zijn, maar niet alles moet altijd met alles verbonden zijn. Die heel grote matrix is dus niet altijd efficiënt.

Slide 49: Soms is het interessanter om verschillende kleinere dingen te nemen. In plaats van alles groter te maken, gaan we verschillende kleine matrices gebruiken zodanig dat die lokaal verbonden kunnen worden en weinig verbonden moeten worden met de rest. Sommige dingen gaat men dan onderling wel nog gaan verbinden.

Slide 50: Wat we tot hiertoe gedaan hebben is de logica, de poorten die erop staan, die staan vast. Het enige wat we konden programmeren waren de verbindingen. We kunnen nog een stap verder gaan, dat is de meer recente programmeerbare logica. We kunnen niet alleen de verbindingen programmeren, maar ook de poorten. Dan komen we tot de FPGA.
Als je gaat kijken heb je een matrix van logische blokken. Dat is die basiscomponent waarvan je ook de functionaliteit (die nog niet bestaat) kan bepalen.
Het is field programmable: gelijk waar programmeerbaar, je moet daarvoor niet naar de fabriek gaan, geen heel speciale apparatuur voor nodig. Het is in dit soort schakelingen dat daarom die transistoren aangestuurd worden vanuit een geheugen.
Wat zit daarin? Je hebt enerzijds de logische blokken en iedere logische blok kan een bepaalde functionaliteit hebben. Je kan er natuurlijk maar iets mee doen als die verbonden zijn met elkaar. Je hebt 2 soorten verbindingen:

- Lange lijnen die lopen over de ganse component want soms kan het gebeuren dat je informatie moet delen met heel wat andere logische blokken en dan ga je typisch een busstructuur gebruiken en die gaat op verschillende plaatsen gebruikt worden. Op die ogenblikken is zo'n lange lijn heel interessant.
- Als je toch snelheid wil hebben, moet je kortere verbindingen hebben. Daarvoor gebruiken we de schakelmatrices: die gaan u de mogelijkheid geven om elke lijn die erop toekomt te verbinden met elke andere lijn, die kan dat zelfs met meer dan een andere lijn verbinden. De lichtere dingen die er nog tussen staan (SMc) dienen om de horizontale en verticale lijnen te raken, de verbinding naar de lijnen die naar de schakelmatrices gaan.

Je hebt bijkomend heel korte verbindingen die rechtstreeks 2 naburige blokken verbinden.
Je moet nog naar de buitenwereld gaan, dus je hebt ook IO-blokken nodig.
Als je dat gaat bekijken binnen een ontwikkelomgeving: de blauwe dingen op slide (via animatie komt die erop te staan): logische blokken. Paarse lijnen zijn de lange lijnen die over de ganse lengte doorlopen. Om een bus aan te sturen met meerdere, dan moet dat met tristate buffers, dat zijn die kleine driehoekjes. Die staan erop, of je die nu gebruikt of niet. De witte lijnen zijn de kortere lijnen. De schakelmatrices maken dus gewoon verbindingen tussen lijnen. De blauwe puntjes (nauwelijks zichtbaar) maken ook verbindingen.

Slide 51: Als je weet dat bepaalde dingen veel gebruikt worden ga je dat niet met basiscomponenten implementeren, dan zet je dat daar gewoon bij.

Gewoonlijk ga je sequentiële schakelingen maken. Wat heb je daarvoor nodig? Geheugen (op die van de labo's zit extra geheugen: het groene op de tekening). Sequentiële schakelingen werken op een klok, dus dat heb je ook nodig (zit er automatisch op, dat is dat zwart blokje DCM). Dat kan nog verder gaan: vermenigvuldiger. Een optelling kan je redelijk efficiënt met die logische blokken maken, een vermenigvuldiger niet, daarom zijn er vermenigvuldigers bijgezet.
Als je naar complexere toepassingen gaat kijken kan er veel meer opgezet worden: meer geheugen, alle componenten die je denkt nodig te hebben: aansturing van ethernet,...
Wat als je bepaalde componenten niet nodig hebt? Je hebt verschillende soorten FPGA's: met bepaalde componenten of zonder.

Slide 52: We hebben 2 manieren om te programmeren: het maken van verbindingen en bijkomend een basiscomponent waarvan we de functionaliteit ook kunnen aanpassen. Dat geeft enorm veel flexibiliteit.
Daar zitten dus geen poorten in waarvan je de schakeling kan wijzigen!
Wat hier is weergegeven wordt geïmplementeerd met een geheugen: een opzoektabel implementeren. Dat geeft voor elke combinatie van de ingangen (hier 4 bits ingangen dus 16 mogelijkheden) aan of de uitgang 1 of 0 moet zijn. Die lookup table bepaalt dus de functionaliteit.
Wat je nu ziet is de waarheidstabel. Op het moment dat je de waarheidstabel hebt moet je niet meer nadenken wat het wordt in termen van AND en OR, je steekt dat in het geheugen en de component doet wat hij moet doen. Je hebt wel de beperking dat je meestal maar met 4 ingangen kan werken. Elke logische blok kan dus gelijk welke functie van 4 ingangsvariabelen realiseren. Dit is de manier waarop je een programmeerbare component kan maken.

Slide 53: Spartan-3: je hebt een logische blok (CLB: configurable logic block) en elke slice is opgedeeld in 2 logische cellen, zie **Slide 54**.

Slide 54: Binnen 1 configureerbare logische blok zitten 8 logische cellen in, dus je gaat altijd 8 logische cellen met een keer gebruiken. Als je hiernaar kijkt kan je denken dat het nodeloos ingewikkeld gemaakt is. Ook hier geldt: wat zetten we erbij? $\rightarrow$ Dingen die we goed denken te kunnen gebruiken.
Door er een beetje hardware bij te zetten (in het lichtgroen) kan je met een logische cel een optelling maken. Het is de moeite waard om dat te hebben.
Evengoed hebben we dikwijls extra geheugen nodig, dat is in het blauw aangegeven.
Dat rode stukje: die kiest tussen 2 ingangen. Door alleen deze erbij te zetten kan je die 2 logische cellen combineren tot iets nieuw. Als je dus 2 logische cellen hebt, heb je 2 keer een functie van 4 variabelen. Maar heel dikwijls heb je meer variabelen nodig. Door die rode selector erbij te zetten, kan je die 2 samenzetten tot 1 functie van 5 variabelen: je splitst uw waarheidstabel in 2 op: 1 waarbij de meest beduidende variabele 0 is en een waarbij de meest beduidende variabelen 1 is. Op die manier, door het feit dat die selector daar wijzigt, kan je een fucntie van 5 variabelen realiseren.

Slide 55: Schakelmatrices: de verbindingen worden gemaakt/verbroken met transistoren. Hier is het getekend met NMOS-transistoren, in de realiteit is

dat ingewikkelder. Je kan het programmeren vanuit een RAM-geheugen dat die poorten aanstuurt.

Slide 56: Je hebt een ontwikkelomgeving specifiek voor het soort componenten dat de fabrikant verkoopt.

Slide 57: CAD-omgeving waarbinnen je het volledige ontwerp kan doen en dat je zoveel mogelijk zal helpen om alle taken te doen. De basistechnieken zijn al geïmplementeerd in de software en zullen voor u verzorgd worden.
Linksboven: project met bestanden. Je kan daar hiërarchisch in werken. Je kan daar verschillende dingen op doen. Je hebt ook een venster met het resultaat.
Rechts: overzichtsschema met ofwel schema ofwel resultaat van simulaties.

Slide 59: Je bent niet verplicht om gebruik te maken van 1 manier van input ingeven. In het voorbeeld op de slide is alles wat gecombineerd. Je hebt een topschema met een paar dingen die al een beetje uitgwerkt zijn, andere dingen zijn nog gewoon blokjes met een subschema waarop je kan klikken, dan wordt een lager niveau geopend en kan je dat ook in detail zien.
Ook nog andere dingen die afhangen van fabrikant tot fabrikant zoals bv. die logiCORE, soort macro. Om dat allemaal afzonderlijk in de bibliotheek te hebben, gebruikt men een soort macro. Dan wordt er nog een venster geopend waarin je instellingen kan maken (zie transitie slide). Dit komt een beetje overeen met VHDL. Je moet niet weten wat daar allemaal onder zit, de omgeving zal daar rekening mee houden.
Hetzelfde met de specifieke klokinstellingen.
Als we het over sequentiële schakelingen hebben gaan we in eerste instantie toestandsdiagrammen gebruiken. Je kan zelf een toestandsdiagram tekenen en die software weet hoe je dat toestandsdiagram efficiënt moet omzetten naar hardware. Alle tussenliggende stappen moet je dus niet doen. Je kan ook een VHDL-beschrijving doen, soms is dit veel compacter.
Je kan dit allemaal gaan combineren in uw schema.

Slide 60: Je kan in die omgeving een logische simulatie doen. Dan krijg je daar de gegeven tekening uit: je kan ingangssignalen aanleggen, zien waar er een reset gebeurt,... Het telt hier op een decimale manier.
Je kan ook elk bitje afzonderlijk bekijken hoe het zich in de tijd gedraagt maar dit is geen tijdsgedrag, alleen een benadering want het toont alleen de logische werking. Er staat geen timinginformatie. Zelfs als hier tijden bij vermeld staan, is het geen echte timinginformatie.

Slide 61: Als je daarmee tevreden bent, moet je uw fysisch ontwerp gaan doen. Je moet gaan zeggen hoe je alles in de logische cellen steekt. Dat heeft dus niks met poorten te maken. Wat als je heel complexe berekeningen hebt op 27 bits, wat ga je in welke cel steken, hoe ga je ze verbinden, waar ga je welke cel zetten,...? $\rightarrow$ Moet allemaal bepaald worden.

Slide 62: Dat gebeurt automatisch. Er zijn verschillende stappen die moeten doorlopen worden: alle schema's moeten samengezet worden. Dan moet dat vertaald worden naar logische primitieven, men moet er een logisch ontwerp

van maken: poorten en flipflops. Maar er staan geen poorten op uw FPGA. Dan komen de specifieke dingen: wat kan je het best gebruiken, hoe ze je die logische schakeling het best om?
2 stappen: logische schakeling maken en dan die zo goed mogelijk vertalen naar wat er op die FPGA komt. Dan heb je bepaald wat op die cellen moet komen en hoe ze verbonden zijn. Maar dan heb je hetzelfde probleem: welke cel zet je op welke plaats?
Hier heb je eveneens een placement en routing stap. Als dat allemaal gedaan is, is uw fysisch ontwerp af, je hebt dan alle informatie: je weet welke cel wat doet, hoe die met andere cellen verbonden is en als fabrikant ken je de specificaties van die logische verbindingen en kan je het tijdsgedrag gaan voorspellen.

Slide 63: Simulatie doen die eruit ziet als getoond, hier is de tijdsinformatie wel de echte tijdsinformatie. De piekjes die je ziet waren er daarnet niet.
Transitie slide: de twee enen moeten nul worden en de tweede nul moet 1 worden. Het tijdsverloop toont wanneer welke bit verandert: er is altijd maar 1 signaal dat tegelijkertijd verandert. Alle piekjes die je zag op het vorige beeld (voor de transitie) zijn ook niet zo erg, de tijdssimulatie zegt u alleen wat zal gebeuren, jij moet dan beslissen of dat OK is of niet.

Slide 65: FPGA: omcirkeld (en helemaal zichtbaar). De rest zijn allemaal dingen om te experimenteren, drukknoppen,...

4.2 Slides: 4_Combinatorisch

Slide 1: We gaan beginnen ontwerpen met de eenvoudigste schakelingen: combinatorische schakelingen. De ingang verandert en zal impact hebben op de uitgang, maar houdt geen rekening met het verleden: heeft dus geen geheugen.

Slide 2: Alle dingen die op schema's zijn uitgelgd, hoe ga je dat in een hardwaretaal beschrijven? We beginnen met de combinatorische schakelingen, we willen die minimaliseren.

Slide 3: De eerste vraag die je je kan stellen is: waarom zou je dat willen? Het is omwille van de randvoorwaarden dat je wil minimaliseren. We streven naar een schakeling die zo goedkoop mogelijk is, zo snel mogelijk werkt en zo weinig mogelijk vermogen verbruikt. Het derde is niet vermeld op de slide, dat is evenredig met de capaciteit maal de frequentie maal de spanning in het kwadraat.
Hoe gaan we zorgen dat we zo weinig mogelijk vermogen verbruiken? Als we de frequentie en de spanning constant houden, hangt het dus af van de capaciteit. Die hangt af van de grootte van de component. Als we dus zo compact mogelijk werken, zullen we ook een minimum aan vermogen verbruiken.
Om 2 implementaties te kunnen vergelijken moeten we een maat hebben om de kostprijs en snelheid van een schakeling te kunnen bepalen.
De kostprijs is het makkelijkste: we hebben gezien dat de kostprijs van een poort evenredig is met het aantal ingagnen bij CMOS. Het hangt er wel vanaf welke technologie gebruikt wordt. Als het een niet-inverterende poort is, die kost nog

eentje extra (alleen voor CMOS). Gaan we naar iets totaal anders kijken zoals de FPGA, daar telt het aantal transistoren niet. Daar hangt de kostprijs er vanaf of je een kleine of een grote FPGA kan gebruiken (met kleine logische cellen of grote). Het is hier evenredig met het aantal logische cellen. De kostprijs wordt hier dus anders bepaald. Dit is moeilijk omdat het moeilijk is om te schatten hoeveel logische cellen je gaat gebruiken.
In de praktijk moet je je daar niet teveel zorgen om maken. Het eerste (CMOS) is dus meer als je zelf een geïntegreerde schakeling gaat maken.
De kostprijs van de schakeling is de som van de kostprijzen van de afzonderlijke elementen.

Slide 4: Vertraging: moeilijker. Vertraging is niet alleen een functie van de complexiteit van de poort, maar ook van de stijg- en daaltijd. Die hebben te maken met hoe die poorten verbonden zijn met elkaar. Dat kan je nooit op een logisch schema zien, dat kan je alleen berekenen indien je een fysisch ontwerp hebt. Om dat toch enigzins te kunnen vergelijken, is er een benadering. Er is een stuk dat evenredig is met het aantal ingangen van de poort en dat is de complexiteit van de poort die je in rekening brengt (hoe meer transistoren, hoe trager het zal zijn). Er staat ook nog een constante bij (0.6 en 1.6) om rekening te houden met de verbindingen. Het probeert beide dingen dus te combineren. Dit geldt weer alleen voor CMOS, niet voor FPGA, daar is het te complex.
De totale vertraging is de som van de vertragingen langs het kritisch pad: dat pad van een ingang naar een uitgang dat voor de grootste vertraging zorgt. Als je dat pad volgt, dat is de traagste manier om van een ingang naar een uitgang te gaan. Het is het traagste pad dat de maximale verwerking bepaalt.
Bv. voor die XOR: donkerblauw is 1 poort (geen 3). Van die één enkele poort kan je de kostprijs en de vertraging berekenen. De poort heeft 4 ingangen en is inverterend, dus $0.6 + 4 * 0.4$. De vertraging van de ganse schakeling is die door ook nog de inverter. Er zijn hier 2 kritische paden: door de inverters en ze hebben dezelfde vertraging (anders hadden we geen 2 kritische paden), dus nog +1.

Slide 6: Karnaughkaarten: wat we tot hiertoe gezien hebben is dat je dat kan oplossen door algebraïsch te werken met Boole-algebra. Maar dat is niet echt handig omdat je niet weet welke theorema's je in welke volgorde moet gebruiken om tot het minimum te komen. De waarheidstabel toont het al meteen in sommige gevallen. In het voorbeeld zie je dat de uitgang 1 is in de 5^e en 6^e rij. Om die twee gevallen te combineren kan je gewoon zeggen dat f=1 als x=1 en y=0, z doet er niet toe. Je kan dat natuurlijk niet zomaar in alle gevallen zien. Je kan evengoed de groene illustratie zien: f=1 als z=0, maar dat is minder makkelijk om te zien omdat het niet meer naast elkaar ligt.

Slide 7: N-kubus met zoveel dimensies als er variabelen zijn. Als we maar 2 dimensies hebben is het doenbaar, ook bij 3, maar daarna is het niet meer te zien. Visueel ben je dus niets met een N-dimensionale kubus, maar het principe is niet slecht.

Slide 8: Oplossing: N-kubus die je platduwt: je maakt die 2-dimensionaal. Op de slide heb je een 3-D kubus die is platgeduwd. 000 ligt naast 001, dat

moet in het 2-D geval nog altijd zo zijn! We kunnen hier heel makkelijk tot 4 variabelen weergeven. Als je 2 dimensies hebt en je wil dat uitbreiden, ga je spiegelen rond een bepaalde as.

Slide 9: Spiegelen: de spiegeling van 1 is 3.
De mens is heel goed in het herkennen van patronen, maar je moet die wel kunnen herkennen. Je hebt de binaire voorstelling en gaat telkens 1 bit veranderen. Als je 5 bits hebt, heb je 5 mogelijkheden: 5 groene veldjes. 5 variabelen gaat minder makkelijk: wat ligt er allemaal naast 5: alle rode. Dat 21 ernaast ligt is minder evident om dat te herkennen. Die ligt niet alleen in het andere vierkant, maar ook gespiegeld in het andere vierkant. Dat maakt het soms heel moeilijk om het nog visueel te interpreteren.
In plaats van te spiegelen kan je ook gewoon dupliceren. In dat geval moet je wel gewoon op dezelfde plaats kijken en dat is al wat makkelijker.

Slide 10: Werkt goed voor een beperkt aantal variabelen, maar eens je meer variabelen hebt dan 4, wordt het moeilijk om het visueel te kunnen oplossen.
Als je een Karnaughkaart met 6 variabelen moet oplossen op het examen, kun je met 99.999% zekerheid zeggen dat je een fout hebt gemaakt.

Slide 11: Hoe gaan we dat gebruiken? We hebben dat daarnet gemaakt om visueel te zien wat naast mekaar ligt. Als je van een logische functie vertrekt, kan je die gebruiken om uw kaart op te vullen. In zo'n kaart ga je zo'n groot mogelijke gebieden maken die een macht zijn van 2. Hoe groter dat gebied is, hoe meer variabelen je in je poort kan laten vallen. Als je dat op het voorbeeld bekijkt: gebied van 4. Hoe goed je ook zoekt, je vindt geen gebied van 8. Dan zoek je nog naar de anderen: onderaan kan je samennemen en ook verticaal. Er mogen gebieden overlappen! Zo heb je een vorm gevonden die veel compacter is dan uw oorspronkelijke beschrijving.
Karnaughkaarten zijn gebaseerd op het herkennen van patronen. De vraag is of er nog andere patronen te herkennen zijn die nuttig zijn voor een implementatie: dambordpatroon. Dat komt overeen met een XOR.

Chapter 5

Les 5

5.1 Slides: 4_Combinatorisch

Slide 12: We hadden het over combinatorische schakelingen waarbij de uitgang afhankelijk is van de ingangen, er is geen voorgeschiedenis.
We hebben Karnaughkaarten bekeken, we gaan dat nu iets verder in detail bekijken, want voorlopig hebben we altijd een aantal ingangen en enkele uitgangen gehad, maar in realiteit heb je soms ongespecifiëerde uitgangen.
Er zijn altijd onmiddelijk 2 oplossingen: SOP en POS, dus er zijn automatisch 2 mogelijke implementaties. Het minimaliseren ging erom dat we een zo goedkoop mogelijke oplossing, een zo snel mogelijke oplossing en een oplossing die zo weinig mogelijk vermogen verbruikt willen. We willen de implementatie doen met een minimum aan transistoren.

Slide 13: Implicanten: producttermen waarvoor de functie 1 is. De 1-mintermen behoren daartoe, zoals je op de slide ziet. Dat zijn niet de enige, je kan ook producttermen hebben waarin minder variabelen zitten, zoals op de rode rij. Dat zijn allemaal implicanten die gebruikt kunnen worden om die functie te realiseren wanneer we op zoek gaan om de ene te realiseren.
Priemimplicanten: speciale soort: dat is de compactste manier om iets voor te stellen, er is geen compactere manier en alle manieren die eronder vallen zijn duurder. Het is een implicant die geen onderdeel is van een grotere. De blauwe is een priemimplicant want er is er geen die groter is. We hebben in het voorbeeld maar 2 priemimplicanten.
Wanneer we gaan implementeren is het het interessantst om de priemimplicanten te gebruiken, want als het geen priemimplicant is, is er altijd een efficiëntere manier om het uit te werken.
Dekking: welke implicanten hebben we allemaal nodig om de functie volledig te realiseren? Dat kunnen er verschillende zijn. De oplossing die gebruik maakt van de priemimplicanten is veel efficiënter.

Slide 14: Wij gaan vertrekken van een Karnaughkaart, in andere gevallen/oplossingen kan je van iets anders vertrekken (bv. tabellen). Wat daarachter komt is essentiëel hetzelfde. We hebben al gezien dat het de priemimplicanten zijn die het interessantst zijn om te gebruiken, dus we gaan die eerst bepalen. Er is

een speciaal soort priemimplicanten: de essentiële priemimplicanten. Dit zijn degenen die je echt nodig hebt om het te implementeren. De essentiële heb je echt nodig en je weet dat je die nodig hebt door het feit dat er minstens 1 1-minterm inzit die in geen enkele andere priemimplicant voorkomt, er is geen enkele andere beschikbaar waarmee je het kan doen, dus die is essentiëel. Dan gaan we zien of we nog iets anders nodig hebben, we vullen dat dan aan (liefst met priemimplicanten) tot alle enen bedekt zijn.

Slide 15: De eerste stap is het maken van de Karnaughkaart. Je moet enen invullen waar ieder van die termen 1 is. De groene vakjes zijn die waar de stelling waar is. Zo doe je dat term per term. Die enen mogen overlappen, ze mogen alleen niet contradicteren.

Slide 16: We hebben nu de Karnaughkaart gemaakt en daarmee gaan we aan de slag. We gaan nu kijken naar een stap-voor-stap oplossing die altijd werkt. Er is ook een snellere manier, maar die is ook gevaarlijker (je ziet makkelijker dingen over het hoofd).
Hoe komen we tot alle priemimplicanten? We gaan die enen één voor één aflopen. Voor iedere 1 gaan we de priemimplicanten zoeken waar die 1 deel van uitmaakt. Als we naar die eerste kijken hebben we 2 1-mintermen die eraan voldoen. We kunnen er geen van 4 zoeken waarin de linksbovenste 1 inzit. We kunnen geen van 4 vinden, de grootste is dus van 2. De bedoeling is dat we alle priemimplicanten met een grootte van 2 opsommen, in dit geval zijn dat er dus 2. Dat resulteert dus onmiddelijk 2 priemimplicanten. Bij de volgende 1 kunnen we 4 enen samennemen en weerom zijn er 2 mogelijkheden, velden van 4, waar die 1 in voorkomt, dus er komen weer twee priemimplicanten bij. Bij de volgende 1 komt er geen priemimplicant bij (we hadden die al), zo kan je verdergaan en priemimplicanten identificeren. Uiteindelijk heb je de volledige lijst van priemimplicanten.

Slide 17: Nu je die priemimplicanten hebt moet je de essentiële gaan zoeken. De essentiële zijn degenen die een 1 bedekken waar geen enkele andere priemimplicant overkomt. Je kan dat zien op de Karnaghkaart. Op veel plaatsen zijn verschillende gebieden die overlappen, bij 2 enen is er maar 1 term die de 1 bedekt. Dat zijn dus termen die essentiële priemimplicanten zijn. Door het feit dat je die gekozen hebt, heb je al een zekere dekking. De rechtsbovenimplicant zal ook andere enen bedekken, we hebben zo al 8 enen bedekt. Zo moet je gaan kijken of er nog enen zijn die niet bedekt zijn (zijn er 2) en van die resterende implicanten een liefst zo klein mogelijke priemimplicant zoeken zodanig dat die resterende enen ook bedekt worden. Tot hiertoe kon je met zekerheid zeggen wat erin moest zitten, maar nu wordt het iets moelijker want je moet het globale overzicht hebben en kijken welke priemimplicanten je gaat gebruiken. In de praktijk is dat vaak moeilijk en gaat men iteratief te werk en gebruikt men een gulzige manier van werken: eerst de priemimplicant zoeken die zoveel mogelijk bedekt, dan de tweede die dat doet,... Je kan in een lokaal optimum terechtkomen, maar het kan zijn dat het niet de meest minimale oplossing is. In praktijk ga je er meestal wel terechtkomen. We gaan van iedere priemimplicant zoeken hoeveel die bedekt en dan gaan we die nemen die de meeste enen

bedekt. Uiteindelijk kom je op de oplossing getoond op de slide, dat is een minimale standaardoplossing.

Slide 20: Je kan ook kijken naar de nullen in plaats van naar de enen. Je krijgt dan de kaart zoals getoond. Ook hier ga je op dezelfde manier tewerk: zo groot mogelijke gebieden zoeken die een nul realiseren. Je zoekt de essentiële, die minimale gebruiken en dan zoeken wat je nog nodig hebt. Op zich is dat niet zo erg verschillend, maar er is een groot gevaar: het is hier iets minder makkelijk om die term te omschrijven die die nullen bedekt. Als je concreet kijkt naar de rood omcirkelde, hoe beschrijf je dat? Moesten daar enen staan, zou je zeggen zw'y'. Maar er staan geen enen, er staan nullen, dus je moet er het complement van nemen, de beschrijving die dat beschrijft is (zw'y')'. Met de wet van De Morgan kan je dat omzetten naar w+y+z' want nu ben je niet op zoek naar en SOP maar naar een POS. Je kan dat via de omweg doen zoals net uitgelegd, je kan ook rechtstreeks naar de kaart kijken en bv. zeggen: er staan 2 nullen, dus op alle andere plaatsen staan enen. Hoe kan je die gaan beschrijven? Alle enen onderaan is w, alle enen rechts is y en alle zijkanten is z' en dat is meteen een OR-verband.
Hier zie je die 5 priemimplicanten. Dan moet je de essentiële gaan zoeken en weerom, over twee nullen komt er maar 1 priemimplicant. Op dit moment staat er in zo'n vakje een 1 of een 0, die zijn complementair ten opzichte van elkaar.

Slide 21: We gaan gebruik maken van volledig gespecifiëerde functies. Dat betekent dat we niet voor iedere combinatie van ingangen een zinvolle uitgang hebben.

Slide 22: Soms kan het ons niet schelen wat er op een uitgang komt voor die ingangscombinatie. Een reden daarvoor is dat het niet kan voorkomen dat die combinatie aan de ingang gaat staan. Dan zegt men dat die uitgang "don't care" is. We hebben nu al 4 mogelijkheden om een signaal aan te geven: 0, 1, Z & don't care. De mogelijkheden om een binair signaal te beschrijven nemen alsmaar toe. Een voorbeeld waarbij je dat makkelijk ziet waarom dit optreedt is een 7-segmentdisplay van een BCD-getal (binary-coded decimal). Je kan dat op zo'n display weergeven. Je hebt 4 ingangen en 7 uitgangen. Als je een 0 wilt tonen, moeten alle uitgangen branden buiten het middelste stuk,...
De tabel is nog niet helemaal opgevuld eens we 9 hebben voorgesteld. Je kan zeggen dat het u niet kan schelen wat eruitkomt. We geven dat aan met een streepje. Er zijn verschillende manieren om dat te doen (in het boek: d, vaak ook X). Wanneer we dat ontwerp maken, verdwijnt die don't care en wordt die omgezet naar een 0 of een 1.

Slide 23: Hoe maak je daar gebruik van? Wat is het voordeel van een don't care vs. allemaal nullen? De don't care heeft het voordeel dat je die zowel voor een 1 als een 0 kan gebruiken zoals het u uitkomt. Je kan die gebruiken om gebieden groter te maken als dat interessant is. Je hebt bv. de kaart op de slide (links), als je die kaart hebt en je gaat daar een SOP van maken, als je daar zo groot mogelijke gebieden zoekt, kan je don't cares gebruiken om die te zien als een 1 en kan je grotere gebieden maken, zo kan je een ingang minder gebruiken aan uw poort. Zo'n don't care geeft u dus de mogelijkheid om uw

implementatie efficiënter te maken: je kan kiezen wat u het beste uitkomt. Als je gaat kijken, wat is er nu met die don't care gebeurd eens je dat gemaakt hebt? Als je de blauwe gebieden als priemimplicanten gebruikt voor de implementatie, de onderste don't care is dan een 1 geworden want je hebt de onderste rij samengenomen (anders mocht je dat niet samennemen). Evengoed zijn de twee bovenste don't cares enen geworden. De tweede onderste is een nul geworden al je heb die niet gebruikt om er een 1 van te maken. Na de implemenatie bestaan er dus geen don't cares meer, je gebruikt dat enkel tijdens het ontwerp, maar eens je de implementatie gemaakt hebt, bestaat dat niet meer. Hier is het nu ook zo dat als je de nullen zou implementeren, dat dat niet automatisch een complementaire implementatie is: rechtse tekening: je neemt de twee bovenste erbij om een groter gebied van nullen te maken en de tweede onderste gebruik je ook als nul, de onderste is een 1 geworden. Het is dus niet zomaar het complement van de linkse tekening! Het is natuurlijk niet zo dat dat de ene keer 0 en de andere keer 1 is, het hangt er vanaf, van uw implementatie. Indien het mogelijk is, ga je zoveel mogelijk don't cares in uw Karnaughkaarten proberen in te schrijven.

Slide 24: Een normale schakeling heeft meer dan 1 uitgang. Op het eerste gezicht lijkt dat niet zo erg want 1 schakeling met 4 ingangen en 7 uitgangen zou je kunnen bekijken als 7 schakelingen met ieder 4 ingangen en 1 uitgang. Je kan dat zo implementeren, maar dat geeft meestal geen efficiënte implementatie.

Slide 25: Je kan poorten die je in de ene implementatie gebruikt ook voor de andere gebruiken. Het voordeel is dat die maar 1 keer kosten. Zoveel mogelijk poorten hergebruiken is waar we naar op zoek zijn, daarom mag je dat niet onafhankelijk van elkaar maken. Je moet alle kaarten bijeen bekijken en dan als 1 probleem oplossen. Stel dat we enkel priemimplicanten gebruiken. Er is een speciaal soort priemimplicanten, die essentiële heb je zeker nodig. Wat je wel kan doen is die 7 kaarten opstellen en in ieder van die kaarten naar de essentiële zoeken. Je weet dat je die nodig hebt om die geïmplementeerd te krijgen. Je kan geluk hebben dat er priemimplicanten overeenkomen. Je gaat in iedere kaart de essentiële priemimplicanten zoeken en die al als basisoplossing gebruiken. In het tweede deel, waarin je de resterende enen gaat proberen oplossen, mag je het niet onafhankelijk van elkaar bekijken want ondertussen heb je al poorten ter beschikking en het zou het interessantst zijn om poorten te hergebruiken. Wat in het rood aangegeven is, is in iedere kaart de essentiële priemimplicant. In de eerste twee kom je toe met de ene essentiële die er is, maar bij de derde zijn er nog 2 enen die niet bedekt zijn door de essentiële. De vraag is nu hoe je die gaat realiseren. Nu gaan we ook naar de andere kaarten kijken, want het zou kunnen dat die al in een andere kaart geïmplementeerd zijn. Dat is al zo: die twee blauwe zijn geen essentiële voor c, maar degene die de middelste implementeert is een essentiële in a en b en de blauwe die de rechtse kolom samenneemt is een essentiële in a. In dit geval bestaan ze beiden al. Als er al meerdere geïmplementeerd zijn, hoe kies je dan? Qua kostprijs maakt dat op zich niets uit. Je kan kijken naar andere randvoorwaarden zoals bv. een zo snel mogelijke schakeling. Die hangt af (bij CMOS) van de fan-out. Dus, wat ga je hier doen? De schakeling zal zo snel mogelijk zijn als je de fan-out van alle poorten zo laag mogelijk kan houden. Dat betekent dat ze allemaal

zo weinig mogelijk gebruikt moeten worden. Je gaat ze gebruiken omwille van de kostprijs, maar je gaat een poort die al veel gebruikt is niet hergebruiken als er een is die weinig gebruikt is. Die van b is al 2 keer gebruikt, die van a (de rechtse kolom) is nog maar 1 keer gebruikt, dus we gaan die gebruiken. Belangrijk is dus dat je dat niet onafhankelijk van elkaar mag bekijken. De uiteindelijke implementatie van de drie uitgangen staat onderaan. Je ziet dat er dingen hergebruikt worden.

Slide 26: Wat het moeilijker maakt is wanneer je meerdere kaarten hebt die je samen moet bekijken. Het kan hierbij gebeuren dat, als je meerdere uitgangen hebt, het interessant is om niet-essentiële priemimplicanten te gebruiken, of zelfs implicanten te gebruiken die geen priemimplicant zijn. Normaal is dat geen goed idee, maar dat kan een goed idee zijn als je er meerdere hebt. We hebben hier twee kaarten/uitgangen voor 4 ingangen. We gaan dat implementeren op de manier juist uitgelegd en dan krijg je kostprijs 32. Dit is evenwel niet de meest compacte implementatie. Je kan gebruik maken van niet-essentiële priemimplicanten, maar alleen als ze al ergens anders voor gebruikt werden, alleen als je daardoor hergebruik kan stimuleren (alleen als je meerdere uitgangen hebt, nooit bij 1 uitgang). Bij hergebruik stijgt de kostprijs niet: het maakt dan niet uit als je 1 poort hebt die duurder is maar wel meer gebruikt kan worden.
Vanonder is men niet op zoek gegaan naar essentiële priemimplicanten, maar is men gaan kijken hoe zoveel mogelijk hergebruikt kon worden. Wat in het groen weergegeven is en de 1 onderaan kan je 2 keer gebruiken. Je kan zeggen dat de 1 bij a niet nuttig is omdat je die kan samennemen met de enen erboven, maar dat zou enkel zijn wanneer je maar 1 kaart hebt, nu gaat dat omdat je meerdere kaarten hebt: het is toch gratis om ze een tweede keer te gebruiken. Bij de twee enen in de derde kolom is geen enkele kaart een priemimplicant, maar je kan het hergebruiken dus uw kostprijs daalt. Het probleem hierbij is dat er geen mooi stappenplan is om dat te vinden. Je moet die kaarten naast elkaar leggen en ernaar beginnen staren en hopen dat er dingen naar boven komen, je gaat misschien dingen moeten proberen. Het is dus trial-and-error. Het probleem is daarmee nog niet helemaal opgelost: alles wat we tot nu toe gezien hebben geldt alleen als je tot een tweelagenimplementatie komt. Daar ben je dikwijls naar op zoek want het is de snelste implementatie, maar in vele gevallen kan het interessant zijn om naar meerdere lagen te zoeken.

Slide 27: Dit soort technieken is in alle CAD-omgevingen geïmplementeerd, maar niet met Karnaughkaarten (is een visuele manier van werken). Er is dus een tegenhanger van: Quine-McCluskey. Die maakt gebruik van tabellen. Is niet echt interessant om dat met potlood en papier te doen, zo kan die tot een goede oplossing komen. De technieken die daarachter zitten zoals essentiële priemimplicanten zoeken,... komen daar ook aan bod. We gaan dat niet nog eens zien omdat het niet nuttig is in dit geval. Het staat in het boek in detail beschreven als het u interesseert. Je moet gewoon weten dat het een tegenhanger is van Karnaugh die hetzelfde doet.

Slide 30: Het kan zijn dat nadat je je specificaties geanalyseerd hebt, je dat als logische bewerking op de manier na de slide beschrijft. We kunnen geen poorten maken met een fan-in van 100, dat is niet mogelijk om technologische

redenen. Als je dus met die technieken van Karnaughkaarten uitkomt op een poort met 100 ingangen, dan kan je die niet maken. In die technieken van daarnet kan je dat niet inbrengen dat er een beperkte fan-in is. Hoe doe je dat dan? Je lost het probleem in stukjes op: eerst hou je er geen rekening mee, dan ga je kijken hoe je uw poorten kunt realiseren met minder ingangen dan je bent uitgekomen. Het komt erop neer dat je haakjes gaat introduceren: de onderste formuleset is eerst een OR-poort met 4 ingangen en dan reduceer je die naar 3 OR-poorten met 4 ingangen. Je kan uw poorten omzetten naar meerdere lagen van OR-poorten. Je kan de haakjes ook anders zetten en dan krijg je ook poorten met 4 ingangen.
Op zichzelf is die laatste lijn niet slechter dan de voorlaatste (zelfde kostprijs), maar qua vertraging is dat niet hetzelfde: in de onderste heb je 3 lagen en die gaat trager zijn. Als we dan toch kunnen kiezen, kiezen we natuurlijk de snelste oplossing. Je gaat altijd een boomstructuur toepassen, dat is de snelste manier van werken. Dat is hier geïllustreerd voor OR-poorten, dat geldt evengoed voor AND-poorten en XOR-poorten. Zo kan je tegemoet komen aan de beperkte fan-in die we soms hebben.

Slide 31: Functionele ontbinding: de functie ontbinden in verschillende delen die je ieder afzonderlijk gaat implementeren. Bij voorkeur ga je die delen zo kiezen dat de uitgang van het ene deel op meerdere plaatsen kan gebruikt worden, dan heb je iets nuttig apart geïmplementeerd. Je gaat nadenken van wat erin zit, waaruit bestaat het probleem dat je wil oplossen en wat zijn de subproblemen? Elk van die subproblemen ga je apart implementeren. Dat is typisch wat je doet als je hiërarchisch gaat oplossen. Ieder van de subschema's ga je laten invullen. Zelfs als je de subschema's optimaal maakt, gaat het totaal geen tweelagenlogica zijn.
Heel dikwijls, zonder het zelf te beseffen, kom je tot meerlagenlogica. We gaan dat straks toepassen. Je komt dan tot een andere (goedkopere) implementatie en gedeeltelijk snellere implementatie. Het probleem hier weer is dat het ook hier niet evident is wat je juist moet gebruiken, wat een goed resultaat zal opleveren. Er is dus heel veel trial-and-error en ervaring begint hier mee te spelen.

Slide 32: Alles wat we tot nu toe gezien hebben gaat er vanuit dat we alleen beschikking hebben over basispoorten in een CMOS-technologie. Maar dat is niet het geval, we kunnen ook AND-OR-invert gebruiken. Als je een FPGA gebruikt, doet het er eigenlijk niet toe wat voor poorten je hebt want 1 blokje in een FPGA kan elk van die functies gaan uitvoeren. Hoe los je het probleem nu op? $\rightarrow$ Door technology mapping.

Slide 33: Je kapt uw probleem in zoveel kleine stukjes die je wel kan oplossen en zet die allemaal samen. We gaan dat hier ook zo doen. Zowel voor FPGA als voor gate arrays. We gaan in 2 stappen werken:

1. We hebben een beschrijving, we gaan die proberen zo compact mogelijk voor te stellen. We gaan doen alsof we het met basispoorten gaan implementeren, dan hebben we de meest compacte beschrijving van de functionaliteit die we moeten implementeren.

2. Een keer we dat hebben gaan we dat implementeren, rekening houdend met de beperkingen die er zijn.

In de praktijk worden we daar meestal bij geholpen want stap 2 is zeer sterk afhankelijk van de implementatie: komt het in een FPGA, in een gate array,... Degene bij wie we dat gaan implementeren weet dat we met dat probleem zitten en zal daar gewoonlijk software voor leveren om die stap te doen want het is te specifiek voor de implementatie.

Slide 34: (EXAMEN: je mag alleen NAND en/of NOR poorten gebruiken met 2 ingangen, niet met meer dan 2 ingangen, ook geen AND en OR). Hoe ga je dan te werk? Je gaat eerst de Karnaugh-technieken toepassen en dan kom je bv. tot de tekening links. Je hebt dan de compactste oplossing gezocht, die moet je gaan omzetten naar een implementatie die met die beperking reking houdt. Het eerste wat je dan doet is die poorten opsplitsen. Daarmee heb je de eerste beperking opgelost: geen enkele poort die meer dan 2 ingangen heeft. Dan moet je dat omzetten naar NAND en NOR. Op die manier heb je een implemenatie die aan alle beperkingen voldoet.

Slide 35: We hebben nu gezien dat er heel veel verschillende implementaties zijn, welke is nu de goeie? We kunnen alleen zoveel mogelijk implementaties bedenken en er dan de beste uitkiezen.

Slide 36: Je kan de canonieke oplossing gebruiken. De canonieke POS is duurder want je hebt meer nullen dan enen.

Slide 37: Karnaugh toepassen om de minimale SOP te zoeken. Vertraging: kritisch pad, alle poorten zijn hetzelfde en soms moet je door een inverter.

Slide 40: Met meer lagen werken, complexe poorten gebruiken,... Wat in het blauw gekleurd is is 1 poort: OR-AND-invertpoort. Dat heeft dus 1 kostprijs van 6 en een vertraging van 3.

Slide 41: In de praktijk is het niet mogelijk om alles te gaan bekijken, er wordt dan meestal gevraagd wat volgens u de beste oplossing zou zijn en te vermelden waarom. Alles nog eens vergeleken: illustreren dat vertraging en kostprijs niet altijd hand in hand gaan. De goedkoopste oplossing is hier niet de snelste oplossing. Het is niet automatisch zo dat de meest compacte oplossing ook de snelste is. Wanneer je het dan moet implementeren moet je weten of snelheid of kostprijs het belangrijkste is voor jou. Afhankelijk daarvan zal je voor 1 van de 2 oplossingen kiezen. Die tabel is geen algemene tabel! Die is anders voor elk probleem dat je moet oplossen, het dient gewoon om te illustreren voor 1 specifiek probleem.

Slide 42: We weten hoe we combinatorische schaklingen maken en wat de mogelijkheden zijn. Nu moeten we dat op een hoger niveau gaan toepassen. Bij de combinatorische schakelingen heb je niet zoveel mogelijkheden: je hebt een aantal bouwblokken die nodig zijn om berekeningen te doen en een beperkt aantal bassischakelingen die bepaalde dingen doen (bv. kiezen uit 10 mogelijkheden). We gaan eerst kijken naar de rekenkundige basisschakelingen.

Slide 43: Eerst moeten we overeenkomen hoe we getallen gaan voorstellen: heel formeel beschrijven wat die cijfers te doen hebben met het getal. De plaats van de cijfers bepaalt de radix (met welke macht van 10 je werkt).

Slide 44: Wij rekenen normaal met het decimale stelsel. Het idee blijft wel hetzelfde. We gaan heel dikwijls niet binair werken omdat het minst grote getal dat we willen voorstellen een enorme opeenvolging van enen en nullen is, dat kunnen wij niet makkelijk onthouden. Men is op zoek gegaan om een aantal bits samen te nemen en dat als 1 cijfer voor te stellen. De grootste macht van 2 die je voor kan stellen met 1 cijfer is een 8 $\rightarrow$ octaal stelsel. Dit zijn 3 bits, is niet handig om neer te schrijven. Hexadeximaal is veel makkelijker. Daarmee kan je met een beperkt aantal cijfers redelijk grote getallen weergeven.

Slide 45: Het is ook niet moeilijk om van het ene naar het andere over te gaan, zolang ze eenzelfde basis hebben.

Slide 46: We gaan daar nu mee rekenen.

Slide 47: Cijferen zoals reeds gekend is ook mogelijk in het binaire talstelsel.

Slide 48: Halve opteller: kolom s = "uitkomst" voor die rij/kolom, c is de carry-over. c_{i+1} is de AND van x_i & y_i, s_i is het dambordpatroon. Het is een halve opteller omdat het maar 2 bits kan optellen. Als je overdracht hebt, moet je 3 bits optellen, dus dat hebben we ook nog nodig.

Slide 49: De Karnaughkaart wordt hier iets moeilijker.

Slide 50: We moeten naar kostprijzen gaan kijken. Als je meerdere cijfers moet optellen wordt de snelheid niet bepaald door van de ingang naar de carry te gaan, maar door van de inkomende carry naar de uitgaande carry te gaan. Van c_i naar c_{i+1} is de vertraging van een AND naar een OR. Dit is beter want op de vorige slide was het een AND en een OR-poort met 3 ingangen.

Slide 51: Waarom ben je geïnteresseerd in de inkomende carry naar de uitgaande carry? Als je die cijfers naast elkaar zet krijg je wat te zien is op de slide. Waar wordt hier nu de snelheid door bepaald? Door de langste weg die je hebt: als de carry verandert kan de volgende carry ook veranderen,... Je moet dus van helemaal rechts naar helemaal links geraken en da's het slechtste geval. Vandaar ripple-carry: rimpelt door van helemaal rechts naar helemaal links. Hoe meer carry's je hebt, hoe langer het duurt. Als je dus een 4-bits opteller wilt hebben, gaat het een poosje duren voor het meest beduidende resultaat zichtbaar gaat zijn. De vertraging van die opteller is dus evenredig met het aantal bits. We gaan nu 1 grote opteller maken van 8 bits met 17 ingangen en 19 uitgangen.

Chapter 6

Les 6

6.1 Slides: 4_Combinatorisch

Slide 50: We hebben gekeken hoe we bouwblokken van combinatorische schakelingen kunnen maken. De eerste die we aan het bekijken waren was een opteller. Uiteindelijk waren we ertoe gekomen dat we dat kunnen opdelen in onderdeeltjes: 1 opteller per cijfer en dan kregen we een dergelijke implementatie en die kunnen we combineren naar meerdere cijfers. En dan krijg je wat op **Slide 51** staat.

Slide 51: Die werken niet onafhankelijk van elkaar. Via de carry gebeurt de overdracht van de ene naar de andere. Dit zorgt voor vertragingen want wanneer er iets gebeurt moet je wachten tot het doorgerimpeld is voor je zeker kan zijn dat het juiste cijfer bekomen is. Een 32-bits opteller is 2 keer zo traag als een 16-bits opteller en we willen dat nu sneller maken. Dat kan door dat niet mooi gestructureerd zoals hier te implementeren, maar door dat als een grote schakeling te zien, bv. met 9 ingangen en 5 uitgangen. Het nadeel is dat we niet meer iets kunnen hergebruiken. De vraag is dan ook of er een tussenoplossing is waarbij we wat kunnen hergebruiken maar dat sneller werkt dan de ripple-carry opteller. Er zijn verschillende mogelijkheden.

Slide 52: Meest courante: carry-lookahead opteller. Wat het meest vertragende was (carry doorgeven) gaan we proberen versnellen. Je kan de dia van **Slide 50** hertekenen en dan krijg je wat op deze slide staat. Als je die signalen (p_i en g_i) hebt, kan je de werking herschrijven zoals getoond op de slide. p en g zijn de signalen voor propagate en generate, met de betekenis op een carry: ik ga in dit cijfer een carry genereren. Wanneer genereer je zeker een overdracht? Wanneer je 1+1 doet, onafhankelijk of er een gepropageerd gaat worden of niet. Wanneer ga je er een voortplanten? Als er 1 aan de ingang is, dan ook 1 aan de uitgang. Een van de ingangsbits moet 1 zijn (staat in p_i). Je kan dat vereenvoudigen, maar nu kunnen we hetzelfde schema behouden. Je kan het verband tussen de uitgaande en binnenkomende carry schrijven als c_{i+1}. De carry hangt enkel onrechtstreeks af van x_i en y_i (denk ik). Uitgaande van de 3 ingangssignalen berekenen we de volgende carry die gebruik maakt van die ingangen en p en g. Wanneer we meerdere cijfers hebben gaan we dit gedeelte dat niet in het

hokje zit in tweelagenlogica implementeren. We zorgen dat de overdracht van de carry in naar de carry out zeer snel gebeurt.

Slide 53: Die blokjes op de dia genereert de p en de g en de som daarvan. Dat gebeurt nog altijd per cijfer, maar daarnaast heb je nog het doorgeven van de carry. Dat gebeurt in het gele blokje en dat is voor alle cijfers samen. Wat is het verschil? Stel dat 1/meer bits veranderen, dan zullen die blauw/groene blokjes ongeveer tegelijkertijd een p en een g kunnen genereren, daar is 1 vertaging onafhankelijk van welke bit je veranderd hebt. Dan heb je van die p en die g die veranderd is via een tweelagenlogica naar alle carry's die er zijn. Die worden allemaal onmiddelijk gegenereerd. Die carry kan dan gebruikt worden om de juiste som te genereren. De totale vertraging heb je dus: door het groen daar propagate en generate, naar de carry en die carry wordt dan nog eens gebruikt om de som te genereren en daar wordt dan de som van genomen. De vertraging hangt nu dus niet meer af van het aantal cijfers. Doordat je aan p en g een betekenis toekent kan je neerschrijven wat er binnenin zit. De vertraging is nu onafhankelijk geworden van het aantal cijfers, maar niet helemaal. Hoe meer cijfers je hebt, hoe groter die producten worden. Je hebt wel tweelagenlogica, maar je hebt altijd maar grotere poorten: zoveel ingangen als er cijfers zijn. De vertraging neemt ook toe met het aantal ingangen, dus ingangen blijven toevoegen is ook geen optie.

Slide 54: Uitgerekend wat de vertraging is: het aantal ingangen zit er nog steeds in, zij het wel met een kleinere factor. Het grootste probleem is dat als je heel veel bits hebt, je dat niet met 1 poort kan realiseren. Dan zal je een boomstructuur van poorten moeten gebruiken. Nog een probleem dat we niet opgelost hebben: de CLA-generator is afhankelijk van het aantal bits (2x zoveel). Wat daarin zit is ook verschillend voor het aantal bits. Als je dat dus wil hebben voor 4 of 5 bits moet je dat anders ontwerpen. Voor elke hoeveelheid cijfers moet je een nieuw ontwerp maken. We kunnen dit oplossen: ervoor zorgen dat we een oplossing hebben waar we redelijk wat kunnen hergebruiken en we toch niet die lineaire toename hebben met het aantal bits. Dat is de schakeling op de slide: elk geel blokje is een CLA zoals op de vorige slide. We gaan die 1 keer ontwerpen voor een vast aantal cijfers, in dit geval voor 4 cijfers. Er komen 4 p's en 4 g's binnen en een carry. Voor 8 bits kunnen we die 2 keer gebruiken, voor 16 bits kunnen we die 4 keer gebruiken. Het is niet zo dat dat onafhankelijk is van elkaar: je moet dat doen zoals op de slide. Eerst worden die samengezet in groepjes van 4, die op hun beurt kunnen in eenzelfde component gestoken worden die de informatie van 16 generates en propagates combineert. Kom je niet toe met 16, leg je nog een laag bij, zo kan je tot 64 gaan. Elke keer je het aantal ingangen verdubbelt, komt er 1 laag bij. We hebben dus een component waarmee we alles kunnen doen. Het is meer hardware dan oorspronkelijk en op zich hebben ze elk geen fan-in probleem, maar er is meer hardware, dus we gaan er meer voor moeten betalen. Wat is de vertraging van dit? We gaan kijken wat het langste pad is: we moeten een aantal keer door de lagen door eer alle carry's gegenereerd zijn. Waar zit de vertraging? Die is evenredig met het aantal lagen en dat neemt logaritmisch toe met het aantal bits. Het heeft dus geen constante vertraging maar een logaritmisch stijgende vertraging. Wanneer je dus naar snelheid gaat kijken is dit meestal de oplossing die gekozen wordt.

Zelfs voor een opteller bestaan er verschillende ontwerpen. Afhankelijk van wat je belangrijk vindt ga je een ander ontwerp kiezen: als kostprijs belangrijk is kies je bv. voor de ripple-carry, als snelheid belangrijk is ga je eerder kiezen voor de carry-lookahead.

Slide 55: Als we nu getallen van elkaar willen aftrekken, kunnen we de opteller gebruiken waarbij we negatieve getallen voorstellen.

Slide 56: De meest gekende is de sign-magnitude: we hebben een teken en een grootte. Bij conventie is een 0 een + en een 1 een -.

Slide 57: Je hebt ook complementsvoorstelling: je gaat van ieder cijfer het complement nemen. Als je in het decimaal stelsel werkt, neem je het 9-complement. Bij binair is dat het 1-complement.
Radix-complement: als we n cijfers hebben is het radixcomplement r^n-D (oorspronkelijk getal). Als je die twee vergelijkt zit er altijd 1 verschil tussen (tussen cijferscomplement en radixcomplement). Dat is geen toeval. Dat heeft ermee te maken dat bij het cijfercomplement van ieder cijfer van 9 afgetrokken werd. Je kan evengoed het volledig getal van 999 aftrekken (want dan zal je nooit moeten gaan lenen). Dat is wat er staat met die D'. Het is belangrijk omdat we dat gaan gebruiken om het radixcomplement te berekenen, dat is veel makkelijker.

Slide 58: Mooie aan het tweecomplement: in het blauw: we kunnen dat beschouwen alsof het de negatieve voorstelling is. Waarom is dat zo? Als je die definitie boven het blauw hebt, kan je geen onderscheid maken tussen r^n en 0 want r^n is een 1 met n nullen achter, maar als je maar n cijfers hebt, zie je die 1 niet meer, dan is dat hetzelfde want je kan het verschil niet meer maken tussen r^n en 0. Je kan geen onderscheid meer maken tussen het radixcomplement en het negatieve getal. Je kan dat dus evengoed gebruiken als voorstelling van het negatieve getal. Eens je dat doet kan je dat optellen en dan optellen met een negatief getal.

Slide 58: Wat als je het omgekeerde nu hebt, hoe weet je dan of dat bv. -3 is of 13? Aan de bitcombinatie kan je dat niet zien: die bitcombinatie kan verschillende dingen voorstellen.

Slide 59: Bitcombinaties: de interpretatie is verschillend van afhankelijk van welke voorstelling je veronderstelt. Een bitcombinatie op zichzelf zegt niks, je weet niet over welk getal het gaat: unsigned, sign-magnitude,... In veel gevallen gaat dat er niet bijstaan, dan betekent het gewoonlijk het 2-complement, maar het kan er evengoed ergens bijstaan.
Probleem op het examen regelmatig: er is een verschil tusesn een 2-compelementvoorstelling (hoe je die bitcombinatie moet interpreteren: 1010 komt overeen met -6) en het 2-complement nemen (je doet een bewerking) dat is iets anders: bij 2-complementvoorstelling komt geen hardware overeen. Bij het nemen van het 2-complement heb je hardware nodig. Je moet dus geen operator er gaan tussenzetten om het 2-complement te berekenen als er staat dat het in 2-complement staat.
Als het sign-magnitude is, is de eerste bit het teken. Bij de complementsvoorstelling

is het anders: de eerste bit geeft het teken aan, maar om de grootte te kennen moet je het complement nemen: 1000: negatief getal, hoe bereken je de grootte? Door van dat getal het 1-complement te nemen: 0111 = 111 - 7. Bij het 2-complement: 1110: je moet het 2-complement nemen: eerst het 1-complement en er dan 1 bij optellen: 0001 en dan +1: 0010. Bij -8: je komt opnieuw 8 uit en je weet dat het negatief is, dus -8. Bij het 2-complement is er maar 1 voorstelling voor 0, bij het 1-complement zijn er 2 voorstellingen voor 0.

Slide 60: Ander probleem dat veel voorkomt: als je een voorstelling hebt in 4 bits en je moet dat eerst omzetten naar een voorstelling in 8 bits, hoe doe je dat? Voor unsigned en sign-magnitude is dat makkelijk: gewoon uitbreiden met nullen vooraan. Bij de complementsvoorstellingen is het iets complexer: als het getal positief is, moet je gewoon nullen toevoegen. Als het negatief is, moet je de radix-1 toevoegen. In het geval van bits is dat een 1 toevoegen (denk ik??). Als je een getal wil uitbreiden waarvan je weet dat het in 2-complementsvoorstelling staat, dan moet je gewoon de tekenbit er extra aan toevoegen aan de meest beduidende bits. Als die 0 was: nullen ervoor, als het 1 was: 1 ervoor.

Slide 61: Als we 2 getallen hebben waarvan er 1/beide negatief kunnen zijn, gaan we eerst kijken naar de tekens. Als die tekens gelijk zijn, is dat het teken van het resultaat. Als de tekens niet aan elkaar gelijk zijn, kan het zijn dat je moet testen op + of - 0. Als ze een verschillend teken hebben, moet je kijken welke van de twee de grootste grootheid heeft. Dat is dus heel wat nadenken, niet gewoon een berekening doen. Afhankelijk van de verschillende gevallen telkens iets anders doen.
Een aftrekking doen vergt alleen dat je het teken moet veranderen.
Dat neemt niet weg dat je dit normaal niet gaat gebruiken. Als je dit wil implementeren in hardware heb je hardware nodig voor ieder van die testen en voor ieder van die blokjes, ook een controle die afhankelijk van de testen bepaalt welk stukje in gang schiet. Daarom gebruiken we dit normaal niet in computers.

Slide 62: Als je kijkt naar de 1-complementsvoorstelling: je telt de getallen op en moet een carry-aanpassing doen. Als er een carry uitkomt, tel je die nog eens bij het resultaat op. Als je negatieve getallen hebt, moet je niet het teken veranderen, maar gewoon het 1-complement nemen. Dat is veel makkelijker met hardware.
Nadeel: je hebt 2 stappen: je moet eerst een optelling doen (met ripple carry: vertraging), dan komt daar een carry uit en moet je nog eens een opteling doen $\rightarrow$ dubbele vertraging ten opzichte van gewone optelling.

Slide 63: Optelling met 2-complement: door de voorstellingswijze is het optellen van een positief of negatief getal hetzelfde. Je neemt gewoon de som en dat blijft hetzelfde. Dit is dus het eenvoudigste. Het nadeel is dat de negatieve waarde berekenen vrij ingewikkeld zou kunnen zijn. Je kan doen wat op de slide staat en dan een 1 bij optellen. Dat is niet zo.

Slide 64: Illustratie aan de hand van voorbeelden: bovenaan staan de carry's. De zwarten zijn normaal. Groene: carry-in van de minst beduidende opteller. Als je een halve opteller gebruikt, hoef je die niet toe te voegen. Grijze: de

carry die eruitkomt. Als je je beperkt tot 4 bits heb je die niet meer nodig en mag je die weggooien. Wat we nu gaan doen om die +1 erbij op te tellen is die bijkomende carry te gebruiken. Die +1 doen we door de carry-ingang te gebruiken. Zo gaan we met 1 opteller een optelling doen en tegelijk +1 zonder dat we daar extra hardware voor nodig hebben. Dus zowel voor optelling als aftrekking in 2-complement heb je alleen de traditionele manier nodig zoals uitgelegd. Daarom gebruikt men meestal 2-complement.
Als je niet het aantal bits verandert, kan je in de problemen geraken (in alle gevallen). Als je de twee grootste getallen met elkaar optelt, krijg je een getal dat je niet meer kan voorstellen: 7+6=niet-voorstelbaar in 2-complement met 4 bits. 13 is dus niet voor te stellen, je hebt een extra bit nodig. Als je dat toch met 4 bits wil doen, hou je dat aantal bits vast. Moesten er toch 2 grote getallen opgeteld worden, wil je kunnen detecteren dat het fout is. Dat kan je doen door dat af te schatten wanneer het resultaat boven de 7 of onder de -8 gaat zijn. Eenvoudige manier: kijk naar de carry's. De laatste gebruikte carry en degene die eruit komt die je normaal nooit gebruikt: altijd hetzelfde normaal: altijd 00 of 11. Hier is het verschillend. Het feit dat die verschillend zijn toont dat je een overflow hebt. Je moet dus alleen die 2 bits vergelijken. Dat geldt ook voor negatieve getallen.

Slide 65: Er is heel veel gemeenschappelijk tussen optelling en aftrekking bij hardware. Je gaat dus veel implementaties hebben waarbij we soms een optelling moeten doen en soms een aftrekking. Je kan dan quasi dezelfde hardware gebruiken. De instructiebit (s) gaat bepalen of er een optelling of een aftrekking gaat gebeuren. Bij optelling ga je optellen zoals de traditionele optelling, aftrekking is door eerst de radix te gebruiken (denk ik?).
XOR: als er een 0 opstaat kan je bepalen dat het de x gewoon doorgeeft, als het 1 is, gaat dat x' doorgeven. Je moet die s hangen aan de onderste carry. Als je de overflow wil berekenen, komt er een XOR bij (test of 2 bits verschillend zijn van elkaar of niet).
Je kan ook nog een stap verder gaan: nog meer dingen combineren. In plaats van een afzonderlijke optelling en aftrekking te nemen, wil je een stuk hardware dat de ene keer als opteller en een andere keer als aftrekker gebruikt kan worden, dat drukt de kostprijs. Kunnen we dat nog uitbreiden?

Slide 66: ALU: 1 component/schakeling dat verschillende dingen kan doen: optellingen, aftrekkingen, logische functies of gelijk wat je beslist dat die allemaal moet kunnen.
Voorbeeld van een die optellingen en aftrekkingen kan doen. Logische bewerkingen: kan je bit bij bit doen, de bits beïnvloeden elkaar niet. Bij rekenkundige berekeningen beïnvloeden de bits elkaar.
Vastgelegd en nu kan je weerom zeggen dat je de specificaties hebt en dat met Karnaughkaarten oplost. Bij complexe dingen is het soms interessant om eerst na te denken en te kijken welke kennis je al hebt. Je kan eerst een uitbreiding doen van het vorige. Hier gaan we dat ook doen. We hebben hier 2 soorten bewerkingen: rekenkundige en logische. Voor de rekenkundige ga je de opteller voorzien. Je kan daar iets bovenzetten: ALE.

Slide 67: Dan krijg je dit schema. Waarom zo getekend? Die ALE moet zo zijn dat het voor alle cijfers hetzelfde is: je moet het maar 1 keer maken en je kan het n keer gebruiken. Eens je dat vastgelegd heb kan je aangeven of het over een rekenkundige of logische bewerking gaat,... Dan moet je vastleggen wat er gaat uitkomen, dat is die F-kolom. Omdat het we het bovenstaande schema al kennen, kunnen we zeggen wat X en Y moeten zijn. Alle logische bewerkingen gaan in de extender gebeuren en alle rekenkundige bewerkingen, daarvoor gaan we zorgen dat die op de juiste manier gebeuren zoals gedefiniëerd.
De extender gaan we wel met Karnaughkaarten doen.

Slide 68: Elke rij in de tabel komt overeen met een kolom bij de Karnaughkaarten. (denk ik). Je moet dan de (essentiële (priem))implicanten gaan zoeken. Snellere manier hierop toepasen: eerst de enen op geïsoleerde plaatsen zoeken. Bv. de rood omcirkelde: valt niet met iets anders te combineren. De eentjes boven het vierkant rechtsonder ook. Je veronderstelt dan dat dit de enige goede oplossing is. Let er wel op dat je altijd de grootst mogelijk oplossing moet zoeken. -1 betekent een 1 voor elk cijfer.

Slide 69: We gaan kijken hoe we een vermenigvuldinger kunnen implementeren in hardware.

Slide 70: De manier van werken is hetzelfde. We gaan elk cijfer in de tweede rij vermenigvuldigen met het cijfer erboven, dat genereert een partiëel product enzovoort en dat tellen we dan op. Dat is een iteratief proces, ook bij hoe wij dat doen. Je kan dat in hardware ook op een iteratieve manier implementeren. Je moet lang wachten tot je uw resultaten hebt. De vraag is of dat sneller kan. Vermenigvuldigen met 2^n is er n nullen achterzetten, gewoon opschuiven: heel eenvoudig te doen.
Ook het meer algemene geval is nog te doen: we moeten die partiële producten berekenen en dan optellen. Die partiële producten zijn onafhankelijk van elkaar. Je kan die dus in parallel berekenen, je moet er alleen veel hardware tegenaan gooien.

Slide 71: Wat betekent dat? Tafels van vermenigvuldiging voor het binaire geval. Als we meer-bits getallen hebben, bv. 2-bits: a_0b_0 en dat geeft het partiële product. Het volgende is juist hetzelfde. Dan moeten we dat nog optellen. Je zou denken dat we een 4-bitsopteller nodig hebben, maar dat is niet zo want die eerste bit moet nergens bij opgeteld worden, pas vanaf de tweede bit. En de laatste is de carry die erbij opgeteld wordt. We moeten dat wel nog aanvullen met een 0.

Slide 72: Meer algemeen, bv. 4x3-bit: de grootte van de schakeling/het aantal transistoren dat je al nodig hebt is een functie van het product van het aantal bits van de getallen, bij 4x3 is dat dus 12. Als je naar grote getallen gaat worden die vermenigvuldigers zeer snel zeer groot.

Slide 73: Voor andere voorstelling zoals 2-complementsvoorstelling: de meest linkse bit is de tekenbit. Als je 1111 schrijft als machten van 2 is de meest linkse bit -8 en niet 8. Dat laatste is niet gewoon uitbreiden met tekenbits want dat

zou 8x-1 geven. Je moet gaan kijken wat je moet doen. Het is dat soort zaken dat ervoor zorgt dat een vermenigvuldiger maken voor een 2-complement maken moeilijker is dan voor natuurlijke getallen. Dat is ingewikkelder. In sommige gevallen levert het op om eerst van 2-complement naar sign-magnitude te gaan en daarmee te rekenen en dan terug te gaan.

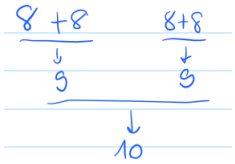
Slide 75: Delen: het grote verschil is dat hier gen partiële producten zijn die onafhankelijk zijn van elkaar, hier moet je eerst kijken of de deler in de eerste X getallen van het deeltal geraakt. Bijna de enige manier om vermenigvuldigingen te doen is op een iteratieve manier. De meesten werken ook iteratief. Zelfs als je er heel veel hardware tegenaan gooit, zelfs dan zijn ze nog traag. Daarom willen we delingen zoveel mogelijk vermijden in de hardware, tenzij het een deling door een macht van 2 is want dan moet je gewoon opschuiven: minst beduidende bit weglaten. Alle anderen zijn ingewikkeld en ga je vermijden. Als je dat toch probeert op het examen, moet je dat ook kunnen verantwoorden op het examen.

Slide 76: Modulorekenen: rest bij deling: eerst delen en dan rest bepalen. Dat is nog ingewikkelder dan een deling, dus dat ga je ook niet in hardware doen, behalve wanneer de deling makkelijk is zoals bij 2. De modulo van een macht van 2 is ook makkelijk. Bij VHDL heb je er 2 soorten: het bovenste en de remainder.
Wij zijn gewoonlijk geïntereseerd in modulo en niet in de remainder.
Als we modulo van een macht van 2 doen, dan is dat gewoon zoveel bits overhouden. Als je modulo 8 (2^3) doet, moet je gewoon de 3 minst beduidende bits overhouden. Je moet zelfs niet weten of dat 1- of 2-complement is of sign-magnitude. Dit kost dus qua hardware niks, je verbindt gewoon de n minst beduidende bits door.

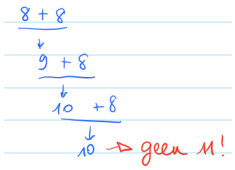
Slide 77: We hebben basisbewerkingen gehad die we makkelijk in hardware kunnen uitvoeren. Nu gaan we komma's introduceren. Normaal spreken de meeste mensen direct over de vlottende kommavoorstelling.

Slide 78: In hardware met vaste komma werken is hetzelfde als met gehele getallen werken omdat een komma met niks in hardware overeenkomt: in hardware is de komma-interpretatie iets imaginair. Je doet dat in het normale rekenen eigenlijk ook zo: 2 kommagetallen met elkaar vermenigvuldigen: eerst doen alsof die er niet stonden, met elkaar vermenigvuldigen en dan de komma plaatsen. Hetzelfde gebeurt in hardware met vaste komma. Je moet gewoon bijhouden waar de komma terecht gaat komen. Voor de hardware zelf maakt dat totaal niks uit.
Wat wel belangrijk is: als je 2 kommagetallen met elkaar optelt/vermenigvuldigt, in wat resulteert dat: hoeveel bits en waar staat de komma? Je moet in gedachten houden hoe je 2 kommagetallen optelt: zo boven elkaar zetten dat de komma's boven elkaar staan en dan doe je de optelling. Je moet 1 van de 2 dus misschien wat verschuiven: nullen toevoegen aan de achterkant, dat is wat op de slide staat. Die voor de komma, daar komt 1 bit bij, dat is nodig om te zorgen dat je in geen enkel geval zorgen moet maken over overlfow.
Tussendoor: hoeveel bits heb je nodig om dat resultaat correct te kunnen

voorstellen? 8 bits (lijkt in tegenspraak met dat van daarnet): de regels gaan er vanuit dat alle mogelijke waarden kunnen optreden. Om echt te weten wat je moet doen, moet je kijken waartussen α kan variëren: tussen 0 en 15. 17*α kan variëren tussen 0 en 17 keer 15. Dat is dus allemaal voorstelbaar met 8 bits.
Stel dat je 4 8-bits getallen moet optellen. Je zou denken dat er met elke optelling 1 bijkomt, maar dat is fout: zie Figuur 6.1.



Je hebt genoeg aan 10 bits. Waarom klopte dat van die 11 niet: zie Figuur 6.2.



Slide 80: Vlottende komma: totaal anders. Tekenbit, exponent en mantisse. Waarom in die volgorde? Als we hardware kunnen hergebruiken, doen we dat graag. Stel dat je hardware hebt om gehele getallen te behandelen en je wil die gebruiken om ook vlottende kommagetallen met elkaar te vergelijken. Die moeten er dan een beetje hetzelfde uitzien. Dat betekent dat de tekenbit helemaal links moet staan want is bij gehele getallen ook zo. Eens je meer naar rechts gaat, hoe minder beduidend uw cijfer is, hoe minder belangrijk het is. Als je twee getallen vergelijkt ga je eerst kijken of de hoogste cijfers verschillend zijn of niet en als die verschillend zijn ga je bepalen welk van de twee het grootste is. Wat het meeste impact heeft op de grootte staat rechts, dus dat moeten we hier ook doen want de exponent is belangrijker dan de mantisse.

Slide 81: NaN is het enige getal dat niet gelijk is aan zichzelf. Wanneer je naar hardware-implementaties kijkt gebruikt men meestal het IEEE-formaat, maar alleen de genormaliseerde voorstellingen ervan. Er zijn implementaties die daar niet mee overweg kunnen. De exponent kan zowel positief als negatief zijn. Hier wordt een heel speciale voorstelling gebruikt: excess-formaat: je gaat bij de exponent een getal bijtellen zodanig dat het altijd positief is. Waarom niet als bv. 2-complement? Zou makkelijker zijn tot je de hardware wil gebruiken om te vergelijken. Als we er altijd iets bij optellen zodanig dat het zuiver positief is, geldt het weer want dan hebben we geen tekenbit meer bij wijze van spreken. Daarom wordt er zoiets speciaals gebruikt: je telt het op op het moment dat je stockeert, wanneer je het gaat interpreteren moet je dat er terug van aftrekken.

Slide 82: Optellen: eerst de exponenten gelijk maken, dan optellen, dan normaliseren. Dat zijn dus heel wat bewerkingen. Dat betekent dat dit ook

nog meer hardware vergt voor vlottende komma dan voor gewone bewerkingen (gehele getallen of vaste komma). Je gebruikt dus vaste komma indien mogelijk. Vermenigvuldiging is makkelijker qua stappen dan optellen omdat je de mantisses gewoon met elkaar kan vermenigvuldigen en de exponenten met elkaar kan optellen.

Slide 83: Speciale gevallen: als je geen getal wil voorstellen, maar bv. symbolen.

Slide 86: ASCII: hoe voorstellen?

Slide 84: BCD: voor elk decimaal cijfer voorzie je 4 bits. Je gebruikt dus maar 10 van de 16 mogelijkheden. Die anderen kunnen en mogen nooit voorkomen. Je moet niet veel bewerkinegn doen op getallen: je hebt een decimale invoer en je wil 2 getallen optellen: van decimaal naar binair en terug is moeilijker. Je kan een opteller ontwerpen die BCD-getallen optelt.

Slide 85: Je kan het complement berekenen, dat is dan niet het 2-complement maar het 10-complement.
Hoe begin je hieraan? Niet met een Karnaughkaart, je begint met na te denken. Als we 2 cijfers willen optellen: een gewone optelling. 10: niet meer voor te stellen met 1 cijfer, ook 7+6 is niet meer voor te stellen met 1 cijfer. Je moet terug van het cijfer dat binair 13 voorstelt moet je 10 aftrekken en een carry genereren naar de volgende. Wat nu in woorden gezegd is geweest kan direct getekend worden zoals te zien is op de slide.

Chapter 7

Les 7

7.1 Slides: 4_Combinatorisch

Slide 87: We gaan de bouwblokken bekijken die we met combinatorische schakelingen kunnen maken. Er zijn nog een aantal bouwblokken die heel veel gebruikt worden, die gaan we nu zien. Buiten de rekenkundige zijn de meest belangrijke bouwblokken die om te kiezen tussen 2/meer waarden: selector of multiplexer.

Slide 88: We hebben een aantal ingangen met een aantal instructie- of controlebits. We gaan er één kiezen en die doorgeven naar de uitgang. Er is altijd 1 uitgang. Die s_0 en s_1 bepalen welke van de ingangen je kiest. Het is weergegeven in een pseudo-waarheidstabel. Deze beschrijft voor die 4 selectie-ingangen wat er op de uitgang komt, het beschrijft het heel goed, is wat overzichtelijker. Het voordeel van die beschrijving is dat je bijna onmiddelijk ziet hoe het geïmplementeerd kan worden. Je kan detecteren met die 4 AND-poorten en als die AND-poort 1 is ga je de ingang doorgeven aan de uitgang. Dat gebeurt door dat te combineren met die AND-poort en dan alle resultaten door te geven naar de uitgang. Weerom zie je dat er een heel algemene techniek is zoals Karnaughkaarten, maar meestal ga je die niet gebruiken want door te redeneren weet je meestal wel wat een goede oplossing is.

Slide 89: Cascaderen = een basiscomponent meerdere keren gebruiken om zo iets ingewikkelders te doen: n-bit opteller, daarvoor gebruik je n keer een 1-bit opteller. Je hebt 4 selectiebits en de onderste kiezen uit 1 van de 4 groepjes en binnen elk groepje kies je uit de minst beduidende bits om te kiezen.
Voordelen: je hebt hier veel meer hardware voor nodig dan dat in 1 keer te maken (zou je denken), maar dat is niet waar. De vertraging lijkt ook groter (door 4 lagen gaan), maar raar genoeg is dat niet waar. Het heeft alleen maar voordelen eigenlijk, te zien op de slide. Je kan componenten hergebruiken, je moet alleen maar een 4-naar-1 voorzien en daaruit kan je die anderen opbouwen. Een van de problemen als je heel grote multiplexers (dus zonder cascadering) wil maken (bv. 16-naar-1), naarmate je meer selectiebits hebt, ga je meer ingangen aan die poorten nodig hebben, die poorten worden groter en groter. Dat heeft tot gevolg dat ze kostelijker worden en dat ze trager worden. Vandaar dat als je dat

gaat bekijken, te grote poorten kunnen we niet maken, met deze implementatie heb je geen fan-inproblemen. Het is ook goedkoper en sneller: een poort van 16 ingangen is heel traag, maar wat blijkt: 2 poorten na elkaar zetten van 4 ingangen is sneller dan 1 poort van 16. 2 lagen gebruiken is sneller dan 1 laag in dit geval. Als je dat gaat uitrekenen zie je dat een 4-lagen netwerk is dan een tweelagennetwerk. Daarom wordt cascaderen vaak gebruikt.
Het is niet altijd goedkoper en sneller! Soms is het goedkoper maar niet sneller. Maar in dit specifieke geval is dat wel zo.

Slide 91: Decoder: een aantal bits en die gaan bepalen welke van die 2^n uitgangen 1 is en de rest is allemaal 0. Om te cascaderen is er die 0, als die 0 is, is alles 0. De implementatie op zichzelf is ook vrij eenvoudig, is rechtstreeks toe te passen. Komt veel voor als adresdecoder bij geheugens. a is het adres en dat wordt gedecodeerd naar een van de uitgangen. Afhankelijk van welk adres aan de uitgangen ligt, wordt er bepaald welke geheugencel gebruikt wordt.

Slide 92: Je kan de inverse gaan zoeken, een demultiplexer: je hebt 1 ingang en je stuurt die naar een van de uitgangen. Ook dat kan je cascaderen: je decodeert eerst de meest beduidende bits, die bepaalt dan welke van die 4 actief is en op basis van daarvan ga je verder decoderen.

Slide 93: De decoder kan ook gebruikt worden om een multiplexer te implementeren: MUX kiest uit 4 ingangen en je kan dat zien als het kiezen van 1 van de 4 en dan de desbetreffende doorgeven. Als je het implementeert zoals op de slide (links) is het wel niet efficiënter dan wat we eerst deden. Je kan dit wel gebruiken om tot een andere implementatie van de multiplexer te komen: er zijn altijd 2 mogelijkheden om dat te maken: met een echte MUX of met een bus en tristate buffer. Dit is wat weergegeven is rechts. Je gaat het echt doorgeven met de tristatebuffer. Je mag dan de uitgangen aan elkaar hangen want je hebt de garantie dat er altijd maar 1 is die inderdaad ook verbonden is met de bus, de anderen zijn losgekoppeld. Dat is een alternatieve implementatie voor een MUX. Is dit nu compacter? Nee → andere manier is dan beter. Toch zijn er soms heel wat redenen om het wel op die manier te doen.

Slide 94: Grote voordeel: gedistribueerde versie van de MUX kan gemaakt worden: staat niet ergens op 1 plaats in de schakeling, wordt verdeeld over de ganse schakeling. Wat je evengoed kan doen is die tristate buffers verdelen over uw schakelingen, over de systemen die in uw systeem zitten. → Computer: je kan borden bijsteken die extra functionaliteit bieden. Er is iets nodig dat zorgt dat er altijd maar 1 van de modules maar informatie op de bus kan zetten.
Ze staan dus niet allemaal op dezelfde plaats: per bord/subsysteem heb je 1 tristate buffer.
Voordeel: heel makkelijk uit te breiden: als je er een kaart wil bijsteken: gewoon tristate buffer bijzetten, de hardware van de andere kaarten verandert daardoor niet, het is niet zo dat je plots een extra OR-poort nodig hebt. Je kan hier ook nooit fan-in problemen mee krijgen: je kan er zoveel bijsteken als je wilt. Dat geeft een enorme flexibiliteit. Dat is dikwijls de reden om geen MUX te gebruiken maar zo'n systeem.
Specifiek voor de FPGA: die tristate buffers zijn niet zo goedkoop, maar bij een

FPGA staan er toch nog bij. Bij een FPGA zijn die meestal gratis en je moet er dan geen logische cel voor kopen.

Slide 95: Van 1 naar $2^n \rightarrow 1$ ingang wordt doorgegeven naar een van de 2^n uitgangen en met de demultiplexer ga je bepalen naar welke uitgang het gaat. Dit is hetzelfde als een decoder, maar waar je bij een decoder zegt dat het de ingangen zijn, dat zijn daar de selectors en waar je bij decoders een enable hebt is dat bij de demux de data.

Slide 96 (HEEL BELANGRIJK): Als je denkt dat je een demux nodig hebt, denk daar nog eens goed over na, omdat in het merendeel van de gevallen je geen demux nodig gaat hebben, is meestal pure verspilling en gaat de hardware niet vooruithelpen.
De gele tekening is meestal het geval dat je nodig denkt te hebben: info die naar verschillende gebruikers (niet gelijktijdig) gestuurd moet worden. Als je in een waarheidstabel ingeeft zie je dat de data ofwel naar de ene of wel naar de andere uitgang gaat gaan. Wat er op de andere plaatsen gebeurt is voor jou eigenlijk niet belangrijk (veel mensen denken van wel). Wat wordt er dan doorgegeven? Niet niks! Je mag uw draden ook niet zwevend laten hangen want dat zorgt voor problemen. Wat geef je dan wel door? Het kan u niet zoveel schelen, maar hardware werkt altijd! Je kan er evengoed iets anders opleggen dan een zwevende ingang, vandaar die don't care's. Er zullen berekeningen mee gebeuren en als je dat laat veranderen wordt er meer vermogen verbruikt, maar dan is dat maar zo: is hardware en er staat spanning op, dus er gaat iets mee gebeuren. Zet er dus zeker geen Z.
Als we dit gaan implementeren moeten we met die don't care's iets gaan doen, die gaan invullen. Wat misschien het makkelijkste is om te doen is die 0 maken, want dan kom je uit bij de demux. Waarom zou je dat invullen met die nullen? Want het is een don't care. Slimmer is om het in te vullen met D en dan krijg je de waarheidstabel rechts onder. Dat implementeren komt met de tekening linksonder overeen. Als die het even goed doet als met de demux, waarom zou je dan hardware gebruiken? Als die 2 qua werking equivalent zijn, gebruik dan geen demux. Het is niet helemaal hetzelfde als daar een schakeling staan hebben, maar het werkt even goed.

Slide 98: Encoder: je hebt 2^n ingangen en n uitgangen, je wil het getal uitkomen dat met het nummer van de uitgang overeenkomt. De f_0 en f_1 geven de plaats aan waar die 1 was.
We zijn meestal hier niet in geïnteresseerd want je hebt meestal niet de garantie dat er maar één 1 op de ingang staat, er kunnen ook 2 enen op staan. Dan zou dat niet meer werken! In de praktijk heb je dus de prioriteitsencoder waarbij er wel meerdere ingangen 1 mogen zijn. Je gaat altijd eerst naar de meest beduidende kijken en je neemt dan de eerste 1 die je tegenkomt, de rest maakt u al niet meer uit: als $d_3 = 1$, dan maken $d_2, \ldots, d_0$ u niet meer uit.
Om te normaliseren moet je vinden waar het eerste beduidende cijfer staat in uw resultaat, hier kan je dat parallel doen: het nummer van de meest beduidende bit komt er hier uit.

Slide 99: Ook dat kan je cascaderen: dezelfde component hergebruiken. Hier is het niet echt dezelfde component gebruiken, maar een aantal basiscomponenten die je hebt voor een zekere grootte gebruiken: prioriteit decoderen: in groepjes onderverdelen, dan kijken of er een 1 zit in die groep en dan bepalen voor welke er een 1 moet uitkomen.

Slide 101: Hoe doe je vergelijkingen? Hoe vergelijk je 2 getallen met elkaar? We gaan ook hier proberen modulair te werken en hardware te hergebruiken. We gaan weer een component maken die 2 cijfers vergelijkt en dat gebruiken om 2 getallen te vergelijken.
Je hebt eerst een tabel die 2 bits vergelijkt: x_1 en x_0 is ons eerste getal, y_1 en y_0 is het tweede getal. Aan de uitgang zijn er twee uitgangen die aangeven of x groter is dan y of kleiner. Als het noch groter noch kleiner is, dan zijn ze geijk aan elkaar.
Als die meest beduidende bits hetzelfde zijn, dan is het de minst beduidende bit die zal bepalen welke van de twee het grootste is.
Als die meest beduidende bits verschillen, dan bepalen die welk getal het grootst is, niet de minst beduidende bits. Bij $y_1 = 1$ en $x_1 = 0$, dan zal getal 2 altijd groter zijn dan getal 1. Dat is een belangrijk principe dat we later gaan nodig hebben om die aan elkaar te hangen. Een keer we die waarheidstabel hebben is het makkelijk om er een Karnaughkaart van te tekenen en dat op te lossen en de twee logische functies te bepalen waarmee het geïmplementeerd kan worden.
Als het de meer beduidende bit is die verschilt, bepaalt die het resultaat, anders de minst beduidende bit. Je kan dit veralgemenen: getal in 2 stukken opdelen: meer beduidend gedeelte (als dit verschillend is, bepaalt dit het verschil) en het minder beduidende gedeelte.

Slide 102: Als x_7 en y_7 verschillen bepalen zij wat het resultaat zal zijn, anders zal het deel erna bepalen wat bepalend is. Je kan die component van **Slide 101** dus hergebruiken waarbij de minder beduidende bits weergegeven worden door andere comparators.
Vereenvoudiging: die x_0 en y_0 kunnen 4 mogelijke waarden aannemen, G en L kunnen maar 3 mogelijke waarden aannemen. Telkens waar er 1 1 staat moet er don't care staan (dus x_0 en y_0 gelijk aan 0). Dan wordt dat deel van de Karnaughkaarten ook vervangen door don't cares en vallen er termen weg in het resultaat.
We gaan dat in de praktijk niet doen om dezelfde reden: bij een opteller kan je de minst beduidende optelling door een halve opteller vervangen (denk ik).
Deze manier van aan elkaar schakelen, daarmee kan je de vergelijking doen. Probleem: vertraging: als er aan het minst beduidende deel iets verandert moet dat weer helemaal doorrimpelen naar links. De vergelijking is rechtevenredig met het aantal cijfers. We kunnen hier in een boomstructuur werken, want we werken met meer en minder beduidende delen. Je kan dat ook zo opsplitsen zoals te zien is op de slide onderaan. Boomstructuren zijn interessant omwille van de vertragingen want nu is het evenredig met het logaritme van het aantal cijfers.

Slide 103: Speciale gevallen: het treedt hier het meeste op. In veel gevallen wil je niet zomaar vergelijken. Dit is helemaal niet eficiënt: comparator per

cijfer. Het kan veel compacter. Moraal van het verhaal: wanneer 1 van die gegevens (voor vergelijker, opteller,...), als dat een constante is bestaat er een compactere implementatie. Je kan dit in rekening brengen bij het ontwerp.
Wat we heel dikwijls willen doen is testen of 2 getallen aan elkaar gelijk zijn. Je kan dat beter doen met iets dergelijks zoals op de slide vanboven: die kijkt of 2 bits aan elkaar gelijk zijn (XNOR). We gaan dat per bit bekijken en de getallen zijn aan elkaar gelijk als alle bits aan elkaar gelijk zijn. Dit is een veel compactere implementatie dan een comparator. Bv. testen of een getal gelijk is aan 0: 1 enkele NOR-poort. Zodra er een 1 binnenkomt, zal er aan de uitgang een 0 komen te staan. AND-poort: testen of je aan uw grootste waarde zit. Ook andere dingen, als het een combinatie is, die getallen, van machten van 2. Eenvoudigste hardware: testen of iets (on)even is: als het oneven is, is de minst beduidende bit een 1, daarvoor heb je geen hardware nodig. Om te testen of het even is, moet je er een NOT-poort bijzetten. Je kiest dus beter om te testen op onevenheid.
Als je wil testen van 10 naar 1. Om te testen of het 0 is is veel makkelijker. Dat soort zaken moet je dus zeker in gedachten houden.

Slide 105: Schuifoperaties: je hebt verschillende manieren om bits te schuiven. Het eenvoudigste is roteren: ofwel naar links ofwel naar rechts schuiven. Wat doe je met de bits die eraf vallen? Je zet die op de vrijgekomen plaatsen. In het andere geval moet je vastleggen wat er moet gebeuren met die vrijgekomen plaatsen, de afvaller ben je kwijt. Het hangt af van welke betekenis je geeft aan uw combinaties van bits. Je wil waarschijnlijk controle hebben over wat er op die plaatsen komt. Als dat een getal is heeft dat een betekenis en kan je er niet zomaar iets willekeurig zetten. Afhankelijk van welke voorstelling je gebruikt van getallen ga je iets anders moeten doen. De bedoeling is natuurlijk dat het getal zijn betekenis blijft behouden. Wat betekent dat nu? Als je de meest beduidende bit laat vallen komt er op de minst beduidende plaats een plaats vrij. Er komt dan een 0 bij.
Als het aan de meest beduidende bit een plaats vrij krijgt, hangt het af van de voorstellingswijze die je gebruikt. Een getal uitbreiden hangt af van zijn voorstelling: als het unsigned is komt er een 0 bij, als het een 2-complement is, is het de tekenbit die herhaald wordt (als de tekenbit een 0 was, komt er een 0 bij, anders een 1).

Slide 106: We gaan zo'n stuk hardware maken dat naar links of rechts kan schuiven of roteren over maximaal 1 plaats. Er is 1 instructiebit die aangeeft of er geschoven wordt of niet, een die aangeeft of dat naar links of rechts gebeurt en een die aangeeft of er geschoven of geroteerd wordt.
De 4 bovenste lijnen zijn de ingang en ga je doorgeven aan de 4 plaatsen aan de uitgang. 4-naar-1 multiplexers: 2 ingangen zijn onmiddelijk doorverbonden: als je niet schuift, maakt het niet uit of het naar links of rechts is: als s_2 0 is, is s_1 don't care, dat is opgelost zoals te zien in y_0. Het is pas wanneer die bit 1 is dat we gaan schuiven, in het ene geval naar links. Op de minst beduidende hangt het er vanaf wat je wilt doen: ofwel roteer je, ofwel is het schuiven en dan moet je bepalen hoe je die invult. Afhankelijk wat vanboven staat is dan aangegeven wat er gebeurt. R = arit'* R_{in}: als het geen rekenkundige voorstelling is (niet voorgesteld als een getal) dan is het (???), anders is het altijd 0.

L: ???
Soms wil je onmiddelijk over 3 of 4 plaatsen schuiven: bv. vlottende komma normalisatie. Als je 10 plaatsen moet opschuiven ga je geen 10 van die hardwaredingen achter elkaar zetten, maar dat doen zoals op **Slide 107**.

Slide 107: Als je maximaal 16 plaatsen moet opschuiven: 16 muxen die kunnen kiezen welke bits gekozen worden. Er zijn een aantal controlebits die zeggen hoeveel plaatsen je moet opschuiven.
Fan-inprobleem: je kan eerst bv. 2 plaatsen opschuiven en daarna 4 of eerst 4 en daarna 2,. . . : er zijn 8 soorten mogelijkheden om op te schuiven.

Slide 108: We gaan kijken hoe we de afgelopen zaken in VHDL kunnen beschrijven. We gaan de verschillen tussen de hardwaretaal en een gewone taal bekijken. De syntax moet je niet leren, je krijgt een appendix op het examen.

Slide 109: Commentaar weergeven: - -, namen geven: let op: geen verschil tussen grote en kleine letters! Letters en cijfers en underscores mogen, mag niet met een underscore eindigen. Getallen voorstellen: op de normale manier. Je kan ook exponenten gebruiken (betekent niet automatisch dat het over vlottende komma gaat!). Wanneer het niet nodig is om vlottende komma te gebruiken, zal VHDL ervan uit gaan dat het vaste komma is. Je kan ook met verschillende basissen werken, moet niet binair. 2#1#E10 = 1*$2^1$0.

Slide 110: Je hebt karakters tussen enkele quotes, karakterreeksen en bitreeksen: geef je weer op eenzelfde manier als een karakterreeks. Verschil: staat een bepaalde letter voor: O (octal), b (binair), X (hexadecimaal). Hoofdletter of kleine letter ervoor maakt niet uit. Een bitreeks stelt geen getal voor, is gewoon een collectie van bits, zoals een karakterreeks niet per sé een woord of zin is.
Links en rechts van de pijl zijn niet aan elkaar gelijk, hebben niks met elkaar te maken!

Slide 112: Wat voor objecten kunnen we gebruiken:

- Constante: om het programma leesbaar te houden, zijn vaste getallen.
- Variabele: op dezelfde manier als in een softwaretaal. Ook die worden gedeclareerd en hebben eventueel een initiële waarde en je kan er berekeningen op doen.
- Signaal: daar hechten we een hardwarebetekenis aan, aan een variabele niet. Als je iets hardwarematig wilt doen zegt een variabele niks, aan signalen hangt een hardwarebetekenis. Die komen bij wijze van spreken overeen met een draad. In het voorbeeld: y stelt een draad voor: die van de AND-poort. Je kan het ook gebruiken voor golfvormen. Je ziet het onmiddelijk in het gebruik en de toekenning is ook anders.

Slide 113: Types: de gewone types met de speciale dingen die je voor hardware nodig hebt: gehele getallen (bij VHDL minstens 32 bit). We moeten exact kunnen aangeven wat de range is.
Bit is 0 en 1, is niet hetzeldfe als true en false, maar wordt wel voorgesteld als

karakters!
Bij simulatie is de tijd ook belangrijk.

Slide 114: Voorbeeld van een fysisch type. Daar geef je de eenheden aan. Tijd wordt voorgesteld als een geheel getal. De basistijdseenheid is 1 femtoseconde. Dit is niet altijd handig, daarom kan je bij de specificatie ineens verdere dingen definiëren.
Met zo'n kleine tijdsresolutie werken heeft zijn effect: het grootste tijdsinterval is 4ms als je met XXX bits werkt 32 of 16). Als je met reactie van buitenuit wil rekening houden kan het ms of s duren. Als je met 64 bits werkt geraak je al wat verder.

Slide 113: Eenheden samennemen in vectoren en matrices en je kan ook subtypes nemen. Je kan ook types gebruiken. VHDL zal nooit toelaten dat je er vanuit gaat dat zaken aan elkaar gelijk zijn als ze niet van hetzelfde type zijn (bv. kommagetallen en gehele getallen). Een manier om dat op te lossen is door subtypes te gebruiken. Die high: hoogste waarde van een geheel getal.

Slide 115: Je kan dat samenzetten in vectoren en matrices. Je hebt niet alleen de traditionele vectoren en matrices, maar ook de onbegrensde types. De normale (begrensde), daar leg je het bereik van de indices vast. MSB: most significant bit; LSB: least significant bit.
Onbegrensd: vectoren of matrices waarbij je het bereik niet aangeeft, je zegt enkel dat het een geheel/natuurlijk getal is. Daar kan je op zichzelf niets mee doen maar het laat u toe om het geheel op te splitsen: je definiëert een type zonder te zeggen hoe groot het moet zijn en dan definiëer je een type waarvan je zegt hoe groot het moet zijn.

Slide 116: Toekenningen in vectoren of matrices. Voor die karakterreeksen/bitreeksen kan je dat als een geheel voorstellen. In het algemeen is het zo dat als je een vector hebt (op de slide van 3 gehele getallen) en je doet een toekenning, als je er voor de rest niks bijzet is het van links naar rechts. Je mag ook namen geven en expliciet zeggen wat op welke plaats komt (onderaan). Het wordt heel dikwijls gebruikt op de manier op de onderste lijn: zet alle bits op 0.

Slide 117: Bits worden met 0 en 1 voorgesteld, maar we moeten ook zwevend en tristate kunnen aangeven. Je kan dat uitbreiden: std_logic: nieuw type.

- U: die draad is niet geïnitialiseerd: als je de spanning opzet weet je niet wat de waarde is van uw draden in de schakeling. Je kan daarvoor X gebruiken, maar men maakt een onderscheid omdat bij niet-geïnitialiseerd, daar wordt alles op gezet. Binnen de kortste keren moeten die U's weg zijn. Als die toch blijven is uw schakeling slecht ontworpen.
- X: duidt op conflicten, dit kan op gelijk welk ogenblik ontstaan. Bv. 2 uitgangen aan elkaar hangen of tegelijk 0 als 1 opzetten.
- W, L, H: hetzelfde als X, 0, 1, maar zwak aangestuurd. Sterke aansturing komt overeen met iets wat door een transistor bepaald wordt en het ander is iets wat veel zwakker is zoals bv. een weerstand die iets op 0 of 1 probeert te trekken, als er geen transistor is die het tegengestelde probeert te doen.

- std_ulogic: standard unresolved logic.

Slide 118: Vectoren en matrices kan je aan elkaar toekennen als ze dezelfde dimensies heben en dezelfde grootte in iedere dimensie. De volgorde van de indices heeft geen belang hierbij (downto vs. to).

Slide 119: Resolved vs. unresolved: VHDL is heel strikt. Uitgangen aan elkaar hangen (getoond op slide) mag bv. niet! VHDL wil zeker zijn dat dat niet gebeurt dus dat is niet toegelaten. Op elke regel zijn er uitzonderingen, dus zelfs CMOS mag je af en toe aan elkaar hangen: uitgangen van tristatebuffers. Andere dingen zoals NMOS poorten mag je ook aan elkaar hangen zonder dat je kortsluiting krijgt. Om ook die speciale gevallen te kunnen beschrijven ga je op die draad nog een stukje tussenzetten (in gedachten): dat stukje in het blauw. Als er iets op out_1 en out_2 staat, bereken je wat er echt op die draad staat. Dat blauwe blokje gaat een AND-functie implementeren.
Je geeft dat aan als een resolutietype. std_logic zal hierover nooit klagen, std_ulogic wel.

Slide 120: Gedefiniëerd normaal voor CMOS. Als je 2 uitgangen aan elkaar hangt voor alle mogelijke combinaties, wat geeft dat? Bv. op de ene uitgang een 1 op de ene uitgang en 0 op de andere, dan krijg je een X. In die functie kan je zoveel draden aan elkaar hangen als je wilt. Hoe beschrijft men dat? De input van die functie is een vector (zonder te specifiëren hoe groot die vector is). Met behulp van die attributen kan je dat dus oplossen zonder exact te weten hoe groot dat is.
Gemakshalve begint men met een zwevende draad en daar alles aan toe te voegen, behalve voor het speciale geval er maar 1 draad is want dan moet je dat niet combineren met Z.

Slide 121: Voorbeeld van sterk en zwak: afgesloten bus. We hebben een aantal tristate buffers en een weerstand/spanningsdeler. Als die bus niet aangestuurd wordt gaat die op een bepaalde waarde staan. Hoe in VHDL: de resistor is RTL. Het is zwak in die zin dat als een van de tristate buffers actief is en een aansturing doet van de draad dat niet via H of L zal zijn maar met 0 of 1 en zal die bepalen wat er uiteindelijk op die draad komt.

Slide 122: Normaal gebruiken we bibliotheken om niet alles zelf te moeten definiëren.

Slide 123: Verwarrend: in VHDL heb je packages en libraries. Een library is geen bibliotheek, het is er maar een stukje van. Je hebt een package waar definities in staan en die worden ergens bewaard, dus een library kan uit meerdere packages bestaan. Een package zit in een folder/library. Op de slide zeggen we dat we alles gaan gebruiken, maar je kan ook gewoon onderdelen gebruiken.

Slide 124: Die bibliotheken maken veel gebruik van overloading: manier om een bewerking die je hebt op nieuwe types te kunnen toepassen zonder de oorspronkelijke functies/programma te moeten herschrijven. In VHDL zit bv. ingebakken hoe je getallen optelt. Op het moment dat men een optelling tegenkomt

gaat men kijken wat men wil optellen: ziet men dat het een bit_vector is gebruikt men die versie, anders een andere versie.
Op die manier kan je dingen uitbreiden.

Slide 125: Bitvector die een betekenis heeft van een unsigned of tweecomplement getal, als je die bibliotheek insluit, die definiëert hoe optellingen op bitreeksen moeten gebeuren, dan kan je in uw programma bitreeksen gaan optellen. Deze gebruik je normaal standaard omdat ze gestandaardiseerd zijn.

Chapter 8

Les 8

8.1 Slides: 4_Combinatorisch

Slide 126: We hebben de aspecten van VHDL behandeld, we zijn begonnen met de syntax van de taal. We gaan nu nog kijken hoe we bewerkingen beschrijven.

Slide 127: In eerste instantie gaat het over logische bewerkingen. Je moet opletten: als je naar de prioriteit kijkt bij een logische functie heeft het complement de hoogste prioriteit, dan de vermenigvuldiging en dan de optelling. Hier heeft NOT de hoogste prioriteit, maar al de rest staat op hetzelfde niveau. Als je twijfelt kan je beter haakjes gebruiken. Je kan op vectoren bewerkingen doen (logische) en die gebeuren bitsgewijs.

Slide 128: Rekenkundige bewerkingen: in VHDL kan je even gemakkelijk een berekening of exponentberekening gebruiken als basisbewerking. Het is niet omdat het niet onmiddelijk een hardware-implementatie is dat het onmogelijk te gebruiken is in VHDL. Die bewerkingen kan je op gehele en reële getallen doen (maar daar heeft modulo gaan betekenis). Als je naar de fysische datatypes kijkt is het heel beperkt: je kan seconden optellen en aftrekken maar niet vermenigvuldigen. Je kan wel een reëel getal vermenigvuldigen.

Slide 129: Schuifoperaties: de linkse operand bestaat uit een bitvector/standard logic vector/booleans/...de tweede waarde is normaal een geheel getal dat aangeeft over hoeveel plaatsen je opschuift. Wat dat doet zie je aaan de term.

Slide 130: Vergelijkingen: /= geeft aan dat het verschillend is van. Je kan je nooit vergissen met de toekenning/vergelijking, je kan hetzelfde symbool (<=) voor 2 betekenissen gebruiken. Op gehele getallen en dergelijke is dat vanzelfsprekend. Je kan dat ook op vectoren toepassen. Als die even groot zijn, is dat ook voor de hand liggend. Het is dan alsof het getallen zijn, maar het is er van geen kanten mee gekoppeld! Let wel: je kan bitvectoren van een verschillende lengte ook vergelijken. Daarom heeft men een procedure: je begint bij de meest beduidende cijfers, als die gelijk zijn ga je kijken naar het tweede beduidende cijfer enzovoort. Je gaat bit per bit vergelijken. De twee vergelijkingen die er

staan zijn gelijk: zijn bitvectoren en er wordt van links vergeleken: de eerste 4 bits links zijn gelijk aan de eerste 4 bits rechts (onder). Boven: linkse is de grootste bitvector. Het is dus beter om geen vergelijkingen op bitvectoren met verschillende lengte te doen.

Slide 131: Specifieke bewerkingen op vectoren: je kan vectoren aaneenschakelen: bundel van vectoren samenvoegen tot bv. 1 grote bundel draden.
De toekenning gebeurt altijd via een links en rechts element: de meest linkse index is 0, de meest rechtse is 1. Voorbeeld: bus 5 wordt verbonden met A 0.

Slide 132 Hoe hardware beschrijven met die syntax?

Slide 133: Hardware beschrijven gebeurt in 2 stukken: declaratie waar je het als een zwarte doos bekijkt, je kijkt naar de in- en uitgangen: wat zijn de poorten die eraan verbonden zijn? Het tweede deel is de architectuur: wat zit er in de doos? Aan die poorten geef je namen en in welke richting ze informatie doorgeven. Wat nieuw is is die generic: je kan generische constanten gebruiken, wat binnen de beschrijving van de architectuur als een constante gezien wordt maar waarvan je nog niet weet wat de waarde van die constante gaat zijn. Dat laat u toe om een beschrijving te maken voor een n-bit opteller. Als je die later gaat gebruiken kan je die n gaan specifiëren. Je moet geen verschillende beschrijvingen doen in VHDL voor al die verschillende soorten tellers. $\rightarrow$ 1 beschriving voor iets wat voor verschillende gevallen gebruikt kan worden.
1 of meerdere architecturen: een component kan verschillende implementaties of beschrijvingen hebben. Wanneer je die gaat gebruiken ga je zeggen waarvoor je die component gaat gebruiken met welke implementatie.

Slide 134: Die architectuur bestaat uit kleinere stukjes hardware. Die kleinere stukjes zijn die parallelle uitdrukkingen: concurrent statements. Dat beschrijft op zichzelf een stukje hardware. Het specifieke daarvan is dat het altijd werkt wanneer de spanning opstaat en dat het allemaal tegelijkertijd werkt. Die twee poorten werken alletwee constant. Zodra er iets verandert zijn ze alletwee aan het werk. Ook de volgorde waarin je die poorten tekent maakt geen verschil. Ook in VHDL maakt die volgorde niet uit bij parallelle uitdrukkingen. Ieder van die 2 uitdrukkingen is een parallele uitdrukking en wordt tegelijkertijd uitgevoerd. Als je die 2 lijnen omwisselt (van de NANDs) heb je juist dezelfde uitdrukking.
Je hebt 2 manieren om parallelle uitdrukkingen te maken: structurele beschrijving (letterlijke vertaling van schema) of abstracter: gedragsbeschrijving. Je zegt dan niet hoe het moet gebouwd/geïmplementeerd worden, gewoon zeggen wat het doet en u geen zorgen maken over hoe het geïmplementeerd gaat worden.

Slide 136: Twee naar 1 multiplexer: 2 ingangen, een selectoringang en een uitgang. Met poorten is dat zoals getoond op de slide. De VHDL beschrijving staat ernaast. Dat is vrij uitgebreid. Je kan daar deels op besparen: eerst aangegeven wat alle interne draden zijn, dan drie keer component: 3 soorten component met bepaalde in- en uitgangen (beschrijft niet wat die doen of hoe die geïmplementeerd zijn $\rightarrow$ zoals op het schema: je tekent een AND poort maar zegt niet hoe die geïmplementeerd is). In een gewone schakeling die gebruik

maakt van poorten weet je dat je die ter beschikking hebt. Al die componenten steek je dus in een bibliotheek. Op het moment dat je dan zelf iets maakt, laad je de bibliotheek en blijft jouw programma overzichtelijker. Er staat dat we 4 poorten hebben. Ze krijgen een naam en wat voor type het is en welke draden gebruikt worden (zo geef je aan dat er met iets anders verbonden wordt).

Slide 137: Naast de beschrijving, of dat nu met een gedrag is of structureel, heb je ook de mogelijkheid om de configuratie aan te geven: verband leggen met implementatie die je gaat gebruiken want dat moet je weten op het moment dat je begint te simuleren. Het gaat de link leggen tussen wat ter beschikking is als component en wat je in de bibliotheek hebt.
Als je geen AND poort hebt met 2 poorten, enkel met 3, dan kan je die gebruiken. Dat ga je alleen maar gebruiken als het echt nodig is. Als je dezelfde namen gebruikt als die in uw implementatiebibliotheek zit, kan VHDL dat raden.

Slide 138: Gedragsbeschrijving is veel krachtiger en heeft iets meer mogelijkheden en is iets ingewikkelder.

Slide 139: Er zijn twee basisuitdrukkingen, die komen met een stuk hardware overeen: toekenning van conditionele signalen.
Wat op de uitgang f komt is d1 als s = 1 of H (dat is hetzelfde). Anders d0 als aan die voorwaarden voldaan is en in alle andere gevallen een X. → Beschrijving. Je ziet een ingebouwde prioriteit: je doet eerst de eerste test en als die waar is stop je daar, als die niet waar is ga je naar de volgende test. Het kan ook zijn dat die testen op zich neveneffecten hebben en dan kunnen uw testen impact hebben op wat je moet doen. Dat is anders bij de toekenning van geselecteerde signalen. Daar ga je op een andere manier werken: afhankelijk van het signaal beschrijf je voor ieder van de mogelijkheden wat het signaal zal zijn. Je gaat dus niet testen.
Alle mogelijkheden moeten beschreven zijn en s moet gekend zijn, af te leiden zijn, wanneer het geïmplementeerd wordt. Hier is het wel een verschil: geen prioriteit: als je d1 en d0 omwisselt gaat dat hetzelfde werken.
Die s gaat dus ook geen neveneffecten hebben want ze wordt maar 1 keer geïmplementeerd.

Slide 140: Voorbeeld van VHDL beschrijving: n-bit opteller (n is in het groen aangeduid). Er staat een defaultwaarde bij. Als je niet aangeeeft wat die n is, dan zal die gelijk zijn aan 4. Bij de beschrijving kan je dan uw n gebruiken, wat die ook moge zijn.
Hier is het met standard_logic dus er moet een bibliotheek gebruikt worden. Welke gaan we gebruiken? Hangt er vanaf wat we ermee willen doen: 2 std_logic optellen met elkaar en dan moet je de juiste bibliotheek inladen. Het is van belang de juiste bibliotheken te gebruiken om te begrijpen wat het juist doet.
Onderaan de beschrijving: intern een bundel draden die je gaat gebruiken, die worden gedeclareerd. Tussen begin en end: parallelle uitdrukkingen, komt overeen met de vergelijkingen in **Slide 139**, maar ze zijn parallel: je mag ze door elkaar halen. Hoe doe je die berekening? Dit is een gedragsbeschrijving, je zegt dus niet hoe de optelling moet geïmplementeerd worden, je zegt gewoon dat die moet gebeuren. Eerst worden de getallen uitgebreid met een extra getal

aan de linkerkant. XOR tussen carry die binnenkomt en naar buitengaat. In de onderste lijn ga je die opnieuw berekenen, die resulteeert in de carry die naar de meest beduidende bit gaat en zo de overflow berekenen.

Slide 141: Naast die twee parallele uitdrukkingen is er nog een derde parallele uitdrukking: proces. Een parallelle uitdrukking dient om een stuk hardware te beschrijven. Proces: een gewoon ordinair programma dat beschrijft wat die hardware precies doet, het beschrijft het gedrag ervan. Het beschrijft niet stap voor stap wat het doet, gewoon het eindresultaat. Dat is het makkelijkste om dat via een traditionele taal te beschrijven. Je gaat dus geen parallelle uitdrukkingen gebruiken, maar sequentiële. Dit is een traditioneel programma.
Parameters: signalen die impact hebben op de hardware. Wat gebeurt er: wanneer gaat er op de uitgang iets wijzigen bij een combinatorische schakeling? Als er aan de ingang iets wijzigt. Zodra een van de signalen aan de ingang verandert wordt het proces opgestart en berekend wat er op de uitgang gaat komen en beschikbaar gesteld. Vandaar dat je ook een aantal bijkomende traditionele uitdrukkingen hebt: if, case, lussen,... Je kan hier variabelen en signalen gebruiken (normaal alleen signalen bij parallelle).

Slide 142: In een programma kan je de gewone trukjes toepassen en variabelen gebruiken. Een van de plaatsen waar vaak verwarring onstaat is het verschil begrijpen tussen variabelen en signalen omdat die verschillende impacten hebben. Signalen worden normaal gebruikt voor fysische dingen: komt normaal overeen met een draad waarop je een golfvorm kan meten. Vermits een parallelle uitdrukking stukjes hardware weergeeeft die met draden verbonden zijn is het logisch dat je alleen signalen kan gebruiken op niveau van de parallelle uitdrukkingen. Daar ga je dus nooit variabelen gebruiken, uitsluitend in een subprogramma of procedure. Als je wat in een variabele zit naar buiten wil brengen moet je dat toekennen aan een signaal voor je dat buiten het proces krijgt.
Toekenning aan variabele en signaal: stel dat de stukjes code uit een proces komen (zowel variabelen als signalen kunnen dus gebruikt worden). Wat binnen die test staat zal altijd waar zijn bij variabelen maar niet bij signalen. Rechts: s wordt 1 (links: v = 1), maar het signaal is niet van waarde veranderd, je zegt gewoon dat dat gaat gebeuren op het einde van het proces.
s <= 1 → s woodt 1 op het einde van het proces. s <= 0 → s wordt 0 in plaats van 1.
Dan gaan we s testen, maar deze heeft nog geen nieuwe waarde gekregen, je kan daar niet voor voorspellen of dat waar gaat zijn of niet.
Welke s wordt er getest: de waarde die het had op het begin van het proces, toen het opgestart werd. Hieraan kan je niet zeggen of die 0 was of 1. Het kan zijn dat het soms uitgevoerd wordt en soms niet.

Slide 143: Zorg dat je dit begrijpt bij het examen. Er is een proces dat iets doet: het stelt een variabele op 0, dan x doorlopen: is een vector van 3 elementen. Als er een 1 is op die plaats in de x, tel je een 1 op bij de term. Op het einde moet je dat toekennen aan het signaal zodanig dat het extern zichtbaar is.
Wat gaat dat proces doen: gaat het aantal enen in x bepalen. De hardware hoeft dat niet iteratief te doen, maar we verwachten dat dit in simulatietijd

ogenblikkelijk uitgevoerd is. Dat is geen berekening zoals het daar beschreven staat, het is een programma zodat we kunnen neerschrijven wat de werking is. Hoe kan je dit implementeren: met twee full adders (adder die de eerste 2 bits optelt en dan een die de derde erbij optelt. Einde: komt overeeen met count). Variabelen vervangen door signaal: tussenresultaten stockeren. Dan krijg je het rechtse. Je zou veronderstellen dat dat hetzelfde doet, maar dat is niet zo. Wat komt er op cnt te staan als x verandert? Als er geen enen inzitten komt er een 0 op te staan en anders komt er de oude waarde +1 op te staan. Waarom: in de eerste stap wordt count 0 (op het einde van het proces), dan gaan we in de lus en als er geen enkele 1 voorkomt gaat er geen nieuwe toekenning zijn en gaat cnt = 0 worden op het einde van het proces. Als er wel een 1 instaat, wordt bij cnt 1 opgeteld. Maar bij welke waarde van cnt? → De waarde van cnt toen het proces werd opgestart. Elke keer je een 1 ziet, ga je zeggen dat in cnt de oude waarde van cnt + 1 moet komen te staan. → Met die signalen werken is niet zo vanzelfsprekend omdat het totaal verschillend is van wat we gewoon zijn. Met signalen is dat anders, daar moet je constant in gedachten houden dat het later zijn waarde pas gaat krijgen, maar je berekent nu al wat het gaat worden. Ander voorbeeld: wat onderaan staat mag je enkel schrijven met variabelen, nooit met signalen: kortsluiting tussen uitgang en andere uitgang, maar vergeet dat even. Wat is het probleem: je krijgt een waarde van cnt, telt daar 1 bij op, dan weer,... → aan hypersnelheid die cnt laten oplopen met altijd 1 meer. Zal aan de snelheid zijn van de vertraging om er één bij op te tellen. Je kan dat ook zien aan de VHDL code: stel nieuwe waarde van cnt. Dan zeg je dat cnt die waarde + 1 gaat zijn. Aan het einde van het proces krijgt cnt ook die waarde, dan kijkt het proces of het veranderd is en dat zal zo zijn en zal dat proces weer opstarten.

Slide 144: Dat proces is een derde manier om parallelle uitdrukkingen te hebben. Voor VHDL bestaat er eigenlijk maar 1 soort parallelle uitdrukking: proces. Die andere twee zijn vereenvoudigde notaties om het de mens makkelijker te maken. Je kan de voorbeelden op de slide als een proces schrijven (zoals links). Eigenlijk is de basisparallelle uitdrukking het proces, alles kan daarmee beschreven worden. Het proces komt overeen met een stukje hardware. Waarom die kortere uitdrukkingen: makkelijker voor ons te lezen en schrijven.

Slide 145: Bij sequentiële uitdrukkingen heb je naast de if en case ook lussen. Dat is iets wat nooit stopt (in tegenstelling tot gewone programmeertaal), het is dus niet abnormaal om oneindige lussen te hebben, maar je gaat het niet altijd nodig hebben. Soms ga je iets een aantal keer willen doen of zo lang het nodig is. Onderaan: index voor for-loop moet je niet declareren, gebeurt automatisch in die lus.

Slide 146: Niet met een aantal paralllelle uitdrukkingen zoals daarnet, maar met 1 proces: een proces dat onmiddelijk de ganse component beschrijft. Ook hier moet je telkens in gedachten houden: op het einde van het proces. Wanneer opgestart: w en 'en' (enable), als die veranderen wordt het proces opgestart. 2 gevallen: en = 1 of en = 0. In alle andere gevallen weet je niet goed wat er moet uitkomen. Als enable = 0 zowel in sterke als zwakke waarden, moeten de uitgangen nul zijn. Als enable 1 is, hangt het af van w. Je moet ook rekening

houden met het feit dat het ook X, Y, don't care en Z kan zijn.
Je zou alle mogelijke combinaties kunnen testen en opschrijven wat het moet zijn, maar of uw ingang nu op een sterke of zwakke 1 of 0 staat, maakt niet uit. Of die ingang nu niet geïnitialiseerd is, je weet niet wat op de uitgang komt.
Je werkt met de bibliotheek, de rechtse functie staat ook in de bibliotheek: wat de waarde ook is, je zet het naar een van de 3 om. Dan hoef je ook niet op alle mogelijkheden te testen, gewoon op 0 en 1 en alle andere gevallen worden in die laatste gevallen opgevangen.
Null: zegt dat er niks gebeurt: no-operation want VHDL kan er niet tegen dat je geen uitdrukking zet, je moet altijd een uitdrukking hebben. Stel dat er op die w op een van de 2 bits een X staat, dan kom je in others, dan ga je niks doen, er komt dan op Y XXX te staan.

Slide 147: Lussen om componenten te genereren. Je hebt bv. een optelling: 1 basiscomponent, een full adder en die gebruik je n keer bv.
Als je een schema hebt moet je die n keer tekenen, bij VHDL niet: je zegt dat je die er n keer wil inzetten.

Slide 148: In plaats van n keer te moeten opschrijven schrijf je dat op die manier, da's veel compacter en beter verstaanbaar en het heeft hetzelfde effect. De compiler gaat dat dan controllen: die parallelle uitdrukkingen n keer na elkaar herhalen. Soms wil je dat conditioneel: bij opteller: minst beduidende bit moet lichtjes anders behanded worden dan de resterende bits. Je moet dat dus ook kunnen aangeven. Dat gebeurt met de tweede uitdrukking op de slide: iets uitvoeren mits aan de voorwaarde voldaan is.
Je kan binnen een generate een andere generate zetten (ook conditioneel), je kan die nesten.

Slide 149: Illustratie: vermenigvuldiger: m*n vermenigvuldiger. Let wel: welke bibliotheek wordt gebruikt: unsigned → op natuurlijke getallen, niet op gehele getallen.
De ene ingang heeft m bits, de andere n en het resultaat heeft n+m bits.

Slide 150: Als je weet hoe het gemaakt moet worden krijg je dit. Dit is hetzelfde als de schema's over de vermenigvuldiger. Je hebt maar 1 component nodig: AND-poort, maar ook opteller, maar dat moet je niet als component zien: VHDL weet wat de betekenis van een optelling is, dat zit in de beschrijving, is een onderdeel van de taal die VHDL verstaat.
Je hebt de verbindingen (type). Er zijn 2 soorten: PCvect: vector van partiële producten, van tweedimensionale producten. Anderzijds heb je een vector van partiële sommen.
Signal: Alle draden die in de vermenigvuldiger nodig zijn.
Er is 1 parallelle uitdrukking is die een for-generate is. Binnen die uitdrukking kan je andere parallelle uitdrukkingen gebruiken, in dit geval 4. Die worden alle 4 tegelijkertijd uitgevoerd. Die worden 4*m keer parallel uitgevoerd als dat ontrold is.
Die eerste gaat een partiëel product maken met behulp van de AND-poorten.
De 3 anderen: if-generates: 3 speciale gevallen: helemaal in het begin, als je nog niks hebt om bij op te tellen, daarna heb je altijd 2 dingen die je met

elkaar gaat optellen (resultaat partiële som van de vorige + partiëel product erbij tellen) en bij die laatste doe je dat ook, maar bij de eerste twee heb je de minst beduidende bit niet meer nodig en die breng je naar buiten. Bij de laatstse heb je alle bits nodig, vandaar dat je 3 gevallen hebt. Die 3 testen op verschillende zaken zodanig dat altijd maar 1 van de 3 voor een bepaalde waarde van i waar is. Er zal er altijd maar 1 zijn die iets genereert.

8.2 Slides: 5_Sequentieel

Slide 1: Een schakeling waarbij de uitgang alleen afhangt van de ingang, daar heb je niet zoveel aan. We gaan nu sequentiële schakelingen bekijken: een schakeling die rekening houdt met wat er gebeurd is, die de voorgeschiedenis in rekening brengt.

Slide 2: Je hebt een geheugen nodig om de voorgeschiedenis in te onthouden. Niet alles van die voorgeschiedenis, we zijn alleen geïnteresseerd in iets bijhouden dat equivalent is met wat er allemaal tevoren al gebeurd is en dat zijn die toestanden. In het geheugen gaan we toestanden onthouden: FSM. Wat zit er naast het geheugen nog in? We moeten de toestand aanpassen want er is iets nieuws gebeurd zijn en we moeten ervoor zorgen dat de juiste informatie op de uitgang gebracht kan worden en dat kan op de manier zoals voorgesteld.
Je hebt een combinatorische schakeling die 2 taken heeft:

1. Bepalen wat de nieuwe toestand zal zijn waar we naartoe gaan, die rekening houdt met de huidige toestand en de ingangen die er nu zijn.

2. Ervoor zorgen dat gebruik makende van de huidige toestanden en de ingangen die je de uitgang hebt die erop moet komen te staan.

Er zijn 2 technieken om het geheugen te implementeren.

1. Component: capaciteit bv. waarop je lading brengt, die blijft daarop staan en heeft een geheugenfunctie. Dat wordt ook gebruikt in heel veel geheugens, maar je kan niet de ideale condensator maken. Als je er lading op brengt en je wacht lang genoeg, dan die is die lading weg. Als het over kleine capaciteiten/capacitoren gaat spreekt men over 1 milliseconde. Hoe kleiner de capaciteit is, hoe sneller die lading weg is.
   Als die informatie niet langer moet bewaard blijven kan je dat wel gebruiken, als je maar rap genoeg werkt, geeft het niet dat die info verdwijnt. $\Rightarrow$ Dynamische logica $\rightarrow$ er is een maximale en minimale snelheid waaraan het moet werken: snel genoeg werken of het gedraagt zich niet meer als een geheugen.

2. Terugkoppelsysteem: de uitgang terugvoeren naar de ingang en je brengt de component in een bepaalde toestand en door het feit dat die uitgang teruggekoppeld is naar de ingang houdt die zichzelf in leven. Het moet wel een terugkoppeling zijn zodanig dat het niet omkeert van waarde tijdens het pad dat het volgt. $\rightarrow$ Wij gaan dit gebruiken: zolang de spanning opstaat blijft het er voor altijd opstaan.

Slide 3: Moore: de uitgang is enkel en alleen een functie van de toestand. Mealy: de uitgang is een functie van de toestand en van de ingangen die er op dat moment zijn. Hier is er een combinatorisch verband tussen de uitgangen de ingangen.

Slide 4: Voor de sequentiële schakeling heb je 2 manieren, tot nu toe hadden we het over de asynchrone schakeling: er verandert iets en daardoor komen we in een nieuwe toestand. Niet zo handig om te ontwerpen: storingen etc. gaan effecten hebben. Nadeel: als een overgangsverschijnsel u in een andere toestand brengt, dan geraak je daar nooit meer vanaf, je bent vanaf dan verkeerd bezig. Daar moeten we dus erg mee opletten, bij combinatorisch is dat niet zo'n probleem: ook aan de uitgang even een overgangsverschijnsel maar daarna weg.
Het is dus niet zo interessant om asysnchrone sequentiële schakelingen te maken, wel synchrone: we bekijken ingangen maar op 1 bepaald ogenblik: als we min of meer zeker zijn dat alle overgangsverschijnselen weg zijn. Het gebeurt op basis van het kloksignaal en dan verandert het op een gecontroleerd moment. Het ontwerp wordt dan veel makkelijker.
De meeste schakelingen zijn synchrone schakelingen.
Stijgende/dalende flank: verandering van 0 naar 1 (stijgend) of 1 naar 0 (dalend).

Slide 5: Hoe wordt geheugen gemaakt? Wat zijn de tijdsgedragparameters?

Slide 6: Basiscomponent: geheugenelement voor 1 enkele bit. Hoe kunnen we dat maken? Eenvoudigste: eenvoudige schakeling en zorgen dat er positieve terugkoppeling is. Wat is terugkoppeling: als je naar de uitgang kijkt, die wordt teruggevoerd naar de ingang en we komen terug op de plaats waar we vertrokken zijn. Positieve terugkoppeling want op het pad dat we volgen wordt het signaaal niet geïnverteerd (anders zou het beginnen oscilleren).

Slide 7: Timingdiagram: altijd 2 uitgangen: Q en Q_n. Voor het ontwerp zou het het interessantst zijn dat we zowel het signaal als zijn complement beschikbaar hebben: converter uitsparen in het ontwerp. Eén uitgang is dus altijd het complement van de eerste uitgang. We hebben 2 signalen: set en reset. Het is geen waarheidstabel wat getoond is! Als s en q op 0 staan wordt de waarde bewaard: huidige uitgang is dezelfde als vorige uitgang $\rightarrow$ wordt op dat moment als geheugen gebruikt. Met de set zorg je dat er een 0 in het geheugen komt, met de reset zorg je dat er een 0 komt in het geheugen.
Als s en r 0 zijn zal q ook 0 zijn (veronderstellen we) en $Q_n = 1$. Als we s op 1 brengen komt Q_n op 0 en komt de Q op 1 (want 2 nullen aan de Q_n ingang) $\rightarrow$ stabiele terugkoppeling: er verandert niks meer. Neem je de set weg, gebeurt er niks want de terugkoppeling zorgt ervoor dat het in die toestand bewaard blijft. Als de reset op 1 komt dan gaat de Q op 0 komen met als gevolg dat de Q_n op 1 komt en dan zitten we weer in een stabiele terugkoppeling. Als we de reset wegnemen blijft de stabiele teostand bewaard. Breng je de set en reset aan, dan kan er op de uitgangen niks anders dan een 0 staan, dan klopt het niet meer dat het een het complement is van het andere. Het is dus geen goed idee om dat te gebruiken: doe dat niet (NA). Als we 1 van de 2 signalen wegnemen zal hetgeen wat het langst blijft aanliggen overblijven. Niet de ene na de andere wegnemen

maar tegelijkertijd: dan staan er allemaal nullen op de ingangen met als gevolg dat de 2 uitgangen 1 worden, dan zullen ze 0 worden, dan terug 1,... → het zal beginnen oscilleren. De frequentie waaraan het oscilleert hangt af van de vertraging van de poorten. Je krijgt een raceconditie: wie is het sterkste/snelste: de snelste zal ervoor zorgen dat hij uiteindelijk wint. Het minste verschil dat er tussen die twee poorten is zal dat niet blijven oscilleren: een van de twee pulsen zal smaller worden en de andere breder. Uiteindelijk ga je dan naar een 1 of een 0, alleen weet je niet wanneer dat gaat gebeuren, je weet niet waar je terechtkomt, dat hangt af van toevalligheden. Het is dus totaal onbruikbaar om een ontwerp mee te maken. Voor een ontwerp moet je regels hebben: als ik dat doe zal dat gebeuren. Die roze piekjes zijn dus een ongedefiniëerde toestand.

Slide 8: Ook met NAND-poorten: wordt hier meer mee gebruikt. Ook hier heb je een set en een reset, maar hier hebben de set en reset een actief lage betekenis: gaan alleen maar effect hebben als ze laag zijn, bij NOR was er enkel effect als ze hoog waren.

Slide 9: Het grote probleem van die latch die we tot hiertoe hadden is dat het een asynchrone sequentiële schakeling is. Overgangsverschijnselen heben invloed op de uitgang en kunnen het in een ongewenste toestand brengen.
We willen die klok dus kunnen controleren: wanneer mogen set en reset bekeken worden. → Geklokte latch gebruiken. Alleen als het kloksignaal 1 is worden de set en reset doorgegeven naar de latch. Wanneer de klok 0 is mogen er overgangsverschijnselen zijn, die hebben geen enkel effect.

Slide 10: In veel gevallen wil je niet zozeer dat er een 1 inkomt met 1 signaal en een 0 met een ander signaal, je wil informatie kunnen bewaren. Dat noemt men een data latch: om 1 bit aan data te bewaren.
Als de data 1 is ga je die setten (latch op 1 brengen), als die data 0 is ga je de latch op 0 brengen. Dat kan je op de manier getoond: data en geïnverteerde gebruiken om te setten of resetten.
Op die manier kunnen we gedurende 1 klokperiode 1 bit aan data onthouden.
Tijdsgedrag: wat zijn de tijdsgedragparameters bij een combinatorische schakeling: vertragingstijd: er verandert iets aan de ingang, hoe lang duurt het eer je dat ziet aan de uitgang?
Verschillende vertragingen uitgerekend. Ofwel van de data-ingang naar een van de uitgangen (als de klok 1 is!). Als de klok verandert (1 wordt) afhankelijk van wat de ingangen op dat moment waren en we veronderstellen dat die niet gelijktijdig veranderen want dan gaat het eerst het ene en dan het andere zijn. Om nul te maken moet je de reset activeren: door inverter dan door de twee NANDs, door de rechtsboven NAND en dan ben je er. Er hangt niet 1 tijd aan, er zijn redelijk wat verschillende tijden. We gaan nooit gevraagd worden om de vertragingstijd te berekenen.
Er zit redelijk wat speling op. Wat is nu de vertraging van die latch? Vaststelling: we willen die latch gebruiken om informatie te bewaren. Het is niet de bedoeling dat als de klok 1 is de data verandert en je dat kan mee volgen, dat is geen geheugenelelement. De bovenste vertragingstijden (van pijl 1) doen niet ter zake want zo gaan we de klok niet gebruiken. Op het examen: als er gevraagd wordt wat de vertraging is voor een geheugenelement, zeg dan nooit

van de dataingang naar de uitgang, dat is totaal irrelevant. De enige zinvolle vertraging die je hebt is van de klok naar de uitgang. Dat is de enige vertraging die telt.

Slide 11: Vertraging is niet de belangrijkste parameter, wel de setup- en de holdtijd.
Setup: op het moment dat de klok verandert, op het moment dat die begint te onthouden (actieve klokflank), vanaf dan mag de uitgang niet meer veranderen: vanaf dan begint hij te onthouden. Rond die actieve klokflank mag je de data niet veranderen want dan kan je problemen krijgen. Een tijdje voor de actieve klokflank moeten die stabiel zijn en een tijdje erna ook. Je moet daarmee rekening houden in uw ontwerp.
Voorbeeld: als D tijdig genoeg verandert (stabiel is wanneer de klok verandert) dan werkt dat zoals het moet. Als je die data kort voor de klok laat veranderen werkt dat niet meer, dan is de toestand ongedefiniëerd. Dit werkt maar als we veronderstellen dat wat op die bovenste NAND-poort staat het omgekeerde is van de onderste NAND-poort. Binnenin het geheugenelement kan je inconsistentie krijgen.
Door te specifiëren dat de setup tijd de tijd van 1 inverter is dat die minstens die tijd ervoor stabiel moet zijn kan je garanderen dat dat probleem niet optreedt.

Slide 12: Ze zijn ook belangrijk voor metastabiliteit: inverter en de transferkarakteristiek daarvan. De latch kan je vereenvoudigd tekenen zoals de groene. De twee assen zijn verwisseld ten opzichte van elkaar bij het blauwe. Wat zijn de stabiele punten, waar gaat het geheugenelement zich in blijven bevinden? Op het snijpunt van die 2 curven. Groen: er is 1 of 0 in opgeslagen.
Rode punt: strict gezien is dat ook een stabiel punt en je kan daar eindeloos in blijven zitten, maar het is een metastabiel punt: het is stabiel maar het gaat daar in normale omstandigheden niet lang in blijven zitten: de minste verstoring/ruis gaat ervoor zorgen dat dat redelijk snel daaruit verdwijnt. Kijken we naar het mechanische equivalent: heuvel met 2 dalen. De stabiele punten zijn dat de bal beneden ligt. Als je heel voorzichtig bent kan je ook op de top een bal leggen en blijft die liggen. Maar de minste trilling/wind is gaat die beginnen bewegen en naar beneden rollen.
Is het zo erg dat je daarin zit? $\rightarrow$ Ja! Dat ligt rond de helft van de voedingsspanning: noch een 0, noch een 1: je moet erdoor maar je moet zorgen dat je er zo snel mogelijk doorbent want het genereert spanningen waar de rest van de schakeling niet mee overweg kan en je weet niet hoe die gaat reageren.
Hoe kan je in dat metastabiel punt terechtkomen en daarin blijven? Je zorgt voor energie voor alle dingen die tegenwringen (capaciteiten): genoeg energie om die opgeladen te krijgen naar iets anders. Je moet doseren hoe hard je ertegen schopt: die komt daar en is juist op dat moment al zijn energie kwijt en heeft net dat beetje extra om verder te rollen niet meer. Hetzelfde gebeurt elektrisch. Als je niet genoeg energie levert om van de ene kant naar de andere te gaan kom je dus in dat punt.
Redenen: als je in een geklokt systeem werkt: hoeveel tijd geef je je om te veranderen, als die klok te smal is heb je niet genoeg tijd noch energie om aan de andere kant te geraken. Daarom heb je specificaties wat de minimumbreedte van de klok is.

Wat gebeurt er als je je er niet aan houdt: iets inconsistent binnenin: de ene wil naar de ene kant en de andere naar de andere kant en dan heb je te weinig netto-energie.
Ook daar kunnen we rekening mee houen en daar moeten we rekening mee houden bij ons ontwerp om zeker te zijn dat er zeker geen schending is van die tijden: gedurende die tijden mag de data niet veranderen.
Als we er niet lang inzitten heeft het niet veel effect als je eruit bent voor de volgende klok komt (je de informatie gebruikt). Als het langer duurt is het wel een probleem. Kunnen we bepalen hoe lang het duurt om uit die metastabiele toestand te geraken? $\rightarrow$ Nee. Hangt ook af van de ruis in de schakeling. Als er veel ruis is in de schakeling, ga je er snel uitgeraken, ook de helling van de curve zal hierin meespelen. Je kan geluk hebben en er binnen een microseconde uitgeraken of pech hebben en er pas na 3 uur uitgeraken (al is die kans klein), je hebt nooit de garantie.
De tijd dat je metastabiel bent neemt exponentiëel af maar wordt niet 0 (denk ik).

Chapter 9

Les 9

9.1 Slides: 5_Sequentieel

Slide 13: Basiselement voor sequentiële schakelingen: geheugenelement. Vorige les hadden we het over de geklokte latch. Dat had zijn voordelen: bepalen wanneer we naar een nieuwe toestand overgaan, hoe we onthouden wat er in een vorige toestand gebeurde. Nadeel: zolang de klok 1 is, werkt dat transparant: veranderingen op de ingang worden ook zichtbaar op de uitgang. Dat is niet de bedoeling. Dankzij de klok willen we tegemoet komen aan het feit dat er altijd overgangsverschijnselen meespelen. Als we ingangen zo weinig mogelijk effect willen laten hebben op de uitgang kunnen we de tijd dat de klok 1 is zo klein mogelijk maken, maar door metastabiliteit mag je dit niet te klein maken.

Slide 14: Het feit dat die transparant is kan voor problemen zorgen: je verwacht dat X opgeslagen wordt in de D-latch en op de volgende klok opgeslagen wordt in de tweede (1 klokperiode vertraging). Als die klok transparant is en de ingang rimpelt door naar de uitgang, dan is dat niet het geval, zoals te zien is op het diagram. De veranderingen op Q_1 gaan onmiddelijk gevolgd worden en binnen dezelfde klokperiode krijgen alle geheugenelementen een andere waarde. De enige oplossing hieraan is de klok kleiner maken maar dat willen we niet. Wat op de tekening staat willen we niet, daarom worden latches gewoonlijk niet gebruikt.
Wat we eigenlijk willen is een element dat op een flank werkt: op 1 ogenblik kijken wat de ingangswaarde is: op 1 bepaald moment. Bv. wanneer we een stijgende flank hebben. De vraag is hoe je zoiets kan maken.

Slide 15: Master-slave principe is makkelijkste manier om dat te maken: elk geheugenelement opdelen in 2 delen (2 latches). Het specifieke is dat die elk op een eigen klok werken: als de ene transparant is, is de andere dat niet en omgekeerd. Er is dus nooit een transparant pad naar de uitgang. Wanneer zien we dan veranderingen naar de uitgang? Links gaat transparant zijn als de klok laag is, rechts als de klok hoog is (binnen 1 segment).
Op het moment dat Clk2 1 wordt en de linkse dus niet meer transparant is dat de informatie zal kunnen doorgegeven worden naar de uitgang. Op dat moment kan de ingang dat niet meer beïnvloeden omdat de master niet meer transparant

is.
De info op de ingang op de stijgende klokflank wordt opgenomen. Het hangt dus niet meer van de klokperiode af om te weten wat er precies gebeurt. Als je er zo 2 na elkaar hebt, is er 1 klokperiode verschil wanneer de info op de volgende verschijnt. In de praktijk is het meestal niet implementatie die gebruikt wordt, maar een edge-triggered flipflop.

Slide 16: Niet echt een correcte naam want de vorige was ook flankgevoelig. Deze is compacter maar ook moeilijker te begrijpen. 3 latches. De meest rechtse is een "normale": set en reset ingang. De eerste twee gedragen zich soms als een latch en soms niet. Als de klok 0 is gaan zowel de set als de reset op 1 staan, dan gedraagt zich dat niet als een latch want de informatie van D wordt geïnverteerd doorgegeven alsook de informatie van B. Wanneer de klok 0 is, zal wat op B staan het inverse zijn van D en wat op A staat hetzelfde zijn als D. Dat gedraagt zich op dat moment gewoon als 2 invertoren na elkaar.
Wanneer de klok 1 wordt gaat het zich plots wel als een latch gedragen: zal onthouden wat op A stond en wat op B stond. Vanaf dat moment zal het dat onthouden. Als de klok terug 0 wordt gaan de set en reset op 1 komen. Als dat gebeurt, zal de outputlatch onthouden wat er was. Blauwe draad: zal zorgen dat de set en reset niet tegelijk actief kunnen zijn. Een keer de linkse 2 als latch beginnen te werken, zal de bovenste als set werken en de onderste als reset.
Je kan dat ook allemaal op timingsdiagramma gaan zetten. Zo'n diagram zal niet gevraagd worden, is ter illustratie.

Slide 17: In de praktijk wordt er gewoonlijk een asynchrone clear voorzien en een asynchrone preset. → Bij het opstarten van het systeem wil je niet op klokken wachten maar in een gekende begintoestand zetten: alle geheugenelementen op 0 gewoonlijk. Om dat te doen, daarvoor te zorgen, wordt er een asynchrone clear/reset gebruikt. Traditioneel is dat een actief laag signaal. In sommige omstandigheden is het ook handig om een preset te hebben om dat niet in de 0 toestand te brengen maar in de 1-toestand.
Onthoud hiervan dat het niet alleen de outputlatch is die asynchroon moet werken, maar ook de twee eersten. → Wanneer we over de kostprijs van een flipflop spreken gaan we het hierover hebben. Je hoeft geen asynchrone clear of reset te gebruiken. Je kan het ook syncrhoon implementeren, dan gebeurt dat zoals getoond op de slide na animatie. Dat is een component die we kennen met extra poorten.

Slide 18: Wanneer we gaan ontwerpen, eigenlijk hebben we nu al 2 soorten latches en flipflops gezien. Strikt gezien gebruikt men een latch wanneer men niveau-gevoelig werkt (het element is transparant wanneer het signaal 1 is). Flipflop wordt vaak gebruikt bij flanken. Flipflop wordt wel als verzamelnaam voor beiden gebruikt.

Slide 19: Set-reset: linksboven: flipflops, die werken op een flank. Je herkent die aan de driehoek bij de klok. Bij de niveautriggering (latches) ontbreekt de driehoek. Het bolletje, als dat ontbreekt, is het de normale actief hoge betekenis. Dat bolletje betekent niet dat er een inverter voor staat. Waarheidstabel van flipflop/latch: karakteristieke tabel: waarde aan ingang aanleggen, hoe gaat

de uitgang daarop reageren. Dit is niet wat we nodig hebben een keer we beginnen te ontwerpen. We gaan van een toestandsdiagram/-tabel vertrekken waarin staat hoe je van de ene toestand naar de andere gaat.
Wat u interesseert is hoe je die ingangen moet aansturen om die uitgang bepaalde veranderingen te laten doen. Je weet welke veranderingen aan de uitgang nodig zijn en wilt weten hoe je de ingang moet aanpassen daarvoor. Dat noemt men de excitatietabel. Je kan deze afleiden uit de karakteristieke tabel. Dit is wat je bij wijze van spreken vanbuiten moet kennen. Waarom staat er don't care? Hoe meer don't cares op de uitgang, hoe compacter en sneller het combinatorisch netwerk wordt gewoonlijk. Probleem: combinatie 1 1 van vanboven mag niet gebruikt worden. Het zou handig zijn als we iets nuttig konden doen met die 4^e lijn.

Slide 20: 4^e lijn gebruikt. We kunnen de flipflop setten, resetten, onthouden en nu ook de uitgang inverteren.
Als je nu kijkt naar de excitatietabel staan er nu nog meer don't cares. De flipflop op zichzelf is iets duurder, maar er zijn meer don't cares.

Slide 21: Data flipflop: gedurende 1 klokperiode gaat die info onthouden die op zijn ingang zit. Eenvoudige karakteristieke tabel en excitatietabel. Als je weet wat er volgende keer moet opkomen, leg je dat nu aan zijn ingang aan.

Slide 22: Toggle-flipflop. Als de ingang 1 is zal die van toestand veranderen. Je kan die niet in een toestand brengen, je kan die toestand alleen veranderen. Dit is typisch wat je nodig hebt in een teller. Tekening vanonder: manier om over te gaan.

Slide 23: Alle basiselementen zijn gekend nu. We gaan kijken of we bouwblokken kunnen maken om op hoger niveau te gebruiken.

Slide 24: Je moet onder controle hebben of nieuwe data ingeladen wordt of de vorige data onthouden wordt (onthouden: telkens opnieuw inladen) via een extra ingang.
Op die manier heb je een element ter beschikking waarin je een gans woord kan onthouden en je zelf onder controle hebt wanneer nieuwe informatie onthouden wordt en wanneer nieuwe informatie ingeladen wordt.

Slide 25: Variante daarop: schuifregister ($\neq$ combinatorisch element dat over 1/meer plaatsen opschuift). Hier wordt de data van een register over 1 plaats opgeschoven. Het principe is ongeveer hetzelfde als het vorige: data bewaren (niet opschuiven), ofwel wel opschuiven en dan wordt de uitgang van de ene gebruikt als ingang van de volgende. Je hebt een extra ingang nodig om te bepalen wat de data is die in de meest linkse flipflop komt. Gewoonlijk wordt het gecombineerd dat je het niet alleen kan schuiven maar ook data kan inladen. Zie animatie.
Wat in het blauw staat is in uw bibliotheek als 1 enkel component beschikbaar. Als je dit soort componenten hebt, heb je een laadbaar schuifregister. Een typische toepassing: van seriëel naar parallel en omgekeerd. SerUit: uitgang waar de bits één voor één gaan uitkomen. Je kan dat ook omgekeerd gebruiken.

Het kan ook gebruikt worden om op een iteratieve manier een vermenigvuldiging of deling te implementeren.

Slide 26: Andere basiscomponenten: tellers: bijhouden hoeveel keer iets al gebeurd is of moet gebeuren.

Slide 27: Teller = register + opteller (waar +1 of -1 wordt bij opgeteld).
Het meest efficiënte is een asynchrone teller, niet omdat er geen klok aan zit: er zit hier wel degelijk een klok aan, maar die klok gaat alleen naar de eerste flipflop en niet naar de anderen: die anderen worden niet geklokt door middel van die klok.
Wat gaan die meer beduidende flipflops gebruiken als klokingang: het feit dat de minder beduidende (meer naar links) van waarde veranderd is.
Het is Q' die als klok dient voor de volgende: op dalende flank van de Q zal de volgende reageren. Dat is een compacte implementatie. Nadeel: asynchroon voor de meer beduidende flipflops. Op het moment dat Q_3 verandert is een hele poos na de klok. Naarmate er een flipflop bijkomt gaat dat altijd maar verder en verder van de klokflank afliggen. Als je teveel bits hebt, gaat de laatste veranderen wanneer de klok nog eens verandert of al eens veranderd is. Dat kan je je niet veroorloven: setupproblemen of overgangsverschijnselen.
Dit is dus geen component om aan zeer hoge rates te gebruiken, toch niet als je vele bits hebt.

Slide 28: Als je snelheid wilt moet je synchroon gaan werken. Je hebt dan wel iets meer hardware nodig: concreet aangeven wanneer de volgende mag wijzigen: hangt af van vorige Q's en enabler.
Je kan dit ook implementeren met 1 AND met 3 ingangen als snelheid belangrijk is.

Slide 29: Veralgemenen: soort componenten dat je dan terugvindt: je wil uw teller kunnen initialiseren, je wil dat die naar boven en naar beneden kan tellen en dat je die kan cascaderen: component die dat voor 4 bit doet. Als je er een voor 8 bit nodig hebt: 2 van 4 bit achter elkaar zetten.
Manier om dat te doen is hier weergegeven. We hebben al een idee hoe dat werkt voor een gewone synchrone teller. Daarnet hadden we een toggle flipflop. Als je dat laadbaar wilt maken kan je er geen T-flipflop zetten, je hebt dan een D-flipflop nodig en in de terugkoppeling zet je een XOR poort.
In het kaki moet je iets algemeners zetten: niet om toe te laten van het te laden (zorgt de multiplexer voor), maar om te beslissen of je telt en of je naar boven of naar beneden telt.
D/U*: up-/downteller. Hoe moet je dat interpreteren: dat signaal heeft 2 taken: aangeven dat er naar boven moet geteld worden en dat er naar beneden moet geteld worden. Dit kan met 1 draad omdat ze complementair zijn. We geven dit expliciet aan in de naam zodanig dat we duidelijke interpretatie hebben. D/U* = 1 is actief hoog voor down, D/U* = 0 is actief laag voor up.
Er is een driebitsteller een een cascaderingsuitgang. Daarnaast is er een asynchrone clear en preset maar een synchrone: als set/reset is: allemaal nullen of enen inladen.

Slide 30: Naar de functionaliteit kijken: 2 uitgangen. Die enable is degene die naar de meer beduidende gaat. Als er mag geteld worden en dat geeft E_i aan, dan is het een XOR die we nodig hebben: van toestand veranderen. Als er niet geteld mag worden, dan moet het hetzelfde blijven. Kan beschreven worden met een enkele XOR.
Tweede uitgang: hangt niet alleen af van enable maar ook van of we naar boven of beneden tellen. In het voorbeeld: de meer beduidende bit moet veranderen als de minder beduidende 1 was en naar 0 gaat. Wanneer moet die uit de enable naar het hogere cijfer dan 1: binnenkomende is 1 en de minder beduidende bit is 1.
Als je naar beneden telt is het het omgekeerde.
Je had het ook met een Karnaughkaart kunnen maken, maar dan had het waarschijnlijk niet met een XOR-poort geïmplementeerd geweest. Was niet fout geweest natuurlijk.

Slide 31: Soms willen we geen binaire teller maar bv. een BCD-teller. Hoe maak je dat? Door te implementeren wat juist gezegd is geweest: tellen op normale manier, maar als je detecteert dat je aan 9 gekomen bent, is het volgende niet 10 maar 0. Je detecteert dat in de onderste AND-poort. Als we dat detecteren, dan zal op de volgende klokflank er niet 10 op staan maar op 0. Je doet dan een synchrone reset door er 0 in te laden. Dat kan in de andere richting even goed toegepast worden: van 0 naar 9 bij het naar beneden gaan.
Alle andere soorten tellers kunnen dus ook met een binaire teller gemaakt worden en dan op het juiste moment de juiste waarde inladen.

Slide 33: Hoe in VHDL? Jammer genoeg bestaat er in VHDL niks om flipflops of geheugenelementen te beschrrijven, dat is altijd impliciet. Dat maakt het gevaarlijker: wat je bedoeld hebt als combinatorische schakeling kan een sequentiële worden.
Waar zit de geheugenfunctie? Als je iets niet verandert verwacht VHDL dat de oude waarde blijft staan. Er is dus bijna altijd impliciet een geheugenfunctie. Je gaat onthouden wat erop stond, je moet dat gewoon toelaten.
2 snippets: bij rechts staat er een else, bij de andere niet. Bij de rechtse ga je voor elke waarde van de klok en de data onmiddelijk kunnen bepalen wat op de uitgang staat. Als de data op de klok verandert komt er een eventueel nieuwe waarde op de uitgang. Links is dat niet het geval. Als de klok daar 0 is mag die data zoveel veranderen als die wilt, dat gaat geen invloed hebben. Wat je daar beschrijft is dus een latch: als de klok 1 is gaat de uitgang hetzelfde zijn als de ingang, de ingang wordt doorgegeven aan de uitgang. Als de data constant blijft en de klok verandert, als de klok 1 geworden is, dan ga je hetzelfde hebben: data wordt doorgegeven naar de uitgang. Op het moment dat de klok 0 wordt gebeurt er niks meer. Als je die code analyseert: je schrijft een latch. Het feit dat je die else er niet bij hebt gezet kan betekenen dat je van een combinatorische naar een latch gaat.

Slide 34: We zijn geïntereseerd in de flank-gestuurde flipflops. Hoe? Niet op de waarde van de klok testen, maar op de verandering van de klok. Die if zegt dat als er een verandering van de klok is en ze is 1 geworden, dan zit je met een stijgende flank. Je kan ook gewoon de functie rising edge gebruiken, dan moet

je alle mogelijkheden niet gaan testen.
Er is nog een andere manier, maar die gaan we later zien.

Slide 35: We willen ook kunnen resetten en clearen. Als het synchroon is: binnen de test op de stijgende flank: alleen naar het resetsignaal kijken op de stijgende flank. Als asynchroon: buiten de test op de stijgende flank. Waarom sommige dingen in het lichtgrijs: die ingangen moeten daar strikt gezien niet bijstaan want gaan alleen effect hebben wanneer er een stijgende klokflank is. Maar het is veiligheidshalve beter om alle ingangen te vermelden voor als je later aanpassingen wil maken.

Slide 36: In plaats van gewoon de data door te geven naar de uitgang kan je ook een bewerking doorgeven, het resultaat van een bewerking. Komt neer op dat je een combinatorisch netwerk mee in rekening brengt in het proces. Op die manier heb je de tekening rechtsboven beschreven met 1 parallelle uitdrukking, met 1 proces in plaats van met 4 parallelle uitdrukkingen. Functioneel gezien zijn ze equivalent. Toch ga je heel dikwijls zien wat in de VHDL code staat omdat het efficiënter is wanneer je gaat simuleren.
Elke parallelle uitdrukking komt met een proces overeen. Bij simulatie is het eficiënter om 1 proces te gebruiken.

Slide 37: Voorbeelden: registers. Op zichzelf is dat zeer eenvoudig: vb. van een schuifregister. We testen hier ook op stijgende klokflank. Als de shift 1 is gaan we opschuiven, anders doen we niks. Q(i) wordt Q(i+1) op het einde van het proces. In die loop ga je onthouden wat het op het einde gaat zijn, maar je gaat intermediair nog niks onthouden. Het doet er niet toe in welke volgorde het gebeurt, het kan evengoed op de rechtse manier geschreven worden: ganse vector schuift 1 plaats op.

Slide 39: Het geheugen gaan we nu gebruiken om de toestand in te bewaren van de sequentiële schakeling. Bij een synchrone sequentiële schakeling gaan we gebruik maken van een klok. Op alle andere ogenblikken kan het ons niet schelen hoe dat verandert. Alleen op het moment, liefst op de stijgende klokflank, dat er iets bekeken wordt.

Slide 40: Hoe maken we zo'n ontwerp: eerst gaan we van onze beschrijving vertrekken en dat vertalen naar een eenduidige beschrijving die niet fout geïnterpreteerd kan worden en maar tot 1 duidelijke werking van het systeem kan leiden. Dat is een toestandsdiagram of -tabel. Die beschrijft hoe je van de ene toestand naar de andere overgaat. We gaan dat aantal toestanden zo klein mogelijk proberen te maken want hoe minder toestanden, hoe goedkoper de schakeling.
De volgende stap is het implementeren van het geheugen. Tot hier toe zijn die toestanden nog vrij abstract, maar je moet gaan zeggen hoe je dat in een flipflop gaat bewaren, met wat komt die bepaalde binaire combinatie overeen?
→ Coderen van de toestanden.
Om het geheugen te implementeren moet je bepalen wat voor flipflop je kiest. Dat heeft impact op hoe je de ingangen kiest.

Laatste stap: implementatie van het combinatorisch deel. Enerzijds om de flipflops aan te sturen en anderzijds om de uitgang aan te sturen.

Slide 41: We hebben gezien dat er 2 types zijn: toestandsgebaseerd (uitgang hangt enkel af van de toestand) en inputgebaseerd (uitgang hangt ook af van de input).

Slide 42: We willen een schakeling maken met 1 in- en uitgang. Op die uitgang moet 1 komen als de vorige 2 klokperiodes de ingang 1 was. Je wil een opeenvolging van twee enen detecteren. Als eerste maken we een toestandsdiagram. Iedere ellips is een toestand. Momenteel krijgen die nog symbolische namen. Wat er wel altijd is, is dat elke toestand een betekenis heeft en je moet daar ook een betekenis aan toekennen, anders wordt het moeilijk te begrijpen wanneer je naar een andere toestand terugkeert of wanneer je een nieuwe toestand nodig hebt.
De uitgang hangt enkel af van de toestand op tijdstip t_i. De uitgang hangt dus alleen af van de toestand en je kan de waarde van die uitgang bij de toestand bij noteren: als de toestand A is, is de uitgang 0. Die kan daarin terechtkomen door te resetten of als er geen enen meer geweest zijn. Betekenis ervan: geeft aan dat er ervoor geen enen waren. In een toestandsdiagram kan je in principe niks asynchroon weergeven, vandaar wordt dat met een speciaal symbool weeergegeven: je hebt het in het echt nodig dat je dat asynchroon in een begintoestand kan krijgen (die gekartelde pijl).
Vanuit die toestand moet je gaan kijken voor elke ingangscombinatie waar je terechtkomt, wat de volgende toestand is. Je kan maar 2 waarden hebben: ofwel 1 ofwel 0. Als die 0 is zijn er nog steeds geen enen opgetreden dus blijf je in die toestand. Als er een 1 optreedt, ga je naar een toestand waar het duidelijk is dat er één 1 voorkwam.
Als je vanuit die toestand een 0 hebt herbegin je en ga je terug naar A. Als je een 1 hebt ga je naar een nieuwe toestand die aangeeft dat er twee enen na elkaar zijn geweest. In dit geval ga je als uitgang 1 hebben dus is er een nieuwe toestand nodig. Als je opnieuw een 1 hebt blijf je in die toestand. Als je een 0 hebt ga je terug naar A.
Nu is er geen enkele toestand meer waarvoor niet alle combinaties beschreven zijn. Dit is een interpretatie van wat bovenaan in woorden staat.

Slide 43: In detail bekijken wat de betekenis is van al die getekende dingen: ellips: toestand → niet de volledige beschrijving van de volledige voorgeschiedenis en wat op dit moment gebeurt, maar het geeft enkel de voorgeschiedenis weer.
Toestand + pijlen/ingangen: beschrijven de volledige toestand van het systeem. Je bevindt u altijd op een pijl, nooit in een toestand. Als uw ingang verandert, verander je niet van toestand maar van pijl.
Je mag overspringen zodra er een klokflank is. Dan ga je naar de toestand waar de pijl naar wees. Je gaat u onmiddelijk opnieuw op een pijl bevinden want je hebt een ingang. Die overgangen en toestanden, de enige verandering die daarin kan komen kan alleen wanneer er een klokflank geweest is. Wanneer we het hebben over een synchrone sequentiële schakeling, dan is dat zo.
Rode kader: heel belangrijk: in een synchrone sequentiële schakeling kan nooit

een kloksignaal staan. Nergens in dat schema staat een kloksignaal. Dat is alleen impliciet aanwezig en dient om van de ene toestand naar de andere te gaan.

Slide 44: We kunnen dat ook in een toestandstabel zetten. Dat is juist hetzelfde om hetzelfde weer te geven. De volgende stap is het aantal toestanden reduceren tot een minimum aantal. In dit geval is dit al een minimum aantal toestanden.

Slide 45: Codering: je hebt toestanden die je nu in een geheugen moet gaan plaatsen. Je hebt minmum 2 flipflops nodig om die 3 toestanden te combineren. Je hebt verschillende mogelijkheden. Je moet dat niet doen zoals getoond, je mag de benaming in principe ook anders doen. Op zich maakt het wel veel verschil wat je daar kiest, gaan we nog zien. Er is normaal altijd een begintoestand en het is de gewoonte om die met 0 overeen te laten komen. Daarna moet je het type flipflop kiezen.

Slide 46: Laatste stap: combinatorische schakeling ontwerpen: bepalen wat de toestand is en aansturing van de uitgang. Hoe vertrek je hier van om dat combinatorisch netwerk te bepalen? Die excitatietabel is hier nodig: verband tussen veranderingen van toestand. Een D-flipflop is makkelijk om mee te werken want wat je aan de ingang moet aanleggen is de volgende toestand. Je moet er niet voor weten wat de huidige toestand is, gewoon wat de volgende is/moet zijn. Nu heb je een verband: uitgangen van combinatorische schakeling, huidige toestand en de ingang. Je kan dat ook in Karnaughkaarten neerschrijven. Er zijn 2 kaarten want je hebt D_1 en D_0.

Slide 47: Implementatie: basis: wat we al hadden, dan pas je toe wat in de Karnaughkaarten staat. De uitgang is gewoon de uitgang van die flipflop. Zelfs geen speciale hardware voor nodig.
Nu hebben we een eerste synchrone schakeling geïmplementeerd. Je moet nakijken of dat wel werkt: controleren of je uw specificaties wel goed vertaald hebt naar een toestandsdiagram. Dit doe je meestal door middel van een tijdsdiagram. Je moet alleen op de stijgende klokflank kijken, alles wat daartussen gebeurt is niet van belang. $\rightarrow$ Verschil met inputgebaseerd systeem (nu een toestandsgebaseerd systeem).

Slide 48: Het is niet omdat je lokale optimaliseringen doet dat je tot globale optimaliseringen komt.
Stel dat we van 4 toestanden vertrekken en we willen niet minimaliseren. Ook hier zullen twee flipflops nodig zijn. We gaan nu de getoonde tabel implementeren. De overeenkomstige schakeling is nu compacter, alleen is het niet duidelijk hoe je er geraakt bent: eerst inputgebaseerd opgelost en dan gezien dat het eenvoudiger is. Als we nu dezelfde implementatie nemen en die aanpassen, komen we ook tot deze implementatie. Vanaf de vorige vertrokken was het niet evident om hier te komen.
De stappen mooi na elkaar uitvoeren geeft meestal wel een compacte representatie maar niet altijd de beste oplossing.

Slide 50: Inputgebaseerd systeem: specificaties zijn ongeveer hetzelfde. We kijken nu niet wat de ingang op de twee vorige klokperiodes was, maar in de vorige en in de huidige: ingang wordt bekeken op hetzelfde moment dat de uitgang gegenereerd wordt. Toestandsdiagram is ongeveer dezelfde. De uitgangen staan nu echter niet bij de toestand maar bij de pijl: de pijl komt overeen met de combinatie van toestand en ingang en dat is wat je bij een toestandsgebaseerd systeeem hebt. Hier zou dat niet veel verschil maken want of w nu 0 is of 1, de uitgang is toch 0 vanuit A. Maar in B is er wel een verschil voor z. Bij een inputsgebaseerd systeem is het iets moeilijker omdat je niet 1 uitgang voor een toestand hebt, maar in een aantal gevallen gaat dat leiden tot een kleiner aantal toestanden omdat je dat niet eerst ergens moet bewaren. In dit geval leidt dat ertoe dat er maar 1 flipflop nodig is terwijl we er daarnet 2 nodig hadden.

Slide 51: De uitgang is nu ok afhankelijk van de ingang. Zie ook overgang!

Slide 52: De rest van de stappen is ongeveer hetzelfde, alleen is het in dit voorbeeld iets simpeler. Je hebt ook veel beperktere kaarten want je hebt maar 2 variabelen. Voor de uitgang speelt ook de ingang mee (niet alleen de toestand).

Slide 53: Als je hier de timing van bekijkt en je vergelijkt met het vorige is er wel een verschil: ingang speelt onmiddelijk mee in de uitgang.

Slide 54: VHDL: om een toestandsdiagram te beschrijven en de implementatie daarvan kan je dat in de Xilinxomgeving gaan tekenen. Die is intelligent genoeg om dat stappenplan zelf te volgen en er zelf de juiste versie uit af te leiden.
In de nieuwste versie kan je dat toestandsdiagram niet meer tekenen want dat werd niet gebruikt. Is even makkelijk in VHDL te gebruiken als dat te tekenen.

Slide 55: VHDL beschrijft hardware: voor ieder stukje heb je hardware. Je kan de synchrone sequentiële schakeling altijd tekenen zoals links op de slide: stuk geheugen waar de toestand in bewaard wordt en dan 2 stukken combinatorische logica om output in te bewaren en de volgende toestand te bepalen.
Rechts is het getekende toestandsdiagram. Hoe dit implementeren? Je kan het op gedragsniveau omschrijven.

Slide 56: Je kan het schema rechts volledig in VHDL beschrijven. Je hebt de ingangen eraan en de twee bits in elke bol op de vorige slide zet je bij uw Output. Dat is uw zwarte doos en verder beschrijf je dan hoe het werkt.
Voor ieder van de blokjes heb je 1 proces. Je geeft aan wat de ingangen zijn, uitgangen moeten niet. Alle mogelijke toestanden die je hebt kunnen daarop komen te staan. Dit is dus een volledige beschrijving van wat rechts staat, op die puntjes na. Wat is er verschillend van de ene sequentiële schakeling naar de andere: ingangen, toestanden kunnen anders zijn, maar groen en de 3 processen blijven hetzelfde. Je kan dus makkelijk een sjabloon maken dat vooor alle synchrone sequentiële schakelingen werkt.

Slide 57: Dit is op zich niet moeilijker dan het toestandsdiagram tekenen. Je hebt bv. een register, wordt ook beschreven in VHDL. Je moet wel invullen hoe

de logica werkt. Niet hoe de codering werkt of de flipflops, de hardware moet dat maar oplossen.

Slide 58: Beschrijft wat er in het toestandsdiagram staat: voor iedere toestand zeg je wat er afhankelijk van de ingangen de volgende toestand zal zijn.

Slide 59: Uitgang bepalen, ook makkelijk. Wanneer je in VHDL iets beschrijft is dat even makkelijk als dat tekenen.

Slide 60: Wat hebben we niet gedaan: flipflops en codering vastgelegd → voor compiler. Als die dat niet goed zou doen, dan kan je hem ook terechtwijzen. Eerste manier: bijkomend attribuut aan toestand meegeven.

Slide 61: We hadden daarnet 3 toestanden nodig en 2 flipflops dus eigenlijk 4 mogelijke toestanden. We hadden van de 4^e toestand een don't care gemaakt. Je moet niet zo zeker zijn dat dat niet kan optreden: bij het opstarten kan je er per ongeluk in terechtkomen. Het kan zijn dat uw schakeling dan nooit meer correct werkt want de voorgeschiedenis gaat wijzigen. Om dat te voorkomen is het veiliger om geen don't cares te gebruiken. Gebruik die extra toestanden om te zorgen dat als je daarin zit zo snel mogelijk terug naar de intitiële toestand te gaan: het gaat even fout zijn maar daarna zorg je voor een correcte werking. Dat kan je doen door de don't cares in te vullen (makkelijk in VHDL: when others).

Chapter 10

Les 10

10.1 Slides: 5_Sequentieel

Slide 62: We gaan de stappen in detail doorlopen en kijken waarom bepaalde keuzes gemaakt zijn. We gaan eerst de specificaties in een toestandsdiagram bepalen.

Slide 63: Modulo 3 teller: 0 1 2 0 1 2 (of omgekeerd). Dat is wat gegeven is. Doordat we weten wat een modulo 3 teller is kunnen we afleiden wat voor in- en uitgangen nodig zijn. We moeten aangeven dat er geteld moet worden want je wilt niet dat er op elke klokflank geteld wordt: count. Direction: in welke richting moet geteld worden. Er is ook een klok (hier niet opgeschreven). Als je naar de uitgang gaat kijken moet je kijken naar uitgangen die de stand van de teller gaat aangeven. M en L: meest en minst beduidende cijfers. Wanneer we die willen gebruiken om te cascaderen hebben we ook een uitgang nodig die zegt wanneer het meer beduidend deel mag tellen. Uit de definitie volgt direct wanneer die uitgang 1 moet zijn. De eerste keuze die we nu moeten maken: toestands- of inputgebaseerd? Op zich niet veel keuze: in dit geval is het inputgebaseerd: het hangt af van de ingangen (onderaan) en de toestand.

Slide 64: Opstellen toestandsdiagram: je begint bij een initiële toestand waarbij je liefst de mogelijkheid voorziet om die in die toestand te dwingen. Voor elke toestand ga je alle ingangscombinaties nagaan en kijken wat de volgende toestand is. Indien nodig creëer je toestanden bij. Dat doe je tot bij alles aangegeven is wat de volgende toestanden zijn en wat de uitgangen zijn. Als het toestandsgebaseerd is staat de uitgang in h, anders op de pijlen.

Slide 65: u_0 met uitgang 00. We tellen naar boven. Nu moeten we alle ingangscombinaties nagaan. Als we niet tellen doet de D er niet toe. De pijl naar zichzelf komt met 2 pijlen overeen: waarbij die 1 is en 0. We vermelden hier ook de uitgang bij. Zoals het hier gedaaan is is zoals op een hybride manier. Er zijn 2 uitgangen bij de toestand en 1 uitgang bij de pijlen. Strikt gezien zou alles bij de pijlen gezet moeten worden, maar je kan het nu ook bij de toestand zetten. Het kan in dit geval dus geen kwaad om de uitgangen bij de toestand te zetten zodat ze voor alle ingangen van de toestand hetzelfde zijn. Je kan dat

niet doen bij Y want die hangt er wel vanaf. u_1: naar boven tellen. d_2: naar beneden tellen. De Y wordt in dit geval 1 want we zitten in de grenswaarde en er wordt geteld. Voor u_0 zijn nu alle mogelijkheden gecreëerd, we hebben nog teostanden waar niet alles gedefiniëerd is → weer 2 nieuwe bij. Ook voor d_2. Uiteindelijk komen we tot het eindresultaat en zijn voor alle toestanden alle ingangscombinaties vastgelegd en wat de uitgangen overal zijn. Dan is het toestandsdiagram klaar.
Dit is niet het enige diagram dat je ervoor kan tekenen (zelfs niet het optimale).

Slide 66: Toestandstabel: komt met hetzelfde overeen, maar een diagram is visueler, een tabel is vaak makkelijker om mee te werken.

Slide 67: De volgende stap is zorgen dat we zo weinig mogelijk toestanden hebben (want dan kunnen we een kleiner geheugen gebruiken). Hoe minder toestanden je hebt, hoe meer rijen met don't cares je hebt. Hoe meer don't cares je hebt, hoe kleiner uw combinatorische schakeling wordt.

Slide 68: De vraag is hoe we twee toestanden die we hebben kunnen samenvoegen tot een nieuwe toestand. Je kan dat als die door elkaar kunnen vervangen worden (equivalente toestanden) → terwijl ze werken kan je het onderscheid tussen de twee niet maken. Waar zie je dat verschil: in uw schakeling zijn er 2 effecten: je hebt een uitgang en je bepaalt wat de volgende toestand is. Als 2 toestanden zorgen dat indien ze tot dan dezelfde ingangen gekregen hebben en op dat moment dezelfde uitgang genereren en naar dezelfde volgende toestand gaan, dan zijn ze equivalent en kunnen we ze door elkaar vervangen. Ze moeten zelfs niet naar dezelfde toestand gaan, mag ook naar equivalente toestanden. Het enige wat we moeten doen is de equivalente toestanden gaan zoeken.
Probleem: is een recursieve definitie: om te weten dat 2 toestanden equivalent zijn, moeten de toestanden waar ze naartoe gaan ook equivalent zijn, maar je zoekt naar equivalente toestanden dus je weet dat nog niet. Daarom werk je iteratief.

Slide 69: Techniek van het boek: partitionering. We gaan alle toestanden in 1 verzameling steken en we gaan die opsplitsen in deelverzamelingen die disjunct zijn en samen de volledige set vormen met dat doel dat het altijd zo is dat in die sets binnen 1 groep alle toestanden die daarbinnen zitten equivalent kunnen zijn (niks spreekt het tegen dat die equivalent kunnen zijn) en toestanden bij verschillende toestanden kunnen niet equivalent zijn. Dit blijf je doen tot er geen wijzigingen meer zijn.
Een keer je daar gekomen bent, kan je alle toestanden binnen een groep samennemen want er is niks dat het tegengaat dat ze equivalent zijn.

Slide 70: Initiëel gaan we alle toestanden in 1 grote verzameling steken. Het makkelijkste om te testen: toestanden zijn niet equivalent als ze verschillende uitgangen produceren. We groeperen dus eerst op toestanden die allemaal dezelfde uitgangen hebben. Je moet niet kijken naar de uitgang op 1 moment maar voor alle ingangscombinaties: 1/1, 1/1, 0/0,... → we starten met 2 groepjes.
Dan ga je itereren en nagaan of in de sets de we nu hebben er tegenstrijdigheden

zitten. Daarvoor gaan we voor zo'n set de volgende toestanden bepalen. Voor de volgende toestanden moet je gaan kijken of de uitgangen in dezelfde groep zitten: voor BDB is er geen probleem, voor CFG ook niet. Voor CEFG: FFEF: OK. ECDG: niet OK want D zit niet in dezelfde groep: F kan niet equivalent zijn met CEG, dus we moeten die afzonderen. Door het feit dat we nu nieuwe groepjes hebben moeten we alle testen opnieuw doen. Nu kom je in de problemen bij ABD: F zit niet meer in hetzelfde groepje als C en G.
Dan splitsen we weer op en zo voort tot je niets meer hebt dat tegenspreekt. Alle toestanden die we nu hebben zijn toestanden die equivalent *kunnen* zijn. We besluiten nu dat ze waarschijnlijk wel equivalent zijn (binnen 1 groep).

Slide 71: Toegepast op de modulo-3 teller. We beginnen met te kijken welke dezelfde uitgangen hebben. Wat zijn dezelfde uitgangen: 000/000/001 bv. Zo maken we 3 groepjes. Voor ieder van die groepjes moeten we nu nagaan wat de volgende toestanden zijn. Wat zien we nu: als we $u_0 d_0$ samennemen, gaan die naar toestanden van hetzelfde groepje. Ook bij de twee anderen zijn er geen problemen. We zien dus vrij snel wat de equivalente toestanden zijn → van 6 naar 3 toestanden.
Dat hadden we in principe van in het begin kunnen zien: er werd artificiëel het onderscheid gemaakt tussen naar boven of naar beneden tellen. Maar misschien wist je dat niet zeker. Daarom is het belangrijk om niet te weinig toestanden te nemen, teveel kan op zich geen kwaad, die gaan weg bij het minimaliseren. Je moet in het begin dus best op veilig spelen, bij de minste twijfel die je hebt.

Slide 72: We hebben nu de optimale toestandstabel om van te vertrekken. De volgende stap is het implementeren van het geheugen: hoe de toestanden voorstellen in het geheugen en hoe gaan we het geheugen maken?

Slide 73: Coderen van de toestanden: hoe vertaal ik de symbolische benaming voor de toestanden die ik heb naar een bepaalde bitcombinatie? Daar zijn heel wat mogelijkheden. Strikt gezien: bij n toestanden heb je $log_2 n$ flipflops nodig. Hoeveel coderingen kan je doen met die n toestanden: n! mogelijkheden om te kiezen. → Het minste dat erbij komt zorgt dat dat getal enorm hard stijgt. Je moet ook altijd naar boven afronden. Alle toestanden gaan uitproberen is onmogelijk in veel gevallen. Kunnen we daar zinvolle eliminatie aan doen? → Niet echt.

Slide 74: Je kan niet zomaar zeggen dat je bepaalde toestanden eens gaat bekijken. Het maakt erg veel verschil dewelke je kiest. In het boek staan er enkele voorbeelden van dezelfde toestandstabel met een andere codering: je kan tot verschillende hardware en verschillende kostprijs komen.
In de praktijk kan je dat zelfs niet aan uw PC overlaten om daar het beste uit te kiezen. Je moet ergens een keuze maken en veronderstellen dat die zinvol is: keuze baseren op andere dingen. Traditioneel zijn er een drietal coderingen die meestal gebruikt worden. Je kan voor ieder van die coderingen nagaan wanneer dat een goede manier van werken is. Zelfs als je 1 van de coderingen kiest, is het niet automatisch zo dat er maar 1 codering bij hoort, in veel gevallen heb je dan nog de keuze uit verschillende coderingen. Er zijn nog een aantal verfijningen mogelijk daarna.

Slide 75: Eerste: voor de hand liggende (als je niet nadenkt). Die is niet altijd slecht: is redelijk goed wanneer die toestand ook met iets fysisch overeenkomt, als die gekoppeld is aan iets. Typisch voorbeeld: teller. In veel gevallen is dat niet zo optimaal. Heeft vooral te maken met wanneer je van een toestand naar een andere gaat, er regelmatig meerdere bits moeten veranderen. → Overgangsverschijnselen. Het kan zijn dat uw tussenresultaten totaal onnauwkeurig zijn. Je moet hier rekening mee houden, zelfs als die verschijnselen niet te lang duren.
Elke verandering van een bit verbruikt vermogen: als 3 bits veranderen: 3 keer vermogen dat verbruikt wordt. Daarenboven, om een bit te doen veranderen, als je die op 1 of 0 zet, dan heb je daar hardware voor nodig om die set of reset aan te sturen. Elke bitverandering die moet gebeuren, daar komt hardware mee overeen. Hoe meer bitveranderingen moeten gebeuren, hoe meer hardware je nodig hebt.

Slide 76: Toegepast op een voorbeeld.

Slide 77: Om aan nadelen tegemoet te komen: minimumcodering. Bij elke overgang zo weinig mogelijk bits laten veranderen: minder vermogenverbruik en waarschijnlijk minder hardware nodig. Typisch het geval bij CMOS implementatie.
Links: normale teller. Als je nagaat hoeveel bits er telkens veranderen: gemiddeld gezien anderhalve bit per wijziging. Je kan die ook op de rechtse manier laten veranderen: er verandert altijd maar 1 bit. De linkse verbruikt dus anderhalve keer zoveel vermogen als de rechtse. Die rechtse heeft wel een nadeel ook, want links kan de toestand rechtstreeks gebruikt worden als uitgang, bij de rechtse gaat dat niet (het is niet omdat de toestand 10 is dat de uitgang dat ook moet zijn) → eventueel nadeel.
In vele gevallen gebruik je die toestanden niet rechtstreeks dus maakt het niet uit. Als het dus geen evidente teller is, heb je er meestal wel het voordeel aan van de minimum bit change te nemen.

Slide 78: Passen we dat toe, zien we dat er geen eenduidige oplossing is. Je moet nagaan wat het gemiddeld aantal bits is dat per overgang verandert. Stel dat alle overgangen even waarschijnlijk zijn, wat je ook kiest, het kan nooit zo zijn dat al die overgangen maar 1 bit verandering hebben. Er gaan altijd 4/6 overgangen zijn die 1 bit veranderen en 2/6 gaan 2 bits verandering hebben. Er zijn dus verschillende combinaties waarbij je geen onderscheid kan maken. Als je het gaat nagaan, als je gaat kijken wat we daarnet als straightforward oplossing hadden, die was ook OK. Je gebruikt dan uiteraard liever de straightforward implementatie.

Slide 79: One-hot encodering: niet zo weinig mogelijk flipflops gebruiken, wel zoveel flipflops als nodig: n. Dit is een verspilling van flipflops, maar je kan bijkomende voorwaarden oplegggen: er is altijd exact 1 flipflop die 1 bevat. Alle anderen zijn dan 0 en alle andere mogelijke combinaties zijn niet toegestaan (niet alles mag 0 zijn bv). Je kan dan elke toestand laten overeenkomen met 1 flipflop. Dit maakt uw ontwerp makkelijker. Er is 1 plaats waar het bijna

als vanzelfsprekend gebruikt wordt: bij FPGA's, die gebruiken meestal one-hot encodering. Waarom: die manier van werken past ideaal bij wat er in een logische cel zit: stukje combinatorische logica waarvan je volledig zelf kan bepalen wat die doet. Enige beperking: je kan maar een beperkt aantal ingangen gebruiken (klein stukje combinatorische logica). Wat zit er nog in: flipflop. Die zit er of je die nu nodig hebt of niet. Het feit dat je beschikt over een kleine combinatorische schakeling met gratis flipflop maakt dat de logica die je nodig hebt om de flipflop aan te sturen beperkter wordt: als alle toestanden in 3 flipsflops zit en je moet die aansturen is dat ingewikkelder dan wanneer je 8 flipflops hebt die je elk afzonderlijk kan aansturen → past goed bij hoe een FPGA gebouwd is.

Slide 80: Toegepast op het voorbeeld.

Slide 81: Tweede stukje: welke flipflop gaan we gebruiken?

Slide 82: 4 types met eigenschappen. Je hebt op zich niet echt een goed idee van wat te kiezen. Wel algemeen, maar geeft geen 'echte' raad. Voordeel: maar 4 types. Als je echt naar het optimale zoekt, kan je 4 ontwerpen maken en daarna beslissen welke van de 4 je gaat gebruiken.

Slide 83: Soms zijn er andere dingen en is het het snelst ontworpen met de D-flipflop.
Bij FPGA: alleen D-flipflops. Je kan de anderen wel gaan emuleren, maar dat is vrij onnozel.

Slide 85: Voor elk type flipflop gaan we nu kijken naar de laatste stap. We beginnen met JK. Je hebt 2 flipflops want 2 toestandsbits. Iedere flipflop heeft 2 ingangen: J en K. We gaan eerst de kaarten voor de meest beduidende opstellen. Vanboven: toestandstabel. We gaan in die tabel kijken hoe we de flipflop moeten aansturen om alles zo efficiënt mogelijk te laten gebeuren → excitatietabel: van 0 naar 0 overgaan: welke ingang moet je aanleggen. ⇒ In die kaarten invullen. Hier zijn onmiddelijk 2 velden ingevuld.
Zo verdergaan voor de volgende enz. Je gebruikt de excitatietabel om dus wat in de toestandstabel staat om te zetten naar een Karnaughkaart.
Gaten: 4^e toestand die er niet instaat. Je weet dat die nooit optreedt dus je vult die met don't cares.
Je kan nu hetzelfde doen voor de minst beduidende flipflop. De 4 ingangen zijn nu voorzien van kaarten. Je moet ook nog de uitgang gaan bepalen. M en L moet je niet doen, dat is hetzelfde als Q_0 en Q_1. Maar Y moet je nog wel maken. Ook hier is een kolom met don't cares.
Zo heb je dus 5 Karnaughkaarten die je kan gaan implementeren.

Slide 87: We hadden gekozen welke flipflops het waren, waarvan voorzien was dat die in een initiële toestand gebracht kunnen worden (wat je hebt in de vorige stap). Dan kaarten implementeren.

Slide 88: Fout gemaakt: 5 afzonderlijke Karnaughkaarten opgelost. Dat mag niet. Een combinatorische schakeling is er een met 4 ingangen en 5 uitgangen. We hebben het beschouwd als 5 schakelingen met 1 uitgang. In het vorige hoofdstuk zagen we dat we zoveel mogelijk moeten zoeken om te delen tussen de kaarten. Grijze bolletjes: tellen niet mee: die berekenen het inverse van Q. Maar het inverse van Q ga je niet berekenen door Q te inverteren, je gebruikt gewoon de uitgang waar de inverse van Q uitkomt. Als 1 grote schakeling bekijken: je moet minder poorten implementeren. ⇒ Bekijk ze allemaal samen om tot een minimale oplossing te komen.

Slide 90: Voor set-reset. Die voor de Y is hetzelfde als in het vorige geval: de schakeling van Y hangt alleen af van de huidige toestand en van de ingang, niet van het type flipflop.

Slide 91: Implementatie voor SR. Vergelijking met vorige: hier is meer hardware nodig ten opzichte van daarnet. Dit komt omdat er in deze kaarten minder don't cares staan dan daarnet. Dit is dus een duurdere oplossingen.

Slide 93: D: geen don't cares, maar er is maar 1 ingang (voordeel) → 1 kaart voor elke flipflop. Er staan geen don't cares in behalve voor de toestand die niet gebruikt wordt.

Slide 94: Implementatie: ook ingewikkelder. Dit is tot nu toe de duurste oplossing. Wat wel is: je kan alleen dit bekijken voor een FPGA. Kostprijs bij FPGA: met logische cellen: schakeling met 4 ingangen en een flipflop. Hier hebben we een schakeling met 4 ingangen en een flipflop. We hebben hier 3 logische cellen.

Slide 95: Toggle-flipflop. We verwachten dat dit de beste oplossing geeft omdat het over een teller gaat.

Slide 96: Kostprijs van 32: niet de beste maar ook hier is het niet op een optimale manier gemaakt (combinatorische schakeling). Als we het hier optimaliseren kom je op de schakeling van **Slide 97**.

Slide 97: Ook dit is in principe niet het goedkopste (JK was dat). Dit illustreert dat de algemene bedenkingen van daarnet dus niet veralgemeend kunnen worden: het is niet omdat het een teller is dat je maakt dat je een T-flipflop moet hebben. In een JK flipflop zit ook een togglemogelijkheid dus kan ook zo gebruikt worden en heeft het bijkomend voordeel van de don't cares. Besluit: je kan niet op voorhand kiezen wat het optimale zal zijn.

Slide 98: Speciaal geval van one-hot codering. Waarom kan je het eenvoudig maken: door het feit dat een flipflop gekoppeld is aan een toestand kan je bepalen wat er aan de ingang komt. Je moet geen kaarten opstellen, gewoon door na te denken weet je hoe het aangestuurd moet worden. Voor een D flipflop: 1 hebben: 1 aan de ingang aanleggen. Alles wat ervoor zorgt dat we in die toestand zullen zitten, dat moeten we gebruiken om te zorgen dat er bv.

een 1 opstaat.
Bij een SR: we komen toe in een 1 en de reset: we gaan weg, het moet 0 worden.

Slide 99: Op vb toepassen: 3 toestanden dus 3 flipflops. Let op dat je die op de juiste manier initialiseert. We initialiseren die zodanig dat die niet allemaal 0 worden. Hoe bepalen we de ingang van die flipflop: kijken wat ervoor zorgt dat we in een van de flipflops gaan zitten. Kijken we naar S_0: als we naar die toestand gaan $\rightarrow$ alle pijlen die daarop toekomen, we moeten zorgen dat daar een 1 op staat. Dat geeft de schakeling onder de eerste flipflop: we zaten in S_0 en C = 0. Een andere pijl die er toekomt is die van CD=11 uit S_1. Er is nog een derde pijl die er toekomt waarbij we in toestand S_2 zaten. De ingangen waren daar 1 en 0. Nu hebben we alles aangegeven waardoor na de volgende keer deze flipflop op 1 moet staan, gewoon door naar het toestandsdiagram te kijken. Zo kunnen we dat flipflop per flipflop doen en dan krijgen we de getoonde schakeling. De combinatorische schakeling die er staat ziet er op zich niet eenvoudiger uit dan die van daarnet.
Normaal is dit compacter dan bij de andere omdat we 1 AND poort hebben per pijl die toekomt en normaal heb je geen toestanden waar 27 pijlen op toekomen. Normaal heb je veel toestanden en een beperkt aantal pijlen en daardoor blijft dit normaal beperkt.
We moeten nu wel nog Y gaan realiseren. Die is hier niet hetzelfde omdat we een andere codering hebben toegepast en deze hangt af van de codering en de ingang. We hebben ook de uitgangen M en L. Normaal zijn die geen uitgang van een flipflop, die moeten evengoed berekend worden. Nu is dat niet nodig: L is alleen 1 in S_1. M is alleen 1 in toestand S_2. Dus hier is het toevallig zo dat je het kan gebruiken zoals hier. Was het een normale teller geweest, was dat niet zo want dan gingen in toestand S_3 M en L 1 zijn. Y: we moeten kijken wanneer Y 1 gaat zijn. Door het feit dat het zo eenvoudig is, is er een duidelijke koppeling tussen toestand en flipflop dus moeten we de stap met kaarten en excitatietabellen niet doen.

Slide 100: Toegepast op een set-reset flipflop. De set is de pijlen die erop toekomen. Reset: pijlen die ervan vertrekken. Als je in dezelfde toestand blijft hoef je niks te doen: niet setten of resetten.
Wat komt erop toe in S_0: 2 pijlen. Er vertrekken ook 2 pijlen. Bij het vertrekken vertrek je telkens van S_0. Er moet geteld worden, of dat nu naar boven of naar beneden is, dat maakt niet uit.
Hier is de Y wel hetzelfde als de vorige keer omdat we dezelfde codering gebruiken. Het kan hier ook nog zo zijn dat je dingen kan hergebruiken. Het is hier veel moeilijker om dingen te hergebruiken. Je zou hier ook kaarten kunnen gaan opstellen maar meestal win je daar niet zoveel bij.

Slide 102: Bij de combinatorische schakelingen gingen we naar het kritisch pad zoeken om te bepalen hoe snel de schakeling werkte: langste vertraging van *een* ingang naar *een* uitgang. Bij een sequentiële schakeling is dat niet zo: wordt bepaald door de klokfrequentie. Het is niet door de klokfrequentie te verhogen dat je meer gaat doen, maar hoe hoger die frequentie is, hoe meer stapjes van uw algoritme je per seconde kan doen dus hoe sneller het werkt. Die maximale frequentie hangt hier niet af van de grootste vertraging tussen de ingang en de

uitgang, die hangt af van het moment dat er een klok komt tot het moment dat je terug een klok kan geven. → Grootste vertraging tussen 2 klokflanken. Bv. in dit geval: stel dat we beginnen te klokken en de rechtsbovenflipflop gaat veranderen. Die uitgang gaat veranderen en aan de ingang van de linkse flipflop komen nieuwe waarden. Hoe lang wachten: vanaf klokflank: vertraging door ingang, combinatorische schakeling en dan nog setuptijd wachten want anders kan je in metastabiliteit gaan. Je moet zorgen dat je minstens de setuptijd voor de klokflank stabiel was. De grootste vertraging voor de edge-triggered flipflop was in dit geval 5. Dan door de 2 poorten (je zoekt de grootste: meeste vertraging) & setuptijd. Als je dat uitrekent kom je in dit geval aan 10.
Als je de eenheidsvertraging van 1ns gebruikt, dan kan deze schakeling aan een maximale frequentie van 100MHz werken. Als die sneller gaat gaan, gaat die niet meer betrouwbaar werken. We willen dit wel altijd zo hoog mogelijk omdat we dan meer kunnen verwerken per tijdseenheid.

Slide 103: Asynchrone sequentiële schakelingen gaan we niet bekijken in de oefenzitting. We gaan die bekijken omdat het belangrijk is om te beseffen waar de verschillen liggen en waarom het moeilijker is om met een asynchrone schakeling te werken. Je moet altijd uw uiterste best doen om die niet te gebruiken.

Slide 104: In een asynchrone schakeling wordt geen klok gebruikt. Je hebt ingangen die veranderen en na verloop van tijd hebben die effect. Je kan niet gaan kijken wat op de ingang staat en kijken wat voor gevolg die hebben. Die kunnen op willekeurige ogenblikken veranderen. Dit is wat het moelijk maakt om het te ontwerpen want je hebt het niet meer zo onder controle.
We hebben al zo'n asynchrone schakeling gezien: set-reset latch.
Hoe begin je eraan om die te ontwerpen? Liefst zouden we synchrone toepassen op de asynchrone. Dat is vrij goed mogelijk, alleen moeten we dat in onze gedachten aanpassen want het grote verschil is dat er geen flipflops inzitten: geen geklokte geheugenelementen. Het geheugen zit 'm in die terugkoppeling en het feit dat die poorten een zekere vertraging hebben.
We kunnen het ook tekenen zoals na de overgang: het blauwe is een imaginair onderdeel waarin alle vertragingen samengezet zijn. De totale vertraging van de terugkoppeling zit in dat moment. Dan kunnen we een onderscheid maken: aan de ingang staat wat de volgende toestand wordt, wat aan de uitgang staat is de volgende toestand. Dit is een gedachtenoefening, je kan dat niet zo gaan implementeren. Daarmee is niet alles opgelost. Je kan voor deze schakeling de toestandstabel gaan opstellen → heel duidelijke, eenduidige werking.
Verschil met synchrone: stabiele toestanden en niet-stabiele toestanden: stel dat we in toestand 1 zitten en we gaan dan resetten (bedoeling dat in 0 komt). We brengen de gepaste ingangen dan aan, voor het vertragingselement komt dan 0 te staan en een tijd later komt op Q ook 0 te staan. Je kan wel zeggen dat je moest overgaan naar 0, maar in de niet-stabiele 0 blijf je niet lang. Alles is hier stabiel, automatisch een tijdje later zie je dat ook op de uitgang. We hebben stabiele toestanden en toestanden waarin we niet kunnen blijven zitten. We hebben geen klok waarbij we kunnen zeggen dat we "nu" in stabiele toestanden zitten. Veranderingen worden gedeeltelijk verwerkt, dan ga je andere ingangen aanpassen waardoor tegengestelde veranderingen kunnen gebeuren in-

wendig. Je kan hier geen rekening mee houden in het ontwerp. We gaan ons dus beperken tot de fundamentele modus: als er iets verandert wachten we tot het effect helemaal uitgewerkt is, tot we in een stabiele werking gekomen zijn. Hoe realistisch is dit? Jammer genoeg is het zo dat in realiteit je dat niet kan eisen. Je kan hopen dat het niet gebeurt, maar als die ingangen niet onder uw controle vallen, dan ben je daar nooit zeker van dat dat inderdaad niet gebeurt. Wat doe je in de praktijk: bij het ontwerp veronderstellen dat het niet gebeurt en daarna zo grondig mogelijk trachten te testen onder de meest absurde ingangscombinaties om te zien of het werkt zoals je dat wil: misschien werkt het ook wel als je twee dingen bijna gelijktijdig verandert.
Rond setup en hold mogen de ingangen niet veranderen.

Slide 105: Als we die veronderstellingen aannemen kunnen we dezelfde manier van werken aannemen als het vorige, alleen hebben we nu 4 stappen in plaats van 5. → Er zit geen flipflop meer in. Elke stap is hier we veel moeilijker.

Slide 106: Opstellen van de toestandstabel. Je kan geluk hebben dat het duidelijk is welke toestanden er zijn, ook in een asynchrone sequentiële schakeling. Maar als die wat ingewikkeld in mekaar zit en dat kan zelfs met een heel eenvoudig voorbeeld: 2 stukken hardware die totaal onafhankelijk van elkaar werken (wel 2 aparte klokken, maar geen gemeenschappelijke klok). Tegenover elkaar werken die asynchroon. En toch wil je informatie uitwisselen tussen die twee. Je wil iets hebben om een handshaking te doen zodanig dat je weet wanneer het goed is om een verandering te doen.
Werking: E geeft aan of informatie opgegeven kan worden. I geeft aan of er informatie is: stijgende flank geeft aan dat je informatie mag beginnen uitwisselen tot E zegt dat het genoeg geweest is (E = enable). Dan stopt het ook met informatie uitwisselen. Je wacht niet tot I 1 is, maar tot I 1 wordt, dan pas wordt Q = 1. Q blijft 1 zolang E 1 blijft.
Hoeveel toestanden zijn er nodig om dit te beschrijven? Je weet dat er geen geheugenelementen inzitten, dus je kan zeggen wat er allemaal verschillend kan zijn: alle mogelijke combinaties van in- en uitgangen. → 8 toestanden.
Er zijn al 2 onmogelijke toestanden. Je zal dus waarschijnlijk niet meer dan 6 toestanden hebben.

Slide 107: Tabel met 6 toestanden, we nemen een toestandsgebaseerd systeem (input zou ingewikkelder zijn). We weten wat de uitgangen ervan zijn. Onderaan staan de 8 mogelijkheden van de in- en uitgangen en de toestanden erbij. Als we daar naar een rij kijken, dan blijft het normaal in de toestand an die rij, dat is dan ook de eerste pijl die je kan tekenen. Dit geldt voor elke rij. Dan moet je de overgangen nakijken. Bv.: je zit in a en E verandert: van 0 naar 1. Je moet ook kijken naar I hier. Als I hetzelfde blijft, ga je naar b. Als I verandert ga je naar c enz. Als E constant blijft (= 1) en I = 0 en gaat naar 1, dan gaat Q ook veranderen: van 0 naar 1. Dan ga je niet van b naar d maar van b naar f. Als je alle combinaties bent nagegaan, in de veronderstelling van die fundamentele modus, dan doen alle resterende plaatsen zich niet voor. Je mag die met don't cares invullen want die mogen niet optreden.
Nu heb je uw tabel. Er is een duidelijk verschil: niet bij de synchrone schakelingen: don't cares in de toestandstabel, bij de vorige was dat niet. Bij asynchrone

ga je dat altijd tegenkomen door die fundamentele modus.

Slide 109: Nu moeten we gaan minimaliseren. Ook hier gaan we minimale toestanden zoeken. In de meeste gevallen levert dit niet zoveel op, juist door die don't cares. 2^e trukje: gebruik maken van compatibele toestanden.

Slide 110: Verschil met equivalente toestanden: bij equivalente toestanden zochten we sets van equivalente toestanden die we samennamen. Bij compatibele doen we dat niet: wat we tot hiertoe hebben, als we die 2 samenzetten, kunnen we dat doen of niet?
Als we don't cares tegenkomen kunnen we die eventueel samennemen. We gaan nooit een set van compatibele toestanden hebben, je gaat het altijd paarsgewijs bekijken. Dat heeft tot gevolg dat het niet is omdat A compatibel is met B en B met C, dat A compatibel is met C (bij equivalentie was dat wel zo).

Slide 111: Stel dat we de gegeven toestandstabel hebben. Eerst gaan we equivalente toestanden samennemen. Bij die don't cares zitten we met een probleem want je weet niet op voorhand wat het gaat worden. Je kan wel een toekenning doen aan een don't care, maar je kan geen set maken waarvan die don't cares wel in orde komen, dat die door het juiste zullen vervangen worden. Het enige dat je kan doen is zeggen dat als er overal een don't care staat voor die ingangscombinatie het geen kwaad kan en er wel iets gekozen zal worden en wat gekozen wordt zal bij implementatie ook geen probleem opleveren.
Alleen toestanden waar in dezelfde kolommen don't care staan kunnen we samennemen. Je krijgt onmiddelijk veel meer setjes en is er niet veel meer samen te nemen. → Je kan maar weinig dingen samennemen.

Slide 112: We gaan nu telkens 2 toestanden nemen en kijken of die compatibel zijn. Je gaat er vanuit dat wat je nu samenzet, het laatste gaat zijn wat je samenzet. Je kan bij F de don't care bij 00 vervangen door A en voor 10 door C bij A bij 11 door H. Je kan zo verbindingen maken tussen toestanden waarbij dit mogelijk is op deze manier. Je ziet dan uiteindelijk dat BMH compatibel zijn met elkaar: ze zijn allemaal compatibel met elkaar. Dat is niet zo voor F: F is compatibel met J en A, maar A is niet compatibel met J. Je kan dat nagaan: de don't care van F vervangen door A of B, maar niet door beiden tegelijkertijd. Je neemt er zoveel mogelijk samen. Die linkse cluster is een twijfelgeval. Je kan AF nemen en JG. Er is gekozen voor AF en CJ.

Slide 113: Je krijgt nu een kleinere tabel en veel don't cares zijn verdwenen omdat je samengezet hebt. Ook hier moet je gaan itereren en weer nagaan. C en G zijn compatibel, maar bij 01 heb je C en G. Als je ze echter samenneemt is dat geen probleem en is dat dezelfde toestand. Je kan zo blijven itereren tot er niks meer samen te nemen is.

Slide 114: Equivalente toestanden: met dezelfde uitgang. Dan die met don't care in dezelfde kolom: geen enkele. 0 equivalente toestanden dus. Compatibiliteit: zo zijn er wel veel. Je kan niet zeggen of het beter is om het een samen te nemen met iets anders, je kan ook andere dingen samennemen. → Maar 3 toestanden nodig, geen 6.

Chapter 11

Les 11

11.1 Slides: 5_Sequentieel

Slide 115: Bij asynchroon is het moeilijker om uit te voeren want je hebt compatibele toestanden hier. Vandaag bekijken we de twee laatste stappen: coderen en implementeren.

Slide 116: Het coderen op zichzelf is hetzelfde, maar we gaan dat ook moeten gebruiken om te zorgen dat een aantal problemen die vaak voorkomen minder voorkomen: races. Wat is een race: aan een ingang verandert er 1 ding en dat heeft effect op 2/meer uitgangen/interne waarden die de toestand voorstellen. Dan kan er een wedstrijd onstaan welk van de twee het eerst verandert. Dat is niks nieuws en bestaat ook bij synchrone sequentiële schakelingen, maar daar helpt de klok u om die overgangsverschijnselen te laten uitsterven. Hier heb je geen klok dus zodra er iets verandert wordt dat teruggekoppeld en heeft dat onmiddelijk effect en je kan dat niet tegenhouden.
Als er in uw ontwerp dus 2 dingen veranderen bij het veranderen van 1 ingang, dan weten we dat dat in praktijk niet gebeurt, dan is er de mogelijkheid tot problemen. Het is maar als dat leidt tot een verkeerde toestand dat we dat een kritische race noemen en dat dan uw schakeling niet correct werkt. Het kan zijn dat je niet naar een stabiele toestand gaat, maar heen en weer gaat tussen toestanden.

Slide 117: Voorbeeld: als we nu 1 ingang laten veranderen (bv. van 01 naar 11), dan gaan we via een onstabiele toestand naar een stabiele toestand. Het is nu niet zo dat omdat het de juiste implementatie is, dat het de juiste werking gaat hebben. Er moeten twee bits veranderen, dus dat zal nooit tegelijkertijd gebeuren. De schakeling werkt in het voorbeeld perfect. Maar als we nu dezelfde logische functie op een andere manier implementeren (vanonder): in plaats van NAND-NAND AND-OR gezet. Er gebeurt dan wat links getoond wordt: die uitgang/toestand gaat van 01 naar 00 naar 10 (tot zover OK), maar dan terug naar 01 00 10 en dat blijft zich eindeloos herhalen. → Cycle: we komen niet in 1 toestand terecht maar cyclen tussen toestanden.
Waarom is dat verschil er: de tweede verandering op a komt na de verandering op b en vanonder is dat niet het geval, daar gebeurt dat veel sneller. Dat hangt

af van de vertragingen die in het systeem zitten (we hebben tragere poorten gebruikt). Waarom blijft zich dat eindeloos herhalen: het pad via de OR naar de onderste AND zal zich eeuwig herhalen. Hier is ook geïllustreerd waar het aan te wijten is: door het feit dat er toevallig meer vertraging op zit dan in het andere geval. Dus met een beetje meer vertraging werkt die schakeling niet meer.
Hoe kunnen we daarop ingrijpen? Uitmeten en proberen en zien wat de vertragingen exact zijn en er zo rekening mee houden. Dit is echter moeilijk om bij het ontwerp te doen, je moet dan de exacte vertragingen kennen, is niet gekoppeld aan de logische functie.
Kritische race: het komt in de foute toestand terecht. Dit is moeilijk aan te tonen met het gegeven voorbeeld, maar je kan dat inzien wanneer je a' steeds smaller laat worden (en als dat naar 0 gaat, die breedte), dan krijg je bij y_1 ook altijd 0 en kom je uiteindelijk in 00 terecht en dat is fout.
Wat is de oplossing: gaan inwerken op de vertraging (zeer moeilijk).

Slide 118: Andere manier: zorgen dat nooit 2 uitgangen tegelijkertijd veranderen. Dit kan door een geschikte codering te kiezen: zorgen dat er altijd maar 1 uitgang tegelijkertijd verandert. Daar kan je voor zorgen bij het ontwerp (in de stap dat je de codering doet): zorgen dat dat nooit gebeurt.
Hoe begin je daaraan: minimumbitchangecodering doen. Je gaat proberen altijd naar een 1-bitchange codering te gaan. Dat lukt uiteraard niet altijd. Bij synchrone sequentiële schakelingen ging dat ook niet, dus dat kan hier evengoed optreden. Hier heb je wel een aantal mogelijkheden meer om dat nog opgelost te krijgen.
Als je niet altijd 1 bit kan laten veranderen kan je gebruik maken van tussentoestanden: bij een geklokt systeem, als je naar een toestand gaat en je moet naar een volgende gaan duurt dat 1 klokperiode langer, hier is dat zo niet: je gaat naar een instabiele tussentoestand die onmiddelijk effect heeft en iets later zit je in de finale toestand: het duurt iets langer maar je komt er wel op een zo kort mogelijke tijd, je moet niet zo heel lang wachten.
We gaan dus niet de beperking opleggen dat we zo snel mogelijk moeten werken, we laten toe om naar een tussentoestand te gaan. We zorgen dat via die tussentoestanden altijd maar 1 bit verandert: in de tijd na elkaar proberen doen (en zo kort mogelijk op elkaar) $\rightarrow$ zo ontstaat er geen race.
Als je bv. van 11 naar 01 gaat: de tweede ingang laat je van C naar D (niet-groen) en dan naar de groene D. Als je dat wil doen, moet je twee bits laten veranderen. Welke oplossing is dan gekozen: niet van niet-groene D naar groene D, maar van groene C naar B-D (de D in de rij van B) en zo naar de groene D.
Je kan hier ook gebruik maken van de don't cares: we hebben die hier bv. ingevuld met 11, maar je kan die ook bijkomend als tussentoestand gebruiken. Da's dus een andere mogelijkheid.
Laatste hulpmiddel: toestanden toevoegen (doe je normaal niet bij synchrone schakelingen want dan komen er bijkomende klokcycli bij). Hier mag je dat als dat geen stabiele toestanden zijn.

Slide 119: (niet te kennen op het examen, gewoon het principe, de redenering kennen): We vertrekken van de tabel linksboven. Hoe kan je zo vlug mogelijk detecteren dat er geen 2 bits tegelijkertijd mogen veranderen: de twee tabellen

daaronder: tabel en toestandsdiagram. Alle stabiele toestanden krijgen een nummer: 7 stabiele toestanden met elk een nummer. Op de andere plaatsen, op die tijdelijke toestanden, daar vul je het nummer in waar je uiteindelijk naartoe moet, niet de tussentoestand. Waarom: dan zie je wat je allemaal kan gebruiken als tussentoestand om in die bepaalde uiteindelijke toestand terecht te komen. Dan ga je een code toekennen. In dit geval zijn er maar 4 toestanden, die zie je dan in de transitiediagram. Alle overgangen die je hebt naar een stabiele toestand toe, die ga je beschrijven in dat diagram. Niet alleen rechtstreeks naar de stabiele toestand toe, maar ook naar tussentoestanden om daar terecht te geraken.
Bv. van C naar A in eerste kolom moet je een overgang maken. Je komt dan uiteindelijk in toestand 1 terecht, dat vermeld je er ook bij. In de derde kolom moet je alle tussentoestanden ook vermelden: van A naar B, van A naar C en van A naar D. 7 staat hier heel dikwijls in: er zijn heel veel mogelijkheden om van gelijk waar in die 7 terecht te komen. Dat wordt hier grafisch voorgesteld.
In dit geval is het eenvoudig te zien wanneer je nooit 2 bits gaat hebben die veranderen. In dit geval is dat alleen als je via een diagonaal gaat. Alle andere stappen zijn altijd maar 1 bit. Dus voor dit soort transitiediagram kan je de eis zetten dat er geen diagonalen gebruikt mogen worden.
Hoe raak je die kwijt: van A naar C moet je niet rechtstreeks gaan, je kan van A naar B naar C gaan, daar staat ook telkens een 7 op. Hetzelfde voor A naar D en D naar C.
Van die 1 en die 3 geraak je niet af omdat er geen weg aan de buitenkant voorzien is om van A naar C te geraken in dit geval. Dus dit is geen goede codering.
Je kan dan andere coderingen proberen: rechtse toestandsdiagram. Hier heb je een don't care in de kolom waar de 4 instaat dus die kan je proberen te gebruiken voor de 4 die je weer hebt op de diagonaal. Je kan die gebruiken zoals in de tabel en transitiediagram vanonder getoond. Zo heb je dan het probleem opgelost: codering waarbij er telkens maar 1 bit verandert en je een redelijke garantie hebt dat er geen (kritische) races optreden. Je moet dan nog die code vertalen, maar die staat er eigenlijk al naast. Zo heb je dan uiteindelijk de oplossing.

Slide 120: Links: geminimaliseerde toestandstabel. De vraag is nu wat de beste codeering is. Je kan er een transitiediagram bijzetten met de codering rechts ervan. Die 6 krijgen we niet weg. In de kolom waar de 6 staat is er geen don't care dus je kan daar geen don't care vervangen. Het enige wat er dus nog overschiet is een toestand bijmaken: transitiediagram eronder. Als je dat invult krijg je de codering in de tabel linksonder.

Slide 121: Met die don't cares moet je opletten: het kan zijn dat je in een van de toestanden terechtkomt, hoe geraak je er dan uit? Nadat je het implementeert kan het zijn dat het allemaal stabiele toestanden geworden zijn. Je kan dat zien door die onderste tabel te maken. Bij de uitgang is er wel een don't care blijven staan, normaal doe je dat niet maar hier kan dat geen kwaad want de overgang die je maakt zijn allemaal door tijdelijke toestanden.
In principe heb je hier ook een race want je gaat van 11 naar 00, maar in dit geval is dat geen kritische omdat alles waar je daar ook terecht komt in die kolom, je gaat daar altijd naar 00. Het is niet dat je altijd naar een andere stabiele toestand kan gaan. Je gaat hier wel overgangsverschijnselen hebben,

maar je kan vrij zeker zijn dat het niet voor problemen gaat zorgen.

Slide 123: Bij de implementatie moet je ervoor zorgen dat er geen hazards optreden. Een hazard is een (normaal) tijdelijke niveauverstoring. Dat kan je niet alleen hebben bij asynchrone schakelingen, dat heb je bij alle combinatorische schakelingen, alleen zorgt het daar normaal niet voor problemen (gewoon overgangsverschijnselen). Hier, ten gevolge van die terugkoppeling en het feit dat we geen klok hebben, kan elk overgangsverschijnsel zich gaan stabiliseren.
Er zijn twee types hazards: statische hazard (signaal moest constant blijven en je ziet even een verandering), dynamische hazard: op de plaats dat je verwacht dat het verandert maar je merkt dat het niet 1 keer verandert: het verandert, gaat even terug en verandert dan uiteindelijk: verandering teveel. Dit is storend wanneer je dit signaal gebruikt als klok.
Weerom heeft het te maken met vertraging. Het is op het moment dat je het tijdsgedrag bekijkt dat je dit soort zaken vaststelt.

Slide 124: Waaraan is die hazard nu te wijten: hou x&z constant op 1 en variëer y van 0 naar 1, dan verwacht je logisch gezien dat er op de uitgang ook 1 komt. Als je dat simuleert zie je dat er ergens een dipje inzit.
Dat komt omdat de informatie van y door de AND en OR gaat, maar ook door de inverter, AND en OR. Hier heb je 1 ingang die langs 2 verschillende paden dezelfde uitgang beïnvloedt. → Verschil tussen race en hazard.
Het is dat verschil in vertraging dat voor dat dipje zorgt.
Dynamische hazard: meestal in een iets ingewikkelder netwerk (meestal meer dan 2 lagen). Hier ga je 3 ingangen constant houden en 1 veranderen. Dan kan je weer nagaan wat er gebeurt. Normaal verwacht je dat f dan gewoon x_1 volgt. De dynamische hazard komt voor waar je middenin een statische hazard hebt. Ook in tweelagennetwerken kan je een dynamische hazard hebben, te zien in het tegenvoorbeld. We noemen dat een tweelagennetwerk, maar hier maakt de inverter wel een verschil: er zijn 3 lagen logica na elkaar. Dat is de reden waarom dit ook dynamische hazards zal creëren wanneer je dat uitwerkt.

Slide 125: Nu hebben we gezien waar dat vandaan komt, we kunnen nu ook kijken hoe we dat kunnen tegengaan. Hazards verhelpen: het probleem ligt 'm in het feit dat er 1 ingang verandert en dat er 2 paden zijn waaarbij de ene van de andere moet overnemen. Bij de multiplexer kwam het signaal eerst langs boven en moest de onderkant overnemen, maar ze hadden niet dezelfde vertraging dus dat gebeurde niet gelijktijdig. → Hoe verhelpen: zorgen dat de ene tak nooit van de andere moet overnemen. Je zorgt ervoor dat er geen disjuncte gebieden in de Karnaughkaarten zijn en dan heb je dat niet voor.

Slide 126: Bij x gaat er vanboven een 1 opstaan in de derde kolom. Als je van rij 1 kolom 3 naar kolom 2 gaat, verandert y en kunnen we problemen hebben (denk ik). Als je een bijkomende term legt (blauwe poort), dan zal die blauwe poort 1 blijven want die wordt niet beïnvloed door y, die is onafhankelijk van y en zal niet veranderen. Die zal zorgen dat de uitgang toch constant blijft, hoewel p & q het niet op het juiste manier van elkaar overnemen. Zorg dus in uw Karnaughkaarten dat er geen disjuncte gebieden zijn.

Slide 127: Dynamische los je op door te zorgen dat er binnenin geen statische zijn. Op de b was er bv. een statische, door de blauwe poort bij te zetten is dat probleem opgelost en zal er geen dynamische hazard op die f optreden.
Da's de manier om ervoor te zorgen dat die hazards niet ontstaan.

Slide 128: Daar kunnen we rekening mee houden wanneer we ons combinatorisch netwerk ontwerpen.
Linksboven: opnieuw don't cares ingevoerd. Hoe implementeren we dit nu: op dezelfde manier als bij een synchrone sequentiële schakeling. Het geheugenelement is hier een D-flipflop. Met andere woorden kunnen we de excitatietabel van de D-flipflop hier gebruiken, ondanks het feit dat er geen flipflop inzit. Op die manier kan je de Karnaughkaarten maken en implementeren. De uitgang is het makkelijkste: gelijk aan y_1 (geen logica voor nodig). De volgende toestanden kan je op de traditionele manier maken. Strikt gezien is het hier weer verkeerd gedaan: kaart per kaart gedaan. In dit geval is dat niet zo slecht: bij het realiseren van de kaarten moest je ervoor zorgen dat er geen disjuncte gebieden waren. Hier zijn er geen disjuncte gebieden dus is in principe vrij van hazards. Je zou kunnen bij de linkse kaart de blauwe rechthoek niet gebruiken zodat je dat in de middelste Karnaughkaart ook kan doen, maar dan krijg je disjuncte gebieden. Nu hebben we de asynchrone sequentiële schakeling ontworpen zonder hazards.
Hiermee zijn niet alle problemen opgelost! Nog meer problemen qua tijdsgedrag die kunnen optreden.

Slide 129: Essentiële hazards: ook te wijten aan het feit dat er 1 ingang verandert en je verwacht dat er 1 uitgang verandert maar dat gebeurt niet. Daarvoor gaan we dezelfde schakeling gebruiken. De don't cares zijn hier weg want het is geïmplementeerd. Nu gaan we extra vertraging introduceren. We gaan (blauw vierkantje) een extra vertraging introduceren met een vertragingselement (in de prakijk gaat dat te wijten zijn aan dat er in dat stuk een enorm lange lijn gebruikt wordt). We gaan hier overdrijven en die vertraging echt heel groot maken. Nu kunnen we nagaan wat er gebeurt wanneer je vertrekt vanuit rij 00 kolom 11 en je gaat van rij 01 kolom 01.
De uitgang van het vertragingselement zal pas na heel lange tijd veranderen. Op de andere plaatsen zal de verandering wel plaatsvinden. Groen = uitgang is op 1 gekomen. De AND met het vertragingselement zal ook 1 worden omdat y_0 1 wordt en door de vertraging blijft die ene draad met het vertragingselement ook 1. Op die manier zijn we voorlopig in de gegeven toestand gekomen.
Als we nu zeggen dat die vertraging voorbij is, dan zien we dat de uitgang van de twee ORs niet meer verandert. De onderste blijft op 1 staan omdat dat zichzelf in leven houdt. Doordat de snelheid van de terugkoppeling te groot was ten opzichte van de signalen die terugkomen hebben we iets in de terugkoppellus gezet dat zichzelf in leven houdt. Dit komt neer op dat we niet van 00 naar 01 gaan maar dat we in 10 terechtkomen (andere stabiele toestand). We komen dus in de verkeerde stabiele toestand terecht en de schakeling werkt niet meer correct. Die essentiële hazard is dus te wijten aan het feit dat er (heel wat) extra vertraging werd toegevoegd.

Slide 130: Kan je dat verhelpen? Veel moeilijker. Je kan het detecteren uit de toestandstabel dat er een kans is op een essentiële hazard: als je in rij 00 kolom 11 ben en je maakt een overgang naar rij 01 kolom 01 en je volgt de rode pijlen, dan komen we in de toestand waar we in de vorige slide terecht zijn gekomen. Er is inwendig op sommige plaatsen I veranderd en op sommige plaatsen veel later nog eens. Op die manier kan je dat detecteren, maar het is een grote beperking: stabiele toestanden verminderen is dus meestal niet mogelijk.
Je kan dit niet bij het ontwerp verhelpen. Het enige wat je hier kan doen is met de vertragingen gaan spelen: extra vertraging hier en daar toevoegen: capaciteit bijzetten waardoor die poort trager gaat werken. Op die manier kan je daar wat mee spelen. Maar dat bij het ontwerp al verhinderen kan je hier niet echt. Je kan achteraf de schakeling heel grondig uittesten en zien of het werkt of niet. Het is niet omdat het zich *kan* voordoen dat dat ook zal gebeuren.

Slide 131: Je kan dus niet alles tijdens het ontwerp oplossen. Je moet dat heel nauwkeurig testen om er min of meer van overtuigd te zijn dat die werkt. Dat is een van de belangrijke redenen om niet met asynchrone schakelingen te werken. Als je dat dus niet ECHT nodig hebt, gebruik dat dan niet. En als je het al gebruikt, maak er dan geen ingewikkelde schakeling van, gewoon een kleine zoals bv. een flipflop.
Doordat het zo weinig gebruikt wordt heb je ook veel minder hulpmiddelen.
Wanneer wel doen: als het echt niet anders kan, voor een kleine schakeling tussen dingen waartussen geen synchronisatie mogelijk is (die totaal onafhankelijk van elkaar werken) of wanneer snelheid heel erg belangrijk is (want asynchrone werkt in principe sneller dan synchrone want die werkt met klok). In de praktijk is zelfs dit niet altijd een reden om naar asynchrone te gaan.

11.2 Slides: 6_FSMD

Slide 1: We gaan overgaan naar een hoger niveau waarbij we voortbouwen op wat we nu kennen maar dat op een iets abstractere manier bekeken gaat worden. We gaan de bouwblokken gebruiken om ingewikkeldere schakelingen te maken. Dan spreken we over het ontwerp van processoren. Eerst kijken we naar niet-programmeerbare (om 1 bepaald probleem optimaal op te lossen, waar je meestal hardware voor nodig hebt). Later gaan we kijken hoe dat veralgemeend kan worden naar een programmeerbare processor.

Slide 2: Hoe NP-processor omschrijven: FSMD. Er komt een datapad bij, specifiek. Je kan het blokschema weergeven zoals getoond. Waarom is dat voldoende om de moeilijkere problemen te gaan oplossen? We splitsen het probleem op in 2 delen: we weten dat we bewerkingen moeten doen, daarvoor gebruiken we het datapad. Voor de rest hebben we nog een tweede deel nodig dat zegt hoe het algoritme verloopt, welke bewerkingen moeten gebeuren en op welk ogenblik. Dat is wat de controller doet, die zegt wat de stappen zijn. Op die manier kunnen we het probleem makkelijk opgelost krijgen want die controller is een gewone sequentiële schakeling die we al kunnen maken: geeft aan wat de stappen zijn en in welke volgorde we die kunnen doorlopen.
Het enige waar we het nog niet over gehad hebben is het datapad. Dit blokschema

is evengoed geldig voor een programmeerbare processor. Het is niet-programmeerbaar als die controller vastligt.

Slide 3: Wat moeten we nog leren ontwerpen: wat er in het datapad zit. Op zichzelf, conceptueel, is dat datapad niet zo ingewikkeld. Je wilt er bewerkingen doen op gegevens. Die moeten ergens in het geheugen bewaard worden, daar bewerkingen op doen en dat resultaat terug wegschrijven in een geheugen. In het datapad zitten 3 dingen:

- Functionele eenheden: om bewerkingen/berekeningen te doen. Al die bouwblokken die we bekeken hebben bij de combinatorische schakelingen.
- Tijdelijk geheugen: resultaten ergens bewaren voor een volgende bewerking (bv. tussenresultaten).
- Verbindingen: moet allemaal verbonden worden met elkaar op een correcte manier. Het zijn die verbindingen die zeggen wat er tegelijkertijd kan gebeuren en welke gegevens die bewerkingen kunnen gebruiken. Er zijn verbindingen van het geheugen naar de eenheden en verbindingen die ervoor zorgen dat het resultaat terug in het geheugen kan geschreven worden. → Basisonderdelen van het datapad.

In het begin spraken we over verschillende niveau's van implementatie en een van die niveau's noemen we het transferlevel (RTL-niveau). Waar komt die naam vandaan (transferniveau): we hebben informatie die in het geheugen/register zit die vanuit dat register via de functionele eenheden terug in een geheugen/register geplaatst wordt. Vandaar die naam: van register naar register transfereren.

Slide 4: Hoe die opbouw/dat ontwerp doen. Die synthese, daarvoor hebben we nog een soort extra bouwblok nodig: geheugen. We hebben registers gezien, maar soms hebben we een groter geheugen nodig.
Voor we tot de implementatie van een algoritme kunnen komen moeten we een overstap maken van de algemene beschrijving van de funcitonaliteit naar een beschrijving die het makkelijk maakt om de implementatie te doen. Voor eenvoudige schakelingen is het makkelijk om dat via het toestandsdiagram te doen. Een keer we naar complexere algoritmen gaan is dat niet meer zo voor de hand liggend. Hoe komen we dan tot zoiets eenduidig?

Slide 5: Ons probleem waarvan we vertrekken: 1 ingang en op die ingang komen waarden. Op de uitgang willen we het maximum van 2 opeenvolgende waarden die op die ingang gekomen zijn. Gewoonlijk vertrek je van een probleem zoals dit, dus niet van een algoritme.
Volgende stap: vertalen naar algoritme. Voor elk probleem dat je hier hebt zijn er altijd heel wat verschillende mogelijkheden om er een algoritme voor te vinden dat dat probleem oplost. Er is nooit maar 1 algoritme dat het oplost. Het moment dat je het begint te implementeren moet je aan bepaalde dingen denken, wat hier ook in het algoritme al vermeld is: wacht op een nieuwe waarde. Die informatie is niet eens vermeld in de probleemsbeschrijving → al iets nauwkeuriger, maar nog veel te vaag om in hardware te implementeren. Hier zit nog geen enkel tijdsverband in: in hoeveel klokcycli moet dit gebeuren? Er wordt alleen uitgelegd hoe je het probleem zou kunnen oplossen.

Volgende stap: meer concreet: wat gebeurt er in elke klokcyclus? Dan zou je dat op een manier kunnen doen zoals rechts onder. Dit is geen toestandsdiagram: bij een toestandsdiagram staat er in de bollen vermeld wat de uitgangen zijn, hier staat vermeld welke bewerkingen er moeten gebeuren: y moet geladen worden met de waarde van x_i. Dat is iets totaal anders dan de uitgang van een schakeling. Wat je hier al kan zien is dat we twee delen hebben: controller en datapad. Alles wat blauw is heeft te maken met de controller. Alles wat groen is heeft te maken met het datapad (testen). Alles wat in het groen staat zijn dus bewerkingen. Let op: op dit moment hebben we al tijdsinformatie ingebracht: als het zo is dat elk van die bollen met een klokperiode overeenkomt, hebben we vastgelegd wat binnen 1 klokperiode gebeurt. Wat hierin staat is een combinatie van die 2 laatste lijnen.
We zijn dus gegaan naar een beschrijving gegaan die goed genoeg is om hardware mee te omschrijven.
Extra signalen toegevoegd want hoe weet je of er nieuwe informatie is? Iets moet zeggen dat we eraan beginnen.
Veronderstelling: startsignaal is niet nodig om de tweede x_i te lezen. We gaan er hier vanuit dat het startsignaal nodig is om te weten wanneer de eerste komt, de derde en de vijfde. Om te weten wannder de tweede komt, die moet maar in de volgende klokcyclus komen (beslis je als ontwerper).
Nu hebben we een beschrijving en zijn we van een algemeen probleem gegaan naar een beschrijving om hardware te ontwerpen.

Slide 6: Eerst overgaan naar een variante: toestands-actietabel. Een toestandstabel geeft voor alle ingangen altijd de volgende teostand aan en wat er op de uitgang komt. Als je heel wat ingangen hebt, dan leidt dat tot een onoverzichtelijke toestandstabel. Daarvoor is een toestandsactietabel veel interessanter, daarin noteer je enkel het strikt noodzakelijke: alleen als er iets gebeurt.
Blauw: controllerpad, groen: datapad (zegt niet wat uitgang is op dat moment, wel wat er moet gebeuren). Je kan niet altijd zeggen wat er ergens in die tabel, wat de inhoud van het register is, dat hangt af van de voorgeschiedenis, wanneer er iets ingeschreven is. Daarom ga je de acties weergeven.
Komt overeen met de toestandstabel zoals we die kennen, maar als de ingang niet gebruikt wordt, gaan we dat niet beschrijven, doet er hier niet toe. Zo kan je het geheel overzichtelijker houden.
Extra uitgang: klaar: geeft aan dat we klaar zijn om aan de volgende stap te beginnen.

Slide 7: Kijken in welke hardware dat resulteert: scheiding: links in het blauw de controller, rechts in het groen datapad. We gaan dingen moeten onthouden, dat steken we in registers (zo hebben we er 2, die komen overeen met de variabelen die we hebben: y en max. x is geen variabele, is gewoon een ingang). We gaan dan vergelijken en dat aan de controller geven die gaat beslissen wat er moet gebeuren. Het is altijd de controller die zegt wat er moet gebeuren. Het datapad zelf heeft 0 aan intelligentie, dat zit allemaal in de controller.
Je ziet dat ook aan de sturing: de controller zegt welke waarden er moeten ingeladen worden in het register. Het datapad doet dus de berekening, ook de testen en de controller doet dan de aansturing van het datapad. $\rightarrow$ Implementatie die

erbij hoort.
Klaar is een uitgang van de controller en daar moeten we in elke toestand aangeven wat de uitgang is. Als we dat zo aangeven wil dat zeggen dat die klaar is, gelijk aan 1, anders zal dat wel gelijk zijn aan 0, normaal geven we dat ook aan.
Dingen aansturen gaat tot effect hebben dat er iets ingeladen wordt op de volgende klokflank en vanaf dan gaat het bewaard blijven. Als die 0 gemaakt wordt blijft die data niet in het register (denk ik).

Slide 9: Wij gaan gebruik maken van een ASM-beschrijving. Dit is een visuele voorstelling van de toestands-actietabel.
Elke rij komt overeen met een blok en elke blok kan overeenkomen met verschillende elementen. Elk blok komt overeen met een klokcyclus.
Zo'n blok bestaat uit drie mogelijke kaders: toestands-, beslissing- en conditioneel kader.

Slide 10: Ter illustratie: TA-tabel en ASM-schema dat erbij hoort, makkelijker om te tonen wat er gebeurt in elke stap. Elke blauwe kader komt overeen met een toestand (ze staan erbij).

Slide 11: De hardware die erbij hoort is juist hetzelfde.

Slide 12: Meer in detail die elementen. Op examen: VHDL beschrijving. Het eerste wat je moet doen is overgaan naar een ASM schema, dat vertalen we dan naar hardware want eens je ASM hebt, is het nogal voor de hand liggend hoe je tot hardware komt. Het is dus belangrijk om uw ASM goed te kunnen opstellen want als je hier al fouten maakt, zal uw hardwareopstelling ook fout zijn.
Hier zien we de drie kaders gebruikt. Het blauwe hoort dus allemaal bij 1 toestand. Hoe weten we dat: in het begin van een toestand zit een toestandskader (rechthoek). Die loopt tot dat we een volgend toestandskader tegenkomen. Zo kan je afbakenen wat er in 1 toestand gebeurt. Belangrijk te onthouden is dat je verkeerdelijk de indruk kan hebben dat je dat nogal rechtstreeks kan vertalen (als je die indruk hebt zit je waarschijnlijk op het foute pad). De fout die je het meest tegenkomt: done wordt 0: of je nu $<=$ of $<-$ "is hetzelfde" $\Leftarrow$ FOUT. In ASM kunnen alleen pijlen staan of een gelijkheidsteken. Het is heel belangrijk om geen VHDL te gebruiken in ASM. Dit toont dat je niet begrijpt hoe het in elkaar zit.
Als we naar die kaders kijken, in het begin is er altijd 1 toestandskader. Er is er altijd 1 per blok die aan de ingang zit. Daarin zitten alle onvoorwaardelijke toekenningen: die onafhankelijk zijn van ingangen. Wat doe je met de rest: bij start kan je op ingangen testen, wat gewoonlijk gebruikt wordt door de controller die dat gebruikt om dingen in het datapad aan te sturen of wat gebruikt kan worden om conditioneel zaken uit te voeren (rondere kader), die gebeurt alleen als start gelijk is aan 1, dan pas gebeurt die toekenning. Dat kan alleen naar beslissingskaders gaan.
Het is niet zo dat je in elk blauw blokje maar 1 test kan hebben en maar 1 conditioneel pad. Van die bovenste kan je altijd maar 1 hebben, die anderen kunnen wel meer voorkomen. Je moet altijd in gedachten houden dat wat in een blauw kader zit allemaal binnen 1 klokperiode gebeurt: gebeurt tegelijkertijd/in

parallel. Het is dus niet zoals bij een normaal stroomschema dat je eerst het bovenste doet, dan de test en dan de uitvoering. Wat er in het echt gebeurt is dat er in die toestand 2 registers een nieuwe waarde kunnen krijgen: done krijgt altijd een nieuwe waarde nl. 0. Het register data gaat eventueel een nieuwe waarde krijgen. Wat er in die rondere kader staat is wat er moet ingeladen worden in dat register. Op het moment dat je in die blok binnenkomt laat uw controller weten aan het datapad of start 1 is, dat moet dus niet dan nog getest worden.
Waarom die f(x) erbij gezet: resultaat van wat er in data moet ingeladen worden, dat kan een heel ingewikkelde berekening zijn die je strikt gezien helemaal moet uitschrijven. Dat is niet zo evident als je een heel ingewikkelde formule hebt. Je lost dat best op door een functie te gebruiken: zeggen dat je nog apart een stukje hardware hebt staan dat die functie doet, je weet dat je dat hebt maar werkt dat hier niet uit. Op dezelfde manier kan je ingewikkelde berekeningen schrijven zoals bij een functie. Daarmee kan je een ASM-schema overzichtelijk houden.

Slide 13: Dingen om op te letten: testen die zorgen dat je geen foute ASM schema's tekent, er zijn een aantal dingen die niet kunnen. Een ASM schema lijkt op een toestandsdiagram. Eigen daaraan is dat je in een bepaalde toestand zit en voor elke ingangscombinatie weet je wat de volgende toestand is. Er is altijd exact 1 volgende toestand. In een ASM schema moet dat ook zo zijn. Dat betekent dat als je van een toestandskader vertrekt en je volgt het pad tot je aan een volgend toestandskader komt, dan moet je altijd in 1 enkele toestand uitkomen, dus de linkse kan niet, dat is een ASM schema dat niet geïmplementeerd kan worden. Het kan ook zijn dat je niet tot een volgende toestand komt, zoals rechts: stel dat de beide condities 0 zijn, dan blijf je eindeloos rondcirkelen, je raakt nooit in een volgende toestand. Dat mag ook niet! Moest de pijl boven de blok uitkomen was het OK geweest, eronder niet.

Slide 14: Je lijkt met een probleem te zitten links want die conditie is nodig zowel in S_0 als S_1. Je zou kunnen denken dat het een fout ASM schema is, maar dat is het niet want voor elke ingangscombinatie en voor elke toestand weet je exact wat de volgende toestand is waarin je terechtkomt. Het enige nadeel is dat je de blauwe kaders er niet kan tekenen. Als je dat wil tekenen met blauwe kaders moet dat zoals rechts: ontdubbelen. Gaat dat verschillende hardware opleveren? Nee $\rightarrow$ zal juist dezelfde hardware opleveren. Het een is dus niet slechter dan het andere.

Slide 15: Alles binnen een blauw (en dus ook geel) blokje gebeurt gelijktijdig, is parallel. Als je dat programma wil uitwerken moet je dat doen zoals getoond. De meest rechte is geen oplossing.
De volgorde binnen de kader maakt dus ook niet uit! Binnen zo'n blauw kader doet het er niet toe in welke volgorde je iets tegenkomt want het gebeurt allemaal tegelijkertijd.

Slide 16: Links: welke waarde krijgt b nu? Eerst laad je c in, daarna 1. De conventie is dat de laatste toekenning geldt, je kan dat ook schrijven zoals rechts en beide schema's zijn alletwee geldig, hoewel het linkse voor iets meer

verwarring kan zorgen.
Het enige wat niet kan is conditionele kaders na beslissingskaders zetten. Wat onder het kruis staat is hetzelfde als wat rechts onderaan staat.

Slide 17: Alles binnen zo'n blok gebeurt tegelijkertijd. Je kan hetzelfde dus op totaal verschillende manieren omschrijven. Die twee ASM schema's doen juist hetzelfde, maar hebben niet dezelfde implementatie. Als je gaat kijken wat er is gebeurd op het moment dat je eruit komt, in beide gevallen gaat dat hetzelfde zijn. Zelfs als je dezelfde functionaliteit beschrijft kan dat met andere ASM schema's.

Slide 18: In een ASM schema heb je pijltjes: acties in het datapad: er wordt iets in het register geladen en er wordt ingeladen wat er op dat moment instond. Als er = staat is dat de uitgang van een controller. Als het niet vermeld staat is het impliciet dat het gelijkgesteld wordt aan 0 in die klokperiode. Als er iets vermeld staat is dat gewoonlijk = 1.

Slide 19: Wat met combinatorische uitgangen? Je doet dat ook met een = maar één die dingen combineert uit het datapad. Als je z = x or y doet, maak je een combinatie van de twee registers en breng je dat naar uitgang z. Om dat helemaal correct te doen moet je = gebruiken, maar het is hier niet 0 of 1, het is een functie, dus je moet dat in elke toestand vermelden: in elke toestand aangeven wat die waarde is. Ondanks het feit dat dit altijd dezelfde berekening is, moet je dat altijd vermelden. Voor die z kan je dat dus niet compacter maken. De prof gebruikt een niet-officiële manier om ASM schema te tekenen: in elke toestand zou die combinatorische dingen moeten herhaald worden, dus je zet het ernaast. 1 probleem: mensen vergeten dat dan te gebruiken in een volgende stap.

Chapter 12

Les 12

12.1 Slides: 6_FSMD

Slide 20: Je kan toestandsgebaseerd werken: geen conditionele kaders en elke keer dat iets gebeurt moet je noodgedwongen een nieuwe toestand nemen. Je moet kijken hoeveel klokcycli je nodig hebt om uw algoritme uit te werken. Bv. je leest de data in en dan kijk je hoeveel enen erin staan.
In het algoritme staat niet aangeduid dat je weet wanneer nieuwe data aanwezig is, daarom is er een bijkomend signaal 'start'. Het is niet zo dat je dat zelf altijd moet invoeren natuurlijk. Zolang er geen start geweest is blijf je wachten en als er een start is ga je het uitvoeren. Je doet zoveel mogelijk tegelijkertijd en die eerste twee lijnen kan je gelijktijdig doen omdat die niet van elkaar afhangen. Die test of de data 0 is kan je niet doen voor uw data binnen is. Dan zit je in uw do-loop: testen op minst beduidend cijfer.
In elke toestand zijn er altijd volgende toestanden, minstens 1. Dat kan eventueel dezelfde toestand zijn, maar er vertrekt altijd 1 pijl uit elke toestand. Hardware blijft altijd werken. Hoe los je dat op: door te herbeginnen.
We hebben het algoritme nu vertaald naar een ASM schema, maar dat is niet de enige manier: we hebben hier bepaalde keuzes gemaakt, je kan er ook andere maken. Een alternatief is getoond via de transitie.
Achteraf gezien zou de oplossing van de transitie een betere oplossing zijn $\rightarrow$ het is dus niet vanzelfsprekend om een algoritme naar het beste ASM schema te gaan omzetten. Het is pas als je verder redeneert dat je ziet dat iets beter is. Dit is ook niet de enige mogelijkheid: we zijn er vanuit gegaan dat er pas data beschikbaar is als start = 1. Maar het kan zijn dat dat samen met dat event gebeurt. Er zijn dus allerhande mogelijkheden en het is niet altijd voor de hand liggend welke oplossing de beste is.

Slide 21: Er is een fout gemaakt in de vorige dia, bij s_4, de test of Data $\neq$ 0. Dat is letterlijk vertaald geweest en dat is verkeerd omdat als je gaat kijken binnen zo'n ASM-blok, alles gebeurt daar gelijktijdig. De toekenning aan data zal gebeuren op het moment dat je s_4 verlaat (bij volgende klokflank). Er is dus niet getest op de data die opgeschoven is maar op de oude data. Dat heeft tot gevolg dat als je nog een 1 staan had op de minst beduidende bit, je voert de loop nog eens uit. Je krijgt hier geen foute resultaten van, maar je gaat 2

klokflanken verspillen.
Het is dus belangrijk te zien dat je daar test op de oude data. Oud: wat je beschikbaar had op het moment dat de klokcyclus begon, op het moment dat je de ASM blok binnenkomt. Waar vind je die oude data terug in het schema: op de uitgang van het register. De nieuwe data is wat het moet worden op de volgende klokflank, wat je klaarzet/al berekent om in dat register opgeslagen te worden, dat is dus de informatie die je aan de ingang van het register terugvindt. Je rekent dus altijd met de oude waarde. Het vorige schema was in dit opzicht dus geen optimale oplossing.
Wat als je wil testen op de nieuwe data? Dat wat daarboven berekend wordt. Je kan die berekening nog eens doen en zo testen of dat verschillend is van 0. Het is nu ook niet zo dat je dat 2 keer moet berekenen: het is wat op dat moment aan de ingang van het register beschikbaar is, je legt een draad daar naartoe aan en je kan dat nog eens gebruiken. Je doorloopt dus niet nodeloos dat bericht nog eens.

Slide 22: Hetzelfde algoritme, alleen laat je nu ook toe dat er conditionele kaders gebruikt worden. Pas in s_2 gebeurt er iets conditioneel. Je kan ook diezelfde test in die lus doen.
Alles wat in zo'n blauwe kader gebeurt, gebeurt tegelijkertijd. Uw schema moet nog steeds werken als je het laatste deel in dat blok vanvoor zet.

Slide 23: Terug inputgebaseerd maar een ander algoritme: in plaats van do while, nu while do. Is iets efficiënter omdat je het niet nodeloos 1 keer doet. Het ASM schema is behoorlijk anders. Het is enigzins verwarrend: wat daarnet in 2 toestanden stond staat nu in 1 toestand, dat kon daarnet in principe ook. Verschil met het vorige: hier veronderstel je dat de data aan de ingang beschikbaar is samen met de start. In dit geval kom je zelfs met 2 toestanden toe om het algoritme te implementeren.

Slide 24: Stilstaan bij het geheugen. Bij synchrone sequentiële schakeling: enkel om toestand in te bewaren. Is een beperkte hoeveelheid geheugen die je nodig hebt. Hier niet: hier gaat het om een algoritme met tussenresultaten en variabelen die opgeslagen moeten worden dus gewoonlijk een groter geheugen nodig. We gaan nu kijken hoe dat in de praktijk geïmplementeerd wordt.

Slide 25: In plaats van een gewone D-flipflop gebruiken we iets dergelijks. Verschil: mux die u toelaat om uit verschillende ingangen te kiezen. Het is de bedoeling dat er geen twee write-enables gelijktijdig bezig zijn want dan heeft dat geen betekenis meer. Hier kan dat niet tot hardwareconflicten leiden (bij tristatebuffers wel), maar het is geen goed idee want zowel ingang A als B zullen onthouden worden, maar dat kan niet: er kan maar 1 bit onthouden worden. Hier heeft B voorrang op A maar dat merk je niet op het moment dat je uw signalen eraan geeft. Anderzijds heb je ook de RE die u toelaat om de uitgang op verschillende plaatsen te gebruiken. Met tristatebuffers omdat je die op bussen kan plaatsen (denk ik).
Aantal ingangen: # schrijfingangen/-poorten. #poorten: leespoorten: waar je data uitleest in het geheugen.

Slide 26: 4x3 registerbank: 4 registers van 3 bits ieder. Iedere lijn is eigenlijk een register en op die manier heb je 4 registers onder elkaar staan. Hier hebben ze elk 2 ingangen en 2 uitgangen. Verschil met gewoon 4 registers naast elkaar: als je 4 registers hebt, heeft ieder van hen 1 in- en uitgang dus 4 schrijf- en leespoorten. In het merendeel van de gevallen heb je er niet zovel nodig. In de meeste gevallen ga je 1/2/3 waarden uitlezen en misschien 1 of hooguit 2 tegelijkertijd wegschrijven. Je kan ze niet in hetzelfde register wegscrhijven natuurlijk.
Hoe bepaal je waar het terechtkomt: adresdecoders die er staan. Links voor het schrijven, rechts voor het lezen. Het specifieke van een registerbank is dat die minder in- en uitgangen heeft en dat zorgt ervoor dat we minder draden gaan moeten legggen.
Het meest kom je een dual-port registerfile tegen: registerbank met 2 poorten: 1 leespoort en 1 schrijfpoort: 1 ingang voor alle registers en 1 uitgang voor alle registers.

Slide 28: RAM: random access memory: je moet de elementen niet op volgorde lezen. Ondertussen wordt dat gebruikt voor een geheugen dat wijzigbaar is en normaal een vluchtig geheugen is: gegevens blijven niet bewaard als je de spanning afzet (in tegenstelling tot niet-vluchtig geheugen). In tegenstelling tot ROM: een keer er data instaat kan die niet gewijzigd worden.
De implementatie van RAM en ROM is vrij verschillend maar van hoe wij het bekijken is er niet zoveel verschil, conceptueel is dat hetzelfde.
Als je dat RAMgeheugen vergelijkt met registerbank: RAM gebruikt je wanneer je een groot geheugen nodig hebt. Een registerbank is moeilijk om groot te maken. Een RAM geheugen is een groot geheugen waarin je heel veel gegevens (tussenwaarden, initiële waarden, eindresultaten) kan raadplegen. Het feit dat je een groot geheugen wil heeft zijn impact op hoe het gemaakt wordt: niet met mux, flipflop, tristate. Je kan een flipflop maken met 4 a 6 transistoren, dat is wat ze hier gebruiken: de eenvoudigste versie ervan om zoveel mogeijk geheugen op de geïntegreerede schakeling te krijgen.
SRAM: gewoon geheugen/fliplop. DRAM: nog compacter gewerkt. Zelfs geen terugkoppeling van transistoren maar condensatoren als geheugenelement: cel neemt heel weinig plaats in, de grootte van 1 transistor. De informatie lekt wel weg van de condensator dus je moet dat regelmatig heropfrissen of je bent het kwijt. Tweede effect: geheugen is gewoonlijk trager: hoe groter het geheugen, hoe langer het duurt om gegevens uit te lezen/in te schrijven. Dit omdat het geheugen alsmaar complexer wordt of van een andere technologie wordt gemaakt.
Je krijgt dus een hiërarchie: registerbanken die zeer snel zijn maar waar je geen te grote geheugens mee maakt normaal. RAM: groter geheugen maar duidelijk trager. Je kan nog verder gaan: schijven die heel traag zijn maar waar je veel informatie op kwijt kan.
Dit betekent dat als je in uw processor geheugen nodig hebt en het is niet te groot, dan gebruik je normaal registers. Als het om resultaten gaat die je maar af en toe nodig hebt kan je dat evengoed in uw RAM-geheugen zetten. Dat het lezen en schrijven dan traag gaat is geen beperkende factor.
Belangrijk verschil: heeft te maken met manier waarop het gebruikt wordt. RAM: gegevens uit lezen en later gegevens in wegschrijven, niet zoals bij regis-

terbank tegelijkertijd lezen en ondertussen restulaat al wegschrijven waarbij je altijd minstens 2 poorten gebruikt.
Hier is dat niet nodig. Vandaar dat een normaal RAM geheugen ook maar 1 poort heeft: ofwel lezen ofwel schrijven, omdat je in de praktijk beiden niet tegelijkertijd doet.
Nog een verschil: een normaal RAM geheugen gebruikt geen klok. Een RAM-geheugen heeft controlesignalen: 1 signaal dat aangeeft of het gebruikt wordt en dan een signaal of het gelezen of geschreven wordt.
Het feit dat er asynchroon geschreven wordt (lezen was altijd al asynchroon, vroeger ook), als uw chip select wegvalt, op dat moment wordt de data weggeschreven.
Belangrijk om erbij stil te staan dat dit geen geklokt geheugen is → bij ontwerp als ascynchroon ook beschouwen. In tegenstelling tot wat we voor de rest veronderstellen dat het geklokt is en synchroon werkt is dat hier niet zo. Je moet dus weten dat als uw adres verandert, dat al effect heeft: of er nu een klok geweest is of niet.

Slide 29: Derde: hoog impedant: lijn wordt niet aangestuurd. Bij geheugen: als je daar niet leest wordt die uitgang in tristate gebracht zodanig dat andere componenten die bus kunnen aansturen.
Tijden: t_{AA}: adres is veranderd: hoe lang duurt het voor er zinvolle informatieop de uitgang staat. Het kan zijn dat het adres al lang aanligt maar dat je gewacht het om de component te selecteren. Op dezelfde manier heb je een tijd dat het duurt tussen dat je het component selecteert en dat er zinvolle data op de uitgang staat.
Donkergearceerde: er staat al iets op de uitgang maar komt niet overeen met wat er op de ingang is aangelegd. Hetzelfde: t_{HZ}: om naar tristate te gaan: kan 0ns duren maar kan tot 30ns duren: je moet lang genoeg wachten voor je iemand anders toestemming geeft.
Belangrijkste hier: de toegangstijd: vertraging van het adres naar de uitgang. Dat noemt men de toegangstijd/vertragingstijd van het geheugen.
Daaronder voor schrijven: vergelijkbaar, alleen moet je zorgen dat data op de ingang staat in de buurt van de actieve flank. In deze component is dat de dalende flank: op dat moment wordt de data op de ingang opgeslagen.
Ook hier is een setuptijd en holdtijd.
t_{RC} en t_{WC}: cyclustijd: geeft aan om de hoeveel tijd je een nieuwe operatie kan doen. Er is een minimumtijd die nodig is om 1 cyclus af te werken. In dit geval is dat 35ns, bij dynamische geheugens gaat die veel groter zijn (toegangstijd van 40-50ns maar cyclustijd van 200ns). → Om de hoeveel tijd kan je een nieuwe lees- of schrijfoperatie beginnen?

Slide 30: Geheugen zonder adres.

Slide 31: Je geeft niet aan waar de data moet geschreven worden, die wordt altijd op een vaste plaats geschreven. In dit geval altijd bovenaan. Als je data inschrijft komt die bovenaan. Door het feit dat je dat weet meot je geen adres meegeven: geen sturing nodig, zit intern in de component. Als er nog eens inschrijft schuift de oude data op en wordt de top vrij en kan je opnieuw data wegschrijven.
Hetzelfde bij LIFO en FIFO. Bij LIFO: laatste wat geschreven werd wordt eerst

gelezen (destructief). Bij FIFO: wat er eerst inkwam wordt eerst uitgelezen. Typische plaats waar het geheugen wordt: stapelgeheugen/stack. → Buffergeheugen waarbij leverancier en verbruiker niet aan dezelfde snelheid werken. Hoe kan je dat maken in hardware? Implementeerbaar met een schuifregister. Dat is doenbaar as die geheugens niet te groot zijn. Als je groter gaaat, gaat men meestal gebruik maken van een apart geheugen.

Slide 32: Meestal een groter geheugen (RAM). Er zijn 2 pointers: 1 dat zegt waar het volgende moet geschreven worden en een ander dat zegt waar het volgende moet gelezen worden. Initiëel zet je het schrijfadres op de onderste plaats, dus 0. Het leesadres is altijd één minder, dus leesadres wijst naar 3. Dit heeft geen zin maar er staat niks in dus je mag er eigenlijk nog niks uitlezen. Hoe detecteer je dat het leeg is: door te weten dat het schrijfadres 0 is.

Slide 33: Als je dan data begint te laden, de data schuift niet op in het geheugen maar het adres wordt altijd met 1 verhoogd en beide adressen worden gelijktijdig verhoogd. Wat je hier dus nodig hebt zijn tellers die altijd 1 verhogen. Als je er nu 1 wil uitlezen gebruik je het leesadres dat naar de oorspronkelijk eerste waarde. Na het uitlezen verminder je je tellers terug. Je ziet ook dat er een strikt verband is tussen het lees- en schrijfadres: altijd 1 lager dan het schrijfadres mod 2^n.

Slide 36: Als het geheugen vol zit moet je dat ook kunnen detecteren, je mag er dan niks meer bijsteken. Je moet dat als gebruiker weten dat je in die toestand bent. Het is niet evident want het schrijfadres is weer 0, dus 0 staat ook voor vol. In plaats van een 2-bit teller te gebruiken kan je ook een 3-bit teller gebruiken: bij 4 weet je dan dat die vol zit en bij 0 is het dan leeg. Op die manier kan je detecteren dat die vol of leeg is.

Slide 37: Uit die manier van werken volgt zowat onmiddelijk het hardwareschema. Om te detecteren of die leeg is moet je zien dat die 11 bits gelijk zijn aan 0.

Slide 38: FIFO: je begint van dezelfde configuratie. In dit geval initialiseer je lezen en schrijven wel op hetzelfde: op 0. Als er nu iets ingeschreven wordt, gaat enkel het schrijfadres omhoog. Het leesadres blijft dus naar de oudste waarde wijzen. Als je daarna leest ga je enkel het leesadres verhogen.
Ze tellen dus beiden rond en de ene loopt achter de andere aan (leesadres haalt schrijfadres in).

Slide 41: Hoe detecteren of vol/leeg: als ze aan elkaar gelijk zijn of wanneer het helemaal vol is want dan heeft het leesadres het schrijfadrs ingehaald.
In beide gevallen moet je ook weer 1 bit meer nemen. Als ze aan elkaar gelijk zijn is het leeg, als ze 2^n verschillen. Als ze dezelfde waarde hebben dan is het leeg, is de ene 1 en de andere 5, dan zit het vol.

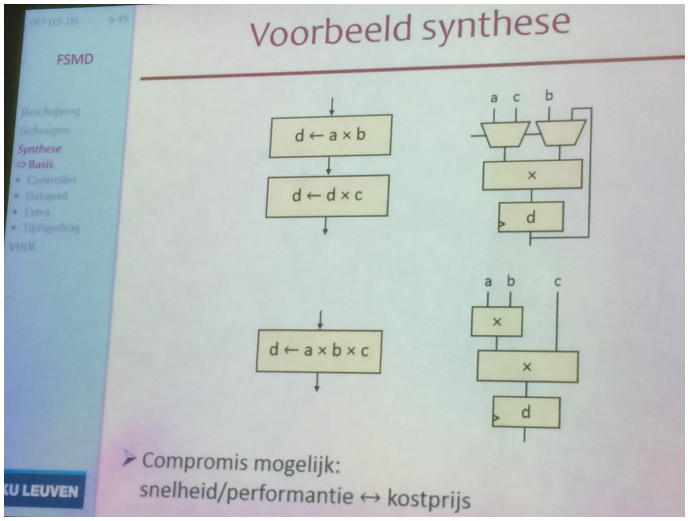
Slide 42: De ipmlementatie daarvan. De testen zijn iets anders & wordt iets anders geïnitialiseerd (denk ik), maar grosso modo blijft het hetzelfde.

Slide 43: We gaan het ASM schema gebruiken om te beslissen hoe het algoritme precies zal evolueren in de loop van de tijd. Als we nu een goed ASM schema hebben, wat is dan de volgende stap om tot de hardware de komen? → synthese van de hardware.

Slide 44: De implementatie van ASM naar hardware is op zich vrij eenvoudig. We hebben een controller (gewone sequentiële schakeling, daar weten we al alles van), we hebben ook het datapad (nog niet gemaakt). Er zitten 3 dingen in het datapad: operatoren (doen de bewerkingen), registers (geheugen voor tussenresultaten), verbindingen (alles met elkaar verbinden).
Registers: waar de tussenresultaten/variabelen opgeslagen worden. Je moet dus gewoon kijken welke variabelen je hebt en aan iedere variabele ken je een register toe.
Stap 2: operatoren: je kan zien welke bewerkingen moeten gebeuren. Voor iedere bewerking voorzie je een stukje hardware en meestal hebben we dat als bouwblok beschikbaar, dat kan je erin zetten als stuk hardware dat je reeds hebt en dat is geïmplementeerd.
Verbindingen: je moet zorgen dat aan de ingang van de operatoren daar de juiste variabelen gebracht kunnen worden: juiste ingangen van de operatoren aan de registers aanbrengen. Anderzijds heb je het resultaat van de bewerking die gestockeerd moet worden in een variabele. Je moet de uitgang dus verbinden met de ingang van eht registers. As je dat allemaal gedaan hebt heb je de hardware-implementatie gehad.
Als je dat op die manier doet werkt dat wel, maar dat is verschikkelijk inefficiënt qua hardware om dat te doen.
Stel dat je 2 of 3 keer een optelling moet doen, strikt gezien heb je dan 3 optellers qua hardware en dat gaat werken. Maar als die niet tegelijkertijd gebruikt worden is het dom om er 3 te voorzien, je kan er beter 1 voorzien en die 3 keer gebruiken: dat is veel efficiënter. We gaan ervoor proberen zorgen dat de dingen die herbruikt kunnen worden, dat we die herbruiken. Dat kan evengoed geheugenplaatsen zijn als verbindingen.
Aan de controller is in principe geen werk, het datapad kunnen we wel optimaliseren. Voor 1 probleem zijn er verschillende algoritmes om dat op te lossen en voor ieder algoritme zijn verschillende implementaties in een ASM schema. Je mag er dus niet zomaar vanuit gaan dat uw ASM schema optimaal is.

Slide 45: Lichtjes anders algoritme: om de minst beduidende bit te berekenen hebben we de data ge-AND met een constante 1 (MASK). Als je uw data daarmee AND krijg je een getal dat overeenkomt met de minst beduidende bit en dan tel je dat daarbij op → lichte variant op het voorgaande. Eerst nagaan wat er allemaal gelijktijdig mag gebeuren. Die eerste 3 dingen mogen gelijktijdig gebeuren want die zijn onafhankelijk van elkaar. Als je de volgorde mag omwisselen hangen die niet van elkaar af en mag je die gelijktijdig uitvoeren. Die Temp berekenen en die gebruiken kan niet in dezelfde periode dus daar zit een duidelijke sequentie in en moet dus in verschillende periodes gebeuren.
Als je in het vervolg iets uitkomt waar twee uitdrukkingen op dezelfde lijn staan, wil dat niet zeggen dat die samen moeten uitgevoerd worden → of dingen op een lijn staan of onder elkaar, dat maakt totaal geen verschil. Het is dus niet omdat iets op dezelfde lijn staat dat het in dezelfde klokcyclus gebeurt. Dingen

in parallel laten gebeuren, voordeel: je hebt maar 1 klokcyclus nodig om die dingen te doen. Uw ganse toepassing gaat dus trager werken. Qua snelheid is het dus interessant om zoveel mogelijk tegelijkertijd te doen. Daar staat iets tegenover: je hebt verschillende hardware nodig om iets gelijktijdig te doen: je kan met dezelfde opteller niet op hetzelfde ogenblik 2 optellingen doen, je hebt dan 2 optellers nodig. Hoe meer je dus parallel zet, hoe meer hardware je nodig hebt, maar ook hoe sneller het gaat werken.
Transitie (check Toledo: vernieuwde versie of zie Figuur **??**). Het onderste gaat veel meer kosten dan het bovenste. Je kan dus kiezen voor de kostprijs (en dan ga je voor het bovenste) of je gaat voor snelheid (en dan kies je het onderste). Dat is een keuze die je maakt op het moment dat je de onderverdeling in klokcycli maakt. Dat doe je wanneer je uw ASM schema maakt. Eens uw ASM schema gemaakt is, ligt dat dus vast. Het is dus belangrijk om erover na te denken terwijl je uw ASM schema tekent: wat ga ik parallel laten gebeuren?



Slide 46: De constante is verdwenen. Constanten voorzie je niet in hardware want als je dat rechtstreeks zou implementeren: je voorziet een register waar altijd 1 instaat. Je hebt daar geen geheugen voor nodig als dat altijd 1 is, je kan dat beter gewoon zo bedraden. Je gaat dus onmiddelijk al wat constant is letterlijk invullen (hou het in gedachten dat je het niet in een register plaatst). Dat heeft nog een ander effect, zichtbaar wanneer we de operatoren opsommen. Ook hier hebben we start. We maken een toestandsgebaseerde oplossing. Je gaat niet naar de test terug met de loop want dan heb je hetzelfde proleem: je gaat hetzelfde testen, dus daarom ga je naar het begin van die toestand. Als je ergens naartoe gaat is het het meest voor de hand liggende dat het het begin van de toestand is.
Als die test waar is dan wordt de data gestockeerd.
Alle variabelen (data, cnt, temp, out) komen overeen met registers. Het nut om ook aan Out een register te spenderen: als je uw berekening helemaal gedaan hebt, gaat het resultaat ook aan de uitgang staan. Als je aan een tweede berekening begint, gaat uw count wel veranderen, maar dat ga je niet zien aan de uitgang

want daar blijft de vorige utigang staan. Er komt pas een nieuwe waarde op de uitgang zodra je klaar bent met rekenen → geen overgangsverschijnselen.

Slide 47: Alle variabelen komen overeen met een register. Pijltjes: of er iets ingeladen wordt. Voor de rest bewerkingen (laden op zichzelf is geen bewerking): die zet je eronder. Wat er als operator staat is niet is gelijk aan (waarbij 1 ingang aan de 0 vasthangt). Bij de combinatorische schakeling (comparatoren): als er iets met constanten is resulteert dat gewoonlijk in veel eenvoudigere hardware. Testen of iets gelijk is aan 0 is gaat met 1 poort. Testen of iets gelijk is, daar hebben we 16 XOR-poorten voor nodig en een AND-poort met 16 ingangen. Het is onnozel om dat te implementeren als je het ook met 1 poort kan doen. Het derde wat je moet doen is de verbindingen leggen. Hoe doe je dat: ASM schema 1 voor 1 doorlopen, bij cnt 0 inladen (dit gebeurt door dat pijltje: resetten). z = signaal dat je naar de controller stuurt. Data ANDen met 1 en dan in Temp steken. Aan de ingang van het bovenste register ligt al iets, dus je moet een multiplexer voorzien om te kiezen welk van de twee echt naar de ingang gaat. Je beperkt het daardoor niet want het gaat nooit zo zijn dat op hetzelfde moment verschillende dingen moeten ingladen worden. Die mux bepaalt welk van de twee ingangen gekozen wordt.
De controller kan enkel weten in welke stap van het algoritme we zitten. Alle aansturingen komen dus van de controller naar het datapad.
Als je dit hebt kan je nog niet beginnen bouwen want er is nergens gespecifiëerd hoeveel bit die data is. Je moet weten met hoeveel bits je werkt. De gemakkelijkste oplossing is te zeggen dat je overal 16 bit neemt. In het algemene geval werkt dat niet want soms kunnen daar bits bijkomen: 2 16 bits optellen: 17 bits. Hier bv. zou je overal 16 bits kunnen gebruiken maar da's een enorme verspilling. Je hebt maar 5 bits nodig om het getal 16 voor te stellen, die uitgang moet dus maar 5 bits zijn, geen 16. Hetzelfde met die teller. Vanuit hardwarestandpunt is dat belangrijk: hardware voor 5 bits kost maar een derde van een van 16 bits. Het heeft ook geen nut om 15 nullen op te slagen, je kan beter minder bits gebruiken → enorme winst. Daarover redeneren gaat u al onmiddelijk een betere implementatie geven vanuit hardwarestandpunt.
Verbidningen zonder het aantal bits erbij, daar ben je niks mee.

Slide 48: Controller: moet de kader van 'naar datapad' als uitgang leveren. Waar don't cares gebruikt konden worden, zijn die ook gebruikt. Als je nog niks gedaan hebt of je wacht op uw volgende data, dan maakt het niet uit wat er in uw inwendige schakeling zit: registers mogen van waarde veranderen en vanalles mag berekend worden, het komt toch niet naar buiten. Het enige waar je voor moet zorgen is dat wat op uw uitgang staat, daar ook blijft staan. Daarom moet er een 0 staan. Een keer je verder gaat in uw schema gaan er steeds midner en minder don't cares staan. Je moet daar geen don't cares invullen, maar hoe meer er staan, hoe compacter uw synchrone sequentiële schakeling zal zijn.
Zo ga je blokje per blokje verder en je kan bij elk van de toestanden aangeven wat moet gebeuren.

Slide 49: De eerste implementatie is nu gedaan. Opmerkingen:

- Er zijn altijd 2 delen: controller en datapad. Zoals ze in de vorige slide getekend waren zijn dat beiden synchrone schakelingen die op dezelfde

klok werken. Wat gebeurt er nu: de controller geeft insturcties aan het datapad en het datapad geeft resultaten van testen terug aan de controller. Als je nu kijkt per klokcyclus, kan wat doorgegeven wordt maar gebruikt worden op de volgende klok. Alles wat je doorgeeft is dus met een klok vertraging. Het gevolg kan dus pas op tijdstip m+2 gegeven worden.

Slide 50: Kan dit sneller? Je kan een inputgebaseerde controller gebruiken (in plaats van toestandsgebaseerd) → wat op ingang staat heeft direct invloed. Daar kan je eventueel makkelijk in winnen. Kan je langs de andere kant ook winnen? In principe wel. Als je in plaats van een synchrone reset een asynchrone reset gebruikt, dan kan je dat ook gebruiken: data wordt nog in dezelfde klokcyclus ingeladen. Dit doet men minder omdat daar gevaren aan gekoppeld zijn: bij een asynchrone reset, die hebben onmiddelijk effect maar die hebben acties tot gevolg die losstaan van de klok. Het geheel gaat zich als een asynchrone(evtentueel sequentiële) schakeling gedragen. Zolang die load actief is gaat die data dus constant aangepast worden. Dat wil je helemaal niet. → gevaar van synchrone en asynchrone met elkaar te mengen.
Dat heeft tot gevolg dat die stap vanaf het datapad naar de controller, dat je daar de vertraging wel kan wegdoen, maar dat die vertraging van de controller naar het datapad er wel gaat zijn. → Altijd 1 klokcyclus vertraging. Dat hoeft niet altijd een rprobleem te zijn (in veel gevallen is dat geen probleem).

Slide 51: Als je hardware wil maken om een sequentie te tellen. Een ASM schema dat erbij zou kunnen horen is bv. het linkse. Dat is niet het correcte omdat het gaat testen op de oude data (dus niet die waarvan er al 1 afgetrokken is), gaat maar uit de lus gaan als de nieuwe data al -1 is, dat gaat 2 1 0 -1 2 1 0 -1 ... tellen. Je moet dus testen of data = 1 in plaats van data = 0.

Slide 52: Test op de data en op de nieuwe data. Je ziet ook dat het niet door twee keer data - 1 staan te hebben dat dat extra harddware gaat vragen: qua hoeveelheid hardware is dat geen verschil.
Hier wordt dat geïllustreerd: als je gaat testen dat data 0 is, dan gaat de controller dat in de volgende klokcyclus klaarzetten. Je test niet of je al op 0 bent maar of je de volgende keer op 0 komt: of je nu op 1 zit. Als je de regels van ASM volgt mag je er vrij zeker van zijn datje dt niet meer moet controlleren.

Slide 53: Als je iets gebruikt uit de controller en het datapad, moet je weten dat er altijd 1 klokcyclus vertraging op gaat zitten. Als je dingen niet uitsluitend van het datapad gebruikt, moet je in gedachten houden (als dat belangrijk is) dat die klokcycli meespelen.

Slide 54: Helemaal uitwerken: inputgebaseerd voorbeeld. Manier van werken is hetzelfde, enkel Temp heb je nu niet meer. 3 variabelen/registers. Operatoren: testen op 0, opschuiven en er 1 bij optellen. Dan nog de verbindingen doorlopen en kijken waar variabelen overal worden gebruikt. Dit is de rechtstreekse implementatie met registers en operatoren. We hebben bouwblokken gezien die registers en operatoren combineren: zie transitie. Je kan weer efficiënter gaan werken dus.

Slide 55: Controller: is eenvoudig want er zijn maar 2 toestanden. Wat je wel ziet: toestandsgebaseerd (denk ik), dus uitgang staat bij de pijlen. In principe vertrekken er 8 pijlen uit dat rechtse: iedere pijl komt met 2 pijlen overeen.

Slide 56: De realisatie van die controller geeft de totale implementatie op deze slide. Initialisatie: resetten aanbrengen: op de plaatsen waar het belangrijk is: in de controller: die moet altijd in de goede initiële toestand staan en normaal alles wat op de uitgang terechtkomt. Die andere zijn interne dingen en mogen dus even verkeerd staan, dat komt uiteindelijk wel goed.

Chapter 13

Les 13

13.1 Slides: 6_FSMD

Slide 57: Vorige les: van ASM naar hardware. Een keer je het ASM schema hebt is het niet zo ingewikkeld om tot een implementatie te komen. Een probleem was dat het niet echt optimale implementaties waren in die zin dat er heel wat hardware dubbel gebruikt werd waar dat niet nodig was. We gaan nu kijken hoe we dat kunnen optimaliseren. We gaan hiervoor kijken naar de controller want optimaliseren houdt niet alleen in dat je de implementatie zo compact en goedkoop mogelijk maakt, maar ook om zo snel mogelijk tot een implementatie te komen, zo flexibel mogelijk want ontwerptijd kost ook. We gaan nu tot een microprogrammeerbare processor komen.

Slide 58: We willen het ontwerp van die schakeling zo snel mogelijk laten gebeuren. Daarvoor gaan we patronen herkennen (staan opgesomd) die heel regelmatig voorkomen. Wanneer je een controller maakt ga je van toestand naar toestand. Dat patroon zie je altijd terugkomen: van 1 naar 2 naar 3... ook kan je subsets hebben zoals bij programma's.

Slide 59: We vertrekken van het algemene schema van een synchrone sequentiële schakeling. In 2 stukken: combinatorische logica.

Slide 60: Het eerste wat we vaststellen is dat als we een toestandsdiagram tekenen, er een natuurlijke opeenvolging van toestanden is. Af en toe zijn er testen, lussen,... maar meestal ga je van een toestand naar een volgende toestand. Daar kan je gebruik van maken als je die toestanden nummert (volgens natuurlijke volgorde), dan is naar de volgende toestand gaan er gewoon 1 bij optellen. Dan moeten we dat niet implementeren in de next state logic, maar gewoon tellen. We gaan zo eigenlijk een opteller implementeren.

Slide 61: Je komt dan tot iets dergelijks. Dat heeft tot gevolg dat het lijkt alsof het ingewikkelder geworden is. Dat kan ook zijn, maar niet per sé. De next state logic is nu veel eenvoudiger geworden. Als je bijna nooit een sprong hebt in uw programma is het veel eenvoudiger want die component staat voor de sprongen. We gaan dus standaardlogica zetten die het merendeel van de tijd

gebruikt kan worden. Het is niet gezegd dat dat compactere hardware oplevert, dat is niet gegarandeerd. Als het gaat om een makkelijker ontwerp te maken is dit zeker een voordeel. Je kan dat rode zelfs als 1 standaard component bekijken.

Slide 62: Wat je ook terugvindt is dat er dikwijls stukken terugkeren (in programma: subroutine die je verschillende keren opnieuw oproept). Dit kan ook voorvallen bij een toestandsdiagram. Je kan dat ook op die manier proberen implementeren: gemeenschappelijke stukken samennemen. Het enige dat bijkomend nodig is is een register om de terugkeertoestand in te bewaren.

Slide 63: Wanneer je geneste subroutines hebt, heb je een LIFO nodig. Elke keer als je ergens inspringt, bewaar je waar je daarna naartoe moet, anders zou je adressen moeten overschrijven. Het kan dus zijn dat je verschillende registers nodig hebt. LIFO is dan flexibeler: wat je er het laatst ingestoken hebt, heb je als eerste nodig als terugkeeradres.

Slide 65: One-hot register: de logica die erbij hoort om te bepalen wat de volgende toestand is is eenvoudiger dan bij een traditioneel register omdat je niet meer moet decoderen/beslissen in welke toestand je zat om de volgende actie te gaan bepalen. Het nadeel is dat het heel duur is omdat we voor n toestanden n flipflops nodig hebben in plaats van het binaire logaritme van n aantal flipflops.
Getoond is een traditioneel register (SReg) met daarachter een decoder. Is dit een intelligente manier om een one-hot encodering te maken? Niet per sé. het is niet automatisch zo dat dat een goed idee is qua hardware. In elk geval kan je het op die manier doen. Voordeel van dit te gebruiken in plaats van enkel het toestandsregister: logica wordt eenvoudiger. Wat moeten we nog ontwerpen: die stukken boven elkaar.

Slide 66: Als je dat allemaal samenzet kom je tot de gegeven tekening. Bij transitie slide: subroutine gedeeld.
Als dat is wat we bekomen als we het samenzetten en we bekijken het lichtgroene samen: dat is een decoder (links): daar gaat 1 van de uitgangen aangeven dat we in de zoveelste toestand zitten. Wat moet die logica (rechts) doen: bepalen wat alle stuursignalen zijn die nodig zijn. Als je dat op die manier beschouwt dan is dat groene eigenlijk een ROM-geheugen: voor een bepaalde toestand wordt aangegeven wat er op de uitgangen van dat geheugen moet staan.

Slide 67: Je krijgt dan een microprogrammacontroller. In dat ROM geheugen ga je programmeren wat er voor elke toestand moet gebeuren. Je gaat geen hardware meer ontwikkelen, gewoon in het geheugen inschrijven wat er moet gebeuren voor de volgende toestand. Dat is qua ontwerpinspanning heel makkelijk: als je dit gaat vergelijken lijkt dat enigzins op een programmeerbare processor (maar hij is het niet, werkt wel zo). In de ROM tabel staan de rechtstreekse aansturingen van de hardware ("instructies") wat er moet gebeuren. Vandaar microprogrammacontroller: gebeurt in de controller zelf.
Het mooie hier is dat er van hardware niks meer moet gebeuren. De aansturingen (CS en CO) zullen een verschillend aantal uitgangen hebben en de

betekenis van het geheugen zal verschillend zijn, maar qua componenten die je moet gebruiken is dat identiek hetzelfde voor een synchrone sequentiële schakeling. Alleen het ROM geheugen moet opgevuld worden. Als je dit hebt zou je ook voor RAM geheugen kunnen kiezen om aanpassingen te doen. Dat vind je terug bij sommige implementaties zodanig dat als men de instructies/manier van werken kan gaan aanpassen naar nieuwe eisen/fouten die ontdekt zijn/andere instructiesets. Je kan die hardware gebruiken om zich bv. de ene keer als een Intelprocessor te gedragen en een andere keer als een andere processor. WCS: writable control store.

Slide 68: Je kan echt winnen op optimaliseringen in het datapad.

Slide 69: Het is niet de bedoeling dat we deze optimalisering kunnen toepassen op het examen, het is belangrijk dat de principes begrepen worden: manier waarop het werkt en waarom bepaalde dingen op bepaalde manieren gebeuren. Eenvoudig voorbeeld: benadering doen van de wortel uit een som van kwadraten. Wordt gebruikt om het vermogen af te schatten van complexe signalen. Het geeft de norm/lengte/amplitude van een vector weer als a en b een reëel en imaginair deel zijn van een getal. De wortel trekken is niet evident dus we gaan een ander algoritme zoeken dat goed genoeg is om een benadering te geven van die wortel. Dit is goed als we een schatting willen hebben.
Wat hier gegeven staat is een voldoende benadering om de wortel uit de som van de kwadraten te gaan berekenen.
Je moet eerst eeen algoritme vinden dat zo eenvoudig mogelijk is maar toch nuttig genoeg om praktisch gebruikt te worden.
Aantonen dat het praktisch bruikbaar is: we hebben dit gekozen met in het achterhoofd dat het in hardware geïmplementeerd gaat worden (dat zie je in die .875 en .5 $\rightarrow$ delen door een macht van 2). De coëfficienten zijn zo gekozen dat ze makkelijk te berekenen zijn. Als je zo'n eerste benadering zoekt kan je wel vinden dat de beste benadering .85 is en .55, maar als je hardware gaat ontwikkelen ga je toch eerst nog nadenken voor je dat zo implementeert, je gaat dat eerst aanpassen zodat het later te implementeren valt. Zo zie je dat ontwerpen niet begint vanaf iets dat vastligt en in hardware moet omgezet worden. Het is alle stappen samen: vanaf het probleem vertrekken tot de uiteindelijke implementatie.

Slide 70: Om te kunnen optimaliseren gaan we proberen tijdens het opstellen van het ASM schema om niet onmiddelijk zoveel mogelijk te optimaliseren (gewoonlijk doe je dat wel). Dat betekent dat je hier zoveel mogelijk tussenresultaten gaat gebruiken, zoveel mogelijk verschillende variabelen, zo kunnen we beter optimaliseren. Je moet het dus eerst zo gedetailleerd mogelijk met zoveel mogelijk variabelen beschrijven zodat je het beter kan optimaliseren. Hetzelfde geldt bij het schrijven van software: als je uw compiler goed wil laten optimaliseren, maak je het zo eenvoudig mogelijk.
Zoveel mogelijk tussenresultaten en geen samengestelde bewerkingen doen in 1 klokcyclus maar spreiden over meerdere klokcycli.
Er is een startsignaal nodig, ook iets om te weten dat het resultaat beschikbaar is en we gaan beslissen dat de berekening (resultaat) gedurende 1 klokperiode beschikbaar gaat zijn en een signaal om aan te geven dat het beschikbaar is. De

ingangen worden bewaard in a en b, daar worden het min en max van berekend nadat er de absolute waarde van berekend wordt. Dan heb je x en y. Je gaat die .875 verwerken als 1-1/8 (eenvoudiger dan vermenigvuldigen). Het uiteindelijke resultaat moet gedurende 1 klokcyclus beschikbaar zijn, maar bij t7 <− max(t6,x) is in die klokcyclus nog niet beschikbaar, die *wordt* dan pas geassigned in de volgenede klokcyclus. DAV = data available, dus enkel tijdens die klokcyclus moet dat 1 worden. Het gaat hier enkel en alleen om het optimalisere van het datapad. Voor de verdere stappen gaan we een deel niet bekijken (zie transitie slide).

Slide 71: Wat kunnen we optimaliseren? In het datapad zitten variabelen, functionele eenheden (operatoren) en verbindingen. We kunnen dat optimaliseren. In plaats van 11 variabelen te gebruiken kunnen we sommige variabelen samenzetten in 1 register/2 variabelen dezelfde naam geven. Wanneer kan je 2 verschillende variabelen in 1 register zetten: als ze niet op hetzelfde moment dat register nodig hebben. → Levensduur. Een variabele begint te leven als die een waarde krijgt en die moet onthouden worden tot die de laatste keer nodig was. Wat kunnen we allemaal in 1 register samenzetten: alle variabelen met niet-overlappende levensduur. De matrix rechtsonder toont in elke component elke variabele nodig is. x moet in S_5 ook nog levend zijn want het is in S_6 nog nodig! Als je in iedere toestand kijkt zie je dat er in S_4 en S_5 telkens 3 variabelen nodig zijn en dat dat het meeste is. We hebben dus wel 11 variabelen, maar we hebben maar 3 registers gelijktijdig nodig. Dat reduceert het aantal variabelen dus al.

Slide 72: We kunnen hetzelfde doen voor de funcitonele eenheden maar dat is minder evident: een register heeft maar 1 functionaliteit: iets onthouden. Een functionele eenheid is geen functionele eenheid want ze doen allemaal iets anders (soms wel hetzelfde). We zien bv. dat het maximum 2 keer berekend wordt, dus daarvoor kunnen we dezelfde hardware gebruiken, maar soms kan het interessant zijn om ook zaken die niet identiek zijn wel samen te zetten: optelling en aftrekking bevatten ongeveer dezelfde hardware. Iets dat beiden kan kost niet veel meer dan iets wat alleen kan aftrekken, dus je kan zo gaan kiezen om dan andere hardware te gebruiken. Het principe blijft wel hetzelfde: we kunnen dingen samenzetten. Het is niet gezegd om het tot 2 operaties altijd te beperken!

Slide 73: Je kan ook verbindingen samenvoegen. Dat kan zijn voordelen hebben (gaan we nog zien), niet alleen om het aantal verbindingen te reduceren, maar ook op het aantal mux en tristate buffers die nodig zijn. → Kan zinvol zijn om die te beperken.
We hebben 11 registers en 9 operatoren. Ieder van die registers wordt gebruikt en ieder van die operatoren wordt gebruikt. Als we een uitgang maar 1 keer tellen, hebben we een 20-tal verbindingen nodig. Als je kijkt naar dat ASM schema hoeveel operanden er maximaal worden gebruikt telkens, dan zijn dat er altijd maximaal 2. Hoeveel resultaten worden er gelijktijdig berekend? Ook maar maximaal 2. Daaruit kan je afleiden dat je maar 4 verbindingen nodig hebt. Zeker op het aantal verbindingen kunnen we dus eveneens besparen.

Slide 74: Groot probleem: al die dingen zijn niet onafhankelijk van elkaar: als je een aantal registers reduceert heeft dat zijn impact op de hardware die nodig is voor de operatoren, op de complexiteit van de registers en de operatoren,... Oorspronkelijk vertrek je van het linkse ASM schema: iedere variabele heeft zijn eigen register en iedere operator heeft zijn eigen (??). Er zijn rechtstreekse verbindingen en geen bussen ofzo nodig want elke operator weet welke operanden hij heeft. Het probleem onstaat wanneer je die gaat samenzetten: als operator 1 &3 ongeveer hetzelfde zijn en 2&4 en 1&2 overlappen niet qua levensduur, 3&4 ook niet. $f_{2,4}$ moet een multiplexer hebben want heeft ingang van zowel 2 als 4 nodig. $\rightarrow$ Bijkomende hardware is dus nodig door het feit dat je het aantal registers reduceert. Rechts hebben we beslist dat die variabelen samenstaan en dat leidt dus tot de nood aan die rode mux'en. We hadden de variabelen ook op een andere manier kunnen samnenemen en dan krijg je het middelste. In dit geval leidt dat ertoe dat er geen enkele mux nodig is want alles wat f_2 & f_4 nodig hebben zit in hetzelfde register. Daardoor zie je dat (door uw variabelen op een andere manier samen te nemen) je de mux'en aan de ingang kwijt bent. Wat je dus doet van optimalisatie heeft ook impact op de andere dingen.

Slide 75: We zoeken weerom een globaal optimum die al die verschillende dingen gelijktijdig kan optimaliseren. Met ons beperkt inzicht kunnen we echter maar 1 ding gelijktijdig doen. We zitten dus met die deadlock: om te zien hoe je de registers optimaal samenzet moet je weten hoe die operatoren samengezet zullen worden, maar ook omgekeerd. De traditionele manier om dat op te lossen is iteratief gaan werken: eerst hetgeen wat het meeste impact heeft optimaliseren. Voor de andere dingen schattingen doen, wat er daarmee waarschijnlijk zal samengenomen worden. Dan optimaliseer je dat en dan ga je verder. In de tweede stap ga je de schatting weggooien en de volgende optimaliseringen doen rekening houdend met wat je al gedaan hebt. Je kan dan zien dat wat je gedaan hebt geen optimale schatting was.

Slide 76: Als we dat willen toepassen, beginnen we meestal met de registers: er zijn heel wat variabelen (meer variabelen dan bewerkingen gewoonlijk). Het samenvoegen van een register, daar kan je goed het effect van afmeten want een register is een register dus 2 registers samenzetten levert de kostprijs van 1 register op en het is in de praktijk gewoonlijk makkelijker om in te schatten wat misschien zinvol is om samen te nemen. Soms wordt er begonnen met het samenvoegen van operatoren, bij FPGA's bv. omdat een register daar gratis is. Je begint zelden met het optimaliseren van de verbindingen want dat is eerder verwaarloosbaar ten opzichte van de rest. In vergelijking met de rest is dat gewoonlijk een ordegrootte minder van kostprijs.

Slide 77: Toepassen op het voorbeeld. Als je dingen wil gaan vergelijken moet je weten wat het allemaal kost: van elk ding een maximum berekenen hoeveel het kost. De cijfers die op deze en de volgende dia staan zijn random, moet je niet vanbuiten kennen. In praktijk ga je de gegeven getallen ook nooit gebruiken want het zijn zeer ruwe benaderingen. In praktijk ga je dat implementeren en de software gaat u zeggen wat de echte kostprijs ervan is (dus geen schatting). Hoe tot die getallen gekomen: bepalen hoeveel logische cellen en transistoren nodig zijn per woord (niet gespecifiëerd hoe lang een woord is). In veel gevallen

kan je dat duidelijk aanduiden hoeveel nodig is voor een bijkomende bit. Je gaat alle dingen die voor meer dan 1 bit gebruikt worden verwaarlozen in de veronderstelling dat als je heel veel bits hebt, dat maar een fractie van de kost is.
Mux voor 2 bits: we gaan die 3 NAND-poorten beschouwen als de kostprijs per bit en dat klopt ongeveer voor de totale kostprijs maar dan neem je de inverter niet in rekening. Dat maakt het wel wat ingewikkelder dus wordt hier verwaarloosd (hou in gedachten als je het zelf uitrekent).
Een register is in dit geval een data flipflop met asynchrone clear en een enable.

Slide 78: Operatoren die nodig zijn.

Slide 79: Illustreren hoe je kan optimaliseren zodanig dat je een optimaler datapad ter beschikking hebt. Er is nog iets bijgevoegd: registers samenvoegen tot registerbank.

Slide 80: We gaan variabelen samenvoegen: met niet-overlappende levensduur samenzetten. Het feit dat we variabelen samenvoegen creëert in veel gevallen bijkomende hardware. Daarom hebben we niet alleen als doel om zo weinig mogelijk registers te hebben, maar ook om zo weinig mogelijk hardware erbij te zetten. Er bestaan verschillende optimaliseringstechnieken die als doel hebben om meer dan 1 ding gelijktijdig te optimaliseren.
Het is niet de enige manier om te optimaliseren en ook niet eigen aan hardware-optimalisatie.
We willen hier 2 dingen: minimum # mux'en. Hoe kan je daarvoor zorgen: als je 2 variabelen hebt die eerst in een verschillend register zaten en je steekt die in 1 register en die worden aan eenzelfde ingang van eenzelfde operator gebruikt, heb je geen mux nodig, dus je bespaart op hardware. Hetzelfde met het aantal ingangen van de registers: ook daar mux'en staan. Als je er daar voor kan zorgen dat het resultaat van een beperking dat zowel in de ene als de andere variabele terecht moet komen op een verschillend ogenblik, als je die in eenzelfde register steekt, win je ook aan hardware. De makkelijkste manier om te zien dat je bespaart is door het te vervangen door tristatebuffers, dan wordt dat duidelijk.
Als we daar nu aan beginnne, denk eraan dat we verschillende dingen tegelijkertijd willen veranderen. We gaan beginnen met een schatting van welke operatoren samengezet gaan worden en ook welke verbindingen (maar dit is te ingewikkeld).
We gaan nu veronderstellingen maken, als die totaal verkeerd zijn moeten we eigenlijk herbeginnen.

Slide 81: We maken gebruik van compatibiliteitsgrafen en we gaan die partitioneren om te minimaliseren. Incompatibiliteitsbogen: mogen nooit samengevoegd worden. Prioriteiten geven aan dat het interessant is om samen te voegen, we geven er een prioriteit aan. Die knopen zijn variabelen.
Prioriteit: het aantal keer dat aan de voorwaarde voldaan is kan je als prioriteitgetal gebruiken.

Slide 82: Al onze knopen/variabelen. De compatibiliteit kunnen we uit de tabel linksonder halen. We zien dat bepaalde dingen niet compatibel zijn (als ze onder elkaar staan in dezelfde kolom). Daarvoor tekenen we incompatibiliteitsbogen. Het is interessant om met zoveel mogelijk variabelen te beginnen die een zo kort mogelijke levensduur hebben. Na het tekenen van de incompatibiliteitsbogen kunnen we de prioriteiten gaan bepalen: kijken wat we kunnen samennemen. Stel dat we geen enkele veronderstelling gemaakt hadden over operatoren die samengenomen gingen worden, dan was het hier gedaan want dan hadden we geen mux'en (als je niks samenneemt heb je geen mux'en) en was er niet zo'n probleem.

Slide 83: Door het feit dat we de veronderstelling gemaakt hebben van max samen te nemen, hebben we een mux nodig. Nu kunnen we gaan kijken of we ervoor kunnen zorgen dat we x en t1 samennemen om geen mux nodig te hebben. → Prioriteit om die samen te nemen. Hetzelfde voor t2 en t6. Ook aan de uitgang kunnen we kijken of we x en t7 in hetzelfde register kunnen steken. Nu is het natuurlijk wel zo, we hebben gzegd 'eenzelfde ingang': links en rechtse ingang gecombineerd. Hetzelfde voor optelling en aftrekking waarbij we dingen zouden kunnen omwisselen. Wat we wel kunnen samennemen zijn t3 en t5 en t5 en t6.

Slide 84: Dit kunnen we gebruiken om te beslissen welke groepjes we hebben. We hebben daarnet gezegd dat we maar 3 registers nodig hebben, maar we moeten nu kijken of we er nog mee komen. We gaan de graaf partitioneren zodanig dat in ieder groepje zoveel mogelijk prioriteiten zitten. Prioriteiten tussen groepjes tellen niet, binnen een groepje wel.

Slide 85: We zien dat er een aantal dingen onderling incompatibel zijn: x, t3 en t4. Ook t5 eigenlijk, maar we nemen er maar 3 omdat we weten dat we minstens 3 registers gaan nodig hebben. We gaan ieder van die gebieden uitbreiden door te proberen zoveel mogelijk prioriteitsbogen erbij te nemen. t1 en t7 kan je bv. samennemen met x en dan heb je die prioriteiten gebruikt.

Slide 87: Dan krijg je de getoonde groepjes.
Nu doet het er niet meer toe, maar we hebben nu nog teveel groepjes en de bedoeling is van er maar 3 over te houden. Je kan al degenen samennemen die niet incompatibel zijn.

Slide 88: Gedaan wat nog moest gebeuren en nu hebben we maar 3 groepen. Het getoonde is 1 van de oplossingen, anderen zijn ook mogelijk.

Slide 89: Schema na eerste optimalisering. Voor de optimalisering zag die er hetzelfde uit, buiten dat er 11 registers waren en er was geen mux.

Slide 90: Nu kan je nagaan hoeveel we ermee gewonnen hebben. We kunnen alles uitrekenen en dan kan je zeggen hoeveel logische cellen het zijn en hoeveel transistoren (TOR = transistor). Maar je gaat die nooit alletwee berekenen: ofwel doe je implementatie in FPGA of in CMOS. Hier doen we beiden om te illustreren dat het tot een verschillende oplossing kan leiden.

Slide 91: Als je dat hebt kan je de gegeven tabel al invullen. We zien al een behoorlijke reductie, ook al bij andere dingen: vanzelf zijn er minder verbindingen nodig → niet allemaal onafhankelijk van elkaar.

Slide 93: Volgende stap: functionele eenheden. Nieuwe operator die zowel het ene als het andere kan doen (wel op verschillende ogenblikken). Dat gaat een zekere hardwarekost hebben die onafhankelijk kan zijn van de hardwarekosten links. We gaan hier op eenzelfde manier proberen optimaliseren. We willen niet alleen het aantal operatoren reduceren maar tegelijkertijd ook de hoeveelheid mux'en reduceren.

Slide 94: Zelfde techniek: knopen zijn hier de operatoren/bewerkingen. Incompatibel: bewerkingen die op hetzelfde moment moeten gebeuren.

Slide 95: Incompatibiliteitsbogen getekend. Het probleem ontstaat bij de prioriteiten: wat moet je nu doen? Je moet niet 2 aan 2 gaan kijken of je ze kan samenzetten en er misschien nog een bijzetten (werkt goed bij registers want dan gaat telkens dezelfde kostprijs weg), maar hier kan de kostprijs opeens stijgen door dingen samen te zetten. Op zich is de samenzettechniek dus niet zo geschikt hiervoor.
We gaan hier 2 aan 2 nakijken wat de kostprijs is van de oude en de nieuwe. Dat moet niet allen 2 aan 2 maar voor alle groepjes van > 1 knopen. Dat is dus een hele bezigheid om dat allemaal na te gaan want je moet elke keer een neiuwe operator ontwerpen.

Slide 96: Hoe maak je een maximumberekening in hardware: vergelijk de twee en kies daarmee (mux) of het het ene of het andere getal is dat het grootste is. De compactste manier om dat te doen is die van elkaar aftrekken en naar het teken kijken. Als de tekenbit 1 is, is a-b < 0 of a $<$ b. In het andere geval kies je a. Waarom zo: je bent niet in het resultaat van de aftrekking geïnteresseerd, enkel in de laatste bit (tekenbit). Voor elke bit moet je de som dus nooit berekenen, enkel de carry. De blauwe kader is dus het teken-blokje.

Slide 97: Wat hebben we nu gewonnen door er 2 samen te zetten? → Specifiek gaan bekijken. We zien hier niet de variabelen staan maar de registers waar het uitkomt. Je hebt nu geen mux'en nodig omdat de juiste veronderstellingen gemaakt zijn toen de registers geminimaliseerd zijn. We hebben dat dus voorzien (dat die twee aan twee samengenomen gingen worden). We winnen dus de kostprijs van 1 max-berekening.

Slide 98: Samenzetten en dan voortgaan.

Slide 99: Eerst kijken wat een abs-bewerking kost. Is ook niet zo moeilijk: kijk naar tekenbit en wissel het getal desnoods van teken.

Slide 100: Daarna moet je een nieuwe operator ontwerpen die zowel een abs kan berekenen als die twee maxima. Voor die twee maxima is er geen odnerscheid want ze komen uit R1 en moeten beiden naar R1. Je hebt wel een stuuringagng

nodig of je max wil berekenen of abs. Hoe geïmplementeerd: keuze gemaakt om dat zo te implementeren, er kunnen goedkopere zijn. Stel nu dat dit de beste implementatie is die er is. Deze opteller wordt gebruikt ofwel om R1-R2 te berekenen ofwel ga je daar -R2 mee berekenen zodanig dat je kan kiezen tussen R2 en -R2 voor de absolute waarde.

Slide 101: Alles combineren, we hebben de getoonde winst.

Slide 102: Samenemen. Er is niet zo heel veel gewonnen met dat samenzetten maar je kan maar proberen. Zo moet je nu alle combianties nagaan.

Slide 103-105: Nog dingen toegevoegd.

Slide 106: We dachten dat optelling en aftrekking goed ging samengaan en we weten hoe we dat moeten implementeren. Maar nu zien we er een mux bijstaan. Die is erbij gekomen door het feit dat de andere waarde soms in R1 steekt en soms in R3 want we konden die niet samenzetten (waren incompatibel met elkaar). We moeten die mux dus wel meerekenen in de kostprijs.

Slide 107-111: Nog uitbreidingen.

Slide 112: Met 3 samen: niet *zo* ingewikkeld, je kan dat maken met een 2 naar 1 mux maken qua hardware.

Slide 113-114: Nog toevoegingen.

Slide 115: De hardware is eigenlijk niet moeilijker geworden, dus we hebben hier winst mee.

Slide 116-120: Nog dingen toevoegen en samennemen.

Slide 121: ABS en SUB hebben dezelfde hardware nodig, waarom staat dat er: heeft evenveel hardware nodig om abs min en sub samen te nemen in plaats van enkel abs en sub.

Slide 122: Nog uitwerking.

Slide 123: Tonen waarom deze manier van werken minder geschikt is: als je alsmaar grotere gebieden neemt is het niet zo dat je blijft winnen. ABS en MIN samennemen: geen winst. ABS en SUB: geen winst. Alledrie wel. Van $>> 3$ en $>> 1$ is hij afgebleven om een goede reden: de kostprijs om op te schuiven voor een vaste hoeveelheid plaatsen kost niks, dat is een kwestie van bedrading. Dat gaan combineren met iets anders zal dus geen bijkomende winst opleveren dus je mag dat gerust apart houden.

Slide 124: Illustatie van **Slide 123**.

Slide 125: Veronderstellen dat het belangrijkste gebeurd is.

Slide 126: We gaan kijken of er een onderscheid is: eerst cijfers van de transistoren die je wint. Als je hier gaat kijken kan je ze samennemen zoals getoond en hou je 4 operatoren over en heb je 94 transistoren gewonnen.
Als je naar transistoren kijkt is dat de beste manier om ze samen te nemen.

Slide 127: Kijk je naar de FPGA is dat veel minder evident want veel kleine getallen. Het is niet echt een voordeel om dingen samen te nemen want je hebt dikwijls nog een aansturing en zo extra hardware nodig. Je zou het kunnen doen zoals op deze slide of zoals op **Slide 128**. De winst is hetzelfde dus daarop kunnen we niet voortgaan. Om dan te kijken wat het beste is moet je kijken wat de kleinere effecten zijn: heeft de ene toch meer hardware/aansturing nodig dan de andere? Een voordeel van de tweede is dat er minder verbindingen zijn, maar dat vraagt dan weer een grotere mux.

Slide 129: Gekozen: voor CMOS omdat die ook geschikt is voor FPGA. Het getoonde is het resultaat na optimalisering, ook na de functionele eenheden.

Slide 130: Hebben we daar nu ook mee gewonnen? → Allemaal gaan uitrekenen. Je moet er ook rekening mee houden dat de mux'en veranderd zijn.
Het aantal mux'en is gewijzigd, dus de kostprijs ervan ook.

Slide 131: Dat heeft tot gevolg dat als je in die tabel kijkt dat de kostprijs van de registers ook veranderd is, niet alleen die van de operatoren. → Het ene grijpt in op het andere. Het aantal registers is niet veranderd, de kostprijs ervan wel.

Chapter 14

Les 14

14.1 Slides: 6_FSMD

Slide 132: Twee van de vier optimaliseringen hebben we al besproken: aantal registers reduceren of het aantal operatoren/functionele eenheden reduceren. Nu is er nog een derde onderdeel: verbindingen.

Slide 133: Verbindingen kosten gewoonlijk minder dan transistoren, maar nemen ook plaats in dus de kostprijs is niet onbestaande. Effect van verbindingen samennemen: je bent dan genoodzaakt om tristate buffers te voorzien want je hebt er een bus van gemaakt. Dus er is extra hardware nodig, tenzij beide draden register 1 allebei aanstuurden. Op eenzelde manier: aan de ingang van de FU heb je normaal een mux nodig als je >= 2 verschillende registers wilt. Als die van dezelfde draad komen heb je die mux niet nodig. We gaan op basis van deze informatie kijken welke draden we gaan reduceren. We gaan proberen te verhinderen dat er tristate buffers bijkomen (liefst zo weinig mogelijk) en anderzijds proberen om het aantal mux'en te minderen.

Slide 134: We gebruiken dezelfde manier met de knopen. Knopen zijn hier de verbindingen: operandverbindingen en resultaatverbindingen. We gaan bij voorkeur die nemen die eenzelfde aansturing hebben.

Slide 135: Voorbeeld van benadering van worteltrekking van som van kwadraten. Ondertussen hebben we al variabelen samengezet in registers en samengezet in functionele eenheden. We gaan nu nagaan wat uit welk register komt en wat als ingang gberuikt wordt van een register. Dat is wat hier is aangeduid op die slide.

Slide 136: Je hebt er 2: van uitgang van register naar functionele operatoren en van FU naar mux'en. Transitie: vroeger: mux die kiest tussen R1 en R3 en wat FU2 genoemd wordt is al de rest.
Waarom expliciet bijgetekend: bij de kostprijs van de FU's hebben we het mux'en er altijd bijgezet. Door verbindingen samen te zetten kunnen er mux'en verdwijnen.

Slide 137: 8 operandverbindingen weer meegegeven. Nog 1 extra: degene die naar de uitgang gaat is ook een verbinding en moeten we ook in rekening brengen → komt er ook nog bij. Gedurende de klokcyclus wordt berekend wat er de volgende keer moet ingeladen worden, dat staat in de kaders.
Als we gaan kijken naar de uitgangen van de registers naar de ingangen van de FU's moeten we kijken: R1 hangt aan de eerste ingang van FU1, FU2 is verbonden met de tweede ingang,... Het moet duidelijk zichtbaar zijn dat het om dezelfde ingang van dezelfde FU gaat, daarom is dat onderscheid zo ook gemaakt.
Zo kunnen we zien op welke klokperiode welke verbindingen in gebruik zijn. Wat kan niet samengenomen worden: die in eenzelfde toestand in verschillende registers staan. Bij S2 kan je B en D samennemen want ze komen samen in R1 uit. Zo kan je telkens nagaan of je dingen kan samennemen. Dat levert dus heel wat onverenigbare verbindingen op.

Slide 138: Stippellijnen zijn incompatibiliteit. Prioriteiten: zelfde aansturing of zelfde gebruik. Nu gaan we dus kijken welke dezelfde aansturing hebben (pijl $->$ = 'komt uit'). Als 2 dingen uit hetzelfde register komen trek je een groene lijn: van dezelfde bron afkomstig. Dezelfde gebruiker: als je kijkt naar de rechterkant van de pijl. Je ziet dat dat alleen bij D en E zo is, dus daartussen komt dan ook nog eens een prioriteitsboog.

Slide 139: Partitionering: met twee zouden we moeten toekomen. We vertrekken van 2 die incompatibel zijn (je kan dat niet goed voorspellen, gewoon proberen welke goed gaan zijn). Je gaat dan die gebieden uitbreiden waarbij je probeert ervoor te zorgen dat er nooit incompatibelen in eenzelfde gebied zitten. Dat is niet zozeer omwille van de prioriteit maar door de incompatibiliteit. Tegelijkertijd heeft dat er ook voor gezorgd dat er al prioriteitsbogen gebruikt werden. Voor de rest is er niks van incompatibiliteit dat ons dwingt van dingen niet samen te zetten dus dan probeer je zoveel mogelijk prioriteiten samen te zetten. Een keuze hierbij kan wel andere zaken vastleggen door de incompatibiliteitsbogen.

Slide 140: We krijgen dus twee bussen en we krijgen dan het getoonde. Er zijn tristate buffers bijgekomen en als je die FU's vergelijkt is er een mux weggegaan.

Slide 141: Weer 8 verbindingen in dit geval maar nu gaat het van de uitgang van een FU naar de ingang van een register. Als je hier naar dezelfde toestand kijkt (S2 bv.): in die klokperiode gaan de FU's iets berekenen en dat klaarzetten aan de ingang van het register. Het klaarzetten aan de ingang is de draad die we nu gaan leggen. Dat FU2 naar R1 gaat wordt gelegd in die klokperiode. Voor elke pijl die je hebt moet je dus een pijl gebruikt hebben.
Onverenigbaarheid: als het weer van 2 verschillende kanten komt kan je het niet samenzetten, wel als het uit eenzelfde FU komt. Hier zijn er veel incompatibiliteiten. Dat heeft ermee te maken dat we weinig registers en weinig FU's hebben.

Slide 142: Incompatibiliteiten, hetzelfde met de prioriteiten.

Slide 143: Ook nu kiezen we er weer twee. Je kan nu ook nog niet weten of dat de minste hardware gaat opleveren.

Slide 144: Er zijn weer tristate buffers bijgekomen maar er zijn mux'en verdwenen aan de ingang van de registers.

Slide 145: De vraag is nu of je ermee gewonnen hebt. Dat is niet zo vanzelfsprekend want een tristate buffer is niet goedkoper. Er zijn dus veel niet-goedkope elementen bijgezet en een paar weggehaald.

Slide 146: Tristatebuffers kosten redelijk wat, maar in een FPGA zijn die gratis, die staan er sowieso. Daardoor kunnen we toch nog door een daling komen omdat we die complexe mux'en kwijtzijn. Maar als je kijkt bij de transistoren is dat helemaal niet het geval. Het is dus inderdaad niet altijd zo dat hoe meer je optimaliseert hoe goedkoper het wordt. Uiteraard is het aantal verbindingen wel gedaald.

Slide 147: Laatste trukje dat regelmatig toegepast wordt. Wel mee oppassen: dat geldt maar onder bepaalde omstandigheden.

Slide 148: We hadden 3 registers en de vraag is nu of, als we die in een registerbank steken, of we dan tot iets compacter komen. Elk register heeft 1 lees- en 1 schrijfpoort. Als je n registers hebt, heb je dus n lees- en n schrijfpoorten. Je gaat die nu nooit allemaal gelijktijdig gebruiken. Het kan efficiënter zijn om die in een registerbank te steken met minder lees- en schrijfpoorten. Dat heeft tot gevolg dat je aan de ingang van de registers minder verbindingen te leggen hebt, ook aan de uitgang. In de praktijk wil dat zeggen dat je verbindingen samenlegt en dat je misschien geen tristate buffers meer nodig hebt aan de uitgang. Die zijn niet op magische wijze verdewenen, die zitten in de registerbank. Je hebt die nodig om die registers goed te laten werken.
We gaan heel eenvoudig rekenen: wat gemeenschappelijk is voor meer dan 1 bit rekenen we niet aan, maar in die registerbank zit heel wat dat gemeenschappelijk is voor heel wat bits tesamen.
Het is dus wel een verbloeming van de waarheid. Eens we dat hebben (registerbank) kunnen we ook kijken wat er gebeurt. Als je weinig registers hebt kan je rustig nagaan wat er gebeurt, anders moet je optimaliseringstechnieken toepassen.

Slide 149: Registerstoegangstabel: per register of er in die klokperiode in geschreven wordt of uit gelezen wordt. Dus is een samenvoeging van de voorgaande tabellen. Uit de grote tabel kan je makkelijk zien waaruit gelezen wordt in welke toestand.

Slide 150: Op een vergelijkbare manier kan je kijken wanneer er waarin geschreven wordt. Dat eindresultaat is de registertoegangstabel.

Slide 151: Die kan je makkelijk gebruiken om te zien wat je kan samenzetten in de registerbank. Oorspronkelijk: aparte registers dus 6 poorten nodig voor die 3 registers. Als je er nu 2 samenzet, bv. R1 en R2, dan zien we dat er op hetzelfde moment in R1 en R2 gelezen en geschreven wordt, dus we hebben 2 leespoorten nodig en 2 schrijfpoorten dus in totaal heb je voor die registerbank ook 4 poorten nodig en in totaal weer 6 poorten.
Als je anderen combineert kan je wel meer geluk hebben. Als je ze alledrie samenzet kom je met 4 poorten toe want er wordt nooit tegelijkertijd in alle drie de registers gelezen en geschreven.

Slide 152: We zien dat we een mux zijn kwijtgeraakt aan de ingang → verdere reductie.

Slide 153: Als we dat gaan bekijken: je ziet duidelijk het verschil tussen wat het beste is voor transistoren en voor FPGA. Bij FPGA kan je zoveel mogelijk optimaliseren en kom je tot een minimumresultaat. Als het over CMOS gaat, telt dat niet, dan doe je die laatste twee optimalisaties niet want dat geeft de beste oplossing. Maar eigenlijk weet je dat niet goed want daar heb je wel meer verbindingen nodig. Je zou die twee nog moeten kunnen combineren om 1 getal uit te komen om te vergelijken, maar dat is niet evident want je weet op zich niet hoeveel transistoren een verbinding waard is. Je kan dat pas nagaan wanneer je een volledige implementatie gedaan hebt.
Dan ga je rekening houden met waar wij geen rekening gehouden hebben. Dan ga je wel zien wat dan het beste is. Het toont aan dat afhankelijk van waar je het wil implementeren, welke technologie je wil gebruiken, dat je tot andere optimale oplossingen komt.

Slide 154: Andere optimaliseringen. Tot nu toe hebben we alleen de controller en het datapad bekeken. Het enige wat we in het datapad gedaan hebben is ervoor zorgen dat de hardware die we hebben zoveel mogelijk samengenomen werd. We hebben niks veranderd aan het tijdsgedrag. Nog 4 dingen vermeld die die twee aspecten behandelt: we hebben controller en datapad en we mogen die niet onafhankelijk van elkaar optimaliseren, daar is een verbinding, die zijn gekoppeld aan mekaar. Anderzijds heb je ook het aspect dat het misschien beter is om heel het ASM schema helemaal opnieuw te bekijken. Het is moeilijk om daar algemene richtlijnen over te geven.

Slide 155: Het eerste gaat over een minimaal instructiewoord: de instructie die vanuit de controller komt en naar het datapad gaat: verbindingen tussen controller en datapad. Informatie die van het datapad terugkomt is meestal redelijk eenduidig en beperkt dus kan je meestal niet veel op ingrijpen.
We gaan als voorbeeld nemen dat dit een datapad is dat we na al de vorige optimaliseringen zijn uitgekomen. We zijn tot het besluit gekomen dat dit ons datapad zou kunnen zijn. We hebben 8 registers samengenomen en 1 apart gehouden en dan nog 1 specifiek register: gecombineerd met teller: gedefiniëerde taak en zo ook gebruiken. Operatoren, waarvan ook een complexe (kunnen verschillende bewerkingen in gebeuren) en operatoren. Verbindingen samengevoegd. We hebben 2 uitgangen en maar 1 verbinding dus ook dat samengezet. Ook andere keuzes gemaakt: uitgang is aan de resultaatbus gekoppeld,... → Beperkingen

die we ons opgelegd hebben om tot zo compact mogelijke hardware te komen. Wat hier niet gelijktijdig kan gebeuren is iets opschuiven en een berekening doen want dat komt op 1 bus, dus je kan het resultaat van 1 maar doen.
Stel nu dat dit dus ons datapad is waartoe we gekomen zijn. Het instructiewoord zijn alle streepjes die aan de linkerkant staan van de blokjes. Al die dingen die vanuit de controller komen zijn opgesomd op de volgende slide.

Slide 156: We zien dat we al onmiddelijk 32 bits nodig hebben om dat aan te sturen: als je naar grotere datapaden gaat kijken is het heel normaal om 100-200 aansturingen nodig te hebben vanuit de controller. De vraag is nu of we die 32 bit nodig hebben, of we niet kunnen reduceren. Eerst gaan we kijken hoe we kunnen besparen, daarna waarom je dat zou doen.

Slide 157: We kunnen erop ingrijpen omdat ons datapad bepaalde beperkingen oplegt, daardoor zijn sommige dingen niet meer mogelijk. Bv.: om bj de teller te beginnen: 3 ingangen, je kan daar maar 4 verschillende acties mee beschrijven. In principe zijn 2 bits voldoende om dat te doen, dus daar zou je al een bit op kunnen winnen.
Zo mogelijk nog evidenter: ieder van die operandbussen kan vanaf 2 kanten aangestuurd worden maar niet gelijktijdig. Als je naar de onderste lijn kijkt, daar staat ofwel de uitgang van het register op ofwel de uitgang van een registerbank. Dat betekent dat die tristatebuffers niet gelijktijdig geactiveerd worden. Ze mogen gelijktijdig niet geactiveerd worden, maar dan heb je een zwevende lijn en da's ook geen goed idee. Het lijkt dus niet onlogisch om die aansturing van die tristate buffers complementair te maken (is best zo). Als die complementair zijn heb je maar 1 aansturing vanuit de controller nodig om te zorgen dat het juiste erop komt te staan. Dat heeft tot gevolg dat er in het datapad nog een stukje hardware bijkomt: inverteren en naar de andere tristatebuffers sturen. Hier zie je dus ook dat het aantal lijnen reduceren dat de hoeveelheid hardware binnen het datapad zal vergroten. Dat geldt voor alle bussen.
Hetzelfde met het vorige: als je maar 2 bits stuurt moet je ze decoderen om 3 bits te sturen, dus ook hardware voor nodig.
Er zijn dikwijls nog andere dingen: je leest iets uit de registerbank. Het heeft op zich geen zin om iets uit die registerbank uit te lezen als je het daarna toch niet gebruikt (als de output enable in niet-doorverbonden toestand staat). Het is logsich om te zeggen van er maar uit te lezen als de tristate buffer aanstaat. Dat geldt voor ieder an die uitgangen. In dit geval zorgt dat ervoor dat er geen bijkomende hardware is. De bijkomende mogelijkheid die je hebt om iets uit te lezen en het niet te gebruiken heeft in deze configuratie toch geen zin.
Die twee operatoren (ALU en barrel shifter), het heeft geen zin om daar gelijktijdig iets mee te berekenen want maar 1 van de twee kan op de uitgang gezet worden. Maar 1 van de twee zal dus maar nuttig werken. De twee instructiebits die je nodig hebt (F, Sh of D), je kan er dus twee samennemen: met 4 bits kom je toe. Als de barrel shifter gebruikt wordt zijn dat bits 2-5, anders 7-9. Als de barrelshifter actief is, kan er onzin komen op de ALU en omgekerd. Je kan hier dus zonder problemen hier 3 bits besparen.

Slide 158: Het eerste effect is dat je minder verbindingen hebt tussen de controller en het datapad. Het is meestal zo dat de controller apart gemaakt

wordt en het datapad ook. Die hardware staat meestal niet tussen elkaar, die staat naast mekaar en dat heeft tot gevolg dat dat meestal niet de kortste verbindingen zijn tussen die twee. Hoe minder verbindingen je hebt tussen die twee hoe beter dus. Strikt gezien wordt uw controller eenvoudiger want je hebt minder uitgnagen. De implementatie ervan zal compacter zijn en je zal minder hardware nodig hebben. 59: Als je aan microprogramming doet bespaar je even geod op hardware. Je kan u bedenken dat je misschien wel minder hardware nodig hebt in uw controller maar wel meer in uw datapad. Zie daarnet: output enable en zijn geïnverteerde waarde nodig. Ofwel laat je die 2 signalen door de controller berekenen en maar 1 in het datapad ofwel 1 in de controller, dat naar het datapad en dat daar laten fixen.
Als we tot een optimale implementatie komen van controller en datapad kan dat ertoe komen dat je een verschuiver krijgt van een inverter die eerst in de controller eruit gehaald werd en dan in het datapad erbij kwam.
Stel nu dat we in dat geval zitten. Het is dus misschien niet zo dat we in het totaal minder hardware nodig hebben. Toch gaat dat in een aantal gevallen wel zo zijn: zolang je niet in meerdere lagen ontwerpt gaat het wel aanleiding geven tot meer hardware: 2 lagenlogica met 2 uitgangen ga je nooit 2 uitgangen met elkaar verbinden. Het gevaar is minstens als je zegt dat als je het met 2 lagen ontwerpt dat je niet zien dat die mekaars complement mogen zijn (ze mogen in theorie ook allebei nul zijn maar het is logisch dat ze mekaars complement zijn). Dat ga je heel dikwijls niet ontdekken als je met meerdere lagen werkt met als gevolg dat je het in de praktijk gewoon niet ziet dat die signalen gekoppeld zijn. Het feit dat er minder gekoppelde uitgangen zijn komt de foutgevoeligheid ten goede want als je 2 uitgangen van de controller hebt en stel dat de logica slecht ontworpen is in de controller of dat je bij microprogrammatie een bit verkeerd gezet hebt, kan het zijn dat de twee tristate buffers tegelijkertijd geactiveerd worden → probleem. Als je daarentegen maar 1 signaal hebt kan het aleen maar problemen geven als de inverter toevallig kapot zou zijn. In alle andere gevallen, welke bedenking/fout je ook maakt in de controller, het kan nooit tot hardwareconflicten leiden in het datapad. → Voordelen van datapad zo kort mogelijk te houden.
Wat nooit te detecteren valt als je de twee los van elkaar ontwikkelt: bij de controller, het ontwerp ervan, kan je geen rekening houden met de beperkingen van het datapad. Zoals het feit dat die instructiebits voor ALU en shift dat je die kan delen, dat kan je maar opleggen als je terdege weet hoe je het datapad in elkaar gestoken hebt en wat daar de mogelijkheden en beperkingen van zijn. Dat kan nooit uit een automatische procedure volgen, dat hangt af van de intelligentie van de ontwerper om zo'n zaken te detecteren.

Slide 159: Zorgen dat het ASM schema dat je niet heel het ontwerp weer moet doormaken.

Slide 160: Kan je ook doen op het examen zonder er veel tijd aan te spenderen. Op het moment dat je uw ASM schema opstelt kan je dat automstisch doen. Aan efficiëntie winnen: uitsmeren over zoveel mogelijk klokcycli vs. zoveel mogelijk in 1 klokcyclus te doen. We gaan proberen van ze onmiddelijk van de uitgang van de ene naar de ingang van de andere operator te geven. Dat is wat hier beschreven staat.

Je kan die - en >> 3 gemakkelijk in 1 klokcyclus doen. Je bent sneller in het rechtse dan in het linkse en je gaat dus meer bewerkingen per seconde kunnen doen → voordeel tenzij er ergens gespecifiëerd staat dat je t5 exact na 2 klokwaarden van die berekening moet hebben. Je gaat het dan moeten vertragen dus geen winst hebben.
Beperking waarop je moet letten: als je dat samenzet krijg je een ingewikkeldere operator die in zijn geheel trager gat werken. Dat kan zijn impact hebben op de snelheid. De maximale klokfrequentie wordt bepaald door het kritisch pad. Het gaat meestal langer duren: voor dit geval alleen niet maar gewoonlijk worden er nog andere dingen gedaan en die gaan evengoed aan die trage klokfrequentie gebeuren dus normaal gaat uw algoritme juist trager werken. Je hebt er dus pas voordeel aan als je daardoor uw klok niet gaat moeten aanpassen. Dat kan bv. wel voordelig zijn als uw kritisch pad niet langs dat samengestelde deel gaat (denk ik).

Slide 161: Je kan ook aan snelheid winnen door zo weinig mogelijk in 1 klokcyclus te doen.

Slide 162: Multicylcing: in meerdere klokcyli iets afwerken. Het voordeel is dat je uw klokfrequentie kan opdrijven waar je normaal bv. 90ns nodig had om een vermenigvuldiging te doen en je spreidt die uit over 3 klokcycli, dan kan je een klokfrequentie van 30ns maken. Die klok gaat er nog altijd even lang over doen om te berekenen, maar het levert wel op in alle andere bewerkingen die niet zo lang nodig hebben. Uw optelling ga je bv. dus wel in 30ns doen, alleen bij vermenigvuldiging ga je wel 90ns nodig hebben. Dit leidt nog steeds niet per sé tot vertragingen wanneer je uw berekening nog niet nodig hebt in de volgende klokperiode: plan die wat vroeger in en dan is die ook op tijd af dus uw algoritme gaat toch sneller uitgevoerd worden. Heb je het echt de volgende klokperiode nodig, kan je extra klokcycli toevoegen tussendoor. Zelfs dan kan je globaal nog tijd winnen.
Je kan goedkopere hardware gebruiken die trager is, maar kan geen kwaad want voor elke optelling heb je bv. 2 klokcycli. → In 1 klokcyclus minder doen en toch sneller resultaat.
Het gaat erom om de paar dingen die het kritisch pad bepalen op een andere manier te behandelen. Verschil met pipelining: hier worden geen tussenresultaten berekend.

Slide 163: Om pipelining te situeren: in algemenere context. Dit wordt gebruikt om het aantal resultaten van een bewerking per seconde zo hoog mogelijk te krijgen. Wat we altijd proberen te doen is de hardware zo snel mogelijk te maken. Op een bepaald moment loop je tegen grenzen aan, kan je niet meer implementeren in hardware, er is een grens aan. Het kan zijn dat die grens niet de goede is, dat het nodig is om sneller te gaan werken (bv. bij realtime toepassingen → data wordt geleverd aan een bepaalde snelheid: beeldinformatie verwerken die aan hoge snelheid binnenkomt. Het kan nodig zijn om om de 10ns een resultaat te hebben en als je ingewikkelde berekeningen nodig hebt, kan dat niet mogelijk zijn).
Je kan ze in parallel doen. Als 1 bewerking doen te traag is doe je maar n bewerkingen gelijktijdig. Door die n groot genoeg te kiezen kan je eender welke

snelheid halen. Nadeel: als je n dingen in parallel wil doen heb je ook n keer zoveel hardware nodig want voor iedere berekening heb je die hardware nodig. Je moet uw data ook nog omzetten van seriëel naar parallel. Gewoonlijk met een ganse buffer. Aan het uiteinde moet hetzelfde gebeuren want je hebt 3 resultaten gelijktijdig beschikbaar terwijl die sequentiëel na elkaar doorgestuurd worden. Dat kan je oplossen door interleaving: die processen iets verschoven ten opzichte van elkaar opstarten, dan heb je gen buffer nodig. Je moet dan enkel zorgen dat uw elementen in de juiste blokken terechtkomen. Aan de uitgang hetzelfde. Je moet gewoon zien dat je op het juiste moment inklokt. → Manier van werken die veel gebruikt wordt bij geheugens: componenten die veel trager werken dan de rest van de logica. Oplossing om niet die explosie aan hoeveelheid hardware nodig te hebben en toch de snelheid op te drijven pipelining. Wassen, drogen en strijken: als alles niet rap genoeg gaat kan je naar een wassalon en kan je verschillende wassen gelijktijdig doen en gelijktijdig drogen en dan 3 mensen nodig om te strijken → manier om het sneller te doen verlopen, maar niemand doet het op die manier want je hebt maar 1 wasmachine staan. Wat doe je: je steekt was in, als die klaar is steek je die in de droogkast en steek je de tweede was in enzovoort. → Werken aan de lopende band.
Voordeel: als je hiernaar kijkt zie je dat de wasmachine constant bezig is, er is geen enkel moment dat die niks aan het doen is. Bij parallel is dat wel het geval: wanneer je droogt/strijkt is de wasmachine niks aan het doen. Je hebt er wel 3 die aan het wassen zijn, maar die gaat wel wachten.
Je komt bij pipelining dus met veel minder hardware toe en toch kan je de throughput naar boven drijven. Dat is de reden waarom pipelining zo succesvol is. Dat kan alleen gebeuren onder bepaalde omstandigheden: als die wasmachine iets vroeger gedaan is, heb je een wasmand nodig om het tussenresultaat tijdelijk in te stockeren. Pipelining werkt maar goed als je het kan onderverdelen en als elk stukje ongeveer even lang duurt en dat resulteert in een tussenresultaat. Als dat niet het geval is ga je niet veel winnen want het is de traagste die bepaalt aan welke snelheid een nieuwe was begonnen mag worden. Het is maar interessant in die omstandigheden.

Slide 164: Als je daaraan voldaan hebt kan je de hoeveelheid verwerkte data verhogen zonder dat je meer hardawre nodig hebt. Hardware die veel tijd in beslag neemt in 2 stukjes opgedeeld. Als je de ASM schema's erbij zet zie je dat in elke klokperiode nieuwe data kan verwerkt worden. Het verschil in de praktijk is dat binnen 1 klokperiode er veel minder moet gebeuren: je kan uw klokfrequentie opdrijven. Als je in elke klokperiode nieuwe data kan leveren ga je rechts 2 keer zoveel data verkrijgen als links. De tijd om 1 resultaat te berekenen is niet veranderd echter (1 klokperiode vs. 2 (wel aan dubbele snelheid)) → = latency time. Maar daarna komende resultaten wel aan een hogere snelheid door.

Slide 165: Het is niet altijd zo makkelijk: als je het ASM schema iets moeilijker maakt: conditioneel element is het trage en ga je dus proberen pipelinen. Hier heb je niet al die voordelen: als je nu gaat kijken: links is het zo dat zelfs onafhankelijk van de waarde van a dat er elke klokperiode nieuwe data verwerkt kan worden. Als je rechts kijkt: daar kan het gebeuren dat je 2 klokperiodes nodig hebt om iets te verwerken. Zelfs als je uw klokperiode hebt kunnen ver-

dubbelen (en je hebt nu maar 1 klokperiode nodig in plaats van 2) heb je nog steeds niks gewonnen: evenveel resultaten per seconde die eruitkomen. Soms is het nog ingewikkelder: soms na 1 klokperiode, soms na 2. Met reële ASM schema's heb je frequent zulke problemen. Het komt erop neer dat je er meer werk aan hebt: je moet uw ASM schema helemaal herdenken. Het helemaal rechtse (vorige was het middelste!) doet helemaal hetzelfde maar heeft als voordeel dat er in elke klokperiode nieuwe data kan verwerkt worden dus kan je aan een dubbele klokfrequentie werken met in elke periode nieuwe data. Door beter na te denken bekom je dus een beter resultaat. Dat betekent dat je in heel veel gevallen van pipelining gebruik kan maken maar dat dat het ASM schema moeilijker maakt.
Je moet pipelining alleen toepassen op plaatsen waar het trager is. Optellers ga je bv. niet pipelinen, wel de vermenigvuldiger.

Slide 166: Je kan in het hardawre shcema ook nog op andere plaatsen pipelinen. Je hebt heel dikwijls dat er van een operator iets berekend wordt, dat gaat naar de controller en zo naar het datapad. Je kan eventueel ook in de lus van de controller pipelining doen. Maar door daar registers te zetten wordt de fasering tussen controller en datapad groter. Dat kan je niet in rekening brenen in uw ASM schema. Het roze/rode kan wel wat opleveren maar maakt het ontwikkelen van het ASM schema dus wel moeilijker.
Wel makkelijker: ASM schema zelf opsplitsen. In plaats van het te zien als een controller: in 5 controllers opsplitsen en voor elke controller een ASM schema voorzien. Hier staan dingen die na elkaar getekend zijn maar dat zijn op zich niet-samenhangende dingen: weinig/geen impact op elkaar. Het kan dan interessant zijn om voor elk van hen een ASM schema te maken. Dat is zeker interessant wanneer die toch verschillende hardware aansturen en toch verschillende hardware gebruiken. In plaats van 1 groot schema met 1 grote controller en datapad heb je dan verschillende kleine schema's die makkelijker te ontwikkelen zijn en het voordeel hebben dat ze allemaal gelijktijdig kunnen werken. → voordeel van opsplitsen ASM schema.

Slide 167: Getoond. Zoals je hier ziet: die 3 onderdelen gaan gelijktijdig werken en als dat gebeurt heb je een optimaal gebruik vn uw hardware.
Nadelen pipelining: klein beetje meer hardware nodig (maar da's nog het misnte). Groot probleem: als de lopende band zich niet als lopende band gedraagt omdat je die moet stilleggen: kan niet meer volgen. Het systeem van pipelining werkt ook maar als je aan de lopende band gegevens kan verwerken. Zodra je moet terugkoppelen (volgende instructie hangt af van huidig resultaat) kan je de lopende band niet laten verdergaan. Het stilleggen van de lopende band heeft vanuit het oogpunt van de hardware het probleem dat dat moet kunnen. Je moet tussenresultaten kunnen bewaren, opnieuw kunnen opstarten,... → Heel wat extra problemen. Als je dit hebt is het beter om geen pipelining toe te passen.

Slide 168: Je kan er heel wat uithalen maar het is geen automatisme zoals een optimale synchrone sequentiële schakeling ontwerpen. Je kan hier wel uw creativiteit laten werken. Hier zijn zoveel verschillende mogelijkheden/algoritmes/ASM

schema's/implementatataties van die ASM schema's dat uw ervaring zeker gaat meespelen.

Slide 169: Kritisch pad binnen een processor: het is een synchrone schakeling dus ook hier loopt het kritisch pad van klok naar klok. Maar het is niet binnenin een controller dat we gaan kijken wat de grootste vertraging is, maar de combinatie controller-datapad dat je moet gaan bekijken. Het is meestal niet binnenin het datapad of binnenin de controller maar binnenin het datapad en de controller en terug dat het loopt.

Slide 170: Zie transities, dan zie je wat waardoor moet. Is dat nu *het* kritisch pad voor deze tekening? Nee → maar een voorbeeld wat het is. Als er u dus ooit gevraagd wordt 'geef mij het kritisch pad' gaat het erover dat je het concept van eht kritisch pad uitlegt: het loopt op die manier of zou op die manier kunnen lopen en dat zijn de componenten die erin zitten.
Het is belangrijk te zien dat datapad en controller samen zorgen voor het kritisch pad.

Slide 171: Problemen die je pas tegenkomt als je grote schakelingen hebt, die echt reële problemen moeten oplossen en met allerhande signalen van de buitenwereld verbonden zijn. Wat zijn die problemen: de twee laatsen in dat lijstje: skew en asynchrone ingangen.

Slide 172: Klok die gebruikt wordt door alle flipflops. Die klok moet naar al die flipflops. We namen tot nu toe aan dat die klok overal tegelijkertijd verandert. Dat is niet het geval: een flipflop die heel dicht staat bij de plaats waar de klok gegenereerd wordt en je kijkt naar een kloksignaal op een plaats heel ver er vandaan dan krijg je wat rechtsboven afgebeeld staat: clock skew, verschuiving van een signaal dat zou moeten samenvallen, maar in praktijk, als je het gaat meten, valt het toch niet samen.
Waaraan te wijten: als je door een lange draad moet en je werkt aan een hoge frequentie, heeft een langere looptijd dan bij een korte draad. Er kan soms iets vertragend opstaan: transistoren die voor vertragingen zorgen. Iets dat door 5 schakelmatrices moet gaat uiteraard meer vertraging hebben dan iets wat door geen enkele moet.
Belasting: lange draad gaat een hoge capacitieve belasting hebben → duurt veel langer eer je de helft van de voedingsspanning hebt. De klok wordt daar dus ook veel later gezien.
Logica in die draad. Waarom zou je dat doen: soms ben je ertoe verplicht. Als je 1 signaal hebt dat 500 flipflops aanstuurt: kan op zich niet: er is geen enkele uitgang die zo'n hoge fan-out heeft. Hoe los je dat op: klokbuffers: op 1 plaats wordt het signaal gegenereerd en dan stuur je dat door 10 buffers onmiddelijk. De uitgang van die 10 buffers stuur je nog eens door een buffer. Gevolg: netwerk van signalen met in principe hetzelfde signaal. Maar ze hebben niet allemaal juist dezelfde vertraging. Stel dat ze eenzelfde belasting hebben en allemaal even lange draden, dan kan het nog zijn dat de verandering vanboven en vanonder niet op hetzelfde moment gebeurt (tekening vanonder).
Andere reden: clock enable.

Slide 173: Clock enable: alternatieve manier om uw flipflop aan te sturen. Tot nu toe veronderstelden we dat als er een geheugenelement is, we een mux nodig hebben (denk ik). Je kan dat ook doen zoals rechts: de klok tegenhouden zodanig dat er niks wordt ingeladen. Heeft het voordeel dat je hier maar 1 AND nodig hebt zelfs al de data over 64 bits gaat. Dat rechtse is qua hardware dus veel interessanter en verbruikt iets minder vermogen. Elke keer de klok 1 wordt gaat de set of reset geactiveerd worden dus signalen veranderen intern van niveau zelfs als er geen datasignaal is. Nadelen: je moet veel voorzichtiger zijn als je dit wil toepassen in een ontwerp: enable mag niet zomaar veranderen. (zie transitie). Als je kijkt wat er aan de uitgang van de AND komt: onbruikbaar. Die genereert 2 stijgende klokflanken en dat kan je niet toelaten. De enige manier om dat te verhinderen: beperking opleggen dat enable enkel mag veranderen als de klok laag is. Dat btekent wel dat je niet zomaar kan ontwerpen en dat wat tussen 2 klokcycli gebeurt ook belangrijk is → bijkomend probleem waar extra anadacht voor nodig is.
Het introduceert ook clock skew. Stel tekening rechtsonder: stel dat de vertraging dezelfde is als die klok van rechts, kom je in metastabiliteit. Zolang de clock skew trager is dan de vertraging waarmee je werkt is er geen probleem, maar een keer dat gelijkaardig wordt heb je wel problemen.

Slide 175: Metastabiliteit konden we geen probleem mee hebben als we ons ontwerp maar goed tekenden: ontwerp zo goed ontwerpen dat er met de setuptijd rekening gehouden werd. Je kan de buitenwereld echter niet opleggen wanneer signalen gaan voorvallen. In een toepassing zoals getoond heb je altijd gevaar op metastabiliteit. Je kan dat oplossen door er een extra flipflop voor te zetten (metastabiliteit niet vermeden, maar nu wordt dat metastabiel en niet het systeem. Is OK want behoort niet tot de normale werking van het systeem). Het hoeft ook niet altijd tot metastabiliteit te leiden: het kan zijn dat het wel in de buurt van de klok is, die verandering, maar dat het geen probleem is. He zal een verschil op de uitgang geven: duurt een periode langer, maar da's geen probleem. Als we met een bepaalde klokfrequentie werken weten we dat dat de resolutie is waaraan we werken, dat dat is hoe (snel) we werken.
2 signalen waarbij de ene het inverse is van de andere. Dan kan het zijn dat de ene overgang wel gezien is en de andere niet en dat de twee signalen niet meer elkaars inverse zijn. Als je synchroniseert mag je dus enkel onafhankelijke signalen synchroniseren. Als je een ganse bus wil synchroniseren, dan hangen die allemaal samen, kan je dat niet op een of andere manier oplossen en achteraf terug genereren. Wat wordt dan gebruikt: je gaat handshaking doen: een signaal bijmaken en ervoor zorgen dat je dat signaal alleen moet synchroniseren.

Slide 176: Enable: op het moment dat die 1 is ben je zeker dat alle datasignalen gestabiliseerd zijn, dat er geen enkele meer aan het veranderen is. Dan moet je enkel de enable synchroniseren (wordt eventueel een beetje vetraagd), maar alle andere signalen gaan stabiel zijn. Zo vermijd je gekoppelde signalen door een enable. → Iets minder veilig want je gaat er vanuit dat je weet hoe lang het signaal moet aangelegd blijven. Als de gebruiker trager is, kan het soms te laat zijn. Meestal is er dus een echte handshake: een van degene die de data doorgeeft en degene die de data verbruikt, de ontvanger. De ontvanger zal zeggen wanneer hij het geaccepteerd heeft waardoor degene die het geleverd

heeft weet dat het niet meer nodig is en het available signaal kan weggenomen worden. Daardoor weet de ontvanger dat de zender het gezien heeft en kan de ganse cyuclus afgesloten worden. Welke klokcycli er ook zijn, dit zal dan altijd tot een veilige manier van werken leiden.

Slide 177: Het kan wel nog altijd metastabiel worden: er kan iets raar (metastabiel) aan de ingang aanliggen. We zijn al best tevreden als het een kans heeft dat het bijna nooit zal gebeuren. Wat gaan we doen: zoveel mogelijk tijd geven om uit metastabiliteit te komen: metastability resolution time. Daarop inspelen: klokperiode zo groot mogelijk maken (maar performantie daalt dan, hebben we dus niet zograag, maar soms is het mogelijk van ze iets groter te maken dan we in eerste instantie willen), we kunnen ook snelle flipflops gebruiken. Voordelen: setuptijd is kleiner en de tijdsconstante om uit metastabiliteit te geraken is ook beter. Een snelle flipflop gaat veel sneller uit metastabiliteit geraken dan een andere. Daarmee speel je zowel op de setuptijd als op de kans in. je moet dat natuurlijk niet overal toepasen, enkel waar metastabiliteit kan zijn. Bij handshaking is dat niet nodig (denk ik). Nadeel: snelle flipflops verbruiken meer vermogen.

Slide 178: Laatste trukje om toe te passen: meer flipflops zetten want al die flipflops mogen in metastabiliteit geraken je hebt n flipflops om uit metastabiliteit te geraken. In praktijk blijkt dat als je er 2 of 3 zet, dat dat uw kans op metastabiliteit zo klein maakt dat je er geen rekening meer mee moet houden. Je kan er ook minder hebben en op tragere klokfrequentie werken. De bemonstering gaat dan wel aan een lagere frequentie zijn.

Chapter 15

Les 15

15.1 Slides: 6_FSMD

Slide 179: We hebben het over het ontwerp van digitale schakelingen. We bekeken hoe niet-programmeerbare processoren geïmplementeerd kunnen worden. → Wat in de praktijk toegepast wordt. We hebben gezien hoe we een ASM-schema naar hardware kunnen omzetten en naar datapad en controller en hoe dat te optimaliseren. Binnen dat hoodfstuk is er nog 1 stuk te behandelen: hoe vertaalt zich dat in VHDL? En hoe kunnen we dat omzetten naar hardware? We hebben enkele details overgeslagen omdat die nog niet nodig waren. Nu gaan we het resterende van VHDL zien. We gaan overgaan, wat we hier gaan bekijken is hoe we VHDL kunnen bekijken om te simuleren. Daar moeten we exact bij weten wat het inhoudt,... → Wat we tot hiertoe nog niet gedaan hebben, enkel logische functies tot nu toe. Als je de manier van simuleren van VHDL begrijpt is het ook veel makkelijker om te begrijpen wat er juist bedoeld wordt in die VHDL en dat is nodig om de vertaling te maken van VHDL naar hardware. Een van de dingen die we in de eenvoudigste vorm gezien hebben is het proces.

Slide 180: Een ingewikkelde beschrijving zodanig dat je een soort programma nodig hebt wordt door een proces beschreven. Het is een parallelle uitdrukking die een hardwarebeschrijving geeft. We zien een proces met gevoeligheidslijst, hier eerst zonder gevoeligheidslijst, da's algemener want daar heb je meer mogelijkheden.
Het proces wordt opgestart en dat voert die sequentiële uitdrukkingen gelijktijdig uit, komt aan het einde van het proces en begint opnieuw → eindeloos sequentiële uitdrukkingen herhalen. Dat heeft op zich geen betekenis want er is geen klok ofzo. Daarom: wait: wacht tot er iets gebeurt. Binnenin VHDL is dat de enige vorm die gebruikt wordt. We hebben ook het proces linksboven (geel) gezien, die is identiek aan het proces zonder gevoeligheidslijst. Elk proces kan je dus tot het rechtsondere omzetten, dat is de meest algemene vorm van een proces.
Alle andere parallelle uitdrukkingen die geen proces zijn kan je ook naar een proces omzetten. VHDL kent dus enkel dat proces, da's de enige parallelle uitdrukking die die kent. Voor de rest zijn dat allemaal manieren omdat het voor

de mens niet makkelijk is om het uitgebreid te zetten in een proces. Inwendig is er dus maar 1 parallelle uitdrukking en da's een prcoes.

Slide 181: Wait: laat u toe om te wachten. Je wacht op tot er terug iets gebeurt. Wat er gebeurt kan je met die voorwaarden weergeven. Het eerste wat er gebeurt is dat alle signalen hun nieuwe waarde krijgen. Van een proces met gevoeligheidslijst daarbij krijgen signalen hun waarde op het einde van het proces → is eigenlijk wanneer er een wait is, maar was impliciet zoals toen vermeld. De enige plaats om te wachten is de wait en wanneer je daar aankomt krijgen signalen hun nieuwe waarden. Dan wacht je tot er iets gebeurt.
Je kan op signalen wachten (gevoeligheid) → als een van de vermelde signalen verandert. Je kan nog een bijkomende voorwaarde geven (until) of een timeout meegeven: hoe lang je maximaal gaat wachten. Onderste is het meest algemene voorbeeld: wachten tot de klok veranderd is en reset 0 is en als dat niet optreedt (rst is nooit 0) wachten we niet langer dan 1ms (simulatietijd). Het meeste zie je het voorlaatste. Daar staat geen on bij. Op zichzelf heeft dat geen betekenis want je kan maar stoppen met wachten wanneer iets gebeurt. De interpretatie die men geeft aan die lijn is hetzelfde als 'wait on clk until clk = '1". Ze moet van iets anders komen en moet 1 worden wel. Er moet een verandering op gebeurd zijn. Je wacht dus op een stijgende klokflank.
Rising_edge: voor het geval je met std_logic werkt waar je niet alleen 0 en 1 hebt maar ook don't cares enzo.
Als je die lijn of die uitdrukking ziet komt dat dus neer op een stijgende klokflank.
Al die voorwaarden zijn optioneel. Je kan ook een wait hebben zonder voorwaarden: je komt daar aan en blijft wachten. Dit kan nuttig zijn voor eenmalige pulsen.

Slide 182: In de beschrijving van een flipflop (hier D-flipflop) kom je wait until clk = '1' tegen. Dit konden we met een proces met gevoeligheidslijst even goed beschrijven.
Klokgenerator: beschrijving in VHDL van klokgenerator. Je wacht hier niet op een signaal dat verandert, gewoon gedurende een zekere tijd. Het proces wordt een eerste keer opgestart, er is een lokale variabele die op 0 wordt gezet. Dan wordt de klok 0 op het moment dat we aan wait komen. Vervolgens inverteren we val (dan wordt clk niet 1, want we hadden gezegd dat het 0 moest worden) en dan wachten we gedurende die pulsbreedte. Dan komen we aan het einde van dat proces en beginnen we direct terug opnieuw.
Dit kon niet met een proces met gevoeligheidslijst beschreven worden want hier wachten we gedurende een bepaalde tijd.

Slide 183: Hoe gebeurt een simulatie?

Slide 184: We gaan op ieder tijdsogenblik kijken wat alle signalen zijn. Als we per fs werken gaan we elke fs checken wat de waarde is. Dat is niet efficiënt want het is niet zo dat er elke fs iets gebeurt. We gaan heel dikwijls dezelfde bewerking doen en dat gaat heel veel simulatietijd op onze pc in beslag nemen.
Hoe efficiënter: je tekent uw signalen aan de ingang en dan ga je kijken wanneer er iets verandert. Wanneer er niks verandert op de ingangen verandert er ook

niks op de uitgangen. Pas wanneer er aan de ingangen iets verandert kan er iets veranderen aan de uitgang. Dat is wat we hier ook gaan toepassen: om de seconde iets veranderen: om de simulatieseconde maar checken.
Je kan dat ook opsplitsen in stappen. Stel dat er op de ingang iets verandert. Het eerste wat er gebeurt is dat er een transactie gecreëerd wordt: gekeken wat het effect is van die verandering aan de ingang op de uitgang. Meestal gaat dat binnen zoveel ns een effect hebben. We gaan bijhouden wat de nieuwe waarde is en vanaf wanneer die optreedt. Dat gaan we doen voor alle ingangen die veranderen en op dat moment is er niks meer te doen. We gaan dan niet meer wachten tot de volgende ingang die verandert, maar we kijken ook naar alle transacties. We gaan kijken wanneer er nog iets verandert en de transacties uitvoeren die op dat moment een nieuwe waarde zouden moeten leveren. Het signaal is actief als er iets mee gebeurt, als er een aanpassing is. Die aanpassing kan evengoed zijn dat het 1 was en het 1 wordt.
Het is belangrijk te weten als er iets gebeurt op een signaal. Als het van 1 naar 1 gaat moet er niks berekend worden want er wordt geen andere waarde op de uitgang gezet. Dat is dus niet altijd een gebeurtenis: gebeurtenis wanneer er ook een verschil is met de oude waarde. Daar kunnen we ook op testen.
Waarom zijn events belangrijk: de uitgang van de ene poort is meestal de ingang van de andere poort dus verander je de ingang van een andere poort en start het hele proces opnieuw op. We kijken dus enkel naar de poorten waar een event heeft plaatsgevonden. Het resultaat van die manier werken is dat je dezelfde simulatie krijgt, alleen heb je er veel minder werk aan.

Slide 185: We gaan het stap voor stap doorlopen.

1. In de PEQ wordt bijgehouden wat berekend moet worden: elke poort waar aan de ingang iets veranderd is, die gaan we stockeren in die PEQ om bij te houden dat daar een verandering mee moet gebeuren.

2. Er kunnen verschillende poorten zijn die aan de ingang een verandering zien. We werken dat 1 voor 1 uit. Probleem: als er 2 poorten aan hun ingang iets zien veranderen gaan ze gelijktijdig hun resultaat berekenen. Meestal werken we met een sequentiële computer waar maar 1 instructie gelijktijdig wordt berekend. Het probleem hierbij is dat als we 2 poorten zouden hebben en die krijgen eenzelfde ingang: zien hun ingang gelijktijdig veranderen. Van die tweede poort is er een andere ingang die een uitgang van de eerste poort is. Strikt gezien, als je dat 1 voor 1 berekent en als er geen vertragingen zouden zijn (resultaat ogenblikkelijk aanwezig), dan ga je die twee verschillende ingangen aan die ene poort gelijktijdig zien: parallel berekend met ingangswaarden die er op dat moment zijn, maar in realiteit is dat niet zo, er is vertraging. Je kan een verschillend resultaat krijgen als je eerst de ene voor de andere berekent of omgekeerd. Hoe lossen we dat op: het resultaat berekenen in 2 stappen opsplitsen: eerst het resultaat berekenen en dan dat resultaat op de uitgang zetten. Nu kunnen we makkelijk doen alsof dat parallel gebeurt door eerst het resultaat te berekenen en nog niks op de uitgang te brengen (kunnen nog niet met nieuwe waarden rekening houden), dan pas brengen we dat op de uitgang. Als je dat zo doet heeft dat geen invloed op de uitgang. Er is in VHDL een verschil tussen een berekening maken en het signaal krijgt

een nieuwe waarde: die twee dingen die we opgesplitst hebben: elk proces kan rustig nieuwe waardeen uitrekenen en pas wanneer ze allemaal hun nieuwe waarde berekend hebben, dan gaan we die de toekenning aan de signalen laten doen. Dat is volledig correct met wat we van de hardware verwachten.

3. Als ze allemaal uitgerekend zijn gaan we ze één voor één toekennen.

4. Als er een gebeurtenis is opgetreden aan de uitgang zoek je alle ingangen die aan die uitgang hangen en voeg je die toe aan de PEQ.

5. Dat blijf je doen tot de PEQ leeg is.

⇒ Zo 1 iteratie is een deltacyclus. Je itereert hier maar dat is allemaal op hetzelfde tijdstip.

6. Als alles gebeurd is kunnen we de overgang maken (naar het volgende tijdstip) en is er delta-convergentie.

Slide 186-194: SR-latch voorbeeld. We gaan veronderstellen dat alle poorten geen vertragingstijd hebben: uitgang is ogenblikkelijk aangepast aan wat aan de ingang is aangelegd (is voor VHDL geen probleem). We nemen aan dat Q en Qn al een initiële waarde hebben. Zolang er niks verandert blijft dat constant. We gaan naar T1 kijken waarop A veranderd is. Als A veranderd is, ziet NAND1 een verandering aan zijn ingang. We zetten die daarom op de PEQ. We voeren dan een uitdrukking uit die queue uit. Het effect daarvan is dat Qn 1 moet worden. Aan de uitgang van die poort is er op dat moment nog niks veranderd. Als je dat gedaan hebt tot de queue leeg is kan je alles wat je hier onthouden hebt aanpassen en Qn 1 maken (op hetzelfde moment want geen vertraging). Daarom is er op Qn een vertraging dus nu ziet NAND2 een verandering aan zijn ingang dus zetten we die in de PEQ. Het resultaat van die berekening is dat Q 0 moet worden. Nu ziet die eerste poort weer een verandering en komt NAND1 weer op de PEQ. We voeren de stappen weer zo uit. Qn moest 1 worden, maar die was al 1. Er is dus een aanpassing van de waarde maar geen event. Dat betekent dat er nu niks meer veranderd is en dat je geconverteerd bent. We hebben dus alles wat er op dat simulatietijdstip gebeurt. We kunnen dus overgaan naar het volgende simulatietijdstip.
Op T2 veranderen 2 ingangen gelijktijdig. NAND1 en NAND2 komen dus samen op de PEQ. We gaan ze wel 1 voor 1 uitrekenen, die waarde onthouden (nog geen toekenning doen) en dan de tweede uitrekenen, waarbij we nog geen rekening houden met de verandering van Qn. Als de PEQ leeg is kunnen we alle aanpassingen die we vonden doen. We krijgen een event op Q, dus de eerste poort komt terug in de PEQ en doorlopen we de derde deltacyclus. We kunnen zo dus naar het volgende simulatietijdstip gaan.
Wat je hier krijgt is juist hetzelfde als wanneer je dat met potlood en papier zou getekend hebben.

Slide 195: We hebben nog niet bekeken hoe we kunnen aangeven dat een poort een vertraging heeft.

Slide 196: Wat je aan een signaal toekent is geen waarde/uitdrukking maar een golfvorm. Die kan heel eenvoudig zijn (constante waarde), maar die kan ook in de loop van de tijd variëren. Deze wordt toegekend aan een signaal.
Als je zo'n expression tegenkomt wordt dat uitgerekend en wordt dat binnen zoveel tijd als transactie gescheduled. Die tijd is altijd een deltatijd: ten opzichte van de huidige simulatietijd.
Voorbeelden: je zegt dat de NAND-poort na 10ns moet aangepast worden. Je kan met golfvormen ook pulsen genereren: op het signaal rst zet je een waarde '1' na 5 ns (ten opzichte van het huidige simulatietijdstip) en waarde 0 na 25ns van het huidige simulatietijdstip. Als die rst 0 was, wordt die 1 5ns na het huidige simulatietijdstip en nog eens 20ns later wordt dat terug 0.
Golfvormen kan je enkel aan signalen toekennen, niet aan VHDL variabelen.
delay_mechanism: alleen voorzien voor het eerst en als je er meerdere zet werk je normaal niet met een delaymechanisme.

Slide 197: 2 soorten vertraging. De meest voorkmende is de transport (maar is niet de default). Transportvertraging is een 'gewone' vertraging: met een bepaalde tijd uitgesteld.
Toch is dat niet het meest logische: als je dergelijk signaal hebt, als je dat zonder vervorming wil doorgeven heb je een systeem nodig met oneindige bandbreedte. Als dat niet het geval is gaan hogere frequenties daaruit verdwijnen. Er bestaan systemen met een redelijk hoge bandbreedte die heel veel frequenties doorlaten (bv. korte draad). Dat is niet meer het geval een keer er poorten/logica tussenstaan/-staat. Alles waar transistoren/logica inzit probeert tegen te werken: zitten capaciteiten in. Capaciteiten die zich verzetten tegen elke spanningverandering. Dat heeft een zekere traagheid/inertie en dat is wat je in de praktijk hebt. Daarom is de inertievertraging de default.
Stel dat je een puls hebt van bv. 2 ns, dan heeft dat niet genoeg energie om door de poort te raken: zo'n grote vertraging dat tegen dat die wilt veranderen, die al terug verdwenen is. Die puls wordt dus geabsorbeerd door de logica, door de intertie die erin zit. Dat geldt niet voor grote pulsen, als je niet te rap verandert gaat dat geen probleem zijn.
Dat is wat een inertievertraging is, je kan aangeven wat de minimumtijd is opdat dat pulsje niet zou verdwijnen. Je moet het niet vermelden, de defaultwaarde is dat die reject time dezelfde is als de a_time, de vertragingstijd. Ook dat is niet onlogisch: als je trage poorten hebt die een grote vertraging hebben, zijn dat poorten met een grote inertie. Als je snelle poorten met kleine vertraging hebt, kunnen die snel reageren. Snelle poorten laten dus veel makkelijker kleine pulsen door dan anderen. Als we kijken naar dat voorbeeld rechtsonder: vertragingstijd van 3ns, dus reject time is 3ns: alle pulsen kleiner dan 3ns verdwijnen. Dat zie je hier ook op dat signaal: verandering van a van laag naar hoog wordt doorgegeven, maar van hoog naar laag niet. De verandering bij a zal wel als transactie gescheduled worden, maar je gaat zien dat dat te kort is en die transactie terug verwijderen.
Op die manier kunnen we vertragingen weergeven.

Slide 199: Testbank: simulatie waarin je uw ontwerp gaat testen. Je hebt een prototype met een aantal ingangen en je legt waarden aan op die ingangen en kijkt wat op de uitgang uitkomt en kijkt of het overeenkomt met wat je

verwacht.
Je hebt het ontwerp dat je wil testen en je biedt die stimuli aan en je kijkt wat er op de uitgang komt. Als je een goede ontwikkelomgeving hebt kan dat grafisch, anders moet dat met de officiële VHDL-taal. Dat op zichzelf is ook een VHDL-beschrijving. Het specifieke is wel dat het geen in- en uitgangen heeft: alles gebeurt intern.

Slide 200: Entitiy testbench is het hoogste niveau dus daar geef je geen in- en uitgangen aan.
We zien een mux en willen die gaan testen door alle mogelijke in- en uitgangscombinaties aan te leggen. De beschrijving van die stimuli is beschreven, het gedrag ervan. Op de slide is dat op de niet-meest evidente manier geschreven zodat we kunnen begrijpen hoe dat in elkaar zit, zo zien we hoe VHDL redeneert. → Belangrijk om te begrijpen om het later naar hardware om te zetten.
Het proces wordt opgestart en op de eerste lijn (na begin) wordt een golfvorm beschreven, dat is de golfvorm die rechtsonder getekend is. Wat gebeurt er in de praktijk: 1 wordt 0 op dit moment (is onmiddelijk) en die andere overgangen worden gewoon onthouden dat er na zoveel ns nog iets moet gebeuren. Op dezelfde manier: die tweede uitdrukking zegt dat het nu 0 is en na 100ns wordt het 1, dat is de tweede golfvorm rechtsonder.
for-loop waarin staat dat select 0 is en na 25ns 1 wordt. Dan komen we een wait tegen en gaan alle waarden toegekend worden. Hoe lang blijven we daar wachten: 50ns. Bij de simulatie worden dan niet ineens 50ns gewacht, het proces blijft 50ns wachten, maar ondertussen zijn er nog transacties gebeurd: select verandert nog in die 50ns. Doordat select 1 wordt daarbinnen zal die testbank in gang schieten en nieuwe waarden berekenen. Op 25ns is alles wat op 25ns moest gebeuren gedaan, gaat verder, het volgende moment is 50ns, dan krijgt alles wat nieuwe waarden moest krijgen die nieuwe waarde. Het gaat naar het begin van die for die opnieuw wordt uitgevoerd en select krijgt een nieuwe waarde 0. Dat gebeurt 4 keer. Het is pas daarna (lus 4 keer doorgelopen) dat we aan het einde van het proces komen en gaan we onmiddelijk naar het begin van het proces terug en voeren we alles opnieuw uit. Alles wat tot nu toe gedaan was wordt volledig herhaald en dat blijft zo doorgaan.

Slide 201: Realistisch voorbeeld: onmiddelijk al heel wat bewerkingen tot gevolg. We moeten ook stimuli aanleggen. Als we dat met de hand zouden doen krijg je wat rechtsboven staat. In VHDL is elke component een parallelle uitdrukking, dat is een proces. Je kan alles ook met processen beschrijven. In dit geval zijn er 4: stimuli en de 3 componenten. Gaat eindeloos blijven wachten (gebeurt dus maar 1 keer). Flipflop rechtsonder: wachten op stijgende klokflank en pas dan uitgang aan met vertraging van 10ns.

Slide 202: Die processen gaan we gebruiken om te simuleren. Wat gebeurt er op simulatietijdstip 0. Het mag geen verschil maken in welke volgorde je de processen oproept, wij beginnen vanboven en werken naar onder.
We zien welke transacties allemaal gepland zijn. We geven eerst alle signalen U: die U's moeten zo snel mogelijk uit de schakeling verdwijnen want pas dan zit je in een gekende toestand. Er wordt gepland dat in 0 wordt en 1 na 20ns → wordt 1 keer opgestart en daarna gebeurt er niks meer mee. Bij het tweede

proces krijgt clk de waarde 0 en wachten we 10ns. Bij het derde proces krijgt D een nieuwe waarde: het inverse van in (is op dat moment nog ongekend) en D wordt ook U → alle bewerkingen op U worden ook U. Je blijft wachten tot de ingang verandert. Het laatste proces begint onmiddelijk met te wachten tot de klok 1 wordt. We moeten nu het resultaat berekenen dat op tijdstip 0 zal gebeuren. We zitten te wachten tot de klok 1 wordt, is die ondertussen 1 geworden? We wachten op een verandering op in, is die ondertussen veranderd? Als je gaat kijken, de klok is veranderd, maar ze is 0 geworden niet 1 dus we moeten niet uit die wait uitgaan. in is wel veranderd: van U naar 0 → het derde proces wordt opnieuw opgestart (te zien in PEQ), hierbij krijgt D een nieuwe waarde. In dit geval was D op hetzelfde ogenblik eerst U en daarna 1 geworden (op hetzelfde ogenblik), het finale effect is dat het 1 geworden is.

Slide 203: Dat proces moeten we voor alle tijdstippen gaan herhalen. Na de eerste transitie zie je wat we na de eerste klokcyclus hadden (op tijdstip 0 gebeurd). De finale waarden op simulatietijdstip 0 zijn 0 voor clk en in en nog steeds U voor D en Q. De eerste transactie die gepland is is op 5ns dus we gaan naar dat tijdstip. Op dat moment krijgt D een nieuwe waarde die we invullen, daardoor gebeurt er een event: D is veranderd van U naar 1. Er is niks dat op D aan het wachten is dus we kunnen overgaan naar de volgende: 10ns. Daar is een proces dat opnieuw ging opgestart worden: tweede. Daar wordt clk = 1 en die gaat opnieuw opgestart moeten worden. De klok is veranderd van 0 naar 1 waardoor het laatste proces nu wordt opgestart. Er wordt dus een nieuwe waarde gepland voor Q (binnen 10 ns). Alles wat op 10 moest gebeuren is gebeurd, dan gaan we naar het volgende ogenblik waarop transacties gepland zijn: klokgenerator staat op 15ns. Het laatste proces wacht nu weer tot de klok 1 wordt.
Op 25ns krijgen Q en D nieuwe waarden, dit is het eerste moment waarop alle uitgangen een andere waarde dan U gekregen hebben. Zo zien we duidelijk hoe alles op elkaar inspeelt: de nieuwe waarde voor D wordt bv. berekend op tijdstip 20 en krijgt pas zijn nieuwe waarde op tijdstip 25 → 2 verschillende zaken.

Slide 204: Nu we weten hoe een simulatie gebeurt is het makkelijker geworden om te kijken hoe we dat kunnen omzetten naar hardware. We moeten elk proces in VHDL omzetten naar een stuk hardware. Hoe: door te begrijpen wat VHDL doet.

Slide 205: Het is de bedoeling dat dat automatisch wordt gesynthetiseerd: automatisch komt er hardware uit. VHDL geeft veel flexibiliteit. Om ervoor te zorgen dat dat toch doenbaar is, dat je een programma kan schrijven dat die automatische omzetting doet kan je niet alles toelaten wat van VHDL mogelijk is.
Dat betekent dat in prakijk je, wanneer het over synthese gaat, je maar een subset van VHDL gaat toelaten. Wij krijgen die beperking niet tijdens het examen. Er zijn beperkingen die zijn ingevoerd om het syntheseprogramma makkelijker te maken. Daarom is er synthetiseerbare VHDL: subset die gestandardiseerd is. Wat zit er niet in: alle zaken waar je niets mee kan doen: alles wat met tijdsgedrag te maken heeft. Je kan niet specifiëren dat een antwoord een vertraging van 10ns gaat hebben want dat kan in de praktijk ook niet: je hebt een

poort en die heeft een vertraging, die kan jij niet kiezen. Vandaar zegt men bij VHDL ook dat er een waarschuwing komt, in het slechtste geval wil die niet synthetiseren. Dat soort zaken zijn niet toegelaten, ook het 'wachten gedurende 5ns' is niet toegelaten want er staat geen component tegenover. Alles wat met tijdsgedrag te maken heeft wordt niet aanvaard op het moment dat je synthese wilt doen.

Slide 206: Het lijkt redelijk eenvoudig om elke uitdrukking in VHDL om te zetten naar hardware. Het probleem treedt op als je een proces hebt dat iets op gedragsniveau beschrijft want dan heb je meestal geen component die dat doet, die dat proces implementeert. Als je iets hebt zoals op de slide getoond is de kans groot dat er geen component tegenover staat. Om dat te vertalen ga je niet het programma (zegt wat het doet, niet hoe het het doet) dat er staat zomaar omzetten lijn voor lijn want nu moet je gaan zeggen hoe je het gaat doen. Wel de goede oplossing: proces bestuderen, begrijpen wat het doet, afhankelijk van alle ingangen die aangeboden worden en een keer je weet wat het moet doen kan je hardware maken/kiezen die hetzelfde doet. Daar zit de intelligentie in die de meeste programma's nog niet hebben: verstaan wat een proces doet. Als je ingewikkelde beschrijvingen hebt, daar lopen ze op vast.
Het voorbeeld op de slide toont maar 1 slide: beschrijft in zijn totaliteit maar 1 stuk hardware. We hebben een interne variabele die we initialiseren op 0, daarna doorlopen we alle bits van A, dan XOR'en we met het resultaat dat we al hebben tot we alle bits gehad hebben. Uiteindelijk komt daar de XOR van alle bits uit. Syntheseprogramma's proberen het toch lijn voor lijn te vertalen en dan kom je uit wat getoond is op de slide. Er staat een generische constante in de code, op het moment van implementatie moet dat gekend zijn, die waarde. Dit is dus de domme implementatie, als je begrepen hebt wat het doet, weet je dat een goede maner om dat te maken getoond wordt linksonder op de slide. Veel syntheseprogramma's gaan het eerst zoals rechts schrijven en het dan omzetten naar dat van vanonder: optimaliseren. → Begrijpen wat het doet en vertalen naar hardware waarvan je weet dat dat hetzelfde doet.

Slide 207: Een keer het sequentiëel is moet je opletten. Er zijn 2 processen die ongeveer hetzelfde zijn (verschillen in het blauw). Boven: variabele om het tussenresultaat te stockeren, vanonder gebruik je het signaal. Dat genereert verschillende hardware: verschil tussen variabele en signaal: signalen krijgen hun waarde op het einde van het volgende proces (want gevoeligheid), bij variabelen op het einde van het proces (denk ik!).
Het verschil in hardware is te zien op de slide: subtiliteiten tussen variabelen en signalen waar je makkelijk overkijkt heeft beduidend invloed op de hardware die gegenereerd wordt.

Slide 208: Wat nog wel/niet toegelaten: alles wat hardware-matig een betekenis heeft (datatypes, bit, boolean, std_logic). Een keer in de imlementatie is er geen verschil meer tussen '1' en 'H'.
Gegroepeerde hardware bits: range gegeven: zo weet je met hoeveel bits je toekomt om het voor te stellen.

Slide 209: Je kan initiële waarden niet aan signalen geven. Je mag dat dus ook niet doen in uw VHDL.
Lussen: while-lus kan je niet doen. Je kan enkel lussen die je kan ontvouwen (waarvan je weet hoe dikwijls ze gaan uitgevoerd worden) gebruiken. Flankgevoelig: ofwel if clock-event en dan is dat het enige dat er mag staan ofwel wait until en dan mag er geen andere wait gebruikt worden. Alle andere dingen mogen niet gebruikt worden. Niveaugevoelig wordt weinig ondersteund en asynchrone wordt niet ondersteund.

Chapter 16

Les 16

16.1 Slides: 6_FSMD

Slide 210: Hoe vertrekken van VHDL om tot een schema te komen? We hebben in vorige les gekeken hoe details worden weergegeven in VHDL en hoe we vertragingen weergeven. Er is een subset van VHDL voldoende om tot synthese te komen.
Je hebt een beschrijving op hoog niveau en daaruit wordt de juiste hardware gegenereerd, maar zo ideaal is het niet. Wat eruit komt is in vele gevallen niet het optimale. Waar moeten we op ingrijpen om die VHDL zo te maken dat er zo goed mogelijke hardware uitkomt?

Slide 211: Waarom krijgt VHDL er geen optimale hardware uit? Heeft te maken met dat jij als auteur VHDL neerschrijft met bepaalde dingen in het achterhoofd en dat het syntheseprogramma dat moet interpreteren en moet trachten te proberen wat jij wou neerschrijven om dat zo optimaal mogelijk te implementeren.

- Wanneer je een beschrijving doet die geïmplementeerd kan worden met een latch in plaats van een flipflop die flankgevoelig is, dan moet het programma gaan nadenken of er wel een latch *moet* gebruikt worden, een flankgetriggerde flipflop ga je bij voorkeur gebruiken. Dat programma moet gaan raden of een flipflop gebruikt mag worden in plaats van een latch. Alles wat je hebt neergeschreven moet eigenlijk geïmplementeerd worden.

- Zelfs als er verschillende mogelijkheden zijn is het zelfs voor een programma moeilijk om alle mogelijkheden te onderzoeken. Er kruipt heel wat tijd in het onderzoeken van elke mogelijkheid.

- Met een aantal dingen kan je geen rekening houden, daar moet je eerst een prototype voor bouwen. Voor de vertraging bv. moet het eerst een keer geïmplementeerd worden, dat weet je niet vooraf. Het zijn meestal zoveel randvoorwaarden dat je niet automatisch tot de beste implementatie komt.

- Op het moment dat je het schrijft geef je aan wat er moet gebeuren en welke codering je wil gebruiken bv. Dat betekent dat het maar makkelijk

te implementeren is als je bepaalde flipflops hebt die erbij horen, maar die zijn niet per sé voorzien in die bibliotheek. Het kan dan zeer omslachtig geïmplementeerd worden. Het kan zijn dat de aangegeven encodering niet zo belangrijk was en dat er andere implementaties mogelijk waren.

⇒ Degene die het schrijft en degene die het moet maken zijn 2 verschillende entiteiten en dit kan leiden tot problemen.
In de praktijk betekent dit dat wat eruit komt deels afhangt van de mogelijkheden tot interpreteren van het syntheseprogramma maar het is ook een kwestie van hoe de VHDL geschreven is. Het op een andere manier schrijven kan tot een totaal andere oplossing leiden. Het is moeilijk om goede VHDL te schrijven want je moet weten hoe het programma het gaat interpreteren.

Slide 212: Voorbeeld: MUX kan je op verschillende manieren schrijven. Je kan met een toekenning van conditionele signalen met een if-uitdrukking werken of met een case of met een toekenning van geseleconstanteerde signalen. In principe geeft dat dezelfde functionaliteit.

- Als je dat met een if doet, krijg je de implementatie te zien op de slide. Door het te schrijven op die manier is de logsiche vertaling in hardware te zien. Er is een ingebouwde prioriteit: kijken naar eerste voorwaarde, dan pas naar rest. Die prioriteit vind je in de hardware terug. Eerst gebeurt de keuze tussen C of niet-C en daarna (als het niet C was) kijk je naar A en B.

- Andere manier: geselecteerde signalen. De logische tegenhanger ervan is dat je 1 grote MUX hebt waarmee je kiest. Je selecteert C zowel bij 10 als bij 11.

Op zich is het niet belangrijk of je nu het ene of het andere schrijft (buiten efficiëntie). Dat syntheseprogramma zou de beste oplossing moeten kiezen, maar ze zijn niet helemaal hetzelfde. Vandaar dat het syntheseprogramma dat niet gaat doen want er is wel degelijk een prioriteit. Als het syntheseprogramma niet weet dat dat mag, die twee door elkaar verwisselen, gaat die dat niet zomaar doen. Bij de if, als de meest beduidende bit een correcte waarde heeft en de minst beduidende heeft geen gekende waarde, dan is dat geen probleem want die meest beduidende wordt gebruikt om C te selecteren en de minst beduidende maakt niet uit. Bij de onderste is dat niet het geval: als je dat logischerwijze bekijkt, als een van de select-ingangen U is, dan komt er ook U op de uitgang. Dat geeft aan dat je het in het algemene geval niet weet. Je zou kunnen veronderstellen wat erop komt, maar algemeen bekeken is het zo dat als een van de ingangsbits U is, de uitgang dat ook is. Voor VHDL is er dus wel degelijk een verschil en het is dus van belang dat je het op de goede manier schrijft.

Slide 213: Een ander voorbeeld is waar sommige syntheseprogramma's misschien niet slim genoeg zijn om tot een optimale oplossig te komen. De rechtstreekse vertaling van de VHDL code getoond staat rechts ervan. Vanuit het standpunt van de hardware is dat niet het meest efficiënte want er staan 2 optellers en er gaat altijd maar 1 van de twee een nuttig resultaat leveren. Het is vanuit hardware-standpunt interessanter van het te doen zoals getoond. Het

geeft hetzelfde resultaat maar gebruikt andere hardware. Als uw syntheseprogramma slim genoeg is zal die weten dat dat op die manier gemaakt moet worden omdat dat hetzelfde is.
Je kan tegenslag hebben, voor iets ingewikkeldere dingen, dat het syntheseprogramma dat niet ziet. De manier om dat in orde te krijgen is van het expliciet te schrijven (zoals getoond in de code onderaan). Soms moet je het programma wat helpen om tot een optimale oplossing te komen.

Slide 214: Wat we nu gaan zien is onderdeel van een oefening op het examen. We gaan vertekken van een VHDL beschrijving, die omzetten naar ASM en dat omzetten naar een datapad en een controller.
De eerste stap is het omzetten naar een ASM-schema.

Slide 215: Hoe wordt dat vertaald? Het probleem is dat je dat niet rechtstreeks kan vertalen: VHDL geeft een beschrijving van wat er moet gebeuren in een bepaald stuk hardware, niet hoe het moet gebeuren, gewoon wat er moet gebeuren. Als je het gaat implementeren moet je weten wat het juist inhoudt om goede hardware te implementeren.
Hoe ga je daarbij tewerk? Een eerste oorzaak van verwarring is: je hebt in VHDL variabelen en signalen. Als je dat gaat bekijken heb je ook bij ASM variabelen en signalen. Bij ASM is een variabele een register. Een signaal is een draad die naar buiten komt. Het idee bestaat van je hebt dat precies alletwee dus een variabele bij VHDL vertaal je naar een variabele bij ASM en hetzelfde voor sigalen. Als je dat doet ben je fout bezig want ze hebben niks met elkaar te maken.
Gebruikt dus verschillende symbolen om de toekenning te doen in VHDL en ASM. Bij VHDL heb je := (variabele) en <= (signaal). Bij ASM heb je registers voor variabelen.
Hoe vertaal je naar ASM? Eigenlijk is het niet zo moeilijk om er een antwoord op te vinden: bij ASM betekent een variabele een register: iets moet onthouden worden. Als iets langer dan 1 klokcyclus blijft leven moet je dat onthouden, anders moet je dat niet onthouden. Da's de reden om het onderscheid te maken. Zowel een VHDL variabele als een VHDL signaal kunnen omgezet worden naar een ASM variabele. Bij VHDL blijft het altijd zijn waarde behouden, maar bij ASM is het belangrijk om te kijken hoe lang het nodig is om het te bewaren: voorbij de volgende klokflank of niet.
In het getoonde voorbeeld is dat nodig: altijd bij optellen dus vorige waarde moet onthouden worden → wordt een ASM variabele. Hier is het beschreven met variabelen. Het onderste is het makkelijkste om te zetten want bij ASM krijgt een variabele zijn nieuwe waarde ten gevolge van de klokflank. Bij VHDL krijgt een signaal zijn nieuwe waarde op de volgende wait, dat is niet juist hetzelfde. Als je waits alleen gebruikt voor klokflanken is dat hetzelfde, anders niet. Bij het onderste is het het geval dat wait gebruikt wordt als klokflank.
Wat vanboven staat: beneden wordt getest op i=2 en bovenaan op i=3 → verschil in ogenblik waarop signalen en variabelen hun waarde krijgen. Als je i bovenaan test is dat al een nieuwe waarde van i: oude waarde van i + 1. Beneden gaat i die nieuwe waarde nog niet gekrgen hebben want die gaat die pas gekregen hebben bij de volgende wait.
Daar moet je dus bijkomend op letten bij het vertalen naar ASM.

Slide 216: Als informatie niet langer nodig is dan alleen maar in dezelfde klokcyclus: i berekenen en i gebruiken, maar je moet die niet onthouden want volgende keer dat je er komt krijgt i een totaal nieuwe waarde die niks te maken heeft met de vorige. Dan heb je geen register nodig om dat in te stockeren (je mag dat, maar dat is een verspilling van hardware). In dat geval is het eigenlijk gewoon een tussenresultaat. Hoe noteer je dat in ASM → tussenresultaten kan je er strikt gezien niet in noteren, je moet die wegwerken. In het ASM schema moet het helemaal uitgeschreven worden en i wegwerken omdat die niet meer bestaat. Signalen als dusdanig bestaan eigenlijk niet in een ASM schema tenzij om naar een uitgang te brengen, om combinatorische dingen te berekenen. In VHDL had je het evengoed op de lichtblauwe manier kunnen schrijven.
Voor zo'n eenvoudige berekening is het niet zo'n probleem.
Je kan hetzelfde behouden, maar je kan in uw ASM schema ook tussenresultaten bewaren (signalen gebruiken). Dat gebeurt op de manier zoals onder het andere ASM-schema. Wanneer kan je dit doen: als je een functie kan afzonderen die je op een combinatorische manier kan neerschrijven. Ook als je testen doet kan dat combinatorisch: met een MUX iets kiezen. Je kan alleen geen functie zetten die ook tussenresultaten bewaart en onthoudt, dat soort zaken zet je daar normaal niet.
Waarom is het soms interessant van het zo te doen: uw schema wordt overzichtelijker (als je maar weet hoe het naar hardware moet vertaald worden) en je splitst het probleem op: je kan nog nadenken hoe je die functie kan implementeren. Het principe blijft wel hetzelfde: die i is weg.
Dit is voor een sequentiële schakeling. Alles wat zuiver combinatorisch is kan ook tussenresultaten hebben die je kan combineren.
Als je naar het proces beneden kijkt: je krijgt op het einde van het proces eventueel een nieuwe waarde (maar b is niet veranderd dus dan gebeurt er niks), a is wel veranderd dus Uit zal wel veranderen. Gans dat proces (niet die 2 uitdrukkingen) kan je schrijven als de getoonde parallelle uitdrukking: j is niet nodig, je moet die eigenlijk niet onthouden, die wordt alleen maar binnen dat proces gebruikt en is verder niet van belang.
Ook ASM heeft dus signalen: die = is vergelijkbaar met een signaal, tegenhanger ervan.
Variabelen en signalen kunnen geëlimineerd worden. Het kan altijd gebeuren dat signalen eigenlijk maar tussenresultaten genereren en geëlimineerd kunnen worden.

Slide 217: Nu we weten hoe we dat gaan vertalen kunnen we kijken, als we nu complexere beschrijvingen hebben (beschrijving in een proces), zal het ingewikkelder worden. Als je daar naar het sequentiële gaat kijken, dan heb je twee mogelijkheden om dat te beschrijven: clk-event of wait.

- clk: rising_edge getoond in de VHDL-code is een clock-event. Als je met een clock-event werkt, staat in dat ganse proces eigenlijk maar 1 klokflank. Dat maakt het makkelijker. Als je dat vertaalt naar een ASM-schema, komt gans dat proces overeen met 1 blok: 1 toestandskader, want er is altijd maar 1 klokflank. Afhankelijk van wat er nog voor de rest gebeurt kunnen er bijkomende kaders zijn.
  Als je ernaar kijkt, is er maar één deel wat op de klokflank gebeurt: wat ervoor staat gebeurt niet op een klokflank. Het is het asynchrone gedeelte

en onder de klokflank, dat is het synchrone gedeelte. In een ASM-schema kan je eigenlijk alleen maar synchrone dingen weergeven. Asynchrone dingen staan niet in het ASM-schema, de asynchrone dingen staan ernaast (eronder) beschreven in de blauwe kader. Die commentaar ga je later moeten implementeren (zorgen dat asynchrone reset op plaatsen waar s en x staan). Die eerste lijn: wordt dat een ASM-variabele of niet. Het antwoord is: hoe lang heb je dat nodig? t krijgt er een nieuwe waarde die niet afhangt van een vorige waarde van t, hangt enkel af van s die wel kan veranderen. Eens we aan het einde van het proces zitten hebben we die nadien niet meer nodig. Met andere woorden dit moet de klokflank niet overleven. Als het op een klokflank is gaat alles uitgevoerd worden, als het niet op een klokflank is gaat alles tot de elsif uitgevoerd worden. Daarom kan t geëlimineerd worden. Dit kan door ofwel die t in te vullen (s ← t vervangen door s ← s+1) of je kan dat in commentaar schrijven en t dan wel gebruiken (zo is het nu getoond). Als je het rechtstreeks kan vervangen, kan je meteen s ← s+1 invullen en dan is uw probleem meteen opgelost.

- **Slide 218:** Gebruik maken van wait. Er kunnen meerdere waits staan in een proces. Synthestiseerbare VHDL zegt dat dat niet mag (er mag er maar 1 zijn, anders is het niet automatisch synthetiseerbaar). Die waits komen overeen met een stijgende klokflank. Binnen 1 ASM-blok staat alles wat binnen 1 klokperiode gebeurt. In VHDL is dat alles tussen 2 waits: alles wat ertussen zit gebeurt binnen 1 klokperiode. Het VHDL-programma is maar een beschrijving van wat er moet gebeuren. Je kijkt wat er tussen twee waits zit en dat moet allemaal gebeuren. Soms kan dat redelijk eenvoudig zijn, het getoonde voorbeeld is niet zo moeilijk. Meestal is wat tussen twee waits staat niet zo moeilijk en kan je het meteen implementeren. Ga die code eens na met het getekende ASM-schema.
  Normaal heb je zoveel toestandskaders als dat je waits hebt in uw VHDL-code. Het is een belangrijke check om te maken na het tekenen van uw ASM-schema: als je 3 waits hebt en 4 toestanden, dat klopt gewoonlijk niet.
  Waar hier die wait staat is de klokovergang, dat is juist voor de toestandskader. Alles wat tussen twee opeenvolgende toestandskaders zit, dat zit allemaal in 1 klokperiode.
  Hier is het veel moeilijker om asynchrone dingen te beschrijven, bij het vorige was dat veel makkelijker. Het enige wat hier gebeurt is kijken naar stijgende klokflanken, je kijkt naar geen enkel ander signaal. Je kan met een wait natuurlijk ook wel asynchrone dingen weergeven. Dat doe je dan door te testen tot de reset verandert bv.

Slide 219: Combinatorisch: alles wat asynchroon is past niet in het schema en zet je ernaast. In de VHDL-code onderaan heb je twee processen: twee parallelle uitdrukkingen, elk een proces (denk ik). Je kan die twee combineren: als a of b verandert krijgt x onmiddelijk een nieuwe waarde en binnen dat proces eronder wordt x enkel gebruikt om y aan te passen. Je kan x dus evengoed elimineren. Hier is het zo eenvoudig dat je het samen kan zetten in plaats van het apart te beschouwen.

Slide 220: Onze ganse beschrijving bestaat heel dikwijls niet uit 1 enkele uitdrukking/proces. Als je nu verschillende processen hebt, hoe combineer je dat dan tot 1 grote oplossing? → Iets complexer dan dat het op het eerste gezicht lijkt. Men zet vaak de twee processen gewoon achter elkaar, maar dat is totaal fout. Dat zijn twee afzonderlijke dingen met elk hardware en die werken tegelijkertijd. Je moet die ASM-schema's dus combineren. Als ze tegelijkertijd opgestart worden, dan ga je verschillende combinaties krijgen van verschillende toestanden van het eerste en het tweede proces. Als je dat echt wil combineren krijg je wat gegeven is voor het geval dat ze gelijktijdig beginnen. Dit is al meteen vrij complex. Je moet voor ieder van de mogelijke ingangen die er zou kunnen zijn moet je dat gaan combineren, voor gelijk welke toestand waarin het zou kunnen zitten. Daaraan beginnen lukt meestal niet (behalve voor eenvoudige gevallen). Hoe los je dat op? De oplossing zit 'm in het feit dat ieder ASM-schema/proces een stuk hardware voorstelt.

Slide 221: Je kan ieder proces afzonderlijk gaan implementeren. Ieder proces creëert een controller en een datapad. Zijn die allemaal onafhankelijk van elkaar? Natuurlijk niet, anders zouden dat totaal losstaande dingen zijn. Waar vind je dat hier terug? In het feit dat er informatie tussen de twee stukken hardware zal uitgewisseld worden. De ASM-schema's voor beide stukken code zijn apart gegeven. Links: als s 0 is: krijgt een nieuwe waarde, anders wachten. Rechts: zolang s = 0, blijven wachten, anders gaan we iets doen. Ofwel gaat het ene dus iets nuttig doen, ofwel het andere. Hoe zorgen die ervoor dat die afwisselend iets doen? → s zal een signaal zijn dat door de twee gebruikt wordt en ook een signaal dat door de twee kan gezet worden. Dat lijkt gevaarlijk omdat het tot conflicten zou kunnen leiden, maar als je dat tegenkomt heb je slechte VHDL geschreven. Je kan makkelijk zien dat de ene s kan zetten en de andere resetten → S/R-flipflop. Je weet dat die nooit tegelijkertijd 1 kunnen zijn.
Op die manier kan je alles oplossen. Je lost het proces per proces op en zet het allemaal samen.
Nadeel: voor elk proces heb je een controller. In plaats van 1 grote controller als je het allemaal zou kunnen samenzetten. Je hebt evengoed een apart datapad. Als je dat in de praktijk gaat uitproberen, in OZ6 heb je zo'n probleem: daar delen ze een datapad.
Als je iets dergelijk doet, teken dan geen twee aparte datapaden (want je kan 2 registers x hebben die eigenlijk hetzelfde zijn), maar maar 1.
Waar eindig je normaal mee: met 1 datapad en zoveel controllers als je processen hebt. Daarin ligt de inefficiëntie want je kan het evengoed tekenen zoals onderaan: dat ASM-schema combineert beiden. Het voordeel is dat die s hier is weggevallen, je hebt die niet meer nodig. Als je het kan samenzetten levert het dikwijls een efficiëntere implementatie op.

Slide 222: Als je gaat kijken naar specifieke implementaties, dan zijn er binnen die standaardmogelijkheden nog uitbreidingen (afhankelijk van de software). Het voordeel ervan is dat het meestal een efficiënter resultaat oplevert. Het nadeel ervan is dat je dat niet kan overdragen naar een andere VHDL-omgeving.

Slide 223: Voorbeelden gegeven: als je niks aangeeft, gaat die 1-hot encodering gebruiken want dat is één van de efficiëntere coderingen. Je hebt ook nog

extra componenten waarbij je extra hardware kan gebruiken.

16.2 Slides: 7_Microprocessor

Slide 1: We gaan kijken naar programmeerbare processoren en hoe dat verbonden is met hoe we tot nu digitale ontwerpen gemaakt hebben. Het past in dat ganse kader.

Slide 2: Programmeerbare vs. niet-programmeerbare processor: we hebben geen vaste controller meer. In het vorige gingen we een oplossing van een bepaald probleem zoeken. In plaats van hardware te maken voor 1 probleem gaan we dat nu ruimer zien en we gaan proberen van ganse categorieën van hardwareproblemen op te lossen. Dat zijn dan micro-processoren → met 1 processor alle mogelijke stukken software draaien. Misschien zijn er bepaalde klassen van algoritmen die je wil versnellen waar je hardware voor wil maken maar niet voor 1 specifiek ding. Typevoorbeeld: digitaal signaal verwerkers: hebben heel specifieke algoritmen waar bepaalde algoritmen dikwijls terugkomen. Je gaat dan toch nog de hardware aanpassen aan het soort bewerkingen dat je wil doen. Hoe brengen we die flexibiliteit binnen: opsplitsen in 2 delen: programmageheugen: staat in wat er allemaal moet gebeuren. Die instructies worden gebruikt om te beslissen wat er in het datapad moet gebeuren. Zoveel aanpassing is er dus niet, we moeten alleen op een andere manier die controller maken, maar aan het datapad verandert er niks.

Slide 3: Stilstaan bij wat erbij gekomen is: We hebben instructies die enigzins vergelijkbaar zijn met wat in de controller zat, maar toch verschillend zijn. Hier wordt heel dikwijls gebruik gemaakt van addresseringen (denk ik).

Slide 4: NP-processor is voor 1 specifiek probleem gemaakt: zo efficiënt, goedkoop en met zo weinig mogelijk vermogen implementeren. Alle verwerking gebeurt dus binnenin de processor want de snelheid is daar belangrijk. Het gaat ook dikwijls over real-time verwerkingen: data komt binnen en moet aan dezelfde snelheid verwerkt worden.
Als je kijkt naar programmeerbare: gaan bewerkingen uitvoeren op data. Hier is het meestal zo dat je ingewikkelde programma's wil laten lopen die uit heel wat instructies bestaan (conditionele testen tussenin). Met eenzelfde hoeveelheid data doe je heel veel bewerkingen. Je gaat daar een ingewikkeld programma op laten lopen. Het is niet zo dat we heel veel data hebben en daar een beetje bewerkingen op doen, het is omgekeerd. Dat heeft tot gevolg dat de manier waarop normaal gewerkt wordt, de data in een geheugen geplaatst wordt: data wordt binnengehaald van extern, wordt op gerekend binnenin processor en wordt terug weggeschreven in geheugen.
Wat maakt het voor hardware van verschil: waar we in het vorige er vanuit gingen dat data daar kwam binnenstromen en dat we de resultaten er zomaar lieten uitlopen, gaan we hier informatie halen uit een geheugen, we gaan die verwerken, we gaan die informatie terugsteken in het geheugen.
Het geheugen is dus een belangrijk onderdeel. Hier heb je een extern geheugen waar alle informatie uitkomt. Strikt gezien kan je dat als onderdeel van het datapad zien, maar toch stelt dat bepaalde eisen, daarom gaan we dat hier als

iets apart zien. Wat er precies gebeurt is wat die instructies zijn. Je kan die instructies vergelijken met de toestanden in een toestandsdiagram. Er is hier een belangrijk onderscheid: een toestand komt overeen met 1 klokcyclus. Een instructie hier is algemener: bv. 'doe een deling', die ga je iteratief moeten doen en het geheel zal de deling uitvoeren. Hier heb je dat een instructie met meerdere klokcylci overeen gaat komen. Het is niet zuiver een toestand weergegeven, het is op hoger niveau aangeven wat er gebeurt.
Je kan die instructies op verschillende manieren gaan schrijven:

- Mnemonisch: hoe je normaal programma's schrijft, hoe je dat in assembleertaal gaat schrijven → makkelijker te lezen.
- Meer hardwaregebonden: legt de nadruk op hoe het in de hardware is. Je gaat uit het geheugen wat op plaats B staat en uit het geheugen halen wat op plaats C staat en dat optellen en stockeren in A.

Slide 5: Iedere instructie is onderverdeeld in velden. Als we het bij NP-processoren hadden over een instructie: data doorgegeven van controller naar datapad: controller die datapad instrueert welke bewerkingen moeten gebeuren. Er zijn daar bepaalde onderdelen en ieder onderdeel stuurt een stuk van het datapad aan. Hier heb je iets vergelijkbaars: velden die op zichzelf op een hoger niveau iets aangeven. Wat moet die instructie doen: meestal in 2 stukken: instructietype en opcode. Waarom: dat eerste stuk geeft aan hoe je de volgende bits moet interpreteren, hoe je de rest moet onderverdelen in velden om te weten wat dat betekent. Dat geeft een klasse van bewerkingen aan (bv. bewerkingen op het register, daar heb je andere velden voor nodig dan om iets in het geheugen te plaatsen/lezen want dan moet je het geheugenadres meegeven). Het instructietype bepaalt dus de layout van uw instructie. De opcode bepaalt meer specifiek wat er gebeurt (het zijn registerbewerkingen bv., maar het zijn optellingen ervan). Het adres zegt vanwaar uw data moet komen en waar het resultaat moet teruggestoken worden. Om dat efficiënter te doen gebruikt men gewoonlijk adresseermodi en af en toe is het nodig om constanten in instructies te kunnen zeggen (bv. incrementeren). In die assembleertaal duid je dat dan aan met een #.

Slide 6: Kijken we naar controller: programmageheugen en daar nog controller bij om de instructies te vertalen naar aansturing van het datapad. Het feit dat we dat zo goed kunnen maken (P-processor) is omdat wat die controller moet doen redelijk generisch is, onafhankelijk van welke instructies je juist gebruikt. Het is iets wat altijd herhaald wordt: instructie ophalen uit geheugen, die wordt gedecodeerd en er wordt bepaald waar je de data moet gaan halen en waar je die moet gaan stockeren, die adressen worden dan gebruikt om de operanden te gaan lezen, dan kan je de bewerkingen doen en dan kan je het resultaat wegsteken en je kan herbeginnen. Dat herhaalt zich eindeloos.
Voor de rest zit er nog de effectieve vertaling van een instructie in: hoe moet een deling precies gebeuren en hoe moet het datapad aangestuurd worden?

Slide 7: We zitten nog altijd met de snelheid: we willen zo krachtig mogelijke processoren. Hoe bepalen we de snelheid: op dezelfde manier als dat we dat vroeger deden: hangt af van het kritisch pad in uw processor, snelheid van het

datapad en hoe snel de controller kan werken. Als je het sneller wil doen werken moet je snellere componenten hebben. Hier komt ook nog eens het externe geheugen bij. Omdat het extern is, is het trager. Het is een groot geheugen en is traag, het is ook nog eens extern dus het is nog trager want je moet over zwaarbelaste lijnen dus je gaat trager werken. Daarom zie je dikwijls dat de uiteindelijke snelheid meer afhangt van hoe snel je instructies kan gaan halen en hoe snel je data kan gaan halen en terug wegsteken, meer dan de berekening doen op zichzelf. Het vraagt meer tijd om iets uit het geheugen te halen dan er een optelling mee te doen bv. Daar heb je varianten op: afhankelijk van hoe je uw instructies maakt (kleine instructies die je snel kan halen maar op zichzelf niet veel gaan doen of grote die je traag gaat halen en veel kunnen doen). In hoeverre gaan de instructies de snelheid bepalen?

Chapter 17

Les 17

17.1 Slides: 7_Microprocessor

Slide 2: Zelfde principe als bij niet-programmeerbare processor.

Slide 3: Het programmeerbare van processoren betekent niet dat die zomaar voor alles gebruikt kunnen worden, je hebt ook specifieke klassen daarbinnen waar die erg geschikt voor zijn.

Slide 6: Alleen het aansturen van het datapad en hoe instructies moeten gedecodeerd worden moet nog opgelost worden.

Slide 7: Een van de grote verschillen: bij de programmeerbare processoren: heel wat informatie in extern geheugen omwille van flexibiliteit: programma, instructies en data. Het idee is dat men daar de data en instructies gaat afhalen, dat zo snel mogelijk verwerken en dan het resutlaat wegschrijven. Omdat het geheugen extern is heeft dat impact op de snelheid. Het blijkt dat in veel gevallen, als je dikwijls naar dat extern geheugen moet, dat de snelheid van uw verwerking gaat beïnvloeden. Je zit daar aan de snelheden van het geheugen vast: aan extern bord/externe chip vast.
Om te illustreren wat de impact is van het feit dat je naar extern geheugen moet gaan kunnen we zien hoe dat de keuzes van de instructies gaat beïnvloeden. → Verschil van kwadraten berekenen. We gaan dat doen zoals helemaal rechts, want dan maar 1 vermenigvuldiging, da's efficiënter dan 2 vermenigvuldiging.
We gaan zien hoe we verschillende instructies kunnen kiezen en hoe dat impact heeft op de snelheid. De veronderstellingen zijn gegeven op de slide:

- Het aantal geheugentoegangen is gelijk aan het aantal instructies. Dat hangt af van hoe complex de operatie is, maar ook van hoeveel adressen je nodig hebt (want dat is in extern geheugen). Een adres in een extern geheugen is meestal even groot als een woord. Als je 16-bit woorden gebruikt, heb je minstens 64k geheugen nodig. Voor de instructies betekent dat: in instructie moet een adres staan (1 woord lang), maar ook wat er moet gebeuren (ook minstens 1 woord). We gaan ervan uit dat het aantal woorden in een instructie daar dusmee overeenkomt.

- Data uit het geheugen halen en resultaten wegschrijven. Elke keer je een woord leest uit het geheugen is dat een woord, hetzelfde met schrijven. Afhankelijk van hoeveel gegevens je moet halen uit het geheugen, daar heb je telkens 1 cyclus voor nodig. Dus voor alle operaties die er moeten gebeuren heb je een extra geheugentoegang nodig.

Slide 8: Als we dat toepassen, voor die bepaalde taak kan je, afhankelijk van de instructies die je hebt, een meer of minder eenvoudig programma hebben. Het kan zijn dat je 2 gegevens uit het geheugen moet halen. Die ganse operatie (+,-,*) vragen telksen 1 instructie. De lengte van het programma is op zich niet indicatief van hoe snel het programma uitgevoerd worden.
Je moet eerst kijken naar het ophalen: iets wat moet gebeuren en dan telkens 3 adressen. Om de instructie te lezen moet je dus 4 keer naar het geheugen. Dan heb je enkel de instructie gelezen. Om ze uit te voeren moet je ook data gaan halen (operanden). Je moet lezen wat er op A staat en op B en je moet dat resultaat wegschrijven in X. → 2 lees- & 2 schrijfoperaties. Per regel dus 7 operaties en in totaal dus 21 operaties.
Dat levert u het kortste programma op, maar er zijn andere mogelijkheden: instructies korter maken zodanig dat ze rapper gelezen zijn, dat zal impact hebben op het programma. Dat is onderaan de slide weergegeven: 2 adressen waarvan je er dus één gaat hergebruiken. Om dat te kunnen doen, heb je een bijkomende instructie nodig waarbij je elementen op een andere plaats in het geheugen kopiëert zodanig dat die niet overschreven worden door het resultaat. Er zijn nu wel 5 instructies. Ze gebruiken niet allemaal evenveel toegangen (het lezen is evenveel). Soms is het dus 2 en soms 3. Als je dat uitrekent: voor dit programma 28 toegangen nodig. Het is dus niet automatisch zo dat als je uw instructies korter maakt, het programma sneller gaat werken.

Slide 9: Met één adres werken. Dit kan door met registers te werken. Als je dat doet, zie je dat de instructies weer eenvoudiger worden (maar 1 adres). Als je kijkt naar de verschillende geheugentoegangen die nodig zijn voor de data is dat altijd maar 1. Weer 3 geheugentoegangen. Het programma is langer geworden, maar er zijn weer maar 21 toegangen nodig.

Slide 10: Met 0 adressen werken: voor uw bewerkingen met een LIFO werken zodanig dat de adressen impliciet zijn (LIFO werkt niet met adressen, je werkt altijd met de bovenste). Zodra je naar het geheugen moet heb je wel een adres nodig. Je kan gebruik maken van het feit dat je meestal in hetzelfde gebied van het gehugen moet werken. Dan ga je met een basisadres werken en met een offset. Wat heb je daarmee gewonnen? → er is een verschil tussen een basis en een offset. Een basis is normaal een volledig adres (eender waar in het geheugen), een offset is meestal een klein getal. Je gaat uw basis zo kiezen dat uw offsets altijd klein zijn. Bij een kleine offset moet je geen apart woord reserveren. Offsets kunnen meestal in dat eerste woord van de instructie erbij gezet worden. Op die manier heb je dus geen extra instructiewoord nodig en dat is waar je wint.
Voor de basis heb je er een nodig, maar je veronderstelt dat die basis gedurende het hele programma hetzelfde blijft. Deze veronderstellingen zijn misschien niet altijd juist, maar voor dit soort programma's is dat een heel logische veronder-

stelling.
Voor de instructies zelf hebben we altijd maar 1 woord nodig. Voor de data hebben we soms een interactie met het geheugen en bij berekeningen niet (want LIFO). We hebben dan maar 13 toegangen nodig. We hebben dan wel een redelijk lang programma, maar wel het snelste tot nu toe. Dit is ook niet het beste wat we kunnen bekomen: we hebben die adressen niet meer nodig omdat we interne registers van de processor hebben gebruikt om informatie op te slaan. Die LIFO is ook geheugen maar zit intern in de processor. Als we registers gebruiken zoals de accumulator, dat is ook intern geheugen. Dat gaat geen extra tijd vergen om die informatie te gaan halen. Met dat in het achterhoofd kunnen we verdergaan.

Slide 11: We kunnen proberen zoveel mogelijk registers te gebruiken. Je kan zorgen dat je er maximaal 1 adres erbij te zetten hebt, zo kan je met minder instructies wekren zonder dat de lengte van de instructie negatief beïnvloed wordt en zonder dat de geheugentoegang negatief beïnvloed wordt. Dit heeft dus maar 12 toegangen → nog beter dan het vorige. Daar kan je mee gaan spelen.

Slide 12: Degene die het meest gebruikt wordt: men gaat er vanuit dat men de operaties in 2 soorten kan onderverdelen: ofwel heb je interactie met het geheugen, ofwel doe je berekeningen. Berekeningen doe je alleen op interne waarden. Uitwisselingen met het geheugen, op dat moment gaan we 2 instructies voorzien die die mogelijkheid hebben (laden en stockeren). Als je dat toepast krijg je het programma zoals op de slide. Dat is ook vrij efficiënt omdat je niet meer naar het geheugen moet voor de berekeningen.
Dit wordt gewoonlijk verkozen omdat in een normaal programma een hoeveelheid gegevens heel veel bewerkingen worden gedaan. Je hebt die gegevens, je gaat daar heel veel bewerkingen op doen. In dit model gebeuren alle bewerkingen op registers. Daarna steek je het opnieuw in het geheugen (eens de bewerkingen gedaan zijn).→ Combinatie die u de mogelijkheid geeft om zo kort mogelijke instructies te hebben en de snelheid op te drijven.

Slide 13: Je kan altijd het adres waar je mee werkt in de instructie inzetten, maar dat is niet zo efficiënt want soms worden die berekend. Daarom zijn er verschillende adresmodi voorzien.
Wat zijn die: manieren om het adres te berekenen. Niet alleen het er zomaar inzetten maar het zo efficiënt mogelijk berekenen. Dat heeft tot gevolg dat je zelfs in veel gevallen de grootte van het adresveld kan reduceren, waardoor de lengte van uw instructie ook positief beïnvloed wordt.
Voor de rest, als je een keer in assembleertaal bezig bent is dat plezieriger omdat het meer overeenkomt met hogere programmeertalen.

Slide 14: Welk zijn de meest courante:

- Eenvoudigste: geen adres nodig, zit impliciet in de instructie in. Als je een reset accumulator hebt, moet je geen adres meegeven, het gaat automatisch over de accumulator. Bij LIFO kan je zeggen dat je wil optellen, dat gaat dan automatisch over de twee waarden bovenaan de LIFO.

- **Slide 5:** Onmiddelijke adressering: is eigenlijk ook niet adresseren: in plaats van het adres te stockeren steek je de waarde zelf erin, dat doe je meestal als je een constante hebt.

- **Slide 16:** Normale manier van werken: directe adressering: adres in geheugen dat verwijst naar een plaats in het geheugen waar de data staat. Dat kan direct naar het geheugen gaan of naar een register. Het voordeel ervan is dat je minder bits nodig hebt om het adres weer te geven. Dit zorgt dat uw instructies korter worden en data ophalen gaat ook sneller → dubbel voordeel. Wanneer je met registers kan werken ga je daar dus voorkeur aan geven.

- **Slide 17:** Indirecte adressering: adres van de data staat in het geheugen. Je hebt dus twee geheugentoegangen nodig om er te geraken. Waarom: normaal ga je ervan uit dat het programma zichzelf niet verbetert, maar dat is geen gezonde manier van werken want dan kan je moeilijk bijhouden wat uw programma aan het doen is. Als je het adres wil aanpassen in het programma doe je dat via indirecte adressering. In veel gevallen gaat het register indirect werken waarbij het adres in het register staat. → Zelfde voordelen.

- **Slide 18:** Relatieve adressering: je gaat adressen berekenen. In het voorbeeld is dat een offset opgeteld met een basis die ergens in een register bewaard wordt. Het zijn dus impliciete registers. Uit de opcode volgt automatisch welk register gebruikt wordt. Je kan daar expliciet een register voor gebruiken, dat is wat het meest gebruikt wordt waarbij je het adres van het register in de instructie erbij gaat zetten. Als je het altijd over een adres hebt gaat het altijd over de twee gecombineerd: adres + offset want die heb je nodig om het effectieve adres te gaan berekenen. Dit gebruikt men typisch om met vectoren te werken waarbij de basis het beginadres is van de vector en de offset is de index van de vector. Dan moet je niet expliciet dat nieuwe adres gaan berekenen, dat gebeurt automatisch door de hardware voor u.

- **Slide 19:** Geïndexeerd: hetzelfde maar dan omgekeerd: basis staat in instructie en offset staat in register. Heeft als nadeel dat het langste woord in geheugen staat. Het hangt er echter vanaf wat uw programma ervoor berekend heeft want dat staat ergens in een impliciet of expliciet register. Als uw programma de index moet berekenen, dan zal die in een register staan. Gaat uw programma de basis berekenen, gaat die kiezen tussen verschillende vectoren maar gaat die altijd geïntereseerd zijn in het vijfde element kan je zo dus ook werken. Je kan die basis ook kleiner maken door beperkingen op te leggen (veelvoud van een bepaald getal zodanig dat de laatste bits altijd nul zijn), maar dat is niet overal toepasbaar.

- **Slide 20:** Indexering met autoincrement of -decrement. Je voorziet extra hardware en geeft aan dat het automatisch moet gebeuren. Je kan ook veel meer complexe berekeningen met adressen implementeren in uw hardware, je gaat gewoon kijken of het de moeite waard is om het in hardware te voorzien.

Slide 21: Hoe gaan we zo'n processor ontwerpen: er zijn 2 manieren om dat te benaderen. Je hebt enerzijds de complexe instructiesetimplementatie en de gereduceerde instructieset. Dat heeft vooral impact op de instructies, maar onrechtstreeks ook op de hardware. Daarom gaan we dat onderscheid maken en kijken hoe dat de hardware implementatie gaat beïnvloeden.

Slide 22: Hoe gebeurt het ontwerp, wat is er verschillend ten opzichte van een niet-programmeerbare-processor. Het begint met het maken/kiezen van een instructieset. De instructiesets kunnen verschillen per processor. Afhankelijk van waar uw processor voor zal moeten dienen moet je de geschikte instructieset gaan kiezen. Dit is op een hoog niveau: wat heb je nodig: optellingen, vermenigvuldigingen, delingen,... Daarna moet je gaan kijken hoe je die instructies gaat uitvoeren met hardware die gemaakt kan worden. Je moet de instructies in stappen gaan onderverdelen en kijken welke algoritmes je gaat gebruiken om de instructies uit te voeren. Zo weet je wat je nodig hebt.
Dan weten we wat we in elke stap moeten doen en zou je er een ASM schema voor kunnen willen maken. Gewoonlijk zit er nog een stap tussen: allocatie van de datapadcomponenten. Je moet gaan kijken wat je nodig hebt in uw datapad. Daarmee rekening houdend, op dat moment weet je welke componenten er zijn, kan je zeggen dat bepaalde veronderstellingen herzien moeten worden: iets anders kiezen of de instructie in software gaan oplossen.
Pas daarna ga je het ASM-schema bepalen waarbij je gaat bepalen wat de verschillende stappen zijn, maar ook wat er in elke klokcyclus gaat gebeuren: wat allemaal gelijktijdig en wat na elkaar.
Daarna komt het ontwerp van de controller en het datapad.
Allocatie datapadcomponenten: zou dat er niet uit gevolgd hebben als we de laatste stap gedaan hadden? Waarom hebben we dat hier nodig en niet bij de niet-programmeerbare-processoren? Er is een andere invalshoek bij programmeerbare processoren en niet-programmeerbare processoren. Bij niet-programmeerbare wil je uw probleem zo snel en efficiënt mogelijk utivoeren in zo weinig mogelijk tijd, met zo weinig mogelijk hardware (kost) en zo weinig mogelijk vermogen. Je gaat ervanuit dat het allemaal in 1 cyclus kan en desnoods smijt je er alle hardware tegenaan die nodig is. Bij programmeerbare processoren is dat niet het geval want daar wil je een flexibele processor, die is sowieso traag want die gaat verschillende instructies moeten doen die één na één moeten uitgevoerd worden, die niet zo snel zijn omdat ze dikwijls naar extern geheugen gaan. De snelheid wordt hier eventueel maar bepaald door de snelheid van het externe geheugen. Hier is het dus geen probleem om eens een extra klokcyclus nodig te hebben.
We kunnen ook niet alles voorzien. We kunnen hooguit alle instructies voorzien die we nodig hebben/willen implementeren, maar we kunnen ook niet alles in 1 klokcylcus gedaan kan worden. Daarom moet je tussendoor al naar het datapad kijken want dat heeft impact op de instructies.
Er zijn die terugkoppelingen omdat je dingen opnieuw gaat bekijken.

Slide 23: Voor we beginnen ontwerpen gaan we het onderscheid tussen de twee klassen van implementaties maken: complex en gereduceerde instructeiset. Bij de complexe ga je in 1 instructie alle dingen proberen te doen die je regelmatig nodig hebt. Je gaat de kracht verschuiven naar de hardware. Je gaat dat

niet in software implementeren, je gaat zorgen dat de nodige hardware aanwezig is om dat te kunnen doen. Een gereduceerde instructieset wil alles eenvoudig houden. Als je blokken wil verplaatsen gaat alles één voor één verplaatst moeten worden.
Als je die benadering neemt heeft dat onmiddelijk ook zijn impact: bij de complexe ga je heel veel instrucies hebben, adresmodi, heel wat hardware hebben. Je gaat een complex datapad hebben → wordt trager.
Dat is het voordeel van die andere: heeft zo'n eeenvoudig datapad dat je daar wel met een hoge klokfrequentie kan werken. Bij het complexe ga je dus een kort programma hebben maar gaat elke instructie lang duren. Bij dat andere gaat het een lang programma zijn maar gaat het kort duren qua klokcycli. Meestal heb je winst aan die laatste manier van werken. Door zoveel mogelijk in software te werken gaat gemiddeld gezien uw programma sneller afgelopen zijn dan wanneer je trage instructies hebt.
Dat andere wordt natuurlijk nog gebruikt. Heeft zijn voordelen voor specifieke toepassingen: signaalprocessoren waar je heel veel nut kan hebben aan een instructie die een Fourier-transformatie uitvoert bv. Daar kan je er voordeel aan hebben omdat het veel gebruikt wordt. Beide ga je nog altijd terugvinden. Op het moment dat het ontwerp start moet je daarover nadenken welke je wil kiezen.
De complexe is makkelijker om te ontwerpen, bij de gereduceerde komt er veel bij.

Slide 24: De verschillende stappen die we moeten doorlopen.

Slide 25: Ontwerp van de instructieset: in de specificaties is bv. opgegeven dat het om 16 bit gaat (data en adressen). Omwille van het feit dat we zo weinig mogelijk tijd willen steken in het halen van instructies, zeggen we dat een instructie nooit uit meer dan 2 woorden mag bestaan. Je moet dus nog een aantal dingen zelf kiezen. Bepaalde dingen volgen enerzijds uit wat al vermeld is. Je krijgt er de gegeven configuratie uit.
Omdat we maar 1 adres ter beschikking hebben is het logisch dat we kiezen dat we bepaalde dingen in het geheugen stockeren en al de rest met registers moet gebeuren.

Slide 26: Opgedeeld in een aantal velden: 2 die bepalen wat moet gebeuren:

- Type van instructie.
- De eigenlijke operatie die moet uitgevoerd worden.

Onderscheid: type ging layout van de rest van de instructie ook nog bepalen. Het soort instructies dat we hebben is hier heel algemeen gehouden: instructies die alleen op registers werken, die op dingen uit het geheugen werken, dingen die nodig zijn om het programma te kunnen laten lopen en instructies die we in een vierde categorie gestoken hebben: 2 bits nodig om categorie aan te geven. De rest hangt af van het type zelf. Als je naar het meest complexe qua velden gaat kijken gaan we operanden hebben waar we adressen nodig hebben, voor registers (???).
Er is gekozen voor 3 bits voor een registeradres met dan nog 5 bits om te zeggen

wat we gaan doen.
Voordeel van velden: kunnen ieder onafhankelijk van elkaar gedecodeerd worden.

Slide 27: Verschillende types van registerbewerkingen: op een register/tussen 2 registers. We hebben er aritmetische, schuifoperaties,...

Slide 28: We moeten daar bitcombinaties aan gaan toekennen. De instructies worden dus gedefiniëerd in bits. Van die bewerking, die 5 bits die we hebben gaan we ook nog verder onderverdelen. Als die eerste bit 0 is gaat het over schuiven. Als het schuiven is, geeft de tweede bit aan of het naar links of rechts is en de 3 laatsten geven aan over hoeveel posities.

Slide 29: Meer registerinstructies. Dit gaat voor de hardware een belangrijke beslissing zijn of je het uit een register moet gaan halen of niet. Voor alle gevallen zijn er twee mogelijkheden. Kwadraat berekenen is geen intelligente instructie om te doen want je kan beter de vermenigvuldiging gebruiken daarvoor. Je hebt daar dus geen aparte instructie voor nodig. Worteltrekking is zo wel nodig bv.

Slide 31: Voor de tweede categorie: interactie met het geheugen, gaat het anders zijn want daar heb je maar 2 instructies: uit het geheugen halen of erin steken. Je hebt dus maar 1 bit nodig. Hoe ga je de bits dan gebruiken? Om adresseermodi te hebben. Bij de registers is er geen adresseermodus voorzien, alle informatie zit in die registers in. eenmaal we naar het geheugen gaan kan het interessant zijn om complexe adressen te gebruiken. Daarom worden daar ook bits voor voorzien.

Slide 32: Niet alle adresseermodi die besproken zijn, zijn gebruikt, enkel degene die courant optreden.

Slide 34: Bits toegewezen: 1 om de instructie aan te geven en 4 bijkomende bits. 1 bit om aan te geven of het tweede woord nodig is of niet. Waarom apart: op het moment dat je ziet dat het een verplaatsinstructie is kan je meteen naar bit 12 om te kijken of er nog iets gedecodeerd moet worden. Als het register indirect is heb je niks meer nodig, als het gewoon indirect is, bestaat het uit twee woorden. Daar kan je dus nog een bijkomende bit voor gebruiken.
Op dit moment is het interessant om erop te wijzen dat niet alle combinaties gebruikt worden: met 3 bits hebben we 8 mogelijkheden maar we gebruiken er maar 6. Anderzijds, zelfs met die bits die hier staan kan je combinaties maken die niet geldig zijn. Je kan bv. iets stockeren met een onmiddelijk adres, dat kan eigenlijk niet $\rightarrow$ niet-toegelaten instructie.
Twee dingen dus: de instructies zijn niet volledig gebruikt en er zijn mogelijkheden die niet gebruikt mogen worden. Wat doe je daarmee? Naar de gebruiker toe zeg je gewoon dat het niet mag. Maar mensen luisteren daar niet naar en gaan dat toch proberen, je mag je gebruikers dus niet vertrouwen. Alle gevallen die niet mogen vorokomen, daar moet je er dus voor zorgen dat die geen kwaad kunnen want als je die bits gewoon gebruikt in de veronderstelling dat die slechte gevallen nooit gaan voorvallen, kan je hardwareconflicten krijgen in uw processor of data die overschreven wordt in een register. Je moet er dus voor zorgen

dat alle dingen die niet mogen optreden, dat je ermee rekening houdt zodat ze geen kwaad kunnen. Als de gebruiker dat dan toch toepast ziet die er niks van. De dingen die jij niet mag gebruiken worden wel gebruikt, die doen wel iets nuttig in sommige gevallen. Waarom vertelt men u dat niet: dikwijls zijn dat instructies die gebruikt worden om de processor te testen, zaken intern te initialiseren. Voor de normale werking zijn die dus niet nodig. Vandaar dat als gebruiker toch niet aangeraden is om niet-bestaande instructies te gebruiken omdat die wel degelijk iets zouden kunenen doen, alleen weet jij niet wat dat juist zou kunnen doen.

Slide 35: We hebben sprongoperaties, conditionele sprongoperaties. Je moet ergens aan zien of je moet springen of niet. Je moet ergens een status hebben die je kan gebruiken om te beslissen of je moet springen of niet. Dat heb je bijkomend nodig. Ook springen naar subroutine en terugkeren. De details leggen we hier nog niet vast, behalve dat er geneste subroutines moeten kunnen zijn. Terugkeeradres: ook register voor gebruiken om dat te stockeren.

Slide 36: Overige instructies: zorgen dat de status gezet wordt. Registers vergelijken met elkaar en afhankelijk van de instructies de status al dan niet zetten. Expliciete instructies om de status te zetten. Initialiseren op 1 of 0.
Instructie die niks doet $\rightarrow$ als je moet wachten.

Slide 37: Machinetaal: je gaat al die bits zetten. Wat gaat de processor dan doen: die bits interpreteren. Je kan dat veld voor veld bekijken en daar automatisch uit afleiden wat er precies moet gebeuren in die instructie. De controller die die instructies vertaalt naar het datapad, die interpreteert de bits die in de instructie staan. Die worden gewoonlijk hexadecimaal geschreven om het voor de mens makkelijker te maken om dat te verstaan.

Slide 38: Assembleertaal: een iets meer verstaanbare taal dan machinetaal. Het is een mnemonische voorstelling. De vertaling van het een naar het ander is identiek, da's dus redelijk eenvuodig. Als je die vertaling dan toch moet doen kan je er gebruik van maken om nog andere dingen doen: gebruik maken van variabelen en labels: naam geven in plaats van een getal want dat heeft voor ons meer betekenis. Die wordt bij de vertaling dan weer omgezet in een getal.

Slide 39: Opsomming vanwat welke opcode doet.

Slide 40: Streepjes zijn don't cares.

Slide 43: Waarom kan het nuttig zijn om iets in assembleertaal te programmeren in plaats van in een hogere taal? Stel dat we het gegeven probleem hebben dat bescrheven is in hogere taal.

Slide 44: Als je dat in assembleertaal zet moet je meer expliciet gaan zeggen wat de processor allemaal moet doen. Je kan zo ook optimaal van de processor gebruik maken. Werken met registers is bv. interessant, te verkiezen boven geheugentoegangen. Je kan zo ook kijken wat de keuzes zijn en wat er meespeelt

in hardware. Of wanneer je om een of andere reden beperkt bent in uw geheugen en je het dus zo compact mogelijk moet maken. Meestal gebruikt men Von Neumann: data en programma staan in hetzelfde geheugen (code en tekst). Je moet dus plaats voorzien voor uw variabelen. Programma en data volgen elkaar heel dikwijls op dus na uw data komt in dit geval het programma. Als je verwijst naar een bepaald adres moet je al uw adressen gaan aanpassen, daarom gebruik je labels. Schuift het label op, zal de assembler daar automatisch het verbeterde adres invullen.

Slide 45: Als je naar het hoge niveau kijkt, de variabelen in het geheugen worden geïnitialiseerd. We gaan zoveel mogelijk registers gebruiken. Slechts als het strikt nodig is gaan we naar het geheugen gaan. In dit geval zijn er 8 registers ter beschikking, dat is het maximum. We gaan die gebruiken als tijdelijke plaats voor de variabelen. Als je efficiënte instructies kan gebruiken, doe dat dan ook. Dan ga je een lus hebben met variabelen die bijhoudt waar je zit en tot waar je moet tellen. Ook die ga je initaliseren en in registers plaatsen. Al die dingen die handig zijn om te gebruiken, daarvoor ga je registers gebruiken. Je kan ook relatieve adressen gebruiken: altijd 1 bij optellen. Dat doe je dus niet zoals op hoog niveau (elke keer opnieuw berekenen), maar gewoon het adres van het eerste element stockeren en daar altijd bij optellen.

Slide 46: Je gebruikt een label zodat je weet waarnaar je moet terugspringen. A[i] ga je één keer gebruiken uit het geheugen en daarna zoveel mogelijk hergebruiken.

Slide 48: De interactie met het geheugen is beperkt, er zijn zoveel mogelijk interne registers gebruikt om de snelheid op te drijven.

Slide 50: Wordt automatisch vertaald naar bitcombinaties. Er waren een aantal don't cares, maar die kan je niet invullen. Overal waar er don't cares stonden zetten we hier een 0. Als je echte instructies bekijkt ga je nooit don't cares zien, men gaat er altijd een 0 zetten. Soms zie je ook echte adressen ingevuld staan (bij 0410). Ook het echte adres van A is aangegeven.

Slide 51: De volgende stap is van te gaan zeggen hoe we die gaan implementeren.

Slide 52: Voor de meesten is dat redelijk makkelijk: optelling is niet zo moeilijk. Als het ingewikkelder is kan dat verschillende iteraties in beslag nemen.

Slide 53: Wat moet er in een instructie gebeuren: je moet niet alleen schuiven, je moet beginnen met de instructie te gaan halen. Het eerste wat we daarna doen is die in een instructieregister steken, de programmateller verhogen, daarna moeten we decoderen: bepalen wat er moet gebeuren vanuit al die bits. Die grote driehoek stelt een traditionele test voor.
Instructieregister: eerste twee bits bepalen wat het betekent. Dat is de volledige decodering van uw instructie. Het decoderen gebeurt hier stap voor stap. Je ziet hier het grote voordeel van met velden te werken en dat niet te zien als iets

dat het een voor een decodeert. Het is beter om dat stukje per stukje te doen en het duidelijk op te kunnen volgen wat er precies gebeurt.
Als we weten dat het schuiven naar rechts is kunnen we het uitvoeren. Als dat gedaan is kunnen we naar de volgende stap gaan.
Je zou kunnen denken dat dit op een ASM-schema trekt, maar dat is het niet. Het is een gewoon stroomschema die één voor een aangeeft wat er moet gebeuren: binnen zo'n blokje gebeurt eerst het eerste en dan het tweede, dus niet allemaal tegelijkertijd.

Slide 54: Dan kan je dat nagaan voor alle andere dingen en ga je één voor één de instructies af.

Slide 55: Als we naar optellingen gaan kijken is het wat meer decoderen: registerinstructies: optelling en aftrekking met 2 registers en dan zeggen we welke instructies moeten uitgevoerd worden.

Slide 56: Verplaatsinstructies: het decoderen is daar wat anders. Je moet zeggen of het over stockeren of laden gaat, of het een 1- of 2-woordinstructie is. Die aarding is het teken dat je dat niet mag gebruiken. → 2 mogelijkheden die we niet gebruik hebben van de 8. Als het over 2 woorden gaat, gaan die combinaties ook niet gebruikt kunnen worden.
Het gaat dus over het laden van een constante, het tweede instructiewoord ga je gewoon gebruiken. De programmateller moet dan nog een keer verhoogd worden.

Slide 57: De andere laadinstructies. In de vierkante kader rechts: om dat uit te voeren heb je 3 geheugentoegangen na elkaar voor nodig: eerst het tweede woord van de instructie, dat gebruiken om adres te gaan halen en dat als adres gebruiken om de data te gaan halen.

Slide 58-64: Als voorbeeld.

Slide 65: Subroutines. Nu moeten we gaan beslissen hoe we dat gaan implementeren. We moeten naar een subroutine kunnen springen, dat mag genest zijn en je moet daar source 1 voor gebruiken bv. Nu ga je kiezen hoe je dat implementeert. Je kan zeggen dat je een LIFO voorziet, of je kan doen zoals hier geïmplementeerd: geen aparte hardware voor LIFO → stap voor stap doen zonder er aparte hardware voor te gebruiken.
Je moet dan gaan beschrijven hoe je dat gaat doen. Je gaat register source 1 gebruiken als het leesadres. Bij LIFO is er een strikt verband tussen een lees- en schrijfadres: leesadres is schrijfadres - 1. Je moet daar dus geen register voor voorzien, je moet dat gewoon onthouden, die berekening. Je gaat ervanuit dat het terugkeeradres de volgende instructie in het programma is. Dat geeft dan de kader rechts (geel).
Nu heb je in detail beschreven hoe dat geïmplementeerd gaat worden → verschil in deze stap: voor complexe operaties wat je in deze stap gaat verduidelijken. Je zegt nog niet wat er allemaal in die klokcycli gaat gebeuren, je zegt gewoon wat er moet gebeuren.

Slide 66: Je gaat eerst het schrijfadres verlagen: was het leesadres. Dat kan je meteen gebruiken om de locatie van de volgende instructie te gaan halen. Hier zie je duidelijk dat het geen ASM-schema is want het zou niet werken als het in parallel uitgevoerd werd. Je kan enkel wat rechts in de kader staat in parallel uitvoeren.

Slide 68: In het datapad een bewerking doen: inhoud van 2 registers vergelijken om te beslissen wat er moet gebeuren.

Slide 69: Voor we aan het ASM-schema komen: allocatie van datapadcomponenten. We hebben er nog niet echt hardware aan gekoppeld. We mogen niet tot op het einde wachten want dan moeten we misschien herbeginnen. We gaan nu kiezen wat er in het datapad moet. Je moet dat ooit eens beslissen want je kan niet alle mogelijke hardware voorzien die wel eens interessant zou *kunnen* zijn. We gaan rekening houdend met wat er moet gebeuren bepalen wat de logische invulling van het datapad is.

Slide 70: We weten al redelijk veel: extern geheugen dat werkt met woorden van 16 bit en 16 bit als adresgeheugen. De registerbank: 3 bits om een register te kiezen, dus een registerbank met 8 registers. Wat moeten we kunnen doen met die registers: in 1 instructie willen we 2 operanden eruit kunnen halen en 1 resultaat wegschrijven. Op dit moment zou je kunnen zeggen dat het wenselijk is om een registerbank te hebben met 2 leespoorten en 1 schrijfpoort (want optelling kan enige tijd duren). Die hebben ook 16-bit woorden. Status: moet maar 1 bit zijn.
Makkelijke deel: operatoren, bepalen wat erin moet zitten. Je kan kijken wat alle bewerkingen zijn die moeten gebeuren. Het kan zijn dat je er bewerkingen hebt tussengezet waarvan het niet evident is hoe je dat in hardware gaat vertalen, daarvoor kan je uw instructie moeten gaan aanpassen. Voor bepaalde dingen (schuifoperaties of vergelijkingen) moet je bijkomende hardware voorzien. Je moet nu gaan kijken of er nog dingen bijgezet moeten worden. Je gaat een deling doen, dat kan met de hardware die we al hebben. Worteltrekking kan moeilijker zijn: liefst andere hardware voor, tenzij je dat niet dikwijls genoeg nodig hebt, dan kan je het verwijderen als instructie. We gaan 1 aanpassing doen: we hebben een enorm snelheidsverlies als we indirect iets uit het geheugen moeten halen: 3 geheugentoegangen voor nodig.

Slide 71: Stel dat je dat als 1 geheel bekijkt en dat 1 toegang 50ns nodig heeft, dan heb je 150ns nodig voor de traagste operatie als je dat in 1 klokcyclus zou willen doen. Dat betekent dat alle operaties heel traag gaan zijn. Dat wil je niet, dus ga je de principes van multicycling en pipelining toepassen. Je gaat 3 cycli voorzien waarin je 1 geheugentoegang doet $\rightarrow$ optelling zal maar 50ns duren.
Je gaat dus niet meer dan 1 lees- of schrijfoperatie doen in 1 cyclus. Je moet het tussenresultaat dan wel ergens kunnen bewaren.

Slide 72: Bijkomend register voorzien: adresregister: register waar (tussen)resultaten van het berekenen van een adres in komen te staan. Dat zal de efficiëntie opdrijven.