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 éé 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 éé specifieke familie van problemen op te lossen.
- Controle-eenheden.

Waarom die tweede groep? → Veel goedkoper! Je hebt ongeveer 1000 keer zoveel energie nodig op een programmeerbare processor dan bij een niet-programmeerbare. Bv i.p.v. 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 elektornica.

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 dan kijken hoe dat geïmplementeerd kan worden.

Slide 11: Vereiste voorkennis.

Examen: FSM is schriftelijk, rest is ook mondeling. 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 specifieert. 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 flip flops). Dan komt er nog een nivau 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 middenste 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, 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 werk. 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 een 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 ga je 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 een 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: men wil het omgekeerde, wat is er de schakeling voor? Maar zelfs dit is een onvolledige beschrijving van wat die schakeling doet want het beschrijft alleen de logische functie en in 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 ik de schakeling zodat die daaraan voldoet? Er zijn 2 manieren om dat te benaderen: een schema tekenen/zoeken dat eraan voldoet. Anderszijds 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 kun je ook altijd omzetten naar een waarheidstabel. Verschilllende logische functies kunnen eenzelfde waarheidstabel geven. De basisfucntionaliteit 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 rechtsreeks 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 wiskunding 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 nog 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 een 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 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 meoten kiezen wat het beste is.

Slide 29: Het probleem met de canieke: 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 at er rijen zijn waar een 1 in voorkwam op het einde. 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 bevat de bovenste elke term 3 ingangen vs. 2. 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 mogelijkehden het interssantst 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 input 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 doen en nagaan wat het tijdsgedrag en elektrische eigenschappen etc zijn. Je moet een prototype bouwen en daarop metingen uitvoeren. Wanneer dat tot een geintegreerde schakeling moet leiden is dat een verpilling van tijd en kosten du snormaal gaat ede fabriekant van die geintegreerde 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 info om te simuleren en beschrijven en dat systeem op een voldoende hoog niveua 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 specifieren. 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 hoogst niveau heb je een beschrijving van het gedrag: "dit blokje moet dat doen" . Je moet ok 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. (OPLIJSTEN) 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 zijn want in de taal is de implementatie exact beschreven. Nadeel: als er nieuwe ideeen komen/er zijn tekorten, dan kan men het niet meer veranderen –¿ nieuwe versie. Slide 38: VHDL is voor digitale systemen, maa 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 differnetiaalvergelijkingen nodig om tot uw oplossing te komen. Slide 39: andere kandidaat die misntens 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 syntaxv erschilt. Slide 40: wat is nu het interessantste: schema of hardwaretaal? –¿ hangt af wat je ermee moet doen. Schemas zijn interssant voor kleine ontwerpen, VHDL is gewoonlijk intersanter wanneer het over groter eontwerpen gaat. Nadelen van VHDL: • 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 beshcrijven heb je onmiddelijk veel linen 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 getrainder. • VHDL bevat alle mogelijkheden: alle mogelijkehden 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: • Het is een standaard: je kan het gebruiken om info 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 hebben (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. N een programma, als uw parameter niet voldoende groot/whatever is, pas je dat gewoon aan. • In een shcema moet je onmiddelijk componenten kiezen om erop te zetten. VHDL heeft het voordeel dat je het gedrag kunt beschrijven, je moet dus niet per se een implementatie te kiezen. Slide 42: beperkingen: • VHDL heeft zoveel mogelijkehden 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 makkleijker 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 wat je neershcrijft. Ind e meeste gevallen raadt hij dat zo strict mogelijk. Je moet in de meeste gevalle de compiler dus wat helpen. Je moet ongeveer weten hoe hij interpreteert wat jij hebt neergeschreven. Je moet je dus 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 ontewrper. Slide 43: schema tekenen: kan in een keer, maar in de praktijk ga je op de hierarchische manier werken. Je gebruikt blokjes die je eventueel nog niet definieert en waarvan je later de ontbrekende blokjes gaat specifieren. 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 bewekringen 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 htzelfde 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 specifieren. 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: entitteit 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 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 geimplementeeerd moet worden, gewoon beschreven wat het doet. Het is ook zo dat voor een bepaalde ocmponent er verschillende implementaties kunnen zijn: gedrag beschrijven, implementatie beschrijven, voor sommige thoerien zal iets goe dijzn, in anderen niet. De architecturen krijgen daarom een naam omdat je zo kan kiezen voor een bepaalde iplementatie. 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 geimplementeerd is. We hebben een component Comparator nodig, dat is een virtuele coponent 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 rehcts vanboven met een schema staat. Belangrijk is om in gedachten te houden dat het een beschrijving van hardware is. De volgorde vna 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 stricte taal dus normaal moet je alles heel uitgebreid beschrijven, maar als je uw namen goed schrijft kan hij raden wat je bedoelt. Als de specs 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 verchilt vrij hard van een gewone programmeertaal, het ziet er alleen uti als een programmeertaal. Vanboven: "gewone" programmeertaal: functie die je definieert. In vhdl beschrijf je gedrag. Het topniveau is het hoofdprogramma: main() die de functie 2 keer gaat gebruiken. Eblagnrijk verschil: bij het vorige waren de uitdrukkingen parallel, hier sequentieel: het ene wordt uitgevoerd na het andere, er is een stricte 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 soft-

wareprogramma 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 spannign afzet. Dat is de enige manier omd ie tot stilstand te brengen. Software: bug als blijft runnen, hardware: bug als 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 specifieren: 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 reele tijd. Het kan zijn dat je een minuut moet rekenen om de simulatie 1 nanoseconde vooruit telatne gaan.

Sides: 3

Slide 3: als we elektronische schakelingen willen implementeren, hoe gebeurt dat ind e 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. Daartussen is een stuk waar het noch hoog, noch laag is: ongedefinieerd. Waarom bereieken? We willen onze schakeling zo robuust mogelijk. We willen niet dat we de schakeling zo nauwkeurig moeten maken zdd 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 feit van een bereiek te gebruiken laat fluctuaties toe. Op diem anier kunnen we dat oplossen en kunnen we veel makklijker ingewikkelde zaken maken, da thebben we ook als we verschillende transistoren op 1 component willen zetten. Ook het gebruik beinvloedt dit: elektornica 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 mogelijkw aarden, hoe kleiner de bereiken zijn. We hebben het nog niet over 0 of 1 gehad. Het hoeft niet per se zo te zijn dat 0 laat is en 1 hoog –¿ positieve logica, maar je kan ook negatieve logica hebben waarbij men het omgekeerd doet. 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 geisoleerd van het blokje p. als de spanning klein si zal er niks gebeuren en zal de halfgeledier zich gedragen als een isolator. Wordt de spanning voldoende groot zal die elektronen aantrekken om een geleidend pad te creeren en wrodt 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 spannign betekent: de negatieve spanning is heel groot. Op die manier kunnen we die 2 soorten schakelaars maken. We

krijgen dit op dezelde geintegreerde 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 spannign is, dan zal de transistor (rood) nie 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.a ls 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 teenwoordig neit 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 du seen 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 zichzelfstaande 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. Die komt omdat de twee uitgangen aan elkaar hangen. Door een bedrading aan te leggen is er een extar functionaliteit. Het nadeel is dat je met NMOS blijft werken en het statisch vermogenverbruik hebt. 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 een onderaan. Vanboven een PMOS en het ander een NMOS. Het een si een pull up, het ander een pull-down. Hier hebben we geen satisch 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 (?? 1.28.15). Slide 11: je mag dit nooit met CMOS doen: die werkt aleen 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 geleid, wat een zeer lage weerstand is, er zal dus een enorm grote stroom doorlopen. Gevolg: uw schakeling ontploft (wordt getoond in de slides normaal). Je moet ds 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 een van de twee zal kapot gaan. CMOS-uitgangne mag je dus nooit zomaar aan elkaar hangen! Slide 12: Als we ze toch aan elkaar moeten hangen, gaan we een speciale oplossing moeten 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 dat erbij komthebben 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.