Digitale Techniek

Opgaven en uitwerkingen

Jesse op den Brouw

Tweede editie

Versie: 2.0α
Datum: 6 mei 2022

DE HAAGSE HOGESCHOOL

De auteur schenkt de royalties aan Stichting KiKa.

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

Inhoudsopgave

1

1.1. Ontwerp een omschakelbare buffer/NOT. Dit is een schakeling die onder besturing van een stuursignaal de ingangswaarde doorgeeft (buffer) of inverteert (NOT).

1.2. Ontwerp een NOT-schakeling met een NAND-poort.

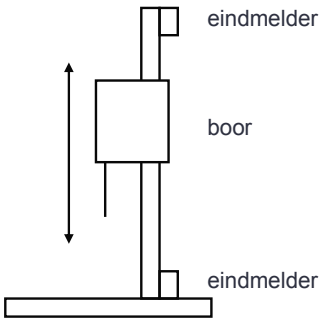
1.3. Ontwerp een NOT-schakeling met een NOR-poort.

1.4. Bepaal de waarheidstabel van een EXOR waarvan één ingang geïnverteerd is.

1.5. De beschrijving van een AND is: de uitgang is 1 als alle ingangen 1 zijn. Hoe zou de beschrijving zijn als er van de nullen wordt uitgegaan?

1.6. Hoeveel verschillende functies zijn er te maken met twee variabelen? En met drie variabelen? En met n variabelen?

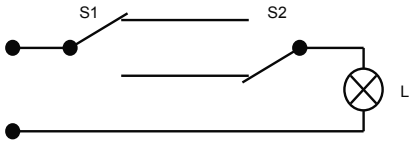
1.7. Een automatische kolomboor (figuur P1.1) heeft twee eindmelders: een aan de bovenkant en een aan de onderkant. Een melder kan open zijn (boor is daar niet) of gesloten zijn (boor is daar wel).



Figuur P1.1: *Kolomboor.*

Hoeveel combinaties van open en gesloten zijn er mogelijk? Welke combinatie komt nooit voor? Stel een tabel op met alle mogelijkheden en geef met een paar woorden aan wat de mogelijkheden inhouden.

1.8. Een elektrisch schema met schakelaars kan ook als een digitaal systeem worden gezien. Een bekende schakeling is de zogeheten wisselschakeling die veel bij trappen voorkomt. Deze schakeling wordt ook wel de hotelschakeling genoemd. Zie figuur P1.2 voor het schakelschema.

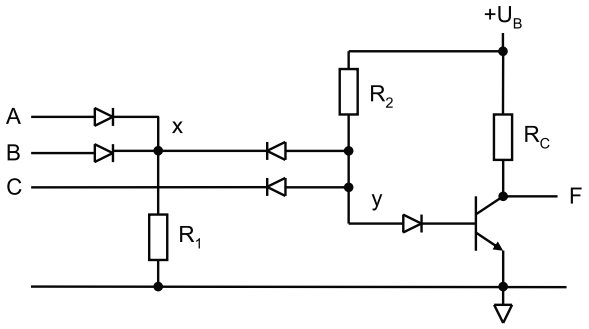


schakelaar in rust: 0
schakelaar geactiveerd: 1
lamp uit: 0
lamp brandt: 1

Figuur P1.2: *Schema wisselschakeling en beschrijving logische signalering.*

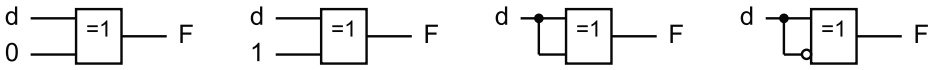
Geef de waarheidstabel van de wisselschakeling.

* **1.9.** In figuur P1.3 is een schakeling gegeven met diodes, weerstanden en een transistor. Stel een waarheidstabel op voor alle signalen in de figuur.



Figuur P1.3: *Transistorschakeling.*

1.10. Aan een EXOR-poort met twee ingangen wordt één ingang als volgt afwisselend aangesloten: 0, 1, normaal of geïnverteerd. Zie hiervoor figuur P1.4. Hieruit is op te maken dat er slechts één ingangssignaal is, namelijk d. Daardoor vereenvoudigd de werking van de poort. Bepaal de werking van deze vier mogelijkheden.



Figuur P1.4: *Vier EXOR-poorten.*

1.11. Als opgave 1.10 maar nu met een EXNOR-poort.

1.12. Toon aan dat een EXOR-poort door middel van een extra NOT-poort kan worden omgezet in een EXNOR-poort.

1.13. Bepaal de eenvoudigste vorm van de functies in tabel P1.1.

1.14. Ontwerp een buffer, NAND, NOR en EXNOR met behulp van NANDs.

1.15. Gegeven de functie: $y = \overline{a} \cdot b + a \cdot \overline{b}$
Ontwerp deze functie met alleen NANDs.

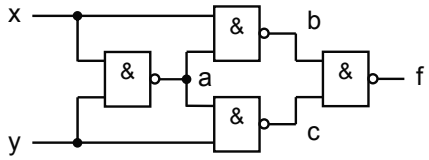
Tabel P1.1: *Drie waarheidstabellen.*

a	b	y
0	0	1
0	1	1
1	0	1
1	1	0

a	b	y
0	0	1
0	1	0
1	0	1
1	1	1

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

1.16. Gegeven de schakeling in figuur P1.5. Bepaal f als functie van x en y.



Figuur P1.5: *Schakeling met NAND-poorten.*

1.17. Gegeven de volgende functie: $f = (a + b) \cdot (c + d)$
Ontwerp voor deze functie een schakeling met alleen NOR-poorten.

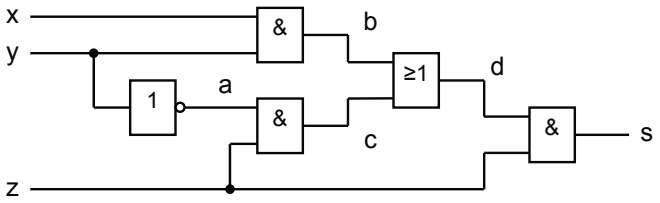
1.18. Gegeven de waarheidstabel in tabel P1.2.

Tabel P1.2: *Waarheidstabel.*

a	b	y
0	0	1
0	1	1
1	0	0
1	1	1

Ontwerp de bijbehorende schakeling met alleen NANDs. Ontwerp de bijbehorende schakeling met alleen NORs. Dit mogen ook poorten zijn met meer dan twee ingangen.

1.19. Gegeven is het schema in figuur P1.6. De vertragingstijden van de poorten is gegeven in tabel P1.3. De tijden zijn in nanoseconden (ns) Bepaal de minimale en maximale vertragingstijd van deze schakeling.



Figuur P1.6: *Een schakeling.*

Tabel P1.3: *Vertragingstijden.*

Poort	$t_{P(min)}$	$t_{P(max)}$
AND	2,4	6,9
OR	3,1	7,2
NOT	1,5	4,3

1.20. Wat is de periodetijd van een signaal met een frequentie van 33,3 MHz? Wat is de periodetijd van een signaal met de frequentie van 3,4 GHz? Wat is de frequentie

van een signaal met een periodetijd van 20 ns? Wat is de frequentie van een signaal met een periodetijd van 104,167 μs?

2

2.1. Zet om van binair naar decimaal:

1111_2 110011_2 01111111_2 1010010110100101_2

2.2. Zet om van decimaal naar binair:

25_{10} 10_{10} 128_{10} 27543_{10}

2.3. Met tien vingers is het mogelijk om van 0 t/m 1023 te tellen door binaire codering te gebruiken. Een 0 is dan een gebogen vinger en een 1 is een gestrekte vinger. Waarom levert het getal 132_{10} toch problemen op in het dagelijks gebruik?

2.4. Zet om van hexadecimaal naar decimaal:

$3FF_{16}$ $4D52_{16}$ $CAFE_{16}$

2.5. Zet om van decimaal naar hexadecimaal:

255_{10} 57005_{10} 32768_{10}

2.6. Het nieuwe internet protocol IPv6 gebruikt 128 bits om computers een uniek IP-adres te geven. Hoeveel unieke adressen zijn mogelijk met IPv6? Voor meer informatie over IPv6 zie [1].

2.7. Hoeveel unieke IPv6-adressen zijn er mogelijk per vierkante meter aardoppervlakte?

2.8. Zet de volgende binaire breuken om in decimale breuken:

$0{,}1001_2$ $0{,}11111111_2$

2.9. Zet de volgende decimale breuken om in een binaire breuken:

$0{,}75_{10}$ $0{,}3_{10}$ $0{,}8_{10}$

2.10. Zet de volgende getallen om:

$11{,}7_{10} \rightarrow$ hex $\quad$ $1F{,}3C_{16} \rightarrow$ dec $\quad$ $3F{,}EA_{16} \rightarrow$ bin $\quad$ $110110{,}110111_2 \rightarrow$ hex

2.11. Hoewel zelden toegepast, komt in de praktijk ook het octale of achttallig talstelsel voor. Hierbij worden alleen de cijfers 0 t/m 7 gebruikt en elk cijfer wordt weergegeven met precies drie bits. Wat is de decimale waarde van 377_8 en 127_8? Wat is de binaire waarde van deze twee octale getallen?

2.12. In een bepaald talstelsel blijkt het getal 41 gelijk te zijn aan het kwadraat van 5. In welk talstelsel staan deze getallen geschreven?

2.13. Op de planeet Mars is een getal ontdekt in een onbekend talstelsel. Na veel puzzelen komen de onderzoekers er achter dat het Martiaanse getal 234 overeenkomt met aardse getal 123_{10}. In welk talstelsel zijn de marsiaanse getallen genoteerd?

2.14. Een ontwerper van een digitale schakeling heeft een getal opgeslagen in een 8-bits variabele. Bij het aanzetten van de schakeling wordt de variabele gereset (op 0 gezet). Wat is de waarde van deze variabele als hij 653 maal met 1 is verhoogd?

* **2.15.** Geef een algemene uitdrukking voor de verzameling getallen die opgeslagen in n bits het getal M oplevert met $0 \leq M \leq 2^n - 1$.

2.16. Hoeveel bits zijn er minimaal nodig om een 8-cijferig decimaal getal op te slaan?

2.17. Bepaal voor elk van de decimale getallen 365, 1024, 7987 en 176890 het aantal bits dat minimaal nodig is om de getallen binair weer te geven.

2.18. Hoeveel procent van de codecombinaties van een twee-cijferig BCD-getal kan nuttig worden gebruikt? En bij drie-cijferig?

* **2.19.** Ontwerp een schakeling die een (zuiver) 3-bits binaire code omzet in een 3-bits Gray-code. Hint: EXOR rules.

* **2.20.** Ontwerp een schakeling die een 3-bits Gray-code omzet in een (zuiver) 3-bits binaire code. EXOR rules again?

2.21. Laat zien dat het eenvoudig is om in de ASCII-code hoofdletters te veranderen in kleine letters (en andersom).

2.22. Met een 7-segment decoder kunnen de cijfers 0 t/m 9 worden gedecodeerd voor aansturing van een 7-segment display zoals te zien is in figuur P2.1. Daarvoor hebben we vier bits nodig. We kunnen hiermee 16 combinaties maken, waarvan we alleen de combinaties 0000 t/m 1001 gebruiken. De zes niet gebruikte combinaties kunnen toch zinvol worden gebruikt voor het afbeelden van A t/m F zodat ook hexadecimale cijfers kunnen worden afgebeeld. Hoe zouden die letters er op een 7-segment display uitzien?

Figuur P2.1: *Alle decimale cijfers op een 7-segment display.*

3

3.1. Gegeven dat $a = 1$, $b = 0$ en $c = 0$. Werk de volgende functies uit. De waarde van d is niet gegeven.
$s = (a + b) \cdot (\overline{b} + c)$ $\quad s = \overline{(\overline{a + b}) \cdot (\overline{c + d})}$ $\quad s = \overline{b} \cdot (c + d \cdot (c + a))$

3.2. Toon aan met behulp van de schakelalgebra dat de absorptiewet $a + a \cdot b = a$ klopt.

3.3. Toon aan met behulp van de schakelalgebra en met behulp van waarheidstabellen dat de consensuswet $(a + b) \cdot (\overline{a} + c) \cdot (b + c) = (a + b) \cdot (\overline{a} + c)$ klopt.

3.4. Gegeven de functie: $s = x \cdot (y + z \cdot \overline{y}) + z \cdot y$
Bepaal de mintermvorm van deze functie. Bepaal de waarheidstabel van deze functie.

3.5. Schrijf de functie $s_{x,y,z} = \sum m(1,3,4,7)$ uit als een som van mintermen.

3.6. Schrijf de functie $s_{x,y,z} = \prod M(2,6,7)$ uit als een product van maxtermen.

3.7. Schrijf de functie $s_{x,y,z} = \sum m(1,2,3,6)$ uit als een product van maxtermen.

3.8. Schrijf de functie $s_{x,y,z} = \prod M(0,1,3,5)$ uit als een som van mintermen.

3.9. Schrijf elk van de functies, gedefinieerd in tabel P3.1, als een som van mintermen (indien mogelijk) en in de somnotatie.

3.10. Schrijf elk van de functies, gedefinieerd in tabel P3.1, als een product van maxtermen (indien mogelijk) en in de productnotatie.

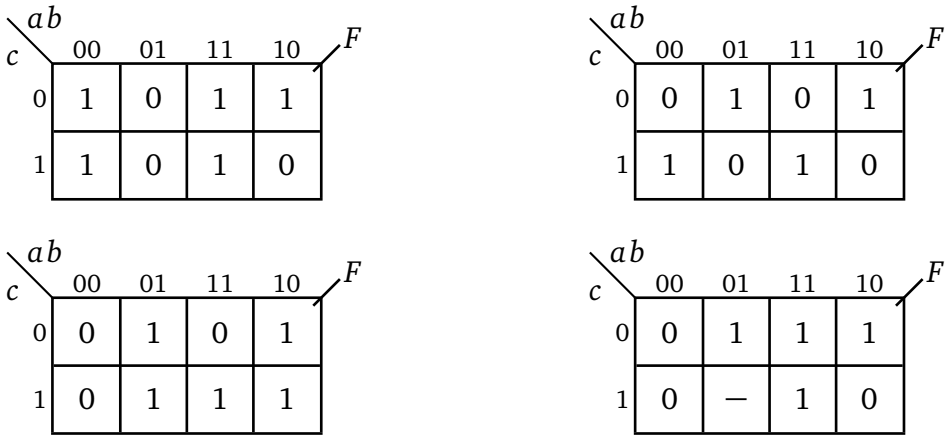
3.11. Gegeven een functie van drie variabelen waarvoor geldt dat de functie voor $xyz = 000$ en $xyz = 111$ don't care is en dat de functie een logische 1 geeft als $x = 0$ terwijl $z = 1$, anders is de functie logisch 0. Geef de waarheidstabel.

Tabel P3.1: *Waarheidstabel bij opgave 3.9 en 3.10.*

x	y	z	S_1	S_2	S_3	S_4	S_5	S_6	S_7
0	0	0	0	1	0	0	0	1	1
0	0	1	1	1	0	1	1	0	–
0	1	0	0	1	0	1	1	1	1
0	1	1	1	0	0	1	1	0	–
1	0	0	0	1	0	0	1	1	1
1	0	1	1	0	0	1	1	0	0
1	1	0	1	0	0	1	1	1	–
1	1	1	1	1	1	1	1	0	0

4

4.1. Minimaliseer de functies gegeven in de Karnaughdiagrammen in figuur P4.1:



Figuur P4.1: *Karnaughdiagrammen met drie variabelen.*

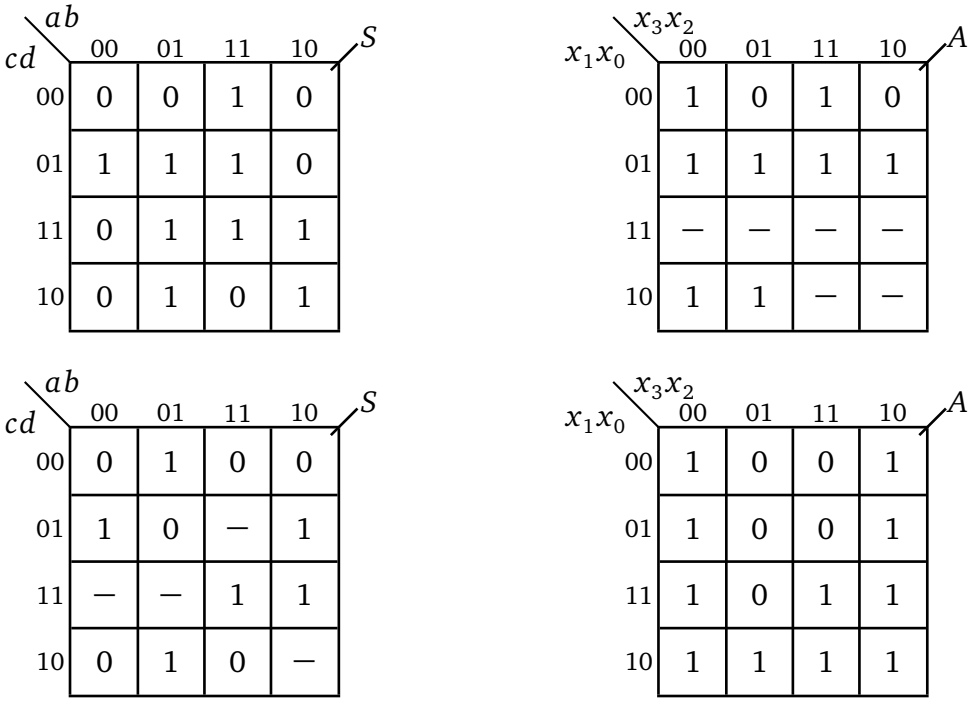
4.2. Gegeven de functies:

$$\begin{aligned} f_{x,y,z} &= \sum m(0,1,2,6) \\ f_{a,b,c} &= \sum m(1,2,5,6,7) \\ f_{s_2,s_1,s_0} &= \sum m(0,3,6) + d(1,2) \end{aligned} \tag{P4.1}$$

Stel de Karnaughdiagrammen op en geef de geminimaliseerde functies. Teken de bijbehorende schakelingen met NOT, AND en OR.

4.3. Ontwerp een *majority gate*. Dit is een schakeling met vier ingangen en één uitgang. De uitgang is 1 als de meerderheid van de ingangen 1 is, anders is de uitgang 0.

4.4. Minimaliseer de functies gegeven in de Karnaughdiagrammen in figuur P4.2:



Figuur P4.2: *Karnaughdiagrammen met vier variabelen.*

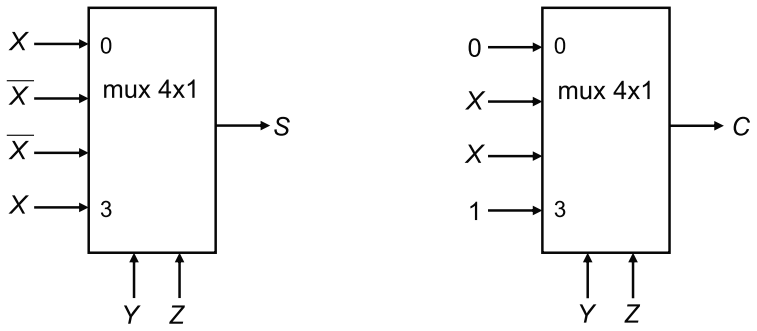
4.5. Gegeven de functies:

$$\begin{aligned} f_{w,x,y,z} &= \sum m(1,2,3,5,7,10,11,12,15) \\ f_{a,b,c,d} &= \sum m(6,7,8,12,14,15) \\ f_{s_3,s_2,s_1,s_0} &= \sum m(4,5,9,10,12) + d(7,8,11,15) \\ f_{x_3,x_2,x_1,x_0} &= \prod M(1,2,3,5,7,10) + D(11,12,15) \end{aligned} \tag{P4.2}$$

Minimaliseer met behulp van Karnaughdiagrammen naar een SOP-vorm.

* **4.6.** Gegeven de functies in opgave 4.5. Minimaliseer de functies met behulp van de Quine-McCluskey-methode naar een SOP-vorm.

4.7. Gegeven twee 4x1 multiplexers in figuur P4.3. Deze worden aangesloten volgens onderstaand schema. Bepaal de waarheidstabellen van S en C.

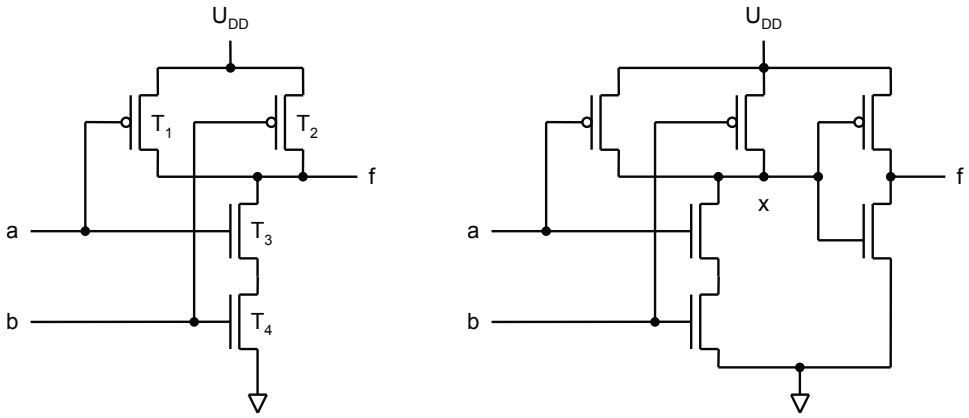


Figuur P4.3: *Logische functies met 4x1 multiplexers.*

4.8. In figuur P4.4 zijn de afbeeldingen van de cijfers op een zevensegmentsdisplay te zien, samen met de namen van de segmenten. Rood betekent dat een segment brandt, grijs betekent dat een segment niet brandt. Een segment brandt als een logische 1 gegeven wordt. Ontwerp een schakeling voor dit display.

Figuur P4.4: *Alle decimale cijfers op een 7-segment display, met segmentaanduiding.*

4.9. In figuur P4.5 zijn twee schakelingen te zien met MOS-transistoren. Bepaal van elk van de schakelingen de waarheidstabel en de logische functie.



(a) *Schakeling met MOS-transistoren.* **(b)** *Schakeling met MOS-transistoren.*

Figuur P4.5: *Twee schakelingen met MOS-transistoren.*

4.10. Ontwerp een NOR-poort met twee ingangen met (C)MOS-transistoren.

4.11. Van een groene led is gegeven dat de werkspanning 2,2 V is bij een werkstroom van 10 mA. De led wordt in combinatie met een serieweerstand aangesloten op een uitgang. De hoge uitgangsspanning is 4,7 V, de lage uitgangsspanning is 0,4 V. Het voedingsspanning van het ic waar de uitgang in zit is 5,0 V. Bereken de waarde van de serieweerstand als de led actief hoog wordt aangesloten. Bereken de waarde van de serieweerstand als de led actief laag wordt aangesloten.

5

5.1. Tel de volgende unsigned binaire getallen op:

$1011001 + 0111011 \qquad 01010 + 01010 \qquad 01111 + 00001$

* **5.2.** Voer de onderstaande unsigned aftrekkingen uit:

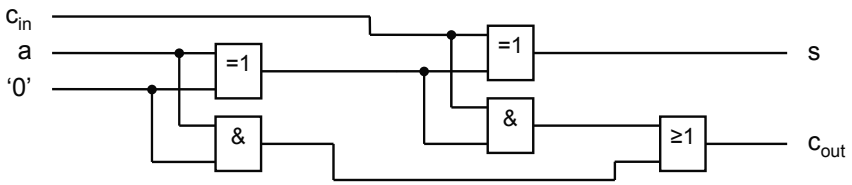
$1011001 - 0111011 \qquad 110000 - 010011 \qquad 010110 - 100101$

5.3. Toon aan dat: $c_{out} = \overline{c_{in}} \cdot (a \cdot b) + c_{in} \cdot (a + b) = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$

5.4. Toon aan dat: $c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in} = a \cdot b + c_{in} \cdot (a \oplus b)$

5.5. Als de functies van s en c_{out} vanuit de nullen zouden worden gemaakt, wordt de functie dan kleiner (minder poorten)?

5.6. In figuur P5.1 is de full adder die eerder is besproken nog eens afgebeeld, maar nu is de b-ingang aan een logische 0 gekoppeld. Vereenvoudig het schema ("minimaliseer b weg"). Doe hetzelfde voor b is logisch 1.



Figuur P5.1: *Full adder met ingang b aangesloten op een logische 0.*

5.7. Ontwerp een 2x2-bits unsigned vermenigvuldiger. Stel de waarheidstabel op en leid de schakelfuncties af.

5.8. Ontwerp een schakeling die test of twee unsigned 4-bits getallen gelijk zijn.

5.9. Ontwerp een 4x4 bits carry save multiplier (tip: uiteraard heeft iemand dat allang gedaan).

5.10. Hoeveel optellers zijn er nodig voor een 5x3-bits vermenigvuldiger? Hoe breed zijn de optellers?

5.11. Reken om van decimaal naar 8-bits two's complement:

$+5, \quad -5, \quad -100, \quad -64, \quad +25, \quad -25$

5.12. Reken uit in two's complement en let daarbij op overflow. Gebruik tekenuitbreiding om de getallen uit evenveel bits te laten bestaan.

$011001 + 100100$	$110011 + 100011$
$011001 - 001110$	$101010 - 010101$
$1101 + 111001$	$101011 - 1100111$
$011100 - 011001$	$10001 + 1100111$

5.13. Geef het decimale equivalent van de volgende hexadecimale two's complement getallen:

$FFFF_{16} \quad 53BA_{16} \quad CB_{16} \quad 7EA3_{16}$

5.14. Wat is de kleinste decimale waarde en de grootste decimale waarde van een hexadecimaal two's complement getal van 8 cijfers?

5.15. Schrijf als hexadecimale two's complement getallen met vier cijfers. Maak gebruik van tekenuitbreiding.

$F_{16} \quad 6_{16} \quad 7A_{16} \quad CB_{16} \quad 35B_{16} \quad D73_{16}$

* **5.16.** Negatieve getallen in BCD-formaat worden weergegeven in ten's complement. Dit is rekenkundig te doen door het BCD-getal af te trekken van 9...9 en daarna er 1 bij op te tellen (uiteraard met een BCD-opteller). Dit aftrekken van 9...9 is het zogenoemde nine's complement. Het voordeel van hiervan is dat per kolom niet geleend hoeft te worden. Ontwerp een digitale schakeling waarin een aangeboden BCD-cijfer wordt afgetrokken van 9. Stel een waarheidstabel op, minimaliseer de functies m.b.v. Karnaughdiagrammen en teken de schakeling van de functies met poorten naar eigen keuze.

5.17. Een nadeel van two's complement is dat het bereik asymmetrisch is, bijvoorbeeld -8 t/m $+7$. Is het mogelijk om met een 4-bits FA toch $+8$ op te tellen bij een getal? Motiveer het antwoord.

5.18. Tel de volgende BCD-gecodeerde getallen bij elkaar op:

$10011001 + 00111001 \quad 0011 + 0111 \quad 010110000110 + 000001110001$

5.19. Het probleem van BCD-getallen is dat er per BCD-cijfer zes binaire codecombinaties niet gebruikt worden. De bekende excess-3-code is een methode om snellere BCD-optellers te maken. Van elk BCD-cijfer kan het excess-3-cijfer worden gemaakt door bij het BCD-cijfer 3 op tellen dus 4 (0100) wordt dan 7 (0111). Ontwerp een digitale schakeling voor de BCD-to-excess-3 encoder. Wat gebeurt er als alle bits van de (geldige) excess-3-code-cijfer worden geïnverteerd?

* **5.20.** Een ouderwetse auto doet mee aan een achteruitrijrace. De kilometerteller staat

op 0. De auto rijdt 123 kilometer achteruit. Wat is de kilometerstand als de kilometerteller vijf decimale cijfers heeft? Ga ervan uit dat de kilometerteller terug telt.

* **5.21.** Een 32-bits single format floating point getal heeft een mantisse van 23 bits. Toon aan dat de decimale variant van de mantisse met ongeveer 7 significante cijfers kunnen worden weergegeven.

5.22. De onderstaande getallen staan in two's çomplement. Bepaal de waarde van de N-, V-, Z- en C-flags na uitvoeren van de optellingen en aftrekkingen.

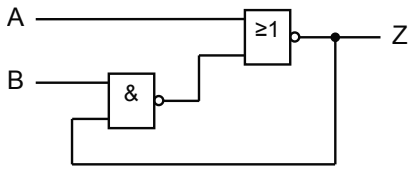
a) $10011001 + 11100100$
b) $10011001 - 00001110$
c) $11111101 + 11111001$
d) $00011100 - 00011001$
e) $01110011 + 11100011$
f) $01101010 - 00010101$
g) $11101011 - 11100111$
h) $11110001 + 11100111$

* **5.23.** Toon aan dat het optellen van twee two's complement getallen met gelijk teken overflow kan optreden. Toon tevens aan dat het optellen van twee two's complement getallen met verschillend teken nooit overflow kan optreden. Doe dit voor het algemene geval van n-bits getallen.

6

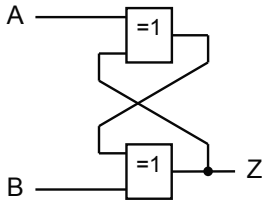
6.1. De SR-latches met overheersende set en reset zijn reeds behandeld. De don't cares waren dan ingevuld als respectievelijk twee enen en twee nullen. Het zijn twee don't cares, dus er zijn vier combinaties mogelijk. Ontwerp SR-latches voor de twee overgebleven combinaties. Zijn de ontwerpen zinvol?

6.2. Gegeven is de schakeling in figuur P6.1. Realiseert deze schakeling een goedwerkend geheugenelement? Motiveer het antwoord.



Figuur P6.1: *Een poging om een latch te realiseren.*

6.3. Gegeven is de schakeling in figuur P6.2. Realiseert deze schakeling een goedwerkend geheugenelement? Motiveer het antwoord.



Figuur P6.2: *Een poging om een latch te realiseren.*

6.4. Is het mogelijk om een SR-latch te maken met de onderstaande SR-combinaties? Motiveer het antwoord.

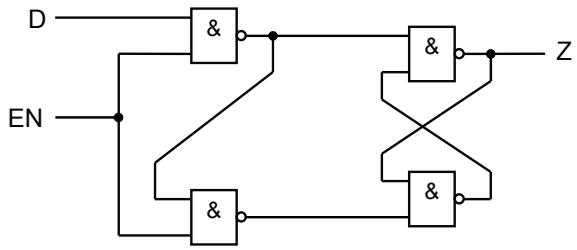
$SR = 11$ onthouden
$SR = 10$ niet gebruikt

$SR = 01$ reset
$SR = 00$ set

6.5. Eerder is het signaaldiagram van de SR-latch met overheersende set besproken. Vul hetzelfde signaaldiagram in voor een SR-latch met overheersende reset.

6.6. In de symbolen van de SR-latches komt de uitgang $\overline{Z}$ voor. Dat suggereert dat dit de inverse is van Z. Geldt dat voor elke combinatie van S en R?

6.7. De schakeling in figuur P6.3 lijkt op een gated D-latch. Is dit een correct werkende gated D-latch? Motiveer het antwoord.



Figuur P6.3: *Een gated D-latch.*

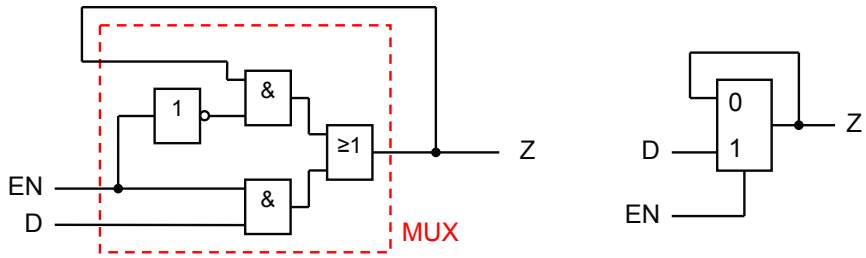
6.8. Een *tweedeler* is een schakeling die op commando van een enable-puls geacht wordt eenmaal van uitgangswaarde te veranderen. Als de uitgang 1 is wordt deze 0 en omgekeerd. Is een tweedeler te maken met één latch en poorten? En met twee latches en poorten? Motiveer het antwoord.

* **6.9.** Voor een master-slave flipflop moet gelden dat:

$$t_{P(max)}(\text{NOT}) + t_h(\text{Z-latch}) < t_{P(min)}(\text{Y-latch})$$

Toon deze voorwaarde aan met een timingdiagram en verklaar dit.

6.10. Een gated D-latch voldoet aan de functie $Z_{nieuw} = \overline{EN} \cdot Z_{oud} + EN \cdot D$ en kan worden gebouwd met behulp van een *multiplexer*, zie figuur P6.4.



Figuur P6.4: *D-latch op basis van een multiplexer.*

Toon aan dat dataoverdracht *niet* betrouwbaar verloopt. Hint: bekijk de situatie $Z = 1$, $D = 1$ en EN gaat van 1→0.

6.11. Probeer een double edge-triggered D-flipflop te ontwerpen. Dit is een flipflop die op *beide* flanken reageert.

6.12. Een T-flipflop is een flipflop die van stand (waarde) verandert onder besturing van een stuursignaal T. Zie voor de functie tabel P6.1. Het (steeds weer) veranderen van stand wordt "toggelen" genoemd. Ontwerp deze T-flipflop, er mag uiteraard *niet* geschakeld worden in de kloklijn.

Tabel P6.1: *Functietabel T-flipflop.*

T	Q^{n+1}
0	Q_n
1	$\overline{Q_n}$

* **6.13.** De functie van een JK-flipflop is:

$$Q^{n+1} = J^n \cdot \overline{Q^n} + \overline{K^n} \cdot Q^n$$

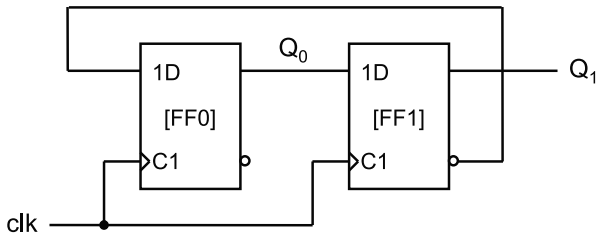
Toon dit aan met behulp van een Karnaughdiagram.

6.14. Van een gated D-latch zijn gegeven t_{su}(EN-to-Z) = 15 ns, t_h(EN-to-Z) = 10 ns. De klok loopt op 5 MHz en heeft een duty cycle van 50%. Bereken de tijd dat D stabiel moet blijven als ervan uitgegaan wordt dat D niet mag veranderen tijdens het transparant zijn van de latch.

6.15. Maak een SR-latch van een edge-triggered D-flipflop met asynchrone preset en clear.

6.16. Eerder is een ontwerp van een schuifregister aan bod gekomen. Deze schakeling schuift (althans op tekening) naar rechts. Ontwerp nu een schuifregister dat zowel rechtsom als linksom kan schuiven (gebruik een stuursignaal).

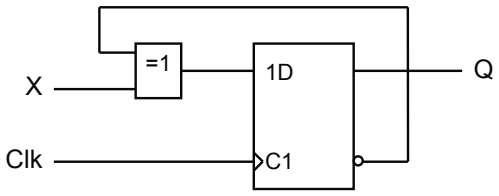
6.17. De schakeling in figuur P6.5 bestaat uit twee flipflops waarbij de uitvoer van de eerste flipflop direct aan de twee flipflop wordt gevoerd terwijl de inverse van de tweede flipflop aan de eerste flipflop wordt gevoerd. We gaan er vanuit dat de beide flipflops geïntialiseerd zijn met 0. Daarna wordt een aantal klokpulsen aangeboden. Geef aan wat de standen zijn die de flipflops doorlopen. Hint: er zijn niet meer dan vier klokpulsen nodig om alle standen van de flipflops te bepalen.



Figuur P6.5: *Een schakeling met twee flipflops.*

6.18. Gegeven is de flipflopschakeling in figuur P6.6. De beginstand van de flipflop is 0.

Op ingang Clk worden 12 klokpulsen aangeboden. Op ingang X wordt het volgende bitpatroon aangeboden: 011101001101 (beginnend met de linker 0), elke (bit-)waarde steeds vlak voor de opgaande flank. Geef bij elke (bit-)waarde van



Figuur P6.6: *Flipflopschakeling.*

ingang X vlak voor de opgaande flank van ingang Clk steeds de (bit-)waarde van uitgang Q vlak na de opgaande flank van ingang Clk. Timing wordt buiten beschouwing gelaten.

7

7.1. Geef de logische functie van onderstaande CSA:

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

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

7.2. Geef de logische functie van onderstaande CSA:

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

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

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

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

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

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

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

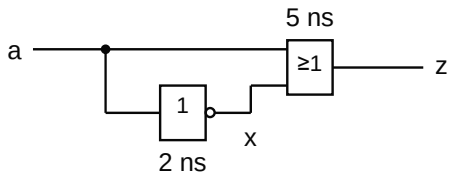
7.8. Geef een VHDL-beschrijving voor een negative edge triggered D-flipflop.

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

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

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

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



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

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

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

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

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

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

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

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

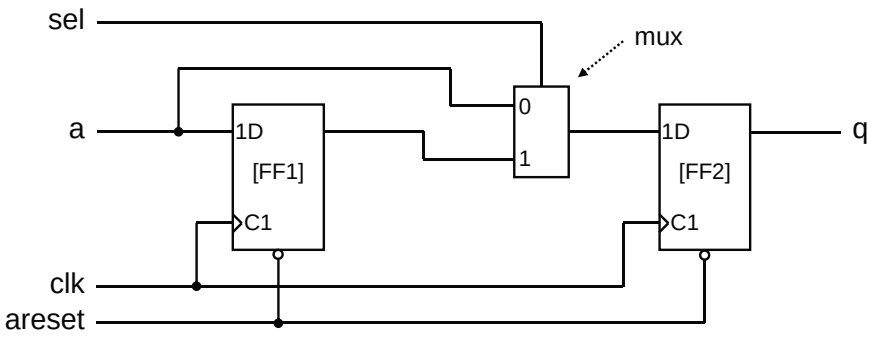
7.16. Geef het schema van de VHDL-code in listing P7.5.

```
1  if rising_edge(clk) then
2      ff1 <= x;
3      if ff1 = '1' and x = '0' then
4          ff2 <= '1';
5      else
6          ff2 <= '0';
7      end if;
8  end if;
```

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

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

7.18. Geef het schema van in de code in listing P7.6. Probeer zo dicht mogelijk de originele code te volgen.



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

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

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

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

8

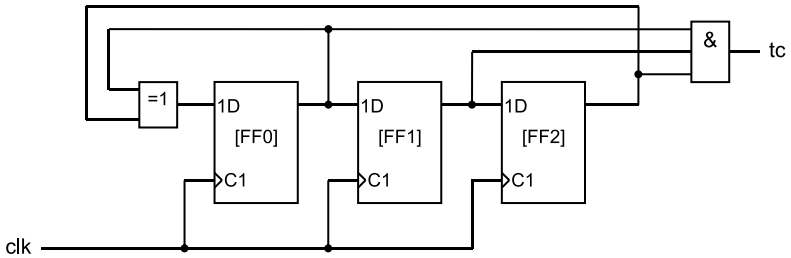
8.1. Ontwerp een 4-bits omhoogteller met JK-flipflops en poorten.

8.2. Ontwerp een 4-bits omlaagteller met T-flipflops en poorten. Ga uit van een 4-bits omhoogteller.

8.3. Ontwerp een 6-teller, een omhoogteller die telt van 0 t/m 5, met losse poorten en T-flipflops.

8.4. Een teller is te ontwerpen met een register en een full-adder. Op één ingang van de full-adder wordt de telstand aangeboden, op de andere ingang wordt de constante 00..01 aangeboden. Implementeer een enable in deze teller zodat de telstand behouden blijft ook al passeert er een klokflank.

8.5. In figuur P8.1 is een schakeling te zijn. De schakeling wordt een *Linear Feedback Shift Register* genoemd. De schakeling lijkt op een schuifregister maar kan ook als een teller worden beschouwd. De reset is achterwege gelaten maar werkt als volgt: als de reset geactiveerd is worden de flipflops geladen met een logische 1. De "teller" doorloopt 7 telstanden. Bepaal de telcyclus.



Figuur P8.1: *Een 3-bits Linear Feedback Shift Register.*

8.6. Bepaal de resolutie van de Duty Cycle voor een *n*-bits digitaal PWM-systeem.

8.7. Het PWM-signaal is hoog als de referentiewaarde groter is dan de tellerwaarde (op enig moment). Daardoor is een DC van 100% niet haalbaar. Een student past

de schakeling aan zodat het PWM-signaal is hoog als de referentiewaarde groter is dan of gelijk is aan de tellerwaarde (op enig moment). Is dit een slimme keuze? Zijn er andere problemen te verwachten? Motiveer het antwoord.

8.8. Bedenk een manier om tot een DC van 100% te komen zonder de vergelijkschakeling aan te passen.

8.9. Ontwerp een VHDL-beschrijving van een 4-bits omhoogteller die ook een begintelstand kan laden.

8.10. Gegeven een deel van de VHDL-beschrijving van de BCD-teller in listing P8.1. Wat doet de teller als deze per ongeluk in stand 12 terecht is gekomen?

```
1   process (clk, areset) is
2   begin
3       if areset = '1' then
4           count <= "0000";
5       elsif rising_edge(clk) then
6           if en = '1' then
7               if count = "1001" then
8                   count <= "0000";
9               else
10                  count <= count + 1;
11              end if;
12          end if;
13      end if;
14  end process;
15
16  tc <= '1' when count = "1001" and en = '1' else '0';
17  q <= count;
```

Listing P8.1: *De VHDL-beschrijving van een 1-decade BCD-teller.*

8.11. Pas de VHDL-code in listing P8.1 zo aan dat de teller alleen (verder) telt als de teller zich in telstand 0 t/m 9 bevindt. Denk ook aan de terminal count.

8.12. Pas de code in listing P8.1 aan zodat de teller een 13-teller wordt.

8.13. Voor het detecteren van telstand 9 zijn in principe alleen telbit 3 en 0 nodig (Q_3 en Q_0). De andere twee zijn don't care. Helaas werkt de constructie

```
when count = "1--1"
```

niet. Pas de code aan zodat voor de test alleen naar de waarden Q_3 en Q_0 gekeken wordt.

8.14. Ontwerp een urenteller in VHDL. De urenteller wordt voor zowel de Amerikaanse als Europese markt ontwikkeld. De urenteller moet aan de volgende eisen voldoen:

- De urenteller heeft een asynchrone reset, actief hoog;
- De urenteller kan tellen in 12-uur-modus of 24-uur-modus middels het signaal `ampm`.
- Als signaal `ampm` logisch '1' is, telt de teller cyclisch van 1 t/m 12;

- Als signaal `ampm` logisch '0' is, telt de teller cyclisch van 0 t/m 23;
- De teller gaat naar de volgende telstand als signaal `up` logisch '1' is;
- Het uur (de tellerwaarde) is beschikbaar als 5-bits unsigned vector.

In listing P8.2 zijn de entity met port-beschrijving en de (interne) teller al ingevuld. Geef de VHDL-code van de architecture van deze urenteller.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity opgave5 is
6      port (clk : in std_logic;                -- The clock
7            areset : in std_logic;             -- The reset
8            up : in std_logic;                 -- Count up
9            ampm : in std_logic;               -- 24 or 12 hour clock
10           hour : out unsigned (4 downto 0)   -- Hour counter
11     );
12 end entity opgave5;
13
14 architecture rtl of opgave5 is
15 signal counter : unsigned(4 downto 0);     -- Internal hour counter
16 begin
17
18     -- Deze code moet ontwikkeld worden
19
20 end architecture rtl;
```

Listing P8.2: *De entity-beschrijving van een 12/24-uursklok.*

8.15. Ontwerp een PWM-generator in VHDL. De PWM-generator moet aan de volgende eisen voldoen:

- De PWM-generator heeft een asynchrone reset, actief hoog.
- De Duty Cycle (DC) kan in stappen van 0,5% worden ingesteld.
- De PWM-generator heeft een referentie-ingang genaamd `ref` waarmee een 8-bits waarde geladen kan worden in het interne referentieregister.
- De referentiewaarde moet steeds aan het begin van een PWM-cyclus worden geladen.
- De PWM-generator heeft een uitgang `pwm_out` die '1' is als de interne referentiewaarde hoger is dat de interne telstand, anders is deze uitgang '0'.
- De PWM-generator heeft een uitgang `top` die '1' is als de interne telstand maximaal is, anders is deze uitgang '0'.

8.16. Een ontwerper heeft de onderstaande code geschreven, zie listing P8.3. Het betreft hier een 1-decade BCD-teller. Merk op dat de beschrijving van de terminal count verweven is met de teller. Leg uit waarom de terminal count *niet* correct functioneert.

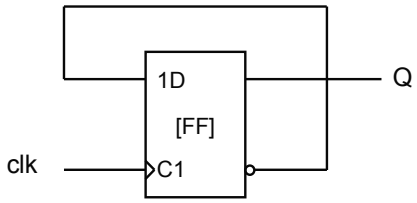
```
1   process (clk, areset) is
2   begin
3       if areset = '1' then
4           count <= "0000";
5       elsif rising_edge(clk) then
6           if en = '1' then
7               if count = "1001" then
8                   count <= "0000";
9                   tc <= '1';
10              else
11                  count <= count + 1;
12                  tc <= '0';
13              end if;
14          end if;
15      end if;
16  end process;
17
18  q <= count;
```

Listing P8.3: *Een deel van de beschrijving van de 1-decade BCD-teller.*

9

9.1. Van de tweedeler in figuur P9.1 zijn de volgende timingparameters gegeven.

$$t_{su}(\text{FF}) = 15 \text{ ns}$$
$$t_h(\text{FF}) = 5 \text{ ns}$$
$$t_{P(min)}(\text{FF}) = 10 \text{ ns}$$
$$t_{P(max)}(\text{FF}) = 35 \text{ ns}$$



Figuur P9.1: *Timingparameters en schema van een tweedeler.*

Bepaal de maximale frequentie waarop dit systeem kan werken. Is betrouwbare dataoverdracht mogelijk?

9.2. Gegeven is een flipflop met enable in figuur P9.2. Bepaal de setup- en holdtijd voor signaal D t.o.v. van de actieve klokflank. Bepaal de maximale frequentie waarop dit systeem betrouwbare kan werken. Teken tijddiagrammen waarmee de berekeningen kunnen worden gemaakt.

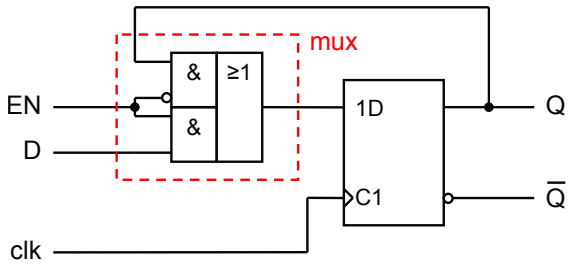
9.3. Gegeven is het schema in figuur P9.3. Het kloksignaal van FF2 is iets vertraagd t.o.v. de klok van FF1. Druk de maximale frequentie uit in de timingparameters van de flipflops en de clock skew (delay). Analyseer de logische werking van dit systeem.

9.4. Gegeven is het schema in figuur P9.4. Het laat dataoverdracht tussen twee flipflops van verschillende ic's zien.

De timing van de flipflops is verschillend:

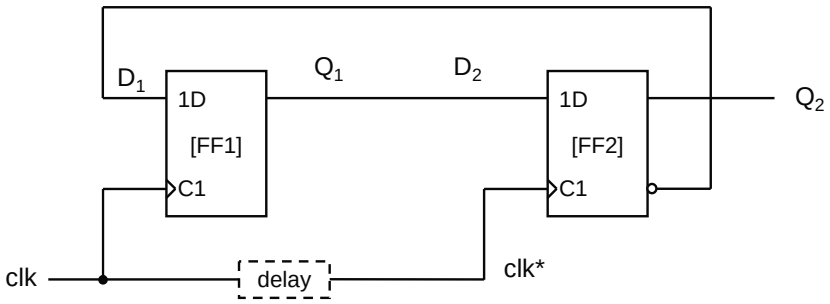
FF1: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 10/3/7/12$ ns

FF2: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 20/6/10/25$ ns

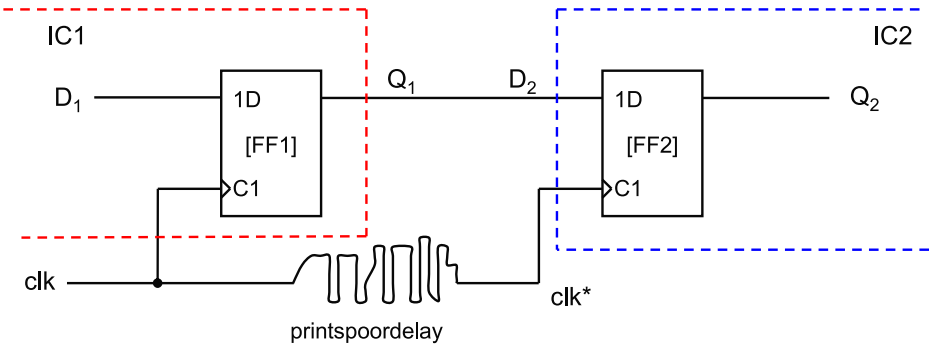


$t_{su}(\text{FF}) = 14$ ns
$t_h(\text{FF}) = 4$ ns
$t_{P(min)}(\text{FF}) = 7$ ns
$t_{P(max)}(\text{FF}) = 17$ ns
$t_{P(min)}(\text{logic}) = 8$ ns
$t_{P(max)}(\text{logic}) = 19$ ns

Figuur P9.2: *Schema en timing van een D-flipflop met enable.*



Figuur P9.3: *Twee flipflops met verbindingen en klokskew.*



Figuur P9.4: *Verbinding tussen verschillende ic's.*

De skewtijd bedraagt 2 ns. Is betrouwbare dataoverdracht mogelijk?

9.5. Gegeven is de onderstaande 4-bits teller.

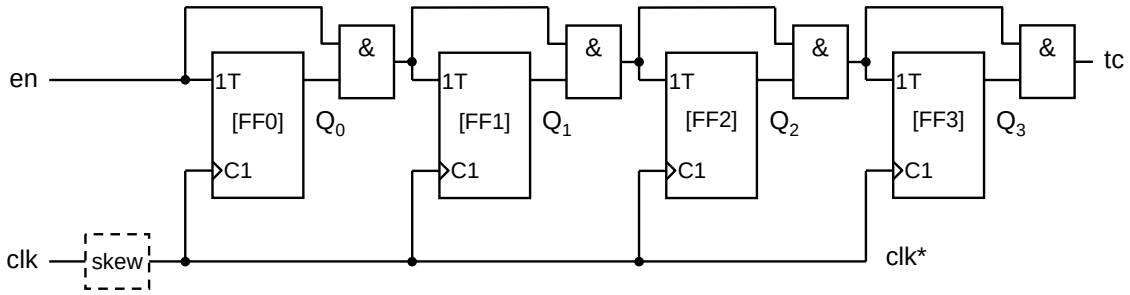
Van de componenten zijn de volgende timingparameters gegeven:

Flipflops: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 2/1/4/7$ ns

AND: $t_{P(min)}/t_{P(max)} = 3/5$ ns

Skew: $t_P = 4$ ns

In dit systeem zijn meerdere paden te ontdekken waarlangs data wordt getrans-



Figuur P9.5: *4-bits teller opgebouwd uit 1-bit tellersecties.*

porteerd.

a) Geef alle paden waarlangs data van flipflop naar flipflop wordt getransporteerd.

Van dit systeem moet de maximale frequentie worden berekend.

b) Teken een tijddiagram met alleen relevante timingparameters waarmee de maximale frequentie berekend moet worden.

c) Bereken de maximale frequentie waarop dit systeem nog betrouwbaar werkt.

Het signaal en (enable) wordt vertraagd aangeboden aan de ingang van de flipflops. Dat heeft invloed op de setup- en holdtijden van signaal en t.o.v. het kloksignaal clk.

d) Teken een tijddiagram met alleen relevante timingparameters waarmee de setuptijd en holdtijd van en (enable) t.o.v. de actieve flank van clk berekend kan worden.

e) Bereken de setuptijd en holdtijd van en (enable) t.o.v. de actieve flank van clk.

De uitgang tc geeft aan of de teller op de maximale stand staat. Na het passeren van de actieve klokflank duurt het even voordat de nieuwe waarde op de uitgang beschikbaar is.

f) Teken een tijddiagram met alleen relevante timingparameters waarmee de minimale en maximale vertragingstijden van uitgang tc t.o.v. de actieve flank van clk berekend kan worden.

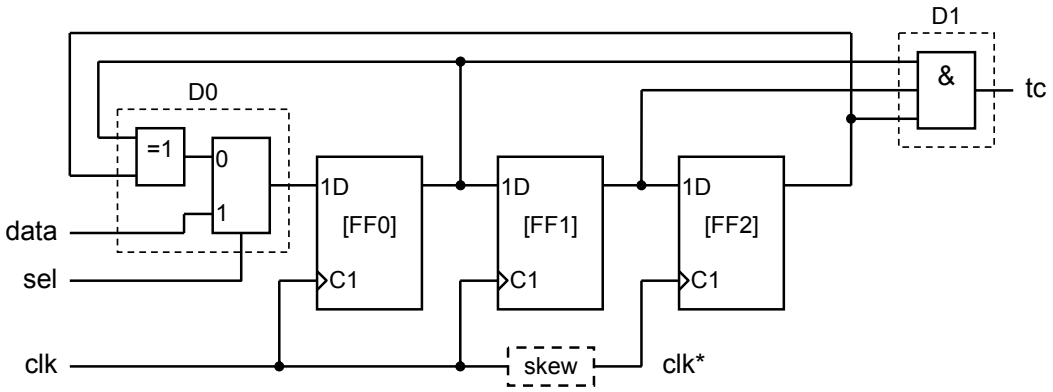
g) Bepaal de minimale en maximale vertragingstijden van de uitgang tc t.o.v. de actieve klokflank.

De opbouw van de teller is zeer elegant en makkelijk uitbreidbaar maar hoe zit het nou met de timing.

h) Beschrijf kort wat de invloed is van het aantal telsecties op de timing, bv. als het aantal secties wordt uitgebreid.

9.6. Gegeven is de schakeling in figuur P9.6. Hierin zijn drie D-flipflops opgenomen (FF0, FF1 en FF2), twee stukken combinatoriek (D0, D1) en één klokvertraging

(skew). D0 is combinatoriek voor de ingang van FF0 en D1 is combinatoriek na de uitgangen van FF0, FF1 en FF2. FF2 krijgt een vertraagd kloksignaal aangeboden (skew).



Figuur P9.6: *Synchrone flipflop-schakeling*

Van de bouwstenen zijn de volgende timingparameters gegeven:

D-flipflops: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 2/1/6/8$ ns

AND: $t_{P(min)}/t_{P(max)} = 2/3$ ns

EXOR: $t_{P(min)}/t_{P(max)} = 3/5$ ns

Mux: $t_{P(min)}/t_{P(max)} = 4/6$ ns

Skew: $t_P = 3$ ns

Van de skew is maar één vertragingstijd gegeven, er wordt geen onderscheid gemaakt tussen de minimale en maximale vertragingstijd. Het betreft hier namelijk clock skew.

Merk op dat er in dit systeem **vier** paden zijn tussen de flipflops waarlangs data wordt getransporteerd.

a) Geef aan welke paden er tussen de flipflops zijn waarlangs data wordt getransporteerd. Geef hierbij duidelijk aan welke stukken combinatorische logica wordt gepasseerd.

Van het systeem moet de maximale frequentie worden berekend. Na onderzoek blijkt het pad FF2 → D0 → FF0 de grootste minimale periodetijd op te leveren.

b) Teken een tijddiagram met alleen relevante timingparameters waarmee de maximale frequentie berekend moet worden.

c) Bereken de maximale frequentie waarop dit systeem nog betrouwbaar werkt.

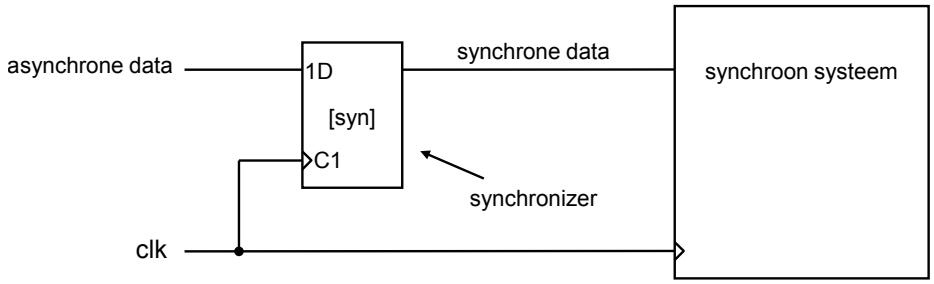
FF2 heeft last van klokskew, het kloksignaal komt vertraagd aan op de klokingang van FF2. Dat heeft invloed op de holdtijd van de dataingang van FF2.

d) Teken een tijddiagram met alleen relevante timingparameters waarmee de *hold slack* van de dataingang van FF2 berekend kan worden.

e) Bereken de *hold slack* van de dataingang van FF2.

10

10.1. Gegeven is het synchrone systeem in figuur P10.1 met een frequentie van 20 MHz. Data komt asynchroon binnen met 100 kHz.



Figuur P10.1: *Een synchronisatiesysteem*

Van de synchronizer 74F50109 is gegeven: $\tau = 0{,}135$ ns, $T_0 = 9{,}8 \cdot 10^6$ s

De setuptijd van het achterliggende systeem is 20 ns. Bereken de MTBF van dit systeem.

10.2. Gegeven is het systeem van in figuur P10.2.

Van flipflops FF1 en FF2 is gegeven $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 5/2/4/10$ ns

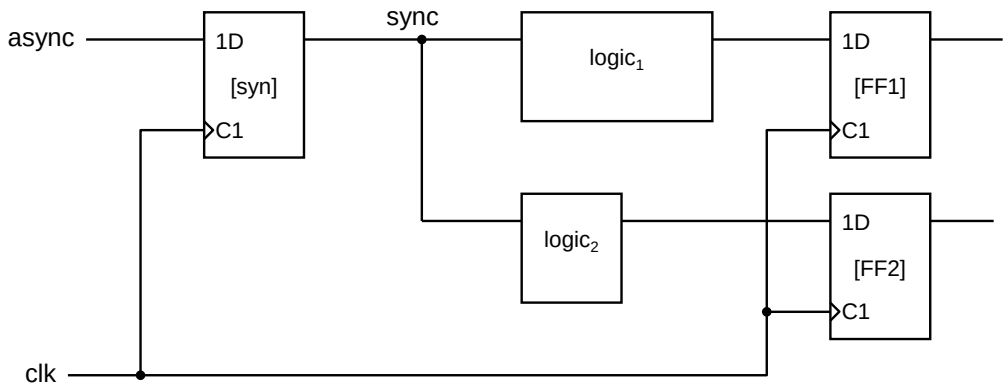
Van de synchronizer is gegeven $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 2/0/3/5$ ns, $\tau = 0{,}135$ ns, $T_0 = 9{,}8 \cdot 10^6$ s.

Van logica$_1$ is gegeven $t_{P(min)}/t_{P(max)} = 10/20$ ns.

Van logica$_2$ is gegeven $t_{P(min)}/t_{P(max)} = 5/12$ ns.

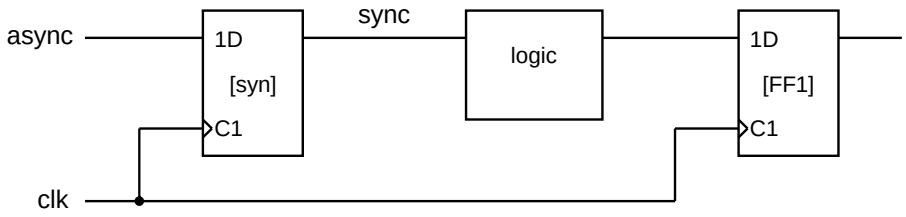
De systeemfrequentie is 25 MHz. De frequentie van de asynchroon binnenkomende data bedraagt 2 kHz.

Bepaal de MTBF van dit systeem.



Figuur P10.2: *Een synchronisatiesysteem*

10.3. Gegeven is de schakeling in figuur P10.3.



Figuur P10.3: *Synchrone schakeling met synchronizer.*

Van flipflop FF1 is gegeven: $t_{su}/t_h/t_{P(min)}/t_{P(max)} = 5/2/4/10$ ns
Van de synchronizer is gegeven: $\tau = 0{,}15$ ns, $T_0 = 9{,}8 \cdot 10^6$ s
Van logica is gegeven: $t_{P(min)}/t_{P(max)} = 7/10$ ns

De systeemfrequentie is 40 MHz. Een ontwerper wil dat bij dit systeem de gemiddelde tijd tussen twee fouten 1 jaar (1 jaar = 365 dagen) bedraagt. Bereken de maximaal toegestane frequentie van het asynchroon binnenkomende signaal om aan de eis van de ontwerper te kunnen voldoen.

11

11.1. Teken het toestandsdiagram van een JK-flipflop.

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

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

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

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

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

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

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

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

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

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

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

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

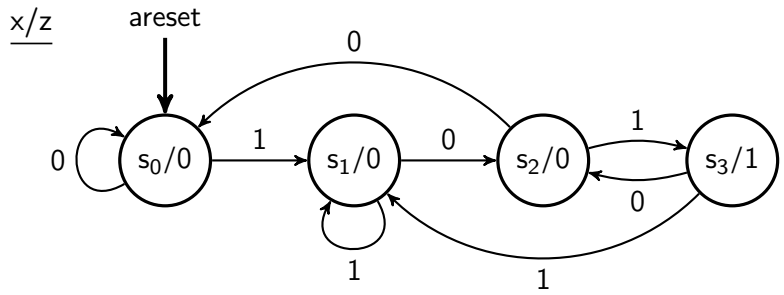
11.14. Ontwerp het toestandsdiagram van een Moore-machine voor het herkennen van het patroon (of reeks) 1001. De machine geeft een 1 af en stopt met herkennen.

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

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

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

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

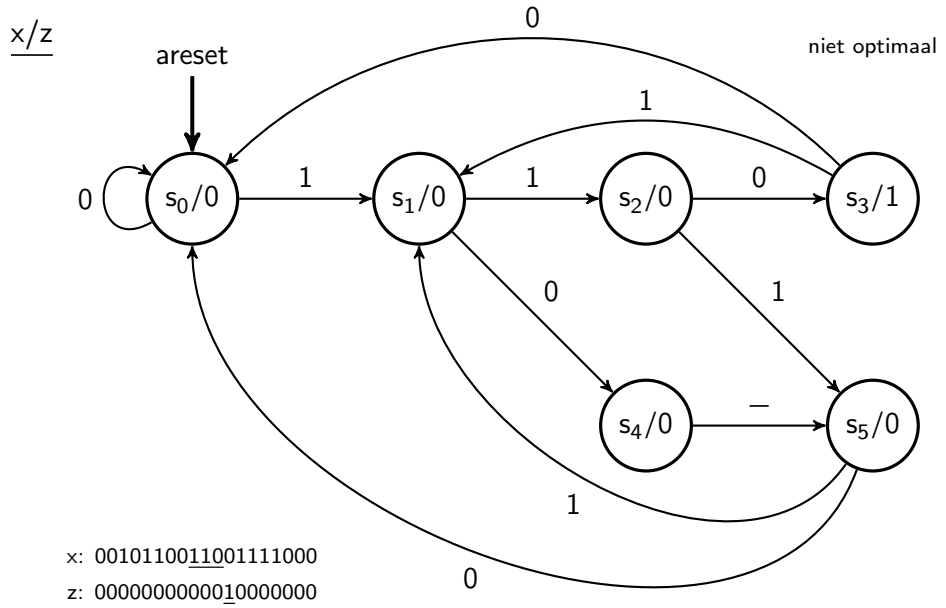


Figuur P11.1: *Herkenner voor het overlappend herkennen van het patroon 101 (Moore).*

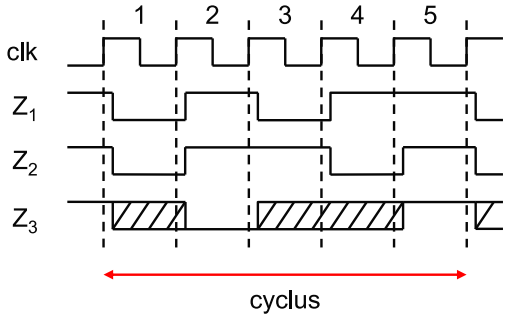
11.19. In figuur P11.2 is het toestandsdiagram van een machine te zien voor het bloksgewijs herkennen van het patroon (of reeks) 110. Een studente beweert dat het toestandsdiagram van deze machine compacter kan worden getekend omdat toestand s_0 en s_5 identiek zijn en samengenomen kunnen worden. Klopt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

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

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

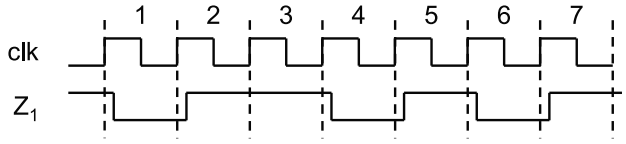


Figuur P11.2: *Herkenner voor het bloksgewijs herkennen van het patroon 110 (Moore).*



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

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



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

11.23. Ontwerp een toestandsdiagram voor een snoepmachine die 35 cent moet herkennen. Toegestane munten zijn 5 cent (S) en 10 cent (D). Bij dit bedrag moet snoepgoed worden geretourneerd. Bij te hoog bedrag moet een signalering geactiveerd worden.

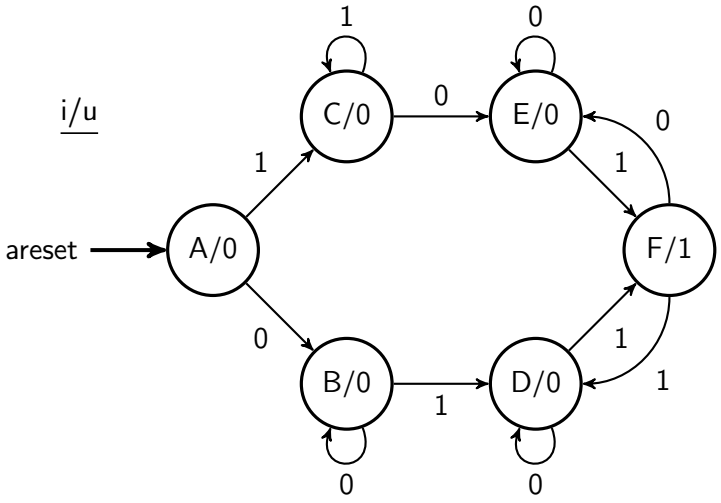
11.24. Wat is het hoogste bedrag dat in de machine in opgave 11.23 kan worden inge-

worpen (dat is niet het bedrag dat de machine uiteindelijk moet herkennen)?

11.25. Hoeveel unieke toestanden heeft machine in opgave 11.23?

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

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



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

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

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

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

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

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

11.29. Geef de VHDL-beschrijving voor het toestandsdiagram in figuur P11.1.

11.30. Geef de VHDL-beschrijving voor het toestandsdiagram in figuur P11.5.

12

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

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

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

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

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

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

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

13

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

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

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

13.4. Geef de instructie (in bits) die ervoor zorgt dat de EXOR van *R*2 en *R*3 in *R*3 terecht komt (snelschrift: **`exor`** `R3,R2`). Het instructieformaat is te zien in figuur P13.1.

15 12	11 8	7 4	3 0
opcode	sub-code	reg_{ds}	reg_{s}

Figuur P13.1: *Formaat van de EXOR-functie.*

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

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

13.7. In het programma in listing P13.1 wordt direct na de **`adc`**-instructies een **`add`**-instructie uitgevoerd. Maar de **`add`**- en **`adc`**-instructies verwerken ook de carry-flag. Moet dan vóór de tweede **`add`**-instructie (regel 11) de carry flag niet gewist worden? Mag de **`add`**-instructie in regel 11 vervangen worden door een **`adc`**-instructie?

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

```
 0:   ldi  r0,13        ; load multiplicand (constant 13)
 1:   ldi  r1,11        ; load multiplier (constant 11)
 2:   ldi  r2,0         ; clear result registers
 3:   ldi  r3,0         ; or use mov R3,R2
 4:   ldi  r4,1         ; loop variable
 5:   ldi  r8,1         ; constant 1
 6:   ldi  r9,0         ; constant 0
 7:   cmp  r4,r1        ; compare R4 with R1 (subtract)
 8:   bhi  @13          ; r4 is higher than r1, so go out of loop
 9:   add  r2,r0        ; add multiplicant (low byte)
10:   adc  r3,r9        ; process carry in high byte
11:   add  r4,r8        ; increment loop counter
12:   bra  @7           ; and again
13:   out  4,R2         ; result to output0
14:   out  5,R3         ; result to output1
15:   bra  @15          ; hold your horses!
```

Listing P13.1: *Programma voor vermenigvuldiging van 11 met 13 (assembler).*

13.9. Stel dat de processor geen **`ldil`**-instructie (load low byte) en **`ldih`**-instructie (load high byte) zou hebben, maar alleen een **`ldi`**-insstructie (load 8-bit constant). Toon aan dat het toch mogelijk is om een 16-bits constante in een register te laden. Benoem voordelen en nadelen.

13.10. Gegeven is de pseudo-code in listing P13.2. Vertaal deze code naar assembler. Gebruik hierbij vergelijk- en spronginstructies.

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

Listing P13.2: *Programma voor een beslissing.*

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

```
a := input0;        // a is an unsigned variable

do
    a := a * 2;
while (a <= 128);
```

Listing P13.3: *Programma voor een herhaling.*

13.12. Gegeven is de wachtlus gebaseerd op de zero flag, te zien in listing P13.4. Verder is gegeven dat de processor loopt op 50 MHz en dat de wachttijd 2/3 seconden bedraagt. Bereken de waarden waarmee de registers geladen moeten worden.

13.13. Herhaal opgave 13.12 maar nu voor een klokfrequentie van 33 MHz en een wachtijd van 0,5 seconden.

13.14. Stel dat de processor geen zero flag heeft. Is het dan toch mogelijk om een wachtlus te implementeren op basis van het programma in listing P13.4? (hint:

```
        ldil r0,<value1>    ; low least byte count
        ldih r0,<value2>    ; low most byte count
        ldil r1,<value3>    ; high least byte count
        ldih r1,<value4>    ; high most byte count

        ldi  r8,1           ; constant 1
        ldi  r9,0           ; constant 0

wait:   sub  r0,r8          ; decrement low word
        sbc  r1,r9          ; decrement high word on carry
        bne  wait           ; repeat if not 0

halt:   bra  halt           ; halt forever
```

Listing P13.4: *Opzet van de wachtlus.*

kijk naar de carry flag, de zero flag kan niet gebruikt worden).

13.15. In listing P13.5 is de pseudo-code te zien voor het berekenen van de deling van unsigned twee getallen. De code voert de deling A/B uit. Na de deling is het quotiënt te vinden in variabele `q` en de rest is te vinden in variabele `a`. Ontwerp een assemblerprogramma voor dit programma.

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

Listing P13.5: *Code voor een deling.*

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

```
    ldi  R1,13
    ldi  R2,45
    exor R1,R2
    exor R2,R1
    exor R1,R2
```

Listing P13.6: *Code "EXOR swap".*

13.17. Ontwerp een programma dat de input eenmaal inleest en het aantal enen bepaalt in de ingelezen waarde. De pseudo-code is gegeven in listing P13.7.

13.18. Het ontwikkelde programma uit opgave 13.17 kent een lus die een aantal klokpulsen duurt. Hoeveel klokpulsen duurt het uitvoeren van de lus?

13.19. De processor kan geen rekenkundige operatie uitvoeren op een register en een constante. Het datapad van de processor en de instruction decoder zijn daar wel voor uitgerust. Ontwerp twee instructies die een constante direct aftrekken van een register, één zonder carry en één met de carry. Zoek geschikte opcodes uit.

```
R3 := 0;                        // result
R1 := 16;                       // counter
R2 := input0;                   // read input once
do                              // loop
    R2 := shr(R2);              // shift right R2, low bit in carry
    R3 := R3 + 0 + carry;       // add carry
    R1 := R1 - 1                // decrement counter
while (R1 <> 0);                // as long as R1 not equal to 0
```

Listing P13.7: *Code voor het tellen van het aantal enen in de input.*

13.20. We hebben in opgave 13.19 gekozen voor het aftrekken van constanten van een register. Is het mogelijk om constanten toch op te tellen bij registers?

13.21. De processor is niet in staat om een vermenigvuldiging in hardware uit te voeren. Gelukkig zijn er nog I/O-adressen beschikbaar die niet worden gebruikt. Ontwerp de VHDL-code voor het realiseren van een hardware-vermenigvuldiger die in de I/O geplaatst moet worden. Ga uit van unsigned getallen. Merk op dat I/O-adressen zijn te benaderen met **`in`**- en **`out`**-instructies dus er zijn geen nieuwe assembler-instructies nodig

13.22. Gegeven zijn twee 128-bits getallen. Het eerste getal is opgeslagen in $R3$, $R2$, $R1$ en $R0$ en het tweede getal is opgeslagen in $R7$, $R6$, $R5$ en $R4$. Geef de benodigde instructies om deze twee getallen op te tellen. Het resultaat komt weer in $R3$, $R2$, $R1$ en $R0$.

13.23. Een 32-bits unsigned getal is opgeslagen in register $R1$ (meest significante word) en register $R0$ (minst significante word). Geef de benodigde instructies om dit 32-bits getal één positie naar links te schuiven.

13.24. Een 32-bits signed getal is opgeslagen in register $R1$ (meest significante word) en register $R0$ (minst significante word). Geef de benodigde instructies om dit 32-bits getal één positie naar rechts te schuiven. Let erop dat de tekenbit behouden moet blijven.

13.25. Onderzoek de mogelijkheid om de registers van 16 naar 32 bits uit te breiden. Welke onderdelen moeten dan worden aangepast? Kunnen alle opcodes hetzelfde blijven? Moeten de instructies ook naar 32 bits worden uitgebreid? Benoem voordelen en nadelen.

A

Uitwerkingen

A.1 Uitwerkingen hoofdstuk 1

Uitwerking opgave 1.1.
Eerst maar eens een inventarisatie van de signalen. Een ingangssignaal moet óf onveranderd worden doorgelaten óf geïnverteerd worden. Er is een signaal nodig dat de data kan aanleveren. De keuze tussen doorlaten of inverteren moet met een tweede signaal gedaan worden, een zogenaamd *stuur*signaal. Natuurlijk is er één uitgang. Het datasignaal noemen we a, het stuursignaal noemen we m (*mode*). De uitgang noemen we f.

We maken een keuze in de besturing: signaal a wordt onveranderd doorgelaten als $m = 0$ en geïnverteerd als $m = 1$. De functie is als volgt te schrijven:

$$f = \begin{cases} a & \text{als } m = 0 \\ \overline{a} & \text{als } m = 1 \end{cases} \tag{A.1}$$

De waarheidstabel en de functie zijn eenvoudig op te stellen en af te leiden.

Tabel A.1: *Waarheidtabel voor functie f*

m	a	f	
0	0	0	a doorgeven
0	1	1	”
1	0	1	a inverteren
1	1	0	”

De functie levert twee enen die niet te combineren zijn:

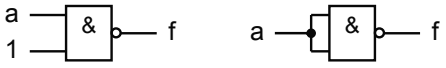
$$\begin{aligned} s_{m,a} &= \overline{a} \cdot m + a \cdot \overline{m} \\ &= a \oplus m \end{aligned} \tag{A.2}$$

De omschakelbare buffer/NOT is de bekende EXOR-functie!

Uitwerking opgave 1.2.

Een NAND kan je op twee manieren schakelen als een NOT (of inverter). De functie van de NAND is $f = \overline{x \cdot y}$. Een mogelijkheid is de signalen x en y met elkaar te verbinden (we noemen het signaal nu a). Een andere mogelijkheid is om één ingang aan een logische 1 te verbinden (de andere ingang noemen we nu ook a). Aldus:

$$x = a, y = a \quad \rightarrow \quad f = \overline{a \cdot a} = \overline{a}$$
$$x = a, y = 1 \quad \rightarrow \quad f = \overline{a \cdot 1} = \overline{a}$$

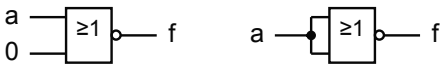


Figuur A.1: *Functies en aansluitingen van een NAND geschakeld als NOT.*

Uitwerking opgave 1.3.

Een NOR kan je op twee manieren schakelen als een NOT (of inverter). De functie van de NOR is $f = \overline{x + y}$. Een mogelijkheid is de signalen x en y met elkaar te verbinden (we noemen het signaal nu a). Een andere mogelijkheid is om één ingang aan een logische 0 te verbinden (de andere ingang noemen we nu ook a). Aldus:

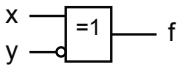
$$x = a, y = a \quad \rightarrow \quad f = \overline{a + a} = \overline{a}$$
$$x = a, y = 0 \quad \rightarrow \quad f = \overline{a + 0} = \overline{a}$$



Figuur A.2: *Functies en aansluitingen van een NOR geschakeld als NOT.*

Uitwerking opgave 1.4.

In onderstaande waarheidtabel is de functie gegeven van een EXOR waarvan één ingang (y) geïnverteerd wordt aangeboden. De verkregen functie is die van de EXNOR.



x	y	$\overline{y}$	$f = x \oplus \overline{y}$
0	0	1	1
0	1	0	0
1	0	1	0
1	1	0	1

Figuur A.3: *Aansluitingen en waarheidstabel van een EXOR met één geïnverteerde ingang.*

Uitwerking opgave 1.5.

De AND is als volgt met nullen te beschrijven:

De uitgang is logisch 0 als één of meer ingangen logisch 0 zijn.

Een andere is:

De uitgang is logisch 0 als de ene ingang logisch 0 is óf de andere ingang logisch 0 is óf beide ingangen logisch 0 zijn.

In beide gevallen is duidelijk de term “of” te lezen: vanuit de 0-en gezien is het een OR-functie.

Uitwerking opgave 1.6.

Met twee variabelen kunnen vier combinaties gemaakt worden. Dat levert o.a. de bekende AND- en OR-functies op. Maar er zijn meer functies te maken: NAND, NOR, EXOR en NOT. In feite kunnen de vier functiewaarden de combinaties 0000 t/m 1111 aannemen. Met twee variabelen zijn dus zestien functies te maken. Met drie variabelen zijn 256 verschillende functies te maken. Met n variabelen zijn er 2^{2^n} functies te maken.

Uitwerking opgave 1.7.

Er zijn twee eindmelders, we noemen ze E_T (top) en E_B (bottom). Elk van deze melders kan óf open óf gesloten zijn, per melder dus twee mogelijkheiden. In totaal zijn er vier mogelijkheden of *combinaties*. Eén combinatie komt in de praktijk (hopelijk) niet voor. We stellen een tabel op om wat meer duidelijkheid te krijgen.

Tabel A.2: *Functietabel van de eindmelders van de kolomboor.*

E_T	E_B	positie boor
open	open	boor is tussen de melders
open	gesloten	boor is beneden
gesloten	open	boor is boven
gesloten	gesloten	kan niet voorkomen

Het mag duidelijk zijn dat de situatie van twee gesloten eindmelders niet voor kan komen. Dit duidt op een systeemfout.

Uitwerking opgave 1.8.

We stellen een tabel op voor open en gesloten schakelaars en voor gedoofde of brandende lamp. We stellen een identieke tabel op maar nu vervangen we de termen “open” en “gedoofd” met een 0 en de termen “gesloten” en “brandt” met 1.

Tabel A.3: *Functietabel en waarheidstabel van de schakeling.*

S_1	S_2	L
open	open	gedoofd
open	gesloten	brandt
gesloten	open	brandt
gesloten	gesloten	gedoofd

S_1	S_2	L
0	0	0
0	1	1
1	0	1
1	1	0

Te zien is dat de schakeling de EXOR-functie realiseert.

Uitwerking opgave 1.9.

Eerst maar eens de schakeling goed bekijken. Links is een OR-schakeling van de signalen A en B te zien. De uitgang van de OR (hulpsignaal x) is samen met signaal C verwerkt tot een AND-schakeling. De laatste stap is een NOT-schakeling van hulpsignaal y. Hieronder is de waarheidstabel gegeven.

Tabel A.4: *Waarheidstabel OR-AND-schakeling met diodes.*

A	B	C	x	y	F	geleiden/sperren diodes
0	0	0	0	0	1	A en B sperren, C geleidt
0	0	1	0	0	1	A, B en C sperren
0	1	0	1	0	1	A spert, B en C geleiden
0	1	1	1	1	0	A en C sperren, B geleidt
1	0	0	1	0	1	A en C geleiden, B spert
1	0	1	1	1	0	A geleidt, B en C sperren
1	1	0	1	0	1	A, B en C geleiden
1	1	1	1	1	0	A en B geleiden, C spert

Uitwerking opgave 1.10.

De functie van de EXOR is $f = x \oplus y$. Aan ingang x wordt de variabele d gekoppeld en aan ingang y worden achtereenvolgens de waarden 0, 1, d en $\overline{d}$ gekoppeld. Hierdoor vereenvoudigd de functie f. Zie onderstaande tabel.

$$\begin{aligned} x = d, y = 0 &\rightarrow f = d \oplus 0 = d \\ x = d, y = 1 &\rightarrow f = d \oplus 1 = \overline{d} \\ x = d, y = d &\rightarrow f = d \oplus d = 0 \\ x = d, y = \overline{d} &\rightarrow f = d \oplus \overline{d} = 1 \end{aligned} \tag{A.3}$$

Uitwerking opgave 1.11.

De functie van de EXNOR is $f = \overline{x \oplus y}$. Aan ingang x wordt de variabele d gekoppeld en aan ingang y worden achtereenvolgens de waarden 0, 1, d en $\overline{d}$ gekoppeld. Hierdoor vereenvoudigd de functie f. Zie onderstaande tabel.

$$\begin{aligned} x = d, y = 0 &\rightarrow f = \overline{d \oplus 0} = \overline{d} \\ x = d, y = 1 &\rightarrow f = \overline{d \oplus 1} = d \\ x = d, y = d &\rightarrow f = \overline{d \oplus d} = 1 \\ x = d, y = \overline{d} &\rightarrow f = \overline{d \oplus \overline{d}} = 0 \end{aligned} \tag{A.4}$$

(Zoooooo, deze uitwerking lijkt wel heel erg op die van opgave 1.10.)

Uitwerking opgave 1.12.
Een EXOR omzetten naar een EXNOR kan op twee manieren: inverteren van de uitgang van een EXOR of inverteren van één ingang van een EXOR (maakt niet uit welke ingang).

Het kan ook met behulp van de schakelalgebra bewezen worden.

$$\begin{aligned}
f &= x \oplus \overline{y} && \text{de functie met één geïnverteerde ingang} \\
&= x\overline{y} + \overline{x}\,\overline{y} && \text{uitschrijven in AND, OR en NOT} \\
&= xy + \overline{x}\,\overline{y} && \text{dubbele inverse verwijderen} \\
&= \overline{\overline{xy} \cdot \overline{\overline{x}\,\overline{y}}} && \text{DeMorgan toepassen} \\
&= \overline{(\overline{x}+\overline{y})\cdot(x+y)} && \text{DeMorgan toepassen} \\
&= \overline{\overline{x}x + \overline{x}y + x\overline{y} + y\overline{y}} && \text{uitvermenigvuldigen} \\
&= \overline{0 + \overline{x}y + x\overline{y} + 0} && \text{reductie inversen} \\
&= \overline{\overline{x}y + x\overline{y}} && \text{de EXNOR-functie} \\
&= \overline{x \oplus y} && \text{geschreven als een NOT-EXOR-functie}
\end{aligned} \tag{A.5}$$

De eerste en de laatste functie zijn resp. een EXOR met één geïnverteerde ingang en een EXOR met geïnverteerde uitgang.

Uitwerking opgave 1.13.
Bij de linker waarheidstabel valt op dat de uitgang y 1 is als ingang a 0 is. De uitgang is ook 1 als ingang b 0 is. De functie voor deze tabel is $y = \overline{a} + \overline{b}$. Merk op dat dit de tabel is van een NAND-poort.

Bij de middelste waarheidstabel valt op dat de uitgang y 1 is als ingang a 1 is. De uitgang is ook 1 als ingang b 0 is. De functie voor deze tabel is $y = a + \overline{b}$.

De rechter tabel levert een 1 als a 0 en b 1 is. Daarnaast levert de tabel een 1 als a 1 en b 0 is. Deze twee (uitgangs-)enen zijn niet te combineren tot een kleinere functie. Merk op dat dit de tabel van een EXOR-poort is.

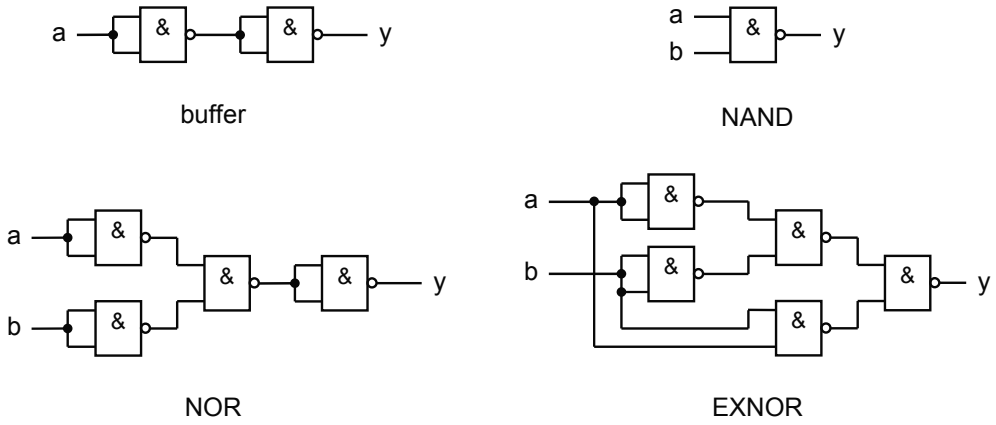
Uitwerking opgave 1.14.
In deze opgave moet een aantal functies worden gerealiseerd met alleen NANDS. De buffer is natuurlijk te realiseren met twee NANDs geschakeld als inverters. De NAND is te maken met één NAND (echt waar!). De NOR en de EXNOR verdienen wat meer aandacht.

De NOR is te schrijven als $y = \overline{a+b} = \overline{\overline{a}\cdot\overline{b}}$ waarbij diverse NANDs geschakeld zijn als inverter.

De EXNOR is wat lastiger:

$$y = \overline{a \oplus b} = a \oplus \overline{b} = \overline{a}\cdot\overline{b} + a\cdot b = \overline{\overline{\overline{a}\cdot\overline{b}} \cdot \overline{a\cdot b}} \tag{A.6}$$

De schema's zijn in figuur A.4 getekend.



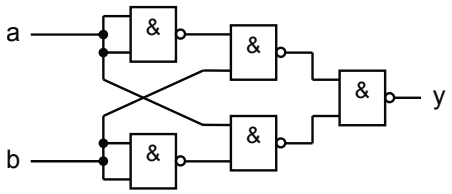
Figuur A.4: *Elementaire poorten opgebouwd uit NAND-poorten.*

Uitwerking opgave 1.15.

De functie is met behulp van De Morgen eenvoudig om te zetten in NANDs:

$$y = \overline{a} \cdot b + a \cdot \overline{b} = \overline{\overline{\overline{a} \cdot b} \cdot \overline{a \cdot \overline{b}}} \tag{A.7}$$

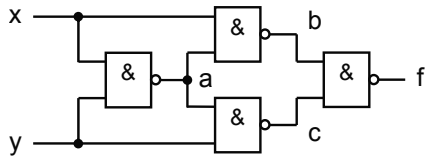
De termen $\overline{a}$ en $\overline{b}$ kunnen gemaakt worden met een NAND geschakeld als inverter. Er zijn in totaal dan vijf NAND-poorten nodig. Zie het schema hieronder.



Figuur A.5: *Schema van een EXOR-poort opgebouwd uit NAND-poorten*

Uitwerking opgave 1.16.

Hieronder is nogmaals de schakeling opgebouwd met vier NAND-poorten gegeven. Bij de interne knooppunten zijn de variabelen a, b en c geplaatst. D.m.v. een waarheidstabel met de functiewaarden van alle uitgangen kan de functie van f bepaald worden.



x	y	a	b	c	f
0	0	1	1	1	0
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	1	1	0

Figuur A.6: *Schema en waarheidstabel van de schakeling met vier NAND-poorten.*

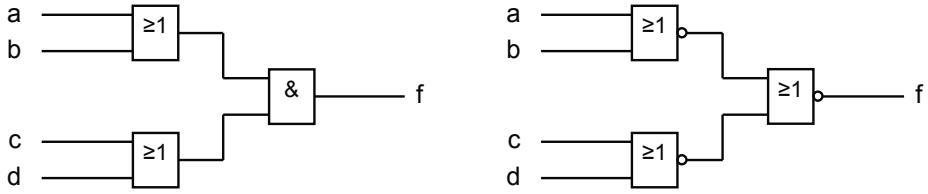
Dit is de functie van een EXOR. Deze schakeling werd in de jaren '70 van de vorige eeuw vaak gebruikt want het was te bouwen met één 7400 TTL-IC.

Uitwerking opgave 1.17.

Een AND-OR-constructie is eenvoudig naar een NOR-NOR-constructie om te zetten door één keer De Morgan toe te passen:

$$f = (a+b)\cdot(c+d) = \overline{\overline{a+b}+\overline{c+d}} \tag{A.8}$$

Het OR-AND-schema en het NOR-NOR-schema zijn hieronder weergegeven.



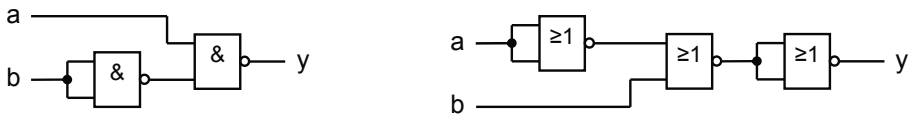
Figuur A.7: *Schema's van een OR-AND-schema en NOR-NOR-schema.*

Uitwerking opgave 1.18.

Snel is te zien dat de functie logisch 1 is als $a = 0$ en/of $b = 1$. In formulevorm $y = \overline{a}+b$. De formule is om te schrijven naar NANDs en NORs.

$$\begin{aligned} y &= \overline{a}+b = \overline{a\cdot\overline{b}} && \text{realisatie met NANDs} \\ y &= \overline{a}+b = \overline{\overline{a}+b} && \text{realisatie met NORs} \end{aligned} \tag{A.9}$$

De termen $\overline{a}$ en $\overline{b}$ kunnen gemaakt worden met een NAND of een NOR geschakeld als inverter. Zie de schema's hieronder.



Figuur A.8: *Realisatie van de functie met NAND- en NOR-poorten.*

Uitwerking opgave 1.19.

We moeten zoeken naar het kortste en langste pad (in tijd). Het kortste pad is snel te vinden. Via ingang z naar uitgang s wordt één AND-poort gepasseerd. Dus:

$$t_{P(min)}(\text{schakeling}) = t_{P(min)}(\text{AND}) = 2{,}4 \text{ ns} \tag{A.10}$$

Het langste pad is via ingang y. Daarbij wordt een NOT-poort, twee AND-poorten en een OR-poort gepasseerd. Dus

$$\begin{aligned} t_{P(max)}(\text{schakeling}) &= t_{P(max)}(\text{NOT}) + 2\cdot t_{P(min)}(\text{AND}) + t_{P(max)}(\text{OR}) \\ &= 4{,}3 + 2\cdot 6{,}9 + 7{,}2 \\ &= 25{,}3 \text{ ns} \end{aligned} \tag{A.11}$$

Uitwerking opgave 1.20.
De periodetijd van een signaal met de frequentie van 33,3 MHz is 30,03 ns (nanoseconden, 1 ns is 10^{-9} s. Dit is de frequentie van een 486DX-33 processor.

De periodetijd van een signaal met de frequentie van 3,4 GHz is 294,12 ps (picoseconden, 1 ps is 10^{-12} s. Dit is de frequentie van o.a. een Core-i7 processor.

De frequentie van een signaal met een periodetijd van 104,167 μs is 9,6 kHz (kilohertz). Dit is een standaard seinsnelheid van een RS-232 seriële verbinding.

A.2 Uitwerkingen hoofdstuk 2

Uitwerking opgave 2.1.
De getallen zijn om te zetten naar decimale equivalenten door alle 2-machten te noteren waarvan het corresponderende bit een 1 is en de 0-en te negeren:

$$\begin{aligned} 1111_2 &= 2^3 + 2^2 + 2^1 + 2^0 \\ &= 8 + 4 + 2 + 1 \\ &= 15 \end{aligned} \qquad \begin{aligned} 110011_2 &= 2^5 + 2^4 + 2^1 + 2^0 \\ &= 32 + 16 + 2 + 1 \\ &= 51 \end{aligned}$$

$$\begin{aligned} 01111111_2 &= 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \\ &= 64 + 32 + 16 + 8 + 4 + 2 + 1 \\ &= 127 \end{aligned}$$

$$\begin{aligned} 1010.0101.1010.0101_2 &= 2^{15} + 2^{13} + 2^{10} + 2^8 + 2^7 + 2^5 + 2^2 + 2^0 \\ &= 32768 + 8192 + 1024 + 256 + 128 + 32 + 4 + 1 \\ &= 42405 \end{aligned}$$

Figuur A.9: *Decimale equivalent van enkele binaire getallen.*

Uitwerking opgave 2.2.
De getallen worden van decimaal naar binair omgezet door herhaald de delen door 2 en rest na deling te noteren. Dit gaat door tot het gehele deel 0 geworden is. Daarna kan het binaire equivalent uitgelezen worden door de resten na deling van onder naar boven te noteren.

Dus 25_{10} is gelijk aan 11001_2 en 10_{10} is gelijk aan 1010_2.

Dus 128_{10} is gelijk aan 10000000_2 en 27543_{10} is gelijk aan 110101110010111_2.

Uitwerking opgave 2.3.
Deze leggen we niet uit....

$$
\begin{array}{rcrcl}
25 & \div 2 & = & 12 & \rightarrow 1 \\
12 & \div 2 & = & 6 & \rightarrow 0 \\
6 & \div 2 & = & 3 & \rightarrow 0 \\
3 & \div 2 & = & 1 & \rightarrow 1 \\
1 & \div 2 & = & 0 & \rightarrow 1 \\
\underline{0} & & & &
\end{array}
\qquad
\begin{array}{rcrcl}
10 & \div 2 & = & 5 & \rightarrow 0 \\
5 & \div 2 & = & 2 & \rightarrow 1 \\
2 & \div 2 & = & 1 & \rightarrow 0 \\
1 & \div 2 & = & 0 & \rightarrow 1 \\
\underline{0} & & & &
\end{array}
$$

Figuur A.10: *Binaire equivalent van enkele decimale getallen.*

$$
\begin{array}{rcrcl}
128 & \div 2 & = & 64 & \rightarrow 0 \\
64 & \div 2 & = & 32 & \rightarrow 0 \\
32 & \div 2 & = & 16 & \rightarrow 0 \\
16 & \div 2 & = & 8 & \rightarrow 0 \\
8 & \div 2 & = & 4 & \rightarrow 0 \\
4 & \div 2 & = & 2 & \rightarrow 0 \\
2 & \div 2 & = & 1 & \rightarrow 0 \\
1 & \div 2 & = & 0 & \rightarrow 1 \\
\underline{0} & & & &
\end{array}
\qquad
\begin{array}{rcrcl}
27543 & \div 2 & = & 13771 & \rightarrow 1 \\
13771 & \div 2 & = & 6885 & \rightarrow 1 \\
6885 & \div 2 & = & 3442 & \rightarrow 1 \\
3442 & \div 2 & = & 1721 & \rightarrow 0 \\
1721 & \div 2 & = & 860 & \rightarrow 1 \\
860 & \div 2 & = & 430 & \rightarrow 0 \\
430 & \div 2 & = & 215 & \rightarrow 0 \\
215 & \div 2 & = & 107 & \rightarrow 1 \\
107 & \div 2 & = & 53 & \rightarrow 1 \\
53 & \div 2 & = & 26 & \rightarrow 1 \\
26 & \div 2 & = & 13 & \rightarrow 0 \\
13 & \div 2 & = & 6 & \rightarrow 1 \\
6 & \div 2 & = & 3 & \rightarrow 0 \\
3 & \div 2 & = & 1 & \rightarrow 1 \\
1 & \div 2 & = & 0 & \rightarrow 1 \\
\underline{0} & & & &
\end{array}
$$

Figuur A.11: *Decimale equivalent van enkele binaire getallen.*

Uitwerking opgave 2.4.

Het uitrekenen geschiedt door de cijfers uit de hexadecimale getallen te schrijven als een 16-macht met in de exponent de positie van het cijfer (beginnend bij 0 voor het meest rechtse hexadecimale cijfer) vermenigvuldigd met het corresponderende cijfer uit het hexadecimale getal. Let er op dat de hexadecimale cijfers A t/m F omgezet moeten worden naar de getallen 10 t/m 15.

Uitwerking opgave 2.5.

$$
\begin{array}{rcrcl}
255 & \div 16 & = & 15 & \rightarrow 15\ (\mathrm{F}) \\
15 & \div 16 & = & 0 & \rightarrow 15\ (\mathrm{F}) \\
\underline{0} & & & &
\end{array}
\qquad
\begin{array}{rcrcl}
57005 & \div 16 & = & 3562 & \rightarrow 13\ (\mathrm{D}) \\
3562 & \div 16 & = & 222 & \rightarrow 10\ (\mathrm{A}) \\
222 & \div 16 & = & 13 & \rightarrow 14\ (\mathrm{E}) \\
13 & \div 16 & = & 0 & \rightarrow 13\ (\mathrm{D}) \\
\underline{0} & & & &
\end{array}
$$

Figuur A.13: *Hexadecimake equivalent van enkele decimale getallen.*

$$
\begin{aligned}
3FF_{16} &= 3\cdot16^2 + 15\cdot16^1 + 15\cdot16^0 \\
&= 3\cdot256 + 15\cdot16 + 15\cdot1 \\
&= 768 + 240 + 25 \\
&= 1023
\end{aligned}
\qquad
\begin{aligned}
4D52_{16} &= 4\cdot16^3 + 13\cdot16^2 + 5\cdot16^1 + 2\cdot16^0 \\
&= 4\cdot4096 + 13\cdot256 + 5\cdot16 + 2\cdot1 \\
&= 16384 + 3328 + 80 + 2 \\
&= 19794
\end{aligned}
$$

$$
\begin{aligned}
CAFE_{16} &= 12\cdot16^3 + 10\cdot16^2 + 15\cdot16^1 + 14\cdot16^0 \\
&= 12\cdot4096 + 10\cdot256 + 15\cdot16 + 14\cdot1 \\
&= 51966
\end{aligned}
$$

Figuur A.12: *Decimale equivalent van enkele hexadecimale getallen.*

Dus 255_{10} is gelijk aan FF_{16} en 57005_{10} is gelijk aan $DEAD_{16}$. Het hexadecimale equivalent van 32768_{10} is 8000_{16}. Dit is een macht van 2 (2^{15}).

Uitwerking opgave 2.6.

IPv6-adressen zijn 128 bits groot. Aangezien elk bit een 0 of een 1 kan zijn, is het aantal adressen 2^{128}. Dat zijn 340.282.366.920.938.463.463.374.607.431.768.211.456 verschillende adressen. Zie http://en.wikipedia.org/wiki/IPv6_address.

Leuk om te weten: het hele IPv4-adressenbereik kan 2^{96} keer in het IPv6-adressenbereik. Dat is 79.228.162.514.264.337.593.543.950.336 keer.

Uitwerking opgave 2.7.

Voor het gemak gaan we ervan uit dat de aarde een bol is (dat is het namelijk niet). Onze aarde heeft een diameter van 6.371 km, dat is 6.371.000 m. De oppervlakte van een bol is $O = 4\pi r^2$. Invullen en uitrekenen levert een opppervlakte van 510.064.471.909.788 m^2. Nu het aantal adressen delen door de oppervlakte, dat levert 667.135.990.959.828.999.515.555 IPv6-adressen per vierkante meter. We schrijven het even in 10-machten om de grootte te overzien: $6{,}67\cdot10^{23}$. Dat is waarschijnlijk wel genoeg voor de komende tijd. Op de website https://rednectar.net/2012/05/24/just-how-many-ipv6-addresses-are-there-really/ zijn nog meer interessante gegevens te vinden.

Uitwerking opgave 2.8.

Binaire breuken omzetten naar het decimale equivalent gaat op dezelfde wijze als het omzetten van gehele getallen, alleen zijn de machten nu negatief. Het gewicht van het eerste cijfer rechts van de komma is $2^{-1} = \frac{1}{2}$ en het tweede cijfer is $2^{-2} = \frac{1}{4}$ etc. Uiteraard worden alleen die machten genoteerd waarvoor het cijfer 1 in het getal staat.

Overigens het het slimmer om gebruik te maken van de gelijkheid $0{,}11111111_2 = 1_2 - 0{,}00000001_2$ zodat het antwoord berekend kan worden met $1 - 1^{-8}$ of $1 - \frac{1}{256}$.

Uitwerking opgave 2.9.

Het omzetten van een decimale breuk naar het binaire equivalent wordt gerealiseerd

$$
\begin{aligned}
0{,}1001_2 &= 2^{-1} + 2^{-4} \\
&= \frac{1}{2} + \frac{1}{16} \\
&= 0{,}5 + 0{,}0625 \\
&= 0{,}5625
\end{aligned}
\qquad
\begin{aligned}
0{,}11111111 &= 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-7} + 2^{-8} \\
&= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256} \\
&= 0{,}5 + 0{,}25 + \cdots + 0{,}0078125 + 0{,}00390625 \\
&= 0{,}99609375
\end{aligned}
$$

Figuur A.14: *Decimale equivalent van enkele binaire fracties.*

door de decimale breuk te vermenigvuldigen met 2, het gehele getal te noteren (0 of 1) en de procedure te herhalen met de fractie totdat het resultaat van een vermenigvuldiging 0,0 oplevert. Zie hieronder.

$$
\begin{array}{lcrcl}
0{,}75 & \times 2 & = & 1{,}5 & \rightarrow 1 \\
0{,}5 & \times 2 & = & 1{,}0 & \rightarrow 1 \\
\underline{0{,}0} & & & &
\end{array}
\qquad
\begin{array}{lcrcl}
0{,}3 & \times 2 & = & 0{,}6 & \rightarrow 0 \\
0{,}6 & \times 2 & = & 1{,}2 & \rightarrow 1 \\
0{,}2 & \times 2 & = & 0{,}4 & \rightarrow 0 \\
0{,}4 & \times 2 & = & 0{,}8 & \rightarrow 0 \\
0{,}8 & \times 2 & = & 1{,}6 & \rightarrow 1 \\
\textit{0,6} & \ldots & & &
\end{array}
$$

Figuur A.15: *Binaire equivalent van enkele decimale fracties.*

Het getal $0{,}75_{10}$ is gelijk aan $0{,}11_2$. Het getal $0{,}6_{10}$ leidt tot de repeterende binaire breuk $0{,}010011001\ldots_2$ wat inhoudt *dat 0,6 niet exact kan worden weergegeven in het binaire stelsel!* Dit geldt trouwens voor veel eindige decimale breuken. In de rechter berekening zijn de getallen $0{,}3_{10}$, $0{,}2_{10}$, $0{,}3_{10}$ en $0{,}8_{10}$ te zien, die zijn uiteraard ook niet exact weer te geven.

Op eenzelfde wijze levert de omzetting van $0{,}8_{10}$ het binaire getal $0{,}11001100\ldots_2$ op.

Uitwerking opgave 2.10.

Het getal 11_{10} is zo basaal, dat schrijven we direct op: B_{16}.

De decimale breuk kan eerst omgezet worden naar een binaire breuk en dan omgezet worden naar een hexadecimaal equivalent. Maar het kan sneller door het direct uit te rekenen als hexadecimale breuk:

$$
\begin{array}{lcrcl}
0{,}7 & \times 16 & = & 11{,}2 & \rightarrow \mathrm{B} \\
0{,}2 & \times 16 & = & 3{,}2 & \rightarrow 3 \\
\textit{0,2} & & & &
\end{array}
$$

Figuur A.16: *Hexadecimaal equivalent van een decimale fractie.*

Het getal $11{,}7_{10}$ is equivalent met $\mathrm{B{,}B3333}\ldots_{16}$. Merk op dat het getal niet exact is weer te geven in het hexadecimale talstelsel.

Het getal $1F,3C_{16}$ naar decimaal omzetten gaat op de gebruikelijke manier.

$$\begin{aligned} 1F,3C_{16} &= 1\cdot16^1 + 15\cdot16^0 + 3\cdot16^{-1} + 12\cdot16^{-2} \\ &= 16 + 15 + \frac{3}{16} + \frac{12}{256} \\ &= 31 + 0{,}1875 + 0{,}046875 \\ &= 31{,}234375 \end{aligned}$$

Uitschrijven van een hexadecimaal getal als binair getal is eenvoudig weg alle hexadecimale cijfers vervangen door de bijbehorende 4-bits binaire code:

$$3F,EA_{16} = 0011.1111,1110.1010_2$$

Let op de plaats van de punten en de komma. Leidende en volgende nullen zouden kunnen worden weggelaten.

Bij het omzetten van het binaire getal $110110,110111_2$ moeten het gehele deel en de fractie eerst geschreven worden in veelvouden van vier bits, een hexadecimaal cijfer moet immers geschreven worden met precies vier bits.

We breiden beide delen uit: het gehele deel krijgt voorlopende nullen, de fractie krijgt volgende nullen: $00110110,11011100_2$. Het getal is nu op te splitsen in delen van precies vier bits: $0011.0110,11011100_2$ en is direct te schrijven als het hexadecimale getal $36,DC_{16}$.

Uitwerking opgave 2.11.
In het octale (achttallig) talstelsel worden de cijfers 0 t/m 7 gebruikt. Het omzetten van een octaal getal naar een decimaal getal geschiedt op de zelfde wijze als bij binaire en hexadecimale getallen.

$$\begin{aligned} 377_8 &= 3\cdot8^2 + 7\cdot8^2 + 7\cdot8^0 \\ &= 3\cdot64 + 7\cdot8 + 7\cdot1 \\ &= 192 + 56 + 7 \\ &= 255 \end{aligned} \qquad \begin{aligned} 127_8 &= 1\cdot8^2 + 2\cdot8^2 + 7\cdot8^0 \\ &= 1\cdot64 + 2\cdot8 + 7\cdot1 \\ &= 64 + 16 + 7 \\ &= 87 \end{aligned}$$

Octale getallen bestaan uit acht verschillende cijfers. Acht is een macht van 2 ($2^3 = 8$) en dat houdt in dat een octaal getal eenvoudig is om te zetten in een binair getal en dus ook in een hexadecimaal getal.

In de programmeertaal C begint een octaal getal met een 0, dus `0377` is 377_8 (255_{10}).

Het toegangsrechtensysteem bij bestandssystemen onder Linux kan worden aangepast en weergegeven d.m.v. octale getallen. Zie http://linuxcommand.org/lts0070.php. Onder Linux is het commando `od` (octal dump) beschikbaar om bestanden octaal (en hexadecimaal) weer te geven. Zie http://en.wikipedia.org/wiki/Od_(Unix).

In de datacommunicatiewereld wordt niet gesproken van een byte maar een *octet*.

Uitwerking opgave 2.12.
Het getal 41 in onbekende talstelsel is gelijk aan het kwadraat van 5 in dit stelsel. Dus $41_x = (5_x)^2$. Daaruit volgt

$$4 \cdot x^1 + 1 \cdot x^0 = (5 \cdot x^0) \cdot (5 \cdot x^0) \tag{A.12}$$

Nu het een en ander herschrijven en vervolgens uitrekenen:

$$\begin{aligned} 4x + 1 &= 25 \\ 4x &= 24 \\ x &= 6 \end{aligned} \tag{A.13}$$

Het getal 41 staat in het zestallig (hexaal?) talstelsel.

Uitwerking opgave 2.13.
Het marsiaanse getal 234 is geschreven in het x-tallig stelsel en dit getal is gelijk aan 123_{10}. Dan geldt $234_x = 123_{10}$. We schrijven het marsiaanse getal uit in machten van x en stellen dit gelijk aan het decimale getal 123.

$$\begin{aligned} 234_x &= 2 \cdot x^2 + 3 \cdot x^1 + 4 \cdot x^0 \\ &= 2 \cdot x^2 + 3 \cdot x + 4 \cdot 1 \\ &= 123 \end{aligned} \tag{A.14}$$

Deze vergelijking kan worden omgewerkt naar een tweedegraads vergelijking

$$2x^2 + 3x - 119 = 0 \tag{A.15}$$

Oplossen met de ABC-formule[1] levert twee antwoorden op: $x_1 = 7$ en $x_2 = -8{,}5$. Een grondtal moet een geheel getal groter dan 1[2] zijn waardoor x_2 afvalt. De marsiaanse getallen zijn in het 7-tallig stelsel geschreven.

Uitwerking opgave 2.14.
Het getal 653 moet worden omgeslagen in acht bits. De vraag is natuurlijk hoe je zo'n getal kan invoeren als het in acht bits moet worden opgeslagen, maar de C-compiler doet dat. De bewerking is

$$653 \bmod 2^8 = 141 \qquad \text{of} \qquad 141 \equiv 653 \pmod{2^8} \tag{A.16}$$

Onderstaande code is een implementatie in C (listing A.1) die continue tekens inleest totdat een niet-cijfer wordt ingevoerd. Met deze code is het goed mogelijk een getal in te voeren dat te groot is om in het resultaat opgeslagen te worden.

[1] Zie http://www.wiskundeonline.nl/lessen/kw_bewijs_abc_formule.htm
[2] Strikt genomen hoeft het grondtal geen geheel getal en/of positief te zijn. Je kan ook getallen weergeven met het grondtal $\sqrt{2}$ or π. Zie http://en.wikipedia.org/wiki/Non-integer_representation

```c
#include <stdio.h>              /* for getchar et al. */

int main(void) {

    unsigned char res;          /* 8-bit unsigned */
    int x;                      /* 16-bit signed */

    res = 0;                    /* clear intermediate result */
    x = getchar();              /* get a character */
    while (isdigit(x)) {        /* if it is a digit ... */
        res = res*10 + (x-'0'); /* mult. prev. by 10, add new digit */
        x = getchar();          /* get next character */
    }
    printf("%d\n", res);
    return 0;
}
```

Listing A.1: *Het inlezen van een integer in 'C'.*

Uitwerking opgave 2.15.

Onderstaande uitdrukkingen geven het verband aan.

$$x \equiv M \pmod{2^n} \quad \rightarrow \quad x = k \cdot 2^n + M \quad \text{met } k \in \mathrm{N} \tag{A.17}$$

Uitwerking opgave 2.16.

Het aantal bits dat nodig is om een 8-cijferig decimaal getal op te slaan, is eenvoudig te bepalen door de 2-log van 10^8 te nemen en het antwoord af te ronden naar het eerst hogere gehele getal (er bestaan immers alleen hele bits). Dus

$$n = \left\lceil \frac{8}{\log 2} \right\rceil = \lceil 26{,}575\cdots \rceil = 27 \tag{A.18}$$

Natuurlijk kunnen er met 27 bits meer dan 10^8 getallen gemaakt worden.

Uitwerking opgave 2.17.

Om het aantal bits dat nodig is om een specifiek decimaal getal op te slaan uit te rekenen moet de 2-log van het getal vermeerderd met 1 uitgerekend worden. Natuurlijk moet weer worden afgerond naar het eerst hogere gehele getal.

$$\begin{aligned}
n_{365} &= \left\lceil \frac{\log(365+1)}{\log 2} \right\rceil = \lceil 8{,}51\cdots \rceil = 9 \\
n_{1024} &= \left\lceil \frac{\log(1024+1)}{\log 2} \right\rceil = \lceil 10{,}001\cdots \rceil = 11 \\
n_{7987} &= \left\lceil \frac{\log(7987+1)}{\log 2} \right\rceil = \lceil 12{,}96\cdots \rceil = 13 \\
n_{176890} &= \left\lceil \frac{\log(176890+1)}{\log 2} \right\rceil = \lceil 17{,}43\cdots \rceil = 18
\end{aligned} \tag{A.19}$$

Op http://www.exploringbinary.com/number-of-bits-in-a-decimal-integer/ is een uitgebreid verhaal te vinden.

Uitwerking opgave 2.18.

Een twee-cijferig BCD-getal gebruikt 100 combinaties (00 t/m 99). Hiervoor zijn acht bits nodig, vier bits per BCD-cijfer. Het aantal nuttige combinaties is $\frac{100}{256} \cdot 100\%$. Dat is ongeveer 39%.

Een drie-cijferig BCD-getal heeft $\frac{1000}{4096} \cdot 100\%$ codecombinaties, dat is ongeveer 24%.

Uitwerking opgave 2.19.

Eerst maar eens een waarheidstabel opstellen voor de omzetter. De ingangen staan als normaal binaire telcode gerangschikt, de uitgangen stellen de bijbehorende Gray-code voor. Daar kunnen de functies voor de Gray-bits worden bepaald.

Tabel A.5: *Waarheidstabel omzetting binaire telcode naar Gray-code.*

b_2	b_1	b_0	g_2	g_1	g_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

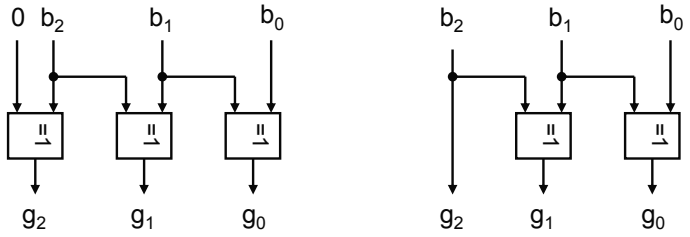
Na enig rekenwerk volgt:

$$\begin{aligned} g_0 &= b_1 \oplus b_0 \\ g_1 &= b_2 \oplus b_1 \\ g_2 &= 0 \oplus b_2 = b_2 \end{aligned} \tag{A.20}$$

Het mag duidelijk zijn dat de functies uit EXOR-poorten bestaan. De functie voor g_2 is als EXOR geschreven waarbij één ingang aan een logische 0 gekoppeld is. Uitbreiding naar meer bits is dus eenvoudig: voor bit g_i geldt:

$$g_i = b_{i+1} \oplus b_i \tag{A.21}$$

De vertragingstijd is slechts één poortvertraging, onafhankelijk van het aantal bits. Zie onderstaande figuur.



Figuur A.17: *Omzetter binaire telcode naar Gray-code.*

Uitwerking opgave 2.20.

Op vergelijkbare wijze als in opgave 2.19 kan ook deze opgave worden opgelost. De ingangen stellen nu de Gray-code voor, de uitgangen stellen de bijbehorende normale binaire code voor. Zie onderstaande tabel.

Tabel A.6: *Waarheidstabel omzetting Gray-code naar binaire telcode.*

g_2	g_1	g_0	b_2	b_1	b_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	0	0
1	1	1	1	0	1

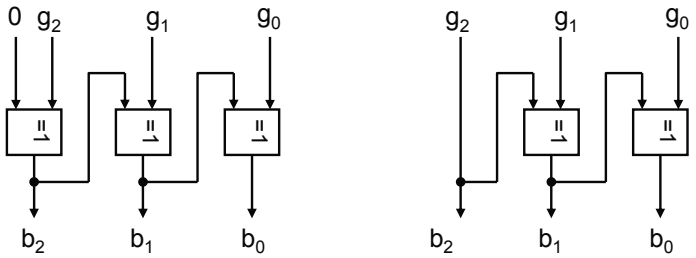
Na enig rekenwerk volgt:

$$\begin{aligned} b_0 &= g_2 \oplus g_1 \oplus g_0 \quad &= b_1 \oplus g_0 \\ b_1 &= g_2 \oplus g_1 \quad &= b_2 \oplus g_1 \\ b_2 &= g_2 \oplus 0 \quad &= 0 \oplus g_2 \end{aligned} \tag{A.22}$$

Het mag duidelijk zijn dat de functies uit EXOR-poorten bestaan. De functie voor g_2 is als EXOR geschreven waarbij één ingang aan een logische 0 gekoppeld is. Uitbreiding naar meer bits is dus eenvoudig: voor bit g_i geldt:

$$b_i = b_{i+1} \oplus g_i \tag{A.23}$$

Let er op dat dit een cascade- of serieschakeling van EXOR-poorten oplevert omdat bij de functie voor b_i de waarde van b_{i+1} nodig is, die moet dan wel bekend zijn. Bij een groot aantal bits levert dit een aanzienlijke vertragingstijd op. Zie onderstaande figuur.



Figuur A.18: *Omzetter van Gray-code naar binaire code.*

Uitwerking opgave 2.21.

Uit de ASCII-tabel blijkt dat de hoofdletters oplopend worden gecodeerd vanaf code 41_{16}. De 'A' is 41_{16}, 'B' is 42_{16} etc. Voor de kleine letters geldt dat deze ook oplopend gecodeerd zijn maar dan vanaf 61_{16}, dus 'a' is 61_{16}, 'b' is 62_{16} etc. Het verschil tussen een hoofdletters en een bijbehorende kleine letter is 20_{16}.

Om een hoofdletter te veranderen in een kleine letter moet bij de code van de hoofdletter 20_{16} worden opgeteld, bij het veranderen van een kleine letter in een hoofdletter moet er 20_{16} worden afgetrokken.

Nu is 20_{16} hetzelfde als 100000_2 of 2^5. Je kan dus ook het 6^e bit van de ASCII-code aanpassen: een 0 levert een hoofdletter op, een 1 levert een kleine letter op. Dat alles mag natuurlijk alleen in het ASCII-bereik van de letters.

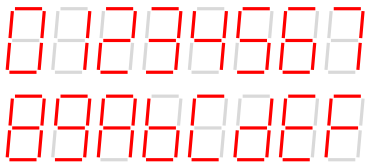
In de programmeertaal C kan je eenvoudig bewerkingen doen op *characters* (tekens). Zie het voorbeeld hieronder.

```c
#include <stdio.h>

int main(void) {

    char A, a, B, b, Negen, negen;

    A = 'A'; b = 'b'; negen = '9';

    a = A + 0x20;           /* a = 'a' */
    B = b - 0x20;           /* B = 'B' */
    Negen = negen + 0x20;   /* Negen = 'Y' */

    printf("%c -- %c,%c -- %c,%c -- %c\n", A, a, B, b, Negen, negen);

    return 0;
}
```

Listing A.2: *Het omzetten van hoofdletters naar kleine letters in 'C'.*

In C zijn de functies `toupper()` en `tolower()` beschikbaar voor de conversie tussen kleine letters en hoofdletters.

Uitwerking opgave 2.22.
Hieronder is een afbeelding te zien van alle 16 hexadecimale cijfers. Merk op dat de 'A' als hoofdletter is geschreven, het is niet mogelijk om een kleine 'a' te maken. De 'b' is als kleine letter te zien, want een hoofdletter 'B' is niet te onderscheiden van een '8'. De 'C' zou ook als kleine letter kunnen worden uitgevoerd, maar meestal wordt gekozen voor de hoofdletter. De 'd' is weer als kleine letter geschreven, een hoofdletter 'D' is niet te onderscheiden van een '0'. De 'E' wordt als hoofdletter geschreven, een kleine letter 'e' kan wel maar is te 'hoog'. De 'F' kan alleen als hoofdletter geschreven worden. Merk trouwens het subtiele verschil tussen de '6' en de 'b' op. (De figuur is gerealiseerd met een aangepaste versie op http://www.texample.net/tikz/examples/segment-display/.)



Figuur A.19: *Alle hexadecimale cijfers op een 7-segment display.*

A.3 Uitwerkingen hoofdstuk 3

Uitwerking opgave 3.1.
Gegeven dat $a = 1$, $b = 0$ en $c = 0$. De waarde van d is niet gegeven.

$$\begin{aligned}
s &= (a+b)\cdot(\overline{b}+c) \\
&= (1+0)\cdot(\overline{0}+0) \\
&= (1)\cdot(1+0) \\
&= 1\cdot 1 \\
&= 1
\end{aligned}
\qquad
\begin{aligned}
s &= \overline{\overline{(a+b)}\cdot\overline{(c+d)}} \\
&= \overline{\overline{(1+0)}\cdot\overline{(0+d)}} \\
&= \overline{\overline{(1)}\cdot\overline{(d)}} \\
&= \overline{0\cdot\overline{d}} \\
&= \overline{0} \\
&= 1
\end{aligned}
\qquad
\begin{aligned}
s &= \overline{b}\cdot(c+d\cdot(c+a)) \\
&= \overline{0}\cdot(0+d\cdot(0+1)) \\
&= 1\cdot(0+d\cdot(1)) \\
&= 1\cdot(0+d\cdot 1) \\
&= 1\cdot d \\
&= d
\end{aligned}
\tag{A.24}$$

Natuurlijk kunnen sommige stappen samengenomen worden. Zo kan $(0+1)$ direct geschreven worden als 1.

Uitwerking opgave 3.2.
Onderstaande is het bewijs voor de stelling $a + a\cdot b = a$.

$$\begin{aligned}
a + a\cdot b &= a\cdot 1 + a\cdot b \\
&= a\cdot(1+b) \\
&= a\cdot 1 \\
&= a
\end{aligned}
\tag{A.25}$$

Uitwerking opgave 3.3.
Laten we de functie even S noemen:

$$\begin{aligned}
S &= (a+b)\cdot(\overline{a}+c)\cdot(b+c) && \text{de functie} \\
&= (a+b)\cdot(\overline{a}+c)\cdot(0+b+c) && \text{middels } x = 0+x \\
&= (a+b)\cdot(\overline{a}+c)\cdot(\overline{a}\cdot a+b+c) && \text{middels } \overline{x}\cdot x = 0 \\
&= (a+b)\cdot(\overline{a}+c)\cdot(\overline{a}+b+c)\cdot(a+b+c) && \text{middels } \overline{x}\cdot x + y = (\overline{x}+y)\cdot(x+y) \\
&= (a+b)\cdot(a+b+c)\cdot(\overline{a}+c)\cdot(\overline{a}+b+c) && \text{termen herschikken} \\
&= (a+b)\cdot(\overline{a}+c) && \text{middels } (x)\cdot(x+y) = (x)
\end{aligned}
\tag{A.26}$$

Uitwerking opgave 3.4.
De functie omwerken naar een som van mintermen gaat als volgt:

$$\begin{aligned}
s_{x,y,z} &= x\cdot(y+z\cdot\overline{y})+z\cdot y \\
&= x\cdot y + x\cdot\overline{y}\cdot z + y\cdot z \\
&= x\cdot y\cdot(\overline{z}+z) + x\cdot\overline{y}\cdot z + (\overline{x}+x)\cdot y\cdot z \\
&= x\cdot y\cdot\overline{z} + x\cdot y\cdot z + x\cdot\overline{y}\cdot z + \overline{x}\cdot y\cdot z + x\cdot y\cdot z \\
&= \overline{x}\cdot y\cdot z + x\cdot\overline{y}\cdot z + x\cdot y\cdot\overline{z} + x\cdot y\cdot z
\end{aligned}
\tag{A.27}$$

In deze functie is x de meest significante en z de minst significante variabele. De som van mintermen en de waarheidstabel zijn als volgt (waarheidstabel en functie):

Tabel A.7: *Waarheidstabel.*

x	y	z	s
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$\begin{aligned} s_{x,y,z} &= \overline{x}\cdot y\cdot z + x\cdot \overline{y}\cdot z + x\cdot y\cdot \overline{z} + x\cdot y\cdot z \\ &= m_3 + m_5 + m_6 + m_7 \\ &= \sum m(3,5,6,7) \end{aligned} \tag{A.28}$$

Uitwerking opgave 3.5.
De functie moet in mintermen uitgeschreven worden:

$$\begin{aligned} s_{x,y,z} &= m_1 + m_3 + m_4 + m_7 \\ &= \overline{x}\cdot\overline{y}\cdot z + \overline{x}\cdot y\cdot z + x\cdot\overline{y}\cdot\overline{z} + x\cdot y\cdot z \end{aligned} \tag{A.29}$$

Uitwerking opgave 3.6.
De functie moet in maxtermen worden uitgeschreven:

$$\begin{aligned} s_{x,y,z} &= M_2\cdot M_6\cdot M_7 \\ &= (x+\overline{y}+z)\cdot(\overline{x}+\overline{y}+z)\cdot(\overline{x}+\overline{y}+\overline{z}) \end{aligned} \tag{A.30}$$

Uitwerking opgave 3.7.
We kunnen de som-van-producten-vorm omzetten in een product-van-sommen-vorm door de nullen uit de functie $s_{x,y,z}$ te nemen:

$$s_{x,y,z} = \sum m(1,2,3,6) = \prod M(0,4,5,7) \tag{A.31}$$

zodat de product-van-sommen-vorm als volgt wordt:

$$s_{x,y,z} = \prod M(0,4,5,7) = (x+y+z)\cdot(\overline{x}+y+z)\cdot(\overline{x}+y+\overline{z})\cdot(\overline{x}+\overline{y}+\overline{z}) \tag{A.32}$$

Uitwerking opgave 3.8.
We kunnen de product-van-sommen-vorm omzetten in een som-van-producten-vorm door de enen uit de functie $s_{x,y,z}$ te nemen:

$$s_{x,y,z} = \prod M(0,1,3,5) = \sum m(2,4,6,7) \tag{A.33}$$

zodat de product-van-sommen-vorm als volgt wordt:

$$s_{x,y,z} = \sum m(2,4,6,7) = \overline{x}\cdot y\cdot\overline{z} + x\cdot\overline{y}\cdot\overline{z} + x\cdot y\cdot\overline{z} + x\cdot y\cdot z \tag{A.34}$$

Uitwerking opgave 3.9.

$$
\begin{aligned}
S_1 &= \sum m(1,3,5,6,7) = \overline{x}\cdot\overline{y}\cdot z + \overline{x}\cdot y\cdot z + x\cdot\overline{y}\cdot z + x\cdot y\cdot\overline{z} + x\cdot y\cdot z \\
S_2 &= \sum m(0,1,2,4,7) = \overline{x}\cdot\overline{y}\cdot\overline{z} + \overline{x}\cdot\overline{y}\cdot z + \overline{x}\cdot y\cdot\overline{z} + x\cdot\overline{y}\cdot\overline{z} + x\cdot y\cdot z \\
S_3 &= \sum m(7) = x\cdot y\cdot z \\
S_4 &= \sum m(1,2,3,5,6,7) = \overline{x}\cdot\overline{y}\cdot z + \overline{x}\cdot y\cdot\overline{z} + \overline{x}\cdot y\cdot z + x\cdot\overline{y}\cdot z + x\cdot y\cdot\overline{z} + x\cdot y\cdot z \\
S_5 &= \sum m(1,2,3,4,5,6,7) = \overline{x}\cdot\overline{y}\cdot z + \overline{x}\cdot y\cdot\overline{z} + \overline{x}\cdot y\cdot z + x\cdot\overline{y}\cdot\overline{z} + x\cdot\overline{y}\cdot z + x\cdot y\cdot\overline{z} + x\cdot y\cdot z \\
S_6 &= \sum m(0,2,4,6) = \overline{x}\cdot\overline{y}\cdot\overline{z} + \overline{x}\cdot y\cdot\overline{z} + x\cdot\overline{y}\cdot\overline{z} + x\cdot y\cdot\overline{z} \\
S_7 &= \sum m(0,2,4) + d(1,3,6)
\end{aligned}
\tag{A.35}
$$

De functie voor S_7 is alleen in de verkorte notatie (of somnotatie) te schrijven.

Uitwerking opgave 3.10.
Om de product-van-maxterm-vorm te krijgen moeten we de nullen uit de functies nemen:

$$
\begin{aligned}
S_1 &= \prod M(0,2,4) = (x+y+z)\cdot(x+\overline{y}+z)\cdot(\overline{x}+y+z) \\
S_2 &= \prod M(3,5,6) = (x+\overline{y}+\overline{z})\cdot(\overline{x}+y+\overline{z})\cdot(\overline{x}+\overline{y}+z) \\
S_3 &= \prod M(0,1,2,3,4,5,6) = (x+y+z)\cdot(x+y+\overline{z})\cdot(x+\overline{y}+z)\cdot(x+\overline{y}+\overline{z})\cdot \\
&\qquad (\overline{x}+y+z)\cdot(\overline{x}+y+\overline{z})\cdot(\overline{x}+\overline{y}+z) \\
S_4 &= \prod M(0,4) = (x+y+z)\cdot(\overline{x}+y+z) \\
S_5 &= \prod M(0) = (x+y+z) \\
S_6 &= \prod M(1,3,5,7) = (x+y+\overline{z})\cdot(x+\overline{y}+\overline{z})\cdot(\overline{x}+y+\overline{z})\cdot(\overline{x}+\overline{y}+\overline{z}) \\
S_7 &= \prod M(1,3,5,6,7)\cdot D(1,3,6)
\end{aligned}
\tag{A.36}
$$

De functie voor S_7 is alleen in de verkorte notatie (of productnotatie) te schrijven.

Uitwerking opgave 3.11.
Hieronder is de waarheidstabel gegeven, zie tabel A.8

A.4 Uitwerkingen hoofdstuk 4

Tabel A.8: *Waarheidstabel.*

x	y	z	s	
0	0	0	-	don't care
0	0	1	1	$x = 0$ en $z = 1$
0	1	0	0	
0	1	1	1	$x = 0$ en $z = 1$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	-	don't care

Uitwerking opgave 4.1.
In figuur A.20 zijn de Karnaughdiagrammen met omrandingen te zien. Bij twee diagrammen zijn twee oplossingen mogelijk die leiden tot een minimale functie. Eén diagram kan niet vereenvoudigd worden.

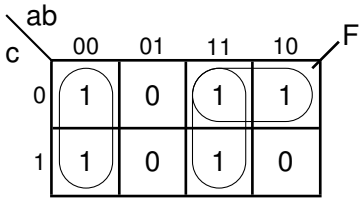
Uitwerking opgave 4.2.
We vullen de functie $f_{x,y,z} = \sum m(0,1,2,6)$ in een Karnaughdiagram in. Op de plaatsen van de mintermen 0, 1, 2 en 6 wordt een 1 ingevuld, de overige plaasten zijn logisch 0. Zie het Karnaughdiagram linksonder. In dit diagram zijn ook de plaatsen van de mintermen aangegeven met een indexcijfer. Het Karnaughdiagram is rechtsonder nog een keer getekend maar nu zijn de omrandingen zichtbaar. De functie kan vereenvoudigd worden tot $f_{x,y,z} = \overline{x} \cdot \overline{y} + y \cdot \overline{z}$.

De tweede functie is $f_{a,b,c} = \sum m(1,2,5,6,7)$ en wordt ingevuld in een Karnaughdiagram, zie linksonder. Rechtsonder is het diagram omrand. De vereenvoudigde functie is $f_{a,b,c} = \overline{b} \cdot c + b \cdot \overline{c} + a \cdot c$. In deze uitwerking is gekozen om minterm 7 samen te nemen met minterm 5, maar is ook mogelijk om minterm 7 samen te nemen met minterm 6. In dat geval wordt de term $a \cdot c$ vervangen door $a \cdot b$.

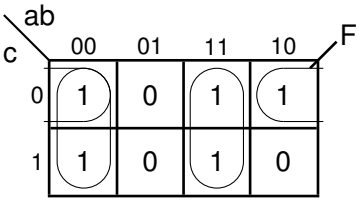
We vullen de functie $f_{s_2,s_1,s_0} = \sum m(0,3,6) + d(1,2)$ in een Karnaughdiagram in. Merk op dat twee functiewaarden als don't care zijn gespecificeerd. We vullen hiervoor een minteken ("dash") in. Zie het Karnaughdiagram linksonder. Bij het uitwerken van het Karnaugh-diagram worden de don't cares meegenomen als logische enen. Hierdoor wordt de functie het meest eenvoudig. De functie is $f_{s_2,s_1,s_0} = \overline{s_2} + s_1 \cdot \overline{s_0}$.

Hieronder zijn de schema's gegeven van de functies in AND-, OR- en NOT-poorten. De schema's zijn direct uit de functies getekend. Zie figuur A.24.

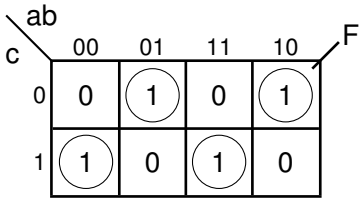
Uitwerking opgave 4.3.
Een *majority gate* is een schakeling waarbij de uitgang logisch 1 is als de meerderheid van de ingangen logisch 1 is, anders is de uitgang logisch 0. Voor een even aantal ingangen geldt dat er minimaal $n/2 + 1$ ingangen 1 moeten zijn. Voor vier ingangen is dat dus bij drie en vier enen. We kunnen direct het Karnaughdiagram invullen. Zie figuur A.25.



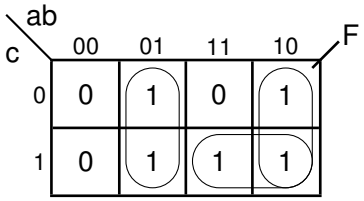
$$F = \overline{a} \cdot \overline{b} + a \cdot b + a \cdot \overline{c}$$



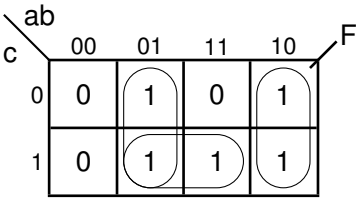
$$F = \overline{a} \cdot \overline{b} + a \cdot b + \overline{b} \cdot \overline{c}$$



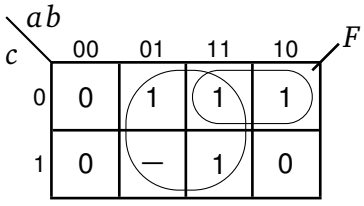
$$F = \overline{a} \cdot \overline{b} \cdot c + \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot \overline{c} + a \cdot b \cdot c$$



$$F = \overline{a} \cdot b + a \cdot \overline{b} + a \cdot c$$

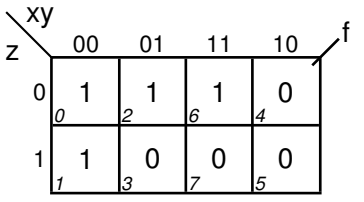


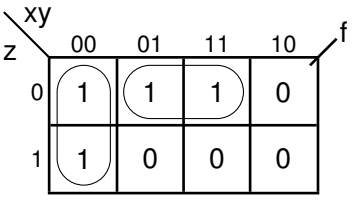
$$F = \overline{a} \cdot b + a \cdot \overline{b} + b \cdot c$$



$$F = b + a \cdot \overline{c}$$

Figuur A.20: *Uitwerkingen van de Karnaughdiagrammen.*





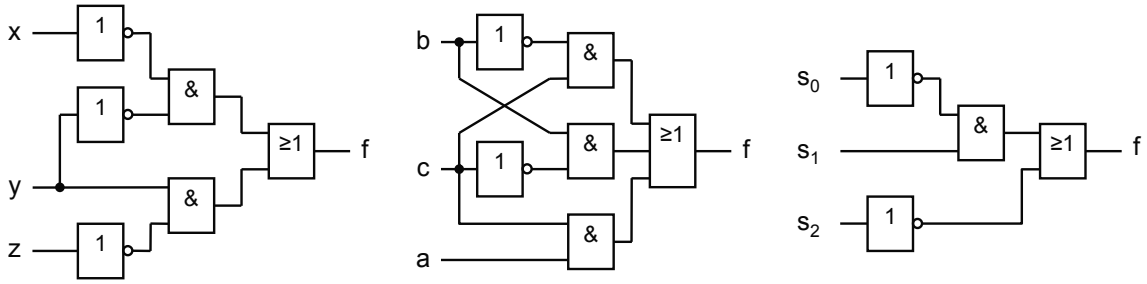
Figuur A.21: *Karnaughdiagrammen.*

De functie is:

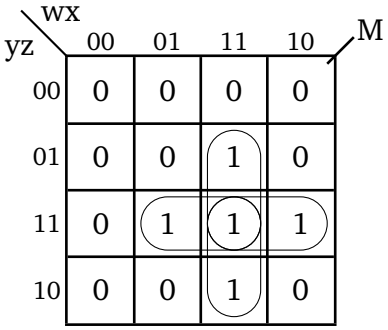
$$M = wxy + wxz + wyz + xyz \tag{A.37}$$

Figuur A.22: *Karnaughdiagrammen.*

Figuur A.23: *Karnaughdiagrammen.*

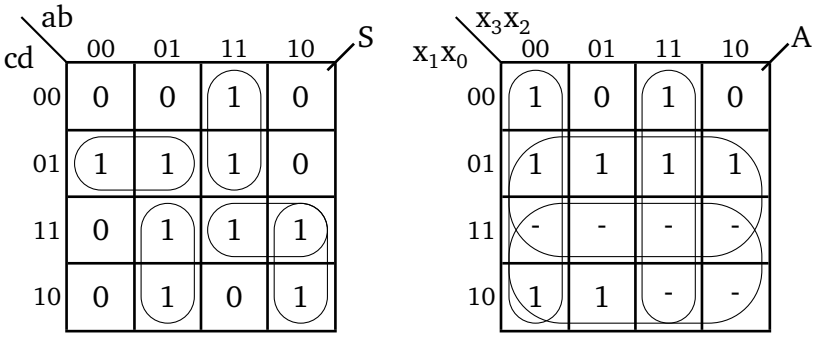


Figuur A.24: *Schema's van de functies.*



Figuur A.25: *Majority gate voor vier ingangen.*

De functie is ook wel te verklaren: het bevat alle mogelijke combinaties van drie variabelen. In het geval dat alle ingangen 1 zijn, zijn alle producttermen 1.



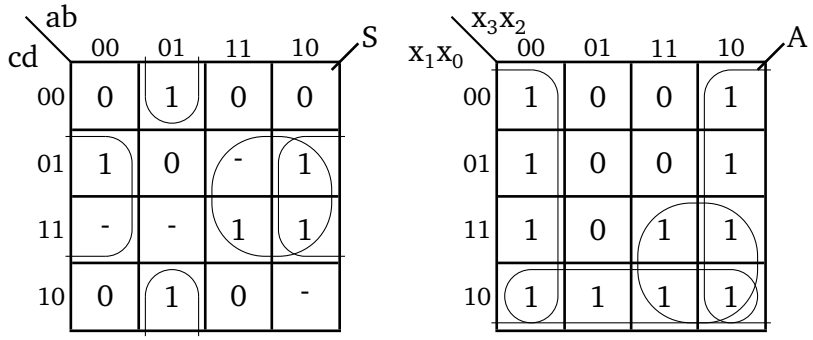
Figuur A.26: *Karnaughdiagrammen.*

Uitwerking opgave 4.4.

Hieronder zijn de Karnaughdiagrammen nogmaals getekend, maar nu met omrandingen.

De functie voor het linker Karnaughdiagram is $S = \overline{a}\,\overline{c}d + acd + ab\overline{c} + \overline{a}bc + a\overline{b}c$. Het is trouwens verleidelijk om de vier enen in het midden van het diagram samen te nemen tot de term bd. Maar zoals te zien is, worden deze enen afgedekt door de overige vier omrandingen.

De functie voor het rechter Karnaughdiagram is $A = x_0 + x_1 + \overline{x_3}\,\overline{x_2} + x_3x_2$. Hier zijn twee omrandingen van acht mintermen te zien. Die leveren per omranding een term van slechts één variabele op.



Figuur A.27: *Karnaughdiagrammen.*

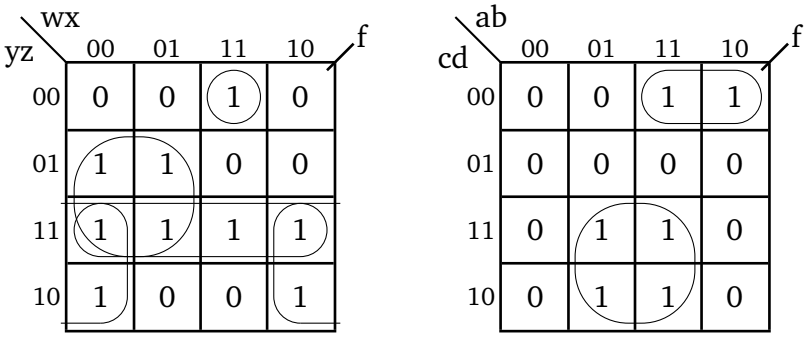
De functie voor het linker Karnaughdiagram is $S = ad + \overline{b}d + \overline{a}b\overline{d}$. Merk op dat de variabele c in deze functie niet voorkomt.

De functie van het rechter Karnaughdiagram is $A = \overline{x_2} + x_3x_1 + x_1\overline{x_0}$.

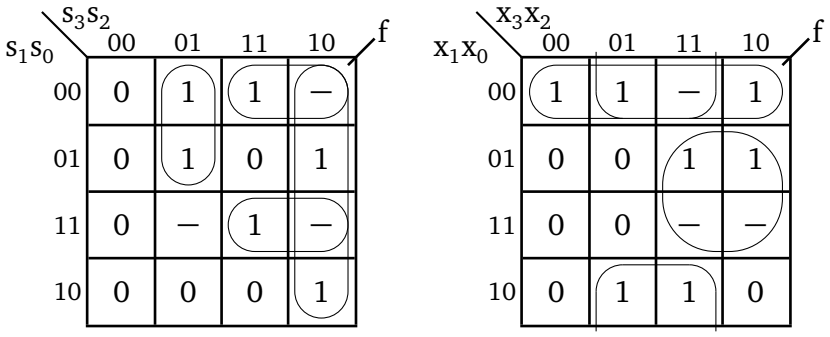
Uitwerking opgave 4.5.

De functie voor het linker Karnaughdiagram is $f_{w,x,y,z} = \overline{w}\cdot z + \overline{x}\cdot y + y\cdot z + w\cdot x\cdot \overline{y}\cdot \overline{z}$. De functie van het rechter Karnaughdiagram is $f_{a,b,c,d} = b\cdot c + a\cdot \overline{c}\cdot \overline{d}$.

Voor wat beteft het linker Karnaughdiagram zijn er meerdere functies mogelijk. Een mogelijke functie is $f_{s_3,s_2,s_1,s_0} = s_3\cdot \overline{s_2} + \overline{s_3}\cdot s_2\cdot \overline{s_0} + s_3\cdot \overline{s_1}\cdot \overline{s_0} + s_3\cdot s_1\cdot s_0$. De functie f_{x_3,x_2,x_1,x_0}



Figuur A.28: *Karnaughdiagrammen.*



Figuur A.29: *Karnaughdiagrammen.*

is in productnotatie gegeven. Dat zijn de nullen in een waarheidstabel of Karnaughdiagram. De don't cares moeten gewoon worden ingevuld. De functie van het rechter Karnaughdiagram is $f_{x_3,x_2,x_1,x_0} = \overline{x_1}\cdot\overline{x_0} + x_2\cdot\overline{x_0} + x_3\cdot x_1$. I.p.v. productterm $x_3\cdot x_1$ kan ook productterm $x_3\cdot\overline{x_1}$ genomen worden.

Uitwerking opgave 4.6.

De combinaties van de functie $f_{w,x,y,z} = \sum m(1,2,3,5,7,10,11,12,15)$ zijn in tabel A.9 te zien. Uit kolom I volgt direct dat minterm 12 niet met andere te combineren is. Dit is dus een priemimplicant.

We hebben nu de volgende priemimplicanten gevonden:

$$1110 = A \qquad -010 = C$$
$$0--1 = B \qquad --11 = D$$

We zetten nu de dekkingstabel op voor de priemimplicanten, zie tabel A.10. We geven met een omcirkelde x aan of de priemimplicant essentieel is. Een priemimplicant is essentieel als er in een kolom van een minterm maar één x voorkomt. Deze minterm is namelijk alleen te combineren met de overige mintermen van de priemimplicant.

Uit de tabel volgt dat alle priemimplicanten essentieel zijn. We kunnen nog de Petrick-

Tabel A.9: *Uitwerking van de tabellen.*

Kolom I.

	w	x	y	z	m.t.	
Groep 1	0	0	0	1	1	✓
	0	0	1	0	2	✓
Groep 2	0	0	1	1	3	✓
	0	1	0	1	5	✓
	1	0	1	0	10	✓
	1	1	0	0	12	A
Groep 3	0	1	1	1	7	✓
	1	0	1	1	11	✓
Groep 4	1	1	1	1	15	✓

Kolom II.

w	x	y	z	m.t.	
0	0	–	1	1,2	✓
0	–	0	1	1,5	✓
0	0	1	–	2,3	✓
–	0	1	0	2,10	✓
0	–	1	1	3,7	✓
–	0	1	1	3,11	✓
0	1	–	1	5,7	✓
1	0	1	–	10,11	✓
–	1	1	1	7,15	✓
1	–	1	1	14,15	✓

Kolom III.

w	x	y	z	m.t.	
0	–	–	1	1,3,5,7	B
0	–	–	1	1,5,3,7	
–	0	1	–	2,3,10,11	C
–	0	1	–	2,10,3,11	
–	–	1	1	3,7,11,15	D
–	–	1	1	3,11,7,15	

Tabel A.10: *Dekkingstabel van de priemplicanten.*

	m_1	m_2	m_3	m_5	m_7	m_{10}	m_{11}	m_{12}	m_{15}
A								(x)	
B	(x)		x	(x)	x				
C		(x)	x			(x)	x		
D			x		x		x		(x)

functie opstellen:

$$\begin{aligned} P_{f_{w,x,y,z}} &= B\cdot C\cdot(B+C+D)\cdot B\cdot(B+D)\cdot C\cdot(C+D)\cdot A\cdot D \\ &= B\cdot C\cdot B\cdot C\cdot A\cdot D \\ &= A\cdot B\cdot C\cdot D \end{aligned} \tag{A.38}$$

Zoals te zien is, zijn alle priemimplicanten nodig voor de dekking van de functie. Dus:

$$f_{w,x,y,z} = w\cdot x\cdot\overline{y}\cdot\overline{z} + \overline{w}\cdot z + \overline{x}\cdot y + y\cdot z \tag{A.39}$$

De combinaties van de functie $f_{a,b,c,d} = \sum m(6,7,8,12,14,15)$ zijn in tabel A.11 te zien.

We hebben nu de volgende priemtimplicanten gevonden:

$$1-00 = A \qquad 11-0 = B \qquad -11- = C$$

We zetten nu de dekkingstabel op voor de priemimplicanten, zie tabel A.12. We geven met een omcirkelde x aan of de priemimplicant essentieel is. Een priemimplicant is essentieel als er in een kolom van een minterm maar één x voorkomt. Deze minterm is namelijk alleen te combineren met de overige mintermen van de priemimplicant.

Uit de tabel volgt dat de priemimplicanten A en C essentieel zijn. Te zien is dat de mintermen gedekt door priemimplicant B gedekt worden door A en C. We kunnen nog

Tabel A.11: *Uitwerking van de tabellen.*

Kolom I.

	a	b	c	d	m.t.	
Groep 1	1	0	0	0	8	✓
	0	1	1	0	6	✓
Groep 2	1	1	0	0	12	✓
Groep 3	0	1	1	1	7	✓
	1	1	1	0	14	✓
Groep 4	1	1	1	1	15	✓

Kolom II.

a	b	c	d	m.t.	
1	–	0	0	8,12	*A*
0	1	1	–	6,7	✓
–	1	1	0	6,14	✓
1	1	–	0	12,14	*B*
–	1	1	1	7,15	✓
1	1	1	–	14,15	✓

Kolom III.

a	b	c	d	m.t.	
–	1	1	–	6,7,14,15	*C*
–	1	1	–	6,14,7,15	

Tabel A.12: *Dekkingstabel van de priemplicanten.*

	m_6	m_7	m_7	m_{12}	m_{14}	m_{15}
A			Ⓧ	x		
B				x	x	
C	Ⓧ	Ⓧ			x	Ⓧ

de Petrick-functie opstellen:

$$\begin{aligned} P_{f_{a,b,c,d}} &= C \cdot C \cdot A \cdot (A+B) \cdot (B+C) \cdot C \\ &= C \cdot C \cdot A \cdot C \\ &= A \cdot C \end{aligned} \tag{A.40}$$

Voor de functie vinden we dus:

$$f_{a,b,c,d} = a \cdot \bar{c} \cdot \bar{d} + b \cdot c \tag{A.41}$$

De combinaties van de functie $f_{s_3,s_2,s_1,s_0} = \sum m(4,5,9,10,12) + d(7,8,11,15)$ zijn in tabel A.13 te zien.

We hebben nu de volgende priemimplicanten gevonden:

$$\begin{aligned} 010- &= A & -111 &= E \\ -100 &= B & 1-11 &= F \\ 1-00 &= C & 10-- &= G \\ 01-1 &= D \end{aligned}$$

We stellen een dekkingstabel op van alle mintermen gecombineerd met de priemimplicanten maar laten de mintermen die als don't care te boek staan weg. Zie tabel A.14.

Te zien is dat priemimplicanten E en F geen mintermen dekken; alle bijhorende mintermen zijn don't care. We kunnen E en F dus direct schrappen.

Tabel A.13: *Uitwerking van de tabellen.*

Kolom I.

	s_3	s_2	s_1	s_0	m.t.	
Groep 1	0	1	0	0	4	✓
	1	0	0	0	8	✓
Groep 2	0	1	0	1	5	✓
	1	0	0	1	9	✓
	1	0	1	0	10	✓
	1	1	0	0	12	✓
Groep 3	0	1	1	1	7	✓
	1	1	0	1	11	✓
Groep 4	1	1	1	1	15	✓

Kolom II.

s_3	s_2	s_1	s_0	m.t.	
0	1	0	–	4,5	A
–	1	0	0	4,12	B
1	0	0	–	8,9	✓
1	0	–	0	8,10	✓
1	–	0	0	8,12	C
0	1	–	1	5,7	D
1	0	–	1	9,11	✓
1	0	1	–	10,11	✓
–	1	1	1	7,15	E
1	–	1	1	11,15	F

Kolom III.

s_3	s_2	s_1	s_0	m.t.	
1	0	–	–	8,9,10,11	G
1	0	–	–	8,10,9,11	

Tabel A.14: *Dekkingstabel van de priemplicanten.*

	m_4	m_5	m_9	m_{10}	m_{12}
A	x	x			
B	x				x
C					x
D		x			
E					
F					
G			ⓧ	ⓧ	

Uit de tabel volgt verder dat priemimplicant G essentieel is We kunnen nu de Petrick-functie opstellen:

$$\begin{aligned} P_{f_{s_3,s_2,s_1,s_0}} &= (A+B)\cdot(A+D)\cdot G\cdot G\cdot(B+C) \\ &= A\cdot B\cdot G + A\cdot C\cdot G + B\cdot D\cdot G + B\cdot C\cdot D\cdot G \end{aligned} \tag{A.42}$$

De laatste term valt af want die bestaat uit vier priemimplicanten. We kunnen nu kiezen tussen de eerste drie.

$$\begin{aligned} f_{s_3,s_2,s_1,s_0} &= A+B+G = \overline{s_3}\cdot s_2\cdot\overline{s_1} + s_2\cdot\overline{s_1}\cdot\overline{s_0} + s_3\cdot\overline{s_2} \\ f_{s_3,s_2,s_1,s_0} &= A+C+G = \overline{s_3}\cdot s_2\cdot\overline{s_1} + s_3\cdot\overline{s_1}\cdot\overline{s_0} + s_3\cdot\overline{s_2} \\ f_{s_3,s_2,s_1,s_0} &= B+D+G = s_2\cdot\overline{s_1}\cdot\overline{s_0} + \overline{s_3}\cdot s_2\cdot s_1 + s_3\cdot\overline{s_2} \end{aligned} \tag{A.43}$$

Elk van de drie functies voldoet aan de minimale functie.

De functie $f_{x_3,x_2,x_1,x_0} = \prod M(1,2,3,5,7,10) + D(11,12,15)$ staat in POS-vorm en schrijven we eerst om naar SOP-vorm: $f_{x_3,x_2,x_1,x_0} = \sum m(0,4,6,8,9,13,14) + d(11,12,15)$. De combinaties zijn te zien in tabel A.15.

We stellen een dekkingstabel op van alle mintermen gecombineerd met de priemimplicanten maar laten de mintermen die als don't care te boek staan weg. Zie tabel A.16.

Tabel A.15: *Uitwerking van de tabellen.*

Kolom I.

	x_3	x_2	x_1	x_0	m.t.	
Groep 0	0	0	0	0	0	✓
Groep 1	0	1	0	0	4	✓
	1	0	0	0	8	✓
Groep 2	0	1	1	0	6	✓
	1	0	0	1	9	✓
	1	1	0	0	12	✓
Groep 3	1	0	1	1	11	✓
	1	1	0	1	13	✓
	1	1	1	0	14	✓
Groep 4	1	1	1	1	15	✓

Kolom II.

x_3	x_2	x_1	x_0	m.t.	
0	–	0	0	0,4	✓
–	0	0	0	0,8	✓
0	1	–	0	4,6	✓
–	1	0	0	4,12	✓
1	0	0	–	8,9	✓
1	–	0	0	8,12	✓
–	1	1	0	6,14	✓
1	0	–	1	9,11	✓
1	–	0	1	9,13	✓
1	1	0	–	12,13	✓
1	1	–	0	12,14	✓
1	–	1	1	11,15	✓
1	1	–	1	13,15	✓
1	1	1	–	14,15	✓

Kolom III.

s_3	s_2	s_1	s_0	m.t.	
–	–	0	0	0,4,8,12	*A*
–	–	0	0	0,8,4,12	
–	1	–	0	4,6,12,14	*B*
–	1	–	0	4,12,6,14	
1	–	0	–	8,9,13,12	*C*
1	–	0	–	8,12,9,13	
1	–	–	1	9,11,13,15	*D*
1	–	–	1	9,13,11,15	
1	1	–	–	12,13,14,15	*E*
1	1	–	–	12,14,13,15	

Tabel A.16: *Dekkingstabel van de priemplicanten.*

	m_0	m_4	m_6	m_8	m_9	m_{13}	m_{14}
A	ⓧ	x		x			
B		x	ⓧ				x
C				x	x	x	
D					x	x	
E						x	x

We stellen nu de Petrick-functie op:

$$\begin{aligned} P_{f_{x_3,x_2,x_1,x_0}} &= A\cdot(A+B)\cdot B\cdot(A+C)\cdot(C+D)\cdot(C+D+E) \\ &= A\cdot B\cdot(C+D) \\ &= A\cdot B\cdot C + A\cdot B\cdot D \end{aligned} \tag{A.44}$$

We kunnen nu kiezen uit twee functies:

$$\begin{aligned} f_{x_3,x_2,x_1,x_0} &= A+B+C = \overline{x_1}\cdot\overline{x_0} + x_2\cdot\overline{x_0} + x_3\cdot\overline{x_1} \\ f_{x_3,x_2,x_1,x_0} &= A+B+D = \overline{x_1}\cdot\overline{x_0} + x_2\cdot\overline{x_0} + x_3\cdot x_0 \end{aligned} \tag{A.45}$$

Uitwerking opgave 4.7.
Een multiplexer kan gezien worden als een schakeling die een som-van-mintermen kan nabootsen. Een multiplexer met stuuringangen s_1 en s_0 en data-ingangen i_0, i_1, i_2 en i_3 heeft de logische functie $F = \overline{s_1}\,\overline{s_0}i_0 + \overline{s_1}s_0i_0 + s_1\overline{s_0}i_2 + s_1s_0i_3$ waarbij F natuurlijk de uitgang is.

In de twee multiplexer-schakelingen wordt s_1 verbonden met Y en s_0 verbonden met Z. Aan de data-ingangen worden nu logische constanten (0 en 1), X of $\overline{X}$ aangeboden.

Kijken we naar functie S dan zien we dat i_0 wordt verbonden met X, i_1 en i_2 met $\overline{X}$ en i_3 met X. De functie voor S is dan $S = \overline{Y}\,\overline{Z}X + \overline{Y}Z\overline{X} + Y\overline{Z}\,\overline{X} + YZX$. We kunnen de functie herschrijven zodat X vooraan staat in de termen.

Kijken we naar functie C dan zien we dat i_0 wordt verbonden met 0, i_1 en i_2 met X en i_3 met 1. De functie voor S is dan $C = \overline{Y}\,\overline{Z}0 + \overline{Y}ZX + Y\overline{Z}X + YZ1$. We kunnen de functie herschrijven als $C = YZ + X\overline{Y}Z + XY\overline{Z}$.

Hieronder zijn de waarheidstabellen gegeven van beide functies. Een geoefend lezer ziet in S de functie voor het sombit van een full-adder en in C de *carry-out*-bit van een full-adder. Merk op dat de volgorde van de variabelen in de tabel afwijkt van wat gebruikelijk is; variabele X is nu het minst significant. Op deze manier kunnen de functiewaarden van C en S goed vergeleken worden met de aan de data-ingangen aangeboden waarden van 0, 1, X en $\overline{X}$.

Tabel A.17: *Waarheidstabel.*

Y	Z	X	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Uitwerking opgave 4.8.
In tabel A.18 is de waarheidstabel te zien. De codes 1010 t/m 1111 zijn als don't cares gespecificeerd omdat deze codes geen decimaal cijfer voorstellen.

De uitgewerkte Karnaughdiagrammen zijn te zien in de figuur A.30.

Voor de segmenten vinden we de volgende functies:

$$\begin{aligned}
a &= x_3 + x_1 + \overline{x_2}\cdot\overline{x_0} + x_2\cdot x_0 \\
b &= \overline{x_2} + \overline{x_1}\cdot\overline{x_0} + x_1\cdot x_0 \\
c &= x_2 + \overline{x_1} + x_0 \\
d &= x_3 + \overline{x_2}\cdot\overline{x_0} + \overline{x_2}\cdot x_1 + x_2\cdot\overline{x_1}\cdot x_0 + x_1\cdot\overline{x_0} \\
e &= \overline{x_2}\cdot\overline{x_0} + x_1\cdot\overline{x_0} \\
f &= x_3 + \overline{x_1}\cdot\overline{x_0} + x_2\cdot\overline{x_1} + x_2\cdot\overline{x_0} \\
g &= x_3 + x_1\cdot\overline{x_0} + x_2\cdot\overline{x_1} + \overline{x_2}\cdot x_1
\end{aligned} \tag{A.46}$$

Tabel A.18: *Waarheidstabel van een zevensegmentendecoder.*

x_3	x_2	x_1	x_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
	overig			—	—	—	—	—	—	—

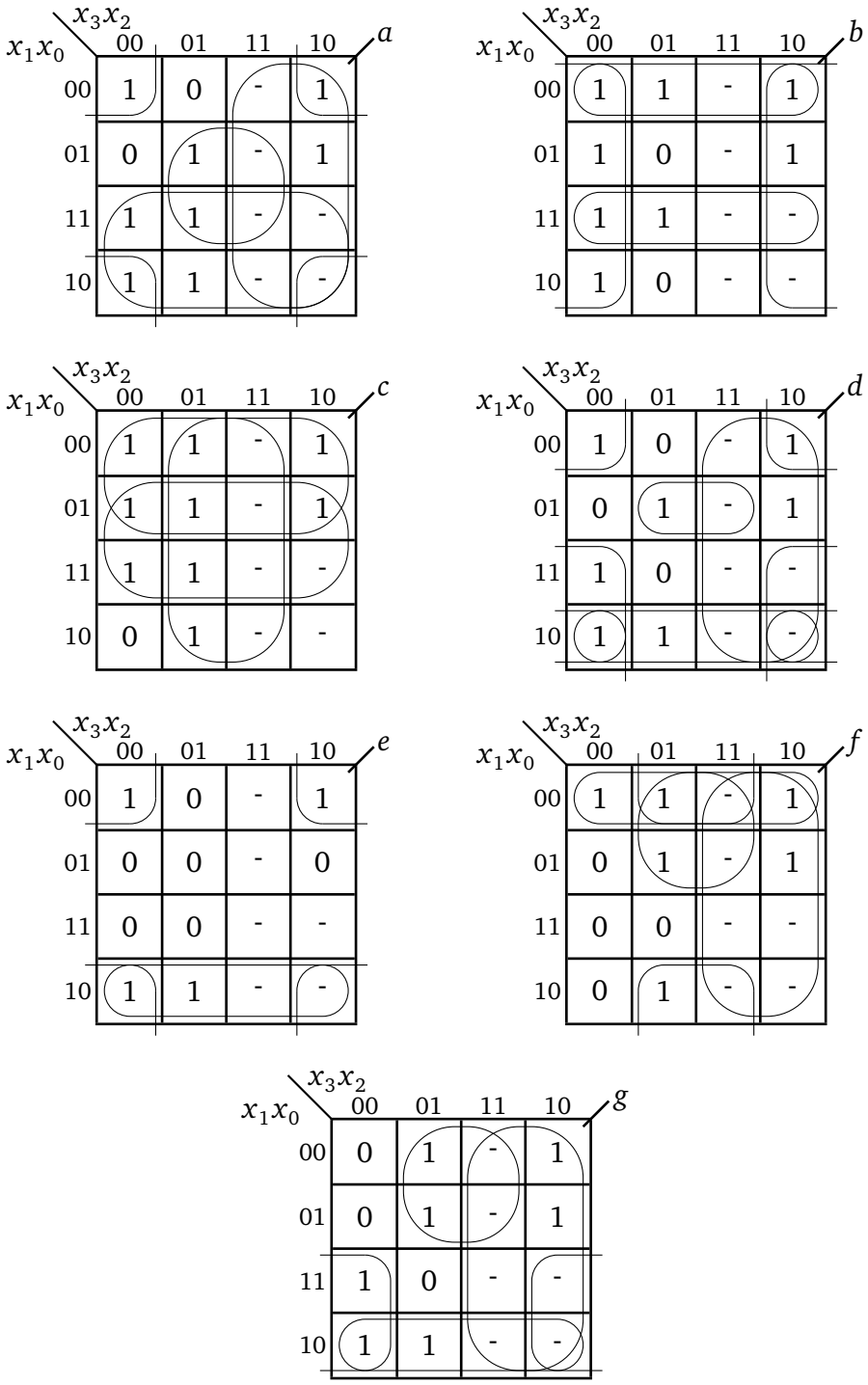
Uitwerking opgave 4.9.

De schakeling links in de figuur is een 2-input NAND-poort. Dat is als volgt te beredeneren. Als beide ingangen 0 zijn, dan geleiden T_1 en T_2. T_3 en T_4 sperren. Er is zodoende een elektrisch pad van U_{DD} naar de uitgang. De uitgang is dus 1. Als $a = 0$ en $b = 1$, dan geleidt T_1 en spert T_2. T_3 spert en T_4 geleidt. Er is weer een pad van U_{DD} naar de uitgang, de uitgang is 1. Eenzelfde situatie doet zich voor als $a = 1$ en $b = 0$. Als de ingangen beide 1 zijn, dan sperren T_1 en T_2 en geleiden T_3 en T_4. Er is nu een pad naar de referentie, de uitgang is 0. Alle combinaties zijn nog eens weergegeven in de onderstaande tabel. De schakeling werkt dus als 2-input NAND. Merk op dat voor deze NAND vier transistoren nodig zijn.

Tabel A.19: *Waarheidstabel.*

a	b	T_1	T_2	T_3	T_4	f
0	0	geleidt	geleidt	spert	spert	1
0	1	geleidt	spert	spert	geleidt	1
1	0	spert	geleidt	geleidt	spert	1
1	1	spert	spert	geleidt	geleidt	0

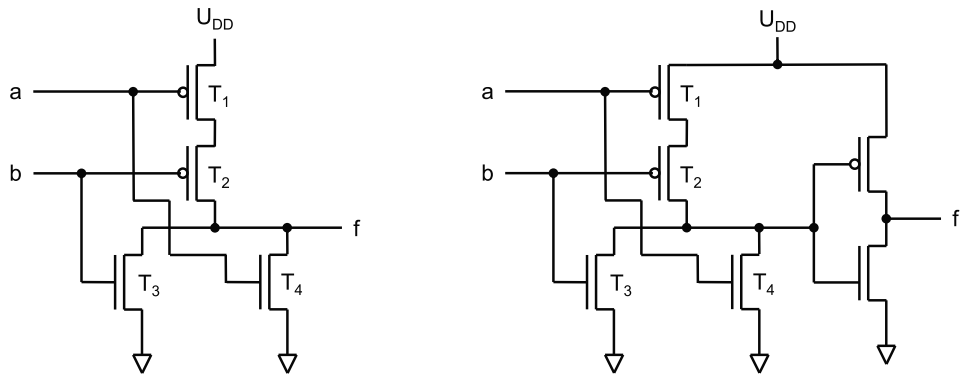
De schakeling rechts in de figuur is een samenstelling van de hierboven beschreven NAND-poort en een NOT-poort. De schakeling werkt in zijn geheel als een AND-poort. Voor deze poort zijn zes transistoren nodig.



Figuur A.30: *Uitgewerkte Karnaughdiagrammen van de functies.*

Uitwerking opgave 4.10.

De hieronder gepresenteerde schakeling is een NOR-poort én een OR-poort met twee ingangen. Zoals te zien is wordt een OR-poort gerealiseerd door een NOR-poort en een inverter. De NOR-poort kan met vier transistoren worden opgebouwd. Voor de OR-poort zijn zes transistoren nodig.



Figuur A.31: *Schema van een NOR-poort (links) en een OR-poort (rechts).*

A.5 Uitwerkingen hoofdstuk 5

Uitwerking opgave 5.1.
Hieronder de optellingen mét carry's. De eerste optelling levert een *overflow* op omdat het antwoord niet in zeven bits past.

1110110		10100		11110		carry's
1011001		01010		01111		getal A
0111011	+	01010	+	00001	+	getal B
1)0010100		10100		10000		resultaat

Figuur A.32: *Enkele optellingen met carry's.*

Uitwerking opgave 5.2.
Hieronder de aftrekkingen mét borrows. De derde aftrekking levert een *underflow* op omdat het antwoord niet in zes bits past.

1111100		111100		1)000010		borrows
1011001		110000		010110		getal A
0111011	−	010110	−	100101	−	getal B
0011110		011010		1)110001		resultaat

Figuur A.33: *Enkele aftrekkingen met borrows.*

Uitwerking opgave 5.3.
Het bewijs is niet zo moeilijk. Er wordt gebruik gemaakt van een absorptiewet en de commutatieve wetten.

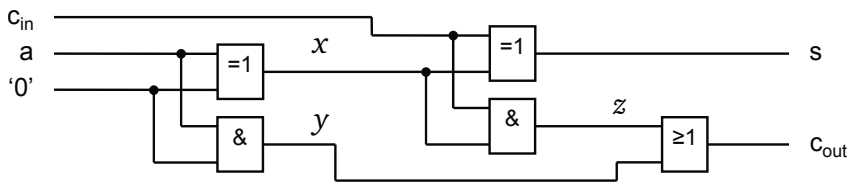
$$\begin{aligned} c_{out} &= \overline{c_{in}} \cdot (a \cdot b) + c_{in} \cdot (a + b) \\ &= \overline{c_{in}} \cdot a \cdot b + c_{in} \cdot a + c_{in} \cdot b \\ &= a \cdot (\overline{c_{in}} \cdot b + c_{in}) + c_{in} \cdot b \\ &= a \cdot (b + c_{in}) + c_{in} \cdot b \\ &= a \cdot b + a \cdot c_{in} + b \cdot c_{in} \end{aligned} \tag{A.47}$$

Uitwerking opgave 5.4.
Dit is wel een lastige opgave. Van cruciaal belang is dat de term $a \cdot b$ uitgebreid moet worden naar $a \cdot b + a \cdot b$. Daarna kan de absorptiewet worden toegepast. Zie hieronder.

$$\begin{aligned} c_{out} &= a \cdot b + c_{in} \cdot (a \oplus b) \\ &= a \cdot b + c_{in} \cdot \overline{a} \cdot b + c_{in} \cdot a \cdot \overline{b} \\ &= a \cdot b + a \cdot b + c_{in} \cdot \overline{a} \cdot b + c_{in} \cdot a \cdot \overline{b} \\ &= (a + c_{in} \cdot \overline{a}) \cdot b + a \cdot (b + c_{in} \cdot \overline{b}) \\ &= (a + c_{in}) \cdot b + a \cdot (b + c_{in}) \\ &= a \cdot b + c_{in} \cdot b + a \cdot b + a \cdot c_{in} \\ &= a \cdot b + a \cdot c_{in} + b \cdot c_{in} \end{aligned} \tag{A.48}$$

Uitwerking opgave 5.5.
Als de functies van s en c_{out} vanuit de 0-en zouden worden gemaakt, levert dat net zulke grote functies op als wanneer ze vanuit de 1-en zouden worden gemaakt. Dat is ook niet zo verwonderlijk. Voor s geldt dat (vanuit de 1-en gezien) het aantal 1-en oneven moet zijn, dus dat geldt dan ook voor de 0-en. Voor c_{out} geldt dat (weer vanuit de 1-en gezien) het aantal 1-en twee of meer moet zijn en ook dat geldt dan natuurlijk voor de 0-en.

Uitwerking opgave 5.6.
Hieronder is het schema gegeven met daarin interne signaalnamen.



Figuur A.34: *Schema van een full adder.*

De ingang waar de constante 0 aan verbonden is herkennen we als de b-ingang van een full-adder. Deze ingang koppelen we ook nog eens aan de constante 1 en rekenen beide schakelingen door.

De functies voor $b = 0$

$$\begin{aligned} x &= a \oplus 0 & &= a \\ y &= a \cdot 0 & &= 0 \\ z &= x \cdot c_{in} & &= a \cdot c_{in} \\ s &= c_{in} \oplus x & &= c_{in} \oplus a \\ c_{out} &= y + z & &= a \cdot c_{in} \end{aligned}$$

De functies voor $b = 1$

$$\begin{aligned} x &= a \oplus 1 & &= \overline{a} \\ y &= a \cdot 1 & &= a \\ z &= x \cdot c_{in} & &= \overline{a} \cdot c_{in} \\ s &= c_{in} \oplus x & &= c_{in} \oplus \overline{a} \\ c_{out} &= y + z & &= a + c_{in} \end{aligned} \tag{A.49}$$

In de functies aan de linkerkant herkennen we de functies van een half-adder.

Uitwerking opgave 5.7.

De vermenigvuldiger vermenigvuldigt twee *unsigned* getallen van twee bits met elkaar. De rekenkundige functie is dan $M = A \times B$ waarbij A samengesteld is uit de bits a_1 en a_0 en B uit b_1 en b_0. Een 2x2 bits vermenigvuldiging levert een antwoord van maximaal vier bits dus M is samengesteld uit de bits m_3, m_2, m_1 en m_0. De grootste waarde van M is trouwens 9, want $A = 3$ en $B = 3$ levert $M = 3 \times 3 = 9$ op.

We stellen een waarheidstabel op met a_1, a_0, b_1 en b_0 op als ingangen en m_3, m_2, m_1 en m_0 als de uitgangen.

Tabel A.20: *Waarheidstabel van een 2x2-bit vermenigvuldiger.*

a_1	a_0	b_1	b_0	m_3	m_2	m_1	m_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1

a_1	a_0	b_1	b_0	m_3	m_2	m_1	m_0
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

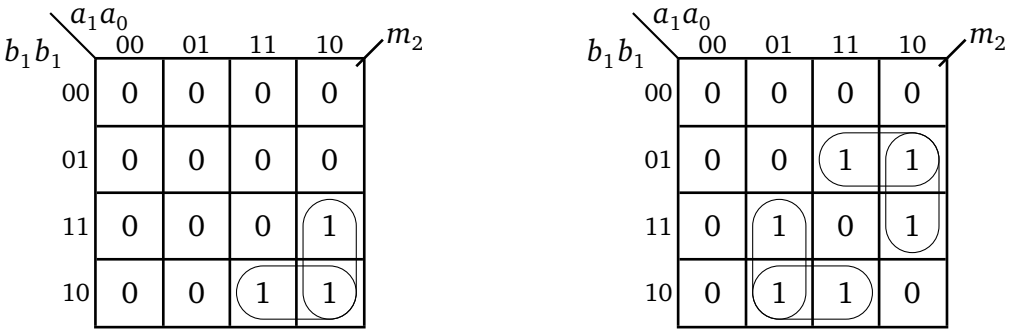
Snel is te zien dat de functie voor m_3 een AND-operatie is van alle ingangen zodat $m_3 = a_1 \cdot a_0 \cdot b_1 \cdot b_0$. De functie voor m_0 is ook redelijk snel te ontdekken: $m_0 = 1$ als $a_0 = 1$ én $b_0 = 1$ dus $m_0 = a_0 \cdot b_0$.

Voor de andere twee functies stellen een Karnaughdiagram op.

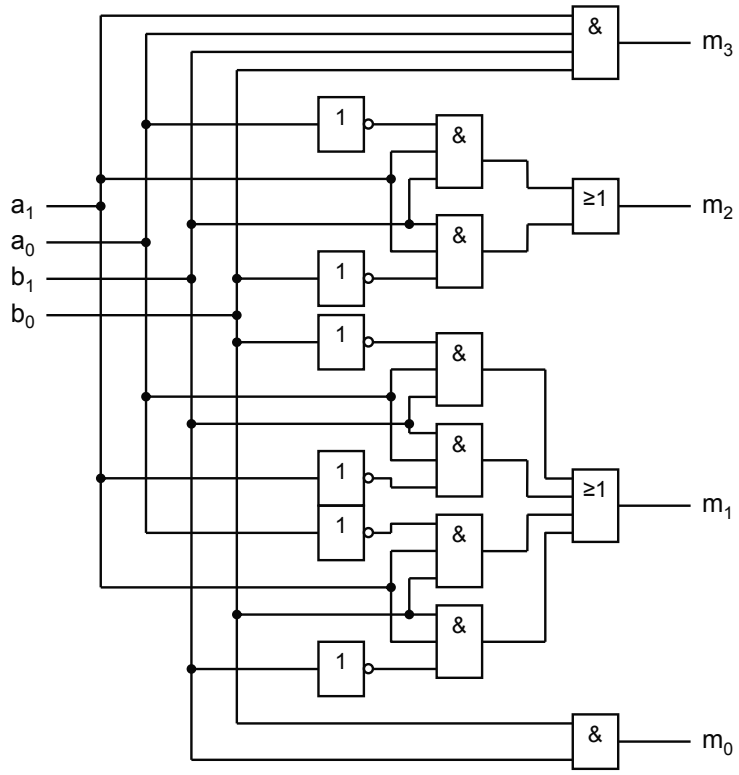
De vier functies voor deze schakeling zijn:

$$\begin{aligned} m_3 &= a_1 \cdot a_0 \cdot b_1 \cdot b_0 \\ m_2 &= a_1 \cdot b_1 \cdot \overline{b_0} + a_1 \cdot \overline{a_0} \cdot b_1 \\ m_1 &= \overline{a_1} \cdot a_0 \cdot b_1 + a_0 \cdot b_1 \cdot \overline{a_0} + a_1 \cdot \overline{a_0} \cdot b_0 + a_1 \cdot \overline{b_1} \cdot b_0 \\ m_0 &= a_0 \cdot b_0 \end{aligned}$$

Het bijbehorende schema met AND-, OR- en NOT-poorten is hieronder gegeven.



Figuur A.35: *Karnaughdiagrammen.*



Figuur A.36: *Schema van de 2x2-bit vermenigvuldiger.*

Uitwerking opgave 5.8.

Als geldt dat twee getallen $A = a_3a_2a_1a_0$ en $B = b_3b_2b_1b_0$ aan elkaar gelijk zijn, dan moeten cijfers op identieke posities binnen de getallen aan elkaar gelijk zijn:

$$A = B \quad \rightarrow \quad a_3 = b_3 \wedge a_2 = b_2 \wedge a_1 = b_1 \wedge a_0 = b_0 \tag{A.50}$$

Het testen of twee bits gelijk zijn kan met de EXNOR-functie. De schakelfunctie is:

$$E = \overline{a_3 \oplus b_3} \cdot \overline{a_2 \oplus b_2} \cdot \overline{a_1 \oplus b_1} \cdot \overline{a_0 \oplus b_0} \tag{A.51}$$

of, m.b.v. De Morgan:

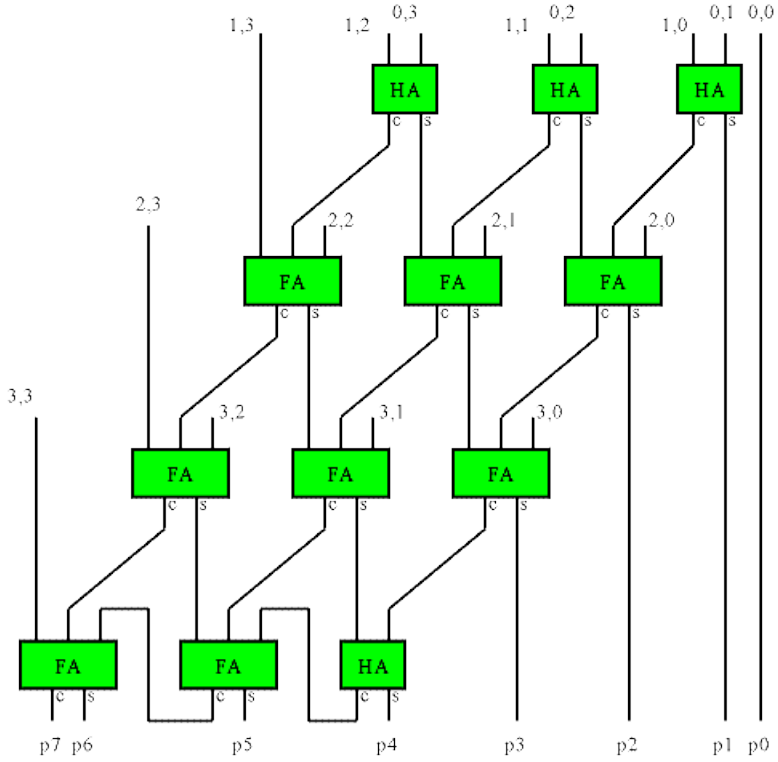
$$E = \overline{(a_3 \oplus b_3) + (a_2 \oplus b_2) + (a_1 \oplus b_1) + (a_0 \oplus b_0)} \tag{A.52}$$

Uitwerking opgave 5.9.

Hieronder is het schema gegeven van 4x4 carry save multiplier. Bij een “normale” multiplier wordt het optellen uitgevoerd met carry ripple adders waarbij de carry steeds wordt doorgegeven aan de hogere macht van dezelfde opteller.

Bij de carry save multiplier wordt de carry van een 1-bit full adder circuit doorgegeven aan de hogere macht van de *volgende* 4-bit opteller.

Er zijn nu wel meer rijen nodig, maar minder kolommen.



Figuur A.37: *Opbouw van een carry save vermenigvuldiger.*

Uitwerking opgave 5.10.

Voor een 5x3-bit vermenigvuldig zijn twee 5-bit optellers, tenminste als de optellers worden uitgevoerd als *carry ripple adders*.

Uitwerking opgave 5.11.

De getallen moeten omgerekend worden naar binaire equivalenten. Van negatieve getallen moet eerst de positieve variant worden omgerekend. Van de getallen +5 en −5 kan direct het binaire equivalent worden uitgerekend (althans van +5).

Het getal +5 als binair getal is 0101_2. Let hierbij op de leidende 0 om het getal positief te houden. Het getal staat nu in 4 bits genoteerd, we moeten het dus aanvullen met meer nullen (het getal is immers positief). Dat levert het binaire getal 00000101_2 op. Dit staat nu al in 2’s complement.

Het getal -5 moet bepaald worden volgens de 'flip-bit, add-one'-methode. We weten al hoe $+5$ in 8-bits formaat geschreven wordt.

$$\begin{array}{rl} 00000101 & (+5) \\ \\ 11111010 & \text{flip bits} \\ \qquad\qquad 1 & + \\ \hline 11111011 & (-5) \end{array}$$

Figuur A.38: *De binaire representatie in 8 bits van de getallen* $+5$ *en* -5*.*

Het getal -5 geschreven in 8 bits 2's complement is 11111011_2.

Eerst moet het getal $+100$ omgezet worden in het binaire stelsel (we laten dat als oefening voor de lezer). Dat levert het binaire getal 01100100_2. Let hierbij op de leidende 0 om het getal positief te houden. Dit getal staat al in 8-bits formaat, dus aanvullen is niet nodig. Omzetten naar de negatieve variant:

$$\begin{array}{rl} 01100100 & (+100) \\ \\ 10011011 & \text{flip bits} \\ \qquad\qquad 1 & + \\ \hline 10011100 & (-100) \end{array}$$

Figuur A.39: *De binaire representatie in 8 bits van de getallen* $+100$ *en* -100*.*

Het getal -100 geschreven in 8 bits 2's complement is 10011100_2.

Het getal -64 moet omgezet worden naar 2's complement. Daarvoor moet eerst de positieve variant 64 omgezet worden naar binair (dat laten we weer over als oefening voor de lezer). Dit levert het getal 01000000_2 op (let hierbij op de leidende 0 om het getal positief te houden. Het getal is al in 8 bits genoteerd, er is geen tekenuitbreiding nodig. Omzetten naar de negatieve variant:

$$\begin{array}{rl} 01000000 & (+64) \\ \\ 10111111 & \text{flip bits} \\ \qquad\qquad 1 & + \\ \hline 11000000 & (-64) \end{array}$$

Figuur A.40: *De binaire representatie in 8 bits van de getallen* $+64$ *en* -64*.*

Het getal -64 geschreven in 8 bits 2's complement is 1100000_2.

Het getal $+25$ is volgens de bekende procedure om te zetten naar het binaire equivalent 011001_2. Uitbreiden naar 8 bits levert 00011001_2 op.

Van het getal -25 wordt alleen het antwoord gegeven: 11100111_2.

Uitwerking opgave 5.12.
Bij deze opgave moet een aantal optellingen en aftrekkingen worden gedaan. Optellingen kunnen direct uitgevoerd worden d.m.v. de bekende kolomsgewijze optelling. Aftrekkingen kunnen worden omgezet in optellingen door bekende vergelijking

$$A - B = A + (-B) \tag{A.53}$$

Hiervoor is wel de negatieve representatie van B nodig. Dit kan eenvoudig in de 2's complement-representatie. Daarnaast moet een aantal getallen met meer bits worden geschreven zodat de op te tellen getallen evenveel bits hebben. Dat kan d.m.v. tekenuitbreiding.

De eerste optelling is 011001 + 100100. Het eerste getal is positief, het tweede getal is negatief. De getallen kunnen direct opgeteld worden. De tweede optelling is 110011 + 100011. Beide getallen zijn negatief en kunnen direct worden opteld.

000000		000110		carry's
011001		110011		getal A
100100	+	100011	+	getal B
0)111101		1)010110		resultaat

Figuur A.41: *Enkele optellingen.*

De linker optelling levert een negatief getal op. Er is geen overflow want de op te tellen getallen hebben verschillend teken. De uitgaande carry is 0. De tweede optelling levert een overflow op. De op te tellen getallen zijn beide negatief en het antwoord is positief. De uitgaande carry is 1 en moet genegeerd worden. Aan de uitgaande carry is niet te zien of er overflow is opgetreden.

Hierna volgen twee aftrekkingen namelijk 011001 − 001110 en 101010 − 010101. De aftrekkingen kunnen worden omgezet in optellingen door het getal dat moet worden afgetrokken om te zetten in de negatieve representatie van het originele getal. Omzetten gaat heel eenvoudig met de procedure *flip bits, add 1* zoals uitgelegd is in opgave 5.11. Het getal 001110 wordt dan 110010 en het getal 010101 wordt 101011.

100000		010100		carry's
011001		101010		getal A
110010	+	101011	+	getal B
1)001011		1)010101		resultaat

Figuur A.42: *Enkele optellingen.*

De uitgaande carry's moeten weer genegeerd worden. De linker optelling levert geen overflow op, de rechter optelling levert wel een overflow. Het betreft hier nu twee negatieve getallen die opgeteld een positief getal opleveren.

De volgende optelling is 1101 + 111001. Hierin is één van de getallen maar in vier bits geschreven en moet worden uitgebreid naar zes bits. Het getal begint met een 1 (negatief) en er moeten dan twee 1-en voor gezet worden. De optelling wordt dan 111101+111001.

Daarna volgt een aftrekking met de getallen $101011 - 1100111$. Het eerste getal moet uitgebreid worden met een 1, het is immers negatief. Het betreft hier een aftrekking dus het tweede getal moet omgezet worden in de negatieve representatie van het originele getal en levert dan 0011001. Nu kunnen de twee optellingen uitgevoerd worden.

110010		1011110		carry's
111101		1101011		getal A
111001	+	1100111	+	getal B
1)110110		1)1010010		resultaat

Figuur A.43: *Enkele optellingen.*

De laatste twee berekeningen zijn een aftrekking en een optelling. De aftrekking betreft $011100 - 011001$ en de optelling betreft $10001 + 1100111$. Er moeten weer omzettingen in 2's complement en tekenuitbreiding gedaan worden. Uiteindelijk levert dat de onderstaande optellingen op:

111000		1001110		carry's
011100		1111001		getal A
100111	+	1100111	+	getal B
1)000011		1)1011000		resultaat

Figuur A.44: *Enkele optellingen.*

Beide uitkomsten leveren geen overflow op. De uitgaande carry's moeten genegeerd worden.

Uitwerking opgave 5.13.

Een hexadecimaal 2's complement getal dat begint met de cijfers 8 t/m F wordt als negatief beschouwd omdat het binaire equivalent van dat cijfer begint met een 1. Zo is 8_{16} gelijk aan 1000_2 en F_{16} gelijk aan 1111_2. Getallen die beginnen met de cijfers 0 t/m 7 zijn positief omdat het binaire equivalent van deze cijfers beginnen met een 0.

Als we het getal $FFFF_{16}$ opschrijven als binair getal dan levert dat $1111.1111.1111.1111_2$ (punten worden gebruikt ter bevordering van het lezen). Zouden we dit omzetten naar het decimale equivalent dan levert dat behoorlijk wat schrijfwerk op:

$$\begin{aligned} FFFF_{16} &= -2^{15} + 2^{14} + 2^{13} + \ldots + 2^2 + 2^1 + 2^0 \\ &= -32768 + 16384 + 8192 + \ldots + 4 + 2 + 1 \\ &= -1 \end{aligned}$$

Handiger is om direct gebruik te maken van de hexadecimale cijfers waarbij het meest significante cijfer als negatief wordt beschouwd is dit cijfer tussen 8 en F ligt:

$8_{16} = -8_{10}$ $9_{16} = -7_{10}$ $A_{16} = -6_{10}$ $B_{16} = -5_{10}$ $C_{16} = -4_{10}$ $D_{16} = -3_{10}$ $E_{16} = -2_{10}$ $F_{16} = -1_{10}$

We kunnen nu FFFF_{16} direct uitschrijven in machten van 16:

$$\begin{aligned}\text{FFFF}_{16} &= -1\cdot 16^3 + 15\cdot 16^2 + 15\cdot 16^1 + 15\cdot 16^0 \\ &= -4096 + 3840 + 240 + 15 \\ &= -1\end{aligned}$$

En de rest van de getallen:

$$\begin{aligned}\text{53BA}_{16} &= 5\cdot 16^3 + 3\cdot 16^2 + 11\cdot 16^1 + 10\cdot 16^0 \\ &= 20480 + 768 + 176 + 10 \\ &= 21434\end{aligned}$$

$$\begin{aligned}\text{CB}_{16} &= -4\cdot 16^1 + 11\cdot 16^0 \\ &= -64 + 11 \\ &= -53\end{aligned}$$

$$\begin{aligned}\text{7EA3}_{16} &= 7\cdot 16^3 + 14\cdot 16^2 + 10\cdot 16^1 + 3\cdot 16^0 \\ &= 28672 + 3584 + 160 + 3 \\ &= 32419\end{aligned}$$

Uitwerking opgave 5.14.
Een hexadecimaal getal van acht cijfers kan worden geschreven als een binair getal van 32 cijfers. De bereik van zo'n getal is

$$-2^{31} \leq M \leq +2^{31} - 1 \tag{A.54}$$

oftewel

$$-2147483648 \leq M \leq +2147483647 \tag{A.55}$$

Uitwerking opgave 5.15.
Tekenuitbreiding bij hexadecimale getallen is niet zo heel ingewikkeld. Als het meest significante cijfer tussen 0 en 7 ligt, moet het getal uitgebreid worden met voorlopende nullen (een 0 levert vier nullen op: $0 \rightarrow 0000$). Ligt het meest significante cijfer tussen 8 en F, dan moet het getal worden uitgebreid met voorlopende F-en (een F levert vier enen op: $\text{F} \rightarrow 1111$). Dus:

$\text{F}_{16} \rightarrow \text{FFFF}_{16}$ $\quad 6_{16} \rightarrow 0006_{16}$ $\quad \text{7A}_{16} \rightarrow \text{007A}_{16}$ $\quad \text{CB}_{16} \rightarrow \text{FFCB}_{16}$ $\quad \text{35B}_{16} \rightarrow \text{035B}_{16}$
$\text{D73}_{16} \rightarrow \text{FD73}_{16}$

Uitwerking opgave 5.16.

We noemen de ingangen $X = x_3x_2x_1x_0$ en de uitgangen $F = f_3f_2f_1f_0$. De werking van de 9's-complement-functie kan worden beschreven met de functie

$$F = 9 - X \qquad \text{voor } 0 \leq X \leq 9 \tag{A.56}$$

Eerst moet de waarheidstabel worden opgezet. De bitcombinaties $X = x_3x_2x_1x_0$ kunnen gezien worden als een getal dat ligt tussen 0 en 15, waarvan alleen de combinaties 0 t/m 9 gebruikt worden, de overige komen niet voor. Dat worden don't cares in de waarheidstabel. De bitcombinaties $F = f_3f_2f_1f_0$ vormen de getallen $9 - X$, dus X moet van 9 afgetrokken worden.

Van alle mogelijkheden wordt een waarheidstabel opgezet.

Tabel A.21: *Waarheidstabel van de 9's complement functie.*

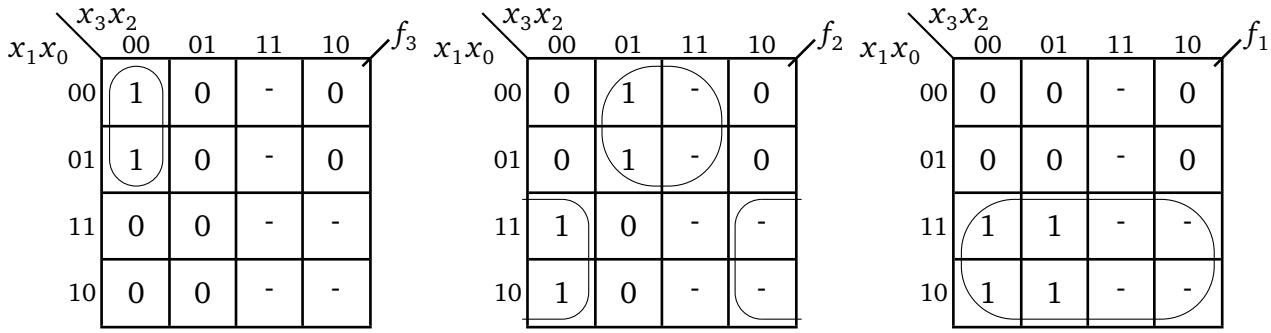
x_3	x_2	x_1	x_0	f_3	f_2	f_1	f_0
0	0	0	0	1	0	0	1
0	0	0	1	1	0	0	0
0	0	1	0	0	1	1	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	0	0
0	1	1	0	0	0	1	1
0	1	1	1	0	0	1	0
1	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	–	–	–	–
1	0	1	1	–	–	–	–
1	1	0	0	–	–	–	–
1	1	0	1	–	–	–	–
1	1	1	0	–	–	–	–
1	1	1	1	–	–	–	–

Vanuit de tabel is direct te zien dat $f_0 = \overline{x_0}$. Voor de overige functies worden Karnaugh-diagrammen opgezet en uitgewerkt. Deze zijn weergegeven in onderstaande figuur.

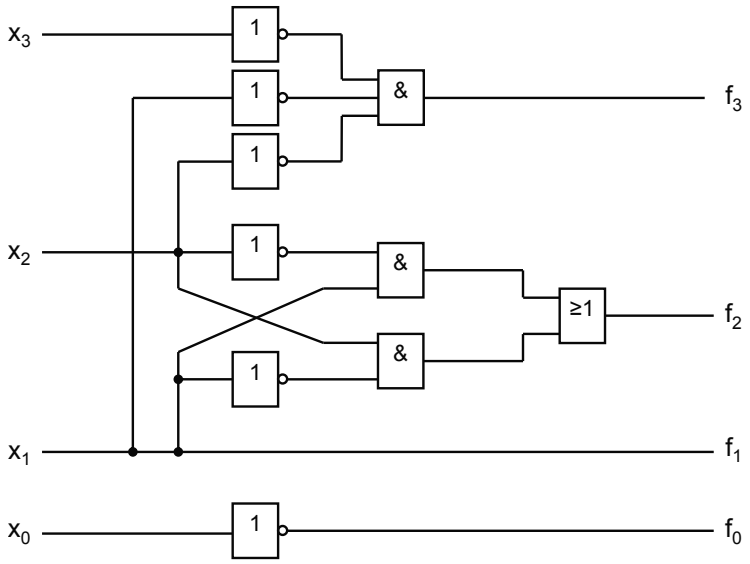
De gevonden functies voor f_3 t/m f_0 zijn:

$$\begin{aligned} f_3 &= \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \\ f_2 &= \overline{x_2} \cdot x_1 + x_2 \cdot \overline{x_1} \\ f_1 &= x_1 \\ f_0 &= \overline{x_0} \end{aligned} \tag{A.57}$$

De schema's van de vier functies zijn weergegeven in onderstaande figuur.



Figuur A.45: *Karnaughdiagrammen voor de functies f_3, f_2 en f_0.*



Figuur A.46: *Schema 9's complement functie.*

Uitwerking opgave 5.17.
Het is een kwestie van interpretatie.

In de 2's representatie systeem wordt een binair getal dat begint met een 1 als negatief beschouwd. Het meest negatieve getal is een 1 gevolgd door alleen 0-en. Maar er is niets op tegen om dit ene getal als positief te beschouwen.

Laten we eens kijken wat er met optellingen gebeurt als we 1000_2 als +8 beschouwen. We tellen bij alle getallen van -7 tot +8(!) de waarde +8 op en interpreteren het resultaat als 2's complement waarbij 1000_2 als +8 wordt beschouwd. Uiteraard gooien we een eventuele carry weg.

De optellingen onder -7 tot 0 leveren correcte antwoorden op, die van +1 tot +8 geven een overflow.

De reden waarom 1000_2 als -8 wordt beschouwd is heel simpel: het levert eenvoudigere hardware op. Begint het getal met een 1, dan is het negatief, anders is het positief. Het tekenbit kan dan ook als onderdeel van het getal worden gezien en heeft dus ook een (negatief) gewicht.

$$
\begin{array}{rcrcr}
-7+8 & \rightarrow & 1001+1000=1)0001 & \rightarrow & +1 \\
-6+8 & \rightarrow & 1010+1000=1)0010 & \rightarrow & +2 \\
-5+8 & \rightarrow & 1011+1000=1)0011 & \rightarrow & +3 \\
-4+8 & \rightarrow & 1100+1000=1)0100 & \rightarrow & +4 \\
-3+8 & \rightarrow & 1101+1000=1)0101 & \rightarrow & +5 \\
-2+8 & \rightarrow & 1110+1000=1)0110 & \rightarrow & +6 \\
-1+8 & \rightarrow & 1111+1000=1)0111 & \rightarrow & +7 \\
0+8 & \rightarrow & 0000+1000=0)1000 & \rightarrow & +8 \\
+1+8 & \rightarrow & 0001+1000=0)1001 & \rightarrow & -7 \\
+2+8 & \rightarrow & 0010+1000=0)1010 & \rightarrow & -6 \\
+3+8 & \rightarrow & 0011+1000=0)1011 & \rightarrow & -5 \\
+4+8 & \rightarrow & 0100+1000=0)1100 & \rightarrow & -4 \\
+5+8 & \rightarrow & 0101+1000=0)1101 & \rightarrow & -3 \\
+6+8 & \rightarrow & 0110+1000=0)1110 & \rightarrow & -2 \\
+7+8 & \rightarrow & 0111+1000=0)1111 & \rightarrow & -1 \\
+8+8 & \rightarrow & 1000+1000=1)0000 & \rightarrow & 0
\end{array}
$$

Figuur A.47: *Gebruik van bitpatroon 1000 als* +8.

Uitwerking opgave 5.18.

Bij de eerste twee optellingen treedt overflow op; het antwoord past niet in het aantal bits waaruit de opgetelde getallen bestaan. De laatste optelling genereert geen overflow.

0001 0000		0000		0001 0000 0000		carry's
1001 1001		0011		0101 1000 0110		getal A
0011 1001	+	0111	+	0000 0111 0001	+	getal B
1) 0010 1000		1) 0000		0110 0101 0111		resultaat

Figuur A.48: *Enkele optellingen met BCD-gecodeerde getallen.*

Uitwerking opgave 5.19.

We noemen de ingangen $X = x_3x_2x_1x_0$ en de uitgangen $F = f_3f_2f_1f_0$. De werking van de excess-3-functie kan worden beschreven met de functie

$$F = X + 3 \qquad \text{voor } 0 \leq X \leq 9 \tag{A.58}$$

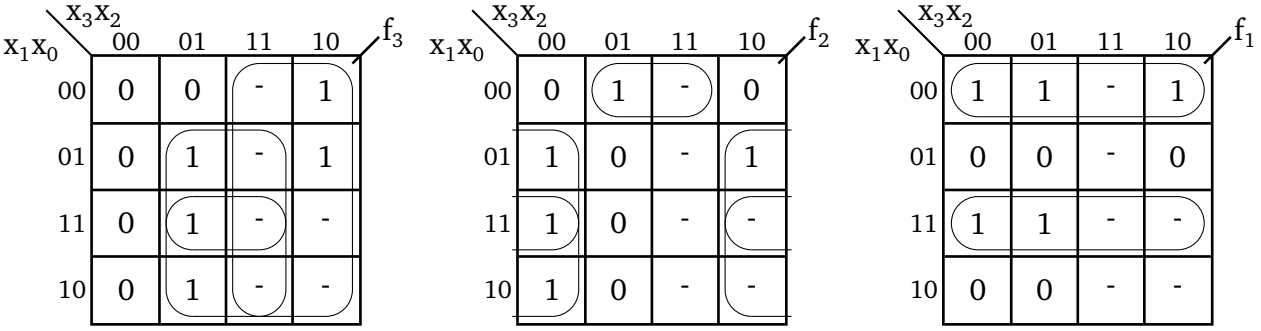
Eerst moet de waarheidstabel worden opgezet. De bitcombinaties $X = x_3x_2x_1x_0$ kunnen gezien worden als een getal dat ligt tussen 0 en 15, waarvan alleen de combinaties 0 t/m 9 gebruikt worden, de overige komen niet voor. Dat worden don't cares in de waarheidstabel. De bitcombinaties $F = f_3f_2f_1f_0$ vormen de getallen $X+3$, dus bij X moet 3 opgeteld worden.

Van alle mogelijkheden wordt een waarheidstabel opgezet.

Vanuit de tabel is direct te zien dat $f_0 = \overline{x_0}$. Voor de overige functies worden Karnaugh-diagrammen opgezet en uitgewerkt. Deze zijn weergegeven in onderstaande figuur.

Tabel A.22: *Waarheidstabel voor omzetter van BCD-code naar Excess-3-code.*

x_3	x_2	x_1	x_0	f_3	f_2	f_1	f_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	—	—	—	—
1	0	1	1	—	—	—	—
1	1	0	0	—	—	—	—
1	1	0	1	—	—	—	—
1	1	1	0	—	—	—	—
1	1	1	1	—	—	—	—



Figuur A.49: *Karnaughdiagrammen BCD-naar-Excess-3-code.*

De gevonden functies voor f_3 t/m f_0 zijn:

$$\begin{aligned} f_3 &= x_3 + x_2 \cdot x_1 + x_2 \cdot x_0 \\ f_2 &= \overline{x_2} \cdot x_1 + \overline{x_2} \cdot x_0 + x_2 \cdot \overline{x_1} \cdot \overline{x_0} \\ f_1 &= \overline{x_1} \cdot \overline{x_0} + x_1 \cdot x_0 \\ f_0 &= \overline{x_0} \end{aligned} \tag{A.59}$$

De functie voor f_3 kan nog geschreven worden als

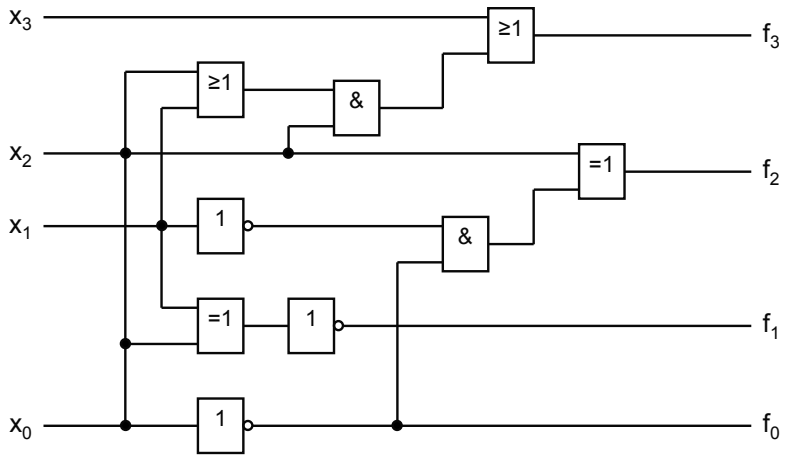
$$f_3 = x_3 + x_2 \cdot (x_1 + x_0) \tag{A.60}$$

waardoor een AND-poort komt te vervallen.

De functies voor f_2 en f_1 kunnen nog omgewerkt worden naar

$$\begin{aligned} f_2 &= x_2 \oplus \left(\overline{x_1} \cdot \overline{x_0}\right) \\ f_1 &= \overline{x_1 \oplus x_0} \end{aligned} \tag{A.61}$$

zodat de inverter voor x_2 komt te vervallen. De schema's van de vier functies zijn weergegeven in onderstaadne figuur. Noot: er zijn ook andere schema's mogelijk die de juiste functies weergeven.



Figuur A.50: *Schema omzetter BCD-naar-Excess-3-code.*

Uitwerking opgave 5.20.

De waarde van de kilometerteller staat in het begin op 00000. Daarna wordt 123 kilometer achteruit gereden en de kilometerteller telt terug (moderne kilometertellers doen dat niet, die tellen altijd vooruit). Om de stand van de kilometerteller te bepalen voeren we een ten's complement omzetting uit. We trekken 123 van 0 af in het ten's complement.

$$\begin{array}{ll} \ldots000) & 00000 \\ \ldots000) & 00123 \quad - \\ \hline \ldots999) & 99877 \end{array}$$

Figuur A.51: *De ten's complement aftrekking van 0 en 123.*

We zien dat de uitkomst 99877 is maar we merken wel op dat er een uitgaande carry is en wel in de vorm van een 9, we rekenen immers in het ten's complement, maar die carry wordt niet meegenomen. We hebben immers maar 5 cijfers om de uitkomst te representeren. De stand op de kilometerteller is dus 99877.

Uitwerking opgave 5.21.

Het aantal bit van de mantisse is 23. Inclusief de leidende 1 (die niet wordt opgeslagen) is de nauwkeurigheid dan 2^{-24}. Het aantal significante decimale cijfers laat zich als volgt berekenen:

$$2^{-24} = 10^x \qquad \longrightarrow \qquad x = -24 \cdot \log 2 = -7{,}2\cdots \tag{A.62}$$

Het aantal significante cijfers is ongeveer 7.

Uitwerking opgave 5.22.

Deze opgave behandelt het optellen en aftrekken van 8-bits getallen. Optellen kan direct worden uitgevoerd met de bekende optelprocedure. Bij aftrekken moet de aftrekker negatief gemaakt worden middels de procedure *flip bits, add 1*. Daarna kunnen de twee getallen gewoon worden opgeteld. Let erop dat bij het bepalen van de flags de carry flag uit de optelling komt. De carry flag wordt dus niet geïnverteerd bij een aftrekking. Dit wordt *subtract with carry* genoemd.

Opgave a) De getallen zijn direct op te tellen met de bekende techniek.

```
      10011001
      11100100  +
-------------------
 1)   01111101
```

Het 8-bits antwoord is ongelijk aan 0, dus $Z = 0$. Er is een uitgaande carry, dus $C = 1$. Het antwoord heeft als tekenbit een 0 dus $N = 0$. Beide getallen zijn negatief (beginnen met een 1), maar het antwoord is positief dus $V = 1$.

Opgave b) Het resultaat kan niet direct berekend worden want het gaat hier om een aftrekking. We moeten dus het tweede getal (de aftrekker genoemd) negatief maken, oftewel de two's complement bepalen. Dat kan met de bekende *flip bits, add 1*-methode. Dus 00001110 wordt 11110010. Dit is te zien in de onderstaande omzetting.

```
   00001110
       ↓
   11110001
          1  +
-------------------
   11110010
```

Nu kunnen we de getallen optellen met de bekende procedure.

```
      10011001
      11110010  +
-------------------
 1)   10001011
```

Hieruit volgt dat $Z = 0$, $C = 1$, $N = 1$ en $V = 0$.

Opgave c) Deze getallen kunnen direct worden opgeteld.

```
      11111101
      11111001  +
-------------------
 1)   11110110
```

Hieruit volgt dat $Z = 0$, $C = 1$, $N = 1$ en $V = 0$.

Opgave d) Dit is een aftrekking dus van de aftrekker moet negatief gemaakt. We laten dat niet expliciet zien maar de procedure *flip bits, add 1* wordt weer gevolgd. Dus 00011001 wordt omgezet in 11100111. We kunnen nu de getallen optellen en de flags bepalen.

```
      00011100
      11100111  +
-----------------
 1)   00000011
```

Hieruit volgt dat $Z = 0$, $C = 1$, $N = 0$ en $V = 0$.

Opgave e) Dit is een optelling en kan direct uitgevoerd worden.

```
      01110011
      11100011  +
-----------------
 1)   01010110
```

Hieruit volgt dat $Z = 0$, $C = 1$, $N = 0$ en $V = 0$.

Opgave f) Dit betreft een aftrekking dus de aftrekker moet negatief gemaakt worden. De aftrekker wordt dan van 00010101 omgezet in 11101011. Daarna kunnen we de getallen optellen.

```
      01101010
      11101011  +
-----------------
 1)   01010101
```

Hieruit volgt dat $Z = 0$, $C = 1$, $N = 0$ en $V = 0$.

Opgave g) Het betref hier een aftrekking dus de aftrekker 11100111 wordt omgezet in 00011001. Nu kunnen we de getallen optellen.

```
      11101011
      00011001  +
-----------------
 1)   00000100
```

Hieruit volgt dat $Z = 0$, $C = 1$, $N = 0$ en $V = 0$.

Opgave h) Deze twee getallen kunnen direct worden opgeteld.

```
      11110001
      11100111  +
-----------------
 1)   11011000
```

Hieruit volgt dat $Z = 0$, $C = 1$, $N = 1$ en $V = 0$.

Uitwerking opgave 5.23.
Overflow in de two's complement representatie kan eenvoudig worden gedetecteerd aan de hand van de te bewerken getallen, het resultaat en de bewerking.

Bij een optelling van twee two's complement getallen kan een overflow alleen maar optreden als de twee getallen hetzelfde teken hebben, met de nadruk op *kan*. Dat is te zien in de onderstaande vergelijkingen. Hierbij worden de grenzen van de two's complement representatie opgezocht. Deze grenzen zijn -2^{n-1} en -1 voor negatieve getallen en 0 en $2^{n-1}-1$ voor positieve getallen.

$$-2^{n-1}+-2^{n-1}=-2^n \tag{A.63a}$$
$$(-1)+(-1)=-2 \tag{A.63b}$$
$$0+0=0 \tag{A.63c}$$
$$(2^{n-1}-1)+(2^{n-1}-1)=2^n-2 \tag{A.63d}$$

De vergelijkingen A.63a en A.63d leveren een overflow op, de vergelijkingen A.63b en A.63c doen dat niet.

Als we twee getallen met verschillend teken optellen kan er nooit overflow optreden. Ook dat is eenvoudig aan te tonen. We zoeken hiervoor weer de grenzen van het bereik van de two's complement representatie op.

$$-2^{n-1}+0=-2^{n-1} \tag{A.64a}$$
$$-1+0=-1 \tag{A.64b}$$
$$-2^{n-1}+(2^{n-1}-1)=-1 \tag{A.64c}$$
$$-1+(2^{n-1}-1)=2^{n-1}-2 \tag{A.64d}$$

De antwoorden van vergelijkingen (A.64a) t/m (A.64d) liggen allemaal in het bereik van de two's complement representatie.

A.6 Uitwerkingen hoofdstuk 6

Uitwerking opgave 6.1.
De twee don't cares kunnen als volgt worden ingevuld:

Hierin is Z_{nieuw} de originele specificatie en zijn O.S. en O.R. de realisatie van resp. overheersende set en overheersende reset. Bij de realisatie van #3 valt op dat de uitgangswaarden bij $SR=11$ precies hetzelfde zijn als de waarde van Z_{oud}, dus de latch onthoudt de waarde (= stand) van Z_{oud}. Dit is dus een latch met een extra onthoudstand ($SR=00$ is er ook één). Bij de realisatie van #4 valt op dat de uitgangswaarden bij $SR=11$ precies de inverse zijn van de waarde van Z_{oud}, dus de latch inverteert de stand. Omdat dit direct werkt (er is geen klok- of enablesignaal om de latch te synchroniseren) zal zodra een stand aan de uitgang bekend is, de inverse worden berekend en na enige poortvertragingen deze nieuwe stand op de uitgang beschikbaar zijn. Deze schakeling oscilleert bij $SR=11$. #3 is een bruikbare latch maar wordt in de praktijk niet gebruikt

Tabel A.23: *Waarheidstabel voor de twee don't cares bij de SR-latch.*

S	R	Z_{oud}	Z_{nieuw}	O.S.	O.R.	#3	#4
0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1
0	1	0	0	0	0	0	0
0	1	1	0	0	0	0	0
1	0	0	1	1	1	1	1
1	0	1	1	1	1	1	1
1	1	0	-	1	0	0	1
1	1	1	-	1	0	1	0

omdat het ontwerp meer poorten kost dan O.S. en O.R. #4 is niet bruikbaar vanwege de oscillatie.

Uitwerking opgave 6.2.

De functie van Z is: $Z_{nieuw} = \overline{A + \overline{B \cdot Z_{oud}}} = \overline{A} \cdot B \cdot Z_{oud}$. Het laatste deel is gevonden met behulp van De Morgan en geeft duidelijk aan dat de latch alleen maar te resetten is ($A = 1$ of $B = 0$) of kan onthouden ($AB = 01$). Setten is niet mogelijk. Dit is dus geen goedwerkend geheugenelement.

Uitwerking opgave 6.3.

De functie van Z is: $Z_{nieuw} = A \oplus B \oplus Z_{oud}$. We kunnen dit omwerken naar: $Z_{nieuw} = \overline{A \oplus B} \cdot Z_{oud} + (A \oplus B) \cdot \overline{Z_{oud}}$. Hieruit wordt duidelijk dat de latch alleen kan onthouden of de inverse kan inlezen. Er is geen mogelijkheid om de latch te setten of te resetten.

Uitwerking opgave 6.4.

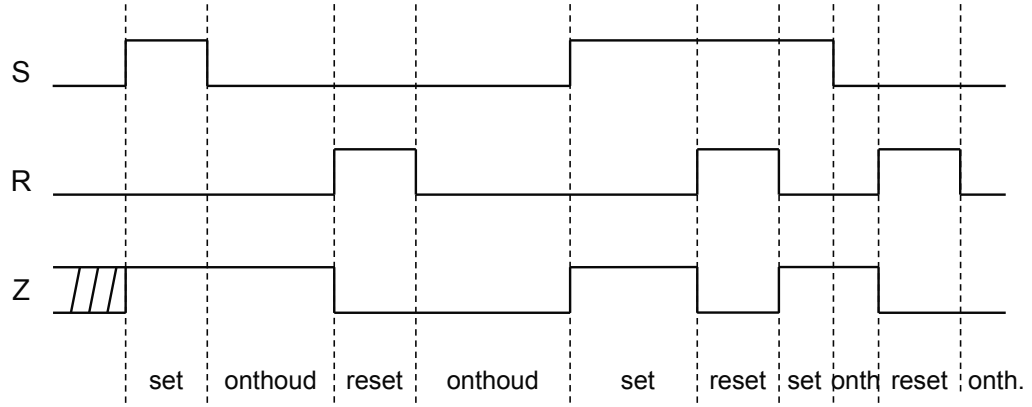
Dit is geen goede codering. De onthoudstand is de "ruststand", dus vanuit set of reset moet je terug kunnen gaan naar onthouden. Dat mag maar met één bitwisseling in de stuurcodes. Het is namelijk praktisch niet mogelijk om twee bits tegelijk te wisselen; er gaat er altijd wel één eerder om. Vanuit set moet je twee bits wisselen om bij onthouden te komen. Dat gaat dus niet goed.

Uitwerking opgave 6.5.

Zie onderstaand timingdiagram. Dit diagram wijkt op één punt af van het diagram voor de SR-latch met overheersende set: als S en R beide logisch 1 zijn, is de uitgang Z logisch 0.

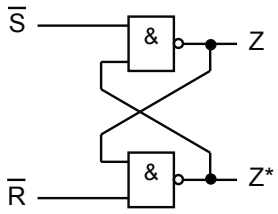
Uitwerking opgave 6.6.

Als we kijken naar de diverse latches die de revue gepasseerd zijn, dan kunnen we constateren dat alleen latches die gebouwd zijn met inverterende poorten (NAND-NAND en NOR-NOR) in aanmerking komen. Zie figuur. In een tabel kunnen we zetten hoe Z en Z^* zich verhouden. Z en Z^* zijn inderdaad elkaars inverse voor de drie combinaties



Figuur A.52: *Signaaldiagram voor de SR-latch met overheersende reset.*

de nuttig te gebruiken zijn. Bij de combinatie $SR = 11$ zijn Z en Z^* beide logisch 1.

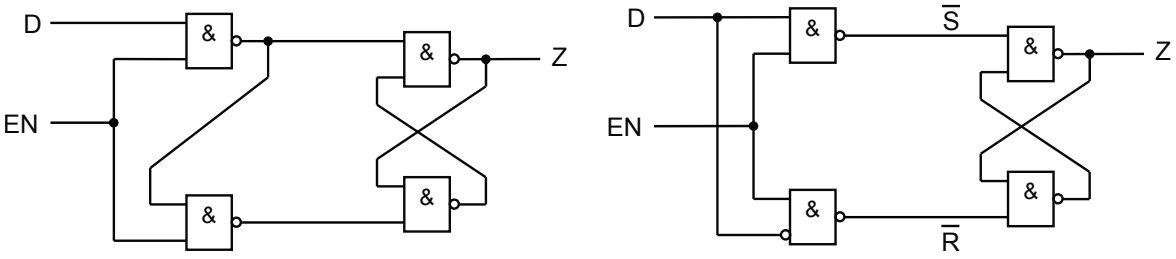


S	R	Z_{oud}	Z_{nieuw}	Z^*_{nieuw}
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

Figuur A.53: *SR-latch met NAND-poorten en bijbehorende waarheidstabel.*

Uitwerking opgave 6.7.

We tekenen twee gated D-latches naast elkaar. Links is de nieuwe te onderzoeken gated D-latch, rechts is de bekende gated D-latch die volgt uit de aangepaste gated SR-latch.



Figuur A.54: *Twee realisaties van een gated D-latch.*

Te zien is dat beide latches bestaan uit een op NAND's gebaseerde SR-latch met voorzet-logica. De SR-latch wordt aangestuurd met de inverse van S(et) en R(eset). De signalen zijn ingetekend in de rechter latch.

De functies voor de set er reset zijn (let hierbij op de naamgeving voor $\overline{S}$ en $\overline{R}$, dit zijn

signaalnamen):

$$\begin{aligned}[\overline{S}] &= \overline{\overline{D \cdot EN}} \\ [\overline{R}] &= \overline{\overline{D} \cdot EN}\end{aligned} \tag{A.65}$$

Dit moet dus ook gelden voor de te onderzoeken latch. Het is duidelijk dat de functie voor de set voor beide schakelingen identiek is. De functie voor de reset voor de te onderzoeken latch is

$$\begin{aligned}[\overline{R}] &= \overline{\overline{[\overline{S}] \cdot EN} \\ &= \overline{\overline{\overline{D \cdot EN} \cdot EN} \\ &= \overline{(\overline{D} + \overline{\overline{EN}}) \cdot EN} \\ &= \overline{\overline{D} \cdot EN + \overline{\overline{EN}} \cdot EN} \\ &= \overline{\overline{D} \cdot EN + 0} \\ &= \overline{\overline{D} \cdot EN}\end{aligned} \tag{A.66}$$

Merk op dat hier een keer De Morgan is gebruikt en diverse andere wetten en theorema's. Het uiteindelijke resultaat voor de reset is hetzelfde voor beide latches dus de *logische* werking is identiek. De te onderzoeken latch is met vier NANDs te bouwen en heeft geen inverter nodig bij de resetingang.

Uitwerking opgave 6.8.

De tweedeler is niet te maken met een (D-)latch. De uitgangswaarde van de latch moet geïnverteerd worden doorgegeven aan de ingang. Zodra de latch transparant is, verandert de uitgangswaarde als gevolg van de nieuwe ingangswaarde. Dit de inverse van de uitgangswaarde. Het geheel wordt instabiel en gaat *oscilleren*. Dit oscilleren volgt uit het transparant zijn van de latch. Door gebruik te maken van twee latches kan dit transparant zijn worden onderbroken door één van de latches gesloten te houden terwijl de andere geopend is. De enable-signalen zijn dus elkaars inverse. Zie figuur.

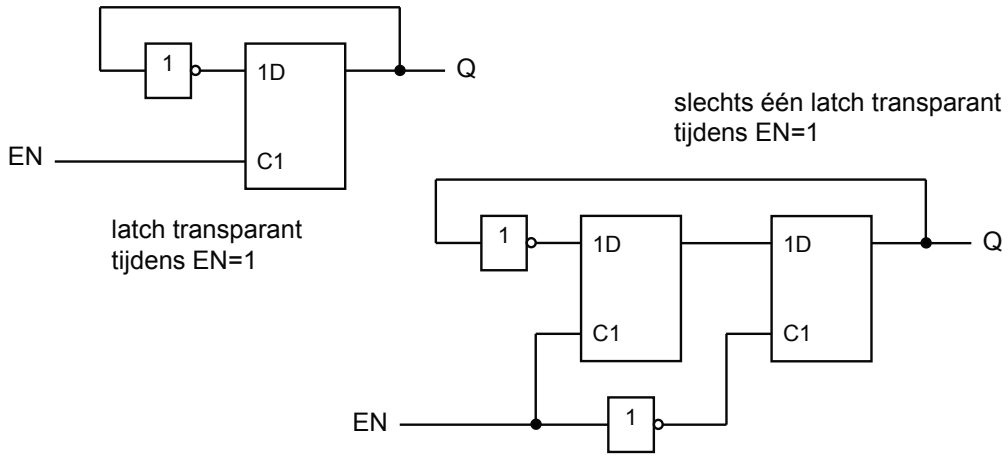
Uitwerking opgave 6.9.

De timing-voorwaarde voor de correcte werking van de latch is

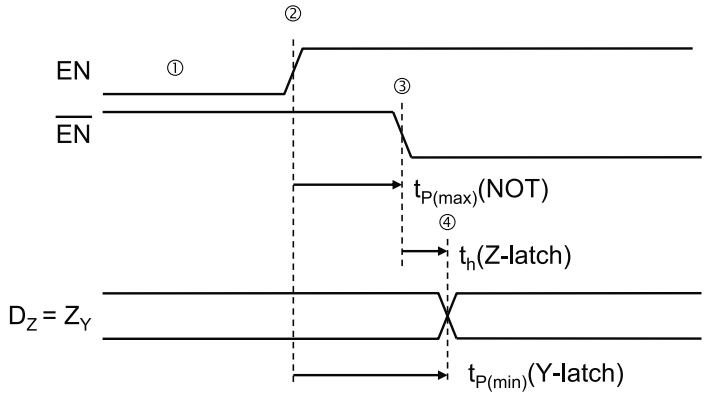
$$t_{P(max)}(\text{NOT}) + t_h(\text{Z-latch}) < t_{P(min)}(\text{Y-latch}) \tag{A.67}$$

Dit is een doordenker. Eerst maar eens het timingdiagram tekenen. Zie onderstaande figuur.

In de bovenstaande figuur zijn het enable-signaal en de geïnverteerde versie te zien. De geïnverteerde versie wordt gegenereerd door een NOT-poort (inverter) en dat kost enige tijd. Het enable-signaal wordt aan de Y-latch aangeboden. De Y-latch heeft een uitgang genaamd Z_Y. Het geïnverteerde enable-signaal wordt aan de Z-latch aangeboden. De Z-latch heeft een ingang D_Z. De uitgang van de Y-latch is direct gekoppeld aan de ingang



Figuur A.55: *Poging tot realisatie van een tweedeler met gated D-latches.*

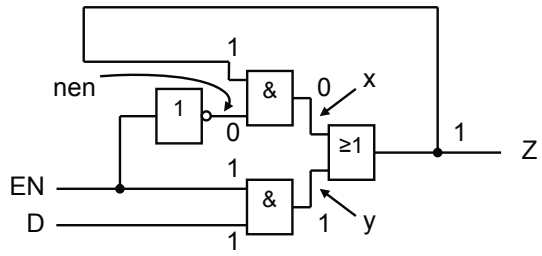


Figuur A.56: *Timing-voorwaarde van een D-flipflop.*

van de Z-latch, dus $D_Z = Z_Y$. De situatie op moment ① is dat de Y-latch gesloten is en de Z-latch open. De Y-latch neemt nog geen data over. Op moment ② gaat de Y-latch open en zal nieuwe data gaan doorgeven, maar door de vertraging van de NOT-poort is de Z-latch óók nog open. Op moment ③ is het geïnverteerde enable-signaal laag (0) waardoor de Z-latch gaat sluiten (Y-latch staat nog steeds open). Dat is dus zeker na de maximale propagatietijd van de NOT-poort. De data moet nog enige tijd stabiel gehouden worden (holdtijd Z-latch) dus pas na moment ④ mag de data op de ingang van de Z-latch veranderen. Gelukkig heeft de Y-latch enige vertraging met het doorgeven van nieuwe data, dus mag de uitgang van de Y-latch pas veranderen (minimale propagatie) nadat holdtijd van de Z-latch is verstreken.

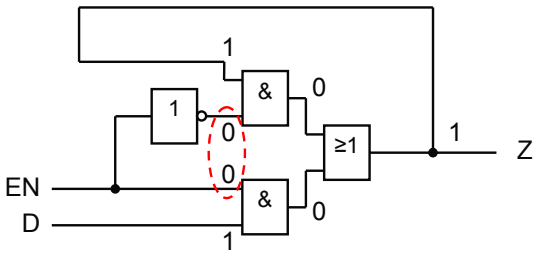
Uitwerking opgave 6.10.

Het probleem zit in de timing. Zie onderstaande figuur. Er zijn twee delen in de D-latch te onderscheiden. De onderste AND-poort (uitgang y) zorgt er voor dat de waarde van D "in de lus" wordt gebracht als $EN = 1$, de bovenste AND-poort (uitgang x) is "gesloten". Als EN nu logisch 0 wordt, zal de onderste AND-poort "sluiten" en zorgt de bovenste AND-poort samen met de lus dat de waarde op Z behouden blijft. De inverse van EN wordt gemaakt met behulp van een inverter (*nen*) en dat kost enige tijd. *Er is dus een moment*

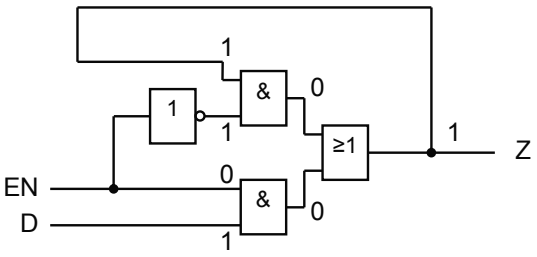


Kort *voordat* EN van 1 naar 0 gaat

dat beide AND-poorten een logisch 0 op één van de ingangen aangeboden krijgen. Zie onderstaande figuur *links*. Merk op dat de signaalverandering door de OR-poort nog

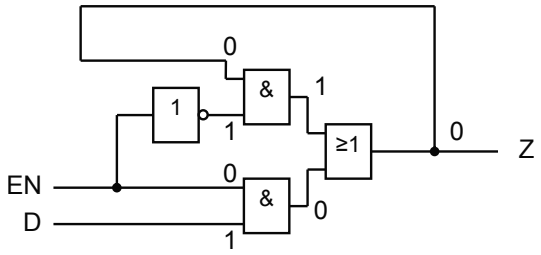


Kort *nadat* EN van 1 naar 0 gaat (signaalverandering door OR-poort nog niet gezien)



De inverter heeft nu de juiste waarde, maar de bovenste AND-poort heeft dat nog niet gezien.

niet is “gezien”, wel door de inverter. Ondertussen is de inverter op de juiste waarde gekomen (logisch 1) nog voordat de OR-poort een logische 0 levert en samen met de logisch 1 van de uitgang levert dit weer een logische 1 op voor de bovenste AND-poort. Zie bovenstaande figuur *rechts*.



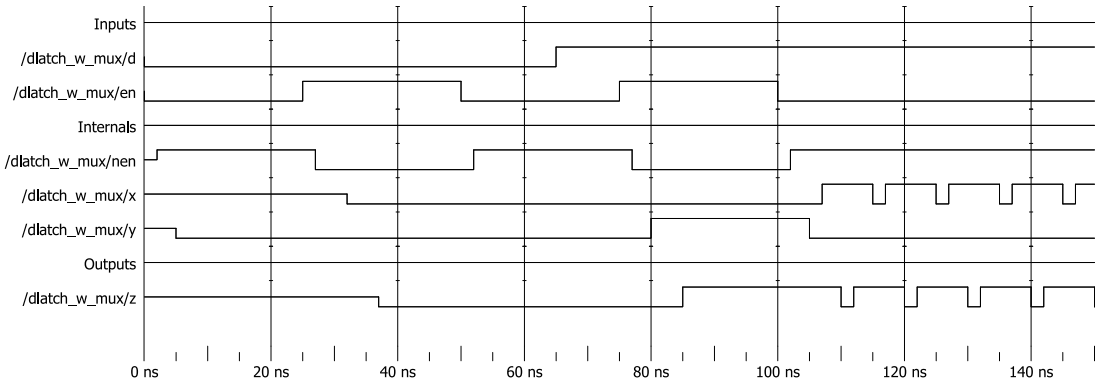
De OR-poort levert een logische 0 maar de bovenste AND-poort geeft ondertussen een logische 1 af, de OR-poort zal na een tijdje dus weer een logische 1 afgeven.

De OR-poort zal uiteindelijk een logische 0 leveren. Deze logische 0 wordt wel weer via de lus “in het systeem” gebracht. Maar de bovenste AND-poort geeft ondertussen een logische 1 af. Zie figuur hiernaast.

Nu zijn er een drie mogelijkheden: de uitgangswaarde blijft oscilleren tussen 0 en 1, de uitgang wordt na een tijd logisch 1 of logisch 0. Dit hangt helemaal af van de eigenschappen van de poorten en verbindingen (wires): propagatietijd is een belangrijke maar ook de *inertial rejection time*: als een aangeboden puls op een ingang te kort is, “ziet” een poort dit niet en de uitgang verandert dan ook niet.

In onderstaande simulatie blijft de uitgang oscilleren. De vertragingstijden van de AND- en OR-poort zijn 5 ns, die van de inverter is 2 ns. Alle poorten hebben een *inertial*

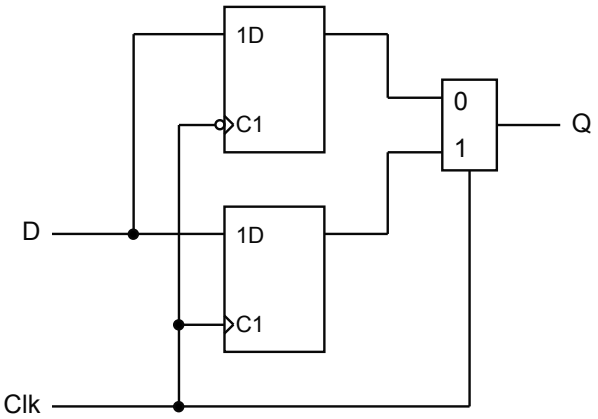
rejection time van 1 ns.



Figuur A.57: *Timing van een gated D-latch op basis van een multiplexer.*

Uitwerking opgave 6.11.

De meest eenvoudige oplossing is om twee flipflops die elk een flank voor hun rekening nemen te combineren. De uitgang is dan de waarde van de flipflop waar het laatst de flank is gepasseerd. Dit kan eenvoudig met een mutiplexer gerealiseerd worden. Zie onderstaande figuur.



Figuur A.58: *Double Edge Triggered flipflop op basis van D-flipflops en een multiplexer.*

Helaas heeft deze schakeling wel wat nadelen. Zo is de uitgang niet *single transition*. Bij het veranderen van het kloksignaal zal steeds de andere flipflop geactiveerd worden, maar ook de multiplexer zal de andere ingang selecteren. De uitgangen van de flipflops geven niet direct de nieuwe waarde af. Als de multiplexer snel is, zal deze eerst nog even de oude waarde van de na de flank geselecteerde flipflop doorgeven. Beter is om een double edge-triggered D-flipflop geheel opnieuw te ontwerpen en dat is lastig. Een leuke realisatie met multiplexers is te vinden op http://atrak.usc.edu/~massoud/Papers/det-ff.pdf.

Uitwerking opgave 6.12.

Het toggelen laat zich op eenvoudige wijze beschrijven. Hiervoor wordt uitgegaan van de D-flipflop. De nieuwe waarde is immers de inverse van de oude waarde: $Q^{n+1} = \overline{Q^n}$.

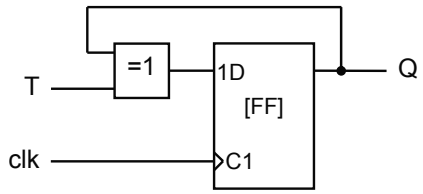
Dit kan prima gedaan worden door een D-flipflop zijn eigen inverse waarde te laten inklokken.

Er wordt nu een signaal T geïntroduceerd waarmee dat toggelen te besturen is. De keuze is arbitrair maar een goede codering is als volgt:

Als T = 0 moet de flipflop zijn waarde vasthouden $\rightarrow Q^{n+1} = Q^n$

Als T = 1 moet de flipflop toggelen (inverse stand) $\rightarrow Q^{n+1} = \overline{Q^n}$

Dit is in z'n geheel te schrijven tot $Q^{n+1} = \overline{T^n} \cdot Q^n + T^n \cdot \overline{Q^n} = T \oplus Q^n$. Dus met behulp van een EXOR-poort is de T-flipflop op basis van een D-flipflop te realiseren. Zie onderstaande figuur.



Figuur A.59: *T-flipflop op basis van een D-flipflop en een EXOR-poort.*

Uitwerking opgave 6.13.

De functie van een JK-flipflop is $Q^{n+1} = J^n \cdot \overline{Q^n} + \overline{K^n} \cdot Q^n$
Toon dit aan met behulp van een Karnaughdiagram. Eerst maar eens de actietabel van de JK-flipflop weergeven:

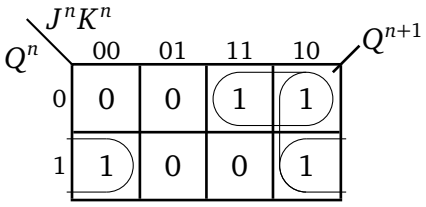
J^n	K^n	Q^{n+1}	actie
0	0	Q^n	onthouden
0	1	0	reset
1	0	1	set
1	1	$\overline{Q^n}$	inverse

Hiernaast de actietabel. Merk op dat bij de laatste mogelijkheid ($JK = 11$) de inverse waarde wordt "ingeklokt". We schrijven de actietabel om naar een waarheidstabel.

J^n	K^n	Q^n	Q^{n+1}	actie
0	0	0	0	onthouden
0	0	1	1	onthouden
0	1	0	0	reset
0	1	1	0	reset
1	0	0	1	set
1	0	1	1	set
1	1	0	1	inverse
1	1	1	0	inverse

Hiernaast is de waarheidstabel gegeven van de JK-flipflop. De eerste zes combinaties zijn identiek aan die van de SR-latch of (de niet besproken) SR-flipflop. De laatste combinatie wordt nu gebruikt om de inverse stand van de flipflop in te klokken. De flipflop "toggelt" dus. Bij een latch geeft dat problemen (oscillatie) maar een flipflop is flankgestuurd.

Vervolgens kan een Karnaughdiagram worden opgesteld en uitwerkt.



De vereenvoudigde functie is

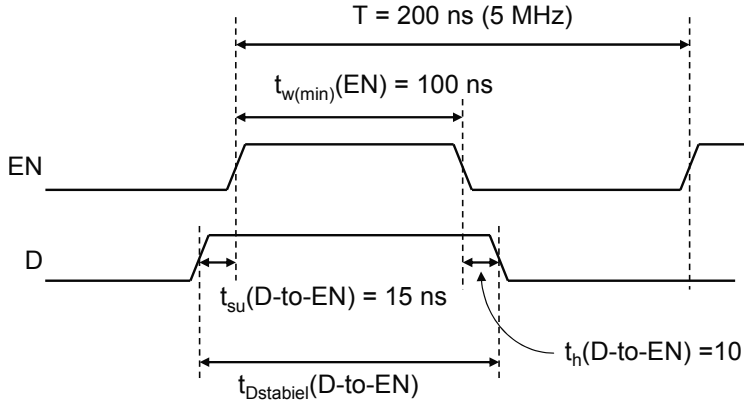
$$Q^{n+1} = J^n \cdot \overline{Q^n} + \overline{K^n} \cdot Q^n$$

De functie is dus zoals in de vraagstelling is gegeven.

Uitwerking opgave 6.14.

Van een latch zijn gegeven t_{su}(EN-to-Z) = 15 ns, t_h(EN-to-Z) = 10 ns. De klok loopt op 5 MHz en heeft een duty cycle van 50%. Wat is de tijd dat D stabiel moet blijven? Hiervoor moet een timingdiagram getekend worden. Daarin moet natuurlijk het kloksignaal getekend worden. Merk op dat het kloksignaal het enablesignaal is (EN = klok). Van het kloksignaal is gegeven een frequentie van 5 MHz, de periodeduur (of periodetijd) is dan 200 ns. Verder is de duty cycle 50% wat inhoudt dat de klok gedurende 100 ns (50% van de periodeduur) logisch '1' is (dat wordt ook wel klok-hoog genoemd). Gedurende die tijd moet D zeker stabiel blijven. Daarnaast zijn nog een setuptijd en een holdtijd gegeven. D moet stabiel blijven van de setuptijd (voordat EN logisch 1 wordt) tot na de holdtijd (nadat EN logisch 0 is geworden). Zie de figuur. De totale tijd dat D stabiel moet zijn is:

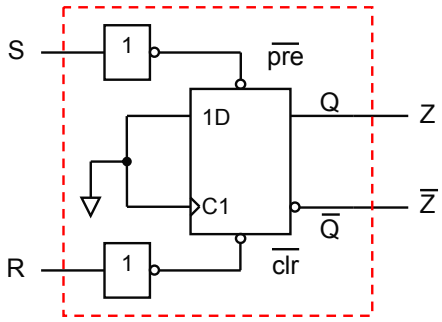
$$\begin{aligned} t_{Dstabiel}(\text{D-to-EN}) &= t_{su}(\text{D-to-EN}) + t_{w(min)}(\text{EN}) + t_h(\text{D-to-EN}) \\ &= 15 + 100 + 10 = 125 \text{ ns} \end{aligned}$$



Figuur A.60: *Timing van de gated D-latch*

Uitwerking opgave 6.15.

Zie onderstaande figuur. De D- en klokingang zijn met een logische 0 verbonden waardoor de flipflop nooit data zal inklokken. De asynchrone preset- en clear-ingangen zijn verbonden (via een inverter) met resp. de S(et)- en R(eset)-signalen. Deze configuratie wordt veel gebruikt in een FPGA. Er wordt een minimum aan logische componenten gebruikt en de timing is veel beter in de hand te houden dan wanneer de schakeling met combinatoriek zou worden gerealiseerd. Overigens is het gebruik van latches niet aan te bevelen in een complex digitaal systeem.

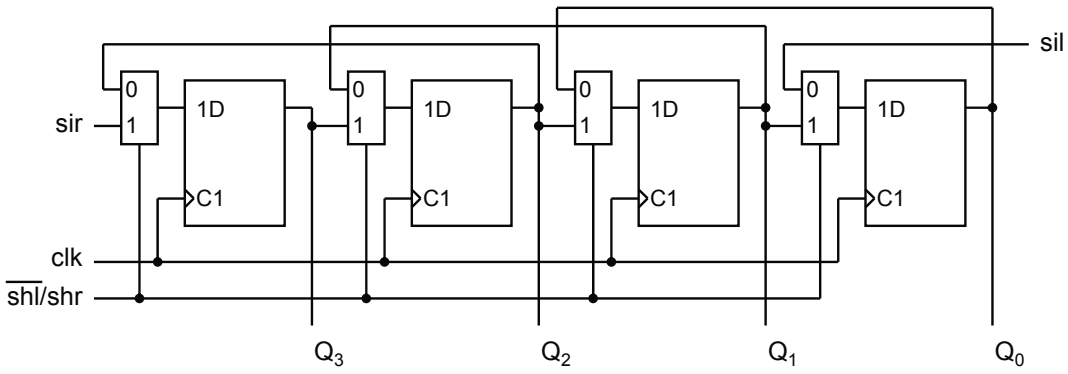


Figuur A.61: *Een SR-lacth op basis van een D-flipflop met asynchrone set en reset.*

Uitwerking opgave 6.16.

Het schuifregister moet twee kanten op kunnen schuiven. Hiervoor is een stuursignaal nodig. We noemen het signaal $\overline{shl}/shr$ (deze naamgeving komt vaker voor, denk hierbij aan het $R/\overline{W}$-signaal bij processoren). Als het signaal logisch 0 is, zal het schuifregister linksom schuiven, als het signaal logisch 1 is, zal het schuifregister rechtsom schuiven.

Tijdens het schuiven wordt en een databit ingeschoven via één van de ingangen sir (serial input right) en sil (serial input left). Natuurlijk verdwijnt er ook een bit, dit bit wordt "eruit geschoven". Verder worden multiplexers gebruikt om de te schuiven bits in de goede flipflops te krijgen.



Figuur A.62: *Een links-rechts schuifregister.*

Uitwerking opgave 6.17.

De flipflops zijn beide in het begin geladen met een 0. We kunnen dus schrijven dat $Q_1^n Q_0^n = 00$. Als er nu een klokpuls wordt aangeboden dan klokt flipflop 1 de waarde van flipflop 0 in. Flipflop 0 klokt de *inverse* waarde van flipflop 1 in. De nieuwe stand na één klokpuls is dus $Q_1^n Q_0^n = 01$. Bij een volgende klokpuls klokt fliflop 1 nu een 1 in en flipflop 0 klokt ook een 1 in. De stand is dan $Q_1^n Q_0^n = 11$. Na de derde klokpuls heeft flipflop 1 een 1 ingeklokt en flipflop 0 een 0. De stand is dan $Q_1^n Q_0^n = 10$. Na de vierde klokpuls heeft flipflop 1 weer een 0 ingeklokt en fliflop 0 weer een 0. De stand is dus $Q_1^n Q_0^n = 00$. Dit was ook de beginstand van de flipflops. De doorlopen cyclus is dus 00-01-11-10.

Uitwerking opgave 6.18.
De flipflop heeft eerst de stand 0. De eerste waarde die aangeboden wordt is ook 0.Daaruit volgt dat D logisch 1 wordt. Zo moeten alle waarden van X worden afgegaan, steeds uitrekenen wat de waarde van Q wordt. Er wordt een tabel opgesteld met X^n, Q^n, D^n en Q^{n+1}. Merk op dat $Q^{n+1} = D^n$ want dat is de werking van een D-flipflop. Zie tabel A.24.

Tabel A.24: *Doorlopen standen van de flipflop.*

nr.	X^n	Q^n	D^n	Q^{n+1}
1	0	0	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1
5	0	1	0	0
6	1	0	0	0
7	0	0	1	1
8	0	1	0	0
9	1	0	0	0
10	1	0	0	0
11	0	0	1	1
12	1	1	1	1
13	–	1	–	–

A.7 Uitwerkingen hoofdstuk 7

Uitwerking opgave 7.1.
Elke regel levert een bijdrage aan de functie s. Eerst wordt de eerste regel uitgewerkt. De uitgang s wordt logisch '1' als $a = 0$. Dit stukje levert de functie $1 \cdot \overline{a}$ op (let op de 1). De tweede regel wordt uitgevoerd als de eerste regel niet waar is, dus als $a = 1$ én $b = 1$. De bijdrage van deze regel is $\overline{\overline{a}} \cdot 1 \cdot b$. De derde regel wordt uitgevoerd als de eerste twee voorwaarden niet waar zijn, dus als $a = 1$ en $b = 0$. De bijdrage is $\overline{\overline{a}} \cdot \overline{b} \cdot 1 \cdot \overline{c}$.

De hele functie is nu op te schrijven een te vereenvoudigen

$$\begin{aligned} s &= 1 \cdot \overline{a} + \overline{\overline{a}} \cdot 1 \cdot b + \overline{\overline{a}} \cdot \overline{b} \cdot 1 \cdot \overline{c} + \overline{\overline{a}} \cdot \overline{b} \cdot \overline{c} \cdot 0 \\ &= \overline{a} + a \cdot b + a \cdot \overline{b} \cdot \overline{c} \\ &= \overline{a} + a \cdot (b + \overline{b} \cdot \overline{c}) \\ &= \overline{a} + a \cdot (b + \overline{c}) \\ &= \overline{a} + (b + \overline{c}) \end{aligned} \tag{A.68}$$

Het is natuurlijk ook mogelijk om een waarheidstabel in te vullen:

Tabel A.25: *Waarheidstabel voor functie s, met uitleg.*

a	b	c	s	
0	0	0	1	eerste regel $a = 0$
0	0	1	1	eerste regel $a = 0$
0	1	0	1	eerste regel $a = 0$
0	1	1	1	eerste regel $a = 0$
1	0	0	1	derde regel $abc = 100$
1	0	1	0	de laatste regel
1	1	0	1	tweede regel $ab = 10$
1	1	1	1	tweede regel $ab = 10$

Uitwerking opgave 7.2.
Het uitwerken van deze functie m.b.v. schakelalgebra is een tijdrovende en lastige zaak. Slimmer is om direct een functietabel op te zetten. Hierin zijn de uitgangswaarden geen constanten maar variabelen. Het opzetten gaat op dezelfde manier als een waarheidstabel. De uitgang s wordt a als $data = 0$. De bijdrage aan de functie is $a \cdot \overline{data}$. De tweede regel levert b op als $data = 1$ en $en = 0$. Nu is het zo dat de tweede regel alleen maar kan worden uitgevoerd (denk aan simulatie) als de eerste regel niet wordt uitgevoerd en dat is als $data = 1$. De voorwaarde $data = 1$ in de tweede regel is dus overbodig! De bijdrage van de tweede regel is $b \cdot data \cdot \overline{en}$. Nog leuker is de derde regel. Deze regel wordt alleen maar uitgevoerd als $en = 0$ maar deze voorwaarde staat ook al in de tweede regel! Dus als de tweede regel wordt uitgevoerd moet $en = 0$ waar zijn (natuurlijk moet $data = 1$ zijn). De derde regel wordt *nooit* uitgevoerd! Al met al de onderstaande tabel:

Tabel A.26: *Waarheidstabel voor functie s, met uitleg.*

$data$	en	s	
0	0	a	eerste regel $data = 0$
0	1	a	eerste regel $data = 0$
1	0	b	tweede regel $data = 1$ en $en = 0$
1	1	d	laatste regel $data = 1$ en $en = 1$

De functie kan heel makkelijk gevonden worden.

$$s = a \cdot \overline{data} + b \cdot data \cdot \overline{en} + d \cdot data \cdot en \tag{A.69}$$

De functie is op een andere manier te schrijven waardoor het gebruik van multiplexers duidelijk te zien is.

$$s = \underbrace{\overline{data} \cdot a + data \cdot (\underbrace{\overline{en} \cdot b + en \cdot d}_{\text{mux}})}_{\text{mux}} \tag{A.70}$$

Uitwerking opgave 7.3.

Hieronder zijn de twee mogelijkheden weergegeven. Links is de versie met CSA en rechts de versie met SSA. Aangezien de EXOR een '1' geeft als de ingangen ongelijk zijn, moeten eerst twee regels worden beschreven waarbij de uitgang '1' wordt. Dit geeft gelijk al aan dat de EXOR niet vereenvoudigd kan worden. In de SSA-variant wordt eerst een vector samengesteld uit de twee afzonderlijke ingangen. Daarna kan EXOR worden beschreven als een waarheidstabel.

```
s <= '1' when a = '0' and b = '1' else
     '1' when a = '1' and b = '0' else
     '0';
```

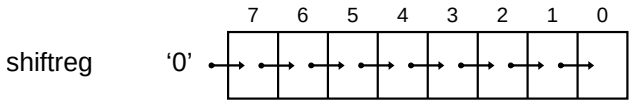
Listing A.3: *CSA-code voor functie s.*

```
ab_vec <= a & b;
with ab_vec select
    s <= '0' when "00",
         '1' when "01",
         '1' when "10",
         '0' when "11",
         'X' when others;
```

Listing A.4: *SSA-code voor functie s.*

Uitwerking opgave 7.4.

Zie de figuur hieronder. De toekenning verloopt als volgt: de '0' wordt toegekend aan element 7. Element 7 wordt toegekend aan 6. Zo gaat dat verder.



Figuur A.63: *Uitbeelding van naar rechts schuiven.*

In code wordt dat:

```
shiftreg(7) <= '0';
shiftreg(6) <= shiftreg(7);
shiftreg(5) <= shiftreg(6);
shiftreg(4) <= shiftreg(5);
shiftreg(3) <= shiftreg(4);
shiftreg(2) <= shiftreg(3);
shiftreg(1) <= shiftreg(2);
shiftreg(0) <= shiftreg(1);
```

Listing A.5: *Code voor het naar links schuiven van een 8-bits waarde.*

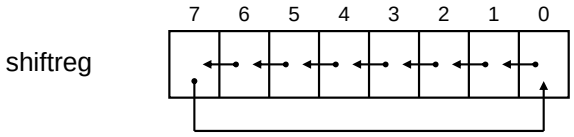
Of sneller:

```
shiftreg <= '0' & shiftreg(7 downto 0);
```

Listing A.6: *Code voor het naar links schuiven van een 8-bits waarde.*

Uitwerking opgave 7.5.

In de onderstaande figuur is weergegeven wat er moet gebeuren. Alle elementen moeten één plaats naar links worden geschoven en het hoogste element moet toegekend worden aan het laagste element. Dit wordt roteren genoemd. Elke microprocessor heeft een rotate-instructie aan boord.



Figuur A.64: *Uitbeelding van het linksom roteren van een 8-bits waarde.*

De VHDL-code moet zo geschreven worden dat element 6 toegekend wordt aan element 7, element 5 aan element 4 etc. Element 0 moet toegekend worden aan element 1. Element 7 wordt toegekend aan element 0.

```
1 shiftreg(7) <= shiftreg(6);
2 shiftreg(6) <= shiftreg(5);
3 shiftreg(5) <= shiftreg(4);
4 shiftreg(4) <= shiftreg(3);
5 shiftreg(3) <= shiftreg(2);
6 shiftreg(2) <= shiftreg(1);
7 shiftreg(1) <= shiftreg(0);
8 shiftreg(0) <= shiftreg(7);
```

Listing A.7: *Code voor het linksom roteren van een 8-bits waarde.*

Deze acht toekenningen kunnen veel sneller opgeschreven worden door gebruik te maken van vector slices.

```
1 shiftreg(7 downto 1) <= shiftreg(6 downto 0);
2 shiftreg(0)          <= shiftreg(7);
```

Listing A.8: *Code voor het linksom roteren van een 8-bits waarde.*

Of nog sneller (als in: minder code):

```
1 shiftreg(7 downto 0) <= shiftreg(6 downto 0) & shiftreg(7);
```

Listing A.9: *Code voor het linksom roteren van een 8-bits waarde.*

Of nóg sneller (als in: nóg minder code):

```
1 shiftreg <= shiftreg(6 downto 0) & shiftreg(7);
```

Listing A.10: *Code voor het linksom roteren van een 8-bits waarde.*

Uitwerking opgave 7.6.

De code is hieronder weergegeven.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity sr_latch_w_domset is
5     port ( s : in    std_logic;
6            r : in    std_logic;
7            q : out   std_logic);
8 end sr_latch_w_domset;
9
```

```
architecture logic of sr_latch_w_domset is
signal q_int : std_logic;
signal notq  : std_logic;
begin

q     <= q_int;
q_int <= s nand notq;
notq  <= r nand q_int;

end logic;
```

Listing A.11: *Code voor een SR-latch met overheersende set.*

Uitwerking opgave 7.7.

Hieronder is een aantal oplossingen gegeven. De eerste twee oplossingen gaan uit van de definitie van de EXOR-poort en zijn min of meer recht-toe-recht-aan.

```
if a = '1' and b = '0' then
    f <= '1';
elsif a = '0' and b = '1' then
    f <= '1';
else
    f <= '0';
end if;
```

Listing A.12: *Code voor een EXOR-poort.*

```
f <= '0'    -- default
if a = '1' and b = '0' then
    f <= '1';
elsif a = '0' and b = '1' then
    f <= '1';
end if;
```

Listing A.13: *Code voor een EXOR-poort.*

Bij onderstaande linker oplossing is gebruik gemaakt van het feit dat in VHDL ook relationele tests kunnen worden beschreven. Hier wordt gekeken of de waarde van a *ongelijk* is aan de waarde van b. In de rechter oplossing is de functie in feite als een waarheidstabel beschreven.

```
if a /= b then     -- not equal
    f <= '1';
else
    f <= '0';
end if;
```

Listing A.14: *Code voor een EXOR-poort.*

```
ab_vec <= a & b;
case ab_vec is
    when "00" => f <= '0';
    when "01" => f <= '1';
    when "10" => f <= '1';
    when "11" => f <= '0';
    when others => f <= 'X';
end case;
```

Listing A.15: *Code voor een EXOR-poort.*

Uitwerking opgave 7.8.

Hieronder de complete VHDL-beschrijving van de negative edge triggered D-flipflop.

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

entity dffneg is
port (clk    : in std_logic;
      d      : in std_logic;
```

```
      q       : out std_logic
     );
end entity dffneg;

architecture behavioral of dffneg is
begin

    process (clk) is
    begin
        if falling_edge(clk) then
            q <= d;
        end if;
    end process;

end architecture behavioral;
```

Listing A.16: *Code voor een negative edge triggered D-flipflop.*

Uitwerking opgave 7.9.

Een schuifregister bestaat uit een groep D-flipflops waarvan de uitgang van een flipflop gekoppeld is aan de ingang van een volgende flipflop. Afhankelijk van de exacte aansluitingen (en het naar buiten brengen van de flipflopinhouden) kan je rechtsom schuiven of linksom schuiven. Er moet gebruik gemaakt worden van flipflops, dat zijn flankgevoelige geheugenelementen. Er is dus een flankbeschrijving nodig in de VHDL-code. Zie onderstaande listing. De asynchrone reset is weggelaten maar moet normaal gesproken wel geïmplementeerd worden.

```
process (clk) is
begin
    if rising_edge(clk) then
        shiftreg <= shiftreg(6 downto 0) & '0';
    end if;
end process;
```

Listing A.17: *Code voor het naar links schuiven van een 8-bits register.*

Uitwerking opgave 7.10.

De code is gegeven in de onderstaande listing.

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

entity bcd_adder is
    port (a : in unsigned(3 downto 0);
          b : in unsigned(3 downto 0);
          cin : in std_logic;
          sum : out unsigned(3 downto 0);
          cbcd : out std_logic;
          valid : out std_logic
         );
end entity bcd_adder;
```

```
14
15 architecture rtl of bcd_adder is
16 begin
17
18     process (a, b, cin) is
19     variable z : unsigned(4 downto 0);
20     begin
21         z := ('0' & a) + ('0' & b) + ("0" & cin);
22         if z > 15 then
23             z := z + 6;
24         end if;
25
26         sum <= z(3 downto 0);
27         cbcd <= z(4);
28
29         if (a < 10) and (b < 10) then
30             valid <= '1';
31         else
32             valid <= '0';
33         end if;
34     end process;
35
36 end architecture rtl;
```

Listing A.18: *Code voor het optellen vabn twee BCD-cijfers.*

Uitwerking opgave 7.11.

Hieronder staan twee listings voor het probleem. Bij de listing links wordt niet de hele array (vector) afgelopen. Variabele `p` wordt eerst gelijk gemaakt aan de waarde van het meest significante element. Daarna moeten de overige elementen verwerkt worden. Dit scheelt één lusdoorgang.

De listing rechts maakt gebruik van de definitie van de OR-poort: de uitgang wordt '1' als één of meer ingangen '1' zijn. Als een array-element '1' blijkt te zijn, is de uitgang ook '1'. Zijn geen van de array-elementen '1' dan wordt variabele `p` nooit op '1' gezet. Na het doorlopen van de lus is `p` dan nog steeds '0'. Merk op dat er geen **`else`** gebruikt mag worden.

```
1 p := x(7);
2 for i in 6 downto 0 loop
3     p := p or x(i);
4 end loop;
5 q <= not p;
```

Listing A.19: *Code 8-input NOR-poort.*

```
1 p := '0';
2 for i in 7 downto 0 loop
3     if x(i) = '1' then
4         p := '1';
5     end if;
6 end loop;
7 q <= not p;
```

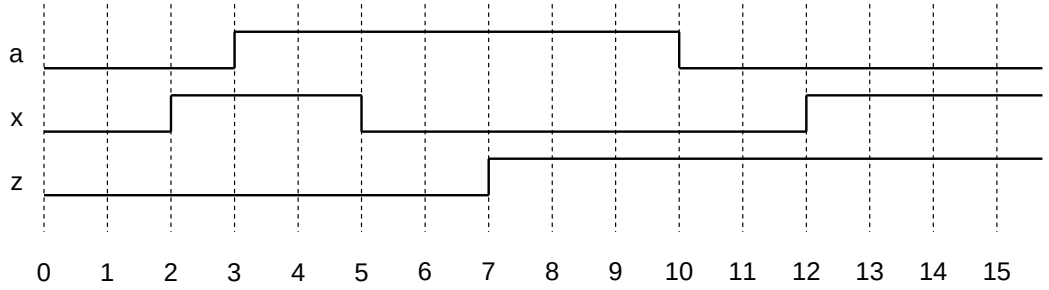
Listing A.20: *Code 8-input NOR-poort.*

Uitwerking opgave 7.12.

Zie uitwerking van opgave 7.13

Uitwerking opgave 7.13.

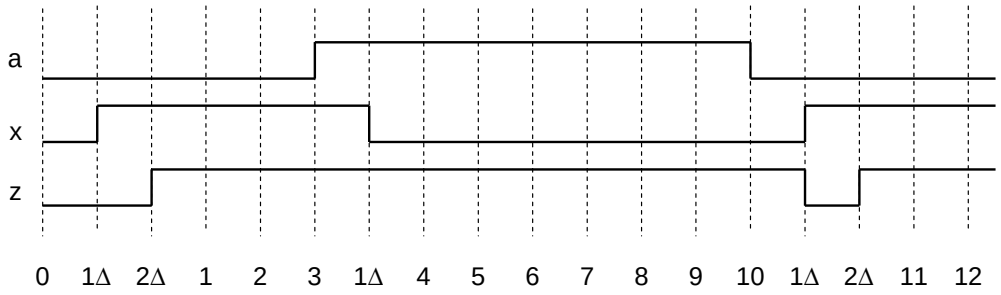
Hieronder is het tijddiagram te zien. Let erop dat de inverter signaalpulsen korter dan 2 ns niet op de uitgang laat zien. Voor de OR-poort is dat 5 ns. Dat laatste kan goed gezien worden in onderstaand tijddiagram. Op tijdstip 10 ns wordt ingang a logisch '0' en 2 ns later (tijdstip 12 ns) wordt signaal x logisch '1'. Dat levert een puls van 2 ns (de puls is laag) die de OR-poort niet doorgeeft.



Figuur A.65: *Timingdiagram.*

Uitwerking opgave 7.14.

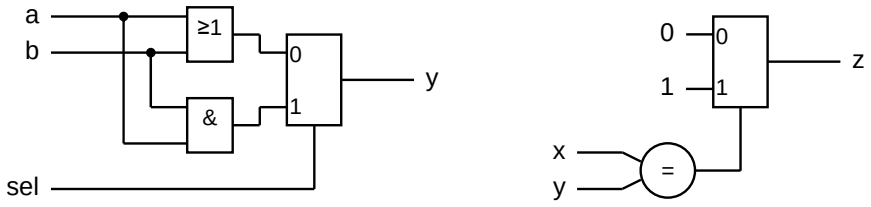
Het tijddiagram heeft zowel 'echte' tijden als delta-delays. Er zijn geen vertragingen opgegeven, dus worden alle toekenningen één delta later actief. Alle pulsen worden doorgegeven, want de minimale pulsbreedte is één delta.



Figuur A.66: *Timingdiagram.*

Uitwerking opgave 7.15.

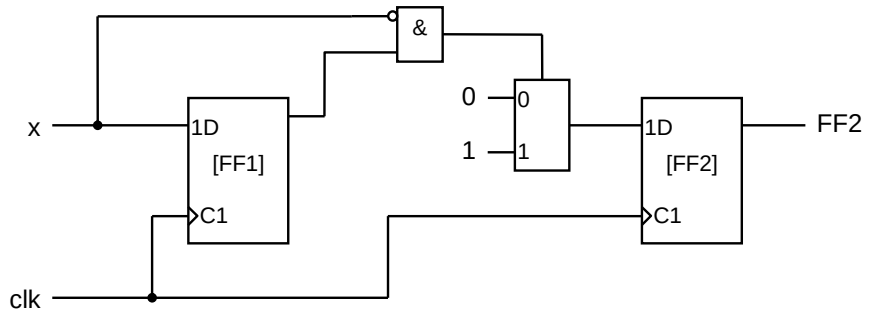
In beide stukken code wordt een **`if ... then ... else`** gebruikt zodat er multiplexers worden gesynthetiseerd. In de code links is signaal `sig` één bit breed, dus het signaal kan direct dienen om de multiplexer aan te sturen. Afhankelijk van de waarde van `sig` moet een OR-functie of een AND-functie gerealiseerd worden. In de code rechts wordt de relationele operator = gebruikt. Dat houdt in dat er een vergelijkschakeling moet worden gesynthetiseerd. Verder is te zien dat als `x` gelijk is aan `y` de uitgang logisch '1' wordt en anders logisch '0'. De constanten '0' en '1' worden op de data- ingangen van de multiplexer aangesloten. In feite hoeft de multiplexer niet gesynthetiseerd te worden, de vergelijkschakeling geeft al een logische '1' als `x` gelijk is aan `y`. De signalen `x` en `y` kunnen vectoren zijn (beide met hetzelfde aantal elementen) of kunnen beide uit één bit bestaan. In dat laatste geval kan de gehele rechter schakeling vereenvoudigd worden tot een EXNOR-poort.



Figuur A.67: *Gerealiseerde schema's voor VHDL-code.*

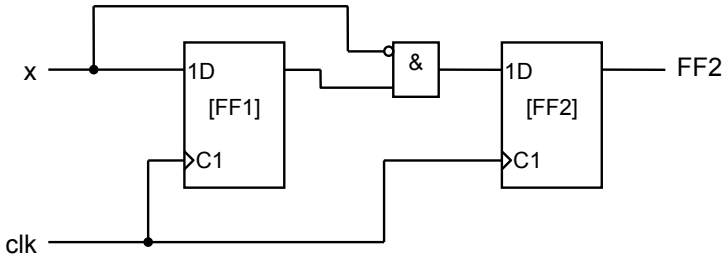
Uitwerking opgave 7.16.

De schakeling bestaat uit twee D-flipflops. Er worden namelijk toekenningen gedaan aan twee signals onder besturing van een klokflank. Er wordt een **`if...then...else`** gebruikt, dat levert een multiplexer op. In de test van de **`if`** wordt een AND-constructie gebruikt (samen met een inverter), dat worden dan logische poorten. Zie onderstaande schakeling.



Figuur A.68: *Gerealiseerd schema voor VHDL-code.*

De constanten '0' en '1' worden op de data-ingangen van de multiplexer aangesloten. In feite hoeft de multiplexer niet gesynthetiseerd te worden, de schakeling voor de test geeft al een logische '1' als deze waar is. Zie onderstaande schakeling.



Figuur A.69: *Geoptimaliseerd schema voor VHDL-code.*

Uitwerking opgave 7.17.

In de figuur zijn twee flipflops te zien. Dat houdt in dat de toekenningen onder flankbesturing moet worden geplaatst. De flipflops hebben een asynchrone reset, actief laag, dat kan met een **`if`**-statement worden beschreven. De multiplexer kan worden beschreven met een **`if ... then .. else ... end if`**. Zie onderstaande code. Een en ander kan wat compacter beschreven worden. Zo kan intern signaal `ff2` vermeden worden en vervangen worden door signaal `q` zelf.

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

entity opgave33 is
    port (clk : in std_logic;
          areset : in std_logic;
          sel : in std_logic;
          a : in std_logic;
          q : out std_logic);
end entity opgave33;

architecture behav of opgave33 is
signal ff1, ff2 : std_logic;
begin

    process (clk, areset) is
    begin
        if areset = '0' then
            ff1 <= '0';
            ff2 <= '0';
        elsif rising_edge(clk) then
            ff1 <= a;
            if sel = '1' then
                ff2 <= ff1;
            else
                ff2 <= a;
            end if;
        end if;
    end process;

    q <= ff2;

end architecture behav;
```

Listing A.21: *VHDL-code voor de schakeling.*

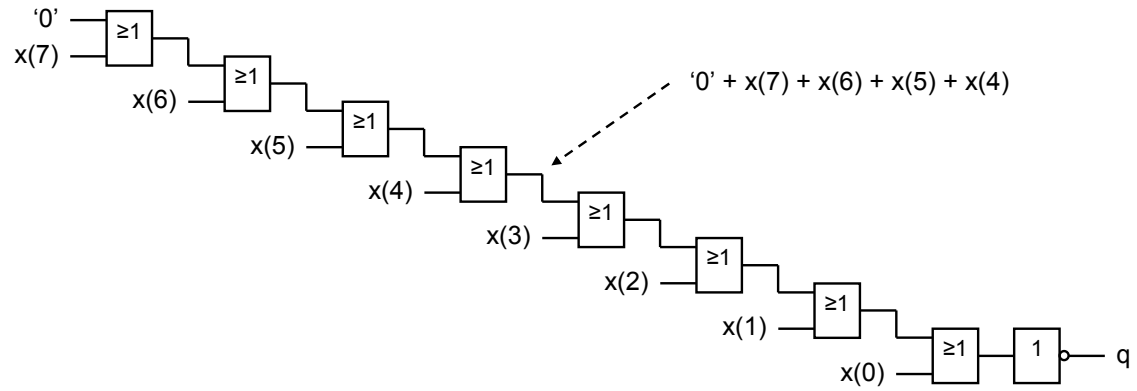
Uitwerking opgave 7.18.

In de code wordt de variabele `p` eerst op '0' gezet. De eerste keer dat de **`for`**-lus wordt doorlopen, wordt de (huidige) waarde van `p` (en die is '0') ge-OR-d met `x(7)` met als resultaat dat `p` nu gelijk aan `x(7)`. In de volgende iteratie wordt `p` ge-OR-d met `x(6)`. Dat gaat zo door tot de lus is afgelopen, dus als laatste wordt `p` ge-OR-d met `x(0)`. We noemen dit ook wel *loop unrolling*. Daarna wordt de inverse van `p` toegekend aan `q`. In onderstaande figuur is het schema te zien. Merk op dat een slimme compiler na uitwerking van de **`for`**-lus en de NOT direct een 8-input NOR-poort genereert.

Uitwerking opgave 7.19.

De beschrijving is in onderstaande listing te zien. We gebruiken de vermenigvuldig-operator `*` om de signalen `a` en `b` te vermenigvuldigen en kennen het resultaat direct toe aan signaal `p`. Let er wel op dat het product `p` uit 64 bits bestaat.

Op zich is dit een prima manier om twee getallen te vermenigvuldigen, maar we geven



Figuur A.70: *Schema voor de 8-input NOR-poort op basis van loop unrolling.*

wel een aantal zaken uit handen. Ten eerste weten we niet hoe de synthesizer de vermenigvuldiging realiseert. We weten dus niet hoe groot de vermenigvuldiger wordt. Ten tweede weten we op voorhand niet hoe lang de vermenigvuldiging duurt. Het is daarom lastig om te voorspellen of de gehele schakeling aan de timingeisen voldoet. Willen we dit onder controle houden, dan moeten we de vermenigvuldiger op een andere manier beschrijven. Een voorbeeld hiervan een *pipe line*-vermengvuldiger, waarbij we de vermenigvuldiging opdelen in kleinere eenheden met tussenliggende registers waarna de deelresultaten opgeteld worden. De vermenigvuldiging kan dan niet in één klokcyclus berekend worden maar de *overall*-snelheid van de schakeling kan nu wel beter geregeld worden.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity mult32 is
6      port (a, b : in unsigned(31 downto 0);
7            p : out unsigned(63 downto 0)
8           );
9  end entity mult32;
10
11 architecture rtl of mult32 is
12 begin
13     p <= a * b;
14 end architecture rtl;
```

Listing A.22: *VHDL-beschrijving van een 32x32-bits unsigned vermenigvuldiger.*

A.8 Uitwerkingen hoofdstuk 8

Uitwerking opgave 8.9.

Een mogelijke oplossing staat hieronder. Hierin is `data_in` een 4-bit ingang waarop de te laden stand staat. Door signaal `load` logisch '1' te maken worden de data-ingangen geladen in de interne stand.

```
1      process (clk) is
```

```
    variable count : unsigned(3 downto 0);
    begin
        if rising_edge(clk) then
            if load = '1' then
                count := data_in;
            elsif enable = '1' then
                count := count + 1;
            end if;
        end if;
        q <= count;
    end process;
```

Listing A.23: *VHDL-code voor een laadbare teller.*

Uitwerking opgave 8.1.

Een JK-flipflop met beide ingangen (J en K) aan elkaar aangesloten, gedraagt zich als een T-flipflop. Zie ook http://nl.wikipedia.org/wiki/Flipflop en bekijk de tabellen van de JK-flipflop en de T-flipflop.

Uitwerking opgave 8.2.

Van een zuivere binaire omhoog-teller waarbij alle telstanden worden doorlopen (0 t/m $2^n - 1$ met n telbits) kan een omlaag-teller worden gemaakt door alle uitgangen van de telbits te inverteren. Dat kan niet als niet de volledige cyclus wordt doorlopen. Zie onderstaande 3-bit teller. De linker kolom laat de opeenvolgende telstanden zien van een omhoog-teller, de rechter kolom van een omlaag-teller.

Tabel A.27: *Telvolgorde van een omhoog- en omlaagteller.*

up	down
000	111
001	110
010	101
011	100
100	011
101	010
110	001
111	000

Uitwerking opgave 8.3.

Eerst even de T-flipflop bespreken. Een T-flipflop is een flankgevoelig geheugenelement dat van stand verandert (een 0 wordt een 1 en een 1 wordt een 0) als de stuuringang (T) logisch '1' is. De stand blijft ongewijzigd als de stuuringang logisch '0' is.

De 6-teller begint bij 000_2 en dan naar 001_2, 010_2, etc. Na stand 101_2 volgt weer stand 000_2 want de teller is aan het einde van zijn cyclus. In de onderstaande tabel zijn de doorlopen telstanden onder elkaar gezet.

Tabel A.28: *Telcyclus van de 6-teller.*

telstand	opmerking
000	begin nieuwe cyclus
001	
010	Q_1 toggelt
011	
100	Q_2, Q_1 toggelt
101	
000	Q_2 toggelt, begin nieuwe cyclus

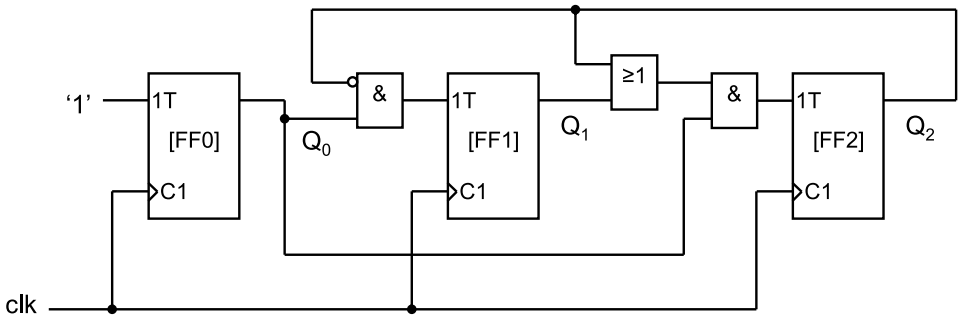
Wat direct opvalt is dat het minst significante telbit (Q_0) bij elke volgende stap wisselt. Dat houdt in dat de bijbehorende flipflop continue moet toggelen. Dat kan eenvoudig gedaan worden door de T-ingang van de flipflop op logisch '1' te zetten. Verder is te zien dat dat het middelste telbit (Q_1) moet toggelen bij stand 001_2 en 011_2 is. De meest significante telbit (Q_2) moet toggelen bij stand 011_2 en 101_2. Nu is 011_2 de enige telstand waarbij $Q_1Q_0 = 11$ dus kan Q_2 verwijderd worden. Stand 101_2 is de enige stand waarbij $Q_2Q_0 = 11$ dus kan Q_1 verwijderd worden.

De standen waarbij de flipflops moeten toggelen moeten uitgecodeerd worden. Er worden schakelfuncties gerealiseerd voor de T-ingangen van de flipflops. Een logische '1' doet een T-flipflop toggelen. Dus wanneer een flipflop moet toggelen moet de bijbehorende schakelfunctie een logische '1' geven.

Na enig rekenwerk worden de volgende functies gevonden:

$$\begin{aligned} T_0 &= 1 \\ T_1 &= \overline{Q_2} \cdot Q_0 \\ T_2 &= Q_2 \cdot Q_0 + Q_1 \cdot Q_0 = Q_0 \cdot (Q_2 + Q_1) \end{aligned} \tag{A.71}$$

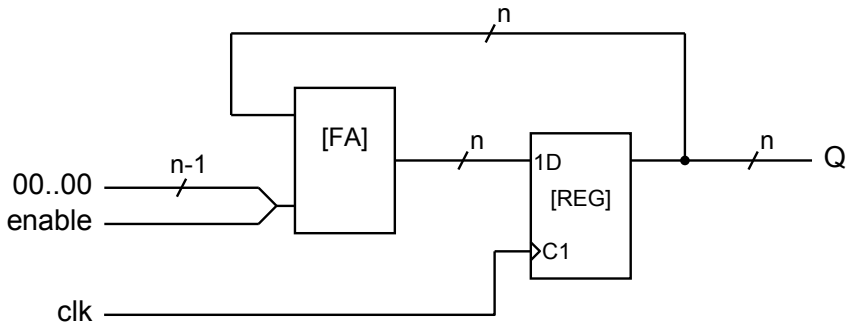
De volledige teller is in onderstaande figuur weergegeven.



Figuur A.71: *Schema van de 6-teller op basis van T-flipflops.*

Uitwerking opgave 8.4.

Een teller is ook op te bouwen met een full adder en een register. Het register onthoudt dan de laatst ingebrachte telstand. Het vermeerderen van de telstand wordt gedaan door de full adder. Op één set ingangen wordt de huidige telstand aangeboden, op de andere set ingangen wordt het binaire getal $00\ldots01_2$ aangeboden. Na een actieve klokflank wordt de nieuwe telstand ingeklokt. Uiteraard werkt de teller modulo 2^n waarbij n het aantal telbits is. Door het getal $00\ldots01_2$ aan te passen kunnen andere telvolgorden realiseren. Een interessante optie is om de minst significante bit van het vaste getal schakelbaar te maken tussen 0 en 1. Is deze bit een 1 dat wordt de teller vermeerderd met 1. Is deze bit 0, dan blijft de huidige telstand bewaard. Deze bit kan dus als een enable-signaal dienen. Een andere interessante optie is om alle bits 1 te maken, dus de telstand te vermeerderen met $11\ldots11_2$. Het effect is dat de telstand met 1 wordt verlaagd. We kunnen dus zo een down-counter maken.



Figuur A.72: *Realisatiw van een enable bij een teller op basis van een opteller.*

Uitwerking opgave 8.10.

Hieronder de (ingekorte) listing van de eerder beschreven BCD-teller.

```
1  begin
2      process (clk, areset) is
3      begin
4          if rising_edge(clk) then
5              if count = "1001" then -- teller in hoogste stand?
6                  count <= "0000"; -- ... dan teller weer 0
7              else
8                  count <= count + 1; -- anders gewoon tellen
9              end if;
10         end if;
11     end process;
12     tc <= '1' when count = "1001" and en = '1' else '0';
13     q <= count;
14 end architecture rtl;
```

Listing A.24: *Verkorte code van de BCD-teller.*

In de beschrijving wordt de gelijkheid van `count` met 9_{10} getest. Dan wordt `count` gelijk aan 0_{10}. Bij ongelijkheid, dus ook als de telstand 12_{10} is, wordt `count` verhoogd. De teller zal doorlopen tot 15_{10} (het is immers een 4-bit teller) en dan naar 0_{10} "omklappen". De uitgang `tc` wordt dan *niet* logisch '1', dat gebeurt alleen als de telstand 9_{10} is (en `en` is '1').

Uitwerking opgave 8.11.

In feite staat de teller in een niet gebruikte telstand en moet daar zo snel mogelijk uit komen. De teller moet dan geladen worden met 0_{10}. Dat kan door de = te vervangen door >= bij beide tests:

```
1 if count >= "1001" then ...
```

Listing A.25: *Test op groter dan of gelijk aan 9.*

```
1 tc <= '1' when count >= "1001" and en = '1' else '0';
```

Listing A.26: *Aangepaste code voor de terminal count.*

Uitwerking opgave 8.12.

De test op het hoogste getal moet aangepast worden, net als de de functie voor de `tc`. Hier wordt ook gebruik gemaakt van de >= operator.

```
1 if count >= "1100" then ...
```

Listing A.27: *Test op groter dan of gelijk aan 12.*

```
1 tc <= '1' when count >= "1100" and en = '1' else '0';
```

Listing A.28: *Aangepaste code voor de terminal count.*

Uitwerking opgave 8.13.

Natuurlijk kunnen de twee benodigde bits getest worden via onderstaande code:

```
1 if count(3) = '1' and count(0) = '1' then ...
```

Listing A.29: *Test voor het patroon 1– –1.*

Maar de bibliotheek `numeric_std` heeft ook een functie `stdmatch` aan boord die twee vectoren vergelijkt met don't cares:

```
1 if stdmatch(count, "1--1") then ...
```

Listing A.30: *Test voor het patroon 1– –1 met stdmatch.*

Uitwerking opgave 8.14.

De telcyclus van de AM/PM-klok is 1-2-3-4-5-6-7-8-9-10-11-12 waarbij telstand 12 gelijk is aan 0 (middernacht) of aan het begin van de middag. Dat houdt in dat de telstand 0 niet voorkomt. De telcyclus van de 24-uursklok is van 0 t/m 23. De telstand 0 komt dus wel voor. Dit levert gelijk een probleem bij de asynchrone reset. Waarmee moet de teller gereset worden? De asynchrone reset is een systeemsignaal en mag niet door combinatorische logica gestuurd worden. Dus een constructie als

```
1 if areset = '1' and ampm = '1' then ....
```

is **verboden**. Wat we wel kunnen doen is bij AM/PM de telstand 0 vervangen door 12. De teller telt dan gewoon van 0 t/m 11 maar 0 wordt vervangen door 12. Bij een 24-uursklok wordt 0 gewoon doorgegeven als 0. Dus volgt

```
hour <= "01100" when ampm = '1' and counter = "00000" else counter;
```

De complete beschrijving van deze urenteller is te vinden in listing A.31.

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

entity opgave5 is
    port (clk : in std_logic;                -- The clock
          areset : in std_logic;             -- The reset
          up : in std_logic;                 -- Count up
          ampm : in std_logic;               -- 0 = 24 hr, 1 = 12 hr
          hour : out unsigned (4 downto 0)   -- Hour counter
   );
end entity opgave5;

architecture rtl of opgave5 is
signal counter : unsigned(4 downto 0);       -- Internal hour counter
begin

    process (clk, areset) is
    begin
        if areset = '1' then
            counter <= "00000";
        elsif rising_edge(clk) then
            if up = '1' then
                if ampm = '1' then  -- 12 hour
                    if counter > "01010" then
                        counter <= "00000";
                    else
                        counter <= counter + 1;
                    end if;
                else -- 24 hour clock
                    if counter >= "10111" then
                        counter <= "00000";
                    else
                        counter <= counter + 1;
                    end if;
                end if;
            end if;
        end if;

    end process;

    hour <= "01100" when ampm = '1' and counter = "00000" else counter;
end architecture rtl;
```

Listing A.31: *Een urenteller voor AM/PM en 24 uur.*

Uitwerking opgave 8.5.

De flipflopfuncties van de LFSR zijn:

$$\begin{aligned} Q_2^{n+1} &= Q_1^n \\ Q_1^{n+1} &= Q_0^n \\ Q_0^{n+1} &= Q_2^n \oplus Q_0^n \end{aligned} \tag{A.72}$$

Als geldt dat bij $Q_2^n Q_1^n Q_0^n = 111$ dan is de opvolgerstand $Q_2^n Q_1^n Q_0^n = 110$, FF2 krijgt immers de waarde van FF1, FF1 krijgt de waarde van FF0 en FF0 krijgt de EXOR van FF2 en FF0. De volledige cyclus is 111-110-010-100-001-011-111. De LFSR doorloopt dus zeven (tel-)standen. Merk op dat de telstand 000 niet voorkomt. Als de LFSR daarin terecht komt, blijft de LFSR in 000. We noemen dit trouwens een *Maximum Length* LFSR. Deze schakeling komt veel voor in *Pseudo Random Generator*, waarbij het genereren van willekeurige getallen nodig is.

Uitwerking opgave 8.6.

Met een n-bit teller kunnen 2^n stappen worden gemaakt. De resolutie is de reciproke daarvan. Duty cycles worden altijd in procenten gegeven (maar er wordt niet met procenten gerekend), dus de resolutie bij n bits is

$$DC = \frac{1}{2^n} \cdot 100\% \tag{A.73}$$

In tabel A.29 zijn de resoluties van enkele bitbreedtes gegeven.

Tabel A.29: *Bitbreedte versus resolutie.*

bits	resolutie (%)
4	6,25
8	0,39
10	0,098
16	0,0015

Uitwerking opgave 8.7.

Een groter-of-gelijk-schakeling lost het probleem bij de maximale waarde op. Als de waarde van de teller gelijk is aan de referentiewaarde, zal de vergelijker een logische '1' geven. Het probleem is nu dat het PWM-signaal niet meer volledig "uit" kan. Als de referentiewaarde 0 is, zal de schakeling namelijk een logische '1' geven als de teller ook op 0 staat. De Duty Cycle is dan minimaal $1/2^n \times 100\%$.

Uitwerking opgave 8.8.

Dit kan heel eenvoudig door de teller die gebruikt wordt voor generatie van het PWM-signaal niet helemaal een volledige 2^n-telcyclus te laten doorlopen, maar bijvoorbeeld een $2^n - 1$-telcyclus. De cyclus loopt van 0 t/m $2^n - 2$ en de gebruiker kan wél $2^n - 1$ als referentiewaarde opgeven. Neem bijvoorbeeld een 8-bit PWM-teller. De teller loopt van 0 t/m 254 en begint dan weer opnieuw. De gebruiker kan wel 255 instellen als referentiewaarde.

Uitwerking opgave 8.16.

De toekenning aan `tc` gebeurt onder klokflanksturing. Dat betekent dat de toekenning pas één klokflank later wordt uitgevoerd, er wordt immers een flipflop voor `tc` gesynthetiseerd. In de praktijk is `tc` dus logisch 1 op telstand 0000_{BCD}. Verder is er geen toekenning gegeven aan `tc` als `enable` '0' is. `tc` zou dus langdurig op '1' kunnen blijven staan. Ook komt `tc` niet voor in de asynchrone reset. Voor correcte werking moet voor `tc` een combinatorische functie worden gerealiseerd.

Uitwerking opgave 8.15.

De Duty Cycle van de PWM-generator moet in stappen van 0,5% kunnen worden ingesteld. Dat betekent dat de PWM-generator 200 stappen moet maken voor één PWM-cyclus. De interne teller, die de stappen bijhoudt, moet dus van 0 t/m 199 tellen en dan weer opnieuw beginnen. We zouden een integer teller en referentiewaarde met een bereik van 0 t/m 199 kunnen gebruiken, maar de aangeboden referentiewaarde is een 8-bits unsigned getal. Het is daarom beter om de interne teller en referentiewaarde als integers te declareren met een volledig 8-bits bereik. Bij synthese wordt de interne teller en de referentiewaarde dus als een 8 bits unsigned getal gesynthetiseerd. Natuurlijk moet de teller van 0 t/m 199 tellen en dan weer opnieuw beginnen. Er moet dus wél getest worden op 199. Op dat moment moet de teller weer op 0 gezet worden én moet een nieuwe referentiewaarde worden ingeladen. De PWM-uitgang is een combinatorische functie en de code moet buiten de klokflank worden beschreven. Ook het top-signaal, dat aangeeft dat de PWM-cyclus geëindigd is, moet als combinatorische functie worden beschreven. Merk op dat een referentiewaarde groter dan 199 kan worden opgegeven. De PWM-uitgang is in dat geval continu '1' (een DC van 100%). Hieronder is de implementatie van de PWM-generator te zien.

A.9 Uitwerkingen hoofdstuk 9

Uitwerking opgave 9.1.

Voor het bepalen van de maximale frequentie moet gekeken worden welk pad de data volgt van flipflop naar flipflop. Er is er maar één, dus dat is makkelijk. Vanuit de uitgang van de flipflop komt de data aan bij de ingang van de flipflop. Het pad is dus FF1 →FF1. Er is geen sprake van klokskew. Dat was ook niet interessant geweest, want de data komt bij dezelfde flipflop aan. De minimale periodetijd van dit systeem wordt opgebouwd uit de vertraging van de data van de flipflop en de setuptijd van de flipflop. Zie figuur A.73.

De minimale periodetijd wordt berekend.

$$T_{min} = t_{P(max)}(\text{FF}) + t_{su}(\text{FF}) = 35 + 15 = 50 \text{ ns} \tag{A.74}$$

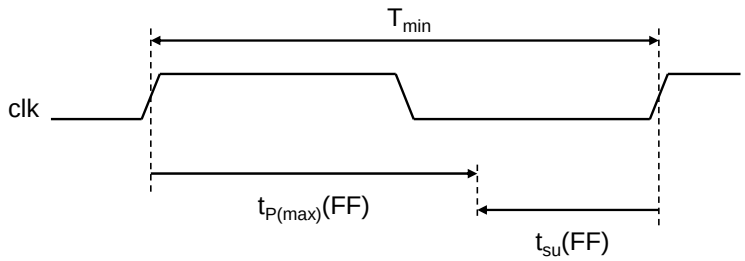
De maximale frequentie is 20 MHz.

De holdtijd is kleiner dan de minimale propagatietijd van de flipflop, dat houdt in dat er geen *hold time violation* is:

$$t_{slack,h} = t_{P(min)}(\text{FF}) - t_h(\text{FF}) = 10 - 5 = 5 \text{ ns} \tag{A.75}$$

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

entity opgave5 is
    port (clk : in std_logic;                -- The clock
          areset : in std_logic;             -- The reset
          ref : in unsigned (7 downto 0);    -- The reference value
          pwm_out : out std_logic;           -- PWM output signal
          top : out std_logic                -- Top of counter
   );
end entity opgave5;

architecture rtl of opgave5 is
-- Internal counter and internal reference
signal counter : integer range 0 to (2**ref'length)-1;
signal reference : integer range 0 to (2**ref'length)-1;
begin

    process (clk, areset) is
    begin
        if areset = '1' then
            counter <= 0;
            reference <= 0;
        elsif rising_edge(clk) then
            if counter = 199 then
                counter <= 0;
                reference <= to_integer(ref);
            else
                counter <= counter + 1;
            end if;
        end if;

    end process;

    pwm_out <= '1' when reference > counter else '0';
    top <= '1' when counter = 199 else '0';

end architecture rtl;
```

Listing A.32: *Realisatie PWM-generator met DC-resolutie van 0,5%.*



Figuur A.73: *Timing maximale frequentie van de tweedeler.*

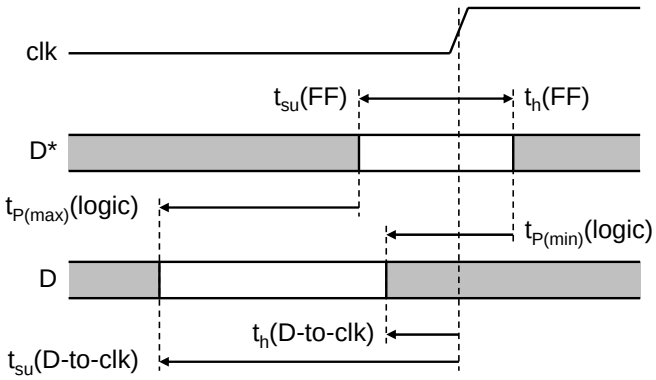
Er is betrouwbare dataoverdracht mogelijk.

Uitwerking opgave 9.2.

De flipflop heeft een setup- en holdtijd ten opzichte van de actieve flank van de klok (de opgaande flank in dit geval). In dat gebied moet de D-ingang van de flipflop (`D*`) stabiel zijn. Voor de flipflop zit nog wat logica en die heeft een bepaalde vertragingstijd. Het zal dan duidelijk zijn dat de setuptijd van de D-ingang (de D-ingang bij de logica, niet de flipflop) t.o.v. van de opgaande flank van de klok "naar voren" wordt verschoven. In dit geval moet de maximale propagatietijd genomen worden. Dat is ook te verklaren: als de data op de D-ingang stabiel wordt gehouden, dan duurt het op zijn langst nog de maximale vertragingstijd voordat de stabiele data bij `D*` aankomt. Precies op dat moment gaat de setuptijd van `D*` in.

Voor de holdtijd geldt eenzelfde verhaal, maar nu moet de minimale vertragingstijd genomen worden. Ook dat is te verklaren. Als de stabiele data weer gaat veranderen, dan zal op zijn vroegst de data bij `D*` na de minimale vertragingstijd veranderen. Precies op dat moment is de holdtijd bij `D*` verstreken.

Eén en ander is in figuur A.74 getekend.



Figuur A.74: *Setup- en holdtijd van de D-flipflop met enable.*

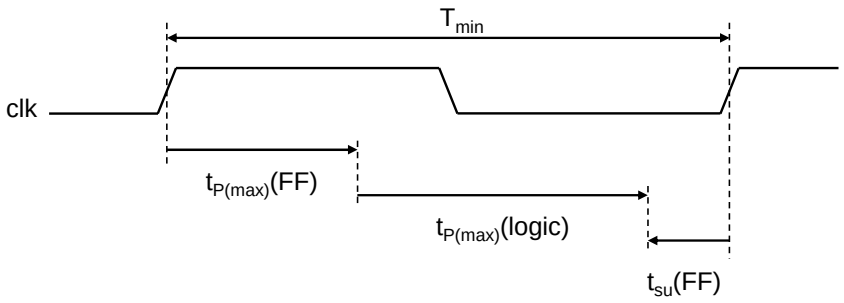
Voor de setup- en holdtijd t.o.v. de klok zijn twee formules op te stellen.

$$\begin{aligned} t_{su}(\text{D-to-clk}) &= t_{su}(\text{FF}) + t_{P(max)}(\text{logic}) \\ t_h(\text{D-to-clk}) &= t_h(\text{FF}) - t_{P(min)}(\text{logic}) \end{aligned} \tag{A.76}$$

Merk op dat in de formule van de holdtijd een minteken staat. Dat komt omdat de holdtijd naar rechts wordt getekend (positieve waarde). Invullen en uitrekenen levert op:

$$\begin{aligned} t_{su}(\text{D-to-clk}) &= 14 + 19 = 33 \text{ ns} \\ t_h(\text{D-to-clk}) &= 4 - 8 = -4 \text{ ns} \end{aligned} \tag{A.77}$$

Merk op dat de holdtijd *negatief* is. Dat houdt in dat de data op `D` nog voor de klokflank mag veranderen. Zie ook de bovenstaande figuur.



Figuur A.75: *Bepaling minimale periodeduur van de D-flipflop met enable.*

Voor het bepalen van de maximale frequentie moet het pad worden bepaald dat de data aflegt van uitgang naar ingang. Het pad is FF → logica → FF. Dat duurt minimaal één klokperiode. In figuur A.75 is dat goed te zien.

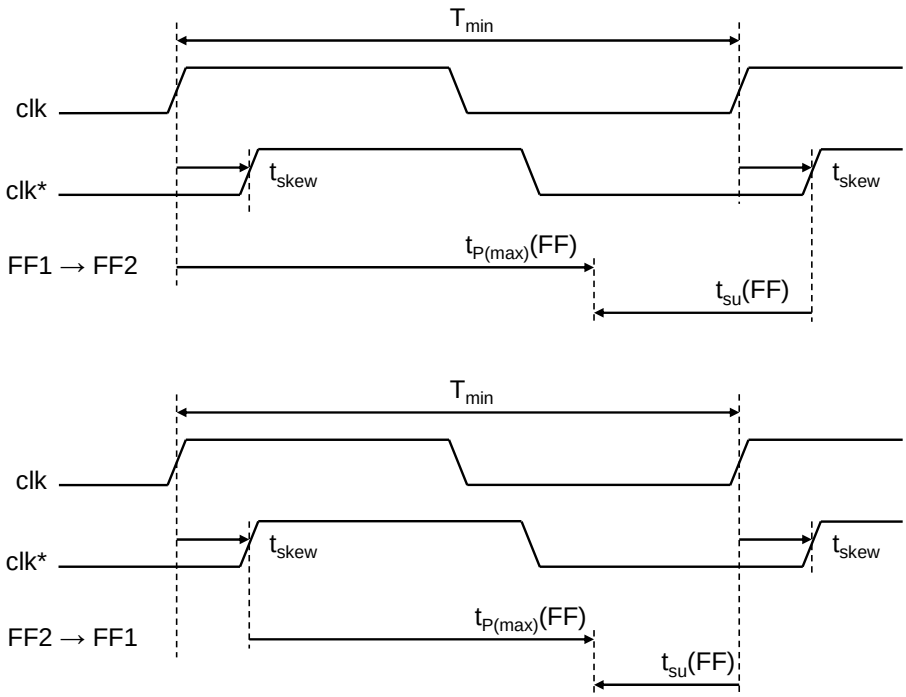
De bijbehorende formules:

$$\begin{aligned} T_{min} &= t_{P(max)}(\text{FF}) + t_{P(max)}(\text{logic}) + t_{su}(\text{FF}) \\ &= 17 + 19 + 14 = 50 \text{ ns} \end{aligned} \tag{A.78}$$

De maximale frequentie bedraagt dan 20 MHz.

Uitwerking opgave 9.3.

Er zijn hier twee paden te ontdekken waarlangs data wordt getransporteerd: FF1 → FF2 en FF2 → FF1. Van beide moet de timing bepaald worden. In figuur A.76 zijn die weergegeven.



Figuur A.76: *Timing van twee rondgekoppelde D-flipflops.*

Beide paden leveren een T_{min}. De grootste waarde van T_{min} levert de minimale periodetijd waarop het gehele systeem nog precies betrouwbaar werkt (als er geen hold violation is).

$$\begin{aligned} \text{FF1} \rightarrow \text{FF2}: &\quad T_{min} + t_{skew} = t_{P(max)}(\text{FF}) + t_{su}(\text{FF}) \\ \text{FF2} \rightarrow \text{FF1}: &\quad T_{min} - t_{skew} = t_{P(max)}(\text{FF}) + t_{su}(\text{FF}) \end{aligned} \tag{A.79}$$

Na herschikking van de term t_{skew} in beide formules wordt duidelijk dat de formule voor het pad FF2 $\rightarrow$ FF1 de grootste T_{min} levert (natuurlijk geldt dat $t_{skew} > 0$).

De logische werking (gedrag) is als volgt te bepalen. De beide flipflops hebben de beginstand $Q_2Q_1 = 00$. Dat betekent dat na de klokflank de inhoud van flipflop 1 logisch 1 wordt, immers wordt de inverse stand van Q_2 ingeklokt. De inhoud van flipflop 2 wordt/-blijft logisch '0'. De inhoud van de flipflops is $Q_2Q_1 = 01$. Na een volgende klokflank is de inhoud van de flipflops gewijzigd in $Q_2Q_1 = 11$. Daarna wordt de inhoud (ook wel stand genoemd) gewijzigd in $Q_2Q_1 = 10$. Hierna volgt dan nog $Q_2Q_1 = 00$. Dat was ook de beginstand van de flipflops. De flipflops doorlopen de cyclus $00 \rightarrow 01 \rightarrow 11 \rightarrow 10$. Dat is de bekende Gray-codering. Dit is dan ook een Gray-teller[3].

Uitwerking opgave 9.4.

Er zijn twee voorwaarden waaraan betrouwbare dataoverdracht moet voldoen:

$$\begin{aligned} \text{setup slack groter dan/gelijk aan 0}: &\quad t_{slack,su} \geq 0 \\ \text{hold slack groter dan/gelijk aan 0}: &\quad t_{slack,h} \geq 0 \end{aligned} \tag{A.80}$$

De setup slack komt ter sprake bij de berekening van de minimale periodetijd. Daar is niets over gezegd (d.w.z. dat de minimale periodetijd prima kan worden berekend). Er is dan ook geen *setup time violation*.

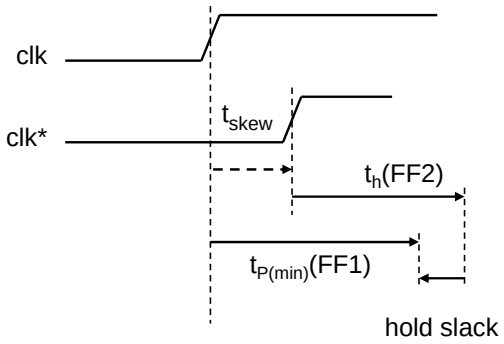
Voor het berekenen van de hold slack moet gekeken worden op welk kloksignaal de ontvangende flipflop zijn data inklokt. In dit geval is dat `clk*`. Deze klok is vertraagd met 2 ns. De holdtijd van de tweede flipflop wordt dus "naar achteren" geduwd. De eerste flipflop, die geen last heeft van skew, haalt zijn data op zijn vroegst na de minimale propagatietijd weg op de actieve flank van `clk`. De tijden kunnen worden getekend in een figuur, zie figuur A.77.

Uit de figuur kan de volgende formule worden opgemaakt:

$$\begin{aligned} t_{slack,h} &= t_{P(min)}(\text{FF}) - t_{skew} - t_h(\text{FF2}) \\ &= 7 - 2 - 6 = -1 \text{ ns} \end{aligned} \tag{A.81}$$

De hold slack is negatief en dat betekent dat er een *hold time violation* is. Er is geen betrouwbare dataoverdracht mogelijk.

[3] De oplettende lezer ziet in deze code ook de telcode van een Johnson counter.



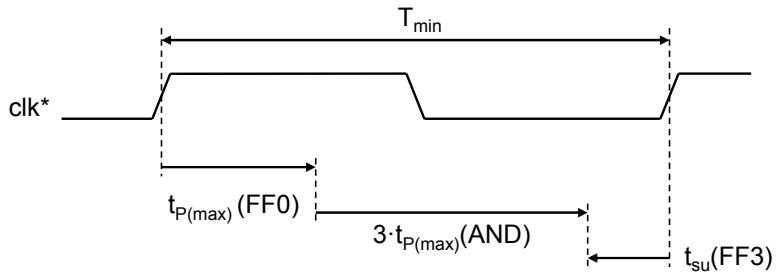
Figuur A.77: *Hold violation bij data-overdracht tussen twee flipflops.*

Uitwerking opgave 9.5.

(Opgave a) In de schakeling zijn de volgende zes paden te ontdekken:

FF0 → 1x AND → FF1
FF0 → 2x AND → FF2
FF0 → 3x AND → FF3
FF1 → 1x AND → FF2
FF1 → 1x AND → FF3
FF2 → 1x AND → FF3

(Opgave b) De flipflops hebben onderling geen last van klokskew, er kan uitgegaan worden van kloksignaal `clk*`. Het langste pad in tijd is onmiskenbaar FF0 → 3x AND → FF3. Hiervan is een timingdiagram te maken, zie figuur A.78.



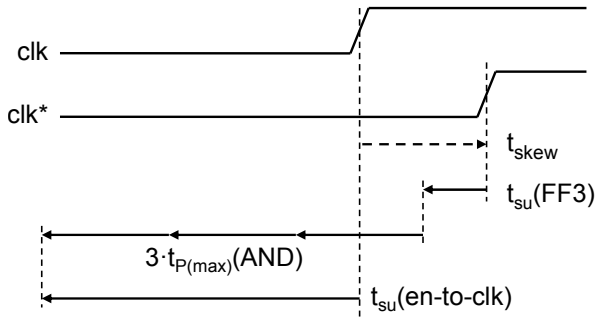
Figuur A.78: *Timingdiagram voor berekening van de minimale periodeduur.*

(Opgave c) De bijbehorende formule voor de berekening van de minimale periodetijd is

$$\begin{aligned} T_{min} &= t_{P(max)}(\text{FF0}) + 3 \cdot t_{P(max)}(\text{AND}) + t_{su}(\text{FF3}) \\ &= 7 + 3 \cdot 5 + 2 \\ &= 24 \text{ ns} \end{aligned} \tag{A.82}$$

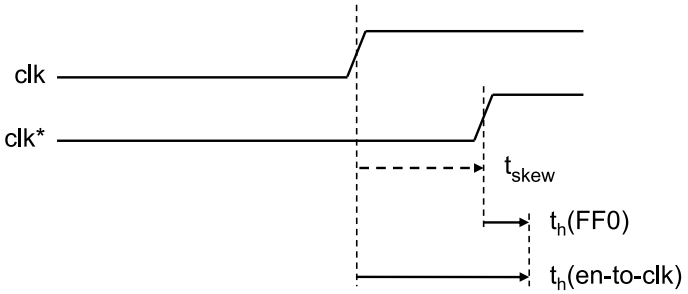
(Opgave d) Om de setuptijd van signaal `enable` t.o.v. van de data-ingang van een willekeurige flipflop te berekenen, moet eerst gekeken worden wat het langste pad (in tijd) is van `enable` tot aan een flipflop. Dat is onmiskenbaar de data-ingang van FF3. Het is logisch dat hier het langste pad (in tijd) gekozen moet worden want een verandering op `enable` komt op z'n laatst bij FF3 aan. Het signaal op de data-ingang moet één setuptijd

voor de klokflank stabiel zijn. Het betreft hier natuurlijk het kloksignaal `clk*`. De AND-poorten leveren na een maximale vertragingstijd zeker een definitieve waarde. T.o.v. van `clk*` moet signaal `en` dus ééns setuptijd en drie vertragingen van de AND-poorten eerder stabiel zijn. Nu wordt wordt echter als referentie het signaal `clk` gebruikt en niet `clk*`. Het enable-signaal moet dus één skewtijd later stabiel zijn. Zie figuur A.79.



Figuur A.79: *Timingdiagram voor berekening van de setuptijd.*

Voor de berekening voor de holdtijd volgt iets vergelijkbaars. Nu moet echter het kortste pad (in tijd) gekozen worden. Dat is natuurlijk de data-ingang van FF0. Dat is te verklaren, want een verandering komt op z'n vroegst bij FF0 aan. De holdtijd van FF0 wordt gemeten vanaf signaal `clk*`. De holdtijd van signaal `en` is pas verstreken na één skewtijd en één holdtijd. Er is geen vertraging van signaal `en` tot aan de data-ingang van FF0, er zit immers geen logica tussen. Zie figuur A.80.



Figuur A.80: *Timingdiagram voor berekening van de holdtijd.*

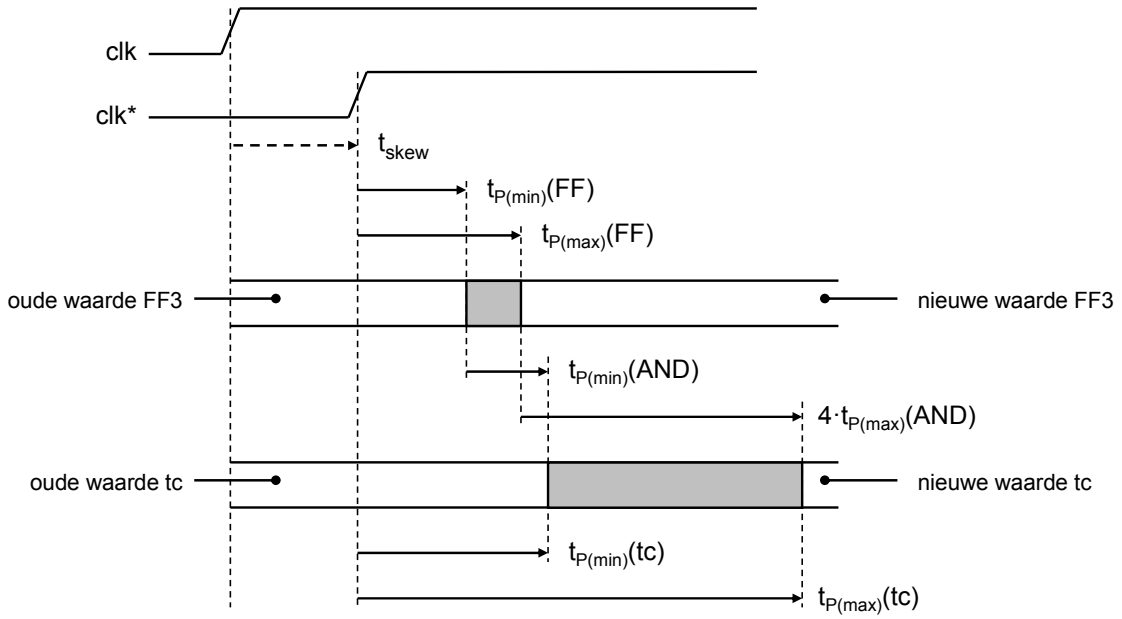
(Opgave e) Uit het bovenstaande kunnen de setuptijd en de holdtijd berekend worden.

$$\begin{aligned} t_{su}(\text{en-to-clk}) &= t_{su}(\text{FF3}) + 3 \cdot t_{P(max)}(\text{AND}) - t_{skew} \\ &= 2 + 3 \cdot 5 - 4 \\ &= 13 \text{ ns} \end{aligned} \tag{A.83}$$

$$\begin{aligned} t_{h}(\text{en-to-clk}) &= t_{skew} + t_{h}(\text{FF0}) \\ &= 4 + 1 \\ &= 5 \text{ ns} \end{aligned} \tag{A.84}$$

(Opgave f) Na de actieve flank van `clk*` moet even gewacht worden tot de uitgangen van flipflops veranderden. Dat is op z'n vroegst na de minimale propagatietijd. Dan is er

nog één poortvertraging nodig om vanaf FF3 bij uitgang `tc` te komen. Voor de maximale propagatietijd van `tc` ligt het anders. Het langste pad van een willekeurige flipflop naar uitgang `tc` begint bij de uitgang van FF0. Dan zijn er nog vier(!) poortvertragingen van de AND-poort nodig om bij uitgang `tc` aan te komen. Vergeet uiteraard de clock skew niet. Zie figuur A.81.



Figuur A.81: *Timingdiagram voor het berekenen van de minimale en maximale vertragingstijd.*

(Opgave g) Het bovenstaande kan in twee formules gezet worden. Daarmee zijn de tijden uit te rekenen.

$$\begin{aligned} t_{P(min)}(\text{clk-to-tc}) &= t_{skew} + t_{P(min)}(\text{FF3}) + t_{P(min)}(\text{AND}) = 4+3+4 = 11 \text{ ns} \\ t_{P(max)}(\text{clk-to-tc}) &= t_{skew} + t_{P(max)}(\text{FF0}) + 4 \cdot t_{P(max)}(\text{AND}) = 4+7+20 = 31 \text{ ns} \end{aligned} \tag{A.85}$$

(Opgave h) Opmerkingen over de teller

De structuur van de teller is bijzonder elegant. Grotere tellers zijn eenvoudig te realiseren. Op het gebied van timing is dit ontwerp geen succes. De maximale frequentie neemt af met het aantal secties. De setuptijd van signaal `enable` en de minimale en maximale vertragingstijd van signaal `tc` neemt toe met het aantal secties.

Uitwerking opgave 9.6.

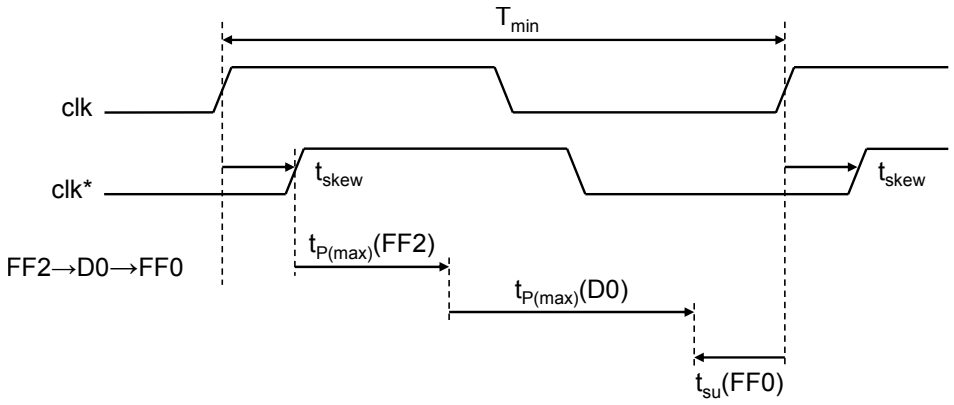
(Opgave a) De volgende paden van flipflop naar flipflop zijn in de schakeling te herkennen:

FF0 → D0 → FF0
FF0 → FF1
FF1 → FF2
FF2 → D0 → FF0

Merk op dat D1 in geen van de paden voorkomt want dat een logica voor een uitgang.

(Opgave b) In figuur A.82 is het timingdiagram getekend. Het langste pad in tijd is van FF2 via D0 naar FF0. De klokingang van FF2 heeft last van skew en die moet meegenomen worden in de berekening van de minimale periodetijd.

Noot: de andere drie paden leveren na uitwerking een kleinere minimale periodetijd dan het hier boven genoemde pad.



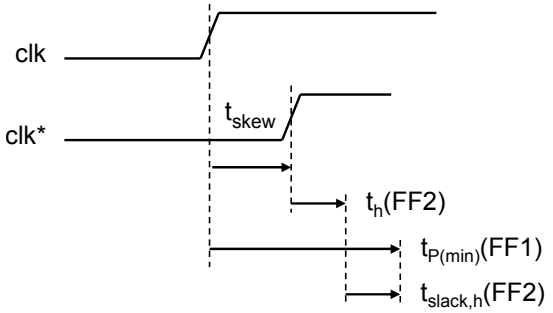
Figuur A.82: *Timingdiagram voor berekening minimale periodetijd.*

(Opgave c) Zie onderstaande berekening.

$$\begin{aligned} T_{min} - t_{skew} &= t_{P(max)}(\text{FF2}) + t_{P(max)}(\text{D0}) + t_{su}(\text{FF0}) \\ T_{min} &= t_{P(max)}(\text{FF2}) + t_{P(max)}(\text{D0}) + t_{su}(\text{FF0}) + t_{skew} \\ &= 8 + (6+5) + 2 + 3 \\ &= 24 \text{ ns} \end{aligned} \tag{A.86}$$

De klokskew heeft negatieve invloed op de minimale periodetijd.

(Opgave d) De *hold slack* is de tijd die over is na het verstrijken van de holdtijd van FF2 en verstrijken van de minimale propagatietijd van FF1. Let er op dat de holdtijd van FF2 naar "achter" wordt geduwd als gevolg van de skew. Zie figuur A.83.



Figuur A.83: *Timingdiagram voor berekening hold slack.*

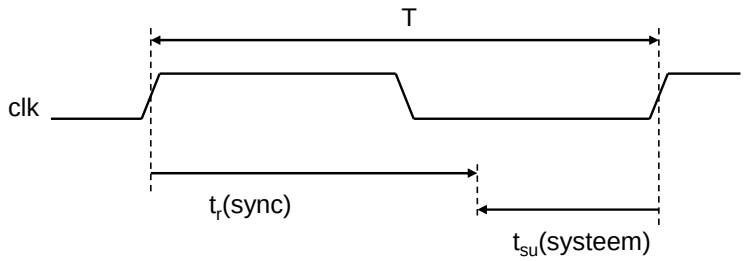
(Opgave e) De hold slack kan als volgt berekend worden:

$$\begin{aligned} t_{skew} + t_h(\text{FF2}) + t_{slack,h}(\text{FF2}) &= t_{P(min)}(\text{FF1}) \\ t_{slack,h}(\text{FF2}) &= t_{P(min)}(\text{FF1}) - t_{skew} - t_h(\text{FF2}) \\ &= 6 - 3 - 1 \\ &= 2 \text{ ns} \end{aligned} \tag{A.87}$$

De hold slack is positief, dat betekent dat de holdtijd eerder is verstreken dan de minimale propagatietijd en dat houdt in dat er betrouwbare data-overdracht mogelijk is (als de maximale frequentie niet overschreden wordt).

A.10 Uitwerkingen hoofdstuk 10

Uitwerking opgave 10.1.
In onderstaande figuur is het timingdiagram getekend.



Figuur A.84: *Timingdiagram van het synchronisatiesysteem.*

Uit deze figuur kan de volgende formule worden opgemaakt.

$$T = t_r(\text{sync}) + t_{su}(\text{systeem})$$

Nu kan de resolutietijd worden bepaald

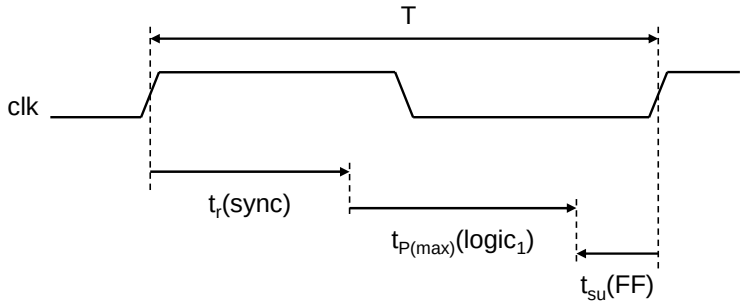
$$\begin{aligned} t_r(\text{sync}) &= T - t_{su}(\text{systeem}) \\ &= 50 - 20 \\ &= 30 \text{ ns} \end{aligned}$$

De synchronizer heeft dus 30 ns om uit z'n metastabiele toestand te raken. Invullen in formule voor berekenen van de MTBF geeft:

$$\begin{aligned} \text{MTBF}(t_r) &= \frac{e^{\frac{t_r}{\tau}}}{T_0 \cdot f_{sys} \cdot f_{data}} = \frac{e^{\frac{30}{0{,}135}}}{9{,}8 \cdot 10^6 \cdot 20 \cdot 10^6 \cdot 100 \cdot 10^3} = \frac{e^{222{,}222}}{19{,}6 \cdot 10^{18}} \\ &= 1{,}651 \cdot 10^{77} \end{aligned}$$

Merk op dat t_r en τ beide in ns gegeven zijn. De gemiddelde tijd tussen twee fouten is $1{,}651 \cdot 10^{77}$ seconden.

Uitwerking opgave 10.2.
In onderstaande figuur is het timingdiagram getekend.



Figuur A.85: *Timingdiagram van het synchronisatiesysteem.*

Uit deze figuur kan de volgende formule worden opgemaakt.

$$T = t_r(\text{sync}) + t_{P(max)}(\text{logic}_1) + t_{su}(\text{FF})$$

Nu kan de resolutietijd worden bepaald

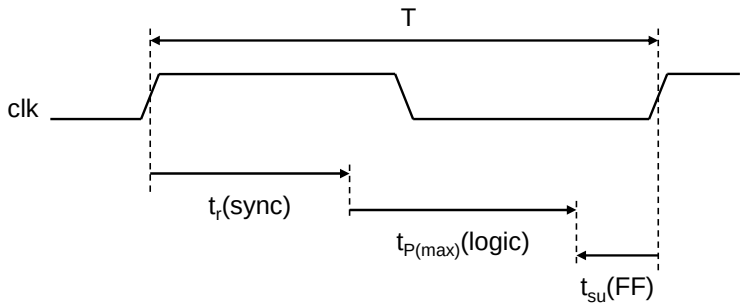
$$\begin{aligned} t_r(\text{sync}) &= T - t_{P(max)}(\text{logic}_1) - t_{su}(\text{FF}) \\ &= 40 - 20 - 5 \\ &= 15 \text{ ns} \end{aligned}$$

De synchronizer heeft dus 15 ns om uit z'n metastabiele toestand te raken. Invullen in formule voor berekenen van de MTBF geeft:

$$\begin{aligned} \text{MTBF}(t_r) &= \frac{e^{\frac{t_r}{\tau}}}{T_0 \cdot f_{sys} \cdot f_{data}} = \frac{e^{\frac{15}{0{,}135}}}{9{,}8 \cdot 10^6 \cdot 25 \cdot 10^6 \cdot 2 \cdot 10^3} = \frac{e^{111{,}111}}{490 \cdot 10^{15}} \\ &= 3{,}671 \cdot 10^{30} \end{aligned}$$

De gemiddelde tijd tussen twee fouten is $3{,}671 \cdot 10^{30}$ seconden. Dat is omgerekend $1{,}164 \cdot 10^{23}$ jaar.

Uitwerking opgave 10.3.
We moeten eerst de maximale resolution time van de synchronizer bepalen. We tekenen een timingdiagram waarin de relatie tussen de diverse tijden duidelijk wordt. Zie figuur A.86.



Figuur A.86: *Timingdiagram van het synchronisatiesysteem.*

Uit deze figuur kan de volgende formule worden opgemaakt.

$$T = t_r(sync) + t_{P(max)}(\text{logic}) + t_{su}(\text{FF})$$

Nu kan de resolutietijd worden bepaald

$$\begin{aligned} t_r(sync) &= T - t_{P(max)}(\text{logic}) - t_{su}(\text{FF}) \\ &= 25 - 10 - 5 \\ &= 10 \text{ ns} \end{aligned}$$

De synchronizer heeft dus 10 ns om uit z'n metastabiele toestand te raken. De MTBF is ook bekend, dat is 1 jaar. Dit moet worden omgerekend naar seconden, dus

$$\begin{aligned} \text{MTBF}(10 \text{ ns}) &= 1 \text{ jaar} = 1 \times 365 \times 24 \times 60 \times 60 = 31536000 \\ &= 3{,}1536 \cdot 10^7 \text{ s} \end{aligned}$$

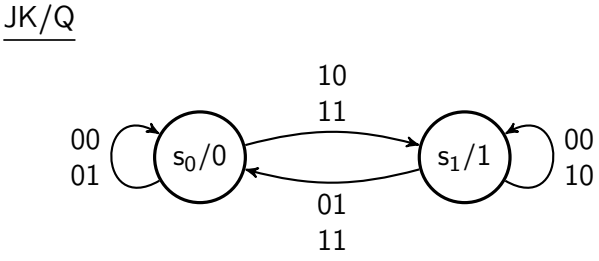
De formule voor het berekenen van de MTBF moet aangepast worden zodat de frequentie van het asynchroon binnenkomend signaal berekend kan worden. Dat kan eenvoudig door MTBF en f_{data} uit te wisselen:

$$\begin{aligned} f_{data} &= \frac{e^{\frac{t_r}{\tau}}}{T_0 \cdot f_{sys} \cdot \text{MTBF}(t_r)} = \frac{e^{\frac{10}{0{,}15}}}{9{,}8 \cdot 10^6 \cdot 40 \cdot 10^6 \cdot 3{,}1536 \cdot 10^7} = \frac{e^{66{,}667}}{1236{,}2112 \cdot 10^{19}} \\ &= 7{,}258933 \cdot 10^6 \end{aligned}$$

De maximale toegestane frequentie van het asynchroon binnenkomend signaal is 7,259 MHz.

A.11 Uitwerkingen hoofdstuk 11

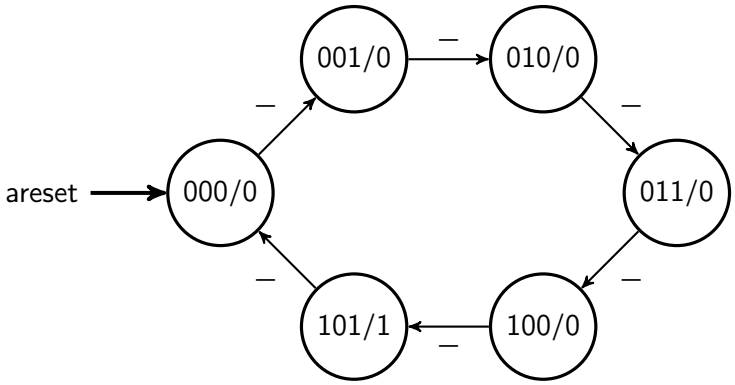
Uitwerking opgave 11.1.
Het toestandsdiagram van de JK-flipflop is te vinden in figuur A.87. De flipflop moet in toestand s_0 blijven als het moet onthouden ($JK = 00$) moet gereset wordt ($JK = 01$). De flipflop gaat naar toestand s_1 als er geset wordt ($JK = 10$) of getoggeld wordt ($JK = 11$). De flipflop moet in s_1 blijven als er onthouden moet worden ($JK = 00$ of geset wordt ($JK = 10$). De flipflop gaat naar toestand s_0 als er gereset wordt ($JK = 01$) of getoggeld wordt ($JK = 11$).



Figuur A.87: *Toestandsdiagram van een JK-flipflop.*

Uitwerking opgave 11.2.

In figuur A.88 is het toestandsdiagram van de 6-teller te zien. De teller telt continue van 0 tot en met 5, er is geen enable aanwezig. Sterker nog, er is niet eens een (stuur-)ingang. We hebben hier gekozen om de codering in de toestanden te plaatsen en niet om symbolische namen te gebruiken. Zo zien we direct de uitgangswaarden van de teller. De uitgang wordt in de hoogste stand (101) een klokcyclus 1. Dit komt overeen met de terminal count van de teller.



Figuur A.88: *Toestandsdiagram van een 6-teller.*

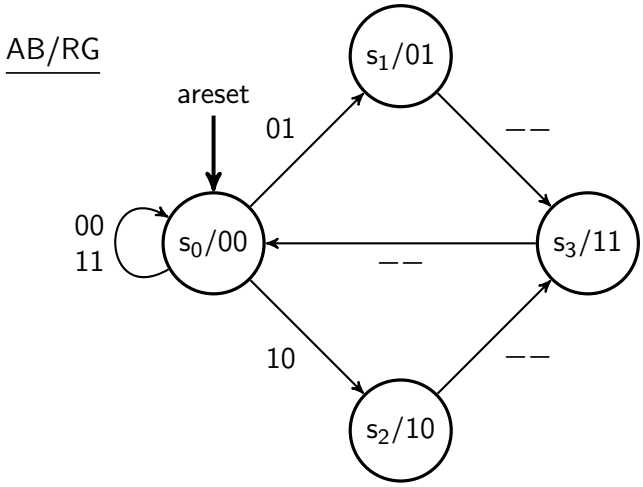
Uitwerking opgave 11.3.

De opgave begint al gelijk met een onduidelijkheid. Er staat dat als de ingang $A = 1$ de machine een bepaalde toestandcyclus moet doorlopen, maar dat staat er ook als $B = 1$. Wat moet er nu gebeuren als zowel $A = 1$ als $B = 1$? Dat staat dus niet vermeld. Als ontwerper moeten we deze onduidelijkheid wegnemen. Meestal volgt een gesprek met de opdrachtgever om het probleem op te lossen. We kiezen er hier voor om de machine in dezelfde toestand te laten als $AB = 11$.

In figuur A.89 is het toestandsdiagram getekend. Alleen in toestand s_0 worden de signalen A en B bekeken. In alle andere toestanden wordt in feite niet naar de ingangen gekeken. Zoals gezegd laten we de machine in toestand s_0 als $AB = 11$. Dat geldt ook voor $AB = 00$, geen van de ingangen is dan immers 1. Alleen als $AB = 01$ of $AB = 10$ worden de andere toestanden doorlopen. Na een opvolgertoestand s_1 of s_2 worden volgt toestand s_2 en gaat de machine weer naar toestand s_0

Uitwerking opgave 11.4.

De toestandtabel is te vinden in tabel A.30. Als de flipflop in s_0 staat blijft de flipflop in deze toestand als er moet worden onthouden ($JK = 00$) of als er gereset wordt ($JK = 01$). Met een set-actie of als er getoggled moet worden, gaat de flipflop naar toestand s_1. Als de flipflop in s_1 staat, blijft de flipflop in deze toestand als er moet worden onthouden ($JK = 00$) of als er geset wordt ($JK = 10$). De flipflop gaat naar toestand s_0 als er gereset moet worden ($JK = 01$) of als er getoggled moet worden $JK = 11$). Merk op dat we hier spreken over een flipflop en niet over een machine. Dat maakt in feite niet uit; elke flipflop is ook een toestandsmachine.



Figuur A.89: *Toestandsdiagram van de RG-machine.*

Tabel A.30: *Toestandstabel van de JK-flipflop.*

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

Uitwerking opgave 11.5.

De teller heeft geen ingangen en gaat bij elke actieve klokflank naar de volgende toestand. Dat maakt het opstellen van een toestandstabel erg makkelijk. Omdat de toestand van de teller ook de uitgangswaarden representeren gebruiken we nu bitcombinaties en niet de bekende symbolische benaming. We hebben echter drie flipflops nodig om de telstand (toestand) weer te geven en dat betekent dat er twee toestanden niet gebruikt worden. We geven de opvolger dan ook aan met don't cares. Zie tabel A.31.

Tabel A.31: *Toestandstabel van de synchrone 6-teller.*

toestand	opvolger
000	001
001	010
010	011
011	100
100	101
101	000
110	–––
111	–––

Uitwerking opgave 11.6.

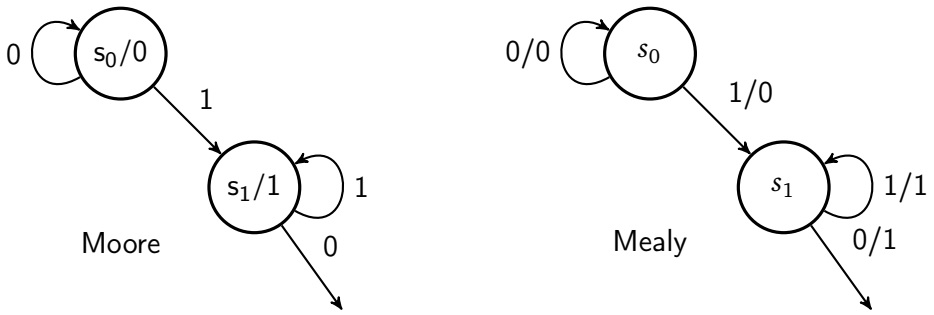
In tabel A.32 is de toestandstabel van de RG-machine te zien. In toestand s_0 wordt gewacht zolang $RG = 00$ of $RG = 11$ is. Als $RG = 01$ is gaat de machine naar toestand

s_1. Bij een ingangscombinatie van $RG = 10$ gaat de machine naar toestand s_2. Vanuit toestand s_1 en s_2 gaat de machine onder elke conditie naar toestand s_3. Daarna gaat de machine altijd naar s_0. Dat betekent dat we alleen bij de regel van toestand s_0 even goed moeten kijken wat de opvolgertoestand is.

Tabel A.32: *Toestandstabel van de RG-machine.*

toestand	opvolger				uitgangen
	00	01	10	11	
s_0	s_0	s_1	s_2	s_0	00
s_1	s_3	s_3	s_3	s_3	01
s_2	s_3	s_3	s_3	s_3	10
s_3	s_0	s_0	s_0	s_0	11

Uitwerking opgave 11.7.
Het is altijd mogelijk om een toestandsdiagram van een Moore-machine als een Mealy-machine te tekenen. Een Moore-machine is namelijk een vereenvoudiging van een Mealy-machine. Bij een toestand in het toestandsdiagram wordt dan bij alle uitgaande pijlen dezelfde uitgangswaarde geschreven. Een voorbeeld is te zien in figuur A.90.



Figuur A.90: *Voorbeeld van Moore-toestanden getekend als Mealy-toestanden.*

Het is niet mogelijk om een Mealy-machine te tekenen als een Moore-machine omdat de uitgangswaarden ook afhankelijk zijn van de ingangswaarden.

Uitwerking opgave 11.8.
Het aantal toestanden is drie en het aantal toestandsbit is twee. We kunnen het aantal unieke codecombinaties uitrekenen:

$$N_D = \frac{2^k!}{(2^k-n)!k!} = \frac{2^2!}{(2^2-3)!2!} = \frac{4!}{1!2!} = \frac{24}{2} = 12 \tag{A.88}$$

Uitwerking opgave 11.9.
Het aantal toestanden is tien. Daaruit volgt dat er minimaal vier toestandsbits nodig zijn om elke toestand uniek te coderen. We kunnen het aantal unieke codecombinaties

uitrekenen:

$$N_D = \frac{2^k!}{(2^k-n)!k!} = \frac{2^4!}{(2^4-10)!4!} = \frac{16!}{6!4!} = 1\,210\,809\,600 \tag{A.89}$$

Uitwerking opgave 11.10.

Eerst moet aan de toestanden een codering worden toegekend. De meest voor de hand liggende codering is $s_0 = 0$ en $s_1 = 1$. Er is dus maar één flipflop nodig. De JK-flipflop heeft twee ingangen zodat de waarheidstabel voor de toestandsfunctie en uitgang acht regels bevat. Zie tabel A.33.

Tabel A.33: *Waarheidstabel met toestandsfunctie en uitgangsfunctie voor de JK-flipflop.*

Q^n	J^n	K^n	Q^{n+1}	D^n	u^n
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	1	0
0	1	1	1	1	0
1	0	0	1	0	1
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	0	1	1

Met behulp van een Karnaughdiagram vinden we voor de JK-flipflop de functies:

$$\begin{aligned} Q^{n+1} &= J^n \cdot \overline{Q^n} + \overline{K^n} \cdot Q^n \\ u^n &= Q^n \end{aligned} \tag{A.90}$$

De uitgang u^n is hetzelfde als de uitgang van de flipflop. Strikt genomen is dit een Medvedev-machine.

Uitwerking opgave 11.11.

In tabel A.34 is de waarheidstabel van de opvolgerfuncties van de 6-teller te zien. We gebruiken hier de binaire telcode omdat de uitgangen van de teller dan direct van de uitgangswaarden van de flipflops kan worden afgetapt. Twee toestanden, te weten $Q_2Q_1Q_0 = 110$ en $Q_2Q_1Q_0 = 111$, worden niet gebruikt en de opvolgers worden als don't care opgegeven.

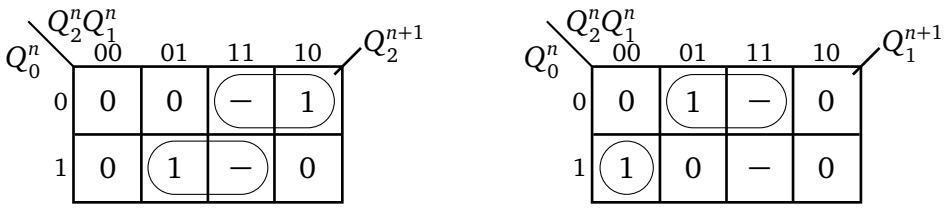
De functies van de flipflops zijn met behulp van Karnaughdiagrammen te vinden, zie figuur A.91. De functie voor Q_0^{n+1} is erg eenvoudig, die laten we achterwege.

Hieronder zijn ze de functies gegeven:

$$\begin{aligned} Q_2^{n+1} &= Q_2^n \cdot \overline{Q_0^n} + Q_1^n \cdot Q_0^n \\ Q_1^{n+1} &= Q_1^n \cdot \overline{Q_0^n} + \overline{Q_2^n} \cdot \overline{Q_1^n} \cdot Q_0^n \\ Q_0^{n+1} &= \overline{Q_0^n} \end{aligned} \tag{A.91}$$

Tabel A.34: *Waarheidstabel met toestandsfuncties voor de 6-teller.*

Q_2^n	Q_1^n	Q_0^n	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	–	–	–
1	1	1	–	–	–



Figuur A.91: *Toestandsfuncties van de 6-teller.*

Uitwerking opgave 11.12.

Voor de toestanden van de RG-machine gebruiken we de binaire telcode: $s_0 = 00$, $s_1 = 01$, $s_2 = 10$ en $s_3 = 11$. We stellen een waarheidstabel op met alle mogelijke combinaties van toestandsbits en ingangswaarden, zie tabel A.35. We noteren de opvolgertoestanden en uitgangswaarden als bitcombinaties. We hebben de kolommen met D_1^n en D_0^n weggelaten omdat we uitgaan van D-flipflops en daarvoor geldt $Q^{n+1} = D^n$.

De functies zijn eenvoudig van aard, deze zijn op te lossen zonder Karnaughdiagrammen te gebruiken:

$$\begin{aligned} Q_1^{n+1} &= \overline{Q_1^n} \cdot Q_0^n + Q_1^n \cdot \overline{Q_0^n} + \overline{Q_1^n} \cdot A^n \cdot B^n \\ Q_0^{n+1} &= \overline{Q_1^n} \cdot Q_0^n + Q_1^n \cdot \overline{Q_0^n} + \overline{Q_1^n} \cdot \overline{A^n} \cdot B^n \\ R^n &= Q_1^n \\ G^n &= Q_0^n \end{aligned} \tag{A.92}$$

Te zien is dat de uitgangswaarden direct van de flipflops kan worden afgetapt.

Uitwerking opgave 11.13.

Voor de toestandscodering gebruiken we de one-hot codering: $s_0 = 0001$, $s_1 = 0010$, $s_2 = 0100$ en $s_3 = 1000$. De toestandsfunctie van een flipflop is op te schrijven door alle inkomende pijlen te bekijken. Elke inkomende pijl vertegenwoordigd één productterm van de toestandsfunctie. Bekijken we toestand s_0, dan zien we twee inkomende pijlen waarbij één pijl twee overgangsvoorwaarden heeft. Hier horen dan twee producttermen bij. Zo blijft de machine in toestand s_0 als R en G beide 0 of beide 1 zijn. Daar horen de producttermen $Q_0^n \cdot \overline{A^n} \cdot \overline{B^n}$ en $Q_0^n \cdot A^n \cdot B^n$. De machine gaat altijd vanuit toestand s_3 naar

Tabel A.35: *Waarheidstabel met toestandsfuncties en uitgangsfuncties voor de RG-machine.*

Q_1^n	Q_0^n	A^n	B^n	Q_1^{n+1}	Q_0^{n+1}	R^n	G^n
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	1	1	0	1
0	1	0	1	1	1	0	1
0	1	1	0	1	1	0	1
0	1	1	1	1	1	0	1
1	0	0	0	1	1	1	0
1	0	0	1	1	1	1	0
1	0	1	0	1	1	1	0
1	0	1	1	1	1	1	0
1	1	0	0	0	0	1	1
1	1	0	1	0	0	1	1
1	1	1	0	0	0	1	1
1	1	1	1	0	0	1	1

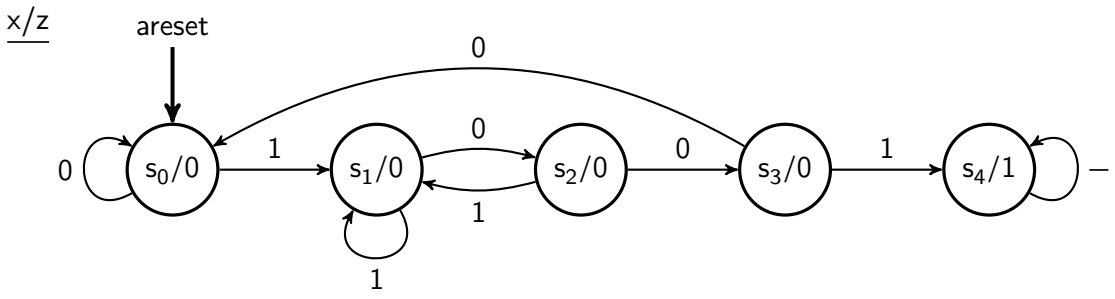
toestand s_0. Daar hoort de *gedegenereerde* productterm Q_n^3 bij. Een gedegenereerde productterm bestaat alleen uit de waarde van de flipflop; er zijn geen overgangsvoorwaarden opgegeven dus alleen de waarde van de flipflop is nodig. Toestand s_1 wordt alleen bereikt als de machine is toestand s_0 staat én $RG = 01$. Daar hoort de productterm $Q_0^n \cdot \overline{A^n} \cdot B^n$ bij. Iets dergelijks geldt ook voor toestand s_2. Daar hoort de productterm $Q_0^n \cdot A^n \cdot \overline{B^n}$ bij. Toestand s_3 wordt (altijd) bereikt als de machine in toestand s_1 of toestand s_2 staat. Daar hoort de term $Q_1^n + Q_2^n$ bij. Dit is een samenstelling (somterm) van de gedegenereerde producttermen Q_1^n en Q_2^n. Alle functies zijn als volgt:

$$\begin{aligned}
Q_0^{n+1} &= Q_0^n \cdot \overline{A^n} \cdot \overline{B^n} + Q_0^n \cdot A^n \cdot B^n + Q_3^n \\
Q_1^{n+1} &= Q_0^n \cdot \overline{A^n} \cdot B^n \\
Q_2^{n+1} &= Q_0^n \cdot A^n \cdot \overline{B^n} \\
Q_3^{n+1} &= Q_1^n + Q_2^n \\
R^n &= Q_2^n + Q_3^n \\
G^n &= Q_1^n + Q_3^n
\end{aligned} \tag{A.93}$$

Uitwerking opgave 11.14.

In figuur A.92 is het toestandsdiagram getekend. Het begint met het opstellen van de vijf toestanden die nodig zijn voor het herkennen van 1001. Daarna volgende de uitzonderingen. Zo blijft de machine in toestand s_0 zolang er een 0 wordt gedetecteerd, bijvoorbeeld bij het patroon 000.. De machine blijft in toestand s_1 als na een 1 nog een 1 (of meerdere) wordt gedetecteerd, bijvoorbeeld bij het patroon 111.. Als na het (deel-)patroon 10 weer een 1 wordt gedetecteerd gaat de machine van toestand s_2 terug naar toestand $_1$ bijvoorbeeld bij het patroon 101.. De machine gaat van toestand s_3 terug naar

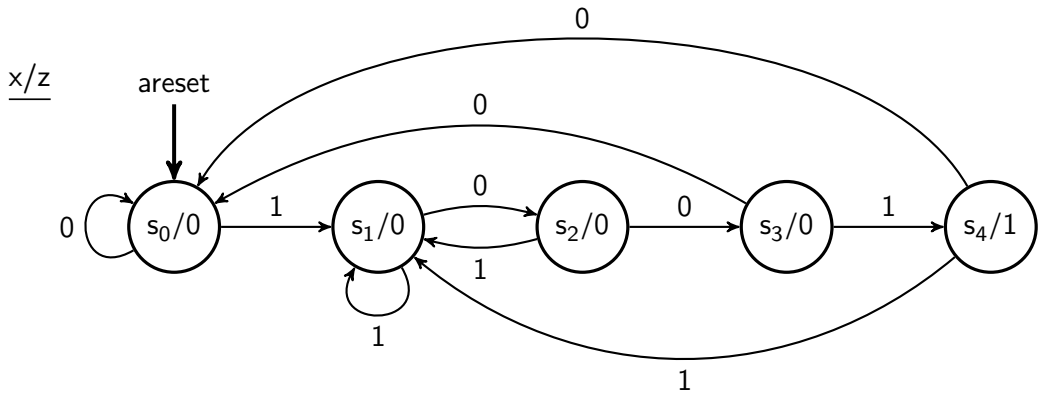
toestand s_0 als na het (deel-)patroon 100 weer een 0 wordt herkend, bijvoorbeeld bij het patroon 1000.. De machine blijft na herkennen in toestand s_4 hangen. De uitgangswaarde blijft dan 1. Eventueel kan de machine uitgebreid worden met een extra toestand waar de machine dan naar toegaat en de uitgangswaarde 0 is zodat er slechts één 1 wordt afgegeven.



x: 00101100110011111000
z: 00000000011111111111

Figuur A.92: *Herkenner voor het eenmalig herkennen van het patroon 1001 (Moore).*

Uitwerking opgave 11.15.
Voor de Moore-variant kunnen we uitgaan van de machine in de vorige opgave. Alleen de overgang bij de laatste toestand moet veranderd worden. Zie figuur A.93.

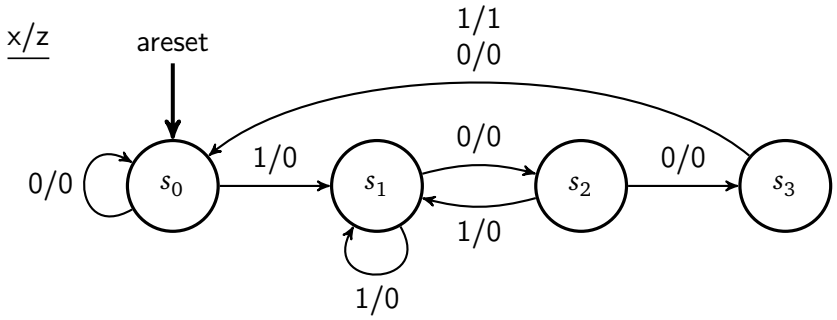


x: 00101100110011111000
z: 00000000010001000000

Figuur A.93: *Herkenner voor het doorlopend herkennen van het patroon 1001 (Moore).*

De Mealy-machine heeft maar vier toestanden. Bij de laatste toestand wordt bekeken of de uitgangswaarde 1 is en dan wordt een 1 afgegeven (bij de overgang uiteraard). Merk op dat de machine onder beide voorwaarden naar toestand s_0 gaat. Zie figuur A.94.

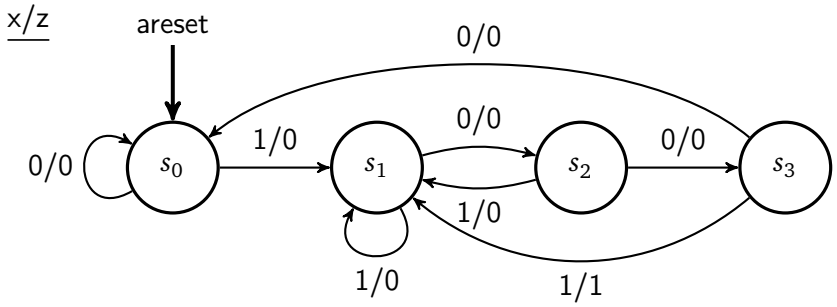
Uitwerking opgave 11.16.
Ook deze machine is weer een variatie op het thema. Het gaat weer om toestand s_3. Slecht één overgang is anders dan de eerder ontwikkelde machine. In toestand s_3 gaat de



x: 00101100110011111000
z: 00000000100010000000

Figuur A.94: *Herkenner voor het doorlopend herkennen van het patroon 1001 (Mealy).*

machine bij een 1 naar toestand s_2. Overlappingen als ..1001001.. worden nu herkend, maar ook doorlopende patronen als ..10011001.. worden hekend.

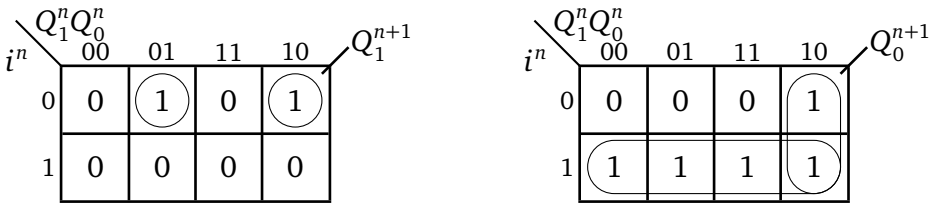


x: 00101100100011111000
z: 00000000100100000000

Figuur A.95: *Herkenner voor het doorlopend overlappend herkennen van het patroon 1001 (Mealy).*

We ontwikkelden de hardware-implementatie met behulp van de binaire telcode als toestandscodering: $s_0 = 00$, $s_1 = 01$, $s_2 = 10$ en $s_3 = 11$. Hiervoor zijn twee flipflops nodig. De machine heeft maar één ingang zodat het aantal ingangen voor de opvolgertoestandslogica en uitgangslogica drie bedraagt. Zie tabel A.36.

In figuur A.96 zijn de functies voor Q_1^{n+1} en Q_0^{n+1} in Karnaughdiagrammen ingevuld. De functie van z^n is erg eenvoudig, dat kan zonder een Karnaughdiagram.



Figuur A.96: *Toestandsfuncties van de patroonherkenner.*

Tabel A.36: *Waarheidstabel met toestandsfuncties en uitgangsfuncties voor de patroonherkenner.*

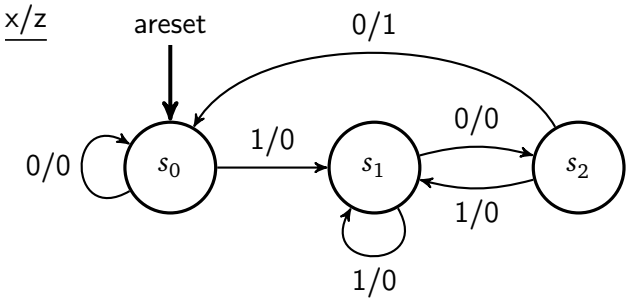
Q_1^n	Q_0^n	i^n	Q_1^{n+1}	Q_0^{n+1}	z^n
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

We vinden voor de toestandsfuncties en uitgangsfunctie het volgende:

$$\begin{aligned} Q_1^{n+1} &= \overline{Q_1^n} \cdot Q_0^n \cdot \overline{i^n} + Q_1^n \cdot \overline{Q_0^n} \cdot \overline{i^n} \\ Q_0^{n+1} &= i^n + Q_1^n \cdot \overline{Q_0^n} \\ z^n &= Q_1^n \cdot Q_0^n \cdot i^n \end{aligned} \tag{A.94}$$

Uitwerking opgave 11.17.

De machine moet een patroon van drie bits kunnen detecteren. Dat houdt in dat de Mealy-machine uit slechts drie toestanden bestaat. Het toestandsdiagram is gegeven in figuur A.97.



Figuur A.97: *Herkenner voor het doorlopend herkennen van het patroon 100 (Mealy).*

Uitwerking opgave 11.18.

Bij een one-hot codering wordt voor elke toestand één flipflop gebruikt die dan ook 1 is als de machine in de betreffende toestand staat. De andere flipflops zijn dan 0. Aldus gebruiken we de codering $s_0 = 0001$, $s_1 = 0010$, $s_2 = 0100$ en $s_3 = 1000$. We zoeken weer naar de pijlen die naar een toestand toewijzen, elke pijl levert één productterm uit

de functie van de flipflop. We vinden:

$$\begin{aligned} Q_0^{n+1} &= Q_0^n \cdot \overline{x^n} + Q_2^n \cdot \overline{x^n} = (Q_0^n + Q_2^n) \cdot \overline{x^n} \\ Q_1^{n+1} &= Q_0^n \cdot x^n + Q_1^n \cdot x^n + Q_3^n \cdot x^n = (Q_0^n + Q_1^n + Q_3^n) \cdot x^n \\ Q_2^{n+1} &= Q^1 \cdot x^n \end{aligned} \tag{A.95}$$

Zoals te zien is, kunnen twee van de drie functies het beste in de POS-vorm genoteerd worden, dat enkele poorten winst op.

Uitwerking opgave 11.19.

De bewering van de studente klopt: toestand s_0 en s_5 zijn *equivalent* en kunnen samengenomen worden tot één toestand. We kunnen dit demonstreren met behulp van een toestandstabel voor de machine, zie tabel A.37.

Tabel A.37: *Toestandstabel van de bloksgewijze herkenner voor het patroon 110.*

toestand	opvolger		uitgang
	0	1	
s_0	s_0	s_1	0
s_1	s_4	s_2	0
s_2	s_3	s_5	0
s_3	s_0	s_1	1
s_4	s_5	s_5	0
s_5	s_0	s_1	0

Als we kijken naar de toestanden s_0 en s_5 dan zien we dat beide dezelfde opvolgertoestanden hebben onder besturing van de ingang én beide dezelfde uitgangswaarde hebben. Dat betekent dat het voor een buitenstaander niet te zien is of de machine in toestand s_0 of toestand s_5 staat; in beide gevallen is de uitgangsreactie hetzelfde. Als de machine in toestand s_0 staat en de ingangswaarde is 0, dan blijft (of gaat) de machine naar toestand s_0. Is de ingangswaarde een 1, dan gaat de machine naar s_1. De uitgangswaarde is 0. Dat gebeurt ook als de machine is s_5 staat. We kunnen toestand s_5 schrappen en een nieuwe toestandstabel opstellen, zie tabel A.38.

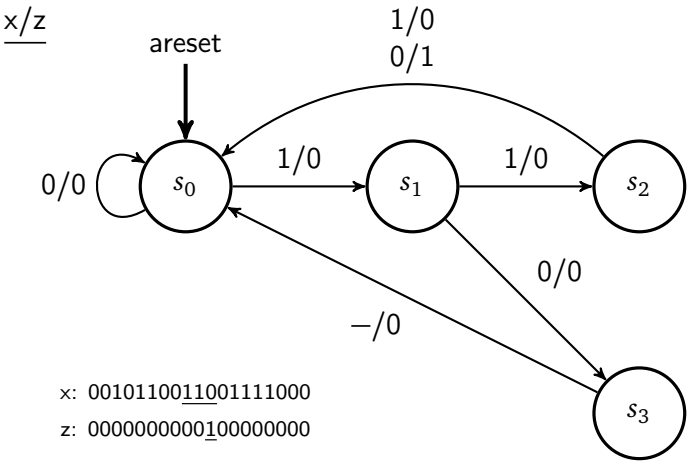
Tabel A.38: *Gereduceerde toestandstabel van de bloksgewijze herkenner voor het patroon 110.*

toestand	opvolger		uitgang
	0	1	
s_0	s_0	s_1	0
s_1	s_4	s_2	0
s_2	s_3	s_0	0
s_3	s_0	s_1	1
s_4	s_0	s_0	0

De oplettende lezer ziet dat ook toestand s_3 dezelfde opvolgers heeft maar hier is de uitgangswaarde anders dat bij toestand s_0/s_5.

Uitwerking opgave 11.20.

Het toestandsdiagram van de Mealy-machine voor het bloksgewijs herkennen van 110, beginnend met een 1 is te zien in figuur A.98. Er zijn slechts vier toestanden nodig. De machine wacht in toestand s_0 zoals de ingangswaarde 0 is. Als de ingangswaarde 1 is, worden drie bits "afgetikt". Als na de eerste 1 en 0 volgt kan het patroon niet meer voorkomen en gaat de machine via toestand s_3 terug naar toestand s_0. Volgt na de eerste 1 nog een 1 dan gaat de machine naar toestand s_2. In deze toestand wordt dan bekeken of vervolgens een 0 of een 1 wordt gedetecteerd. De uitgangswaarde is dan naar gelang de ingangswaarde een 0 of een 1.

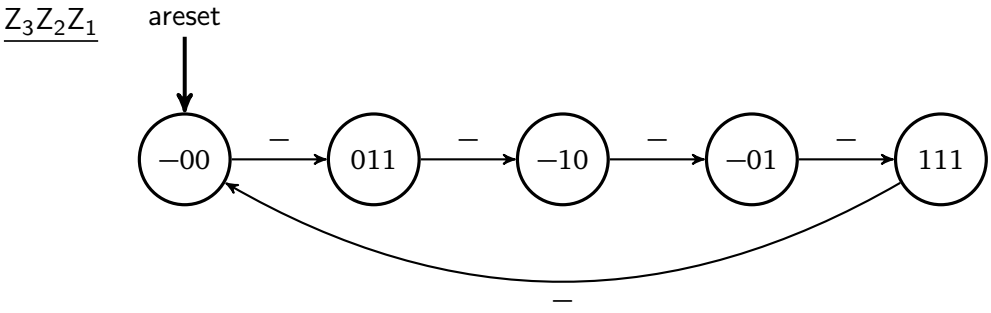


Figuur A.98: *Herkenner voor het bloksgewijs herkennen van het patroon 110 (Mealy).*

Uitwerking opgave 11.21.

We hebben in figuur A.99 het toestandsdiagram getekend met in de toestanden de uitgangswaarden voor Z_3, Z_2 en Z_1. Merk op dat bij drie van de vijf toestanden de waarde van Z_3 niet belangrijk is. We hebben ze op don't care gesteld. De meest linkse toestand (−00) wordt dus door *twee* toestandscoderingen gekenmerkt, te weten 000 en 100. Iets dergelijks geldt ook bij de toestanden −10 en −01. We kunnen van deze don't cares gebruik maken bij het opstellen van de waarheidstabel voor de opvolgertoestand (de uitgangswaarden komen direct van Z_2 en Z_1). Zo kan de opvolger van toestand 011 gespecificeerd worden als −10; het maakt dus niet uit wat de waarde van Z_3 wordt. Merk op dat de don't cares alleen bij specificatie bestaan. Bij uitwerken van de functies worden de don't cares omgezet naar nullen en enen.

In tabel A.39 is de waarheidstabel voor de toestandsfuncties te zien. Merk op dat de uitgangen direct van de flipflops komen; er is dus geen uitgangslogica. In een aantal gevallen kan de waarde van Z_3 als don't care worden gespecificeerd. Zo is de opvolger van toestand 011 de toestand −10; het maakt niet uit wat de waarde van Z_3 is. Dat betekent echter wel dat voor beide codecombinaties (010 en 110) de juiste opvolger moet worden opgegeven. Noot: in tegenstelling tot gebruikelijk gebruiken we hier Z i.p.v. Q als functievariabelen.



Figuur A.99: *Toestandsdiagram van machine op basis van een timingdiagram.*

Tabel A.39: *Waarheidstabel met toestandsfuncties voor machine.*

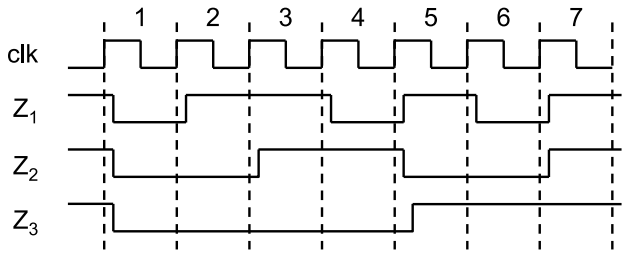
Z_3^n	Z_2^n	Z_1^n	Z_3^{n+1}	Z_2^{n+1}	Z_1^{n+1}
0	0	0	0	1	1
0	0	1	1	1	1
0	1	0	–	0	1
0	1	1	–	1	0
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	–	0	1
1	1	1	–	0	0

De toestandsfuncties zijn eenvoudig van aard:

$$\begin{aligned} Z_3^{n+1} &= Z_0^n \\ Z_2^{n+1} &= \overline{Z_1^n} + \overline{Z_2^n} \cdot Z_0^n \\ Z_1^{n+1} &= \overline{Z_1^n} + \overline{Z_0^n} \end{aligned} \tag{A.96}$$

Uitwerking opgave 11.22.

In het timingsdiagram is te zien dat uitgang Z_1 per cyclus drie keer 0 is en vier keer 1. Er zijn dan minstens twee (extra) flipflops nodig om de een toestandsmachine te ontwerpen die deze cyclus uitvoert. Het volledige timingsdiagram is te vinden in figuur A.100.

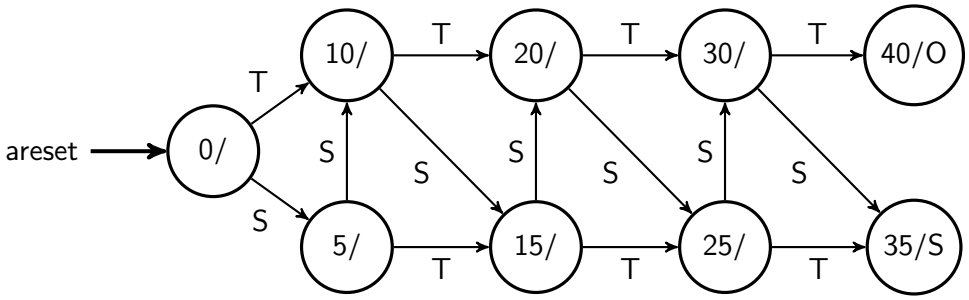


Figuur A.100: *Timingdiagram van een toestandsmachine.*

De machine doorloopt de cyclus: 000-001-011-010-101-100-111. Toestand 100 komt niet voor.

Uitwerking opgave 11.23.

In figuur A.101 is het gestileerde toestandsdiagram te zien van een herkenner voor 35 cent. Geaccepteerde muntstukken zijn 5 cent (S) en 10 cent (T). Er zijn in totaal negen toestanden. Het is mogelijk om het bedrag geheel in 5 cent muntstukken in te werpen, dus elke toestand vertegenwoordigt een veelvoud van 5 cent.



Figuur A.101: *Herkenner voor het herkennen van 35 cent (Moore).*

Uitwerking opgave 11.24.

Het hoogste bedrag dan kan worden ingeworpen is 40 cent. Het kan als volgt beredeneerd worden. Het gewenste bedrag is 35 cent. De kleinste bedrag dat kan worden ingeworpen is 5 cent. Het bedrag dat het dichts bij 35 cent ligt is dus 30 cent, namelijk gewenst bedrag minus kleinste munt. In deze situatie kan nu de grootste munt worden ingeworpen, dus 10 cent. Dat betekent dat er nooit meer dan 40 cent kan worden ingeworpen. In formulevorm:

$$hb = gb - k + g \tag{A.97}$$

Hierin is hb het hoogste bedrag, gb is gewenste bedrag, k is kleinste munt en g is grootste munt. Deze formule geldt alleen als de waarden van de toegestane munten een tweevoud zijn.

Uitwerking opgave 11.25.

De machine heeft negen unieke toestanden. Dat is als volgt uit te rekenen. Het hoogste bedrag is 40 cent. De kleinste munt is 5 cent. Dat betekent dat als er alleen maar muntstukken van 5 cent in worden geworpen de machine in de toestanden 5 cent, 10 cent, 15 cent, ..., 35 cent (40 cent kan op deze manier niet worden gehaald). Het aantal unieke toestanden is dus 40 cent gedeeld door 5 cent plus 1 want 0 cent is ook een toestand. In formulecvorm:

$$t = \frac{hb}{k} + 1 = \frac{gb - k + g}{k} + 1 = \frac{gb + g}{k} \tag{A.98}$$

Hierin is t het aantal unieke toestanden en de overige variabelen hebben dezelfde betekenis als in de vorige opgave. Ook deze formule geldt alleen als de waarde van de muntstukken een tweevoud is.

Uitwerking opgave 11.26.

De munten zijn allemaal een tweevoud van elkaar, dus kunnen de formules uit de vorige opgave gebruikt worden. Het hoogste bedrag is:

$$hb = gb - k + g = 35 - 5 + 20 = 50 \tag{A.99}$$

dus 50 cent. Het aantal unieke toestanden is:

$$t = \frac{gb + g}{k} = \frac{35 + 20}{5} = 11 \tag{A.100}$$

Er zijn dus 11 unieke toestanden.

Uitwerking opgave 11.27.

Zo op het eerste gezicht is er geen vereenvoudiging mogelijk. Geen van de toestanden heeft een identiek opvolger en uitgangswaarde. Zie hiervoor tabel A.40.

Tabel A.40: *Toestandstabel voor machine.*

toestand	opvolger		uitgang
	0	1	
A	B	C	0
B	B	D	0
C	D	C	0
D	D	F	0
E	E	F	0
F	D	E	1

We zien echter het volgende bij toestand D en E. Als de machine in D staat en er wordt een 0 gedetecteerd, dan blijft de machine in D. Bij een 1 gaat de machine naar F. Dat geldt ook voor toestand E. De machine blijft in toestand E als een 0 wordt gedetecteerd. Bij een 1 gaat de machine naar toestand F. In toestand F gaat de machine naar D bij een 0 en naar E bij een 1.

Als de machine nu de volgende toestanden doorloopt: $D-D-D-D-F-D-D$ dan is de uitgangsreactie $0-0-0-0-1-0-0$. Diezelfde uitgangsreactie is ook te zien als de machine de toestanden $E-E-E-E-F-E-E$ doorloopt. Het is voor een toeschouwer niet te achterhalen of de machine nu in toestand D of E staat (als de machine niet in F staat natuurlijk). Ergo, toestanden D en E zijn identiek en kunnen samengenomen worden. Zie tabel A.41.

Uitwerking opgave 11.28.

We stellen een waarheidstabel op uit de functies. Merk trouwens op dat de nieuwe waarden van de Q's logisch 0 zijn als X logisch 0 is. Hieronder de tabel na invullen. Zie tabel A.42.

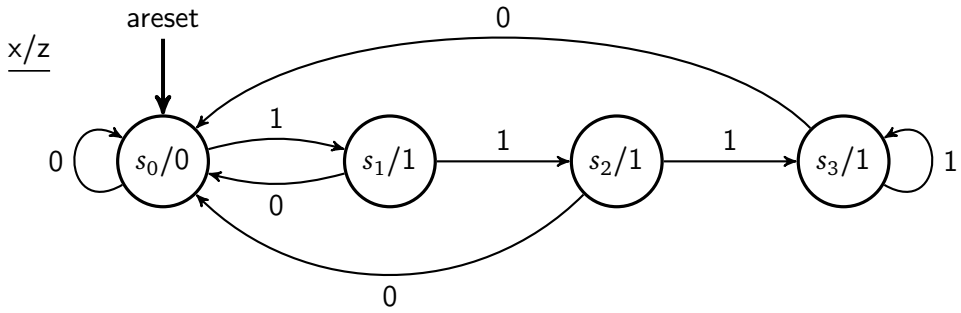
Uit de waarheidstabel is het volgende toestandsdiagram te tekenen, waarbij de volgende toestandscodering wordt gebruikt: $s_0 = 00$, $s_1 = 01$, $s_2 = 10$ en $s_3 = 11$. Zie figuur A.102.

Tabel A.41: *Toestandstabel voor machine.*

toestand	opvolger		uitgang
	0	1	
A	B	C	0
B	B	D/E	0
C	D/E	C	0
D/E	D/E	F	0
F	D/E	D/E	1

Tabel A.42: *Waarheidstabel bij toestandsfuncties van opgave 11.28.*

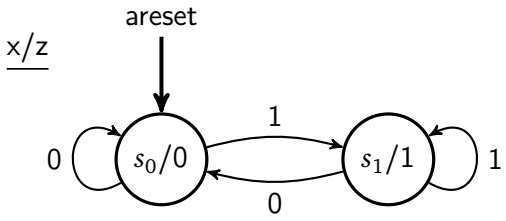
Q_1^n	Q_0^n	X^n	Q_1^{n+1}	Q_0^{n+1}	Z^n
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	0	1
1	0	0	0	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	1	1	1



Figuur A.102: *Toestandsdiagram machine opgave 11.28.*

Het is mogelijk dit toestandsdiagram te vereenvoudigen. Vereenvoudigen houdt in dat twee (of meer) toestanden te combineren zijn (zijn dan *identiek*) en te vervangen zijn door één toestand. Twee toestanden zijn identiek als ze onder dezelfde invoer(reeks) dezelfde uitvoer(reeks) genereren ongeacht welke van de twee toestanden de begintoestand is.

Bekijk toestand s_2 en s_3. Als vanuit s_2 of s_3 een aantal logische 1-en óf een logische 0 wordt verwerkt, is de uitvoer hetzelfde. Toestand s_2 en s_3 zijn dus identiek en kunnen worden gecombineerd. Dat geldt ook voor toestand s_1 en s_2. Dus toestand s_1 en s_2 zijn identiek én toestand s_2 is identiek aan s_3. De toestanden s_1, s_2 en s_3 zijn dus identiek en kunnen worden gecombineerd. In figuur A.103 is het geminimaliseerde toestandsdiagram getekend. Merk op dat dit het toestandsdiagram van een D-flipflop is.

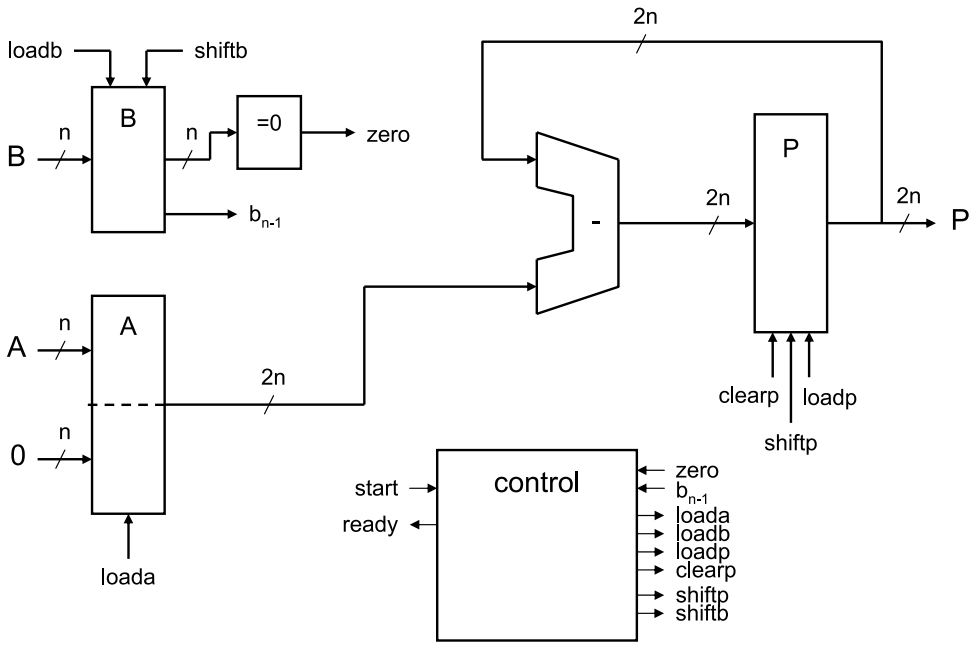


Figuur A.103: *Vereenvoudigd toestandsdiagram machine opgave 11.28.*

A.12 Uitwerkingen hoofdstuk 12

Uitwerking opgave 12.1.

In figuur A.104 is de derde versie van de vermenigvuldiger te zien. We gaan nu P naar links schuiven. We schuiven A niet. P krijg dus een extra stuursignaal $shiftp$ en A heeft slechts alleen nog een load-signaal. Verder is te zien dat de optelacties afhankelijk zijn van het meest significante bit van B. Dat komt omdat we P nu steeds gaan links schuiven. Zo worden na elke optelling ($P = P_A$) de meer significante bits netjes bij P opgeteld.

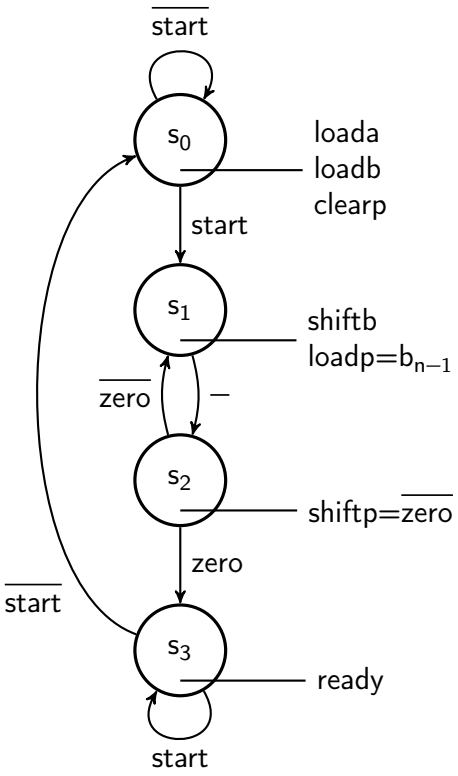


Figuur A.104: *Derde versie van de sequentiële vermenigvuldiger. Hierbij wordt P geschoven.*

Het toestandsdiagram, te zien in figuur A.105 wordt echter uitgebreid met één toestand omdat het laden van P en het schuiven van P niet tegelijkertijd kan plaatsvinden. Verder is te zien dat het optellen van A bij P alleen mag gebeuren als er nog enen in B zijn, anders wordt P één keer teveel geschoven.

Uitwerking opgave 12.2.

P.M.



Figuur A.105: *Het toestandsdiagram van de besturing van de derde versie vermenigvuldiger.*

Uitwerking opgave 12.3.

Het toestandsdiagram moet worden aangepast zodat de inhoud van P bewaard blijft als de machine in de rusttoestand is (hier s_0). We kunnen dit aanpassen door het toevoegen van een extra toestand zoals te zien is in figuur A.106. In toestand s_0 wordt alleen gewacht totdat $start$ geactiveerd wordt. Pas in toestand s_1 worden diverse registers geladen of op gewist.

Dit is niet de enige mogelijkheid. Het is ook mogelijk om het laden van A en B en het wissen van P in toestand s_0 afhankelijk te maken van de waarde van $start$. Als $start$ geactiveerd is worden de registers aangepast. Als $start$ niet geactiveerd is (en de machine blijft dan in s_0) dan worden de inhouden van A, B en P niet aangepast. Dit is het gedrag (van de uitgangen) van een Mealy-machine.

Uitwerking opgave 12.4.

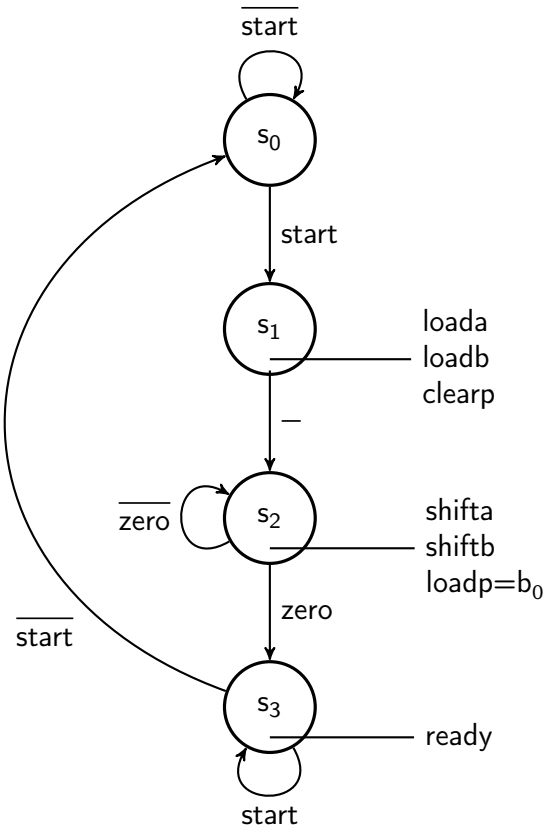
P.M.

Uitwerking opgave 12.5.

We kunnen de vermenigvuldiging van de twee 4-bits getallen A en B als volgt noteren:

$$A \times B = (2^n - 1) \times (2^n - 1) = 2^{2n} - 2 \cdot 2^n + 1 = 2^{2n} - 2^{n+1} + 1 \tag{A.101}$$

Verder geldt dat een $2n$-bits getal de maximale waarde heeft van $2^{2n} - 1$. We kunnen dus



Figuur A.106: *Het toestandsdiagram van de besturing van de twee versie van de sequentiële vermenigvuldiger met extra toestand zodat de inhoud van P bewaard blijft in toestand s_0.*

stellen dat:

$$2^{2n} - 1 > 2^{2n} - 2^{n+1} + 1 \tag{A.102}$$

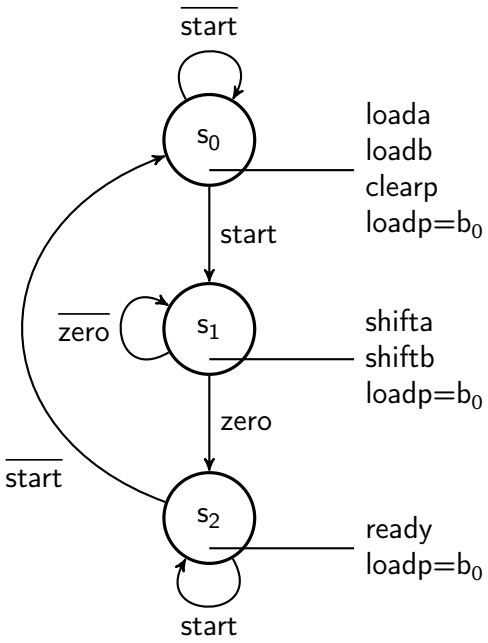
Dit geldt voor elke $n > 0$.

Uitwerking opgave 12.6.

Als we signaal $loadp$ continue verbinden met signaal b_0, dan kunnen we het toestandsdiagram weergeven zoals is te zien in figuur A.107. De vraag is of dit toestandsdiagram een goedwerkende machine oplevert. We kunnen in ieder geval zeggen dat er voor toestand s_1 niets veranderd. We moeten alleen de toestanden s_0 en s_2 bekijken.

Te zien is dat in toestand s_0 zowel $clearp$ als $loadp$ geactiveerd worden. Als we ervoor zorgen dat $clearp$ overheersend is t.o.v. $loadp$ dan werkt de machine in deze toestand correct. In toestand s_2 is de waarde van b_0 altijd 0. Dat komt omdat de machine vanuit s_1 naar s_2 gaat als alle bits van B 0 zijn. Ook in deze toestand werkt de machine correct.

Resumerend kunnen we zeggen dat de machine correct werkt als signaal $clearp$ overheersend is t.o.v. signaal $loadp$.



Figuur A.107: *Het toestandsdiagram van de besturing van de twee versie van de sequentiële vermenigvuldiger met b_0 verbonden aan $loadp$.*

Uitwerking opgave 12.7.

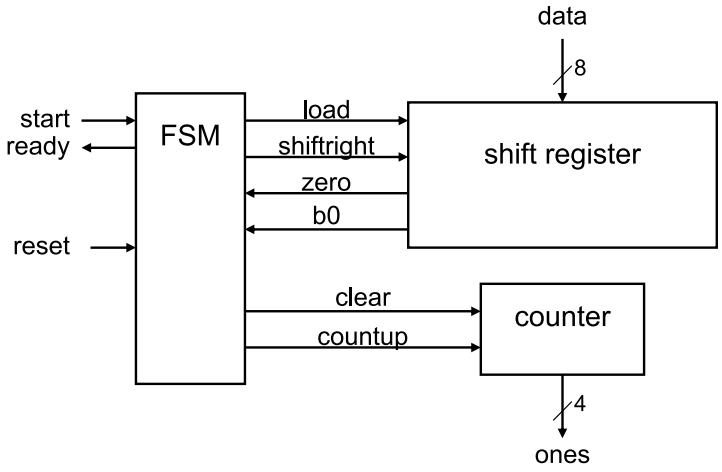
In listing A.33 is de pseudo-code voor de enenteller te zien. Er zijn twee variabelen (of registers) nodig. Register *A* is een teller en register *B* is een schuifregister dat naar rechts kan schuiven.

```
A := 0;
B := input;
while B <> 0 do
    if b0 = 1 then
        A = A+1;
    end if;
    right-shift B;
end while;
```

Listing A.33: *Pseudo-code enenteller.*

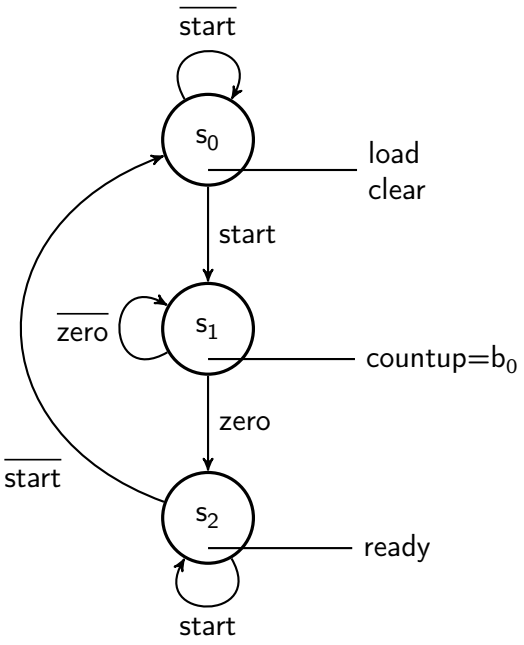
In figuur A.108 is het datapad van de enenteller te zien. *A* is uitgevoerd als een teller die op 0 gezet kan worden middels een $clear$-ingang en met één verhoogd kan worden middels een $countup$-ingang. De telstand wordt via signaal *ones* naar buiten uitgevoerd. Het schuifregister heeft een $load$-ingang om een 8-bits waarde te laden en een $shift right$-ingang om de inhoud één plaats naar rechts te schuiven. Er wordt een 0 ingeschoven (niet getekend). Het schuifregister geeft twee statussignalen terug, te weten het signaal $zero$ dat logisch 1 wordt als alle bits in het schuifregister 0 zijn en een signaal $b0$ dat het minst significante bit uit het schuifregister aan de toestandsmachine (FSM) aanbiedt. De FSM bestuurd het datapad. Het heeft een $start$-ingang waarmee een startopdracht kan worden gegeven. Als de machine klaar is wordt signaal $ready$ geactiveerd.

In figuur A.109 is het toestandsdiagram van de machine te zien. Het begint met toestand



Figuur A.108: *Het datapad voor het bepalen van het aantal enen in een register.*

s_0 waar wordt gewacht totdat $start$ is geactiveerd. In deze toestand wordt het schuifregister geladen en de teller op 0 gezet (clear). In toestand s_1 wordt gewacht zolang het schuifregister niet 0 is. De $countup$-ingang wordt geactiveerd als b_0 logisch 1 is. Dit is dus een Mealy-uitgang. Als het schuifregister 0 is wordt naar toestand s_2 gegaan. In deze toestand wordt de $ready$-uitgang geactiveerd.



Figuur A.109: *Het toestandsdiagram van de besturing van de enenteller.*

A.13 Uitwerkingen hoofdstuk 13

Uitwerking opgave 13.1.

De `nop`-operatie is handig als de ALU "niets" moet doen. Te denken valt aan een sprong-instructie waarbij de PC geladen worden met een nieuw adres. De ALU moet dan in ieder geval de flags niet aanpassen en hoeft ook geen bewerking te doen.

Uitwerking opgave 13.2.

De `pass`-operatie is handig als de inhoud van een register of een constante ongewijzigd in een (ander) register moet worden geladen. De flags worden aangepast.

Uitwerking opgave 13.3.

Er zijn zeker meer logische operaties te verzinnen. Te denken valt aan NOR, NAND en EXNOR. Alleen zijn deze operaties uit te voeren door een OR-, AND- of EXOR-operatie uit te voeren gevolgd door een NOT-operatie. Ze zijn dus niet echt nodig in een processor.

Uitwerking opgave 13.4.

We moeten de opcode en sub-opcode opzoeken via de relevante informatie. De opcode is 0100. De sub-opcode is 0110. We moeten nu de bitpatronen voor de registers vinden. Als doel- en bronregister dient $R3$, dus reg_{ds} is 0011. Als tweede bronregister dient $R2$, dus reg_{s} is 0010. Het volledige bitpatroon van de instructie is dus 0100.0110.0011.0010 (de punten zijn gebruikt om de leesbaarheid te bevorderen).

Uitwerking opgave 13.5.

De instructie **`and`** `R2,R3` zorgt ervoor dat de bit-voor-bit AND van $R2$ en $R3$ wordt geladen in $R2$, d.w.z. dat de statement $R2 \longleftarrow R2 \wedge R3$ wordt uitgevoerd (De $\wedge$ betekent AND). De instructie **`and`** `R3,R2` doet precies hetzelfde alleen zijn nu de doel- en bronregisters omgedraaid maar dat maakt voor de AND-operatie zelf niet uit, alleen wordt nu $R3$ geladen met het resultaat, dus $R3 \longleftarrow R3 \wedge R2$. Hetzelfde geldt voor de OR en de EXOR. Deze bewerkingen bezitten de *commutatieve eigenschap*.

Uitwerking opgave 13.6.

Het programma kan verplaatst worden naar een ander (begin-)adres. Dat komt omdat de sprongadressen van de conditionele branches zijn opgeslagen als een *ofsset*. Deze offset is berekend ten opzichte van het Program Counter. We noemen dat *position independend code*. Let er wel op dat de **`jmp`**-instructie een absoluut adres als gegeven heeft. Na gebruik van deze instructie kan het programma *niet* zomaar verplaatst worden maar moeten de nieuwe (absolute) sprongadressen opnieuw worden uitgerekend.

Uitwerking opgave 13.7.

Nee, de tweede **`add`**-instructie telt twee registers bij elkaar op zonder inkomende carry. De uitgaande carry wordt wel beïnvloed. De tweede **`add`**-instructie mag vervangen worden door een **`adc`**-instructie ook al wordt hierbij de inkomende carry opgeteld. De **`adc`**-instructie die gebruikt wordt voor het bijwerken van het product zorgt er namelijk voor dat de uitgaande carry altijd 0 is. Dat kan eenvoudig worden aangetoond door de

vermenigvuldiging van de maximale waarden van de gebruikte registers te berekenen. Een vermenigvuldiging van twee unsigned 16-bits getallen levert namelijk maximaal

$$(2^{16}-1)\times(2^{16}-1)=2^{32}-2\cdot 2^{16}+1 \tag{A.103}$$

en dat getal is kleiner dan $2^{32}-1$, het maximum van een 32-bits unsigned getal.

Uitwerking opgave 13.8.
De NOT-operatie kan ook worden uitgevoerd door een EXOR-operatie met allemaal enen. Dus:

$$\overline{R_x}=R_x\oplus 1111111111111111 \tag{A.104}$$

De enen kunnen via een constante aan de ALU worden doorgegeven. De **not**-instructie is dus in wezen overbodig.

Uitwerking opgave 13.9.
Ook als de processor geen **ldil**- en **ldih**-instructies heeft, kan een 16-bits constante worden geladen. Daarvoor is wel een hulpregister nodig en het kost enkele instructies. De opzet is te zien in de onderstaande listing

```
1   ; load register R0 with constant $12ab
2   ldi   r0,$ab       ; R0 is $00AB
3   ldi   r8,$12       ; temp register, R8 is $0012
4   swap  r8,r8        ; swap high and low byte, R8 is $1200
5   or    r0,r8        ; R0 is OR of $1200 and $00AB = $12AB
```

Listing A.34: *Laden van een 16-bits constante.*

Het voordeel is natuurlijk dat er geen extra instructies nodig zijn zodat de processor eenvoudiger wordt (lees: minder hardware). Het nadeel is dat er nu vier instructies nodig zijn om een 16-bits constante te laden. Daarbij is het zo dat de *load*-instructies de vlaggen niet beïnvloeden. In de bovenstaande opzet gebeurt dat wel, omdat er een OR-operatie wordt uitgevoerd.

Uitwerking opgave 13.10.
De uitwerking is te zien in listing A.35. Eerst wordt $R1$ vergeleken met 5. Als dat niet het geval is wordt gesprongen naar adres 4 en wordt $R2$ geladen met 10. Als $R1$ gelijk is aan 5, wordt er niet gesprongen en wordt de volgende instructie uitgevoerd, het laden van $R2$ met 8. Daarna wordt gesprongen naar adres 5 zodat de **ld**-instructie op adres 4 niet wordt uitgevoerd.

Uitwerking opgave 13.11.
De uitwerking van de herhaling in assembler is hieronder te zien. We beginnen met het laden van de ingang (`input0`) in register $R0$. Daarna vermenigvuldigen we $R0$ met twee door eenmaal naar links te schuiven. Daarna laden we de constante 128 in register $R1$ en vergelijken het register met $R0$. Merk op dat hier niet mogelijk is om een register

```
1 if:       ldi   r8,5          ; load constant 5
2           cmp   r1,r8         ; compare r1 with 5
3           bne   else          ; not equal, goto else
4           ldi   r2,8          ; load if true
5           bra   endif         ; goto end of if
6 else:     ldi   r2,10         ; load if false
7 endif:    ...
```

Listing A.35: *Assembler-code voor een beslissing.*

direct met een constante te vergelijken. Als $R0$ kleiner is dan of gelijk is aan 128, voeren we de herhaling nog een keer uit.

```
1         in      r0,$00      ; read input
2 do:     lsl     r0,r0       ; R0 := R0 * 2
3         ldi     r1,128      ; load constant 128
4         cmp     r0,r1       ; compare R0 with 128
5 while:  bls     do          ; if lower or same, do again
```

Listing A.36: *Uitwerking herhaling.*

Uitwerking opgave 13.12.
We kunnen het aantal herhalingen van de wachtlus uitrekenen met de formule:

$$\text{aantal keer lus doorlopen} = \frac{f_{proc} \cdot \text{wachttijd}}{\text{klokpulsen per lusdoorloop}} \tag{A.105}$$

Invullen en uitrekenen levert 11111111,11... op (dit is een decimaal getal). De deling gaat niet geheel op. Dat betekent dat een exacte wachttijd niet gerealiseerd kan worden. We zullen moeten afronden op het dichts bijzijnde gehele getal. Hexadecimaal is dat $00A98AC7_{16}$. Hieruit volgt dat $R1$ geladen moet worden met $00A9_{16}$ en $R0$ met $8AC7_{16}$.

Uitwerking opgave 13.13.
Invullen en uitwerken levert 5500000 op, hexadecimaal $0053EC60_{16}$. Dus $R1$ moet geladen worden met 0053_{16} en $R0$ moet geladen worden met $EC60_{16}$.

Uitwerking opgave 13.14.
De wachtlus kan ook worden gerealiseerd met het testen van de carry flag, maar dit kost één extra lusdoorloop. Hiervoor bekijken we wat er met de inhouden van de registers gebeurt op het moment dat daar 00000000_{16} (in twee registers) in staat. Eerst wordt register $R0$ verlaagd. De inhoud wordt dan $FFFF_{16}$ en de carry flag wordt op 1 gezet. Register $R0$ heeft immers een roll-over gehad. Daarna wordt $R1$ verlaagd want de carry flag is 1. Ook dit register heeft een roll-over en wordt $FFFF_{16}$. De carry flag wordt dan op 1 gezet. Ten opzichte van testen met de zero flag wordt dus één extra lusdoorloop uitgevoerd. We kunnen op dezelfde manier het aantal lusdoorlopen uitrekenen maar moeten er dan wel één aftrekken. De **bne**-instructie wordt vervangen door de **bcc**-instructie.

Uitwerking opgave 13.15.

De assemblerprogramma van de deling is te zien in de onderstaande listing. Merk op dat de deling alleen werkt voor unsigned getallen en het deeltal (waardoor gedeeld wordt) moet ongelijk aan 0 zijn. Daar testen we niet op.

```
1         ldi     r0,165          ; load dividend
2         ldi     r1,13           ; load divisor
3         ldi     r2,0            ; clear quotient
4         ldi     r3,1            ; constant 1
5
6 while:  cmp     r0,r1           ; is dividend >= divisor
7         blo     ewhile          ; no so stop
8         sub     r0,r1           ; subtract divisor from dividend
9         add     r2,r3           ; quotient one up
10        bra     while           ; and again
11
12 ewhile: bra    ewhile
```

Listing A.37: *Assemblerprogramma voor delen.*

Uitwerking opgave 13.16.

Na het uitvoeren van deze vijf instructies zijn de waarden (of inhouden) van registers $R1$ en $R2$ verwisseld. Dit wordt in de literatuur de *exor swap* genoemd. Zie ook https://en.wikipedia.org/wiki/XOR_swap_algorithm. Het voordeel van deze manier van verwisselen van de waarden is dat er geen tijdelijke variabele (of register) nodig is. In listing A.38 zijn de waarden van $R1$ en $R2$ als commentaar aan de code toegevoegd.

```
1     ldi   R1,13     ; R1 = 0000.1101
2     ldi   R2,45     ; R2 = 0010.1101
3     exor  R1,R2     ; R1 = 0010.0000
4     exor  R2,R1     ; R2 = 0000.1101
5     exor  R1,R2     ; R1 = 0010.1101
```

Listing A.38: *Code exor swap.*

Uitwerking opgave 13.17.

Het programma is te zien in de onderstaande listing. Merk op dat we het programma beginnen met een **nop**-instructie. Dat is nodig omdat na een reset `input0` nog niet ingelezen is: `input0` is immers een I/O-register en dat wordt bij een reset op 0 gezet. Pas op de eerstvolgende klokflank wordt `input0` voor het eerst geladen met data van buitenaf.

Daarna laden we `input0` eenmaal in register $R2$ in. Vervolgens laden we enkele constanten. In regel 9 schuiven we register $R2$ een plaats naar rechts. De minst significante bit komt in de carry flag. In regel 10 tellen we de carry op bij het resultaat. Merk op dat $R8$ 0 is. Daarna trekken we 1 af van de teller ($R1$) en doorlopen we de lus opnieuw als de teller ongelijk aan 0 is.

```
1              nop     0           ; waste one cycle
2
3              in      r2,0        ; read input0
4              ldi     r3,0        ; the result
5              ldi     r1,16       ; number of bits
6              ldi     r8,0        ; constant 0
7              ldi     r9,1        ; constant 1
8
9  do:         lsr     r2,r2       ; shift low bit in carry
10             adc     r3,r8       ; add only the carry
11             sub     r1,r9       ; decrement counter
12             bne     do          ; do again
13
14 halt:       bra     halt
```

Listing A.39: *Assemblerprogramma voor het tellen van enen.*

Uitwerking opgave 13.18.

Het aantal klokpulsen voor een lusdoorgang bedraagt vier.

Uitwerking opgave 13.19.

De processor kan geen constante van een register aftrekken. Het datapad is daar wel geschikt voor. We kunnen hiervoor dus een instructie ontwerpen. Helaas is het datapad 16 bits breed en dat is ook de breedte van de instructies. Aangezien we ook nog een opcode en een register moeten opnemen, is het aantal bits voor een constante beperkt. Stel dat we de constante beperken tot 8 bits. Dan blijven er nog 8 bits over voor de opcode en het register. We zouden de instructie als volgt kunnen samenstellen:

Figuur A.110: *De nieuwe aftrekinstructie (zonder carry).*

We moeten nog beslissen of we de 8-bits constante als signed of unsigned beschouwen. Een signed constante moet via tekenuitbreiding geschikt gemaakt worden voor 16 bits. Bij een unsigned constante moeten de overige bits met nullen worden gevuld.

Deze nieuwe instructie trekt af zonder de inkomende carry mee te nemen. We hebben dus nog een aftrekinstructie nodig die dat wel doet. Dat kost echter nog een opcode en we zijn er al aardig doorheen. Als we kijken waar de constanten voor gebruikt worden, dan zien we dat het vaak de getallen 0 en 1 betreft om een register op 0 te zetten of om een register te verhogen of te verlagen. We kunnen er ook voor kiezen om de constante met slechts vier bits op te slaan. Er zijn dan 8 bits over voor de opcode en sub-opcode. We hebben dan wel 16 sub-opcodes zodat er veel verschillende instructies te realiseren zijn. Denk hierbij aan met of zonder carry aftrekken. Ook hier kunnen we getallen als signed of unsigned beschouwen (of gewoon nieuwe instructies bedenken).

15 12	15 12	7 4	3 0
0111	0000	reg_d	Imme. 4 bit

Figuur A.111: *Alternatief voor de nieuwe aftrekinstructie (zonder carry).*

Uitwerking opgave 13.20.

Dit hangt helemaal af hoe we de instructies vormgeven. Als we de constanten als signed beschouwen dan kunnen we optellen realiseren door een negatief getal te gebruiken. Zijn de constanten unsigned dan kan het niet en moeten we extra instructies ontwerpen. Er zijn dan vier opcodes nodig en dat redden we nog net. Kijken we waarvoor de constanten veel gebruikt worden dan kunnen we beter uitgaan van 4-bits constanten. Dat beperkt het gebruik van opcodes.

Uitwerking opgave 13.21.

Het vermenigvuldigen kan makkelijk in de I/O gerealiseerd worden. Daarvoor hebben we twee I/O-adressen nodig: een voor de *vermenigvuldiger* en een voor het *vermenigvuldigtal* (beide worden ook wel *factoren* genoemd). Eerst schrijven we de vermenigvuldiger naar het eerste I/O-adres. Er gebeurt dan verder nog niets. Daarna schrijven we het vermenigvuldigtal naar het tweede I/O-adres. Dat is het teken dat de hardware de vermenigvuldiging moet uitvoeren. Het minst significante word van het product komt dan in het eerste I/O-adres en het meest-significante word komt in het tweede I/O-adres. Daarna is het product uit te lezen en in registers te plaatsen.

Voor de twee I/O-adressen nemen we de adressen 16 (10_{16})en 17 (11_{16}). Dat doen we zodat er nog uitbreidingen mogelijk zijn voor extra invoer en uitvoer. Een vermenigvuldiging ziet er in software als volgt uit:

```
1    out     $10,r0      ; write multiplicant
2    out     $11,r1      ; write multiplier and start multiplication
3
4    in      r0,$10      ; read low word product
5    in      r1,$11      ; read high word product
```

Listing A.40: *Vermenigvuldigen via de I/O.*

De implementatie is te zien in listing A.41. We hebben ervoor gekozen om de hele I/O te beschrijven, zodat de relatie met de invoer en uitvoer goed te zien is. We hebben een variabele opgenomen die 32 bits breed is (regel 4). We hebben hier gebruik gemaakt van de *attribute* `length` om de lengte van het datapad op te vragen. Mocht die ooit veranderen dan schaalt de variabele mee. De eigenlijke vermenigvuldiging is te zien in regels 11 t/m 16. Zodra I/O-adres 17 wordt geschreven, start de vermenigvuldiging (regel 13). Daarna wordt het resultaat gesplitst in twee delen en teruggeschreven in adressen 16 en 17. Merk op dat het uitvoeren van de vermenigvuldiging met vier instructies te realiseren is.

```
-- The I/O consists of 64 addresses of 16 bits per address
io_proc: process (clk, areset_int, ioaddress, io) is
-- The product has twice the number of bits
variable mul : unsigned(2*data_type'length-1 downto 0);
begin
    if areset_int = '1' then
        io <= (others => (others => '0'));
    elsif rising_edge(clk) then
        if enio = '1' then
            io(to_integer(ioaddress)) <= datain;
            if to_integer(ioaddress) = 17 then
                -- Start multiplication
                mul := io(16) * datain;
                io(17) <= mul(mul'length-1 downto mul'length/2);
                io(16) <= mul(mul'length/2-1 downto 0);
            end if;
        end if;
        -- io(0-3) will always be the inputs, whatever we write to it
        io(0) <= input0;
        io(1) <= input1;
        io(2) <= input2;
        io(3) <= input3;
    end if;
    datafromio <= io(to_integer(ioaddress));
end process;
```

Listing A.41: *Hardware-implementatie van de vermenigvuldig.*

Uitwerking opgave 13.22.

Het optellen van twee 128-bits getallen kan met vier optelinstructies. Dit is te zien in de onderstaande listing. We beginnen met het optellen van de minst significante words en werken toe naar het optellen van de meest significante words. Op deze manier kan elke breedte van een veelvoud van 16 bits worden opgeteld.

```
add  r0,r4
adc  r1,r5
adc  r2,r6
adc  r3,r7
```

Listing A.42: *Optelling van twee 128-bits getallen.*

Uitwerking opgave 13.23.

We moeten het 32-bits getal zien als twee 16-bits getallen. Eerst schuiven we de 16 minst significante bits één plaats naar links. Dat kan met de **`lsl`**-instructie. De meest significante bit hiervan komt in de carry flag. Daarna gebruiken we een roteerinstructie die de carry flag naar binnen schuift en de meest significante bit in de carry flag plaatst. Het assembler programma is te zien in de onderstaande listing.

```
1    lsl   r0,r0
2    rol   r1,r1
```

Listing A.43: *Naar links schuiven van een unsigned 32-bits getal.*

Uitwerking opgave 13.24.

We moeten het 32-bits getal zien als twee 16-bits getallen. Eerst schuiven we de 16 meest significante bits één plaats naar rechts met behoud van de tekenbit. Dat kan met de `asr` -instructie. De minst significante bit hiervan komt in de carry flag. Daarna gebruiken we een roteerinstructie die de carry flag naar binnen schuift en de minst significante bit in de carry flag plaatst. Het assembler programma is te zien in de onderstaande listing.

```
1    asr   r1,r1
2    ror   r0,r0
```

Listing A.44: *Naar rechts schuiven van een signed 32-bits getal.*

Uitwerking opgave 13.25.

Veel moderne processoren werken direct met 32-bits eenheden. Het is dan ook niet verwonderlijk dat we hier naar kijken. De registers moeten natuurlijk naar 32 bits worden uitgebreid. Dat geldt ook voor het datapad. Alle ALU-operaties kunnen worden uitgebreid naar 32 bits. Merk op dat bij rekenkundige operaties de bepaling van de carry en overflow flags moeten worden aangepast. Het principe blijft echter gelijk, maar nu worden andere bits gebruikt. Voor de logische operaties verandert er niets. De enige operatie die wel verandert is de **`swap`**-operatie. Die werkt nu op 32 bits dus bits 0 t/m 15 worden verwisseld met bits 16 t/m 31.

De RAM en de I/O kan zonder veel problemen worden aangepast. Dat komt omdat de instructies werken op een geheel adres, dus dat blijft zo.

De user ROM kan het best worden uitgebreid naar een 32-bits breedte. Dat heeft te maken met het feit dat uit de user ROM constanten kunnen bevatten en die zijn dan in één instructie in te laden. Het voordeel is dat er veel meer instructies kunnen worden gerealiseerd. Zo zouden we de rekenkundige en logische instructies kunnen uitbreiden met één doelregister en twee bronregisters. Hiervoor zijn 12 bits nodig. Er blijven dan 20 bits over voor de opcode. Het is echter niet mogelijk om een 32-bits constante te verwerken.

De Program Counter zou kunnen worden uitgebreid zodat er meer user ROM geïmplementeerd kan worden. Als we uitgaan van een 4-bits opcode kan de PC uitgebreid worden tot 28 bits. Dan kan 256 GW (giga-words) worden geadresseerd. De vraag is of dat nodig is. Met vier bits voor de opcode en vier bits voor de conditie-keuze houden we 24 bits over voor de branch-offset.

Er is ook een keerzijde aan de uitbreiding naar 32 bits. Zo zal de kloksnelheid flink omlaag gaan. Met name de rekenkundige operaties kosten nu meer tijd. Een en ander heeft te maken met de implementatie van hardware-optellers (ALU en PC).

Bibliografie

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

[1] S. Deering en R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. Engels. Jul 2017. URL: https://www.rfc-editor.org/info/rfc8200 (bezocht op 18-04-2018) (blz. 5).