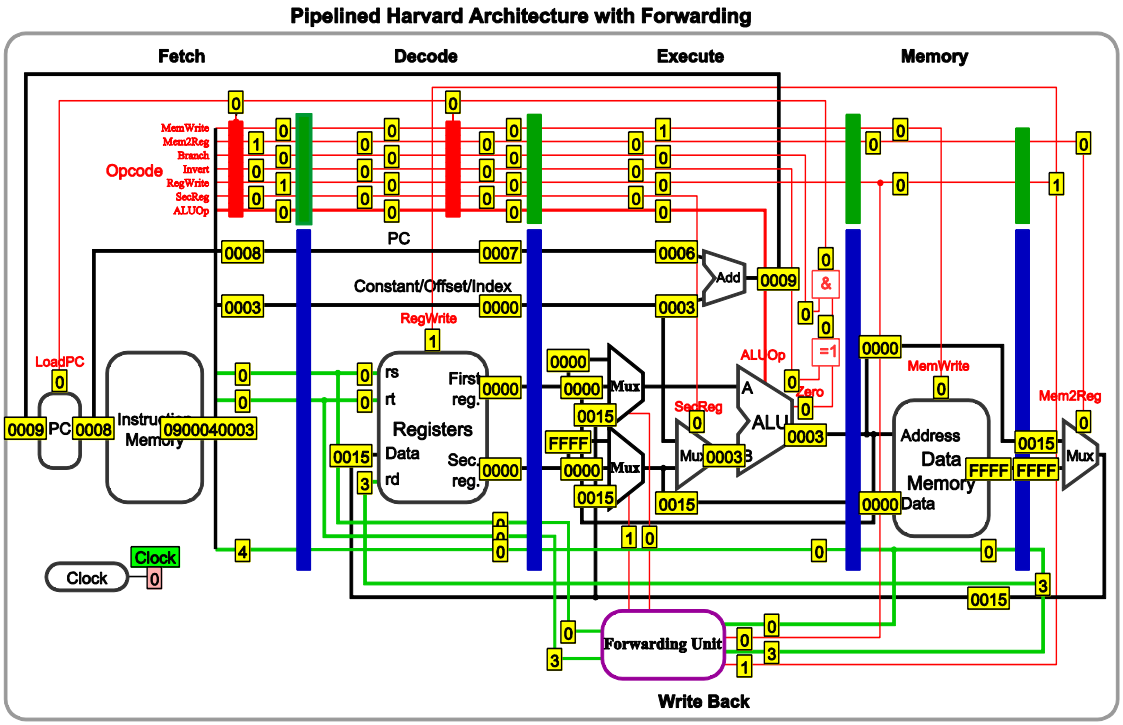
Van 0 en 1 tot pipelined processor

Met simulaties van schakelingen in SIM-PL



Syllabus Computersystemen

Uitgave 2015

Auteur: Ben Bruidegom
Bewerkt door: Bas Terwijn

Van 0 en 1 tot pipelined processor
Met simulaties van schakelingen in SIM-PL

Auteur: Ben Bruidegom bewerkt door Bas Terwijn

Uitgave: juli 2015

Grafische vormgeving & productie: Ben Bruidegom

Voorwoord

In de huidige curricula voor opleidingen in de Informatica, Technische informatica, E-technology en verwante opleidingen is inhoud en omvang van vakken als Embedded Systems, Digitale techniek en Architectuur & Computerorganisatie een voortdurend punt van discussie. Toch is het evident dat kennis van de principes waarop de werking van de hardware en de hardware/software-interface van computersystemen berust een bredere kijk geeft op de uitoefening van het vak in de beroepspraktijk. Ook geeft kennis van het hardware niveau inzicht in de snelheid van programma's. Deze kennis helpt dus om efficiëntere code te schrijven. Het cursusmateriaal is grotendeels gebaseerd op simulaties die inzicht geven in de werking van schakelingen van poortniveau tot en met pipelined processor. Door een hoge mate van visualisatie is de werking van de vele aspecten van computersystemen in betrekkelijk korte tijd te doorgronden. Het materiaal bestaat uit een basismodule, een module digitale techniek en een module computerarchitectuur.

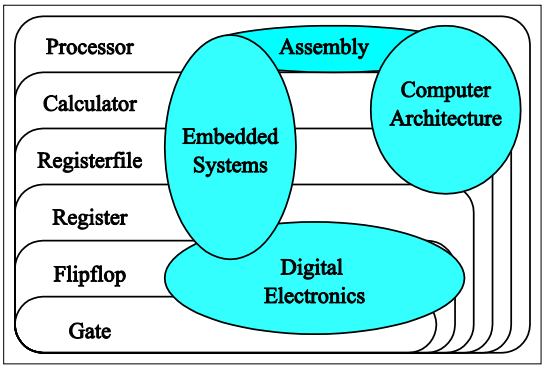
Basismodule
Het doel van de basismodule is om de opbouw en werking van een programmeerbare machine te laten begrijpen. De eerste twee hoofdstukken, "Poorten en logische schakelingen" en "Hoe rekent een computer?" leggen een basis hiervoor. Het derde hoofdstuk, "Hoe werkt een rekenmachine?" laat zien wat er precies gebeurt met de hardware als een 'programma' wordt uitgevoerd.

Module digitale techniek
Deze module is gericht op het systematisch ontwerpen van combinatorische en sequentiële schakelingen zoals Finite State Systems. Hierbij wordt gebruik gemaakt van regels uit de Boole-algebra. In het laatste deel van deze module worden meer complexe digitale schakelingen zoals decoders, tellers, registers en toepassingen van deze schakelingen behandeld.

Module computerarchitectuur; pipelining, function calls en caching
In de eerste hoofdstukken van de module wordt de hardware van de rekenmachine uit de basismodule in twee stappen uitgebreid tot een volledige "five stage pipelined processor" volgens het Harvard model. Op een viertal machines worden dezelfde programma's geïmplementeerd. Het doel hiervan is inzicht te krijgen in het verschil in CPU-time en dus performance van de verschillende machines. Er is hierbij gekozen voor een zeer kleine instructieset. Toch kunnen hiermee programma's voor veel toepassingen zoals sorteerroutines in de taal assembler worden geschreven.
Met een kleine uitbreiding van het Harvard model is er een simulatiemodel waarmee Function Calls en de werking van een stack worden gevisualiseerd. Ook is een hoofdstuk gewijd aan de geheugen hiërarchie. Twee modellen van caches, de direct mapped cache en een set-associative cache komen hierbij aan de orde.

Flexibele cursus

Hiernaast is weergegeven welke lagen van een computersysteem aan bod komen. Uit het materiaal kan op eenvoudige wijze een cursus worden samengesteld voor vakken als Embedded Systems, Digital Electronics of Computerorganization. Na de basismodule kan worden gekozen voor de module digitale techniek of de module computerarchitectuur of een mix van allebei. Ook kunnen met dit materiaal deficiënties in kennis van studenten die niet via de reguliere weg instromen in een opleiding, eenvoudig aangevuld worden en is het materiaal geschikt voor zelfstudie.



Hardware benadering

Er is gekozen om dit onderwerp vanuit de ‘hardware’ te benaderen. Dit houdt in dat alle voorbeelden van schakelingen en systemen ook echt gebouwd kunnen worden in hardware. Tijdvolgordediagrammen maken het signaalverloop door de componenten duidelijk.

SIM-PL

De gebruikte simulatiesoftware is SIM-PL. SIM-PL is een aan de Universiteit van Amsterdam ontwikkelde auteursomgeving waarmee docenten en studenten op een eenvoudige manier schakelingen en componenten kunnen ontwerpen, bouwen en testen. Er is een bibliotheek met een groot aantal componenten, schakelingen en machines aanwezig.

Website

De nieuwste versie van de simulatiesoftware SIM-PL kan worden gedownload via:

http://csa.science.uva.nl/sim-pl/ Dit is de voormalige site: www.science.uva.nl/amstel/SIM-PL/.

Ook zijn hier de antwoorden op toets- en tentamenvragen te vinden. Voor docenten zijn er presentaties en voorbeelden van tentamens beschikbaar.

Dank

De auteur is veel dank verschuldigd aan Wouter Koolen-Wijkstra. Wouter ontwikkelde de SIM-PL-software en bouwde de architecturen van de laatste hoofdstukken. Ook is dank verschuldigd aan Ronald Heijeler voor zijn vele nuttige op- en aanmerkingen.

Wij hopen dat je veel plezier aan dit materiaal zult beleven.

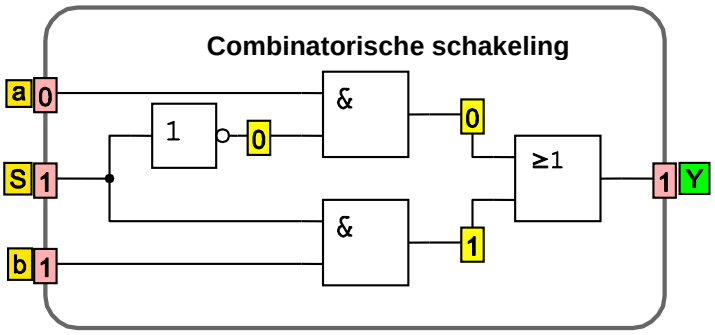
Ben Bruidegom.

Inhoudsopgave

BASISMODULE **7**
Hoofdstuk 1: Poorten en combinatorische schakelingen **7**
1.1 Leerdoelen 7
1.2 De bit 8
1.3 Logische operatoren 8
1.4 Poorten 9
1.5 Combinatorische schakeling en bijbehorende 'formule' 10
1.6 Vragen en opdrachten 11
1.7 SIM-PL 13
1.8 Verdieping: Propagatietijd en tijdvolgordediagram 17
1.9 Bouw en test je eigen multiplexer 19
1.10 Begrippenlijst 21
1.11 Vragen en opgaven 22
Hoofdstuk 2: Hoe rekent een computer? **23**
2.1 Leerdoelen 23
2.2 Binaire representatie van getallen 24
Van binair naar decimaal 24
Van decimaal naar binair 24
2.3 Hexadecimale talstelsel 25
2.4 Optellen 26
Hardwarecomponenten om getallen op te tellen 27
2.5 Opdrachten met gehele getallen zonder teken 27
2.6 Gehele getallen met teken: De two's complement representatie 30
2.7 Sign extension 32
2.8 Operatoren voor bitmanipulatie 33
2.9 Vragen en opgaven met gehele getallen met teken 33
2.10 ASCII en UNICODE 36
2.11 Begrippenlijst 37
2.12 Vragen en opgaven 38
Hoofdstuk 3: Hoe werkt een rekenmachine? **39**
3.1 Leerdoelen 39
3.2 Overzicht van de rekenmachine 40
3.3 Instructies 42
3.4 Practicum met de Rekenmachine I 44
3.5 Hoe krijg ik een constant getal in een register? 46
Architectuur Rekenmachine II 47
3.6 Opdrachten 48
3.7 Practicum met de rekenmachine II 49
3.8 Begrippenlijst 51
3.9 Vragen 52
Hoofdstuk 7: Hoe werkt een "loopje" nu precies? **53**
7.1 Leerdoelen 53
7.2 Benodigde extra hardware om Branch-instructies uit te voeren 54
7.3 Instructieset van de 'Loopsmachine' en implementatie van conditionele statements en herhalings statements 55
7.4 Waar is de snelheid van een computer van afhankelijk? 57
7.5 Opgaven en practicum met de "Calculator with loops" 58
7.6 Begrippenlijst 61
7.7 Tentamenvragen 62
Hoofdstuk 8: De Harvard Single Cycle processor **63**
8.1 Leerdoelen 63
8.2 Uitbreiding hardware 64
8.3 De instructies Load Word en Store Word en de instructieset van de Harvard machine 64
8.4 Operanden die in het geheugen staan 66
8.5 Executietijd, klokfrequentie, performance en Amdahl's law 67
8.6 Opgaven en pract. met de Harvard Single Cycle machine 70
8.7 Begrippenlijst 72
8.8 Tentamenvragen 73
Hoofdstuk 9: De Harvard pipelined processor **75**
9.1 Inleiding en leerdoelen 75
9.2 De werking van de pipelined machine 76

9.3 Pipeline hazards 77
9.4 De pipelined machine met forwarding 80
9.5 De pipelined machine met branch prediction 82
9.6 Opgaven en practicum met de pipelined machines 83
9.7 Terugblik en historisch perspectief 86
9.8 Begrippenlijst 87
9.9 Tentamenvragen 88
Hoofdstuk 10: Hoe werkt een procedure? 91
10.1 Leerdoelen 91
10.2 Wat gebeurt er precies als een procedure wordt uitgevoerd? 92
10.3 Procedures en de stack 93
10.4 Recursieve procedures 95
10.5 Het datapad van de instructies JSR en RETURN 97
10.6 Practicum met de Jumper Machine 98
10.7 Historisch perspectief 101
10.8 Begrippenlijst 101
10.9 Tentamenvragen 102
Hoofdtuk 11: Caches in de Memory Hierarchy 105
11.1 Inleiding en leerdoelen 105
11.2 Direct-mapped cache en 2-way set associative cache 107
11.3 Geïmplementeerde caches in SIM-PL 109
11.4 Theorievragen en opdrachten 110
11.5 Practicumopdrachten caching 112
11.6 Begrippenlijst 114
11.7 Tentamenvragen 115
11.8 Moore's law bestaat al 51 jaar. Hoe lang nog? 116
Hoofdstuk 12: Een busgeorganiseerde rekenmachine 117
12.1 Inleiding en practicummateriaal 117
12.2 Data-opslag in een register 118
12.3 Data-overdracht via een bus 119
12.4 Practicum een busgeorganiseerde rekenmachine 122
12.5 Sequencer 127
12.6 Begrippenlijst 128

BASISMODULE



Hoofdstuk 1: Poorten en combinatorische schakelingen

1.1 Leerdoelen

De basisbouwstenen waaruit computers zijn opgebouwd worden poorten genoemd. Deze poorten kun je combineren tot een schakeling waarmee bijvoorbeeld getallen kunnen worden opgeteld. Om schakelingen te bouwen en te testen gebruik je het computerprogramma SIM-PL. Het vakgebied dat hier bij hoort heet ‘Digitale Techniek’.

Aan het einde van dit hoofdstuk weet je:
- wat een bit is;
- wat een logische operator is;
- wat een waarheidstabel is;
- wat een Inverter is en wat AND, OR en XOR-poorten zijn;
- wat een logische schakeling is;
- wat een combinatorische schakeling is;
- wat de wetten van De Morgan zijn;
- wat de propagatietijd van een poort is;
- wat een tijdvolgordediagram is;
- wat een multiplexer is.

Aan het einde van dit hoofdstuk kun je:
- van een combinatorische schakeling een waarheidstabel maken;
- van een formule een combinatorische schakeling tekenen;
- met de SIM-PL-Editor een schakeling bouwen;
- met de SIM-PL-Executer een schakeling testen.

1.2 De bit

Een computer werkt alleen met ‘nullen’ en ‘enen’. Is deze uitspraak waar? Ja, want een 0 of een 1 hoeft namelijk niet per se een getal voor te stellen. Het is mogelijk daar ook een andere betekenis aan toe te kennen. Een voorbeeld: Een computerchip bevat miljoenen transistoren. Deze transistoren werken als schakelaars. De spanning op de uitgang van een schakelaar is 0 volt of 5 volt. Aan deze spanningsniveaus kennen we een waarde toe: 0 volt = ‘0’[1)] en 5 volt = ‘1’. Een ander voorbeeld komt uit de logica: Een 0 kan betekenen dat een ‘bewering’ niet waar is (FALSE). De waarde 1 zou dan betekenen dat de bewering wel waar is (TRUE). Hieronder nog enkele voorbeelden van toestanden of signalen die twee mogelijke waarden kunnen aannemen:

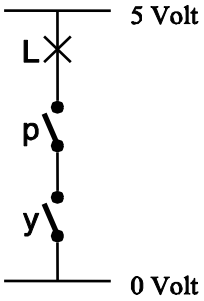
- Dicht / Open
- Uit / Aan

De term 'bit' is een afkorting van 'binary digit'. Deze term duidt op een beperking tot twee onderscheiden situaties. Om verwarring te voorkomen met de decimale cijfers 0 en 1 worden binaire cijfers aangeduid met de symbolen '0' en '1'. In tabellen wordt echter 0 en 1 gebruikt.

1.3 Logische operatoren

Operatoren zijn bewerkingen op getallen. In het geval 12 + 23 is het teken + de rekenkundige operator voor optellen. Om logische operatoren te verduidelijken beschouwen we eerst enkele schakelingen uit de elektrotechniek.

Serieschakeling
In figuur 1.1 zijn twee schakelaars in serie weergegeven. We stellen het volgende:
L = ‘1’ dan brandt de lamp;
L = ‘0’ dan brandt de lamp niet;
p = ‘1’ dan is het schakelcontact gesloten;
p = ‘0’ dan is het contact niet gesloten.
Hetzelfde geldt voor y.
De lamp L brandt alleen als van beide schakelaars de contacten zijn gesloten. De logische operator die we nodig hebben om de schakeling te beschrijven met een formule is de AND-operator. De formule die bij deze schakeling hoort is: L = p AND y.



Figuur 1.1

Waarheidstabel

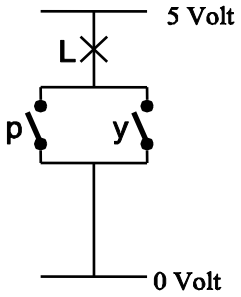
p	y	L
0	0	0
0	1	0
1	0	0
1	1	1

Tabel 1.1

Om de relatie tussen de waarden op de ingangen en de uitgang van een schakeling te beschrijven, wordt een waarheidstabel gebruikt. In de linker kolom van de tabel staan alle mogelijke combinaties van ingangswaarden en in de rechter kolom staat de bijbehorende uitgangswaarde. Tabel 1.1 geeft de relatie tussen p, y en L weer (L is een functie van p en y).

[1)] In vakliteratuur kom je ook vaak L(ow) en H(igh) tegen.

Parallelschakeling
In figuur 1.2 zijn twee schakelaars parallel weergegeven. De lamp brandt als het contact van minstens één van beide schakelaars is gesloten. De logische operator die hierbij hoort is de OR-operator.
De formule die bij deze schakeling hoort is: L = p OR y.



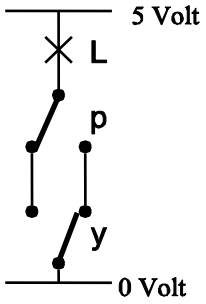
Figuur 1.2

p	y	L
0	0	0
0	1	1
1	0	1
1	1	1

Tabel 1.2

Tabel 1.2 hoort bij figuur 1.2.

Hotelschakeling
In figuur 1.3 is een zogenaamde hotelschakeling weergegeven. Met deze schakeling kun je met twee wisselschakelaars het licht op twee plaatsen aan- of uitschakelen. Deze schakeling wordt vaak toegepast bij trapportalen en in slaapkamers.



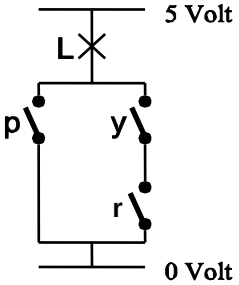
Figuur 1.3

p	y	L
0	0	0
0	1	1
1	0	1
1	1	0

Tabel 1.3

In de getekende stand zijn p en y beide ‘0’.
De operator die bij deze schakeling hoort is de XOR-operator (eXclusive OR-operator). De lamp brandt alleen als óf p óf y ‘1’ is, dat wil zeggen in de niet getekende stand is geschakeld. In formulevorm: L = p XOR y.

Serie-parallelschakeling
In figuur 1.4 is een serie-parallelschakeling weergegeven.



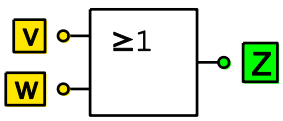
Figuur 1.4

p	y	r	L
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Tabel 1.4

Doordat deze schakeling drie schakelaars heeft, zijn er acht ($2*2*2 = 2^3$) combinaties van verschillende schakelstanden van p, y en r mogelijk. De tabel die bij deze schakeling hoort, heeft dus acht regels. De lamp brandt als p = ’1’ of als y en r beide ‘1’ zijn.
De formule die bij deze schakeling hoort, is:
L = p OR (y AND r).

1.4 Poorten



Een poort is een schakeling opgebouwd uit transistoren. Een poort heeft één of meer ingangen en één uitgang. De uitgang en de ingangen kunnen alléén de waarde ‘0’ en ‘1’ aannemen. Poorten worden daarom *logische of digitale schakelingen* genoemd. De belangrijkste poorten zijn: de inverter, de AND-poort, de OR-poort en de XOR-poort. De bij deze poorten behorende logische operatoren zijn respectievelijk NOT, AND, OR en XOR. Ieder type poort heeft een eigen symbool. Hiernaast is het symbool van een OR-poort met twee ingangen weergegeven. Het symbool bestaat uit een vierkant. De twee lijnen links van dit vierkant zijn ingangen. De willekeurig gekozen letters v en w geven iedere ingang een naam. De lijn rechts van het vierkant is de uitgang. Deze heeft de hoofdletter Z als naam. De tekst in het vierkant, hier ≥1, geeft de relatie tussen ingangen en uitgang aan (≥ 1 betekent hier minstens 1). De symbolen zijn die van de IEC-standaard (International Electrotechnical Commission).

In dit hoofdstuk komen de volgende poorten voor:

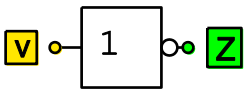
Inverter

Waarheidstabel

v	Z
0	1
1	0

Tabel 1.5

Een inverter heeft één ingang. Het signaal op deze ingang wordt geïnverteerd (‘1’ wordt ‘0’ ofwel ‘hoog’ wordt ‘laag’ en omgekeerd). De waarde van de uitgang Z is de logische omkering (negatie, ontkenning, inverse) van de ingang v. Het ‘bolletje’ bij de uitgang Z van het symbool geeft de inversie aan. Er geldt: Z = NOT v.



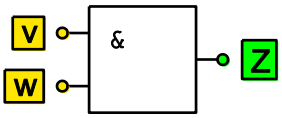
AND-poort

Waarheidstabel

v	w	Z
0	0	0
0	1	0
1	0	0
1	1	1

Tabel 1.6

De AND-poort heeft twee of meer ingangen. Alle ingangssignalen moeten ‘hoog’ zijn om een hoog uitgangssignaal te verkrijgen. Een AND-poort met twee ingangen is hier weergegeven. Het symbool & (ampersand) wordt vaak gebruikt als ‘en’-teken.
Er geldt: Z = v AND w.



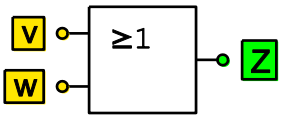
OR-poort

Waarheidstabel

v	w	Z
0	0	0
0	1	1
1	0	1
1	1	1

Tabel 1.7

De OR-poort heeft twee of meer ingangen. Indien één of meer ingangen ‘1’ zijn, is de uitgang ‘1’. Alleen als alle ingangen ‘0’ zijn, is de uitgang ‘0’. Een OR-poort met twee ingangen is hier weergegeven. Het symbool ≥1 betekent minstens 1. Er geldt: Z = v OR w.



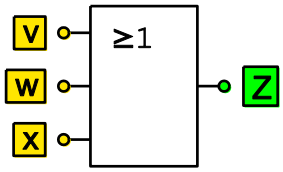
OR-poort met drie ingangen

Waarheidstabel

v	w	x	Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Tabel 1.8

AND- en OR-poorten kunnen meer dan twee ingangen hebben. Hiernaast is de OR-poort met drie ingangen weergegeven.
Er geldt: Z = v OR w OR x.



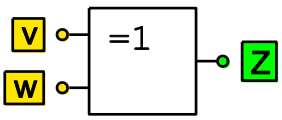
XOR-poort

Waarheidstabel

v	w	Z
0	0	0
0	1	1
1	0	1
1	1	0

Tabel 1.9

De XOR-poort (eXclusive OR) heeft altijd twee ingangen. Alleen als precies één van de ingangen ‘1’ is, wordt de uitgang ‘1’. De tekens =1 in het symbool betekenen hier: precies 1.
Er geldt: Z = v XOR w.



We vergelijken dit met de volgende beweringen:

- Jan woont in Venlo. Als dat zo is dan is v = ‘1’ (TRUE).
- Jan woont in Winterswijk. Als dat zo is dan is w = ‘1’ (TRUE).

We nemen aan dat Jan niet in beide steden woont. De ene mogelijkheid sluit de ander uit. Dus:

- Jan woont óf in Venlo óf in Winterswijk.

1.5 Combinatorische schakeling en bijbehorende ‘formule’

In figuur 1.5 is een combinatie van een AND-poort en een OR-poort weergegeven. Een

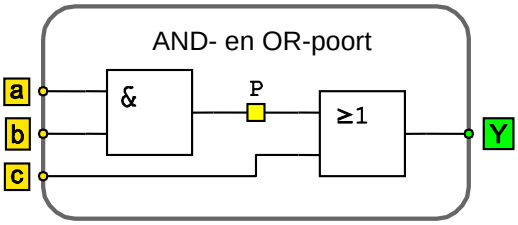
dergelijke combinatie van poorten wordt een *combinatorische schakeling*[2)] ook wel *digitale schakeling* genoemd. In de bijbehorende tabel 1.10 is een extra kolom weergegeven. Het punt P, van probe (testpunt), geeft de waarde weer van de uitgang van de AND-poort.

Van schakeling → formule

De 'formule' die bij de schakeling hoort, is: Y = (a AND b) OR c. Ook de formule: Y = c OR (a AND b) hoort bij deze schakeling. Het is nodig in formules haakjes te plaatsen. Worden deze weggelaten zodat: Y = a AND b OR c, dan kunnen we dat ook interpreteren als: Y = a AND (b OR c). Dit is een formule die bij een andere schakeling hoort.

a	b	c	P	Y
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

Tabel 1.10



Figuur 1.5

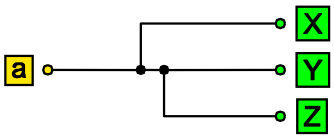
Van formule → schakeling

Stel: van de formule Y = (a AND b) OR c moet een schakeling worden getekend. Hoe pak je dit aan?

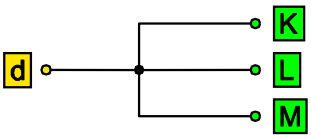
- werk van links naar rechts; teken poortingangen altijd links en de poortuitgang rechts;
- begin met de poort te tekenen die het dichtst bij de poortingangen staat, in dit geval de AND-poort;
- voeg op dezelfde wijze de OR-poort toe;
- teken op de juiste manier de nodige verbindingen en aftakkingen.

Het tekenen van verbindingen en aftakkingen

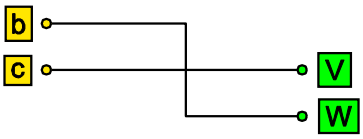
Twee aftakkingen worden bij voorkeur weergegeven door twee 'driesprongen' en niet als één 'viersprong'. Dit is afgesproken om verwarring met kruisende lijnen te voorkomen. Kruisende lijnen maken onderling geen contact.



Juiste weergave aftakkingen
Aftakkingen worden getekend als 'driesprongen'.



Onjuiste weergave aftakkingen
Aftakkingen worden niet getekend als 'viersprongen'.



Kruisende lijnen
Bij kruisende lijnen is er geen verbinding tussen de lijnen cV en bW

1.6 Vragen en opdrachten

Opdracht 1.6.1: Combinatie OR/AND-poort

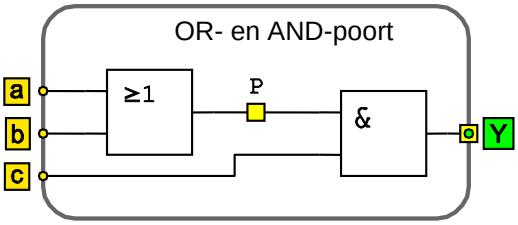
De schakeling in figuur 1.6 heeft drie ingangen: a, b en c en een uitgang: Y. Geef de formule die bij deze schakeling hoort.
Y = ……..
Vul tabel 1.11, de waarheidstabel die bij deze schakeling hoort verder in.

a	b	c	P	Y
0	0	0		

[2)] Deze definitie van combinatorische schakeling is onvolledig. Bij een combinatorische schakeling wordt de waarde van de uitgang bepaald door de *momentele* waarden van de ingangen. In hoofdstuk 5 komen logische schakelingen aan de orde waarbij ook de vorige toestand van de schakeling een rol speelt.

0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Tabel 1.11



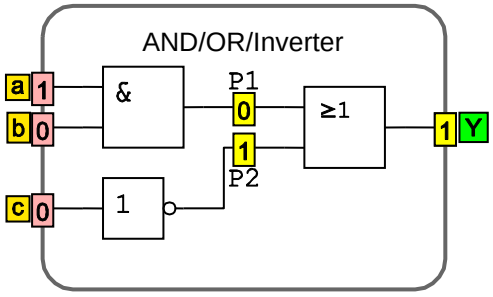
Figuur 1.6

Opdracht 1.6.2: Combinatie AND/OR/Inverter

De schakeling weergegeven in figuur 1.7 heeft drie ingangen: a, b en c en een uitgang: Y.
Vraag: De formule bij de schakeling van figuur 1.7 is: Y = …….
Vul tabel 1.12 in.

a	b	c	P1	P2	Y
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Tabel 1.12



Figuur 1.7

Opdracht 1.6.3: Van formule naar schakeling met drie poorten

Ontwerp een schakeling voor de volgende formule:
Y = (NOT a) AND (b OR c)
Geef het resultaat weer in figuur 1.8. Maak gebruik van IEC-symbolen!

Figuur 1.8

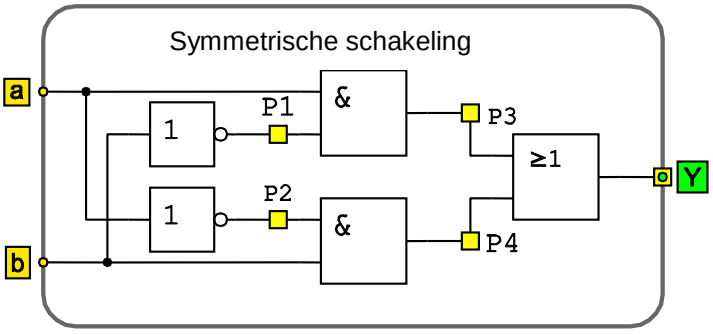
Opdracht 1.6.4: Van formule naar schakeling met vier poorten

Ontwerp een schakeling voor de volgende formule:
Y = (a AND (NOT b)) XOR (c AND d)
Geef het resultaat weer in figuur 1.9. Maak gebruik van IEC-symbolen!

Figuur 1.9

Opdracht 1.6.5: Symmetrisch opgebouwde schakeling

De schakeling in figuur 1.10 heeft twee ingangen: a en b en een uitgang: Y. Vul tabel 1.13 in.



Figuur 1.10

Met welke poort komt deze tabel overeen als je kijkt naar de kolommen a, b en Y?
Antwoord: Dit is de tabel van de …… -poort.
Wat is de formule die bij deze schakeling hoort?
Antwoord: Y = ……..

a	b	P1	P2	P3	P4	Y
0	0					
0	1					
1	0					
1	1					

Tabel 1.13

1.7 SIM-PL

SIM-PL is een computerprogramma dat de werking van logische schakelingen zoals computers helpt verduidelijken. Het softwarepakket (zie ook pag. 4) simuleert schakelingen van een paar poorten tot complete processoren. SIM-PL is een auteursyteem waarmee docenten en studenten vrij eenvoudig zelf componenten en schakelingen kunnen construeren.
SIM-PL bestaat uit twee programma's:

- Een simulator.
  Met de simulator, executer genoemd, worden schakelingen en hun gedrag gesimuleerd.
- Een editor.
  Met de editor worden basiscomponenten zoals poorten gebouwd. Met deze zelfgebouwde en al bestaande componenten kunnen logische schakelingen worden samengesteld.

Opdracht 1.7.1: Werken met de SIM-PL-simulator

De eerste opdracht is de simulatie van een combinatorische schakeling met drie poorten.

Open hiertoe de folder SIM-PLx.x.x. Dubbelklik op de file Executer.jar. Klik op de icoon: 
Hierdoor wordt een bestaand werkblad (worksheet genaamd) geopend. Klik op Components → H1Poorten → AND_OR_Inv.sim-pl-ws.

De schakeling die op je scherm verschijnt, is opgebouwd uit een OR-poort, een AND-poort en een inverter (zie figuur 1.11). De schakeling heeft drie ingangen a, b en c en één uitgang Y. Alle beginwaarden, ook die van de uitgang en van de beide testpunten P, zijn '1'. De waarden van P en Y kunnen onjuist zijn omdat de simulator de schakeling nog niet heeft doorgerekend. Om een simulatie uit te voeren zijn er vier knoppen met pijlen die de volgende betekenis hebben:


Initialisatie. De simulator keert terug naar zijn begintoestand. Alle in- en uitgangen krijgen hun beginwaarde '1'.

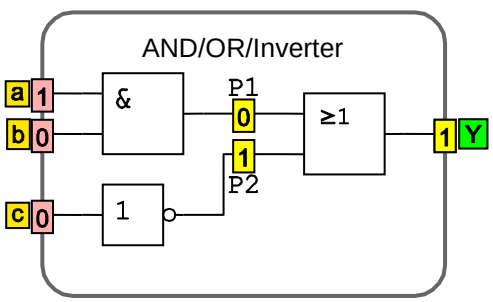

Step. Hiermee wordt de kleinst mogelijke simulatiestap gegeven.


Cycle. Hiermee worden alle simulatiestappen tot de volgende klokpuls gegeven. Dit houdt in dat de gehele schakeling doorgerekend wordt.


Multiple Cycles. Hiermee worden alle simulatiestappen gedurende een zelf te kiezen aantal klokpulsen gegeven.

Een simulatie uitgaande van de begintoestand met de 'Step'-functie

Klik twee keer op de groene pijl (Step). De uitgang van de AND-poort en van de Inverter zijn nu doorgerekend. P1 is '1' gebleven en P2 is '0' geworden. De derde keer dat je op 'Step' klikt wordt de OR-poort doorgerekend. De waarde op de uitgang van de schakeling Y is nu bekend. De simulatie is voltooid. Klik op de rode pijl. Alle waarden hebben nu weer de beginwaarde '1'



Figuur 1.11

gekregen. Een nieuwe simulatie kan worden gestart.

Een simulatie uitgaande van de begintoestand met de 'Cycle'-functie

Klik op de bruine pijl (Cycle). De simulatie wordt nu in één keer uitgevoerd.

Het invoeren van nieuwe waarden op de ingangen

Klik op ingang b en verander de waarde in '0'. Doe hetzelfde met ingang c. Verander ingang a niet. Klik weer twee keer op de groene pijl. P1 is '0' geworden en P2 is '1' geworden. Bedenk voordat je de volgende 'Step' geeft wat uitgang Y zal worden.
Voer de volgende ingangswaarden in: a = '0', b = '1' en c = '1'. Voorspel de waarden van P1, P2 en Y voordat je de simulatie uitvoert.
Voer daarna de simulatie uit.

De menu-optie waarheidstabel

Ga naar de menu-optie Tools → Show Truth-table.
Is de waarheidstabel in overeenstemming met de waarheidstabel die je ingevuld hebt bij opdracht 1.6.2?

Het aansturen van de simulator met een programma

Laad de schakeling met de filenaam: SymmetrischeSchak.sim-pl-ws in de executer. Deze file kun je vinden in de folder: Components/PoortenH1. Je ziet nu een tweede window verschijnen dat de Program Editor wordt genoemd. Kies van dit window de optie: File → Open → TestProgramSymSchak.sal.
Er verschijnt het onderstaande 'programma'. Met dit programma kan de schakeling worden aangestuurd.

```
# Testprogramma symmetrische schakeling
# tijdstip   waarde a      waarde b
  0:         a = 0;        b = 0;
 10:         a = 1;        b = 0;
 20:         a = 0;        b = 1;
 30:         a = 1;        b = 1;
```

Het programma bestaat uit drie kolommen. De linker kolom geeft het tijdstip aan waarop de simulatie wordt uitgevoerd. De andere kolommen geven de waarden aan die de ingangen a en b krijgen op dat tijdstip. Als voorbeeld: op tijdstip 10 krijgt ingang a de waarde '1' en ingang b de waarde '0'. De tekst op de regel achter het #-symbool is commentaar dat toegevoegd kan worden om het programma te verduidelijken.
TestProgramSymSchak.sal bestaat dus uit vier instructies die achtereenvolgens op de tijdstippen 0, 10, 20 en 30 worden uitgevoerd.
Klik op 'Compile'. Als de code correct is verschijnt het groene vinkje. Het programma is nu vertaald naar een code die de schakeling aanstuurt. Klik op de rode pijl. De eerste programmaregel licht op. Het programma staat nu klaar om te worden uitgevoerd. Voer het programma instructie voor instructie uit met behulp van de bruine pijl. Hierdoor wordt iedere keer de code '10 tijdstippen verder' uitgevoerd. Controleer of de uitkomst van iedere programmaregel overeenkomt met de overeenkomstige rij van de tabel van opdracht 1.6.5.
Gebruik de rode pijl om de simulatie opnieuw te starten. Klik eens op de gele pijl en typ 4 in. De vier programmaregels worden nu in één keer uitgevoerd.

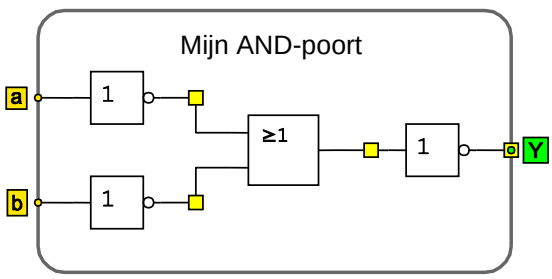
Het zelf 'genereren' van een programma

Met de optie File → Generate Truth Table van de Program Editor kan automatisch een programma worden gemaakt dat de schakeling aanstuurt. Dit programma voert dezelfde opdrachten uit als het programma: TestProgramSymSchak.sal.

Opdracht 1.7.2: Het nabouwen van een schakeling met de Editor van SIM-PL

De opdracht luidt: Bouw de schakeling weergegeven in figuur 1.12 na.
Open hiertoe de folder SIM-PLx.x.x. Dubbelklik op de file Editor.jar.

Klik op de icoon:  'Create a new, empty component'. Er verschijnt het hieronder weergegeven window. Kies de optie 'Complex'. Met deze optie kan een poortschakeling worden gemaakt met bestaande poorten. Geef de

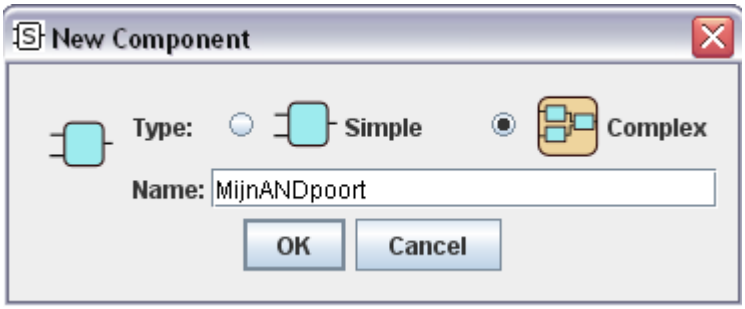


Figuur 1.12

component de naam ‘MijnANDpoort’ en klik op OK. Er verschijnt nu een leeg tekenblad in beeld. Sla de (lege) component op in de folder:
Componenten/H1Poorten onder de naam MijnANDpoort. De extensie ‘.sim-pl’ wordt automatisch aan de filenaam toegevoegd.

Gereedschappen om een schakeling te maken

Boven het lege tekenblad staan een aantal iconen met de volgende functies:



Component. Een icoon om componenten aan de schakeling toe te voegen. Een component kan zowel een poort als een bestaande poortschakeling zijn.

Ingang en uitgang. Iconen om de schakeling te voorzien van ingangen en uitgangen. De linker icoon is voor ingangen en de rechter voor uitgangen.

Verbindingen. Een icoon om verbindingen te maken tussen de ingangen van de schakeling en (sub)componenten, tussen (sub)componenten onderling en tussen (sub)componenten en de uitgang(en) van de schakeling.

Probe. Een icoon om ‘probes’ aan de schakeling toe te voegen. Deze probes worden geplaatst op een verbinding. Zij geven de toestand van de verbinding weer tijdens de simulatie.

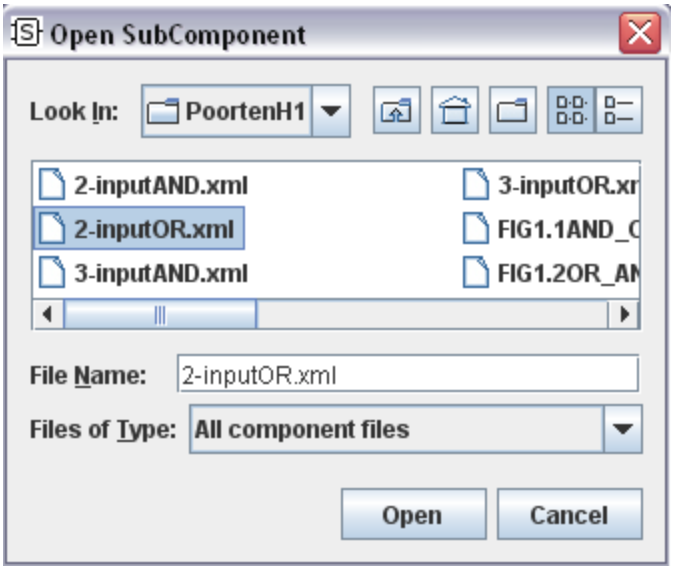
Iconen om de lay-out van de schakeling te maken en teksten toe te voegen.

Select. Een icoon om een onderdeel van de schakeling te selecteren.

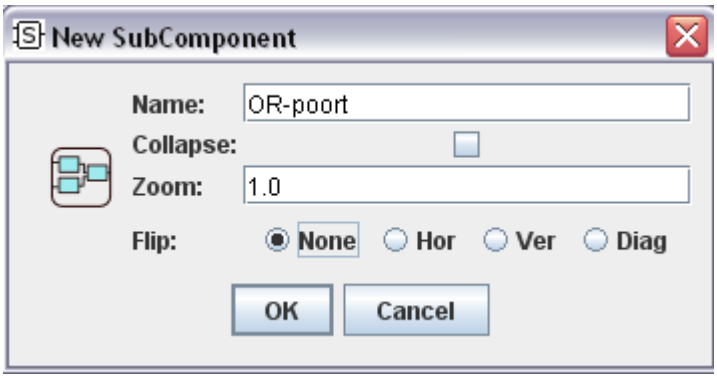
Het editen van de schakeling

1. Componenten

Verzamel eerst de benodigde poorten. Begin met de OR-poort. Klik op de icoon ‘component’.
Klik binnen het tekenblad. Selecteer de component ‘2-inputOR’. Klik op ‘Open’. Er verschijnt een nieuw window. Verander de naam van de component in OR-poort en klik op OK. De component verschijnt nu op het tekenblad. Klik nogmaals binnen het tekenblad. Selecteer de component ‘Inverter’. Verander de naam in InverterLB (Links Boven) Klik nogmaals binnen het tekenblad en selecteer nog twee keer een ‘Inverter’. Geef deze inverters verschillende namen bijvoorbeeld InverterLO (LO Links Onder) en InverterR (R van Rechts).



Klik op de icoon ‘Select’ en sleep de componenten naar de gewenste positie.



2. In- en uitgangen

Klik op de icoon ‘Ingang’ en voorzie de schakeling van twee ingangen. Geef de beide ingangen de namen a respectievelijk b. Klik op de icoon ‘Uitgang’. Geef de uitgangen van de schakeling de naam Y.

3. Verbindingen

Klik op de icoon ‘Verbindingen’. Maak de eerste verbinding door te klikken op een uitgang van een poort en daarna te klikken op een ingang van een andere poort. De verbinding heeft de naam ‘Wire’ gekregen. Links op het scherm bij ‘Property’ kun je deze naam aanpassen tot bijv. OutputWire, door op het desbetreffende onderdeel te klikken. Ook de dikte en de kleur van de lijn kun je op deze manier instellen.

Property	Value
wireColor	
wireName	OutputWire
wireStroke	

Maak ook alle andere verbindingen tussen de poorten en de in- en uitgangen van de schakeling.

4. Probes

Breng probes aan op die verbindingslijnen van de schakeling waarvan je tijdens de simulatie de waarde wilt waarnemen.

5. Lay-out component

Gebruik de ‘tekentools’ om de lay-out van de schakeling te verbeteren en tekst toe te voegen. De opdracht is nu gereed. ‘Save’ de schakeling. Sluit de Editor nog niet.

Opdracht 1.7.3: Het testen van de schakeling ‘MijnANDpoort’

Om de schakeling met de Executer te testen moet er eerst een nieuw werkblad worden aangemaakt.

Maken van een nieuw werkblad

Start de Executer. Klik hiertoe op de icoon:  **‘Create a new worksheet’ voor het maken van een nieuw werkblad.**

Ga naar Componenten → H1Poorten → MijnANDpoort.SIM-PL.

Klik het ‘Time Sequence Diagram-window’ weg en configureer de overige windows naar eigen smaak. Er zijn verschillende zoomopties. Zo wordt bijvoorbeeld met de toetscombinatie: Ctrl = de schakeling groter afgebeeld. Deze zoomopties kun je ook onder de menuoptie ‘view’ vinden.

Testen van MijnANDpoort

Test de schakeling. Gebruik ook de optie ‘Tools → Show Truth Table’.

Als de schakeling correct is nagebouwd, dan blijkt dat de waarheidstabel van de schakeling overeenkomt met die van een AND-poort.

Opdracht 1.7.4: Het veranderen van de schakeling ‘MijnANDpoort’ in de schakeling ‘MijnORpoort’

Keer terug naar de Editor. Gebruik de ‘Save As’-optie om de schakeling MijnANDpoort op te slaan onder de filenaam ‘MijnORpoort’ in de folder H1Poorten.

Naam van de schakeling veranderen

Verander de naam van de schakeling van MijnANDpoort in MijnORpoort. Klik hiertoe op het naamveld linksboven. Hierdoor verschijnt het nevenstaande window. Verander de naam in het hierdoor bestemde veld. Laat de ‘preferedCycleLenght’ en het ‘preferredNumberFormat’

Property	Value
name	MijnORpoort
preferredCycleLength	10
preferredNumberFor...	Hexadecimal

ongewijzigd.

Schakeling aanpassen
Verwijder de OR-poort. Alle draden waarmee de OR-poort is verbonden verdwijnen nu ook. Klik op de icoon: component. Open de component 2-inputAND.xml en zet deze op de plaats van de OR-poort. Sluit de in- en uitgangen op dezelfde wijze aan als in de eerste opdracht en verander de tekst in: MijnORpoort. Voeg probes toe. Save de schakeling en sluit de Editor.

Schakeling testen
Maak een nieuw werkblad aan en test 'MijnORpoort' op dezelfde manier als 'MijnANDpoort'. Met welke poort komt deze schakeling overeen? Antwoord: ……..

Opdracht 1.7.5: Wetten van De Morgan

De wetten van De Morgan
De wetten van De Morgan zijn twee wetten uit de logica die een verband leggen tussen de logische operatoren AND, OR en NOT. Afgeleide vormen van deze twee wetten zijn:
1. a AND b = NOT ((NOT a) OR (NOT b))
2. a OR b = NOT ((NOT a) AND (NOT b))

De schakelingen 'MijnORpoort' en 'MijnANDpoort' bewijzen dat de wetten van De Morgan geldig zijn. Controleer de wetten van De Morgan ook door het invullen van tabel 1.14 en 1.15.

a	b	NOT a	NOT b	(NOT a) OR (NOT b)	NOT ((NOT a) OR (NOT b))	a AND b
0	0					
0	1					
1	0					
1	1					

Tabel 1.14

a	b	NOT a	NOT b	(NOT a) AND (NOT b)	NOT ((NOT a) AND (NOT b))	a OR b
0	0					
0	1					
1	0					
1	1					

Tabel 1.15

Propositielogica
Achter de structuur van Nederlandse zinnen gaan logische vormen schuil. Zo is de bewering 'Het regent *of* het regent *niet*' waar en de bewering 'Het regent *en* het regent niet' onwaar.
Ga na of de volgende twee beweringen overeen komen:
Ik ga zeilen alleen als het waait *en* als jij meegaat.
Ik ga *niet* zeilen alleen als het *niet* waait *of* als jij *niet* meegaat.

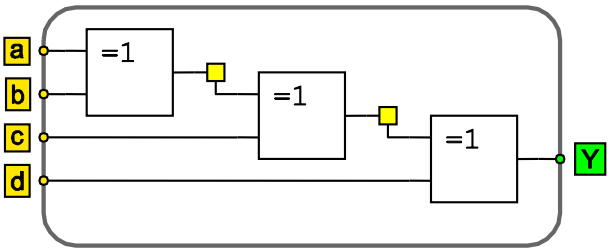
1.8 Verdieping: Propagatietijd en tijdvolgordediagram

Propagatietijd
Iedere poort of schakeling heeft een zogenaamde propagatietijd. Dit is de tijd die verstrijkt tussen een signaalverandering op één of meerdere ingangen en de reactie hierop van de uitgang. Iedere poort of schakeling geeft een signaal vertraagd door. De Engelse term is: 'propagation delay time'. Dit wordt meestal afgekort tot: t_{pd}. Bij de huidige stand van de chiptechnologie is de t_{pd} van een poort ongeveer 0.01 nanosec (ns = $1*10^{-9}$ sec). De poorten van de SIM-PL-simulatieomgeving hebben allen een t_{pd} van 0.01 ns.

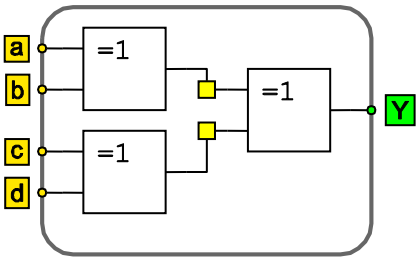
Propagatietijd van een combinatorische schakeling.
De propagatietijd van een combinatorische schakeling is de *langste tijd* die op kan treden tussen een signaalverandering op één of meerdere ingangen en de reactie hierop van de uitgang. Bij het ontwerpen van een schakeling moet er naar worden gestreefd de propagatietijd

zo kort mogelijk te laten zijn.

Voorbeeld: In figuur 1.13 en figuur 1.14 zijn twee oplossingen van de formule: Y = a XOR b XOR c XOR d weergegeven:
1. Y = ((a XOR b) XOR c) XOR d. Deze formule is geïmplementeerd in figuur 1.13;
2. Y = (a XOR b) XOR (c XOR d). Deze formule is geïmplementeerd in figuur 1.14.



Figuur 1.13

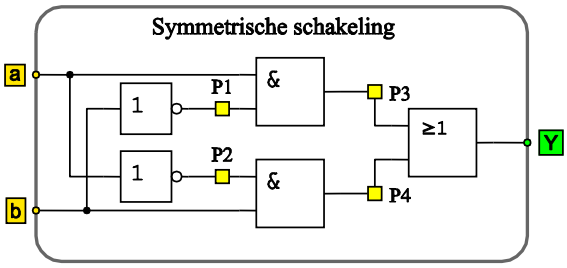


Figuur 1.14

In figuur 1.13 zijn de poorten in serie (in cascade) geschakeld. De langste weg die een signaal van één der ingangen naar de uitgang Y kan afleggen, gaat via drie poorten, namelijk van ingang a of b naar uitgang Y. De maximale t_{pd} van de schakeling is dus 3 * 0.01 ns = 0.03 ns. In figuur 1.14 zijn de poorten meer parallel geschakeld. De signalen van alle ingangen naar Y gaan via twee poorten dus de t_{pd} van deze schakeling is 0.02 ns. Omdat computerschakelingen zo snel mogelijk berekeningen moeten kunnen uitvoeren, is de implementatie zoals weergegeven in figuur 1.14 beter dan die van figuur 1.13.

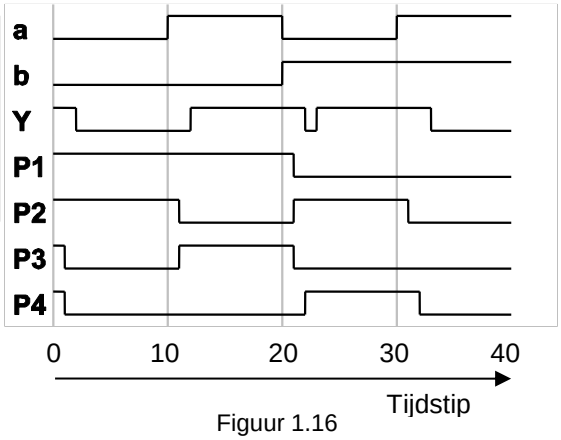
Vraag: Wat is van de schakeling afgebeeld in figuur 1.15 de maximale t_{pd} in ns? Antwoord: …..

Tijdvolgordediagram of Time Sequence Diagram
Een tijdvolgordediagram geeft het signaalverloop op meerdere punten in een schakeling zodanig weer, dat de onderlinge samenhang zichtbaar wordt.
Als voorbeeld beschouwen we nogmaals de schakeling van figuur 1.10, nu figuur 1.15 genaamd, en het bijbehorende testprogramma.



Figuur 1.15

```
# Testprogramma symmetrische schakeling
# tijdstip   waarde a      waarde b
  0:         a = 0;        b = 0;   # regel 1
 10:         a = 1;        b = 0;   # regel 2
 20:         a = 0;        b = 1;   # regel 3
 30:         a = 1;        b = 1;   # regel 4
```



Figuur 1.16

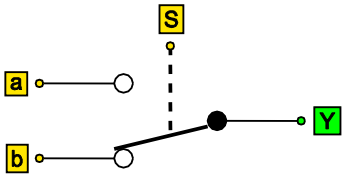
In het tijdvolgordediagram van figuur 1.16 is het verloop in de tijd van de signalen op de ingangen a en b, op de uitgang Y en op de testpunten P1, P2, P3 en P4 tijdens het uitvoeren van het testprogramma weergegeven. Op de horizontale as is de tijd weergegeven. Om de 10 tijdstippen zijn in lichtgrijs verticale strepen weergegeven. Eén tijdstip is 0.01 ns.

Signaalverloop bij het uitvoeren van het programma
Eerst kijken we naar wat er gebeurt als regel 2 van het programma wordt uitgevoerd. Op tijdstip 10 gaat a van ‘0’ naar ‘1’ en b blijft ‘0’. Na 0.01 ns veranderen P2 en P3. Na 0.02 ns op tijdstip 12 heeft Y zijn eindwaarde bereikt. Vervolgens kijken we naar wat er gebeurt als regel 3 van het programma wordt uitgevoerd. Op tijdstip 20 gaat a van ‘1’ naar ‘0’ en b van ‘0’ naar ‘1’. Y verandert na 0.02 ns van ‘1’ naar ‘0’ en vervolgens na 0.03 ns weer naar de oorspronkelijke waarde ‘1’.

De oorzaak is de ‘looptijdverschillen’ van de signalen op P3 en P4. Tenslotte kijken we naar de executie van regel 4. Op tijdstip 30 gaat a van ‘0’ naar ‘1’ en blijft b ‘1’. Na 0.01 ns verandert P2. Na 0.02 ns verandert P4. Nu wordt de eindwaarde van Y pas na 0.03 ns op tijdstip 33 bereikt. Dit is de (maximale) t_{pd} van de schakeling.

De ‘Clock’ van SIM-PL
In de SIM-PL-simulatieomgeving kun je voor elke schakeling de duur van één klokpuls instellen. In de Editor is hiervoor het invoerveld: preferedCycleLength (zie opdracht 1.7.4). De standaard ingestelde waarde, meestal ‘default’ genoemd, is 10. De duur van één klokpuls is 10 tijdsstippen, dus 10 * de t_{pd} van één poort.
In de Executer kun je de duur van één klokpuls instellen via de menuoptie: Simulate → Cycle Length.

1.9 Bouw en test je eigen multiplexer



Figuur 1.17

Een veel voorkomende toepassing van poortschakelingen is het schakelen van datastromen. Een mechanisch equivalent van een multiplexer, een wisselschakelaar, is in figuur 1.17 afgebeeld. De stand van de schakelaar wordt bepaald door ingang S. S staat voor Select. Als S = ‘0’, dan wordt de waarde op ingang b doorgelaten naar uitgang Y. Deze stand van de schakelaar is in figuur 1.17 weergegeven. Als S = ‘1’, dan wordt de waarde op ingang a doorgelaten naar Y. Deze schakeling wordt multiplexer of dataselector genoemd. De naam multiplexer wordt vaak afgekort tot Mux.

Formule
De formule die bij deze schakeling hoort is: Y = (a AND S) OR (b AND (NOT S)).

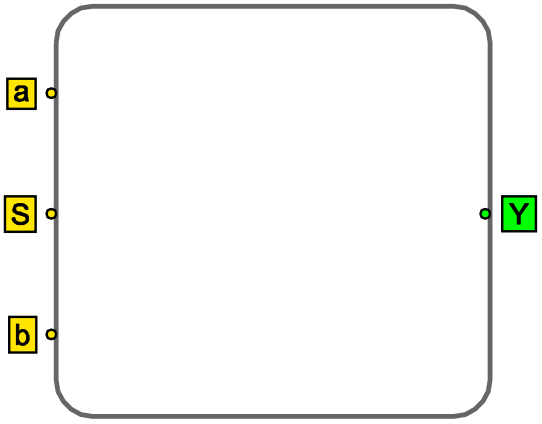
> De manier waarop deze formule is verkregen, wordt in dit hoofdstuk niet behandeld. Dit vereist een methode om uit een tabel een formule te maken en kennis van regels uit de Boole-algebra om de ontstane formule te vereenvoudigen. Deze wat moeilijkere onderwerpen worden in hoofdstuk 4 behandeld.

Opdracht 1.9: Ontwerp, bouw en test een multiplexer
Opdracht 1.9a: Vul tabel 1.16 in.

S	a	b	a AND S	b AND (NOT S)	Y
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Tabel 1.16

Opdracht 1.9b: Ontwerp de schakeling en geef je ontwerp weer in figuur 1.18.

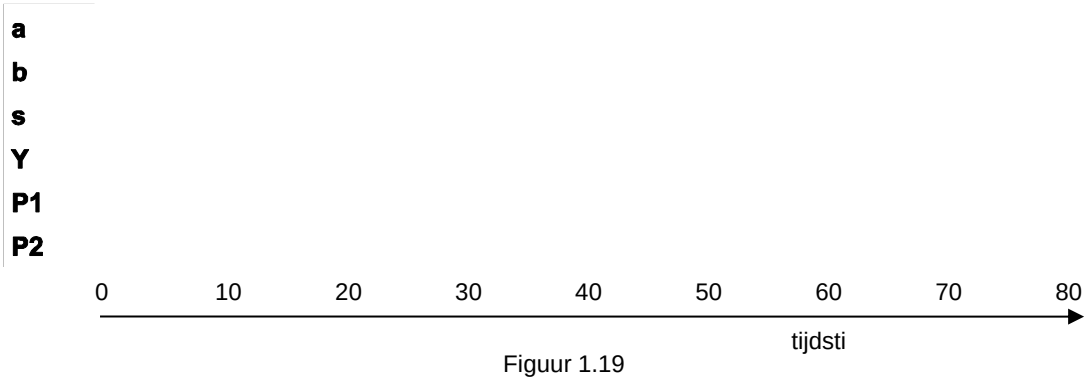


Figuur 1.18

Opdracht 1.9c: Implementeer je ontwerp met de SIM-PL-editor. Haal de benodigde poorten uit de folder H1Poorten. Voeg twee probes P1 en P2 toe op de twee ingangen van de OR-poort. Verander ook de namen van de verbindingen waarop de probes zijn geplaatst in P1 en P2. Sla de schakeling op onder de naam MijnMux.

Opdracht 1.9d: Test je ontwerp m.b.v. de SIM-PL-executer. Creëer hiervoor een nieuw werkblad.
Ga met de muis naar probe P1 en gebruik de rechter muisknop om Graph P1 aan te vinken. Doe hetzelfde met P2.
Test je ontwerp als volgt:

1. Test je schakeling met de menuoptie Tools → Show Truth Table.
2. Schrijf een programma om de schakeling te testen. Gebruik hiervoor de optie File → Generate Truth Table van de Program Editor om automatisch een programma te genereren. Verander de 'time interval' niet, deze is standaard ingesteld op 10. Compileer en run het programma. Beweeg met de muis boven de signalen in het Time Sequence Diagram om van een signaal de waarde op het bijbehorende tijdstip af te lezen. Geef een schets van het tijdvolgorde diagram weer in figuur 1.19.



Figuur 1.19

Vraag: Wat is de maximale propagatietijd van de schakeling afgebeeld in figuur 1.18?

1.10 Begrippenlijst

Bit. Afkorting van: 'binary digit'. Deze term duidt op een beperking tot twee onderscheiden situaties. Gewoonlijk worden deze aangeduid met de symbolen '0' en '1'.

Combinatorische schakeling. Schakeling die uit poorten is samengesteld. De waarde van het uitgangssignaal wordt bepaald door de momentele waarden van de ingangssignalen.

Default. Vooraf ingestelde waarde.

Digitale techniek. Techniek waarbij digitale schakelingen worden toegepast.

Digitale schakeling. Zie logische schakeling.

SIM-PL-Editor. Softwareprogramma om digitale schakelingen te bouwen.

SIM-PL-Executer. Softwareprogramma om simulaties van digitale schaklingen uit te voeren.

Inverteren. Van logische waarde wisselen.

Logische schakeling. Schakeling waarvan in- uitgangen alléén de logische waarden ‘0’ en ‘1’ kunnen aannemen.

Multiplexer. Schakeling met minimaal twee gegevensingangen, één of meer selectie-ingangen en één uitgang. De code op de selectie-ingang(en) bepaalt welke gegevensingang wordt doorverbonden met de uitgang. Een selectie-ingang wordt ook wel een adresingang genoemd.

Poort. Logische schakeling met één of meer ingangen en één uitgang, waarbij een eenduidig verband bestaat tussen de ingangswaarden en de uitgangswaarde.

Probe. Testpunt in een schakeling dat de toestand ervan tijdens de simulatie weergeeft.

Propagatietijd. De tijd langste tijd die op kan treden tussen een signaalverandering op één of meerdere ingangen, en de reactie hierop van de uitgang.

Propositie. Bewering die waar (True) of onwaar (False) is.

Propositielogica. Een logica die zich bezighoudt met geldige redeneringen in de vorm van proposities.

t_{pd}. Afkorting van propagation delay time (voortplantingsvertragingstijd) Zie propagatietijd.

SIM-PL. Simulatieomgeving die de werking van digitale systemen zoals computers helpt verduidelijken.

Tijdvolgorde-diagram. Schema waarin het signaalverloop op belangrijke punten in een schakeling wordt weergegeven, zodanig dat de onderlinge samenhang duidelijk wordt.

Waarheidstabel. Beschrijving van de relatie tussen de waarden op de ingangen en de uitgang(en) van een poort of een combinatorische schakeling.

Wetten van De Morgan. De wetten van De Morgan zijn twee wetten uit de logica die een verband leggen tussen de logische operatoren AND, OR en NOT.
Afgeleide vormen van deze twee wetten zijn:
1. NOT(a AND b) = (NOT a) OR (NOT b);
2. NOT(a OR b) = (NOT a) AND (NOT b).

1.11 Vragen en opgaven

Ter voorbereiding op een toets of tentamen zijn aan dit hoofdstuk enkele vragen toegevoegd.
De antwoorden op de vragen kun je vinden op www.science.uva.nl/amstel/SIM-PL

1. Is elke combinatorische schakeling een digitale schakeling? Zo nee, waarom niet?

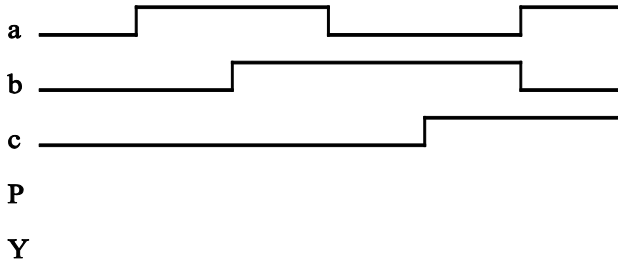
2. Ontwerp een schakeling voor de volgende formule: Y = (v AND (NOT w)) OR p.

3. Ontwerp een schakeling voor de volgende formule: Y = (v AND (NOT w)) OR p OR (NOT y) OR (NOT z). Maak hierbij gebruik van een OR-poort met vier ingangen.

4. Wat is de propagatietijd van de schakeling van vraag 3?

5. Geef de waarheidstabel van de schakeling weergegeven in figuur 1.14.

6. Maak het tijdvolgordediagram weergegeven in figuur 1.20 compleet. De signalen a, b en c worden aangeboden aan de schakeling weergegeven in figuur 1.5. Er hoeft geen rekening te worden gehouden met de propagatietijd van de poorten.



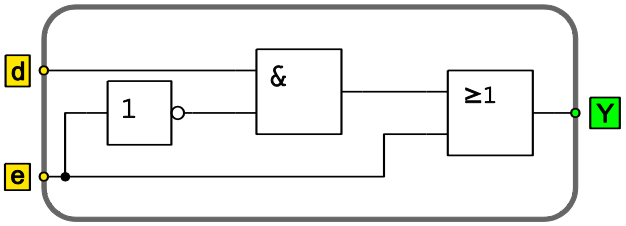
Figuur 1.20

7. 7a: Geef van figuur 1.21 de formule weer.

   7b: Geef van figuur 1.21 de waarheidstabel weer.

   7c: Met welke poort komt deze waarheidstabel overeen?

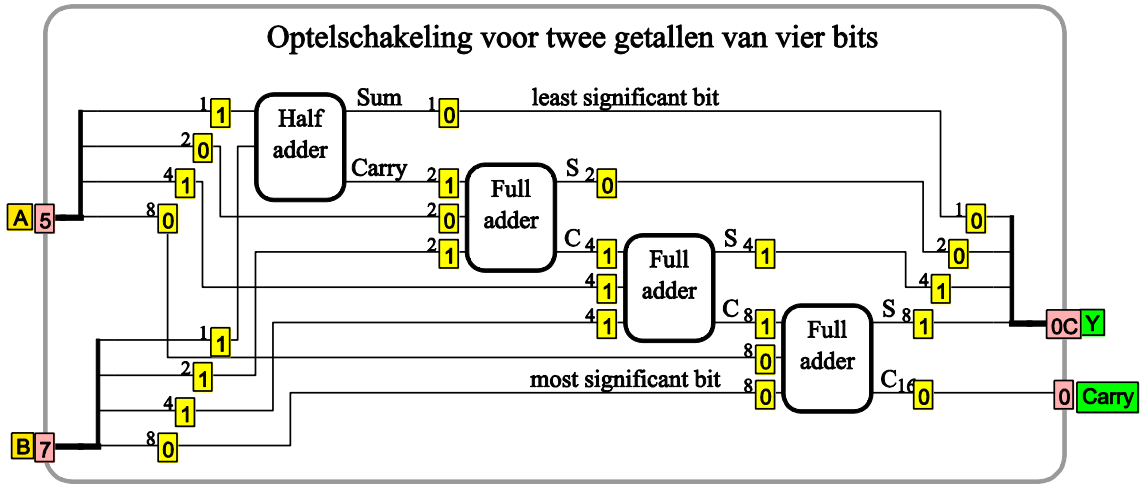
   7d: Wat is de formule die bij deze poort hoort?



Figuur 1.21

   7e: Open de worksheet: Opgave1.11.7.sim-pl-ws en gebruik de optie Truth table om je waarheidstabel te verifiëren.

De formule van opgave 7d is dus gelijk aan de formule van opgave 7a. *De inverter en de AND-poort zijn dus overbodig*. Bij het ontwerpen van schakelingen is het nodig om zo weinig mogelijk poorten te gebruiken. De Boole-algebra, één van de onderwerpen uit hoofdstuk 4, is ons daarbij behulpzaam. Opgave 7 is een voorbeeld van één der zogenaamde *absorptie wetten* uit deze algebra.



Hoofdstuk 2: Hoe rekent een computer?

2.1 Leerdoelen

Een belangrijke functie van een computer is rekenen. Berekeningen worden uitgevoerd door de Arithmetic Logic Unit, ALU genoemd. Met een ALU kunnen zowel rekenkundige operaties, zoals optellen en aftrekken als logische operaties, zoals NOT, AND, OR en XOR op twee operanden (getallen) worden uitgevoerd.

Aan het einde van dit hoofdstuk weet je:

- wat binaire getallen zijn;
- wat hexadecimale getallen zijn;
- wat een ‘half adder’ is;
- wat een ‘full adder’ is;
- hoe een schakeling werkt waarmee twee getallen van vier bits worden opgeteld;
- dat de *two’s complement code*wordt gebruikt voor de representatie van negatieve getallen;
- waarom deze representatie is gekozen;
- wat het datatype *Integer* inhoudt;
- wat het datatype *Unsigned Integer* inhoudt;
- wat een *ALU-overflow* is;
- wat het verschil is tussen een *carry* en een *overflow;*
- wat *sign extension* inhoudt;
- wat bitwise operatoren zijn;
- wat shift operatoren zijn.

Aan het einde van dit hoofdstuk kun je:

- binaire en hexadecimale getallen omzetten naar decimale getallen en andersom;
- een programma schrijven dat berekeningen uitvoert met een optelschakeling;
- van een positief getal een negatief getal maken en andersom;
- getallen optellen en aftrekken in two’s complement code;
- schakelingen ontwerpen om getallen op te tellen;
- een 16-bits getal met teken omzetten naar een 32-bits getal met teken;
- een bit uit een bitstring van waarde laten veranderen waarbij de overige bits onveranderd blijven.

2.2 Binaire representatie van getallen

Notatie decimaalgetal

Een andere notatie voor het decimale (tientallige) getal 123 is 100 + 20 + 3. Dit kunnen we schrijven als: $1*10^2 + 2*10^1 + 3 * 10^0$. Elke positie in een getal heeft een bepaald *gewicht*. Voor decimale getallen zijn dat de opeenvolgende machten van tien, dus: 1, 10, 100, 1000 enzovoort.

Notatie binair getal

Een computerbit kent maar twee toestanden: '0' of '1'. De structuur van tweewaardige of binaire getallen is geheel vergelijkbaar met die van het decimale stelsel. Als voorbeeld nemen we het binaire getal 111. De rechter 1 heeft het gewicht $2^0 = 1$; De middelste 1 heeft het gewicht $2^1 = 2$ en de linker 1 heeft het gewicht $2^2 = 4$. De decimale waarde van het binaire getal 111 is dus: 4 + 2 + 1 = 7. De notatie is vergelijkbaar met de decimale notatie, maar nu met 2 als grondtal[3)] in plaats van 10 (tien). Tweede voorbeeld om van een binair getal de decimale waarde te berekenen is: $101_{Bin} = 1*2^2 + 0*2^1 + 1 * 2^0 = 4 + 0 + 1 = 5$. Voor binaire getallen zijn de '*gewichten*' die bij elke bitpositie horen, de opeenvolgende machten van twee. Dus: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 enzovoort.

Decimale getal	Binaire getal 2^3 8	2^2 4	2^1 2	2^0 1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Tabel 2.1

Van binair naar decimaal

Voorbeeld: Wat is het binaire getal 11010111 in decimale code? Om dit getal om te zetten naar een decimale code beginnen we met de meest rechter bit. Op deze positie staat een '1'. Deze heeft het gewicht $2^0 = 1$. Vanaf deze positie ga je één bit naar links. Hier staat ook een '1' met gewicht $2^1 = 2$. Dit herhaal je tot en met de meest linker bit. De decimale waarde van het binaire getal 11010111 = 1 + 2 + 4 + 0 + 16 + 0 + 64 + 128 = 215.

In tabel 2.2 is dit nogmaals weergegeven. De leesrichting van de tabel is van rechts naar links.

waarde bit	1	1	0	1	0	1	1	1
vermenigvuldigen	*	*	*	*	*	*	*	*
gewicht bit	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
waarde * gewicht	128	64	0	16	0	4	2	1

Tabel 2.2

Van decimaal naar binair

Om een decimaal getal om te zetten naar een binair getal gebruiken we een eenvoudig algoritme dat uit de volgende stappen bestaat:

1. is het decimale getal even?
   a. ja, noteer het cijfer '0';
   b. nee, noteer het cijfer '1' en trek 1 van het decimale getal af;
2. deel het decimale getal door twee. Is het getal even?
   a. ja, noteer een '0' links van het eerder genoteerde cijfer;
   b. nee, noteer een '1' links van het eerder genoteerde cijfer en trek 1 van het getal af;
3. ga naar stap 2. Herhaal deze procedure zolang het overgebleven getal $\neq 0$ is;
4. is het overgebleven getal 0, dan is het decimale getal omgezet in een binaire code.

In tabel 2.3 wordt het decimale getal 89 omgezet in binaire code. De leesrichting van de tabel is van rechts naar links. Het resultaat is het binaire getal 1011001.

[3)] Het grondtal is het getal waarop het talstelsel is gebaseerd.

decimale waarde	1	2	5	11	22	44	89
even/oneven	oneven	even	oneven	oneven	even	even	oneven
binaire waarde	1011001	011001	11001	1001	001	01	1
overgebleven getal	0	2	4	10	22	44	88
deel door twee	klaar	1	2	5	11	22	44

Tabel 2.3

2.3 Hexadecimale talstelsel

Een getal in een computer is vaak 16- of 32-bits. Als voorbeeld een 16-bits getal: 1001110111100011. Een dergelijk getal is niet leesbaar. Om dit getal iets leesbaarder te maken worden deze bits in groepjes van 4 verdeeld: 1001 1101 1110 0011. Ieder groepje van 4 bits willen we met één cijfer[4)] noteren. Omgerekend naar het decimale stelsel wordt dit: 9 13 14 3. Het decimale stelsel voldoet niet omdat er slechts 10 cijfers zijn. De getallen 13 en 14 moeten cijfers worden en dus eigen symbolen krijgen.

Met vier bits zijn er $2^4 = 16$ verschillende symbolen nodig om alle waarden, van 0 tot en met 15, weer te kunnen geven.
Het hexadecimale of zestientallige stelsel heeft 16 cijfers: de cijfers 0 tot en met 9, aangevuld met de hoofdletters A, B, C, D, E en F. Het cijfer A is het symbool voor het decimale getal 10, B voor het decimale getal 11, C voor 12, D voor 13, E voor 14 en F is het symbool voor het cijfer met de decimale waarde 15.
Vier bits noteer je als één hexadecimaal cijfer. De binaire code: 1001 1101 1110 0011 gelijk aan de hexadecimale code 9 D E 3 ofwel 9DE3.

Decimaal	Binair	Hexa-decimaal
0	0000	0
1	0001	1
2	0010	2
…	……	….
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10
17	0001 0001	11
18	0001 0010	12
19	0001 0011	13
20	0001 0100	14
30	0001 1110	1E
32	0010 0000	20
40	0010 1000	28
255	1111 1111	FF
256	1 0000 0000	100

Tabel 2.4

Hexadecimale getallen
Hexadecimale getallen zijn een verkorte schrijfwijze van binaire getallen. Getallen noteren in hexadecimale notatie is handig om binaire getallen van 16, 32 of 64 bits verkort en in een meer leesbare notatie weer te geven.

[4)] Verschil tussen cijfer en getal: Het decimale stelsel bevat de cijfers 0 tot en met 9. Het getal 13 is samengesteld uit twee cijfers: 1 en 3.

Talstelsels
Wij zijn gewend aan het tientallige stelsel, maar zonder vermelding van het talstelsel weten we niet zeker wat er met bijvoorbeeld 1101 wordt bedoeld.
Het binaire 1101 = $1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 8 + 4 + 0 + 1 = 13$ in het decimale stelsel.
Het decimale 1101 = $1*10^3 + 1*10^2 + 0*10^1 + 1*10^0 = 1000 + 100 + 0 + 1 = 1101$ in het decimale stelsel.
Het hexadecimale 1101 = $1*16^3 + 1*16^2 + 0* 16^1 + 16^0 = 4096 + 256 + 0 + 1 = 4353$ in het decimale stelsel.
In het verleden kwam het ook voor dat rijen bits in groepen van drie werden verdeeld. Er zijn dan acht cijfers nodig. Dit wordt het octale stelsel genoemd.
Het octale 1101 = $1*8^3 + 1*8^2 + 0*8^1 + 1*8^0 = 512 + 64 + 0 + 1 = 577$ in het decimale stelsel.
Om aan te geven dat het hexadecimale getal 1101 en niet het binaire of decimale getal 1101 wordt bedoeld, noteren we 1101 als 1101_{Hex} of als 0x1101.

2.4 Optellen

De manier van optellen in het binaire stelsel verschilt niet wezenlijk van die in het decimale stelsel.

Voorbeeld
Op de basisschool hebben we de volgende methode geleerd om de getallen 39 en 25 op te tellen: eerst tellen we 9 en 5 op met als resultaat 14. Van dit getal schrijven we de 4 op. De 1 tellen we op bij de 3 van 39 en de 2 van 25. Het resultaat is 6. Dit cijfer noteren we links van de 4. Resultaat: 64.

Tweede voorbeeld
Het tweede voorbeeld is het optellen van twee getallen van één bit. In tabel 2.5 zijn de vier mogelijke combinaties weergegeven. Bij de meest rechtse optelling geldt: 1 + 1 = 10 (twee). Het linker cijfer heeft het 'gewicht' 2 en moet dus links van de 0 worden opgeschreven. Dit levert als resultaat een getal van *twee bits* op. Men kan ook stellen: 1 + 1 = 0 én 1 onthouden. De optelling levert dus twee resultaten op: een bit die de som geeft (hier 0) én een bit die onthouden moet worden en links van de som wordt geschreven. Deze laatste wordt de transportbit of 'carry' genoemd.

0	0	1	1
0 +	1 +	0 +	1 +
0	1	1	10

Tabel 2.5

Derde voorbeeld
Het derde voorbeeld is het optellen van twee getallen van vier bits A en B. In de tabellen 2.6 en 2.7 is de methode weergegeven volgens welke de hardware van een computer twee getallen van vier bits optelt. Het resultaat is een 5-bits getal. Het symbool '↑' betekent: 1 onthouden en optellen bij de bits of cijfers ter linker zijde. Bij deze optelling komt het drie keer voor dat er niet twee, maar drie bits moeten worden opgeteld; namelijk een bit van getal A en de overeenkomende bit van getal B en de carry van de optelling van de bit ter rechter zijde.

Getal	Decimaal	Binair
A	13	1101
B	7	0111
A + B	20	10100
	↑	↑↑↑↑

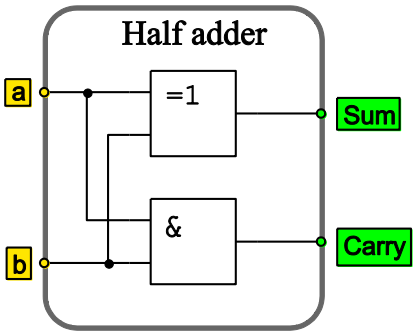
Tabel 2.6

Getal	Decimaal	Binair
transport	1	1111
A	13	1101
B	7	0111
A + B	20	10100

Tabel 2.7

Hardwarecomponenten om getallen op te tellen

Uit het derde voorbeeld blijkt dat voor het optellen van twee 4-bits binaire getallen aan hardware nodig is:



Figuur 2.1

- *één optelschakeling voor de minst significante bit* (bit met het laagste gewicht, dus meest rechter bit) van beide getallen. Dit betekent een optelschakeling voor twee getallen van 1 bit. Een dergelijke schakeling wordt een **'half adder'** genoemd. De schakeling heeft twee uitgangen: een 'Sum-' en een 'Carry-uitgang' (transportbit-uitgang). In figuur 2.1 is de schakelingweergegeven;

- *drie optelschakelingen voor de overige bits.* Dit betekent drie keer een optelschakeling voor drie getallen van 1 bit. Die drie getallen van 1 bit zijn: een bit van getal A en de overeenkomende bit van getal B en de transportbit of carry van de bit ter rechter zijde. Een dergelijke schakeling wordt een **'full adder'** genoemd. Ook deze schakeling heeft een 'Sum-' en een 'Carry-uitgang'.

2.5 Opdrachten met gehele getallen zonder teken

Decimaal	Binair
	11001100
	01011010
	100000000
19	
96	
4097	

Tabel 2.8

Opdracht 1: Van binair naar decimaal en vice versa

Vul tabel 2.8 in.

Opdracht 2: Van hexadecimaal naar decimaal en vice versa

Als we van het hexadecimale getal 1B3 de decimale waarde willen berekenen schrijven we ieder cijfer van dit getal als een macht van 16.
$1B3_{Hex} = 1*16^2$, $B*16^1$, $3 * 16^0$.
De decimale waarde van $1B3_{Hex} = 1*16^2 + 11*16^1 + 3 * 16^0 = 256 + 176 + 3 = 436_{Dec}$.

In tabel 2.9 is een getal weergegeven in decimale of in binaire of in hexadecimale code. Als een getal gegeven is in decimale code, vul dan de bijbehorende binaire en hexadecimale waarden in. Doe hetzelfde als een getal in één van de andere representaties is weergegeven.

Deci-maal	Binair	Hexa-decimaal
		A
11		
	1100	
		D
31		
		12
	0100 0000	
		1A
100		
		32
		80
254		
		111

Tabel 2.9

Opdracht 3: Optellen van twee 5-bits getallen

Opdracht: Tel het getal A op bij het getal B van de tabel 2.10. Vul ook de waarde van de Carry in. Controleer je antwoord door de getallen naar decimale code om te zetten en daarna op te tellen. Doe hetzelfde met de getallen C en D.

Getal	Decimaal	Binair
Carry	…	……
A	...	11110
B	…	01001
A + B	...	…….
Getal	Decimaal	Binair
Carry	…	……
C	...	11101
D	…	01110
C + D	…	…….

Tabel 2.10

Opdracht 4: Half adder

In figuur 2.1 is het schema van een *half adder* weergegeven.
Opdracht a: Geef de formules voor de 'sum' en de 'carry'.
Sum = …….. Carry = …….

Opdracht b: Vul tabel 2.11 in.
Komt tabel 2.11 overeen met tabel 2.5?
Antwoord: …..

a	b	Carry Gewicht = 2^1 =2	Sum Gewicht = 2^0 =1
0	0		
0	1		
1	0		
1	1		

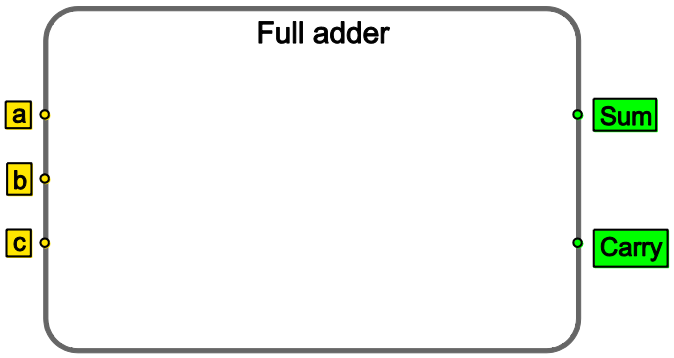
Tabel 2.11

Opdracht 5: Full adder

Opdracht a: Vul de tabel van de 'full adder' in.

a	b	c	Carry	Sum
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Tabel 2.12



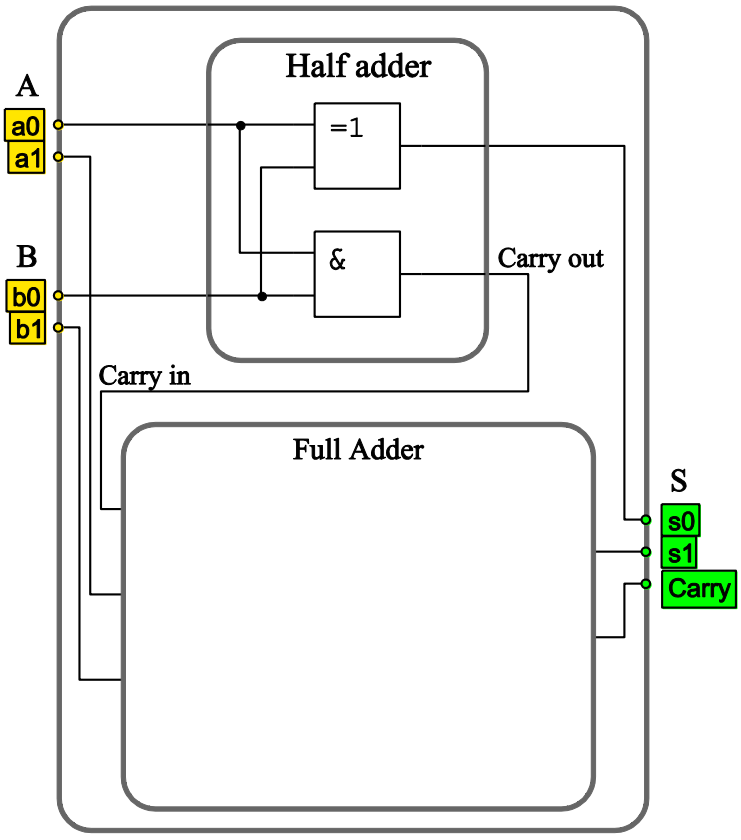
Figuur 2.2

Opdracht b: Ga voor iedere regel in tabel 2.12 na of onderstaande formules correct zijn.
Sum = (a XOR b) XOR c.
1e formule carry: *Carry = ((a XOR b) AND c) OR (a AND b).*
2e formule carry: *Carry = (a AND b) OR (a AND c) OR (b AND c).*

Opdracht c: Geef een ontwerp van een 'full adder'. Maak hierbij gebruik van de 1e formule voor de carry: *Carry = ((a XOR b) AND c) OR (a AND b).*
Geef je ontwerp weer in figuur 2.2.
Stel dat alle poorten een propagatietijd van 1 ns hebben. Wat is de grootste t_{pd} van deze full adder? Antwoord: ……

Opdracht 6: Optellen van twee 2-bits getallen

In figuur 2.3 is een schakeling weergegeven bestaande uit een half adder en een full adder.
Met deze schakeling kan het 2-bits getal A, bestaande uit de bits a0 en a1, worden opgeteld bij het getal B dat uit b0 en b1 bestaat. De uitkomst van de optelling S bestaat uit drie bits: s0, s1 en Carry. s0 is de sum-uitgang van de half adder en s1 is de sum-uitgang van de full adder.
De decimale waarde die bij deze code hoort is:
$S = C_{arry}.2^2 + s_1.2^1 + s_0.2^0$.
De carry van de half adder, *Carry out* genoemd, is verbonden met één van de ingangen van de full adder. Deze ingang is *Carry in* genoemd.



Figuur 2.3

Opdracht: Geef een tweede ontwerp van een 'full adder'. Maak hierbij gebruik van de 2e formule voor de carry: *Carry = (a AND b) OR (a AND c) OR (b AND c).*
Geef je ontwerp weer in figuur 2.3.

Wat is de t_{pd} van deze full adder-schakeling? Antwoord: ……
Wat is de t_{pd} van de hele 2-bits optelschakeling? Antwoord: ……

Opdracht 7: SIM-PL-opdracht

Implementeer één van de ontworpen 'full adders' met de SIM-PL-Editor en sla deze op in de folder 'H2Rekenen' onder de naam MijnFullAdder. Maak met de SIM-PL-Executer een nieuw werkblad en test je ontwerp.

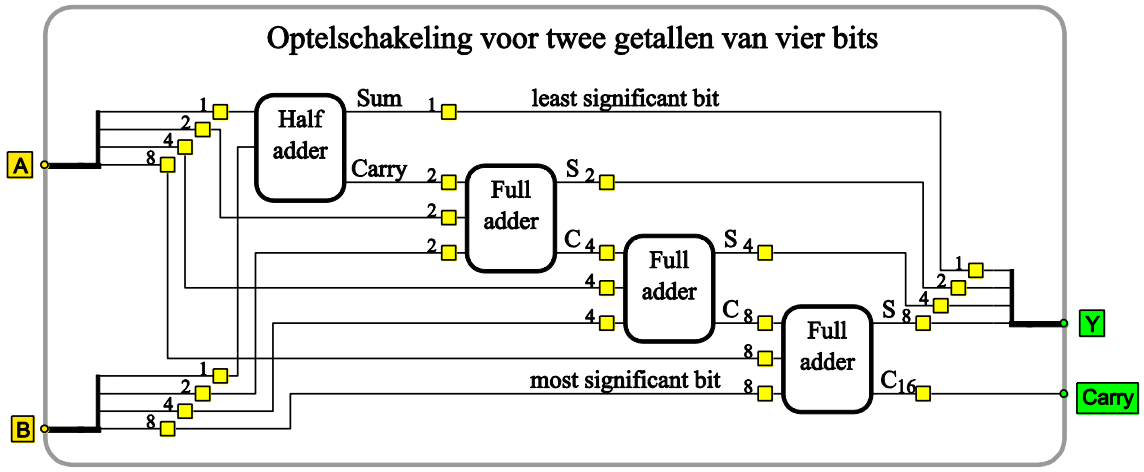
Opdracht 8: Optelschakeling voor twee getallen van twee bits

Implementeer een schakeling waarmee twee 2-bits getallen kunnen worden opgeteld. Maak hierbij gebruik van de component HalfAdder.sim-pl en de zelf ontworpen component MijnFullAdder. Maak met de SIM-PL-Executer een nieuw werkblad en test je ontwerp.

Opdracht 9: Optelschakeling voor twee 4-bits getallen

Figuur 2.4 geeft een optelschakeling weer voor het optellen van twee 4-bitsgetallen A en B. Deze 4-bits getallen worden ontrafeld in afzonderlijke bits waarvan het gewicht is weergegeven. Ook op alle in- en uitgangen van de adders is het gewicht van iedere bit weergegeven. De signalen op de ingangen van een adder moeten hetzelfde gewicht hebben. De sum-uitgang heeft ook dit gewicht en de carry-uitgang heeft een twee keer zo groot gewicht. Bij uitgang Y worden de afzonderlijke bits weer samengevoegd tot een 4-bits getal.

SIM-PL-opdracht: Laad de component 4bitOpteller.sim-pl-ws uit de folder Components → H2Rekenen in de Executer.



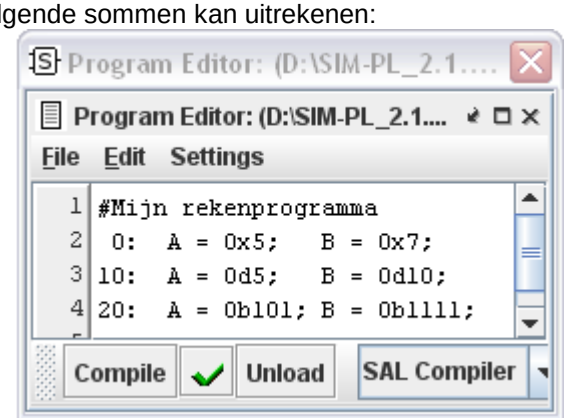
Figuur 2.4

Notaties van binaire, hexadecimale en decimale getallen in SIM-PL
Hexadecimale getallen worden in programmeeromgevingen als Java en C weergegeven door de karakters 0x voor het getal te plaatsen. Bijv. 0x100.
Zowel in dit boek als in de SIM-PL-software worden:

- binaire getallen aangeduid met het prefix (voorvoegsel) 0b, bijvoorbeeld: 0b11001110;
- hexadecimale getallen aangeduid met het prefix 0x, bijvoorbeeld: 0x3DA5;
- decimale getallen aangeduid met het prefix 0d, bijvoorbeeld: 0d1024.

Gebruik het window ‘Program Editor’ om het programma ‘MijnRekenprogramma.sal’ te laden en uit te breiden zodat het achtereenvolgens de volgende sommen kan uitrekenen:
0x5 + 0x7;
0d5 + 0d10;
0b1101 + 0b1111;
0xA + 0xE;
0d15 + 0d15

Een deel van het programma is hiernaast weergegeven. De linker kolom geeft het tijdstip aan dat de instructie wordt uitgevoerd. De volgende instructie moet steeds 10 tijdstippen later worden uitgevoerd. Compileer het programma, laad het programma en voer het uit. Gebruik de optie: Settings → Number Format om van talstelsel te wisselen.



2.6 Gehele getallen met teken: De two’s complement representatie

Om vragen als: “Wat gebeurt er als twee getallen van elkaar worden afgetrokken en het antwoord is negatief?” en “Hoe noteer ik een minteken met enen en nullen”, te kunnen beantwoorden is een getalrepresentatie nodig waarmee zowel positieve als negatieve gehele getallen kunnen worden voorgesteld. In het decimale stelsel zetten we het –teken voor het getal. Het ligt voor de hand een extra bit aan de code toe te voegen. Deze tekenbit geeft aan of het getal positief dan wel negatief is. Uit dit hoofdstuk zal blijken dat er een andere oplossing is die veel minder poorten vereist en die daardoor berekeningen veel sneller uitvoert.

Nogmaals de binaire code

Een algemene voorstelling van de decimale waarde van een 4-bits binair getal A = $a_3a_2a_1a_0$ is: $a_3.2^3 + a_2.2^2 + a_1.2^1 + a_0.2^0$.
Bij het getal 1010 in binaire code zijn a_3 en a_1 ‘1’ en a_2 en a_0 ‘0’. De bijbehorende decimale waarde is: $1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10$. Het domein van een 4-bits binair getal ligt tussen 0000 en 1111. Het bereik van de bijbehorende decimale getallen loopt van 0 (0000) naar 15 (1111).

Two’s complement code

Bij de *two’s complement* representatie wordt aan de meest linker bit, hier a_3, een negatieve waarde toegekend. Aan de andere bits wordt een positieve waarde toegekend. De decimale waarde van een 4-bits getal in two's complement code is: $-a_3.2^3 + a_2.2^2 + a_1.2^1 + a_0.2^0$.
Bij het getal 1010 in two’s complement code zijn a_3 en a_1 ‘1’. De bijbehorende decimale waarde is $-2^3 + 2^1 = -6$. Het bereik loopt van -8 (1000) naar +7 (0111). Het bereik aan de +kant is gehalveerd. Met vier bits kunnen nu eenmaal niet meer dan 16 verschillende decimale getallen worden weergegeven.

Tekenbit
Als het meest linker bit *‘1’* is, is het getal *negatief* want: 2^3 is groter dan $2^2 + 2^1 + 2^0$. Het getal in two’s complement code 1111 levert de decimale waarde -1 op omdat -8 + 4 + 2 + 1 = -1.
Ga na dat ook het 6-bits getal getal 111111 de waarde -1 oplevert.
Het msb (meest linker bit; most significant bit) geeft aan of het getal positief of negatief is en fungeert dus als tekenbit.

Van positief getal naar negatief getal

Hoe maak je van een positief decimaal getal een negatief decimaal getal en omgekeerd?
Een negatief getal in de two’s complement notatie wordt geconstrueerd door:

1. de modulus (is getal zonder teken) van het getal in binaire code te schrijven;

getal	*0110*	*6*
complement	1001	-7
+1	1	
getal	1010	-6

Tabel 2.13

2. alle bits te inverteren (complement nemen; NOT-operatie uitvoeren op alle bits);
3. bij het getal één op te tellen.

getal	*1110*	*-2*
complement	0001	1
+1	1	
getal	0010	2

Tabel 2.14

Een eventueel optredende carry van de hoogste bit wordt genegeerd. De andere carry-bits worden wel gebruikt.
In tabel 2.13 is weergegeven hoe je van +6, -6 maakt.
Om van een negatief getal een positief getal te maken pas je dezelfde constructie toe. In tabel 2.14 is weergegeven hoe je van -2, +2 maakt.

Een speciaal geval: het getal 0.
In tabel 2.15 is weergegeven wat er gebeurt als we deze constructie uitvoeren op het getal 0.
Het resultaat is weer 0. Weliswaar is het carry-bit '1', maar het carry-bit heeft *geen betekenis* in de two's complement-code.

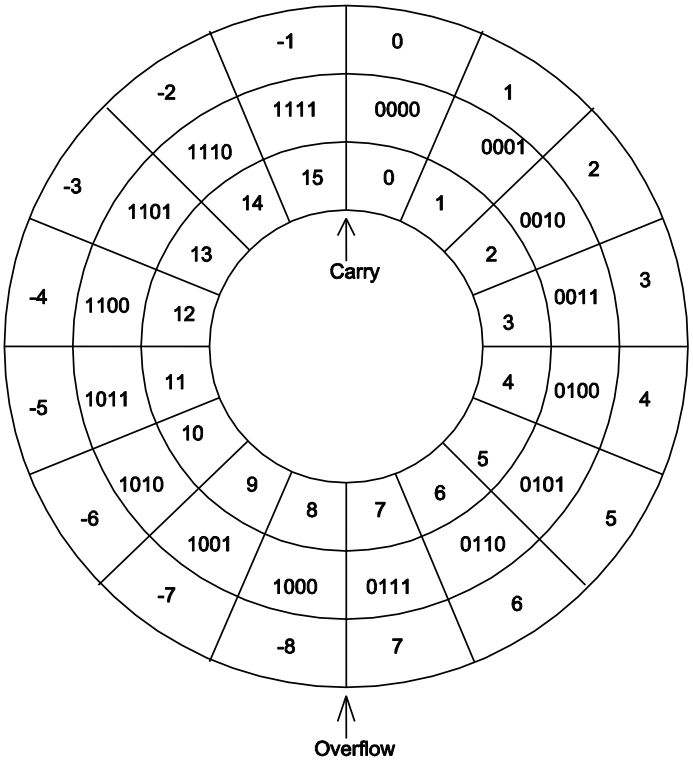
getal	*0000*	*0*
complement	1111	-1
+1	1	
getal	10000	0

Tabel 2.15

Bits are just bits!
Bits zijn niet meer dan 'eentjes' en 'nulletjes'. Bits krijgen pas betekenis na afspraken. Zo kan de bitstring: 11110011 het getal 243 betekenen volgens de binaire code, maar ook -13 volgens de two's complement code. Deze keten van enen en nullen kan echter ook een machinetaal-instructie voorstellen of het teken: ≤ volgens de extented ASCII-code (zie paragraaf 2.11).

Getallencirkel (4-bits)

Alle gehele getallen die met vier bits kunnen worden gerepresenteerd, zijn weergegeven in figuur 2.5 door middel van een getallencirkel. In de middelste ring is een 4-bits code weergegeven. Beschouwen we deze code als *binaire code,* dan geeft de *binnenste ring* de bijbehorende decimale waarden van 0 t/m 15 weer. Beschouwen we de code van de middelste ring als *two's complement code,* dan geeft de *buitenste ring* de bijbehorende decimale waarden van -8 t/m +7 weer.
Je kunt de cirkel gebruiken om twee getallen op te tellen of af te trekken. Optellen van twee getallen gebeurt met de wijzers van de klok mee. Voorbeeld 4 + 3. Zoek het getal 4 en ga 3 vakjes verder met de wijzers van de klok mee. Je komt dan op 7 uit.
Tweede voorbeeld: 4 + 6. In de binnenste ring kom je op 10 uit. In de buitenste ring bestaat 10 niet. Het getallenbereik wordt overschreden als het getal groter is dan 7. Bij deze optelling hoort de waarde -6.



Figuur 2.5

Domein, bereik, carry en overflow van een 4-bits code
Uit figuur 2.5 blijkt dat er twee interpretaties van de code in de middelste cirkel mogelijk zijn:

1. *als getallen zonder plus- of minteken (binaire code):* bereik 0 … 15. Wordt dit bereik overschreden, dan treedt *een carry* op. Dit is het geval als het getal >15 wordt. Voorbeeld: 15 + 3 = 2 + een carry. De carry heeft het gewicht 16. Dus 15 + 3 = 2 + 16 = 18.
2. *als getallen met plus- of minteken (two's complement-code):* bereik: -8 … +7. Wordt de grens van dit bereik in één van beide richtingen overschreden, dan treedt een

zogenaamde *overflow* op. Voorbeeld: Wordt +7 bij + 5 opgeteld, dan wordt de grens van het bereik (+7) overschreden en treedt een zogenaamde *overflow* op. Deze overflow heeft het gewicht 16. Je kunt het resultaat interpreteren als: 7 + 5 = -4 + 16 = 12.

Two's complement code en datatype
Variabelen in een programmeertaal zijn altijd van een bepaald *datatype*. Bijvoorbeeld:
int a;
unsigned int b;
De variabele a is van het type Int. Int is een afkorting van integer (geheel getal met teken). Om een integer op te slaan wordt de two's complement code gebruikt.
De variabele b is van het type unsigned int (geheel getal zonder teken). Een unsigned integer wordt gerepresenteerd in de binaire code.

Algoritme om twee getallen af te trekken

Twee bladzijden terug is een methode uitgelegd hoe je van een positief getal een negatief getal maakt. Deze methode is:

1. de modulus (is getal zonder teken) van het getal in binaire code schrijven;
2. alle bits inverteren (is NOT-operatie uitvoeren op alle bits);
3. bij het getal één optellen.

Deze methode gebruiken we om twee getallen af te trekken. De aftreksom: A - B wordt teruggebracht tot een optelsom door B om te zetten in een negatief getal. Als voorbeeld nemen we de som: 4 – 2. Het getal 2 wordt omgezet naar -2 en opgeteld bij het getal 4. Dus 4 – 2 = 4 + (-2) = 4 + (NOT(2) + 1) = 4 + NOT 2 +1 = 4 – 3 + 1 = 2.

Aftrekken van getal B van getal A geschiedt door getal B om te zetten naar –B en deze –B op te tellen bij A.
In formulevorm: A - B = A + (-B) = A + (NOT(B) + 1).

In tabel 2.16 zijn twee voorbeelden weergegeven waarbij van een 4-bits getal een ander 4-bits getal wordt afgetrokken. De uitkomst van het eerste voorbeeld is 10010. Aan de hoogste bit (waarde van de carry van de hoogste bit) kennen we geen betekenis toe. Dus het resultaat is: 0010.

4 - 2 =	2
0100 - 0010 =	0100 + 1101 + 1 = 10010
4 - (-2) =	6
0100 - 1110 =	0100 + 0001 +1 = 0110

Tabel 2.16

2.7 Sign extension

Gehele getallen met teken kunnen in een 8-bits formaat (byte), 16-bits formaat (word), 32-bits formaat (doubleword) en 64-bits formaat zijn opgeslagen. Bij veel processoren zijn constante getallen in het Instruction Memory opgeslagen als 16-bits getal. De processorregisters zijn meestal 32 bits. Binnen een dergelijk systeem moet een 16-bits getal 'vertaald' worden naar een 32-bits getal. Dit gebeurt d.m.v. een zgn. sign extender. Is het getal positief, dus de 'most significant bit' is 0, dan worden aan het getal 16 nullen aan de linkerkant toegevoegd. Is het getal negatief, de 'most significant bit' is dus 1, dan worden aan de linkerkant 16 enen toegevoegd.
De sign extender converteert bijvoorbeeld het binaire getal 0000 0000 0000 0011 in 0000 0000 0000 0000 0000 0000 0000 0011. De decimale waarde van beide getallen is uiteraard dezelfde namelijk 3.

Laten we volgens de methode van paragraaf 2.6 de 16-bits code voor -3 construeren. Zie tabel 2.17.

getal	*0000 0000 0000 0011*	*3*
complement	1111 1111 1111 1100	-4
+1	1	
getal	1111 1111 1111 1101	-3

Tabel 2.17

De sign extender verandert dit 16-bits binaire getal 1111 1111 1111 1101 in het 32-bits binaire getal 1111 1111 1111 1111 1111 1111 1111 1101. De decimale waarde van beide getallen is -3; in hexadecimale code: FFFD resp. FFFFFFFD.

2.8 Operatoren voor bitmanipulatie

Er zijn twee typen operatoren voor bitmanipulatie:

- Bitwise operatoren
- Shift operatoren

Bitwise operatoren
Logische operatoren uitgevoerd op meerdere bits worden bitwise operatoren genoemd. Bij bitwise operaties staat elke bit op zichzelf. Er is geen *carry-bit* zoals we die bij een rekenkundige operatie als optellen zijn tegengekomen. Enkele bitwise operatoren zijn NOT, AND, OR en XOR.
In tabel 2.18 zijn bitwise operaties uitgevoerd op de 4-bits getallen 5 en 3.

Operator	bitwise NOT
5	0101
NOT 5 = 10	1010
Operator	**bitwise AND**
5	0101
3	0011
5 AND 3 = 1	0001
Operator	**bitwise OR**
5	0101
3	0011
5 OR 3 = 7	0111
Operator	**bitwise XOR**
5	0101
3	0011
5 XOR 3 = 6	0110

Tabel 2.18

Shift operatoren
Shift operatoren verschuiven bits. De operator << (Shift Left; SHL) schuift alle bits naar links en de operator >> (Shift Right; SHR) schuift alle bits naar rechts.
Voorbeeld: Willen we bijv. het getal 13_{DEC} drie bitposities naar links schuiven dan noteren we dat als: 13 << 3. Aan de rechter kant worden er nullen ingeschoven.
Ga na dat 13 << 3 gelijk is aan 13 * 2^3 = 104.
Bij een operatie als a >> 3 schuiven alle bits van getal a drie posities naar rechts en worden aan de linker kant nullen ingeschoven. De meest rechtse bits verdwijnen hierbij.
Zo is 13 >> 3 is gelijk aan 13 / 2^3 = 1. In tabel 2.19 is dit weergegeven

Operator	<< (SHL)
13	00001101
13 SHL3	01101000
Operator	**>> (SHR)**
13	00001101
13 SHR 3	00000001

Tabel 2.19

Bitmanipulaties: Set bit, reset bit en toggle a single bit.
Set bit: Stel dat een machine een 8-bits output port heeft met bits A7.. A0. Iedere outputbit van deze machine is met een afzonderlijk apparaat verbonden. Op welke manier kunnen we nu één van de apparaten aan- of uitzetten zonder de toestand van de andere apparaten te beïnvloeden? Als voorbeeld nemen we het geval dat het apparaat aangesloten op A6, is uitgeschakeld. We kunnen A6 aanzetten door eerst een zogenaamd masker te maken met allemaal nullen op bit 6 na. Dit masker verkrijgen we m.b.v. de SHL operator. Hierna wordt de OR-operator gebruikt om van de outputcode alleen A6 te veranderen. Voor het aanzetten van een willekeurige bit geldt de volgende procedure:
Set_bit (bitnr) = (1 SHL bitnr) OR code.

Set_bit_A6	
mask = 1 SHL 6	0**1**000000
outputcode	1**0**100010
outputcode OR mask	1**1**100010

Tabel 2.20

Toggle bit: Een andere manier om de toestand van een bit te veranderen is om gebruik te maken van de XOR-operator. In tabel 2.21 is weergegeven dat m.b.v. de XOR-operator alleen A2 van toestand is veranderd. Passen we deze operatie twee keer toe dan ontstaat de oorspronkelijke toestand weer.

Toggle_bit_A2	
mask = 1 SHL 2	00000**1**00
outputcode	10100**0**10
outputcode XOR mask	10100**1**10
outputcode XOR mask	10100**0**10

Tabel 2.21

Reset bit: Zie paragraaf 2.9.

2.9 Vragen en opgaven met gehele getallen met teken

Opdracht 1: 4-bits getallen

1. Construeer het getal -3 vanuit het getal +3. Vul hiertoe tabel 2.22 verder in.
2. Construeer op dezelfde wijze het getal +6 vanuit getal -6 en vul tabel 2.23 in.
3. Ga na dat voor 4-bits getallen het grootste positieve getal 7 (0111) en het grootste negatieve getal -8 (1000) is.

getal	*0011*	*3*
complement	……	…..
+1	1	
getal	……	-3

Tabel 2.22

4. Wat is het grootste positieve getal en het grootste negatieve getal bij 8-bits getallen?
   Antwoord: Het grootste positieve getal is: …… en het grootste negatieve getal is: ……

getal	*1010*	*-6*
complement	…….	……
+1	…….	
getal	…….	6

Tabel 2.23

Opdracht 2: Aftrekken van twee 4-bits getallen

Bereken 7 - 6 en -4 - 5.
Vul hiertoe tabel 2.24 verder in.
Opmerking: -4 - 5 = -9. -9 valt echter buiten het bereik van 4-bits getallen in two's complement code.
Er treedt hier een overflow op. Deze heeft het gewicht 16. Dus -4 -5 = -9 + 16 = 7.

7 - 6 =	1
0111 - 0110 =	……
-4 - 5 =	7
1100 - 0101 =	……

Tabel 2.24

Hexadecimale getallen en two's complement code

Hexadecimale code is niet meer dan een verkorte schrijfwijze van ketens van enen en nullen en dus kunnen zowel de binaire code als de two's complement code als hexadecimale getallen worden weergegeven.
Als voorbeeld berekenen we de uitkomst van 100 – 103 = -3 en beantwoorden de vraag: *Wat is de hexadecimale code van het getal -3?*
Tabel 2.25 laat zien hoe het antwoord FFFD tot stand is gekomen.

100 - 103 =	-3
0064 - 0067 =	0064 + FF98 + 1 = FFFD

Tabel 2.25

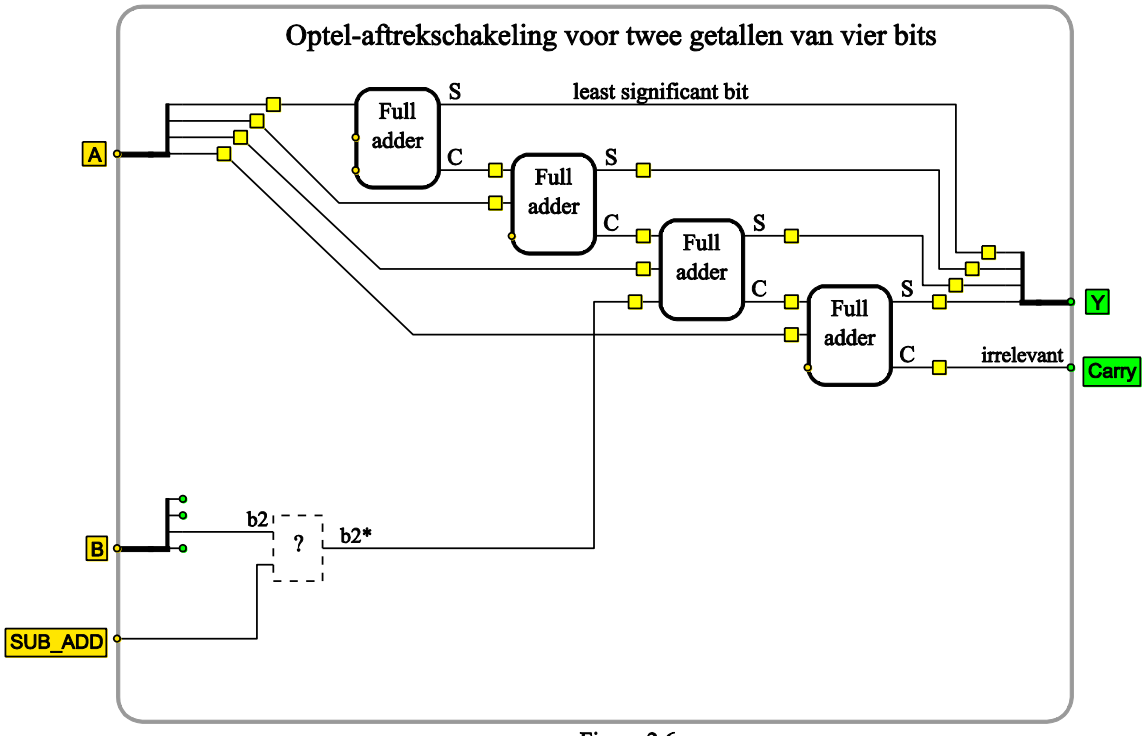
Opdracht 3: Aftrekken van twee 4-bits getallen in hexadecimale code

Ga na dat als we van het getal 0067_{Hex} alle bits inverteren het resultaat $FF98_{Hex}$ is en maak de som: 7 - 11 voor een 16-bits machine. Vul het resultaat in tabel 2.26 in.

7 - 11 =	-4
0007 – 000B =	……

Tabel 2.26

Opdracht 4: Uitbreiden optelschakeling tot optel-aftrekschakeling



Figuur 2.6

In figuur 2.6 is een deel van het ontwerp van een schakeling weergegeven die twee getallen van 4 bits moet kunnen optellen of aftrekken. De volgende veranderingen zijn aangebracht vergeleken met de optelschakeling van figuur 2.4:

1. op bit b2 na zijn verbindingen met ingang B gewist;
2. er is een ingang SUB_ADD toegevoegd. Een betere notatie voor deze ingang is: $\overline{\text{SUB/ADD}}$[5]. Voor deze ingang geldt: als $\overline{\text{SUB/ADD}}$= '0', dan worden de getallen op de ingangen A en B bij elkaar opgeteld. Als $\overline{\text{SUB/ADD}}$= '1', dan wordt het getal B van het getal A afgetrokken;

$\overline{\text{SUB/ADD}}$	b2	***b2****
0	0	0
0	1	1
1	0	1
1	1	0

Tabel 2.27

3. bit b2 is samen met de $\overline{\text{SUB/ADD}}$-ingang aangesloten op een onbekende component. De uitgang van deze component, b2*, wordt aangesloten op de 'oude' plaats van b2. Voor de onbekende component geldt tabel 2.27. Bij optellen wordt b2 doorgelaten en bij aftrekken wordt b2 geïnverteerd.
4. de half adder is vervangen door een full adder.

Vragen:

1. Met welke poort kunnen we tabel 2.27 realiseren? Antwoord: …….
2. Als $\overline{\text{SUB/ADD}}$ '1' is, dus bij de operatie aftrekken, moeten we 1 bij het resultaat optellen. Dit mag niet gebeuren als $\overline{\text{SUB/ADD}}$ '0' is. Hoe kunnen we dit realiseren? Antwoord: ……

Opdracht 5: 4-bits optel/aftrekschakeling

Ontwerp, bouw en test met SIM-PL een schakeling waarmee twee 4-bits getallen in de two's complement-representatie zowel opgeteld als afgetrokken kunnen worden, afhankelijk van de code op de SUB_ADD-ingang. Gebruik hiervoor als basis de schakeling van figuur 2.6. Deze schakeling vind je onder de naam 4bitOptelAftrekSchakeling.simpl in de folder H2Rekenen. In deze folder staan ook andere benodigde componenten.

Test met programma

Start de executer, zodra je ontwerp gereed is en maak een nieuwe worksheet. Test je ontwerp m.b.v. de file: TwosComplement.sal. Laad deze in de Program Editor. Compileer en run deze code en test de schakeling. Gebruik ook de optie Settings → Number Format → Decimal om de weergave van negatieve getallen te bekijken.
Geef het volledige schema van de schakeling ook weer in figuur 2.6.
wordt toegepast.

Two's complement code is efficiënt

Waarom is er voor de weergave van negatieve getallen niet gekozen voor de binaire code met een extra tekenbit, de zogenaamde sign-magnitude code? Deze code is voor de mens veel eenvoudiger te begrijpen. Het antwoord op deze vraag volgt uit de oplossing van de laatste opdracht. Aftrekken m.b.v. de two's complement code vergt weinig extra hardware t.o.v. de optelschakeling. Berekeningen kosten dus weinig tijd. Bij de sign-magnitude code is dit niet zo.

Opdracht 6: Bitwise operatoren op 4-bits getallen.

- Wat is NOT 9? Antwoord: …..
- Wat is 9 AND 5? Antwoord: ….
- Wat is 9 OR 5? Antwoord: …..
- Wat is 9 XOR 5? Antwoord: ….

Opdracht 7: Reset_bit (bitnr)

Kies een 8-bits code A7.. A0 waarbij bit A4 '1' is. Zorg ervoor dat bit A4 '0' wordt en dat hierbij

[5] De streep boven ADD van het woord $\overline{\text{SUB/ADD}}$ is een *notatie* om aan te geven dat de opdracht ADD uitgevoerd wordt door een '0'. Boven SUB staat geen streep. Deze opdracht wordt uitgevoerd door een '1'. Deze manier van noteren maakt de relatie duidelijk tussen het signaal op de $\overline{\text{SUB/ADD}}$-ingang en de operatie die wordt uitgevoerd. De streep boven ADD betekent dus niet NOT ADD. In SIM-PL is deze notatie niet mogelijk.

alle andere bits ongewijzigd blijven. Gebruik hiervoor de operatoren SHL, NOT en AND.
Oplossing:
-
-
-

Opdracht 8: Sign extension.

Zet de 4-bits codes 0110 en 1001 om in de bijbehorende 8-bits codes. Geef ook de decimale waarden weer.
Antwoord: 0110 → ……. = decimaal: …. .
Antwoord: 1001 → ……. = decimaal: …. .

2.10 ASCII en UNICODE

ASCII

ASCII is een afkorting van American Standard Code for Information Interchange. Een ASCII-teken bestaat uit zeven bits. Sommige van de codes uit het begin van tabel 2.28 (1 tot 32) worden gebruikt voor besturingstekens. Bijvoorbeeld TAB, CR (Carriage Return) en LF (Line Feed). Anderen codes, zoals ACK (acknowledge), werden vroeger gebruikt voor datatransmissie tussen terminals.

American Standard Code for Information Interchange (ASCII)											
Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	32	20		64	40	@	96	60	`
1	1	SOH	33	21	**!**	65	41	**A**	97	61	**a**
2	2	STX	34	22	**"**	66	42	**B**	98	62	**b**
3	3	ETX	35	23	**#**	67	43	**C**	99	63	**c**
4	4	EOT	36	24	**$**	68	44	**D**	100	64	**d**
5	5	ENQ	37	25	**%**	69	45	**E**	101	65	**e**
6	6	ACK	38	26	**&**	70	46	**F**	102	66	**f**
7	7	BEL	39	27	**'**	71	47	**G**	103	67	**g**
8	8	BS	40	28	**(**	72	48	**H**	104	68	**h**
9	9	TAB	41	29	**)**	73	49	**I**	105	69	**I**
10	A	LF	42	2A	*****	74	4A	**J**	106	6A	**j**
11	B	VT	43	2B	**+**	75	4B	**K**	107	6B	**k**
12	C	FF	44	2C	**,**	76	4C	**L**	108	6C	**l**
13	D	CR	45	2D	**-**	77	4D	**M**	109	6D	**m**
14	E	SO	46	2E	**.**	78	4E	**N**	110	6E	**n**
15	F	SI	47	2F	**/**	79	4F	**O**	111	6F	**o**
16	10	DLE	48	30	**0**	80	50	**P**	112	70	**p**
17	11	DC1	49	31	**1**	81	51	**Q**	113	71	**q**
18	12	DC2	50	32	**2**	82	52	**R**	114	72	**r**
19	13	DC3	51	33	**3**	83	53	**S**	115	73	**s**
20	14	DC4	52	34	**4**	84	54	**T**	116	74	**t**
21	15	NAK	53	35	**5**	85	55	**U**	117	75	**u**
22	16	SYN	54	36	**6**	86	56	**V**	118	76	**v**
23	17	ETB	55	37	**7**	87	57	**W**	119	77	**w**
24	18	CAN	56	38	**8**	88	58	**X**	120	78	**x**
25	19	EM	57	39	**9**	89	59	**Y**	121	79	**y**
26	1A	SUB	58	3A	**:**	90	5A	**Z**	122	7A	**z**
27	1B	ESC	59	3B	**;**	91	5B	**[**	123	7B	**{**
28	1C	FS	60	3C	**<**	92	5C	****	124	7C	**|**
29	1D	GS	61	3D	**=**	93	5D	**]**	125	7D	**}**
30	1E	RS	62	3E	**>**	94	5E	**^**	126	7E	**~**
31	1F	US	63	3F	**?**	95	5F	_	127	7F	**DEL**

Tabel 2.28

Unicode

Met 127 codes is maar een klein deel van de tekens weer te geven die wereldwijd in diverse talen worden gebruikt. Daarom is de UNICODE ingevoerd. Deze code heeft voor het opslaan van een karakter 16 bits nodig. Hiermee kan aan 65536 verschillende karakters een unieke code worden gekoppeld. Zo wordt bijvoorbeeld de hoofdletter B opgeslagen als 0042_{Hex}.

2.11 Begrippenlijst

ASCII. Afkorting van American Standard Code for Information Interchange en is een standaard om een aantal letters, cijfers, leestekens en andere symbolen weer te geven.

Bitwise operator. Operator die logische operaties op meerdere bits uitvoert. Hierbij staat iedere bit op zichzelf.

Byte. Acht bits.

Carry. Bij een optelling optredende transportbit. (Bij handmatige optelling het cijfer dat men moet 'onthouden').

Full adder. Optelschakeling van drie 1-bits getallen.

Half adder. Optelschakeling voor twee 1-bits getallen.

Hexadecimaal getal. Getal in het zestientallige stelsel. Bedoeld om binaire code verkort weer te geven. Een getal wordt weergegeven met behulp van de set symbolen: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E en F.

Integer. Representatie van een geheel getal met teken in two's complement code.
Bij een 8-bits getal is het bereik: -128 .. +127.
Bij een 16-bits getal is het bereik: -32768 .. +32767.
Bij een 32-bits getal is het bereik: -2.147.483.648 .. +2.147.483.647.

lsb. (least significant bit). De bit met de kleinste gewichtsfactor.

msb. (most significant bit). De bit met de grootste gewichtsfactor.

Sum. Het resultaat van een optelling van twee of meer getallen, in het bijzonder van twee of meer 1-bits getallen, waarbij de sum ook uit één bit bestaat.

Sign extension. De conversie die plaatsvindt als het aantal bits toeneemt van een getal met teken. Doordat bij een positief getal 'nullen' worden toegevoegd en bij een negatief getal 'enen' blijft de waarde behouden.

Overflow. Het overschrijden van het bereik bij een integer.
Bij een 8-bits getal treedt een overflow op als een getal groter wordt dan 127 of kleiner dan -128.

Shift operator. Operator die de bits van een getal een aantal posities naar links of naar rechts verschuift.

Two's complement-representatie. Representatie van een geheel getal met teken (integer), waarbij het msb (most significant bit) het teken van het getal aangeeft. Is dit msb gelijk aan 0, dan is het getal positief; is het gelijk aan 1, dan is het getal negatief.
Om de schrijfwijze voor een negatief getal te vinden, neemt men van dat getal de absolute waarde, inverteert vervolgens elke bit en telt er 1 bij op.

Unicode. Een internationale standaard voor de identificatie van grafische tekens en symbolen.

Unsigned integer. Representatie van een positief geheel getal in binaire code.
Bij een 8-bits getal is het bereik: 0 .. 255.
Bij een 16-bits getal is het bereik: 0 .. 65535.
Bij been 32-bits getal is het bereik: 0 .. 4.294.967.295.

Word. 16 of 32 bits

2.12 Vragen en opgaven

Ter voorbereiding op een toets of tentamen zijn aan dit hoofdstuk enkele vragen toegevoegd. De antwoorden op deze vragen kun je vinden op www.science.uva.nl/amstel/SIM-PL.

1. Geef de waarheidstabel en een schema van een full adder weer.

2. Geef het schema weer van een schakeling waarmee twee 4-bits getallen kunnen worden opgeteld. Gebruik half en full adders als bouwstenen. Geef, voor zowel in- als uitgangen, aan welke bit het laagste gewicht heeft.

3. Geef het schema weer van een schakeling waarmee twee 4-bits getallen kunnen worden opgeteld of afgetrokken afhankelijk van een $\mathrm{SUB}/\overline{\mathrm{ADD}}$-signaal en die volgens de two's complement code werkt.

4. Stel dat de propagatietijd van een half adder 1 ns en van een full adder 2 ns is. Wat is de propagatietijd van de schakeling van figuur 2.4.

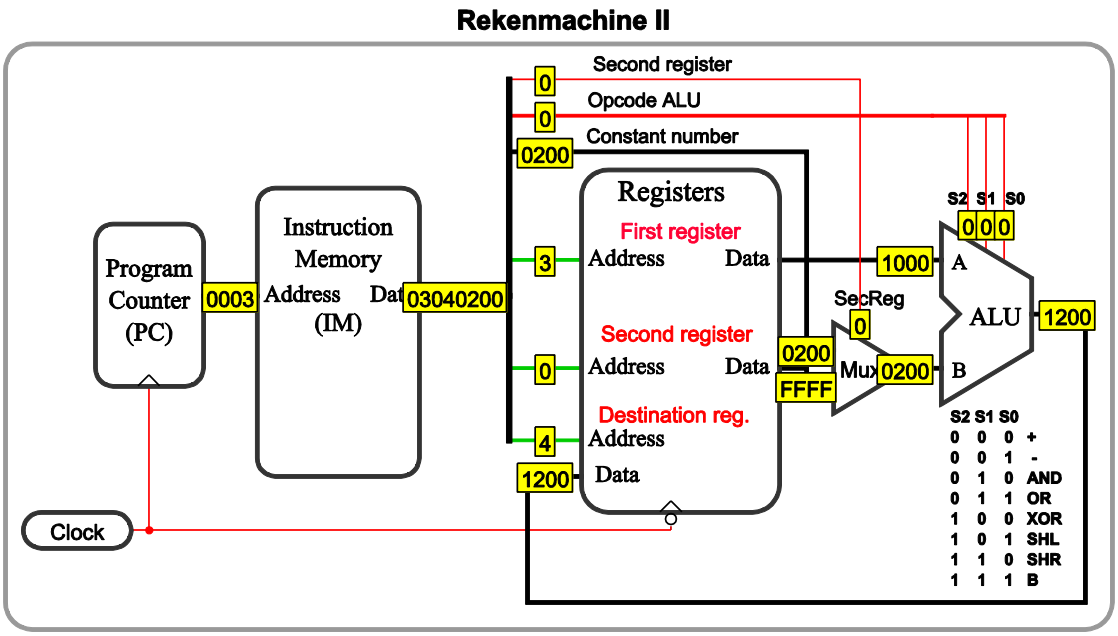
5. Geef voor een 4-bits getal de two's complement code van de decimale getallen -4 t/m +3 weer.

6. Gegeven een 8-bits machine. Bij welk decimaal getal treedt een carry op en bij welk getal treedt een overflow op?

7. Wat is het grootste getal en het kleinste getal dat kan worden gerepresenteerd door een 8-bits computerwoord in binaire code? Geef ook de bijbehorende hexadecimale en decimale waarden. Zelfde vraag, maar dan voor de two's complement code. Vul tabel 2.29 in.

8-bits	Binaire code	Hexadeci-male code	Decimale code	Two's complement code	Hexadeci-male code	Decimale code
Grootste getal						
Kleinste getal						

Tabel 2.29



Hoofdstuk 3: Hoe werkt een rekenmachine?

3.1 Leerdoelen

Een rekenmachine is een elektronisch apparaat waarop tenminste de volgende berekeningen kunnen worden uitgevoerd: optellen en aftrekken. Het hart van de rekenmachine is een Arithmetic Logic Unit. Deze voert de eigenlijke berekening uit. De behandelde machines zijn programmeerbaar. Door het schrijven van een programma kun je een aantal berekeningen achter elkaar uitvoeren. De (tussen)resultaten van deze berekeningen worden opgeslagen in registers. Een programma bestaat uit een reeks instructies. Een instructie voert een berekening uit en/of verplaatst gegevens tussen de verschillende componenten van de machine. Instructies worden opgeslagen in het instructiegeheugen. Programma's worden geschreven in de taal 'assembler'. Is de programmacode correct, dan wordt dit programma vertaald naar 'enen' en 'nullen' en kan het worden uitgevoerd.

Aan het einde van dit hoofdstuk weet je:

- uit welke componenten een rekenmachine is samengesteld;
- de functie van deze componenten;
- wat registers zijn;
- wat adressen zijn;
- wat het verschil tussen assembly-taal en machinetaal is;
- wat een operator is en wat operanden zijn;
- wat een instructieformaat is;
- wat een instructieset is;
- wat een instructie van het type immediate is;
- wat een 'operate code' op 'opcode' is;
- waar constante getallen worden opgeslagen;
- hoe een eenvoudige programmeerbare rekenmachine werkt.

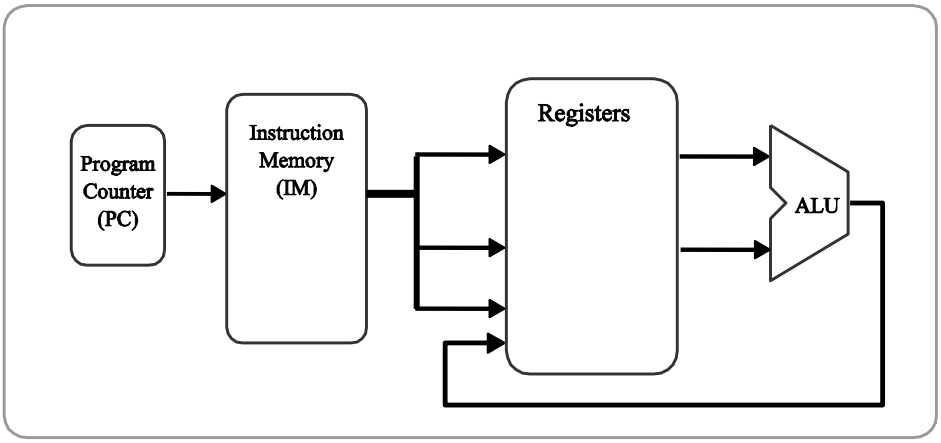
Aan het einde van dit hoofdstuk kun je:

- een assembly-programma compileren, loaden en executeren;
- van een instructie het datapad tekenen;
- een programma in de taal 'assembly' schrijven;
- de operate code en het instructieformaat van een instructie bepalen.

3.2 Overzicht van de rekenmachine

Figuur 3.1 geeft het blokschema van de rekenmachine weer. De machine bestaat uit vier hoofdcomponenten:

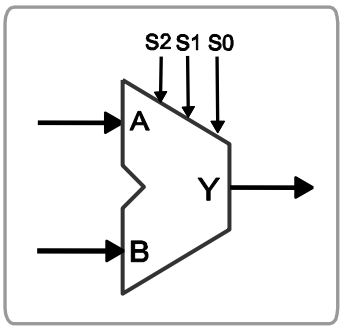
- Program Counter (PC). Deze teller houdt bij welke instructie wordt uitgevoerd.
- Instruction Memory. Hier worden de instructies opgeslagen.
- Registers. Hierin worden (tussen)resultaten van berekeningen in opgeslagen.
- ALU (Arithmetic Logic Unit). De ALU is het rekenorgaan van de machine.



Figuur 3.1 Blokschema rekenmachine

ALU

De ALU kan zeven verschillende bewerkingen op twee getallen van 16 bits uitvoeren. Naast de rekenkundige operatoren optellen en aftrekken zijn dat de bitwise operatoren AND, OR en XOR en de shift-operaties SHL (SHift Left) en SHR (SHift Right). Daarnaast kan het getal op ingang B worden doorgelaten.
Welke operatie wordt uitgevoerd, wordt bepaald door de ingangen S2, S1 en S0. Als bijvoorbeeld S2, S1 en S0 alle drie '0' zijn, dan worden de getallen op de ingangen A en B opgeteld. Als S2, S1 en S0 alle '1' zijn, dan wordt het getal op ingang B doorgelaten. In tabel 3.1 is de relatie vastgelegd tussen S2, S1 en S0 en de operatie die de ALU uitvoert.



Figuur 3.2 ALU

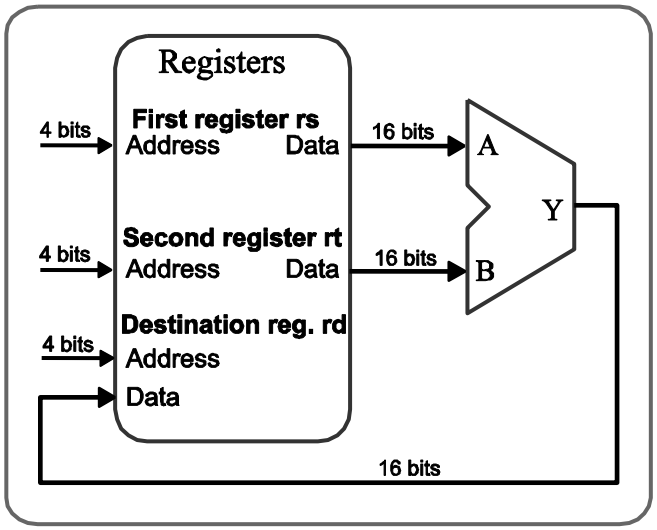
S2	S1	S0	Operator	Functie
0	0	0	+ (plus)	Y = A + B
0	0	1	- (min)	Y = A - B
0	1	0	& (bitwise AND)	Y = A AND B
0	1	1	| (bitwise OR)	Y = A OR B
1	0	0	^ (bitwise XOR)	Y = A XOR B
1	0	1	<< (Shift left)	Y = A SHL B
1	1	0	>> (Shift right)	Y = A SHR B
1	1	1	B wordt doorgelaten	Y = B

Tabel 3.1

Een ALU is een combinatorische schakeling. De waarden op de uitgangen wordt alléén bepaald door de waarden op de ingangen. Een ALU kan daarom een uitkomst van een berekening niet onthouden. Het ALU-resultaat wordt daarom in één van de registers opgeslagen.

Registers

Registers zijn nodig om (tussen)resultaten van een berekening in op te slaan. Ook de getallen waarmee de ALU rekent komen uit de registers. De rekenmachine in ons voorbeeld heeft 16 registers. Ieder van deze registers kan één 16-bits getal bewaren. De registers zijn van 0 t/m 15 genummerd. Met een 4-bits binaire code kun je één van deze 16 registers selecteren. Deze 4-bits code wordt het registeradres, kortweg 'adres' genoemd.
Vraag: Wat is het aantal bits dat in alle registers samen kan worden opgeslagen?
Antwoord: het totale aantal bits is …….



Figuur 3.3 Datapad: Registers en ALU

Drie registers
Een instructie als: c = a + b, rekent met drie getallen a, b en c. Er zijn drie registers nodig om deze getallen te bewaren. Het *first register rs*[6]*)* bewaart het getal dat bij a hoort, het *second register rt* bewaart het getal dat bij b hoort en het *destination register rd* bewaart de uitkomst c van de optelling die de ALU heeft berekend.
Welk van de 16 registers het First Register rs is, wordt bepaald door de code op de adresingang, dus het adres van rs. Het 16-bits getal in register rs gaat naar de A-ingang van de ALU. Het adres op de adresingang van het Second Register rt bepaalt welk register met de B-ingang van de ALU wordt verbonden. Het adres op de adresingang van het Destination register rd bepaalt in welk register het resultaat van de ALU- berekening wordt opgeslagen.

Getallen en adressen
In figuur 3.4 is de toestand van de registers weergegeven na het uitvoeren van een instructie die twee getallen optelt. In register 3 staat het decimale getal 12, in register 4 het decimale getal 8 en in register 7 het decimale getal 20. De andere registers bevatten geen getallen. SIM-PL maakt er 1111…..1 van.

Hogere programmeertaal
In een hogere programmeertaal als Java of C ziet een instructie die twee getallen optelt er uit als:
c = a + b; (betekenis: c wordt a + b)
a, b en c worden variabelen genoemd. In werkelijkheid zijn a, b en c de (register)adressen waarin resp. de getallen 12, 8 en 20 zijn opgeslagen. Variabele a correspondeert met adres 3, variabele b met adres 4 en variabele c met adres 7.

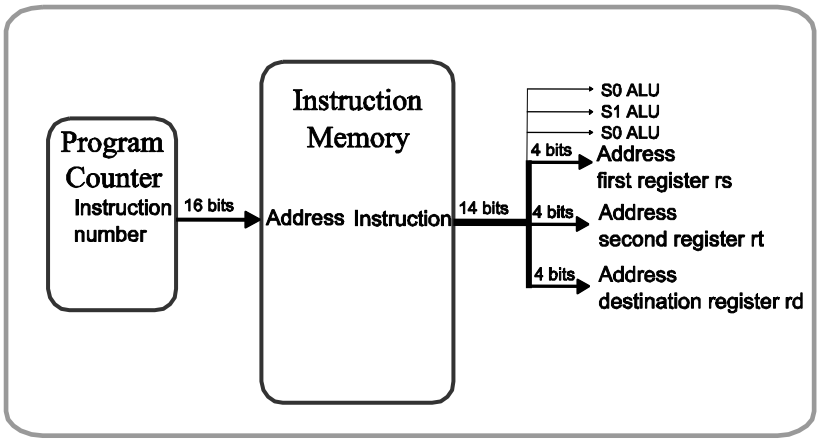
adres	getal
0	1111111111111111
1	1111111111111111
2	1111111111111111
3	0000000000001100
4	0000000000001000
5	1111111111111111
6	1111111111111111
7	0000000000010100
8	1111111111111111
9	1111111111111111
10	1111111111111111
11	1111111111111111
12	1111111111111111
13	1111111111111111
14	1111111111111111
15	1111111111111111

Figuur 3.4: registers

[6] De letters rs betekenen *register source (bron).* De tweede bron waar de ALU gegevens vandaan haalt, wordt rt genoemd.

Program Counter (PC) & Instruction Memory (IM)

De Program Counter bepaalt welke instructie wordt uitgevoerd. Het getal in de Program Counter, is het nummer van de instructie. Dit nummer wordt doorgegeven aan de adresingang van het Instruction Memory. Als een programma wordt gestart is het getal dat in de teller staat 0. Zodra de eerste instructie is uitgevoerd wordt dit getal 1. Het getal in de PC wordt na elke instructie met één verhoogd, waardoor steeds de volgende instructie van een programma wordt uitgevoerd. De Program Counter is 16 bits en geeft dus een 16-bits adres door naar het Instruction Memory. Op ieder adres van het IM is een instructie opgeslagen als een keten van 'enen' en 'nullen'.



Figuur 3.5 Program Counter en Instruction Memory

Vraag: Hoeveel instructies kunnen er in het Instruction Memory worden opgeslagen?
Antwoord: …….. instructies.

3.3 Instructies

Machinetaal

De taal waarmee iedere instructie in het instructiegeheugen wordt opgeslagen, heet machinetaal. Machinetaal bestaat uit een keten van enen en nullen. Zo wordt bijvoorbeeld de instructie: 'tel het getal in register 3 bij het getal in register 4 op en stop het resultaat in register 7', in het instructiegeheugen opgeslagen als: 000 0011 0100 0111.

Assembler code

Instructies in machinetaal zoals 000001101000111 zijn voor een programmeur geen werkbare taal. Daarom is een symbolische taal 'assembler' genaamd ingevoerd. De syntaxis van de instructie voor optellen in assember is: ADD rd, rs, rt. De syntaxis is de exacte beschrijving van de vorm van een instructie. Bovenstaande instructie wordt dan: ADD $7, $3, $4.

Relatie tussen registernamen en registernummers in assembler
De assembler die in deze cursus wordt gebruikt, duidt registernamen aan met het $-teken. Met $7 wordt bedoeld register 7, met $3 register 3 en met $4 register 4.

Het bovenstaande 15-bits binaire getal kan worden omgezet naar het hexadecimale stelsel door het getal eerst in groepjes van vier bits op te splitsen, te beginnen vanaf de rechter kant. Per vier bits wordt daarna het bijbehorende hexadecimale cijfer ingevuld.
Van 000001101000111 maken we 000 0011 0100 0111 en dat wordt 0x0347. 0x geeft aan dat het een hexadecimaal getal is.

Operator en operanden

Een instructie is samengesteld uit een operator en één of meer operanden. Bij bovenstaande instructie is de operator ADD en zijn de operanden $7, $3 en $4.

Instructieformaat

Hoe ziet een instructie er in detail uit? Iedere instructie bestaat uit 15 bits die samen het zgn. instructiewoord vormen. De drie meest linker bits geven de operatie aan die de ALU moet uitvoeren (tel op). De andere twaalf bits zijn verdeeld in drie velden. Ieder veld bevat het 4-bits adres van resp. rs, rt en rd. Het instructiewoord voor de instructie: 'tel op het getal in register 3 bij het getal in register 4 en stop het resultaat in register 7' is: 000 0011 0100 0111. In hexadecimale code is dit instructiewoord: 0347.

Veld	ALU-veld S2,S1,S0	First Register operand rs	Second Register operand rt	Destination Register operand rd
Aantal bits	3	4	4	4
Voorbeeld Betekenis	000 optellen	0011 register 3	0100 register 4	0111 register 7

Tabel 3.2 Instructieformaat van een instructie

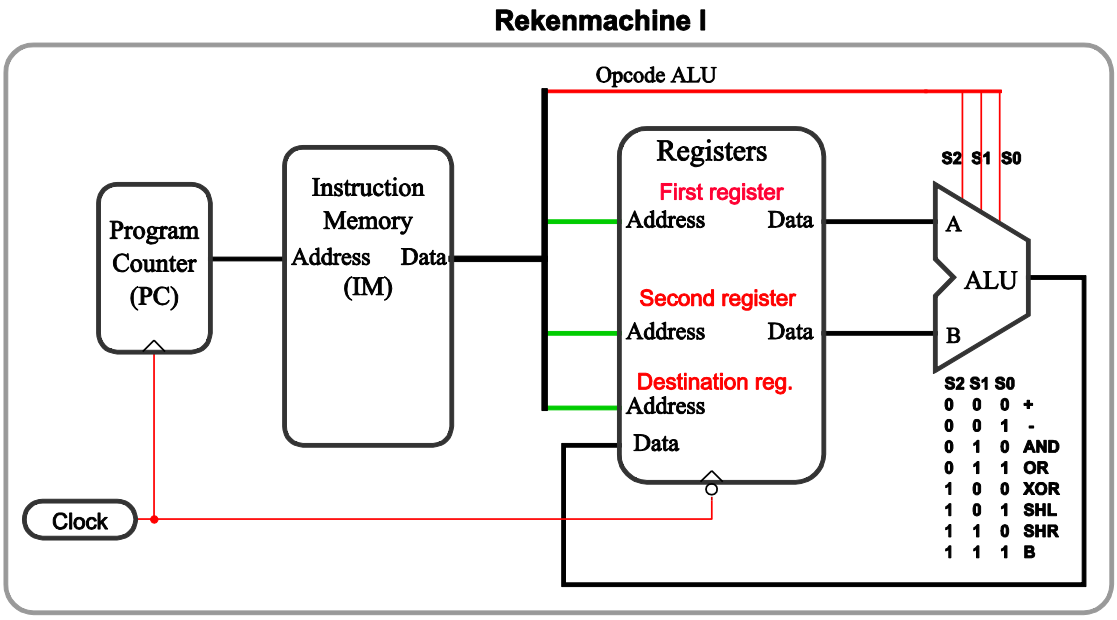
Instructieset

Bij elke operatie die de ALU kan uitvoeren, behoort een overeenkomende instructie van de rekenmachine. De acht instructies zijn in tabel 3.3 weergegeven.

Instructie	Betekenis	Voorbeeld	Betekenis
ADD rd, rs, rt	Optellen registers	ADD $7, $3, $4	r7 ← r3 + r4
SUB rd, rs, rt	Aftrekken registers	SUB $7, $3, $4	r7 ← r3 – r4
AND rd, rs, rt	Bitwise AND registers	AND $7, $3, $4	r7 ← r3 & r4
OR rd, rs, rt	Bitwise OR registers	OR $7, $3, $4	r7 ← r3 | r4
XOR rd, rs, rt	Bitwise XOR registers	XOR $7, $3, $4	r7 ← r3 ^ r4
SHL rd, rs, rt	Shift left register	SHL $7, $3, $4	r7 ← r3 << r4
SHR rd, rs, rt	Shift right register	SHR $7, $3, $4	r7 ← r3 >> r4
COPY rd, rt	Kopieer register	COPY $7, $4	r7 ← r4

Tabel 3.3 Instructieset rekenmachine

De complete rekenmachine

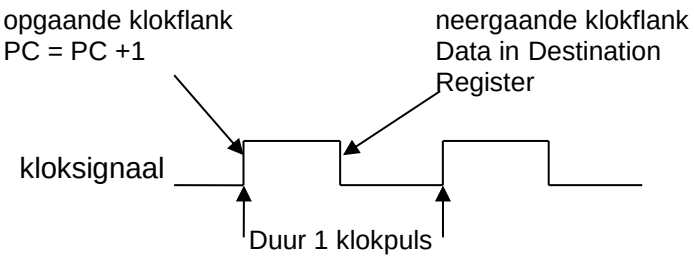


Figuur: 3.6 Rekenmachine I

In figuur 3.6 is het schema van de complete rekenmachine weergegeven zoals die met SIM-PL is gemaakt. Eén component is nog niet ter sprake gekomen en dat is de systeemklok.

Timing

De systeemklok speelt een centrale rol in het aansturen van de componenten van de schakeling. Iedere instructie wordt in één klokpuls uitgevoerd. Op de opgaande flank van een klokpuls wordt de Program Counter met één verhoogd zodat de volgende instructie kan worden uitgevoerd. Op de neergaande klokflank wordt het resultaat van een berekening in het destination register geladen. De tijd daartussen is nodig om het Instruction Memory en de registers te lezen en de ALU-berekening uit te voeren.
Opmerking: Een opgaande klokflank is de overgang van het '0'-niveau naar het '1'-niveau van het kloksignaal en een neergaande klokflank is de overgang van het '1'-niveau naar het '0'-niveau.



Figuur 3.7: Klokpuls van de systeemklok

3.4 Practicum met de Rekenmachine I

Starten van de simulator

Open de folder SIM-PLx.x.x. Dubbelklik op de file Executer.jar. Klik op de icoon:  (Open an existing worksheet). Klik op Componenten→ H3Rekenmachines. Open de file Rekenmachine_I.simpl-ws.
De User Interface laat vier windows zien:

- Component Window. Hier wordt de Rekenmachine I afgebeeld.
- Instance Editor: Instruction Memory. Hier kun je het Instruction Memory bekijken.
- Instance Editor: Registers. Hier kun je de Registers bekijken.
- Program Editor. Dit venster wordt gebruikt om een assembly-programma te laden of te schrijven.

```
# Rekenmachine_I WASM program        #-teken is het teken voor commentaar
@include "Rekenmachine_I.wasm"       # Deze regel altijd toevoegen

.data MyData : REGISTERS             # Het programma start met data in registers
3: WORD  b  7                        # register 3 krijgt waarde 7
4: WORD  c   5                       # register 4 krijgt waarde 5
7: WORD  a                           # a, b en c zijn zgn aliasen voor $7, $3 en $4

.code MyCode : REKENMACHINE_I, MyData     # Hier begint het programma
ADD $6, $3, $4                       # register 6 wordt register 3 + register 4
SUB a, b, c                          # register 7 wordt register 3 - register 4
COPY $8, $6                          # register 8 krijgt waarde van register 6
```

Opdracht 1: Laden, compileren en executeren van een programma
Ga naar de menu-optie File van de Program Editor. Open het programma 'Opdracht1.wasm'. Het volgende programma verschijnt:

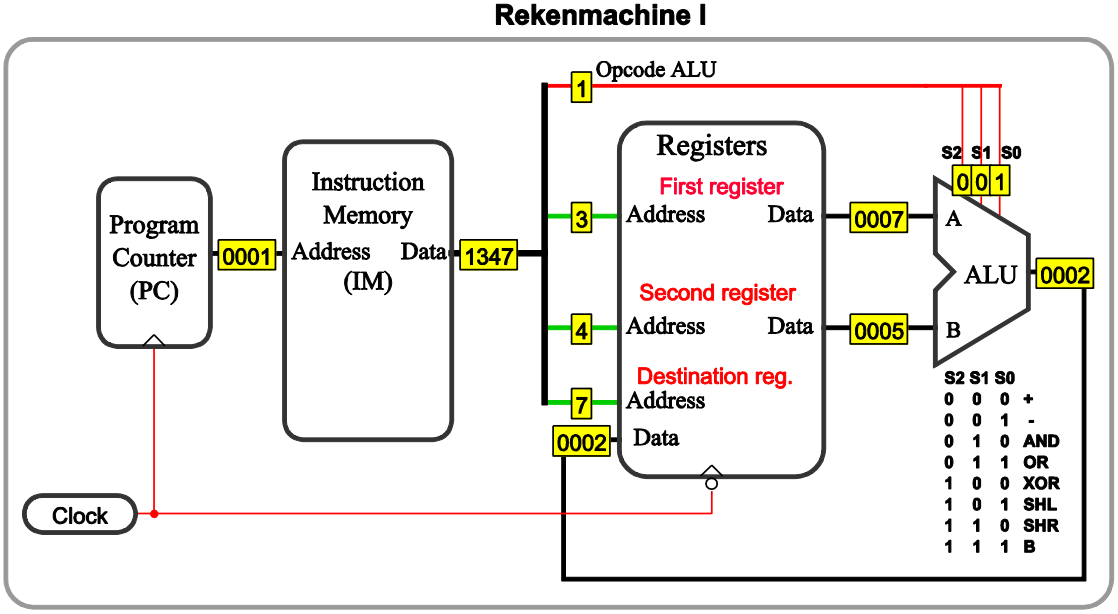
Klik op Compile. Het programma, dat uit drie instructies bestaat, wordt vertaald naar machinetaal. Een groen vinkje verschijnt als de programmacode syntactisch correct is. Alle initiële waarden (waarden die de simulator aanneemt na laden van de schakeling) zijn '1'. Een getal in één van de 16-bits registers wordt weergegeven in hexadecimale code als: FFFF. Het Instruction Memory is 15 bits. Daarom is de beginwaarde van elke geheugenplaats hier 7FFF.

Klik op de rode pijl:  . Er gebeuren drie dingen:

- De machinecode wordt geladen in het Instruction Memory. De drie vertaalde instructies van het programma staan op de eerste drie geheugenplaatsen als 0346, 1347 en 7068 in hexadecimale code.

- Register 3 en register 4 zijn geladen met de getallen 7 respectievelijk 5.
- De eerste instructie in de Program Editor licht op.

Klik op de goudbruine pijl:  . De eerste instructie wordt hierdoor uitgevoerd. Kijk naar het window dat de registers weergeeft en ga na of het resultaat van de eerste berekening in het juiste register staat ($C_{Hex} = 12_{Dec}$). Klik nogmaals op de goudbruine pijl. Figuur 3.8 geeft de toestand van de machine weer na het uitvoeren van de tweede instructie.



Figuur 3.8: Toestand van de machine na executie van de instructie SUB $7, $3, $4.

Beantwoord de volgende vragen:
Van het programma zijn al twee instructies uitgevoerd. Waarom is de waarde van de Program Counter 1 en niet 2? Antwoord: …….
Beschouw het window linksonder op het scherm. Staat in het Instruction Memory de code 1347 op geheugenplaats 1?
Antwoord: ……..
Wat betekenen de cijfers 1, 3, 4 en 7 van deze code?
Antwoorden: 1 = ……
3 = ……
4 =…….
7 = ……
Wat is de waarde van S2 en S1 en van S0? Antwoord: …….
Bij welke operator uit tabel 3.1 behoren deze waarden? Antwoord: ……
Staat het resultaat van de berekening in het juiste register? Antwoord: ……

Voer de volgende opdrachten uit:
Verander via de optie ‘Settings’ het Number Format naar Binary en daarna naar Decimal.
Voer de laatste instructie uit van het programma en controleer het resultaat.
Klik nogmaals op de rode pijl. Alle componenten nemen hierdoor weer hun initiële waarden aan. Klik op de gele pijl:  . Vul in het invoerveld ‘3’ in. De drie instructies van het programma worden nu in één keer uitgevoerd.

Opdracht 2: Programma dat vier getallen optelt
Schrijf een programma dat vier getallen die zijn opgeslagen in de registers 1 tot en met 4 optelt. Het resultaat moet terechtkomen in register 5. De getallen in de registers 1 tot en met 4 mogen door het programma niet worden gewijzigd. Hoeveel getallen kan de ALU tegelijk optellen?
Aanbeveling: Save het programma ‘Opdracht1.wasm’ onder de naam ‘Opdracht2.wasm’ door gebruik te maken van de Save As-optie en pas het ‘oude programma’ aan.

Opdracht 3: Bitwise AND

Bekijk de tekst in hoofdstuk 2.8 over bitwise operatoren.
In het tekstvenster hiernaast is het programma 'Opdracht3.wasm' weergeven.
Opdracht: Voer dit programma eerst uit op papier.
Wat staat er in register 6 nadat de eerste instructie is uitgevoerd?
Antwoord: ……

```
@include "Rekenmachine_I.wasm"

.data MyData : REGISTERS
3: WORD  0x5555        # 0x is hexadecimaal

.code MyCode : REKENMACHINE_I, MyData
ADD $6, $3, $3
AND $7, $3, $6
```

Wat staat er in register 7 nadat de tweede instructie is uitgevoerd?
Antwoord: ……
Controleer je antwoorden door executie van het programma Opdracht3.wasm.

Opdracht 4: Vermenigvuldigen

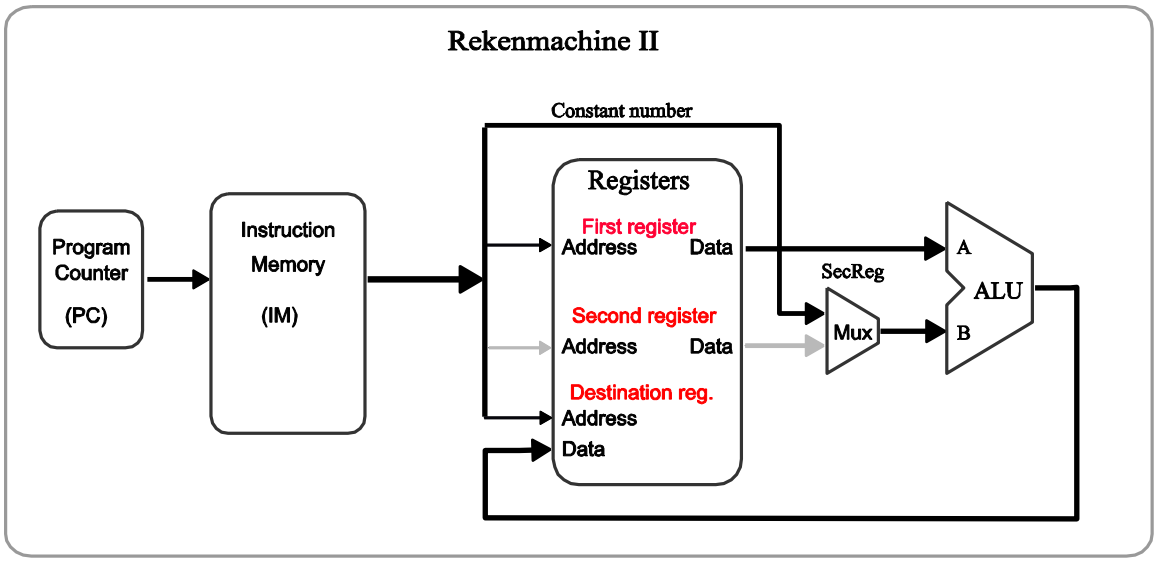
Met welke instructie kun je een getal met twee vermenigvuldigen zonder de operator SHL te gebruiken? Antwoord: …. .
Schrijf een programma dat het getal dat in register 8 staat, met vier vermenigvuldigt. Het enige register dat gebruikt mag worden is register 8.

3.5 Hoe krijg ik een constant getal in een register?

De meest voorkomende instructies die een computer uitvoert, zijn instructies waarbij met constante getallen wordt gerekend. Een voorbeeld is het optellen van een constant getal bij het getal in een register. Een ander voorbeeld is een instructie die een constant getal in een register laadt. Het datapad van de rekenmachine wordt uitgebreid, zodat ook deze zogenoemde immediate instructies kunnen worden uitgevoerd.

Instructies van het type immediate

Een instructie van het type immediate voert een ALU operatie uit op een getal uit een register en een constant getal. Het resultaat wordt weer opgeslagen in een register.
Het constante getal is opgeslagen in het instructiegeheugen en maakt deel uit van het instructiewoord. De syntax van de immediate-instructie ADDI (ADD immediate) is: ADDI rd, rs, imm. Imm stelt het constante getal voor. Het Second Register rt wordt bij dit type instructie niet gebruikt.
Voorbeeld: ADDI $7, $8, 0d100 (0d = decimaal).
Betekenis: r7 ← r8 + 100.
In woorden: in register 7 komt de som van r8 (inhoud register 8) en 100.



Figuur 3.9 Datapad van een immediate-instructie

Uitbreiding datapad
Om immediate-instructies te kunnen uitvoeren wordt het datapad uitgebreid met een extra 16-bits verbinding tussen het Instruction Memory en de B-ingang van de ALU. De delen van het datapad die worden gebruikt tijdens het uitvoeren van de ADDI-instructie, zijn in figuur 3.9 zwart weergegeven.

Instructie LOADI (Load Immediate)
Bij een LOADI-instructie wordt een constant getal in een register geladen. De ALU wordt bij deze instructie alleen gebruikt om data door te laten.
De syntax van een LOADI-instructie is: LOADI rd, imm.

Instructieset en de architectuur van Rekenmachine II

De instructieset van de rekenmachine heeft er acht nieuwe instructies bij: LOADI, ADDI, SUBI, ANDI, ORI, XORI, SHLI en SHRI. In tabel 3.4 is de complete instructieset weergegeven.

Instructie	Betekenis	Voorbeeld	Betekenis
ADD rd, rs, rt	Optellen registers	ADD $5, $6, $7	r5 ← r6 + r7
SUB rd, rs, rt	Aftrekken registers	SUB $5, $6, $7	r5 ← r6 - r7
AND rd, rs, rt	Bitwise AND registers	AND $5, $6, $7	r5 ← r6 & r7
OR rd, rs, rt	Bitwise OR registers	OR $5, $6, $7	r5 ← r6 | r7
XOR rd, rs, rt	Bitwise XOR registers	XOR $5, $6, $7	r5 ← r6 ^ r7
SHL rd, rs, rt	Shift Left register	SHL $5, $6, $7	r5 ← r6 << r7
SHR rd, rs, rt	Shift Right register	SHR $5, $6, $7	r5 ← r6 >> r7
COPY rd, rt	Copy register	COPY $3, $2	r3 ← r2
ADDI rd, rs, imm	Optellen register en constante	ADDI $5, $6, 0x1234	r5 ← r6 + 0x1234
SUBI rd, rs, imm	Aftrekken register en constante	SUBI $7, $6, 0x1234	r7 ← r6 - 0x1234
ANDI rd, rs, imm	Bitwise AND register en constante	ANDI $5, $6, 0d34	r5 ← r6 & 0d34
ORI rd, rs, imm	Bitwise OR register en constante	ORI $5, $6, 0d34	r5 ← r6 | 0d34
XORI rd, rs, imm	Bitwise XOR register en constante	XORI $5, $6, 0d34	r5 ← r6 ^ 0d34
SHLI rd, rs, imm	Shift Left register	SHLI $5, $6, 0d5	r5 ← r6 << 5
SHRI rd, rs, imm	Shift Right register	SHRI $5, $6, 0d5	r5 ← r6 >> 5
LOADI rd, imm	Laad getal in register	LOAD $1, 0x 0020	r1 ← 0x0020

Tabel 3.4: Instructieset van Rekenmachine II

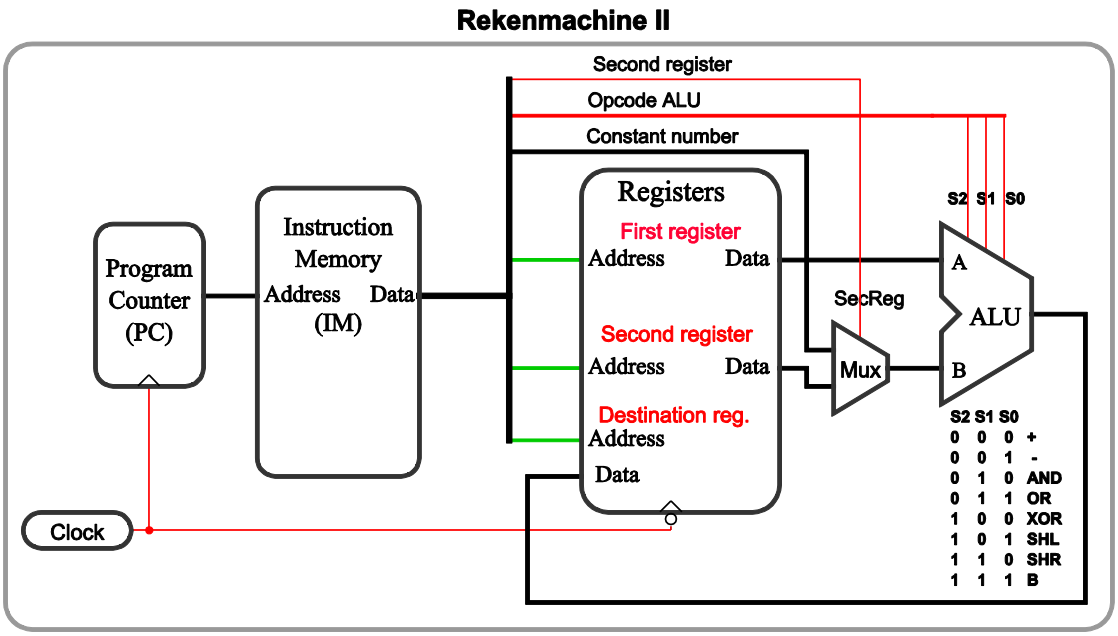
Constante en variabele getallen
Een constant getal is een getal dat door het programma niet kan worden gewijzigd. Dit in tegenstelling met een variabel getal. Een variabel getal is een adres en de inhoud ervan kan dus worden gewijzigd. Een instructie als ADDI $5, $6, 12 kom je in een hogere programmeertaal als Java of C tegen als a = b + 12. Deze instructie telt de constante 12 op bij de variabele b en kent de som toe aan de variabele a.

Architectuur Rekenmachine II

In figuur 3.10 is de complete architectuur van Rekenmachine II weergegeven.

Extra component: multiplexer (Mux)[7)]
Naast een constant getal uit het Instruction Memory moet het mogelijk blijven om het 'Second register' met de B-ingang van de ALU te verbinden. Hiervoor is een schakelaar, meestal Mux genoemd, nodig. Afhankelijk van het controlesignaal SecReg, wordt één van de twee ingangen van de Mux doorgeschakeld naar de B-ingang van de ALU. Als SecReg '0' is, dan wordt het constante getal uit het instructiegeheugen doorgelaten en als SecReg '1' is, dan wordt het getal van het 'Second register' doorgelaten.

[7)] Een multiplexer is een schakelaar. De opbouw van een multiplexer wordt behandeld in hoofdstuk 1.



Figuur 3.10: Schema van de Rekenmachine II

Operate code (opcode)

Iedere instructie heeft zijn eigen unieke code. Voor SUBI is dat 0001. Deze code wordt operate code (meestal opcode) genoemd. De opcode bestaat bij deze rekenmachine uit vier bits. Drie van deze bits bepalen de operatie die de ALU uitvoert, de vierde bit, SecReg stuurt de multiplexer aan.

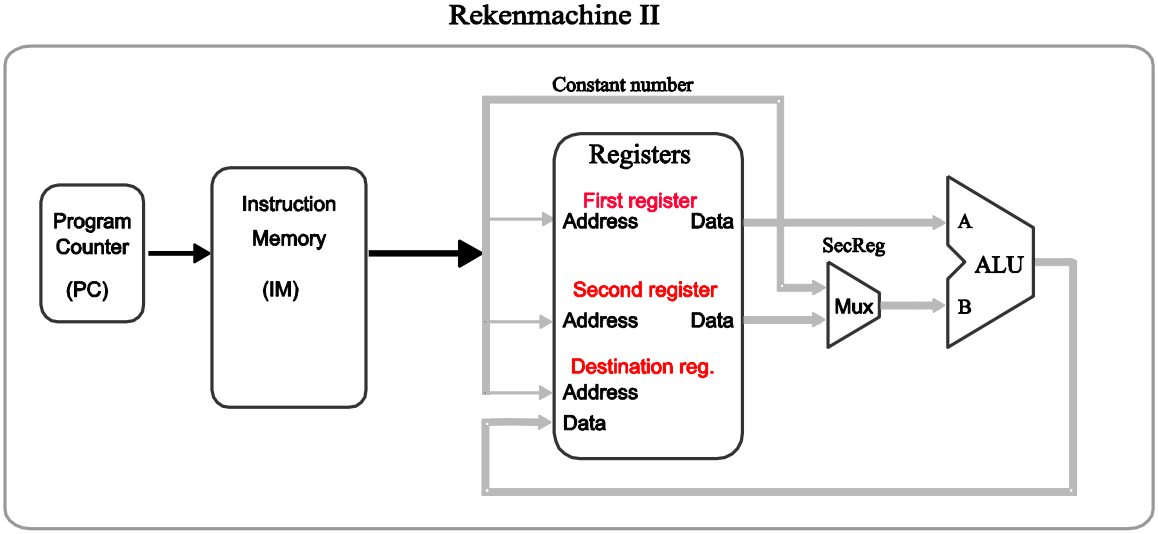
Instructieformaat

Iedere (machinetaal)instructie is opgebouwd volgens een vast formaat. Om immediate-instructies te kunnen uitvoeren wordt aan het instructieformaat van deze machine één veld toegevoegd om het constante getal in op te slaan. Het formaat bestaat uit vijf velden met een vaste bitgrootte. De vijf velden bevatten achtereenvolgens: opcode, (register)adres van het eerste getal, adres van het tweede getal, adres waar het ALU-resultaat naar toe gaat en een veld voor een constant getal. Tabel 3.5 geeft (een deel van) het instructieformaat weer.

3.6 Opdrachten

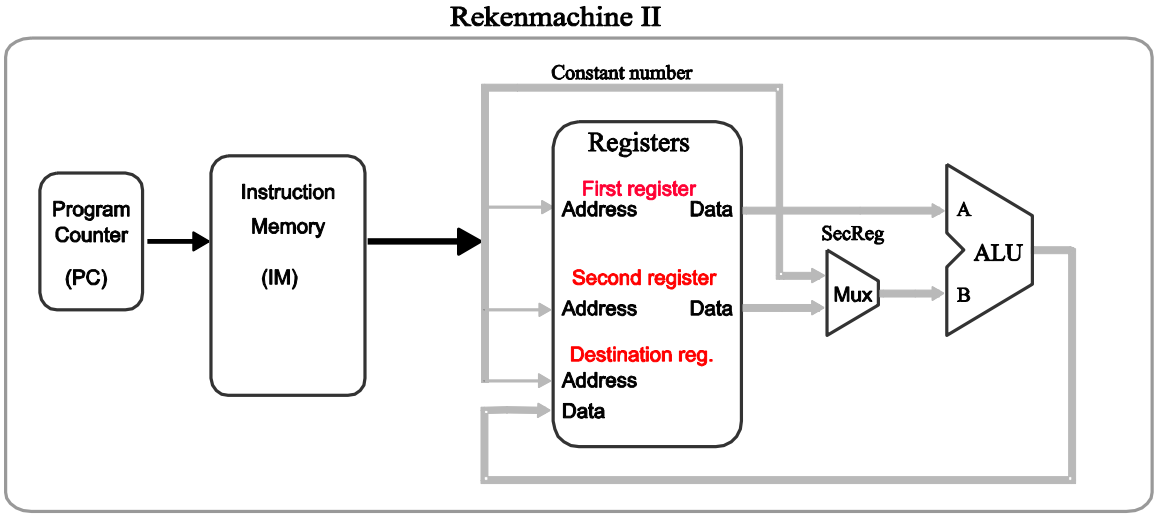
Opdracht 1: Datapad LOADI-instructie

Arceer het datapad van de LOADI-instructie in figuur 3.11. Geef ook aan welke adresingangen worden gebruikt bij deze instructie.



Figuur 3.11 Datapad van de instructie LOADI

Opdracht 2: Datapad COPY-instructie
Accentueer het datapad van de COPY-instructie in figuur 3.12. Geef ook aan welke adresingangen worden gebruikt bij deze instructie.



Figuur 3.12 Datapad van de instructie COPY

Instructieformaat
Vul de velden behorende bij de instructies AND, COPY, SUBI, ANDI en LOADI van tabel 3 in. Als een veld niet wordt gebruikt, vul dan x in. Bij de SIM-PL-software worden niet gebruikte velden 0.

Instructieformaat 16 bit rekenmachine (totaal 32 bits)						
Instructie	Opcode		First Register operand rs	Second Register operand rt	Destination Register operand rd	Constant getal
	Mux SecReg	Opcode ALU S2S1S0				
Aantal bits	1	3	4	4	4	16
ADD	1	000	rs	rt	rd	x
SUB	1	001	rs	rt	rd	x
AND						
COPY						
ADDI	0	000	rs	x	rd	getal
SUBI						
ANDI						
LOADI						

Tabel 3.5: Deel van het instructieformaat van rekenmachine II

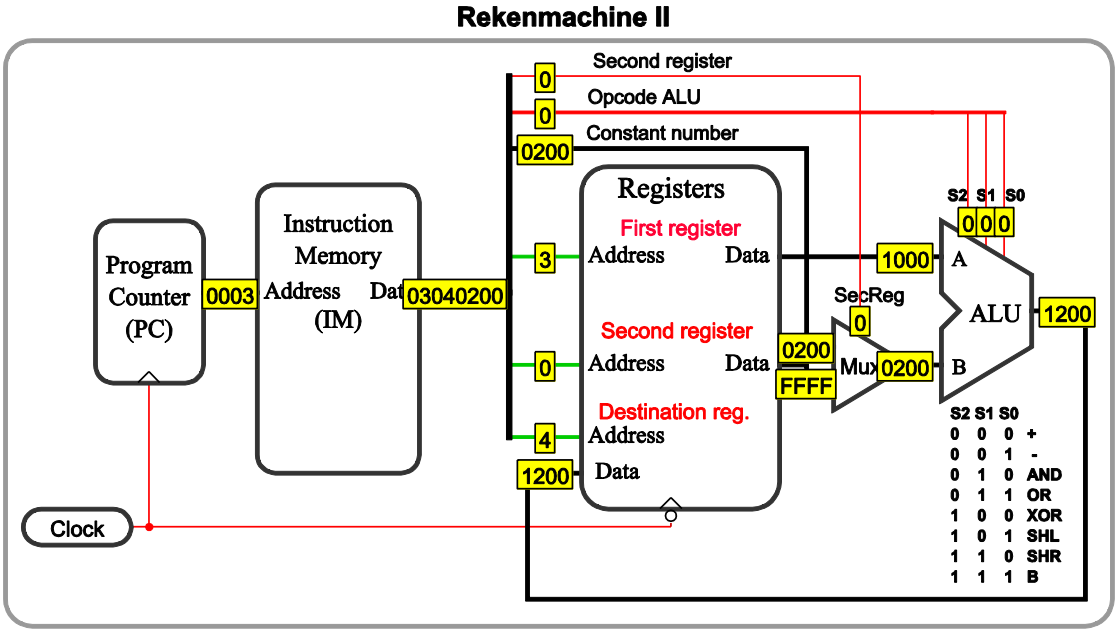
3.7 Practicum met de rekenmachine II

Opdracht 1: Executeren van voorbeeldprogramma
Start de Executer en kies Open → Componenten → H3Rekenmachines → Rekenmachine_II.sim-pl-ws. Open in het Program Window de file 'Rekenen.wasm'. Het volgende programma verschijnt nu in het Program Window:

```
@include "Rekenmachine_II.wasm"
.code MyCode : REKENMACHINE_II, REGISTERS
LOADI $1, 0x3000        # Laad 3000hex in Register 1
LOADI $2, 0x2000        # Laad 2000hex in Register 2
SUB $3, $1, $2          # Register 3 ← Reg1 - Reg2
ADDI $4, $3, 0x0200     # Register 4 ← Reg3 + 0200hex
```

Compileer en laad het programma. Executeer de vier instructies.
Figuur 3.13 geeft de toestand van de machine weer, na executie van de vierde en laatste instructie: ADDI $4, $3, 0x0200.

Toelichting
De instructie ADDI $4, $3, 0x0200 telt het getal 0200_{Hex} op bij de inhoud van register 3 en slaat het resultaat op in register 4.
De Program Counter begint bij 0. Dus bij de vierde instructie heeft de PC de waarde 3.
De inhoud van adres 3 van het instructie geheugen = 0304200_{Hex}. Deze code bestaat uit (vlnr):
0 = opcode (0000_{Bin})
3 = adres First register rs
0 = adres Second register (hier niet gebruikt)
4 = adres Destination register rd
0200 = constant getal 0200_{Hex}.
Dit constante getal gaat via de 'SecReg-multiplexer' naar de ALU. De ALU telt dit getal op bij de data van het getal in register 3: 0x1000. De som 0x1200 wordt in register 4 geschreven.



Figuur 3.13: Toestand van de machine na executie van ADDI $4, $3, 0x0200

Opdracht 2: Programma met instructies van het type immediate
Schrijf een programma dat de decimale getallen 512 en 255 uit het Instruction Memory optelt en van de som 256 aftrekt. Het resultaat moet terecht komen in register 7. Gebruik niet meer dan drie instructies. Het decimaal getal 512 noteer je als 0d512. Geef in tabel 3.6 de hexadecimale waarde van alle decimale getallen die worden ingevoerd en die door de machine worden berekend.

Decimaal	Hexadecimaal
512	
255	
som	
256	
verschil	

Tabel 3.6

Opdracht 3: Programma dat een negatief getal oplevert.
Schrijf een programma dat van het decimale getal 100 het decimale getal 103 aftrekt. Wat is in binaire code en in hexadecimale code het getal -3?
Antwoord: -3_{Dec} = ……$_{Bin}$ = ……$_{Hex}$

Met de binaire code kunnen geen negatieve getallen worden weergegeven. Met de zogenaamde 'two's complemente code' kan dat wel. Deze machine werkt dus met de two's complement code. Dit is de code voor berekeningen met gehele getallen met teken.

Opdracht 4: Swap($1, $2)
Ga na dat (5 XOR 9) XOR 5 gelijk is aan 9 en dat (5 XOR 9) XOR 9 gelijk is aan 5. Schrijf een programma dat de inhoud van $1 en $2 verwisselt zonder gebruik te maken van een derde register. Gebruik de instructie LOADI om de beide registers een initiële waarde te geven en maak verder alleen (drie keer) gebruik van de instructie XOR.

Conversie van big endian naar little endian in een 32-bits systeem

Niet iedere processor slaat de bytes van een woord op in dezelfde bytevolgorde in het data memory (zie hoofdstuk 8). Er zijn twee systemen: big-endian en little-endian genoemd. Big-endian systemen slaan de *most significant byte* van een woord op in het laagste adres (in bytes) en de least significant byte op het hoogste addres. Little-endian systemen, doen dit andersom. Als voorbeeld beschouwen we het 4-bytes getal: 0x0A0B0C0D. In tabel 3.7 is de volgorde weergegeven van de opgeslagen bytes in beide systemen.

Getal in register	0A	0B	0C	0D
Memory address	a+3	a+2	a+1	a
Little-endian	0A	0B	0C	0D
Big-endian	0D	0C	0B	0A

Tabel 3.7

Opdracht 5: Bitmanipulatie met masking en shifting

```
# Bitmanipulatie met masking en shifting
# Converteer 0xABCD naar 0xDCBA

@include "Rekenmachine_II.wasm"

.code MyCode : REKENMACHINE_II
LOADI $1, 0xABCD
SHLI $2, $1, 0d12        # D000 ← ABCD << 12
ANDI $3, $1, 0x00F0      # 00C0 ← ABCD & 00F0
SHLI $3, $3, 0d4         # 0C00 ← 00C0 << 4
OR $2, $2, $3            # DC00 ← D000 | 00C0
# up to you to finish the exercise
```

Rekenmachine-II werkt met 16-bits getallen dus een Word bestaat uit twee bytes. Een conversie van little-endian naar big-endian is een weinig uitdagende opdracht. Daarom voeren we deze opdracht uit met *nibbles*. Een 4-bits getal wordt een *nibble* genoemd. Een register van deze machine bevat dus 4 nibbles.
Converteer het getal 0xABCD naar het getal 0xDCBA. Een deel van de code is hieronder weergegeven.

3.8 Begrippenlijst

ALU. Arithmetic Logic Unit, het rekenorgaan van een computer.

Assembler. Een programma dat een symbolische versie van een instructie vertaalt naar machinetaal.

Assembly taal. Een symbolische taal die kan worden vertaald door een assembler naar machinetaal.

Instructie formaat. Een vorm van representatie waarbij een instructie is samengesteld uit binaire velden.

Instruction Memory. Een geheugen waarin de instructies worden opgeslagen.

Instruction set. De lijst van commando's die op een gegeven architectuur geëxecuteerd kan worden.

Machinetaal. Een reeks '1'-nen en '0'-len waarin elke computer instructies in het geheugen opslaat.

Opcode, operate code. Het veld van de instructie (in machinetaal) dat bepaalt welke instructie wordt uitgevoerd.

Operand. Getal of adres.

Program counter (PC). Een teller die het adres bevat van de instructie die geëxecuteerd wordt.

Registeradres. Het registernummer ofwel de aanduiding van de locatie waar een register zich bevindt.

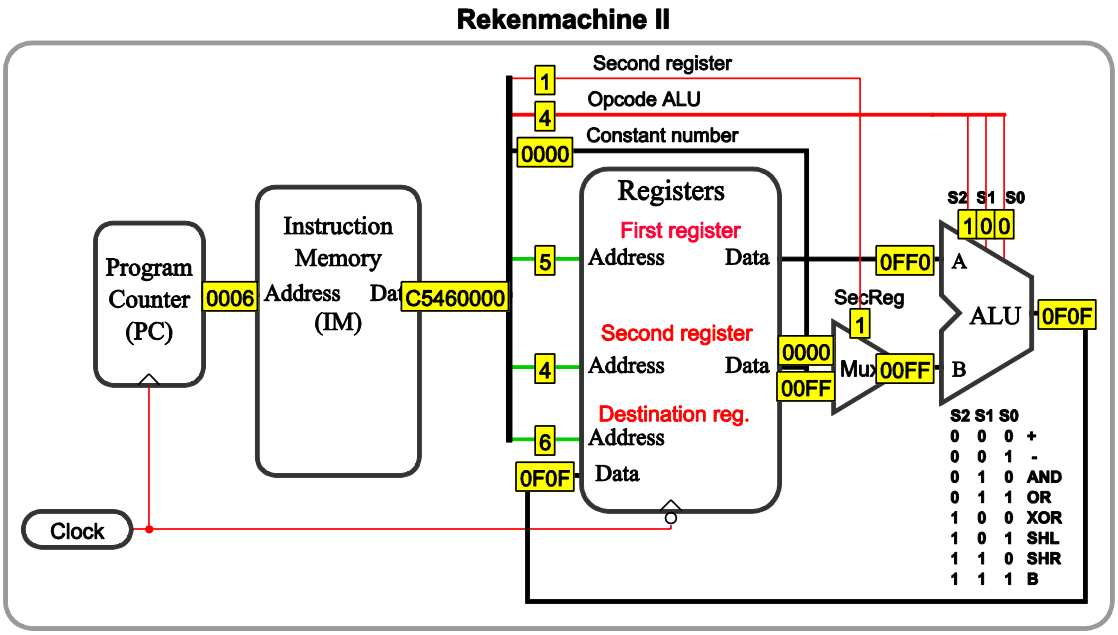
Register. Geheugenelement van 16, 32 of 64 bits waarin (tijdelijk) gegevens kunnen worden opgeslagen die nodig zijn voor berekeningen.

Syntaxis. De exacte beschrijving van de vorm/structuur van een instructie.

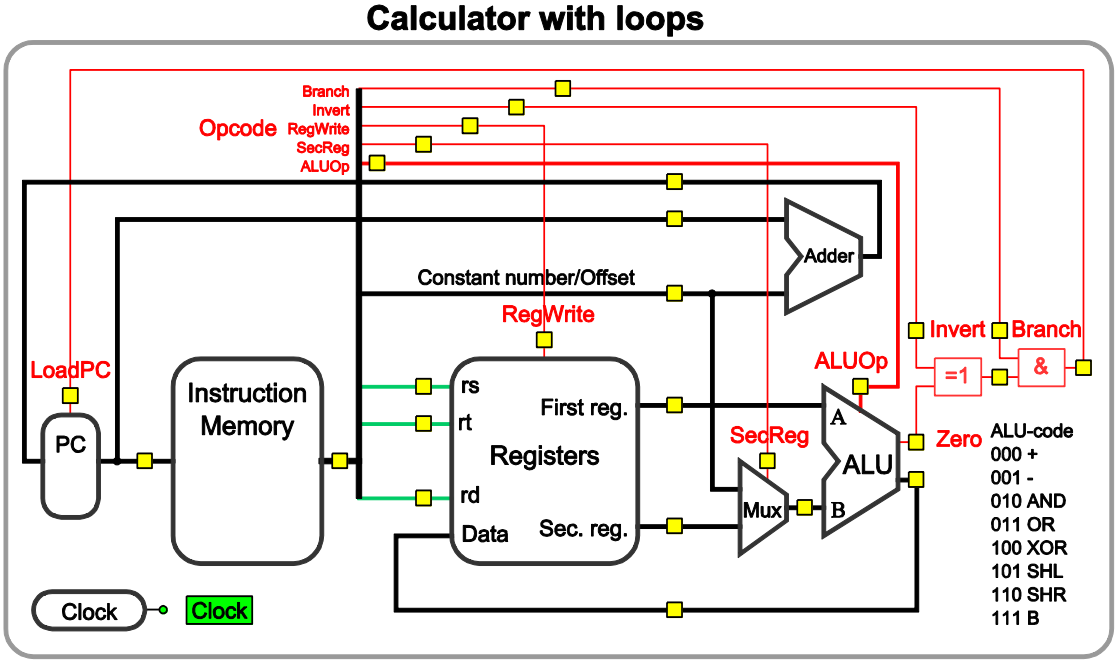
3.9 Vragen

Beschouw figuur 3.14 en beantwoord hierover de volgende vragen:

1. Welke operatie voert de ALU uit?
2. Welke instructie heeft de machine zojuist uitgevoerd? Licht je antwoord toe en geef de syntax van de instructie weer.
3. Wat is de code in machinetaal van deze instructie.
4. Wat is de opcode van de zojuist uitgevoerde instructie?
5. Hoeveel instructies zijn er al voor deze instructie geëxecuteerd?
6. Wat is het adres van de volgende instructie?
7. Waarvoor is de component 'Mux' nodig?
8. Welke componenten hebben de component 'Clock' nodig om de machine goed te laten werken?



Figuur 3.14



Hoofdstuk 7: Hoe werkt een “loopje” nu precies?

7.1 Leerdoelen

Als we een ‘if statement’ of ‘while loop’ programmeren in bijvoorbeeld een taal als C of Java, voert de machine een zogenaamde sprong- of branch-instructie uit. In de rekenmachines van hoofdstuk 3 werd na elke instructie de Program Counter met één verhoogd, waardoor de volgende instructie van het programma wordt uitgevoerd. Bij branch-instructies is dit niet het geval. De waarde van de Program Counter wordt bijvoorbeeld met drie verlaagd ofwel de Program Counter ‘springt’ drie instructies terug.

Aan het eind van dit hoofdstuk weet je:

- Hoe een Branch-instructie werkt;
- het mechanisme waardoor een programma naar een andere dan de volgende instructie kan springen;
- met welke componenten de Rekenmachine II hiertoe moet worden uitgebreid;
- wat de functie van iedere extra component is;
- waardoor de klokfrequentie van een computer wordt bepaald;
- waar de snelheid van een computer van afhangt;
- hoe je de waarde van een bepaalde bit uit een bitstring kan bepalen.

Aan het einde van dit hoofdstuk kun je:

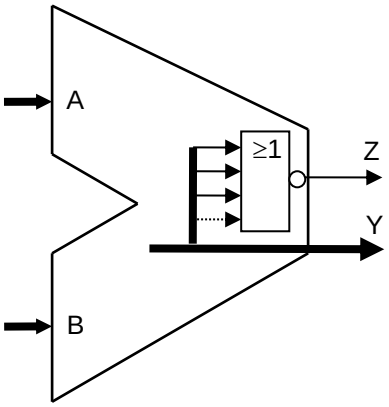
- de opcode van een Branch-instructie bepalen;
- een programma schrijven dat een vermenigvuldiging uitvoert door gebruik te maken van ‘shifting’ en ‘masking’;
- bepalen hoelang de executie van een programma duurt.

7.2 Benodigde extra hardware om Branch-instructies uit te voeren

De benodigde extra hardware bestaat uit:
1. componenten die de voorwaarden scheppen om een sprong mogelijk te maken. Dit zijn:
    a. de ALU krijgt een extra 1-bits uitgang, Zero genaamd;
    b. twee extra poorten: een AND-poort en een XOR-poort;
    c. de opcode wordt met twee bits uitgebreid: een Branch-bit en een Invert-bit;
    d. de Program Counter krijgt een 1-bits LoadPC-ingang en een 16-bits data-ingang;
2. en een extra Adder om het adres te berekenen waar de PC naar toe moet springen.

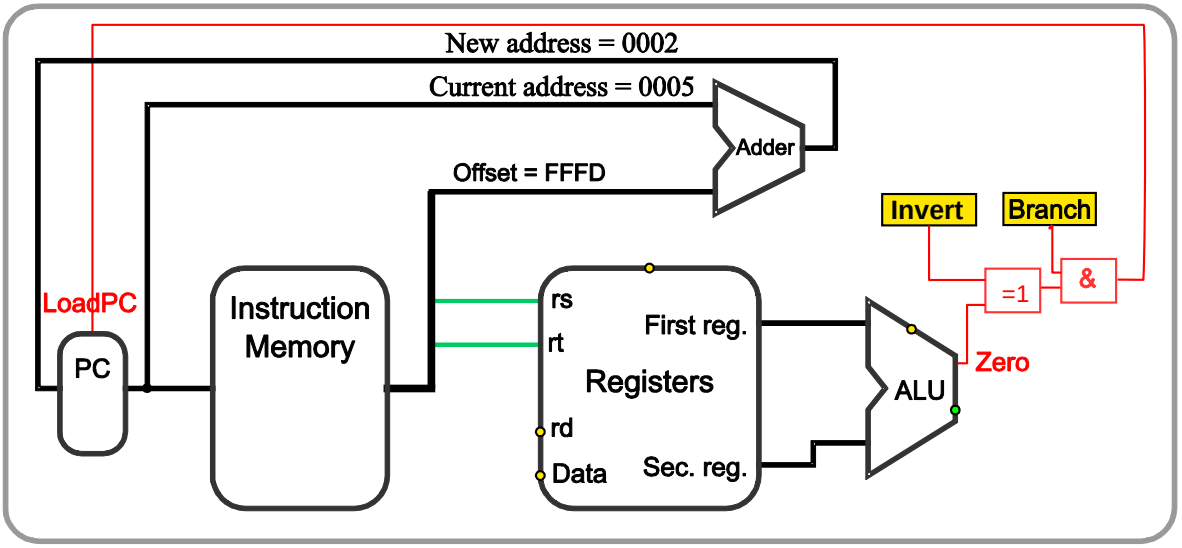
Waardoor kan een programma naar een andere instructie ‘springen’?

Zero-bit ALU
De uitgang van de ALU wordt voorzien van een Zero-bit ook Zero-flag genoemd. Het Zero-bit is dan en slechts dan 1 als de uitgang van de ALU de waarde 0 geeft, dus als alle 16 bits van de ALU-uitgang 0 zijn. Voor de implementatie is een NOR-poort met 16 ingangen gebruikt (NOR-poort is een OR-poort met een geïnverteerde uitgang; zie figuur 7.1).



Figuur 7.1: ALU met Zero-flag

AND-poort en XOR-poort voor detectie van sprongvoorwaarden
Om de PC naar een andere instructie te kunnen laten springen, is het nodig de architectuur uit te breiden met o.a. een AND-poort (zie figuur 7.2). De bovenste ingang van deze AND-poort is met de Branch-bit verbonden en alleen 1 bij een branch-instructie. Als de andere ingang ook 1 is dan is de AND-poort-uitgang 1 en vindt een ‘sprong’ plaats doordat deze uitgang met de LOADPC-ingang van de Program Counter is verbonden. Welke branch-instructie uitgevoerd wordt hangt af van de waarde van het Invert-bit. Als het Zero-bit 1 is en het Invert-bit 0, dan is de uitgang van de XOR-poort 1 en vindt bijvoorbeeld een sprong plaats bij een instructie als Branch Zero. Als het Zero-bit 0 is en het Invert-bit 1, dan is de uitgang van de XOR-poort ook 1 en vindt bijvoorbeeld een sprong plaats bij een instructie als Branch Not Zero.



Figuur 7.2: Datapad en adrespad van een branch-instructie

In formulevorm: LOADPC = Branch AND ((Zero AND NOT(Invert)) OR (NOT(Zero) AND Invert)). Dit is gelijk aan: Branch AND (Zero XOR Invert)

Een sprong in het programma wordt alleen uitgevoerd als er een Branch-instructie is (Branch-bit is 1) én als de onderste bit van de AND-poort 1 is. Dit laatste is het geval als:
1. de Zero-uitgang van de ALU 1 én het Invert-bit 0 is (bij o.a. de instructie Branch Zero) óf
2. de Zero-uitgang van de ALU 0 én het Invert-bit is 1 is (bij o.a. de instructie Branch Not Zero).

Naar welke instructie springt de Program Counter?
De uitgang van de AND-poort is verbonden met de LoadPC-ingang van de Program Counter. Als deze uitgang 1 is dan wordt op de eerstvolgende positieve klokflank de PC 'geladen' met de waarde op zijn16-bits data- ingang. Deze nieuwe waarde is het adres van de volgende instructie die wordt uitgevoerd. Om te bepalen waar de PC naar toe springt, is een optelschakeling, in figuur 7.2, Adder genoemd, toegevoegd. De optelschakeling berekent het nieuwe adres van het instructiegeheugen. Dit nieuwe adres is de som van het huidige adres (is huidige waarde van de PC) en de 'offset' die in het instructiegeheugen is opgeslagen.
Voorbeeld: (zie figuur 7.2) Stel de waarde van de PC 5 is. Als LoadPC 1 is, dan moeten er drie instructies worden 'teruggesprongen', dus van adres 5 naar adres 2. De offset moet in dat geval -3 zijn. In two's complement code is dit FFFD.

7.3 Instructieset van de 'Loopsmachine' en implementatie van conditionele statements en herhalings statements

Instructieset
De instructieset wordt uitgebreid met vijf Branch-instructies: BZ (Branch Zero), BNZ (Branch Not Zero), BEQ (Branch EQual), BNE (Branch Not Equal) en BRA (BRanch Always). De eerste vier instructies zijn zogenaamde conditionele instructies, dit houdt in dat de Program Counter met een nieuwe waarde wordt geladen als aan een bepaalde conditie is voldaan. De instructie BRA is een zogenaamde niet-conditionele instructie. Deze instructie laadt altijd de Program Counter met een nieuwe waarde.
Voorbeeld: Als voorbeeld van een conditionele instructie beschouwen we de instructie "Branch if Zero" (BZ). Deze instructie wordt gebruikt om de Program Counter (PC) naar een andere instructie dan de volgende te laten springen met als voorwaarde dat het getal in het Second Register 0 is. Dit Second Register is met de B-ingang van de ALU verbonden. De waarde op B wordt bij deze instructie doorgelaten naar de uitgang van de ALU. Als het getal op deze uitgang 0 is, dus als alle bits van de ALU-uitgang 0 zijn en het Invert-bit is 0, dan moet de PC springen. De volledige instructieset is in tabel 7.1 weergegeven.

Mnemonic	Betekenis	Voorbeeld	Betekenis
ADD rd, rs, rt	Optellen registers	ADD $5, $6, $7	r5 ← r6 + r7
SUB rd, rs, rt	Aftrekken registers	SUB $5, $6, $7	r5 ← r6 – r7
AND rd, rs, rt	Bitwise AND registers	AND $5, $6, $7	r5 ← r6 & r7
OR rd, rs, rt	Bitwise OR registers	OR $5, $6, $7	r5 ← r6 | r7
XOR rd, rs, rt	Bitwise XOR registers	AND $5, $6, $7	r5 ← r6 ^ r7
SHL rd, rs, rt	Shift Left register	SHL $5, $6, $7	r5 ← r6 << r7
SHR rd, rs, rt	Shift Right register	SHR $5, $6, $7	r5 ← r6 >> r7
COPY rd, rt	Copy register	COPY $3, $2	r3 ← r2
ADDI rd, rs, imm	Optellen register en const.	ADDI $5, $6, 0x1234	r5 ← r6 + 0x1234
SUBI rd, rs, imm	Aftrekken register en const.	SUBI $7, $6, 0x1234	r7 ← r6 - 0x1234
ANDI rd, rs, imm	Bitwise AND register en const	ANDI $5, $6, 0d34	r5 ← r6 & 0d34
ORI rd, rs, imm	Bitwise OR register en const.	ORI $5, $6, 0d34	r5 ← r6 | 0d34
XORI rd, rs, imm	Bitwise XOR register en const	XORI $5, $6, 0d34	r5 ← r6 ^ 0d34
SHLI rd, rs, imm	Shift Left register	SHLI $5, $6, 5	r5 ← r6 << 5
SHRI rd, rs, imm	Shift Right register	SHRI $5, $6, 5	r5 ← r6 >> 5
LOADI rd, imm	Laad constante in register	LOADI $1, 0x 0020	r1 ← 0x0020
BZ rt, label	Branch if rt gelijk is aan 0	BZ $6, end	If (r6 == 0) goto 'end'
BNZ rt, label	Branch if rt ongelijk is aan 0	BNZ $6, end	If (r6 != 0) goto 'end'
BEQ rs, rt, label	Branch if rs gelijk is aan rt	BEQ $6, $8, loop	If (r6 == r8) goto 'loop'
BNE rs, rt, label	Branch if rs ongelijk is aan rt	BNE $6, $8, loop	If (r6 != r8) goto 'loop'
BRA label	Branch always	BRA label	PC ← PC + offset

Tabel 7.1: Instructieset: Calculator with loops

Branch-instructies springen naar een **relatief** adres. Hiermee wordt bedoeld relatief t.o.v. de huidige waarde van de Program Counter. Als de offset bijvoorbeeld -3 is, dan is de nieuwe waarde van de Program Counter drie lager dan de huidige waarde. Dus is ook het nieuwe adres van de volgende instructie drie lager dan het huidige adres.
Bij de instructie *BEQ $2, $3*, *Label* is Label het adres waar de Program Counter naar toe moet springen. Dus Label is een **absoluut** adres! De SIM-PL Assembler berekent de waarde die label aanneemt.

Toelichting van de instructie: Branch if Equal
Een voorbeeld van een branch-instructie is de instructie 'Branch if EQual' (BEQ).
De syntax is: *BEQ rs, rt, Label*.
Voorbeeld: BEQ $2, $3, Label.
Betekenis: if (register2 == register3) goto Label.
Met een woord als 'Label' wordt aangegeven naar welke instructie de PC moet springen.
De voorwaarden waarbij bij deze instructie 'wordt gesprongen' zijn: Zero = 1; Branch is 1 en Invert is 0.
Vraag: Welke operatie moet de ALU bij een instructie BEQ uitvoeren? Antwoord:……

De grootste van twee getallen bepalen

In sorteerroutines worden getallen naar grootte gesorteerd. Hiervoor is het nodig van twee getallen te bepalen welke de grootste is. Dit gebeurt door de beide getallen van elkaar af te trekken en na te gaan of het resultaat een negatief getal oplevert. De ALU van de 'Loops machine' is echter niet voorzien van een extra uitgang N (Negative) die aangeeft of na een berekening op de uitgangen van de ALU een negatief getal staat. Een getal is negatief als de hoogste bit 1 is (zie hoofdstuk 2). Toch kan met een paar instructies de grootste van twee getallen worden bepaald. Hiervoor is o.a. een zogenaamde bittest nodig.

Testen van een bit
Voor het testen van een bit is de bitwise AND-operator nodig (zie hoofdstuk 2.7). Met deze operator kun je bepalen wat de waarde van een bepaalde bit is. Voorbeeld: van een getal b wil je weten of dit negatief is. Dit is zo als de hoogste bit 1 is. Bij een 8-bits getal is de hoogste bit b_7. Neem een bitstring waarvan alleen de te testen bit b_7 1 is. Alle andere bits zijn 0. Een dergelijke bitstring wordt een masker genoemd. Door een bitwise AND-operatie op b en het masker uit te voeren worden de andere bits van b 'gemaskeerd'. Alleen de waarde van de te testen bit is zichtbaar. Opdracht: Vul tabel 7.2 in.

b	01101000	11100010
masker	10000000	10000000
b AND masker		
b>0? ja/nee		

Tabel 7.2

BGT (Branch Greater Than)
In het tekstvenster staan drie instructies waarmee we een pseudo instructie 'Branch Greather Than' hebben gecreëerd. De syntax van deze pseudo instructies is: BGT a, b, label.

```
# BGT a, b, label          # if( a > b ) goto label
SUB temp, a, b
ANDI temp, temp, 0x8000    # temp is 0 of 0x8000
BZ temp, label             # temp 0? goto label
```

Implementatie van conditionele statements en herhalings statements

Hieronder zijn enkele voorbeelden van basisstructuren van programma's weergegeven. Links is van deze voorbeelden de code in de programmeertaal C weergegeven en rechts in de assembly code van de SIM-PL Loopsmachine.

if .. then .. else
Hieronder is weergegeven hoe we gebruik kunnen maken van een statement van de vorm:
If(condition) then statement_1 else statement_2.

```
if( i == j ) { a = a + b;
  }
else   {a = a – b;
}
```

```
BEQ i, j, if          # if( i == j ) goto label if
SUB a, a, b           # else
BRA end               # goto end
if: ADD a, a, b
end: HALT             # einde programma
```

De variabelennamen i, j, a, en b zijn aliassen (andere namen) voor bijv. registers $0, ., $4.

while loop
Een voorbeeld van een structuur als:
While(condition) do {block of code}.

```
while( a < b ) {
        a = a + 1;
        b = b – 1;
}
```

```
while:                  # if( a ≥ b ) goto done
   SUB temp, a, b
   ANDI temp, temp, 0x8000
   BZ temp, done

ADDI a, a, 1
SUBI b, b, 1
BRA while               # goto label while
done: HALT              # einde programma
```

for loop
Een voorbeeld van een for-loop:

```
int a = 0;
for( i = 0; i <= 4; i = i + 1 ) {
        a = a + 8;
}
```

```
LOADI tmp, 4            # i ← 4
LOADI a, 0              # a← 0
LOAD i, 0               # tmp ← 0
for:    BEQ i, tmp, end # if (i == 4) goto end
        ADDI a, a, 8    # (tussen)resultaat
        ADDI i, i, 1    # i = i +1
        BRA  for        # goto for
end: HALT               # einde programma
```

7.4 Waar is de snelheid van een computer van afhankelijk?

In deze paragraaf komen enkele basisbegrippen aan de orde die met deze vraag samenhangen.

Propagatietijd
Iedere poort of schakeling heeft een zogenaamde 'propagation delay time' t_{pd}. Dit is de tijd die verstrijkt tussen een signaalverandering aan één of meerdere ingangen en de reactie van de uitgang hierop. Dus iedere poort of schakeling geeft een signaal vertraagd door. De poorten in deze cursus hebben allen een t_{pd} van 1. Bij de huidige stand van de chiptechnologie komt 1 ongeveer overeen met 0,01 nanosec ($0{,}01*10^{-9}$ sec). De propagatietijd van een schakeling is de som van de propagatietijd van alle deelschakelingen die achtereenvolgens worden gebruikt bij het uitvoeren van een instructie. Voeren we bij de Loopsmachine de instructie ADD uit, dan worden achtereenvolgens de PC, het Instruction Memory, de registers, de Mux, de ALU en nogmaals de registers (voor het opslaan van het ALU-resultaat) gebruikt. De propagatietijden van al deze schakelingen moeten worden opgeteld om de totale propagatietijd van de schakeling te bepalen. Deze totale propagatietijd stellen we op iets minder dan 100 keer de tijd van één poort, dus totaal 100 * 0,01 ns = < 1 ns.

Frequentie
Frequentie geeft aan hoe vaak iets *in een bepaalde tijd* gebeurt. Frequentie wordt weergegeven in hertz (Hz). 1 Hz is één gebeurtenis per seconde. Frequentie is het omgekeerde van de duur van één, zich steeds herhalende, gebeurtenis.

Kristal-oscillator/klok
Een computersysteem gebruikt een klok om de schakelingen in de processor aan te sturen. Bij de Loopsmachine wordt de PC door een opgaande klokflank aangestuurd. Het terugschrijven van het ALU-resultaat naar register rd vindt plaats door een neergaande klokflank (zie fig. 3.7).

De 'Clock' van SIM-PL
De klok van de computer stuurt de schakelingen aan. In de SIM-PL simulatieomgeving kun je voor elke machine de duur van één klokpuls (Cycle) instellen. De duur van deze klokpuls moet zodanig worden gekozen dat een instructie geheel moet zijn afgehandeld voordat de volgende instructie mag beginnen. De duur van één klokpuls t is bij de machine van dit hoofdstuk 100 tijdstippen, dus 100 * de t_{pd} van 1 poort dus t = 100 * 0,01 ns = 1 ns.

De klokfrequentie $f = \frac{1}{t} = \frac{1}{1*10^{-9}} = 10^{9} = 1$ GHz.

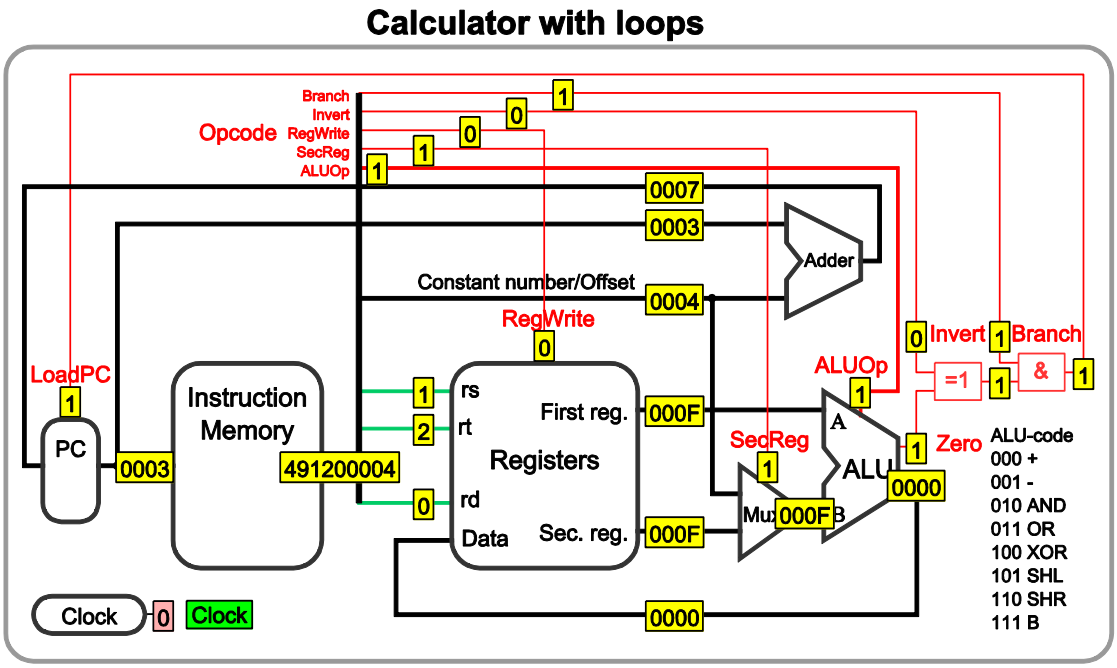
1 GHz = 1 gigahertz = 1 miljard Hz.

Doordat iedere instructie één klokpuls duurt, kunnen er dus 1 miljard instructies per seconde door de machine worden uitgevoerd[8]. Meestal wordt het aantal instructies per seconde weergegeven in 'Million Instructions Per Second (MIPS). De Loopsmachine kan dus 1000 MIPS uitvoeren. Het aantal MIPS van een machine zegt echter lang niet alles over de werkelijke prestaties van die machine.

Historisch perspectief: Wet van Moore
Gordon Moore, één van de oprichters van processorfabrikant Intel, stelde in 1965:
"Elke 18 maanden verdubbelt het aantal transistoren op een processorchip."
De eerste processoren waren 4-bits machines. Door het steeds toenemende aantal transistoren op een chip werd het mogelijk: 8-bits, 16-bits, 32-bits en 64-bits machines te bouwen waardoor berekeningen steeds sneller konden worden uitgevoerd.
Later is de wet van Moore opnieuw formuleert tot:
"Elke 18 maanden verdubbelt de prestatie van een processorchip."
Naast het *aantal* transistoren zijn er een aantal andere oorzaken dat deze wet uit 1965 in 2014 nog steeds geldig is. Eén van de oorzaken is de afnemende propagatietijd van transistoren. In de loop der jaren werden transistoren ontwikkeld die steeds sneller konden schakelen. De klokfrequentie kon hierdoor steeds hoger worden, waardoor het aantal MIPS steeds toenam. De laatste jaren neemt de klokfrequentie echter nauwelijks meer toe. 4 GHz lijkt een bovengrens voor de klokfrequentie te worden. Om toch de prestaties te laten toenemen, worden o.a. meerdere processoren op één chip geplaatst. Het originele artikel van Moore is te downloaden via: ftp://download.intel.com/research/silicon/moorespaper.pdf

7.5 Opgaven en practicum met de "Calculator with loops"

Opdracht 1: Onbekende instructie
In figuur 7.3 is de toestand van de machine weergegeven na het executeren van een onbekende instructie.



Figuur 7.3

Beantwoord de volgende vragen:
1. Wat is het volgende adres voor het Instructie Memory? Antwoord: ……..
2. Wat is de (decimale) waarde van de offset? Antwoord: …….

[8] We nemen hierbij aan dat het om een 'echte machine' gaat, dus niet om een simulatie van een machine.

3. Welke operatie voert de ALU uit? Antwoord: ……..
4. Welke instructie voert de machine uit? Geef de syntax ervan weer. Antwoord: ……
5. Leg uit hoe het sprongmechanisme bij deze instructie werkt. Antwoord: …… ……
6. Een register-writelijn (RegWrite) laadt de data aanwezig op de registerdata-ingangen in de registers als de waarde op de lijn 1 is op een neergaande klokflank. Waarom is bij deze machine een RegWrite-lijn nodig? RegWrite was bij het vorige model, de rekenmachine II, niet nodig.
   Antwoord:……..……..

Opgave 2: Instructieformaat Calculator with loops
Vul de ontbrekende velden van tabel 7.3 in.

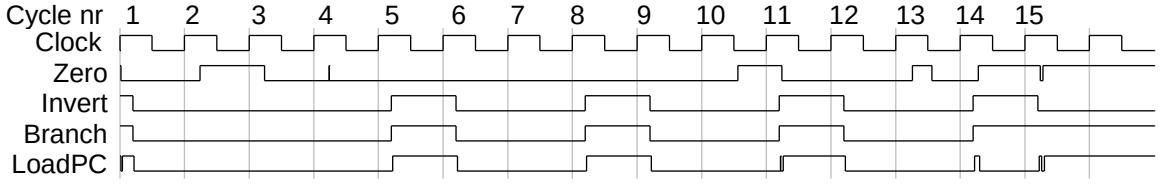
	Instructieformaat Loop-machine (totaal 33 bits)								
Instructie	Opcode					First Register rs	Second Register rt	Destination Register rd	Constant getal/ offset
	Branch	In-vert	Reg Wr	Sec Reg	ALU Op				
Aantal bits	1	1	1	1	3	4	4	4	16
ADD				1	000	rs	rt	rd	x
COPY				1	111	x	rt	rd	x
ADDI				0	000	rs	x	rd	getal
LOADI				0	111	x	x	rd	getal
BZ									
BNZ									
BEQ									
BNE									
BRA									

Tabel 7.3: Instructieformaat: Calculator with loops

Opdracht 3: Vermenigvuldigen door herhaald optellen
In het tekstvenster hiernaast is de broncode weergegeven van een vermenigvuldiging van het getal 8 met het getal 4. Dit wordt bereikt door de inhoud van register 0 steeds met 8 te verhogen totdat de waarde van register 6 gelijk is aan 0. Links van de broncode is het nummer van de klokpuls weergegeven die bij iedere instructie hoort.
In figuur 7.4 is het tijdvolgordediagram weergegeven dat wordt gegenereerd tijdens het executeren van het programma.

```
                @include "CalculatorWithLoops.wasm"
                # Vermenigvuldiging: 4 * 8 door herhaald optellen

                .code MyCode : LOOPS_MACHINE
Nr klokpuls
1               LOADI $6, 4     # register 6 ← 4
2               LOADI $0, 0     # (begin)resultaat ← 0

                loop:
3,6,9,12        ADDI $0, $0, 8 # (tussen)resultaat
4,7,10,13       SUBI $6, $6, 1 # inhoud r6 ← inhoud r6 -1
5,8,11,14       BNZ  $6, loop # if ( inhoud r6 != 0 ) goto loop
15              HALT            # einde programma
```

In dit diagram is het signaalverloop van de klok, het Zero-bit, het Invert-bit, het Branch- bit en LoadPC weergegeven.



Figuur 7.4 Time Sequence diagram van het bovenstaande programma

Beantwoord de volgende vragen:
1. Bij welke cycles is het Branch-bit 1? Antwoord: bij klokpulsen ……,……,…..
2. Bij welke cycles springt de Program Counter naar een ander adres dan het eerstvolgende? Antwoord: Bij klokpulsen ……, ……..,….
3. Hoeveel keer wordt de 'loop' doorlopen? Antwoord: ……
4. Waarom blijft LOAD-PC 0 tijdens de 14-de klokpuls? Antwoord ……..

5. Hoeveel clockcycles heeft het programma nodig om de vermenigvuldiging uit te voeren? Antwoord …..
6. Wat is de executietijd van het programma? Antwoord …..
7. Je kunt dit hele programma vervangen door één instructie. Welke instructie? Antwoord ……..

Spikes en glitches (zie figuur 7.4)
Bij het uitvoeren van de 14-de instructie wordt LOAD-PC heel even 1. Dit komt doordat de Zero-uitgang iets later 1 wordt dan het Invert-bit en het Branch-bit. Dit is het gevolg van de propagatietijden van de registers en de ALU. De stand van de Program Counter verandert hierdoor (gelukkig) niet. Alleen als LoadPC 1 is *tijdens een positieve klokflank*, verandert de stand van de PC.
Een korte signaalverandering zoals beschreven wordt een 'spike' genoemd. Tijdens de 15-de instructie gaat Zero kortstondig naar 0. Dit verschijnsel heeft een vergelijkbare oorzaak en wordt een glitch genoemd.

Opdracht 4: Open met de executer uit de folder: H7CalculatorWithLoops de file: CalculatorWithLoops.sim-pl-ws. Laad in de Program Editor de file 'Opdracht4.wasm'. Compile, Load dit programma. Run dit programma instructie voor instructie. Ga na of de uitkomst correct is. Vervang het label 'loop' in de broncode door het juiste adreswaarde en Run nogmaals het programma. Voer een aantal keren een Halt-instructie uit.
Vraag: Wat zijn de waarden van Invert, Branch, Zero en offset tijdens een Halt instructie?
Antwoord: Invert is ……, Branch is ……. , ALU Zero is ……en offset is ……

Beter leesbare programma's door voor registers aliassen te gebruiken.
In de Program Editor van SIM-PL is het mogelijk aan een registernummer een naam te geven. Het tekstvenster hiernaast laat zien hoe dat moet. Voeg voor de eigenlijke programmacode de volgende tekst toe: Hierdoor mag je in alle programmaregels waar '$0' voorkomt, deze vervangen door 'result'. Hetzelfde geldt voor '$6' en 'b'.

```
.data MyRegisters: REGISTERS
0x0: WORD result
0x6: WORD b
```

Ook moet de regel: *.code MyCode : LOOPS_MACHINE* vervangen worden door:
.code MyCode : LOOPS_MACHINE, MyRegisters
Het is ook mogelijk een register tijdens het laden van de assembly code al een waarde te geven bijvoorbeeld:
0x6: WORD b 4
Register 6 heeft de waarde 4 bij de start van het programma en de instructie: "LOADI b, 4" is in dat geval overbodig geworden.

Opdracht 5: aliassen voor registers
Save de file "opdracht4.wasm" als "opdracht5.wasm".
Voeg toe of verander alle in het tekstvenster cursief weergegeven tekst en pas ook de rest van de broncode aan zodat er geen registernummers meer in voorkomen.

```
@include "CalculatorWithLoops.wasm"
# Vermenigvuldiging: 8 * 4 door herhaald optellen

.data MyRegisters: REGISTERS
0x0: WORD result
0x6: WORD b

.code MyCode : LOOPS_MACHINE, MyRegisters
  LOADI b, 4        #  b ← 4
  LOADI result, 0   #  result ← 0

loop:
….
```

Opdracht 6: Vijf keer negen
Gebruik de "Save AS"-optie om de filename te wijzigen in Opdracht6.wasm. Wijzig het programma zodat 9 met 5 wordt vermenigvuldigd. Gebruik de instructie BNE i.p.v. BNZ. Maak gebruik van drie registers.

Opdracht 7: De grootste van twee getallen bepalen
Schrijf een programma dat twee getallen a en b in twee opeenvolgende registers plaatst. Het programma moet de volgende code uitvoeren: if(a > b) swap(a,b)[9]. Test het programma met geschikte waarden voor a en b. Pas de file Opdracht7.wasm aan.

[9] Swap(a, b): verwissel de inhoud van register a met die van register b.

Opdracht 8: Een efficiënter vermenigvuldigalgoritme dan herhaald optellen

```
Multiplicand  1210     Multiplicand  1010     Multiplicand  1010
Multiplier    1301     Multiplier    1101     Multiplier    1101
              1210                    1010                  1010
             0000                    0000                  0000
            3630                    1010                  01011
           1210                    1010                   1010
Product   1574210     Product   10000010                   01100
                                                          1010
                                                Product  10000010
```

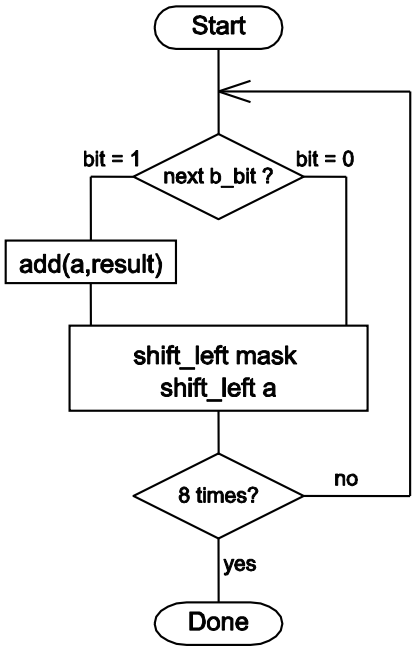
Vermenigvuldigen
Linksboven is een vermenigvuldiging weergegeven van de decimale getallen 1210 en 1301 volgens een methode die in het basisonderwijs wordt toegepast. De methode is:
Vermenigvuldig het meest rechtse cijfer van de vermenigvuldiger (multiplier) met het vermenigvuldigtal (multiplicand). Vermenigvuldig hierna het op één na meest rechter cijfer van de vermenigvuldiger met het vermenigvuldigtal en schrijf dit deelproduct schuin onder het deelproduct van de eerste vermenigvuldiging. Herhaal dit met de overige cijfers van de vermenigvuldiger. Tel alle deelproducten op en het product is bekend.
Met binaire getallen kun je dezelfde methode gebruiken. Middenboven is de vermenigvuldiging van de binaire getallen 1010 en 1101 weergegeven. Het resultaat is een 8-bits getal! Omdat een ALU slechts twee getallen tegelijk kan optellen, zijn meerdere optellingen nodig. Eerst worden de twee eerste deelproducten opgeteld. Bij de som hiervan wordt het derde deelproduct opgeteld etc. Voor het optellen van twee 4-bits getallen is dus drie keer een optelling nodig. Rechtsboven is dit weergegeven.

Er zijn ook toepassingen waarbij d.m.v. een programma een vermenigvuldiging wordt uitgevoerd. Hiervoor zijn een *bittest en een 'shift_left'-instructie* nodig (zie hoofdstuk 3).

Het aantal optellingen kan worden gereduceerd door te vermijden dat 0000 bij een tussenresultaat wordt opgeteld. Het bovenstaande voorbeeld ziet er na deze reductie als volgt uit:

```
Multiplicand  1010
Multiplier    1101
              1010
             101000
            0110010
            1010000
Product   10000010
```

Opdracht: Schrijf een programma waarmee twee positieve getallen van 8 bits kunnen worden vermenigvuldigd. Een 'stroomdiagram' van het programma is in figuur 7.5 weergegeven.



Figuur 7.5

Wat is de executietijd van het programma? De executietijd is ….....ns.

7.6 Begrippenlijst

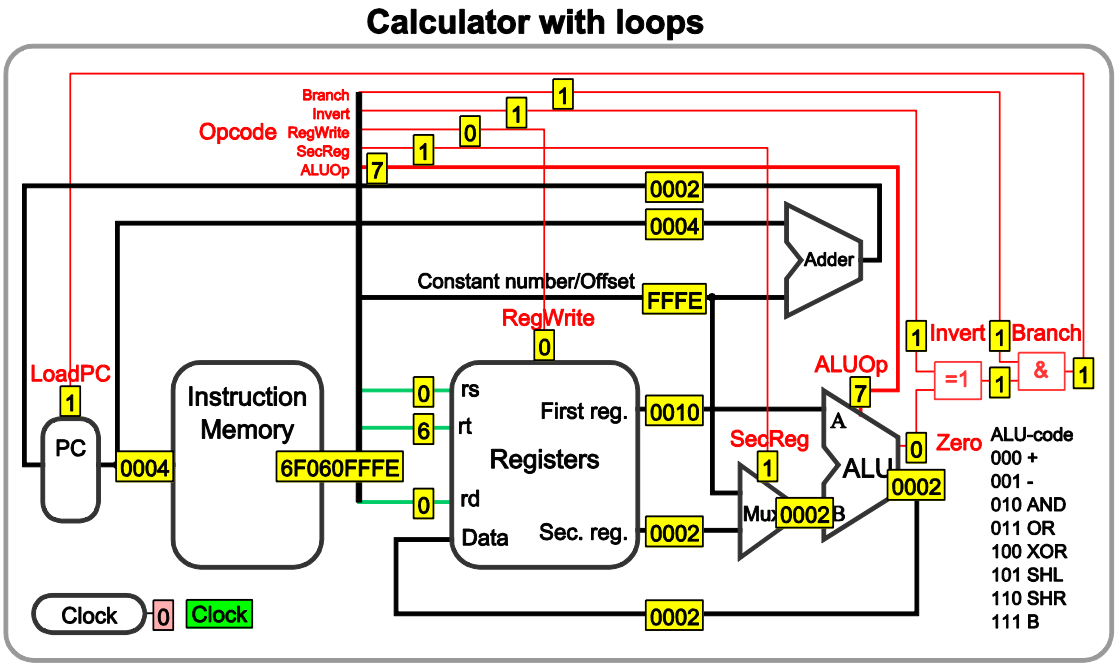
Branch-instructie. Type instructie waardoor de PC, meestal afhankelijk van een ALU-conditie, een sprong in het programma kan maken.

Zero-bit. Een ALU-uitgang die aangeeft of het ALU-resultaat na een berekening 0 is.

Invert-bit. Een bit in de opcode dat het zero-bit al dan niet inverteert.

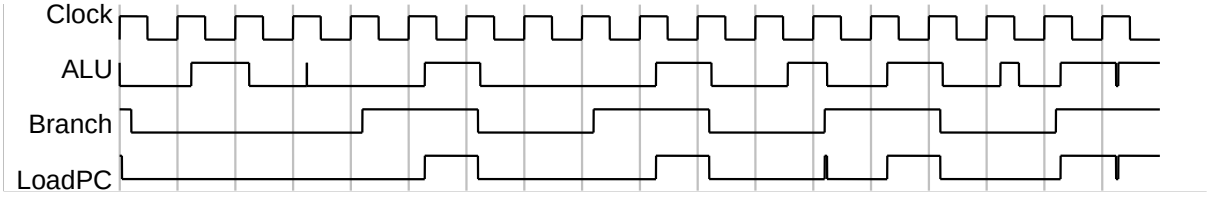
7.7 Tentamenvragen

1. Leg uit op welke manier de executie van een ‘loopje’ precies plaatsvindt. Beschouw hierbij de machine afgebeeld in figuur 7.6.
2. Geef een expressie in termen van Boole-algebra voor de ALU-uitgang Zero.
3. Beschouw figuur 7.6.
    a. Waar is de component “Adder” voor nodig?
    b. Op welk adres staat de volgende instructie die wordt uitgevoerd?
    c. Wat is de “Offset”?
    d. Welke operatie voert de ALU uit?
    e. Welke instructie wordt uitgevoerd? Geef de syntax ervan weer.
    f. Waarvoor is een “RegWrite”-ingang nodig bij de component registers?
    g. Uit hoeveel bits bestaat de inhoud van één adres van het Instruction Memory?



Figuur 7.6

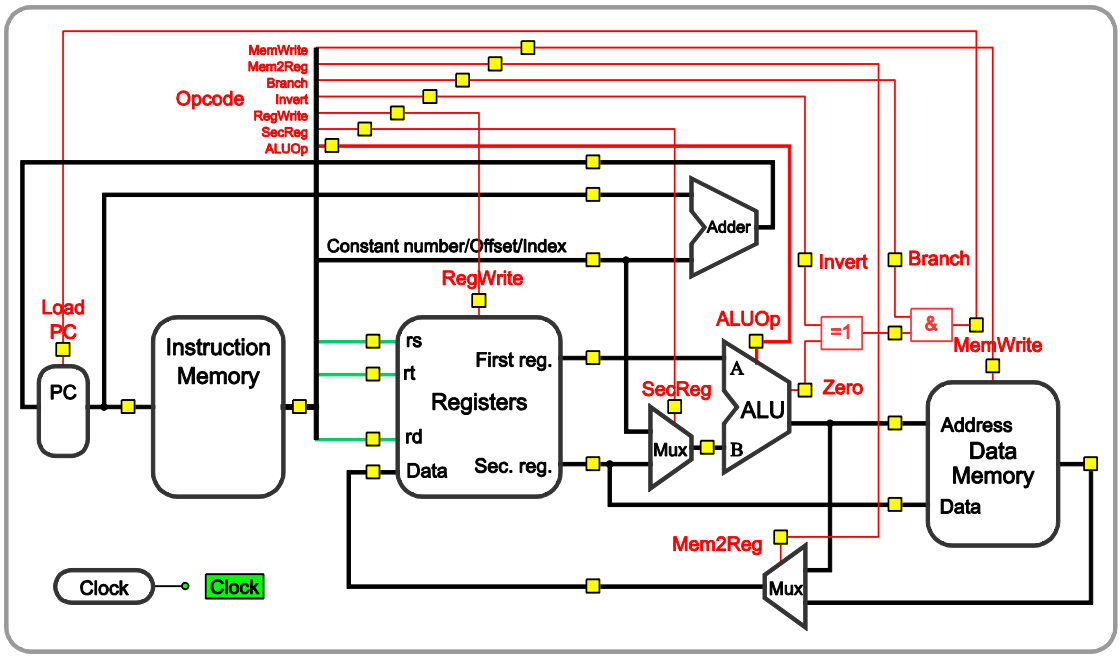
4. Beschouw het tijdvolgordediagram van figuur 7.7.
    a. Hoeveel keer wordt een branch-instructie aangeroepen?
    b. Hoeveel keer springt de PC hierdoor naar een andere instructie dan de eerstvolgende?



Figuur 7.7 Time Sequence diagram van een programma

5. Waarom kan een instructie BEQI (Branch if Equal Immediate), bijvoorbeeld BEQI $4, 5, label op de machine van figuur 7.6 niet worden geïmplementeerd? Antwoord …….

16 bit Harvard Architecture



Hoofdstuk 8: De Harvard Single Cycle processor

8.1 Leerdoelen

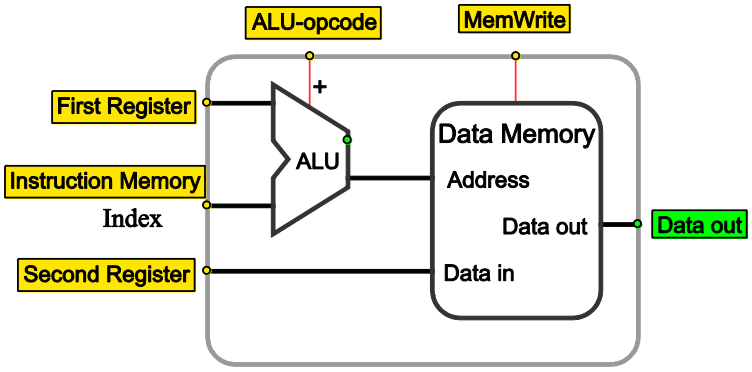
In de bovenstaande figuur is een complete processor volgens het Harvard model weergegeven. De kern van alle moderne processoren bestaat uit een dergelijke eenvoudige architectuur. Een Harvard processor heeft één geheugen voor het bewaren van het programma en een tweede geheugen voor het bewaren van de gegevens. Dit laatste geheugen wordt het Data Memory genoemd. Het uitvoeren van één instructie bestaat uit een aantal datatransfers tussen PC, Instruction Memory, Registers, ALU en Data Memory. Een groot voordeel van de Harvard architectuur is dat al deze datatransfers binnen één klokpuls worden uitgevoerd. Dus voor iedere instructie is slechts één klokpuls nodig. Een dergelijke machine wordt een Single Cycle machine genoemd. Een datatransfer wordt ook een (instructie)fase genoemd.

Aan het eind van dit hoofdstuk weet je:
- hoe een Harvard-processor werkt;
- wat een Data Memory is;
- wat een Load/Store machine is;
- dat iedere instructie uitgevoerd wordt in een aantal fases;
- dat van het instructietype afhangt welke fases dat zijn;
- wat de CPU time is;
- wat Amdahl´s law inhoudt,

Aan het einde van dit hoofdstuk kun je:
- een programma schrijven dat gegevens opgeslagen in het Data Memory kopieert naar andere geheugenlocaties;
- de CPU-tijd van een programma zowel meten als berekenen;
- een programma een reeks getallen laten sorteren naar grootte, gebruikmakend van de zogenaamde bubble-sort methode.

8.2 Uitbreiding hardware

Data Memory
Aan de machine van het vorige hoofdstuk is een Data Memory toegevoegd. Dit datageheugen heeft 2^{16} geheugenplaatsen. Om gegevens naar ieder van deze 2^{16} geheugenplaatsen te kunnen schrijven of uit ieder van deze 2^{16} geheugenplaatsen te kunnen lezen is dus een een 16-bits adres nodig. Dit adres is niet opgeslagen in een register maar bestaat uit twee componenten: de inhoud van het First Register en de index. De ALU berekent het adres door deze waarden op te tellen. Het Data Memory heeft aparte in- en uitgangen voor data, Data in en Data out genaamd. Data in is verbonden met het Second Register rt. Data out is via een multiplexer verbonden met de registers. Om gegevens naar het geheugen te kunnen schrijven, is een extra bit 'Memory Write' (MW) nodig. Alleen als deze bit 1 is wordt naar het geheugen geschreven. Op de uitgang Data out staat altijd de inhoud van het adres. Een 'Memory Read-bit' die aangeeft dat er uit een geheugenplaats wordt gelezen is bij deze machine niet nodig.



Figuur 8.1: ALU and Data Memory

Een extra multiplexer
Bij de machine van het vorige hoofdstuk was de uitgang van de ALU altijd verbonden met de data-ingang van de registers. Bij deze machine kan de data zowel afkomstig zijn van de ALU als van het Data Memory. Daarom is een extra multiplexer nodig met de naam Mem → Reg. Deze multiplexer wordt gestuurd door de bit M2R (Memory to Register).
Voor de M2R-bit geldt:

- M2R = 1: data van het geheugen naar één van de registers;
- M2R = 0: data van de ALU naar één van de registers.

8.3 De instructies Load Word en Store Word en de instructieset van de Harvard machine

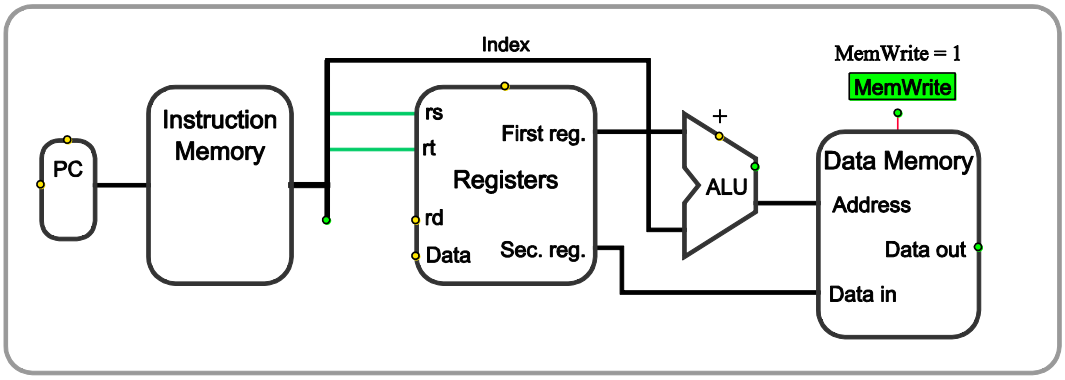
De communicatie tussen het Data Memory en de rest van de machine gebeurt met slechts *twee* instructies:

1. Store Word (SW);
2. Load Word (LW).

Deze machine wordt daarom een *Load/Store machine* genoemd.

Store Word
De instructie Store Word transporteert data van het Second Register rt, naar het datageheugen. Het geheugenadres waar de data naar toe moet, wordt bepaald door de som van het First Register en een constante uit het Instruction Memory. Deze constante wordt *index* genoemd.

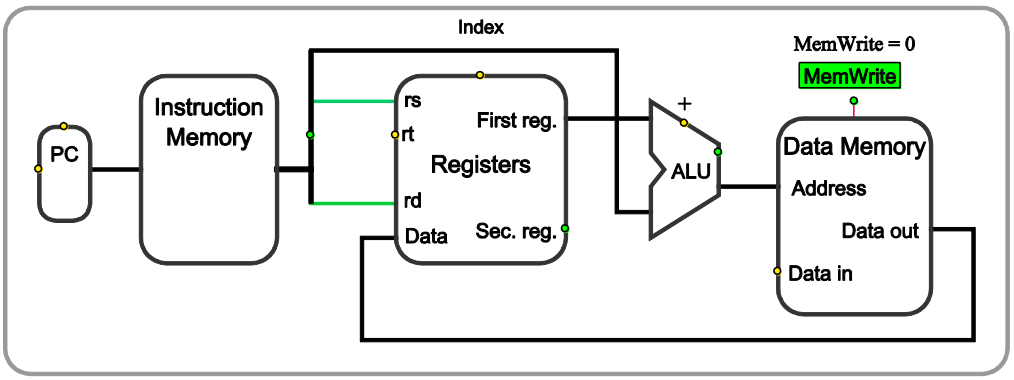
De syntax van een Store Word-instructie is: SW rt, index, rs.
Voorbeeld: SW $0, 0x1FD, $1.
Betekenis: Register 0 → Geheugenadres($1FD_{Hex}$ + Register 1).



Figuur 8.2: Datapad Store Word-instructie

Load Word

De instructie Load Word transporteert data van het datageheugen naar één van de registers. Het geheugenadres wordt op dezelfde manier bepaald als bij een Store Word instructie.



Figuur 8.3: Datapad Load Word-instructie

De syntax van een Load Word-instructie is: LW rd, index, rs.
Voorbeeld: LW $0, 0x1FD, $1.
Betekenis: Register 0 ← Geheugenadres($1FD_{Hex}$ + register 1).

Instructieset
In tabel 8.1 is de instructieset van de Harvard Single Cycle machine weergegeven.

Mnemonic	Betekenis	Voorbeeld	Betekenis
ADD rd, rs, rt	Optellen registers	ADD $5, $6, $7	r5 ← r6 + r7
SUB rd, rs, rt	Aftrekken registers	SUB $5, $6, $7	r5 ← r6 – r7
AND rd, rs, rt	Bitwise AND registers	AND $5, $6, $7	r5 ← r6 & r7
OR rd, rs, rt	Bitwise OR registers	OR $5, $6, $7	r5 ← r6 | r7
XOR rd, rs, rt	Bitwise XOR registers	AND $5, $6, $7	r5 ← r6 ^ r7
SHL rd, rs, rt	Shift Left register	SHL $5, $6, $7	r5 ← r6 << r7
SHR rd, rs, rt	Shift Right register	SHR $5, $6, $7	r5 ← r6 >> r7
COPY rd, rt	Copy register	COPY $3, $2	r3 ← r2
ADDI rd, rs, imm	Optellen register en const.	ADDI $5, $6, 0x1234	r5 ← r6 + 0x1234
SUBI rd, rs, imm	Aftrekken register en const.	SUBI $7, $6, 0x1234	r7 ← r6 - 0x1234
ANDI rd, rs, imm	Bitwise AND register en const	ANDI $5, $6, 0d34	r5 ← r6 & 0d34
ORI rd, rs, imm	Bitwise OR register en const.	ORI $5, $6, 0d34	r5 ← r6 | 0d34
XORI rd, rs, imm	Bitwise XOR register en const	XORI $5, $6, 0d34	r5 ← r6 ^ 0d34
SHLI rd, rs, imm	Shift Left register	SHLI $5, $6, 5	r5 ← r6 << 5
SHRI rd, rs, imm	Shift Right register	SHRI $5, $6, 5	r5 ← r6 >> 5
LOADI rd, imm	Laad constante in register	LOADI $1, 0x 0020	r1 ← 0x0020
BZ rt, label	Branch if rt gelijk is aan 0	BZ $6, end	If (r6 == 0) goto ‘end’
BNZ rt, label	Branch if rt ongelijk is aan 0	BNZ $6, end	If (r6 != 0) goto ‘end’
BEQ rs, rt, label	Branch if rs gelijk is aan rt	BEQ $6, $8, loop	If (r6 == r8) goto ‘loop’
BNE rs, rt, label	Branch if rs ongelijk is aan rt	BNE $6, $8, loop	If (r6 != r8) goto ‘loop’
BRA label	Branch always	BRA label	PC ← PC + offset
SW rt, index, rs	Store Word to memory	SW $0, 0x1234, $1	r0 → Mem(r1 +1234_{Hex})
LW rd, index, rs	Load Word to register	LW $0, 0x1234, $1	r0 ← Mem(r1 +1234_{Hex})

Tabel 8.1: Instructieset van de Harvard Single Cycle processor

8.4 Operanden die in het geheugen staan

De inhoud van een geheugenplaats bewerken
Veronderstel dat een lijst met gegevens bestaande uit 10 getallen is opgeslagen in het Data Memory op de adressen 50 t/m 59. Hoe kunnen we nu het getal op bijvoorbeeld adres 52 met 10 verhogen? Er zijn namelijk geen instructies om direct een berekening op een geheugenplaats uit te voeren. Daarom moet eerst het getal op adres 52 naar één van de registers worden getransporteerd. Hiervoor kiezen we register $1. Het adres 52 laten we uit twee delen bestaan:

1. het beginadres van het blok met gegevens, hier adres 50. Dit adres wordt bewaard in $2;
2. het verschil tussen het adres en het beginadres, hier 2. Dit wordt de index genoemd.

Nadat het getal van het geheugen naar een register is gekopieerd, voeren we de berekening uit en tenslotte wordt het berekende getal teruggeschreven naar geheugenplaats 52.

```
# Assembly code voor de Harvard machine van de instructie: A[2] =  A[2] + 10;
LOADI $2, 50          # Het beginadres van het blok met gegevens.
LW $1, 2, $2          # De waarde op geheugenplaats 52 gaat naar register 1.
ADDI $1, $1, 10       # De berekening wordt uitgevoerd.
SW $1, 2, $2          # De berekende waarde teruggeschreven naar het geheugen.
```

Wat veel voor komt moet snel worden uitgevoerd!
Bewerkingen op lijsten en array’s komen veelvuldig voor. Er is daarom voor gekozen het geheugenadres uit twee componenten, het beginadres van de lijst en de index van de lijst, te laten bestaan die door de ALU worden samengevoegd tot het geheugenadres.

Getallen in een lijst schrijven
Een tweede voorbeeld is een programma waarbij de getallen 1, 2, 4, 8, 16 en 32 op de respectievelijke geheugenplaatsen 10 t/m 15 van het Data Memory worden geschreven. Net als bij de registers is het ook bij het Data Memory mogelijk aliassen voor adressen te gebruiken. Ook kunnen in SIM-PL geheugenlocaties van gegevens worden voorzien aan het begin van de uitvoering van het programma net zoals dat bij de registers het geval is.

```
@include "16bitHarvard.wasm"
# Stores the numbers 1, 2, 4, 8, 16, 32 in
# memory locations 10 .. 15.

# number = 1;
# for( i = 0; i < 6; i = i + 1 ) { a[i] = number;  number = number + number; }

.data MyMemory : DATAMEM
10 :    WORD base               # base address array

.data MyRegisters : REGISTERS
0:      WORD i                  # loopindex
1:      WORD number
2:      WORD six

.code MyCode  : HARVARD, MyMemory, MyRegisters
        LOADI i, 0              # index = 0
        LOADI number, 1         # first number
        LOADI six, 6            # 6 numbers

for:    SW number, base, i      # store number at address (base + i)
        ADD number, number, number
        ADDI i, i, 1            # next number and next address
        BNE i, six, for         #  < 6 numbers stored?

end: HALT
```

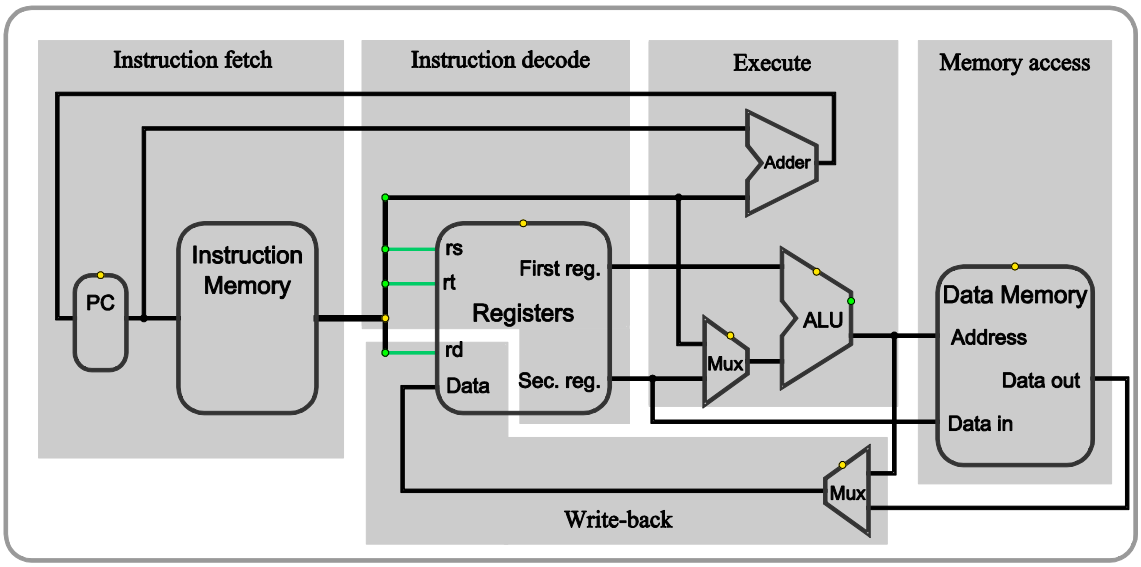
8.5 Executietijd, klokfrequentie, performance en Amdahl's law

Stel dat de verschillende componenten van een Single Cycle machine zoals de Harvard machine de volgende propagatietijd hebben:

- Memory units (Instruction Memory en Data Memory): 0,2 ns (1 nanosecond = 10^{-9} sec.).
- ALU: 0,2 ns.
- Register file (read and write): 0,2 ns.
- De delay time van de andere componenten is te verwaarlozen.

Executietijd van een Load Word-instructie
Iedere instructie kent verschillende fases die steeds een ander deel van de hardware gebruiken.
De relatie tussen de betreffende fase en de daarbij gebruikte hardware (zie figuur 8.4) en bijbehorende delay time is de volgende:

- **Instruction fetch:** de PC wijst de instructie aan en deze verschijnt op de uitgangen van het Instruction Memory. Dit duurt 0,2 ns.
- **Instruction decode:** de instructie wordt uiteengerafeld in adresvelden, opcode en de (constante) index. Op de uitgang van het first register verschijnt de waarde die bij rs hoort. Dit duurt 0,2 ns (register read).
- **Execute:** de ALU berekent het geheugenadres. Dit duurt 0,2 ns.
- **Data Memory access:** De waarde op dit adres verschijnt op de uitgang van het Data Memory. Dit duurt 0,2 ns.
- **Write-back:** Deze waarde wordt geschreven naar één van de registers. Dit duurt 0,2 ns (register write).



Figuur 8.4: Breaking the hardware of the machine into five parts

Iedere fase duurt 0,2 ns. De totale tijd nodig om de instructie Load Word uit te voeren is dus 1 ns. Een Single Cycle machine voert de instructie Load Word uit in één cycle. De som van al deze propagatietijden wordt de cycle time[10] t_c genoemd. Deze is dus 1 ns. De hoogst mogelijke klokfrequentie f_{clock} waarmee de machine kan worden aangestuurd, clock rate genaamd, is de inverse hiervan.

$$f_{clock} = \frac{1}{t_c} = \frac{1}{1*10^{-9}} = 1*10^{9} = 1\,\mathrm{GHz}.$$

Executietijd van een Branch Equal-instructie
Wat is nu de totale tijd die de instructie Branch Equal nodig heeft om te worden uitgevoerd? Voor deze instructie geldt:

- **Instruction fetch:** Instructie Memory lezen; 0,2 ns.
- **Instruction decode:** registers lezen en instructie decoderen; 0,2 ns.
- **Execute:** ALU branchvoorwaarde laten bepalen en Adder offset laten berekenen en PC aanpassen; 0,2 ns.

Er wordt bij deze instructie geen gebruik gemaakt van het Data Memory en er wordt ook geen waarde teruggeschreven in één van de registers. Kan deze instructie worden uitgevoerd in 0,6 ns? Qua hardwaregebruik wel, maar niet in een Single Cycle Harvard machine. Er zijn machines die dat wel kunnen, zogenaamde *Multicycle machines.*

Bij een Single Cycle zoals de Harvard machine duurt de executietijd van iedere instructie even lang als de instructie die de meeste tijd in beslag neemt.

CPU time
De CPU executietijd van een programma (CPU time) is gelijk aan het aantal cycles dat het programma nodig heeft maal de cycle time. Voor een Single Cycle machine geldt dat de CPU executietijd van een programma gelijk is aan het aantal uitgevoerde instructies (Instruction count) maal de cycle time. De CPU time is ook gelijk aan het aantal instructies gedeeld door de Clock rate.

$$\text{CPU time} = \text{Instruction count} * \text{Clock cycle time} = \frac{\text{Instruction count}}{\text{Clock rate}}$$

[10] Cycle time t_c is duur van één klokperiode.

Performance en performance ratio

Om systemen qua prestatie te kunnen vergelijken is het begrip *performance* ingevoerd. Hoe korter de executietijd is van een programma, dat draait op systeem P, des te hoger is de performance van dit systeem. In formulevorm: $\text{Performance}_P = \frac{1}{\text{CPUtime}_P}$.

De *performance ratio* n van een systeem Q t.o.v. een systeem P is, bij executie van hetzelfde programma: $n = \frac{\text{Performance}_Q}{\text{Performance}_P} = \frac{\text{CPUtime}_P}{\text{CPUtime}_Q}$.

Amdahl's law

Stel dat de propagatietijd van de ALU verlaagd wordt van 0,2 ns naar 0,1 ns, een verbetering met een factor 2. Wat betekent dit voor de CPU time?
Herhalen we de berekening van de propagatietijd voor het uitvoeren van een Load Word-instructie, dan neemt deze af van 1 ns naar 0,9 ns. De Clock cycle time wordt nu ook 0,9 ns. De Clock rate gaat omhoog met 11% naar 1,1 GHz.
Heeft het nut de ALU nog sneller te maken zonder de andere componenten te verbeteren? Een nieuwe halvering van de propagatietijd van de ALU zal de CPU time nog maar met 5,5% doen verbeteren. Als de delay time van de andere componenten hetzelfde blijft, heeft het dus relatief weinig nut om de propagatietijd van de ALU verder te verkorten.

Amdahl's law: The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

In formulevorm:
Stel P is het deel van de hardware dat is verbeterd en S de factor hoeveel dit deel is verbeterd, dan is de performanceverbetering (speed up) van een systeem = $\frac{1}{1 - P + \frac{P}{S}}$

Voor het voorbeeld van de ALU geldt: P = 0.2 en S = 2. Vullen we deze waarden in dan bereiken we performanceverbetering met een factor: $\frac{1}{(1-0.2) + \frac{0.2}{2}} = \frac{1}{0.9} = 1.11.$

Indien twee componenten P1 en P2 van een machine worden verbeterd met een factor S1 resp. S2 dan wordt de verbetering van de performance: $\frac{1}{1 - (P1 + P2) + \frac{P1}{S1} + \frac{P2}{S2}}$.

8.6 Opgaven en pract. met de Harvard Single Cycle machine

Opdracht 1: Gebruik figuur 8.4 om voor ieder type instructie na te gaan welke fases worden gebruikt. Vul hiertoe tabel 8.2 verder in (J betekent dat een fase wordt gebruikt, N dat deze niet wordt gebruikt).

Fase / Type instr.	Instruction fetch	Instruction decode	Execute	Memory Access	Write Back
Aritmetisch (bijv. ADD)	J	J	J	N	J
Immediate (bijv. ADDI)					
Branch					
Store Word					
Load Word					

Tabel 8.2: Type instructie met bijbehorende fases

Opdracht 2: Opcode en instructieformaat van de Harvard Single Cycle machine

Vul de ontbrekende velden in tabel 8.3 in.

	Instructieformaat 16 bit Harvard Single Cycle machine (totaal 37 bits)										
Instructie	Opcode							First Register rs	Second Register rt	Dest. Reg. rd	getal/ offset/ index
	MW	M2R	Br	Inv	RW	Sec Reg	ALU				
Aant. bits	1	1	1	1	1	1	3	4	4	4	16
ADD			0	x	1	1	0	rs	rt	rd	x
COPY			0	x	1	1	7	x	rt	rd	x
ADDI			0	x	1	0	0	rs	x	rd	getal
LOADI			0	x	1	0	7	x	x	rd	getal
BZ			1	0	0	1	7	x	rt	x	offset
BNE			1	1	0	1	1	rs	rt	x	offset
BRA			1	0	0	1	1	0	0	x	offset
SW											
LW											

Tabel 8.3: Instructieformaat: Harvard Single Cycle machine

Opdracht 3: Instructie LWI
Er zijn bij de Harvard machine slechts twee instructies waarmee met het Data Memory wordt gecommuniceerd. We willen een derde instructie toevoegen: “Load Word Indexed” (LWI). LWI transporteert data van het Data Memory naar een register. Het geheugenadres wordt bepaald door de som van het first register rs en het second register rt. De syntax van LWI is: LWI rd, rs, rt. Bepaal de opcode van deze instructie. Vul hiertoe tabel 8.4 in.

	Instructieformaat 16 bit Harvardmachine (totaal 37 bits)										
Instructie	Opcode							First Register rs	Second Register rt	Destination Register rd	getal/ offset/ index
	MW	M2R	B	Inv	RW	Sec Reg	ALU				
Aant. bits	1	1	1	1	1	1	3	4	4	4	16
LWI											

Tabel 8.4: Instructieformaat: Harvard Single Cycle machine

Opgave 4: Cycle time en performance ratio
Machine A heeft de waarden zoals hieronder zijn weergegeven.

- Memory units 0,2 ns.
- ALU: 0,2 ns.
- Register file (read or write): 0,2 ns.
- De propagation delay van andere componenten is te verwaarlozen.

Voor een verbeterde versie van machine A, machine B genaamd geldt:

- Memory units: 0,2 ns.

- ALU: 100 ps (ps = picosec = 10^{-12} sec).
- Register file (read or write): 150 ps.
- De propagation delay van andere componenten is te verwaarlozen.

Wat is de cycle time van machine B? Antwoord ……
Wat is de performanceverbetering van machine B t.o.v. machine A.
Antwoord: ……

Opgave 5: Instruction count
Er wordt uitgegaan van dezelfde machines als bij opgave 4.
Hieronder is de source code van een programma weergegeven.

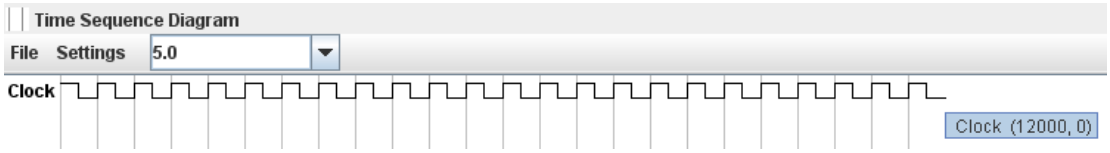
Beantwoord de volgende vragen:
Hoeveel keer wordt de loop doorlopen? Antwoord ….
Wat is het totale aantal instructies dat wordt geëxecuteerd? Antwoord ….
Wat is de CPU time als het programma op machine A draait? Antwoord ……
Wat is de CPU time als het programma op machine B draait? Antwoord ……

Bereken de performance ratio n voor de eerder genoemde machines A en B.
Antwoord: …….

```
@include "16bitHarvard.wasm"
# Add 5 values stored in memory locations 2 .. 6
# and store the result in the first free location (7)

.data MyMemory : DATAMEM
0x2 :    WORD  start    0x6
         WORD           0x7
         WORD           0x8
         WORD           0x9
         WORD           0d10

.code MyCode : Harvard, MyMemory
         LOADI $5, 5        # 5 values
         LOADI $6, 0        # Start at element 0
         LOADI $0, 0        # Clear $0

loop:    LW $1, start, $6
         ADD $0, $0, $1
         ADDI $6, $6, 1
         BNE $6, $5, loop

         SW $0, start, $6
```

Opgave 6: Meten van de CPU time met SIM-PL
Hieronder is het Time Sequence Diagram dat ontstaat tijdens het executeren van een programma weergegeven. Als we de muis bij het einde van dit diagram houden dan lezen we de waarden 12000 en 0. Dit houdt in dat op tijdstip 12000 de waarde van het signaal 0 is. We nemen aan dat:

- Op deze laatste opgaande flank wordt de laatste instructie van het programma uitgevoerd;
- De Clock cycle time is 500 SIM-PL tijdstippen;
- Het aantal cycles is: 12000/500 = 24 cycles;
- Dus de Instruction count = 24;
- Stel dat 1 SIM-PL tijdstip overeen komt met 0,4 ps;
- De Clock cycle time is dan 500 * 0,4 = 200 ps;
- De gemeten CPU time = Instruction count * Clock cycle time = 24 * 200 = 4800 ps = 4,8 ns;



Meten van de CPU time

Start de Executer en open components → H8HarvardArchitecture de worksheet 16bitHarvard.sim-pl-ws. Open the file: Opgave5.wasm. ‘Run’ het programma. Hoeveel klokpulsen zijn hiervoor nodig? Antwoord: ……. Wat is de CPU time?
Is het antwoord in overeenstemming met het resultaat van opgave 5 voor machine A?
Antwoord ……

Opdracht 7: 'Copy block'
Schrijf een programma dat getallen, opgeslagen in vijf opeenvolgende geheugenplaatsen, kopieert naar een ander deel van het geheugen. De geheugenplaats die volgt op die van het laatste getal, bevat het getal 0. Het eerste getal staat op adres: 0x10. Dit getal moet worden gekopieerd naar adres 0x20. Het volgende getal op adres 0x11 moet worden gekopieerd naar adres 0x21 etc. Het programma stopt met kopiëren zodra het getal 0 verschijnt. Dit getal moet ook worden gekopieerd.
Hoeveel klokpulsen zijn er nodig om het programma uit te voeren? Antwoord …….klokpulsen.
Wat is de CPU time van dit programma? Antwoord …….. msec.
Neem hierbij aan dat de propagation delay van de Memory units, ALU en Register file (read and write) 0,2 ns is. De propagation delay van andere componenten is te verwaarlozen.
Deze antwoorden worden gebruikt in hoofdstuk 9.

Opgave 8: Bubble sort
De opgave is om een lijst van vijf getallen te sorteren naar grootte. Na het sorteren staat het grootste getal aan het einde van de lijst. Maak gebruik van het zogenaamde "Bubble sort algoritme". Hierbij zien we dat grotere getallen als "bellen" naar het einde van de lijst worden verplaatst.
Het algoritme werkt als volgt: eerst worden de twee getallen aan het begin van de lijst vergeleken en verwisseld als ze in een verkeerde volgorde staan. Hierna wordt hetzelfde gedaan met de getallen op plaats twee en plaats drie en daarna nogmaals met de getallen op plaats drie en plaats vier. Dit wordt herhaald tot het einde van de lijst. Het grootste getal staat nu aan het einde van de lijst. De hele procedure wordt herhaald, maar nu tot en met het voorlaatste getal. De hele procedure wordt nogmaals herhaald, maar nu tot en met het voor voorlaatste getal. Zo gaan we door totdat de hele lijst gesorteerd is[11)]. Je kunt twee getallen a en b qua grootte vergelijken door ze eerst van elkaar af te trekken. Van het resultaat controleer je of het hoogste bit 1 is door de instructie ANDI en het masker 0x8000 te gebruiken.
Hoeveel klokpulsen zijn er nodig om het programma uit te voeren? Antwoord …….klokpulsen.
Wat is de CPU time van dit programma? Antwoord …….. msec.
Neem hierbij aan dat de propagation delay van de Memory units, ALU en Register file (read and write) 0,2 ns is. De propagation delay van andere componenten is te verwaarlozen.
Deze antwoorden worden gebruikt in hoofdstuk 9.

Opgave 9: Selection sort
De opgave is om een lijst van vijf getallen te sorteren naar grootte. Na het sorteren staat het kleinste getal aan het begin van de lijst. Maak gebruik van het zogenaamde selection sort algoritme. Het algoritme werkt als volgt: eerst wordt het kleinste getal van de ongesorteerde lijst bepaald. Dit getal wordt verplaatst naar het begin van de lijst. Hierna wordt het op een na kleinste getal bepaald en verplaatst naar plaats twee van de lijst. Dit wordt steeds herhaald tot het einde van de lijst. Verwisseling van twee getallen vindt plaats in een blok code met het label swap. Hoeveel klokpulsen zijn er nodig om je programma te executeren?
Is selection sort een efficiëntere methode voor het sorteren van vijf getallen dan bubble sort?

8.7 Begrippenlijst

Amdahl's law. De prestatieverbetering ten gevolge van de verbetering van één bepaalde component wordt beperkt tot de mate van gebruik van die component.

CPU time. De tijd dat de processor er over doet om een programma uit te voeren.

Cycle time. De periode van één klokpuls.

Data memory. Een geheugen waarin bij een Harvard architectuur de data wordt opgeslagen.

Data memory access (MEM). De fase waarbij een waarde in/uit het geheugen wordt geschreven/gelezen wordt.

Execution or address calculation (EX). De fase waarbij de ALU of de Adder een berekening uitvoert.

[11)] Op de site www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html zijn animaties van verschillende sorteeralgoritmes te zien.

Harvard architectuur. Een machine met gescheiden geheugens voor instructies en data.

Instruction count. Het aantal instructies dat de processor uitvoert.

Instruction decode and register file read (ID). De fase waarbij de instructie wordt gedecodeerd en de registers worden gelezen.

Instruction fetch (IF). De fase waarbij de instructie wordt opgehaald uit het Instruction Memory.

Load/Store machine. Een machine die met slechts twee instructies, Load en Store, met het geheugen communiceert.

Performance. De inverse van de CPU time.

Write-back (WB). De fase waarbij in de register file wordt geschreven.

8.8 Tentamenvragen

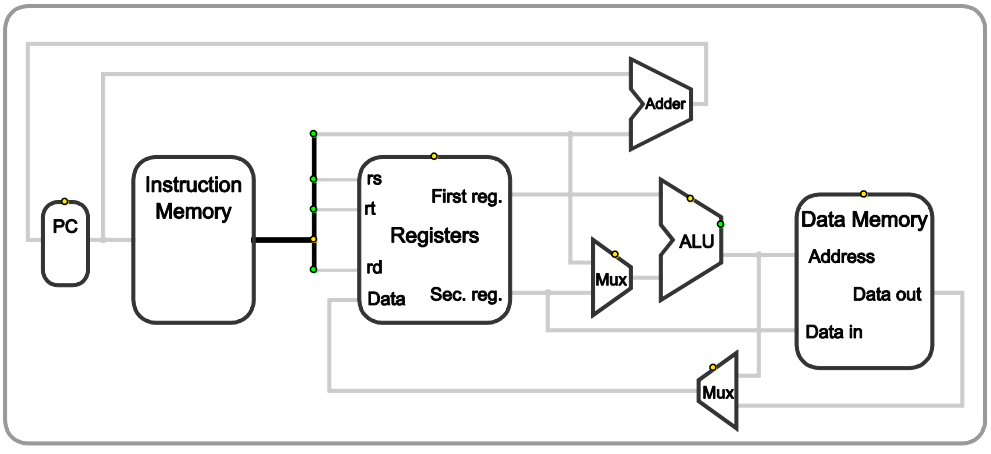
1. Uit welke vijf hoofdcomponenten is een Harvard machine opgebouwd? Geef de functie van elke component weer.

2. Waarom wordt de machine uit dit hoofdstuk een Load/Store machine genoemd?

3. Uit welke fases bestaat het executeren van een “Store Word”-instructie.

4. Wat is de opcode van een instructie?

5. In figuur 8.5 is een machine weergegeven met alleen de data- en adreslijnen tussen de verschillende componenten dus zonder de controlelijnen.
    a. Arceer het datapad en het adrespad van een ADDI-instructie.
    b. Arceer het datapad en het adrespad van een Load Word-instructie.
    c. Arceer het datapad en het adrespad van een Branch Zero-instructie.



Figuur 8.5: Data- en adrespad van een ????-instructie

6. Wat houdt Amdahl’s law in?

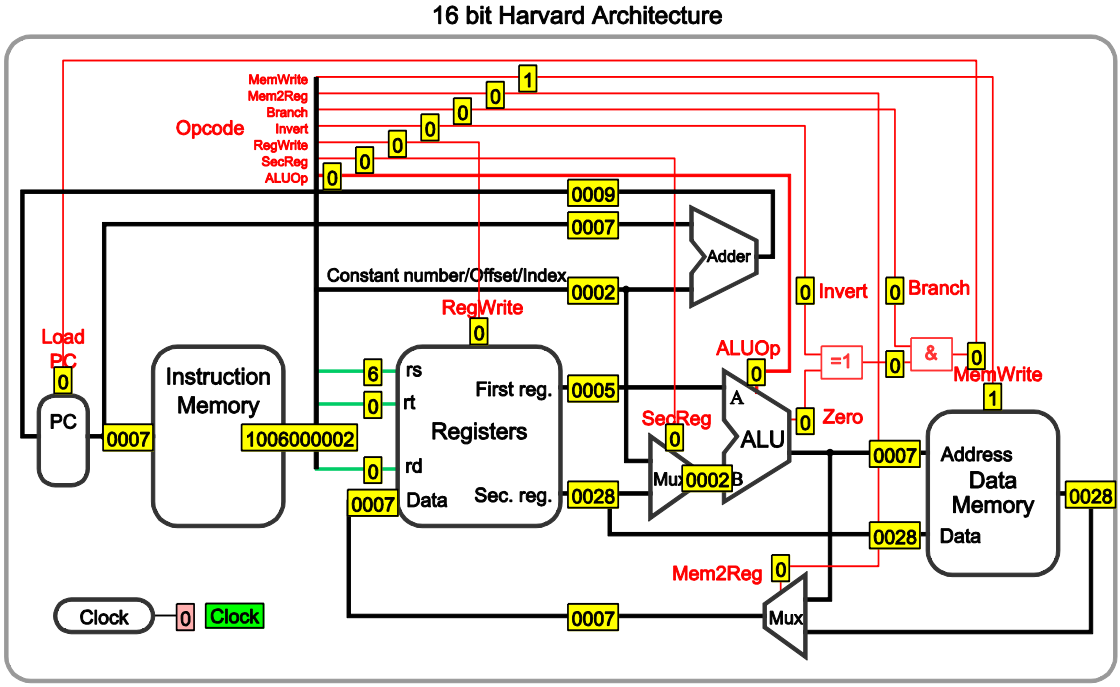
7. Een Single Cycle Harvard machine heeft naast een ALU ook nog een Adder. Beschrijf de functie van deze Adder. Bij welk type instructies wordt deze Adder gebruikt.

8. Van een Single Cycle Harvard machine hebben de componenten de volgende propagation delay time:
    - Memory units 200 ps (picoseconds);
    - ALU 200 ps;
    - Register file (read or write) 200 ps;

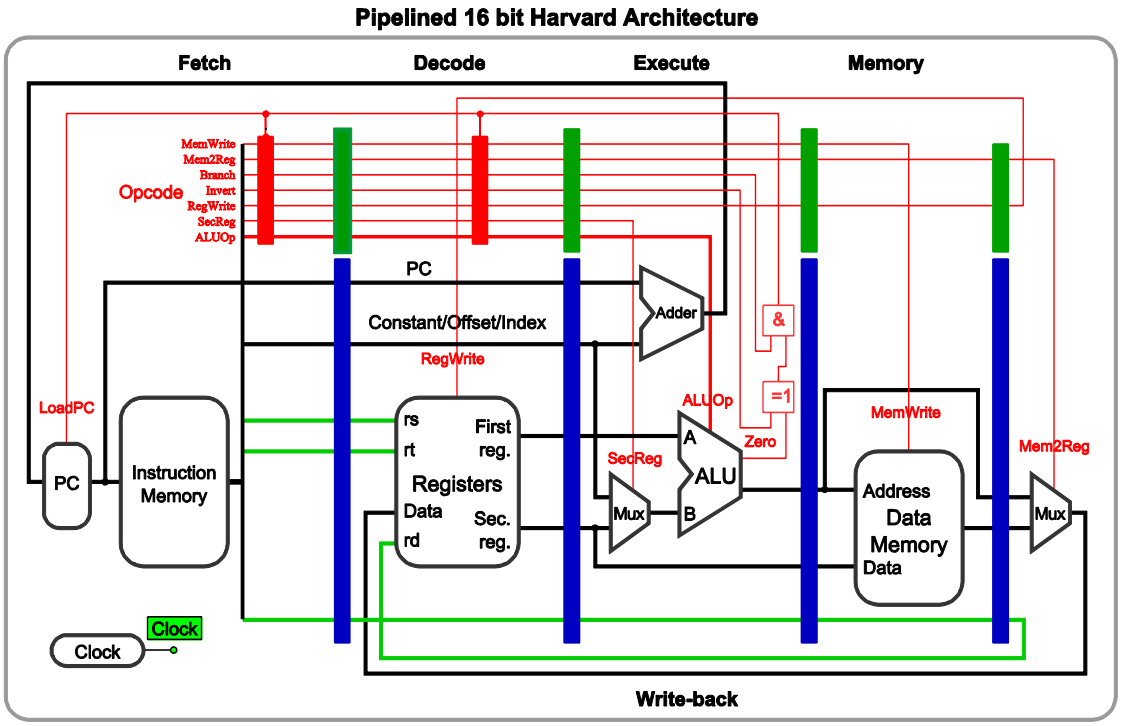
De propagation delay van de andere componenten is te verwaarlozen.
Wat is de (maximale) klokfrequentie van deze machine? Licht het antwoord toe.

9. Van de machine van opgave 8 verschijnt een verbeterde versie. De propagatietijd van de registerfile is teruggebracht naar 100 ps. Wat is nu de maximale klokfrequentie? Wat is de performance verbetering t.o.v. de machine in vraag 8?

10. In figuur 8.6 is een Harvard Single Cycle processor weergegeven. In deze figuur staat de term: Constant number/Offset/Index.
    a. Wat wordt bedoeld met “Constant number”? Bij welke instructies is deze term van toepassing?
    b. Wat wordt bedoeld met “Offset”? Bij welke instructies is deze term van toepassing?
    c. Wat wordt bedoeld met “Index”? Bij welke instructies is deze term van toepassing?
    d. Geef een voorbeeld van een instructie waarbij geen van de drie bovengenoemde termen van belang is.
    e. Welke instructie wordt in figuur 8.6 uitgevoerd? Geef de syntax ervan weer.
    f. Wat is de code in machinetaal van deze instructie?
    g. Wat is de volgende stand van de Program Counter?



Figuur 8.6



Hoofdstuk 9: De Harvard pipelined processor

9.1 Inleiding en leerdoelen

Figuur 8.4 van het vorige hoofdstuk laat zien dat één instructie in vijf fases wordt uitgevoerd. Bij iedere fase wordt een ander deel van de hardware gebruikt. Zo wordt bijvoorbeeld bij de Execute fase de ALU gebruikt terwijl de andere componenten niet actief zijn. Om alle componenten tegelijk aan het werk te krijgen is de pipeline bedacht. Door de zeer eenvoudige structuur van de Single Cycle Harvard machine is deze zeer geschikt om er een zogenaamde pipelined machine mee te construeren. Bij een pipelined machine wordt de hardware in delen opgedeeld. Bij de hierboven weergegeven machine zijn dat er vijf: IF (Instruction Fetch/PC + IM); ID (Instruction Decode and register file read); EX (Execute or address calculation/ALU); MEM (Data Memory Access) en WB (Write Back/Register file write). Tussen deze delen zijn geheugenelementen geplaatst waarin de 'tussenresultaten' d.m.v. een klokpuls worden opgeslagen. In bovenstaande figuur zijn de geheugenelementen als verticale balken weergegeven. Met deze machine kunnen vijf instructies tegelijkertijd worden uitgevoerd.

Aan het einde van dit hoofdstuk weet je:

- hoe een pipelined machine werkt;
- wat CPI betekent;
- waardoor een data hazard ontstaat en hoe je die kunt vermijden;
- wat een branch hazard is en hoe je die kunt verhelpen;
- wat een Von Neumann architectuur inhoudt;
- wat de Von Neumann bottleneck is;
- wat een Multicycle machine inhoudt;
- het verschil tussen een RISC- en een CISC-architectuur.

Aan het einde van het hoofdstuk kun je:

- een programma schrijven voor de Pipelined Harvard machine met zo min mogelijk NOP's (No Operation instructies);
- de performanceverbetering meten en berekenen voor de volgende gevallen:
    - Pipelined Harvard machine t.o.v. Single Cycle Harvard machine;
    - Pipelined Harvard machine met forwarding t.o.v een Pipelined machine zonder forwarding;
    - Pipelined Harvard machine met branch prediction t.o.v een Pipelined machine zonder branch prediction;

9.2 De werking van de pipelined machine

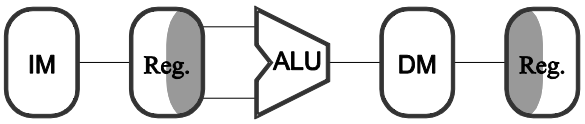
Bij de machine in dit hoofdstuk bestaat het uitvoeren van een instructie uit vijf fases: IF, ID, EX, MEM en WB. Iedere fase duurt één (clock)cycle. Na de eerste cycle is van de eerste instructie de IF-fase uitgevoerd. Na de tweede cycle is van de eerste instructie de ID-fase en van de tweede instructie de IF-fase uitgevoerd. Na de derde cycle is van de eerste instructie de EX-fase, van de tweede instructie de ID-fase en van de derde instructie de IF-fase uitgevoerd. En zo verder. In tabel 9.1 is de uitvoering van een programma met **zes instructies** weergegeven. De eerste instructie is gereed na 5 cycles, de tweede instructie na 6 cycles en de derde na 7 cycles. Totaal zijn voor 6 instructies 6 + 4 is 10 cycles nodig.
Bij een ideale pipelined machine is de propagation delay van alle hardware componenten voor alle fases gelijk. Iedere fase heeft bij deze machine één cycle nodig. De cycle time is gelijk aan de component met de grootste delay. Stel dat deze 0,2 ns bedraagt. De CPU time voor het uitvoeren van deze zes instructies bedraagt 10 * 0,2 ns is 2 ns. De bijbehorende klokfrequentie is dus 5 GHz. Waren deze zes instructies uitgevoerd met de Single Cycle Harvard machine dan was de CPU time 1 ns per instructie dus totaal 6 ns. Met de pipelined machine wordt bij dit voorbeeld een prestatieverbetering van een factor 3 bereikt.

cycle / Instr. nr.	1	2	3	4	5	6	7	8	9	10
1	IF	ID	EX	MEM	WB					
2		IF	ID	EX	MEM	WB				
3			IF	ID	EX	MEM	WB			
4				IF	ID	EX	MEM	WB		
5					IF	ID	EX	MEM	WB	
6						IF	ID	EX	MEM	WB

Tabel 9.1: Executie van instructies in een pipelined machine

Een pipelined machine met 5 ‘stages’ (fases) heeft een 5 keer hogere klokfrequentie dan een Single Cycle machine. Voor iedere fase is één cycle nodig.

Grafische representatie van de pipeline componenten
Figuur 9.1 is een grafische representatie van de componenten die achtereenvolgens gebruikt worden in een pipelined machine. De Fetch-fase (IF) gebruikt het Instruction Memory (IM); de Decode-fase (ID) leest twee registers (Reg); de Execute-fase voert de ALU-berekening uit; de MEM-fase leest of schrijft het Data Memory (DM) en in de Write Back-fase (WB) wordt naar één van de Registers geschreven. De component Registers komt hierin dus twee keer voor: bij de ID-fase en bij de WB-fase. In de ID-fase wordt het betreffende register gelezen (rechterdeel van Reg.). In de WB-fase wordt het betreffende register geschreven (linkerdeel van Reg.).



Figuur 9.1: Representatie van de pipeline

CPI
Bij de Single Cycle machine wordt iedere instructie uitgevoerd in 1 cycle. Bij de pipelined machine wordt iedere instructie opgedeeld in fases, hier vijf, die *ieder* een klokpuls nodig hebben. Om de CPU time van een programma te berekenen, voeren we het begrip CPI (Cycles Per Instruction) in. Hiermee wordt het *gemiddelde aantal* klokpulsen per instructie bedoeld. In het geval zoals in tabel 9.1 is weergegeven worden 6 instructies uitgevoerd en zijn hiervoor 10 cycles nodig. De CPI is in dit geval 10/6 = 1,667.

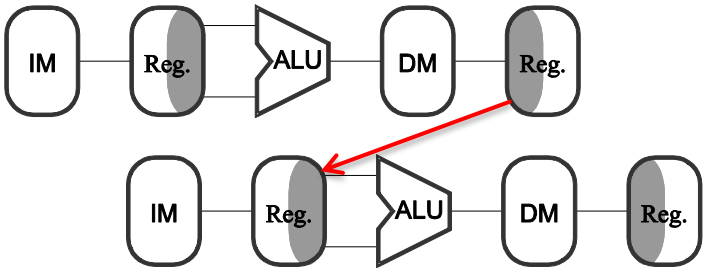
$$\text{CPU time} = \text{Instruction count} * \text{CPI} * \text{Clock cycle time} = \frac{\text{Instruction count} * \text{CPI}}{\text{Clock rate}}$$

Maximale performance ratio Pipelined machine / Single Cycle machine
Stel dat een programma voor de pipelined machine bestaat uit 1000 instructies. Met de kennis die we nu hebben zijn er 1004 cycles nodig om dit programma te executeren. In dit geval is de CPI 1004/1000 = 1,004 en de cycletime 0,2 ns.
De CPU time is dus 1000 * 1,004 * 0,2 ns ≈ 0,2 µs.
Als op de Single Cycle machine hetzelfde programma wordt uitgevoerd dan is de CPI 1 (per definitie) en de cycletime is 1 ns. De CPU time is: 1000 * 1 * 1 ns = 1 µs. In dit, geïdealiseerde voorbeeld wordt een prestatieverbetering van een factor 5 bereikt!

Latency en throughput
De time delay voor het uitvoeren van één instructie wordt de *latency* van de pipeline genoemd. Deze is zowel voor een Single Cycle als voor een pipelined machine dezelfde als de delay's van de pipelineregisters worden verwaarloosd. Verkorting van de executietijd van een programma heeft bijna altijd een verhoging van de *throughput*, het aantal uitgevoerde instructies in een bepaalde tijd, tot gevolg. De throughput is dus aanzienlijk hoger bij een pipelined machine dan bij zijn Single Cycle equivalent.

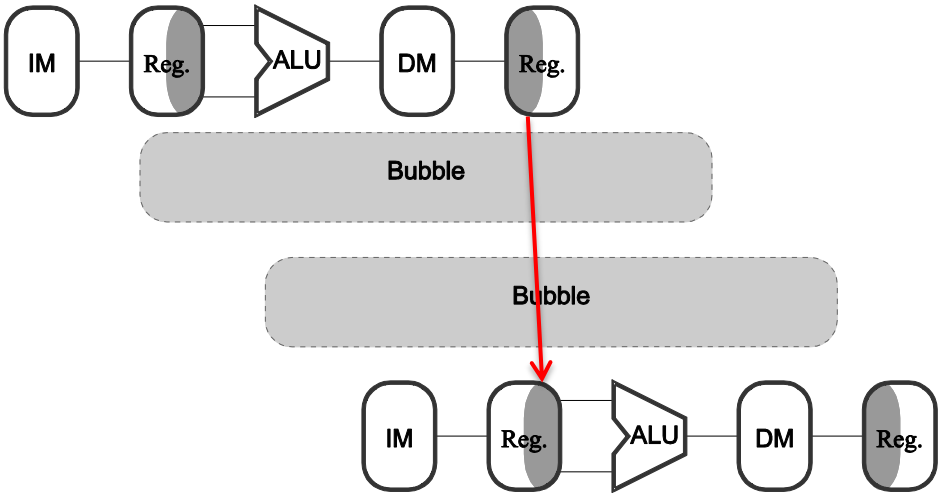
9.3 Pipeline hazards
Er zijn situaties bij gepijplijnde machines waarbij de volgende instructie niet in de volgende klokpuls kan worden geëxecuteerd. Deze gevallen worden pipeline hazards genoemd. We behandelen twee typen hazards, data hazards en branch hazards.

Data Hazards
In figuur 9.2 is, in grafische vorm, een zogenaamde data hazard weergegeven. Deze treedt op als het resultaat van een instructie gebruikt wordt als argument bij een direct daaropvolgende instructie. De benodigde waarde is op dat moment nog niet geschreven in het betreffende register. Dit wordt data-afhankelijkheid genoemd.



Figuur 9.2: Data hazard

Voorbeeld
Stel dat er in register $1 het getal 4 en in register $2 het getal 2 staat. We voeren de instructie ADDI $1, $2, 10, gevolgd door de instructie ADD $3, $1, $2 uit. Na drie cycles staat de tweede instructie klaar voor de EX-fase en heeft de nieuwe waarde van $1, het getal 12 (2 + 10), nodig. Pas twee cycles later is deze waarde beschikbaar als van de eerste instructie de WB-fase is uitgevoerd. Dan pas wordt in register $1 de waarde 12 geschreven, dat is twee cycles te laat. Bij de tweede instructie wordt nu een argument met een foute waarde, 4 gebruikt. Dit levert als resultaat de waarde 6 op i.p.v. 14 in $3. Dit wordt een data hazard genoemd.
Het is dus noodzakelijk om voor het uitvoeren van de tweede instructie, twee cycles te wachten. Hierdoor ontstaan twee zogenaamde pipeline bubbles, ook pipeline stalls genoemd zoals in figuur 9.3 is weergegeven.



Figuur 9.3: Door een data hazard ontstaan er twee bubbles

Er zijn drie methoden om executiefouten te elimineren:

- **NOP's**: Instructie(s) genaamd NOP (NO Operation) implementeren op de plek van de bubble(s). Deze oplossing beïnvloedt de executietijd nadelig.
- **Scheduling:** Instructies verwisselen (schedulen) zodat tussen data-afhankelijke instructies, twee data-onafhankelijke instructies worden geplaatst.
- **Forwarding unit:** De hardware uitbreiden met een forwarding unit (zie paragraaf 9.4).

NOP's implementeren

Het programma ziet er dan als volgt uit (de data-afhankelijkheid is vet weergegeven):

ADDI **$1**, $2, 10
NOP
NOP
ADD $3, **$1**, $2

Instructies schedulen

Een tweede oplossing is instructies te verwisselen. Een moderne compiler doet dat automatisch. Hieronder is een voorbeeld gegeven met drie (vetgedrukte) data-afhankelijke constructies. Bij het linker programma zijn de hazards opgelost met NOP-instructies en bij het rechter programma d.m.v. het schedulen van instructies.

```
ADDI $1, $2, 10
NOP
NOP
ADD $3, $1, $2
SUB $4, $5, $6
NOP
NOP
SUB $7, $4, $6
LW $9, ref1, $5
NOP
NOP
ADDI $5, $9, 12
```

Door instructies die geen argumenten van de voorgaande twee instructies gebruiken achter elkaar te plaatsen worden de NOP's geëlimineerd en daardoor wordt de CPU time met zes cycles verkort. Hieronder is dit weergegeven.

```
ADDI $1, $2, 10
SUB $4, $5, $6
LW $9, ref1, $5
ADD $3, $1, $2
SUB $7, $4, $6
ADDI $5, $9, 12
```

Branch hazards

Bij een branch-instructie zoals de instructie BEQ wordt in de derde cycle beslist wat de volgende instructie wordt. Als de branch niet wordt genomen, ofwel de Program Counter maakt geen sprong, worden de twee instructies die al in de pipeline zitten gewoon uitgevoerd. Niets aan de hand dus. Wat gebeurt er als de branch *wel* wordt genomen? Dan springt de PC in de derde cycle naar het, door PC en offset bepaalde, adres van de nieuwe instructie. Deze komt in de IF-fase. Hoe zit het nu met de twee instructies na de BEQ-instructie die al in de pipeline zitten? In de meeste gevallen mogen die niet worden uitgevoerd.
Er zijn drie methoden om de gevolgen van deze bubbles te reduceren:

- **Scheduling:** De instructies die in de pipeline zitten, moeten worden uitgevoerd onafhankelijk of de branch genomen wordt of niet. De twee instructies na een branch-instructie worden dus altijd uitgevoerd. Dit wordt het zogenaamde *branch delay slot* genoemd. In plaats van twee NOP's kunnen er twee instructies worden geplaatst die, binnen de cyclus van de loop altijd moeten worden uitgevoerd, onafhankelijk of de branch genomen wordt of niet, bijvoorbeeld het incrementeren van tellers. Een voor de hardware geoptimaliseerde compiler kan hiervoor zorgen.
- **Flushing:** De twee instructies die in de pipeline zitten worden "geflusht", d.w.z. dat de code gaat bestaan uit nullen, dezelfde code als voor een NOP-instructie. Bij een branch die genomen wordt treden dus twee pipeline bubbles op.
- **Prediction unit:** Door de hardware uit te breiden met een zgn. dynamic branch prediction unit (zie paragraaf 9.5). Hierbij wordt naar het voorgaande gedrag gekeken van een bepaalde branch. De prediction unit bepaalt wat de volgende instructie wordt op basis van het voorgaande gedrag.

"ADD 5 values" met flushing

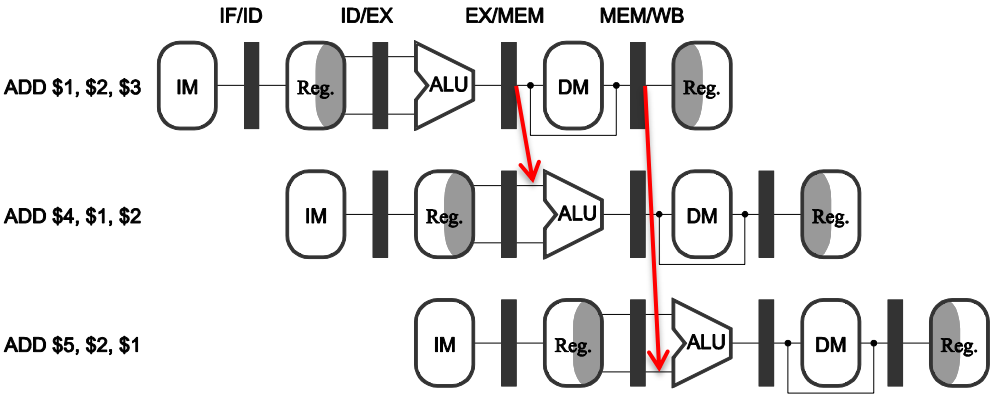
De machine, waarop het programma draait dat in het tekstvenster hiernaast is weergegeven, flusht de twee instructies na de BNE-instructie, SW en Halt, als de branch wordt genomen.

```
@include "PipelinedHarvard.wasm"
# Add 5 values stored in memory locations 2 .. 6 and store
# the result in the first free location (7)

.data MyMemory : DATAMEM
0x02 :   WORD   start      0x6
         WORD              0x7
         WORD              0x8
         WORD              0x9
         WORD              0d10

.code MyCode : HARVARD, MyMemory
         LOADI $6, 0       # Start at element 0
         LOADI $0, 0       # Clear $0
         LOADI $5, 5       # 5 values

loop:    LW $1, start, $6
         ADDI $6, $6, 1
         NOP
         ADD $0, $0, $1
         BNE $6, $5, loop

         SW $0, start, $6  # Flushed if taken, executed if not
HALT                       # Flushed if taken, executed if not
```

De loop wordt 5 keer doorlopen, 4 keer met "branch taken" en de laatste keer met "branch not taken". Het aantal benodigde cycles voor het gedeelte dat de loop doorlopen wordt is dus: 4 * 7 + 1 * 5 = 33 cycles. 3 cycles zijn nodig voor het initialisatie gedeelte, 1 voor de SW-instructie en 4 voor de pipeline.
Het totaal aan cycles voor het programma is 33 + 3 +1 + 4 = 41 cycles. Het aantal uitgevoerde instructies is 24. NOP's zijn hierbij niet meegeteld.
De CPI = 41/24 = 1,7.
Tot welke waarde nadert de CPI als de loop 1000 keer wordt doorlopen? Antwoord ……..

9.4 De pipelined machine met forwarding

Een hardware oplossing om data hazards te vermijden is het toepassen van forwarding. Bij bijna alle pipeline bubbles zijn de benodigde waarden nog niet geschreven in de register-file maar wel beschikbaar in het pipelineregister tussen de ALU en het DM (EX/MEM) of in het pipelineregister tussen het DM en de WB (MEM/WB).



Figuur 9.4: Benodigde waarden zijn beschikbaar in de pipeline registers

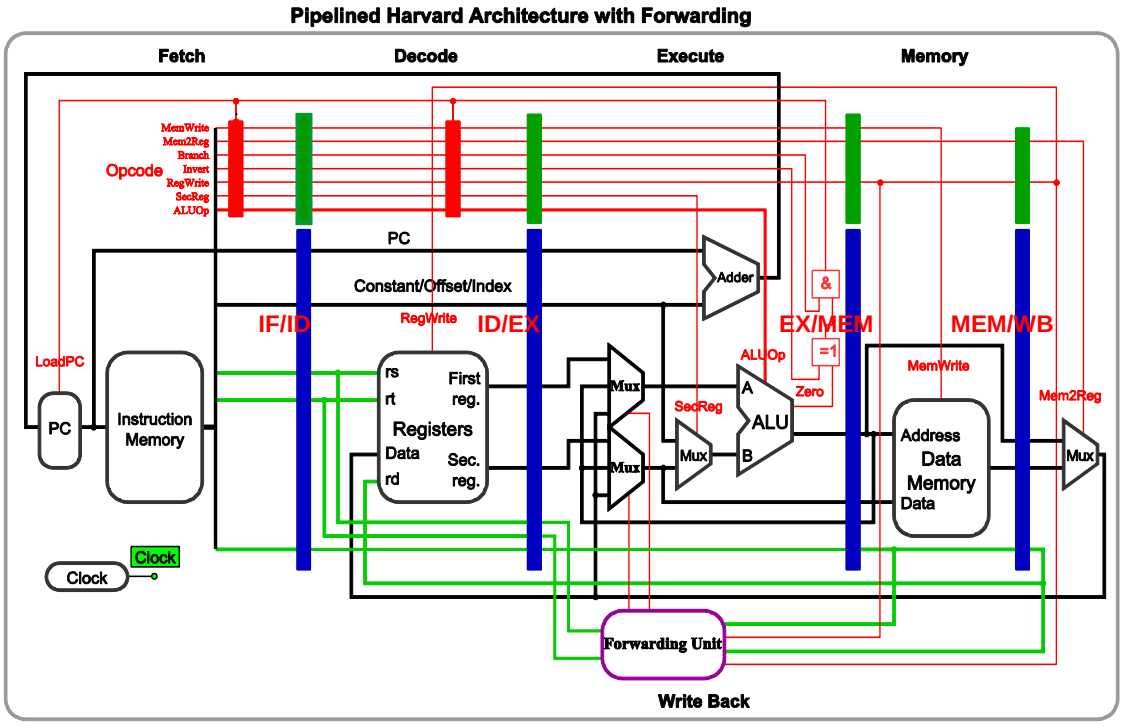
Na iedere cycle worden in de pipelineregisters IF/ID, ID/EX, EX/MEM en MEM/WB de gegevens en de bijbehorende adressen en controlesignalen van die gegevens opgeslagen. Hiervan maken we gebruik. In figuur 9.4 is weergegeven hoe we de twee bubbles tussen de eerste twee instructies kunnen vermijden door een verbinding te leggen tussen het EX/MEM pipeline register en de bovenste ingang van de ALU. Dit EX/MEM register bevat wel de nieuwe (juiste) waarde van $1. We kunnen de bubble tussen de tweede en de derde instructie vermijden door een verbinding te leggen tussen het MEM/WB pipeline register en de onderste ALU-ingang. Dit register bevat, een cycle later, de waarde van $1. Zo zijn er ook gevallen dat verbindingen noodzakelijk zijn tussen EX/MEM en de onderste ALU-ingang en tussen MEM/WB en de bovenste ALU-ingang. Denk hierbij aan een instructievolgorde als:
ADD **$1**, $2, $3
ADD $4, $8, **$1**
ADD $5, **$1**, $2

Forwarding unit

In figuur 9.5 is een pipelined machine met een forwarding unit weergegeven. De forwarding unit stuurt twee multiplexers aan die afhankelijk van de hieronder genoemde condities de A-ingang en de B-ingang van de ALU voorzien van de juiste waarden. Er zijn vier gevallen waarbij een waarde naar voren moet worden gehaald:

1. EX/MEM rd = ID/EX rs # Voorbeeld: ADD **$1**, $2, $3; ADD $4, **$1**, $2
2. EX/MEM rd = ID/EX rt # Voorbeeld: ADD **$1**, $2, $3; ADD $4, $8, **$1**
3. MEM/WB rd = ID/EX rs # Voorbeeld: ADD **$1**, $2, $3; ADDI $9, $9, 2; ADD $5, **$1**, $2
4. MEM/WB rd = ID/EX rt # Voorbeeld: ADD **$1**, $2, $3; ADDI $9, $9, 2; ADD $5, $2, **$1**

Toelichting van het eerste geval: Als het adres van rd opgeslagen in het pipeline register EX/MEM gelijk is aan het adres van rs in ID/EX is de waarde van de ALU-uitgang (opgeslagen in EX/MEM) nodig voor de ALU-ingang rs. Dit is bijvoorbeeld het geval bij de volgende twee opeenvolgende afhankelijke instructies: ADD **$1**, $2, $3; ADD $4, **$1**, $2. De bovenste multiplexer laat in dit geval de gegevens uit het EX/MEM pipelineregister door.
Toelichting van het vierde geval: Als het adres van rd in MEM/WB gelijk is aan het adres van rt in ID/EX is de waarde van de ALU opgeslagen in MEM/WB nodig voor de ALU-ingang rt. Bij de volgende drie opeenvolgende instructies: ADD **$1**, $2, $3; ADDI $9, $9, 2; ADD $5, $2, **$1** is dat het geval. De onderste multiplexer laat in dit geval de gegevens uit het MEM/WB pipelineregister door.
Voor het tweede en het derde geval gelden overeenkomstige voorwaarden.
Data hazards treden alleen op bij instructies waarbij naar de Register-file wordt geschreven.



Figuur 9.5: Pipelined machine met forwarding

Voorbeeld: "ADD 5 values" met forwarding

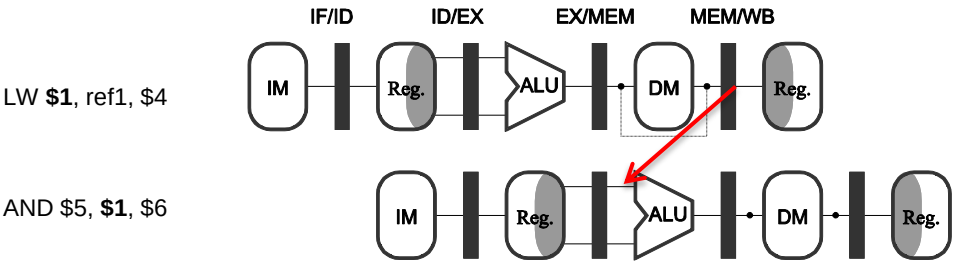
De machine, waarop het programma draait dat in het tekstvenster hiernaast is weergegeven, flusht de twee instructies na de BNE-instructie (SW en Halt) als de branch wordt genomen.

De loop wordt 5 keer doorlopen, 4 keer met "branch taken" en de laatste keer met "branch not taken". Het aantal benodigde cycles voor het gedeelte dat de loop doorlopen wordt is dus: 4 * 6 + 1 * 4 = 28 cycles.
Het totaal aan cycles voor het programma is 3 + 28 +1 + 4 = 36. Het aantal uitgevoerde instructies is 24.
De CPI = 36/24 = 1,5.
Tot welke waarde nadert de CPI als de loop 1000 keer wordt doorlopen?
Antwoord ……..

```
@include "PipelinedForwarding.wasm"
# Add 5 values stored in memory locations 2 .. 6 and
# store the result in the first free location (7)

.data MyMemory : DATAMEM
0x02 :    WORD  start     0x6
          WORD            0x7
          WORD            0x8
          WORD            0x9
          WORD            0d10

.code MyCode : HARVARD, MyMemory
          LOADI $6, 0     # Start at element 0
          LOADI $0, 0     # Clear $0
          LOADI $5, 5     # 5 values

loop:     LW $1, start, $6
          ADDI $6, $6, 1
          ADD $0, $0, $1
          BNE $6, $5, loop

          SW $0, start, $6  # Flushed if taken, exec. if not
HALT                        # Flushed if taken, exec. if not
```

De Load Word hazard

Kunnen alle data hazards met forwarding opgelost? Helaas is dit niet zo.



Figuur 9.6: Load Word hazard

Bij een Load Word-instructie is de waarde pas in de vierde fase in het MEM/WB pipeline register aanwezig. Als deze waarde wordt gebruikt bij de erop volgende instructie, treedt een bubble op die niet met forwarding kan worden opgelost. Je kunt niet terug in de tijd! In figuur 9.6 is dit weergegeven.

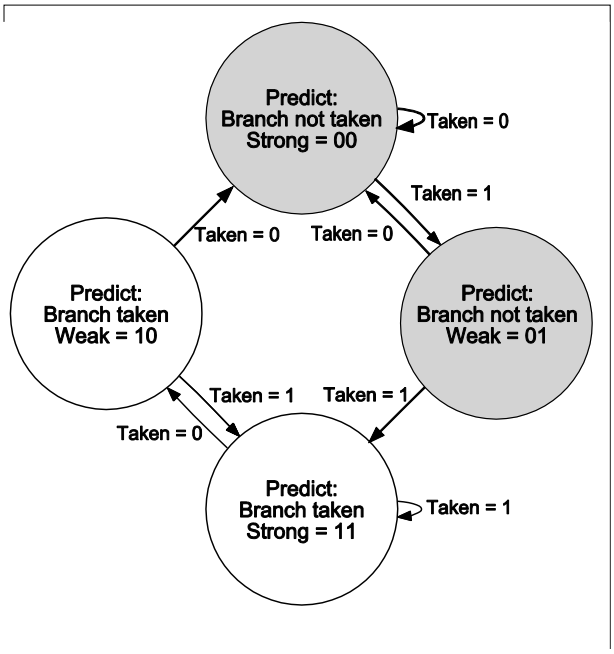
9.5 De pipelined machine met branch prediction

Wat is een dynamische branch predictor?
Een Branch Predictor is een schakeling die voorspelt of tijdens een spronginstructie de Program Counter een sprong maakt of niet, waardoor het leegmaken van de pipeline zo veel mogelijk beperkt kan worden. Vooral bij *for-loops* is de kans veel groter dat de loop nogmaals moet worden doorlopen dan dat de loop wordt beëindigd. De predictor past zich aan tijdens de uitvoering van een programma en daarom is er sprake van een dynamische branch predictor. Iedere branch heeft zijn een eigen predictor. De status van iedere branch wordt bijgehouden in een zgn. *branch history table* ook wel *branch prediction buffer* genoemd.

State Diagram
Figuur 9.7 toont het state diagram van een branch predictor. Elke keer als er bij een branch-instructie niet wordt gesprongen is de ingang Taken 0 en elke keer als er wordt gesprongen is deze ingang 1. De predictor kent vier toestanden:

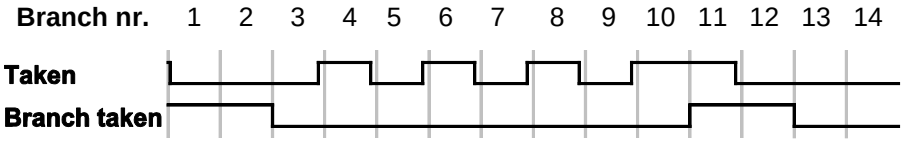
1. Branch not taken Strong = 00 (Grote kans dat er geen sprong gaat plaatsvinden).
2. Branch not taken Weak = 01 (Redelijk grote kans dat er geen sprong gaat plaatsvinden).
3. Branch taken Weak = 10 (Redelijk grote kans, dat er gesprongen wordt).
4. Branch taken Strong = 11 (Grote kans, dat er gesprongen wordt).

Toelichting
Stel dat de begintoestand is: Branch not taken Strong = 00. Na een Branch-instructie waarbij wordt gesprongen (Taken = 1) wordt de toestand: Branch not taken Weak = 01. Na een tweede Branch-instructie waarbij weer wordt gesprongen wordt de toestand: Branch taken Strong = 11 bereikt. De machine gaat er nu vanuit dat de kans groot is dat er bij een nieuwe branch-instructie weer wordt gesprongen, zoals bij een “while do-loop” meestal het geval is. Om de predictor weer in de branch not taken toestand te krijgen, moet er *twee keer niet* worden gesprongen (Taken = 0). De machine komt dan via de toestand 10 weer terecht bij de beginsituatie 00.



Figuur 9.7: State diagram branch predictor

Timing diagram
Figuur 9.8 laat het gedrag zien van de Branch Predictor. De initiële waarde is: Branch taken strong. Nadat er twee keer niet wordt gesprongen (Taken = 0),



Figuur 9.8: Tijdvolgordediagram branch predictor

verandert de status in Branch taken Weak en daarna in Branch not taken Strong. Gedurende branches 4 t/m 9 wisselt de predictor tussen Branch not taken Weak en Branch not taken Strong. De uitgang Branch taken verandert dan niet. Bij branch nummer 11 verandert de status van Branch not taken Weak in Branch taken Strong en de uitgang Branch taken wordt 1. Het

omgekeerde gebeurt bij branch 13. Als de ingang Taken minstens twee keer achtereen dezelfde waarde heeft is de voorspelling dat ook een volgende keer Taken dezelfde waarde houdt.

"ADD 5 values" met branch predictor

De initiële waarde van de predictor voor de loop in het tekstvenster hiernaast is: Branch not taken Strong. De loop wordt 5 keer doorlopen. De eerste twee keer staat de predictor in de toestand Branch not taken en kost het doorlopen van de loop 7 cycles, 5 voor de instructies en 2 t.g.v. het flushen van de pipeline. Bij de derde en de vierde keer dat de loop doorlopen wordt staat de predictor op Branch taken, dus kost het doorlopen van de loop maar 5 cycles. De laatste keer is de voorspelling weer onjuist. Het aantal benodigde cycles voor het gedeelte dat de loop doorlopen wordt is dus: 2 * 7 + 2 * 5 + 1 * 7 = 31 cycles.
Het totaal aan cycles voor het programma is 3 + 31 +1 + 4 = 39 cycles. Het aantal uitgevoerde instructies is 24. De CPI = 39/24 ≈ 1,6.

```
@include "PipelinedHarvard.wasm"
# Add 5 values stored in memory locations 2 .. 6 and store
# the result in the first free location (7)

.data MyMemory : DATAMEM
0x02 :   WORD  start    0x6
         WORD           0x7
         WORD           0x8
         WORD           0x9
         WORD           0d10

.code MyCode : HARVARD, MyMemory
         LOADI $6, 0    # Start at element 0
         LOADI $0, 0    # Clear $0
         LOADI $5, 5    # 5 values

loop:    LW $1, start, $6
         ADDI $6, $6, 1
         NOP
         ADD $0, $0, $1
         BNE $6, $5, loop

         SW $0, start, $6
HALT
```

Tot welke waarde nadert de CPI als de loop 1000 keer wordt doorlopen? Antwoord ……..

9.6 Opgaven en practicum met de pipelined machines

De drie pipelined machines van dit practicum hebben vrijwel dezelfde instructieset als de Harvard machine. Het enige verschil is dat aan de machines een instructie NOP (NO Operation) is toegevoegd. De eerste machine, de Pipelined Harvard, heeft geen forwarding en geen branch predictor en flusht de twee instructies die na een genomen branch-instructie nog in de pipeline zitten. Aan de tweede machine, gebruikt vanaf opgave 7, is een forwarding unit toegevoegd. De derde machine heeft een branch predictor maar geen forwarding. Met deze machine wordt vanaf opgave 9 gewerkt.

Opgave 1: Register read and register write

```
ADD $2, $3, $1
SUB $4, $3, $5
ADD $5, $3, $7
ADD $7, $6, $1
ADD $8, $2, $6
```

Welke registers worden er aan het einde van de zesde klokpuls gelezen? Antwoord ……
Welk register wordt dan geschreven? Antwoord ……

Opgave 2: Data Hazards en aantal NOP's

Beredeneer hoeveel NOP's er minimaal nodig zijn tussen de volgende vier instructies bij een machine zonder forwarding.
ADD $1, $2, $3
ADDI $4, $1, 2 (aantal NOP's …….)
SUB $5, $1, $2 (aantal NOP's …….)
SW $5, ref1, $1 (aantal NOP's …….)

Opgave 3: CPI

Een programma bevat 10^3 instructies in het format: LW, ADD, LW, ADD, …. etc. Een van de argumenten van de instructie ADD hangt af van de voorgaande instructie LW en vice versa.
a. Wat is de CPI bij een pipelined machine zonder forwarding? Antwoord ……
b. Wat is de CPI als er forwarding wordt toegepast? Antwoord ……

Opgave 4: Data Hazards en de pipelined machine

Start uit de folder "H9PipelinedHarvardArchitecture" de worksheet "Pipelined Harvard Processor.sim-pl-ws".
Open het programma "Opgave4.wasm" (zie tekstvenster).
Executeer dit programma.
De kleur van de instructie laat zien welke fase wordt uitgevoerd. IF groen, ID blauw, EX paars, MEM bruin en WB geel.
Beantwoord de volgende vragen:
- Waarom komt het (verkeerde) getal 9 in register 3 terecht? Verklaar de 'waarde 9'.
- Verbeter het programma door NOP's te verplaatsen. De CPU time mag hierbij **niet toenemen**. Geef de veranderingen van je code weer in het tekstvenster hiernaast.

Run het verbeterde programma. Na hoeveel klokpulsen is het programma geëxecuteerd? Dit neem je waar door na te gaan wanneer de Write Back-fase van de laatste instructie: LW $4, ref3, $0 wordt uitgevoerd.
Antwoord: Na . …..klokpulsen. Wat is de CPI van dit programma?
Antwoord: CPI = …..
NOP's tellen (uiteraard) niet mee.
Geef van het verbeterde programma de waarde van het argument rd van iedere instructie weer gedurende het pipeline proces in tabel 9.2. Voor de eerste twee cycles is deze waarde al ingevuld.

```
@include "PipelinedHarvard.wasm"

.data MyRegisters : REGISTERS
0x0 : WORD 0x0

.data MyMemory : DATAMEM
0x01 : WORD ref1 0xA
0x02 : WORD ref2 0xB
0x03 : WORD ref3 0x5555

.code MyCode : HARVARD, MyRegisters, MyMemory
LW $1, ref1, $0
NOP
LW $2, ref2, $0
NOP
ADD $3, $1, $2
NOP
SW $3, ref3, $0
NOP
LW $4, ref3, $0
HALT
```

```
@include "PipelinedHarvard.wasm"

.data MyRegisters : REGISTERS
0x0 : WORD 0x0

.data MyMemory : DATAMEM
0x01 : WORD ref1 0xA
0x02 : WORD ref2 0xB
0x03 : WORD ref3 0x5555

.code MyCode : HARVARD, MyRegisters, MyMemory
LW $1, ref1, $0




HALT
```

Clock cycle	IF	ID	EX	MEM	WB
1	1	0	0	0	0
2	2	1	0	0	0
3					
4					
5					
6					
7					
8					

Tabel 9.2

Opgave 5: Copy block op de pipelined machine

Pas het programma dat je geschreven hebt bij opdracht 7 van hoofdstuk 8 paragraaf 6 aan voor de pipelined machine. *Minimaliseer het aantal NOP's.*
Hoeveel branch hazards treden er op? Antwoord …… hazards.
Hoeveel cycles kosten deze hazards? Antwoord …… cycles.
Hoeveel klokpulsen zijn er nodig om het programma uit te voeren? Antwoord …….klokpulsen.
Wat is de CPI voor dit programma (NOP's tellen hierbij niet mee)? Antwoord ……
Neem aan dat de propagation delay van de Memory units, ALU en Register file (read and write) 0,2 ns is. De propagation delay van andere componenten is te verwaarlozen.
Wat is de CPU time van dit programma? Antwoord …….. msec.
Wat is de performance ratio tussen de pipelined Harvard machine en zijn Single Cycle equivalent voor het programma copy block? Antwoord ……..

Opgave 6: Bubble sort

Pas het programma dat je geschreven hebt bij opdracht 8 van hoofdstuk 8 paragraaf 6 aan voor de pipelined machine. *Minimaliseer het aantal NOP's!*

Hoeveel klokpulsen zijn er nodig om het programma uit te voeren? Antwoord …….klokpulsen.
Wat is de CPI voor dit programma (NOP's tellen hierbij niet mee)? Antwoord ……
Neem aan dat de propagation delay van de Memory units, ALU en Register file (read and write) 0,2 ns is. De propagation delay van andere componenten is te verwaarlozen.
Wat is de CPU time van dit programma? Antwoord …….. msec.
Wat is de performance ratio tussen de pipelined Harvard machine en zijn Single Cycle equivalent voor het programma bubble sort? Antwoord ……..

Opgave 7: Copy block op de pipelined machine met forwarding
Start uit de folder "H9PipelinedHarvardArchitecture" de worksheet ""Pipelined Forwarding Processor".
Pas het programma dat je geschreven hebt bij opdracht 5 van deze paragraaf aan voor de pipelined machine met forwarding. *Minimaliseer het aantal NOP's!*
Hoeveel klokpulsen zijn er nodig om het programma uit te voeren? Antwoord …….klokpulsen.
Wat is de CPI voor dit programma (NOP's tellen hierbij niet mee)? Antwoord ……
Wat is de CPU time van dit programma? Antwoord …….. msec.
Wat is de performance ratio tussen de pipelined machine met en de pipelined machine zonder forwarding. Antwoord ……..

Opgave 8: Bubble sort op de pipelined machine met forwarding
Pas het programma dat je geschreven hebt bij opdracht 6 van deze paragraaf aan voor de pipelined machine. *Minimaliseer het aantal NOP's!*
Hoeveel klokpulsen zijn er nodig om het programma uit te voeren? Antwoord …….klokpulsen.
Wat is de CPI voor dit programma (NOP's tellen hierbij niet mee)? Antwoord ……
Wat is de CPU time van dit programma? Antwoord …….. msec.
Wat is de performance ratio tussen de pipelined machine met forwarding en die zonder forwarding? Antwoord ……..

Opgave 9: Copy block op de pipelined machine met branch predictor
Start uit de folder "H9PipelinedHarvardArchitecture" de worksheet "Pipelined BranchPredict Processor".
Pas het programma dat je geschreven hebt bij opdracht 5 van deze paragraaf aan voor de pipelined machine met branch predictor. *Minimaliseer het aantal NOP's!*
Hoeveel klokpulsen zijn er nodig om het programma uit te voeren? Antwoord …….klokpulsen.
Wat is de CPI voor dit programma (NOP's tellen hierbij niet mee)? Antwoord ……
Wat is de CPU time van dit programma? Antwoord …….. msec.
Wat is de performance ratio tussen de pipelined machine met branch predictor en die zonder branch predictor? Antwoord ……..

Opgave 10: Bubble sort op de pipelined machine met branch predictor
Pas het programma, dat je eerder geschreven hebt bij opdracht 6 van deze paragraaf, aan voor voor de pipelined machine met branch predictor. *Minimaliseer het aantal NOP's!*
Hoeveel klokpulsen zijn er nodig om het programma uit te voeren? Antwoord …….klokpulsen.
Wat is de CPI voor dit programma (NOP's tellen hierbij niet mee)? Antwoord ……
Wat is de CPU time van dit programma? Antwoord …….. msec.
Wat is de performance ratio tussen de pipelined machine met branch predictor en die zonder branch predictor? Antwoord ……..

Opgave 11: Vergelijking bubble sort op vier machines
Vul in tabel 9.3 de resultaten van je metingen op de 4 machines in.

Bubble sort	Single Cycle Harvard	Pipelined Harvard	Pipelined with forwarding	Pipelined with branch pred.
Nr of cycles				
CPI				
CPU time				

Tabel 9.3: CPU-time van 4 machines

Opgave 12: Bonusopdracht Selection sort
Pas het programma dat je geschreven hebt in hoofdstuk 8 aan voor de Pipelined machine met forwarding.
Minimaliseer de executietijd! Hoeveel klokpulsen zijn er nodig om je programma te executeren?

9.7 Terugblik en historisch perspectief

Terugblik

Je hebt kennis gemaakt met de werking van een eenvoudig processormodel. Je hebt inzicht gekregen in de architectuur van een processor volgens het Harvard model. Je weet uit welke componenten de processor is opgebouwd en dat er verschillende typen instructies zijn en dat ieder type een eigen datapad heeft.
Door de twee gescheiden geheugens van een Harvard machine is het mogelijk in één klokpuls één instructie uit te voeren. Bij de in dit hoofdstuk behandelde machine is het aantal instructies slechts dertien. Deze machine is een extreem voorbeeld van een Reduced Instruction Set Computer (RISC). Ook hebben alle instructies hetzelfde instructieformaat. Bij andere RISC-machines zoals de Mips zijn er maar een klein aantal (vier) verschillende instructieformaten aanwezig. Voor de realisatie van een kleine instructieset bestaande uit basisinstructies zijn relatief weinig schakelingen nodig. Het is zelfs bij de machine van dit hoofdstuk niet nodig een instructie te decoderen, want iedere controlelijn is één bit van de opcode. Het decoderen gebeurt dus ‘hardwired’. Hierdoor is de executietijd kort. Ingewikkelder instructies worden ‘opgeknipt’ in basisinstructies. Voor implementatie van een pipeline is een eenvoudige structuur zoals de Harvard machine zeer geschikt.
Andere voordelen van RISC-architecturen zijn:

1. geschikt voor het bouwen van efficiënte compilers voor hogere programmeertalen;
2. heeft minder poorten nodig en is relatief eenvoudig te maken en verder te miniaturiseren.

Historisch perspectief: Von Neumann architecture

Een verre voorloper van de Harvard machine is de Von Neumann machine. Deze machine heeft een gemeenschappelijk geheugen dat zowel voor de gegevens als voor het programma wordt gebruikt. Voor het uitvoeren van een instructie zijn bij deze machine minstens twee klokpulsen nodig. Het is namelijk niet mogelijk om binnen één klokpuls eerst het instructiedeel en daarna het datadeel van het geheugen te benaderen. Mede daarom staat de Harvard architectuur aan de basis van alle moderne computers.

Von Neumann architecture refers to computer architectures that use the same storage device for both instructions and data. The term originated from *First Draft of a Report on the EDVAC* (Electronic Discrete Variable Electronic Computer) (1945), a paper written by the famous mathematician John von Neumann that proposed the stored program concept.

The key principles of this concept are:
- Instructions are represented as numbers
- Programs can be stored in memory to be read or written just like numbers

Von Neumann computers spend a lot of time moving data to and from the memory, and this slows the computer (this problem is called von Neumann bottleneck).

CISC

Nadat RISC-architecturen begin jaren 80 waren ontwikkeld, werden andere processoren vaak CISC (Complex Instruction Set Computers) genoemd. CISC processoren kennen vele instructies, waarvan het dataformaat ook nog eens heel divers is. Voorbeelden hiervan zijn VAX- en IBM-computers en processoren van Intel. Vanaf de 486 processor heeft ook de kern van de Intel-processoren een RISC-architectuur.

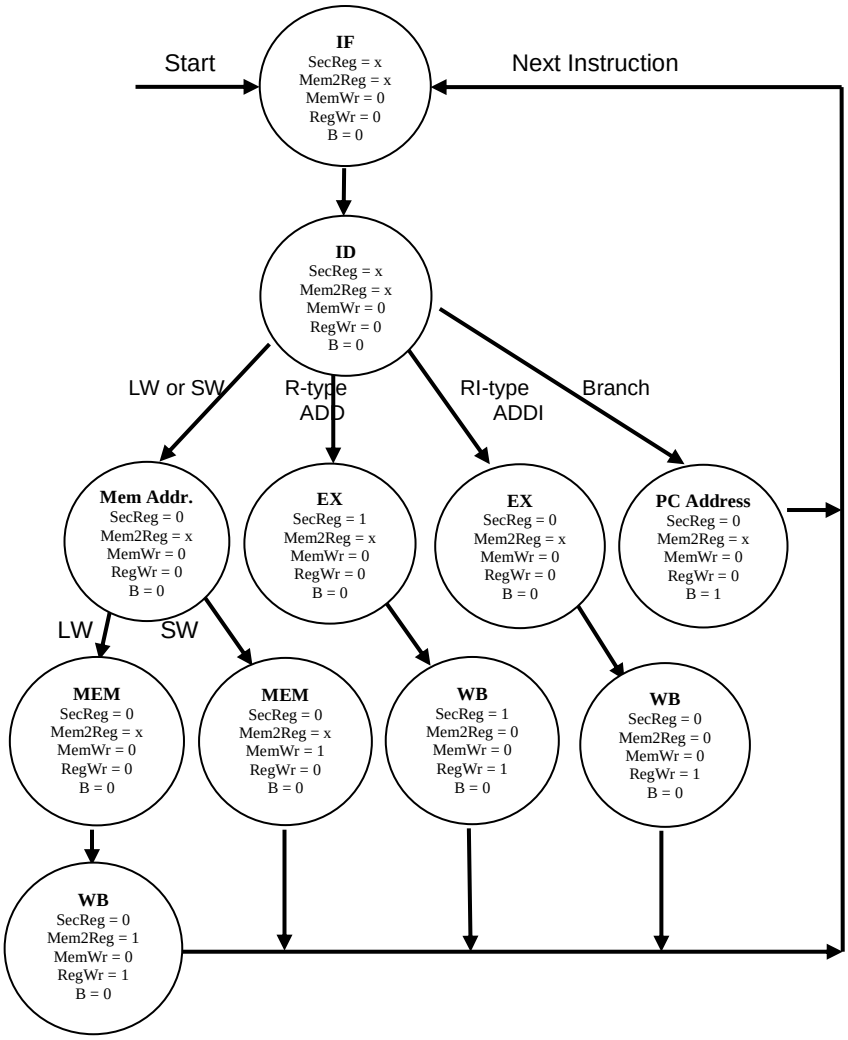
Multicycle machines

In het tijdperk dat er nog geen RISC-architecturen waren, was één van de methoden om een verbetering van de performance te verkrijgen gebaseerd op het aantal fasen dat iedere instructie gebruikte. Iedere fase heeft zijn eigen cycle. Uit tabel 9.4 blijkt dat voor het uitvoeren van een Branch-instructie maar drie cycles nodig zijn. Voor een Load Word-instructie zijn dat er vijf. De executietijd van een Branch-instructie is dan ook 40% lager dan van een Load Word-instructie. Als een programma naast andere instructies evenveel Branches als Load Words bevat dan is de CPI van dat programma precies 4. T.o.v. een Single Cycle machine wordt dan een verbetering van de performance gerealiseerd van 20%.

Fase / Type instr.	Instruction fetch	Instruction decode	Execute	Memory Access	Write Back	Aantal cycles
Aritmetisch (bijv. ADD)	J	J	J	N	J	4
Immediate (bijv. ADDI)	J	J	J	N	J	4
Branch	J	J	J	N	N	3
Store Word	J	J	J	J	N	4
Load Word	J	J	J	J	J	5

Tabel 9.4: Type instructie met bijbehorende fases

Finite State Machine
Bij de aansturing van een Multicycle machine wordt gebruik gemaakt van een zgn. Finite State Machine. In figuur 9.9 is het state diagram van Multicycle architectuur weergegeven.



Figuur 9.9: Finite state control van een Multicycle machine

9.8 Begrippenlijst

Branch hazard. Pipeline bubbles die ontstaan als bij een branch-instructie de branch genomen wordt. De volgende instructies die al in de pipeline zitten worden hierbij gewist.

Branch Prediction Unit. Een schakeling die voorspelt of de branch genomen wordt of niet.

CISC. Afkorting van Complex Instruction Set Computer. Dit is een processor die werkt met een groot aantal instructies en een groot aantal verschillende instructieformaten.

CPI. Het gemiddelde aantal klokpulsen per instructie

Data hazard. Een fout die optreedt bij data-afhankelijkheid. D.w.z. dat het resultaat van een eerdere instructie nodig is voor een volgende instructie terwijl deze nog niet beschikbaar is in de register-file.

Forwarding. Het resultaat van een ALU-bewerking die nodig is voor een volgende of daaropvolgende instructie wordt afgetapt van één der pipeline registers en teruggevoerd naar één der ALU-ingangen.

Latency. De time delay voor het uitvoeren van één instructie.

RISC. Afkorting van Reduced Instruction Set Computer. Dit is een processor die werkt met een kleine set basisinstructies en een beperkt aantal instructieformaten.

Throughput. Het aantal uitgevoerde instructies in een bepaalde tijd.

Von Neumann architecture. Een machine met één geheugen voor zowel instructies als data.

Von Neumann bottleneck. Het tijdverlies dat optreedt doordat de processor in een Von Neumann architectuur voor één Load- of Store-instructie twee keer toegang tot het relatief trage geheugen nodig heeft.

9.9 Tentamenvragen

1. Wat is de relatie tussen: CPU time, CPI, Clock rate en Instruction count?

2. Wat is de relatie tussen: CPU time, CPI, Clock Cycle time en Instruction count.

3. Verklaar het verschil in "performance" tussen een Single Cycle machine en een Pipelined machine.

4. Wat is het verschil tussen de “clocking methodology” van de “single-cycle” en de “pipelined” implementatie.

5. Leg uit dat pipelining de throughput verbetert, maar niet de instructie latency.
6. In een 5-stage stage pipeline zonder forwarding wordt het nevenstaand programma geëxecuteerd. De delay van de componenten van alle fases is 20 ns. Hoeveel cycles zijn hiervoor nodig. Wat is de CPU time en de CPI van dit programma? Welk register wordt geschreven aan het einde van de vijfde cycle?

```
ADD $2, $3, $1
SUB $4, $3, $5
ADD $5, $3, $7
ADD $7, $6, $1
ADD $8, $2, $6
ADD $12, $3, $7
ADD $13, $3, $6
```

7. De machine van opgave 6 wordt vernieuwd. De delay van beide geheugens blijft 20 ns. De delay van de componenten van de overige fases wordt verlaagd naar 10 ns. De IF-fase en de MEM-fase nemen nu twee klokpulsen in beslag. Het aantal stages neemt dus toe tot 7. Hoeveel cycles zijn er in dat geval nodig voor het programma van opgave 7? Wat is de CPU time en de CPI in dit geval.

8. Geef een manier aan om een pipeline toch, ondanks een "conditionele jump", gemiddeld in de tijd gezien beter gevuld te laten.
9. Welke twee “hazards” kunnen bij een pipeline optreden? Geef aan op welke manier deze hazards zo goed als mogelijk verholpen kunnen worden.

10. Geef drie methoden om een data hazard op te lossen.

11. Wat is “forwarding”? Welke “hazard” wordt hiermee opgelost? Bij welke combinatie van instructies is het ondanks forwarding toch nodig een “bubble/NOP” te implementeren?

```
ADD $2, $3, $1
AND $12, $2, $5
OR $13, $6, $2
SUB $7, $6, $1
SW $8, 100, $12
LW $9, 100, $13
```

12. In een 5-stage stage pipeline *zonder* forwarding wordt het nevenstaand programma geëxecuteerd.
    a) Herschrijf het programma zodat het correct werkt door NOP's toe te voegen. Laat hierbij de volgorde van de instructies ongemoeid.
    b) Herschrijf het programma door de instructies te verwisselen. Minimaliseer hierbij het aantal NOP's.

13. Laat aan de hand van een tekening zien hoe door toepassing van een "instruction pipeline" de executie-tijd verkort kan worden. Beschrijf welke hinderlijke situatie onstaat wanneer er een conditionele jump moet worden uitgevoerd.

14. Laat zien dat als een pipeline N stages kent de speed-up in het ideale geval N is.

15. Leg uit dat met "instruction scheduling" load-use data hazards opgelost kunnen worden. Geef een voorbeeld met behulp van de nevenstaande code:

Instr 0	LW $1,100, $2
Instr 1	ADD $3, $1,$5
Instr 2	AND $3, $4,$5
Instr 3	SUB $2, $4, $2

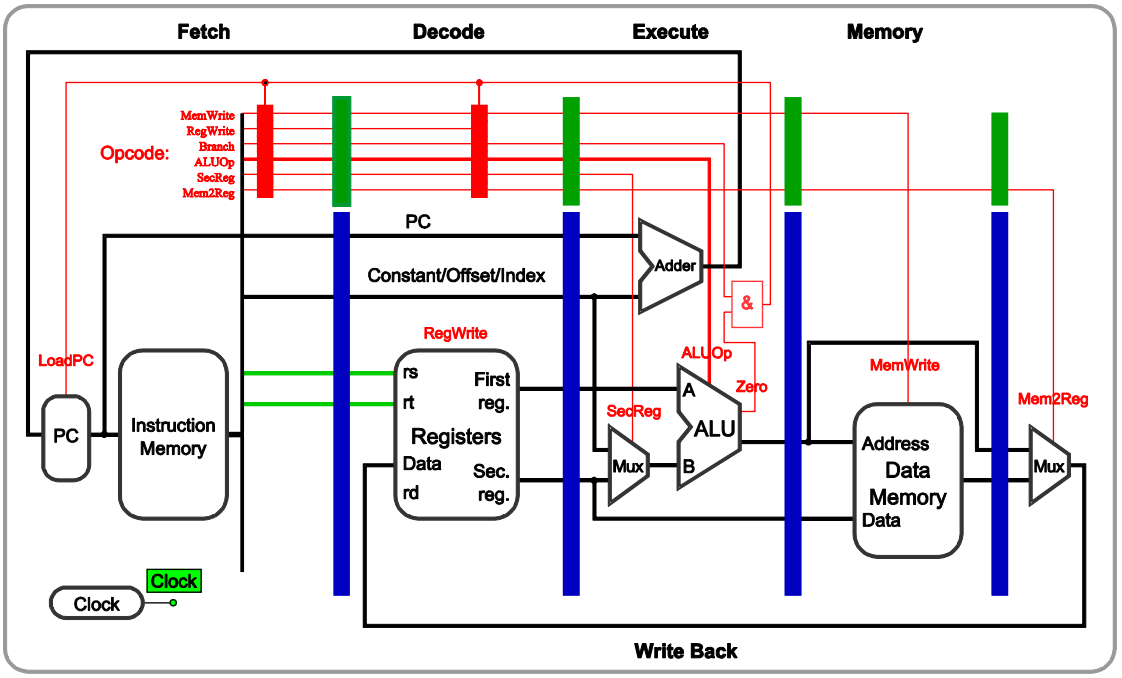
16. Wat is het nadelige effect van "pipeline stages" die niet elk even veel tijd nemen?

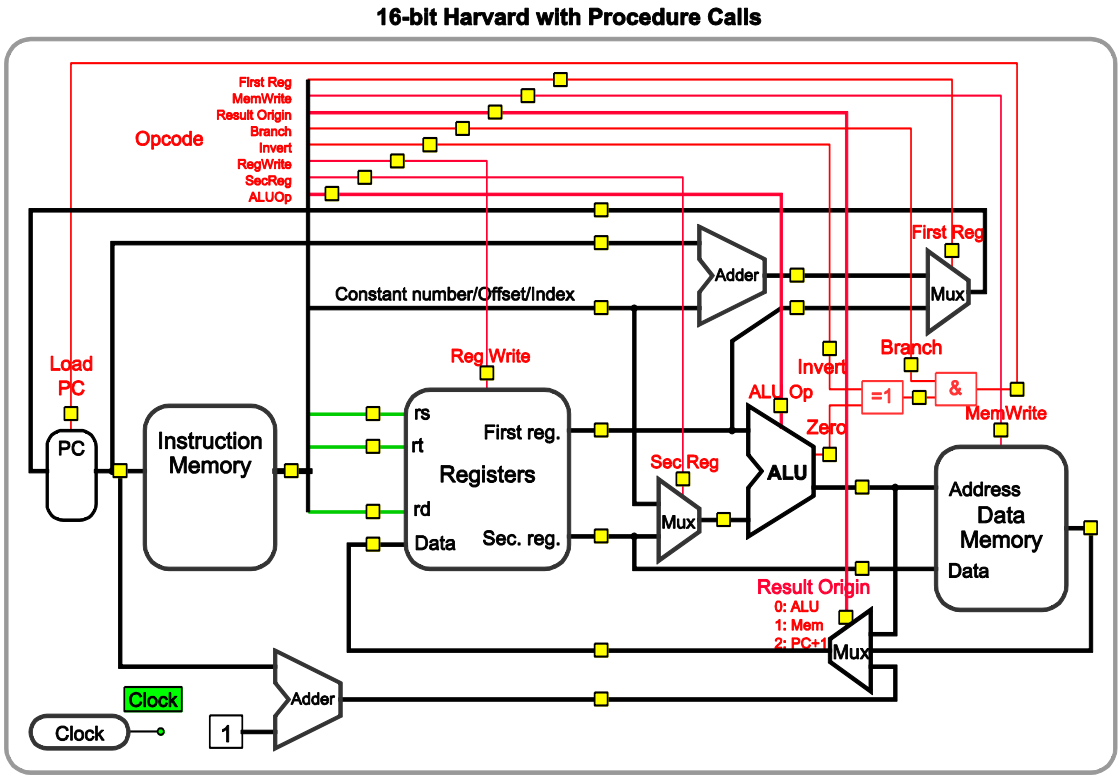
17. Met instruction scheduling kan het probleem van branch of control hazards in een pipeline in sommige gevallen opgelost worden. Beschrijf een andere manier om het probleem van control hazards aan te pakken.

18. Een Single Cycle Harvard machine voert een programma van 100 instructies uit in 100ns. Een 5-stage pipelined machine voert hetzelfde programma uit in 25 ns. Wat is de Cycle length van de pipelined machine? Wat is de CPI van deze machine voor dit programma?

19. Van de component Registerfile ontbreekt zowel de bedrading van de RegWrite-ingang als van de ingang rd. Teken de bedrading van deze ingangen in onderstaand schema.

Pipelined 16 bit Harvard Architecture





Hoofdstuk 10: Hoe werkt een procedure?

10.1 Leerdoelen

Om structuur aan programma's te geven en blokken code als module te kunnen inzetten op meerdere plaatsen in een programma (en andere programma's), wordt gebruik gemaakt van procedures. Een procedure wordt ook subroutine, functie of static method genoemd. Om een procedure uit te kunnen voeren, wordt de instuctieset van de Harvard Machine uit hoofdstuk 8 uitgebreid tot een "Jumper Machine" door toevoeging van twee instructies: een instructie JSR (Jump to SubRoutine) en een instructie RETURN. RETURN laat het programma terugkeren naar de plek van aanroep. Procedures communiceren met de rest van een programma via parameters en 'return values'. Het is dus noodzakelijk in een 'contract' vast te leggen, in welke registernummers parameters, return values en terugkeeradres worden opgeslagen. Registers hebben dus niet langer een "general purpose"-functie zoals bij de modellen uit voorgaande hoofdstukken.

Aan het eind van dit hoofdstuk weet je:

- het mechanisme waardoor een programma naar een procedure springt;
- hoe de communicatie tussen de aanroeper en de procedure verloopt;
- het mechanisme waardoor terug naar het hoofdprogramma wordt gesprongen;
- met welke componenten de Harvard Machine hiertoe moet worden uitgebreid;
- wat een 'stack' is en hoe deze werkt;
- het verschil tussen leaf- en nonleaf functies;
- wat een recursieve procedure is.

Aan het einde van dit hoofdstuk kun je:

- een programma schrijven dat een aantal procedures bevat;
- een programma schrijven met functies die weer functies aanroepen.

10.2 Wat gebeurt er precies als een procedure wordt uitgevoerd?

Een algemene voorstelling van een procedure is: (val_1, .., val_k) = f(arg_1, … , arg_n). Het hoofdprogramma of de procedure van waaruit de procedure wordt aangeroepen, noemen we de 'aanroeper' en de procedure de 'aangeroepene'. Het uitvoeren van een procedure verloopt in een aantal stappen. We onderscheiden de volgende 6 stappen:

1. Plaats argumenten op een plek zodat ze toegankelijk zijn voor de procedure.
2. Bewaar de toestand van de aanroeper.
3. Draag de uitvoering van het programma over aan de procedure.
4. Voer de procedure uit.
5. Plaats de resultaten op een plek zodat ze toegankelijk zijn voor de aanroeper.
6. Draag de uitvoering van het programma weer over aan de aanroeper, nét na de aanroep.

De implementatie van deze 6 stappen is:

1. Sla argumenten op in daarvoor bestemde registers $arg1 …$arg4.
2. Bewaar het adres van de eerstvolgende instructie na terugkeer van de procedure in register $ra (ra is returnadres).
3. Voer de instructie Jump SubRoutine (JSR) uit.
4. Voer de instructies die bij de procedure horen uit.
5. Sla de resultaten van de functieaanroep op in registers $val1 en $val2.
6. Voer instructie RETURN $ra uit.

Voorbeeld 1: Procedure die geen andere procedure aanroept

We lichten de hierboven genoemde 6 stappen toe aan de hand van een procedure met de naam func. Deze procedure roept geen andere procedure aan en wordt daarom ook wel een niet geneste of een 'leaf procedure' genoemd. Het is in dit specifieke geval niet nodig registerwaarden in het geheugen op te slaan, omdat er voldoende registers beschikbaar zijn. Hiernaast staat de definitie van deze procedure in de taal C en hieronder is de assembler code voor de Jumper Machine weergegeven. De procedure wordt aangeroepen met de waarden 10, 20, 3 en 4.

```
int func(int j, int k, int l, int m)
 {
        int f;
        f = (j+k) - (l+m);
        return f;
}

v = func( 10, 20, 3, 4 );
```

Zoals eerder vermeld, zijn de registers niet allemaal meer beschikbaar voor algemeen gebruik maar hebben een aantal ervan een speciale functie. Aliassen hiervoor zijn:

- $arg1..$arg4: vier argument registers om parameters in op te slaan.
- $val1..$val2: twee registers bestemd voor return values.
- $ra: hier wordt het terugkeeradres in opgeslagen.
- $tmp1..$tmp4: registers voor tijdelijk gebruik.

```
LOADI $arg1, 10          # step 1
LOADI $arg2, 20          # step 1
LOADI $arg3, 3           # step 1
LOADI $arg4, 4           # step 1
JSR func                 # step 2 & 3 stores PC + 1 into register $ra and jumps to func.
HALT                     # the result is now in $val1.

func:  #----------------------  procedure func ------------------------------------
ADD $tmp1, $arg1, $arg2  # step 4 compute j + k and store in temporary register $temp1.
ADD $tmp2, $arg3, $arg4  # step 4 compute l + m.
SUB $tmp3, $tmp1, $tmp2  # step 4 compute f = (j+k) - (l+m).
COPY $val1, $tmp3        # step 5 transfer result f to register $val1.
RETURN $ra               # step 6 return to caller.
```

Opmerking: Verwar de betekenis van het c-statement *return f* niet met de assembler-instructie *RETURN $ra* want f is een waarde en $ra is een register.

10.3 Procedures en de stack

Het komt veel voor dat registerwaarden moeten worden bewaard omdat deze verder in het hoofdprogramma nog worden gebruikt. Ook kan het aantal argumenten van een procedure groter zijn dan vier en het aantal return values groter zijn dan twee. Het aantal registers van de 'Jumper Machine' uit dit hoofdstuk is beperkt tot in totaal 16. Dat is vaak niet genoeg om alle benodigde waarden in op te slaan. Waar moeten deze waarden dan wel worden bewaard? De enige plek waar dat kan is in het datageheugen[12].

Stack

Registerwaarden worden in een apart deel van het datageheugen opgeslagen, de zogenaamde stack. De stack groeit van hoge naar lage adressen. Het hoogste adres in de door ons gebruikte machine is 511 (0x01FF). Het adres van het laatst geplaatste item op de stack wordt opgeslagen in een register met de naam $sp. Dit is een afkorting van stack pointer. Dit adres wordt de top van de stack genoemd.

Kaart van het geheugen	
adres	data
511	Stack
510	001E
509	0028
sp → 508	0032
Heap[13] Array's e.d.	
Static data Text Reserved	

Tabel 10.1

```
@include "16bitJumper.wasm"
# int leaf_example(int j, int k, int l, int m)
# {
# int f;
# f = (j+k) - (l+m);
# return f;
# }

.data MyData : REGISTERS
@include "register_constantsLeaf.wasm"
# $s1 = 0x001E; $s2 = 0x0028; $s3 = 0x0032
# these values must be saved and restored

.code MyCode : REGSTACK, MyData
# this is the main program
# v = leaf_example(10,20,3,4);
LOADI $arg1, 10
LOADI $arg2, 20
LOADI $arg3, 3
LOADI $arg4, 4
JSR leaf_example          # stores PC in register $ra
HALT                      # the result is now in $val0

  #---------- procedure leaf_example ----------
leaf_example:
SUBI $sp, $sp, 3          # make space for 3 items on stack
SW $s1, 2, $sp            # save $s1 for use afterward
SW $s2, 1, $sp            # save $s2 for use afterward
SW $s3, 0, $sp            # save $s3 for use afterward
ADD $s1, $arg1, $arg2     # compute g + h
ADD $s2, $arg3, $arg4     # compute i + j
SUB $s3, $s1, $s2         # compute (g+h) - (i+j)
COPY $val1, $s3           # transfer result to return value reg.
LW $s3, 0, $sp            # restore $s3
LW $s2, 1, $sp            # restore $s2
LW $s1, 2, $sp            # restore $s1
ADDI $sp, $sp, 3          # remove 3 items from top stack
RETURN $ra                # return to caller
```

Voorbeeld 2: Leaf procedure waarbij registerwaarden moeten worden opgeslagen.

We gebruiken hetzelfde programma als in voorbeeld 1. Alleen gaan we er nu vanuit dat tijdens het uitvoeren van de procedure de waarden van drie registers moeten worden bewaard. Dit zijn de registers $s1, $s2 en $s3. Met s wordt bedoeld saved temporary register. Dit zijn registers voor tijdelijk gebruik waarvan de waarde moet worden bewaard omdat die verderop in het programma nog gebruikt gaat worden. In bovenstaande tekstbox is de assembly code ervan weergegeven. De beginwaarde van de stack pointer is 511. Na de instructie SUBI $sp, $sp, 3 wijst de sp naar adres 508.

Voorbeeld 3: Non-leaf procedure.

Een non-leaf of geneste procedure is een procedure die een andere procedure aanroept. Bij het onderstaande voorbeeld worden drie getallen opgeteld door de functie Add3. In het hoofdprogramma wordt een functie Add5 aangeroepen. Deze functie telt vijf getallen op door

[12] De tijd die het kost om naar het datageheugen te schrijven, is een veelvoud van de tijd die nodig is om naar een register te schrijven. Compilers worden geoptimaliseerd om dit zo min mogelijk te laten plaatsvinden.

[13] De heap is het geheugengebied waar dynamisch gealloceerde gegevens worden bewaard.

twee keer de functie add3 aan te roepen. Add5(j, k, m, n, o) = Add3(Add3(j, k, m), n, o). Het terugkeeradres $ra moet meteen na aanroep van Add5 worden bewaard, omdat dit adres anders wordt overschreven bij de aanroep van Add3. Dit geldt ook voor de argumenten n en o. De procedure Add5 wordt aangeroepen met de waarden 1, 2, 3, 4 en 5.

```
@include "16bitJumper.wasm"

.data MyData : REGISTERS
@include "register_constants.wasm"

.code MyCode : REGSTACK, MyData
        # the main program
PC
0       LOADI     $arg1, 1
1       LOADI     $arg2, 2
2       LOADI     $arg3, 3
3       LOADI     $arg4, 4
4       LOADI     $s1, 5
5       JSR       Add5
6       HALT                        # endless loop

        #---------- Add5 procedure ------------------------
        Add5:
7       SUBI      $sp,$sp,3         # create space for 3 items on the stack
8       SW        $ra, 2, $sp       # save $ra  for use afterward
9       SW        $arg4, 0, $sp     # $arg4 → stack
10      SW        $s1, 1, $sp       # $s1 → stack
11      JSR       Add3              # computes $val1 = $arg1 + $arg2 + $arg3
12      COPY      $arg1, $val1      # $arg1 ← $val1 = $arg1 + $arg2 + $arg3
13      LW        $arg2, 0, $sp     # $arg2 ← $arg4
14      LW        $arg3, 1, $sp     # $arg3 ← $s1
15      JSR       Add3              # computes $val1 = $val1 + $arg4 +$s1
16      LW        $ra, 2, $sp       # restore $ra
17      ADDI      $sp, $sp, 3       # remove 3 items from top stack
18      RETURN    $ra               # return to caller

        #---------- Add3 procedure ------------------------
        Add3:
19      ADD       $val1, $arg1, $arg2
20      ADD       $val1, $val1, $arg3
21      RETURN    $ra
```

Opdracht: Bestudeer de bovenstaande code en beantwoord de volgende vragen:
Vraag 1: Welke waarden neemt het register $ra achtereenvolgens aan tijdens het uitvoeren van het bovenstaande programma?
Antwoord: ……., ……..,……..,……..

Vraag 2: Vul in tabel 10.2 in welke getallen op welke adressen op de stack worden bewaard. De stack begint vanaf adres 511. Op dit adres staat 0xFFFF.

Data memory	
adres	data
sp → 508	…….
509	…….
510	…….
511	FFFF

Tabel 10.2

10.4 Recursieve procedures

Een procedure is recursief als deze zichzelf aanroept. Recursie is een complex onderwerp. Voor sommige problemen is het echter handig dat procedures zichzelf kunnen aanroepen. Een eenvoudig voorbeeld van een recursief algoritme is: f(0) = 0; f(n) = n + f(n-1) voor n > 0. De vertaling van dit algoritme naar een functie in de taal C is:

```
int f( int n )
{
        if ( n == 0 )
                return 0;
        else
                return( n + f( n - 1 ));
}
```

Als de procedure wordt aangeroepen met de waarde 4, dan gebeurt er het volgende:
f(4) = 4 + f(3) waarbij:
f(3) = 3 + f(2) waarbij:
f(2) = 2 + f(1) waarbij:
f(1) = 1 + f(0) waarbij:
f(0) = 0.
De procedure wordt aangeroepen met de resp. argumenten 4, 3, 2, 1 en 0. Het aantal keren dat f wordt aangeroepen is 5. Dit aantal is gelijk aan het aantal 'rechter sluithaakjes +1' in de expressie f(4) = 4 + (3 + (2 + (1 + (0)))).

Het argument en het terugkeeradres moet bij elke functieaanroep op de stack worden bewaard omdat ze anders worden overschreven. In omgekeerde volgorde moeten deze waarden weer van de stack worden afgehaald. Dit gaat als volgt:
Aanroep 1: f(4); stack ← terugkeeradres aanroep 1 en waarde 4
Aanroep 2: f(3); stack ← terugkeeradres aanroep 2 en waarde 3
Aanroep 3: f(2); stack ← terugkeeradres aanroep 3 en waarde 2
Aanroep 4: f(1) = 1 stack ← terugkeeradres aanroep 4 en waarde 1
Aanroep 5: De voorwaarde: n == 0 is bereikt.
stack → terugkeeradres 4 en waarde 1; f(1) = 1 + 0 = 1
stack→ terugkeeradres 3 en waarde 2; f(2) = 2 + 1 = 3
stack →terugkeeradres 2 en waarde 3; f(3) = 3 + 3 = 6
stack → terugkeeradres 1 en waarde 4; f(4) = 4 + 6 = 10

Opdracht: Bestudeer de code op de volgende bladzijde en beantwoord de volgende vragen:

Vraag 1: Welke waarden neemt het register $ra achtereenvolgens aan tijdens het uitvoeren van dit programma?
Antwoord: ……., ……..,……..,……..

Vraag 2: Vul in tabel 10.3 in welke getallen op welke adressen op de stack worden bewaard. Vul ook in bij de hoeveelste functieaanroep dit gebeurt.

adres	nummer functie aanroep	waarde
502		
503		
504		
505		
506		
507		
508		
509	1	0002
510	1	0004
511		FFFF

Tabel 10.3

```
# Opgave 7 Recursion.wasm
@include "16bitJumper.wasm"
.data MyData : REGISTERS
@include "register_constants.wasm"
.code MyCode : REGSTACK, MyData

            # this is the main program it computes f(4)
PC
0           LOADI           $arg1, 4        # f expects n in $arg1
1           JSR             f               # call procedure f
2           HALT

            #---------------------- procedure f------------------------------
            # $ra contains the return adress
            # $arg1 is the single input register. On call it contains n
            # $val1 is the single output register. On return it contains f(n)
3  f:       BZ              $arg1, f_end            # check for recursion base case
4           SUBI            $sp, $sp, 2             # make place for two items on the stack
5           SW              $ra, 0, $sp             # save return address
6           SW              $arg1, 1, $sp           # store n, we still need it at the end

            # compute f(n-1)
7           SUBI            $arg1, $arg1, 1         # put n-1 in $arg1
8           JSR             f                       # recursive call

            # now f(n-1) is in $val1
9           LW              $tmp1, 1, $sp           # read n from the stack
10          ADD             $val1, $tmp1, $val1     # compute f(n) = n + f(n-1)

11          LW              $ra, 0, $sp             # restore the return address from the stack
12          ADDI            $sp, $sp, 2             # remove 2 items from top stack
13          RETURN          $ra                     # return

            # --------- f_end handle the special case when n = 0 ---------
14  f_end:  COPY            $val1, $zero            # f( 0 ) = 0; $zero is always 0
15          RETURN          $ra                     # we can have multiple returns from a
procedure
```

10.5 Het datapad van de instructies JSR en RETURN

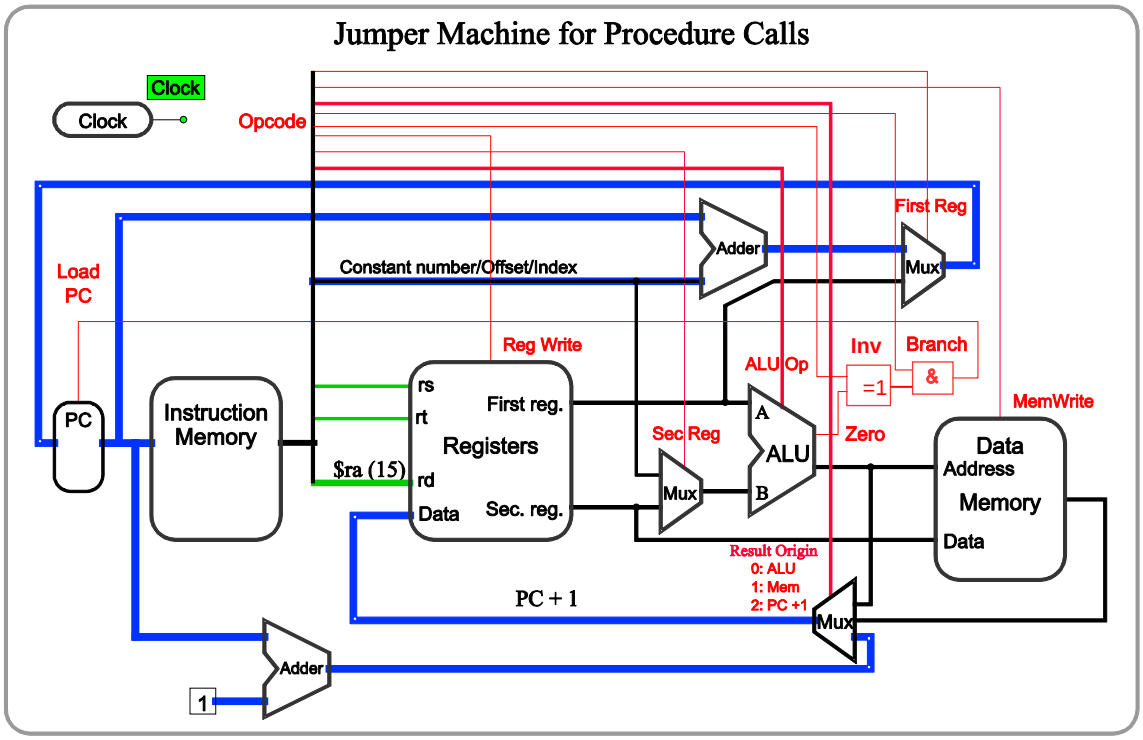
Voor het uitvoeren van de instructies JSR en RETURN is het nodig de Harvard Machine uit te breiden met extra hardware.

Jump to SubRoutine
De instructie JSR label zorgt ervoor dat de PC naar de instructie aangegeven door label springt. Dit is hetzelfde als bij de instructie BRA (BRanch Always) gebeurt. Gelijktijdig wordt het adres, van de instructie die moet worden uitgevoerd als de procedure is afgelopen, opgeslagen in register $ra. De instructie JSR (Jump to SubRoutine) bestaat dus uit twee datatransfers die gelijktijdig worden uitgevoerd:

1. PC ← PC + Offset # Net zoals bij de instructie Branch Always.
2. $ra ← PC + 1 # Het adres van de volgende instructie wordt opgeslagen in $ra.
   # Dit is de instructie die na RETURN moet worden uitgevoerd.

In figuur 10.1 is het datapad van de beide datatransfers met dikke lijnen weergegeven. Voor de datatransfer '$ra ← PC + 1' is een opteller geïmplementeerd die de waarde van de PC met 1 verhoogt. Ook zijn er nu drie bronnen van waaruit naar het destination register rd kan worden geschreven: De ALU, het Data Memory en de PC. Daarom is de multiplexer "Mem → Reg" vervangen door de multiplexer "Result Origin". Deze multiplexer heeft twee controlebits en drie data-ingangen. Voor de controlebits deze multiplexer geldt:

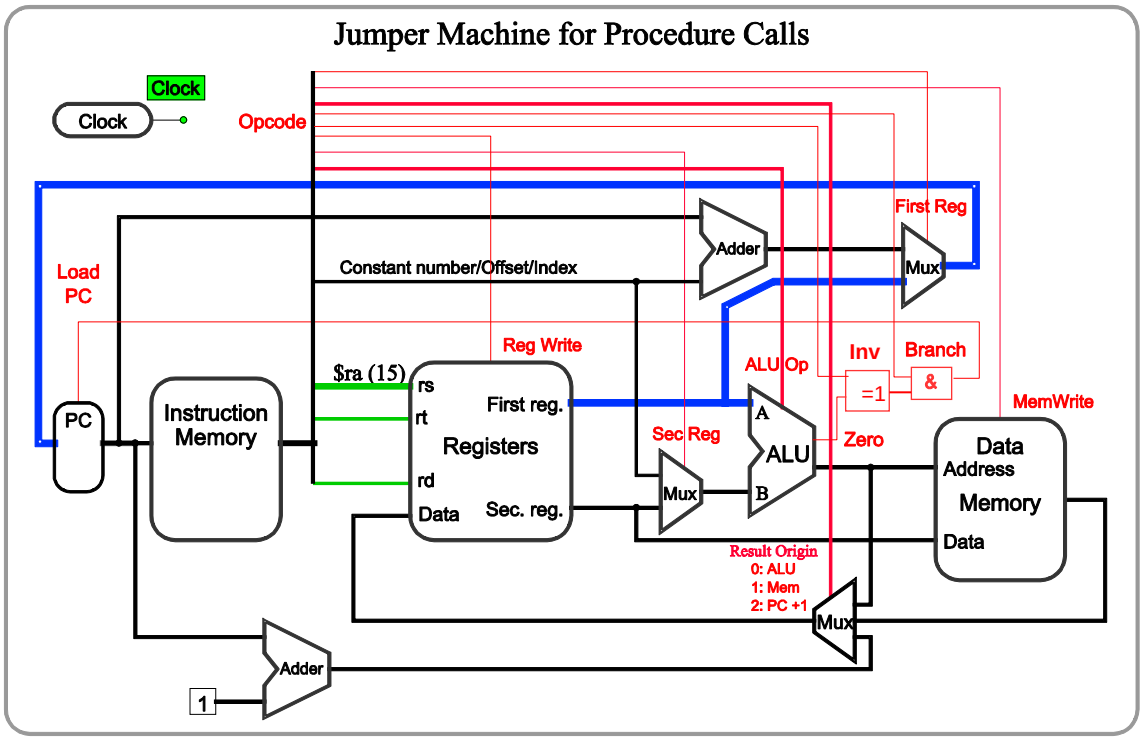
- Result Origin = 0, 0: data van de ALU naar één van de registers;
- Result Origin = 0, 1: data van het geheugen naar één van de registers;
- Result Origin = 1, 0: data van de PC naar één van de registers.



Figuur 10.1: Datapad van de instructie Jump to Subroutine

RETURN
De instructie JSR slaat het adres van de volgende instructie, gezien vanuit de aanroeper, op in register $ra. RETURN doet het omgekeerde. De waarde van $ra gaat naar de Program Counter, waardoor het programma terugkeert naar de plek net na de aanroep. Om deze datatransfer te kunnen uitvoeren, is een extra multiplexer “First Reg” aangebracht, zodat de uitgang van het First Register verbonden kan worden met de PC. Voor de aansturing van deze multiplexer geldt:

- First Reg = 0: de Adder is verbonden met de Program Counter;
- First Reg = 1: het First register is verbonden met de Program Counter;



Figuur 10.2: Datapad van de instructie RETURN from Subroutine

10.6 Practicum met de Jumper Machine

Tabel 10.4 geeft ‘het contract’ weer tussen ‘de aanroeper’ en ‘de aangeroepene’ bij een procedureaanroep. In dit contract zijn de afspraken vastgelegd over welke functie aan welk registernummer is toegekend. De kolom ‘Preserved’ geeft aan of de inhoud van het desbetreffende register moet worden bewaard. Deze registeraanduidingen ($zero, $val1, . . .) kunnen gebruikt worden door het bestand register_constants.wasm in een programma te @include’en.
Tip: Gebruik de slider om het stack-gedeelte van het Data Memory op het scherm te bekijken.

Index	Alias	Function	Preserved
0	$zero	Always zero	implicit
1…2	$val1...$val2	Return values	no
3…6	$arg1…$arg4	Procedure parameters	no
7…10	$tmp1…$tmp4	Temporaries	no
11…13	$s1…$s3	Saved temporaries	yes
14	$sp	Stack pointer	yes
15	$ra	Return address	yes

Tabel 10.4: Contract gebruik registers

Opgave 1 instructieformaat JSR- en RETURN-instructies
Vul tabel 10.5 verder in. Geef in het ALU veld aan welke operatie wordt gebruikt: ADD, SUB, AND of B[14]. Geef in het veld: Res. Org. (Result Origin) de bron aan waar de gegevens voor rd vandaan komen: ALU, Mem of PC +1. Markeer velden die er niet toe doen met een x.

Instructie	Instructieformaat Jumper Machine (totaal 39 bits)											
	Opcode								First Reg. rs	Sec. Reg. rt	Dest. Reg. rd	getal/ offset/ index
	Frst Re	M W	Res Org	B	Inv	RW	Sec Re	ALU				
Aant. Bits	1	1	2	1	1	1	1	3	4	4	4	16
BEQ		0	x	1	0	0	1	SUB	rs	rt	x	offset
SW		1	x	0	x	0	0	ADD	rs	rt	x	index
LW		0	Mem	0	x	1	0	ADD	rs	x	rd	index
JSR *label*												
RETURN *reg*												

Tabel 10.5: Instructieformaat: JSR- en RETURN-instructies Harvard-machine with procedure calls

Opgave 2
Waar of onwaar? Verklaar je antwoord.

- Het is mogelijk om de waarde van de PC met één instructie in het datageheugen te schrijven?

- Het is mogelijk om een RETURN uit te voeren en tegelijkertijd het tweede register naar het geheugen te schrijven op een constant adres? We doen dit door de ALU een 'laat de waarde op ingang B door'-operatie te laten uitvoeren, zodat het immediate veld wordt gebruikt als adres.

- De instructie: JSR label slaat een relatief adres in de registerfile op. M.a.w, het adres aangeduid door label wordt verminderd met de PC (van voor de sprong).

- De register-instructie RETURN reg. springt naar een relatief adres. M.a.w, de waarde van *reg* wordt bij de PC (van voor de sprong) opgeteld.

Opgave 3
We willen de instructieset uitbreiden met een conditionele instructie RETZ (RETurn if Zero), waarbij de Zero vlag van de ALU wordt gebruikt om deze conditie te testen.
De syntax is: RETZ rt, reg. Vul tabel 10.6 in. Is het mogelijk om RETZ op de Jumper Machine te implementeren zonder aanpassing van de hardware? Zo nee, waarom niet?

Instructie	Instructieformaat Jumper Machine (totaal 39 bits)											
	Opcode								First Reg. rs	Sec. Reg. rt	Dest. Reg. rd	getal/ offset/ index
	Frst Reg	M W	Res Org	B	Inv	R W	Sec Re	ALU				
Aant. Bits	1	1	2	1	1	1	1	3	4	4	4	16
RETZ rt, *reg*												

Tabel 10.6: Instructieformaat: RETZ-instructie

[14] Laat de waarde op ingang B, de onderste ingang van de ALU, door.

Opgave 4

Executeer het programma Opgave4.wasm. De code van dit programma is hiernaast weergegeven. Het programma berekent de functie: id(n) = n.
Beantwoord de volgende vragen:

- Welke *twee* data-transfers vindt er plaats bij het uitvoeren van de instructie: JSR id?
- Waarom is het nodig 1 bij de waarde van de PC op te tellen?
- Wat gebeurt er precies bij het uitvoeren van de instructie: RETURN $ra?

```
# This program performs a single procedure call
# to the identity function

 @include "16 bitJumper . wasm "

 . data MyData : REGISTERS
 @include " register_constants . wasm "

 . code MyCode : REGSTACK , MyData

 # this is the main program
 LOADI $arg1, 4   # set up single argument
 JSR id           # and call identity function
 HALT             # endless loop

  #--------------- procedure id-----------------------
  id:
  # id procedure : id(n) = n
  # $ra contains the return address
  # $arg1 is the only used input register, it contains n
  # $val1 is the only used output register. It receives id(n) = n
 COPY $val1, $arg1        # simply copy input to output
 RETURN $ra               # and return
```

Opgave 5: Non-leaf example

Run het programma: opgave5.wasm
De code ervan staat in programma-voorbeeld 3. Controleer de antwoorden die je hebt ingevuld in tabel 10.2 en de antwoorden op de bijbehorende vragen.

Opgave 6: Max4

Schrijf een procedure max(j,k). De procedure retourneert de grootste waarde van de argumenten j en k. In het hoofdprogramma wordt een functie aangeroepen max4. Deze functie retourneert ook de grootste waarde van de vier argumenten. De procedure roept drie keer de procedure max aan. Max4(j, k, l, m) = max(max(j, k), max(l,m)).

Opgave 7: Recursie

Executeer het programma: opgave7Recursion.wasm. De assembler code is weergegeven in de tekstbox bij paragraaf 10.4. De C-code ervan is hier linksonder weergegeven. Controleer de antwoorden die je hebt ingevuld in tabel 10.3 en de antwoorden op de bijbehorende vragen.
Save het programma als opgave7bRecursion.wasm en wijzig de code zodat de C-routine wordt uitgevoerd die hier rechtsonder is weergegeven. Roep de procedure aan met de waarde 4.

```
int f( int n )
{
        if ( n == 0 )
                return 0;
        else
                return( n + f( n - 1 );
}
```

```
int f( int n )
{
        if ( n == 0 || n == 1 )
                return n;
        else
                return( n + f( n - 1 );
}
```

Opgave 8

Herschrijf het programma bubble-sort uit hoofdstuk 8 voor de “Jumper Machine”.
Er moeten twee procedures in voorkomen namelijk sort en swap. Swap wordt vanuit de procedure sort aangeroepen.

Opgave 9: Fibonacci

Open de file opgave9Fibonacci.wasm. Dit programma berekent de reeks van Fibonacci:
fib(0) = 0
fib(1) = 1
fib(n + 2) = fib(n + 1) + fib(n)

Executeer het programma.
Bekijk hoeveel cycles nodig zijn voor n = 2, 4, 6 en 8. Vul hiertoe tabel 10.7 in.

n	*2*	*4*	*6*	*8*	*20*
Nr. of cycles					x
Nr. of cycles na verbetering					

Tabel 10.7

Dit programma is niet efficiënt. Wijzig het zo dat tussenresultaten in het datageheugen worden opgeslagen, zodat ze niet opnieuw moeten worden berekend. Bekijk voor het aangepaste programma nogmaals hoeveel cycles er nodig zijn voor n = 2, 4, 6, 8 en n = 20. Let op dat de stack pointer 'hoog' genoeg begint, zodat de opgeslagen data niet wordt overschreven!

10.7 Historisch perspectief

Stack
Stack betekent stapelgeheugen. Oorspronkelijk was dit een geheugen waar de gegevens gestapeld werden opgeslagen. Een nieuw item kwam boven op 'de top van de stack'. Je kon alleen het laatst geplaatste item van 'de top van de stack' verwijderen. Een andere naam voor een dergelijke datastructuur is een LIFO (Last In First Out).

Stacks zijn zo populair dat ze een eigen jargon hebben namelijk pop en push.

```
push(x)     # Plaats item bovenop op de stack
v = pop()   # Verwijder bovenste item van de stack
```

Implementatie van een datastructuur met de operaties pop en push.
Stel M is Array van items en i is de index van het bovenste item.

```
push(x)     # i = i+1; M[i] = x;  De index wordt met 1 verhoogd en x wordt naar M[i]
            # geschreven.
v = pop()   # v = M[i]; i = i – 1; Het bovenste item wordt van de stack gehaald en de index
            # wordt met 1 verlaagd.
```

Forth en RPN
Begin jaren 70 van de vorige eeuw is een programmeertaal met de naam Forth ontwikkeld, die gebruik maakt van een stack i.p.v. registers. De taal maakt gebruik van vele commando's om de top van de stack te manipuleren. Voorbeelden zijn dup (dupliceer de top van de stack, swap (verwissel de twee bovenste waarden van de stack) en drop (verwijder het bovenste item van de stack. Parameters worden op de stack geplaatst voordat ze worden uitgevoerd. Bij de berekening (6 – 4) * 5 ziet de top van de stack er uit als 6, 4, -, 5, *. Eerst worden 6 en 4 van de stack gehaald voordat deze worden afgetrokken. Hierna wordt het verschil met 6 vermenigvuldigd. Deze volgorde van operanden en operatoren staat ook bekend als RPN (Reverse Polish Notation).

10.8 Begrippenlijst

Functie. Zie procedure.

Leaf procedure. Een procedure die geen andere procedure aanroept.

Non-leaf procedure. Procedure die één of meer andere procedures aanroept.

Procedure. Subprogramma dat via parameters en return values met andere delen van het programma communiceert.

Recursieve procedure. Procedure die zich zelf aanroept.

Stack. Het deel van het datageheugen waar registerwaarden tijdens het uitvoeren van procedures tijdelijk worden bewaard.

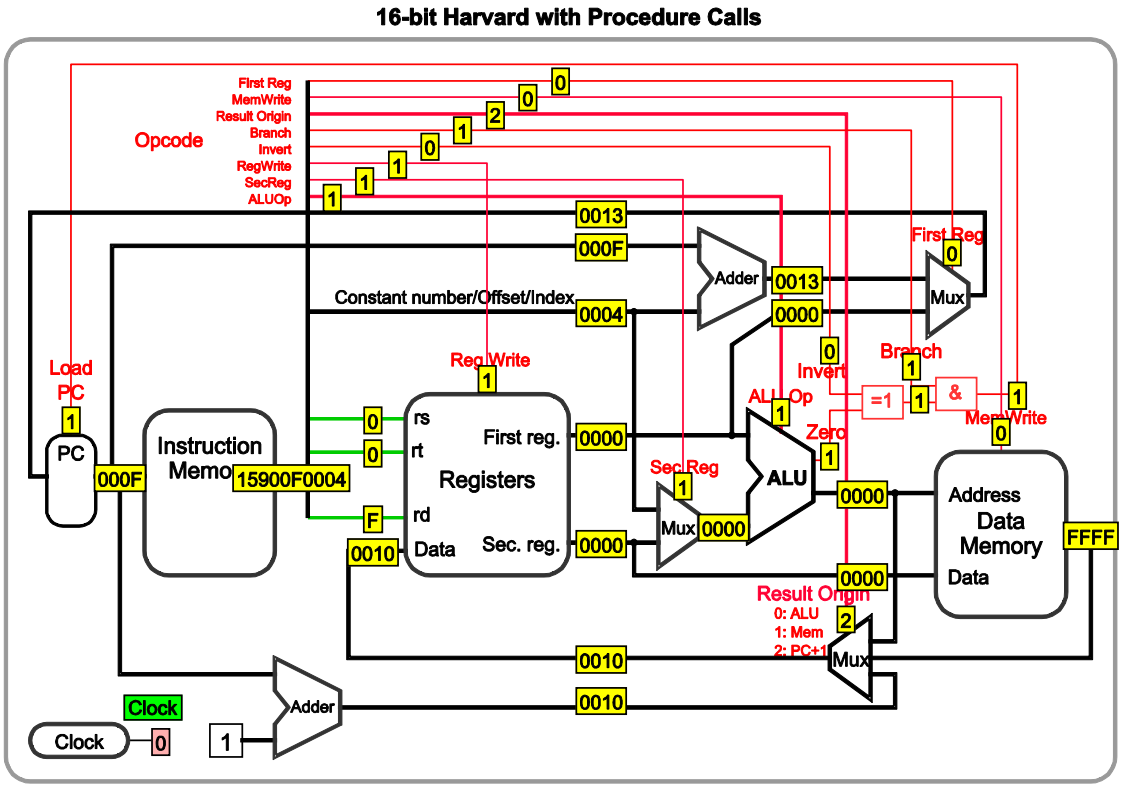
10.9 Tentamenvragen

1. Noem de 6 stappen die bij het uitvoeren van een procedure plaatsvinden.

2. Wat is een stack en waarom is deze nodig bij het uitvoeren van non-leaf procedures?

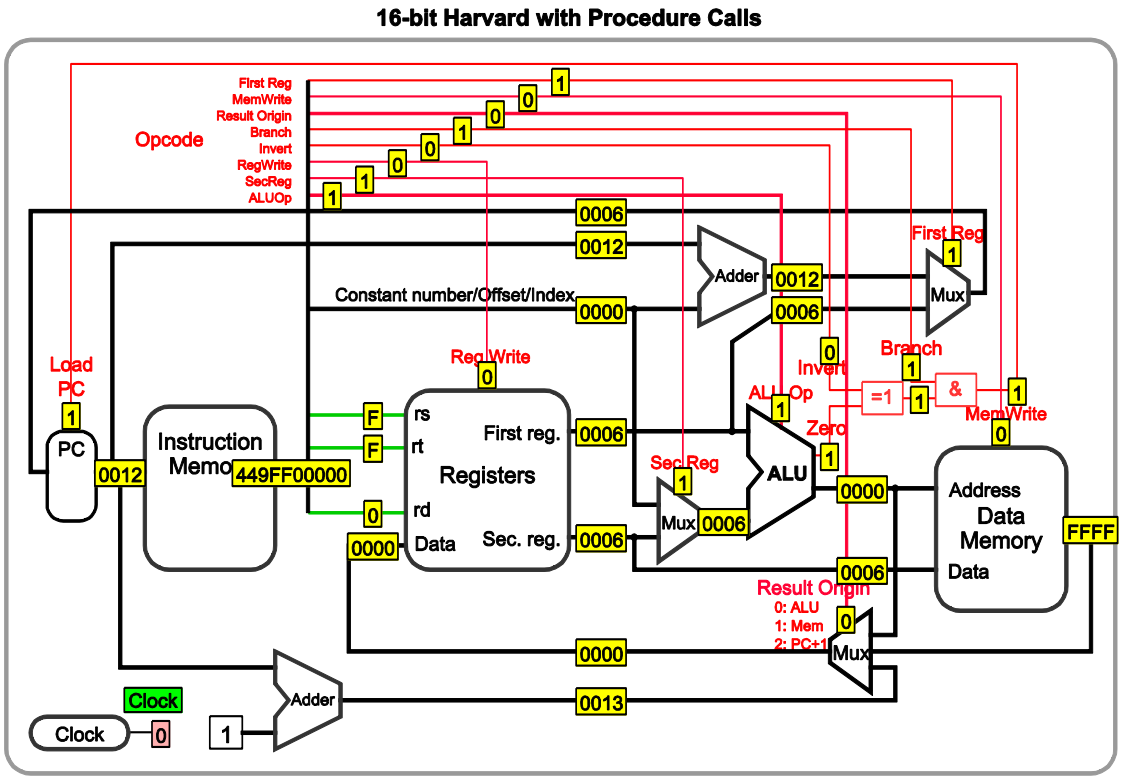
3. De Zero vlag van de ALU wordt gebruikt om condities te testen. We beschouwen de condities Equals (x == y) en Zero (x == 0). Is het mogelijk een conditionele versie van de volgende instructies te maken? Zo ja: welke van de bovenstaande condities kunnen worden getest? Zo nee, waarom niet?
   - JSR
   - RETURN

4. In figuur 10.3 is de Jumper Machine weergegeven na het uitvoeren van een instructie?
   a. Welke instructie wordt uitgevoerd? Geef de syntax ervan weer.
   b. Welke datatransfer(s) vindt/vinden er plaats? Licht je antwoord toe.
   c. Waar dient de Adder die linksonder is weergegeven voor?



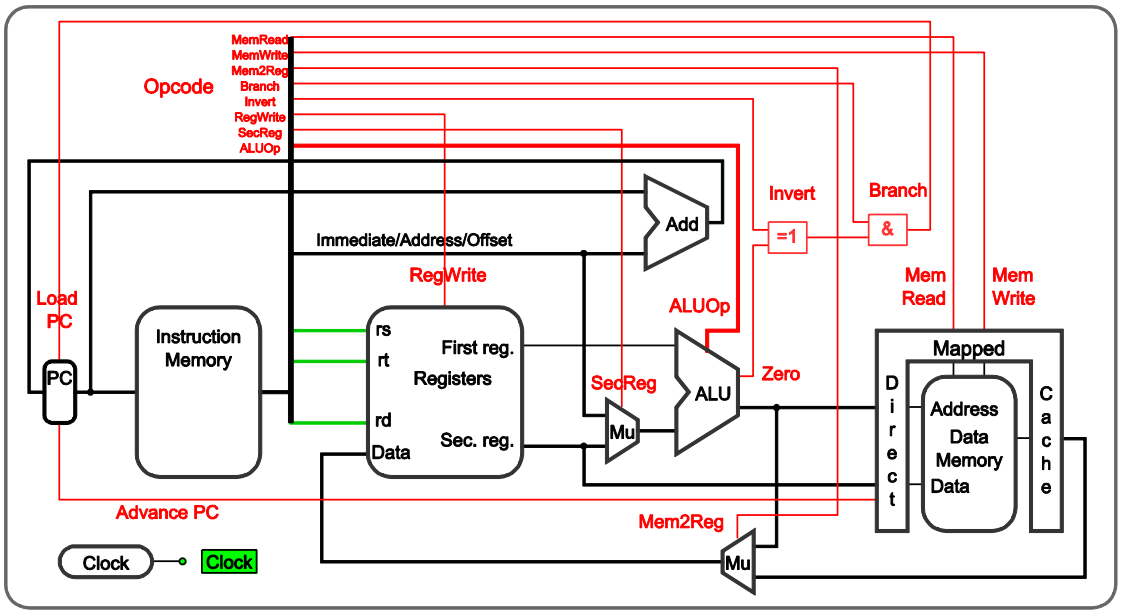
Figuur 10.3: Welke instructie wordt hier uitgevoerd?

5. In figuur 10.4 is de Jumper Machine weergegeven na het uitvoeren van een instructie?
    a. Welke instructie wordt uitgevoerd? Geef de syntax ervan weer.
    b. Welke datatransfer(s) vindt/vinden er plaats? Licht je antwoord toe.
    c. Wat is de opcode van deze instructie in hexadecimale code (zie ook tabel 10.5)?



Figuur 10.4: Welke instructie wordt uitgevoerd?

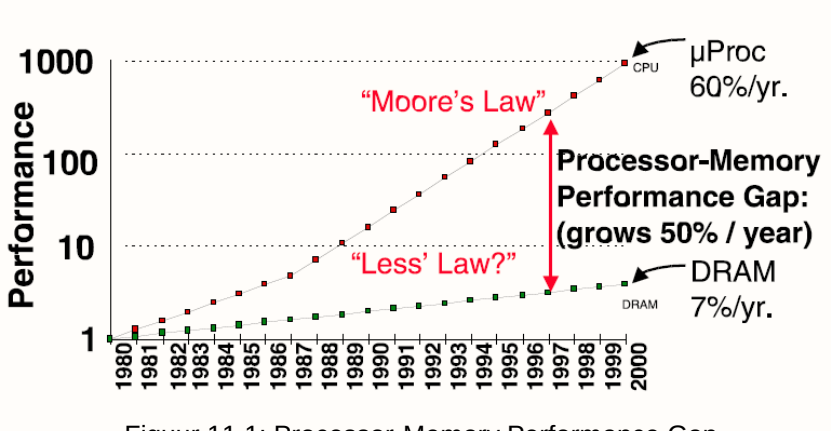
16 bit Harvard Architecture with direct mapped write-through cache



Hoofdtuk 11: Caches in de Memory Hierarchy

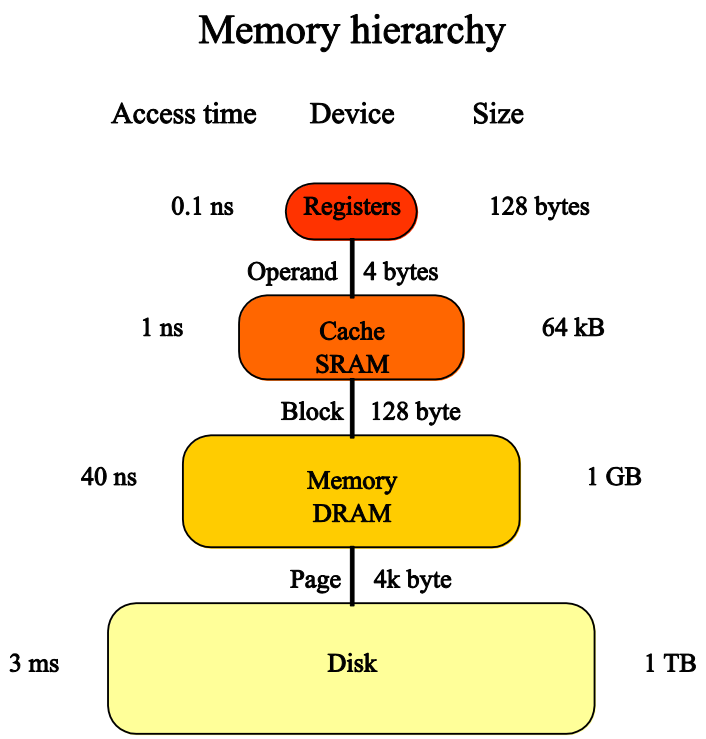
11.1 Inleiding en leerdoelen

De wet van Moore vertelt ons dat de prestaties van een processor (CPU) met 60% per jaar toenemen. Hoe zit dat met het werkgeheugen van een computer? Het werkgeheugen van een computer bestaat uit Dynamic RAM (DRAM). Voor het DRAM gelden heel andere waarden. Weliswaar heb je iedere anderhalf jaar twee keer zoveel geheugen voor dezelfde prijs maar de tijd benodigd om deze geheugencellen te lezen of schrijven is momenteel ongeveer 40 ns en wordt per jaar slechts 7% korter. De CPU cycle time van een processor nam de laatste 35 jaar af van 1000 ns (1 MHz) naar 0,5 tot 0,25 ns (2 tot 4 GHz). Het gevolg is dat er 80 tot 160 cycles nodig zijn om data van het geheugen naar de processor te transporteren (zie figuur 11.1).



Figuur 11.1: Processor-Memory Performance Gap

Volgens Amdahl's law heeft het verder verbeteren van de CPU geen zin zolang dit probleem niet is opgelost. Daarom werden zogenaamde "caches" geïntroduceerd om de "speed gap" tussen processor en memory te overbruggen. Deze caches, bestaande uit Static RAM-cellen (SRAM), zijn niet meer weg te denken in de huidige high-performance processors. Het proces van het gebruik van caches staat bekend als caching.



Figuur 11.2: Geheugenhiërarchie

In figuur 11.2 is een voorbeeld van een geheugenhiërarchie van een computersysteem weergegeven. De op één na onderste laag is het werkgeheugen dat uit DRAM bestaat. Tussen dit geheugen en de registers van de processor moet zo efficiënt mogelijk uitwisseling van gegevens kunnen plaatsvinden. Hiertoe is een cache bestaande uit 64 kB SRAM-cellen tussen beide blokken geplaatst. Het lezen of schrijven van DRAM kost 40 ns en van SRAM 1ns. Uitwisseling van data tussen Memory en Cache gaat in blokken met een grootte van 128 byte. Het kost 1 ns om 4 bytes (1 word) uit de cache naar één van de registers te transporteren. Daarnaast spelen ook de kosten een rol. Snelle klok getriggerde geheugencellen die in registers voorkomen bevatten vele transistoren. Een SRAM-cel heeft 6 transistoren en een DRAM-cel heeft er slechts 1 (zie de module Digitale Techniek).
Waaruit bestaat nu de winst die een cache oplevert? Stel dat er een lijst met gegevens moet worden bewerkt die in het DRAM aanwezig is. Het eerste gegeven van deze lijst wordt tegelijk met de gegevens die in hetzelfde geheugenblok staan uit het DRAM in de cache geladen. Als het tweede, derde en vierde gegeven van de lijst moeten worden bewerkt zijn die dus al in de cache aanwezig en kunnen relatief snel naar een register worden getransporteerd. Dit levert de beoogde tijdwinst op. Er is zowel een cache voor data als één voor de programma code (Instruction cache). Deze laatste wordt in dit hoofdstuk niet behandeld.

Aan het einde van dit hoofdstuk weet je:

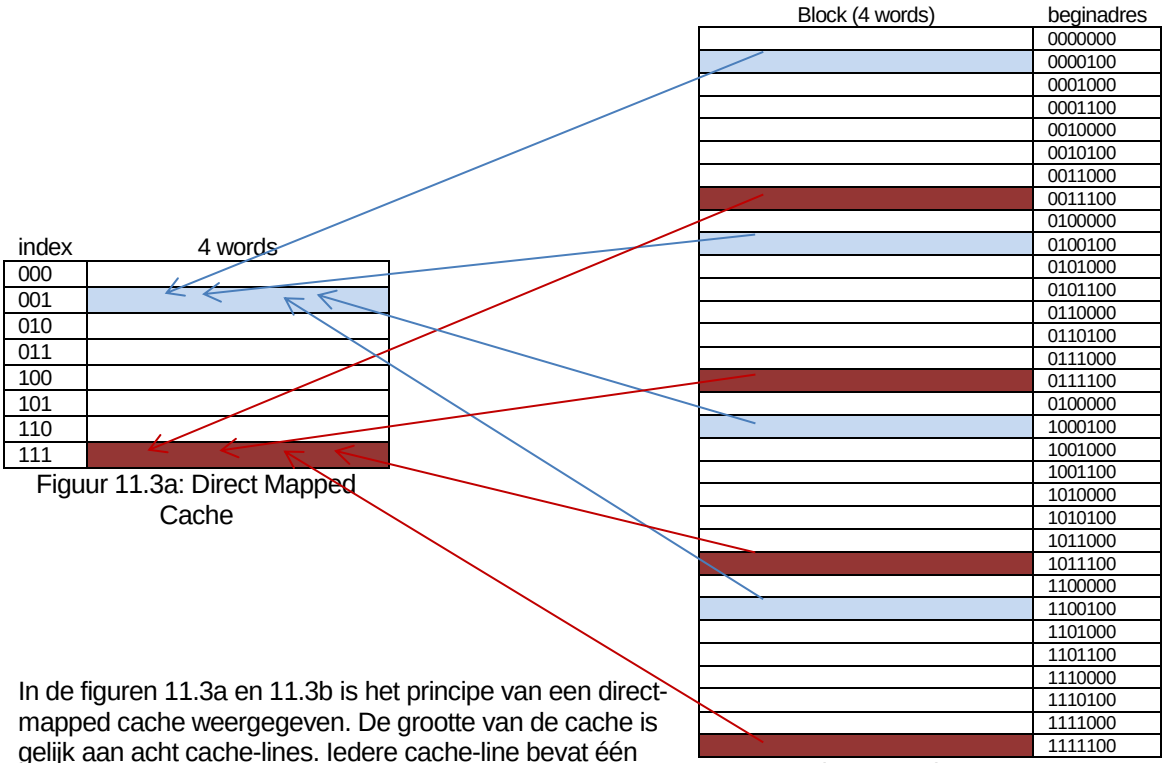
- hoe een direct-mapped cache werkt;
- hoe een set-associative cache werkt;
- wat een tag is;
- wat een valid-bit;
- wat de miss rate is;
- wat een miss penalty is;
- het verschil tussen een write-through cache en een write-back cache.

Aan het einde van het hoofdstuk kun je:

- de miss rates en CPU time bepalen van dezelfde programma's geëxecuteerd op machines met verschillende soorten caches.

11.2 Direct-mapped cache en 2-way set associative cache

Direct-mapped cache



Figuur 11.3a: Direct Mapped Cache

Figuur 11.3b: Memory

In de figuren 11.3a en 11.3b is het principe van een direct-mapped cache weergegeven. De grootte van de cache is gelijk aan acht cache-lines. Iedere cache-line bevat één blok bestaande uit 4 words. Ieder word is 16 bits. Het data-geheugen in dit voorbeeld heeft een grootte van 32 blokken. Het eerste blok, blok 0, bevat de geheugenadressen 0000000, 0000001, 0000010 en 0000011. Het volgende blok begint op adres 0000100. De inhoud van de blokken die beginnen met geheugenadressen 0000100, 0100100, 1000100 en 1100100 worden alle vier afgebeeld ("mapped") op locatie (index) 001 van de cache. De gegevens op de geheugenadressen 0011100, 0111100, 1011100 en 1111100 komen in cache-locatie met index 111. De index bestaat uit bits adres bits a_4, a_3 en a_2 van het geheugen.

Tag, valid-bit en block-offset

Tag
Hoe weten we nu welke van de vier mogelijke blokken uit het DRAM op bijv. index 001 van de cache is terechtgekomen? Daarvoor is een *tag-veld* in de cache aanwezig. In het bovenstaande geval bestaat dit uit de twee most significant bits van het geheugenadres. Stel dat geheugenblok 0100100 op index 001 van de cache terecht komt. De bijbehorende tag is dan 01.

Valid-bit
Als een computer wordt opgestart neemt iedere bit uit de cache een (willekeurige) waarde aan. Ook kan na vele instructies een cache locatie nog ongebruikt zijn. Daarom is iedere cache-line voorzien van een *valid-bit* die aangeeft dat de informatie op de betreffende plek geldig is.

Block offset
Eén blok bevat 4 woorden. Om deze te kunnen onderscheiden is de blok offset ingevoerd. Dit zijn de twee least significant bits van het geheugenadres.

Voorbeeld
In tabel 11.1 is een direct-mapped cache weergegeven bestaand uit 8 *cache-lines.* Iedere cache-line bestaat uit vier Words, een Tag en een Valid-bit. Het aantal bits van het tag-field hangt af van de grootte van het DRAM-geheugen. Stel dat het geheugen uit 4096 (2^{12}) words bestaat. Een geheugenadres bevat dus 12 bits aangeduid met a_{11} .. a_0. Stel dat we de inhoud van geheugenadres: 1001 0010 0100 naar de cache transporteren. Niet alleen dit adres maar ook de

inhouden van de adressen die in ditzelfde block zitten; 1001 0010 01**01**, 1001 0010 01**10** en 1001 0010 01**11** worden in de cache geladen. Het geheel komt terecht in de cache-line met index 001. Dit zijn adres-bits $a_4a_3a_2$. Het tag-field bestaat uit de overgebleven 7 adres-bits a_{11} … a_5 (1001001).
De cache kan 4 * 8 = 32 (2^5) words bevatten. Iedere cache-line bevat 1 + 7 + 4 * 16 = 72 bits. Het totale aantal bits van deze cache is 8 * 72 = 576 bits.

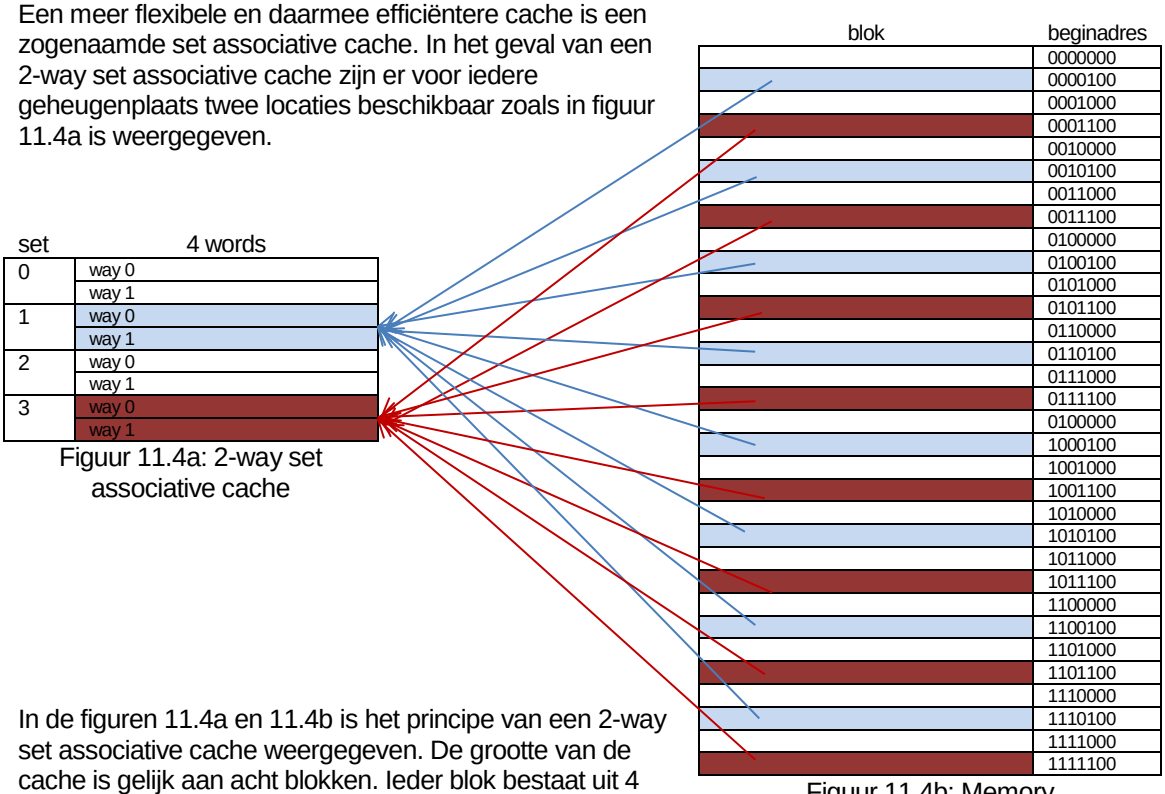
Index	Valid	Tag	Word 3	Word 2	Word 1	Word 0
Nr of bits	1	7	16	16	16	16
000						
001	1	1001001	….	….	….	….
010						
011						
100						
101						
110						
111						

Cache-line

Tabel 11.1: Direct mapped cache

Two-way set associative cache

Een meer flexibele en daarmee efficiëntere cache is een zogenaamde set associative cache. In het geval van een 2-way set associative cache zijn er voor iedere geheugenplaats twee locaties beschikbaar zoals in figuur 11.4a is weergegeven.



Figuur 11.4a: 2-way set associative cache

Figuur 11.4b: Memory

In de figuren 11.4a en 11.4b is het principe van een 2-way set associative cache weergegeven. De grootte van de cache is gelijk aan acht blokken. Ieder blok bestaat uit 4 geheugenplaatsen. Het data-geheugen in dit voorbeeld heeft een grootte van 32 * 4 geheugenplaatsen. De inhoud van de blokken die beginnen met geheugenadressen 0000100, 0010100, 0100100, 0110100, 1000100, 1010100, 1100100 en 1110100 komen terecht in set 1 van de cache. Is locatie way 0 bezet dan komen ze terecht in way 1. De adressen, 0001100, 0011100, 0101100, 0111100, 1001100, 1011100, 1101100 en 1111100 komen in set 3.

Cache hit, cache miss en miss penalty

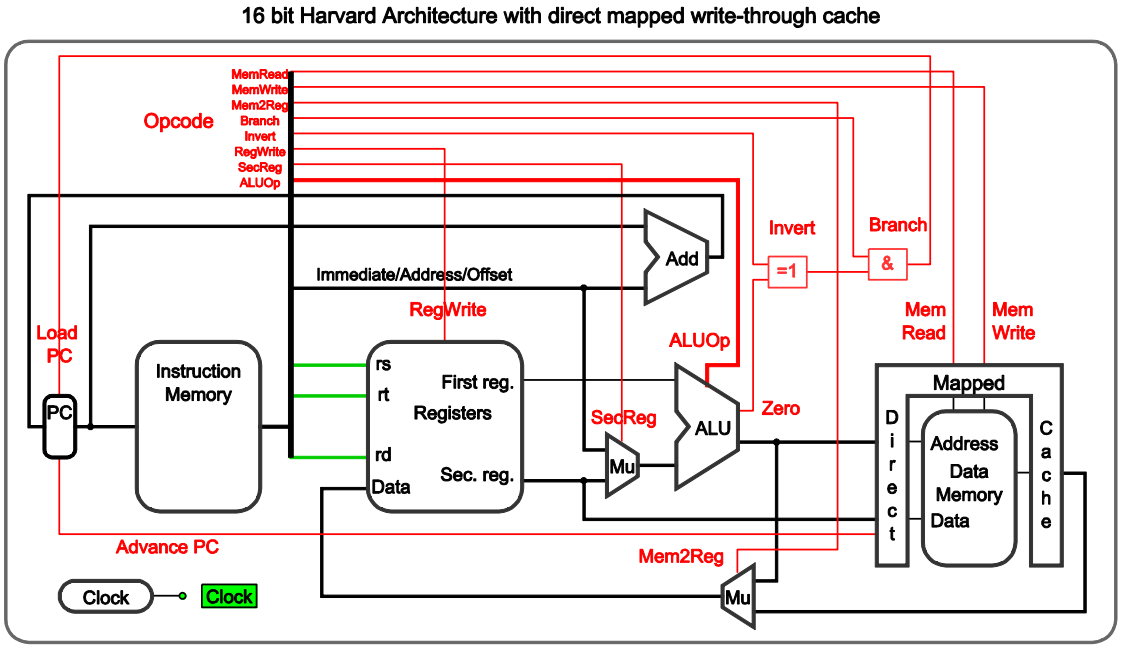
Wordt om een bepaald gegeven gevraagd en is een kopie hiervan wel/niet aanwezig in de cache dan is er sprake van een cache hit respectievelijk cache miss. Bij een cache miss wordt de extra benodigde tijd om dit gegeven uit het DRAM te halen de miss penalty genoemd.

Voorbeeld:
Stel dat een programma zowel de inhoud van de geheugenplaats 0000101 als van 0100101 steeds weer nodig heeft. Bij een direct-mapped cache komt eerst 0000101 op de cache-line met index 001 terecht. Deze wordt overschreven zodra adres 0100101 op dezelfde locatie terecht komt en 0000101 is dus niet meer beschikbaar en moet dus weer uit het DRAM worden gehaald wat steeds een miss penalty oplevert. In het geval van een 2-way set associative cache komen beide in set 1 op respectievelijk way 0 en way 1 terecht en blijven daar voor gebruik beschikbaar.

Least recently used (LRU)
Stel dat in het bovenstaande voorbeeld ook de data van adres 1100100 nodig is. Deze komt ook in set 1 terecht en dus zal één van de bestaande blokken moeten worden overschreven, maar welke? De meest toegepaste strategie is om in dat geval degene te overschrijven die het langste niet gebruikt is.

Write-through en write-back
Hierboven is het ophalen van gegevens uit het werkgeheugen naar de cache beschreven. Voor de omgekeerde richting, het schrijven van gegevens van de cache naar het datageheugen zijn er twee strategieën. De eenvoudigste manier is om de data in de cache en het data memory altijd consistent te houden. D.w.z. dat zodra er een nieuwe waarde van een register naar de cache wordt geschreven dan wordt deze waarde ook naar de desbetreffende DRAM-geheugenplek geschreven. Dit wordt *write-through* genoemd. Een alternatieve methode is om een nieuwe waarde alleen naar de cache te schrijven en dus nog niet naar het geheugen. Schrijven naar het geheugen vindt plaats als het betreffende block wordt herplaatst. Dit wordt *write-back* genoemd. Om de inconsistentie tussen cache en geheugen in dat geval weer te geven wordt iedere cache-line uitgebreid met een zgn. *dirty-bit*.

11.3 Geïmplementeerde caches in SIM-PL



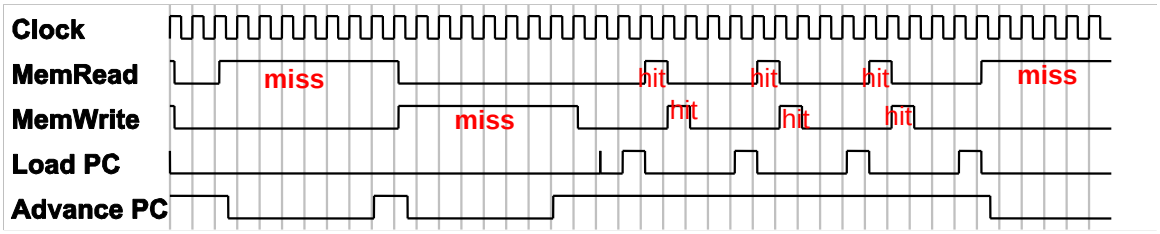
Figuur 11.5: 16-bit processor met Direct Mapped Cache

In figuur 11.5 is de 16 bit Harvard processor uit hoofdstuk 8 uitgebreid met een direct-mapped cache tussen het Data Memory en de processor. De cache past de write-through strategie toe. De direct-mapped cache bevat 16 blokken van elk 4 woorden. Een woord is 16 bits. Het datageheugen heeft de volgende eigenschappen:

1. Het aantal klokpulsen nodig voor een lees- of schrijfactie van het datageheugen is 5½. Er kan 1 lees- of schrijfoperatie per cycle worden aangeboden. Het aantal cycles dat cache en datageheugen samen gebruiken op een cache miss is 8. (Een cache hit kost 1 cycle.)
2. Er is een 4-woord leespoort aangebracht. Als adres 4i+j wordt opgevraagd ($0 \le j \le 3$), dan worden woorden 4i + 0, 4i + 1, 4i + 2, 4i + 3 tegelijk door het datageheugen geretourneerd als één 64-bit blok.

3. De schrijfpoort is nog steeds één enkel woord breed. De schrijfpoort is gebufferd. Na een schrijfoperatie wordt het 64-bit blok dat het geschreven woord bevat beschikbaar op de leespoort als in 2.

Als er een cache miss optreedt worden de benodigde woorden uit het datageheugen gelezen. Dit duurt een aantal (8) cycles. Het controlesignaal AdvancePC wordt door de cache laag gehouden zolang op het datageheugen wordt gewacht. Dit heeft als effect dat de Program Counter niet wordt opgehoogd. Tijdens het wachten op het geheugen verzet de processor geen nuttig werk.
Figuur 11.6 laat van een aantal signalen het tijdvolgordediagram zien tijdens het executeren van een klein programma. Vijf keer wordt de instructie LW aangeroepen om gegevens uit het geheugen op te halen. Bij iedere LW wordt het MemRead-signaal actief. Bij de eerste LW treedt er een cache miss op. Het MemRead-signaal is 8 klokpulsen hoog om (een blok) data van het DataMem naar één der registers te transporteren. Dit wordt een cache miss genoemd.



Figuur 11.6

De drie volgende keren wordt data gebruikt die al in de cache aanwezig is. Dus er zijn drie "hits" die ieder één klokpuls duren en die weer gevolgd wordt door een nieuwe miss. Bij de instructie SW, waarbij gegevens naar een ander stuk van het geheugen worden geschreven zie je ongeveer eenzelfde patroon. Het MemRead is 8 klokpulsen 'hoog' bij een cache miss en 1 klokpuls hoog bij een hit. De eerste keer veroorzaakt SW een miss en bij de drie opvolgende SW-instructies een hit.
Totaal zijn er, voor de totaal negen instructies, 6 hits en 3 misses. De zogenaamde *hit rate* is hier 6/9. De miss penalty is de tijd nodig om een blok in de cache te laden. Dus hier de tijd die behoort bij 7 klokpulsen.

11.4 Theorievragen en opdrachten

Het datageheugen van de Harvard processor weergegeven door figuur 11.5 bevat 2^9 = 512 woorden van elk 16 bits. De direct-mapped cache bevat 16 blokken van 4 woorden.
Vraag1a: Geef van elk van de in tabel 11.2 getoonde velden: Tag[15)], Index en Block offset aan hoeveel bits ze bevatten.

Tag	Index	Block offset

Tabel 11.2: Address

V	Tag	Data 3	Data 2	Data 1	Data 0

Tabel 11.3: Cache line

Opdracht 1b: Doe hetzelfde voor tabel 11.3.

Opdracht 1c: Bits in a Cache
Bereken het totale aantal bits dat gebruikt is voor de implementatie van deze cache-component.
Antwoord …..

Het geheugen wordt uitgebreid tot 2^{16} = 65536 woorden van 16 bits.
Vraag 1d: Zelfde vragen (zie 1a, 1b en 1c) als voor de cache die bij het geheugen van 512 woorden hoort.

Tag	Index	Block offset

Tabel 11.4: Address

[15)] Om het aantal bits van de Tag te bepalen ga je na hoeveel keer de cache in het DRAM past. Stel dat de cache 1024 keer in het geheugen past. Je hebt dan een 10 bits code nodig (2^{10} = 1024) om aan te geven welk deel van het geheugen in de cache zit, dus het Tag-veld is 10 bits.

Opdracht 2

Bestudeer het onderstaande programma. Vul in tabel 11.5 in waar volgens jou de getallen 1 t/m 5; 6 t/m A en {7, 9, B, D, F} terechtkomen. Vul ook de bijbehorende velden V en Tag in. *Het is niet de bedoeling dit programma te executeren en daarvan de resultaten in te vullen. Bij de volgende paragraaf worden je "vermoedens" gecontroleerd.*[16)]

Hoeveel cache misses zijn er opgetreden tijdens een memory read-instructie?
Hoeveel cache misses zijn er opgetreden tijdens een memory write-instructie?

Index	V	Tag	Word 3	Word 2	Word 1	Word 0
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

Tabel 11.5: Cache van de Harvard machine

```
@include "16bitCached.wasm"

.data MyData : DATAMEM
0d10:   WORD base       0d1
        WORD            0d2
        WORD            0d3
        WORD            0d4
        WORD            0d5
0d64:   WORD base2      0d6
        WORD            0d7
        WORD            0d8
        WORD            0d9
        WORD            0d10
0d168:  WORD base3

.code MyCode  : HARVARD, MyData
        LOADI $5, 5     # 5 getallen
        LOADI $6, 0     # Start at 0

loop:   LW $1, base, $6
        LW $2, base2, $6
        ADD $3, $1, $2
        SW $3, base3, $6
        ADDI $6, $6, 1      # next adres
        BEQ $6, $5, end  # 5 getallen?
        BRA loop         # Branch Always
end:    HALT
```

Vraag 3: Waar of onwaar? Beargumenteer uw antwoord.

1. Bij een fully associative cache kan elk geheugenblok in iedere cache-line terecht komen. Een Direct-mapped en 2-way set associative caches zijn extreme gevallen van fully associative caches?
2. Beschouw het Tag- en Index-veld van een adres. Bij een direct-mapped cache is het Tag veld groter, en het Index veld kleiner dan bij een set-associative cache?
3. Direct-mapped, set-associative en fully-associative caches met dezelfde capaciteit vereisen precies evenveel bits opslag?

[16)] De cache bevat 64 getallen. Waar een getal terecht komt is het Memory Address modulo 64.

11.5 Practicumopdrachten caching

Index	V + Tag	Data 3	Data 2	Data 1	Data 0
0	9	0009	0008	0007	0006
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

Tabel 11.6: Cache van de Harvard machine

Opgave 1a: Open de worksheet 16bitHarvardMachineDirectMappedCache uit de folder “H11Caches”. Run het programma “Opgave1.wasm” De cache is geïnitialiseerd met nullen en het datageheugen met enen zodat zichtbaar is als een block in de cache is geladen. Vul tabel 11.6 in en controleer of de antwoorden van de theorieopgave 2 correct zijn. Het block met index 0 is al ingevuld. Het meest linkse cijfer (9) bestaat uit het Valid bit (8) + de Tag (1). De Tag is 1 omdat het kopieën van gegevens uit de geheugen plaatsen 64 tot en met 127 zijn.

Beredeneer hoeveel cache misses er zijn opgetreden tijdens een memory read-instructie?
Beredeneer hoeveel cache misses er zijn opgetreden tijdens een memory write-instructie?

Index	V+Tag	Data 3	Data 2	Data 1	Data 0

Tabel 11.7: Opgave 1b

Opgave 1b: Beredeneer eerst wat de antwoorden zijn en executeer daarna je programma om je antwoorden te verifiëren! De plek waar de tweede dataset {6, 7, 8, 9, A} in het geheugen staat begint nu niet vanaf adres 0d64 maar vanaf adres 0d138.
Welke blokken van de cache bevatten, nadat het programma is geëxecuteerd, (zinvolle) data? Vul in tabel 11.7 de waarden die deze blokken hebben gekregen in. Vul ook het bijbehorende veld V + Tag in.

Beredeneer hoeveel cache misses zijn er nu opgetreden tijdens een memory read-instructie?
Beredeneer hoeveel cache misses zijn er nu opgetreden tijdens een memory write-instructie?
Pas het programma aan door de regel:*0d64: WORD base2*
te veranderen in: *0d138: WORD base2.*

Index	V+Tag	Data 3	Data 2	Data 1	Data 0

Tabel 11.8: Opgave 1c

Opdracht 1c: Beredeneer eerst wat de antwoorden zijn en executeer daarna je programma om je antwoorden te verifiëren!
In het eerder weergegeven programma veranderen we zowel de plek waar de tweede dataset {6, 7, 8, 9, A} in het geheugen staat als de plek waar de getallen naartoe worden geschreven.

De regel :*0d64: WORD base2* wordt veranderd in: *0d138: WORD base2 en*
de regel :*0d168: WORD base3* wordt veranderd in: *0d200: WORD base3.*

Welke blokken van de cache bevatten, nadat het programma is geëxecuteerd, (zinvolle) data? Vul in tabel 11.8 de waarden die deze blokken hebben gekregen in. Vul ook het bijbehorende veld V + Tag in.

Hoeveel cache misses zijn er nu opgetreden tijdens een memory read-instructie?
Hoeveel cache misses zijn er nu opgetreden tijdens een memory write-instructie?

Opgave 2: Run het programma “Opgave2.wasm” uit de folder “H11Caches”. De cache is geïnitialiseerd met nullen en het datageheugen met enen zodat zichtbaar is als een block in de cache is geladen.

- Vul tabel 11.9 in.

index	V	Tag	Data 3	Data 2	Data 1	Data 0
0						
1						
4						
5						
6						
15						

Tabel 11.9

Bekijk het “Time Sequence Diagram”.

- Wat is de “miss rate” in dit programma?
- Wat is de “miss penalty” voor deze machine?
- Meet met de muis de cycle length van de clock in het “Time Sequence Diagram” en controleer de gemeten waarde via de menuoptie Simulate → Cycle Length. 1 tijdstip komt overeen met 0,01 nanosec.
- Wat is de CPU time van dit programma?
- Wat is de CPI voor dit programma?

Opgave 3: Run het programma “Opgave3.wasm” uit de folder “16bitCached”.
Bekijk het “Time Sequence Diagram”.

- Wat is de “miss rate” in dit geval?
- Wat is de CPU time in dit geval?

Verklaar waarom het executeren van programma “Opgave3.wasm” 35 ns langer duurt dan het executeren van programma “Opgave2.wasm”.

Opgave 4: Er is ook een machine met een two-way set associatieve cache geïmplementeerd. Start de worksheet: 16 bit Harvard MachineTwoWaySetAssociativeCache uit de folder “H11 Caches”. Run het programma “Opgave4.wasm”.
Bekijk het “Time Sequence Diagram”.

- Wat is nu de “miss rate”?
- Wat is nu de CPU time?
- Wat is de performance ratio tussen deze machine en die uit de vorige opgave.

Opgave 5a:
Gebruik de direct mapped cache en beschouw Listing 1 en Listing 2. In beide wordt een tweedimensionaal array A doorlopen maar dit gebeurt in een verschillende volgorde. Ga er van uit dat het array A als volgt in het datageheugen wordt bewaard:
A[y][x] staat in woord 8y + x; $0 \le x, y < 8$.

1. Bekijk de tijdvolgordegrafiek in Figuur 11.7. Behoort deze bij Listing 1 of Listing 2? Beargumenteer je antwoord.
2. Implementeer beide programma’s in assembler. De gebruikte ALU kan vermenigvuldigen noch bit-shiften. Bereken het adres van A[y][x] daarom incrementeel, d.w.z. gebruik een (extra) register voor het adres, en pas de inhoud van dit register iedere binnenste en buitenste lus aan. De hieronder weergegeven code in de “binnenloop” kan en moet worden vermeden!

```
ADD $t, $y, $y  # 2*y
ADD $t, $t, $t  # 2*(2*y) = 4*y
ADD $t, $t, $t  # 2*(4*y) = 8*y
ADD $t, $t, $x  # 8*y + x
```

Om beide programma’s zinvolle vergelijkende metingen te kunnen uitvoeren moet het aantal statements in zowel de “binnenloop” als de “buitenloop” van beide programma’s

gelijk zijn. Gebruik hiervoor zonodig een instructie die niets doet (bijv. ADDI $3, $3, 0).

3. Meet voor beide programma's het aantal klokpulsen dat nodig is voor de executie. Gebruik hiervoor het SIM-PL Time Sequence Diagram en controleer het antwoord. Geef de waarden in tabel 11.10 weer. Meet ook de hit rate van beide programma's.

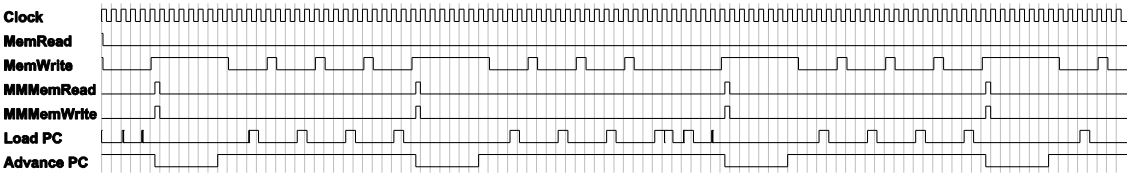
Opmerking: De cache kan 64 woorden bevatten en het array A[8][8] past hier precies in.

```
int A[8] [8];
for( int y = 0 to 7 ){
      for( int x = 0 to 7 ){
            A [y] [x] = 8*y+x;
      }
}
```

Listing 1

```
int A[8] [8];
for( int x = 0 to 7 ){
      for( int y = 0 to 7 ){
            A [y] [x] = 8*y+x;
      }
}
```

Listing 2



Figuur 11.7: Onbekende tijdvolgordediagram

Opgave 5b: Breid het array uit naar 12 bij 12 elementen en meet nogmaals voor beide programma's het aantal cycles en de hit rate. Verklaar uw antwoorden.

	*8*8 Listing 1*	*8*8 Listing 2*	*12*12 Listing 1*	*12*12 Listing 2*
Aantal cycles Hit rate				

Tabel 11.10

11.6 Begrippenlijst

Direct mapped cache. Een cachestructuur waarbij iedere geheugenplaats wordt afgebeeld op precies één plaats in de cache.

Fully associative cache. Een cachestructuur waarbij iedere geheugenplaats kan worden afgebeeld op iedere plaats in de cache.

Miss rate. Het deel van de geheugenaanroepen waarbij de gegevens in een andere laag niet worden gevonden.

Miss penalty. De extra tijd nodig om een benodigd Word uit het geheugen te halen als het niet in de cache aanwezig is.

Set-associative cache. Een cachestructuur waarbij iedere geheugenplaats kan worden afgebeeld op meerdere plaatsen (tenminste twee) in de cache.

Tag. Het veld in een cache-line dat de vereiste adresinformatie bevat om na te gaan of het bijbehorende block overeen komt met het gevraagde word.

Write-back. Als een cache-locatie wordt geschreven wordt pas naar het geheugen geschreven als het hele blok wordt ge-update. In de cache-line wordt d.m.v. een "dirty bit" aangegeven als de waarde van beide locaties verschilt.

Write-through. Als een cache-locatie wordt geschreven wordt tevens naar het geheugen geschreven om consistentie tussen die twee plaatsen te waarborgen.

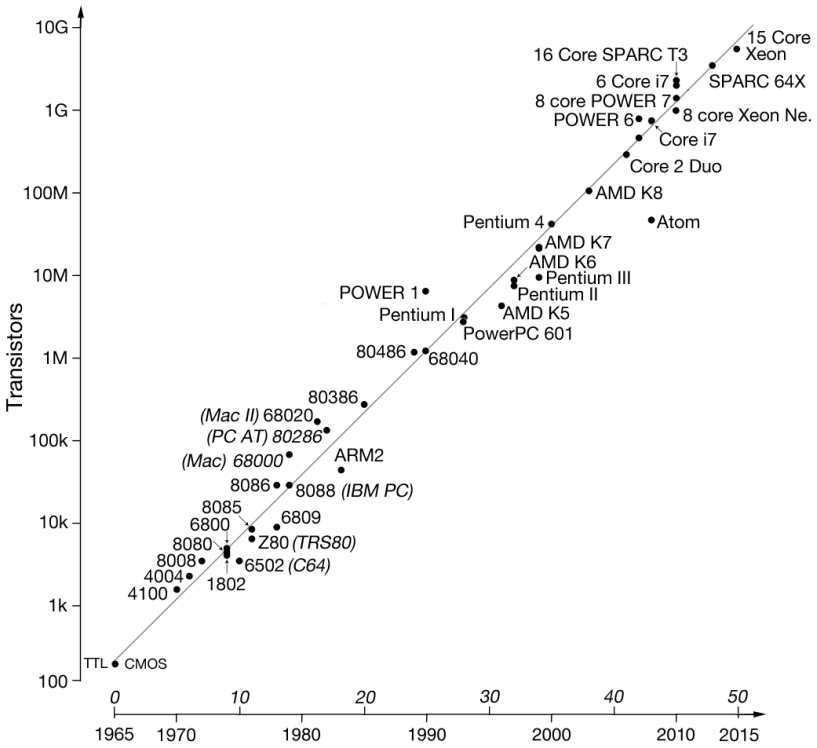
11.7 Tentamenvragen

1. Geef een afbeelding van de memory hiërarchie. Benoem elk opslag medium. Geef twee specifieke eigenschappen van elke laag in de memory hiërarchie weer.
2. Teken de memory hiërarchie en geef daarbij aan een indicatie voor de grootte en de “access time” van de componenten. Neem hiervoor de waarden van de figuur aan het begin van dit hoofdstuk.
3. Welke twee lokaliteiten zijn er. Leg uit waarom deze lokaliteiten noodzakelijk zijn voor het effectief kunnen implementeren van een geheugen hiërarchie.
4. Wat is spatiele - en wat is temporele lokaliteit? Gelden deze lokaliteiten voor zowel data als voor instructies in gelijke mate? Geef voorbeelden.
5. Wat is een cache-blok?
6. Een byte geadresseerde machine heet 2^{18} geheugenadressen voor DRAM. Er is een direct mapped write-through cache geïmplementeerd met 16 cache blokken. Ieder blok kan 16 bytes bevatten.
   Geef weer hoeveel bits het tag-veld, het index-veld en het block-offset-veld bevat. Is er in de cache-line een “dirty-bit” aanwezig? Verklaar je antwoord.
7. Een byte geadresseerde machine heet 2^{18} geheugenadressen voor DRAM. Er is een fully associative cache geïmplementeerd met 16 cache blokken. Ieder blok kan 16 bytes bevatten.
   Is er in dit geval zowel een tag-veld, een index-veld en een block-offset-veld aanwezig? Geef het aantal bits weer dat ieder veld bevat.
8. Leg uit wat onder de “mis penalty” wordt verstaan bij caches.
9. Is het zinvol de data en de instructies in verschillende caches onder te brengen? Verklaar je antwoord.
10. De data-stroom door een cache moet op een doordachte manier geschieden. Beschrijf twee "replacement strategies".
11. De data-stroom door een cache moet op een doordachte manier geschieden. Beschrijf twee twee "write strategies". Bij welke is er een zgn. dirty-bit aanwezig? Licht je antwoord toe.
12. Noem enkele methoden om de cache miss rate te reduceren en leg uit hoe ze werken.
13. Noem enkele methoden om de cache miss penalty te reduceren en leg uit hoe ze werken.
14. Er wordt een cache ontworpen voor een computer met 2^{32} bytes aan geheugen. De cache heeft 512 blokken. Ieder blok is 16 bytes.
    a. Bereken voor zowel een ’fully associative cache’ als een ’direct-mapped cache’ hoeveel bytes de twee verschillende cache soorten in totaal zullen innemen.
    b. Teken zowel een ’fully associative cache’ als een ’direct mapped cache’ en benoem de velden. Geef in de tekening aan hoeveel bits de verschillende velden beslaan.
15. Behandel de voor- en nadelen van de ’fully associative cache’ in vergelijking met de ’direct mapped cache’.
16. Beschrijf de verschillen tussen een ‘fully associative cache’, een ‘direct mapped cache’ en een ‘set-associatice cache’.
17. Behandel de begrippen ’write through’ en ’copy back’.

11.8 Moore's law bestaat al 51 jaar. Hoe lang nog?

De wet van Moore

De wet van Moore vertelt ons dat iedere twee jaar het aantal transistoren op een chip verdubbelt. In figuur 11.8 is dit weergegeven.



Figuur 11.8: Aantal transistoren per processor gedurende de laatste 50 jaar

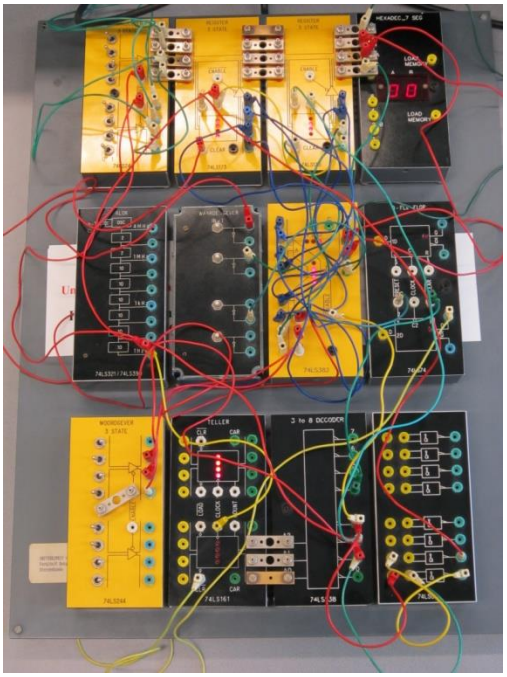
Het is een misvatting te denken dat alleen het aantal transistoren bepalend is voor de prestatietoename van de huidige computers. Het heeft bijvoorbeeld nauwelijks zin om de woordbreedte te vergroten naar 128 bits. Naast toename van het aantal transistoren hebben de volgende items bijgedragen aan een opmerkelijke performanceverbetering:

- Technologische verbeteringen: Met name het afnemende energieverbruik. Nog geen 15 jaar geleden dacht men dat toekomstige processoren gekoeld zouden moeten worden. Alleen tijdens het schakelen van een transistor wordt energie verbruikt. Hoe hoger de klokfrequentie van een processor des te meer er wordt geschakeld en des te heter wordt de processor chip. De maximale frequentie is 4 Giga Hertz. Deze neemt de laatste jaren niet meer toe. Van groot belang voor het energiegebruik is dat de voedingsspanning van processoren drastisch is verlaagd van 5 Volt naar 2 Volt en lager. Hier is nog winst te behalen.
- Architectuurverbeteringen:
    - Pipelined processoren;
    - Caches, tegenwoordig 3 lagen en allen geïntegreerd op de processorchip;
    - Meerdere processoren op één chip. Ook hier is nog veel winst te behalen.
- Software:
    - Betere compilers
    - Parrallel programming languages

Blijven deze ontwikkelingen zo doorgaan?
Intel chips zijn nu gebaseerd op 14 nanometer technologie. 14 nm is de afmeting van de gate van een transistor. Pas in 2017 wordt de volgende stap gemaakt naar 10 nm technologie. Dit is later dan verwacht. De oorzaak is de gigantische investeringen die daarvoor nodig zijn. De wet van Moore gaat dus voorlopig nog door, weliswaar in een wat trager tempo.

Hoofdstuk 12: Een busgeorganiseerde rekenmachine

12.1 Inleiding en practicummateriaal

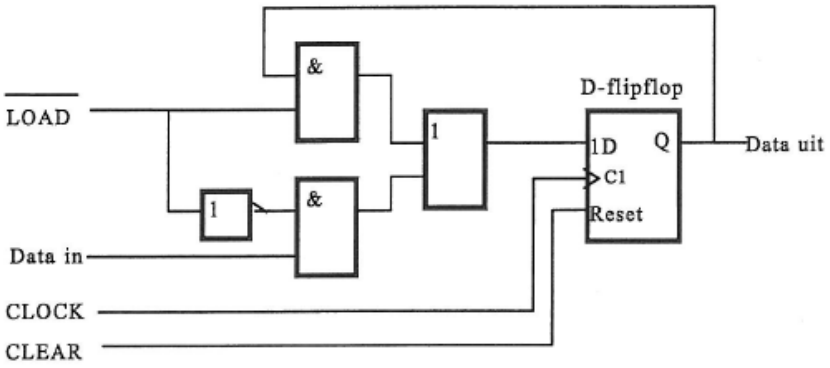


Bij het uitvoeren van een instructie door een computer worden gegevens (data) over de zogenaamde databus verzonden. Dit betekent dat informatie wordt uitgewisseld tussen verschillende registers waarbij meestal ook de ALU betrokken is. De databus bestaat uit een aantal lijnen waarop registers en de ALU zijn aangesloten. Verbindingen met de databus worden onderscheiden naar gelang het een ingang of een uitgang betreft. Daadwerkelijke verbinding met registeringangen komt tot stand door middel van een $\overline{\text{LOAD}}$-signaal. Het moment waarop de informatie van de databus werkelijk in een register wordt opgeslagen, vindt plaats op het moment dat de systeemklok van niveau wisselt, dus tijdens een flank.
Aan de zijde van de uitgangen van de registers en van de ALU is een voorziening getroffen om te voorkomen dat meerdere uitgangen tegelijkertijd via de databus met elkaar verbonden zijn. Daarom is elk systeem dat een uitgang heeft op de databus, voorzien van een three-state (of 3-state) buffer. De functie daarvan is die van schakelaar waarmee de verbinding met de databus kan worden tot stand gebracht dan wel verbroken. De sturing van de three-state buffer geschiedt door middel van een signaal op een aparte ingang aangeduid met $\overline{\text{ENABLE}}$. Om de datatransfers van een instructie op het juiste moment en in de goede volgorde te laten plaats vinden is een sequencer nodig.
In dit hoofdstuk wordt een “accumulatormachine” in elkaar gezet met behulp van zogenaamde PIDAC-modules. In deze modules zijn geïntegreerde circuits (ic’s) gemonteerd van de 7400 LSTTL serie. De schakelingen van de gebruikte ic’s zijn beschreven in hoofdstuk 6 van het boek Van 0 en 1 tot processor. De kleur van de stekerbussen van de modules heeft de volgende betekenis:

- Geel en wit: ingangen
- Groen: uitgang
- Rood: in- of uitgang
- Zwart: 0 volt

12.2 Data-opslag in een register

Registers worden gebruikt voor opslag van tussenresultaten van berekeningen. Registers bestaan uit D-flipflops voorzien van een poortnetwerk. In figuur 1 is het schema van een één bits register weergeven.



Figuur 1: Eén bit register

Als de $\overline{\text{LOAD}}$-ingang 1 is, staat op de uitgangen van de AND-poorten respectievelijk 'Data uit' en 0. De uitgang van de OR-poort heeft dus de waarde 'Data uit'. De flipflop zal op de eerstvolgende flank deze waarde overnemen. Dat is dus de onthoudfunctie. Als de $\overline{\text{LOAD}}$-ingang 0 is, staat op de uitgangen van de AND-poorten respectievelijk 0 en 'Data in'. De flipflop zal op de eerstvolgende flank dus de waarde 'Data in' overnemen. Dat is de 'laad'-functie. Bij moderne digitale schakelingen wordt gebruik gemaakt van slechts één zogenaamde systeemklok. Deze is direct verbonden met de klokingangen van alle flipflops van de registers.

Het vier bit register 74LS173

Op het practicum wordt het register type 74LS173 gebruikt. Dit register is samengesteld uit vier flankgetriggerde D-flipflops en een viervoudige "2-line to 1-line data-selector" (zie ook boek: Van 0 en 1 tot processor fig. 6.6). Hieronder is de waarheidstabel van dit register weergegeven.

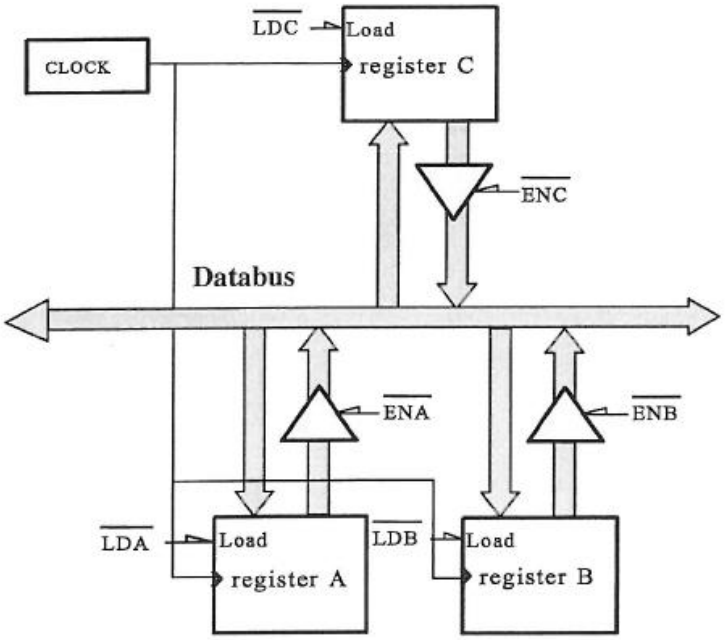
Uit de tabel blijkt dat het op nul zetten (CLEAR) plaats vindt als CLEAR 1 is, zonder dat er een klokflank nodig is. Het laden van een getal vindt plaats als $\overline{\text{LOAD}}$ is 0 en er een opgaande klokflank is. CLEAR is een asynchrone actie; laden is een synchrone actie

CLEAR	$\overline{\text{LOAD}}$	CLOCK	D_3	D_2	D_1	D_0	Q_3	Q_2	Q_1	Q_0
1	x	x	x	x	x	x	0	0	0	0
0	1	↑	x	x	x	x	q_3	q_2	q_1	q_0
0	0	↑	d_3	d_2	d_1	d_0	d_3	d_2	d_1	d_0

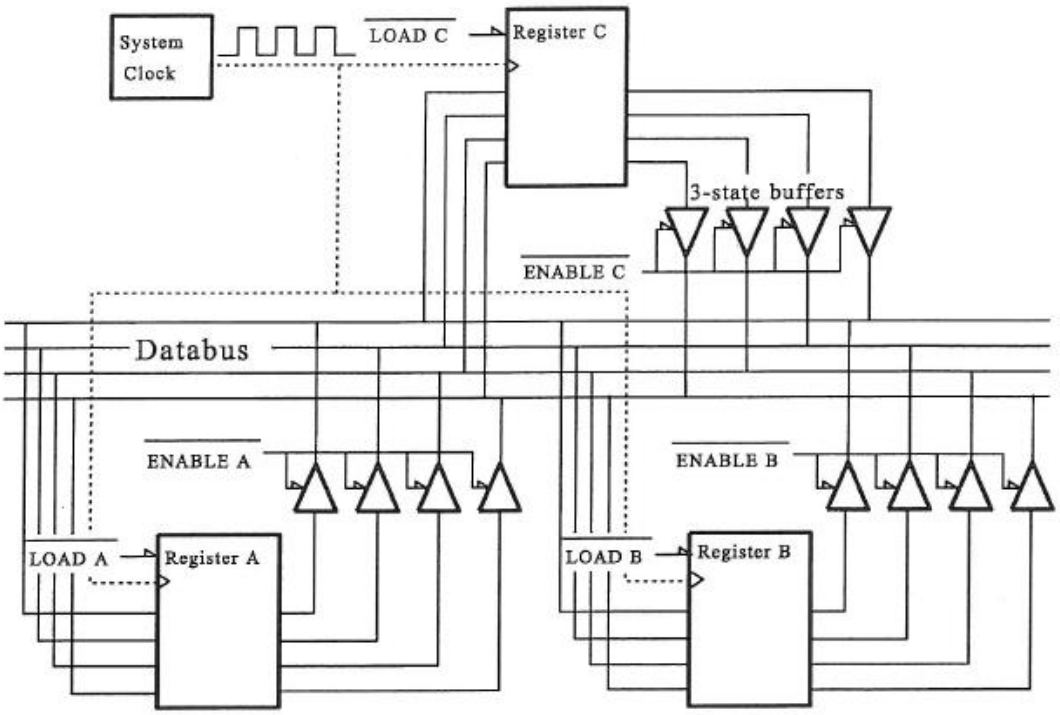
Tabel 1 waarheidstabel van het vier bit register 74LS173

12.3 Data-overdracht via een bus

In figuur 2 staat een schakeling weergegeven van drie registers, die via een databus met elkaar verbonden zijn. De bus waarlangs het datatransport plaats vindt, bestaat in de praktijk uit een aantal koperen printsporen. Bij veel recente processoren is dat aantal 64. Alle registers en andere computerbouwstenen die wat hun uitgang betreft, direct betrokken zijn bij data-uitwisseling, hebben voor elk spoor een three-state buffer. In de schakeling van figuur 3 is dit gedetailleerder weergegeven met een bus die bestaat uit vier sporen. Uit figuur 3 blijkt dat de buffers van één register door één enkele ingang, de $\overline{\text{ENABLE}}$ -ingang worden bestuurd.



Figuur 2: Databus met drie registers



Figuur 3: 4 bit databus met drie registers

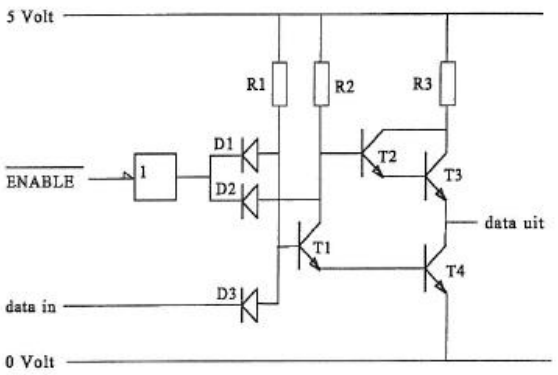
Als de $\overline{\text{ENABLE}}$ -ingang 1 is, is de uitgang zwevend. Praktisch gesproken betekent dit dat de

elektrische weerstand tussen het register en de databus bijna oneindig groot is waardoor er geen contact is. Wanneer de $\overline{\text{ENABLE}}$-ingang 0 wordt gemaakt is de buffer feitelijk een doorverbinding en is de waarde van de bufferuitgang gelijk aan die van de bufferingang.

Three-state buffer

Voor degenen die inzicht en ervaring hebben in het lezen van elektrische schakelingen, is in figuur 4 het elektronisch schema van de three-state buffer opgenomen. De drie mogelijkheden zijn als volgt:

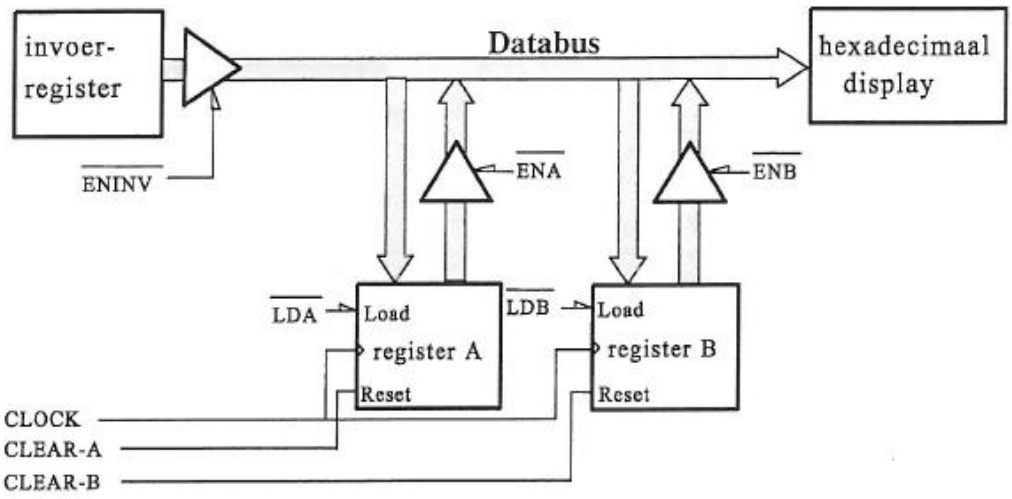
1. de uitgang kan 0 zijn. In dat geval geleidt transistor T4 en geleidt transistor T3 niet,
2. de uitgang kan 1 zijn. In dat geval geleidt transistor T4 niet en transistor T3 wel.
3. de uitgang is zwevend. Beide transistoren geleiden niet.



Figuur 4: Three-state buffer

Dataoverdracht tussen drie registers via een bus

In figuur 5 zijn opnieuw drie registers (register A, register B en een in voerregister) en een databus getekend. Verder is in de schakeling een hexadecimaal display opgenomen.

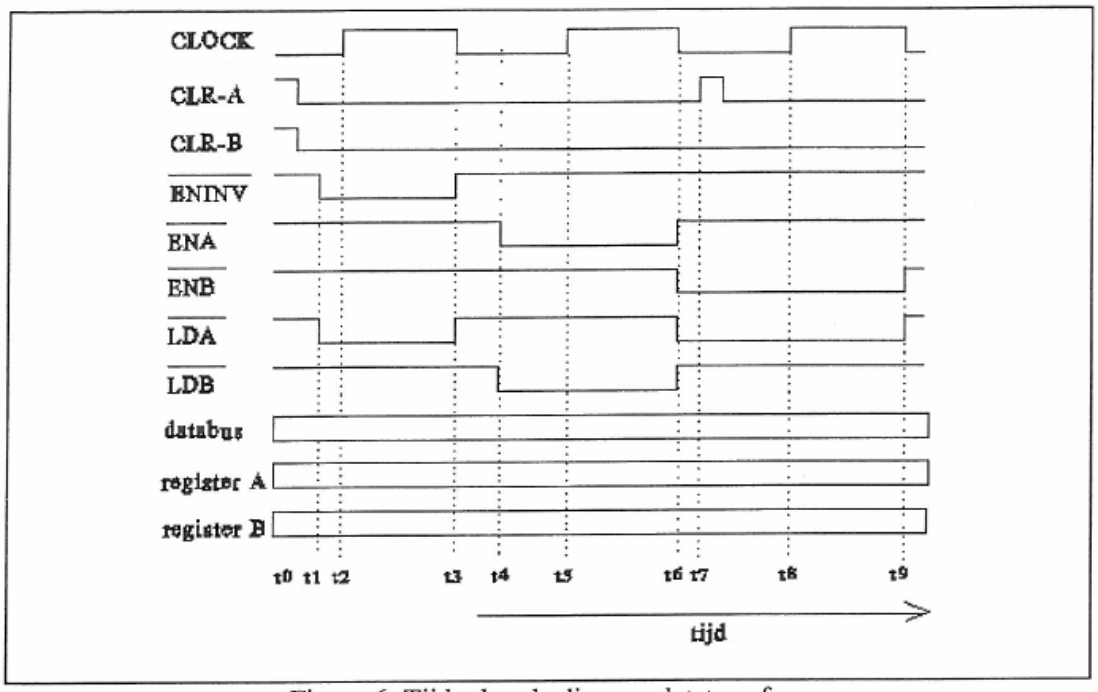


figuur 5: Data-transport via bus

Met deze schakeling kunnen de volgende datatransfers plaatsvinden:

- MOVE A, INV Kopieer de inhoud van het invoerregister naar het A-register,
- MOVE B, INV B ← INV,
- MOVE B, A B ← A,
- MOVE A, B A ← B.

In figuur 6 is weergegeven op welk moment besturingssignalen actief moeten zijn om een datatransfer uit te voeren. Op tijdstip t_0 zijn alle $\overline{\text{LOAD}}$- en $\overline{\text{ENABLE}}$-ingangen 1, zodat alle registeruitgangen zwevend zijn. Tevens zijn de CLEAR-ingangen van de beide registers 1 waardoor de register-flipflops gereset zijn. Op tijdstip t_1 wordt $\overline{\text{ENINV}}$ 0. De inhoud van het invoerregister verschijnt dan op de databus. De waarde ervan is op het display af te lezen. Tegelijkertijd (dus op t_1) wordt de $\overline{\text{LOAD}}$–ingang ($\overline{\text{LDA}}$) van register A 0. De eerstvolgende positieve klokflank verschijnt op tijdstip t_2. Op dit moment wordt register A geladen. Op t_3 wordt de bus weer vrijgegeven voor een volgende transfer doordat $\overline{\text{ENINV}}$ en $\overline{\text{LDA}}$ beide 1 worden.



Figuur 6: Tijdvolgorde diagram datatransfers

Opdracht 1: Datatransfers
Beschrijf hieronder wat er gebeurt op de tijdstippen t_4 t/m t_9 van figuur 6.

- t_4:
- t_5:
- t_6:
- t_7:
- t_8:
- t_9:

12.4 Practicum een busgeorganiseerde rekenmachine

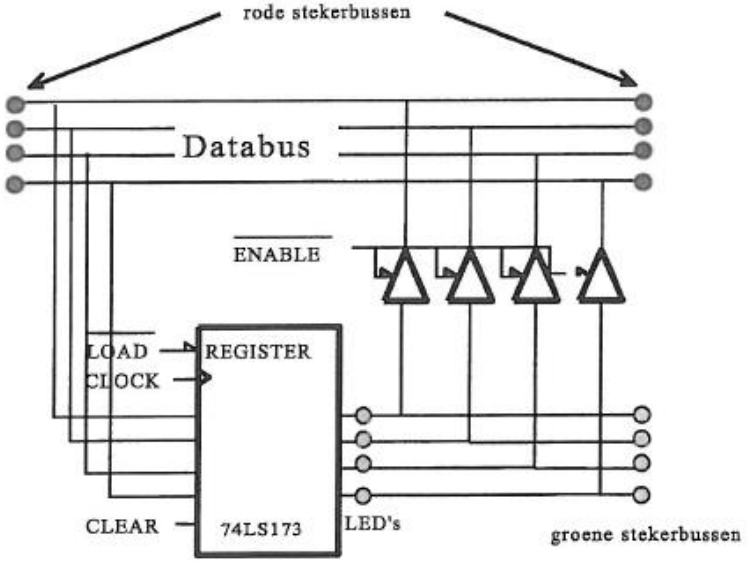
In de volgende experimenten wordt stapsgewijs een busgeorganiseerde rekenmachine gebouwd waarmee operaties op twee vier bit getallen kunnen worden uitgevoerd.

- *In experiment 1 wordt de werking van three-state buffers bestudeerd.*
- *In experiment 2 wordt de werking van een module met register en three-state buffers uitgezocht.*
- *In experiment 3 wordt een schakeling gebouwd waarmee vier-bits data tussen registers via een bus wordt getransporteerd.*
- *In experiment 4 wordt een ALU aan de schakeling toegevoegd. Met de schakeling die dan ontstaat, kan de data bewerkt worden door achtereenvolgens de juiste datatransfers handmatig uit te voeren.*
- *In experiment 5 wordt een tweede invoerregister toegevoegd. Het is dan mogelijk langs twee verschillende wegen data in te voeren en met de ALU te bewerken.*
- *In experiment 6 wordt een sequencer gebouwd. Deze schakeling is nodig om iedere datatransfer op het juiste moment te laten plaatsvinden.*
- *In experiment 7 wordt een sequencer aan de schakeling toegevoegd. Nu is het mogelijk de datatransfers automatisch te laten uitvoeren.*

Experiment 1: Module met three-state invoerregister
Gebruik de bovenste helft van een module met opschrift "WOORDGEVER 3 STATE". Deze module bevat twee series van vier three-state buffers. De uitgangen van deze buffers kunnen in de zwevende toestand gebracht worden door op de gemeenschappelijke $\overline{\text{ENABLE}}$-ingang een 1 te zetten. Sluit op deze module een HEXADECIMAAL DISPLAY aan. Maak de $\overline{\text{ENABLE}}$-ingang achtereenvolgens 0 en 1 en verklaar de waarde op het display.

Experiment 2: Module met register en bus

De PIDAC-module met het opschrift REGISTER 3 STATE bevat behalve het register 74LS173, vier three-state buffers en een vier-bits databus. In figuur 7 zijn deze aangeduid als rode stekerbussen. Sluit op de linker rode stekerbussen een 3-state woordgever aan en op de rechter een display. Sluit op de CLOCK-ingang een WAARDEGEVER aan.



Figuur 7: PIDAC-module: "REGISTER 3 STATE"

- Clear het register.
- Kopieer een getal van het invoerregister naar de 74LS173.
- "Disable" ofwel maak de $\overline{\text{ENABLE}}$ -ingang 1 van het invoerregister
- "Enable" ofwel maak de $\overline{\text{ENABLE}}$ -ingang 0 van de buffers tussen registeruitgangen en databus. Welke waarde geeft het display weer?
- Laad de waarde 3 in het register door de $\overline{\text{LOAD}}$ -ingang 0 te maken en daarna een opgaande klokflank te geven. Zet op het invoerregister de waarde 6. "Enable" beide $\overline{\text{ENABLE}}$ -ingangen. Welke waarde geeft het display weer? Verklaar je antwoord.

Dataconflict
Er mag dus hoogstens één databron op de bus beschikbaar/actief zijn!
Bij digitale circuits van het type LSTTL neemt een uitgang die de waarde 0 heeft, veel meer stroom op dan een ingang die de waarde 1 heeft kan leveren. Dus bij een kortsluiting "winnen" nullen het van enen. Bij ic´s van het type CMOS is dat niet zo.

Experiment 3: Datatransport via bus

Bouw een schakeling waarmee 4-bit data uitgewisseld kunnen worden tussen een invoerregister en twee registers via een bus volgens figuur 5.

Gebruik als invoerregister een three-state woordgever (geel deksel) en als klok een waardegever.

- Laad een getal in register A en “disable” daarna de woordgever.
- Kopieer dit getal van register A naar register B via de databus.
- Clear register A en kopieer de waarde van B terug naar A.

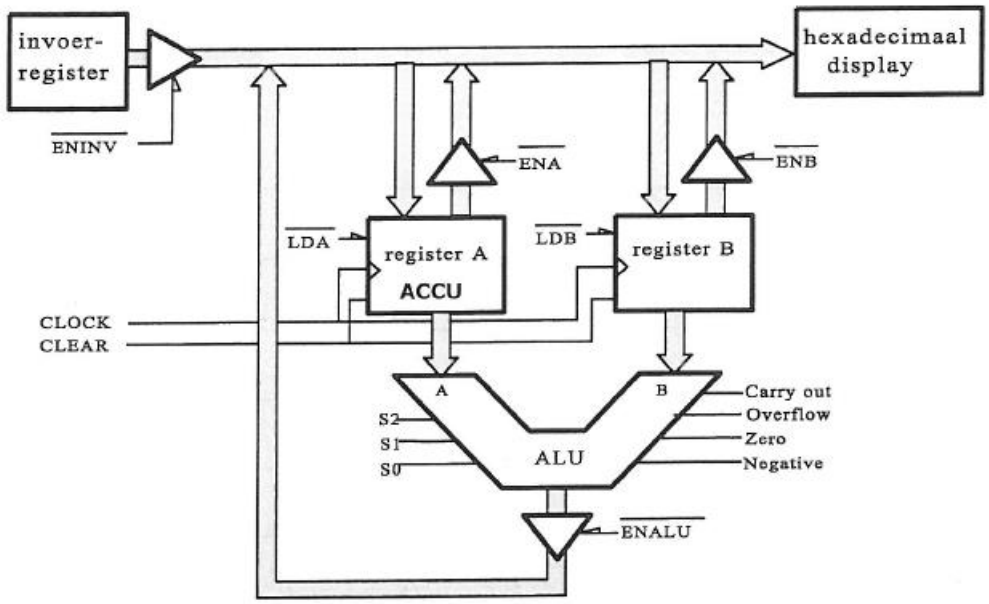
Experiment 4: Datatransfers via ALU naar registers

S_2	S_1	S_0	operatie	
0	0	0	$\overline{B}$	complement van B
0	0	1	B	B wordt doorgegeven
0	1	0	A - B	rekenkundige -
0	1	1	A plus B	rekenkundige +
1	0	0	$A \oplus B$	bitwise XOR
1	0	1	A + B	bitwise OR
1	1	0	A . B	bitwise AND
1	1	1	1111	-1

Tabel 5: Functietabel ALU

Modificeer de schakeling op de volgende wijze (zie figuur 8):

- Voeg aan de schakeling een ALU toe.
- Verbind de uitgangen van de registers vóór de buffers (groene stekerbussen) met de ALU



Figuur 8: Een busgeorganiseerde rekenmachine

Voer de volgende instructies uit:

- LDA # Load Accumulator; ACCU ← invoerregister.
- SUB # ACCU ← ACCU – invoerregister

Het uitvoeren van deze laatste instructie gaat in twee fasen :

fase 1: (hulp)register B ← invoerregister

fase 2: register A ← register A - register B.

Accumulatormachine
Een van de allereerste processoren maakte gebruik van slechts één register. In dit register werden alle tussenresultaten van ALU-bewerkingen opgeslagen. Een gebruikelijke naam voor dit register is ACCU**.** Dit is een afkorting van accumulator of verzamelregister.

Experiment 5: Een bus georganiseerde accumulatormachine
Voeg aan de vorige schakeling een tweede invoerregister toe. Voer de volgende instructie uit:
SUB in1, in2 # ACCU ← invoerregister 1 - invoerregister 2.

Om deze instructie uit te voeren zijn 3 datatransfers nodig. Ofwel voor het executeren van deze instructie wordt een zogenaamd microprogramma bestaande uit micro-instructiestappen uitgevoerd. Schrijf dat programma door in tabel 2 de vereiste logische niveaus bij iedere datatransfer in te vullen.

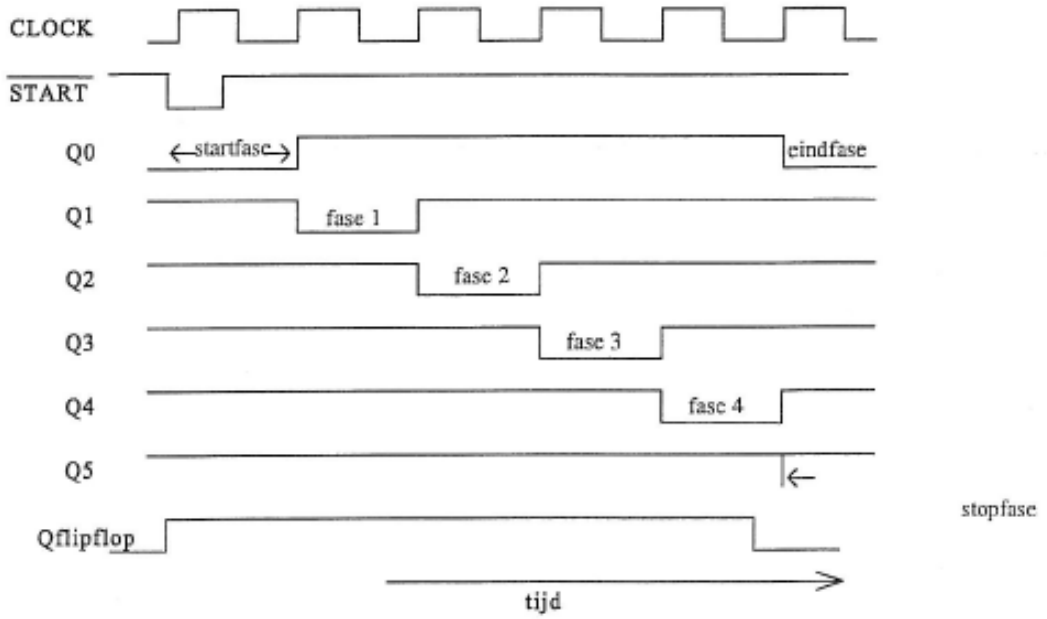
fase / opdracht-ingang	bij aan-vang	fase 1	fase 2	fase 3	na afloop
$\overline{\text{ENINV1}}$	1	....	....	....	1
$\overline{\text{ENINV2}}$	1	....	....	....	1
$\overline{\text{ENA}}$	1	1	....	....	1
$\overline{\text{ENB}}$	1	1	....	....	1
$\overline{\text{ENALU}}$	1	1	....	....	1
$\overline{\text{LDA}}$	1	....	....	....	1
$\overline{\text{LDB}}$	1	....	....	....	1
S_2-ALU	x	x	....	....	x
S_1-ALU	x	x	....	....	x
S_0-ALU	x	x	....	....	x

Tabel 2: Logische niveaus van de stuuringangen voor iedere fase.

Houd de schakeling intact voor verdere experimenten!

12.5 Sequencer

Om de achtereenvolgende fasen van een bewerking volledig automatisch te laten verlopen, wordt een sequencer gebruikt. In figuur 12 is weergegeven welke signalen een Sequencer moet genereren om bijvoorbeeld een tijdsafhankelijk proces, dat in vier fases verloopt, te

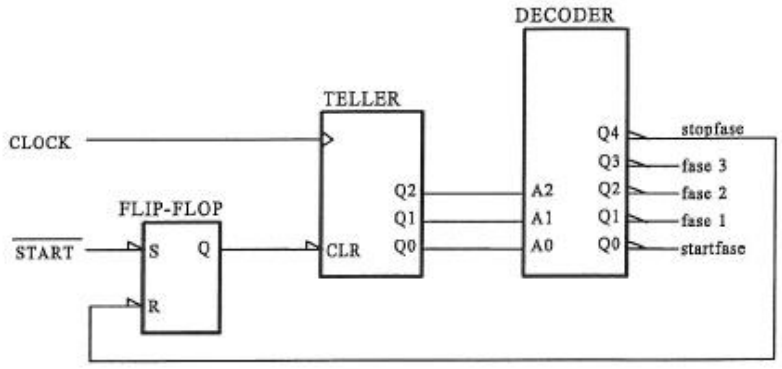


Figuur 12: Timing diagram van de sequencer

kunnen executeren. De cyclus wordt door een extern $\overline{\text{START}}$-signaal in combinatie met de eerstvolgende positieve klokflank gestart. Door een door de schakeling zelf gegenereerd $\overline{\text{STOP}}$-signaal wordt de cyclus beëindigd. Na de start begint iedere fase op een opgaande flank van een klokpuls. Elke actie wordt bestuurd door de klok en is dus synchroon.

In figuur 13 is de sequencer geïmplementeerd door middel van een vier-bits synchrone teller, een decoder en een Set-Reset-latch. Als teller gebruiken we het ic 74LS161. Als de COUNT-ingang 1 is wordt de tellerstand bij elke opgaande flank met 1 verhoogd. Als decoder gebruiken we een 3 naar 8 decoder type 74LS138. Deze decoder heeft "laag actieve" uitgangen. Afhankelijk van de code op de ingangen is één uitgang 0 en zijn de andere uitgangen 1. Voor de Set-Reset latch is het handig om de $\overline{\text{SET}}$ en $\overline{\text{RESET}}$-ingangen van een D-flipflop type 74LS74 te gebruiken. Deze is opgebouwd uit NAND-gates (zie boek figuur 5.35)

Experiment 6: Bouw en test een sequencer met een start-, een stop- en drie signaalfases



Figuur 13: Sequencer

Experiment 7: Geautomatiseerde rekenmachine
Voeg aan de schakeling van experiment 5 een sequencer toe met een start-, een stop en drie signaalfases.
Sluit de uitgangen van de sequencer aan op de juiste $\overline{\text{LOAD}}$- en $\overline{\text{ENABLE}}$-ingangen.
Start de executie van dit experiment door één druk op een knop. Welke (druk)knop wordt hier bedoeld?
Eén van de opdrachtingangen, $\overline{\text{LDA}}$, moet in twee fases 0 worden. Ga na dat een AND-poort nodig is om deze ingang bij de juiste micro-instructiestappen te activeren.
Zorg ervoor dat steeds op de neergaande klokflank de volgende fase ingaat en dat op de opgaande flank steeds een register wordt geladen. De set-up time is hierdoor lang genoeg.

12.6 Begrippenlijst

Accu, accumulator. Het register waarin de tussenresultaten van ALU-bewerkingen worden opgeslagen (verzamelregister) bij een zgn. accumulatormachine.

Databus. Een aantal parallel lopende lijnen waarover het datatransport in een computer plaatsvindt. Naast datalijnen bevat een bus ook controlelijnen en adreslijnen. Een voorbeeld van een controlelijn is de lijn waarover het uitgangssignaal van de systeemklok wordt getransporteerd.

Datatransfer. De overdracht van data van het ene register naar het andere (al of niet via een bus en/of ALU).

$\overline{\text{ENABLE}}$**-ingang.** Een ingang waarmee de waarde op uitgang van een three-state buffer wordt doorgelaten of geblokkeerd.

$\overline{\text{LOAD}}$**-ingang.** Een ingang van een register waarmee (nieuwe) gegevens kunnen worden geladen. Bij de meeste registers is voor het laden van data ook nog een klokflank nodig.

Micro-instructie. Elementaire instructie zoals ADD of MOVE. Deze bestaat uit meerdere datatransfers.

Micro-instructiestap. Eén van de fasen waarin een micro-instructie geëxecuteerd wordt.

Sequencer. Schakeling die de timing genereert voor het uitvoeren van een sequentie van gebeurtenissen.

Three-state buffer. Een schakeling waarvan de uitgang of zwevend is of gelijk is aan de inputwaarde 0 of 1.