

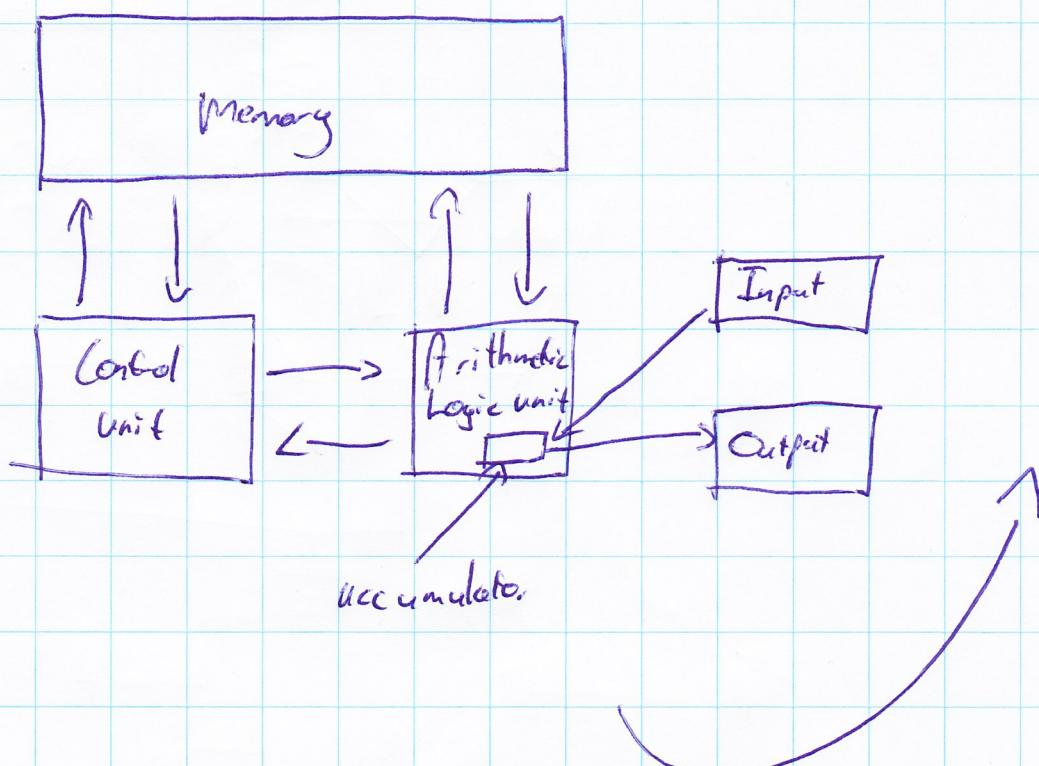
1.

Processoren

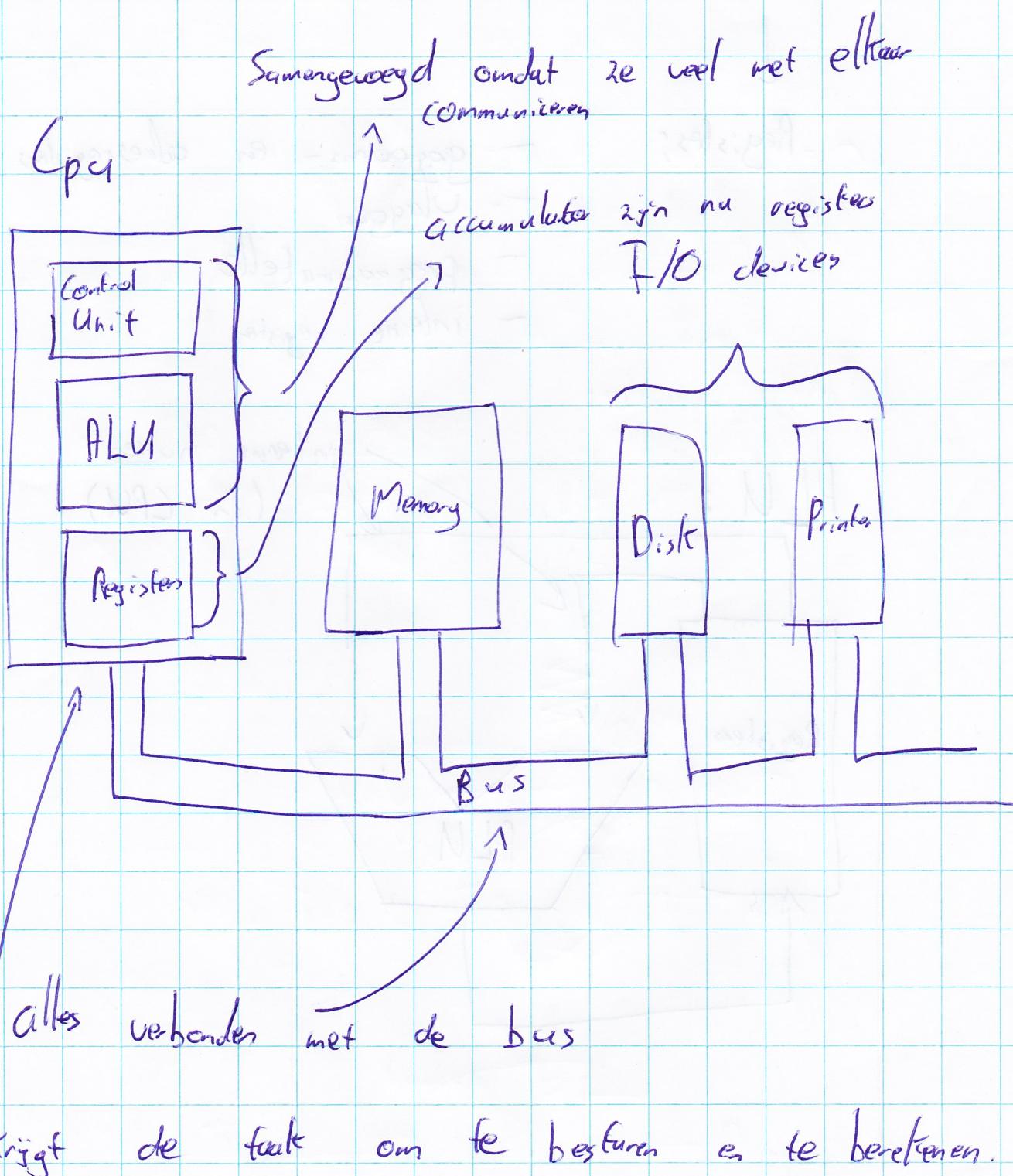
Von Neumann architectuur: (historisch)

Hoofdonderdelen:

- besturingseenheid
- rekenseenheid
- geheugen
- in- en uitvoerapparaten



Von Neumann architectuur



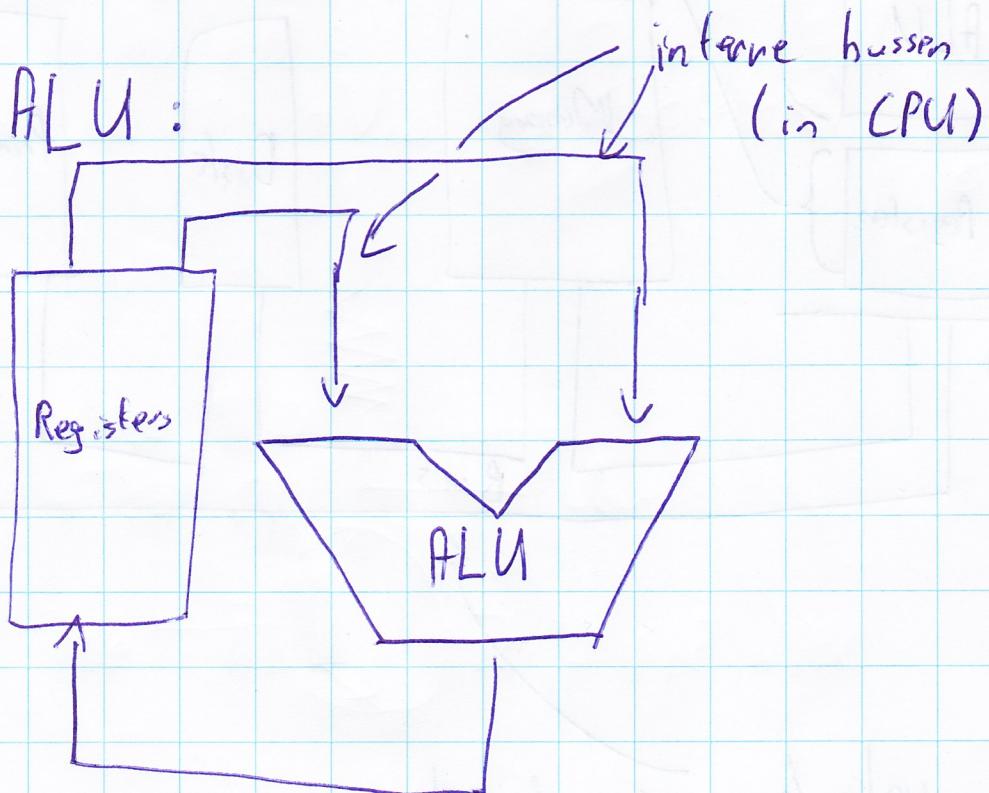
CPU



Registers:

- gegevens - en adresregisters
- vlaggen
- programma teller
- interne registers

ALU:



Gedrag van de CPU:

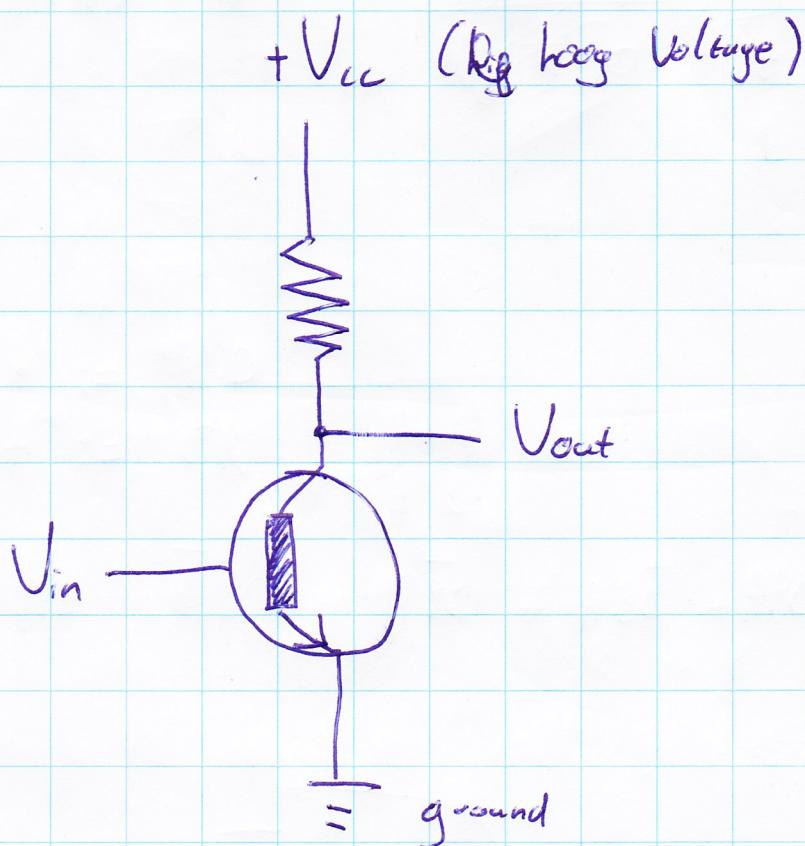
1. fetch (programmatische ophogen & instructie lezen)
2. decode (instructie decoderen)
3. execute (instructie uitvoeren)
4. pause

Art of Processor Design: dit goed laten verlopen.

2.

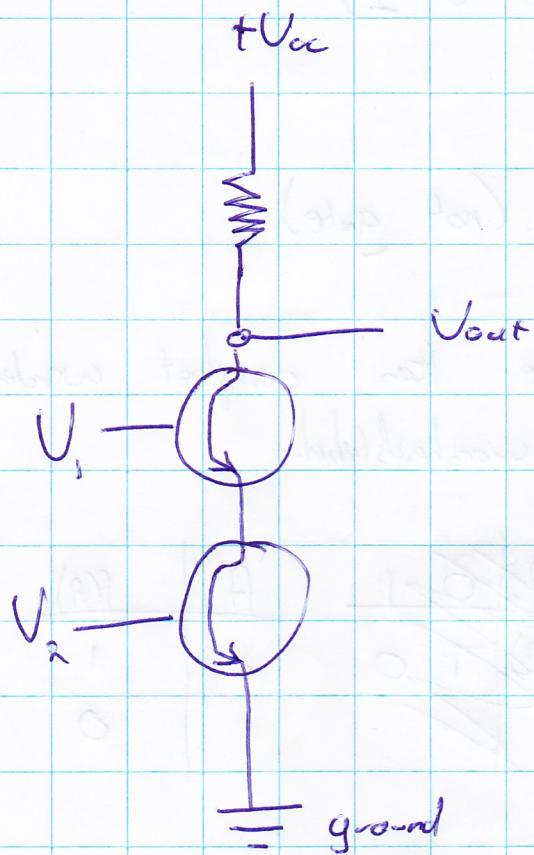
$$O = 0 - 0,5 \text{ Volt}$$
$$I = 1 - 1,5 \text{ Volt}$$

Transistor:

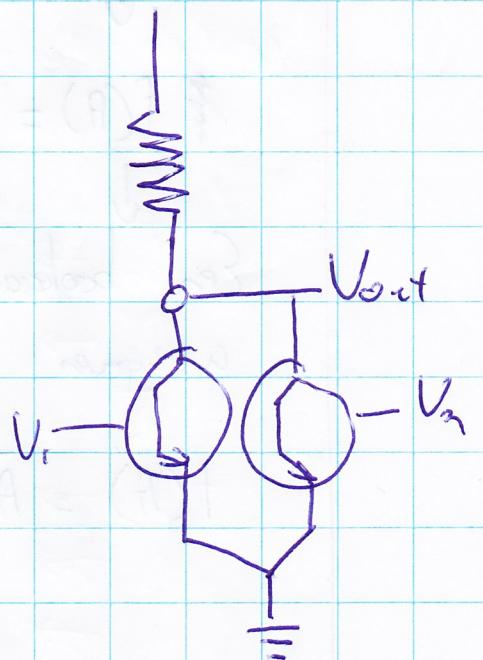


Als V_{in} hoog is gaat de transistor zich opladen als een diode en wordt V_{out} laag. Dit is dus een inverter.
Hiermee kunnen ook NOR en NOT-gates worden gemaakt.

NAND:



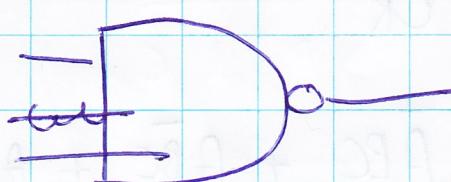
NOR:



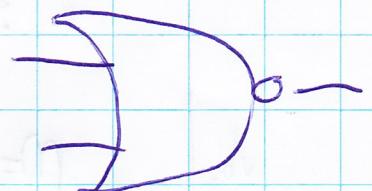
Gates:



Not



Nand



Nor



and



OR

Boolean algebra: algebra met booleanse waarden
(0 of 1)



$$f(A) = \bar{A} \quad (\text{not gate})$$



Een booleanse functie kan compleet worden
weergeven in een waarheids tabel.

$$f(A) = \bar{A} :$$

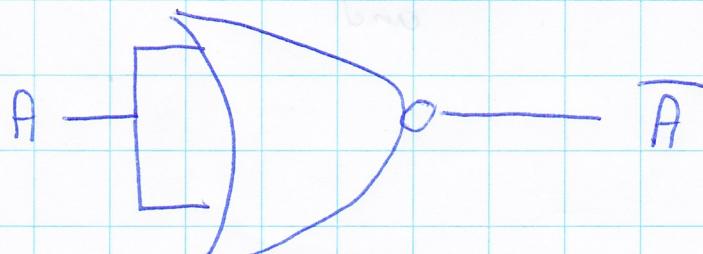
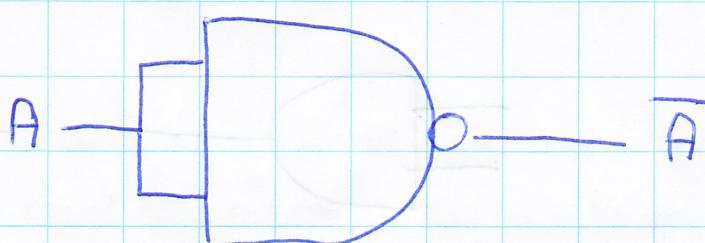
~~$$\begin{array}{c|c} A & f(A) \\ \hline 0 & 1 \\ 1 & 0 \end{array}$$~~

| A | f(A) |
|---|------|
| 0 | 1 |
| 1 | 0 |

$\cdot := \text{AND}$

$+$:= OR

v.b. $M = \bar{A}BC + A\bar{B}\bar{C} + AB\bar{C} + A\bar{B}C$



Booleaanse algebra regels:

$$AB + AC = A(B + C)$$

$$\overline{ABC} = \overline{A} + \overline{B} + \overline{C}$$

$$(\neg(A \wedge B \wedge C) = \neg A \vee \neg B \vee \neg C)$$

Regels

AND

OR

Identiteit

$$1A = A$$

$$0 + A = A$$

Nul

$$0A = 0$$

$$1 + A = 1$$

Idempotent

$$AA = A$$

$$A + A = A$$

Inverso

$$A\bar{A} = 0$$

$$A + \bar{A} = 1$$

(Commutatief)

$$AB = BA$$

$$A+B = B+A$$

Associatief

$$(AB)C = A(BC)$$

$$(A+B)+C = A+(B+C)$$

Absorptie

$$A(A+B) = A$$

$$A + AB = A$$

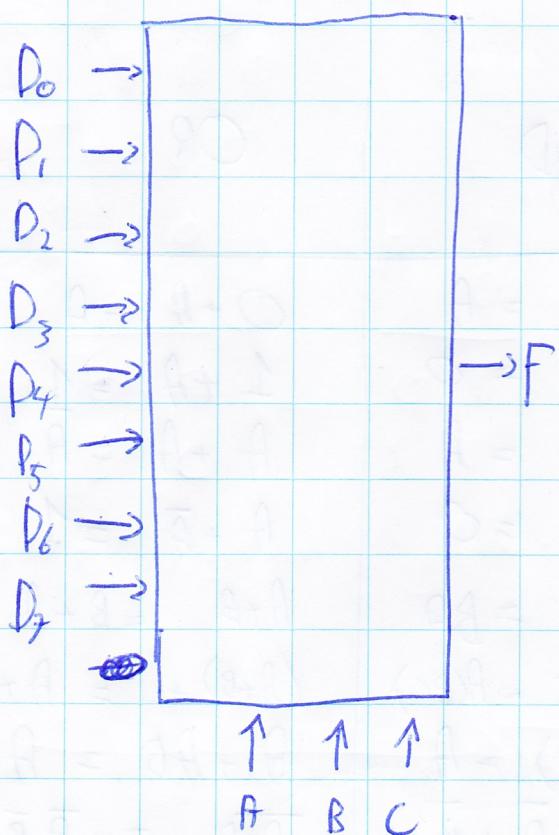
De Morgan

$$\overline{AB} = \overline{A} + \overline{B}$$

$$\overline{A+B} = \overline{A}\overline{B}$$

Multiplexer

Een multiplexer heeft 2^n inputs en n control - inputs:

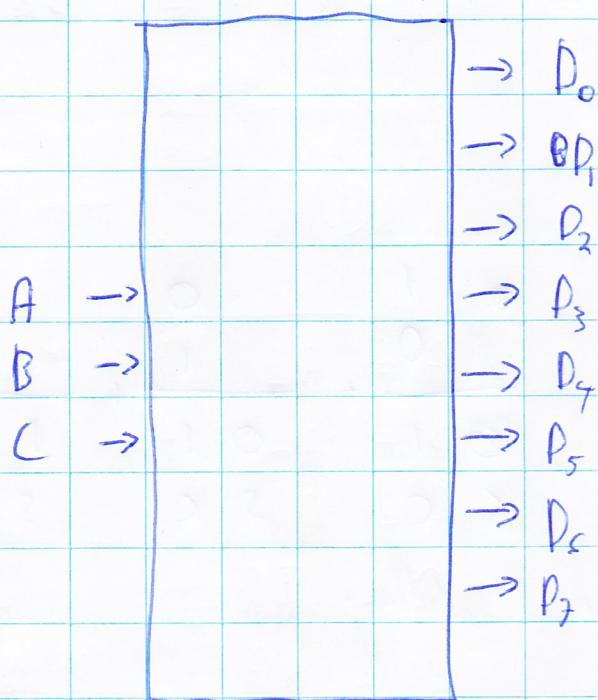


De control - inputs A, B en C bepalen welke data input D_n er wordt getozen.

Vb. Stel $A = 1$ $B = 0$ $C = 1$ ($101_2 = 5$)
dan wordt D_5 ingang D_5 uitgang als output

De code:

Een decoder neemt n inputs en heeft 2^n outputs.



Vb Stel $A = 0, B = 1$ en $C = 0$ ($2^3 = 8$) dan wordt D_2 gelijk aan 1.

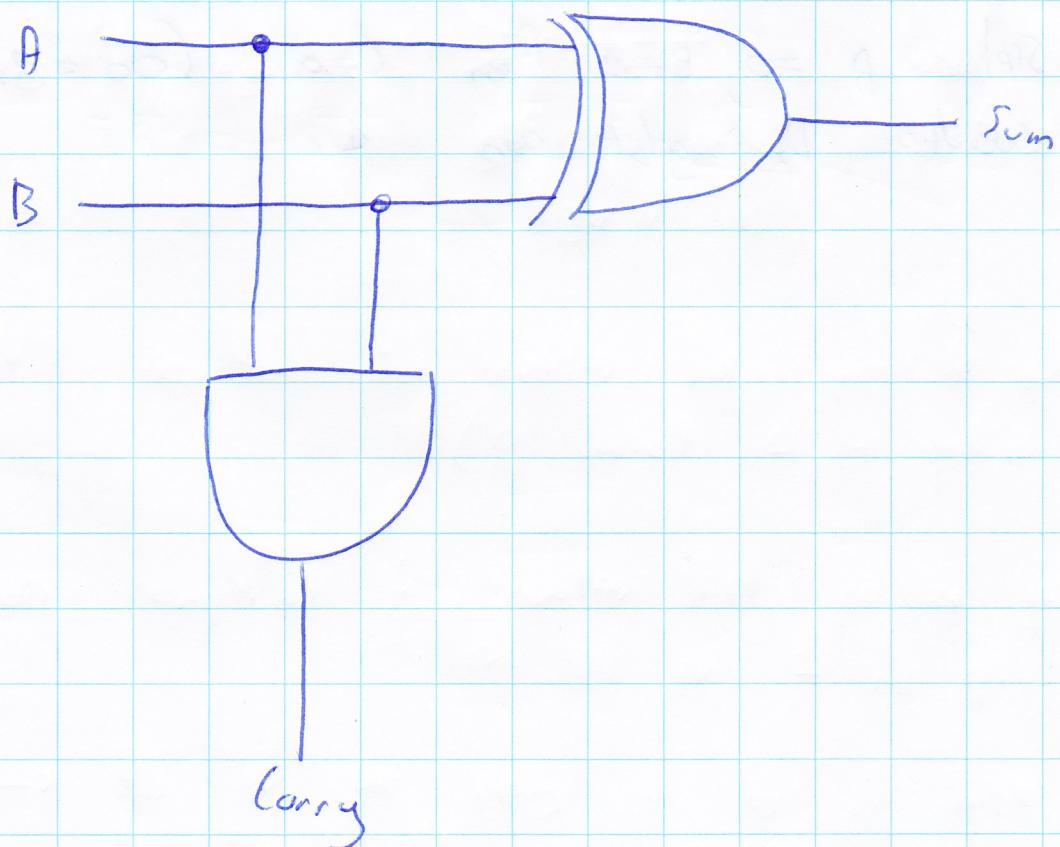
Half adder:

Optellen van twee 1-bits getallen A en B:

$$Sum = A \oplus B$$

$$Carry = AB$$

$$\begin{array}{r} A \\ B \\ \hline 0 & + & 0 & + & 0 & + & 0 \\ 0 & & 1 & & 1 & & 1 \\ \hline 00 & 01 & 01 & 01 & 10 \\ S & C & S & C & S & C \end{array}$$

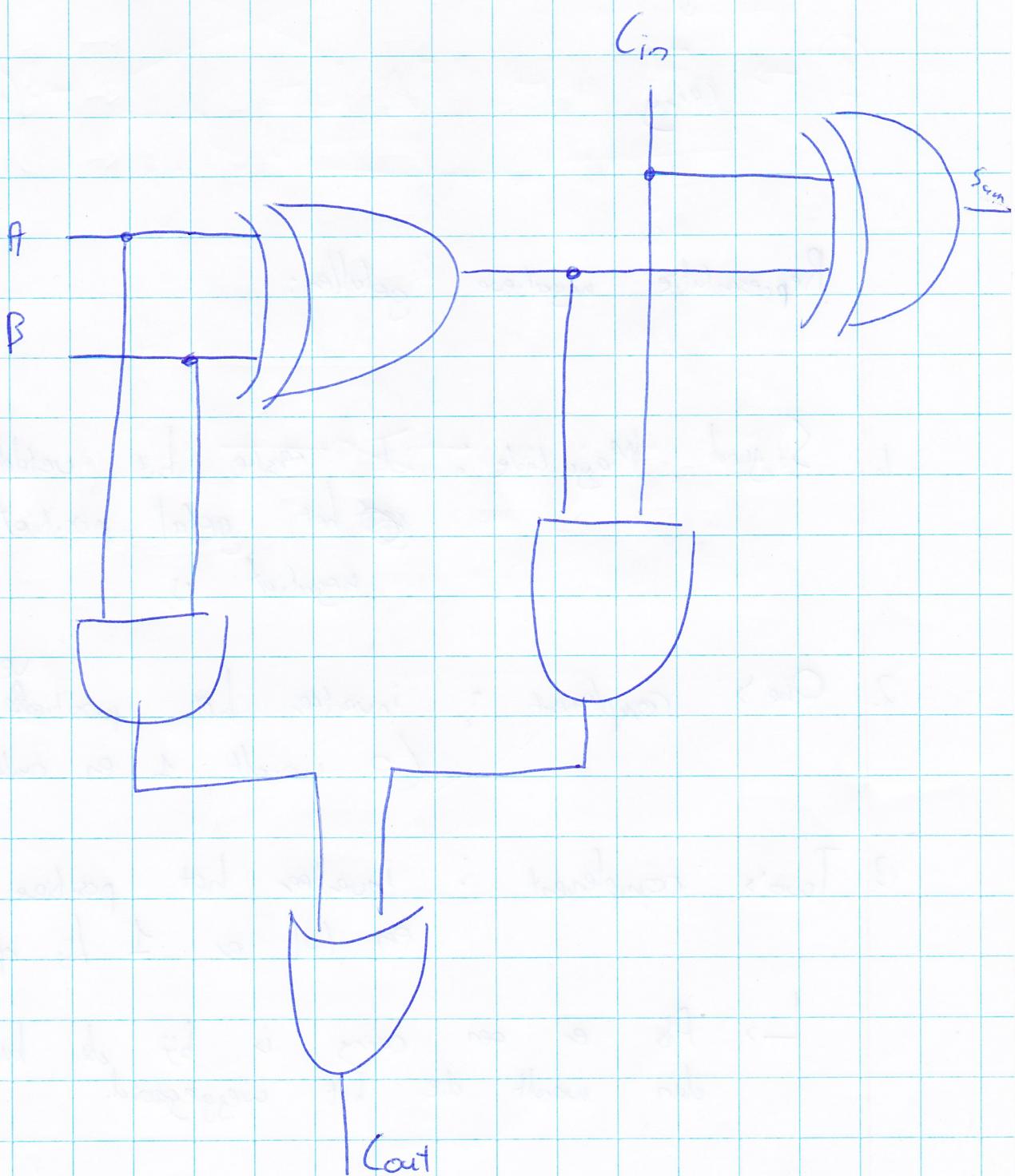


Full adder:

Optellen van twee 1-bits getallen met carry-in:

$$Sum = A \oplus B \oplus C_{in}$$

$$Carry = AB + (A \oplus B) \cdot C_{in}$$



3.

Binair optellen:

$$\begin{array}{r} 1 \ 1 \\ 010 \quad 0 \ 1 \ 0 \\ \hline \underline{1 \ 1 \ 0} + \end{array}$$

$$\begin{array}{r} 1 \ 0 \ 0 \ 0 \\ \downarrow \\ \text{Carry} \end{array}$$

Representatie negatieve getallen:

1. Signed Magnitude: de eerste bit verteld of het getal positief of negatief is.

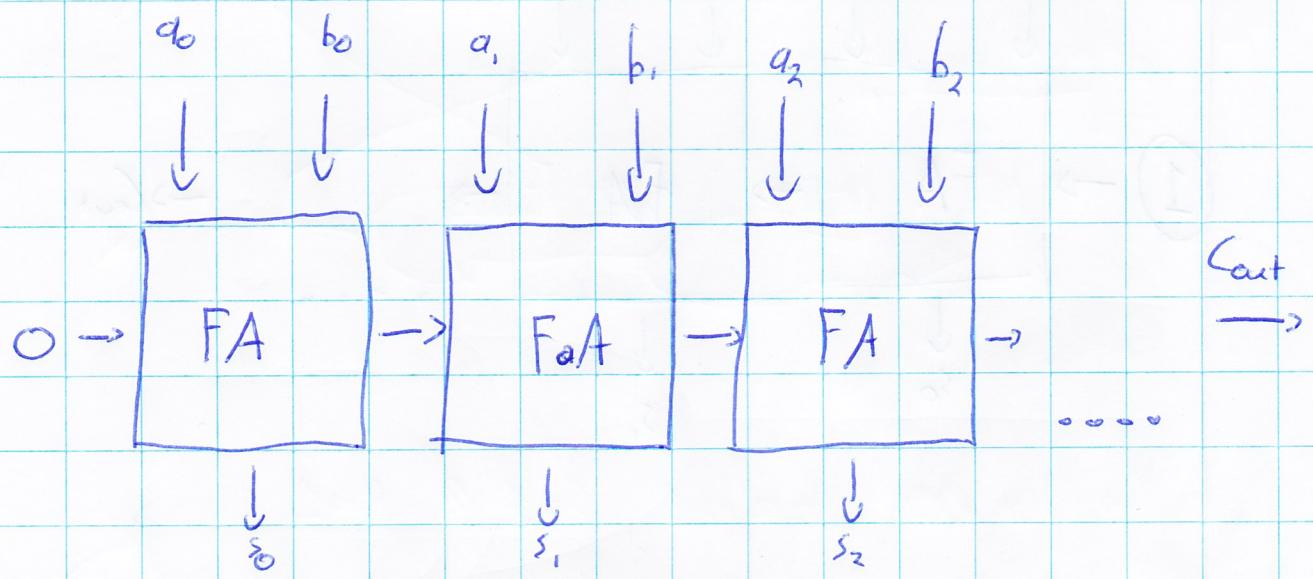
2. One's complement: inverteer het positieve getal (0 wordt 1 en andersom)

3. Two's complement: inverteer het positieve getal en tel er 1 bij op.

↳ Als er een carry is bij de buitenste bit dan wordt die bit weggegooid.

Adder met n-bits getallen:

Door n full adders te combineren krijgen we een adder voor n -bits getallen.



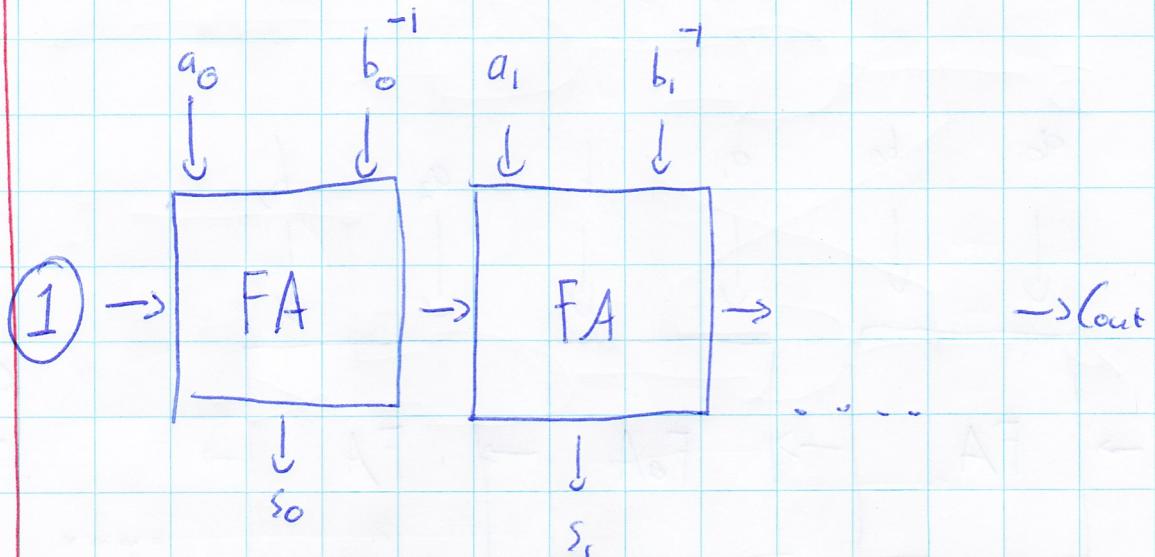
$$a = a_n \dots a_2 a_1 a_0$$

$$b = b_n \dots b_2 b_1 b_0$$

Effetek Aftrekken in schakelingen

$$x - y = x + (-y)$$

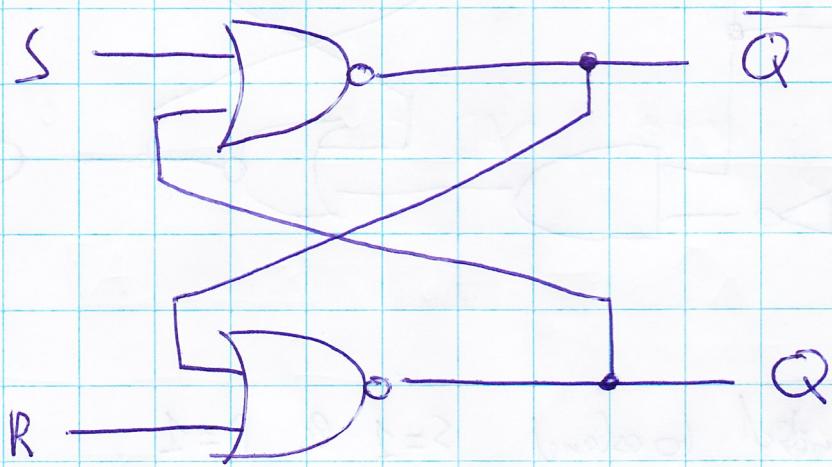
De adder wordt



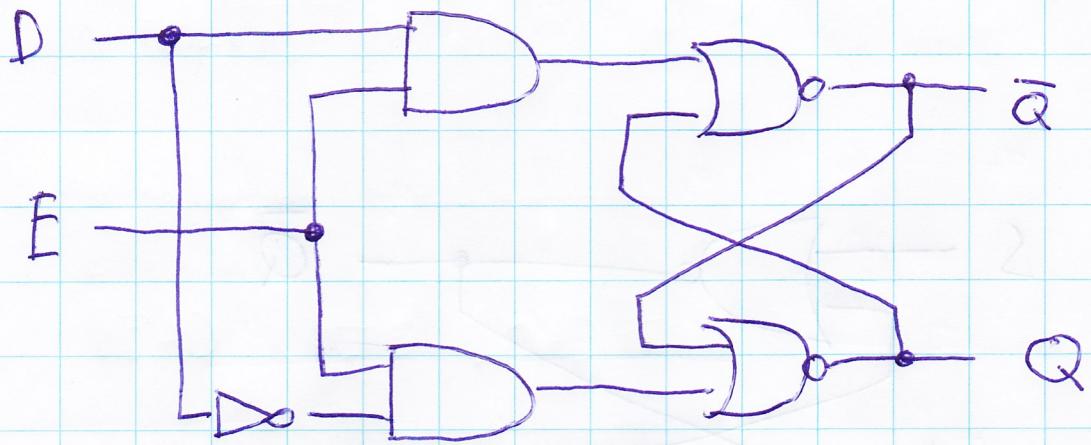
$$b_n^{-1} = b \rightarrow \text{Do} \rightarrow b^{-1}$$

Gebogen in een schakeling:

SR - Latch:



D Latch



↳ vermind toestand $S=1$ & $R=1$

$E = 0$: latch houdt waarde vast

$E = 1$: waarde van D wordt in latch opgeslagen.

4.

Programma in het RAM-geheugen:

- bestaat uit machinecodes

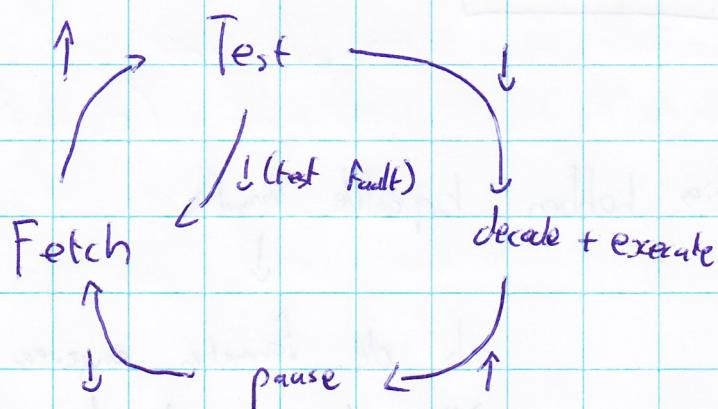
Fetch: er wordt een stukje programma in de control unit geladen (en de program counter wordt opgehoogd).

↓
met de ALU

Decode: het stukje programma wordt in de control unit gedecodeerd.

E. Test Execute: het stukje programma wordt uitgevoerd.

CPU:



5.

Instructies:

- Programma flow beïnvloeden

HALT

- Gegevens transport

READ, WRITE, LOADHI

- Berekenen

ADD, XOR, ...

- Stack

PUSH, POP



Instructies hebben bepaalde formaten



als de formaten overeen gelijk zijn dan wordt de control unit makkelijker te begrijpen

Lommige assembly instructies hebben geen machinecode instructie:

- Jump
- TPA/Movz
- CmpF

...



worden door de assembler vertaald.

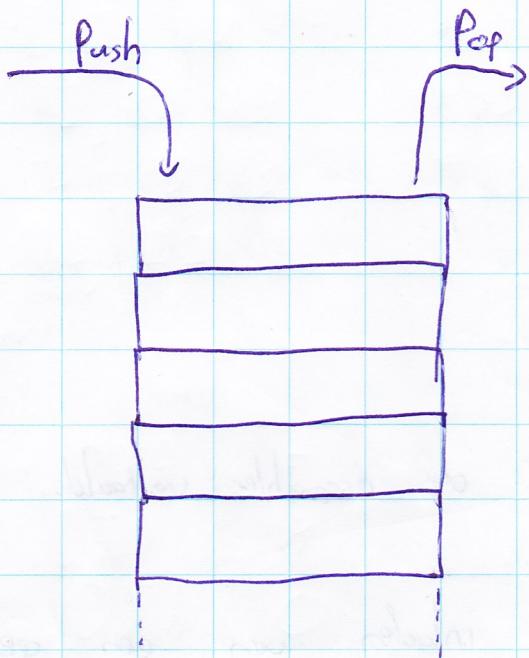
LOADHI: inclusief van een gefal van 32 bits.



laad eerst 23 eerste bits in en voeg dan met een ADD commando de laatste bits toe.

De stack

Volgorde: Last in, first out



Stackpointer: wijst aan waar de het eind van de stack zich bevindt
(In het prakticum R14)

Compiler: vertaald naar assembly code

Assembler: vertaald naar machine code (0 en 1)

6.

Programmeertalen: abstracte, wiskundige notatie
Bijvoorbeeld:

Assembly: mnemonische instructienamen
↳ hoort bij een type CPU) assemble

Machine code: bitpatronen

Assembly: begint altijd met .Start 0

Syntax:

Algemeen: label: Instructie COND opstanden # commando

Constante of globale variabiele definities:

i: .DATA 42

Bij elke register schrijven voor het waar wordt gebruikt.

jb. .START O

Bereken $K = i + j - 3$

READ [i], R1

R1 = i

READ [j], R2

R2 = j

ADD R1, R2, R3

R3 = R1 + R2

ADD -3, R3, R3

R3 = K

WRITE R3, [K]

HALT

i: .DATA 5
j: .DATA 8
k: .DATA 0

Pseudo instructies:

.START const : plaats volgende instructie op adres const.

.ALGN const : Rond adres van volgende instructie naar boven af op een veelvoud van const.

.DATA c1, c2, ... : Reserver en initialiseer gehangen.

.LOAD const, Rn : Laad constante in register Rn
(mbv LOADHI & ADD)

- (PL L label : functieaanroep (label binnen bereikt Jump)
- (ALL FAR label : functieaanroep (label buiten bereikt Jump)

Rotatie : bits worden van plek verschoven en bits aan het einde van de rij worden naar het begin verplaatst.

Shift : bits worden van plek verschoven en bits aan het einde van de rij verdwijnen.
Er worden 0-en ingevoegd aan het begin van de rij.

Kan worden gebruikt voor vermenigvuldigen:

$$x \ll n = x \cdot 2^n$$

$$x \gg n = \lfloor x \cdot 2^{-n} \rfloor$$

(vermenigvuldigen)
(delen)

Conversie van ASCII teksten:

$$7 + '0' = '7'$$

$$3 - '0' = 3$$

Waar komen de bron- en doeloperanden van een instructie vandaan?

- accumulator
- register
- constante
- Push of Pop
- geheugen
 - vast adres
 - berekend adres (register + offset)
- indirect adres (pointer o.i.d.)

Addressing modes: de manieren waarop de Operanden van een instructie worden gespecificeerd.



vb. Relleninstructies kunnen van accumulator OF van een register of sommige instructies maken gebruik van vaste registers.

MOVE vs READ

MOVE : verplaatsen van register waarde of constante naar register.

READ : verplaatsen van het geheugen naar een register.

WRITE : verplaatsen van een register naar het geheugen.

Getallen vergelijken:

CMPF
↓
op1 , op2

Berekend op1 - op2 en zet vlaggen onder staart nits op.

Condities:

- T
- E
- L
- G
- Z
- C
- O
- N
- F

Uitvoeren:

uitvoeren

altijd

$r = 0$

$r < 0$

$r > 0$

$Z = 1$

$C = 1$

$O = 1$

$N = 1$

naast

kunnen ook met

◦ NI...] (of [...])

Zo Jump label buiten bereik:

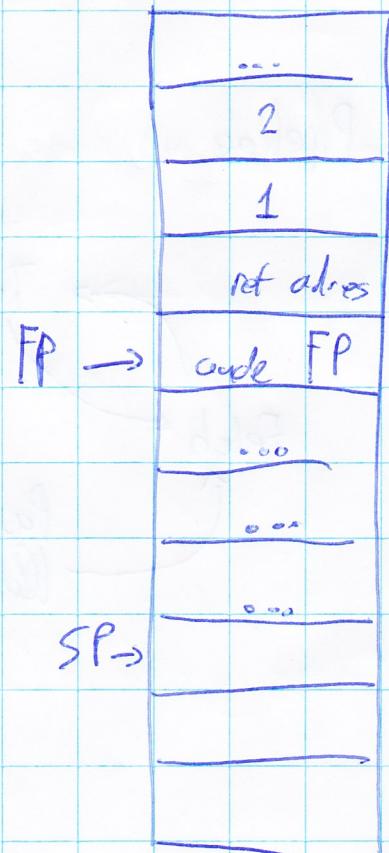
.LOAD const, R105

Functie:

- stukje code dat je meerdere keren wilt aanroepen.
- Voor het aanroepen van de functie het terugkeeradres op de stack zetten.

Als de functie voorbij is moet de stack worden als voor de functie aangevallen.

↓
Frame pointer →



Nu kunnen de parameters aan
de hand van de FP aank
wifgelezen.

Altijd Administratie van een functie

Functie begin:

- zet de oude FP op de stack (push FP)
- Frame pointer initialiseren ($FP := SP$)
- ruimte reserveren voor lokale variabelen
 - $(SP := SP - 4 \cdot \text{aantal variabelen})$

Functie-einde:

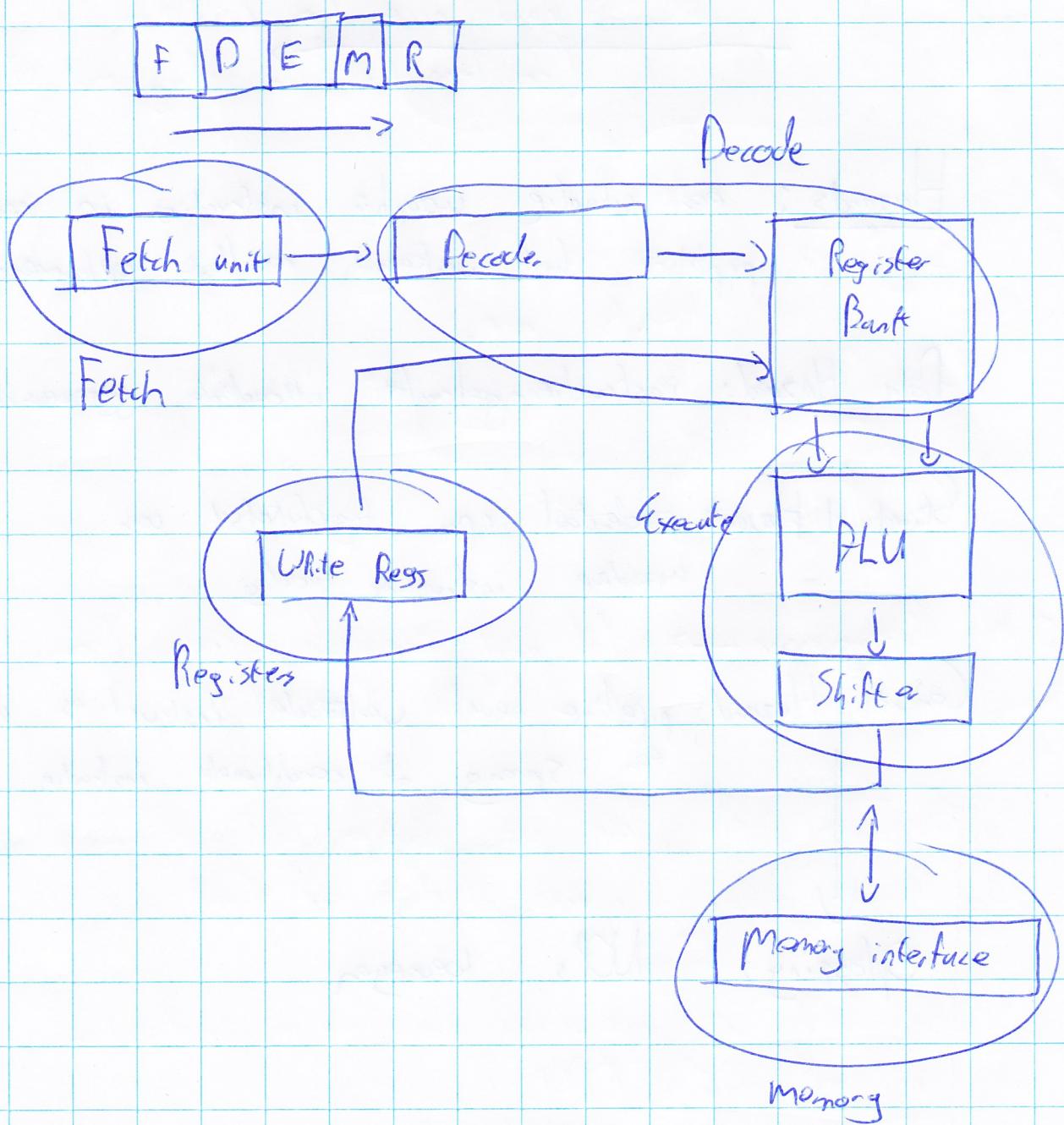
- lokale variabelen vrijgeven ($SP := FP$)
- oude FP herstellen (POP FP)
- terugkerend adres van de stack halen en
er naartoe springen (RET)

Pipeline

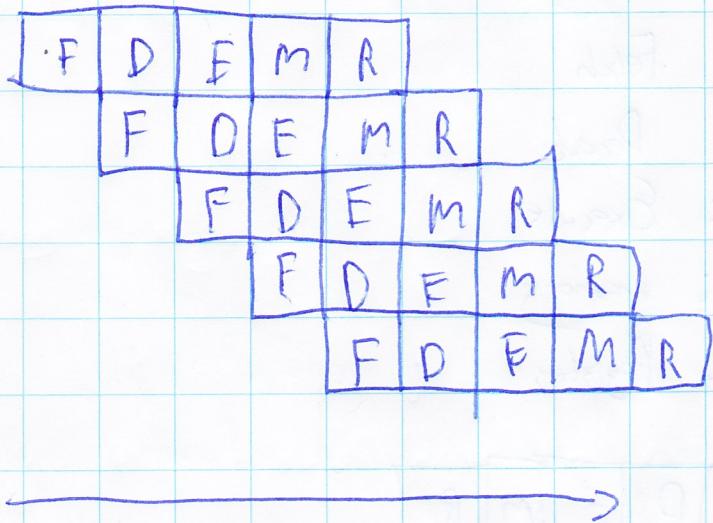


De onderdelen van een CPU hoeven niet in elke fase gebruikt te worden.

- [F] : Fetch
- [D] : Decode
- [E] : Execute
- [m] : memory
- [R] : Registers



Pipelines fases sneller achter elkaar zetten:



Hazards: een situatie waarbij instructies in een pipeline het verkeerde resultaat opleveren.

Data Hazard: instructie gebruikt ouderde gegevens.

Structural Hazard: onderstaal op tegelijkertijd voor meerdere instructies nodig.

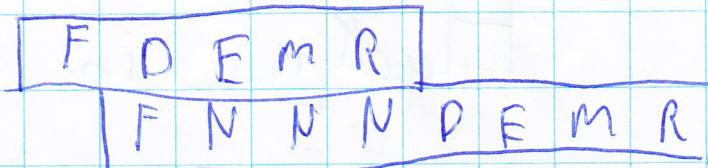
Control Hazard: pipeline beurt verkeerde instructies na een Sprong of conditionele instructie.



Oplossing: NOP's toevoegen

Data Hazards:

SUB R1, R2, R3
ADD S, R3, R4



af: Er kan ook een speciale lus worden gebouwd uit de ALU terug in de ALU om Data hazards te voorkomen.

Register forwarding

Structural Hazards:

- functionele onderdelen dupliceren, of
- Pipeline stall

Control Hazards:

- pipeline flush \rightarrow alles wat in de pipeline zit wordt er uitgegooid. (NOPS invoegen).
- Branch delay slot: instructies na JUMPs worden altijd uitgevoerd
- Speculative execution: "galt" of de branch wel/niet genomen wordt, pipeline flush alleen als do galt (niet) verkeerd was.