

Computer Arkitektur

Processor Arkitektur I:

Den Sekventielle Y86-64

Forelæsning 6
Brian Nielsen

*Credits to
Randy Bryant & Dave O'Hallaron (CMU)*

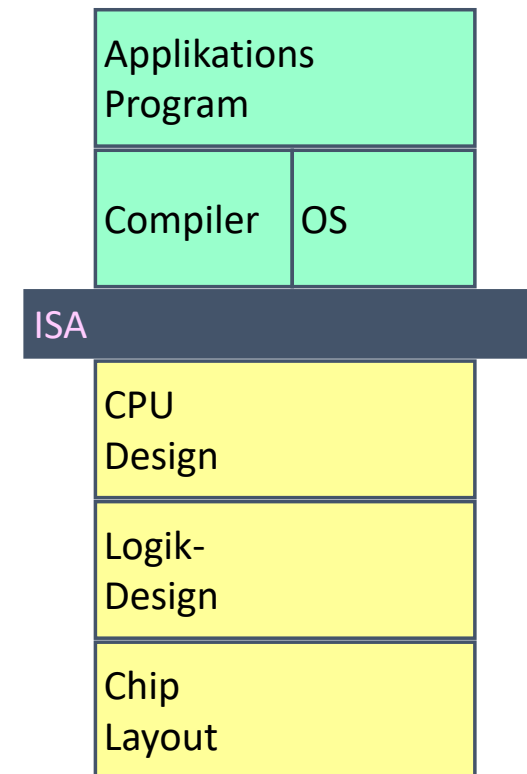
Instruktionssæt Arkitektur (ISA)

- **Assembler Syn**

- Programmørsynlig processor tilstand
 - Registre, hukommelse,...
- Instruktioner
 - `addq, pushq, ret, ...`
 - Indkodning som byte-kode

- **Abstraktionslag**

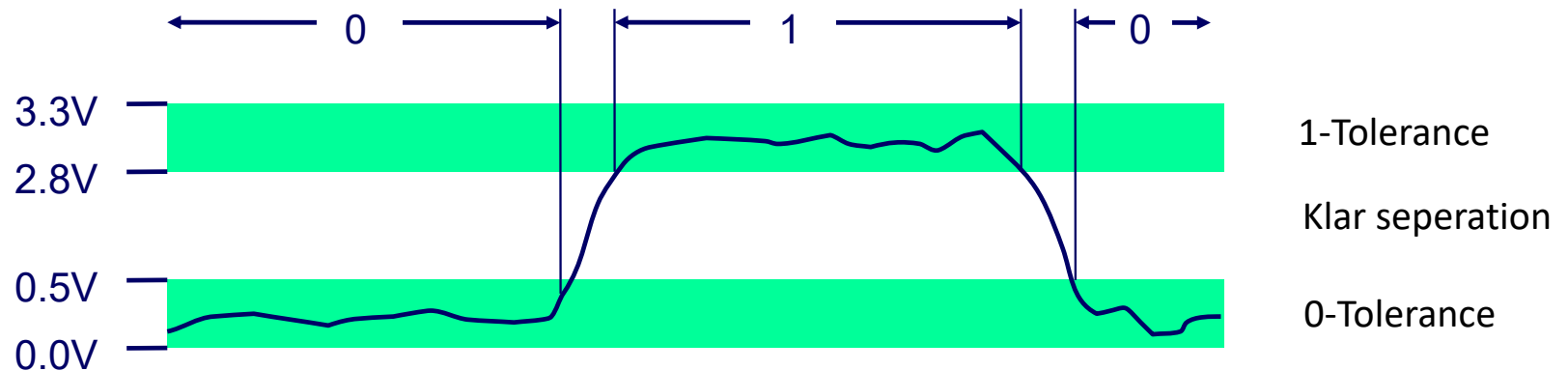
- Over: hvordan programmerer man maskinen?
- Funktionel adfærd
 - Processor udfører instruktioner sekventielt
- Under:
 - Hvordan konstrueres det i digital logik ?
 - Yde-evne?
 - Flere instruktioner udføres parallel!



Digitale kombinatoriske kredsløb

Netværk af Gates uden cykler!

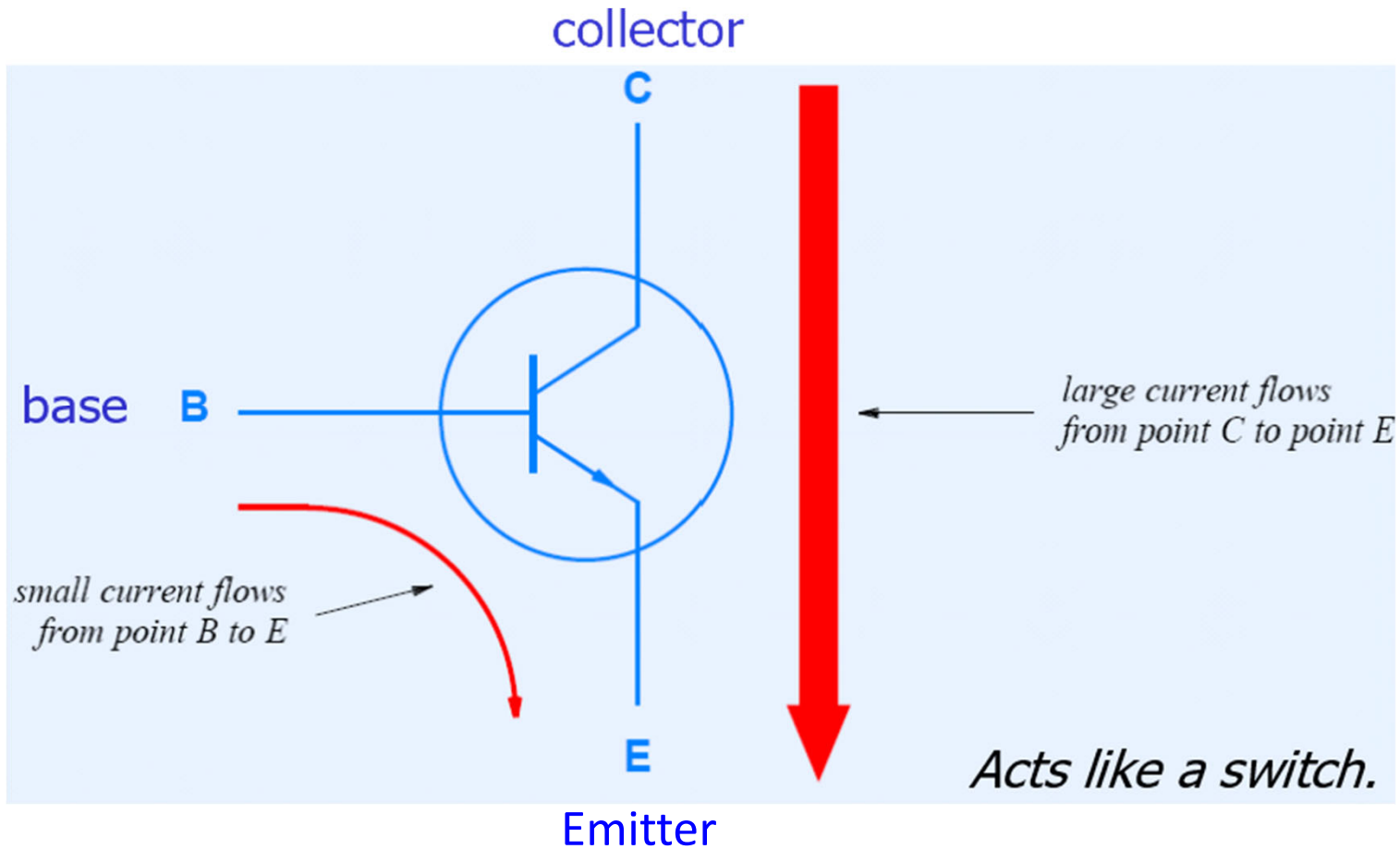
Digitalt Signal



- Brug tærskelværdier for elektrisk spænding/ladning til at diskretisere et kontinuert elektrisk signal
- Simpleste version: 1-bit signal
 - Enten højt område (1) eller lavt område (0)
 - Med en separation imellem dem
 - Robust: påvirkes ikke af støj eller kredsløb af lav kvalitet
 - Vi kan lave digitale kredskøb simpel, småt, og hurtigt

Den grundlæggende byggeblok i digitale kredsløb



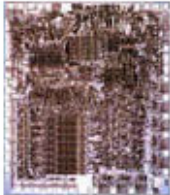
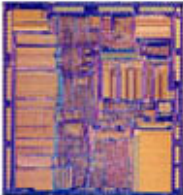
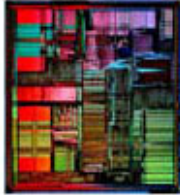
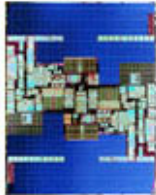
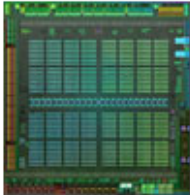
Transistor



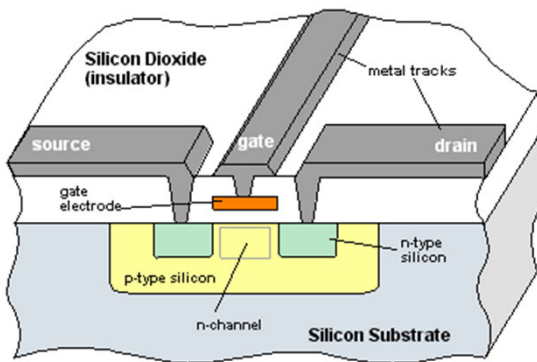
Realisering i halv-leder materiale

Very Large Scale Integration

- <https://www.computerhistory.org/siliconengine/>

1950s	1960s	1970s	1980s	1990s	2000s	2010s
Silicon Transistor	TTL Quad Gate	8-bit Microprocessor	32-bit Microprocessor	32-bit Microprocessor	64-bit Microprocessor	3072-Core GPU
						
1 Transistor	16 Transistors	4500 Transistors	275,000 Transistors	3,100,000 Transistors	592,000,000 Transistors	8,000,000,000 Transistors

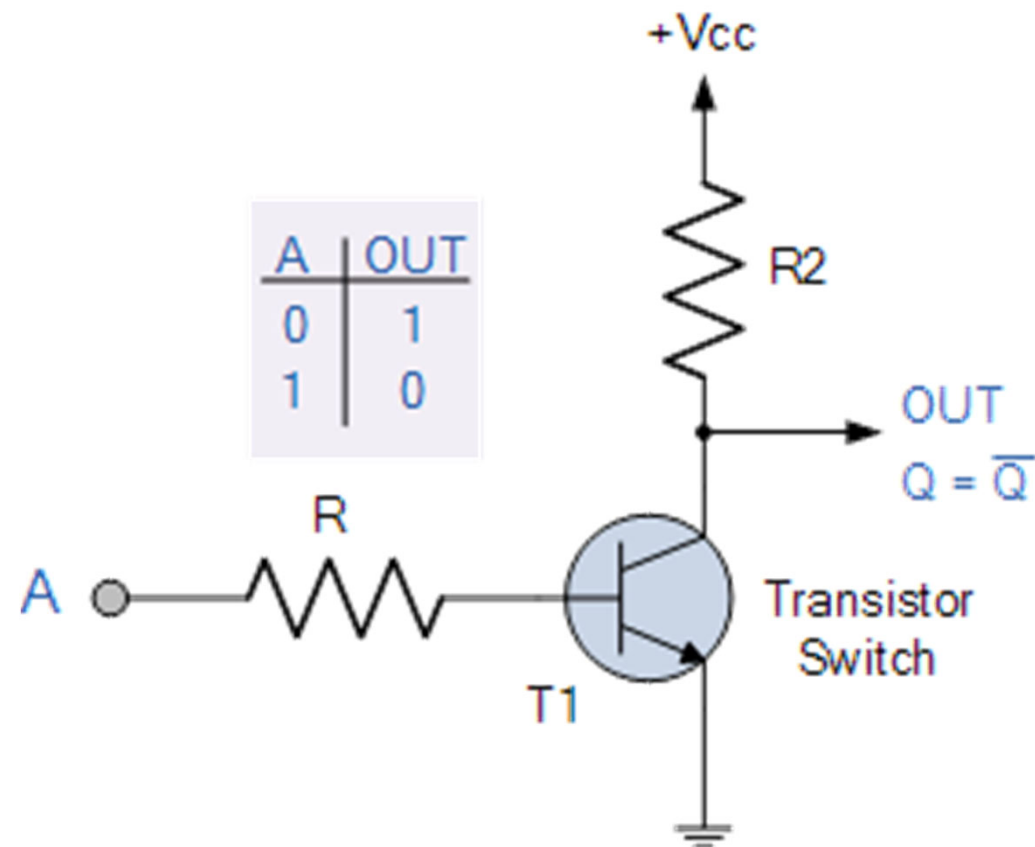
NMOS Transistor
(n-channel MOSFET)



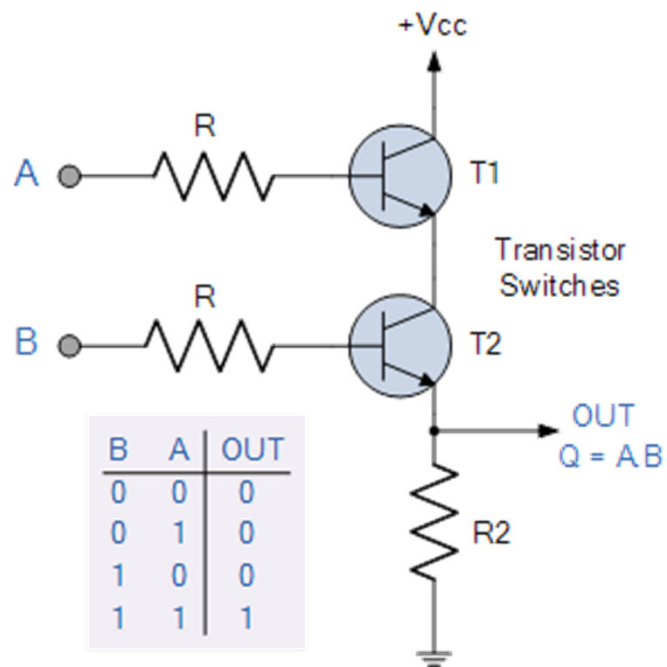
Et par korte intro videoer:

- C-MOS teknologi: <https://www.youtube.com/watch?v=stM8dgcY1CA>
- Fabrikation af IC'ere: <https://www.youtube.com/watch?v=vK-geBYygXo>

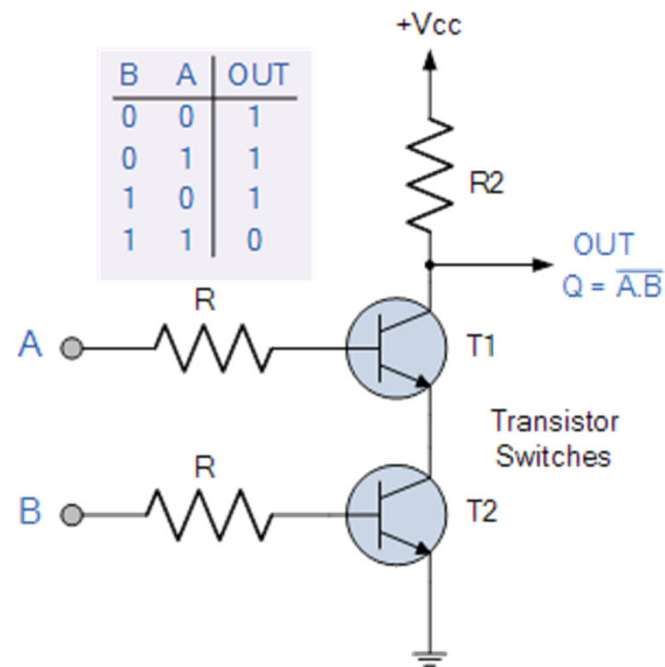
Eksempel: Not



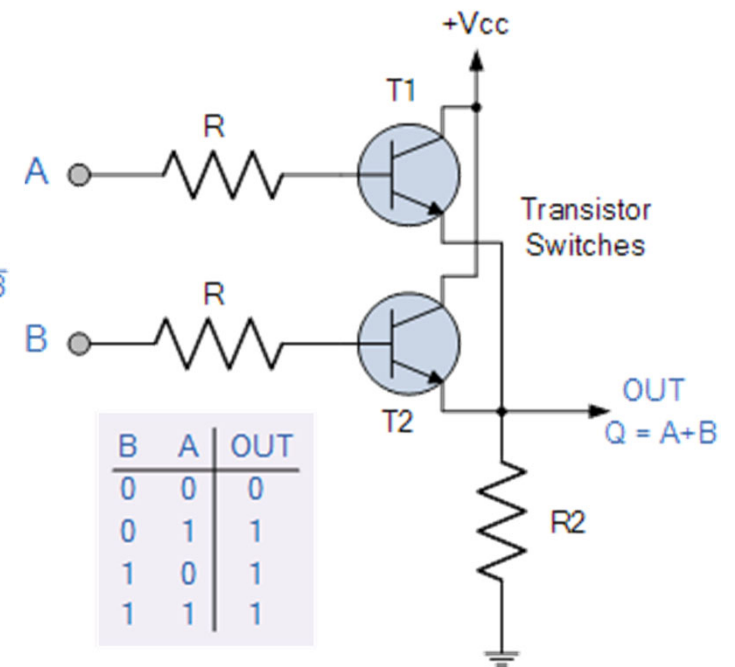
Eksempler: AND, NAND, OR



AND

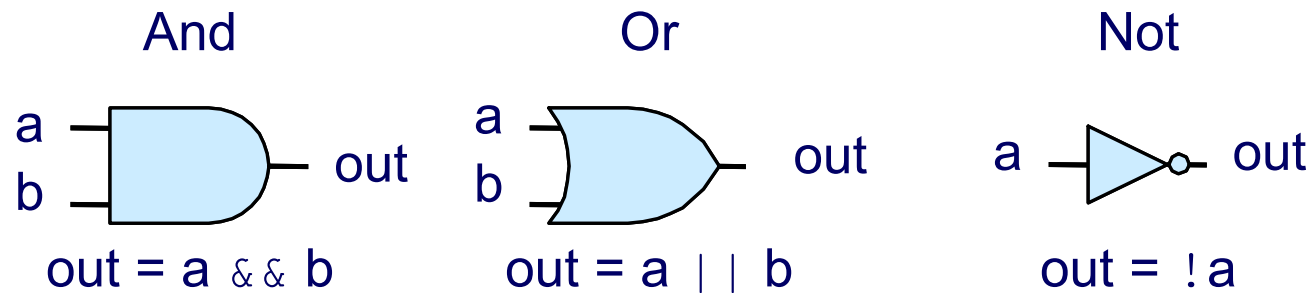


NAND



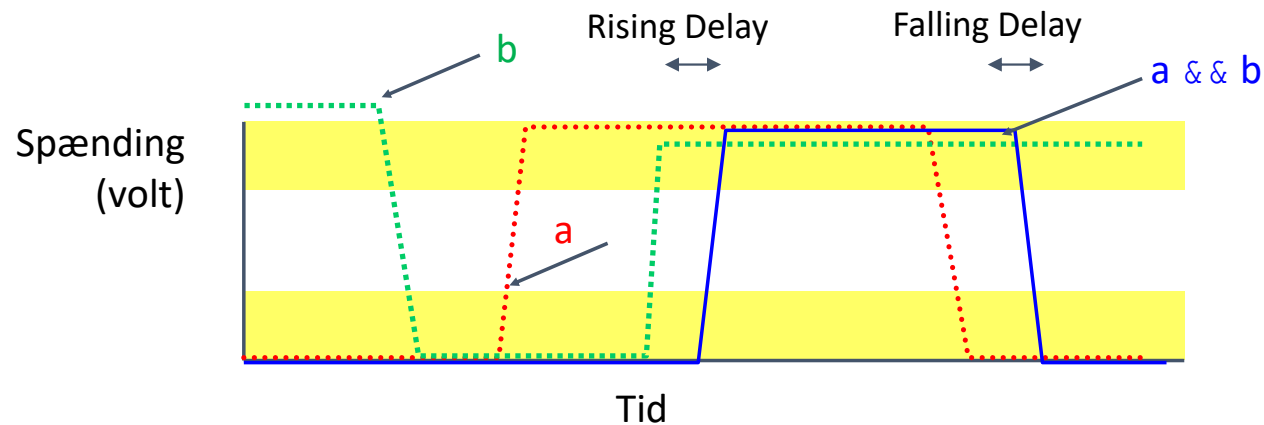
OR

Beregning med Logik "Gates"

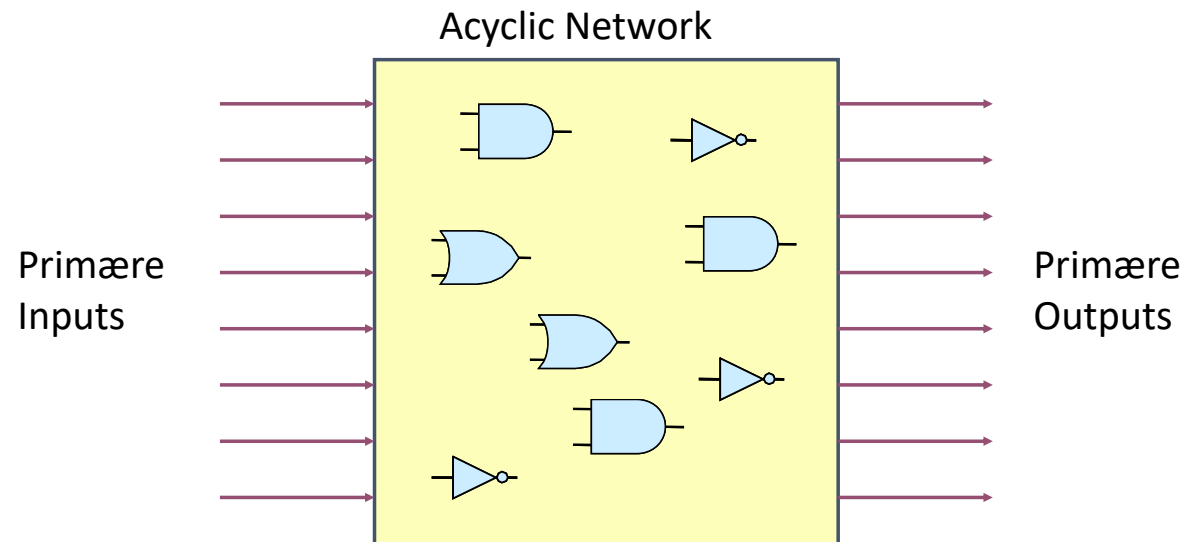


Syntax lånt fra de logiske operationer i C

- Output er en **Boolsk funktion** af input
- Svarer **kontinuerligt** på ændringer af inputs
 - Med en lille forsinkelse

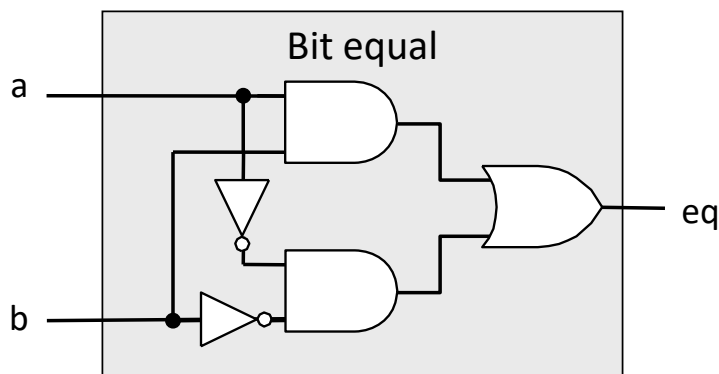


Kombinatoriske (Digitale) Kredsløb



- **Acyclic Netværk af Logiske Gates**
 - Reagerer kontinuerligt på ændringer i primære inputs
 - Signal udbredes igennem netværket
 - Primære outputs bliver (efter en forsinkelse) en Boolsk funktion af primære inputs
- Modsætning: **Sekventielle digitale kredsløb**: indeholder cykler og hukommelselementer/registre

Bit Sammenligning



Sandhedstabel for eq

A	B	A == B
0	0	1
0	1	0
1	0	0
1	1	1

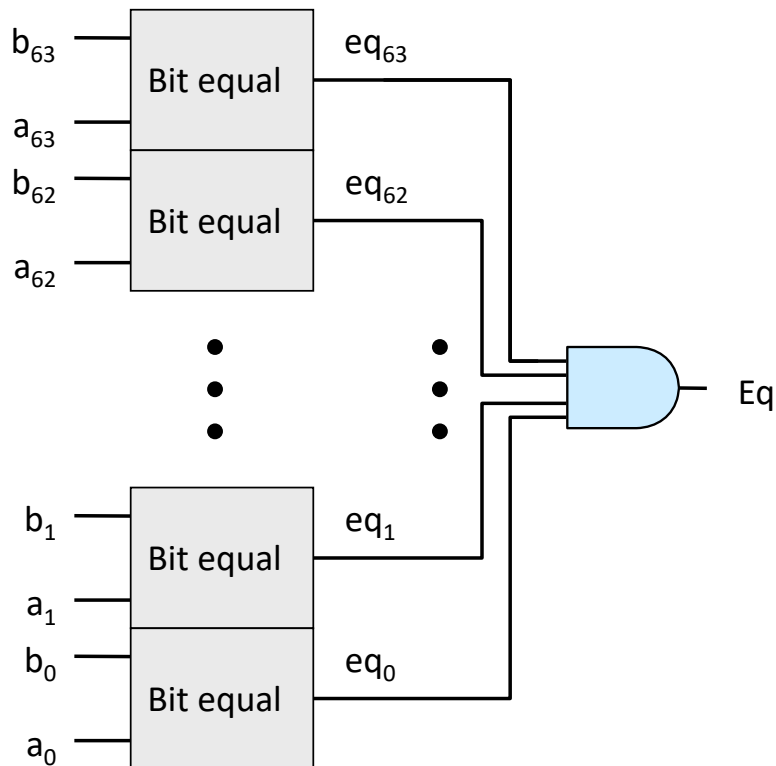
HCL Expression

```
bool eq = (a&&b) || (!a&&!b)
```

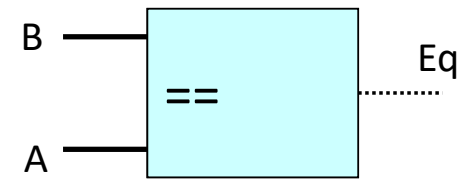
- Giver 1 hvis a og b er ens
- Hardware Control Language (HCL)
 - Meget simpelt hardware beskrivelsessprog
 - Boolske operationer bruger syntax, der ligner C's logiske operationer
 - Her anvendt til at beskrive kontrol logik for processorer
- Moderne digital logik/IC design:
 - Funktionalitet beskrives, simuleres, verificeres i sådanne sprog
 - Synteseres ("compileres") til kredsløb (fx. Application Specific Integrated Circuit (ASIC) eller Field Programmable Gate Array FPGA)
 - VHDL / Verilog

Sammenligning af ord: 64-bit ord A,B

$A = a_0 \dots a_{63}$
 $B = b_0 \dots b_{63}$



Ord-niveau Diagram
Repræsentation



HCL Udtryk

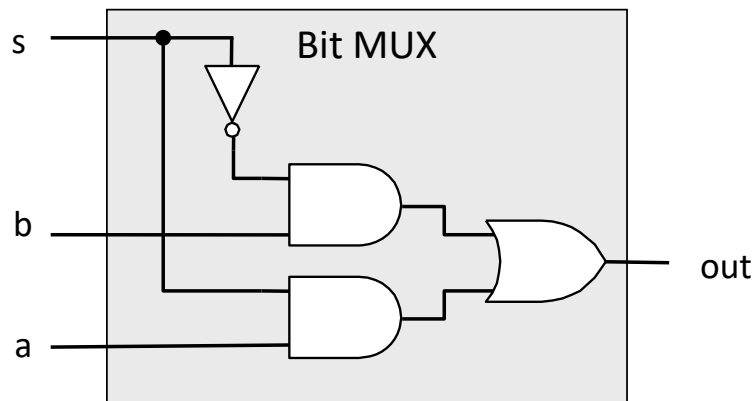
```
bool Eq = (A == B)
```

- 64-bit ord størrelse
- HCL repræsentation
 - Ligheds operator
 - Giver en Boolsk værdi

Princip: opbygning af komplekse kredsløb ved sammensætning af mange simple

Bit-Niveau Udvælger (Multiplekser)

Kontrol bit til
seleksion



A	B	S	A MUX B
0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1

HCL Udtryk

```
bool out = (s&&a) || (!s&&b)
```

- Udvælger vha. kontrol signal s blandt data-signaler a and b
- Output a når s=1, b når s=0

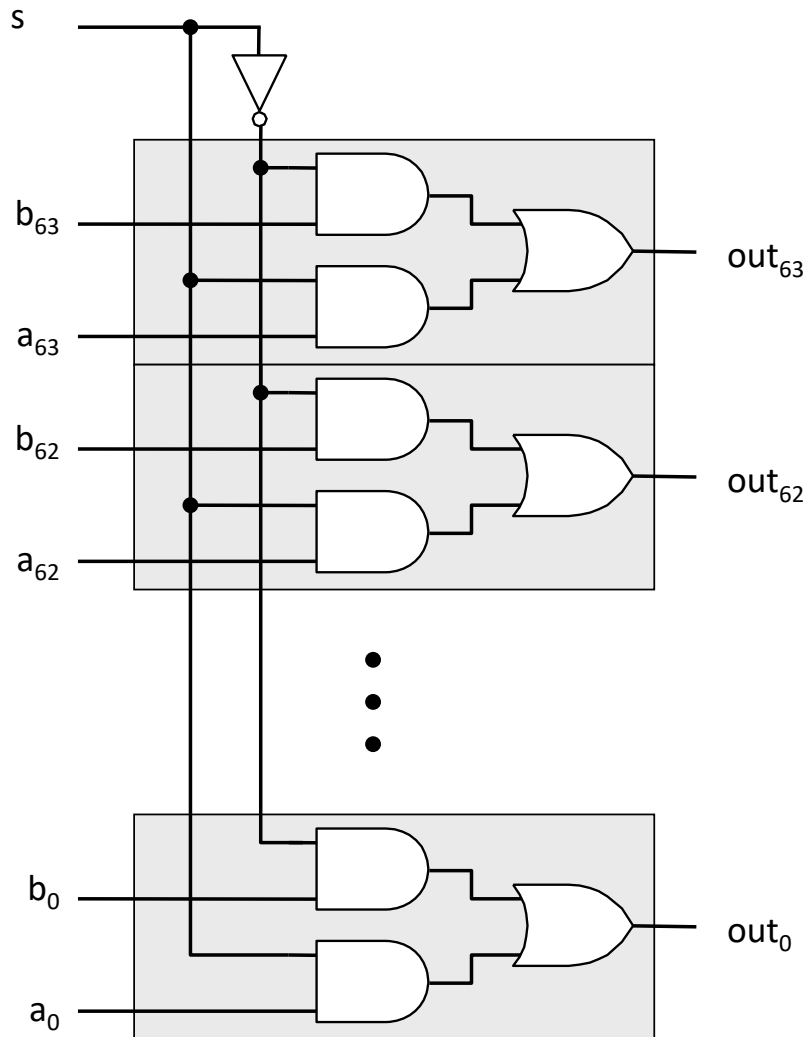
Enhed, som samler to eller flere datasignaler og samler dem til et enkelt signal eller udtryk

Ord Niveau Udvælger (Multiplexer)

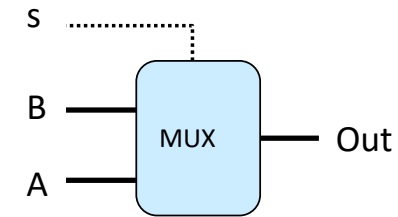
64 bit ord A,B

$A = a_0 \dots a_{63}$

$B = b_0 \dots b_{63}$



Ord-Niveau Diagram Symbol



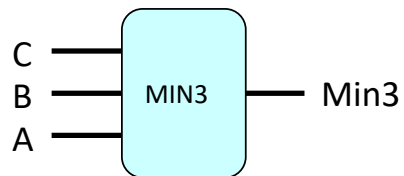
HCL Repræsentation

```
int Out = [  
    s : A;  
    1 : B;  
];
```

- Udvælger input ord A eller B afhængigt af kontrol signal s
- HCL repræsentation
 - **Case udtryk a la C-switch**
 - Serie af tests: par af betingelse:værdi
 - Output:
 - værdien for første sande test
 - Afsluttende "1" angiver "Default"

HCL Ord-Niveau Eksempler

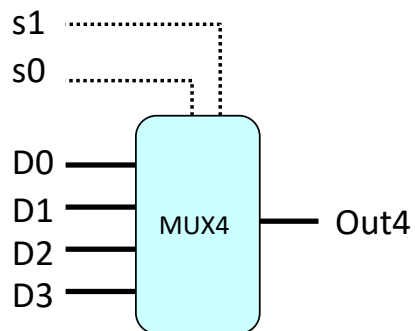
Minimum af 3 ords



```
int Min3 = [  
    A < B && A < C : A;  
    B < A && B < C : B;  
    1               : C;  
];
```

- Find minimum of tre input ord
- HCL case udtryk
- Sidste case garanterer match

4-Vejs Multiplexer

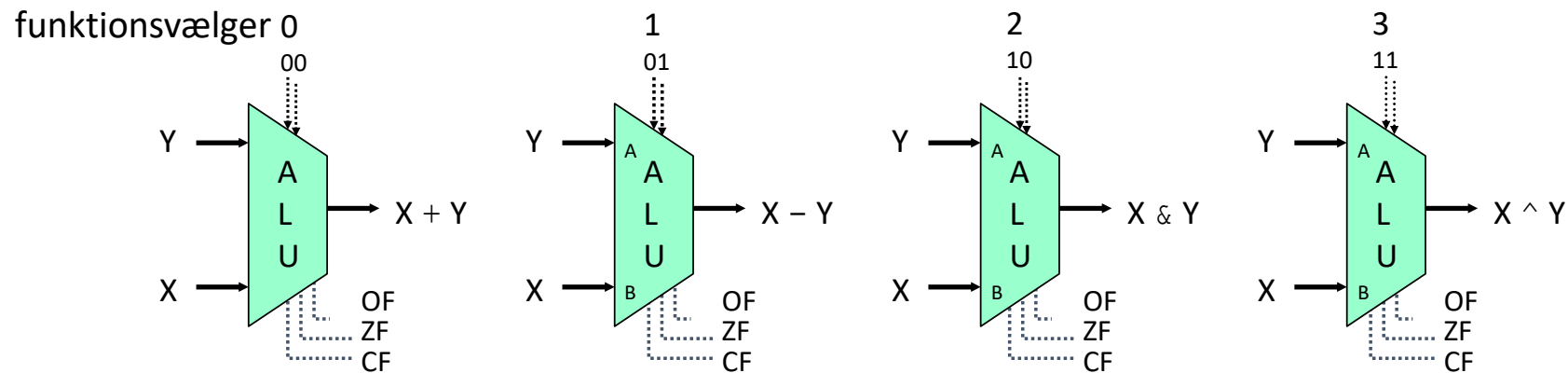


```
int Out4 = [  
    !s1&&!s0: D0;  
    !s1      : D1;  
    !s0      : D2;  
    1        : D3;  
];
```

- Udvælger 1 af 4 inputs baseret på to kontrol bits
- HCL case udtryk
- Simplificere tests ved at antage **sekventiel matchning** (a la switch i C)

Aritmetisk Logisk Enhed

Challenge 6: addér to bytes!

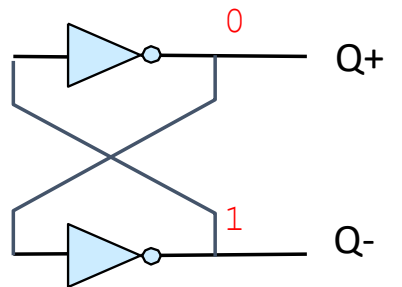


- Arithmetic Logic Unit (ALU)
- Kombinatorisk logik
 - Reagerer kontinuerligt på inputs
- Kontrol signal udvælger den ønskede funktion
 - Svarende til 4 aritmetiske/logiske operationer i Y86-64
- Beregner også værdier for “condition codes”

Digitale sekventielle kredsløb

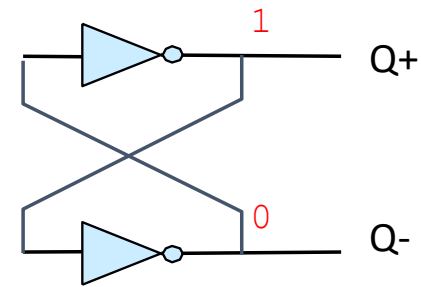
Med hukommelse!

Lagring af 1 Bit: Bi-stabilt Element

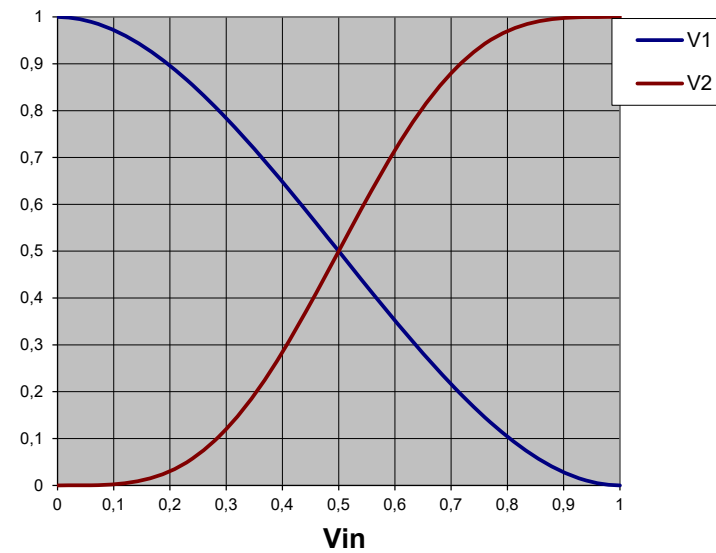
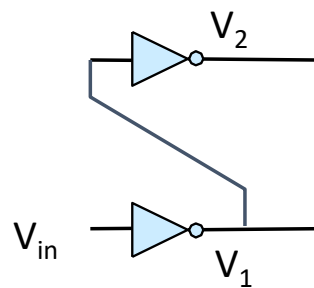


Stabil tilstand 1

Bistabilt Element

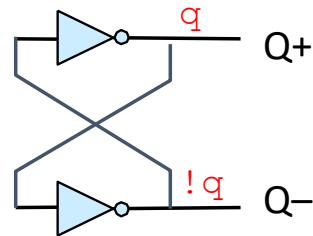


Stabil tilstand 2

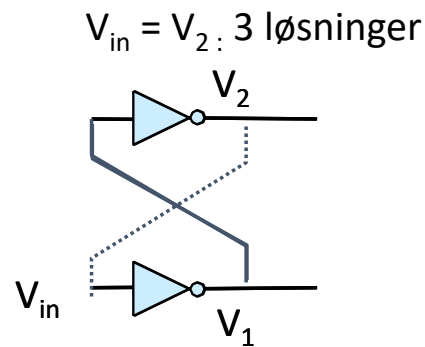


Lagring af 1 Bit (fortsat.)

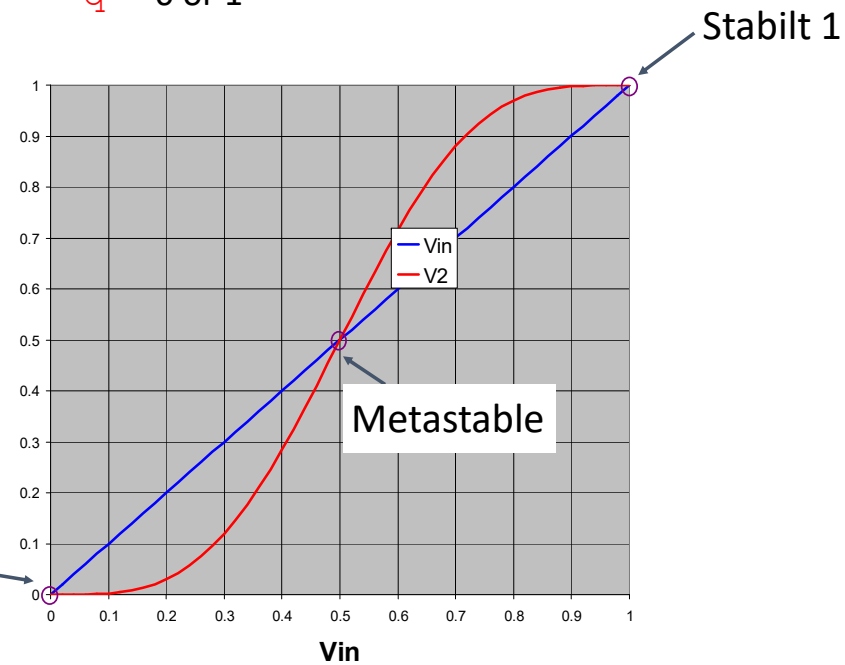
Bistabilt Element



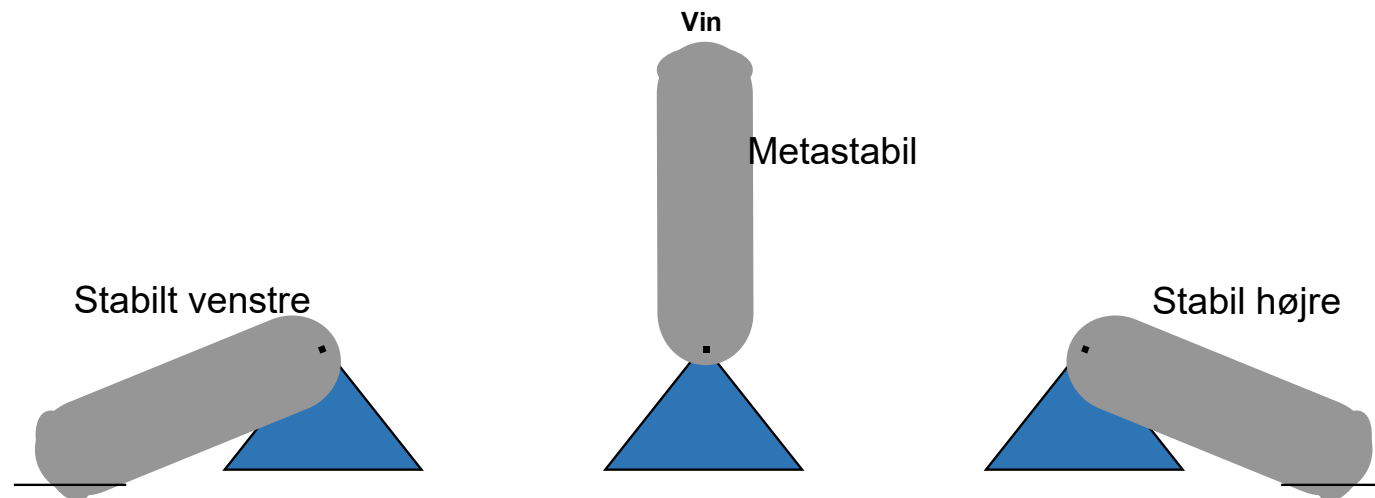
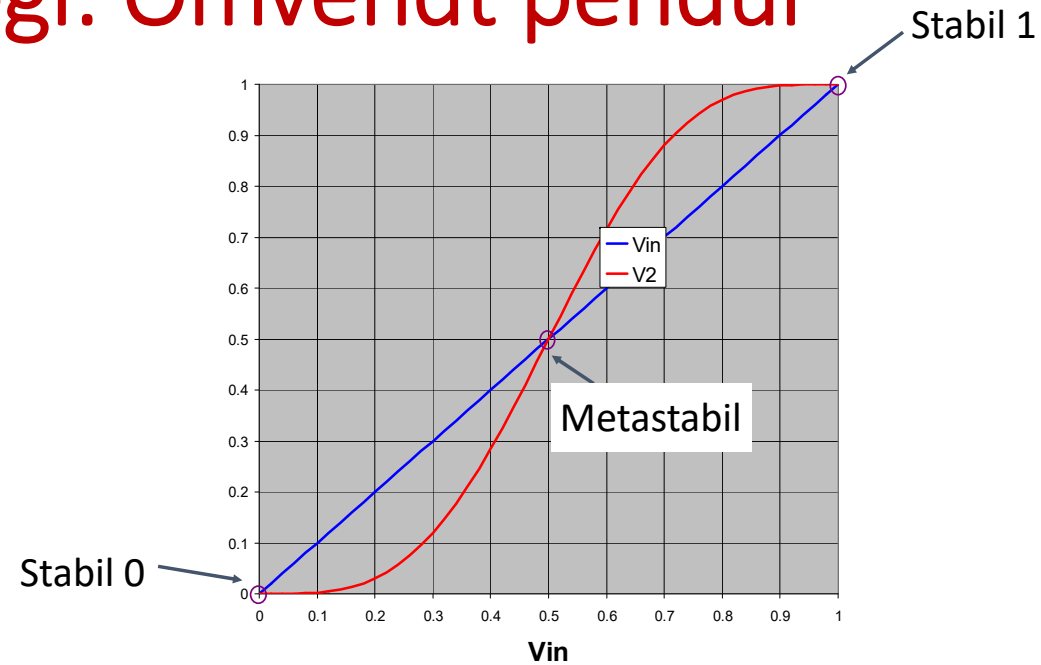
$q = 0$ or 1



Stabilt 0



Fysik Analogi: Omvendt pendul

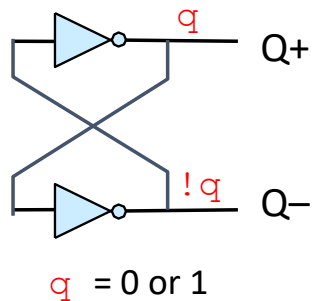


Lagring og aflæsning af 1 Bit

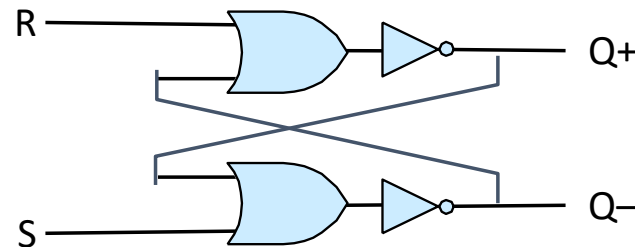
Reset-set lås

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0

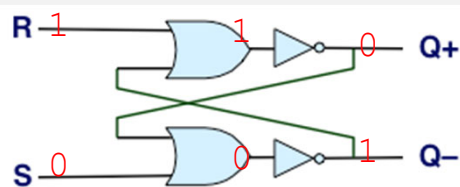
Bistabil Element



R-S Latch

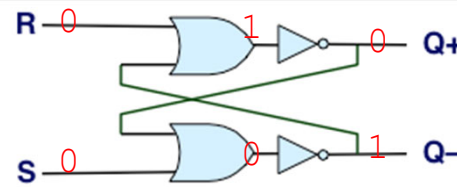


Resetting (skrive 0)



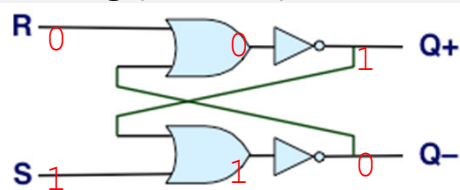
nulstil R

Lager



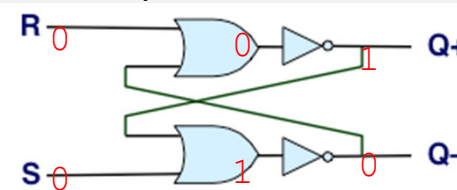
Skrevne
0 bit
bevares

Setting (skrive 1)



nulstil S

Memory



Skrevne
1 bit
bevares

R=1, S=1: Forbudt!

Hardware Register

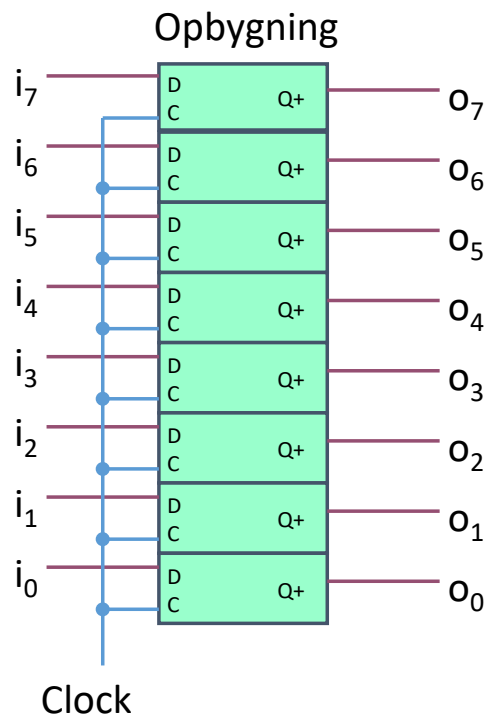
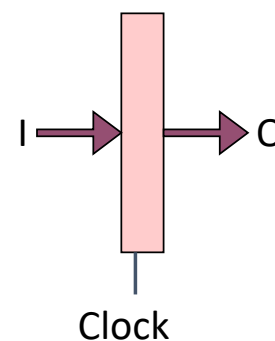
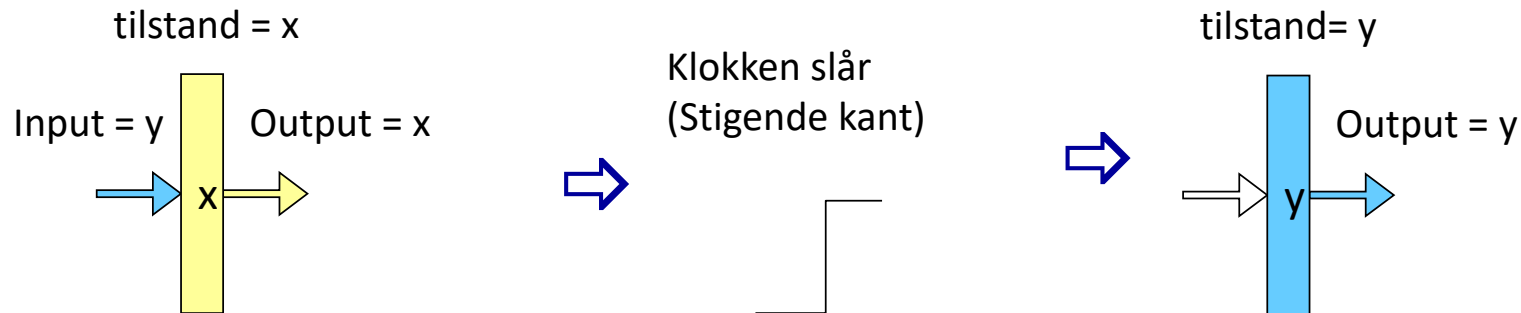


Diagram symbol

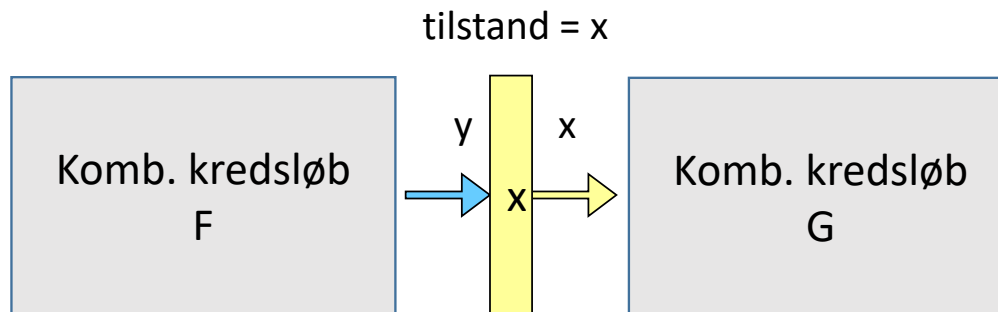


- Gemmer et data-ord
 - Processorer anvender mange sådanne hardware registre internt
- Samling af 1-bit latches, der aktiveres med et clock signal
- Ændres kun når “klokken slår”

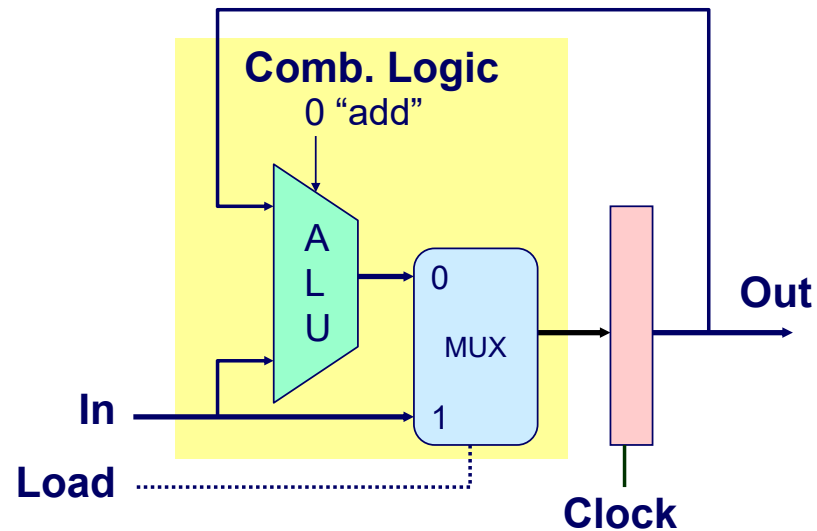
Register Styring



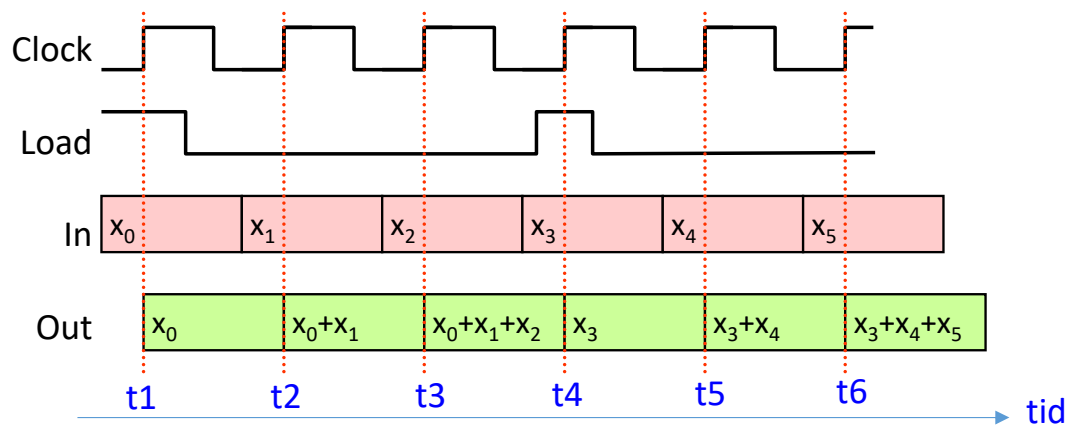
- Gemmer data bits y fra input, og bits fra x findes på output
- Indskrives input når klokken stiger "rising edge triggered"
- Fungerer som barriere, som separerer ændringer på input siden fra output
- Output forbliver stabilt selvom input ændres



Tilstandsmaskine Eksempel



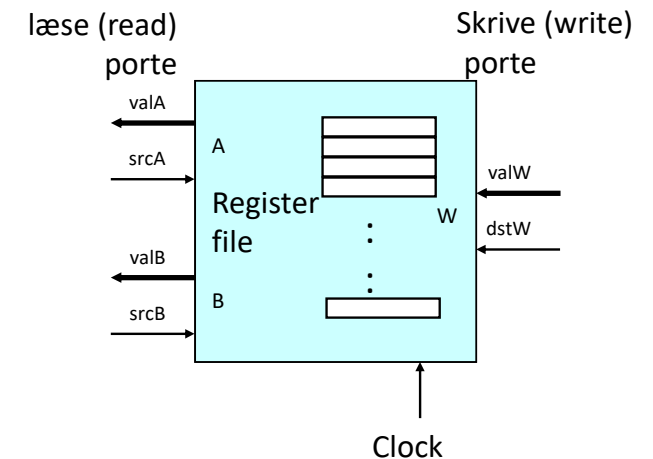
- Akkumulator kredsløb
- Indlæs eller addere i hver cyklus
- Clock= elektrisk signal, der svinger med fast periode tid = takt = clock frekvens



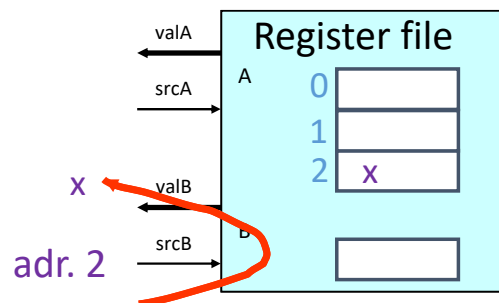
Load bestemmer om register værdi
tages fra In eller ALU ud
Load=1: gem ny data
Load=0: gem adderet resultat

Random-Access Memory

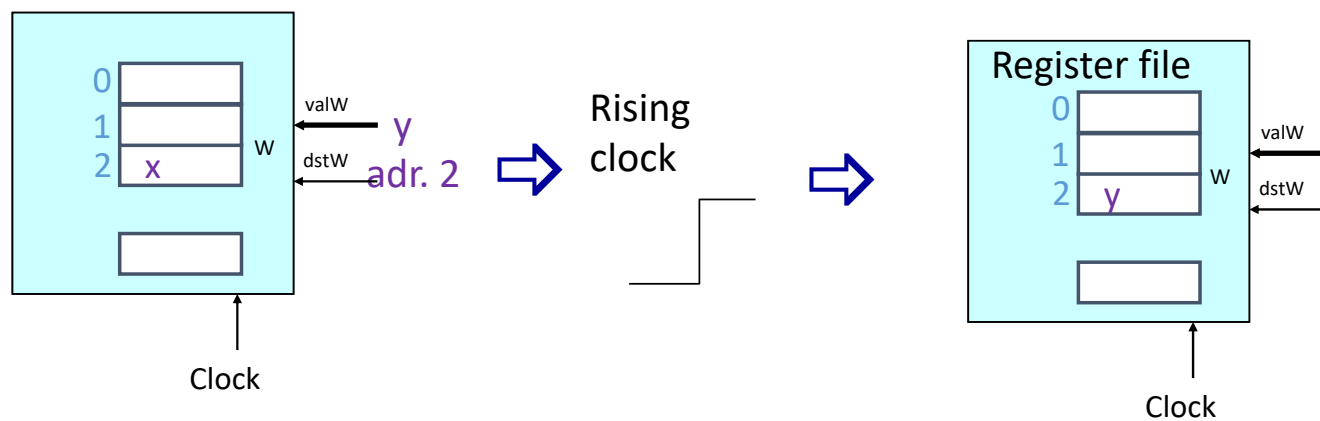
- Kredsløb, der kan gemme et antal ord
 - Adresse input (srcA, srcB, dstW) bestemmer hvilket ord som læses eller skrives
- Register bank (register file)
 - Gemmer værdier af program registre
 - %rax, %rsp, etc.
 - Register-id fungerer som adresse
- Flere Porte
 - Kan læse eller skrive flere ord i en cyklus
 - Hver port har sin egen adresse og data input/output



Register Bank Timing



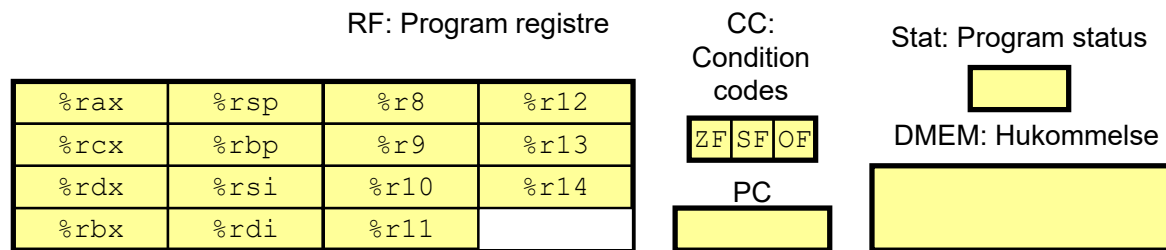
- Læsning
 - Fungerer som kombinatorisk logik
 - Output data genereres udfra input adressen
 - Efter lille forsinkelse
- Skrivning
 - Som hardware register
 - Updateres kun når klokken slår



Y86-SEQ processoren

- Forenklet og destilleret udgave af X86 med sekventiel instruktions-behandling
- Færre og enklere instruktioner
- Adfærd af hardware logik er udtømmende beskrevet!

Y86-64 Processor Programmør Synlig Tilstand



- Program Register (RF: Register File)
 - 15 registre (%r15 udeladt). Hver på 64 bits
- Condition Codes
 - Enkelt-bit flag, der sættes af aritmetiske eller logiske instruktioner
 - ZF: Zero SF: Negative OF: Overflow
- Program Tæller (PC, %rip)
 - Angiver adresse på næste instruktion
- Program Status
 - Flag, der angiver enten normal operation eller fejl-situation
- Hukommelsen
 - Byte-adressérbar array
 - Ord gemmes i little-endian orden
 - DMEM: Dynamic Memory

Y86-64 Instruktions-Sæt #1: Instruktions-kode

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
"No-operation" nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
rmrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Indkodning!

1–10 bytes læses fra hukommelsen

- Eksakte længde kan udledes fra første byte
- Første byte=instruktionsspecifikation
 - 4 øverste bits = Instruktions-kode
 - 4 nederste bits = Funktions-kode

Y86-64 Instruktions-Sæt #2:Funktions-kode

Byte	0	1	2	3	4	5	6	7	8	9	
									rrmovq	2 0	
halt	0	0							cmovle	2 1	
nop	1	0							cmovl	2 2	
cmovXX rA, rB	2	fn	rA	rB					cmove	2 3	
irmovq V, rB	3	0	F	rB	V				cmovne	2 4	
rmmovq rA, D(rB)	4	0	rA	rB	D				cmovge	2 5	
mrmovq D(rB), rA	5	0	rA	rB	D				cmovg	2 6	
OPq rA, rB	6	fn	rA	rB							
jXX Dest	7	fn	Dest								
call Dest	8	0	Dest								
ret	9	0									
pushq rA	A	0	rA	F							
popq rA	B	0	rA	F							

Y86-64 Instruktions-Sæt #3

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

addq

subq

andq

xorq

Y86-64 Y86-64 Instruktions-Sæt #4

Byte	0	1	2	3	4	5	6	7		
									jmp	<div>7</div> <div>0</div>
halt	<div>0</div> <div>0</div>								jle	<div>7</div> <div>1</div>
nop	<div>1</div> <div>0</div>								j1	<div>7</div> <div>2</div>
cmovXX rA, rB	<div>2</div>	<div>fn</div>	<div>rA</div>	<div>rB</div>					je	<div>7</div> <div>3</div>
irmovq V, rB	<div>3</div>	<div>0</div>	<div>F</div>	<div>rB</div>	V				jne	<div>7</div> <div>4</div>
rmmovq rA, D(rB)	<div>4</div>	<div>0</div>	<div>rA</div>	<div>rB</div>	D				jge	<div>7</div> <div>5</div>
mrmovq D(rB), rA	<div>5</div>	<div>0</div>	<div>rA</div>	<div>rB</div>	D				jg	<div>7</div> <div>6</div>
OPq rA, rB	<div>6</div>	<div>fn</div>	<div>rA</div>	<div>rB</div>						
jXX Dest	<div>7</div>	<div>fn</div>	Dest							
call Dest	<div>8</div>	<div>0</div>	Dest							
ret	<div>9</div> <div>0</div>									
pushq rA	<div>A</div>	<div>0</div>	<div>rA</div>	<div>F</div>						
popq rA	<div>B</div>	<div>0</div>	<div>rA</div>	<div>F</div>						

Indkodning af Register

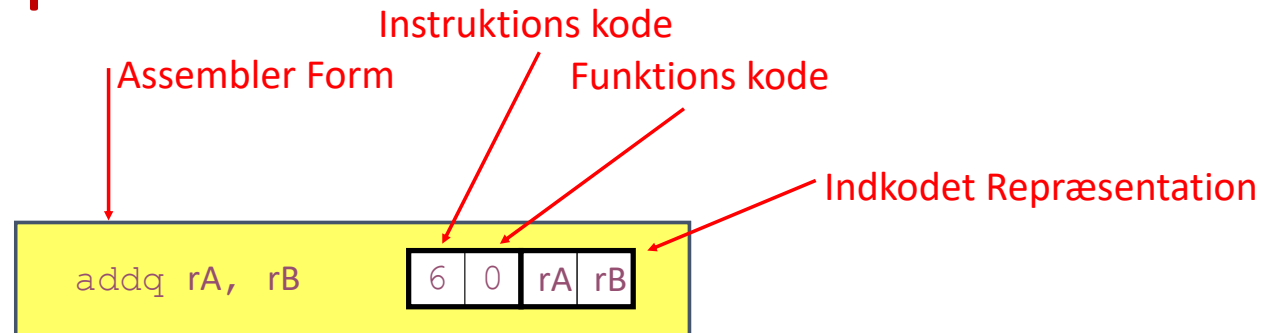
- Hvert register har 4-bit ID

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

- Samme som i x86-64
- Register ID 15 (0xF) indikerer “intet register”
 - Bruges i hardware beskrivelsen flere steder

Eksempel på Y86 instruktion

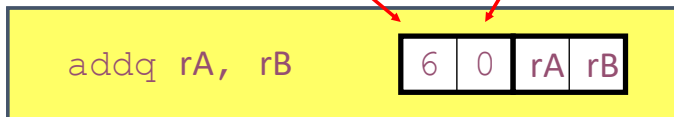
- Addition



- Addér værdi i register rA til værdien i register rB
 - Gem resultat i register rB
 - Begrænsning: Y86-64 tillader kun addition på register data
- Sætter condition codes afhængigt af resultatet
- F.x., `addq %rax, %rsi` Indkodning: 0x 60 06
- Indkodning fylder 2 bytes
 - Første byte angiver instruktions type
 - Anden byte angiver source og destinations registre

Arithmetiske og Logiske Operationer

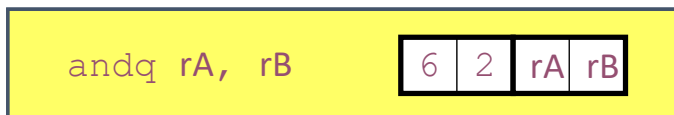
Instruktions kode Funktions kode
Add



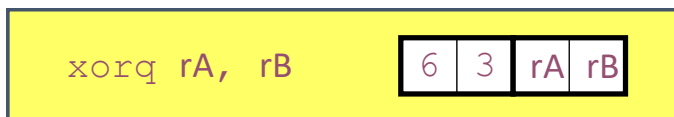
Subtract (rA from rB)



And



Exclusive-Or



- Generisk benævnt “OPq”
- Indkodning varierer kun i “funktions koden”
 - Low-order 4 bytes in first instruction word
- Sætter condition codes som side effekt

Move Operationer

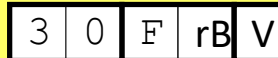
Register → Register

`rrmovq rA, rB`



Immediate → Register

`irmovq V, rB`



Register → Memory

`rmmovq rA, D(rB)`



Memory → Register

`mrmovq D(rB), rA`



- A la x86-64 `movq` instruktionen: `movq D(Rb, Ri, s), Rd`
- Simplere format for hukommelses adresser
- Navngivet forskelligt for tydeliggøre forskel

Move Instruktions Eksempler

X86-64

```
movq $0xabcd, %rdx
```

Indkodning:

```
movq %rsp, %rbx
```

Indkodning:

```
movq -12(%rbp), %rcx
```

Indkodning:

```
movq %rsi, 0x41c(%rsp)
```

Indkodning:

Y86-64

```
irmovq $0xabcd, %rdx
```

30 F2 cd ab 00 00 00 00 00 00

```
rrmovq %rsp, %rbx
```

20 43

```
mrmovq -12(%rbp), %rcx
```

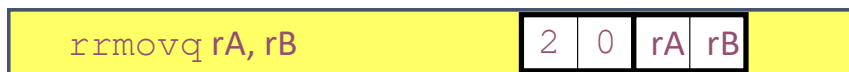
50 15 f4 ff ff ff ff ff ff ff

```
rmmovq %rsi, 0x41c(%rsp)
```

40 64 1c 04 00 00 00 00 00 00

Conditional Move Instruktionser

Move Unconditionally



Move When Less or Equal



Move When Less



Move When Equal



Move When Not Equal



Move When Greater or Equal



Move When Greater



- Genersik benævnt “`cmovXX`”
- Alle laver register->register flytning
- Indkodning bruger forskellig “function code”
- Baseret på værdier af condition codes
- Bemærk `rrmovq` instruktionen
 - register->register move
 - Ubetinget kopiering af værdi fra source til destination register

Jump Instruktioner

Jump (Conditionally)



- Generisk benævnt “jXX”
- Indkodning adskiller sig med “function code” fn
- Baseret på værdier af condition codes
- Samme som condition codes på x86-64
- Indkoder den fulde destinations adresss
 - Modsat X86-64, som benytter PC-relativ adressering

Jump Instruktioner

Jump Unconditionally

<code>jmp Dest</code>	7	0	Dest
-----------------------	---	---	------

Jump When Less or Equal

<code>jle Dest</code>	7	1	Dest
-----------------------	---	---	------

Jump When Less

<code>jl Dest</code>	7	2	Dest
----------------------	---	---	------

Jump When Equal

<code>je Dest</code>	7	3	Dest
----------------------	---	---	------

Jump When Not Equal

<code>jne Dest</code>	7	4	Dest
-----------------------	---	---	------

Jump When Greater or Equal

<code>jge Dest</code>	7	5	Dest
-----------------------	---	---	------

Jump When Greater

<code>jg Dest</code>	7	6	Dest
----------------------	---	---	------

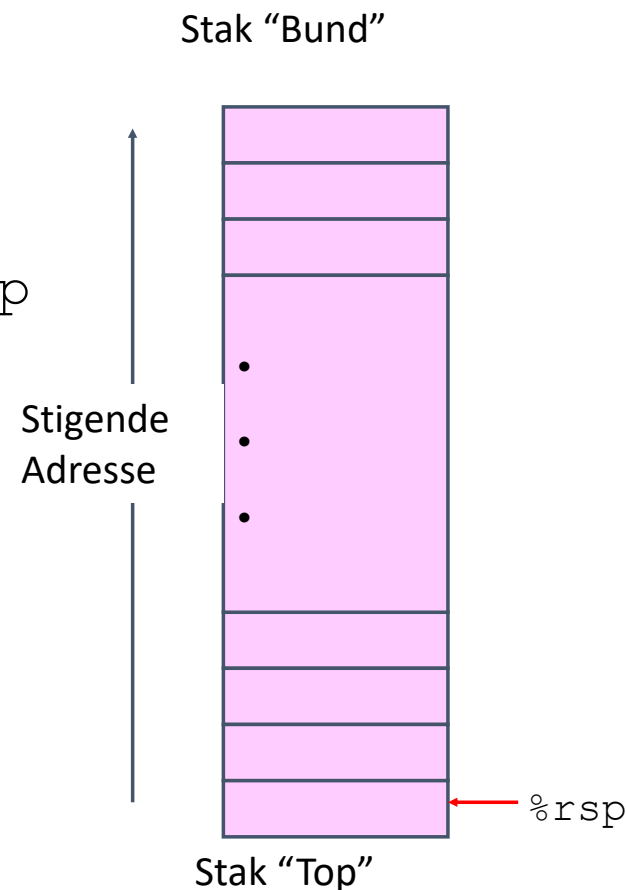
Stak Operationer



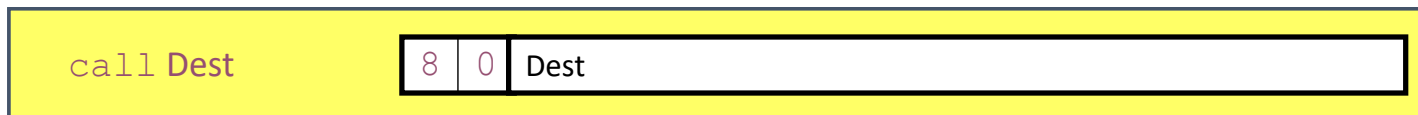
- Dekrementér `%rsp` med 8
- Gemmer ord fra `rA` i hukommelsen på adresse angiver i `%rsp`
- A la x86-64



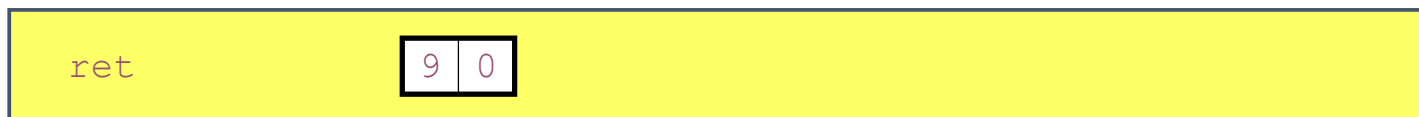
- Læser ord i hukommelsen på adresse angivet i `%rsp`
- Gemmer det i `rA`
- Inkrementér `%rsp` med 8
- A la x86-64



Procedure Kald og Retur



- Skubber adresse på næste instruktion på stakken
- Starter udførelsen af instruktioner på Dest adresse
- A la x86-64



- Afstakker værdi
- Bruges som adresse for næste instruktion (retur adresse)
- A la x86-64

Diverse Instruktioner

`nop`

1	0
---	---

- Udfør ingenting
- “No-operation”

`halt`

0	0
---	---

- Stop udførelsen af instruktioner
- x86-64 har tilsvarende instruktion, men kan ikke udføres i “user-mode”
- Vores anvendelse: stop simulator
- Nul-indkodningen sikrer, at programmet stopper når det rammer hukommelse initialiseret med nuller

Status Koder

Mnemo-kode	Code
AOK	1

- Normal kørsel

Mnemonic	Code
HLT	2

- Halt instruktion udført

Mnemonic	Code
ADR	3

- Adresseringsfejl (enten efter instruktion eller data) indtruffet

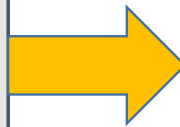
Mnemonic	Code
INS	4

- Invalid instruktion forsøgt udført (Bitmønster giver ikke gyldig instruktion)

- Ønsket adfærd
 - Hvis alt er OK, fortsæt kørsel
 - Ellers, stop program kørsel

Y86-64 Eksempel på program Struktur #1

```
/* Find number of elements in
   null-terminated list */
long len(long a[])
{
    long len;
    for (len = 0; a[len]; len++)
        ;
    return len;
}
```



```
init:                                # Initialization
    . . .
    call Main
    halt

    .align 8                          # Program data
array:
    . . .

Main:                                # Main function
    . . .
    call len    . . .

len:                                  # Length function
    . . .

    .pos 0x100                        # Placement of stack
Stack:
```

- Ingen C compiler til Y86
 - Program starter på adresse 0
 - Skal selv opsætte stak
 - Placering i hukommelsen
 - Opsæt pointers
 - Må ikke overskrive kode!
 - initializér øvrigtdata
- Generer X86 ASM med gcc; oversæt manuelt til Y86

Y86-64 Eksempel på program Struktur #2

```
init:
    # Set up stack pointer
    irmovq Stack, %rsp
    # Execute main program
    call Main
    # Terminate
    halt

# Array of 4 elements + terminating 0
    .align 8
Array:
    .quad 0x000d000d000d000d
    .quad 0x00c000c000c000c0
    .quad 0x0b000b000b000b00
    .quad 0xa000a000a000a000
    .quad 0
```

- Program starter på adresse 0
- Skal selv opsætte stak
 - Placering i hukommelsen
 - Opsæt pointers
 - Må ikke overskrive kode!
- initializér øvrigt data
- Kan bruge symbolske navne

Y86-64 Eksempel på program Struktur #3

```
Main:
    irmovq array,%rdi
    # call len(array)
    call len
    ret
```

- Opsæt kald til `len`
 - Følg x86-64 procedurekaldskonventioner
 - Overfør adresse på `array` som argument

Assemblering af et Y86-64 Program

```
unix> yas len.ys
```

- Genererer objekt kode filen `len.yo`
 - Ligner disassembler output

```
0x054: | len:
0x054: 30f8010000000000000000 |   irmovq $1, %r8      # Constant 1
0x05e: 30f9080000000000000000 |   irmovq $8, %r9      # Constant 8
0x068: 30f0000000000000000000 |   irmovq $0, %rax     # len = 0
0x072: 5027000000000000000000 |   mrmovq (%rdi), %rdx  # val = *a
0x07c: 6222 |   andq %rdx, %rdx     # Test val
0x07e: 73a00000000000000000 |   je Done             # If zero, goto Done
0x087: | Loop:
0x087: 6080 |   addq %r8, %rax      # len++
0x089: 6097 |   addq %r9, %rdi      # a++
0x08b: 5027000000000000000000 |   mrmovq (%rdi), %rdx  # val = *a
0x095: 6222 |   andq %rdx, %rdx     # Test val
0x097: 74870000000000000000 |   jne Loop            # If !0, goto Loop
0x0a0: | Done:
0x0a0: 90 |   ret
```


Simulering af Y86-64 Program

```
unix> yis len.yo
```

- Simulerer instruktionssættet
 - Beregner den effekt, som hver instruktion har på processor tilstand
 - Udskriver ændringer

```
Stopped in 33 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
```

```
Changes to registers:
```

%rax:	0x0000000000000000	0x0000000000000004
%rsp:	0x0000000000000000	0x0000000000000100
%rdi:	0x0000000000000000	0x0000000000000038
%r8:	0x0000000000000000	0x0000000000000001
%r9:	0x0000000000000000	0x0000000000000008

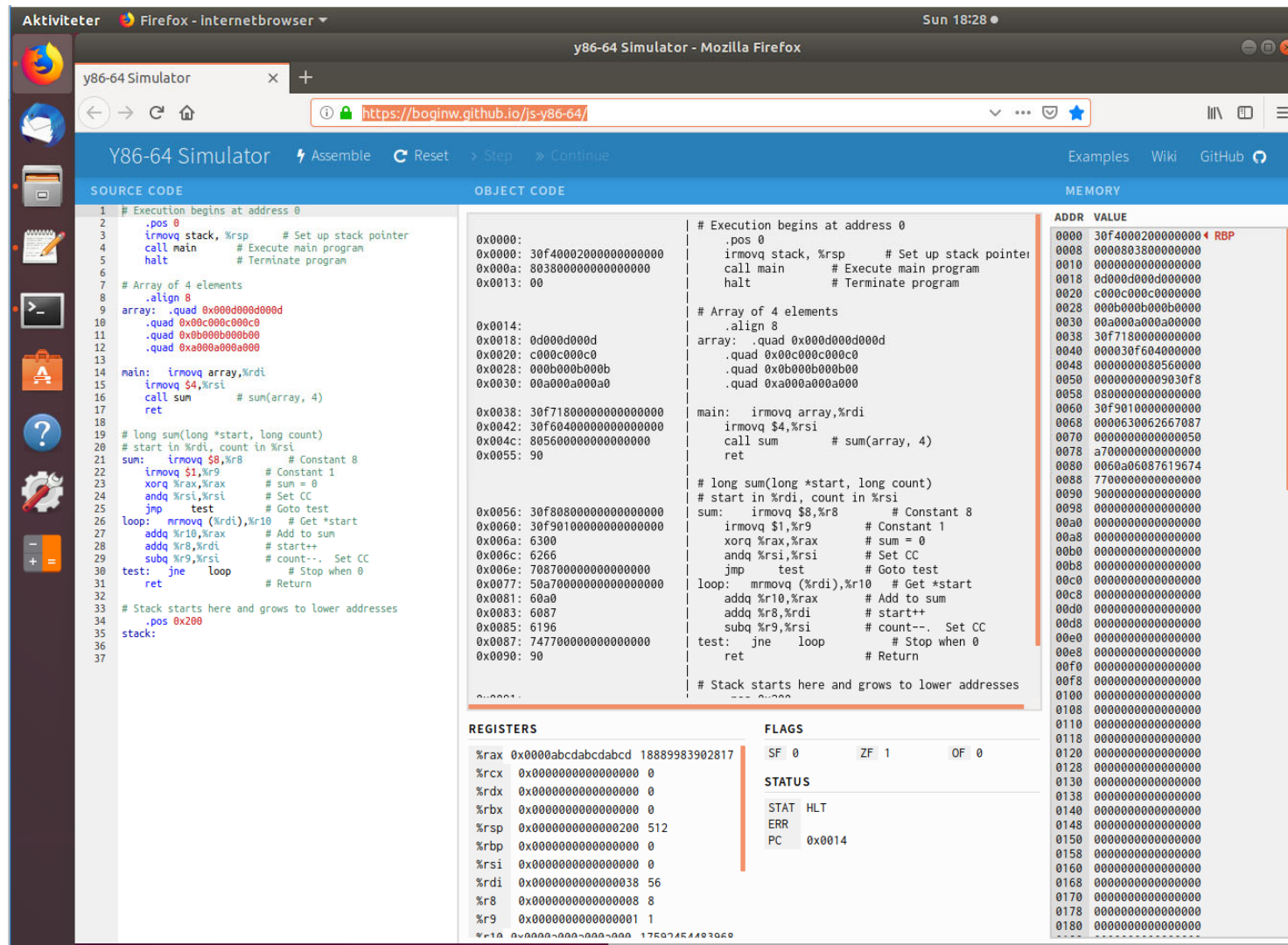
```
Changes to memory:
```

0x00f0:	0x0000000000000000	0x0000000000000053
0x00f8:	0x0000000000000000	0x0000000000000013

```
unix> ssim -g len.yo
```

GUI Version

Alternativ Simulering af Y86-64: <https://boginw.github.io/js-y86-64/>

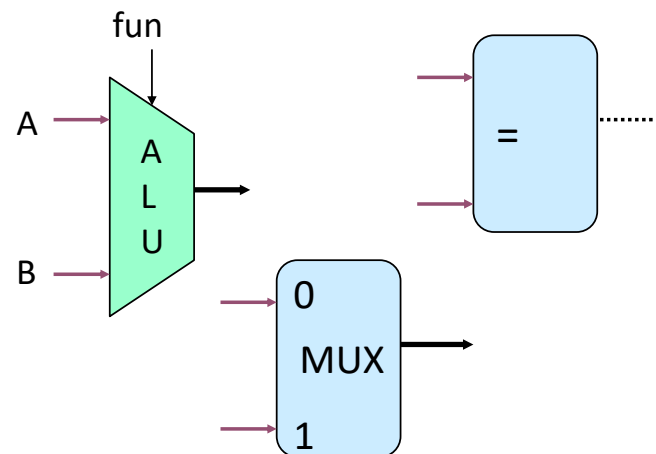


Hardware design af Y86-SEQ

Bygge Blokke

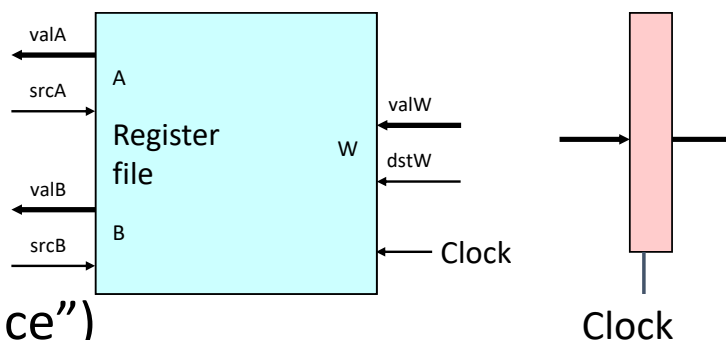
- Kombinatorisk Logik

- Beregner Boolske funktioner af inputs
- Reagerer kontinuerligt på ændringer af inputs
- Opererer på data og implementerer kontrol



- Hukommelses Elementer

- Gemmer bits
- Adressérbar hukommelse
- Hardware registre
- Skrives kun når clock stiger
- (Brug tidsstyring til at undgå samtidig R/W til samme register – "race")



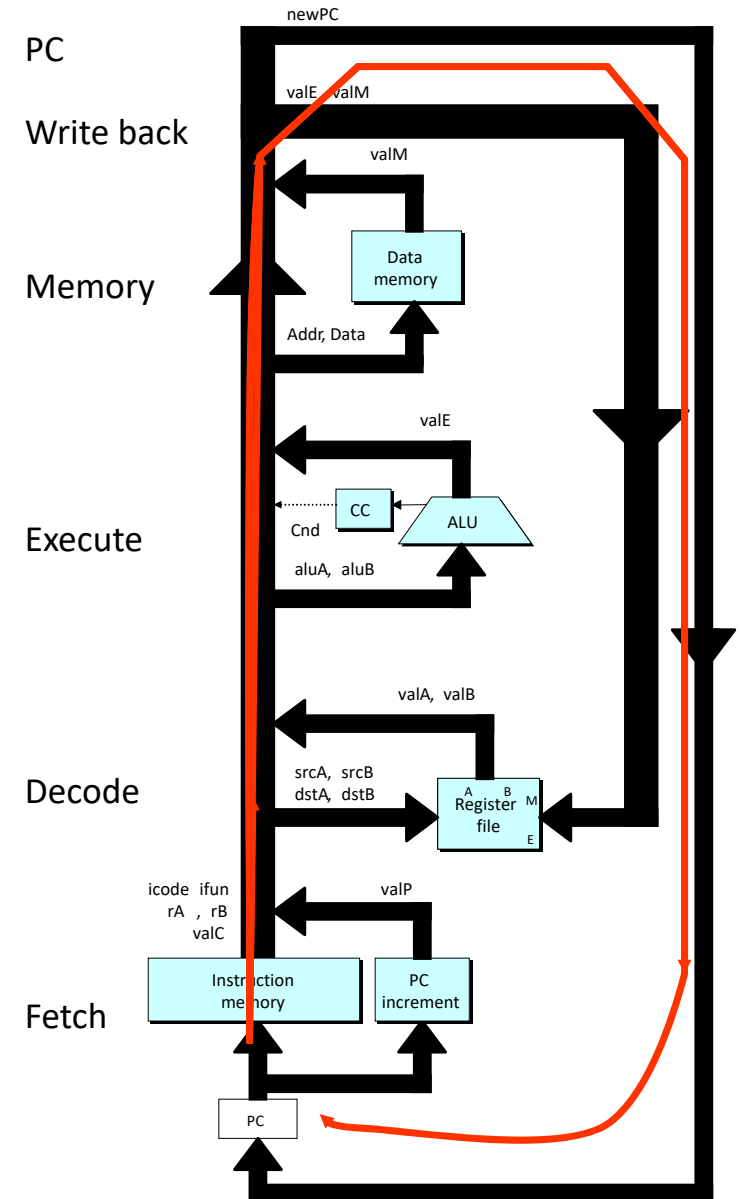
Model for SEQ-processor

- Tilstand

- Program counter register (PC)
- Condition code register (CC)
- Register Bank
- Hukommelse: én hukommelse, med logisk opdeling i
 - Data-hukommelse: til læsning/skrivning af program data
 - Instruktions-hukommelse: til læsning af instruktioner

- Instruktionsflow

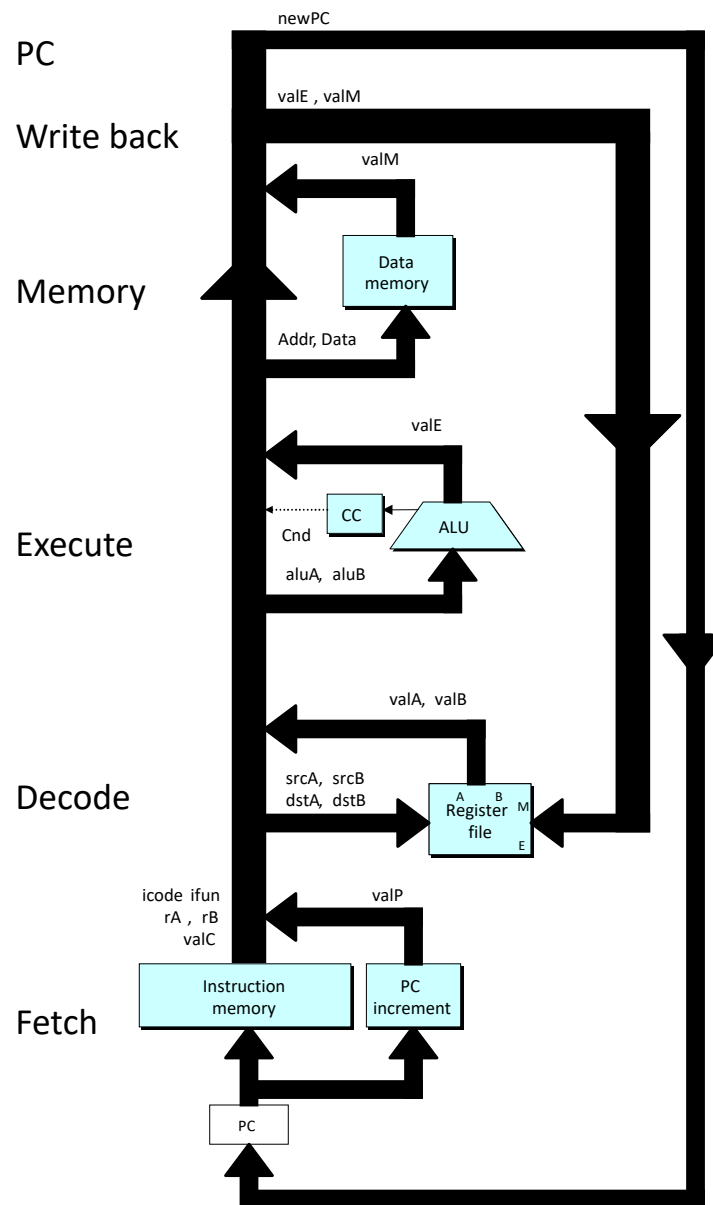
- Læs instruktion på adressen udpeget af PC
 - Behandl instruktionen i udførelsestrin
 - Opdatér program counter



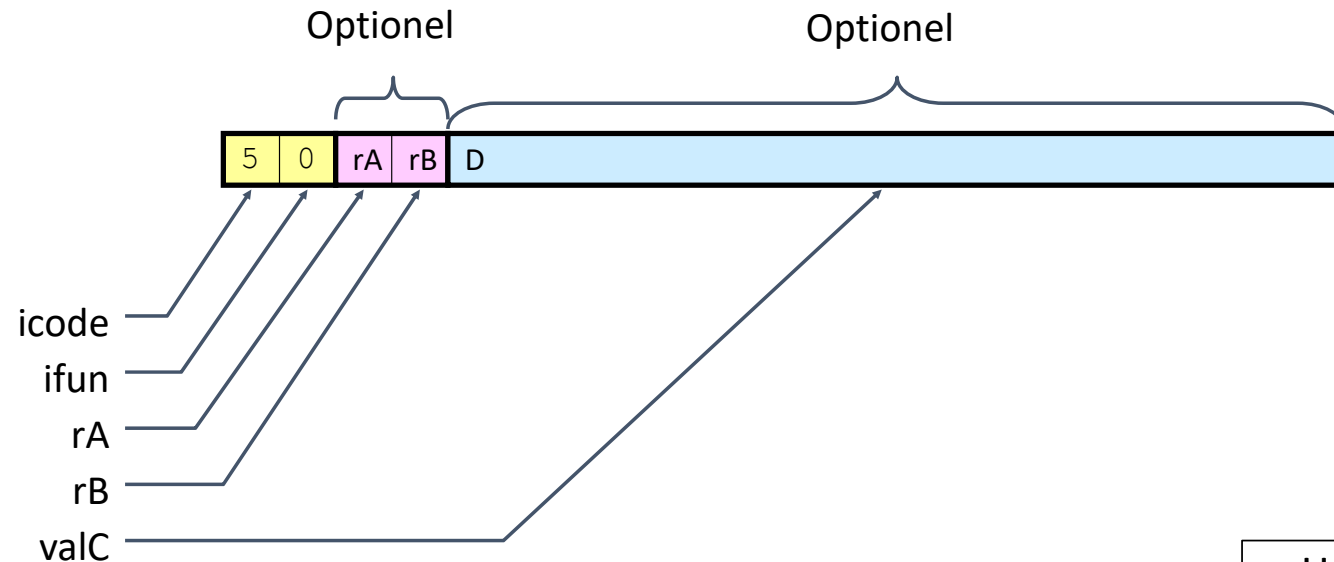
SEQ Trin

1. Fetch
 - Læs instruktion fra instruktionshukommelse
 - Beregn værdi for næste PC
2. Decode
3. Execute
 - Beregn værdi eller adresse
4. Memory
 - Læs eller skriv data i hukommelsen
5. Write Back
 - Skriv resultat til program registre
6. PC
 - Opdatér program tæller
7. Gentag fra 1.

BEMÆRK Signalnavne



Instruktions Dekoding



- Instruktions Format

- Instruktions byte
- Optionel register byte
- Optionel konstant ord

icode:ifun

rA:rB

valC, (fx D)

- Hvert register har 4-bit ID

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

Udførelse af Arit./Logisk Instruktion



$OP \in \{\text{add, sub, and, xor}\}$

1. Fetch
 - læs 2 bytes, beregn næste PC
2. Decode
 - Udlæs operand registre rA, rB
3. Execute
 - Udfør operation "OP"
 - Sæt condition codes
4. Memory
 - Intet at gøre
5. Write back
 - Opdatér register rB
6. PC Update
 - Sæt PC til beregnede næste PC

Udførelse af Arit./Log. Operationer

NB! Array notation til adgang til hukommelse og register bank

1. Fetch
 - læs 2 bytes, beregn næste PC
2. Decode
 - Læs operand registre
3. Execute
 - Udfør operation
 - Sæt condition codes
4. Memory
 - Intet at gøre
5. Write back
 - Opdatér register
6. PC Update
 - Sæt PC til beregnede næste PC

	OPq rA, rB // $OP \in \{\text{add, sub, and, xor}\}$	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+2$	Læs instruktions byte Læs register byte Beregn næste PC
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	Læs operand A Læs operand B
Execute	$\text{valE} \leftarrow \text{valB OP valA}$ Set CC	Udfør ALU operation Sæt condition code register
Memory		
Write back	$R[\text{rB}] \leftarrow \text{valE}$	Skriv resultat tilbage
PC update	$\text{PC} \leftarrow \text{valP}$	Opdatér PC

- Beskriver nøje instruktions udførelse som sekvens af simple skridt
- Bruger samme generelle skabelon for alle instruktioner

Udførelse af `rmmovq`

`rmmovq rA, D(rB)`

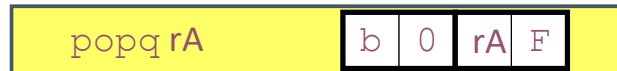


1. Fetch
 - Læs 10 bytes
2. Decode
 - Læs operand registre
3. Execute
 - Beregn effektive adresse: $valE = (rB) + D$
4. Memory
 - Skriv til hukommelsen
5. Write back
 - Gør intet
6. PC Update
 - Incrementér PC med 10

	<code>rmmovq rA, D(rB)</code>	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$	Læs instruktions byte Læs register byte Læs displacement D Beregn næste PC
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	Læs operand A Læs operand B
Execute	$valE \leftarrow valB + valC$	Beregn effektive adresse
Memory	$M_8[valE] \leftarrow valA$	Skriv værdi til hukommelsen
Write back		
PC update	$PC \leftarrow valP$	Opdater PC

Bruger ALU til adresse beregning

Udførelse af popq



1. Fetch
 - Læs 2 bytes
2. Decode
 - Læs stak pointer
3. Execute
 - Inkrementér stak pointer med 8
4. Memory
 - Læs fra tidligere stak pointer
5. Write back
 - Opdater stack pointer
 - Skriv resultat til register
6. PC Update
 - Incrementér PC med 2

popq rA		
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+2$	Læs instruktion byte Læs register byte Beregn next PC
Decode	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$	Læs stak pointer Læs stak pointer
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Inkrementér stack pointer
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Læs fra stak
Write back	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$	Opdater stak pointer Skriv resultat i register
PC update	$\text{PC} \leftarrow \text{valP}$	Opdater PC

- Brug ALU til øgning af stak pointer
- Skal opdatere to registre
 - Ét til at gemme den afstakkede værdi
 - Ny stak pointer

Udførelse af Conditional Moves

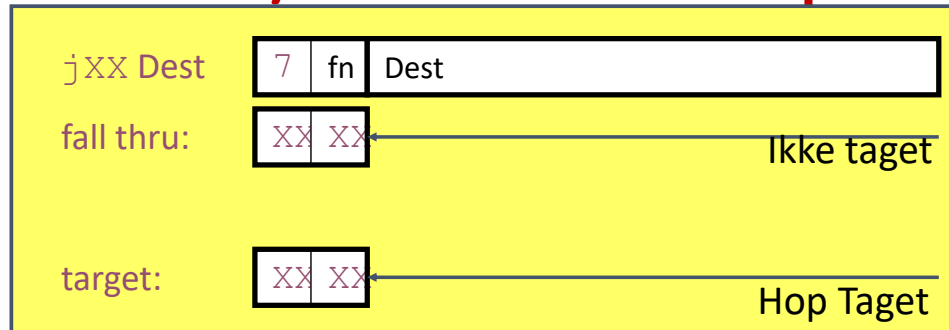
cmovXX rA, rB



1. Fetch
 - Læs 2 bytes
2. Decode
 - Læs operand registers
3. Execute
 - Hvis !cnd, så sæt destination register til 0xF
4. Memory
 - Gør intet
5. Write back
 - Opdater register (eller udelad)
6. PC Update
 - Incrementér PC med 2

	cmovXX rA, rB	
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$ valP $\leftarrow PC+2$	Læs instruktions byte Læs register byte Beregn ny PC
Decode	valA $\leftarrow R[rA]$ valB $\leftarrow 0$	Læs operand A
Execute	valE $\leftarrow valB + valA$ If ! Cond(CC,ifun) rB $\leftarrow 0xF$	Før valA gennem ALU (register skal ikke opdateres)
Memory		
Write back	R[rB] $\leftarrow valE$	Skriv resultat til register
PC update	PC $\leftarrow valP$	Opdater PC

Udførelse af Jumps

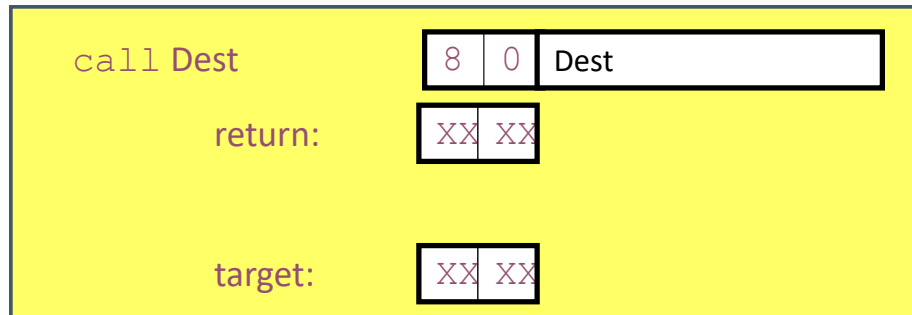


1. Fetch
 - Læs 9 bytes
 - Incrementér PC med 9
2. Decode
 - Gør intet
3. Execute
 - Evaluér om jump skal foretages eller ej baseret på jump betingelsen og condition codes
4. Memory
 - Gør intet
5. Write back
 - Gør intet
6. PC Update
 - Sæt PC til Dest hvis hop tages, eller til ny PC hvis ingen hop

	jXX Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+9$	Læs instruktions byte Læs destination adresse "Fall through" adresse
Decode		
Execute	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	Foretag hop?
Memory		
Write back		
PC update	$\text{PC} \leftarrow \text{Cnd} ? \text{valC} : \text{valP}$	Opdater PC

- Beregn begge adresser
- Vælg afhængigt af based på condition codes og betingelse

Udførelse af call

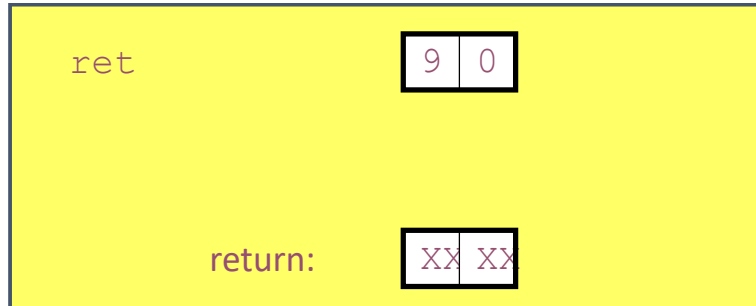


- Fetch
 - Læs 9 bytes
 - Inkrementér PC med 9
- Decode
 - Læs stak pointer
- Execute
 - Dekrementér stak pointer med 8
- Memory
 - Skriv inkrementeret PC til ny værdi for stak pointer
- Write back
 - Opmaskér stak pointer
- PC Update
 - Sæt PC til Dest

	call Dest	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $valC \leftarrow M_8[PC+1]$ $valP \leftarrow PC+9$	Læs instruktions byte Læs destination adresse Beregn retur adresse
Decode	$valB \leftarrow R[\%rsp]$	Læs stak pointer
Execute	$valE \leftarrow valB + -8$	Dekrementér stak pointer
Memory	$M_8[valE] \leftarrow valP$	Skriv retur adresse på stak
Write back	$R[\%rsp] \leftarrow valE$	Opmaskér stak pointer
PC update	$PC \leftarrow valC$	Sæt PC til destination

- Brug ALU til dekrementering af stak pointer
- Gem inkrementeret PC

Udførelse af `ret`



1. Fetch
 - Læs 1 byte
2. Decode
 - Læs stak pointer
3. Execute
 - Incrementér stak pointer med 8
4. Memory
 - Læs retur adresse fra gammel stak pointer
5. Write back
 - Opdatér stak pointer
6. PC Update
 - Sæt PC til retur adresse

	ret	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Læs instruktions byte
Decode	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$	Læs operand stak pointer Læs operand stak pointer
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Inkrementér stak pointer
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Læs retur adresse
Write back	$R[\%rsp] \leftarrow \text{valE}$	Opdatér stak pointer
PC update	$\text{PC} \leftarrow \text{valM}$	Sæt PC til retur adresse

- Brug ALU til inkrementering af stak pointer
- Læs retur adresse fra hukommelsen

Beregningsskridt: Generel Opskrift

		OPq rA, rB	
Fetch	icode,ifun	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Læs instruktions byte
	rA,rB	$\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	[Læs register byte]
	valC		[Læs konstant ord]
	valP	$\text{valP} \leftarrow \text{PC}+2$	Beregn næste PC
Decode	valA, srcA	$\text{valA} \leftarrow R[\text{rA}]$	Læs operand A
	valB, srcB	$\text{valB} \leftarrow R[\text{rB}]$	Læs operand B
Execute	valE	$\text{valE} \leftarrow \text{valB OP valA}$	Udfør ALU operation
	Cond code	Set CC	Sæt/brug cond. codes
Memory	valM		[Hukommelse læs/skriv]
Write	dstE	$R[\text{rB}] \leftarrow \text{valE}$	Skriv ALU resultat til register
back	dstM		[Skriv resultat til hukommelse]
PC update	PC	$\text{PC} \leftarrow \text{valP}$	Opdatér PC

- Alle instruktioner følger same generelle mønster
- Forskelligt præcist hvad der beregnes i hvert skridt

Beregningsskridt for CALL

		call Dest	
Fetch	icode,ifun	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Læs instruktions byte
	rA,rB		Læs register byte
	valC	$\text{valC} \leftarrow M_8[\text{PC}+1]$	[Læs konstant ord]
	valP	$\text{valP} \leftarrow \text{PC}+9$	Beregn næste PC
Decode	valA, srcA		Læs operand A
	valB, srcB	$\text{valB} \leftarrow R[\%rsp]$	Læs operand B
Execute	valE	$\text{valE} \leftarrow \text{valB} + -8$	Udfør ALU operation
	Cond code		Sæt/brug cond. codes
Memory	valM	$M_8[\text{valE}] \leftarrow \text{valP}$	[Hukommelse læs/skriv]
Write	dstE	$R[\%rsp] \leftarrow \text{valE}$	Skriv ALU resultat til register
back	dstM		[Skriv resultat til hukommelse]
PC update	PC	$\text{PC} \leftarrow \text{valC}$	Opdatér PC

- Alle instruktioner følger same generelle mønster
- Forskelligt præcist hvad der beregnes i hvert skridt

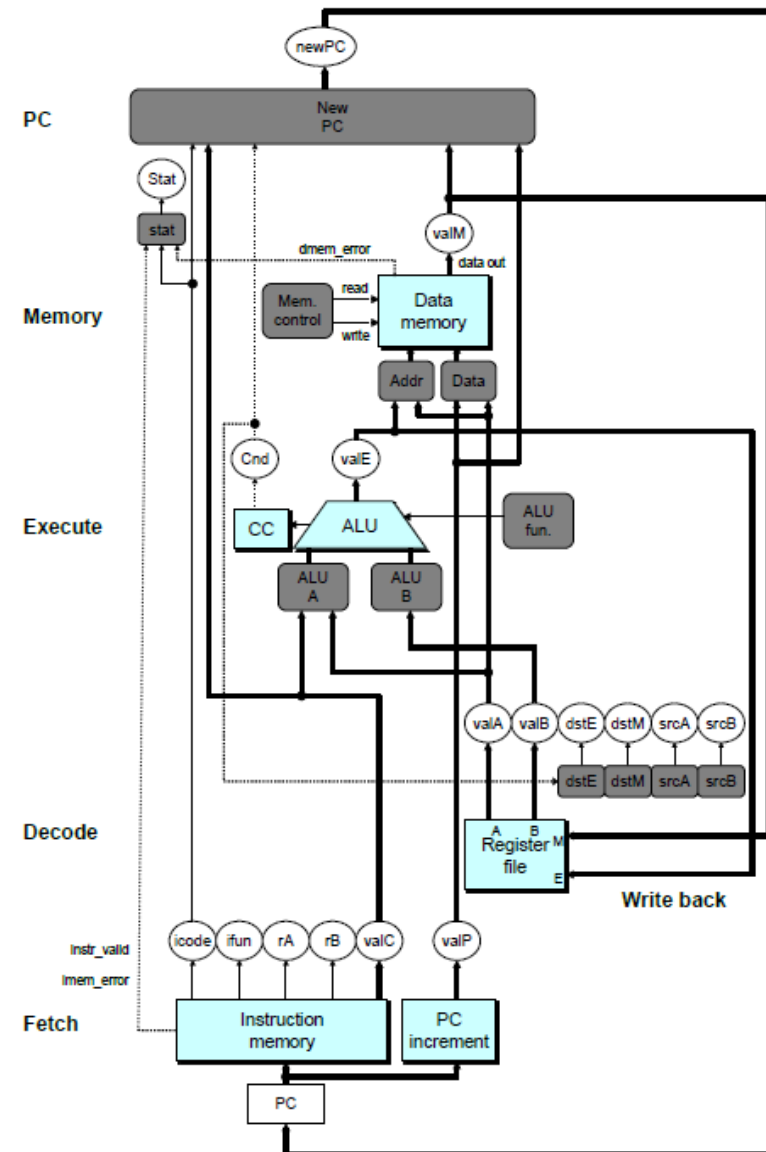
Logik design for Y86-SEQ

SEQ Hardware Opbygning

Hvordan kan vi realisering af Y86-SEQ modellen ved brug af digital logik?

Signatur forklaring

- Blå rektangler: hardware bygge blokke
 - Fx., Hukommelse, registre, ALU
- Grå boxes: kontrol logik
 - Kan beskrives i HCL
- Hvide ovale: signal navne (labels)
- Tykke linier: 64-bit værdi
- Tynde lines: 4-8 bit værdi
- Prikkede lines: 1-bit værdi

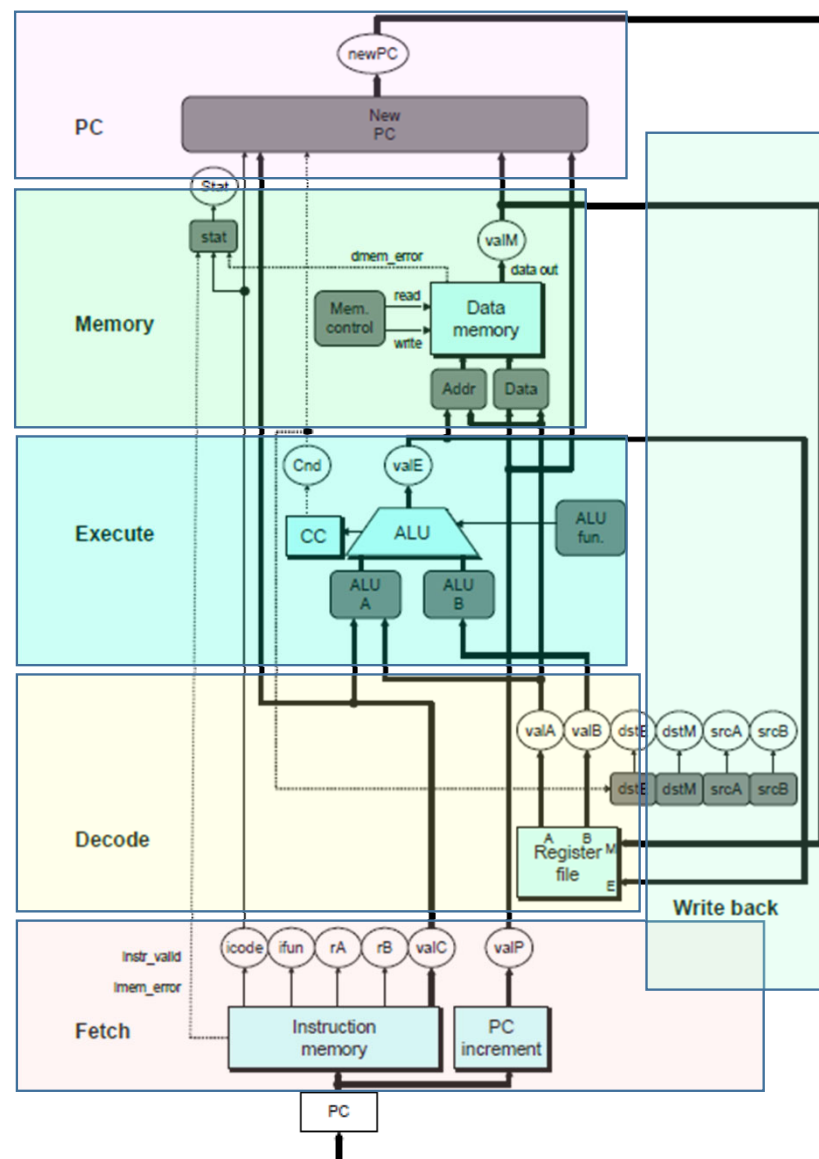


Ikke alle forbindelser/kontrol logik er vist

SEQ Hardware

Signatur forklaring

- Blå rektangler: hardware byggeblokke
 - Fx., Hukommelse, registre, ALU
- Grå bokse: kontrol logik
 - Kan beskrives i HCL
- Hvide ovale: signal navne (labels)
- Tykke linieres: 64-bit værdi
- Tynde lines: 4-8 bit værdi
- Prikkede lines: 1-bit værdi



SEQ Hardware

Beregnete Værdier

•Fetch

icode	Instruktions-kode
ifun	Instruktions-function
rA	Instr. Register ID A
rB	Instr. Register ID B
valC	Instruktions konstant
valP	Inkrementeret PC

•Decode

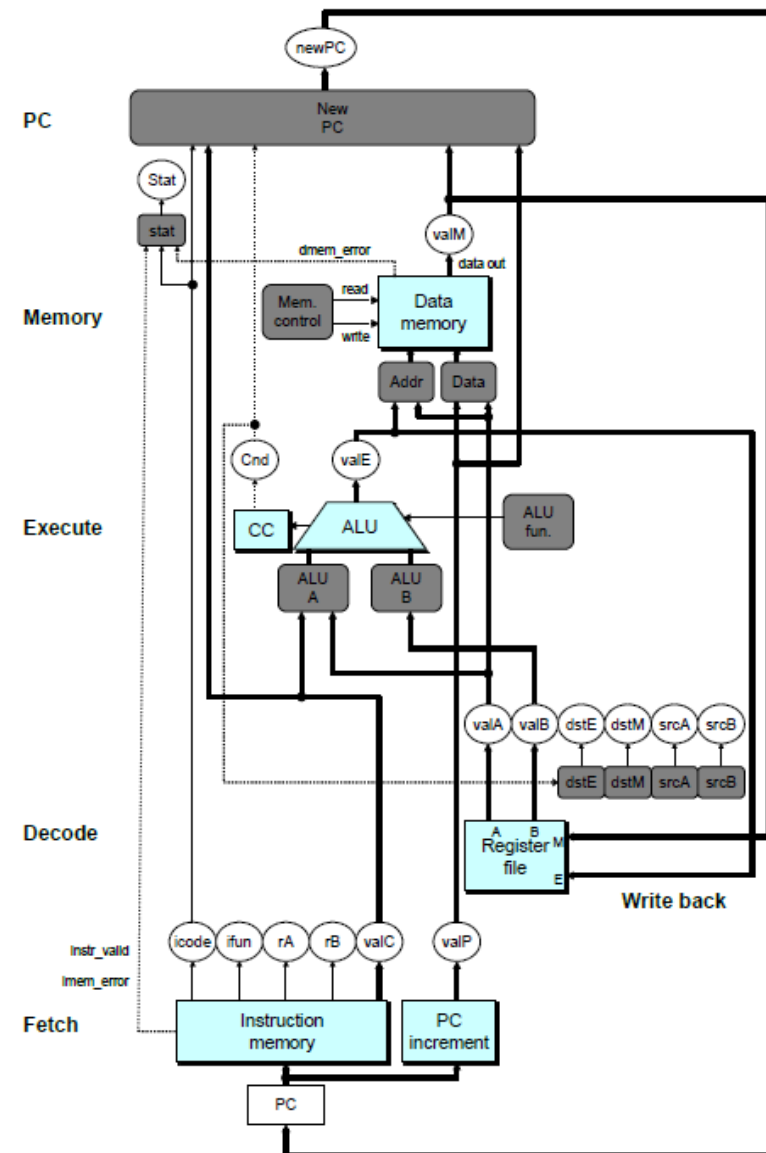
srcA	Register ID A
srcB	Register ID B
dstE	Destination Register ID E (Exec.)
dstM	Destination Register ID M (Mem.)
valA	Værdi fra register A
valB	Værdi fra register B

•Execute

- valE ALU resultat
- Cnd Branch/move flag

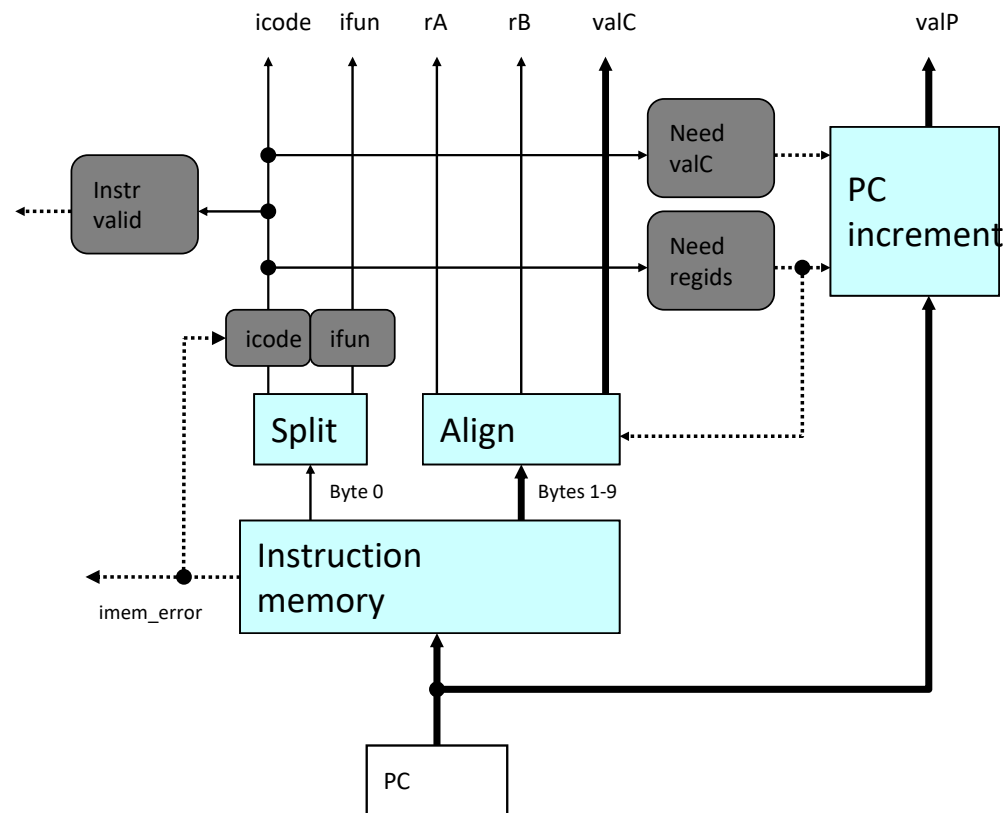
•Memory

- valM Værdi fra hukommelse



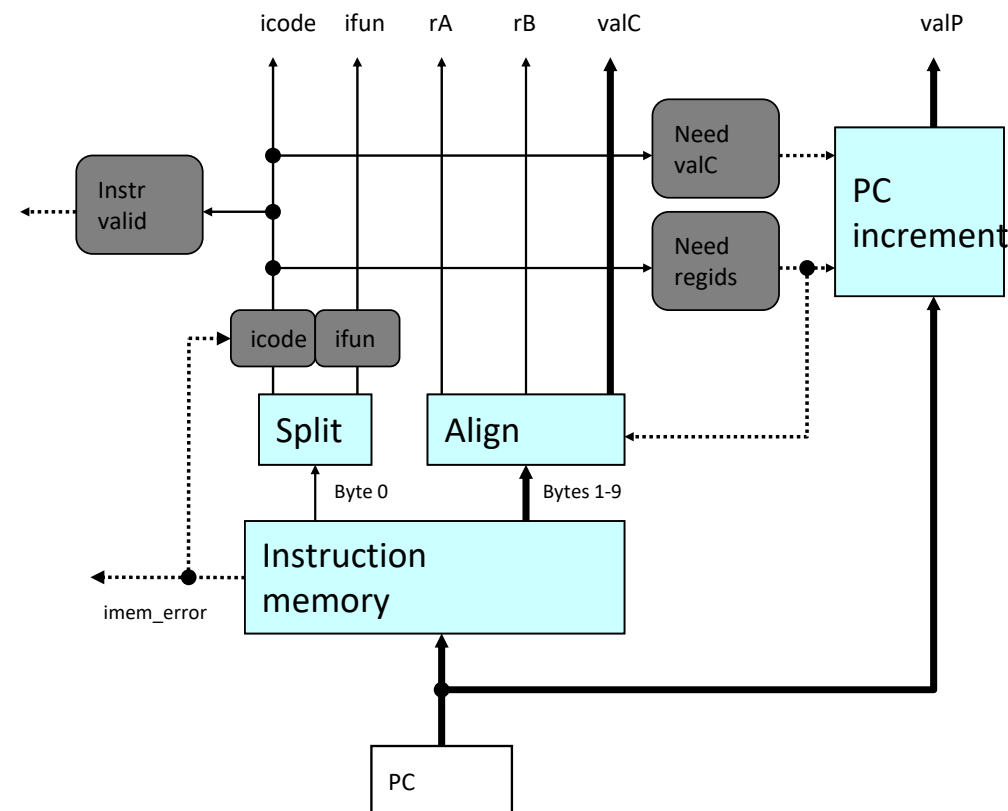
Fetch Logik

- Eksisterende byggeblokke
 - PC: Register til PC
 - **Instruktionshukommelse**: Læs 10 bytes (fra adresse PC t.o.m PC+9)
 - Signalér ugyldig adresse
 - **Split**: Opdel instruktions byte i icode og ifun
 - **Align**: Hent felterne for rA, rB, og valC
 - **PC increment**: Forøg PC med det som instruktionen fylder
- De grå blokke skal vi selv designe ;-)



Fetch Logik

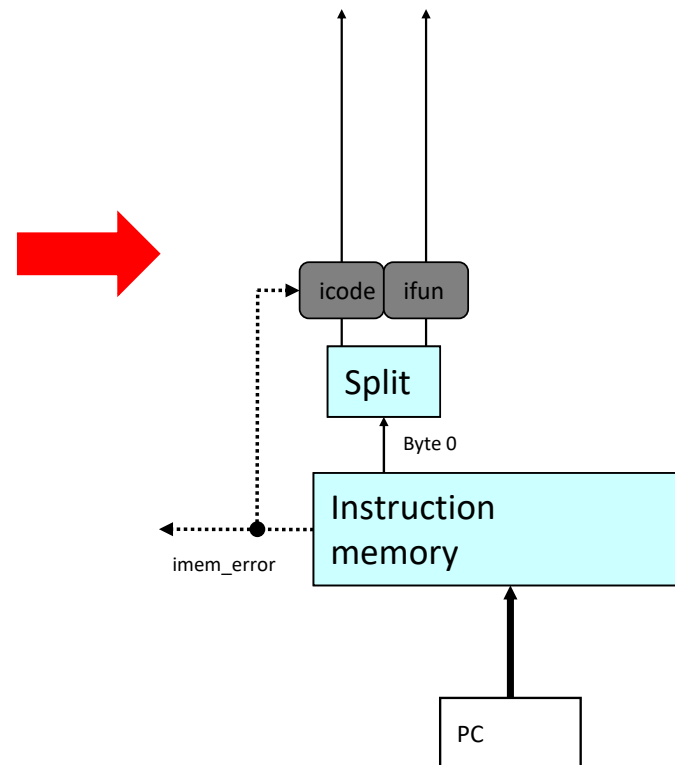
- Egen kontrol Logik
 - “icode”, “ifun”: Genererer ”no-op” hvis adressen er ugyldig
 - “Instr. Valid”: Er denne instruktion gyldig?
 - “Need regids”: Indeholder denne instruktion en register byte?
 - “Need valC”: Indeholder denne instruktion et konstant ord?



Kontrol Logik for Fetch trin i HCL

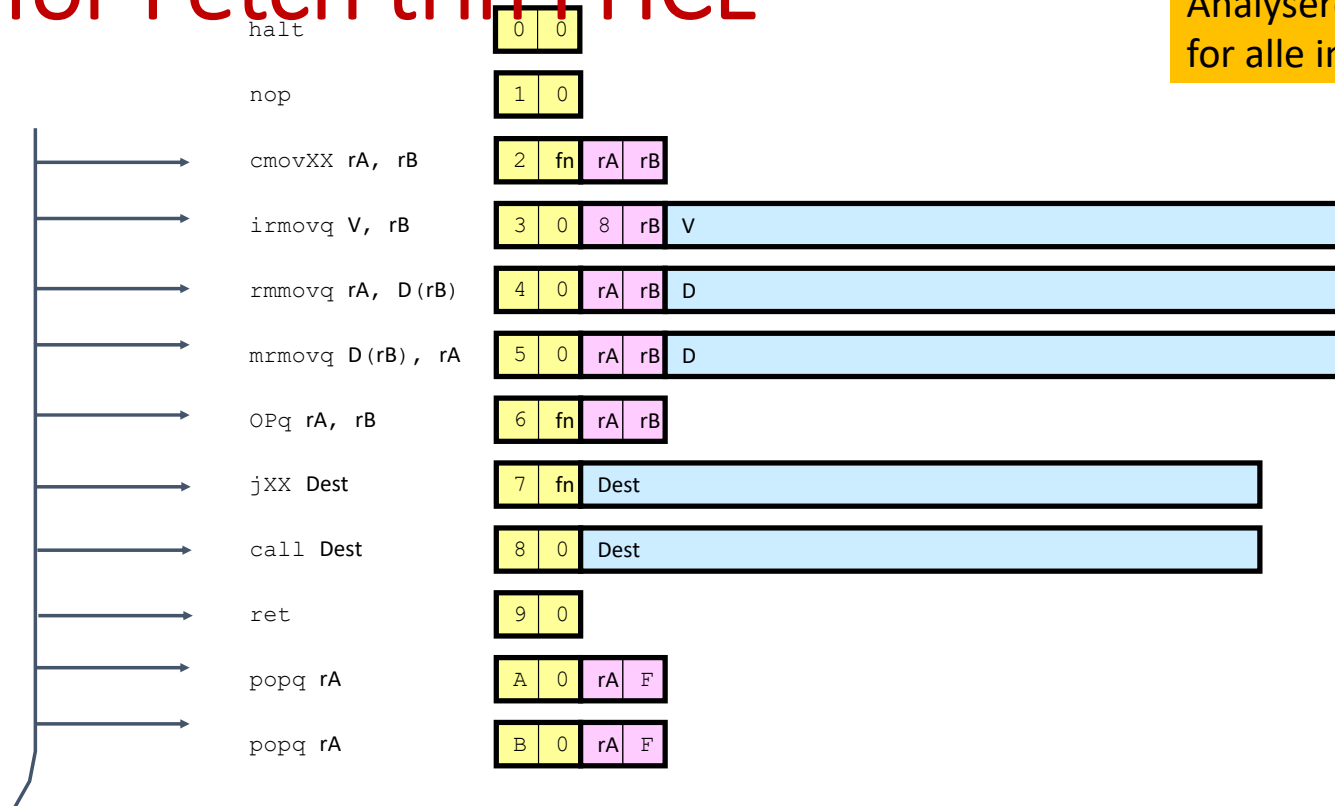
```
# Determine instruction code
int icode = [
    imem_error: INOP;
    1: imem_icode;
];

# Determine instruction function
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;
];
```



Kontrol Logik for Fetch trin i HCL

Analyserer formatet
for alle instruktioner



```
bool need_regids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPOPQ,
               IIRMOVQ, IRMMOVQ, IMRMVQ };
```

```
bool instr_valid = icode in
    { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMVQ,
      IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPOPQ };
```

CMOV, RRMV
deler ICODE #2

Logik for Dekodningstrin

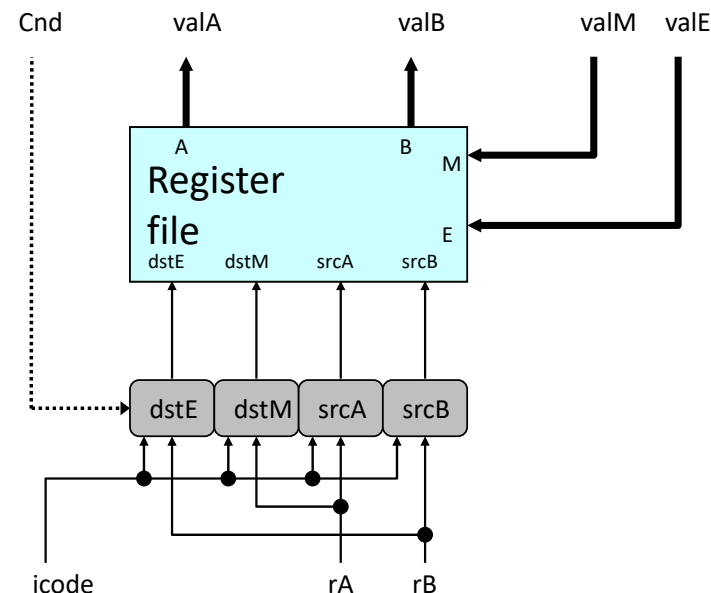
- Register Bank
 - Læse porte A, B
 - Skrive porte E, M
 - Adresser er register IDs el 15 (0xF) (ingen operation)

Kontrol Logik

- **srcA, srcB**: adresser på læseporte
- **dstE, dstM**: adresser på skriveporte

Signaler

- **Cnd**: Angiver om der skal udføres conditional move
 - Beregnes i Execute trin



Source srcA

Source for A er register ID fra instruktionen (hvis benyttet), medmindre det er en stak-operation: brug %rsp

	OPq rA, rB	
Decode	valA \leftarrow R[rA]	Læs operand A
	cmovXX rA, rB	
Decode	valA \leftarrow R[rA]	Læs operand A
	rmmovq rA, D(rB)	
Decode	valA \leftarrow R[rA]	Læs operand A
	popq rA	
Decode	valA \leftarrow R[%rsp]	Læs stak pointer
	jXX Dest	
Decode		Ingen operand
	call Dest	
Decode		Ingen operand
	ret	
Decode	valA \leftarrow R[%rsp]	Læs stak pointer

Samlet dekode-trinene for alle instruktioner

```
int srcA = [
    icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;
    icode in { IPOPQ, IRET } : RRSP;
    1 : RNONE; # Don't need register
];
```

Destination dstE

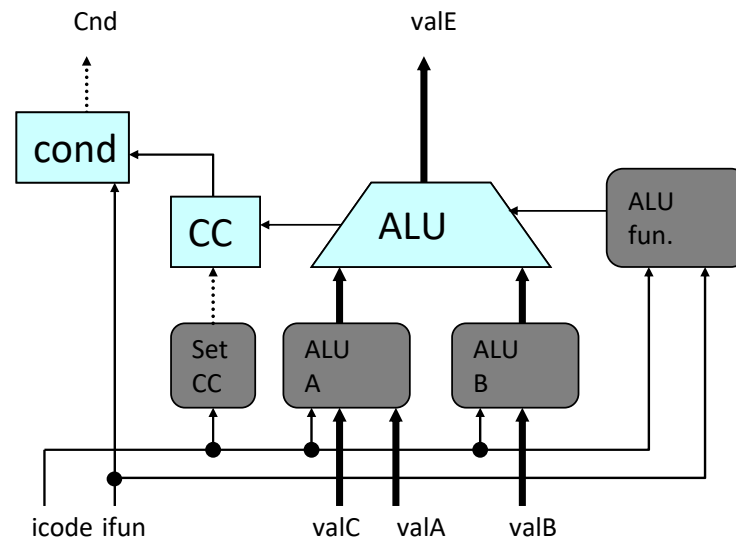
	OPq rA, rB	
Write-back	R[rB] ← valE	Skriv resultat
	cmovXX rA, rB	
Write-back	R[rB] ← valE	Betinget skriv af resultat
	rmmovq rA, D(rB)	
Write-back		Ingen
	popq rA	
Write-back	R[%rsp] ← valE	Opdatér stak pointer
	jXX Dest	
Write-back		Ingen
	call Dest	
Write-back	R[%rsp] ← valE	Opdatér stak pointer
	ret	
Write-back	R[%rsp] ← valE	Opdatér stak pointer

Samlet write-back -
trinene for alle
instruktioner

```
int dstE = [
    icode in { IRRMOVQ } && Cnd : rB;
    icode in { IIRMOVQ, IOPQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # Don't write any register
];
```

Kontrol logic til Execute

- Enheder
 - ALU
 - Implementerer de 4 krævede funktioner
 - Generates condition code values
 - CC
 - Register med de 3 condition code bits
 - cond
 - Beregner conditional jump/move flag
- Kontrol Logic
 - “Set CC”: Skal condition code register ændres?
 - “ALU A”: Input A til ALU
 - “ALU B”: Input B til ALU
 - “ALU fun.”: Hvilken funktion skal ALU beregne?



ALU Input A

	OPq rA, rB	
Execute	$\text{valE} \leftarrow \text{valB} \text{ OP } \text{valA}$	Udfør ALU operation
	cmovXX rA, rB	
Execute	$\text{valE} \leftarrow 0 + \text{valA}$	Send valA igennem ALU
	rmmovq rA, D(rB)	
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Beregn effektive adresse
	popq rA	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Inkrementér stack pointer
	jXX Dest	
Execute		Ingen operation
	call Dest	
Execute	$\text{valE} \leftarrow \text{valB} + -8$	Decrementér stack pointer
	ret	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Inkrementér stack pointer

Samlet execute-
trinene for alle
instruktioner

```
int aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPC } : 8;
    # Other instructions don't need ALU
];
```

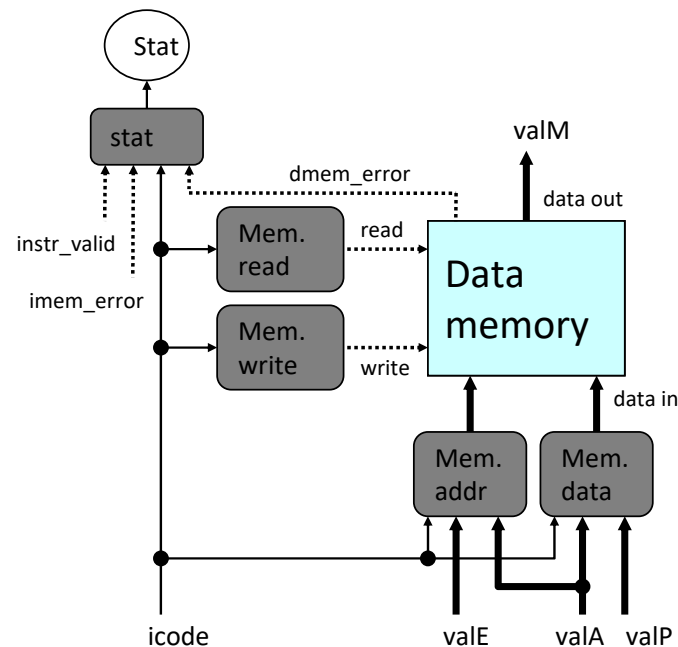
ALU Operation

	OPl rA, rB	
Execute	$\text{valE} \leftarrow \text{valB} \text{ OP } \text{valA}$	Udfør ALU operation
	cmovXX rA, rB	
Execute	$\text{valE} \leftarrow 0 + \text{valA}$	Send valA igennem ALU
	rmmovl rA, D(rB)	
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Beregn effektive adresse
	popq rA	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Incrementér stak pointer
	jXX Dest	
Execute		Ingen operation
	call Dest	
Execute	$\text{valE} \leftarrow \text{valB} + -8$	Decrementér stak pointer
	ret	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Incrementér stak pointer

```
int alufun = [
    icode == IOPQ : ifun;
    1 : ALUADD;
];
```

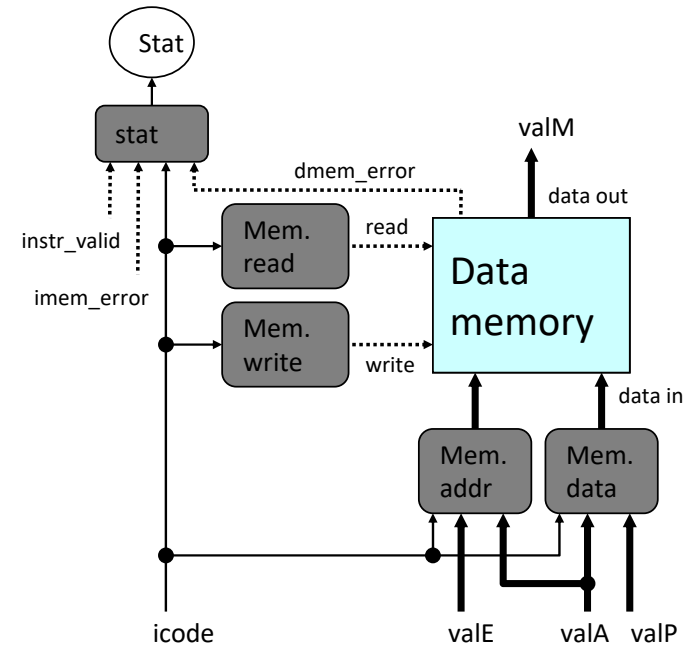
Kontrol logic til Memory trin

- Memory
 - Læser eller skriver et ord til hukommelsen
- Kontrol Logic
 - **stat**: Hvad er instruktionens status?
 - **Mem. read**: Skal et ord læses?
 - **Mem. write**: skal et ord skrives?
 - **Mem. addr.**: Vælg adresse
 - **Mem. data.**: Vælg data



Instruktions Status

- Kontrol Logic
 - **stat**: Hvad er instruktionens status?



```
## Determine instruction status
int Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid: SINS;
    icode == IHALT : SHLT;
    1 : SAOK;
];
```

Hukommelses Adresse

Samlet Memory-
trinene for alle
instruktioner

	OPq rA, rB	
Memory		Ingen operation
	rmmovq rA, D(rB)	
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	Skriv værdi til hukommelsen
	popq rA	
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Læs fra stak
	jXX Dest	
Memory		Ingen operation
	call Dest	
Memory	$M_8[\text{valE}] \leftarrow \text{valP}$	Skriv retur adresse på stak
	ret	
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Læs retur adresse

```
int mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    # Other instructions don't need address
];
```

Hukommelseslæsning

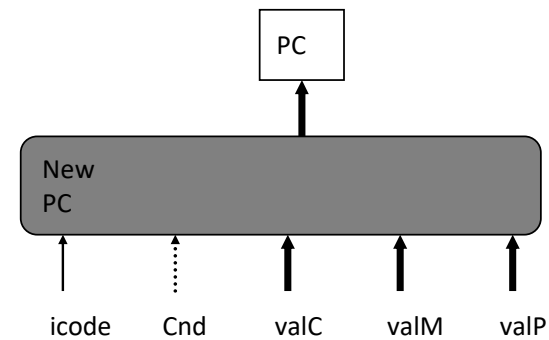
Samlet Memory-
trinene for alle
instruktioner

	OPq rA, rB	
Memory		Ingen operation
	rmmovq rA, D(rB)	Skriv værdi til hukommelsen
Memory	$M_8[valE] \leftarrow valA$	Ingen operation
	mrmmovq D(rA), Rb	
Memory	$valM \leftarrow M_8[valE]$	Læs værdi til hukommelsen
	popq rA	
Memory	$valM \leftarrow M_8[valA]$	Læs fra stak
	jXX Dest	
Memory		Ingen operation
	call Dest	
Memory	$M_8[valE] \leftarrow valP$	Skriv retur adresse på stak
	ret	
Memory	$valM \leftarrow M_8[valA]$	Læs retur adresse

```
bool mem_read = icode in { IMRMOVQ, IPOPOPQ, IRET };
```

Logik til opdatering af PC

- Ny PC
 - Udvælg næste værdi for PC



PC Opdatering

	OPq rA, rB	
PC update	PC ← valP	Opdatér PC
	rmmovq rA, D(rB)	
PC update	PC ← valP	Opdatér PC
	popq rA	
PC update	PC ← valP	Opdatér PC
	jXX Dest	
PC update	PC ← Cnd ? valC : valP	Opdatér PC
	call Dest	
PC update	PC ← valC	Sæt PC til destination
	ret	
PC update	PC ← valM	Sæt PC til retur adresse

Samlet PC-Update
trinene for alle
instruktioner

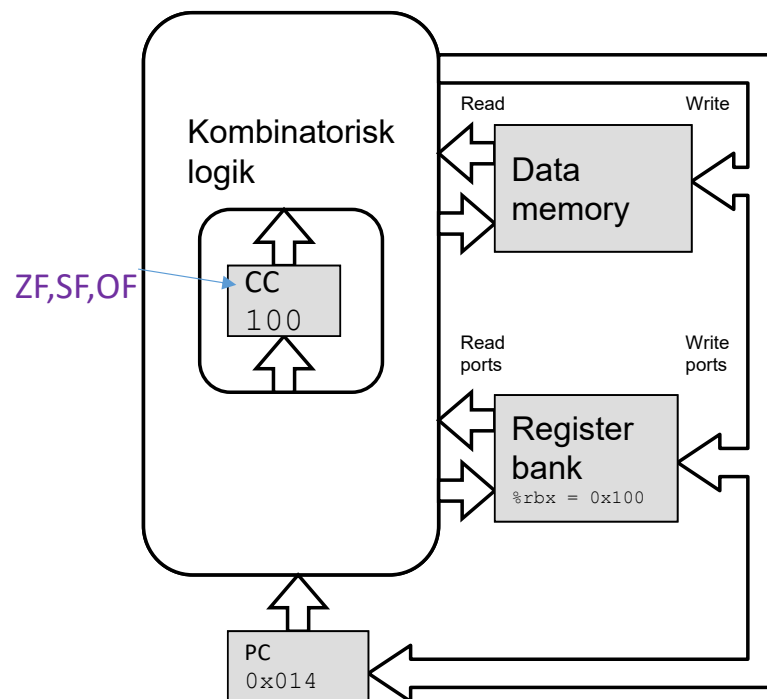
Conditional JMP

Opdatering af PC
afhænger af
instruktionskode!

```
int new_pc = [
    icode == ICALL : valC;
    icode == IJXX && Cnd : valC;
    icode == IRET : valM;
    1 : valP;
];
```

Eksekvering i Y86-SEQ

SEQ Eksekvering



- Tilstand består af:

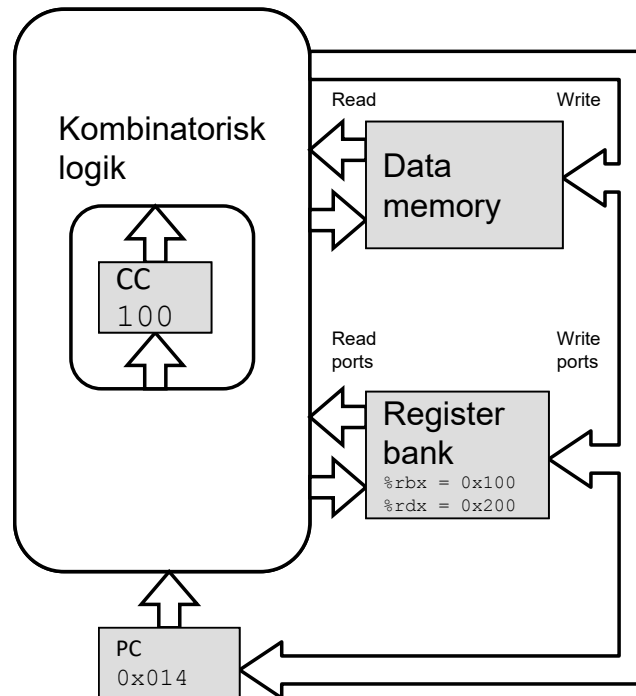
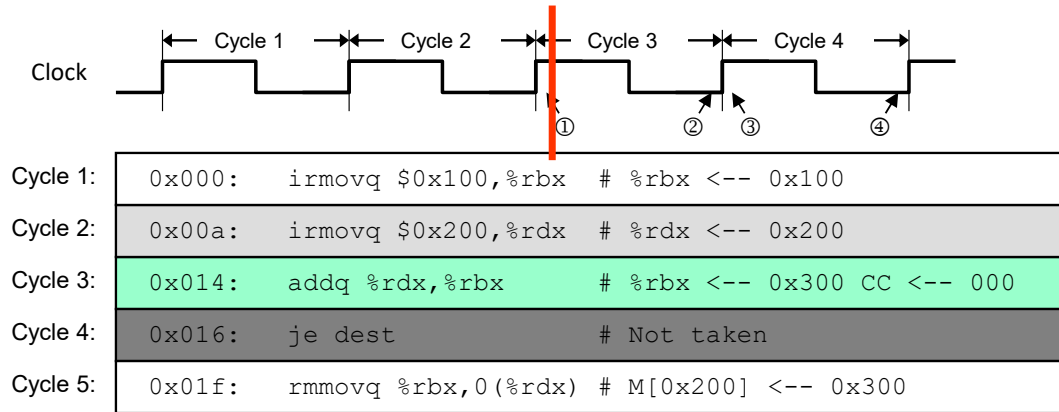
- PC register
- Cond. Code register
- Data Hukommelse
- Register bank

Alle opdateres når clock signalet stiger

- Kombinatorisk Logik:

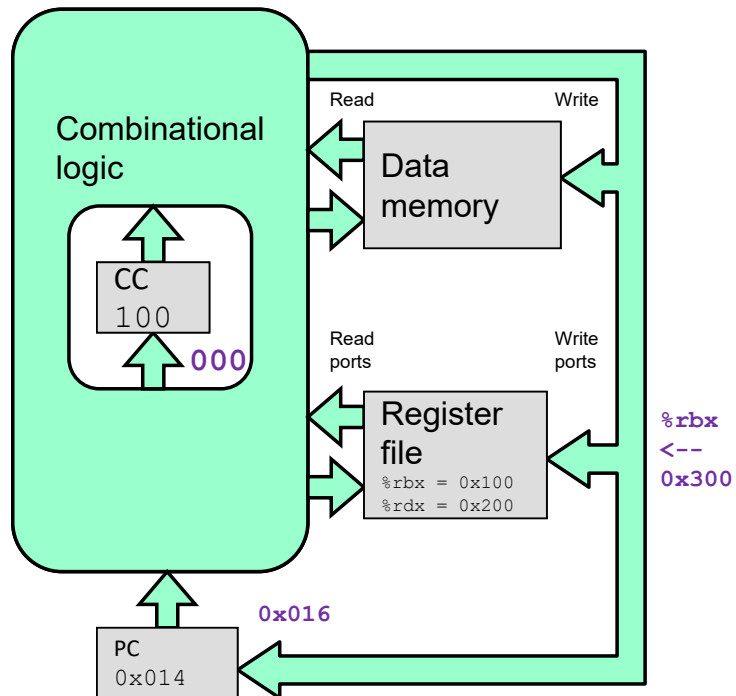
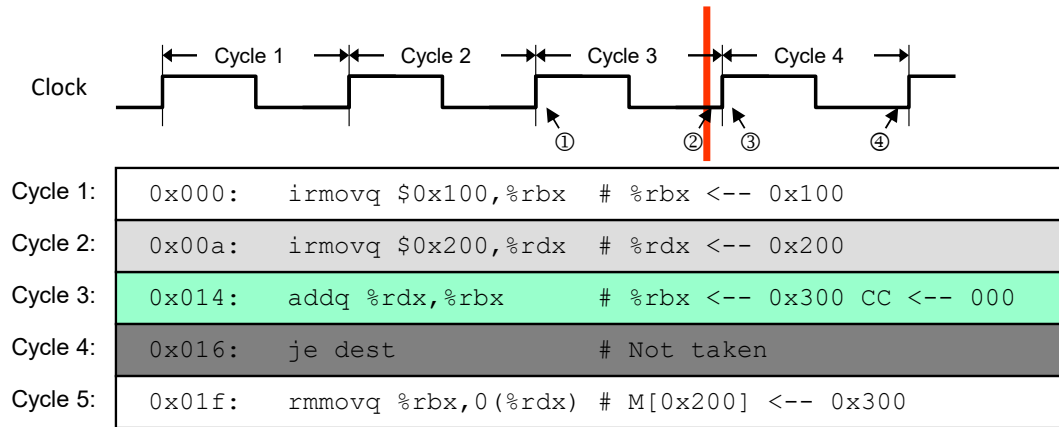
- ALU
- Kontrol logik
- Hukommelseslæsning
 - Instruktions hukommelse
 - Register bank
 - Data hukommelse

SEQ Eksekvering #1



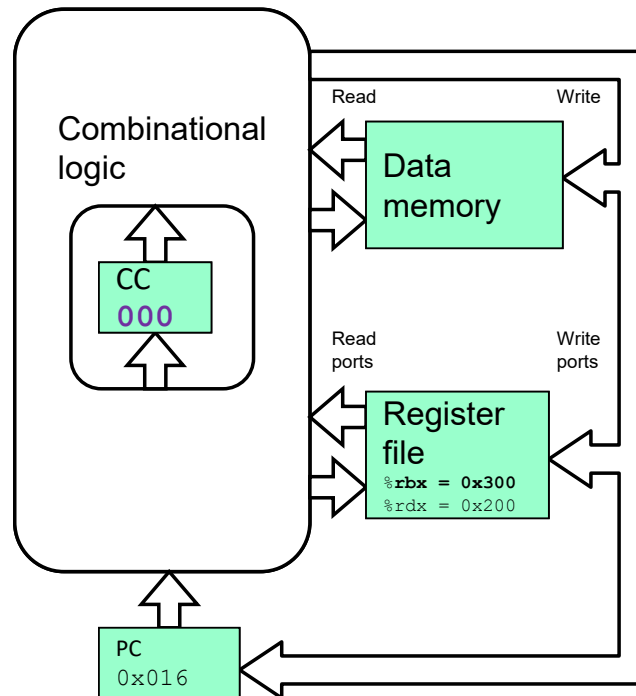
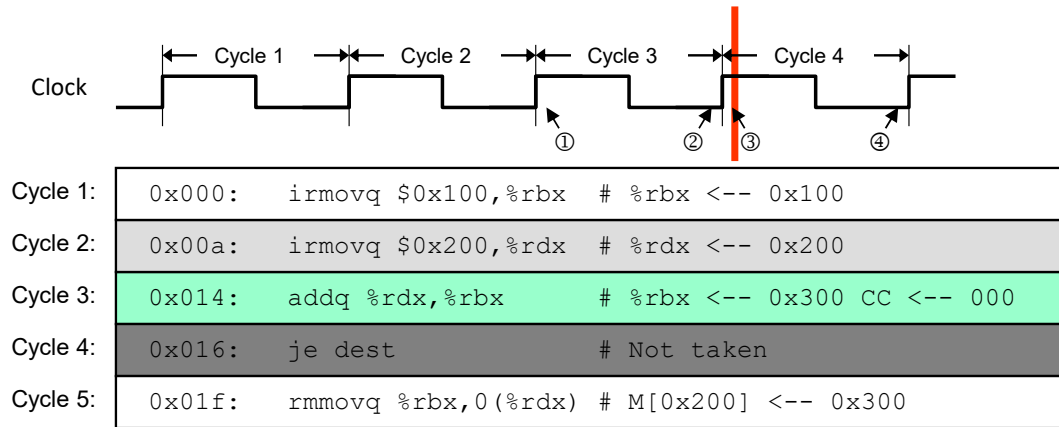
- Klokken lige slået til cycle 3
- Tilstand sat iht. anden `irmovq` instruktion
 - Satte %rdx=0x200,
 - Satte PC til 0x014
- Kombinatorisk logik **starter på at reagere på** tilstandsændring

SEQ Eksekvering #2



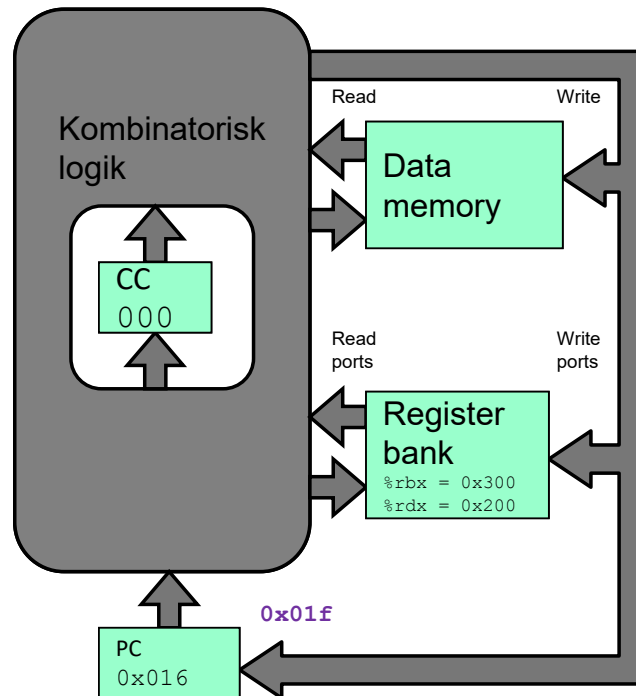
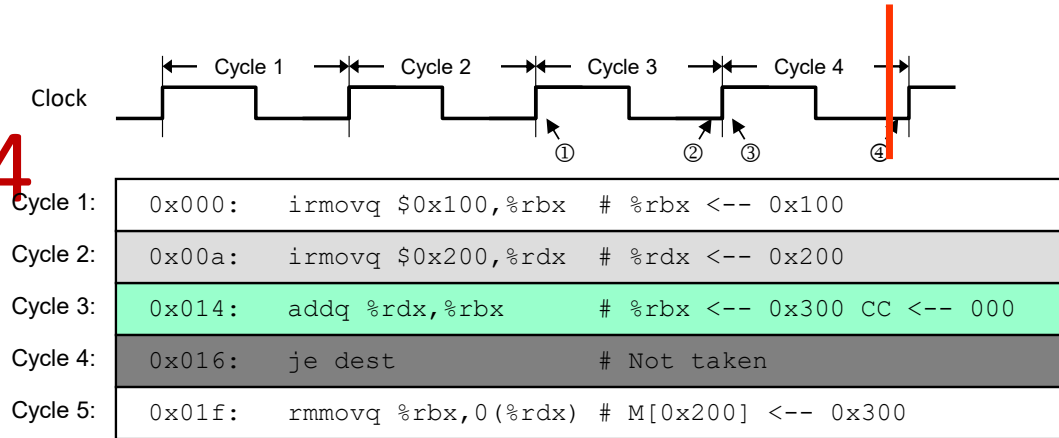
- Tilstand sat iht. anden `irmovq` instruktion
- kombinatorisk logik har **genereret resultat af** `addq` instruktion

SEQ Eksekvering #3



- Klokken lige slået til cycle 4
- Tilstand sat iht. addq instruktion
 - %rbx=0x300
 - CC=000
 - Pc=0x16
- Kombinatorisk logik starter på at reagere på tilstandsændring

SEQ Eksekvering #4



- Tilstand sat iht. `addq` instruktion
- kombinatorisk logik genererer resultat for `je` instruktion (ingen hop: `ZF=0`)
 - Ny PC skal være 0x1f

Y86-SEQ Resumé

- Implementation
 - Udtrykker hver instruktion som en serie af simple skridt
 - Følger same generelle trin (F,D,E,M,W) for hver instruktionstype
 - Byg registre, hukommelse, kombinatoriske byggeblokke (fx ALU)
 - Forbind med kontrol logik
- Begrænsninger
 - *For langsom!*
 - I én cyklus, skal instruktionen udbredes igennem instruktionshukommelse, register bank, ALU, og data hukommelse
 - Betyder at klokken skulle køre meget langsomt
 - De enkelte hardware enheder ville kun være aktive en lille del af hver cyklus
 - \Rightarrow Pipelining og instruktionsniveau parallelitet

