

Den simpleste maskine

Finn Schiermer Andersen,

Ekstern lektor, DIKU

Abstraktionsniveauer

1. Højniveau programmeringssprog: Erlang, OCaml, F# osv
2. Maskinnære programmeringssprog: C (og C++)
3. Assembler / Symbolsk Maskinsprog: x86, ARM, MIPS
4. Arkitektur (ISA): Maskinsprog - ordrer indkodet som tal
5. Mikroarkitektur: ting som lager, registre, regneenheder, afkodere og hvordan de forbindes så det bliver en maskine
6. Standard celler: Simple funktioner af få bit (1-4) med et eller to resultater. Lagring af data (flip-flops)
7. Transistorer
8. Fysik. Eller noget der ligner

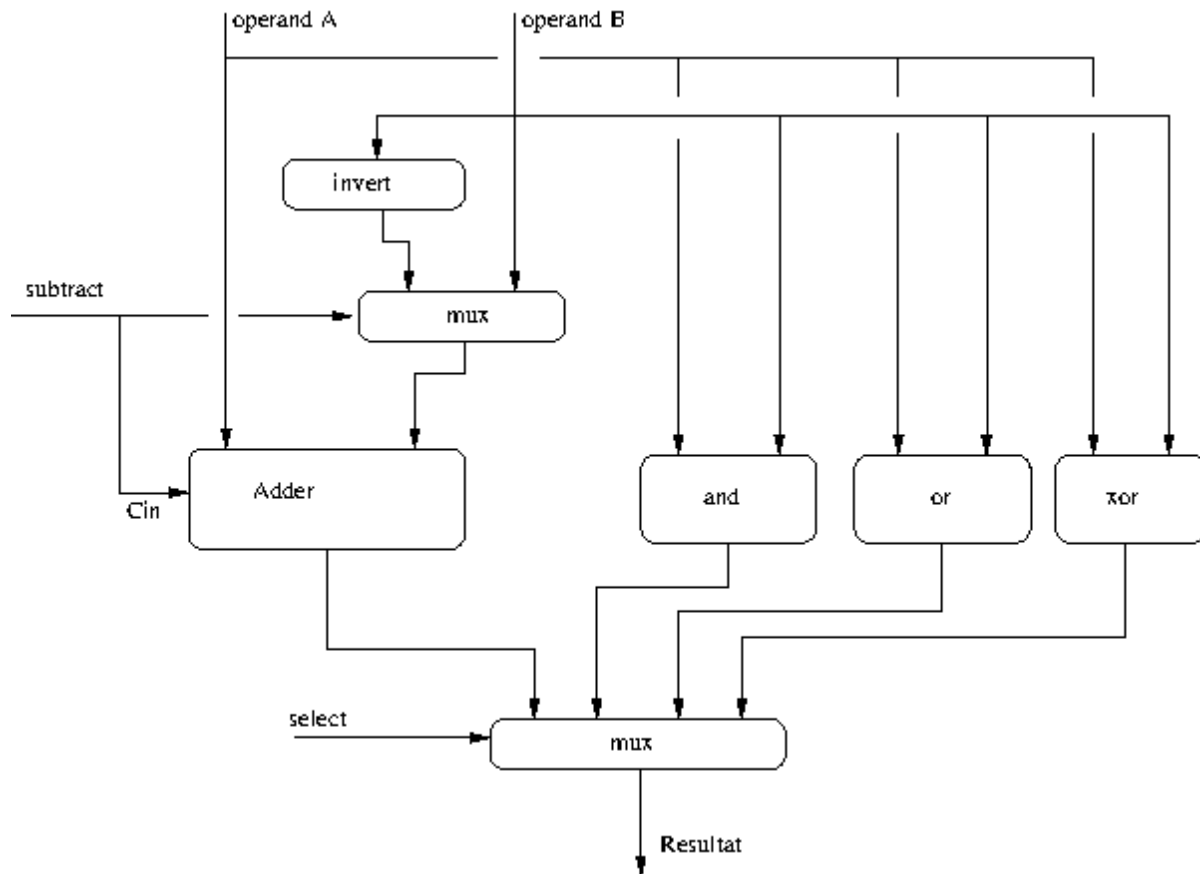
Fremover beskæftiger vi os med niveauerne 2-5 (evt en smule fra 6)

Vi vil nu bygge den Simpleste Maskine

Næsten

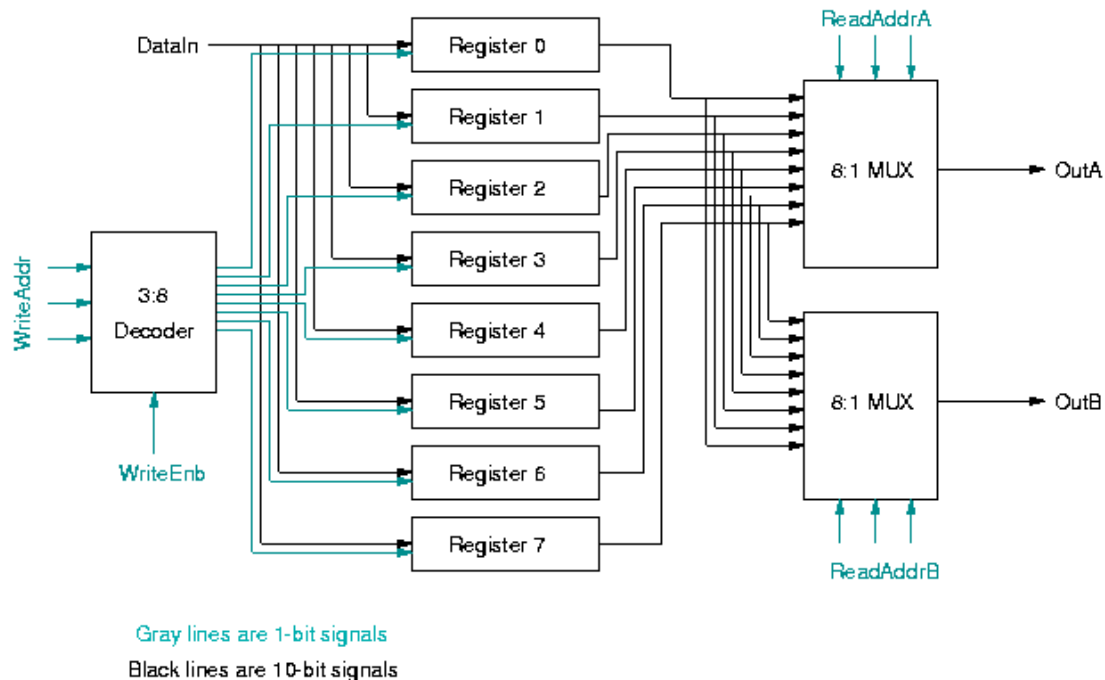
Recap (I)

Hvad er dette?



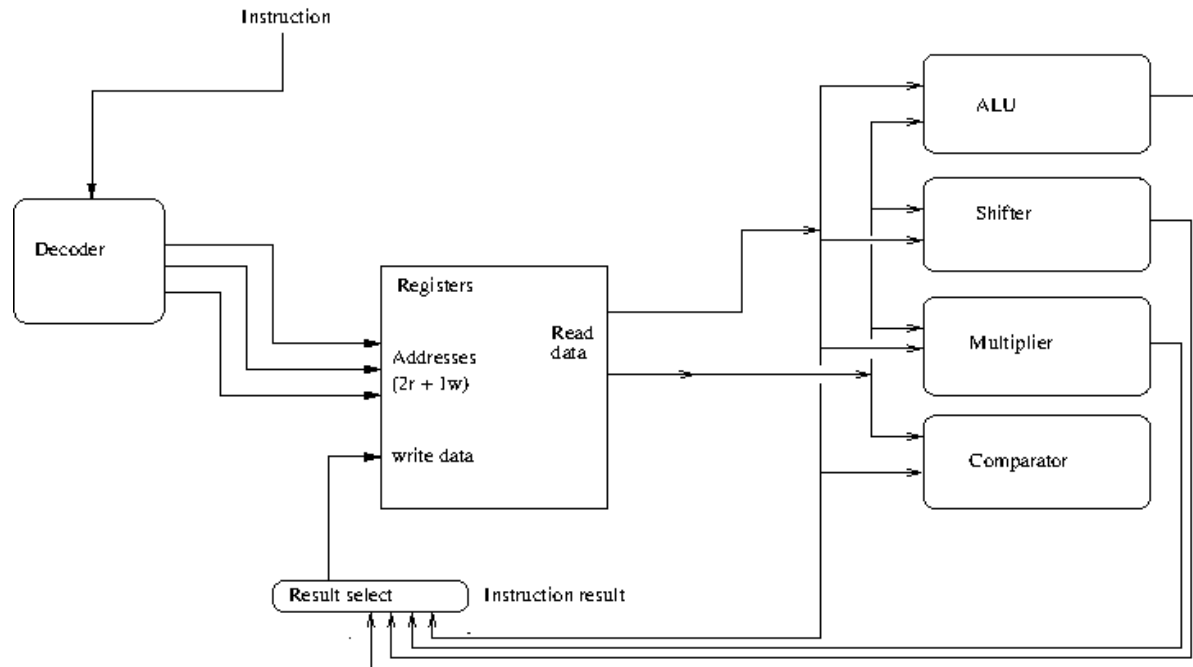
Recap (II)

Og hvad er dette?



De to komponenter er de vigtigste dele af en "datavej".. den del af maskinen hvor beregninger foregår

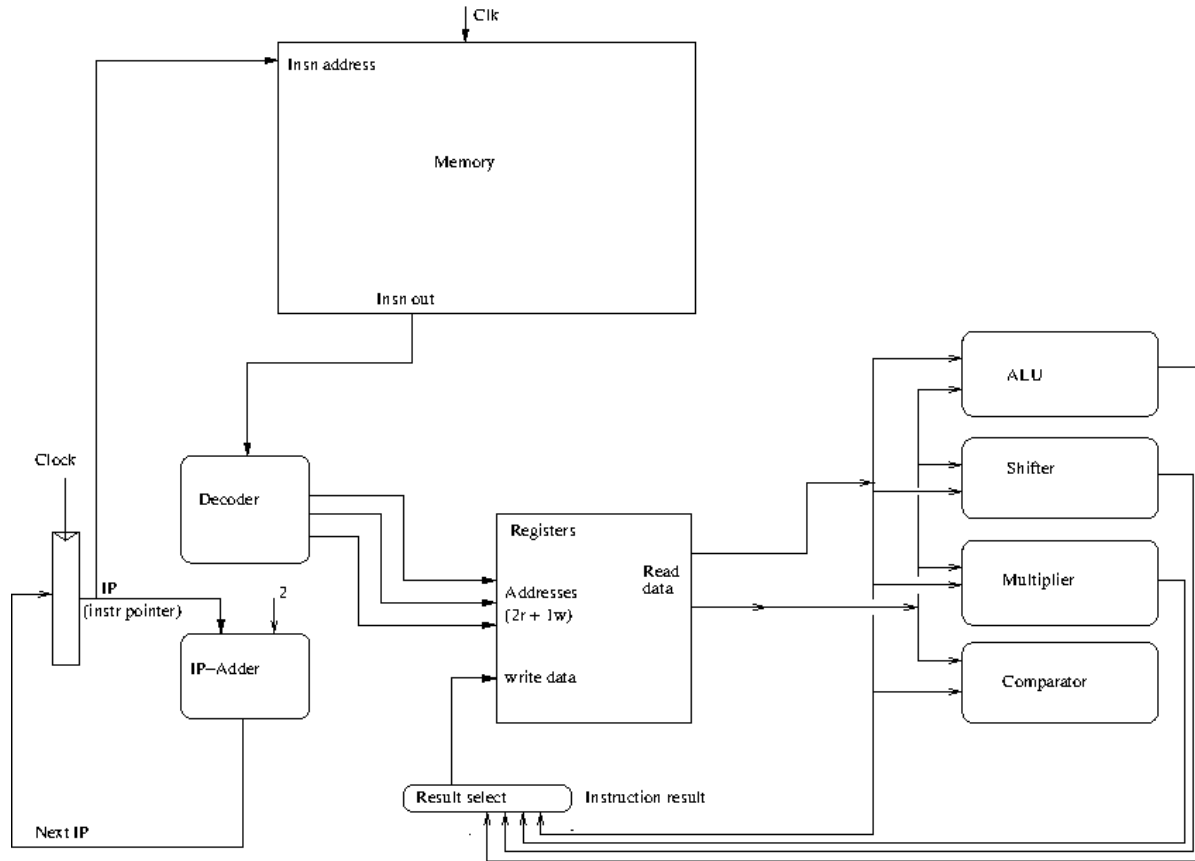
En simpel datavej (Datapath)



Mangler der ikke nogle forbindelser før det her kan virke? Hvilke?

Hvor kommer instruktionen der styrer det hele mon fra?

Vi henter instruktioner - en ad gangen



Nejjjj, hvor heldigt - de har alle samme størrelse (hvor kan man se det?)

Vores første instruktioner

00000000 00000000	stop
0001aaaa ddddssss	register/register arithmetic: op s,d
00100001 ddddssss	movq s,d

'dddd' og 'ssss' er registre, 'd' for destination, 's' for source.

Vi bruger x86 navne for registre:

0000 %rax	0001 %rbx	0010 %rcx	0011 %rdx
0100 %rbp	0101 %rsi	0110 %rdi	0111 %rsp
1000 %r8	1001 %r9	1010 %r10	1011 %r11
1100 %r12	1101 %r13	1110 %r14	1111 %r15

'aaaa' angiver aritmetisk operation som følger:

0000 add	0001 sub	0010 and	0011 or
0100 xor	0101 mul	0110 sar	0111 sal
1000 shr	1001 imul		

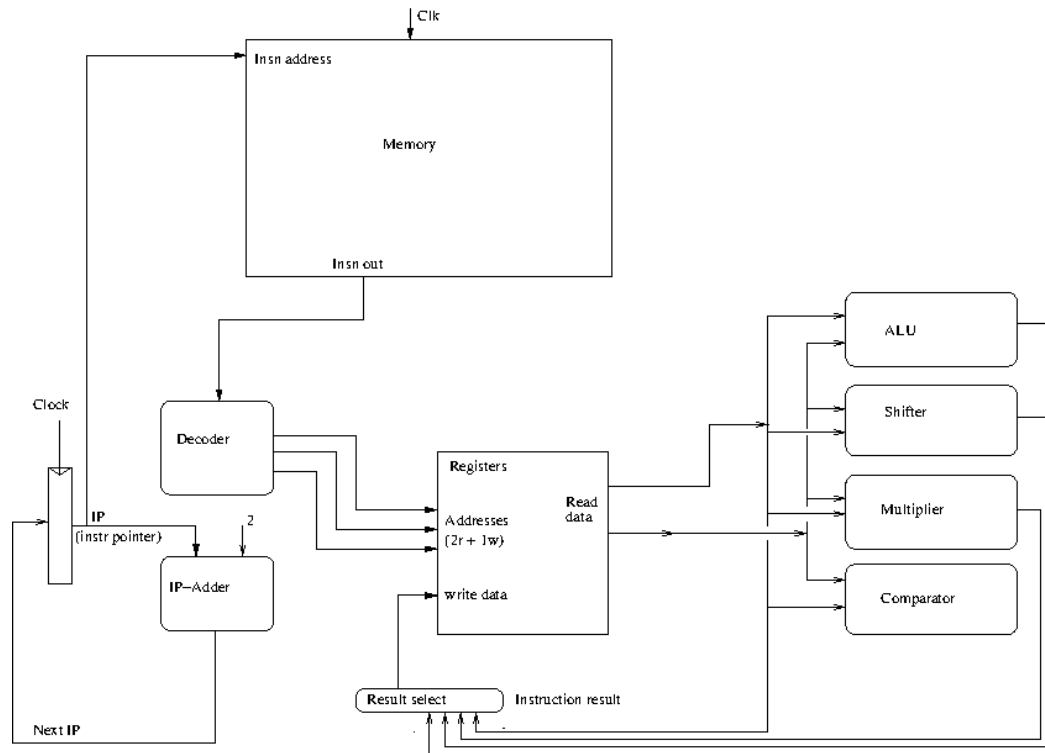
Et eksempel

Addq %rdx,%rax # Læg indhold af register 'rdx' til 'rax' (dest til højre)

Vi tilføjer et "q" for "quad" for at ligne x86 som bruges i bogen.

Lad os da lige udføre en instruktion

00010011 00010011 Orq %rdx,%rbx # bitvis or 'rdx' til 'rbx'



Vi får brug for konstanter

Vi vil gerne kunne introducere konstanter i beregningerne. Det er super praktisk! For eksempel

```
{  
    long vv = 1000;  
    vv = vv + 45;  
}
```

bliver til

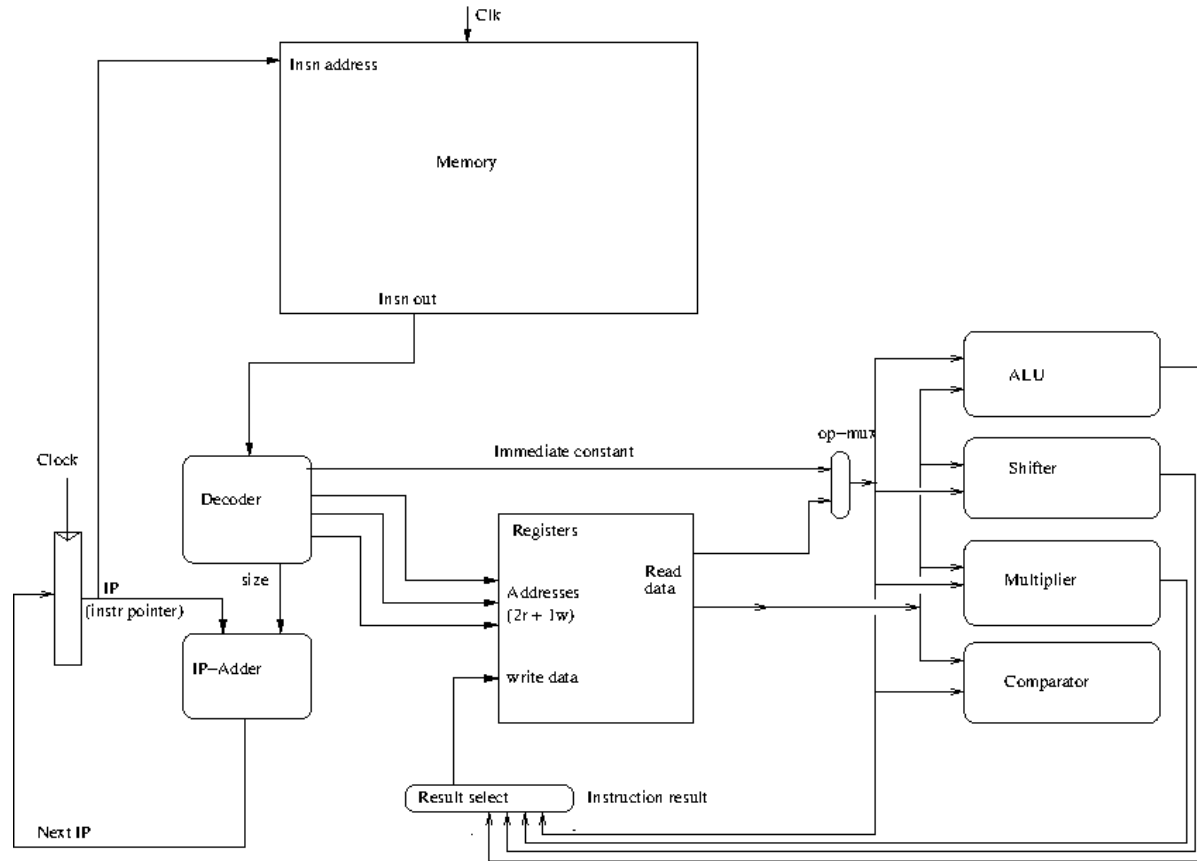
```
Movq $1000,%rax  
Addq $45,%rax
```

Nye instruktioner indkodes således:

0101aaaa dddd0000 ii...32...ii	imm/register arithmetic: op i,d
01100100 dddd0000 ii...32...ii	movq \$i,d

De nye instruktioner fylder mere.....

En datavej for de nye instruktioner



Hvad har vi tilføjet for at kunne understøtte instruktionerne?

Smartere adresseberegninger

x86 har særlige instruktioner til at beregne adresser. De hedder "leaq" og kan addere flere argumenter. Disse instruktioner gør det særlig bekvemt at lave beregninger der svarer til følgende C konstruktioner:

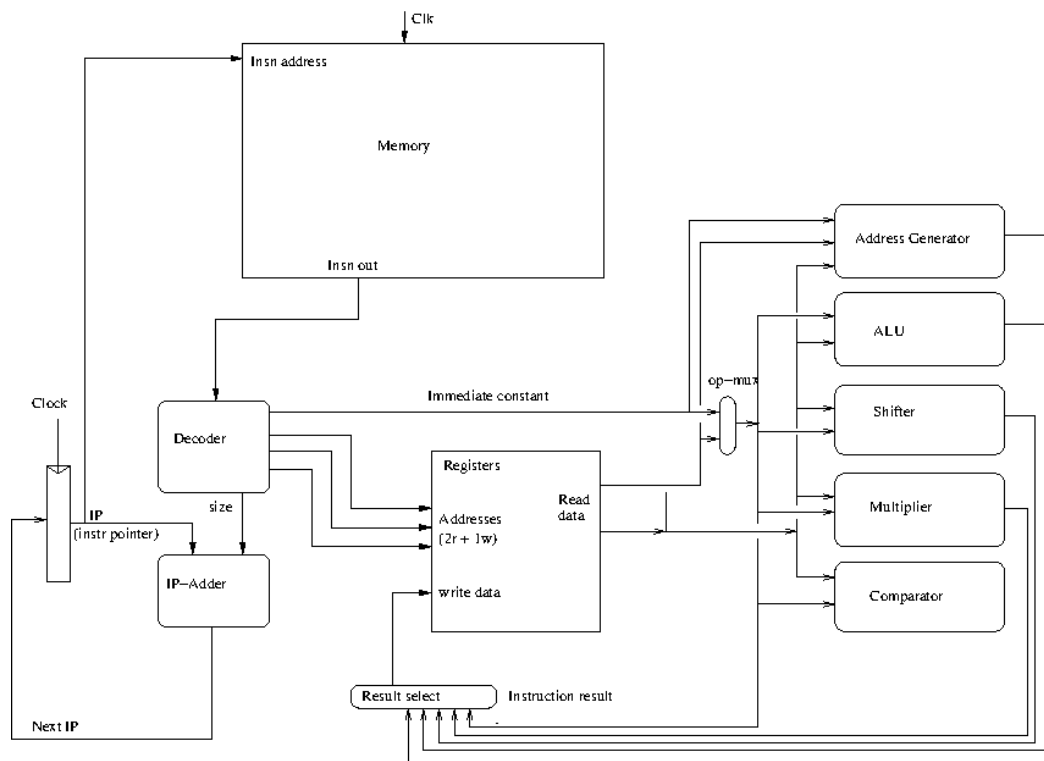
- Bestem adresse på felt i struct
- Bestem adresse på element i vector (af simple typer)
- Bestem adresse på element i en vector (af simple typer) inden i en struct

10000001 ddddssss	leaq (s),d	s -> d
10010010 dddd0000 zzzzvvvv	leaq (,z,(1<<v)),d	z<<v -> d
10010011 ddddssss zzzzvvvv	leaq (s,z,(1<<v)),d	s+(z<<v) -> d
10100100 dddd0000 ii...32...ii	leaq i,d	i -> d
10100101 ddddssss ii...32...ii	leaq i(s),d	i+s -> d
10110110 dddd0000 zzzzvvvv ii...32...ii	leaq i(z,(1<<v)),d	i+(z<<v) -> d
10110111 ddddssss zzzzvvvv ii...32...ii	leaq i(s,z,(1<<v)),d	i+s+(z<<v) -> d

'zzzz' angiver et register (som s og d nævnt tidligere), 'ii...32...ii' er en 32-bit konstant. 'vvvv' er en meget lille konstant som angiver hvor meget der skal skiftes.

Nogle af instruktionerne er også nyttige til almindelig aritmetik på grund af deres større fleksibilitet.

Smartere adresseberegninger



Måske lidt for nemt - men vi tilføjer altså bare en ny "magisk" byggekloids som kan udføre de her instruktioner.

Kan du påpege andre ændringer der (implicit) må indgå i denne nye arkitektur?

Adgang til lageret

Registrene i datavejen er ikke nok. Vi har brug for meget mere plads til data. Al adgang til data i lageret er baseret på en pointer (lager adresse) som befinder sig i et register. Lageret er byte-addresseret (Men den maskine vi bygger nu kan kun læse/skrive quad-words (YUCK!))

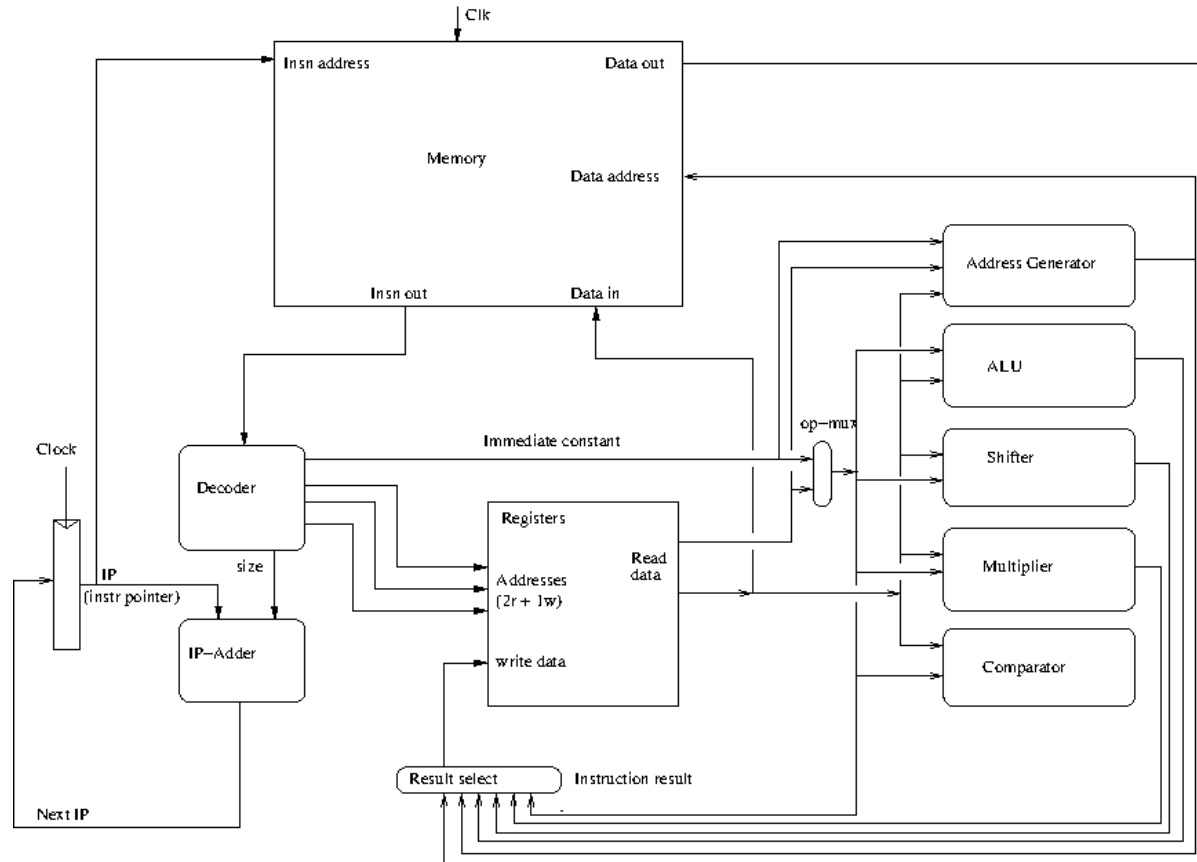
Nye instruktioner:

00110001 ddddssss	movq (s),d
00111001 ddddssss	movq d,(s)
01110101 ddddssss ii...32...ii	movq i(s),d
01111101 ddddssss ii...32...ii	movq d,i(s)

Pointeren er i '(s)'. Bemærk at brugen af 's' og 'd' for disse instruktioner er fastlagt af placeringen af registrene i instruktionsformatet, og ikke om de er "source" eller "destination". Det kan forvirre.

På linie med x86 kaldes alle instruktionerne 'movq'.. men i daglig tale kaldes nogle af dem 'load' og nogle 'store'. Hvilke er mon 'load' og hvilke 'store'

En datavej med adgang til lageret (for data)



Fra C til Prime (for lagertilgang)

```
givet: long* p; long akku;  
{  
    akku += *p;  
}
```

Vi placerer 'p' i '%rbp' og 'akku' i '%rax':

```
movq (%rbp),%r12  
addq %r12,%rax
```


Fra C til Prime (II)

Lidt mere kompliceret (adgang til et felt i en struct):

```
givet: struct { long a; long b;} *p; long akku;  
{  
    akku += p->b;  
}
```

Vi placerer igen 'p' i '%rbp' og 'akku' i '%rax':

```
movq 8(%rbp),%r12  
addq %r12,%rax
```

Hvorfor '8' ?

Fra C til Prime (III)

Kopiering af en struktur

```
Givet: struct { long a; long b; long c } *a,*b;  
{  
    *a = *b;  
}
```

Vi placerer 'a' i '%r12' og 'b' i '%r13'

```
movq (%r13),%rax  
movq %rax,(%r12)  
movq 8(%r13),%rax  
movq %rax,8(%r12)  
movq 16(%r13),%rax  
movq %rax,16(%r12)
```

Et programeksempel:

Hvad gør dette program?

Start:

```
leaq data,%rsi
movq $0,%rax
movq $0,%rbp
leaq (%rsi, %rax, 8), %r11
movq (%r11), %rdx
addq $1, %rax
addq %rdx,%rbp
leaq (%rsi, %rax, 8), %r11
movq (%r11), %rdx
addq $1, %rax
addq %rdx,%rbp
stop
```

```
.align 8
```

data:

```
.quad 42
.quad 21
```

".align 8": Pseudo inst - næste element placeres på adresse delelig med 8.

".quad Z": Pseudo inst - reserver ord (8 bytes) og placer værdien 'Z' i ordet.

Trick question

Hvilken komponent i datavejen har vi ikke brugt til noget?

Nyttige programmer i en svær tid

<https://github.com/finnschiermer/x86prime>

- "Prasm": Kan assemble et x86prime program til hexadecimal notation
- "Prun": Kan simulere udførelse af et x86prime program
- "Primify": Kan oversætte (med begrænsninger) x86 assembler (output fra gcc) til x86prime assembler.

x86prime er skrevet i OCaml, som er et sprog ret tæt på F#. Der er intet krav om at I skal forstå programmet, I skal bare kunne bruge det.

I kan bruge (en virtuel maskine) med Linux (f.eks Ubuntu eller Mint) for let at kunne installere x86prime.

Alternativt udleverer vi nogle scripts som bruger en service på en af DIKUs maskiner. De virker kun når man er online (og servicen også er), men kræver til gengæld ingen besværlig installation.

Prasm (Prime Assembler)

Prime laver en indkodning af et symbolsk assembler program til et hexadecimalt format:

```
00000000 :          # Start:
00000000 : A45030000000    #   leaq data, %rsi
00000006 : 640000000000    #   movq $0, %rax
0000000c : 644000000000    #   movq $0, %rbp
00000012 : 93B503          #   leaq (%rsi, %rax, 8), %r11
00000015 : 313B            #   movq (%r11), %rdx
00000017 : 500001000000    #   addq $1, %rax
0000001d : 1043            #   addq %rdx, %rbp
0000001f : 93B503          #   leaq (%rsi, %rax, 8), %r11
00000022 : 313B            #   movq (%r11), %rdx
00000024 : 500001000000    #   addq $1, %rax
0000002a : 1043            #   addq %rdx, %rbp
0000002c : 0000            #   stop
00000030 :          #   .align 8
00000030 :          # data:
00000030 : 2a00000000000000 #   .quad 42
00000038 : 1500000000000000 #   .quad 21
```

Det er en stor hjælp :-)

Prun (Prime Run)

Prun kører (simulerer) et program i hex format:

./prun fragment.hex Start -show

Starting execution from address 0x0

00000000 : a4 50 00000030	leaq 0x30, %rsi	%rsi <- 0x30
00000006 : 64 00 00000000	movq \$0, %rax	%rax <- 0x0
0000000c : 64 40 00000000	movq \$0, %rbp	%rbp <- 0x0
00000012 : 93 b5 03	leaq (%rsi, %rax, 8), %r11	%r11 <- 0x30
00000015 : 31 3b	movq (%r11), %rdx	%rdx <- 0x2a
00000017 : 50 00 00000001	addq \$1, %rax	%rax <- 0x1
0000001d : 10 43	addq %rdx, %rbp	%rbp <- 0x2a
0000001f : 93 b5 03	leaq (%rsi, %rax, 8), %r11	%r11 <- 0x38
00000022 : 31 3b	movq (%r11), %rdx	%rdx <- 0x15
00000024 : 50 00 00000001	addq \$1, %rax	%rax <- 0x2
0000002a : 10 43	addq %rdx, %rbp	%rbp <- 0x3f
0000002c : 00 00	stop	

Terminated by STOP instruction

Spørgsmål og (forhåbentlig) Svar