# Lektion 2

## En Simpel Computer

Emil Lykke Diget

Computerarkitektur og Operativsystemer
Syddansk Universitet

Viden

- **Kombinatoriske logiske kredse**
- Memorytyper
- Memoryinterface incl. timing
- Adressedekodning
- Interrupt og exceptions
- **Computerdesign**
- Register-transfer level
- Datapath
- **Control unit**
- **Instruktionssæt**
- Pipeline
- Cache
- Processer og tråde
- Context switch
- Inter-process synkronisering og kommunikation, kritiske sektorer og semaphores

Færdigheder

- Redegøre for principperne og algoritmerne bag operativsystemets centrale funktioner
- **Forstå opbygningen af en moderne CPU**
- Kende de almindeligt forekomne memorytyper
- Forstå centrale begreber omkring et operativsystems afvikling af et program

Kompetencer

- Implementere operativsystemsfunktioner i et RTOS (Real Time Operating System)

- Lektion 1: Kombinatoriske Logiske Kredse
- Lektion 2: En Simpel Computer
- Lektion 3: Hukommelse
- Lektion 4: Mikroarkitektur
- Lektion 5: Micro-assembly og IJVM
- Lektion 6: Optimering af Mikroarkitekturdesign

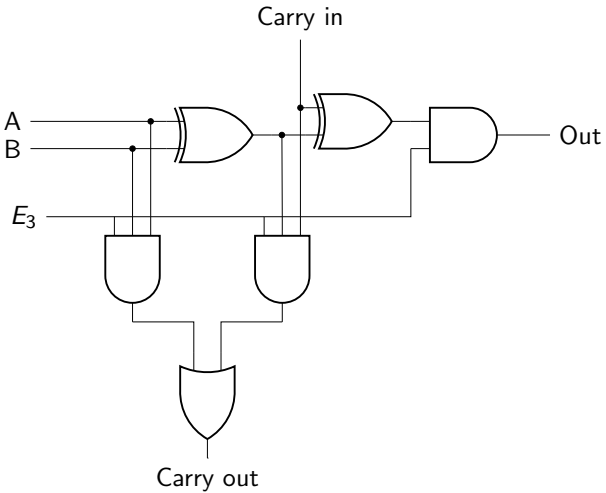
# Repetition

- Opskriv og reducer det boolske udtryk for denne tabel.
- Tegn et kombinatorisk logisk diagram.
- Hvilket komponent?

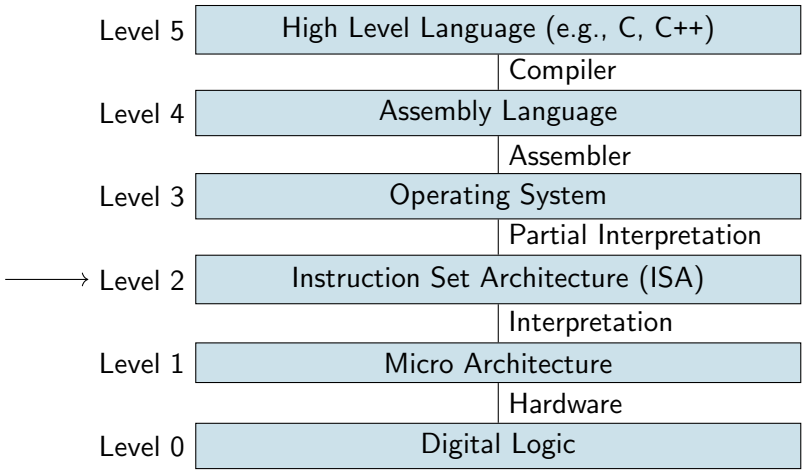
| C | B | A | Y | X |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$$
\begin{aligned}
X &= A\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}\overline{B}C + ABC \\
&= \overline{C}(A\overline{B} + \overline{A}B) + C(\overline{A}\overline{B} + AB) \\
&= \overline{C}(A\overline{B} + \overline{A}B) + C\overline{((A+B)(\overline{A}+\overline{B}))} \\
&= \overline{C}(A\overline{B} + \overline{A}B) + C\overline{(A\overline{A} + A\overline{B} + B\overline{A} + B\overline{B})} \\
&= \overline{C}(A\overline{B} + \overline{A}B) + C\overline{(A\overline{B} + \overline{A}B)} \\
&= \overline{C}(A \oplus B) + C\overline{(A \oplus B)} \\
&= C \oplus (A \oplus B)
\end{aligned}
$$

$$
\begin{aligned}
Y &= AB\overline{C} + A\overline{B}C + \overline{A}BC + ABC \\
&= C(A\overline{B} + \overline{A}B) + AB \\
&= C(A \oplus B) + AB
\end{aligned}
$$


Carry in
A
B
$E_3$
Out
Carry out

# Lagdelt Computermodel

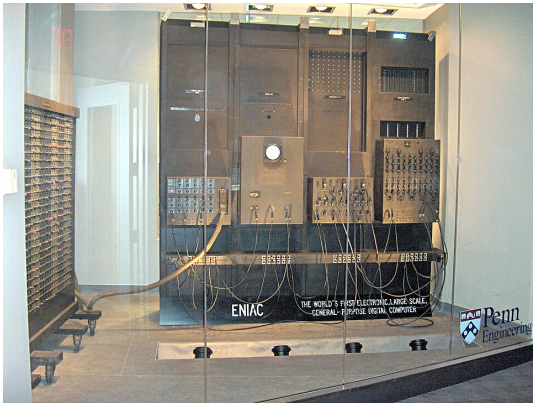


# Indhold

# Indhold

Tidligere var regnemaskiner bygget til at løse en **specifik** og **prædefineret** opgave.

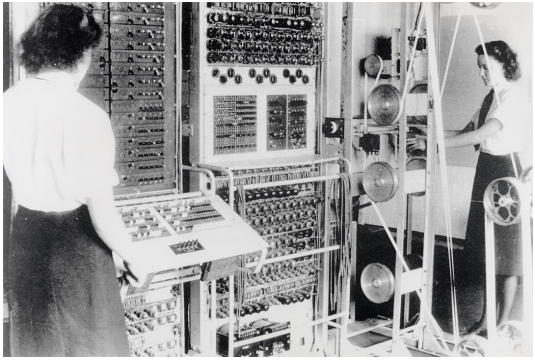
- Reprogrammering er en besværlig process, hvor det fysiske kredsløb skal flyttes rundt, kontakter skal de-/aktiveres og kabler skal kobles til og fra.




Figur: ENIAC (Electronic Numerical Integrator and Computer)

Tidligere var regnemaskiner bygget til at løse en **specifik** og **prædefineret** opgave.

- Reprogrammering er en besværlig process, hvor det fysiske kredsløb skal flyttes rundt, kontakter skal de-/aktiveres og kabler skal kobles til og fra.



Figur: Colossus

# Indhold

# Von Neumann-arkitektur I

Omkring 1950 skrev John von Neumann en serie artikler, der introducerede von Neumann-arkitekturen.


[1] J. Von Neumann, „Theory and organization of complicated automata," **Burks (1966)**, s. 29–87, 1949

[2] J. Von Neumann m.fl., „The general and logical theory of automata. 1951," **John von Neumann-Collected Works**, årg. 5, s. 288–326, 1951

[3] J. Von Neumann, „The theory of automata: Construction, reproduction, homogeneity," **Burks (1966)**, s. 89–250, 1952

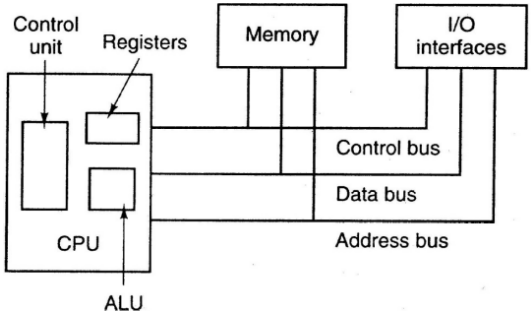
# Von Neumann-arkitektur II

- Disse artikler var med til at forme strukturen af den moderne computer.
- I grunden handler det om, at programinstruktioner er gemt i hukommelsen ved siden af dataen.
- Program og data er altså i den samme hukommelse, og det er programmet, der beslutter, hvad der skal ske.
- Fleksibilitet.
- Arkitekturen har ikke noget prædefineret formål.
- Reprogrammering.

# Von Neumann-arkitektur: Oversigt

Bestående af tre uafhængige komponenter (delsystemer):

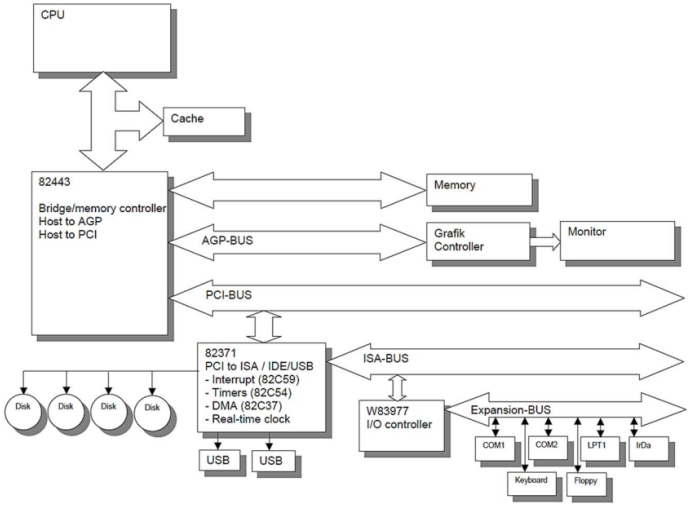
- Central processing unit (CPU)
  - "Hjernen".
  - Kontrolenhed (control unit)
  - Registre
  - Arithmetic logic unit (ALU)
- Hukommelse (memory)
  - Gemmer program og data; random access memory (RAM) og read-only memory (ROM).
- Input/Ouput interface (I/O)
  - Del information med omverdenen; keyboard, skærm, printer, etc.




[4]

# Von Neumann-arkitektur

## Eksempel: Pentium II



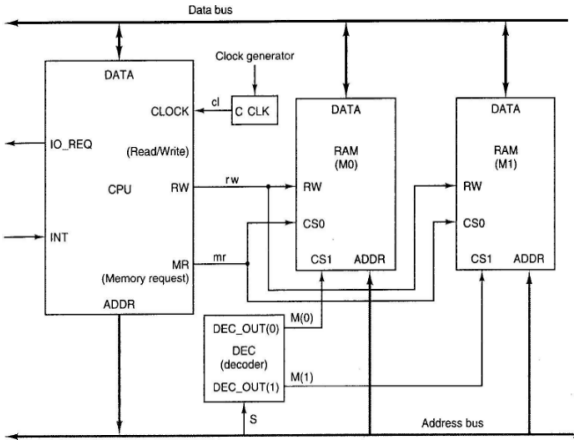
# Indhold

# En Simpel Computermodel

Denne mikrocomputger består af en CPU, en clock generator, en dekoder og to hukommelsesmoduler.

Hvert modul indeholder 8 adresser, hver 8-bit lange.

16 adresser i alt; adressebussen er 4-bit.

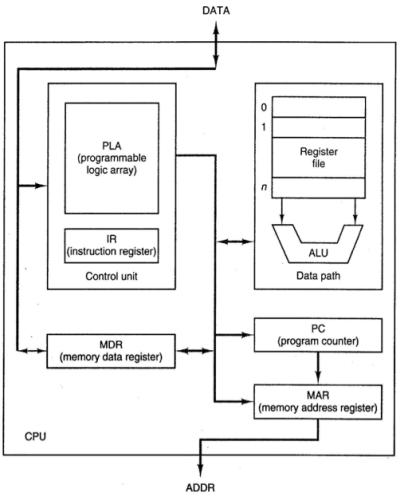


Både data og instruktioner er på data-bussen fra det samme hukommelse.

Data flyttes til **MDR**.
Instruktioner der skal dekodes og eksekveres flyttes til **IR**.

**MAR** udpeger adresses i hukommelsen, der skal læses eller skrives.

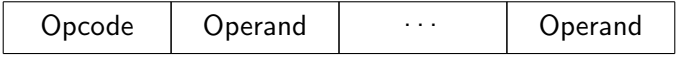
**PC** holder styr på flowet ved at gemme adressen af den næste instruktion.





[4] Zargham, *Computer architecture: single and parallel systems*, 1996

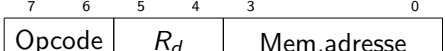
Generelt format af en instruktion:



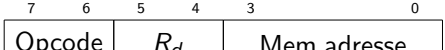
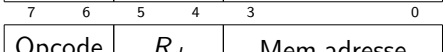
Vores microprocessor har kun fire instruktioner:

| Instruktion | Opcode | Format | Betydning |
|---|---|---|---|

Vores microprocessor har kun fire instruktioner:

| Instruktion | Opcode | Format | Betydning |
|---|---|---|---|
| `LOAD` | 00 |  | $R_d \Leftarrow$ Mem.(adr.) |

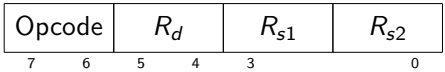
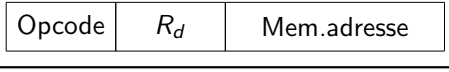
Vores microprocessor har kun fire instruktioner:

| Instruktion | Opcode | Format | Betydning |
|---|---|---|---|
| `LOAD` | 00 |  | $R_d \Leftarrow$ Mem.(adr.) |
| `STORE` | 01 |  | Mem.(adr.) $\Leftarrow R_d$ |

Vores microprocessor har kun fire instruktioner:

| Instruktion | Opcode | Format | Betydning |
|---|---|---|---|
| `LOAD` | 00 |   | $R_d \Leftarrow$ Mem.(adr.) |
| `STORE` | 01 |   | Mem.(adr.) $\Leftarrow R_d$ |
| `ADDR` | 10 |   | $R_d \Leftarrow R_{s1} + R_{s2}$ |

Vores microprocessor har kun fire instruktioner:

| Instruktion | Opcode | Format | Betydning |
|---|---|---|---|
| `LOAD` | 00 |   | $R_d \Leftarrow$ Mem.(adr.) |
| `STORE` | 01 |   | Mem.(adr.) $\Leftarrow R_d$ |
| `ADDR` | 10 |   | $R_d \Leftarrow R_{s1} + R_{s2}$ |
| `ADDM` | 11 |   | $R_d \Leftarrow R_d +$ Mem.(adr.) |

# En Simpel Computermodel

## Opgave (15 min.)

Skriv et program, der lægger to tal sammen, der ligger på adresse 13 og 14 og gem resultatet på adresse 15.

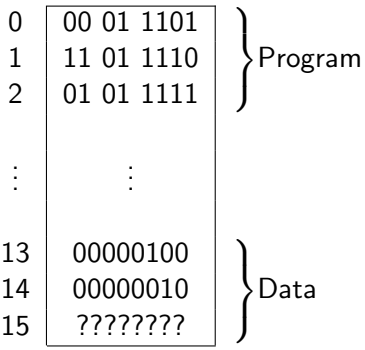
# En Simpel Computermodel

## Opgave (15 min.)

Skriv et program, der lægger to tal sammen, der ligger på adresse 13 og 14 og gem resultatet på adresse 15.

```
LOAD  1, 13 ;       R1 <= mem(13)
ADDM  1, 14 ;       R1 <= R1 + mem(14)
STORE 1, 15 ;       mem(15) <= R1
```

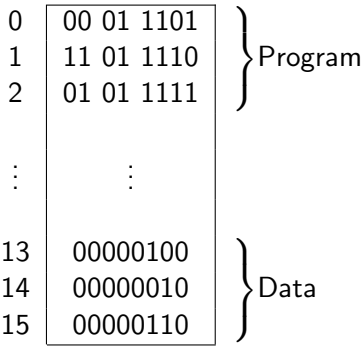
Hukommelse




```
LOAD  1, 13 ;   R1 <= mem(13)
ADDM  1, 14 ;   R1 <= R1 + mem(14)
STORE 1, 15 ;   mem(15) <= R1
```

Adresse og program er i hukommelsen samtidigt.

Hukommelse

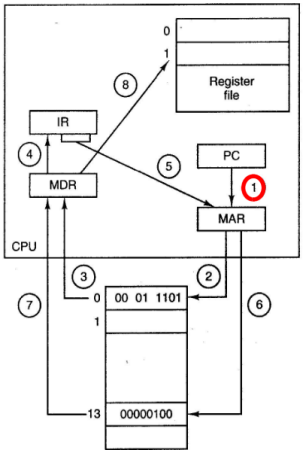



```
LOAD  1, 13 ;   R1 <= mem(13)
ADDM  1, 14 ;   R1 <= R1 + mem(14)
STORE 1, 15 ;   mem(15) <= R1
```

Adresse og program er i hukommelsen samtidigt.

# En Simpel Computermodel

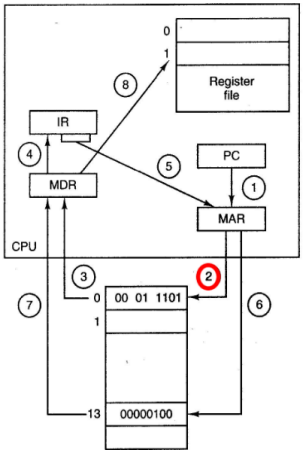
## Eksekvering af `LOAD`




Indholdet af *Program Counter* (PC = 0) kopieres til *Memory Address Register* (MAR).

# En Simpel Computermodel

## Eksekvering af `LOAD`

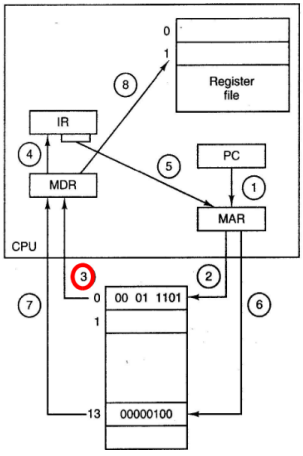



*Memory Address Register* (MAR) peger på den første instruktion i hukommelsen, her er det `LOAD`.


[4] Zargham, *Computer architecture: single and parallel systems*, 1996

# En Simpel Computermodel

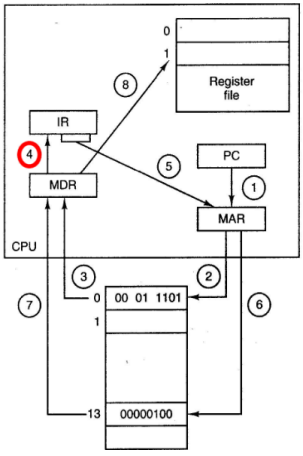
## Eksekvering af `LOAD`




Kontrolenheden (Control Unit) anmoder om at læse indholdet på adresse 0 og bringer indholdet til *Memory Data Register* (MDR).
Samtidigt inkrementeres værdien af PC.

# En Simpel Computermodel

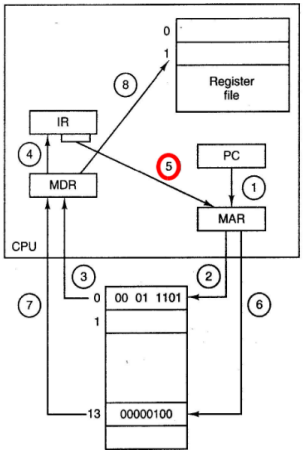
## Eksekvering af `LOAD`



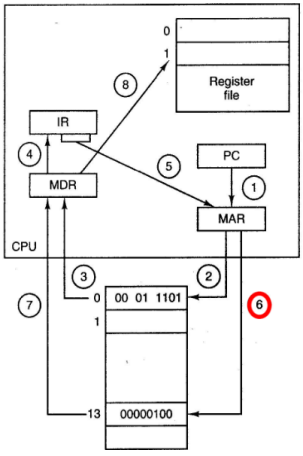

Indholdet af MDR kopieres til *Instruction Register* (IR) hvor instruktionen bliver dekodet, altså 00 $\Rightarrow$ `LOAD`.

# En Simpel Computermodel

## Eksekvering af `LOAD`



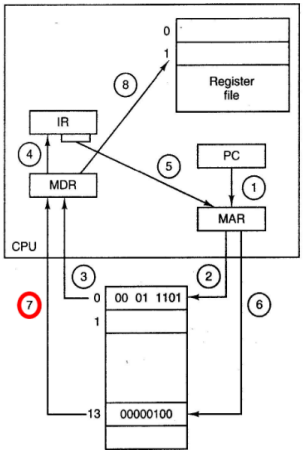

Kontrolenheden vil kopiere de 4 LSBs fra *instruktionsregisteret* (IR) til *Memory Address Register* (MAR) (1101 eller 13).




MAR peger nu på adresse 1101 (13).

# En Simpel Computermodel

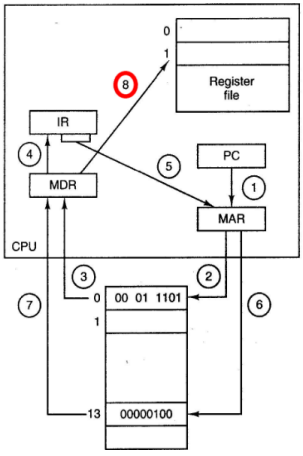
## Eksekvering af `LOAD`




Kontrolenheden læser indholdet af adresse 13 og kopierer det til MDR.

# En Simpel Computermodel

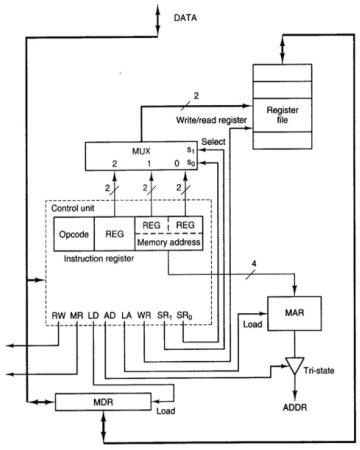
## Eksekvering af `LOAD`




Bit 4 og 5 bestemmer hvilket register, værdien skal gemmes i. Indholdet af MDR kopieres til register 1.

# En Simpel Computermodel

## Kontrolenheden (Control Unit)



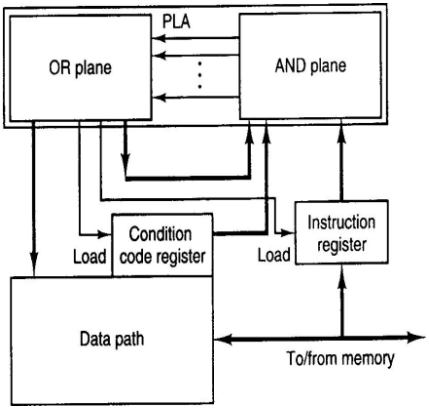

Kontrolenheden har forbindelser til registrene i CPUen.

Kontrolenheden eksekverer en sekvens af operationer, der korresponderer til en specific operation, f.eks. `LOAD`.

Denne sekvens kaldes for **mikro-instruktion**.

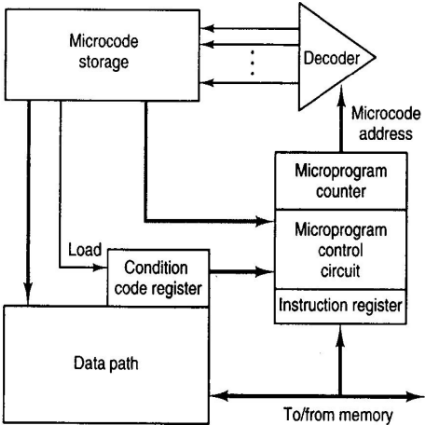


[4] Zargham, *Computer architecture: single and parallel systems*, 1996




Her benyttes *Programmable Logic Array* (PLA) til at implementere kredsløbet.

- **Fordel**: Effektiv.
- **Ulempe**: Ufleksibel; ændringer i kontrolenheden kræver redesign.




Hver instruktion eksekveres som en slags sub-rutine bestående af micro-instruktioner.

- **Fordel**: Fleksibilitet.
  En computer i en computer.
- **Ulempe**: Mindre effektiv.

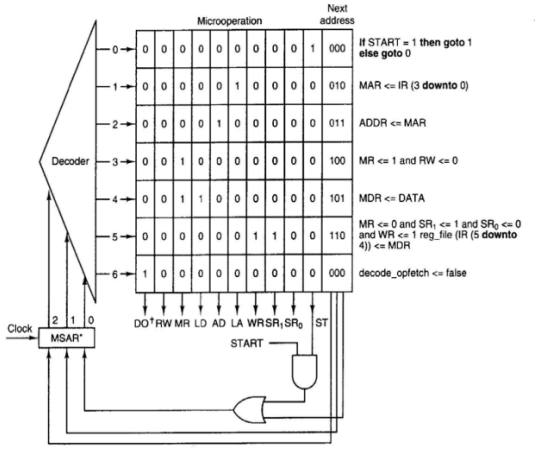
# Design af Kontrolenhed

Man skal tænke over følgende, når man designer størrelsen af mikroinstruktionerne (micro-instrucstion words):

- **Minimér** størrelsen af mikrokode.
- **Minimér** størrelsen af mikroprogram – længden af microkode-hukommelsen.
- **Maximér** fleksibiliteten til at tilføje og ændre microinstruktioner.
- **Maximér** parallel udførsel af microinstruktioner.

# Horisontal Instruktionsdesign

## LOAD



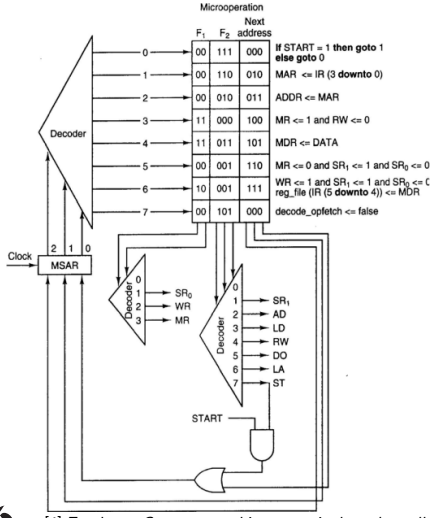

- **Fordel**: Stor fleksibilitet. Kortere eksekveringstid.
- **Ulempe**: Bredere microword-størrelse.
- **Farlig**: Man kan aktivere flere kontrolsignaler på samme tid, der kan skade CPUen.



[4] Zargham, *Computer architecture: single and parallel systems*, 1996

# Vertikal Instruktionsdesign

## LOAD




- **Fordel**: Smallere microword-størrelse.
- **Ulempe**: Mindre fleksibilitet. Længere eksekveringstid.
- Kontrolsignaler, der **ikke** må aktiveres samtidigt, kan grupperes i forskellige dekodere.



[4] Zargham, *Computer architecture: single and parallel systems*, 1996

# Desing af Instruktionssæt

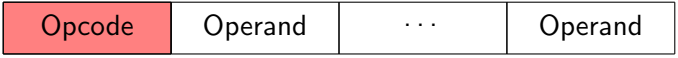
Design af instruktionssæt er vigtigt, når man designer en processor.


Følgende skal overvejes:

- Hvor mange instruktioner er nødvendige?
- Hvilke slags operationer er nødvendige?
- Hvor mange operand-felter og hvilke typer skal tillades i hver instruktion?

# Hvor mange instruktioner er nødvendige?

Kan hjælpe med at bestemme størrelsen af opcode-feltet.




- Flere instuktioner $\Leftrightarrow$ bredere opcode-felt.
- Hver operation fylder mere i hukommelsen.
- Et program kan skrives med færre og mere præcise instruktioner.

Leder til CISC-designet; Complex Instruction Set Computer.
Kompleksitet kan leder til at hver instruktion kan tage flere clock-cykler, men programmet i helhed kan være hurtigere at køre, da der er færre instruktioner.

# Hvor mange instruktioner er nødvendige?

Kan hjælpe med at bestemme størrelsen af opcode-feltet.


- Færre instruktioner $\Leftrightarrow$ smallere opcode-felt.
- Hver operation fylder mindre i hukommelsen.
- Et program skal skrives med flere instruktioner for at have samme funktionalitet.

Leder til RISC-designet; Reduced Instruction Set Computer.
Simplicitet betyder at instruktioner ofte eksekveres i en clock-cyklus, så hver instruktion er hurtigere at eksekvere.

# Hvor mange instruktioner er nødvendige?

Afvejning mellem at implementere funktionalitet som...

| Instruktion i CPU | Subroutine i program |
|---|---|
| Leder til CISC | Leder til RISC |
| Kort access-tid | Længere access-tid |
| Langsommere end RISC pga. kompleksitet | Hurtigere end CISC pga. simplicitet |
| Mindre program | Større program (med subroutiner eller biblioteker) |
| Større strømforbrug | Mindre strømforbrug |

# Hvilke slags operationer er nødvendige?

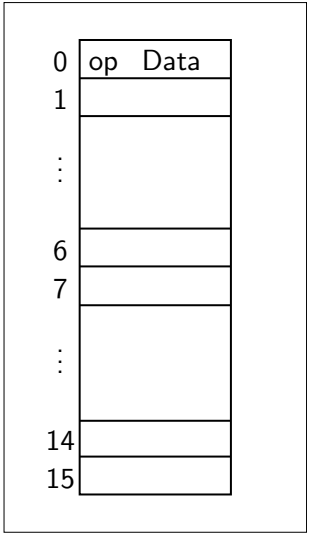
Instruktionssættet kan variere mellem forskellige processorer. De fleste vil have operationer, der kan kategoriseres som følger:

- **Datatransfer**: Flyt (el. kopier) data fra en lokation til en anden.
- **Aritmetik**: Simple aritmiske operationer.
- **Logik**: Boolske operationer.
- **Kontrol**: Relateret til instruktionseksekvering, f.eks. `SKIP`, `RETURN`, `HALT`.
- **System**: Operativsystem-instruktioner, f.eks. systemkald.
- **I/O**: Flyt data ml. hukommelse og eksterne enheder.

# Hvor mange operand-felter og hvilke typer skal tillades i hver instruktion?

Størrelsen af et operand-felt er typisk meget begrænset.
Dog er det nødvendigt at referere til en stor mængde hukommelse.
Dette kan løses ved at bruge forskellige adresse-modes. De mest normale (på engelsk):

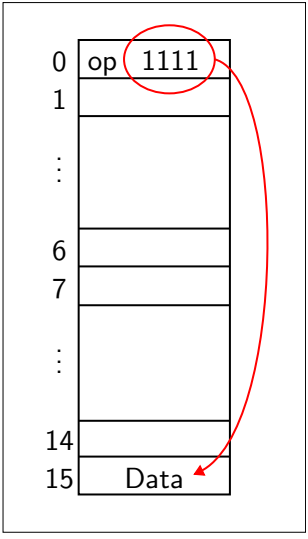
| Adresserings-type | Beskrivelse |
|---|---|
| Immediate | Data |
| Direct | Adresse til data |
| Indirect | Adresse til adresse til data |
| Displacement (indexed) | Adresse til offset og lokal adresse |
| Stack | En slags indirect-type |



# Hvor mange operand-felter og hvilke typer skal tillades i hver instruktion?

Størrelsen af et operand-felt er typisk meget begrænset.
Dog er det nødvendigt at referere til en stor mængde hukommelse.
Dette kan løses ved at bruge forskellige adresse-modes. De mest normale (på engelsk):

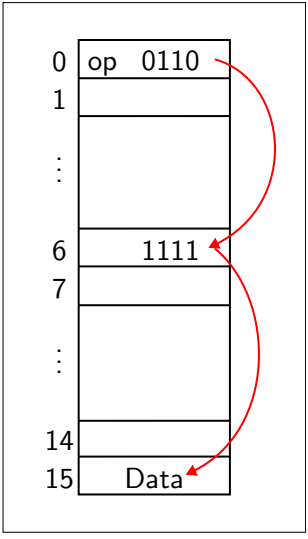
| Adresserings-type | Beskrivelse |
|---|---|
| Immediate | Data |
| Direct | Adresse til data |
| Indirect | Adresse til adresse til data |
| Displacement (indexed) | Adresse til offset og lokal adresse |
| Stack | En slags indirect-type |



# Hvor mange operand-felter og hvilke typer skal tillades i hver instruktion?

Størrelsen af et operand-felt er typisk meget begrænset.
Dog er det nødvendigt at referere til en stor mængde hukommelse.
Dette kan løses ved at bruge forskellige adresse-modes. De mest normale (på engelsk):

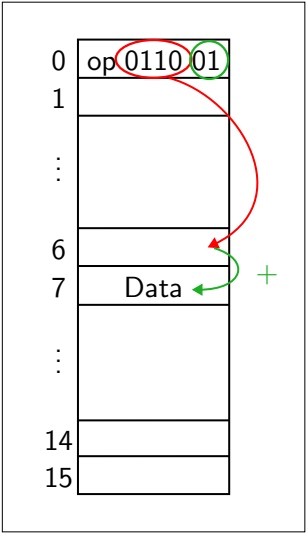
| Adresserings-type | Beskrivelse |
|---|---|
| Immediate | Data |
| Direct | Adresse til data |
| Indirect | Adresse til adresse til data |
| Displacement (indexed) | Adresse til offset og lokal adresse |
| Stack | En slags indirect-type |



# Hvor mange operand-felter og hvilke typer skal tillades i hver instruktion?

Størrelsen af et operand-felt er typisk meget begrænset.
Dog er det nødvendigt at referere til en stor mængde hukommelse.
Dette kan løses ved at bruge forskellige adresse-modes. De mest normale (på engelsk):

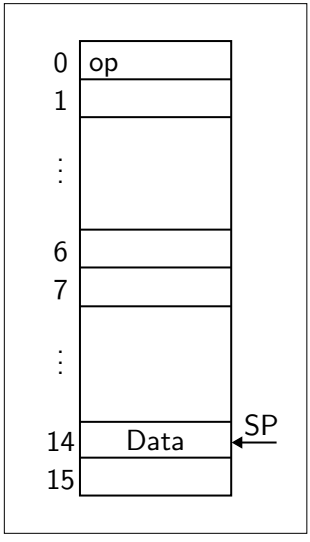
| Adresserings-type | Beskrivelse |
|---|---|
| Immediate | Data |
| Direct | Adresse til data |
| Indirect | Adresse til adresse til data |
| Displacement (indexed) | Adresse til offset og lokal adresse |
| Stack | En slags indirect-type |



# Hvor mange operand-felter og hvilke typer skal tillades i hver instruktion?

Størrelsen af et operand-felt er typisk meget begrænset.
Dog er det nødvendigt at referere til en stor mængde hukommelse.
Dette kan løses ved at bruge forskellige adresse-modes. De mest normale (på engelsk):

| Adresserings-type | Beskrivelse |
|---|---|
| Immediate | Data |
| Direct | Adresse til data |
| Indirect | Adresse til adresse til data |
| Displacement (indexed) | Adresse til offset og lokal adresse |
| Stack | En slags indirect-type |



# Indhold

# Opsummering

- Von-Neumann-arkitektur; en general (simpel) computer bestående af CPU, hukommelse og I/O.
- En simpel CPU
- Funktionen af kontrolenheden og dens design af mikrokode.
- Design af instruktionssæt.
- Adressering af hukommelse.

# Referencer I


[1] J. Von Neumann, „Theory and organization of complicated automata,“ **Burks (1966)**, s. 29–87, 1949.

[2] J. Von Neumann m.fl., „The general and logical theory of automata. 1951,“ **John von Neumann-Collected Works**, årg. 5, s. 288–326, 1951.

[3] J. Von Neumann, „The theory of automata: Construction, reproduction, homogeneity,“ **Burks (1966)**, s. 89–250, 1952.

[4] M. R. Zargham, **Computer architecture: single and parallel systems**. Prentice-Hall, Inc., 1996.

# Opgave

Benyt en digital-logik simulator som f.eks. Digital
`https://github.com/hneemann/Digital`.
Implementér to kontrolenheder med load-operationen; en med horisontal instruktionsdesign og en med vertikal instruktionsdesign.