Lektion 4

Mikroarkitektur

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

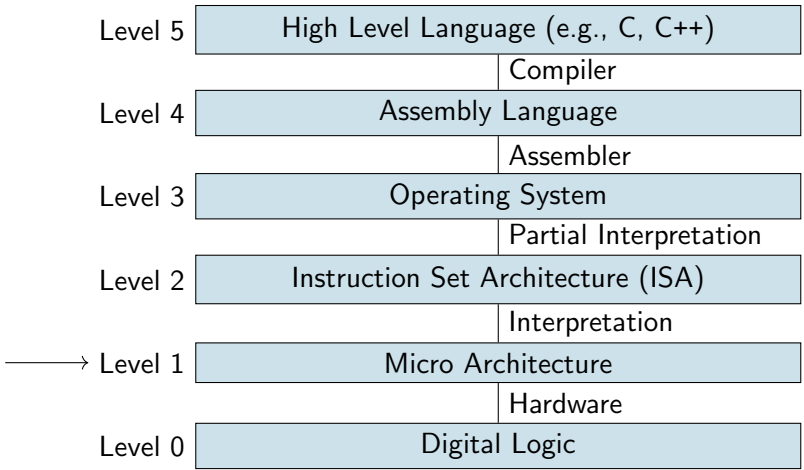
- Implementere operativsystemsfunktioner i et RTOS (Real Time Operating System)

- 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

Indhold

Indhold

Mikroarkitektur-niveau



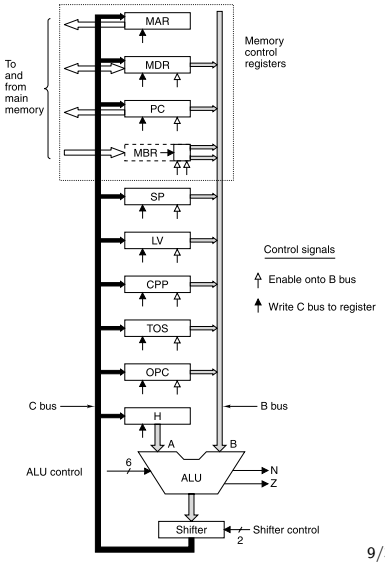
- Mikroarkitekturen implementerer ISA (Instruction Set Architecture), der ligger ovenover.
- Design af mikroarkitekturen afhænger af:
  - den specifikke ISA
  - pris
  - ydelse

- Benytter sig af fetch-decode-execute cyklus af ISA-instrukser.
- Man kan tænke på det som et programmeringsproblem:

```
PC = start_addr
while true:
    instr = memory[PC]                    # find næste instruktion
    PC = PC + 1                           # inkrementer PC
    instr_type = get_instr_type(instr)    # opcode
    data_loc = find_data(instr, opcode) # find data (-1 hvis ingen)
    if (data_loc >= 0):                   #
        data = memory[data_loc]           # fetch data
    execute(opcode, data)                 # eksekver instruktion
```

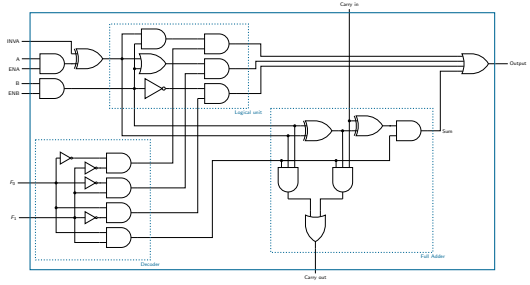
Eksempel: Mic-1

Variable i en computer kaldes for computerens state.
Registre-navne kommer fra ISA-niveauet.
32-bit registre.

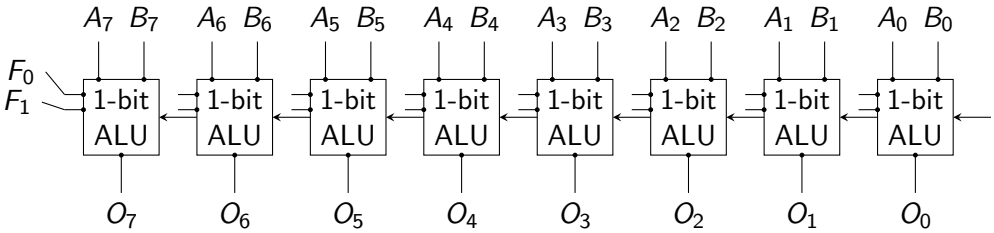


[1] Tanenbaum, Austin og Chandavarkar, *Structured computer organization*, 2013

Mic-1 ALU



F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	$A + B$
1	1	1	1	0	1	$A + B + 1$
1	1	1	0	0	1	$A + 1$
1	1	0	1	0	1	$B + 1$
1	1	1	1	1	1	$B - A$
1	1	0	1	1	0	$B - 1$
1	1	1	0	1	1	$-A$
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	-1



Figur: 8-bit ALU ved serieforbindelse.

MIC1-processoren har en 32-bit ALU.

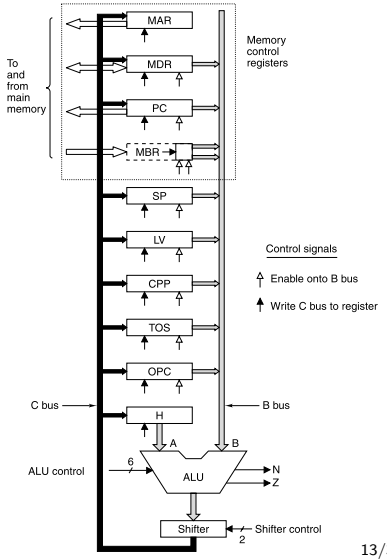
Indhold

Data Path

Definition

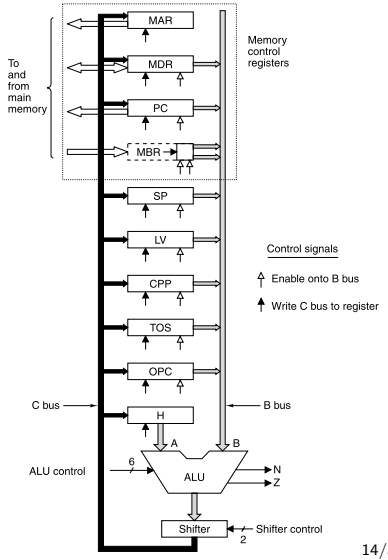
Data-pathen er den del af CPUen, der indeholder ALUen, dens inputs og outputs.

Register	Forkortelse for	Beskrivelse
MAR	Memory Address Register	Pointer til en adresse
MDR	Memory Data Register	Holder data til/fra hukommelse
PC	Program Counter	Pointer til programinstruktion
MBR	Memory Byte Register	Holder byte fået fra PC
SP	Stack Pointer	Pointer til toppen af stacken
LV	Local Variable	Pointer til lokal variabel i hukommelse
CPP	Constant Pool Pointer	Pointer til område i hukommelsen med konstanter
TOS	Top of Stack	Samme indhold som på lokationen af SP
OPC	Optional Program Counter	Et register der kan bruges som midlertidig data-lokation
H	Hold	Holder en værdi til at blive brugt i næste operation

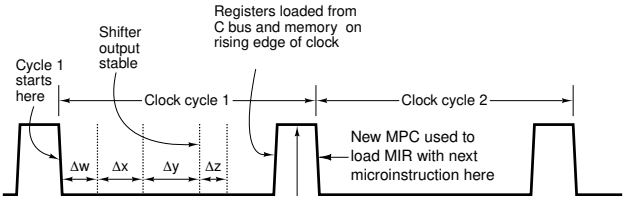


Data Path

- De fleste registre kan skrive til B-bussen.
- ALUens output går i skifteregistret
  - `SSL8`: Shifter indhold 8 bit til venstre. Fylder nederste 8 bit med 0.
  - `SRA1`: Shifter indhold 1 bit til højre. Lader 1. MSB være.
- Muligt at læse og skrive til samme register i en cyklus.



Data Path Timing



En kort puls er lavet i starten af hver clock cycle.

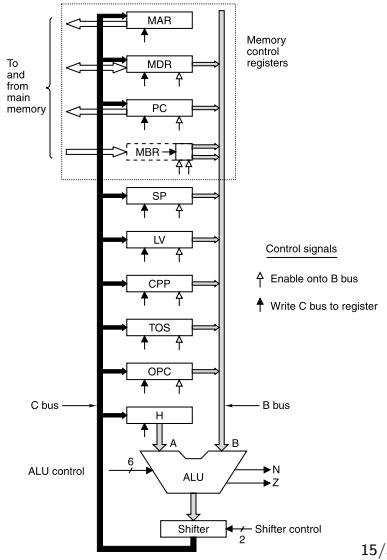
Δw: Kontrolsignal/-bits til ALU sættes op.

Δx: Registre loades på B-bussen.

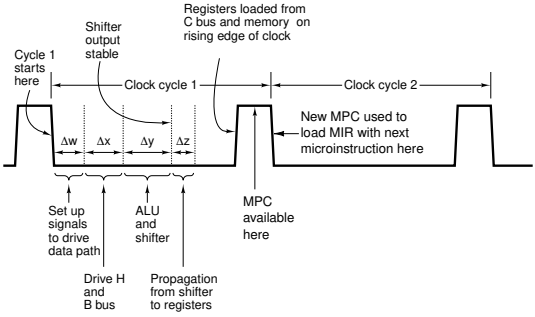
Δy: ALU og shifteren opererer.

Δz: Resultatet propagerer langs C-bussen til registrene.

Ved rising edge gemmes resultatet i registrene.



Data Path Timing



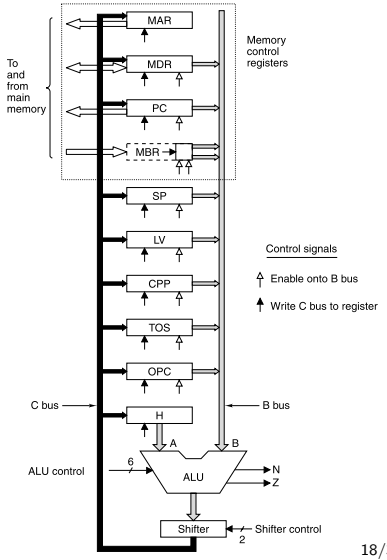
- Husk på, den er bygget op af kombinatoriske kredsløb, så de kører hele tiden.
- Timingen giver en god intuition af, hvad der foregår i løbet af en cyklus.
- Man skal sikre sig, at beregningstiden er kortere end clock-cyklus.

Indhold

Hukommelse

To porte til hukommelsen.

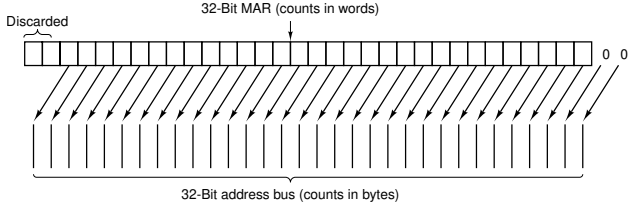
- 32-bit port styret af:
  - `MAR`: Memory Address Register
  - `MDR`: Memory Data Register
- 8-bit port styret af:
  - `PC`: Program Counter
  - `MBR`: Memory Byte Register



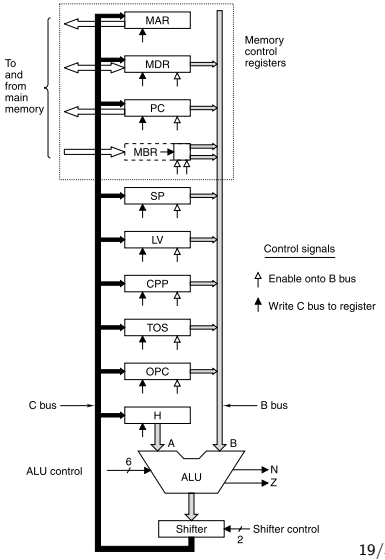
Hukommelsesoperation

To måder at tilgå hukommelse.

- Via `MAR` (Memory Address Register) og `MDR` (Memory Data Register)
  32-bit register, der tilgår hukommelsen i word-størrelse (32-bit), dvs. $2^{32} = 4\,\text{GB}$, dvs. 1 G adresser.



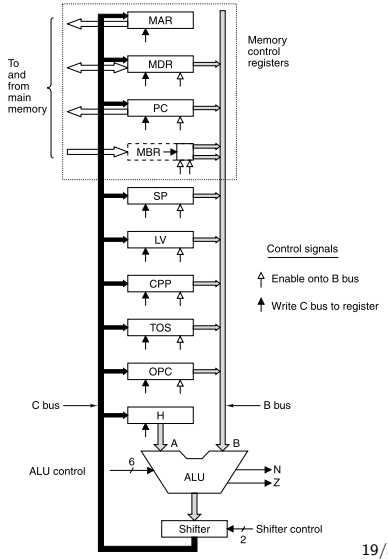
Figur: Den fysiske hukommelse er 8-bit, så dette trick er nødvendigt.



Hukommelsesoperation

To måder at tilgå hukommelse.

- Via `PC` (Program Counter) og `MBR` (Memory Byte Register)
  32-bit register, der tilgår hukommelsen i byte-størrelse (8-bit), dvs. 4 G adresser.



(MBR) Memory Byte Register

Når 8-bit registret MBR skal lægge data på B-bussen, kan det gøres på to forskellige måder.

- Uden fortegn (unsigned)
  MBR lægges på de 8 LSB og de øvre 24 bit sættes til 0.

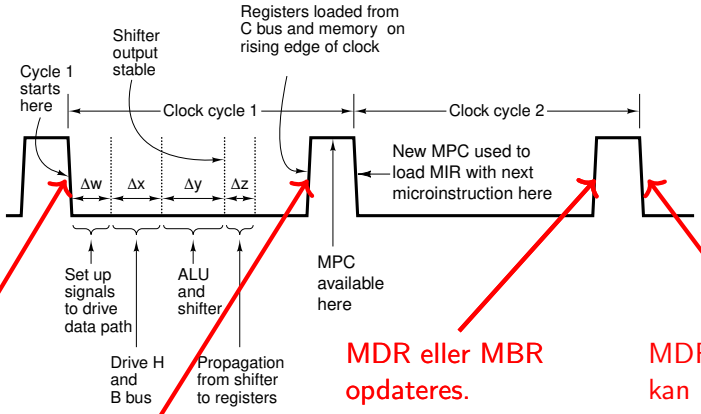
00000000	00000000	00000000	MBR

- Med fortegn (signed)
  MBR lægges på de 8 LSB, og de øvre 24 bit sættes, så de følger MSB.
  Det kaldes for fortegnsforlængelse (sign extension).

00000000	00000000	00000000	0 . . . MBR

11111111	11111111	11111111	1 . . . MBR

Timing af Hukommelsesadgang



Hukommelse læs- eller skrive-signal

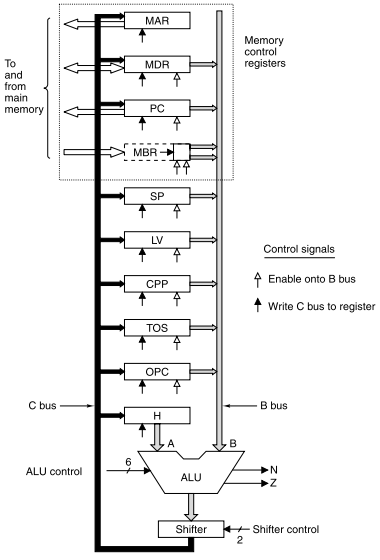
MAR er indlæst, og operationen kan udføres.

MDR eller MBR opdateres.

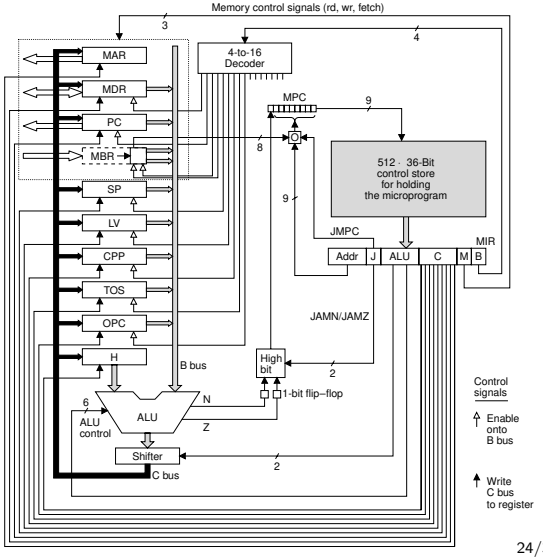
MDR eller MBR kan bruges (2 cykler senere).

Indhold

Mic-1 – Oversigt



Mic-1 – Oversigt

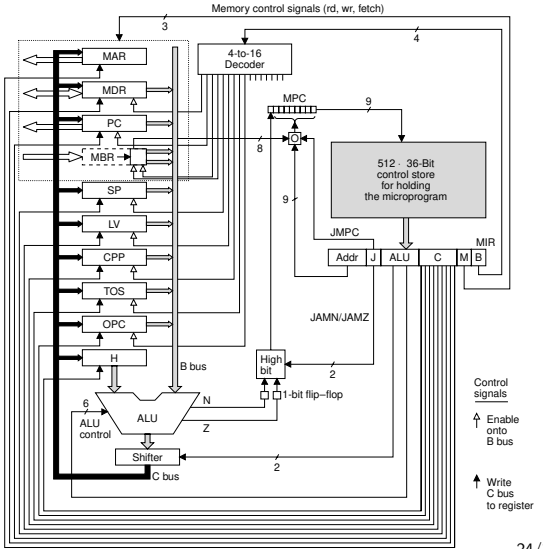


Mic-1 – Oversigt

Control store indeholder alle mikroinstruktioner. Read-only.

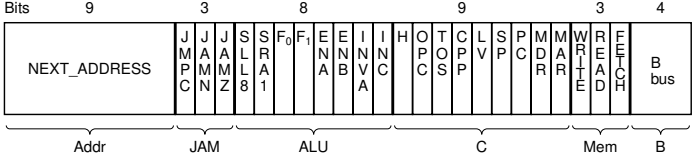
MPC (Micro Program Counter) er ikke en rigtig *Program Counter*.
MIR (Micro Instruction Register) register til at holde på mikroinstruktion.
Mikroinstruktioner eksekveres ikke i rækkefølge – de specificerer deres efterfølger.

Kontrast til et ISA-program, hvor næste instruktion er efter den nuværende.



Micro Instructions Register (MIR)

Mic-1 benytter sig af en særlig intern hukommelse, kaldet Control Store, til at gemme på mikroinstruktioner.
Disse styrer data-flowet og ALU-funktioner.
En mikroinstruktion (36-bit) indlæses fra Control Store til MIR, og bestemmer derved hvad der skal ske i Data Path i den nuværende clock-cyklus.



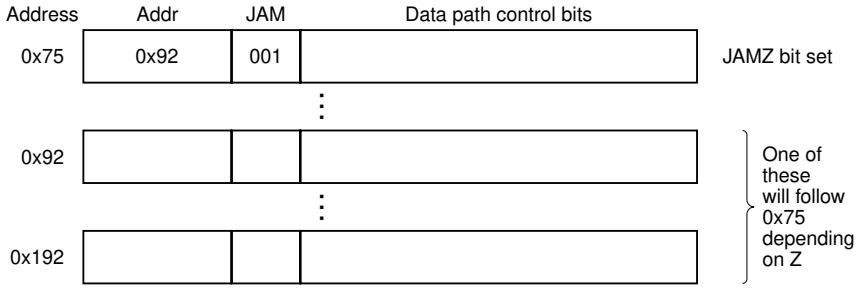
Addr: Indeholder adressen på næste mikroinstruktion.
JAM: Afgør, hvordan næste mikroinstruktion vælges.
ALU: Kontrolsignaler til ALU og shifter.
C: Vælger hvilke registre, der skal læse fra C-bussen.
Mem: Kontrol-signal til tilgang af hukommelsen.
B: Enkoder, der vælger hvilket register, der skriver til B-bussen.

B bus registers

0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBRU	8 = OPC
4 = SP	9-15 none

Forgrening I

Forgrening bestemmes af JAM-feltet i mikroinstruktionen.



High-bit beregnes ved:

$$F = (\text{JAMZ} \cdot Z) + (\text{JAMN} \cdot N) + \text{NEXT\_ADDRESS}[8]$$

Forgrening II

Hvis JMPC $= 1$, bliver de 8 LSB af MIR ORet med indholdet af MBR.

Typisk, vil man kun have inholdet af MBR.

High-bit (9ende) kan også sættes, så NEXT_ADDRESS er enten 0x100 eller 0x000.

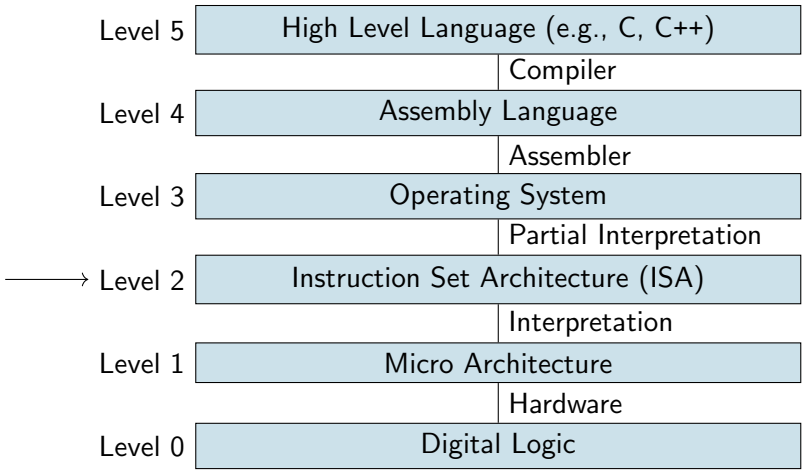
Muliggør en effektiv implementering af forgrening (branching) eller hop (jump).

$$\text{MPC} = \text{NEXT\_ADDRESS} + (\text{JMPC} \cdot \text{MBR})$$

Bruges til bred version af instruktion; f.eks. `istore` (0x36) og `wide_istore` (0x136).

Indhold

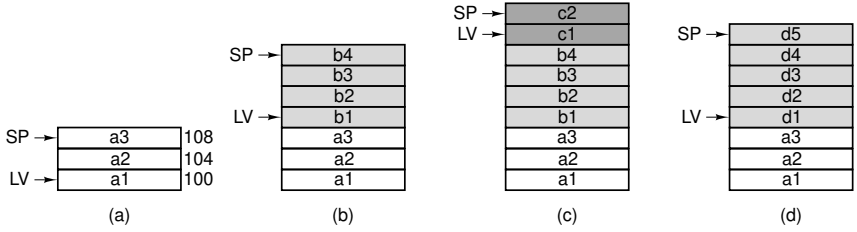
Mikroarkitektur-niveau



Lokale Variable

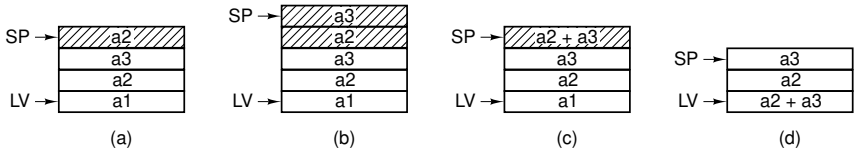
Mic-1 bruger stakken (stack) til lokale variable.
To registre bruges som pointere

- **LV** (Local Variable) peger til den nederste af de lokale variable i den nuværende procedure.
- **SP** (Stack Pointer) peger til den øverste af de lokale variable i den nuværende procedure.



Operand-stack

IJVM (Integer Java Virtual Machine) er en stack-maskine.
Dvs. operationer sker på stakken.



Eksempel: $a_1 = a_2 + a_3$.

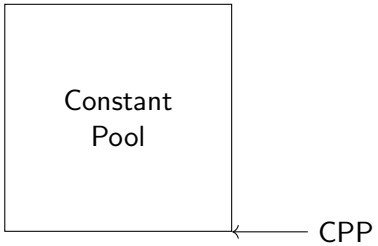
```
PUSH a2     // Læg variabel på stack (a).
PUSH a3     // Læg variabel på stack (b).
ADD         // Adder de øverste to variable og læg resultatet
            // på stacken (c).
POP a1      // Kopier (pop) toppen af stacken til en variabel (d).
```

IJVM Hukommelsesmodel

Data-pathen indeholder registre, der beskriver hukommelsesmodellen, der har tre dele.

Konstant-pølen (constant pool)
Dette område kan ikke skrives af et IJVM-program og består af konstanter, strenge og pointere til andre områder af hukommelsen, der kan refereres til.

CPP adresserer ord (words) (4 byte).

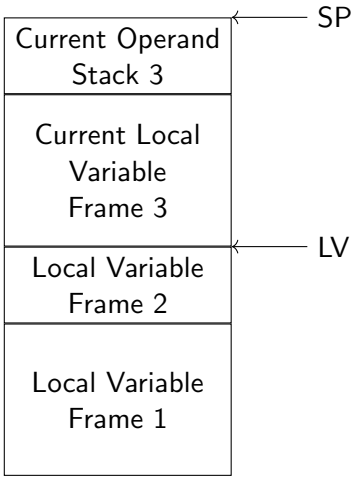


IJVM Hukommelsesmodel

Data-pathen indeholder registre, der beskriver hukommelsesmodellen, der har tre dele.

Lokal-variabel-ramme (local variable frame)
For hver gang en metode køres, allokeres et område af hukommelsen så længe metoden "er i live".

SP og LV adresserer ord (words) (4 byte).

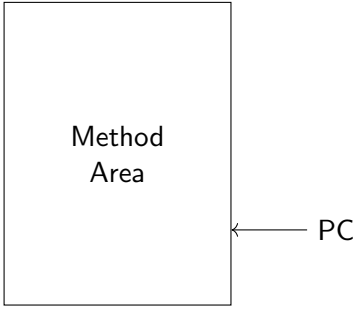


IJVM Hukommelsesmodel

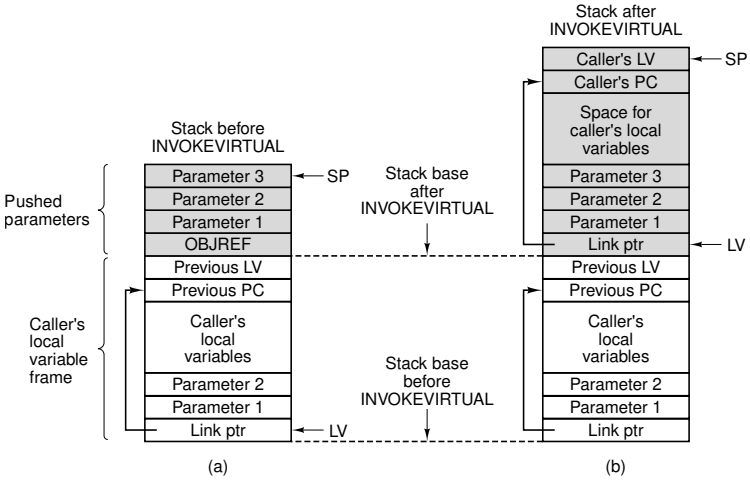
Data-pathen indeholder registre, der beskriver hukommelsesmodellen, der har tre dele.

Metode-område
Område af hukommelsen, hvor programmerne ligger.

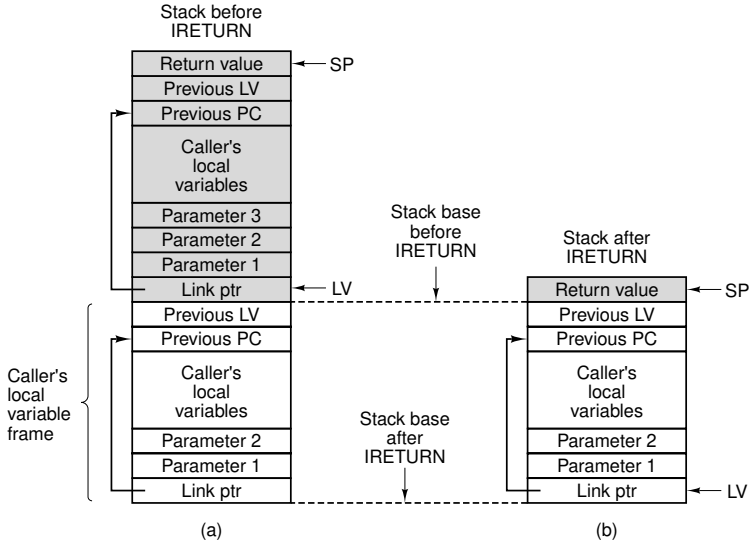
PC adresserer bytes.



Funktionskald



Funktionskald



IJVM Instruktionssæt

Definition (Mnemonic)

"Something such as a very short poem or a special word used to help a person remember something."

`https://dictionary.cambridge.org/dictionary/english/mnemonic`

Hex	Mnemonic	Meaning
0x10	BIPUSH byte	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO offset	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ offset	Pop word from stack and branch if it is zero
0x9B	IFLT offset	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ offset	Pop two words from stack; branch if equal
0x84	IINC varnum const	Add a constant to a local variable
0x15	ILOAD varnum	Push local variable onto stack
0xB6	INVOKEVIRTUAL disp	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE varnum	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC W index	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

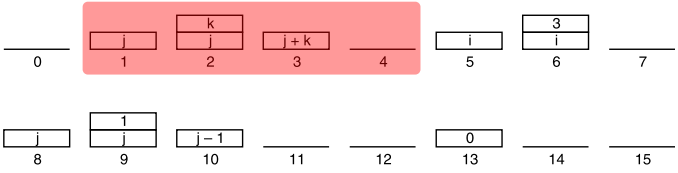
Eksempel

Java → Assembler → Maskinkode

```
i = j + k;
if (i == 3)
    k = 0;
else
    j = j – 1;
```

```
1       ILOAD j          // i = j + k
2       ILOAD k
3       IADD
4       ISTORE i
5       ILOAD i          // if (i == 3)
6       BIPUSH 3
7       IF_ICMPEQ L1
8       ILOAD j          // j = j – 1
9       BIPUSH 1
10      ISUB
11      ISTORE j
12      GOTO L2
13  L1: BIPUSH 0         // k = 0
14      ISTORE k
15  L2:
```

```
0x15 0x02
0x15 0x03
0x60
0x36 0x01
0x15 0x01
0x10 0x03
0x9F 0x00 0x0D
0x15 0x02
0x10 0x01
0x64
0x36 0x02
0xA7 0x00 0x07
0x10 0x00
0x36 0x03
```

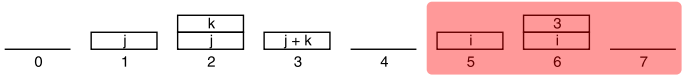


Eksempel

Java → Assembler → Maskinkode

```
i = j + k;
if (i == 3)
    k = 0;
else
    j = j – 1;
```

```
 1       ILOAD j          // i = j + k        0x15 0x02
 2       ILOAD k                             0x15 0x03
 3       IADD                                0x60
 4       ISTORE i                            0x36 0x01
 5       ILOAD i          // if (i == 3)     0x15 0x01
 6       BIPUSH 3                            0x10 0x03
 7       IF_ICMPEQ L1                        0x9F 0x00 0x0D
 8       ILOAD j          // j = j – 1       0x15 0x02
 9       BIPUSH 1                            0x10 0x01
10       ISUB                                0x64
11       ISTORE j                            0x36 0x02
12       GOTO L2                             0xA7 0x00 0x07
13  L1:  BIPUSH 0         // k = 0           0x10 0x00
14       ISTORE k                            0x36 0x03
15  L2:
```

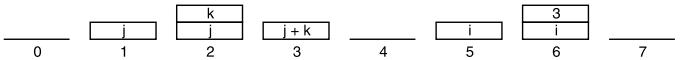


Eksempel

Java → Assembler → Maskinkode

```
i = j + k;
if (i == 3)
    k = 0;
else
    j = j – 1;
```

```
 1      ILOAD j          // i = j + k        0x15 0x02
 2      ILOAD k                             0x15 0x03
 3      IADD                                0x60
 4      ISTORE i                            0x36 0x01
 5      ILOAD i          // if (i == 3)      0x15 0x01
 6      BIPUSH 3                            0x10 0x03
 7      IF_ICMPEQ L1                        0x9F 0x00 0x0D
 8      ILOAD j          // j = j – 1        0x15 0x02
 9      BIPUSH 1                            0x10 0x01
10      ISUB                                0x64
11      ISTORE j                            0x36 0x02
12      GOTO L2                             0xA7 0x00 0x07
13  L1: BIPUSH 0         // k = 0            0x10 0x00
14      ISTORE k                            0x36 0x03
15  L2:
```



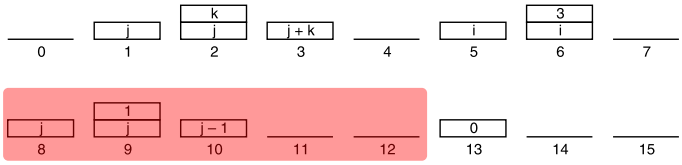
Eksempel

Java → Assembler → Maskinkode

```
i = j + k;
if (i == 3)
    k = 0;
else
    j = j – 1;
```

```
 1      ILOAD j          // i = j + k
 2      ILOAD k
 3      IADD
 4      ISTORE i
 5      ILOAD i          // if (i == 3)
 6      BIPUSH 3
 7      IF_ICMPEQ L1
 8      ILOAD j          // j = j – 1
 9      BIPUSH 1
10      ISUB
11      ISTORE j
12      GOTO L2
13  L1: BIPUSH 0         // k = 0
14      ISTORE k
15  L2:
```

```
0x15 0x02
0x15 0x03
0x60
0x36 0x01
0x15 0x01
0x10 0x03
0x9F 0x00 0x0D
0x15 0x02
0x10 0x01
0x64
0x36 0x02
0xA7 0x00 0x07
0x10 0x00
0x36 0x03
```



Referencer I

[1] A. S. Tanenbaum, T. Austin og B. Chandavarkar, **Structured computer organization**, eng, 6. edition. International edition. Boston, Mass: Pearson, 2013.

Opgave

Den følgende kode bruges til at finde den største fælles divisor (greatest common divisor) af to positive integers, $a, b > 0$.
Oversæt koden til assembler og til maskinekode til Mic-1:

```
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a

    return a
```

Tegn et diagram af stakken i løbet af eksekveringen.