Lektion 5

Micro-assembly og IJVM

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

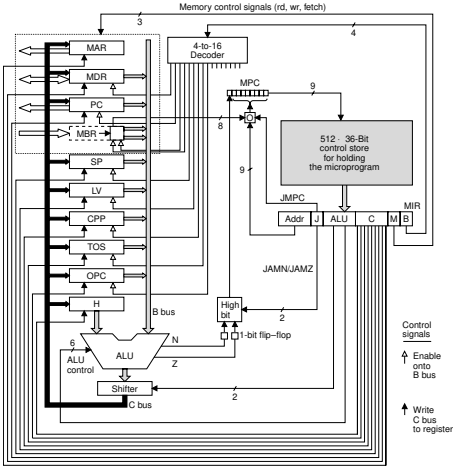
Kompetencer

- 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

Oversigt over Mic-1

Repetition

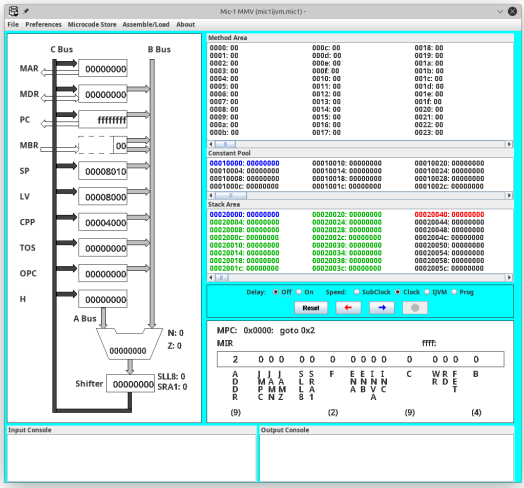


Indhold

Indhold

Simulering af Mic-1

I dagens lektion skal vi benytte os af en simuleret Mic-1, givet i `Mic1MMW`.



Assembler

Simulatoren kan assemble jeres IJVM-program for jer, så I ikke behøver at gøre det selv.

```
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:
```

Program skrevet i filen `lect04_ex.jas` og assembles til `lect04_ex.ijvm`.

```
$ xxd lect04_ex.ijvm
```

```
00000000: 1dea dfad 0001 0000 0000 0004 0000 0040  ...............@
00000010: 0000 0000 0000 0029 1000 5959 3601 3602  .......)..YY6.6.
00000020: 3603 1502 1503 6036 0115 0110 039f 000d  6.....6.........
00000030: 1502 1001 6436 02a7 0007 1000 3603 a7ff  ....d6......6...
00000040: da                                       .
```

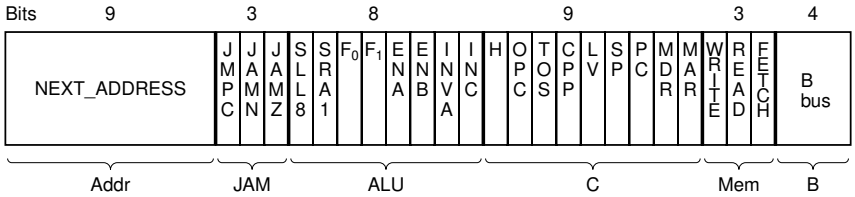
Assembler

Opgave (30 min.)

- Download `Mic1MMV.zip` og pak den ud.
- Åben `Mic1MMV/bin` og kør `Mic1MMV_hr.jar`.
- Skriv programmet fra forrige program i en fil og brug simulatoren til at assemble fra `.jas` til `.ijvm`.
  Kopier f.eks. en af programmerne i `examples`-mappen.
- Lav et hex-dump af den assemblede `.ijvm`-fil og verificer at koden er det, I forventer.

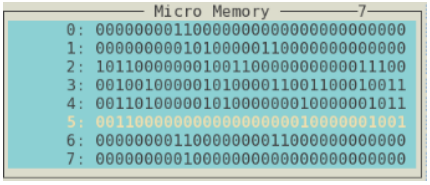
Indhold

Ved at dekode Mikro Instruktion Register (MIR), kan man se præcist hvad en instruktion gør.



Mikroinstruktionerne behøver ikke at være i rækkefølge i control store'en.
Den første mikroinstruktion **skal** placeres på en bestemt adresse.

Det vil være rigtig besværligt at implementere alle IJVM-instruktionerne direkte i control store.



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

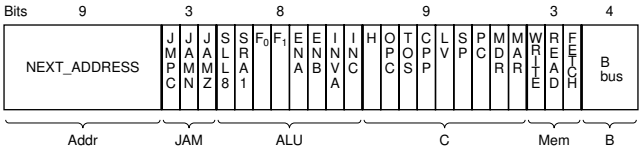
Der er brug for et symbolsk mikroprogrammeringssprog.

Micro Assembly Language (MAL)

Et symbolsk mikroprogrammeringssprog

Skal løse følgende opgaver:

- Oversætte symbolske instruktioner til maskinkode (bit-mønstre).
- Beregne absolutte adresser for symbolske labels af mikroinstruktioner.
- Ansvar for beregning af NEXT_ADDRESS-feltet for hver mikroinstruktion.



En mulighed er bare at skrive alle de signaler, der skal aktiveres.

Example

I én cyklus vil vi inkrementere værdien i SP.
Vi vil også starte en læse-operation, og den næste instruktion skal være den på adresse 122 i control store.

```
ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122
```

`WSP`: Write SP.

Example

I én cyklus vil vi inkrementere værdien i SP.
Vi vil også starte en læse-operation, og den næste instruktion skal være den på adresse 122 i control store.

```
SP = SP + 1; rd
```

`WSP`: Write SP.

Dette sprog kaldes for MAL.

Micro Assembly Language (MAL)

Syntax

- MAL er skræddersyet til Mic-1.
- I hver cyklus (linje i MAL) kan alle registre skrives til.
- Kun ét register kan sende på B-siden af ALUen.
- På ALUens A-side kan man vælge $+1$, 0, -1 og `H`-registret.

Micro Assembly Language (MAL)

Syntaxeksempler

Lovlige:

- `MDR = SP`
- `MDR = H + SP` (kommutativ)

Lovlige:

- `MDR = SP`
- `MDR = H + SP` (kommutativ)

Ulovlige:

- `MDR = SP + MDR` Kan ikke udføres, da en af operander skal være `H`-registret.
- `H = H - MDR` Kan ikke udføres, da `H` skal være subtrahend, altså

$$\text{forskellen} = \text{minuenden} - \text{subtrahend}$$

Lovlige:

- `MDR = SP`
- `MDR = H + SP` (kommutativ)

Ulovlige:

- `MDR = SP + MDR` Kan ikke udføres, da en af operander skal være `H`-registret.
- `H = H - MDR` Kan ikke udføres, da `H` skal være subtrahend, altså

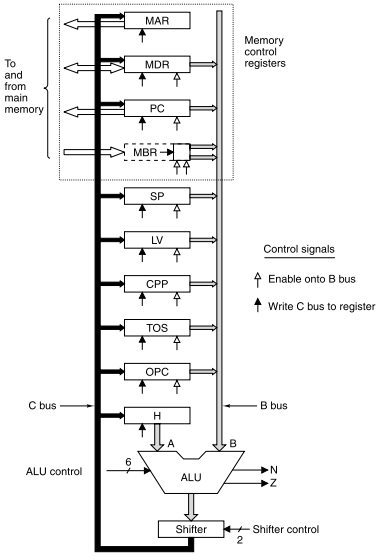
$$\text{forskellen} = \text{minuenden} - \text{subtrahend}$$

Det er op til mikro-assembleren at fejle, hvis der er ulovlige udtryk i mikrokoden.

- Oversigt over alle tilladte operationer.
- SOURCE er alle de registre, der kan skrive til B-bussen, dvs.
  MDR, PC, MBR (signed), MBRU (unsigned), SP, LV, CPP, TOS eller OPC.
- DEST er alle de registre, der kan læse fra C-bussen, dvs. MAR, MDR, PC, SP, LV, CPP, TOS, OPC eller H.
- Shifteren på ALUens output gør at alle kommandoer kan shifte et byte til venstre ved at tilføje <<8 til mikroinstruktionen, f.eks. H = MBR<<8.

DEST = H
DEST = SOURCE
DEST = H
DEST = SOURCE
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE − H
DEST = SOURCE − 1
DEST = −H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = −1

Mic-1-oversigt



To måder at tilgå hukommelsen på:

- `MAR`/`MDR` kan læses og skrives med `rd` og `wr`, 4-byte data.
- `PC`/`MBR` kan læses med `fetch`, 1-byte opcode/operand.

Hvad er galt i følgende kode?

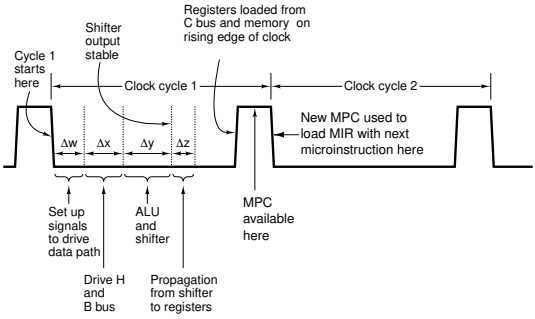
```
MAR = SP; rd
MDR = H
```

To måder at tilgå hukommelsen på:

- `MAR`/`MDR` kan læses og skrives med `rd` og `wr`, 4-byte data.
- `PC`/`MBR` kan læses med `fetch`, 1-byte opcode/operand.

Hvad er galt i følgende kode?

```
MAR = SP; rd
MDR = H
```



Husk, at hukommelsen tager tid at læse, så man skal bruge en cyklus på noget andet.

```
MAR = SP; rd
OPC = MDR
MDR = H
```

- Mikroassembleren fylder normalt `NEXT_ADDRESS`-feltet ud selv.
- Betingelsesløs forgrening (unconditional branch):

```
goto label
goto Main1
```

I teorien skulle hver linje have en `goto`, men hvis det bare er til næste linje, er den implicit.

- Betinget forgrening (conditional branch)
  Bestemt af `Z` og `N` bits i ALUen.
  Testes ved:

  ```
  Z = TOS
  N = TOS
  ```

  Sætter alle bit i C-feltet til 0.
- Sæt `JAMZ`:

  ```
  Z = TOS; if (Z) goto L1; else goto L2
  ```

  Husk på at `L1` skal være `0x100` væk fra `L2`.

- Notation til brug af JMPC-bit.

```
goto (MBR OR value)
```

Gå til opcoden i MBR ORet med en værdi.
Benyttes i Main og i Wide (se senere slides).

Bemærk: Kun de nederste 8-bit er forbundet til MPC.

Indhold

Instruktionssættet

byte, *const* og *varnum* er 1 byte.

disp, *index* og *offset* er 2 byte.

HEX-kolonnen viser adressen på den første mikroinstruktion i control-store.

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*	Adhd 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

IJVM-Implementationen i Mic-1

Den fulde implementering er beskrevet i bogen og på de følgende slides.

Label	Operations	Comments
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch
nop1	goto Main1	Do nothing
iadd1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Add top two words; write to top of stack
isub1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
isub2	H = TOS	H = top of stack
isub3	MDR = TOS = MDR – H; wr; goto Main1	Do subtraction; write to top of stack
iand1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iand2	H = TOS	H = top of stack
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Do AND; write to new top of stack
ior1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
ior2	H = TOS	H = top of stack
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Do OR; write to new top of stack
dup1	MAR = SP = SP + 1	Increment SP and copy to MAR
dup2	MDR = TOS; wr; goto Main1	Write new stack word
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
pop2		Wait for new TOS to be read from memory
pop3	TOS = MDR; goto Main1	Copy new word to TOS
swap1	MAR = SP – 1; rd	Set MAR to SP – 1; read 2nd word from stack
swap2	MAR = SP	Set MAR to top word
swap3	H = MDR; wr	Save TOS in H; write 2nd word to top of stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP – 1; wr	Set MAR to SP – 1; write as 2nd word on stack
swap6	TOS = H; goto Main1	Update TOS

IJVM-Implementationen i Mic-1

bipush1	SP = MAR = SP + 1	MBR = the byte to push onto stack
bipush2	PC = PC + 1; fetch	Increment PC, fetch next opcode
bipush3	MDR = TOS = MBR; wr; goto Main1	Sign-extend constant and push on stack
iload1	H = LV	MBR contains index; copy LV to H
iload2	MAR = MBRU + H; rd	MAR = address of local variable to push
iload3	MAR = SP = SP + 1	SP points to new top of stack; prepare write
iload4	PC = PC + 1; fetch; wr	Inc PC; get next opcode; write top of stack
iload5	TOS = MDR; goto Main1	Update TOS
istore1	H = LV	MBR contains index; copy LV to H
istore2	MAR = MBRU + H	MAR = address of local variable to store into
istore3	MDR = TOS; wr	Copy TOS to MDR; write word
istore4	SP = MAR = SP – 1; rd	Read in next-to-top word on stack
istore5	PC = PC + 1; fetch	Increment PC; fetch next opcode
istore6	TOS = MDR; goto Main1	Update TOS
wide1	PC = PC + 1; fetch;	Fetch operand byte or next opcode
wide2	goto (MBR OR 0x100)	Multiway branch with high bit set
wide_iload1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_iload2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_iload3	H = MBRU OR H	H = 16-bit index of local variable
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = address of local variable to push
wide_istore1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_istore2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_istore3	H = MBRU OR H	H = 16-bit index of local variable
wide_istore4	MAR = LV + H; goto istore3	MAR = address of local variable to store into
ldc_w1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
ldc_w2	H = MBRU << 8	H = 1st index byte << 8
ldc_w3	H = MBRU OR H	H = 16-bit index into constant pool
ldc_w4	MAR = H + CPP; rd; goto iload3	MAR = address of constant in pool

IJVM-Implementationen i Mic-1

Label	Operations	Comments
iinc1	H = LV	MBR contains index; copy LV to H
iinc2	MAR = MBRU + H; rd	Copy LV + index to MAR; read variable
iinc3	PC = PC + 1; fetch	Fetch constant
iinc4	H = MDR	Copy variable to H
iinc5	PC = PC + 1; fetch	Fetch next opcode
iinc6	MDR = MBR + H; wr; goto Main1	Put sum in MDR; update variable
goto1	OPC = PC – 1	Save address of opcode.
goto2	PC = PC + 1; fetch	MBR = 1st byte of offset; fetch 2nd byte
goto3	H = MBR << 8	Shift and save signed first byte in H
goto4	H = MBRU OR H	H = 16-bit branch offset
goto5	PC = OPC + H; fetch	Add offset to OPC
goto6	goto Main1	Wait for fetch of next opcode
iflt1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iflt2	OPC = TOS	Save TOS in OPC temporarily
iflt3	TOS = MDR	Put new top of stack in TOS
iflt4	N = OPC; if (N) goto T; else goto F	Branch on N bit
ifeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
ifeq2	OPC = TOS	Save TOS in OPC temporarily
ifeq3	TOS = MDR	Put new top of stack in TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Branch on Z bit

IJVM-Implementationen i Mic-1

if_icmpeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
if_icmpeq2	MAR = SP = SP – 1	Set MAR to read in new top-of-stack
if_icmpeq3	H = MDR; rd	Copy second stack word to H
if_icmpeq4	OPC = TOS	Save TOS in OPC temporarily
if_icmpeq5	TOS = MDR	Put new top of stack in TOS
if_icmpeq6	Z = OPC – H; if (Z) goto T; else goto F	If top 2 words are equal, goto T, else goto F
T	OPC = PC – 1; goto goto2	Same as goto1; needed for target address
F	PC = PC + 1	Skip first offset byte
F2	PC = PC + 1; fetch	PC now points to next opcode
F3	goto Main1	Wait for fetch of opcode
invokevirtual1	PC = PC + 1; fetch	MBR = index byte 1; inc. PC, get 2nd byte
invokevirtual2	H = MBRU << 8	Shift and save first byte in H
invokevirtual3	H = MBRU OR H	H = offset of method pointer from CPP
invokevirtual4	MAR = CPP + H; rd	Get pointer to method from CPP area
invokevirtual5	OPC = PC + 1	Save return PC in OPC temporarily
invokevirtual6	PC = MDR; fetch	PC points to new method; get param count
invokevirtual7	PC = PC + 1; fetch	Fetch 2nd byte of parameter count
invokevirtual8	H = MBRU << 8	Shift and save first byte in H
invokevirtual9	H = MBRU OR H	H = number of parameters
invokevirtual10	PC = PC + 1; fetch	Fetch first byte of # locals
invokevirtual11	TOS = SP – H	TOS = address of OBJREF – 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = address of OBJREF (new LV)
invokevirtual13	PC = PC + 1; fetch	Fetch second byte of # locals
invokevirtual14	H = MBRU << 8	Shift and save first byte in H
invokevirtual15	H = MBRU OR H	H = # locals
invokevirtual16	MDR = SP + H + 1; wr	Overwrite OBJREF with link pointer
invokevirtual17	MAR = SP = MDR;	Set SP, MAR to location to hold old PC
invokevirtual18	MDR = OPC; wr	Save old PC above the local variables
invokevirtual19	MAR = SP = SP + 1	SP points to location to hold old LV

IJVM-Implementationen i Mic-1

Label	Operations	Comments
ireturn1	MAR = SP = LV; rd	Reset SP, MAR to get link pointer
ireturn2		Wait for read
ireturn3	LV = MAR = MDR; rd	Set LV to link ptr; get old PC
ireturn4	MAR = LV + 1	Set MAR to read old LV
ireturn5	PC = MDR; rd; fetch	Restore PC; fetch next opcode
ireturn6	MAR = SP	Set MAR to write TOS
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto Main1	Save return value on original top of stack

I starten af mikroprogrammet er adresserne til IJVM-instruktionerne defineret.

Mikroprogrammet findes i `examples/MAL/mic1ijvm.mal`.

Ændringer i mikroprogrammet skal også afspejles i konfigurationsfilen i `lib/ijvm.conf`.

```
// labeled statements are "anchored" at
// the specified control store address
.label nop1             0x00
.label bipush1          0x10
.label ldc_w1           0x13
.label iload1           0x15
.label wide_iload1      0x115
.label istore1          0x36
.label wide_istore1     0x136
.label pop1             0x57
.label dup1             0x59
.label swap1            0x5F
.label iadd1            0x60
.label isub1            0x64
.label iand1            0x7E
.label iinc1            0x84
.label ifeq1            0x99
.label iflt1            0x9B
.label if_icmpeq1       0x9F
.label goto1            0xA7
.label ireturn1         0xAC
.label ior1             0xB0
.label invokevirtual1   0xB6
.label wide1            0xC4
.label halt1            0xFF
.label err1             0xFE
.label out1             0xFD
.label in1              0xFC
```

Eksempel

Main

`Main`-instruktionen bliver kaldt i starten, så den skal være så kort som mulig. Tanenbaum har designet hhv. soft- og hardware, så den bare er én linje.

```
Main1   PC = PC + 1; fetch; goto (MBR)
```

- Antager at opcoden ligger i `MBR`.
- Inkrementerer `PC` og fetcher næste byte (opcode eller operand).
- Hopper til næste byte; `goto (MBR)`.

Eksempel

NOP

Den første (og eneste) linje i NOP-instruktionen, nop1, har adressen 0x00.

```
nop1    goto Main1
```

Gør ingenting, udover at bruge en cyklus. Kan være brugbar.

Eksempel

IADD

Lægger de to øverste værdier på stakken sammen og lægger resultatet på stakken.

```
iadd1   MAR = SP = SP - 1; rd
iadd2   H = TOS
iadd3   MDR = TOS = MDR + H; wr; goto Main1
```

`iadd1`: Dekrementerer `SP` og gemmer den i `MAR`. Start læse fra hukommelsen.

`iadd2`: Gem `TOS` i `H`; ALUens venstre input.

`iadd3`: Read er færdig, så det andenøverste ord i stakken er nu i `MDR`, det øverste i `H`.
Læg dem sammen og gem som den nye `TOS` og også toppen af stak i hukommelsen, `MDR`.
Skriv til hukommelsen og gå tilbage til main.

`ISUB`, `IAND` og `IOR` virker nogenlunde på samme måde.

Eksempel

BIPUSH

Lægger en signed *byte* på stakken.

```
bipush1 SP = MAR = SP + 1
bipush2 PC = PC + 1; fetch
bipush3 MDR = TOS = MBR; wr; goto Main1
```

`bipush1`: Inkrementer `SP` og gem den nye adresse i `MAR` til senere skrivning.

`bipush2`: Inkrementer `PC` så næste opcode findes frem.

`bipush3`: Indholdet af `MBR` er det byte, der skal lægges på stakken.
Gem `MBR` i `TOS` og `MDR`. Automatisk fortegnsforlængelse (sign-extension).
Skriv til hukommelsen og gå tilbage til main.

Eksempel

ILOAD

Gemmer lokal variabel nr. *index* (unsiged) på stakken.

ILOAD	INDEX
0x15	

```
iload1  H = LV
iload2  MAR = MBRU + H; rd
iload3  MAR = SP = SP + 1
iload4  PC = PC + 1; fetch; wr
iload5  TOS = MDR; goto Main1
```

Eksempel

ILOAD

```
iload1  H = LV
iload2  MAR = MBRU + H; rd
iload3  MAR = SP = SP + 1
iload4  PC = PC + 1; fetch; wr
iload5  TOS = MDR; goto Main1
```

`iload1`: Gem pointer til de lokale variable `LV` i `H`.

`iload2`: Læg unsigned index i `MBRU` sammen med `H`. Læs fra memory.

`iload3`: Inkrementer `SP` og gem i `MAR`.

`iload4`: Data fra `iload2` er klar. Inkrementer `PC`. Fetch opcode. Skriv.

`iload5`: `MDR` fra `iload2` lægges i `TOS`. Gå tilbage til main.

Eksempel

WIDE ILOAD

Hvis man har mere end 256 lokale variable, kan man benytte sig af WIDE-prefixet.

WIDE	ILOAD	INDEX	INDEX
0xC4	0x15	Byte 1	Byte 2

```
wide1        PC = PC + 1; fetch; goto (MBR OR 0x100)
wide_iload1 PC = PC + 1; fetch
wide_iload2 H = MBRU << 8
wide_iload3 H = MBRU OR H
wide_iload4 MAR = LV + H; rd; goto iload3
```

Eksempel

WIDE ILOAD

```
wide1       PC = PC + 1; fetch; goto (MBR OR 0x100)
wide_iload1 PC = PC + 1; fetch
wide_iload2 H = MBRU << 8
wide_iload3 H = MBRU OR H
wide_iload4 MAR = LV + H; rd; goto iload3
```

`wide1`: Inkrementerer `PC`. Fetcher første index. Går til `0x15 OR 0x100`, dvs. `wide_iload1`.

`wide_iload1`: Inkrementerer `PC`. Fetcher anden index.

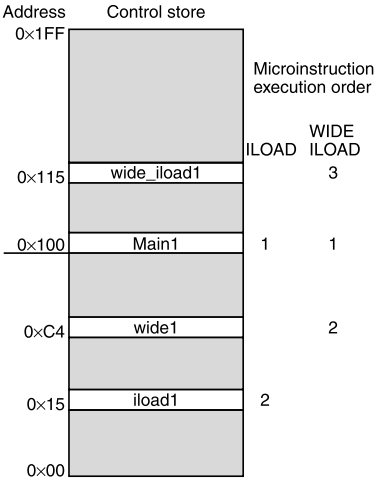
`wide_iload2`: Første index er klar. Shift en byte til venstre og gem i `H`.

`wide_iload3`: Anden index er klar. Logisk or med `H` for at lave 16-bit index.

`wide_iload4`: Regn adresse i hukommelse, læs og fortsæt i `iload3`.

Eksempel

ILOAD og WIDE ILOAD



Eksempel

IINC

Læg en konstant til en lokal variabel.

IINC	INDEX	CONST
0x84		

```
iinc1   H = LV
iinc2   MAR = MBRU + H; rd
iinc3   PC = PC + 1; fetch
iinc4   H = MDR
iinc5   PC = PC + 1; fetch
iinc6   MDR = MBR + H; wr; goto Main1
```

Eksempel

IINC

```
iinc1   H = LV
iinc2   MAR = MBRU + H; rd
iinc3   PC = PC + 1; fetch
iinc4   H = MDR
iinc5   PC = PC + 1; fetch
iinc6   MDR = MBR + H; wr; goto Main1
```

`iinc1`: Kopier `LV` til `H`

`iinc2`: Læg `INDEX` (unsigned) til `LV`, gem i `MAR` og læs.

`iinc3`: Inkrementer `PC` for at få `CONST`.

`iinc4`: Data fra `iinc2` er tilgængelig og gem i `H`.

`iinc5`: Inkrementer `PC` for at få næste opcode.

`iinc6`: Værdi fra `iinc3` ligger nu i `MBR` (`CONST`). Læg `MBR` sammen med `H` og gem. Skriv til hukommelse og gå til main.

Eksempel

GOTO

Betingelsesløst forgrening til et andet sted i IJVM-koden.

GOTO	OFFSET	
0xA7	Byte 1	Byte 2

```
goto1   OPC = PC - 1
goto2   PC = PC + 1; fetch
goto3   H = MBR << 8
goto4   H = MBRU OR H
goto5   PC = OPC + H; fetch
goto6   goto Main1
```

Eksempel

GOTO

```
goto1   OPC = PC - 1
goto2   PC = PC + 1; fetch
goto3   H = MBR << 8
goto4   H = MBRU OR H
goto5   PC = OPC + H; fetch
goto6   goto Main1
```

goto1: Gem adressen til denne IJVM-instruktion (goto1).

goto2: Inkrementer PC for at få 8 MSB af OFFSET. Fetch næste 8 LSB af OFFSET.

goto3: Sign extend og shift 1 byte til venstre og gem i H.

goto4: Or sammen med de nederste 8 LSB i MBRU og gem i H.

goto5: Læg offset til OPC og fetch den nye opcode.

goto6: Gå til main, der hopper til den nye instruktion.

Referencer I

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

Opgave

Idag skal I give jer i kast med Mic-1-simulatoren. Filerne kan findes på itslearning.

- For at forstå, hvad en instuktion gør, kan det være nyttigt at gennemgå, præcis hvad den gør. Tegn, eller beskriv, derfor en tidslinje (hvad er der i hvilke registre til hver tid, etc.) for følgende instruktioner:
  - `ISUB`
  - `DUP`
  - `ISTORE`
  - `IF_ICMPEQ`
- Udvid instruktionssættet i `examples/MAL/mic1ijvm.mal` med en `POP2`- og en `BIPUSH2`-instruktion og skriv et IJVM-program, der bruger dem.