Rechnerarchitektur

Nachklausurtutorium Teil 3

Celina Marie Sieberg

Fachschaft Informatik
Heinrich-Heine-Universität Düsseldorf

12.09.2019

Ablauf der heutigen Übung

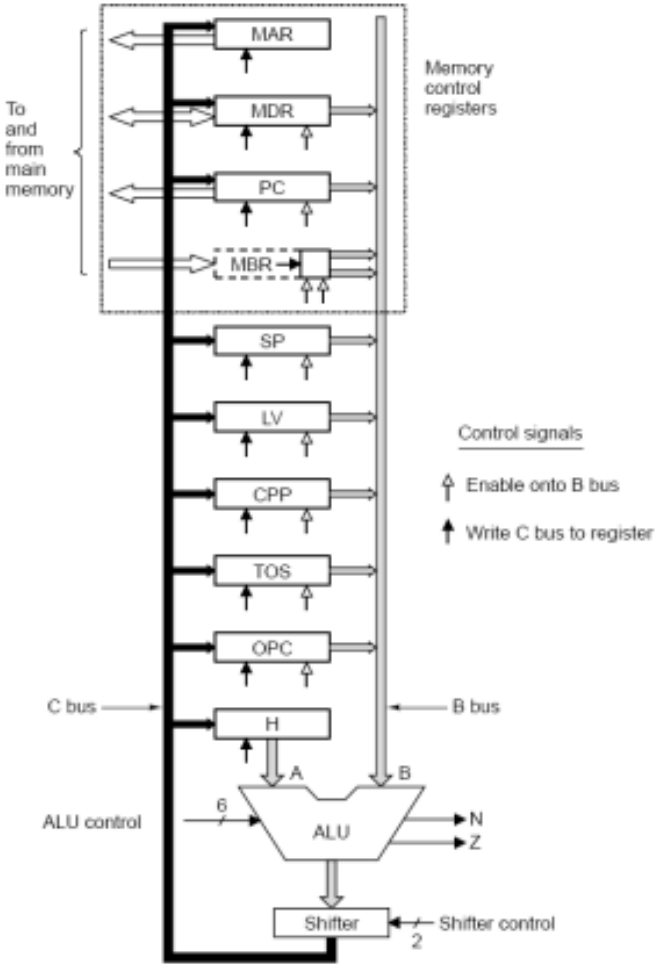
1. ALU
2. Microarchitektur
3. Mal-Instruktionen
4. ISTORE
5. Übungsaufgaben
6. Fragen

Mikroarchitektur

- IJVM
  - Endlosschleife:
    hole nächsten IJVM-Befehl
    führe IJVM-Befehl durch kleines Mikroprogramm aus
- Mikroprogramm
  - besteht aus Mikroinstruktionen
- Mikroinstruktionen
  - Kette von Bits
  - die Bits beschreiben den Zustand von aus(0) oder ein(1)
  - wird in einem Zyklus ausgeführt

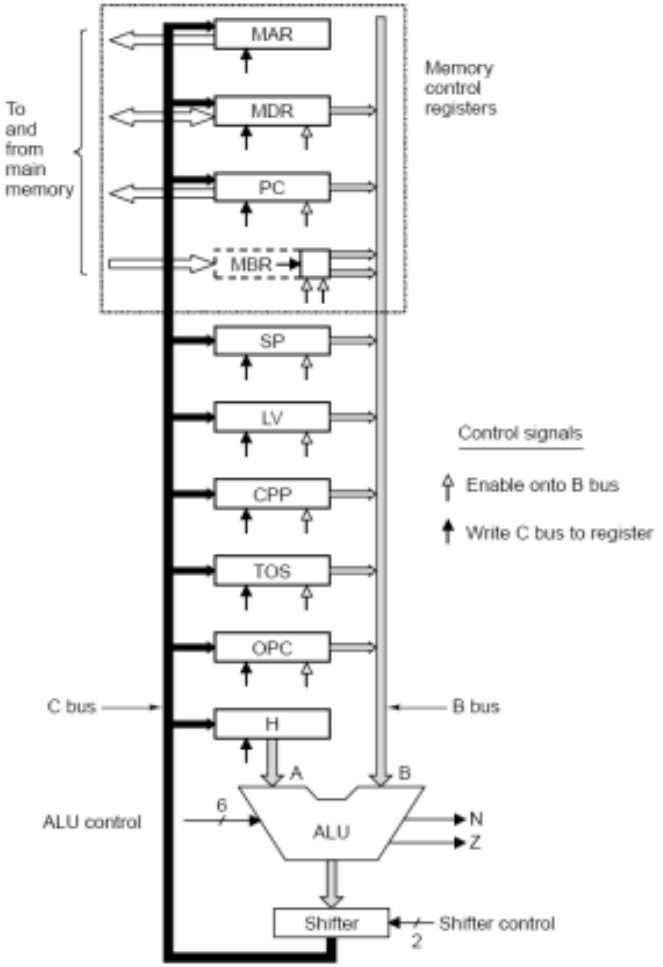
Datenpfad der IJVM

- Register
  - Speicherzugriff:
    - MDR: Memory Data Register
    - MAR: Memory Address Register
    - PC: Program Counter
    - MBR: Memory Buffer Register
  - SP: Stack Pointer
  - LV: Local Variable
  - CPP: Constant Pool Pointer
  - TOS: Top Of Stack
  - OPC: Offset Program Counter
  - H: Help register

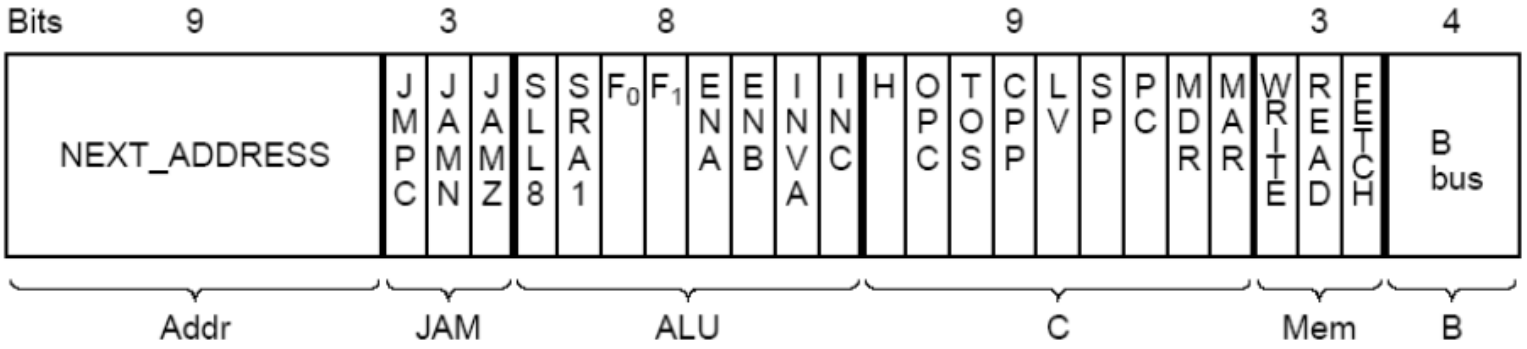


Datenpfad der IJVM

- Busse
  - B: Inhalt eines Registern auf Eingang B der ALU legen (auslesen)
  - C: Ausgang der ALU auf fast alle Eingänge der Register legen (speichern)
- ALU:
  - Ergebnis geht an den Shifter
- Shifter:
  - modifiziert das Ergebnis
  - 2 Steuersignale



Mikroinstruktionen



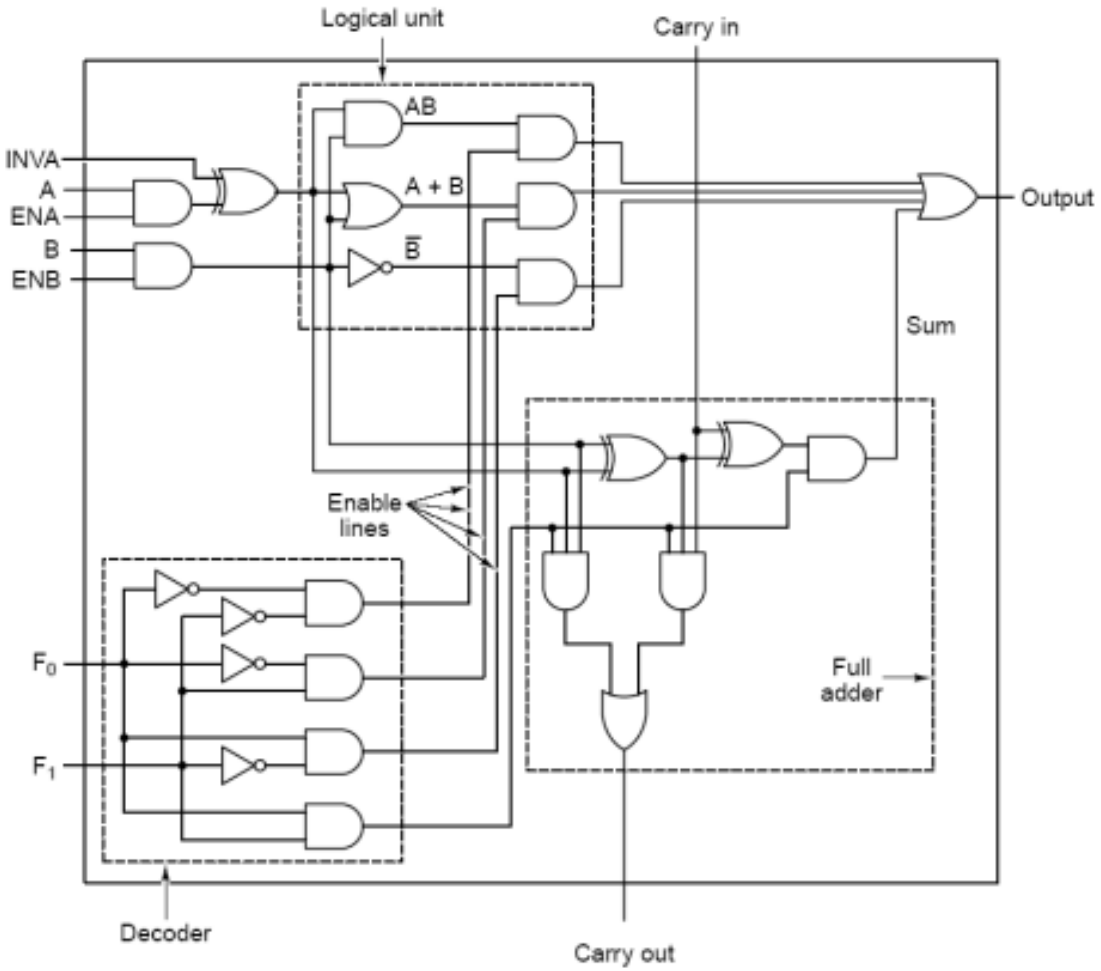
B bus registers

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

- Eine Mikroinstruktion besteht aus 36 Bits:
  - Address: 9 Bits
  - JAM (Bedinkte Sprünge): 3 Bits
  - ALU: 8 Bits
  - C (speichern): 9 Bits
  - Mem (Speicherzugriff): 3 Bits
  - B (auslesen): 4 Bits

ALU

6 Eingabgssignale zum Steuern der ALU, 3 Ausgangssignale (Ergenbnis, N-Bit, Z-Bit)



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

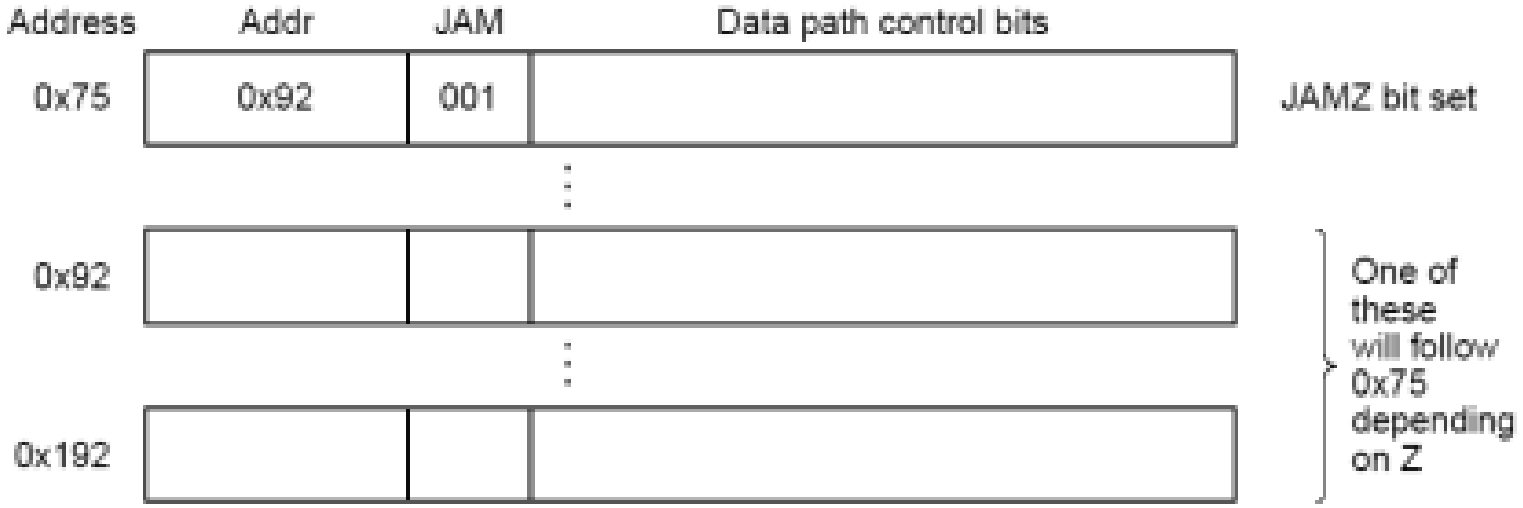
Shifter

- modifiziert das Ergebnis
- 2 Steuersignale:
  - SLL8 (Shift Left Logical 8)
    - verschiebe alle Bits um 8 Stellen nach links
    - letzten 8 Bit = 0
  - SRA (Shift Right Arithmetic 1)
    - verschiebe alle Bits um 1 Stelle nach rechts
    - erste Stelle bleibt gleich

Bedingte Sprünge (JAMN/JAMZ)

- Bei bedinkten Sprüngen ist das vorderste Bit typischerweise 0:
  - NEXT_ ADDRESS[8] = 0
- N und Z werden bei der ALU gesetzt:
  - N → wird gesetzt wenn das Ergebnis der ALU negativ ist
  - Z → wird gesetzt wenn das Ergebnis der ALU 0 ist
- JAMN (Jump on Negative)
  - das vorderste Bit der Adresse und N werden verodert:
    MPC[8] = NEXT_ ADDRESS[8] OR N
    N = 0 → MPC[8] = 0 N = 1 → MPC[8] = 1
- JAMZ (Jump on Zero)
  - das vorderste Bit der Adresse und Z werden verodert:
    MPC[8] = NEXT_ ADDRESS[8] OR Z
    Z = 0 → MPC[8] = 0 Z = 1 → MPC[8] = 1
- → MPC[8] = (JAMZ AND Z) OR (JAMN AND N) OR NEXT_ ADDRESS[8]

Bedingte Sprünge (JAMN/JAMZ)



JMPC

- JMPC (Jump on PC)
  - direkter Sprung zu einer Mikroinstruktion
  - Mikroinstruktion wird aus MBR geladen
  - Mikroinstruktion_ ADDRESS OR NEXT_ ADDRESS (verodert)

Stack

- bestimmter Bereich im Hauptspeicher
  - Enthält Wörter der Größe 32-Bit
  - wächst von unten nach oben
  - Ein Wort kann oben auf den Stack gelegt werden
  - Ein Wort kann oben von dem Stack genommen werden
- SP
  - Zeigt auf das oberste Wort auf dem Stack
- TOS
  - Zeigt auf das oberste Wort des Stacks
  - muss im Mikroprogramm selber gesetzt werden

read und write

- read
  - ”Ließt das oberste Wort vom Stack”
  - Im Detail:
    - bei dem read-Vorgang werden die Register MAR und MDR verwendet
    - in MAR steht die Adresse, wo das Wort steht welches gelesen werden soll
    - in MDR wird das gelesene Wort gespeichert
    - Zusammengefasst: in MDR wird das Word an der Adresse MAR gespeichert
- write
  - SSchreibt ein Wort oben auf den Stack”
  - Im Detail:
    - bei dem write-Vorgang werden die Register MAR und MDR verwendet
    - in MAR steht die Adresse, an die das Wort gespeichert werden soll
    - in MDR steht das Wort, welches gespeichert werden soll
    - Zusammengefasst: an der Adresse MAR wird das Wort in MDR gespeichert
- Da MAR in der Regel = SP gilt, kommt die Situation zustande, dass rd und wr von Stack lesen und auf den Stack schreiben

Program Counter (PC)

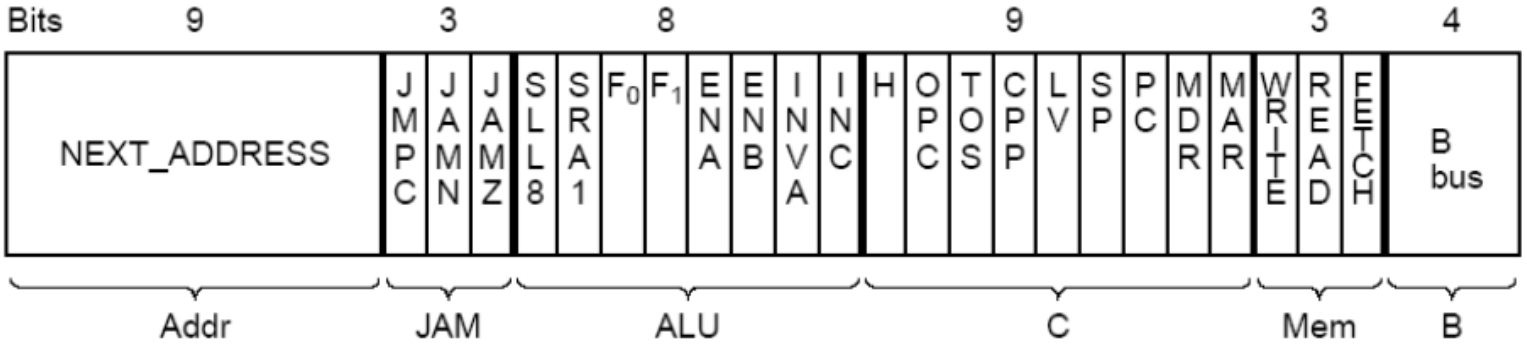
- zeigt auf Adressen anstatt Werte
- zeigt auf die aktuelle Stelle im IJVM Programm
- Holt die Adresse des nächsten Mikroprogrammes:
  - PC = PC + 1; fetch
  - fetch: Signal anlegen
  - nach einem Zyklus Wartezeit ist die Adresse in MBR

Micro Assembly Language (MAL)

- Jede Zeile entspricht einer Mikroinstruktion:

```
0x100: 0x101 000 00110101 000000100 001 0001 PC=PC+1;fetch;goto 0x101
0x101: 0x102 000 00000000 000000000 000 0000 goto 0x102 (warten!)
0x102: 0x000 100 00000000 000000000 000 0000 goto MBR

0x060: 0x061 000 00110110 000001001 010 0100 MAR=SP=SP-1;rd;goto 0x061
0x061: 0x062 000 00010100 100000000 000 0111 H=TOS;goto 0x062
0x062: 0x100 000 00111100 001000010 100 0000 MDR=TOS=MDR+H;wr;goto 0x100
```



B bus registers

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

Micro Assembly Language (MAL)

- Umrechnen nach MAL:
  - NEXT_ ADDRESS: in Hexadezimal umrechnen (Manchmal wird diese direkt in Hexadezimal angegeben)
  - JAM: lesen welcher JAM gesetzt ist
  - ALU: lesen welche Variable gesetzt sind → herrauslesen welche Operation bei der ALU verwendet wird
  - ”C-Bus”: lesen welche Variablen gesetzt sind
  - Mem: lesen welche Variable gesetzt ist
  - ”B-Bus”: in Dezimal umrechnen → herrauslesen welche Variable gesetzt ist

- → C_1 = C_2 = ... = ALU-Operation (mit B=B-Bus, A=H); Mem; goto NEXT_ ADDRESS
  (goto NEXT_ ADDRESS kann entfallen, dann wird NEXT_ ADDRESS + 1 aufgerufen)

Micro Assembly Language (MAL)

- Ergänzung zur ALU-Operation:
  - SLL8 ist gesetzt: ALU-Operation $<< 8$
  - SRA1 ist gesetzt: ALU-Operation $>> 1$ → werdet ihr nicht benötigen, da dieses auch nicht in der Vorlesung besprochen wurde

Micro Assembly Language (MAL)

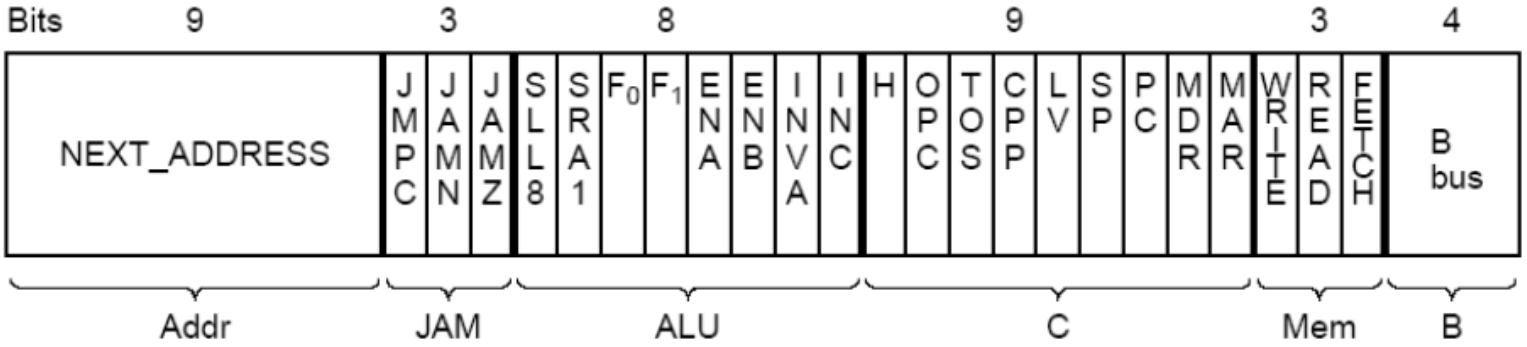
- Wenn JAMZ gesetzt ist
  → $C_1 = C_2 = ... =$ ALU-Operation (mit B=B-Bus, A=H); Mem; if(Z) goto (NEXT_ ADDRESS OR 0x100); else goto NEXT_ ADDRESS
- Wenn JAMN gesetzt ist
  → $C_1 = C_2 = ... =$ ALU-Operation (mit B=B-Bus, A=H); Mem; if(N) goto (NEXT_ ADDRESS OR 0x100); else goto NEXT_ ADDRESS
- Wenn JMPC gesetzt ist
  → $C_1 = C_2 = ... =$ ALU-Operation (mit B=B-Bus, A=H); Mem; goto MBR OR NEXT_ ADDRESS
  - wenn NEXT_ ADDRESS = 0x000 ist → goto MBR
  - wenn MBR = 0x000 ist → goto NEXT_ ADDRESS

Micro Assembly Language (MAL)

- 0x100: 100000001 000 00110101 0000001001 010 0100
  - NEXT_ADDRESS: 0x101
  - JAM: 0
  - ALU: F_0, F_1, *ENB*, *INC* sind gesetzt $\rightarrow B + 1$
  - C: PC
  - Mem: Fetch
  - B: 1 $\rightarrow$ PC
- $\rightarrow$ PC = PC + 1; fetch; goto 0x101

Micro Assembly Language (MAL)

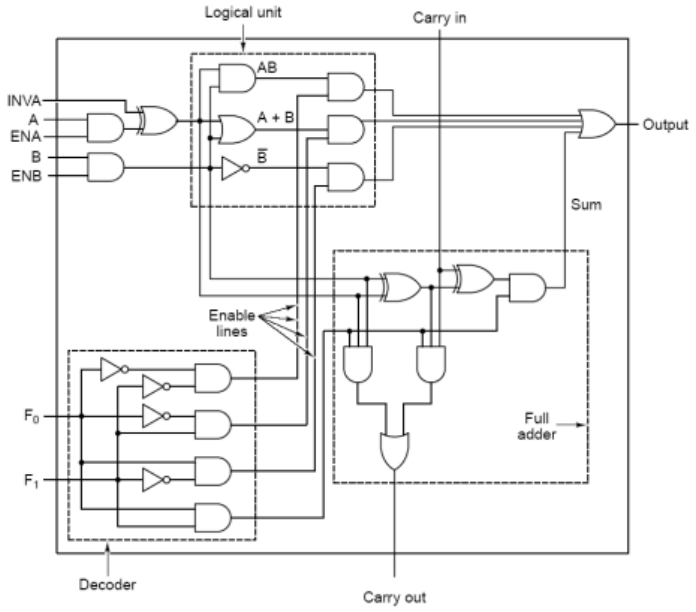
- Beispiel:
  - 0x083: 010000100 000 00010100 1000000000 000 0101
    0x084: 010000101 000 00111100 0000000001 010 0011
    0x085: 010000110 000 00110101 0000000100 001 0001
    0x086: 010000111 000 00010100 1000000000 000 0000
    0x087: 010001000 000 00110101 0000000100 001 0001
    0x088: 100000000 000 00111100 0000000010 100 0010



B bus registers

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

Micro Assembly Language (MAL)



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

Bits	9	3			8								9									3			4
	NEXT_ADDRESS	JMPC	JAMN	JAMZ	SLL8	SRA1	F_0	F_1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B bus
	Addr	JAM			ALU								C									Mem			B

B bus registers

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

Micro Assembly Language (MAL)

- Umrechnen von MAL:
  - C_1, C_2, ... : stehen vor jedem "="
  - ALU-Operation: lesen welche Operation verwendet wird → herrauslesen welche Variablen dafür gesetzt werden
  - B: lesen welche Variable für B eingesetzt ist → Dezimalzahl herraussetzen → in Binär umrechnen
  - Mem: lesen ob wr, rd oder fetch verwendet wird
  - JAM: lesen ob if(Z)..., if(N)... oder goto MBR eingesetzt sind
  - NEXT_ ADDRESS: steht nach (else) goto oder ist NEXT_ ADDRESS + 1
- → NEXT_ ADDRESS JAM ALU C Mem B

Micro Assembly Language (MAL)

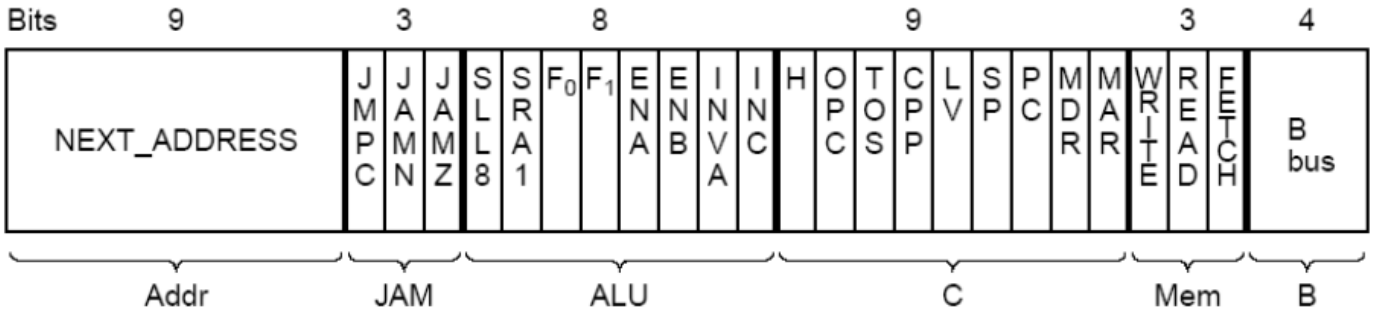
- MAR = SP = SP - 1; rd; goto 0x061
  - C_1 = MAR, C_2 = SP → 000001001
  - ALU-Operation: B - 1 → F_0 = 1, F_1 = 1, ENB = 1, INVA = 1 → 00110110
  - B: B = SP → 4 → 0100
  - Mem: rd wird verwendet → 010
  - JAM: if(Z)..., if(N)... oder goto MBR werden nicht verwendet → 000
  - NEXT_ ADDRESS: 0x061
- → 0x061 000 00110110 000001001 010 0100

Micro Assembly Language (MAL)

- Beispiel:

```
MAR = SP = LV; rd

LV = MAR = MDR; rd
MAR = LV + 1
PC = MDR; rd; fetch
MAR = SP
LV = MDR
MDR = TOS; wr; goto Main1
```



B bus registers

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

MAL-Programm selber schreiben

- Aufgabe:
  Angenommen die Variablen a und b liegen übereinander auf dem Stack, Top of Stack sei a. Geben Sie ein Mikroprogramm in MAL-Notation an, das a NOR b berechnet und das Ergebnis wieder auf den Stapel legt.

- Programm:

```
MAR = SP = SP-1; rd          // lesen (1 Wartezyklus), Stack
                             um eins dekrementieren
H = TOS;                     // TOS (Wert a), nach H kopieren
H = H OR MDR;                // MDR jetzt geladen, Ergebnis
                             nach H schreiben
H = NOT(H);                  // Berechnung abschliessen
MDR = TOS = H; wr goto Main1 // Speichere Ergebnis auf Stack
```

MAL-Programm selber schreiben

- Aufgabe:
  Angenommen die Variablen a und b liegen übereinander auf dem Stack, Top of Stack sei a. Geben Sie ein Mikroprogramm in MAL-Notation an, das a AND b berechnet und das Ergebnis wieder auf den Stapel legt.

IJVM

Byte	Kommando	Kurz-Beschreibung
0x60	IADD	Ersetze die zwei obersten Worte auf dem Stack durch die Summe ihrer Werte.
0x64	ISUB	Ersetze die zwei obersten Worte auf dem Stack durch die Differenz ihrer Werte.
0x7E	IAND	Ersetze die beiden obersten Worte auf dem Stack durch ihre bitweise Boole'sche Und-Verknüpfung.
0x80	IOR	Ersetze die beiden obersten Worte auf dem Stack durch ihre bitweise Boole'sche Oder-Verknüpfung.
0x10	BIPUSH *Byte*	Lege *Byte* auf Stack.
0x15	ILOAD *VarNum*	Lege die durch *VarNum* spezifizierte Variable auf den Stack.
0x36	ISTORE *VarNum*	Entferne den obersten Wert vom Stack und speichere diesen Wert in der durch *VarNum* spezifizierten Variablen.
0x13	LDC_W *Index*	Lege Konstante aus dem Konstanten-Pool auf den Stack.
0x84	IINC *VarNum Const*	Addiere *Const* zu der durch *VarNum* spezifizierten Variablen.
0x57	POP	Entferne das oberste Wort auf dem Stack.
0x59	DUP	Dupliziere das oberste Wort auf dem Stack.
0x5F	SWAP	Vertausche die beiden obersten Worte auf dem Stack.
0x00	NOP	Tue gar nichts.
0xA7	GOTO *Offset*	Erhöhe PC um *Offset*.
0x99	IFEQ *Offset*	Entferne das oberste Wort vom Stack und verzweige, falls dieses Wort gleich 0 ist.
0x9B	IFLT *Offset*	Entferne das oberste Wort vom Stack und verzweige, falls dieses Wort kleiner als 0 ist.
0x9F	IF_ICMPEQ *Offset*	Entferne die beiden obersten Wort vom Stack und verzweige, falls diese gleich sind.
0xB6	INVOKEVIRTUAL *Idx*	Aufruf einer Prozedur.
0xAC	IRETURN	Rückkehr von einer Prozedur.

IJVM - Ergänzung zum Tutorium

- Der Befehl ISUB
  - ”Pop two words from stack; subtract the top word from the second to top word, push the answer”
  - d.h. hier wird das zulezt auf den Stack gelegte Wort von dem davor auf den STack gelegten Wort abgezogen.
  - Beispiel:

    ILOAD a
    ILOAD b
    ISUB

    → berechnet: a - b

Java zu IJVM

- Mit *ILOAD* werden zuerst alle benötigten (für eine Operation) Variablen oder Werte auf den Stack gelegt
  - Achtung: auf die Reihnfolge achten. z.B. beim Subtrahieren
- Operation wird durchgeführt → Ergebnis liegt danach auf dem Stack
- mit *ISTORE* holt man sich die Varable vom Stack
- Besonderheit bei Verzweigungen oder Schleifen:
  - Es wird mitangegeben, in welcher Zeile fortgesetzt werden soll

Java zu IJVM

- $i = j + k;$
  $if(i == 3)$
  $\quad k = 0;$
  $else$
  $\quad j = j - 1;$

```
1        ILOAD j   // i = j + k      0x15 0x02
2        ILOAD k                    0x15 0x03
3        IADD                       0x60
4        ISTORE i                   0x26 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:
```

Java zu IJVM

- Beispiel:
  $if(a == b)$
  $\quad a = 0;$
  $else$
  $\quad b = b + 1;$

IJVM zu Java

- Man guckt sich an welche (für eine Operation) benötigten Variablen oder Werte mit *ILOAD* auf den Stack gelegt wurden
  - Achtung: auf die Reihnfolge achten. z.B. beim Subtrahieren
- Man guckt sich an welche Operation durchgeführt wird
- Man guckt sich an in welcher Variablen das Ergebnis mit mit *ISTORE* gespeichert wird
- Besonderheit bei Verzweigungen oder Schleifen:
  - Man muss im Code hin und her springen und nicht stur von oben nach unten lesen

IJVM zu Java

```
1        ILOAD j   // i = j + k      0x15 0x02
2        ILOAD k                    0x15 0x03
3        IADD                       0x60
4        ISTORE i                   0x26 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:
```

- $i = j + k;$
  $if(i == 3)$
  $\quad k = 0;$
  $else$
  $j = j - 1;$

IJVM zu Java

- Aufgabe:

```
Programm:
1:        ILOAD  a
2:        ILOAD  b
3:        ISUB
4:        IFLT   L1
5:        ILOAD  a
6:        ISTORE i
7:        GOTO   L2
8:   L1:  ILOAD  b
9:        ISTORE i
10:  L2:
```

IJVM ↔ Hexadezimal

- Erste Spalte:
  - Einfaches Herrauslesen aus der Tabelle:

Byte	Kommando	Kurz-Beschreibung
0x60	IADD	Ersetze die zwei obersten Worte auf dem Stack durch die Summe ihrer Werte.
0x64	ISUB	Ersetze die zwei obersten Worte auf dem Stack durch die Differenz ihrer Werte.
0x7E	IAND	Ersetze die beiden obersten Worte auf dem Stack durch ihre bitweise Boole'sche Und-Verknüpfung.
0x80	IOR	Ersetze die beiden obersten Worte auf dem Stack durch ihre bitweise Boole'sche Oder-Verknüpfung.
0x10	BIPUSH *Byte*	Lege *Byte* auf Stack.
0x15	ILOAD *VarNum*	Lege die durch *VarNum* spezifizierte Variable auf den Stack.
0x36	ISTORE *VarNum*	Entferne den obersten Wert vom Stack und speichere diesen Wert in der durch *VarNum* spezifizierten Variablen.
0x13	LDC_W *Index*	Lege Konstante aus dem Konstanten-Pool auf den Stack.
0x84	IINC *VarNum Const*	Addiere *Const* zu der durch *VarNum* spezifizierten Variablen.
0x57	POP	Entferne das oberste Wort auf dem Stack.
0x59	DUP	Dupliziere das oberste Wort auf dem Stack.
0x5F	SWAP	Vertausche die beiden obersten Worte auf dem Stack.
0x00	NOP	Tue gar nichts.
0xA7	GOTO *Offset*	Erhöhe PC um *Offset*.
0x99	IFEQ *Offset*	Entferne das oberste Wort vom Stack und verzweige, falls dieses Wort gleich 0 ist.
0x9B	IFLT *Offset*	Entferne das oberste Wort vom Stack und verzweige, falls dieses Wort kleiner als 0 ist.
0x9F	IF_ICMPEQ *Offset*	Entferne die beiden obersten Wort vom Stack und verzweige, falls diese gleich sind.
0xB6	INVOKEVIRTUAL *Idx*	Aufruf einer Prozedur.
0xAC	IRETURN	Rückkehr von einer Prozedur.

```
      ILOAD j   // i = j + k      0x15 0x02
      ILOAD k                     0x15 0x03
      IADD                        0x60
      ISTORE i                    0x26 0x01
      ILOAD i   // if (i == 3)    0x15 0x01
      BIPUSH 3                    0x10 0x03
      IF_ICMPEQ L1                0x9F 0x00 0x0D
      ILOAD j   // j = j - 1      0x15 0x02
      BIPUSH 1                    0x10 0x01
      ISUB                        0x64
      ISTORE j                    0x36 0x02
      GOTO L2                     0xA7 0x00 0x07
L1:   BIPUSH 0 // k = 0           0x10 0x00
      ISTORE k                    0x36 0x03
L2:
```

IJVM ↔ Hexadezimal

- Zweite und dritte Spalte:
  - Eine zweite Hexadezimalzahl erhalten die Komandos, die einen Wert oder eine Varable mitgegeben bekommen (z.B. ILOAD oder BIPUSH)
    - Bei Werten werden diese einfach in Hexadezimal umgerechnet (z.B. wird die Zahl 3 mitübergeben → lautet die Hexadezimalzahl: 0x03)
    - Bei Variablen werden diese einfach durchnummeriert (am einfachsten ist es die erste verwendete Variable mit der 1 zu nummerieren und die zweite mit der 2 und so weiter).
      D.h. dass wenn die Variable, die mitübergeben wird, bei der Nummerierung die 2 zugewiesen bekommen hat, würde die Hexadezimalzahl 0x02 lauten
  - Eine zweite und dritte Hexadezimalzahl erhalten die Komandos, die einen Sprung enthalten (z.B. IFEQ oder GOTO)
    - siehe nächste Folie

IJVM ↔ Hexadezimal

- Zweite und dritte Spalte:
  - Eine zweite und dritte Hexadezimalzahl erhalten die Komandos, die einen Sprung enthalten (z.B. IFEQ oder GOTO)
    - Die beiden Hexadezimal Zahlen ergeben zusammen eine Entfernung zum Ende des Sprunges in Byte
    - Diese Entfernung wird ab dem Anfang der der Methode gerechnet
    - Jeder "Methodenname" an sich benötigt ein Byte (rot unterstrichen) und die Varablen, die übergeben werden, benötigen 1 Byte, bzw 2 Byte (grün unterstrichen) → zusammenaddiert ergibt dieses die Entfernung, die übergeben werden soll

```
      ILOAD j   // i = j + k      0x15 0x02
      ILOAD k                     0x15 0x03
      IADD                        0x60
      ISTORE i                    0x26 0x01
      ILOAD i   // if (i == 3)    0x15 0x01
      BIPUSH 3                    0x10 0x03
      IF_ICMPEQ L1  ← 2 Byte      0x9F 0x00 0x0D
      ILOAD j   // j = j - 1      0x15 0x02
      BIPUSH 1                    0x10 0x01
      ISUB                        0x64
      ISTORE j                    0x36 0x02
      GOTO L2  ← 2 Byte           0xA7 0x00 0x07
L1:   BIPUSH 0 // k = 0           0x10 0x00
      ISTORE k                    0x36 0x03
L2:
```

IJVM zu Hexadezimal

- Aufgabe:

```
Programm:
1:        ILOAD  a
2:        ILOAD  b
3:        ISUB
4:        IFLT   L1
5:        ILOAD  a
6:        ISTORE i
7:        GOTO   L2
8:   L1:  ILOAD  b
9:        ISTORE i
10:  L2:
```

IJVM zu Hexadezimal

- Lösung:

```
Programm:                  Erklaerung zu (a)      (b)
1:       ILOAD  a          // if(a-b < 0)         (0x15 0x01)
2:       ILOAD  b                                   0x15 0x02
3:       ISUB                                       0x64
4:       IFLT   L1                                  0x9B 0x00 0x0A
5:       ILOAD  a          // false: i = a          0x15 0x01
6:       ISTORE i                                   0x36 0x03
7:       GOTO   L2                                  0xA7 0x00 0x07
8:   L1: ILOAD  b          // true: i = b         (0x15 0x02)
9:       ISTORE i                                  (0x36 0x03)
10: L2:
```

Beispiele MAL-Notation

Label	Operations	Comments
Main 1	`PC = PC + 1; fetch; goto(MBR)`	MBR holds opcode; get next byte; dispatch

-

Label	Operations	Comments
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 in stack
isub2	`H = TOS`	H = top of stack
isub3	`MDR = TOS = MDR - H; wr; goto Main1`	Do subtraction; write to top of stack

-

Label	Operations	Comments
pop1	`MAR = SP = SP -1; rd`	Read in next-to-top of stack
pop2		Wait for new TOS to be read from memory
pop3	`TOS = MDR; goto Main1`	Copy new word to TOS

-

Label	Operations	Comments
Main 1	`PC = PC + 1; fetch; goto(MBR)`	MBR holds opcode; get next byte; dispatch

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

Label	Operations	Comments
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

Label	Operations	Comments
Main 1	`PC = PC + 1; fetch; goto(MBR)`	MBR holds opcode; get next byte; dispatch

Label	Operations	Comments
swap1	`MAR = SP - 1; rd`	Read 2nd word from stack; set MAR to SP
swap2	`MAR = SP`	Prepare to write new 2nd word
swap3	`H = MDR; wr`	Save new TOS; write 2nd word to stack
swap4	`MDR = TOS`	Copy old TOS to MDR
swap5	`MAR = SP - 1; wr`	Write old TOS to 2nd place on stack
swap6	`TOS = H; goto (MBR1)`	Update TOS

Label	Operations	Comments
Main 1	`PC = PC + 1; fetch; goto(MBR)`	MBR holds opcode; get next byte; dispatch

Label	Operations	Comments
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 = offset of method pointer from CPP
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`	Safe old PC above the local variables
invokevirtual19	`MAR = SP = SP + 1`	SP points to location to hold old LV
invokevirtual20	`MDR = LV; wr`	Save old LV above saved PC
invokevirtual21	`PC = PC + 1; fetch`	Fetch first opcode of new method
invokevirtual22	`LV = TOS; goto Main1`	SET LV to point to LV Frame