13.2
Übergang zur realen Maschine

Bernd Becker – Technische Informatik II

Unterschiede zwischen abstrakter und realer Maschine

1. Bei realer Maschine nur *ein Speicher* M
   für Daten und Befehle.
   M ist *endlich*.
   Für i ∈ { 0, ..., 2^{32}-1 } ist M(i) Inhalt der
   i-ten Speicherzelle.
   Speicherzellen können Elemente aus { 0,1 }^{32}
   aufnehmen.

Unterschiede zwischen abstrakter und realer Maschine (ff)

2. CPU-Register PC, ACC, IN1, IN2 können nur Elemente $w \in \{ 0,1 \}^{32}$ aufnehmen.

   w heißt *Wort*.

   Verschiedene Interpretationsmöglichkeiten von Worten:

   - Binärzahlen (z.B. bei Adressen aus M)
   - 2er-Komplement (z.B. Zahlen bei arithmetischen Operationen)
   - Bitstrings (z.B. bei logischen Operationen, die zusätzlich betrachtet werden)

Unterschiede zwischen abstrakter und realer Maschine (ff)

3. Befehle sind ebenfalls Worte aus $\{ 0,1 \}^{32}$.

   → Parameter $i \in \mathbf{N}_0$ bzw. $i \in \mathbf{Z}$ werden kodiert durch m-stellige Binärzahlen bzw. 2er-Komplementzahlen mit $m < 32$.

   → *Maschinensprache* bzw. *Instruktionssatz*

Schreibweisen:

$b^j = \underbrace{(b,\ldots,b)}_{j \text{ mal}}$ für $b \in \{0,1\}$

$< A > := B$ (A Register oder Speicherzelle,

$B \in \{0, \ldots, 2^{32} - 1\}$)

bedeute $A := bin_{32}(B)$ (Beispiel: <PC> := <PC> +1)

$[A] := B$ (A Register oder Speicherzelle,

$B \in \{-2^{31}, \ldots, 2^{31} - 1\}$)

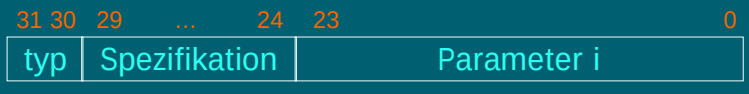
bedeute $A := twoc_{31}(B)$

Instruktionsformate

Sei $I = i_{31}, \ldots, i_0 \in \{0,1\}^{32}$.

$I[y, x] := i_y, i_{y-1}, \ldots, i_x$ für $0 \leq x \leq y \leq 31$

Allgemeines Instruktionsformat:

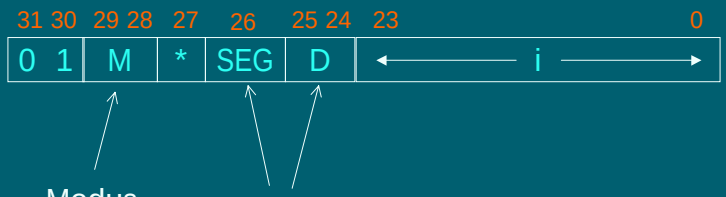


Typ einer Instruktion:

I [31, 30]	Typ
0 0	Compute
0 1	Load
1 0	Store, Move
1 1	Jump

Load - Befehle

Prinzip:



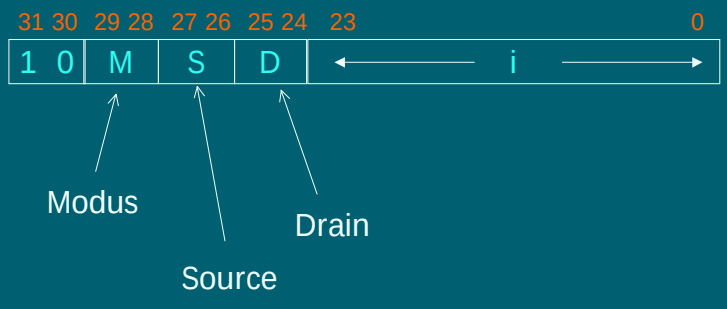
Load – Befehle (ff)

Typ	Modus	SEG	Befehl	Wirkung	
0 1	0 0	0	LOAD i	ACC := M(<i>)	<PC> := <PC> + 1
0 1	0 1	0	LOADIN1 i	ACC := M(<IN1> + [i])	<PC> := <PC> + 1
0 1	1 0	0	LOADIN2 i	ACC := M(<IN2> + [i])	<PC> := <PC> + 1
0 1	1 1	0	LOADI i	ACC := 0^8 i	<PC> := <PC> + 1

Durchführung von Rechnungen <x> + [y] : siehe später

Store, Move - Befehle

Prinzip:



Store, Move - Befehle (ff)

Typ	Modus	Befehl	Wirkung	
1 0	0 0	STORE i	M(<i>) := ACC	<PC> := <PC> + 1
1 0	0 1	STOREIN1 i	M(<IN1> + [i]) := ACC	<PC> := <PC> + 1
1 0	1 0	STOREIN2 i	M(<IN2> + [i]) := ACC	<PC> := <PC> + 1
1 0	1 1	MOVE S D	D := S	<PC> := <PC> + 1

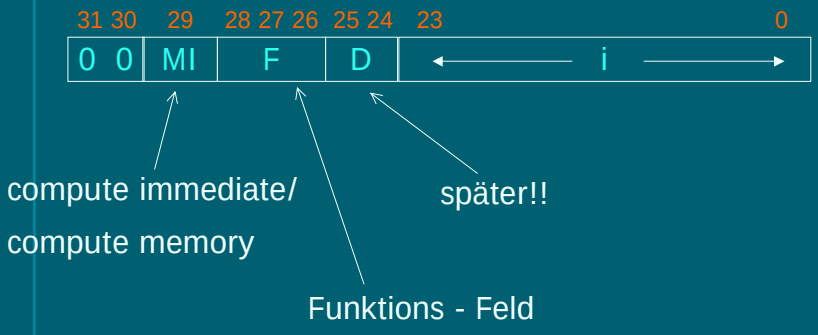
außer bei D = 0 0 (PC)

Kodierung S, D

S, D	Register
0 0	PC
0 1	IN1
1 0	IN2
1 1	ACC

Compute - Befehle

Prinzip:



Compute - Befehle (ff)

Typ	MI	F	Befehl	Wirkung	
0 0	0	0 1 0	SUBI i	[ACC]:=[ACC] – [i]	<PC> := <PC> + 1
		0 1 1	ADDI i	[ACC]:=[ACC] + [i]	<PC> := <PC> + 1
		1 0 0	OPLUSI i	ACC := ACC $\oplus$ 0^8 i	<PC> := <PC> + 1
		1 0 1	ORI i	ACC : = ACC $\vee$ 0^8 i	<PC> := <PC> + 1
		1 1 0	ANDI i	ACC := ACC $\wedge$ 0^8 i	<PC> := <PC> + 1
0 0	1	0 1 0	SUB i	[ACC]:=[ACC] – [M(<i>)]	<PC> := <PC> + 1
		0 1 1	ADD i	[ACC]:=[ACC] + [M(<i>)]	<PC> := <PC> + 1
		1 0 0	OPLUS i	ACC := ACC $\oplus$ M(<i>)	<PC> := <PC> + 1
		1 0 1	OR i	ACC := ACC $\vee$ M(<i>)	<PC> := <PC> + 1
		1 1 0	AND i	ACC := ACC $\wedge$ M(<i>)	<PC> := <PC> + 1

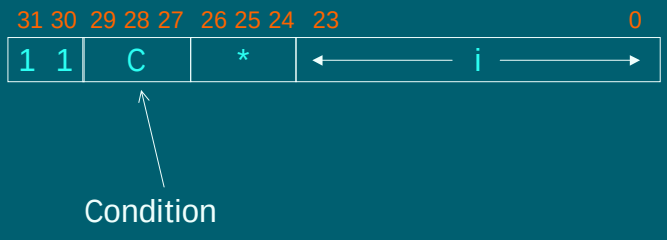
Bemerkung zu OPLUS

$ACC := ACC \oplus 0^8\, i_{23} \ldots i_0 \cong$

$(ACC_{31} \oplus 0, \ldots, ACC_{24} \oplus 0, ACC_{23} \oplus i_{23}, \ldots, ACC_0 \oplus i_0)$

Jump - Befehle

Prinzip:



Kodierung der Bedingung c:

C	Bedingung c
0 0 0	nie
0 0 1	>
0 1 0	=
0 1 1	≥
1 0 0	<
1 0 1	≠
1 1 0	≤
1 1 1	immer

Kodierung nach Schema:

nur I [29] = 1 ⇔ < wird abgefragt

nur I [28] = 1 ⇔ = wird abgefragt

nur I [27] = 1 ⇔ > wird abgefragt

Andere Abfragen durch Kombinationen, z.B.

C = 101 : < oder > , also ≠

Jump - Befehle (ff)

Typ	Befehl	Wirkung
1 1	$JUMP_c$ i	$< PC >:= \begin{cases} < PC > + [i] \text{ , falls } [ACC] \text{ c } 0 \\ < PC > + 1 \text{ , sonst} \end{cases}$

Unbedingte Sprünge werden durch C = 111 ausgedrückt.

Bei C = 000 : *Keine* Wirkung des Befehls außer Inkrementieren des Befehlszählers

→ NOP – Befehl (No Operation)

Andere Befehle

... sind durchaus sinnvoll und bei anderen Architekturen evtl. schon als Grundbefehl vorhanden.

Hier:

Zusammensetzen aus Grundbefehlen mit Hilfe von *Unterprogrammen*.

Beispiel: Shift - Operationen

Linksshift: lsh: $\mathbf{B}^n \rightarrow \mathbf{B}^n$,
$(a_{n-1}, \ldots, a_0) \rightarrow (a_{n-2}, \ldots, a_0, 0)$

zyklischer Linksshift: cls: $\mathbf{B}^n \rightarrow \mathbf{B}^n$,
$(a_{n-1}, \ldots, a_0) \rightarrow (a_{n-2}, \ldots, a_0, a_{n-1})$

Rechtsshift: rsh: $\mathbf{B}^n \rightarrow \mathbf{B}^n$,
$(a_{n-1}, \ldots, a_0) \rightarrow (0, a_{n-1}, \ldots, a_1)$

zyklischer Rechtsshift: crs: $\mathbf{B}^n \rightarrow \mathbf{B}^n$,
$(a_{n-1}, \ldots, a_0) \rightarrow (a_0, a_{n-1}, \ldots, a_1)$

Realisierung mit vorhandenen Maschinenbefehlen: siehe Übung !

Addition und Sign Extension

Probleme bei Additionen:

1. Addition verschieden langer Zahlen
   (z.B. [ACC] + [i])
2. Addition von Binärdarstellungen und Zweierkomplementzahlen
   (z.B. M(<IN1> + [i]) := ACC)

!

Zu 1.

Lösung durch Sign Extension:

Sei $y \in \{ 0,1 \}^{24}$.

Dann sei $sext(y) := y^{8}_{23}y$

sext(y) heißt sign extension von y

Es gilt: $[y] = [sext(y)]$ (→ Übung)

Damit wird [ACC] + [i] zurückgeführt auf

[ACC] + [sext(i)]

zu 2.: Lemma:

Sei $x \in \mathbf{B}^{32}$, $y \in \mathbf{B}^{32}$, $0 \leq \langle x \rangle + [y] < 2^{32}$

und es sei

$\langle x \rangle + \langle sext(y) \rangle = \langle c,s \rangle$ mit $c \in \mathbf{B}$, $s \in \mathbf{B}^{32}$.

Dann gilt: $\langle x \rangle + [y] = \langle s \rangle$

D.h. es genügt, x und sext(y) als Binärzahl

zu interpretieren und den Übertrag zu ignorieren

(wenn keine Überlaufbehandlung nötig).

Beweis: Übung