Kapitel 2 – Kodierung

1. Kodierung von Zeichen
2. Kodierung von Zahlen
3. **Anwendung: ReTI**

Albert-Ludwigs-Universität Freiburg

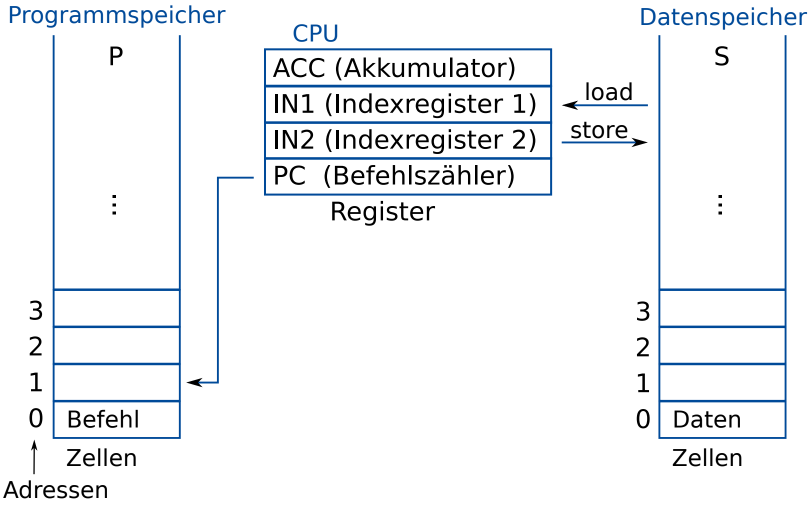
Prof. Dr. Christoph Scholl

Institut für Informatik

WS 2015/16

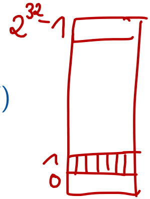
UNI FREIBURG

Realisierung von ReTI



Unterschiede abstrakter/realer ReTI

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



- CPU-Register $PC, ACC, IN1$ und $IN2$ können nur Elemente $w \in \mathbb{B}^{32}$ aufnehmen. w heißt Wort.
    - Ein Wort kann als Binärzahl (z.B. Adresse im M), Zweierkomplementzahl oder Bitstring interpretiert werden.
- Befehle sind ebenfalls Wörter aus $\mathbb{B}^{32}$.

Notation

- $b^j = \underbrace{(b, \ldots, b)}_{j \text{ mal}}$ für $b \in \{0, 1\}$
- $\langle A \rangle := B$

  (A Register oder Speicherzelle, $B \in \{0, ..., 2^{32} - 1\}$)
  bedeute $A := bin_{32}(B)$
  - Beispiel: $\langle PC \rangle := \langle PC \rangle + 1$ $PC := bin_{32}(\langle PC \rangle + 1)$
- $[A] := B$

  (A Register oder Speicherzelle, $B \in \{-2^{31}, ..., 2^{31} - 1\}$)
  bedeute $A := twoc_{32}(B)$
  B wird als Zweierkomplement-Zahl interpretiert.

4

Befehlsformate

- Zur Erinnerung: Der Befehlssatz von ReTI besteht aus Load-/Store-, Compute-, Indexregister- und Sprungbefehlen.

- Sie werden als Wörter aus $\mathbb{B}^{32}$ kodiert. Etwaige Parameter sind in der Kodierung enthalten.
  - Notation: Sei $I = \underline{i_{31}, \ldots, i_0} \in \mathbb{B}^{32}$.
    $I[y,x] := i_y, i_{y-1}, \ldots, i_x$ für $0 \leq x \leq y \leq 31$.

- Allgemeines Instruktionsformat:

i_{31} i_0

31 30	29 ... 24	23 ... 0
Typ	Spezifikation	Parameter i
$I[31,30]$	$I[29,24]$	$I[23,0]$

Typ einer Instruktion

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

31 30	29 … 24	23 … 0
Typ	Spezifikation	Parameter i

UNI FREIBURG

6

Load-Befehle: Kodierungsprinzip

31 30	29 28	27 26	25 24	23 ... 0
0 1	M	*	D	i

- M: Modus
- D: Vorerst irrelevant

7

Load-Befehle: Kodierung

Typ	Modus	Befehl	Wirkung	
0 1	0 0	LOAD i	$ACC := M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$
0 1	0 1	LOADIN1 i	$ACC := M(\langle IN1 \rangle + [i])$	$\langle PC \rangle := \langle PC \rangle + 1$
0 1	1 0	LOADIN2 i	$ACC := M(\langle IN2 \rangle + [i])$	$\langle PC \rangle := \langle PC \rangle + 1$
0 1	1 1	LOADI i	$ACC := 0^8 i$	$\langle PC \rangle := \langle PC \rangle + 1$

„load immediate"

Durchführung von Rechnungen $\langle x \rangle + [y]$ später.

Store-, Move-Befehle: Prinzip

31 30	29 28	27	26	25 24	23 ... 0
1 0	M	S		D	i

- M: Modus
- S: Source
- D: Destination

für MOVE

Kodierung S, D

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

9

Store-, Move-Befehle: Kodierung

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

außer bei D = 00 (PC)

10

Compute-Befehle: Prinzip

31 30	29	28 27 26	25 24	23 … 0
0 0	MI	F	D	i

- MI: „compute **m**emory"/„compute **i**mmediate"
- F: Funktionsfeld
- D: Vorerst irrelevant

11

Compute-Befehle: Kodierung

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

compute immediate

compute memory

Bitstring-Operationen am Beispiel von OPLUS

$\oplus$: exklusives Oder

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

	ACC_{31}	...	ACC_{24}	ACC_{23}	...	ACC_0
$\oplus$	0	...	0	i_{23}	...	i_0
	$ACC_{31} \oplus 0$			...		$ACC_0 \oplus i_0$

- $ACC := \underline{ACC} \oplus 0^8 i_{23} \dots i_0$
  $\cong \underline{(ACC_{31} \oplus 0, \dots, ACC_{24} \oplus 0, ACC_{23} \oplus i_{23}, \dots, ACC_0 \oplus i_0)}$

Sprungbefehle: Prinzip

31 30	29 28 27	26 25 24	23 ... 0
1 1	C	*	i

$JUMP_C$ i

- C: Condition

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

größer oder gleich

kleiner oder gleich oder größer

Bedingungskodierung nach Schema

C	Bedingung c
0 0 0	nie
0 0 1	>
0 1 0	=
0 1 1	$\geq$
1 0 0	<
1 0 1	$\neq$
1 1 0	$\leq$
1 1 1	immer

nur $I[29] = 1 \Leftrightarrow <$ wird abgefragt
nur $I[28] = 1 \Leftrightarrow =$ wird abgefragt
nur $I[27] = 1 \Leftrightarrow >$ wird abgefragt

Andere Abfragen durch Kombinationen,
z.B. $C = 101 : <$ <u>oder</u> $>$, also $\neq$.

15

Sprungbefehle: Kodierung

Typ	Befehl	Wirkung
1 1	$\text{JUMP}_c\ i$	$\langle PC \rangle := \begin{cases} \langle PC \rangle + [i], & \text{falls } [ACC]\ c\ 0 \\ \langle PC \rangle + 1, & \text{sonst} \end{cases}$

- Unbedingte Sprünge werden durch $C = 111$ ausgedrückt.
- Bei $C = 000$: Keine Wirkung des Befehls außer Inkrementieren des Befehlszählers
  $\Rightarrow$ NOP - Befehl (No Operation)

  (In Kap. 8 werden wir kurz auf die Notwendigkeit von NOP - Befehlen bei Pipelining eingehen.)

Zusätzliche Befehle

- Zusätzliche Befehle sind durchaus sinnvoll und bei anderen Architekturen evtl. schon als Grundbefehl vorhanden.
- Nicht vorhandene Befehle müssen hier durch Befehlsfolgen “simuliert” werden.
- Beispiel: Multiplikation, siehe Kapitel 1.2..

Addition und Sign Extension

- Probleme bei Additionen:
  1. Addition verschieden langer Zahlen (z.B. $[ACC] + [i]$).
  2. Addition von ~~Binärdarstellungen~~ und Zweierkomplementzahlen (z.B. $M(\langle IN1\rangle + [i]) := ACC$).

- Zu 1: Lösung durch Sign Extension $y = y_{23} \dots y_0$
  - Sei $y \in \mathbb{B}^{24}$. $sext(y) := y_{23}^8 y$ heißt sign extension von y.
  - Es gilt: $[y]_2 = [sext(y)]_2$ (Beweis: Übung)
  - Dann wird $[ACC] + [i]$ zurückgeführt auf $[ACC] + [sext(i)]$.

- Zu 2: Siehe nächste Folie.

$$[y]_2 = \sum_{i=0}^{22} y_i \cdot 2^i - y_{23} \cdot 2^{23}$$

$$[y_{23} y_{23} \dots y_0]_2 = \sum_{i=0}^{23} y_i \cdot 2^i - y_{23} \cdot 2^{24}$$

$$= \sum_{i=0}^{22} y_i 2^i + y_{23} \cdot 2^{23} - y_{23} \cdot 2^{24}$$

$$= \sum_{i=0}^{22} y_i 2^i - y_{23} \cdot 2^{23} \underbrace{(2-1)}_{1}$$

$$= [y_{23} \dots y_0]_2$$

Addition von Binär- und Zweierkomplementzahlen

Lemma

Sei $x \in \mathbb{B}^{32}, y \in \mathbb{B}^{24}, 0 \leq \langle x \rangle + [y] < 2^{32}$ und sei
$\langle x \rangle + \langle sext(y) \rangle = \langle c, s \rangle$ mit $c \in \mathbb{B}, s \in \mathbb{B}^{32}$.
Dann gilt: $\langle x \rangle + [y] = \langle s \rangle$

- Bedeutung: Kommt es beim Addieren nicht zum Überlauf ($0 \leq \langle x \rangle + [y] < 2^{32}$, so kann man x und y als Binärzahlen interpretieren, addieren und Übertrag ignorieren!
- Zunächst ohne Beweis.
- Wir können somit Parameter i bei den Befehlen ohne Fallunterscheidung nach i positiv / negativ verwenden!

Zusammenfassung

- Kodierungen von Zeichen: Codes fester Länge (z.B. ASCII) sind einfacher aber weniger effizient als Codes variabler Länge (z.B. Huffman).
- Zweier-Komplement-Kodierung von Festkomma-Zahlen erlaubt in Verbindung mit Sign Extension effiziente Umsetzung arithmetischer Operationen in Hardware eines Rechners (tatsächliche Implementierung siehe Kapitel 3.5).
- Der Befehlssatz von ReTI ist auf der nächsten Folie zusammengefasst.

Load-Befehle		$I[25:24] = D$	
$I[31:28]$	Befehl	Wirkung	
0100	LOAD D i	$D := M(\langle i \rangle)$	
0101	LOADIN1 D i	$D := M(\langle IN1 \rangle + [i])$	$\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$
0110	LOADIN2 D i	$D := M(\langle IN2 \rangle + [i])$	
0111	LOADI D i	$D := 0^8 i$	

Store-Befehle		MOVE: $I[27:24] = SD$	
$I[31:28]$	Befehl	Wirkung	
1000	STORE i	$M(\langle i \rangle) := ACC$	
1001	STOREIN1 i	$M(\langle IN1 \rangle + [i]) := ACC$	$\langle PC \rangle := \langle PC \rangle + 1$
1010	STOREIN2 i	$M(\langle IN2 \rangle + [i]) := ACC$	
1011	MOVE S D	$D := S$	$\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$

Compute-Befehle		$I[25:24] = D$	
$I[31:26]$	Befehl	Wirkung	
000010	SUBI D i	$[D] := [D] - [i]$	
000011	ADDI D i	$[D] := [D] + [i]$	
000100	OPLUSI D i	$D := D \oplus 0^8 i$	$\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$
000101	ORI D i	$D := D \vee 0^8 i$	
000110	ANDI D i	$D := D \wedge 0^8 i$	
001010	SUB D i	$[D] := [D] - [M(\langle i \rangle)]$	
001011	ADD D i	$[D] := [D] + [M(\langle i \rangle)]$	
001100	OPLUS D i	$D := D \oplus M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$
001101	OR D i	$D := D \vee M(\langle i \rangle)$	
001110	AND D i	$D := D \wedge M(\langle i \rangle)$	

Jump-Befehle		
$I[31:27]$	Befehl	Wirkung
11000	NOP	$\langle PC \rangle := \langle PC \rangle + 1$
11001	$\text{JUMP}_{>}\, i$	
11010	$\text{JUMP}_{=}\, i$	
11010	$\text{JUMP}_{\geq}\, i$	$\langle PC \rangle := \begin{cases} \langle PC \rangle + [i], & \text{falls } [ACC]\ c\ 0 \\ \langle PC \rangle + 1, & \text{sonst} \end{cases}$
11011	$\text{JUMP}_{<}\, i$	
11100	$\text{JUMP}_{\neq}\, i$	
11110	$\text{JUMP}_{\leq}\, i$	
11111	JUMP i	$\langle PC \rangle := \langle PC \rangle + [i]$

Kodierung S,D

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