

Kapitel 2 – Kodierung

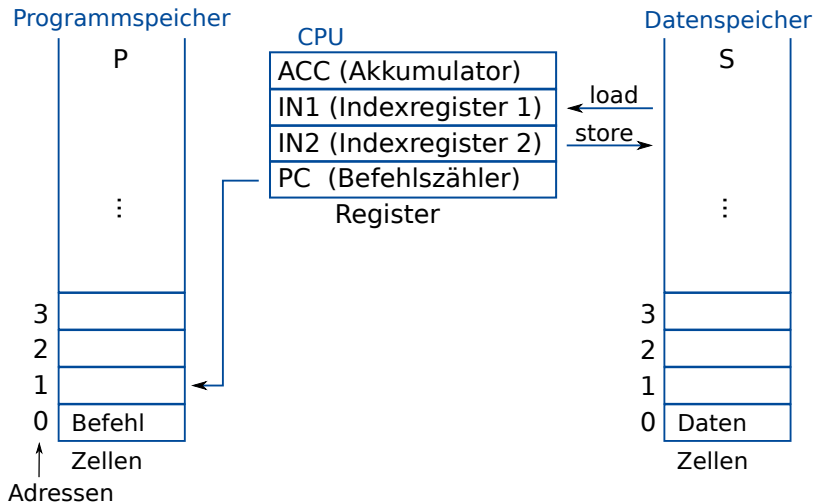
1. Kodierung von Zeichen
2. Kodierung von Zahlen
3. **Anwendung: ReTI**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur
WS 2016/17

Realisierung von ReTI



Unterschiede abstrakter/realer Rechner

Adressbreite / Raum
32 Bit

$$2^{32} \cdot 4 \text{ Byte} = 16 \text{ GiB}$$

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

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

■ $\underline{\langle A \rangle} := B$

(A Register oder Speicherzelle, $B \in \{0, \dots, 2^{32} - 1\}$)
bedeute $A := \text{bin}_{32}(B)$

■ Beispiel: $\underline{\langle PC \rangle} := \underline{\langle PC \rangle} + 1$

■ $\underline{[A]} := B$

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

ADDI i:

„ACC = ACC + i“

- 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 $I = i_{31}, \dots, i_0 \in \mathbb{B}^{32}$.
 $I[y, x]$:= i_y, i_{y-1}, \dots, i_x für $0 \leq x \leq y \leq 31$.

- Allgemeines Instruktionsformat:

I:

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

$I[31:30] \cong \text{Typ des Befehls!}$

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	

Load-Befehle: Kodierungsprinzip

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

nicht genutzt

■ M: Modus

■ D: Vorerst irrelevant

— ↳ Destination

Load-Befehle: Kodierung

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

32 Bit i nur 24 Bit

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

0⁸ i

00000000 $i_{23} \dots i_0$
32 Bit

d.h. wir füllen mit 0 auf

Store-, Move-Befehle: Prinzip

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

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

Kodierung S, D

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

Store-, Move-Befehle: Kodierung

Typ	Modus	Befehl	Wirkung	
<u>1 0</u>	<u>0 0</u>	STORE i	<u>$M(\langle i \rangle) := ACC$</u>	<u>$\langle PC \rangle := \langle PC \rangle + 1$</u>
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 $S D$	$D := S$	<u>$\langle PC \rangle := \langle PC \rangle + 1$</u>

↑ ↗
außer bei $D = 00$ (PC)

Compute-Befehle: Prinzip

31 30	29	28 27 26	25 24	23	...	0
<u>0 0</u>	<u>MI</u>	<u>F</u>	<u>D</u>		i	

- MI: „compute memory”/„compute immEDIATE”
- F: Funktionsfeld
- D: Vorerst irrelevant
Destination

Compute-Befehle: Kodierung

↓ EXOR "exklusiv-oder"

Typ	MI	F	Befehl	Wirkung	
<u>0 0</u>	<u>0</u>	0 1 0	SUB <u>i</u>	$[ACC] := [ACC] - [i]$	$\langle PC \rangle := \langle PC \rangle + 1$
		0 1 1	ADD <u>i</u>	$[ACC] := [ACC] + [i]$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 0	OPLUS <u>i</u>	$ACC := ACC \oplus 0^8 i$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 1	OR <u>i</u>	$ACC := ACC \vee 0^8 i$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 1 0	AND <u>i</u>	$ACC := ACC \wedge 0^8 i$	$\langle PC \rangle := \langle PC \rangle + 1$
0 0	<u>1</u>	0 1 0	SUB <u>i</u>	$[ACC] := [ACC] - [M(\langle i \rangle)]$	$\langle PC \rangle := \langle PC \rangle + 1$
		0 1 1	ADD <u>i</u>	$[ACC] := [ACC] + [M(\langle i \rangle)]$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 0	OPLUS <u>i</u>	$ACC := ACC \oplus M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 1	OR <u>i</u>	$ACC := ACC \vee M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 1 0	AND <u>i</u>	$ACC := ACC \wedge M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$

Bitstring-Operationen am Beispiel von OPLUS

a	b	EXOR(a,b)
0	0	0
0	1	1
1	0	1
1	1	0

■ $ACC := ACC \oplus 0^8 i_{23} \dots i_0$

$$\cong (\underbrace{ACC_{31} \oplus 0, \dots, ACC_{24} \oplus 0}_{\uparrow}, \underbrace{ACC_{23} \oplus i_{23}, \dots, ACC_0 \oplus i_0}_{\uparrow})$$

Sprungbefehle: Prinzip

nicht benutzt

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

■ C: Condition

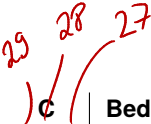
gilt „ACC < 0“ dann Sprung!

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

NOP → No operation,
nur PC = PC + 1

unbedingter Sprung

Bedingungskodierung nach Schema



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

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 :< oder >, also ≠.

Sprungbefehle: Kodierung

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

- *„immer“* **Unbedingte Sprünge** werden durch **C = 111** ausgedrückt.
- Bei **C = 000**: Keine Wirkung des Befehls außer Inkrementieren des Befehlszählers
⇒ **NOP** - **Befehl** (No Operation)

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

- 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. $[ACC] + [i]$). 32Bit / 24Bit
- 2 Addition von Binärdarstellungen und Zweierkomplementzahlen (z.B. $M(\langle IN1 \rangle) + [i] := ACC$).
Bin. Zer-kompl.

■ Zu 1: Lösung durch Sign Extension

- Sei $y \in \mathbb{B}^{24}$. $s_{ext}(y) := y^8_{23}y$ heißt sign extension von y .
① ① 100...0 1111...
- Es gilt: $[y] = [s_{ext}(y)]$. (Beweis: Übung)
-2^n -2^n 0
- Dann wird $[ACC] + [i]$ zurückgeführt auf $[ACC] + [s_{ext}(i)]$.
32Bit 32Bit

■ Zu 2: Siehe nächste Folie.

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 $\langle x \rangle + \text{sext}(y) = \langle c, s \rangle$ 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!

- 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[31 : 28]	Befehl	Wirkung
<u>0100</u>	LOAD $D\ i$	$D := M(\langle i \rangle)$
0101	LOADIN1 $D\ i$	$D := M(\langle IN1 \rangle + [i])$
0110	LOADIN2 $D\ i$	$D := M(\langle IN2 \rangle + [i])$
<u>0111</u>	LOADI $D\ i$	$D := 0^8 i$
Store-Befehle		
I[31 : 28]	Befehl	Wirkung
<u>1000</u>	STORE i	$M(\langle i \rangle) := ACC$
1001	STOREIN1 i	$M(\langle IN1 \rangle + [i]) := ACC$
1010	STOREIN2 i	$M(\langle IN2 \rangle + [i]) := ACC$
1011	MOVE S D	$D := S$
Compute-Befehle		
I[31 : 26]	Befehl	Wirkung
<u>000010</u>	SUBI $D\ i$	$[D] := [D] - [i]$
<u>000011</u>	ADDI $D\ i$	$[D] := [D] + [i]$
000100	OPLUSI $D\ i$	$D := D \oplus 0^8 i$
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 $D\ i$	$D := D \oplus M(\langle i \rangle)$
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
<u>11000</u>	NOP	$\langle PC \rangle := \langle PC \rangle + 1$
11001	JUMP _{>} i	$\langle PC \rangle := \begin{cases} \langle PC \rangle + [i], & \text{falls } [ACC] < 0 \\ \langle PC \rangle + 1, & \text{sonst} \end{cases}$
11010	JUMP ₌ i	
11010	JUMP _{>} i	
11011	JUMP _{<} i	
11100	JUMP _≠ i	
11110	JUMP _{<} 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
<u>1 1</u>	ACC

