

Rechnertechnik

Prof. Dr. rer. nat. Peter Gerwinski

31. Mai 2021

Rechnertechnik

1 Einführung

2 Vom Schaltkreis zum Computer

2.1 Logik-Schaltkreise

2.2 Binärdarstellung von Zahlen

2.3 Vom Logik-Schaltkreis zum Addierer

2.4 Negative Zahlen

2.5 Vom Addierer zum Computer

2.6 Computer-Sprachen

2.7 Programmieren in Assembler

2.8 Struktur von Assembler-Programmen

3 Architekturmerkmale von Prozessoren

4 Der CPU-Stack

...

2.8 Struktur von Assembler-Programmen

Beispiel 2: Redcode (ICWS '88) – Core War[s] (Krieg der Kerne)

Virtuelle Maschine: Memory Array Redcode Simulator (MARS)

Instruktionen:

dat B – Daten – „Du hast verloren!“

mov A, B – kopiere A nach B

add A, B – addiere A zu B

sub A, B – subtrahiere A von B

jmp A – unbedingter Sprung nach A

jmz A, B – Sprung nach A, wenn $B = 0$

jmn A, B – Sprung nach A, wenn $B \neq 0$

djn A, B – „decrement and jump if not zero“

cmp A, B – „compare“: überspringe, falls gleich

spl A – „split“: Programm verzweigen

Adressierungsarten:

grundsätzlich: Speicher relativ

– unmittelbar

\$ – direkt

@ – indirekt

< – indirekt mit Prä-Dekrement

Programm „Nothing“:

jmp 0

Programm „Knirps“:

mov 0, 1

2.8 Struktur von Assembler-Programmen

Unbedingte Verzweigung

Beispiel: Endlosschleife von „Dwarf“

```
bomb  dat #0
start add #4, bomb
      mov bomb, @bomb
      jmp start
      end start
```

Instruktionen:

```
dat B
mov A, B
add A, B
sub A, B
jmp A
jnz A, B – „jump if zero“
jmn A, B – „if not zero“
djn A, B – „dec. & jmn“
cmp A, B – vgl. & überspr.
spl A – verzweigen
```

Adressierungsarten:

grundsätzlich:
Speicher relativ

– unmittelbar

\$ – direkt

@ – indirekt

< – indirekt mit
Prä-Decrement

2.8 Struktur von Assembler-Programmen

Bedingte Verzweigung

Beispiel: Kopierschleife von „Mice“

```
ptr    dat #0
start  mov #12, ptr
loop   mov @ptr, <dest
       djn loop, ptr
       spl @dest
       add #653, dest
       jnz start, ptr
dest   dat #833
       end start
```

Instruktionen:

```
dat B
mov A, B
add A, B
sub A, B
jmp A
jnz A, B – „jump if zero“
jmn A, B – „if not zero“
djn A, B – „dec. & jmn“
cmp A, B – vgl. & überspr.
spl A – verzweigen
```

Adressierungsarten:

grundsätzlich:
Speicher relativ

– unmittelbar

\$ – direkt

@ – indirekt

< – indirekt mit
Prä-Decrement

2.8 Struktur von Assembler-Programmen

Mehrere Threads

Beispiel: Multithreading von „Mice“

ptr	dat #0
start	mov #12, ptr
loop	mov @ptr, <dest
	djn loop, ptr
	spl @dest
	add #653, dest
	jmz start, ptr
dest	dat #833
	end start

Instruktionen:

dat B
mov A, B
add A, B
sub A, B
jmp A
jmz A, B – „jump if zero“
jmn A, B – „if not zero“
djn A, B – „dec. & jmn“
cmp A, B – vgl. & überspr.
spl A – verzweigen

Adressierungsarten:

grundsätzlich:
Speicher relativ

– unmittelbar

\$ – direkt

@ – indirekt

< – indirekt mit
Prä-Decrement

2.8 Struktur von Assembler-Programmen

Selbstmodifizierender Code

Beispiel: Selbsterkennung von „Fini“

```
num    dat #-2
start  mov num, <pos
       jmp start
pos    dat
       end start
```

Instruktionen:

```
dat B
mov A, B
add A, B
sub A, B
jmp A
jnz A, B – „jump if zero“
jmn A, B – „if not zero“
djn A, B – „dec. & jmn“
cmp A, B – vgl. & überspr.
spl A – verzweigen
```

Adressierungsarten:

grundsätzlich:
Speicher relativ

– unmittelbar

\$ – direkt

@ – indirekt

< – indirekt mit
Prä-Decrement

2.8 Struktur von Assembler-Programmen

Ehemaliger Praktikumsversuch (2014)

Inspiration für Projekt?

Schreiben Sie ein
Redcode-Programm,
das die Gegner
Nothing, **Knirps**
und **Mice** besiegt.

ICWS-88-Standard,
max. 64 Prozesse,
Speichergröße zufällig

Teams bis zu 3 Personen
sind zulässig.

Instruktionen:

dat B
mov A, B
add A, B
sub A, B
jmp A
jmz A, B – „jump if zero“
jmn A, B – „if not zero“
djn A, B – „dec. & jmn“
cmp A, B – vgl. & überspr.
spl A – verzweigen

Adressierungsarten:

grundsätzlich:
Speicher relativ

– unmittelbar

\$ – direkt

@ – indirekt

< – indirekt mit
Prä-Decrement

Rechnertechnik

1 Einführung

2 Vom Schaltkreis zum Computer

2.1 Logik-Schaltkreise

2.2 Binärdarstellung von Zahlen

2.3 Vom Logik-Schaltkreis zum Addierer

2.4 Negative Zahlen

2.5 Vom Addierer zum Computer

2.6 Computer-Sprachen

2.7 Programmieren in Assembler

2.8 Struktur von Assembler-Programmen

3 Architekturmerkmale von Prozessoren

3.1 Speicherarchitekturen

3.2 Registerarchitekturen

3.3 Befehlssätze

4 Der CPU-Stack

...

3 Architekturmerkmale von Prozessoren

3.1 Speicherarchitekturen

Bezeichnungen

- *Bit* = 0 oder 1 – kleinste Einheit an Information
- *Byte* = Zusammenfassung mehrerer *Bits*
zu einer Binärzahl, die ein Zeichen (*Character*) darstellen kann,
häufig 8 Bits (*Oktett*)
- *Speicherwort* = Zusammenfassung mehrerer Bits
zu der kleinsten adressierbaren Einheit, häufig 1 Byte
- *RAM* = *Random Access Memory* = Hauptspeicher
- *ROM* = *Read Only Memory* = nur lesbarer Speicher

3.1 Speicherarchitekturen

Verschiedene Arten von Speicher

- *Prozessor-Register*
können direkt mit ALU verbunden werden,
besonders schnell (Flipflops),
überschaubare Anzahl von Registern
- *Hauptspeicher*
kann direkt adressiert und mit Prozessor-Registern abgeglichen werden,
heute i. d. R. dynamischer Speicher (Kondensatoren)
- *I/O-Ports*
sind spezielle Speicheradressen, über die
mit externen Geräten kommuniziert wird
- *Massenspeicher*
liegt auf externem Gerät, wird über I/O-Ports angesprochen,
Festplatte, Flash-Speicher, ...

3.1 Speicherarchitekturen

- *Von-Neumann-Architektur*

Es gibt nur 1 Hauptspeicher, in dem sich sowohl die Befehle als auch die Daten befinden.

Vorteil: Flexibilität in der Speichernutzung

Nachteil: Befehle können überschrieben werden.

—→ Abstürze und Malware möglich

3.1 Speicherarchitekturen

- *Von-Neumann-Architektur*

Es gibt nur 1 Hauptspeicher, in dem sich sowohl die Befehle als auch die Daten befinden.

- *Harvard-Architektur*

Es gibt 2 Hauptspeicher. In einem befinden sich die Befehle, im anderen die Daten.

Vorteil: Befehle können nicht überschrieben werden

—> sicherer als Von-Neumann-Architektur

Nachteile: Leitungen zum Speicher (Bus) müssen doppelt vorhanden sein, freier Befehlsspeicher kann nicht für Daten genutzt werden.

3.1 Speicherarchitekturen

- *Von-Neumann-Architektur*
Es gibt nur 1 Hauptspeicher, in dem sich sowohl die Befehle als auch die Daten befinden.
- *Harvard-Architektur*
Es gibt 2 Hauptspeicher. In einem befinden sich die Befehle, im anderen die Daten.
- Weitere Kombinationen
Hauptspeicher und I/O-Ports gemeinsam oder getrennt,
Hauptspeicher und Prozessorregister gemeinsam oder getrennt

3.1 Speicherarchitekturen

Beispiele:

- Intel IA-32 (i386, Nachfolger und Kompatible):
Von-Neumann-Architektur (plus Speicherschutzmechanismen),
Prozessorregister und I/O-Ports vom Hauptspeicher getrennt
- Atmel AVR (z. B. ATmega):
Harvard-Architektur (Befehlsspeicher als Flash-Speicher grundsätzlich auch
schreibbar),
Prozessorregister und I/O-Ports in gemeinsamem Adressbereich mit
Hauptspeicher
- 6502 (heute: Renesas-Mikro-Controller):
Von-Neumann-Architektur,
I/O-Ports in gemeinsamem Adressbereich mit Hauptspeicher,
Prozessorregister und Hauptspeicher getrennt

3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur $=$, $+=$, $-=$, $*=$, $/=$, ...

3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur $=$, $+=$, $-=$, $*=$, $/=$, ...

Beispiel: $c = a + 2 * b;$

3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur $=$, $+=$, $-=$, $*=$, $/=$, ...

Beispiel: $c = a + 2 * b;$

C:

```
R = b;  
R *= 2;  
R += a;  
c = R;
```

3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur =, +=, -=, *=, /=, ...

Beispiel: $c = a + 2 * b;$

C: Mehrere Register:

```
R = b;      movl (b), %eax
R *= 2;     imull $2, %eax, %eax
R += a;     addl (a), %eax
c = R;      movl %eax, (c)
            (IA-32-Assembler)
```

3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur $=$, $+=$, $-=$, $*=$, $/=$, ...

Beispiel: $c = a + 2 * b$;

C:	Mehrere Register:	Akkumulator:
$R = b$;	<code>movl (b), %eax</code>	<code>load (b)</code>
$R *= 2$;	<code>imull \$2, %eax, %eax</code>	<code>mul \$2</code>
$R += a$;	<code>addl (a), %eax</code>	<code>add (a)</code>
$c = R$;	<code>movl %eax, (c)</code>	<code>store (c)</code>
	(IA-32-Assembler)	(Pseudo-Assembler)

3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur $=$, $+=$, $-=$, $*=$, $/=$, ...

Beispiel: $c = a + 2 * b$;

C:	Mehrere Register:	Akkumulator:	Register-Stack:
$R = b$;	<code>movl (b), %eax</code>	<code>load (b)</code>	<code>push (a)</code>
$R *= 2$;	<code>imull \$2, %eax, %eax</code>	<code>mul \$2</code>	<code>push (b)</code>
$R += a$;	<code>addl (a), %eax</code>	<code>add (a)</code>	<code>push \$2</code>
$c = R$;	<code>movl %eax, (c)</code>	<code>store (c)</code>	<code>mul</code>
	(IA-32-Assembler)	(Pseudo-Assembler)	<code>add</code>
			<code>pop (c)</code>

3.2 Registerarchitekturen

Beispiele:

- Intel IA-32 (i386, Nachfolger und Kompatible):
Mehrere Register, für verschiedene Zwecke spezialisiert (unübersichtlich),
Fließkommaregister: Stack-Architektur
- Atmel AVR (z. B. ATmega):
32 Register
- 6502 (heute: Renesas-Mikro-Controller):
3 Register: A, X, Y. Nur A kann rechnen → Akkumulator
- Java Virtual Machine (JVM):
Stack-Architektur
- Redcode:
Jede Speicherzelle fungiert als Register

3.3 Befehlssätze

- *Complex Instruction Set Computer (CISC)*

Umfangreiche Befehlssätze, mächtige Befehle

→ komfortable manuelle Programmierung in Assembler

→ längere Abarbeitungszeit der einzelnen Befehle

Realisierung: „Prozessor im Prozessor“ – *Mikroprogramme*

Beispiele: IA-32, AMD-64

3.3 Befehlssätze

- *Complex Instruction Set Computer (CISC)*

Umfangreiche Befehlssätze, mächtige Befehle

- *Reduced Instruction Set Computer (RISC)*

wenige, wenig mächtige Befehle

→ Programmierung in Assembler für Menschen unkomfortabel

→ schnelle Abarbeitung der Befehle

Beispiele: Atmel AVR, Redcode

3.3 Befehlssätze

- *Complex Instruction Set Computer (CISC)*
Umfangreiche Befehlssätze, mächtige Befehle
- *Reduced Instruction Set Computer (RISC)*
wenige, wenig mächtige Befehle
- *Very Long Instruction Word (VLIW)* und
Explicitly Parallel Instruction Computing (EPIC)
mehrere Befehle gleichzeitig ausführbar
—→ mehr Rechenleistung möglich
—→ Programmierung sehr aufwendig
Beispiel: IA-64

3.3 Befehlssätze

- *Complex Instruction Set Computer (CISC)*
Umfangreiche Befehlssätze, mächtige Befehle
- *Reduced Instruction Set Computer (RISC)*
wenige, wenig mächtige Befehle
- *Very Long Instruction Word (VLIW)* und
Explicitly Parallel Instruction Computing (EPIC)
mehrere Befehle gleichzeitig ausführbar
- *Orthogonaler Befehlssatz*
jeder Befehl mit jeder Adressierungsart kombinierbar