

# Rechnertechnik

Prof. Dr. rer. nat. Peter Gerwinski

26. April 2022

# 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 Struktur von Assembler-Programmen

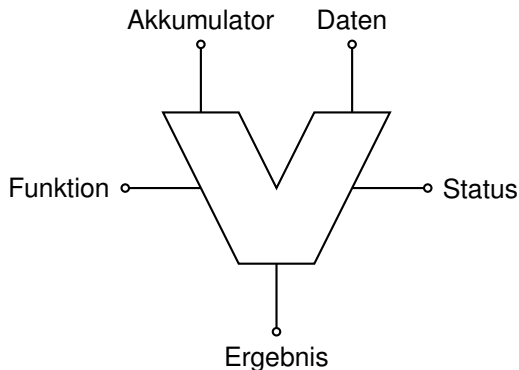
## 3 Architekturmerkmale von Prozessoren

## 4 Der CPU-Stack

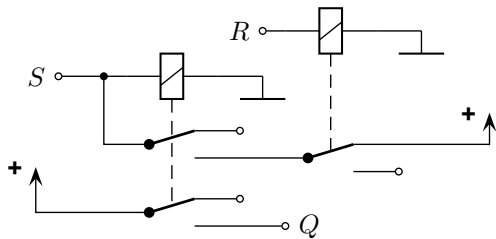
...

## 2.5 Vom Addierer zum Computer

Schaltkreis, der wahlweise eine von mehreren Verknüpfungen durchführt:  
*arithmetisch-logische Einheit – arithmetic logic unit (ALU)*

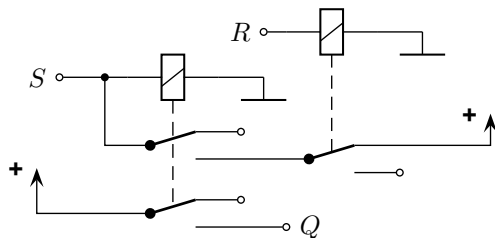


Siehe z. B.: <https://en.wikipedia.org/wiki/File:74181aluschematic.png>



## 2.5 Vom Addierer zum Computer

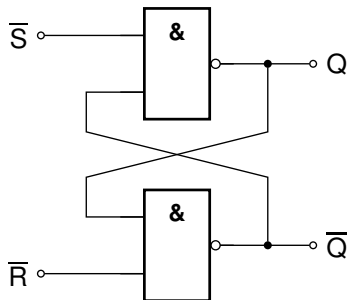
Information speichern



*Selbsthalterschaltung*  
*1-Bit-Speicherzelle*

## 2.5 Vom Addierer zum Computer

Information speichern



*Bistabile Kippstufe – Bistabiler Multivibrator – Flip-Flop*  
1-Bit-Speicherzelle

## 2.5 Vom Addierer zum Computer

Information speichern

*Kondensator*

1-Bit-Speicherzelle



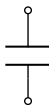
## 2.5 Vom Addierer zum Computer

Information speichern

*Kondensator*

*dynamische* 1-Bit-Speicherzelle

→ benötigt *Refresh*-Schaltung





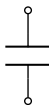
## 2.5 Vom Addierer zum Computer

Information speichern

*Kondensator*

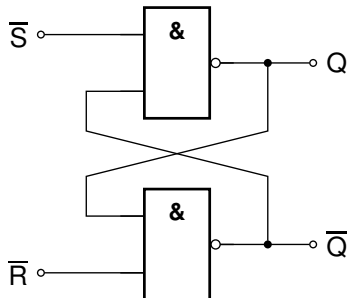
*dynamische* 1-Bit-Speicherzelle

→ benötigt *Refresh*-Schaltung



Flip-Flop

*statische* 1-Bit-Speicherzelle



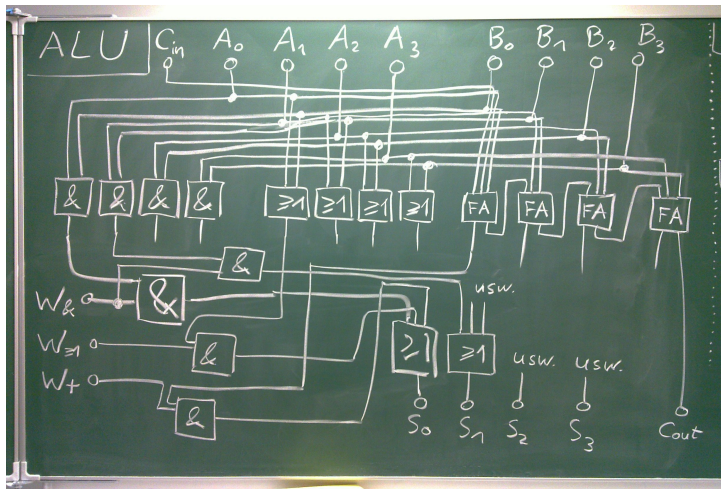
## 2.5 Vom Addierer zum Computer

Bau eines Turing-vollständigen Computers

## 2.5 Vom Addierer zum Computer

Bau eines Turing-vollständigen Computers

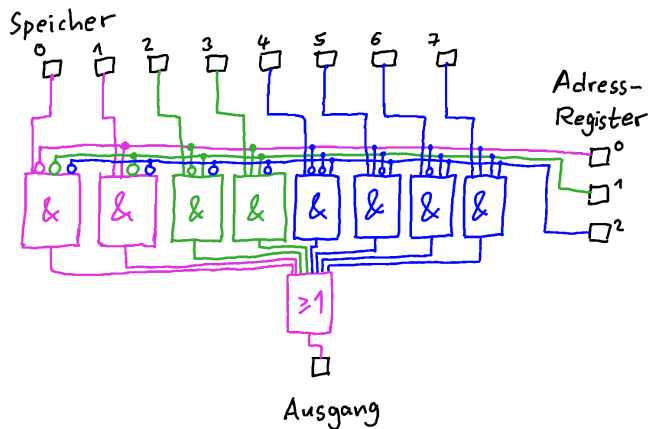
- Rechenwerk: Arithmetisch-logische Einheit (ALU)



## 2.5 Vom Addierer zum Computer

Bau eines Turing-vollständigen Computers

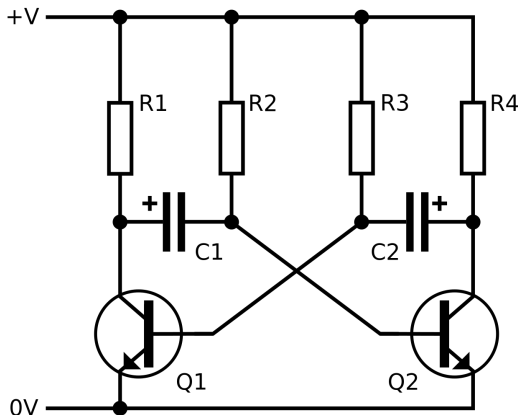
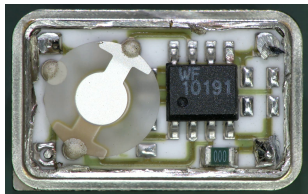
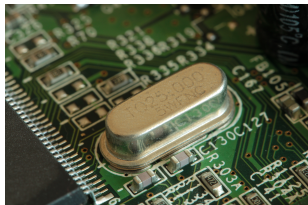
- Rechenwerk: Arithmetisch-logische Einheit (ALU)
- Speicher: Register, adressierbarer Hauptspeicher



## 2.5 Vom Addierer zum Computer

Bau eines Turing-vollständigen Computers

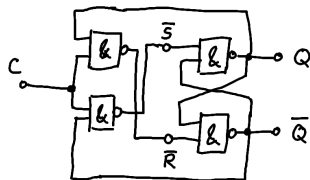
- Rechenwerk: Arithmetisch-logische Einheit (ALU)
- Speicher: Register, adressierbarer Hauptspeicher
- Takt:



## 2.5 Vom Addierer zum Computer

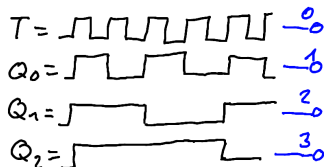
### Bau eines Turing-vollständigen Computers

- Rechenwerk: Arithmetisch-logische Einheit (ALU)
- Speicher: Register, adressierbarer Hauptspeicher
- Takt: Speicher durchgehen



„Stromstoßschalter“

Funktions-Eingänge der ALU



Adress-Register  
Speicher Bit für Bit auslesen



Zähler-Schaltung

## 2.5 Vom Addierer zum Computer

### Bau eines Turing-vollständigen Computers

- Rechenwerk: Arithmetisch-logische Einheit (ALU)
- Speicher: Register, adressierbarer Hauptspeicher
- Takt: Speicher durchgehen und Befehle abarbeiten
  - Register-Ladebefehl:  
„Lade Speicherzelle 42 an den A-Eingang der ALU.“
  - Rechenbefehl:  
„Berechne die Summe der beiden Register an der ALU.“
  - Register-Speicherbefehl:  
„Speichere den Status-Ausgang der ALU in Speicherzelle 137.“

## 2.5 Vom Addierer zum Computer

### Bau eines Turing-vollständigen Computers

- Rechenwerk: Arithmetisch-logische Einheit (ALU)
- Speicher: Register, adressierbarer Hauptspeicher
- Takt: Speicher durchgehen und Befehle abarbeiten
  - Register-Ladebefehl:  
„Lade Speicherzelle 42 an den A-Eingang der ALU.“
  - Rechenbefehl:  
„Berechne die Summe der beiden Register an der ALU.“
  - Register-Speicherbefehl:  
„Speichere den Status-Ausgang der ALU in Speicherzelle 137.“
  - Sprungbefehl:  
„Lade den Inhalt der Speicherzelle 1117  
in das Adressregister für den nächsten Befehl.“



## 2.5 Vom Addierer zum Computer

### Bau eines Turing-vollständigen Computers

- Rechenwerk: Arithmetisch-logische Einheit (ALU)
- Speicher: Register, adressierbarer Hauptspeicher
- Takt: Speicher durchgehen und Befehle abarbeiten
  - Register-Ladebefehl:  
„Lade Speicherzelle 42 an den A-Eingang der ALU.“
  - Rechenbefehl:  
„Berechne die Summe der beiden Register an der ALU.“
  - Register-Speicherbefehl:  
„Speichere den Status-Ausgang der ALU in Speicherzelle 137.“
  - Sprungbefehl:  
„Lade den Inhalt der Speicherzelle 1117  
in das Adressregister für den nächsten Befehl.“
  - Bedingter Sprungbefehl:  
„Wenn das  $C_{out}$ -Bit der ALU den Wert 1 hat,  
springe nach Speicherzelle 23.“

## 2.5 Vom Addierer zum Computer

### Bau eines Turing-vollständigen Computers

- Rechenwerk: Arithmetisch-logische Einheit (ALU)
- Speicher: Register, adressierbarer Hauptspeicher
- Takt: Speicher durchgehen und Befehle abarbeiten
- Peripherie: Kommunikation mit der Außenwelt
  - Output-Port:  
Ausgang einer Speicherzelle nach draußen führen
  - Input-Port:  
Anstelle einer Speicherzelle einen eingehenden Draht abfragen

## 2.5 Vom Addierer zum Computer

Bau eines Turing-vollständigen Computers

- Rechenwerk: Arithmetisch-logische Einheit (ALU)
- Speicher: Register, adressierbarer Hauptspeicher
- Takt: Speicher durchgehen und Befehle abarbeiten
- Peripherie: Kommunikation mit der Außenwelt

→ in Maschinensprache programmierbar

# 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 Struktur von Assembler-Programmen
- 3 Architekturmerkmale von Prozessoren**
- 4 Der CPU-Stack**
- ...

## 2.6 Computer-Sprachen

### 2.6.1 Maschinensprache

#### Computer

- Rechenwerk (ALU)
- Speicher: Register, adressierbarer Hauptspeicher
- Takt: Befehle abarbeiten
- Peripherie: Kommunikation mit der Außenwelt

→ in Maschinensprache programmierbar

In jedem Takt:

- dort aus dem Hauptspeicher lesen, wohin das Register **IP** zeigt  
→ *Befehl* (*Instruction* – **IP** = *Instruction Pointer*)
- den *Befehl* an den *Funktion*-Eingang der *ALU* legen
- auf ähnliche Weise weitere Daten an den *Akkumulator*- und den *Daten*-Eingang der *ALU* legen
- auf ähnliche Weise den *Ergebnis*-Ausgang der *ALU* in den Hauptspeicher schreiben → Befehl ausgeführt
- Register **IP** hochzählen → nächster Befehl

→ Maschinensprache

## 2.6 Computer-Sprachen

### 2.6.1 Maschinensprache

- Daten aus dem Speicher in Register einlesen
- Daten aus Register in den Speicher schreiben

—> Lade- und Speicher-Befehle

- Daten aus Registern (oder Speicher) an ALU legen, Ergebnis in Register (oder Speicher) schreiben

—> arithmetische Befehle

## 2.6 Computer-Sprachen

### 2.6.1 Maschinensprache

- Daten aus dem Speicher in Register einlesen
- Daten aus Register in den Speicher schreiben

—> Lade- und Speicher-Befehle

- Daten aus Registern (oder Speicher) an ALU legen, Ergebnis in Register (oder Speicher) schreiben

—> arithmetische Befehle

- Daten aus Register oder Speicher in das IP-Register laden

—> Sprungbefehl

## 2.6 Computer-Sprachen

### 2.6.1 Maschinensprache

- Daten aus dem Speicher in Register einlesen
- Daten aus Register in den Speicher schreiben

—> Lade- und Speicher-Befehle

- Daten aus Registern (oder Speicher) an ALU legen, Ergebnis in Register (oder Speicher) schreiben

—> arithmetische Befehle

- Daten aus Register oder Speicher in das **IP**-Register laden

—> Sprungbefehl

- Sprungbefehl nur dann, wenn im Status-Ausgang der ALU ein bestimmtes Bit gesetzt ist

—> bedingter Sprungbefehl



## 2.6 Computer-Sprachen

### 2.6.1 Maschinensprache

- Daten aus dem Speicher in Register einlesen
- Daten aus Register in den Speicher schreiben

—> Lade- und Speicher-Befehle

- Daten aus Registern (oder Speicher) an ALU legen, Ergebnis in Register (oder Speicher) schreiben

—> arithmetische Befehle

- Daten aus Register oder Speicher in das IP-Register laden

—> unbedingter Sprungbefehl

- Sprungbefehl nur dann, wenn im Status-Ausgang der ALU ein bestimmtes Bit gesetzt ist

—> bedingter Sprungbefehl

## 2.6 Computer-Sprachen

### 2.6.1 Maschinensprache

- Daten aus dem Speicher in Register einlesen
- Daten aus Register in den Speicher schreiben

→ Lade- und Speicher-Befehle

- Daten aus Registern (oder Speicher) an ALU legen, Ergebnis in Register (oder Speicher) schreiben

→ arithmetische Befehle

- Daten aus Register oder Speicher in das **IP**-Register laden

→ unbedingter Sprungbefehl

- Sprungbefehl nur dann, wenn im Status-Ausgang der ALU ein bestimmtes Bit gesetzt ist

→ bedingter Sprungbefehl

→ Der Computer kann „alles“ – *Turing-Vollständigkeit*

## 2.6 Computer-Sprachen

### 2.6.1 Maschinensprache

- Lade- und Speicher-Befehle  
arithmetische Befehle  
unbedingte und bedingte Sprungbefehle

→ Der Computer kann „alles“ – *Turing-Vollständigkeit*

- Maschinensprache = Zahlen → für Menschen schwer handhabbar

→ Namen für die Befehle: *Mnemonics*

→ *Assembler*-Sprache

## 2.6.2 Maschinensprache → Assembler

Beispiel: Intel-x86-16-Bit-Assembler

- Lade- und Speicher-Befehle
- arithmetische Befehle
- unbedingte und bedingte Sprungbefehle
- Register

mov, ...  
add, sub, inc, dec, xor, cmp, ...  
jmp, jz, jae, ...  
ax, bx, ...

## 2.6.2 Maschinensprache → Assembler

### Beispiel: Intel-x86-16-Bit-Assembler

- Lade- und Speicher-Befehle
- arithmetische Befehle
- unbedingte und bedingte Sprungbefehle
- Register

mov, ...  
add, sub, inc, dec, xor, cmp, ...  
jmp, jz, jae, ...  
ax, bx, ...

### Beispiel: Atmel-AVR-8-Bit-Assembler

- Lade- und Speicher-Befehle
- arithmetische Befehle
- unbedingte und bedingte Sprungbefehle
- Register

ldi, lds, sti, ...  
add, sub, subi, eor, cp, ...  
rjmp, brsh, brlo, ...  
r0, r1, ...

→ für jeden Prozessor anders

## 2.6.3 Maschinensprache $\longrightarrow$ Assembler $\longrightarrow$ Hochsprachen

### Beispiel: Intel-x86-16-Bit-Assembler

- Lade- und Speicher-Befehle
- arithmetische Befehle
- unbedingte und bedingte Sprungbefehle
- Register

mov, ...  
add, sub, inc, dec, xor, cmp, ...  
jmp, jz, jae, ...  
ax, bx, ...

### Beispiel: Atmel-AVR-8-Bit-Assembler

- Lade- und Speicher-Befehle
- arithmetische Befehle
- unbedingte und bedingte Sprungbefehle
- Register

ldi, lds, sti, ...  
add, sub, subi, eor, cp, ...  
rjmp, brsh, brlo, ...  
r0, r1, ...

$\longrightarrow$  für jeden Prozessor anders

Hochsprache  $\longrightarrow$  für jeden Prozessor gleich

## 2.6.3 Maschinensprache $\longrightarrow$ Assembler $\longrightarrow$ Hochsprachen

### Compiler-Sprachen

- *Compiler* übersetzt Hochsprachen-*Quelltext* in die Assembler-Sprache
- *Assembler* übersetzt Assembler-Quelltext in die Maschinensprache
- Compiler und Assembler sind Programme,  
geschrieben in Maschinensprache, Assembler oder einer Hochsprache
- Beispiele: Fortran, Algol, Pascal, Ada, C, C++, ...

## 2.6.3 Maschinensprache $\longrightarrow$ Assembler $\longrightarrow$ Hochsprachen

### Compiler-Sprachen

- *Compiler* übersetzt Hochsprachen-*Quelltext* in die Assembler-Sprache
- *Assembler* übersetzt Assembler-Quelltext in die Maschinensprache
- Compiler und Assembler sind Programme, geschrieben in Maschinensprache, Assembler oder einer Hochsprache
- Beispiele: Fortran, Algol, Pascal, Ada, C, C++, ...

### Interpreter- oder Skript-Sprachen

- *Interpreter* liest Hochsprachen-*Quelltext* und führt ihn sofort aus
- Der Interpreter ist ein Programm, geschrieben in Maschinensprache, Assembler oder einer Hochsprache
- Beispiele: Unix-Shell, BASIC, Perl, Python, ...



## 2.6.3 Maschinensprache $\longrightarrow$ Assembler $\longrightarrow$ Hochsprachen

### Compiler-Sprachen

- *Compiler* übersetzt Hochsprachen-*Quelltext* in die Assembler-Sprache
- *Assembler* übersetzt Assembler-Quelltext in die Maschinensprache
- Compiler und Assembler sind Programme, geschrieben in Maschinensprache, Assembler oder einer Hochsprache
- Beispiele: Fortran, Algol, Pascal, Ada, C, C++, ...

### Interpreter- oder Skript-Sprachen

- *Interpreter* liest Hochsprachen-*Quelltext* und führt ihn sofort aus
- Der Interpreter ist ein Programm, geschrieben in Maschinensprache, Assembler oder einer Hochsprache
- Beispiele: Unix-Shell, BASIC, Perl, Python, ...

### Kombinationen

- *Compiler* erzeugt *Zwischencode* für eine *virtuelle Maschine*
- *Interpreter* liest Hochsprachen-*Zwischencode* und führt ihn sofort aus
- Die virtuelle Maschine ist ein Programm, geschrieben in Maschinensprache, Assembler, einer Hoch- oder Skript-Sprache
- Beispiele: UCSD-Pascal, Java, ...

# 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 Struktur von Assembler-Programmen
- 3 Architekturmerkmale von Prozessoren**
- 4 Der CPU-Stack**
- ...

## 2.7 Struktur von Assembler-Programmen

### Beispiel 1: IA-32-Assembler

```
addl $1, %eax  
movb %al, b  
cmpb (%ebx), %dl  
jbe .L2
```

## 2.7 Struktur von Assembler-Programmen

### Beispiel 1: IA-32-Assembler

Befehl    Größen-Suffix



```
addl $1, %eax  
movb %al, b  
cmpb (%ebx), %dl  
jbe .L2
```

## 2.7 Struktur von Assembler-Programmen

### Beispiel 1: IA-32-Assembler – Adressierungsarten

Befehl    Operanden



```
addl $1, %eax
movb %al, b
cmpb (%ebx), %dl
jbe .L2
```

## 2.7 Struktur von Assembler-Programmen

### Beispiel 1: IA-32-Assembler

unmittelbar

addl \$1, %eax ← Register

movb %al, b ← Speicher (absolut)

cmpb (%ebx), %dl

jbe .L2

indirekt mit Register

Speicher (relativ)

```
graph TD
    U[unmittelbar] --> S1["addl $1, %eax"]
    R[Register] --> S1
    SA[Speicher absolut] --> S2["movb %al, b"]
    I[indirekt mit Register] --> S3["cmpb (%ebx), %dl"]
    SR[Speicher relativ] --> S4["jbe .L2"]
```

## 2.7 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

## 2.7 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



## 2.7 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

## 2.7 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

## 2.7 Struktur von Assembler-Programmen

Selbstmodifizierender Code

Beispiel: Selbsterkennung von „Fini“

```
num    dat #-2
start  mov num, <pos
      jmp start
pos    dat #-3
      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

## 2.7 Struktur von Assembler-Programmen

### Praktikumsversuch

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

ICWS-86-Standard,  
max. 64 Prozesse,  
Speichergröße zufällig  
von 512 bis 8192

Teams bis zu 3 Personen  
sind zulässig.

Ursprünglich stand oben „ICWS-88“, daher wird hier auch ICWS-88 akzeptiert. ICWS-86 ist jedoch „sportlicher“.

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