Hochschule Bochum
Bochum University
of Applied Sciences

Grundlagen Rechnertechnik

Prof. Dr. Peter Gerwinski

6. November 2012

2.6 Maschinensprache $\longrightarrow$ Assembler $\longrightarrow$ Hochsprachen

Beispiel: Intel-x86-16-Bit-Assembler

- Lade- und Speicher-Befehle — mov, ...
  arithmetische Befehle — add, sub, inc, dec, xor, cmp, ...
  unbedingte und bedingte Sprungbefehle — jmp, jz, jae, ...
- Register — ax, bx, ...

Beispiel: Atmel-AVR-8-Bit-Assembler

- Lade- und Speicher-Befehle — ldi, lds, sti, ...
  arithmetische Befehle — add, sub, subi, eor, cp, ...
  unbedingte und bedingte Sprungbefehle — rjmp, brsh, brlo, ...
- Register — r0, r1, ...

$\longrightarrow$ für jeden Prozessor anders

Hochsprache $\longrightarrow$ für jeden Prozessor gleich

2.6 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, ...

2.6 Maschinensprache $\longrightarrow$ Assembler $\longrightarrow$ Hochsprachen

C nach Assembler übersetzen:

```
$ gcc -S pruzzel.c
```

erzeugt pruzzel.s,
Assembler für den Standard-Prozessor
(hier: 32-Bit-Intel-Architektur – IA-32).

2.6 Maschinensprache $\longrightarrow$ Assembler $\longrightarrow$ Hochsprachen

C nach Assembler übersetzen:

```
$ gcc -S pruzzel.c
```

erzeugt pruzzel.s,
Assembler für den Standard-Prozessor
(hier: 32-Bit-Intel-Architektur – IA-32).

```
$ avr-gcc -S pruzzel.c
```

erzeugt pruzzel.s,
Assembler für 8-Bit-Atmel-AVR-Prozessoren.

2.6 Maschinensprache $\longrightarrow$ Assembler $\longrightarrow$ Hochsprachen

C-Programme auf Assembler-Ebene debuggen:

```
$ gcc -g pruzzel.c -o pruzzel
$ gdb -tui ./pruzzel
(gdb) break main
(gdb) run
(gdb) layout split
(gdb) nexti
```

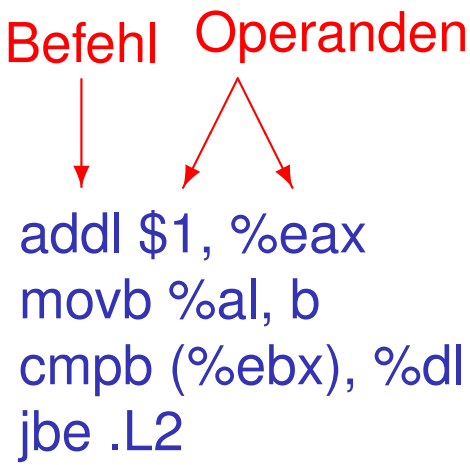
2.7 Struktur von Assembler-Programmen

IA-32-Assembler

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

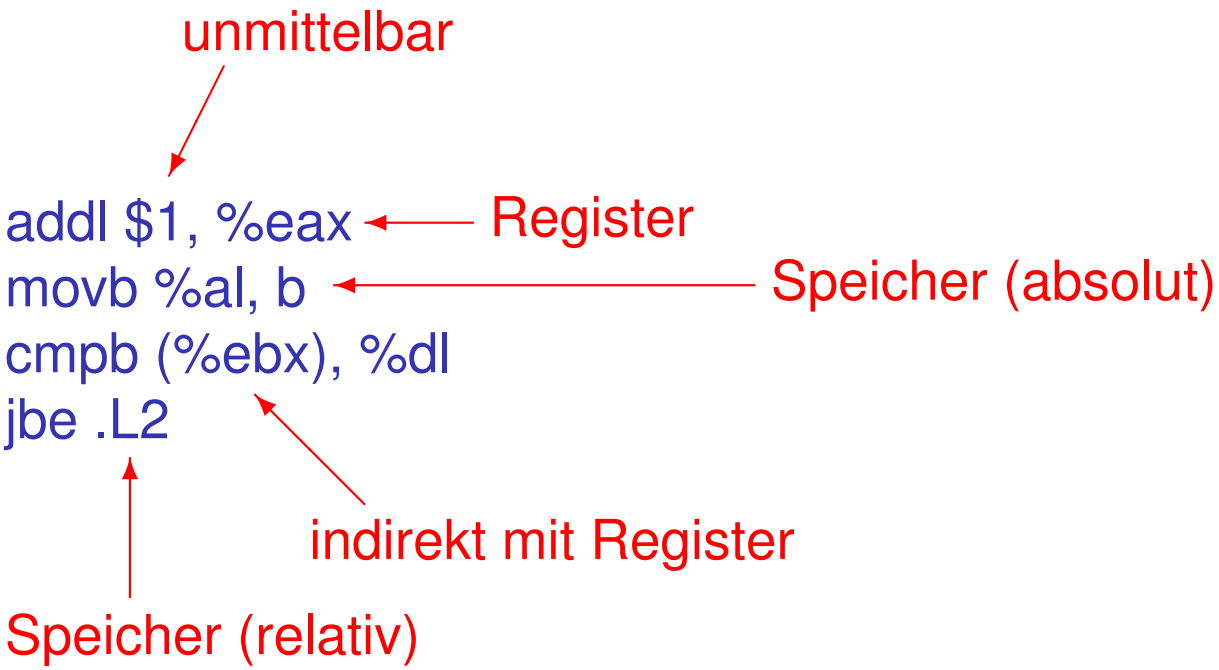
2.7 Struktur von Assembler-Programmen

IA-32-Assembler



2.7 Struktur von Assembler-Programmen

IA-32-Assembler – Adressierungsarten



2.7 Struktur von Assembler-Programmen

Core War[s] (Krieg der Kerne) – Redcode (ICWS '88)
Virtuelle Maschine: Memory Array Redcode Simulator (MARS)

Instruktionen:

dat B – Daten
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

2.7 Struktur von Assembler-Programmen

Core War[s] (Krieg der Kerne) – Redcode (ICWS ’88)
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

2.7 Struktur von Assembler-Programmen

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