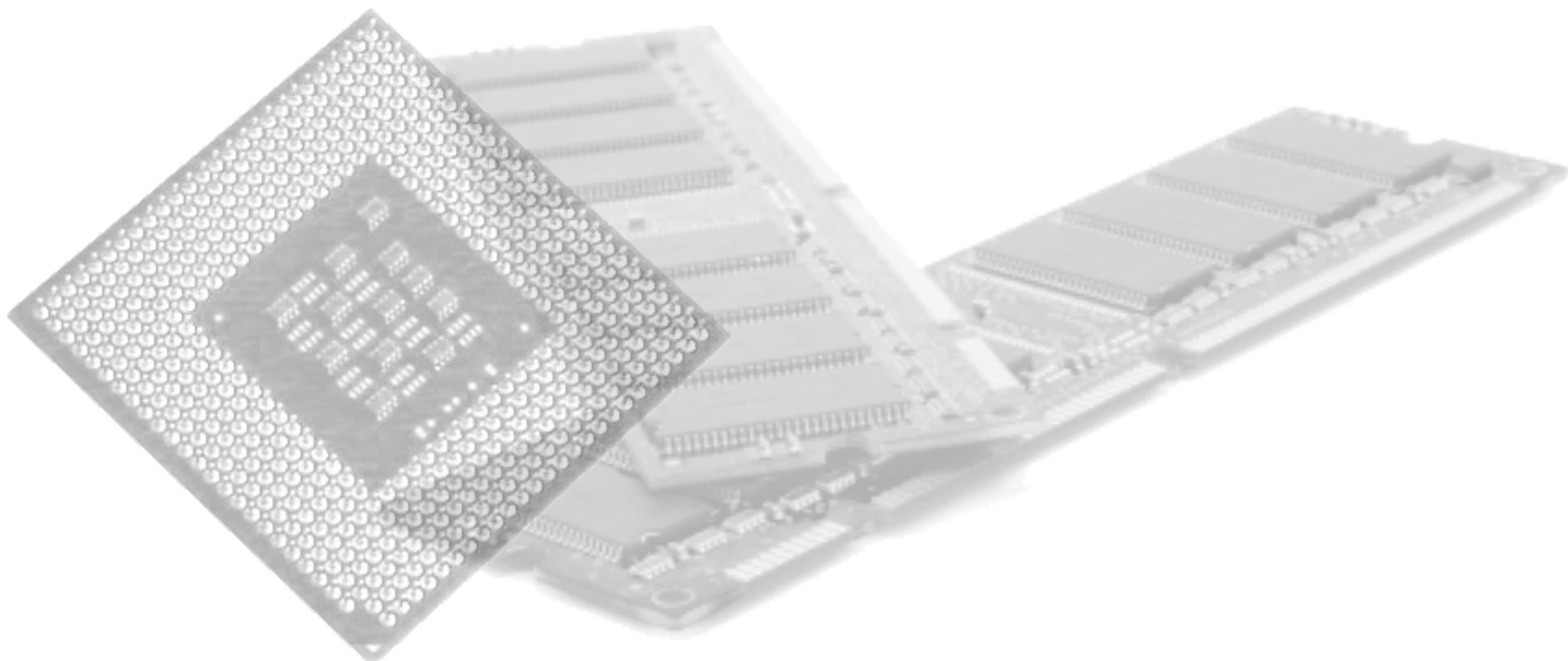
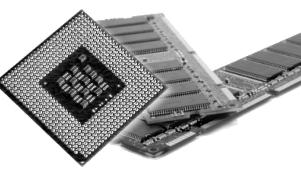


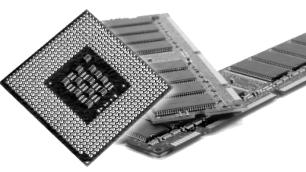
Sprungbefehle / Schleifen (Loops)





Sprungbefehle - Unbedingt

- JMP <Ziel>
 - Springe an bestimmte Stelle im Codesegment
 - Wird immer ausgeführt (! Endless Loop !)

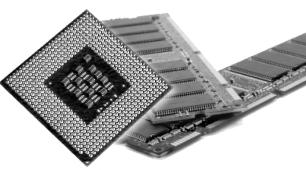


Sprungbefehle - Unbedingt

```
5 section .text
6 global CMAIN
7 CMAIN:
8     mov ebp, esp; for correct debugging
9
10    mov eax, 12h
11    add eax, 8h
12    jmp Springe
13
14    mov eax, 12h ← Befehl wird nicht ausgeführt!
15
16 Springe:
17    PRINT_HEX 4, eax
18
19    xor eax, eax
20    ret
```

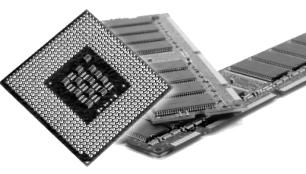


1a



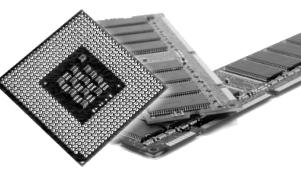
Sprungbefehle - Bedingt

Befehl	Beschreibung	Vorzeichen	Flags
JO	Overflow		OF = 1
JNO	kein Overflow		OF = 0
JS	Vorzeichen		SF = 1
JNS	kein Vorzeichen		SF = 0
JE	Übereinstimmt (Vergleich)		ZF = 1
JZ	Null		
JNE	nicht Übereinstimmt (Vergleich)		ZF = 0
JNZ	nicht Null		
JB	kleiner		
JNAE	nicht größer oder gleich	kein Vorzeichen	CF = 1
JC	Carry		
JNB	wenn es nicht kleiner ist		
JAE	größer oder gleich	kein Vorzeichen	CF = 0
JNC	nicht Carry		
JBE	kleiner oder gleich		
JNA	nicht größer	kein Vorzeichen	CF = 1 oder ZF = 1
JA	größer		
JNBE	nicht kleiner oder gleich	kein Vorzeichen	CF = 0 und ZF = 0



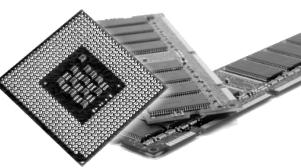
Sprungbefehle - Bedingt

Befehl	Beschreibung	Vorzeichen	Flags
JL JNGE	kleiner nicht größer oder gleich	Vorzeichen	SF <> OF
JGE JNL	größer oder gleich nicht kleiner	Vorzeichen	SF = OF
JLE JNG	kleiner oder gleich nicht größer	Vorzeichen	ZF = 1 oder SF <> OF
JG JNLE	größer nicht kleiner oder gleich	Vorzeichen	ZF = 0 und SF = OF
JP JPE	Gerade gerade Parität		PF = 1
JNP JPO	Ungerade ungerade Parität		PF = 0
JCXZ JECXZ	CX Register ist 0 ECX Register ist 0		CX = 0 ECX = 0



Sprungbefehle - Bedingt

- Prüfe CPU-Flag(s) oder Register
- Falls Bedingung erfüllt: Sprung zur Marke
- Sonst Weiterführung



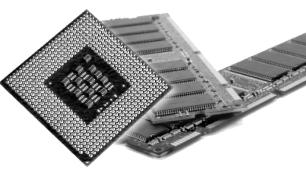
Sprungbefehle - Bedingt

- ist AX (100) kleiner als 80 ?

```
MOV AX, 100
CMP AX, 80
JL k1
PRINT_STRING groesser
ret
k1:
PRINT_STRING kleiner
```

Output

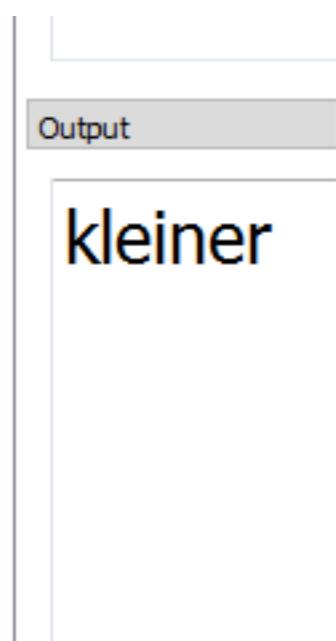
groesser / gleich



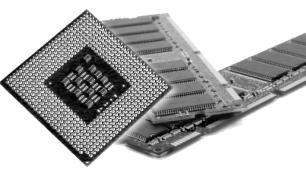
Sprungbefehle - Bedingt

- Bei -100 ???

```
MOV AX, -100
CMP AX, 80
JL kl
PRINT_STRING groesser
ret
kl:
PRINT_STRING kleiner
```



- JL beachtet Vorzeichen!



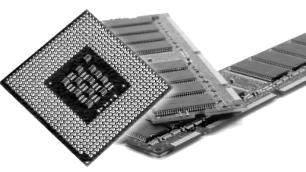
Sprungbefehle - Bedingt

- Mit **JB** wird das Vorzeichen ignoriert!!!

```
MOV AX, -100
CMP AX, 80
JB k1
PRINT_STRING groesser
ret
k1:
PRINT_STRING kleiner
```

Output

groesser / gleich



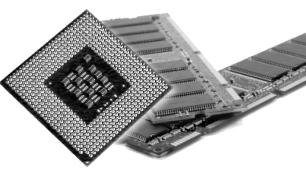
Sprungbefehle - Bedingt

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6     mov ebp, esp; for correct de
7     ;write your code here
8     mov eax, 0h
9     addiere:
10    add eax, 198h
11    jnc addiere
12    PRINT_HEX 4, eax
13    xor eax, eax
14    ret
```

The screenshot shows a debugger interface with three main panes:

- Input:** An empty text input field.
- Output:** An empty text output field.
- Registers:** A table showing CPU register values:

Register	Hex	Info
eax	0x00000098	152
ecx	0x004018cc	4200652
edx	0x0008e3c8	582600
ebx	0x7efde000	2130567168
esp	0x0028ff2c	0x28ff2c
ebp	0x0028ff2c	0x28ff2c
esi	0x00000000	0
edi	0x00000000	0
eip	0x0040139e	0x40139e <addiere>
eflags	0x00000203	[CF IF]
cs	0x00000023	35
ss	0x0000002b	43
ds	0x0000002b	43
es	0x0000002b	43



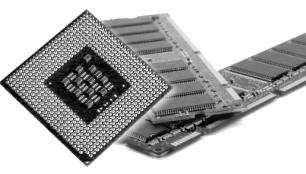
Schleifenbefehle

- LOOP <Ziel>

```
MOV AX, 0
MOV CX, 100
Schleife:
ADD AX, 3
LOOP Schleife
PRINT_DEC 2, AX
```

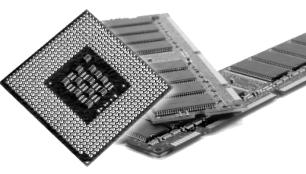


- Verwendet Counter-Register (CX) für die Anzahl der Durchläufe
- Ablauf: CX wird um eins verringert, anschließend Prüfung auf 0
- Bei 0 Weiterführung, sonst Sprung

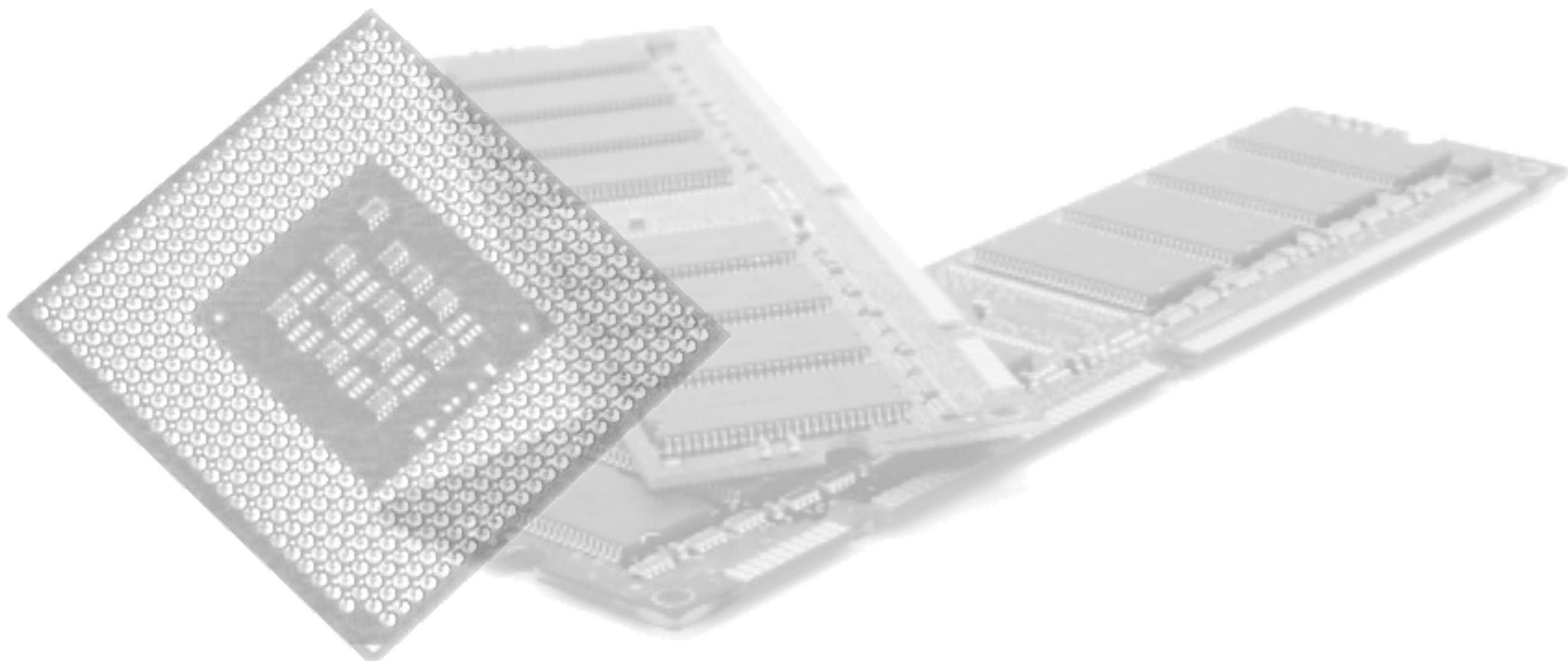


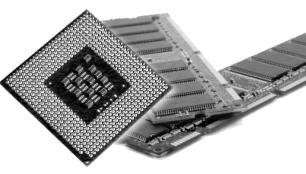
Schleifenbefehle

- Schleifen mit zusätzlichen Prüfungen
 - **LOOPE / LOOPZ <Ziel>** (*solange Zero-Flag = 1*)
 - **LOOPNE / LOOPNZ <Ziel>** (*solange Zero-Flag = 0*)
- Ähnlich zu Loop
- **Zusätzlich** Prüfung des Zero-Flags



Vergleichsbefehle

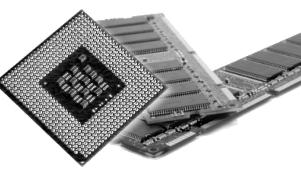




Vergleichsbefehle

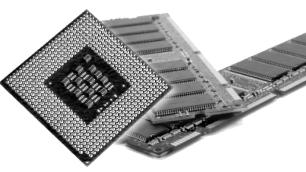
- Compare (nummerisch) - CMP
 - **CMP <Ziel>, <Quelle>**
 - Nummerischer Vergleich von Quelle mit Ziel
 - Vergleich wird durch Subtraktion durchgeführt (wobei das Ergebnis nicht gespeichert wird)
 - Nutz CPU-Flags für Ergebnis:

cmp dst, src	ZF	CF
dst = src	1	0
dst < src	0	1
dst > src	0	0



Vergleichsbefehle

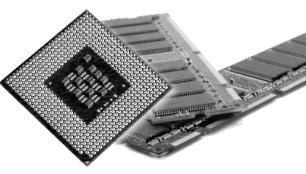
- Compare String – CMPS
 - **CMPSB** (Compare String Byte)
 - **CMPSW** (Compare String Word)
 - **CMPSD** (Compare String Double)
 - **CMPSQ** (Compare String Quad)
 - Vergleiche ein Zeichen von der Speicherstelle die in DS:SI steht mit der von ES:DI
 - Entspricht ebenfalls Subtraktion



Vergleichsbefehle

```
4 global CMAIN
5 CMAIN:
6     ; write your code here
7     mov eax, 1
8     cmp eax, 1
9     je gleich
10    jb kleiner
11    PRINT_STRING "größer"
12    jmp ende
13    gleich:
14    PRINT_STRING "gleich"
15    jmp ende
16    kleiner:
17    PRINT_STRING "kleiner"
18    ende:
19    xor rax, rax
```

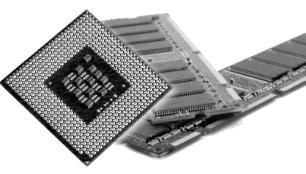
The screenshot shows a debugger interface with two panes. The top pane, labeled 'Input', contains the assembly code from line 4 to 19. The bottom pane, labeled 'Output', shows the result of the execution: the word 'gleich'.



Vergleichsbefehle

```
3 section .data
4 txt1 DB "a", 0
5 txt2 DB "b", 0
6 section .text
7 global CMAIN
8 CMAIN:
9     mov rbp, rsp; for correct debugging
10    ; write your code here
11    mov rsi, txt1
12    mov rdi, txt2
13    cmpsb
14    je gleich
15    PRINT_STRING "ungleich"
16    jmp ende
17    gleich:
18    PRINT_STRING "gleich"
```

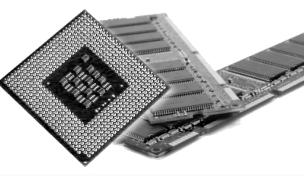
The screenshot shows a debugger interface with two main panes. The top pane is labeled 'Input' and displays the assembly code from line 3 to 18. The bottom pane is labeled 'Output' and displays the string 'ungleich'.



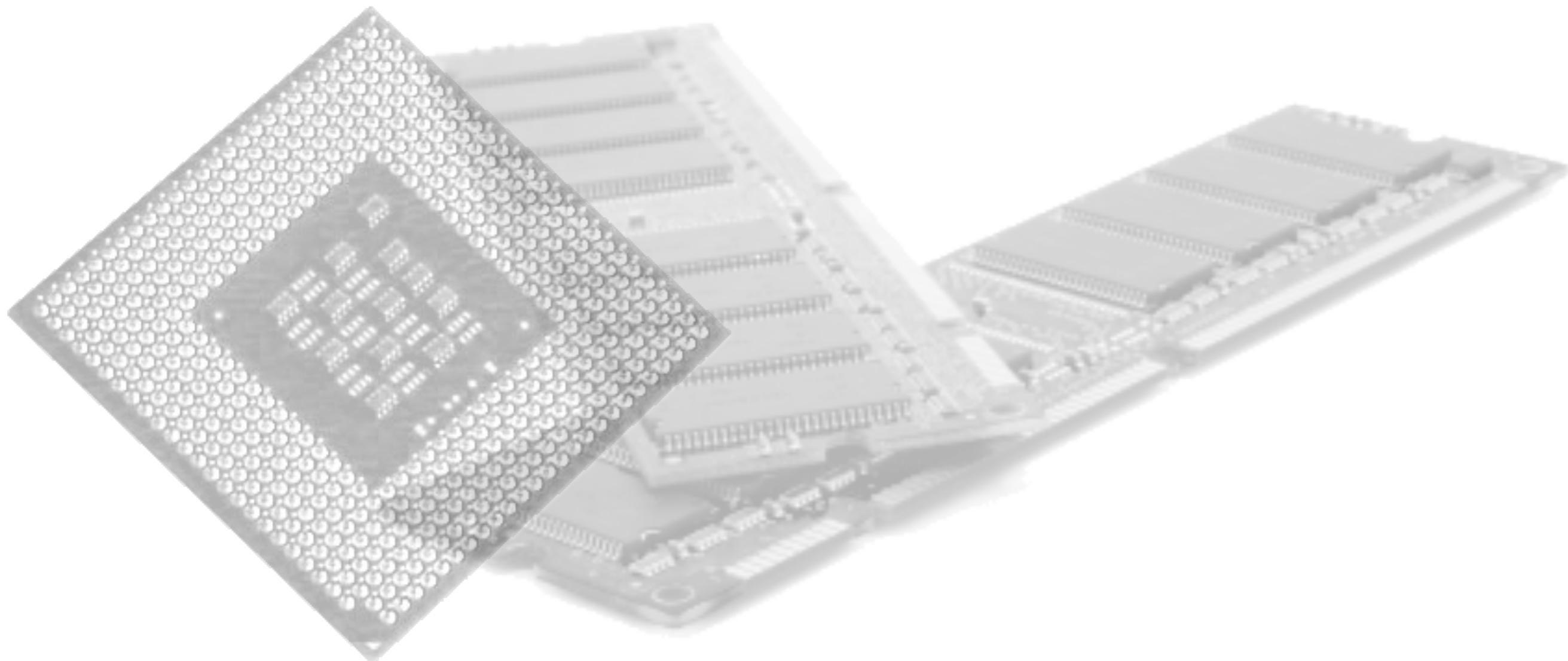
Vergleichsbefehle

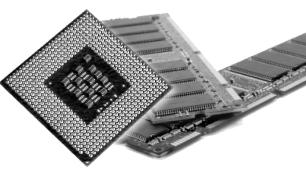
```
3 section .data
4 txt1 DB "Test", 0
5 txt2 DB "Test", 0
6 section .text
7 global CMAIN
8 CMAIN:
9     mov rbp, rsp; for correct debugging
10    ; write your code here
11    mov rsi, txt1
12    mov rdi, txt2
13    mov rcx, 5
14    repe cmpsb
15    je gleich
16    PRINT_STRING "ungleich"
17    jmp ende
18    gleich:
```

The screenshot shows a debugger interface with two main panes: 'Input' and 'Output'. The 'Input' pane contains the assembly code from line 3 to 18. The 'Output' pane shows the execution results: the word 'gleich' is displayed, indicating that the comparison between the two strings resulted in equality.



Makros / Unterprogramme

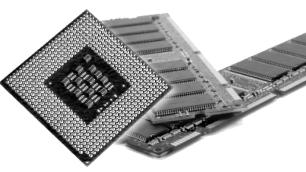




Makros

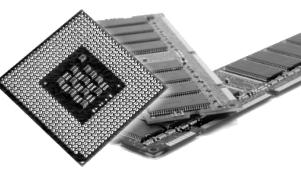
- %macro <Name> <Anzahl Parameter>
 - Code
 - Parameter mit %1, %2, %3, usw. verwendbar
- %endmacro

```
%macro AUSGEBEN 1
    NEWLINE
    PRINT_HEX 4, %1
%endmacro
```

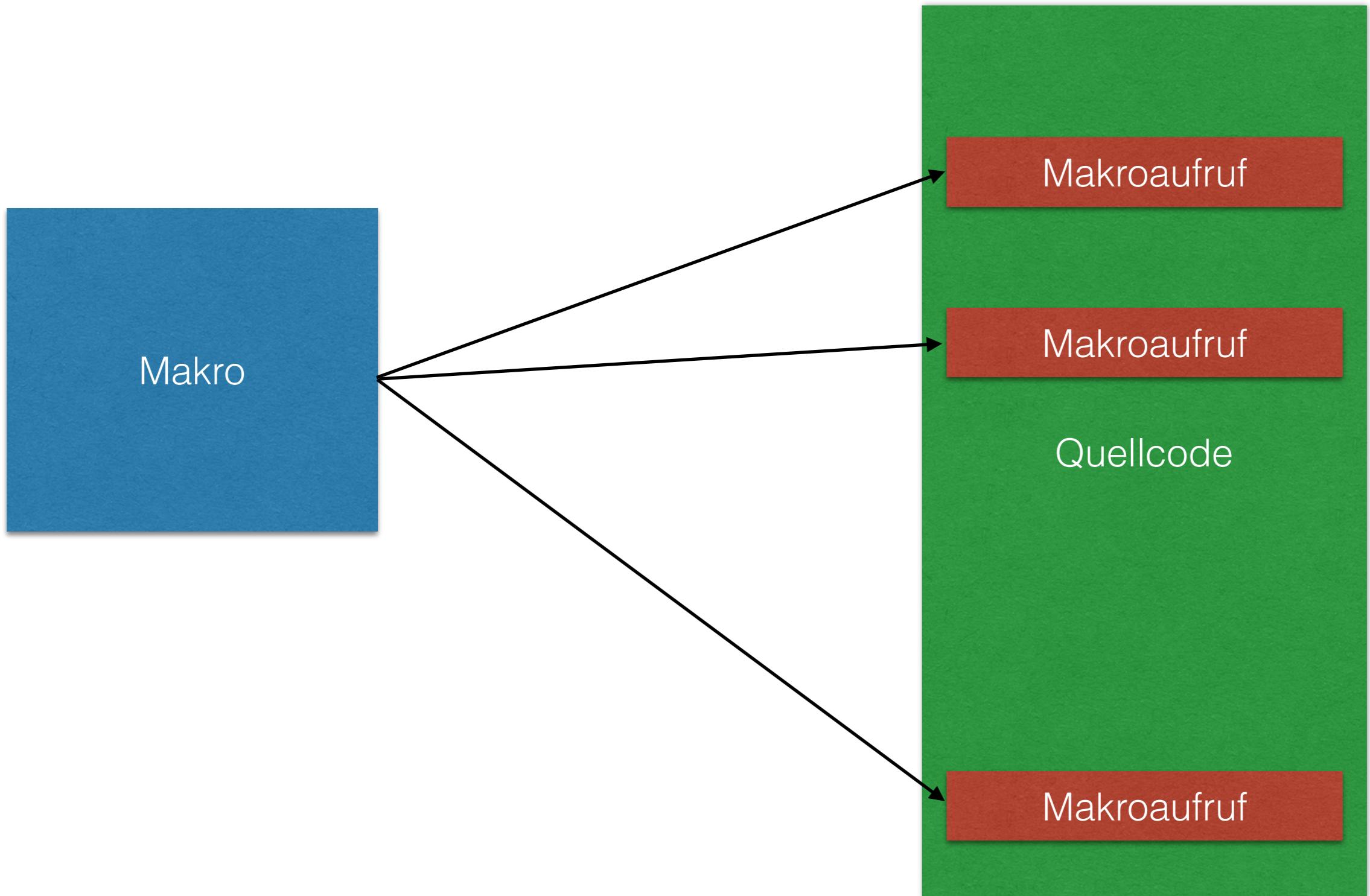


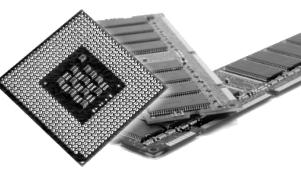
Makros

- Aufruf erfolgt per Name (mit Parametern)
 - AUSGEBEN 4
 - <Name> <Parameter>, <Parameter>, ...
- Beim Kompilieren wird jeder Aufruf durch den Makrocode ersetzt
- Das Makro kann an jeder Stelle im Quellcode stehen (ABER: vor den ersten Aufruf)
- Praktischer ist es die Makros in eine externe Datei auszulagern (siehe io.inc)

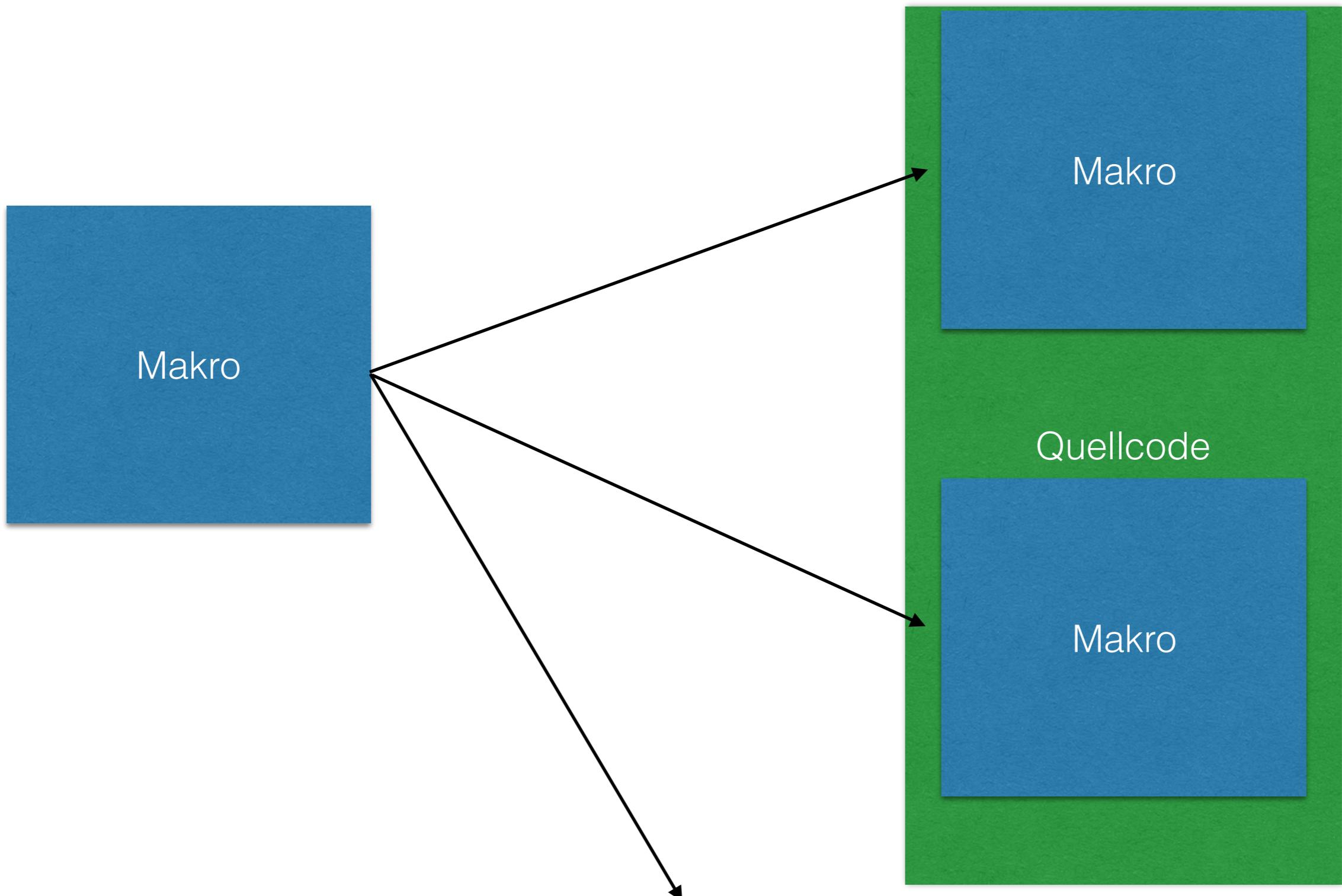


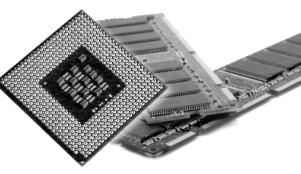
Makros





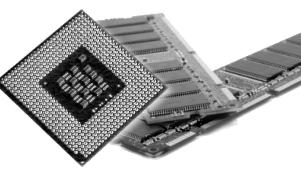
Makros





Makros

- Vorteile
 - “Erweiterung” des Befehlssatzes
 - individuelle höhere Assemblersprache
 - schnellere Entwicklung
- Nachteile
 - hoher Speicherplatzbedarf (nach dem Ersetzen)



Unterprogramme

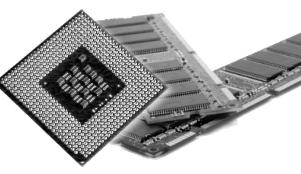
- Vorteile
 - häufig vorkommende Programmteile nur einmal vorhanden
 - modularer Programmaufbau
 - Unterprogrammbibliothek
- Nachteile
 - anspruchsvollere Programmierarbeit



Unterprogramme

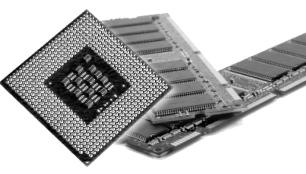
- <Name>:
 - Code

```
proz:  
PRINT_HEX 4, eax  
ret
```
 - RET
- Aufruf per Call: CALL <Name>



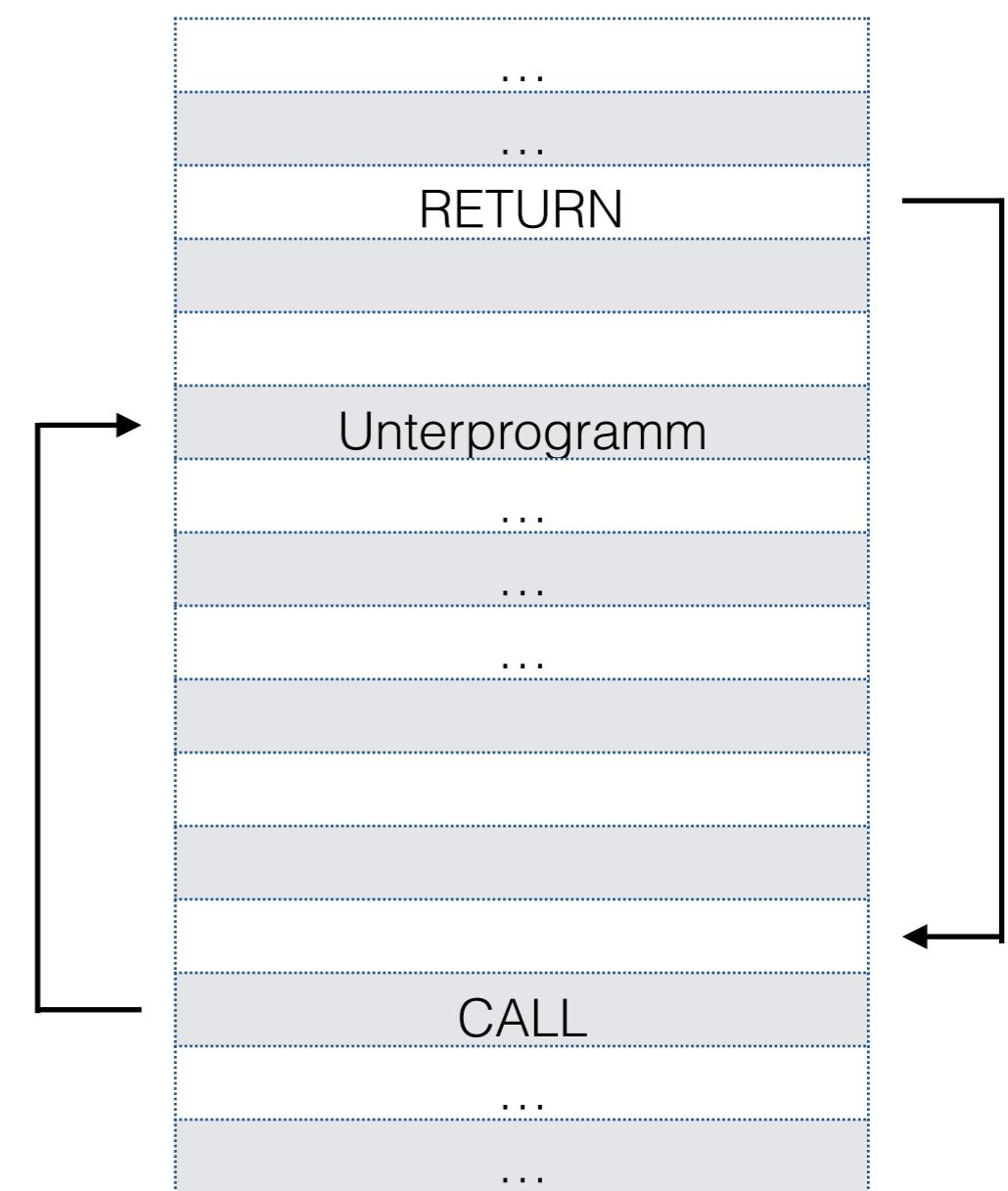
Unterprogramme

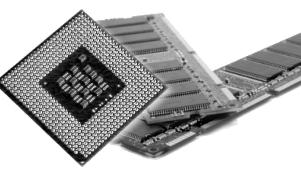
- keine Parameterübergabe möglich
 - entweder über die Register
 - oder über den Stack



Unterprogramme

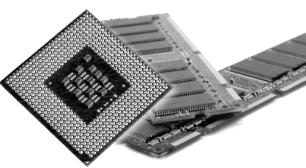
- Aufruf: CALL
- Rücksprung: RETURN
- Intrasegment Call:
 - IP auf den Stack
 - bei Return wieder zurückschreiben
- Intersegment Call:
 - CS und IP auf den Stack
 - bei Return wieder zurückschreiben





Define

- %define <Name> <Ersetzen>
- Bsp.: %define haha 2
- mov eax, haha
- Beim Kompilieren: mov eax, 2
- Anweisungen mit % sind für den Compiler



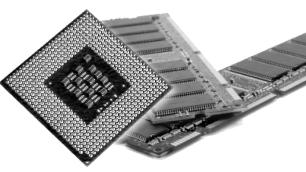
Define

```
1 %include "io.inc"
2 %define m mov
3
4 section .text
5 global CMAIN
6 CMAIN:
7     ;write your code here
8
9     m eax, 12h
10    PRINT_HEX 4, eax
11
12    ret
```

Input

Output

12

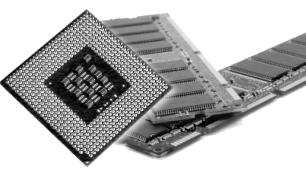


Demo

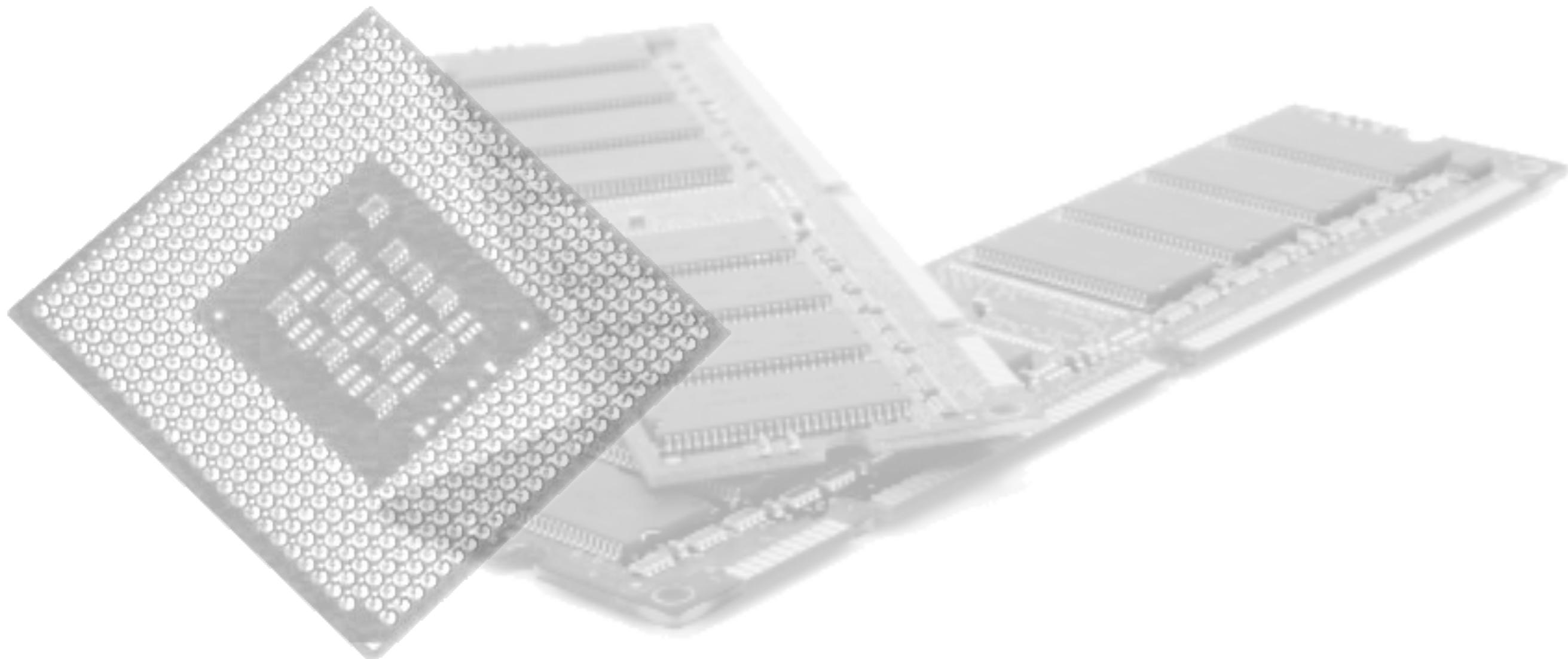
```
1 %include "io.inc"
2 %macro AUSGEBEN 1
3     NEWLINE
4         PRINT_HEX 4, %1
5 %endmacro
6 %define haha 2
7
8 section .text
9 global CMAIN
10 CMAIN:
11     mov ebp, esp; for correct debugging
12
13     mov eax, haha
14     CALL proz
15     NEWLINE
16     PRINT_HEX 4, ebx
17     AUSGEBEN 4
18     ret
19
20     proz:
21     PRINT_HEX 4, eax
22     ret
```

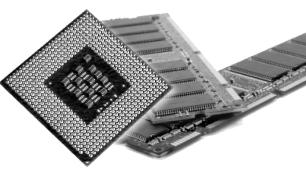
The screenshot shows a debugger window with two panes: 'Input' and 'Output'. In the 'Input' pane, assembly code is displayed. In the 'Output' pane, the results of the assembly code execution are shown:

Output
2
7efde000
4



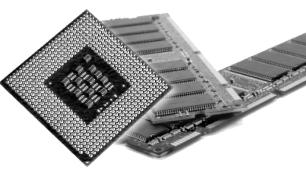
C-Funktionen





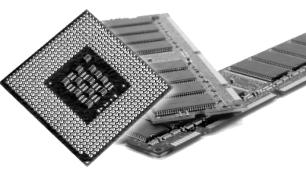
C-Funktionen in Assembler

- In Assembler können auch C-Funktionen aufgerufen werden
- Hilfreich bei der Ein- und Ausgabe
- z.B. printf
- Parameterübergabe über die Register (oder Stack) möglich



Bsp. printf (64 Bit Mac)

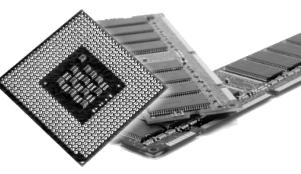
```
1 section .data
2     msg    db    "Hello world!", 0
3     fmt    db    "%s", 10, 0
4
5 section .text
6
7 global main
8 extern printf
9
10 main:
11     mov rbp, rsp; for correct debugging
12
13     push rbp
14
15     mov rdi, fmt
16     mov rsi, msg
17     mov rax, 0
18     call printf
19
20     pop rbp
21
22     xor rax, rax
23     ret
```



Bsp. printf (64 Bit Mac)

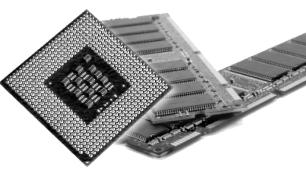
- extern printf
- Stack leeren (sichern)
- rdi -> erster Parameter
- rsi -> zweiter Parameter
- rax -> leeren (Return Value)
- call printf
- Stack wiederherstellen

```
1 section .data
2     msg    db      "Hello world!", 0
3     fmt    db      "%s", 10, 0
4
5 section .text
6
7 global main
8 extern printf
9
10 main:
11     mov rbp, rsp; for correct debug
12
13     push rbp
14
15     mov rdi, fmt
16     mov rsi, msg
17     mov rax, 0
18     call printf
19
20     pop rbp
21
22     xor rax, rax
23     ret
```



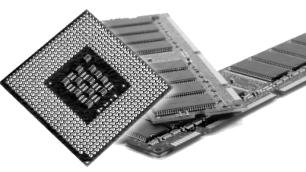
Funktionen Register

Register	Funktion
rax	return value
rdi	1. argument
rsi	2. argument
rdx	3. argument
rcx	4. argument
r8	5. argument
r9	6. argument



DEMO printf (64 Bit Mac)

```
1  section .data
2      msg      db      "Hello world!",10, 0
3      msg2     db      "Hello world!",10, 0
4      fmt      db      "%s%s", 10, 0
5
6  section .text
7
8  global main
9  extern printf
10
11 main:
12     mov rbp, rsp; for correct debugging
13
14     push rbp
15
16     mov rdi, fmt
17     mov rsi, msg
18     mov rdx, msg2
19     mov rax, 0
20     call printf
21
22     pop rbp
23
24     xor rax, rax
25     ret
```



printf Win32

```
1 %include "io.inc"
2 section .data
3 fmt DB "Test: %d %d", 0
4 Zahl1 DD 0xA
5
6 section .text
7 global CMAIN
8 extern printf
9 CMAIN:
10    mov ebp, esp; for correct debugging
11    ;write your code here
12    mov eax, 0xFF
13
14    push eax
15    push dword [Zahl1]
16    push fmt
17
18    call printf
19
20    mov esp, ebp
21
22    xor eax, eax
23    ret
```