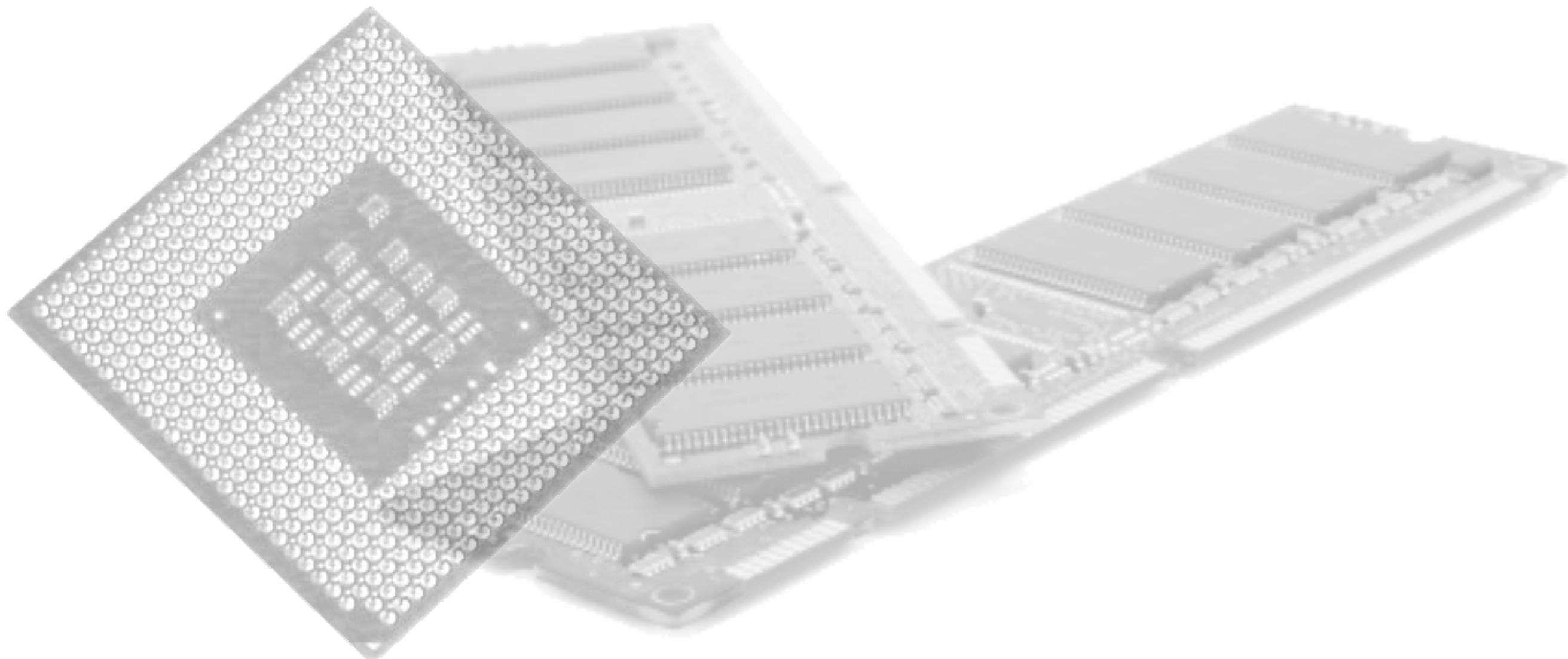
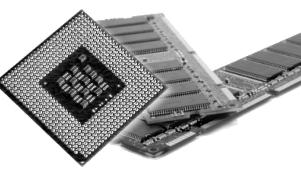


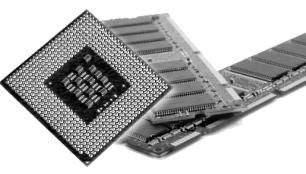
Adressierungsarten





Adressierungsarten

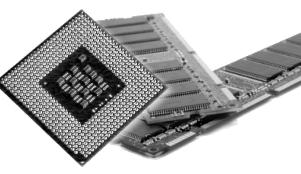
- unmittelbare Adressierung
- direkte / absolute Adressierung
- Registeradressierung
- Registerindirekte Adressierung
- Indizierte Adressierung



unmittelbare Adressierung

- Bsp.:

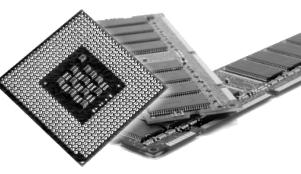
```
MOV EAX, 0FFFFh
ADD EAX, 45h
```
- Quelloperand ist eine Konstante
- Kann nie als Ziel stehen
- die Konstante steht direkt hinter dem Opcode im Speicher



direkte / absolute Adressierung

- Bsp.:

```
MOV EAX, [1000]
```
- Ein Operand ist eine Speicheradresse
- Adressierung über DS und ES beachten
 - ohne Angabe: $(DS * 16) + WERT$



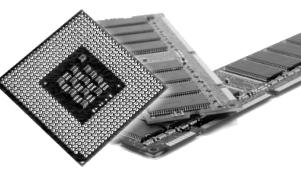
Registeradressierung

- Bsp.:

MOV EAX, EBX

INC EAX

- Alle Operanden stehen in Registern
- Kompakter Opcode
- Schnelle Ausführung



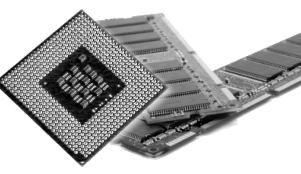
Registerindirekte Adressierung

- Registerindirekte Adressierung

```
MOV ECX, [EBX] ; DS:EBX-Inhalt
```

```
ADD EAX, [EBX] ; DS:EBX-Inhalt
```

- Inhalt des Indexregisters ist die Speicheradresse



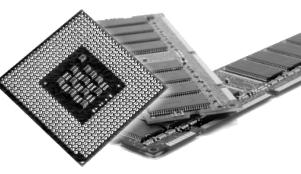
Registerindirekte Adressierung

- Registerindirekte Adressierung plus Displacement

```
MOV ECX, [EBX + 4] ; DS:EBX-Inhalt + 4
```

```
ADD EAX, [EBX + 100] ; DS:EBX-Inhalt + 100
```

- Inhalt des Indexregisters plus konstante Verschiebung



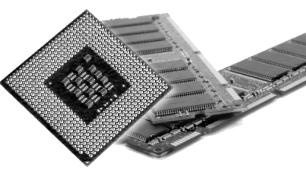
Indizierte Adressierung

- Bsp.:

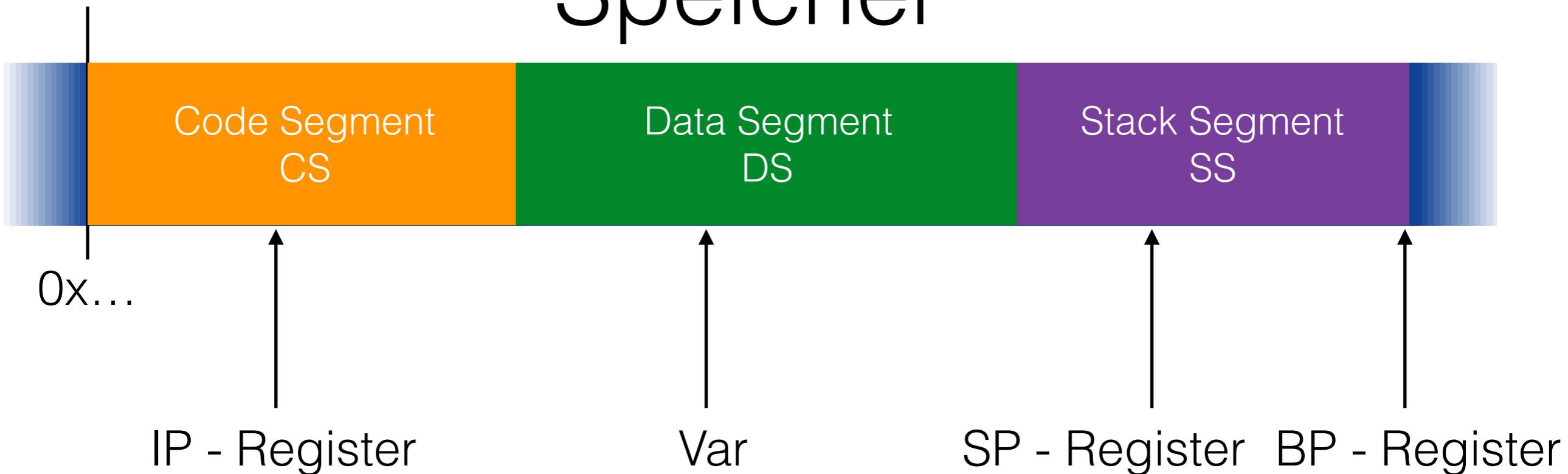
MOV EAX, [ESI+EBX]

ADD EDX, [ESI+4]

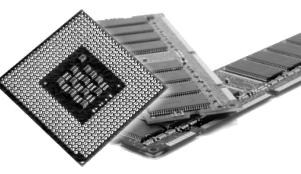
- Neben einer Konstanten auch noch ein Variabler Teil (z.B. SI Register)



Speicher

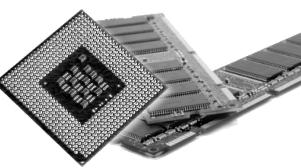


- IP: Adresse des aktuellen Befehls (Instruction Pointer)
- Adresse der Variablen in Daten werden berechnet:
 - DS + Var



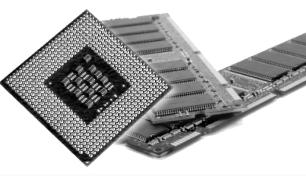
Zusammenfassung

- `mov eax, ebx` **Inhalt** von ebx
- `mov eax, [1122]` **Inhalt** von **Speicherstelle** 1122
- `mov eax, [ebx]` In ebx steht die **Adresse** der **Speicherstelle**
- `mov eax, [Var]` **Inhalt** der **Speicherstelle** Var
- `mov eax, Var` **Adresse** der **Speicherstelle** Var

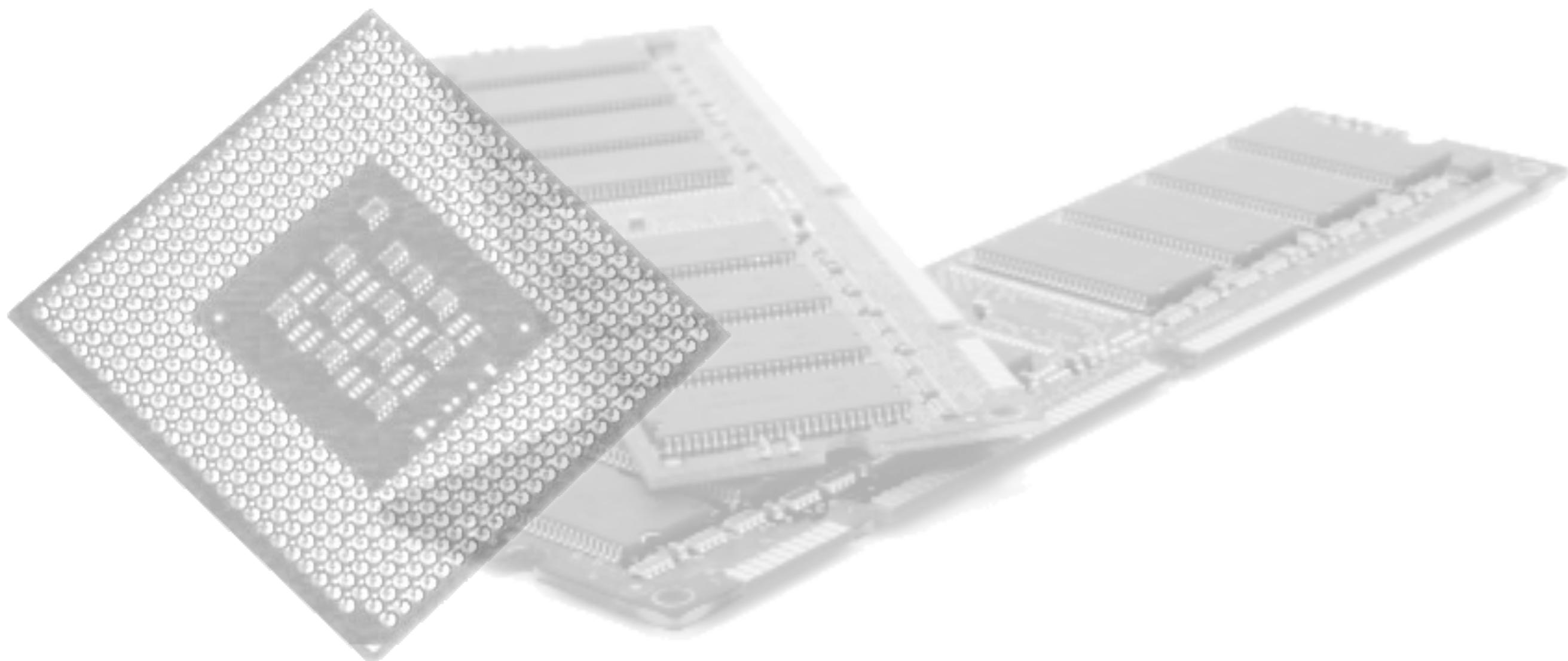


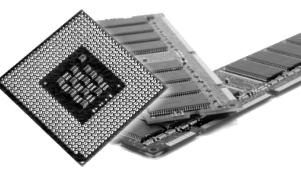
Demo Adressierung

```
2 %include "io64.inc"
3
4 section .data
5
6 Var DQ 0xFFFF
7
8 section .text
9 global CMAIN
10 CMAIN:
11     ;write your code here
12     mov rax, [Var]
13
14     mov rax, Var
15
16     mov rbx, 0x1111
17
18     mov [rax], rbx
19
20     xor rax, rax
21     ret
22
```



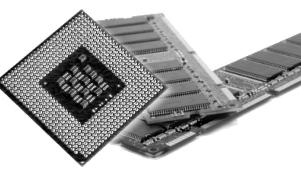
Shift-Operationen





SHL

- SHL <Ziel>, <Anzahl>
- Verschiebt den Inhalt des Zieloperanden und die angegebene Anzahl an Bits nach links
- rechts werden Nullen nachgeschoben
- links herausfallende Bits gehen verloren
- jede Verschiebung entspricht einer Multiplikation mit 2
- Anzahl kann auch das CL Register sein
- shl eax, 2 (2 Bit nach links verschieben = * 4)
- shl eax, cl



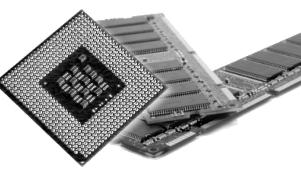
SHL

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---



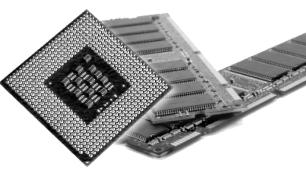
0	0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---	---

- $1101_2 = 13_{10}$
- $11010_2 = 26_{10}$

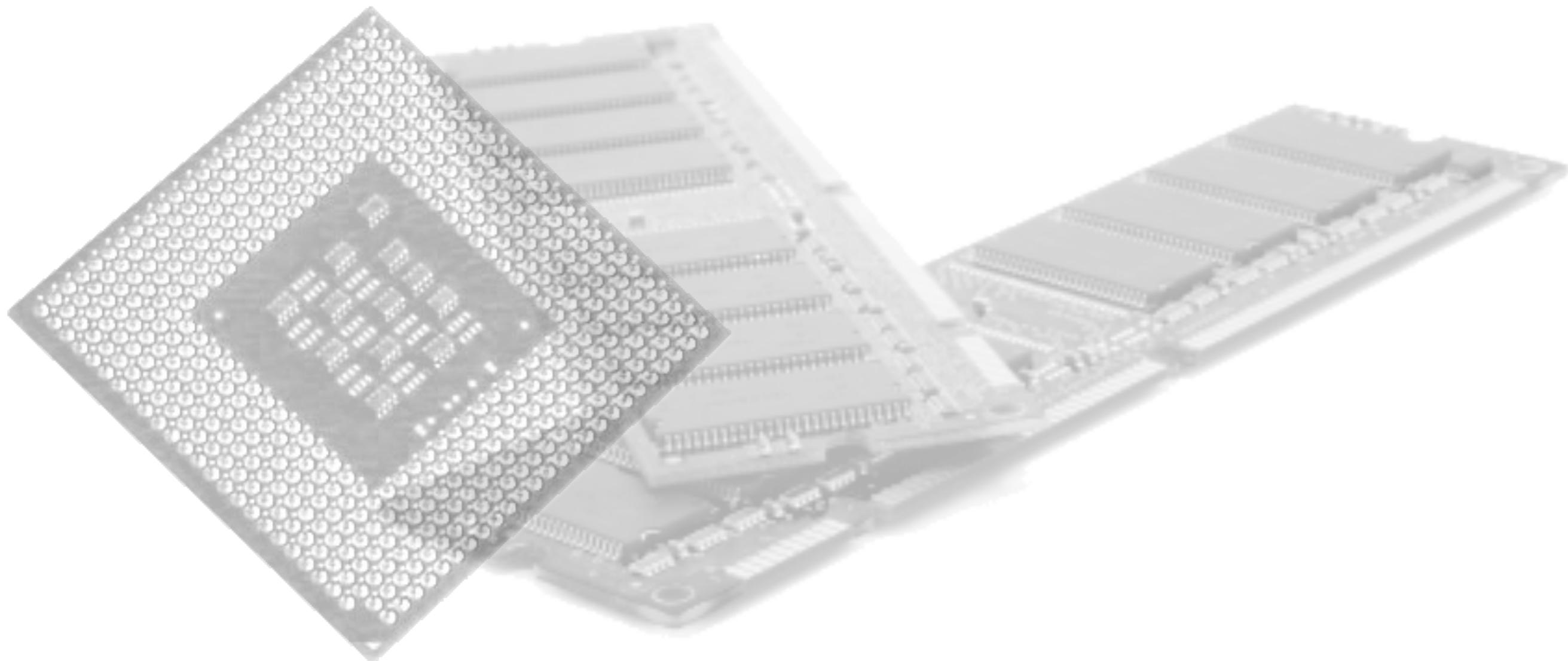


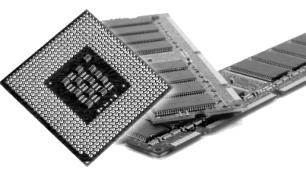
SHR

- SHR <Ziel>, <Anzahl>
- Verschiebt den Inhalt des Zieloperanden und die angegebene Anzahl an Bits nach rechts
- links werden Nullen nachgeschoben
- rechts herausfallende Bits gehen verloren
- jede Verschiebung entspricht einer Division mit 2
- Anzahl kann auch das CL Register sein
- shr eax, 1 (1 Bit nach rechts verschieben = / 2)
- shr eax, cl



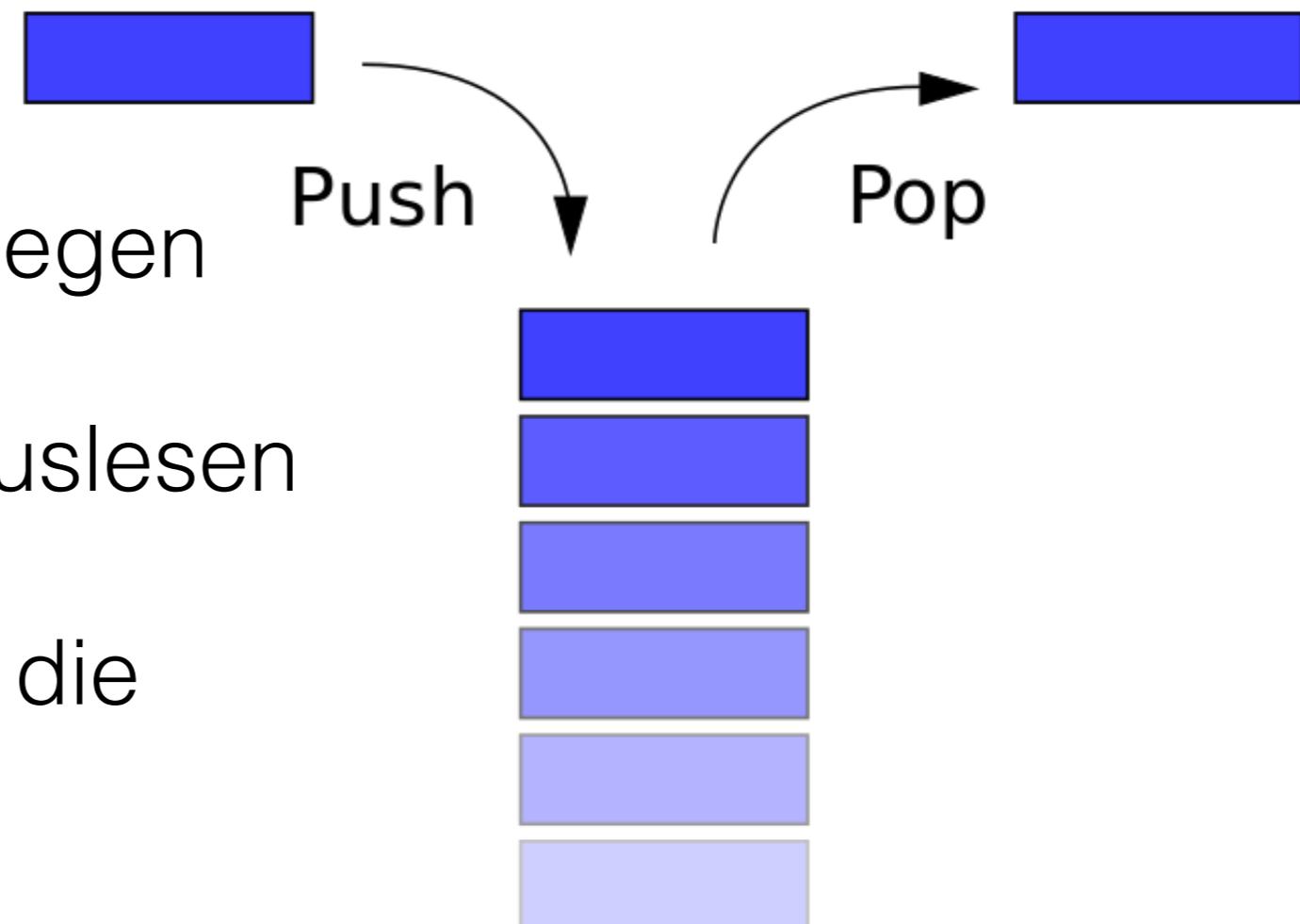
Stack

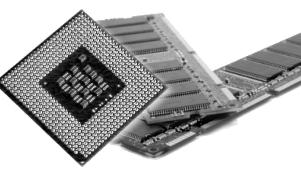




Stack

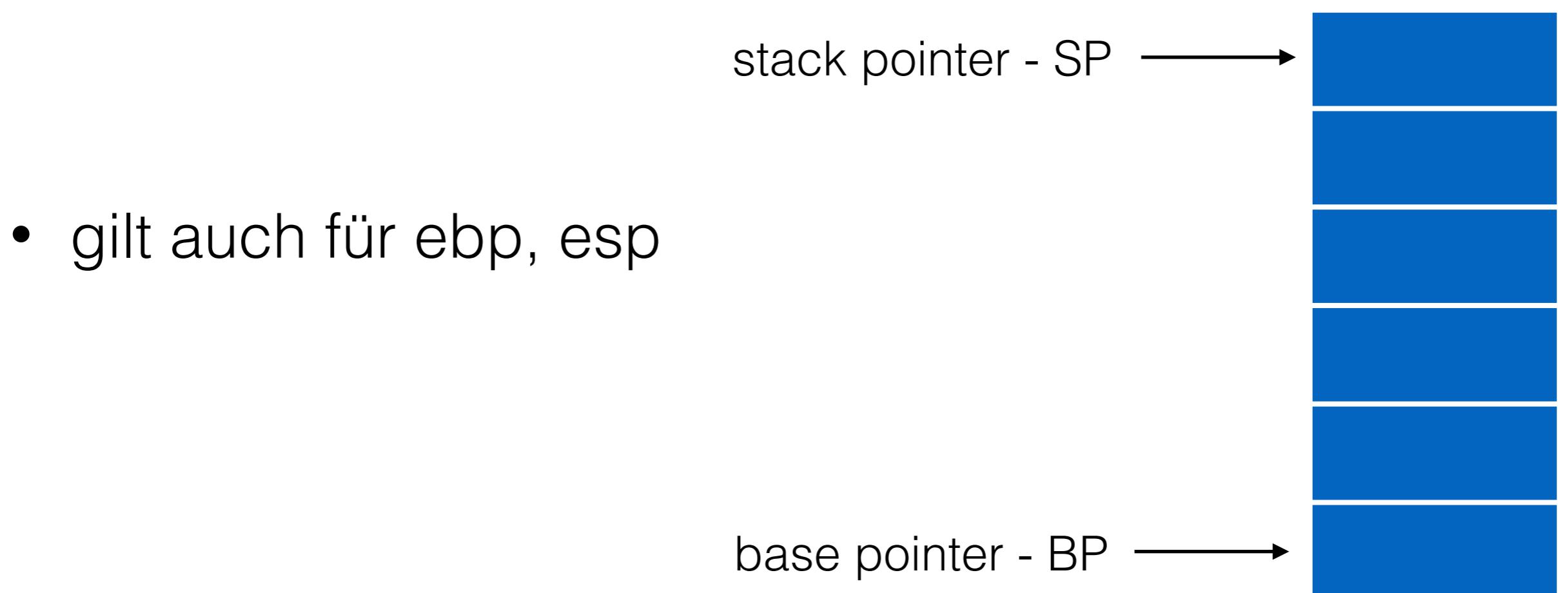
- **LIFO** - last in, first out
- PUSH - Daten oben ablegen
- POP - Oberste Daten auslesen
- kein direkter Zugriff auf die unteren Daten möglich

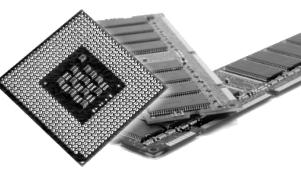




Stack

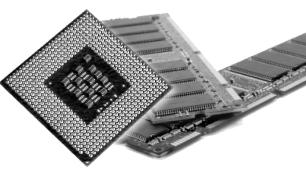
- Befindet sich im Speicher
- Adresse durch SP und BP





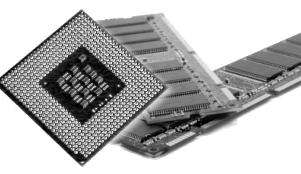
PUSH

- push <Quelle>
- Wert wird auf dem Stack abgelegt
- Stackpointer wird entsprechend angepasst (verringert)



POP

- pop <Ziel>
- In Ziel wird der oberste Wert vom Stack geschrieben
- Stackpointer wird entsprechend angepasst (erhöht)
- Wert ist immer noch im Speicher, aber mittels pop kann nicht mehr darauf zugegriffen werden, da der Stackpointer auf das vorherige Element zeigt

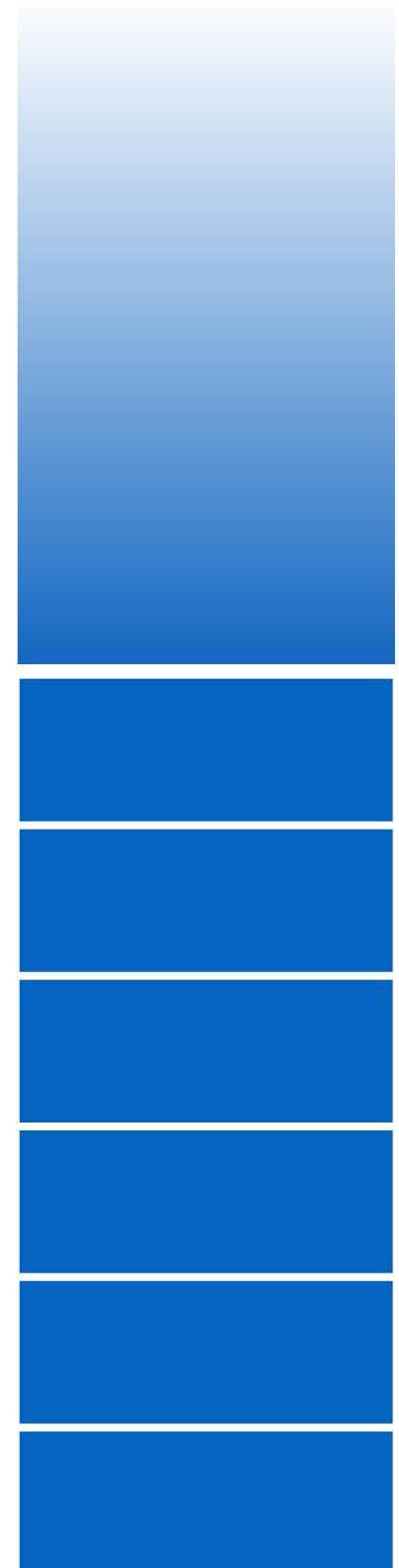


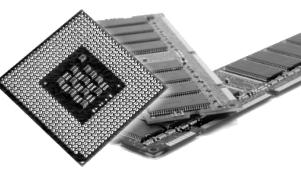
Stack

- Stack am Ende vom Speicher
- => Stack Pointer Adresse verringert sich je mehr Elemente im Stack sind
- $SP = BP \Rightarrow$ leerer Stack

stack pointer - SP →

base pointer - BP →





Stack direkt ändern

mov eax, 12h

push eax

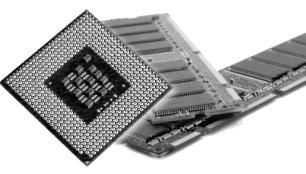
mov DWORD [esp], 22h

PRINT_HEX 4, [esp]

pop eax

NEWLINE

PRINT_HEX 4, eax



Stack direkt ändern

```
mov eax, 12h
push eax

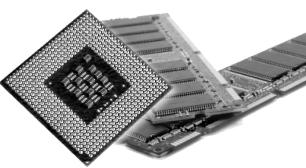
mov DWORD [esp], 22h

PRINT_HEX 4, [esp]

pop eax

NEWLINE
PRINT_HEX 4, eax
```

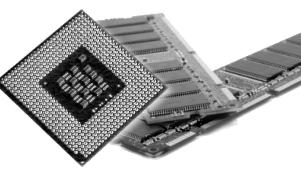
The screenshot shows a debugger interface with a vertical scrollbar on the right. On the left, there is a text area labeled "Output". Inside the output area, the value "22" is printed twice, once on each line. This indicates that the value 22 was pushed onto the stack and then popped off again.



Stack Demo

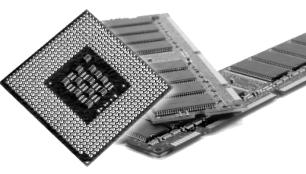
```
7 ;write your code here
8
9     mov eax, 0fffff12fh
10    push eax
11
12    mov DWORD [esp], 22h
13
14    PRINT_HEX 4, [esp]
15
16    pop eax
17
18    sub esp, 4
19 →    pop eax
20
21    NEWLINE
22    PRINT_HEX 4, eax
23
```

Register	Hex	Info
eax	0x00000022	34
ecx	0x004019b8	4200888
edx	0x0008e3c8	582600
ebx	0x7efde000	2130567168
esp	0x0028ff28	0x28ff28
ebp	0x0028ff2c	0x28ff2c
esi	0x00000000	0
edi	0x00000000	0
eip	0x00401415	0x401415 <main+13
eflags	0x00000206	[PF IF]
cs	0x00000023	35
ss	0x0000002b	43
ds	0x0000002b	43
es	0x0000002b	43
fs	0x00000053	83
gs	0x0000002b	43

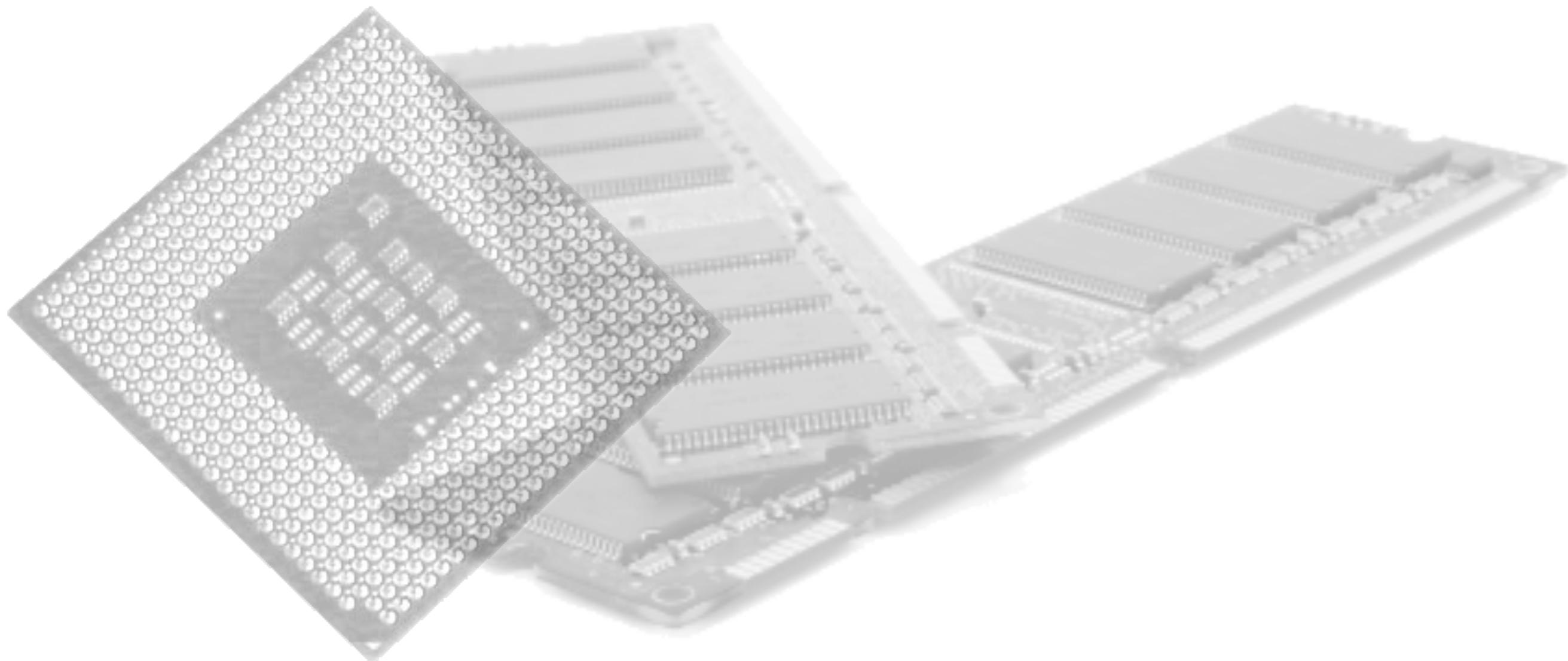


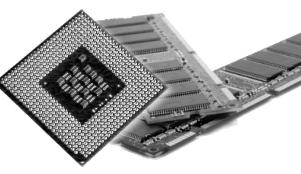
SASM Debugging

- SASM Debugging:
 - mov esp, ebp
- Der Stack Pointer bekommt den gleichen Inhalt wie der Base Pointer
- => Stack wird geleert
- => Debugger sieht keine alten Werte mehr



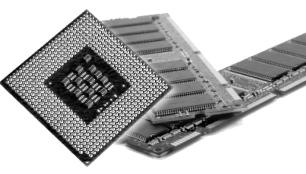
SASM Makros





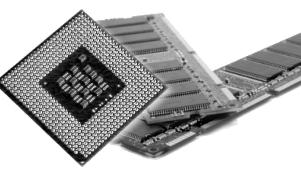
"io.inc" NASM macro library

- PRINT_UDEC / PRINT_DEC
- PRINT_HEX
- PRINT_CHAR
- PRINT_STRING
- NEWLINE
- GET_UDEC / GET_DEC
- GET_HEX
- GET_CHAR
- GET_STRING



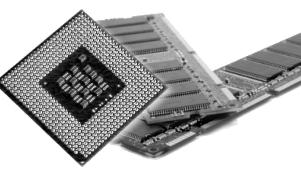
PRINT

- PRINT_HEX <Größe>, <Quelle>
- Größe: In Byte: 1, 2, 4, 8 (nur x64)
- Quelle: Daten zum ausgeben (Die Adresse, d.h. bei Variablen ohne [])
- Gleicher Aufbau bei PRINT_UDEC / PRINT_DEC
 - UDEC - unsigned Decimal



PRINT

- PRINT_CHAR <Quelle>
- Nur die Daten als Quelle: z.B. Charakter oder String
- Gleicher Aufbau bei PRINT_STRING

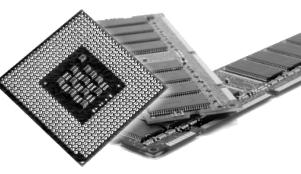


PRINT

- NEWLINE
- Gibt einen Zeilenumbruch aus

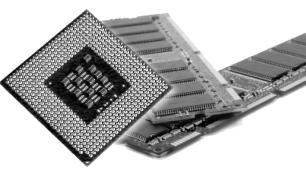
```
1 %include "io.inc"
2
3 section .data
4 Text DB "Text eins", 0
5
6 section .text
7 global CMAIN
8 CMAIN:
9     ;write your code here
10    PRINT_STRING Text
11    NEWLINE
12    PRINT_STRING "Text zwei"
13
14    xor eax, eax
15    ret
```





GET

- GET_UDEC / GET_DEC <Größe>, <Ziel>
- GET_HEX <Größe>, <Ziel>
- GET_CHAR <Ziel>
- GET_STRING <Ziel>, <maximale Größe>
- gelesen wird vom Input Feld
- jede Eingabe in eine neue Zeile

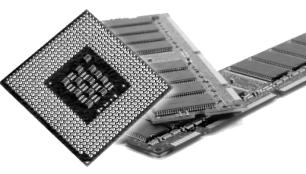


Demo 1

```
1 %include "io.inc"
2
3 section .data
4 Text DB "Text eins", 0
5
6 section .text
7 global CMAIN
8 CMAIN:
9     ; write your code here
10    PRINT_STRING Text
11    NEWLINE
12    PRINT_STRING "Text zwei"
13
14    xor eax, eax
15    ret
```

The screenshot shows a debugger window with two panes. The top pane, labeled 'Input', contains the assembly code from the left. The bottom pane, labeled 'Output', shows the results of the code execution: 'Text eins' followed by 'Text zwei'.

```
Input
Text eins
Text zwei
```

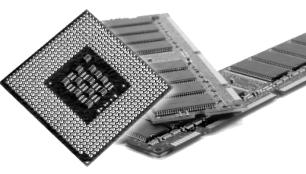


Demo 2

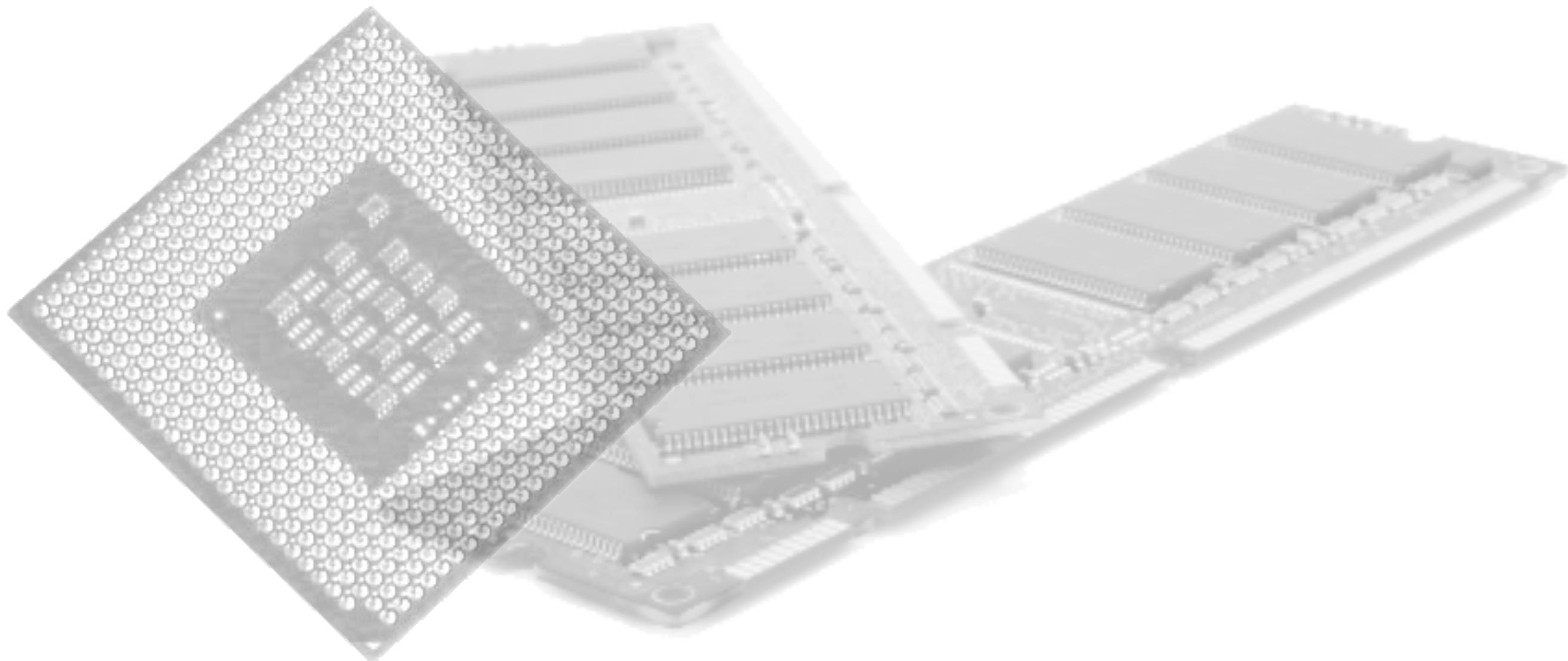
```
1 %include "io.inc"
2 section .data
3 Text DB "Ergebnis: ", 0
4 section .text
5 global CMAIN
6 CMAIN:
7     ; write your code here
8     GET_DEC 4, eax
9     GET_DEC 4, ebx
10
11    add eax, ebx
12
13    PRINT_STRING Text
14    PRINT_DEC 4, eax
15    xor eax, eax
16    ret
```

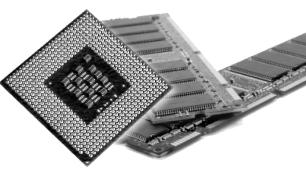
The screenshot shows a debugger interface with several panes:

- Input:** Displays the assembly code being run.
- Registers:** Shows the values of registers EAX and EBX. EAX contains 5 and EBX contains 7.
- Stack:** Shows the memory dump starting with the string "Ergebnis: ".
- Output:** Displays the result of the addition, which is 12.



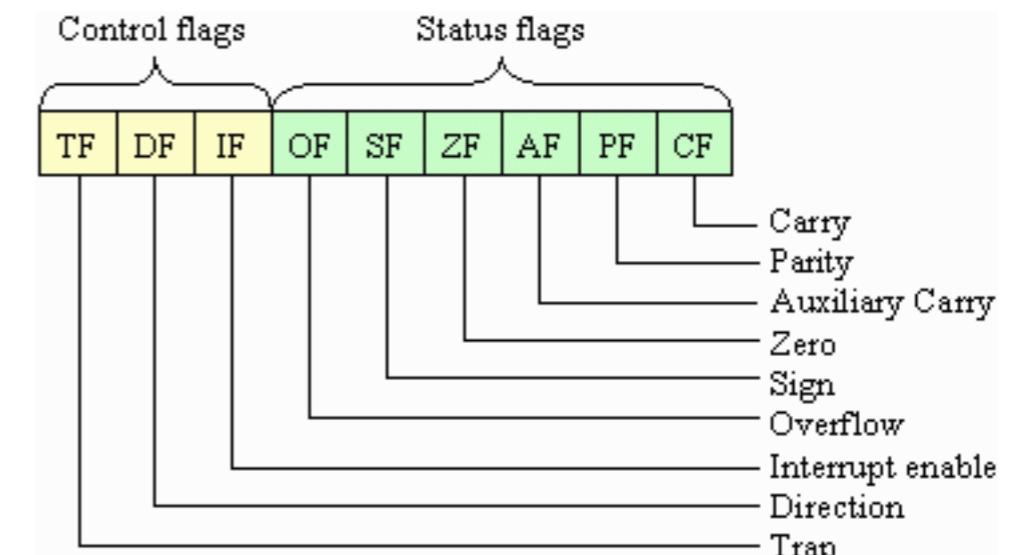
Flags

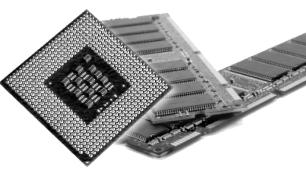




Flags

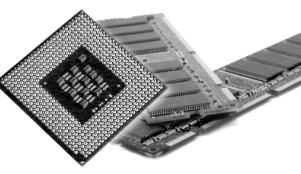
- Control flags:
 - Trap
 - Direction
 - Interrupt enable
- Status flags
 - Overflow
 - Sign
 - Zero
 - Auxiliary Carry
 - Parity
 - Carry





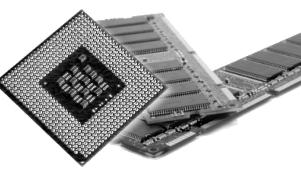
Trap flag

- Versetzt den Prozessor in den Single-Step-Modus
- ermöglicht das Debuggen
- Ein Befehl wird ausgeführt und dann angehalten
- Während der Pause können die Register ausgelesen werden
- Kann nicht direkt gesetzt werden:
 - Flags erst auf den Stack legen
 - Stackelement ändern
 - Flags wieder aus dem Stack lesen



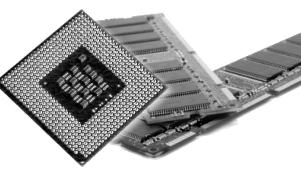
Direction flag

- Kontrolliert die Schreibweise von Strings
 - von links nach rechts
 - von rechts nach links



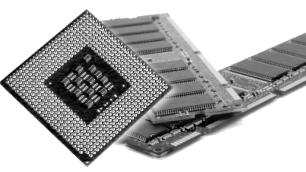
Interrupt flag

- Bestimmt ob Interrupts abgearbeitet werden oder nicht
 - CLI (Clear Interrupts) - keine Interrupts zugelassen
 - STI (Set Interrupts) - Interrupts zugelassen
- betrifft nur maskierbare Unterbrechungen



Overflow flag

- **Arithmetischer** Überlauf
- Bsp.: 8 Bit
 - Dezimal: $127 + 127 = 254$
 - Ergebnis Binär: 1111 1110 => kein Überlauf beim Register
 - ABER: arithmetisch ist die führende Eins das Vorzeichen und damit könnte das Ergebnis auch eine negative Zahl sein (2er Komplement)



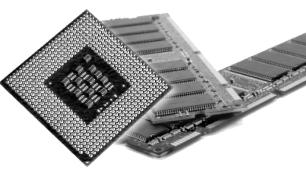
Overflow flag

- AL:
- BL:
- ADD AL, BL

$$\begin{array}{r} 0 \ 1 \ 1 \ 1 \\ + \\ 0 \ 1 \ 1 \ 1 \end{array}$$

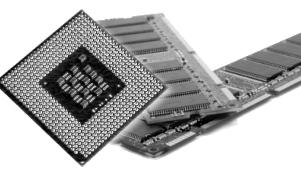
- AL:
 $=$

1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---
- Dezimal: $127 + 127 = 254$
- Ergebnis Binär: $1111 \ 1110 \Rightarrow$ kein Überlauf beim Register
- ABER: arithmetisch ist die führende Eins das Vorzeichen und damit könnte das Ergebnis auch eine negative Zahl sein (2er Komplement)



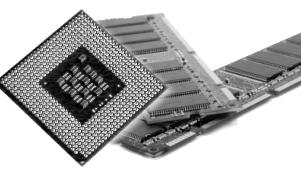
Sign flag

- Wird gesetzt, wenn das Ergebnis einer mathematischen Operation negativ ist
- Wird nicht bei der Multiplikation und Division gesetzt



Zero flag

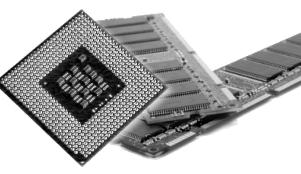
- Wird gesetzt wenn
 - ein arithmetisches Ergebnis 0 ist
 - eine logische Operation 0 wird



Auxiliary Carry flag

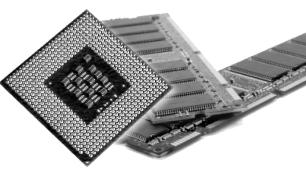
- Wird bei einem Überlauf der ersten 4 Bit gesetzt
- Wird z.B. für die Codierung von BCD verwendet

0	0	0	0	1	1	1	1	+1
0	0	0	1	0	0	0	0	



Parity flag

- Wird gesetzt, wenn das Ergebnis eine gerade Anzahl von 1er besitzt
- Ergebnis: 11010 => ungerade Anzahl => PF: 0
- Ergebnis: 11011 => gerade Anzahl => PF: 1



Carry flag

- Überlauf
- Bsp.:

- mov al, 0FFh
- mov bl, 0FFh
- add al, bl
- Ergebnis passt nicht mehr in al
- => CF = 1

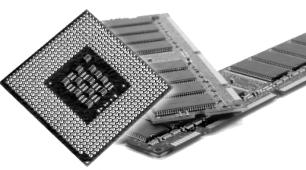
1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---

+

1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---

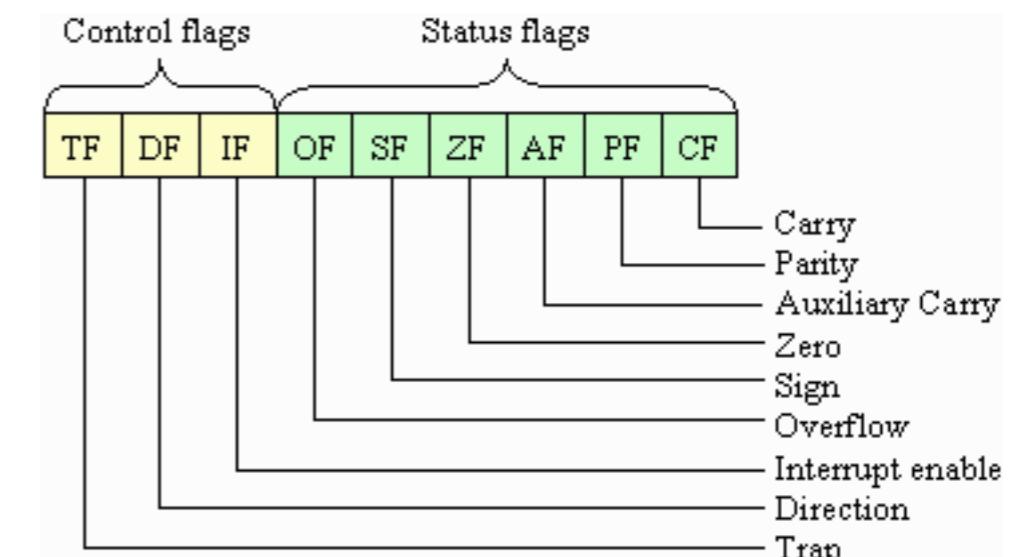
= 1

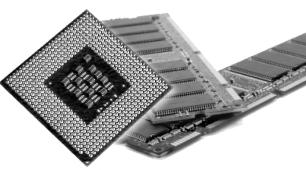
1	1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---	---



Flags

- **PUSHF**
 - schiebt die Flags auf den Stack
- **POPF**
 - holt die Flags vom Stack
- Debugging von Flags:
 - PUSHF
 - POP AX
 - => Flags stehen in AX





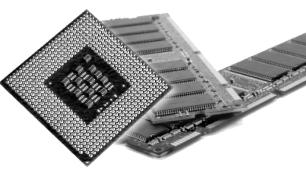
Flag Demo

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6     ;write your code here
7
8     mov eax, 0xFFFFFFFFh
9     mov ebx, 0xFFFFFFFFh ;2h
10
11    sub ebx, eax
12
13    PUSHE
14
15    pop eax
16
17    PRINT_HEX 4, eax
18
19
20    xor eax, eax
21    ret
```

Input

Output

246



Flags SASM

- In SASM sind einige Flags auch direkt rechts über das Register Fenster beim Debugging auslesbar

Registers		
Register	Hex	Info
eax	0x0000003c	60
ecx	0x00401958	4200792
edx	0x0008e3c8	582600
ebx	0x7efde000	2130567168
esp	0x0028ff2c	0x28ff2c
ebp	0x0028ff2c	0x28ff2c
esi	0x00000000	0
edi	0x00000000	0
eip	0x004013b6	0x4013b6 <main+10>
eflags	0x00000283	[CF SF IF]