K02

# Assembler – x86-64 ISA

1. Speicher eines Prozesses, Speicherklassen
2. Register – x86-64
3. Befehle – x86-64
4. Programmierung, Einbettung in C/C++

# Warum Assembler lernen?

- Man bekommt einen Einblick, wie eine CPU arbeitet und wie sie mit Speicher und IO-Geräten interagiert.
- Rückkopplungseffekte: Zu wissen was „ganz unten“ passiert, hilft auch, die Sprachelemente höherer Programmiersprachen effizient nutzen bzw. kombinieren zu können.
- Handoptimierung: Im Speziellen gibt es Situationen, in denen der Mensch effizienteren Maschinencode schreiben kann, als das Compiler tun.
  - ACHTUNG: Nicht überbewerten! Compiler produzieren heutzutage **im Allgemeinen** effizienteren Code als „Standard-Assembler-Entwickler“.
- Reverse-Engineering: z.B. Sicherheitsingenieur, der (a) gegebenes Schadprogramm (Virus o.ä.) oder (b) Programm auf Sicherheitslücken analysieren soll. Da in diesen Fällen oft kein Source-Code verfügbar ist, bleibt nur Analyse des Maschinencodes, z.B. zur Laufzeit mittels Debugger bzw. Disassembler.

# Befehlssatzarchitektur

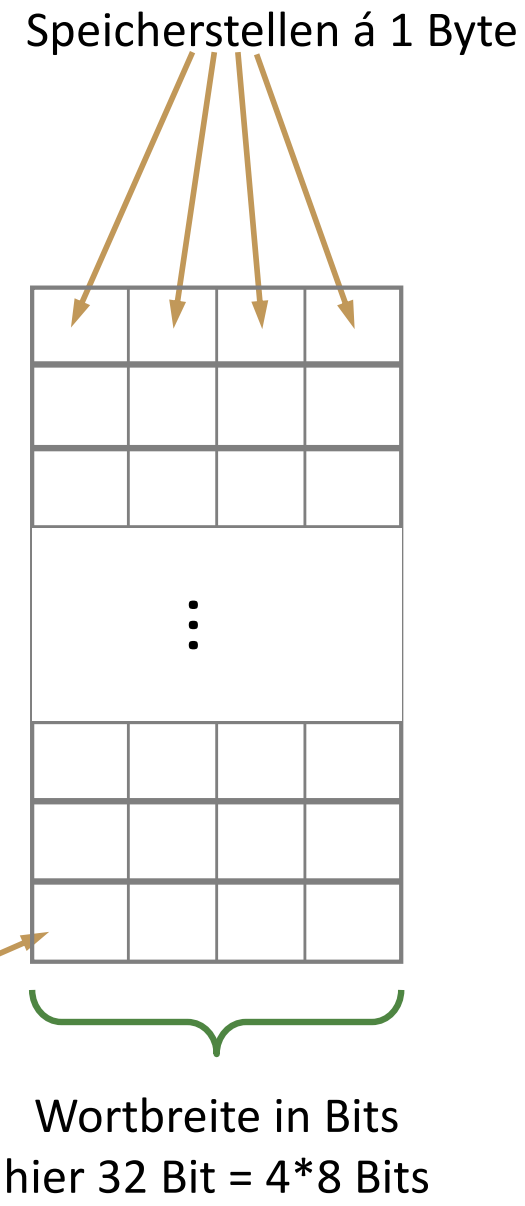
*engl. Instruction Set Architecture (ISA)*

- Allgemein: eine Befehlssatzarchitektur ist die formale Spezifikation des Befehlssatzes und grundlegender Verhaltensweisen eines Prozessors, bestehend vor allem aus:
  - Befehle bzw. Befehlsmenge, die eine CPU ausführen kann
    - binäre Kodierung
    - Verhaltensweise jedes Befehls (Semantik)
  - Interruptverhalten
  - Startadresse der Befehlsverarbeitung
  - Initialisierung der Register nach Reset
  - Unterstützung von Multitasking und Speicherschutzmechanismen

- Wir beschäftigen uns hier mit der x86-64-Architektur (amd64, intel 64)
  - Zählt zu den *Complex Instruction Set Computing*-Architekturen (*CISC)*

# Speichermodell

- Lineare Sequenz von Speicherstellen: meist ein Byte
  - Jede Speicherstelle hat eindeutige Adresse, welche eine positive Zahl ist.
  - Adressierung beginnt meist bei **`0`**; Adresse **`0`** ist “speziell”.
- Ein oder mehrere aufeinander abfolgende Speicherstellen bilden je ein Wort: kleinste Einheit (nicht weiter teilbar) hinsichtlich Transfer zwischen CPU und Speicher.
- Innerhalb einer CPU haben Wörter die gleiche Breite:
  - Je nach CPU 8, 16, 32, 64 Bit breit.




Sei dies die erste Speicherstelle mit Adresse **`0x0001`**, gefolgt von Adressen **`0x0002`**, **`0x0003`**, **`0x0004`**

# Speicheradressen (i)

Heutzutage unterscheiden viele Betriebssysteme zwischen physischem und virtuellem Speicher:

- Prozesse erhalten vom Betriebssystem einen eigenen virtuellen Adressraum (der meist grösser ist als der physische Adressraum).

1. Physische Adresse
   - Adresse, die innerhalb der Hardware benutzt wird.

2. Logische (virtuelle) Adresse
   - Adresse, die ein Programm/Prozess „sieht“.

*Memory Management Unit* (MMU) verwaltet Abbildung zwischen beiden und erfüllt darüber hinaus auch Speicherschutzaufgaben: Sperrung von Speicherbereichen, z.B. Kernel-Space nicht direkt zugänglich vom User-Space.

# Speicheradressen (ii)

Darüber hinaus existieren zwei Adressierungsarten:

1. Absolute Speicheradresse
   - Direkte Angabe der Adresse einer Speicherstelle.

2. Segment-Adresse – z.B. innerhalb der x86-Architektur zu finden
   - Segementselektor eines Speichersegmentes plus Offset, d.h. Addressierung relativ zu Speichersegment.
   - Hatte in der x86-Architektur in der Vergangenheit hohe Bedeutung, um mehr Speicher adressieren zu können, als der Adressbus der CPU breit ist; benötigt aber Hardwareunterstützung.

# Byte-Reihenfolgen *(engl. byte oder, endianess)*

- Das Befüllen von aufeinander abfolgenden Speicherstellen mit den Bytes, die z.B. eine Zahl repräsentieren, kann in unterschiedlicher Reihenfolge geschehen. Dies bezeichnet man als Byte-Reihenfolge.
  - Kommt analog beim Schreiben von Dateien und beim Übertragen von Daten, z.B. in einem Netzwerk, zur Anwendung.

- Es existieren zwei vorherrschende Reihenfolgen*
  - Little-Endian
    - z.B. x86, Z80, 6502
  - Big-Endian
    - z.B. MIPS, SPARC, Motorola-68000

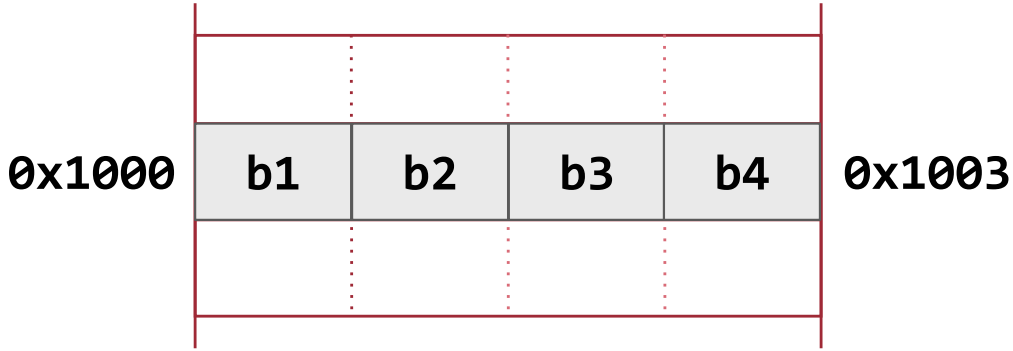
} Beide *Bi-Endian* z.B. ARM ≧ 3, PowerPC, Alpha (teilw. mit Einschränkungen)

*Es existieren weitere exotische Reihenfolgen *Middle-Endian* und *Mixed-Endian*.

# Little-Endian

- Das kleinstwertige Byte wird zuerst gespeichert, dass heisst an der kleinsten Speicheradresse.

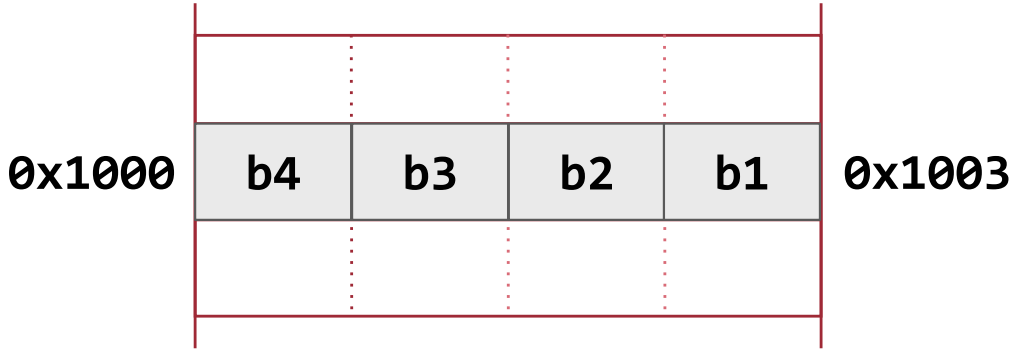
Beispiel: Zahl $\mathbf{23456789}_{dec} = \underbrace{\mathbf{01}}_{\mathbf{b4}}\ \underbrace{\mathbf{65}}_{\mathbf{b3}}\ \underbrace{\mathbf{EC}}_{\mathbf{b2}}\ \underbrace{\mathbf{15}}_{\mathbf{b1}}{}_{hex}$



# Big-Endian

- Das höchstwertige Byte wird zuerst gespeichert, dass heisst an der kleinsten Speicheradresse.

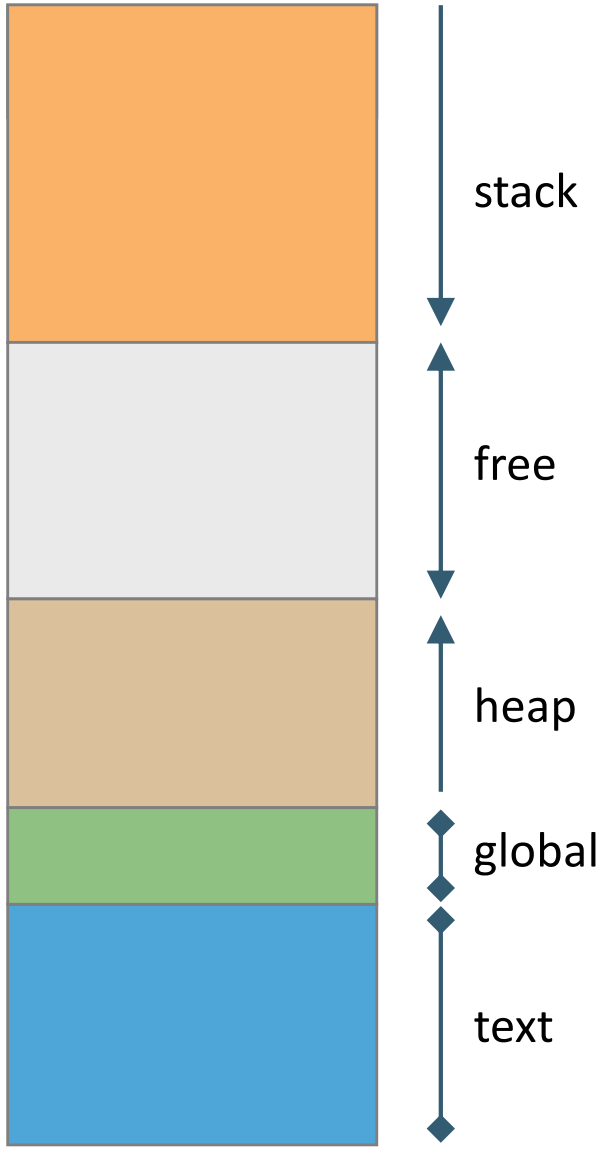
Beispiel: Zahl $\mathbf{23456789}_{dec} = \underbrace{\mathbf{01}}_{\mathbf{b4}}\ \underbrace{\mathbf{65}}_{\mathbf{b3}}\ \underbrace{\mathbf{EC}}_{\mathbf{b2}}\ \underbrace{\mathbf{15}}_{\mathbf{b1}}{}_{hex}$



# Der Speicher eines Prozesses

**Die drei Speicherklassen**

- Es gibt vier wesentliche Speichersegmente für Prozesse (= Programmausführungen)
  - Stack: Parameter, automatische und temporäre Variablen
  - Heap: dynamisch allokierte Variablen
  - Global: statische Variablen
  - Code („text“): das kompilierte Programm
- Heap und Stack wachsen und schrumpfen
- Code- und Global-Segmente sind fest
- Code-Segment ist „read-only“



K02

# Assembler – x86-64 ISA

1. Speicher eines Prozesses, Speicherklassen
2. **Register – x86-64**
3. Befehle – x86-64
4. Programmierung, Einbettung in C/C++

# CPU-Register

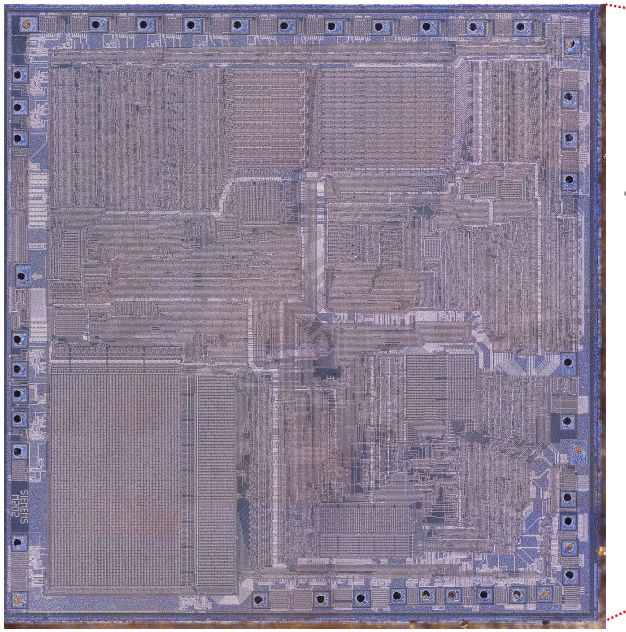
- Speicher mit definierter Breite in Bits, welches direkt mit den eigentlichen Recheneinheiten der CPU verbunden ist.
- Standardbreite entspricht der Wortgrösse.
  - Je nach CPU kann es grössere Register geben, als auch nochmalige Unterteilung und Zugriff auf Teile möglich sein.

Registertypen:

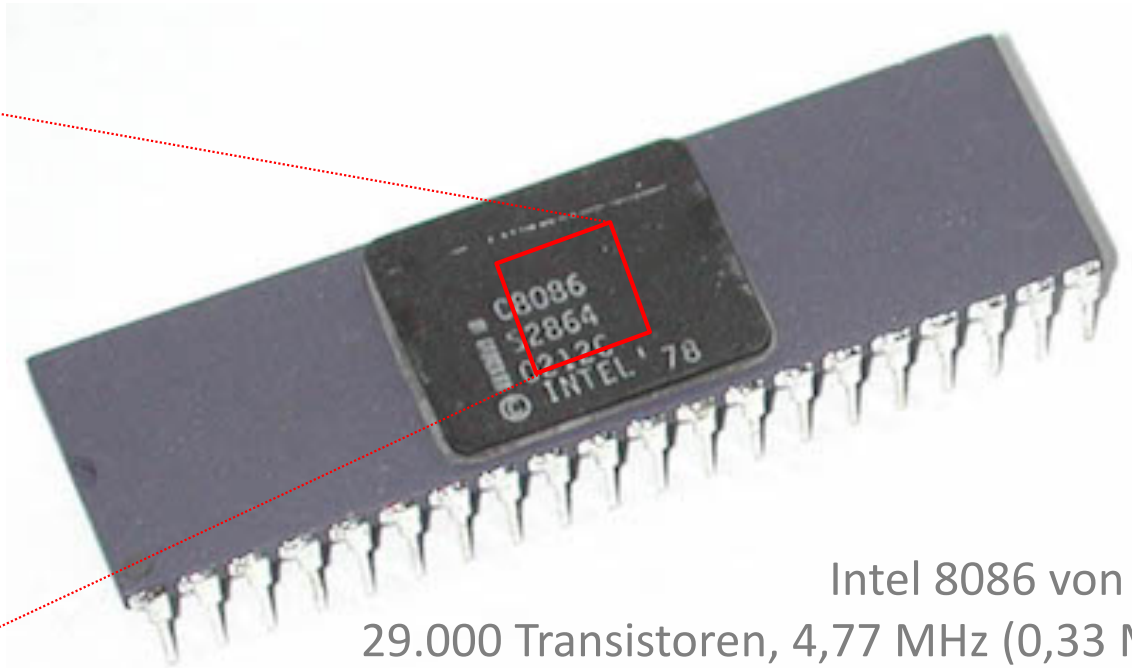
- General Purpose Register (GPR) – Daten oder Adressen
- Special Purpose Register (SPR) – z.B. Instruction/Stack Pointer
- Statusregister – z.B. Vorzeichen, Überlauf, Interrupt u.a.; meist benutzt, um weiteren Befehlsablauf zu bestimmen
- Floating Point Register
- und andere ...

# x86 Register

- Der erste 8086-Prozessor hatte 14 16 Bit-Register.
- Heutige x86-64-Prozessoren haben 64 Bit-Register und zusätzlich zahlreiche Registererweiterungen mit 128 bis zu 512 Bit breiten Registern.



ca. 6mm Kantenlänge

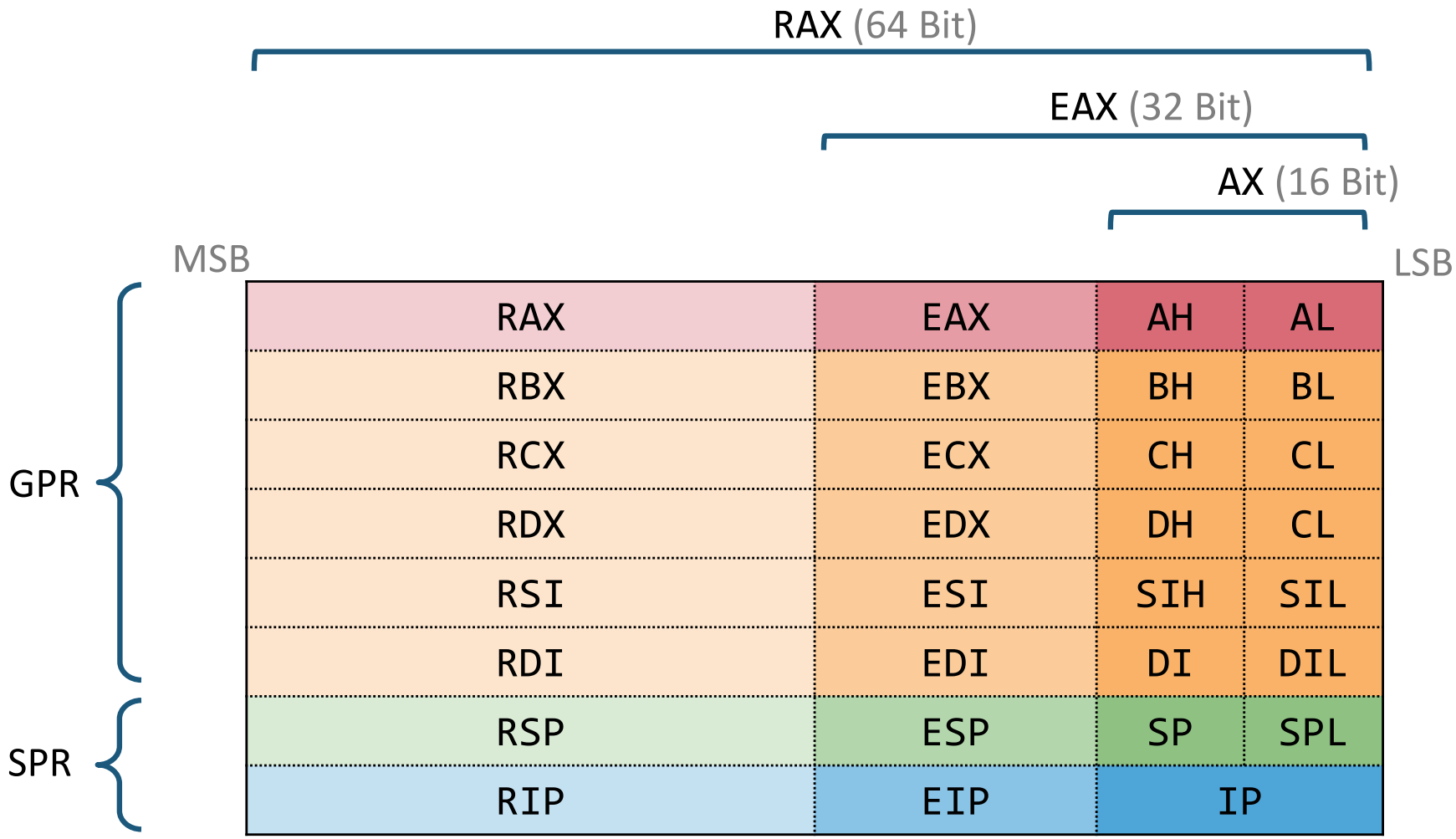



Intel 8086 von 1978
29.000 Transistoren, 4,77 MHz (0,33 MIPS)

*Quelle: https://commons.wikimedia.org/wiki/File:L_intel-c8086.jpg*

# x86 Standardregister (i)

- Die acht Standardregister, die in die Kategorie GPR oder SPR fallen:



# x86 Standardregister (ii)

`AL, AH, AX, EAX, RAX`

- Akkumulator
- Operand für ALU und meist Resultat.

`BL, BH, BX, EBX, RBX`

- Base Register
- Ähnlich Akkumulator: Operand für ALU.

`CL, CH, CX, ECX, RCX`

- Counter
- Speziell für Zählerstände gedacht (Schleifen).

`DL, DH, DX, EDX, RDX`

- Data Register
- Ähnlich Akkumulator/Base Register aber speziell für Daten (anstatt Adressen) gedacht.

# x86 Standardregister (iii)

`SPL, SP, ESP, RSP`

- Stack Pointer
- Hält die aktuelle Adresse des zuletzt abgelegten Elements (also oberstes Element).
- Wird von betreffenden Befehlen implizit dekrementiert bzw. inkrementiert.

`IP, EIP, RIP`

- Instruction Pointer (Befehlszeiger)
- Adresse des nächsten zu verarbeitenden Befehls im Code-Segment.
- Für Programmier nicht direkt zugreifbar.

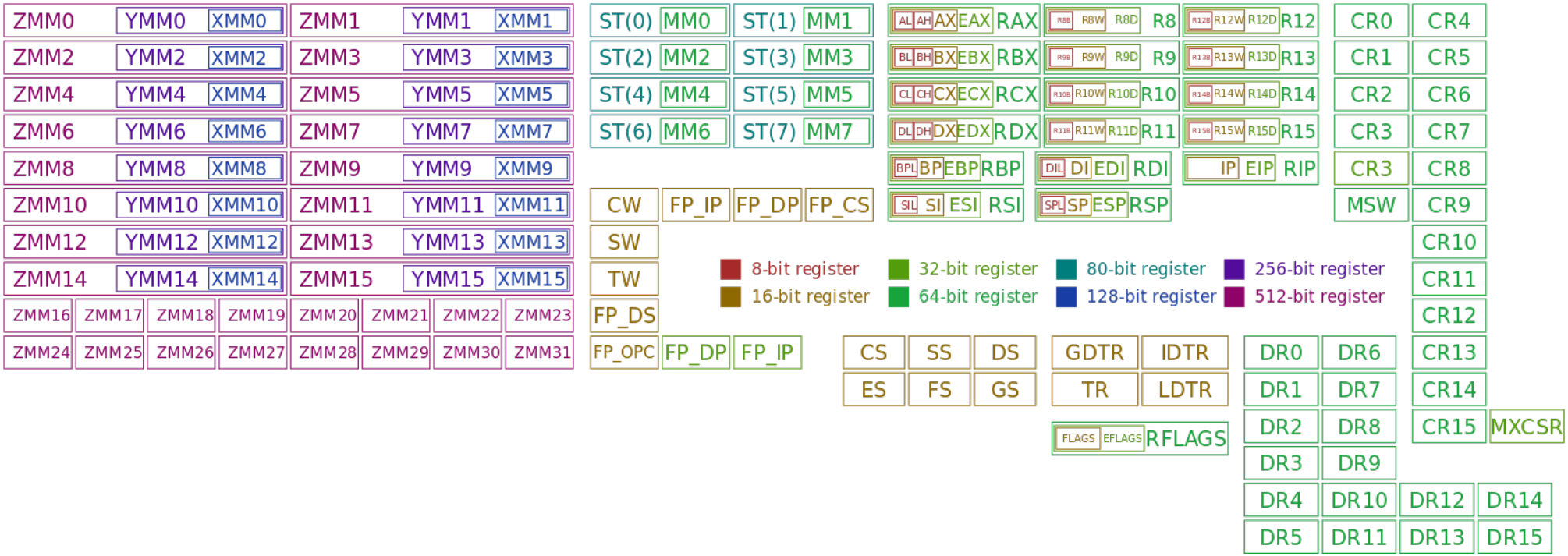
`BPL, BP, EBP, RBP`

- Base Pointer
- Auch Adresse auf Stack; meist zum Festhalten der Grenze zwischen zwei Stack Frames oder der Grenze zwischen lokalen Variablen und Funktionsargumenten verwendet.

`ST0,...,ST7` reserviert für Gleitkommazahlen

und weitere ...

# x86 Register - Übersicht




Quelle: https://commons.wikimedia.org/wiki/File:Table_of_x86_Registers_svg.svg

# Registerinhalt eines Prozesses lesen

- GNU- oder LLVM-Debugger (**`gdb`** bzw. **`lldb`**) können benutzt werden, um aktuellen Inhalt der Register eines Prozesses auszulesen:

```
> gdb -p <pid>
> info registers
```

```
> lldb -p <pid>
> register read
```

```
General Purpose Registers:
       rax = 0xfffffffffffffe00
       rbx = 0x000000000000000a
       rcx = 0x00007f956fabd64a
       rdx = 0x000000000000000a
       rdi = 0xffffffffffffffff
       rsi = 0x00007ffdf82ef6a0
       rbp = 0x0000000000000000
       rsp = 0x00007ffdf82ef688
        r8 = 0x0000000000000000
        r9 = 0x0000000000000000
       r10 = 0x0000000000000000
       r11 = 0x0000000000000246
       r12 = 0x0000000000000000
       r13 = 0x0000000001f4d048
       r14 = 0x0000000000000001
       r15 = 0x0000000001df0988
       rip = 0x00007f956fabd64a
    rflags = 0x0000000000000246
        cs = 0x0000000000000033
        fs = 0x0000000000000000
        gs = 0x0000000000000000
        ss = 0x000000000000002b
        ds = 0x0000000000000000
        es = 0x0000000000000000

(lldb) _
```

# Latency Numbers every Programmer should know

```
L1 cache reference ..........................  0.5 ns
Branch mispredict ............................  5 ns
L2 cache reference ........................... 7 ns
Mutex lock/unlock ........................... 25 ns
Main memory reference ...................... 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns = 3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns = 20 µs
SSD random read ........................ 150,000 ns = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns = 250 µs
Round trip within same datacenter ...... 500,000 ns = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns = 1 ms
Disk seek ........................... 10,000,000 ns = 10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns = 20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns = 150 ms
```

*Assuming ~1GB/sec SSD

Quelle: https://gist.github.com/hellerbarde/2843375

# Latency Numbers every Programmer should know

Lets multiply all these durations by a billion:

| | | | |
|---|---|---|---|
| L1 cache reference | 0.5 | s | One heart beat |
| Branch mispredict | 5 | s | Yawn |
| L2 cache reference | 7 | s | Long yawn |
| Mutex lock/unlock | 25 | s | Making a coffee |
| Main memory reference | 100 | s | Brushing your teeth |
| Compress 1K bytes with Zippy | 50 | m | One episode of a TV show |
| Send 2K bytes over 1 Gbps network | 5.5 | h | From lunch to end of work day |
| SSD random read | 1.7 | d | A normal weekend |
| Read 1 MB sequentially from memory | 2.9 | d | A long weekend |
| Round trip within same datacenter | 5.8 | d | A medium vacation |
| Read 1 MB sequentially from SSD* | 11.6 | d | Waiting ~2 weeks for a delivery |
| Disk seek | 16.5 | weeks | A semester in university |
| Read 1 MB sequentially from disk | 7.8 | months | Almost producing a human being |
| Send packet CA->Netherlands->CA | 4.8 | years | Average time it takes to complete a bachelor's degree |

Quelle: https://gist.github.com/hellerbarde/2843375

K02

# Assembler – x86-64 ISA

1. Speicher eines Prozesses, Speicherklassen
2. Register – x86-64
3. **Befehle – x86-64**
4. Programmierung, Einbettung in C/C++

# Befehle – Syntaxvarianten

- Es gibt (leider) zwei verschiedene x86-Assembler-Dialekte:

1. AT&T-Syntax

Transfer- bzw. Leserichtung

```
   movq   %rsp,    %rbp
; instr source, dest
```

2. Intel-Syntax

Transfer- bzw. Leserichtung

```
   mov    rbp,  rsp
; instr dest, source
```

# Befehlsoperanden (i)

- Sind entweder Daten oder Adressen, auf die ein Befehl angewendet wird.
- x86-Befehle haben null bis maximal drei Operanden.
- Ein Operand kann entweder *immediate*, *register*, oder *memory* sein.
  - Immediate: eine Konstante bzw. Literal (auch *inline value* genannt).
  - Register: Wert in einem Register.
  - Memory: Wert, der an einer Speicheradresse gespeichert ist.

# Befehlsoperanden (ii)

Beispiele:

1. AT&T-Syntax

```
movq   %rsp,    %rbp
movl   $0xff,   %ebx
movl   (%ebx), %eax
```

2. Intel-Syntax

```
mov     rbp, rsp
mov     ebx, 0ffh
mov     eax, dword prt [ebx]
```

```
$  = immediate
%  = register
() = memory

movb       byte = 1 byte
movw       word = 2 byte
movl       long = 4 byte
movq       quad = 8 byte
```

```
immediate, register auto-
matisch detektiert
[] = memory

Size based on register
identifier or specifier:
byte       byte = 1 byte
word       word = 2 byte
dword      long = 4 byte
qword      quad = 8 byte
```

# Befehlsarten

Es existieren drei wesentliche Befehlsarten:

1. Befehle zum Datentransfer.
2. Arithmetische und logische Verknüpfungen/Operationen.
3. Kontrollflusssteuerung bzw. (bedingte) Sprünge.

Nachfolgend ein Überblick der wichtigsten Befehle – repräsentativ aber nicht umfassend.

# Datentransfer (i)

**mov** – Move

Kopiert Daten von Operand zu Operand.
Achtung! Transfer memory-to-memory mit **mov** nicht direkt möglich; nur mittels „Umweg“ über Register.*

```
; Syntax
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
```

```
; Kopiere Wert in ebx nach eax.
mov eax, ebx

; Speichere Wert 5 in das Byte der
; Speicherstelle, die in eax ist.
mov byte ptr [eax], 5
```

*Es existieren Ausnahmen und CPUs/ISAs in denen das möglich ist.

# Datentransfer (ii)

**push** – Push stack

Legt Operand oben auf Stack ab, nachdem zuerst der Stack Pointer (SP) dekrementiert wurde; z.B. um 4 bei Wortbreite von 32 Bit; bzw. 8 bei 64 Bit Wortbreite. Arbeitet immer in der Wortbreite der CPU.

```
; Syntax
push <reg>
push <mem>
push <const>
```

```
; Lege Wert in rax auf Stack.
push rax

; Lege Wert der Speicherstelle, die
; in rax ist auf Stack.
push [rax]
```

# Datentransfer (iii)

**pop** – Pop stack

Entfernt oberstes Element vom Stack (4 oder 8 Byte je nach Wortbreite) und kopiert Wert in Operand. Danach wird Stack Pointer (SP) entsprechend inkrementiert.

```
; Syntax
pop <reg>
pop <mem>
```

```
; Lege Wert auf Stack in rax.
pop rax

; Lege Wert auf Stack in
; Speicherstelle, die in rax ist.
push [rax]
```

# Datentransfer (iv)

**lea** – Load effective address

Legt Adresse des zweiten Operand in Register welches durch ersten Operand angegeben ist. Dies geschieht ohne das der Speicherinhalt geladen wird.

```
; Syntax
lea <reg>, <mem>
```

```
; Adresse die sich aus dem Ausdruck
; ebx+4*esi ergibt, wird in edi
; gelegt.
lea edi, [ebx+4*esi]
```

# Arithmetische Operationen

**add** – Integer Addition; Ergebnis im ersten Operand.

**sub** – Integer Subtraktion; Ergebnis im ersten Operand.

**inc, dec** – Integer Inkrement, Dekrement des Operanden.

**imul** – Integer Multiplikation; Ergebnis in ersten Operand.

```
; Syntax
imul <reg>,<reg>[,<const>] ; 2. * 3. Operand = 1. Operand
imul <reg>,<mem>[,<const>]
```

**idiv** – Integer Division; Quotient,Rest in rdx,rax bzw. edx,eax.

```
; Syntax
idiv <reg> ; rdx:rax / <reg> = rdx:rax; Quotient:Rest
idiv <mem> ; rdx:rax / <mem> = rdx:rax; Quotient:Rest
```

# Logische Operationen

**and** – Bitweise Konjunktion; Ergebnis im ersten Operand.

**or, xor** – Bitweise Disjunktion, Kontravalenz; Ergebnis im ersten Operand.

**not** – Bitweise Negation.

**neg** – Zweierkomplement (zur Darstellung vorzeichenbehafteter Integer).

**shl, shr** – Bitweise links, rechts schieben.

```
; Syntax
shl <reg>,<const>
shl <mem>,<const>
shr <reg>,<const>
shr <mem>,<const>
```

```
; Beispiele
shl eax, 1 ; Multipliziere Wert in eax mit 2.
shr ebx, cl; Dividiere ebx / 2^n, wobei n Wert in cl
```

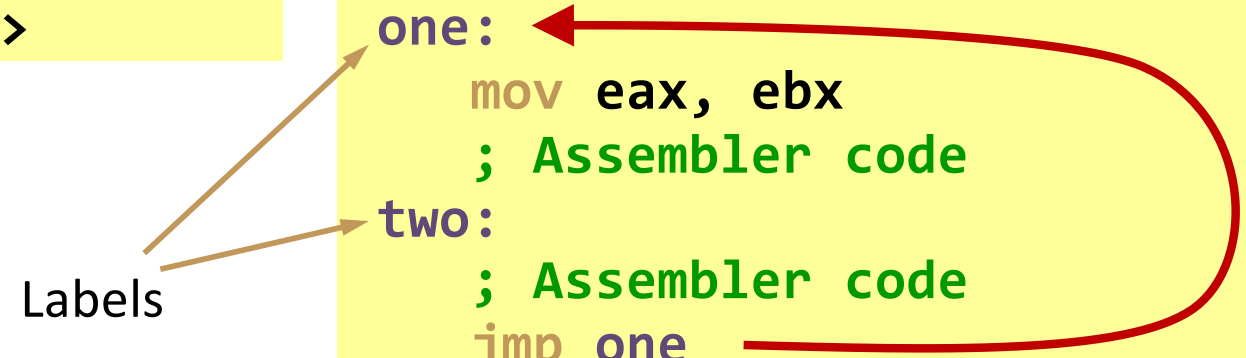
# Sprungoperationen (i)

**jmp** – Jump

Lade Instruction Pointer mit Adresse des Befehls, der unter genanntem Label zu finden ist. Dadurch Sprung zum Befehl mit Label.

```
; Syntax
jmp <label>
```

```
; Beispiel
one:
    mov eax, ebx
    ; Assembler code
two:
    ; Assembler code
    jmp one
```



# Sprungoperationen (ii)

**`jcondition`** – Conditional Jump

Basierend auf Wert im Spezialregister *Machine Status Word* (MSW), wenn Test true liefert, dann Lade Instruction Pointer mit Adresse des Befehls, der unter genanntem Label zu finden ist. Dadurch Sprung zum Befehl mit Label. Andernfalls, setze mit nächstem Befehl fort.

```
; Syntax
je  <label>; jump when equal
jne <label>; jump when not equal
jz  <label>; jump when result zero
jg  <label>; jump when greather than
jge <label>; jump when gr or equal
jl  <label>; jump when less than
jle <label>; jump when less or equal
```

```
; Beispiel: Springe
; zu one wenn
; eax <= ebx
one:
    mov eax, ebx
two:
    cmp eax, ebx
    jle one
```

# Sprungoperationen (iii)

**`call, ret`** – Subroutine call, return

**`call`** legt Adresse des nächsten Befehls auf Stack und springt dann zur Adresse des Operand (indem `EIP` entsprechend gesetzt wird).

**`ret`** Lädt zuerst Sprungadresse vom Stack (pop) in Instruction Pointer, löscht optional Anzahl Bytes des Operanden vom Stack (durch Inkrementieren von ESP) und fährt dann mit nächsten Befehl fort (im `EIP`).

```
; Syntax
call <label>
ret  [<reg>]; Operand optional
ret  [<const>]; Operand optional
```

# Weitere Befehle zur Kontrollflusssteuerung

**`loop`** – Jump until ECX/RCX is zero

Dekrementiert ECX/RCX und springt zur Adresse des Operanden, solange ECX/RCX nicht null.

**`enter`** – Create Stack frame with specified amount of space

**`leave`** – Destroy current Stack frame and restore previous frame

**`hlt`** – Halt processor. Resume execution on next interrupt.

**`nop`** – No operation. Does not do anything except waiting an instruction cycle.

K02

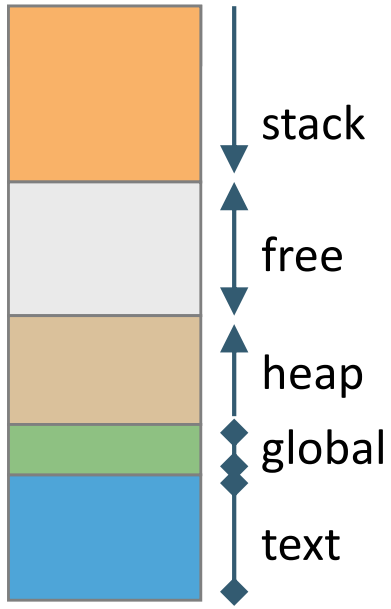
# Assembler – x86-64 ISA

1. Speicher eines Prozesses, Speicherklassen
2. Register – x86-64
3. Befehle – x86-64
4. **Programmierung, Einbettung in C/C++**

# Assembler-Programmstruktur

- Assembler-Programme können in drei Bereiche aufgeteilt sein, was eng mit den Speicherklassen eines Prozesses zusammenhängt:

1. Die `data`-Sektion
   - Deklaration von Konstanten bzw. initialisierten Daten
2. Die `bss`-Sektion – nicht immer notwendig
   - Deklaration von mit 0 oder uninitialisierten Variablen/Speicherbereichen
3. Die `text`-Sektion
   - Programmcode, also Befehle
   - Benötigt zusätzlich Deklaration eines Einstiegspunktes, worüber dem Betriebssystem gesagt wird, wo mit der Programmausführung zu beginnen ist.



# Assembler-Direktiven

- Kommandos, die nicht zum Befehlssatz des Prozessors gehören, aber vom Übersetzer – dem Assembler – ausgewertet werden.
- **`data`**-, **`bss`**-, und **`text`**-Sektion werden mittels solcher Direktiven abgegrenzt.
- Es existieren zahlreiche weitere Direktiven.

# Hello World in Assembler

Für 32bit Linux-Plattform im ELF-Format.

hello.asm

```
section .text
   global _start    ;must be declared for linker (ld)
_start:             ;tells linker entry point
   mov edx,len      ;message length
   mov ecx,msg      ;message to write
   mov ebx,1        ;file descriptor (stdout)
   mov eax,4        ;system call number (sys_write)
   int 0x80         ;call kernel interrupt
   mov eax,1        ;system call number (sys_exit)
   int 0x80         ;call kernel

section .data
   msg db 'Hello, world!', 0xa ;string to be printed
   len equ $ - msg             ;length of the string
```

```
> nasm -f elf hello.asm
> ld -m elf_i386 -s -o hello hello.o
> ./hello
hello world!
```

**nasm** ist der Assembler, der Objekt-code erzeugt.
**ld** ist der Linker, der das ausführbare Programm erzeugt.

# Hello World in Assembler

Für Mac OS X-Plattform im Mach-O-Format.

hello.asm

```
.section __DATA,__data
str:
  .asciz "Hello world!\n"
.section __TEXT,__text
.globl _main
_main:
  movl $0x2000004, %eax           # preparing system call 4
  movl $1, %edi                   # STDOUT file descriptor is 1
  movq str@GOTPCREL(%rip), %rsi   # The value to print
  movq $100, %rdx                 # the size of the value to print
  syscall
  movl $0, %ebx
  movl $0x2000001, %eax           # exit 0
  syscall
```

```
> as hello.asm -o hello.o
> ld hello.o -e _start -o hello
> ./hello
hello world!
```

**as** ist der Assembler, der Objekt-code erzeugt.
**ld** ist der Linker, der das ausführbare Programm erzeugt.

# Aufrufkonvention *engl. calling convention*

- Mit der *Prozeduralen Programmierung* wurde das Konzept der *Prozedur* erschaffen: ein Unterprogramm welches man, modulo Sichtbarkeiten, von beliebigen Stellen aus aufrufen kann.
  - Prozeduren können verschachtelt sein. Insbesondere sollen sie sich auch selbst aufrufen können (um Rekursion zu ermöglichen).
  - Subroutinen, Funktionen, Methoden sind letztlich alle eine Form einer Prozedur.
- Um Prozeduraufrufe in einem linearen Speicher zu realisieren, wird praktisch immer ein Stack als elementare Datenstruktur benutzt.
- Dabei muss eine Konvention definiert werden, die beschreibt:
  1. Wie Parameter (mittels Stack) an eine Prozedur weitergeleitet werden.
  2. Wie Ergebnisse an das übergeordnete Programm zurück gegeben werden.
  3. Wie lokale Daten einer Prozedur verwaltet werden, so dass sie mit dem Ende der Prozedur automatisch wieder gelöscht werden.
  4. Wie man am Ende einer Prozedur zur der Stelle im Hauptprogramm zurück kehren kann, an der man fortsetzen möchte.

# Aufrufkonventionen (ii)

- Es gibt zahlreiche unterschiedliche Aufrufkonventionen.
- Historisch betrachtet wurden diese im Rahmen der unterschiedlichen CPU-Plattformen, Compiler und/oder Betriebssysteme definiert; also **nicht** vorrangig auf der Ebene einer Programmiersprache, sondern dessen Implementierung.

- Innerhalb der x86-Platform existieren drei häufig verwendete Konventionen:
  - **cdecl** – wird von vielen C- und C++-Compilern verwendet.
  - **stdcall** – de facto-Standard für Win32-API.
  - **fastcall** – versucht die ersten beiden Parameter über Register ECX und EDX an die Prozedur zu übergeben (und ist damit in solchen Fällen schneller als Konventionen, die ausschliesslich den Stack zur Parameterübergabe benutzen).

# **cdecl**-Aufrufkonvention im Detail

*caller* – aufrufende Prozedur

*callee* – aufgerufene Prozedur

1. Parameter werden vom caller auf Stack gelegt (**`push`** oder **`mov`** mit RBP als Basisadresse und Offset); vor Aufruf der Prozedur, von rechts nach links.
2. Callee legt Integer- und Adressen als Rückgabewert in EAX ab; Gleitkommazahlen in `ST0`.
3. EAX, ECX und EDX stehen callee zur Verfügung. Alle anderen Register müssen vor Verwendung vom callee gerettet werden (z.B. indem er sie auf den Stack legt) und vor Rücksprung vom callee wiederhergestellt werden.
4. Callee darf Stack selbst vergrössern (**`push`**); muss aber vor Rücksprung alle selbst abgelegten Elemente wieder entfernen (**`pop`** oder Inkrementieren von ESP).
5. Caller baut nach Rücksprung Stack, den er unter 1. erzeugt hat selbst wieder ab (durch Inkrementieren von ESP).

# Einfache Funktion in Assembler

C++

```
int add(int n) {
   return n + 42;
}
```

Assembler – Direktiven weggelassen

```
add(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        add     eax, 42
        leave
        ret
```

Stack-Frame erzeugen

Stack-Frame abbauen

# Einbettung Assembler in C++

**Gegeben sei folgende C++-Funktion:**

```
// Addiere 100000 mal Array y zu Array x
// Annahme: Arrays haben gleiche Länge.
#define TIMES 100000
void calc(int *x, int *y, int length) {
  for (int i = 0; i < TIMES; i++) {
    for (int j = 0; j < length; j++) {
      x[j] += y[j];
    }
  }
}
```

Gesucht ist eine Implementierung in Assembler, die in eine C++-Funktion eingebettet ist.

# Einbettung Assembler in C++

**Inline Assembler in C++-Funktion:**

Einbettung mittels Schlüsselwort **__asm__** oder **asm** je nach Compiler.

```
void calc(int *x, int *y, int length) {
  __asm__ {
      mov edi,TIMES
    start:
      mov esi,0
      mov ecx,length
    label:
      mov edx,x
      push edx
      mov eax,DWORD PTR [edx + esi*4]
      mov edx,y
      mov ebx,DWORD PTR [edx + esi*4]
      add eax,ebx
      pop edx
      mov [edx + esi*4],eax
      inc esi
      loop label ; decrement ecx and jump unless 0
      dec edi
      cmp edi,0
      jnz start
  };
}
```

**ACHTUNG!**
Diese Implementierung ist deutlich langsamer als von Compilern erzeugter Maschinencode. Warum?

Quelle http://stackoverflow.com/q/9601427