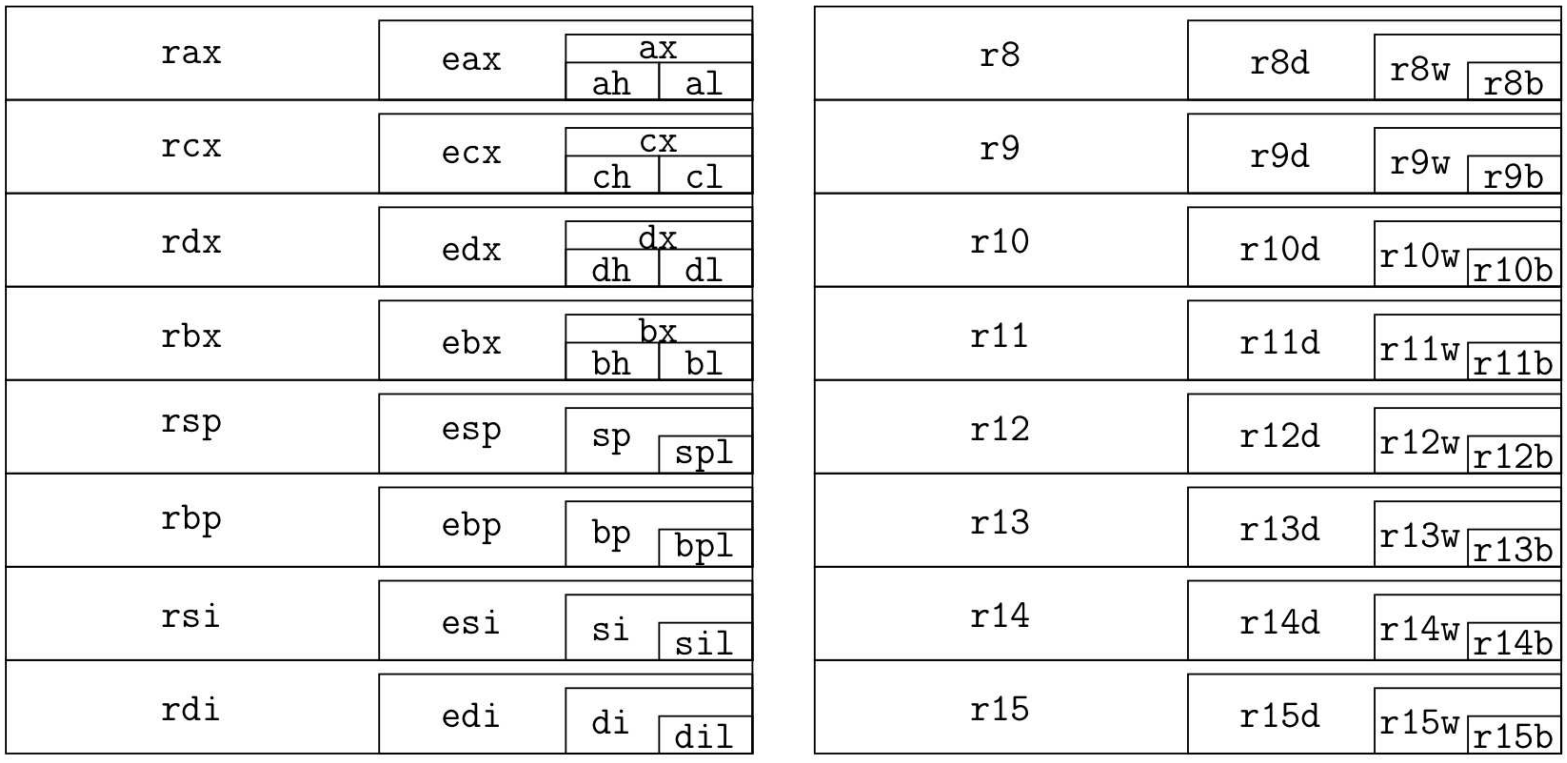
x86-64

64 Bit Register



Wichtige Instruktionen im Überblick

Mnemonic	Beispiel	Pseudocode
MOV dst, src	`mov rax, rdx`	`rax = rdx`
MOV dst, imm	`mov rax, 2`	`rax = 2`
ADD dst, src	`add rax, rdx`	`rax = rax + rdx`
ADD dst, imm	`add rax, 2`	`rax = rax + 2`
SUB dst, src	`sub rax, rdx`	`rax = rax - rdx`
SUB dst, imm	`sub rax, 2`	`rax = rax - 2`
MUL src	`mul rcx`	`rdx:rax = rax · rcx` (unsigned)
IMUL src	`imul rcx`	`rdx:rax = rax · rcx` (signed)
IMUL dst, src	`imul rax, rdx`	`rax = rax * rdx`
IMUL dst, src, imm	`imul rax, rdx, 2`	`rax = rdx * 2`
CMP src, src	`cmp rax, rdx`	Vergleicht `rax` mit `rdx`
JMP label	`jmp .Lskip`	Springt zum Label `.Lskip`
Jcc label	`je .Lskip`	Bedingter Sprung zu `.Lskip`
AND dst, src	`and rax, rdx`	`rax = rax ∧ rdx`
OR dst, src	`or rax, rdx`	`rax = rax ∨ rdx`
XOR dst, src	`xor rax, rdx`	`rax = rax ⊕ rdx`
NOT dst	`not rax`	`rax = ¬rax`
IDIV src	`idiv rcx`	`rax = rdx:rax / rcx` (signed) `rdx = rdx:rax % rcx`
DIV src	`div rcx`	`rax = rdx:rax / rcx` (unsigned) `rdx = rdx:rax % rcx`

Befehl

`add rbx, rdx;  adc rax, rcx` : addiert zwei 128-Bit Zahlen in (`rax: rbx`) und (`rcx: rdx`) und speichert das Ergebnis in (`rax: rbx`)

Der Befehl `adc` ("add carry") führt eine normale Addition durch und addiert zum Ergebnis den Wert des Carry-Flags zuvor

`cmovl edi, r9d` : `cmovl` ist eine Variante der "conditional move" Instruktion `cmovCC` . Wenn die Statusflags l (less-than) anzeigen (z.B. nach einem `cmp`), dann wird der Wert von `r9d` nach `edi` geschrieben. Achtung: Die oberen 32 Bit von `edi` werden hier immer auf 0 gesetzt

SF ≠ OF

`popcnt rax, rdi` : Zählt die Anzahl der gesetzten Bits in `rdi` und schreibt das Ergebnis in `rax` . Ist die Instruktion auf einer CPU nicht verfügbar, so wird beim Versuch diese dennoch auszuführen, aber von der CPU eine "Invalid Opcode" Exception ausgelöst.

`movsx rdx, eax` : Erweitert einen 32-bit Wert (Quellregister kleiner: hier `eax`) vorzeichenbehaftet auf 64-bit (Zielregister größer: hier `rdx`)

`cqo` unterstützt keine(n) Operanden, es wird hier immer `rax` (64 bit) auf `rdx: rax` (128 bit) sign extended

mit 64 Immediates

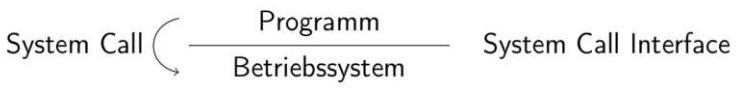
64 Bit Immediates nur bei `mov`-Instruktionen erlaubt

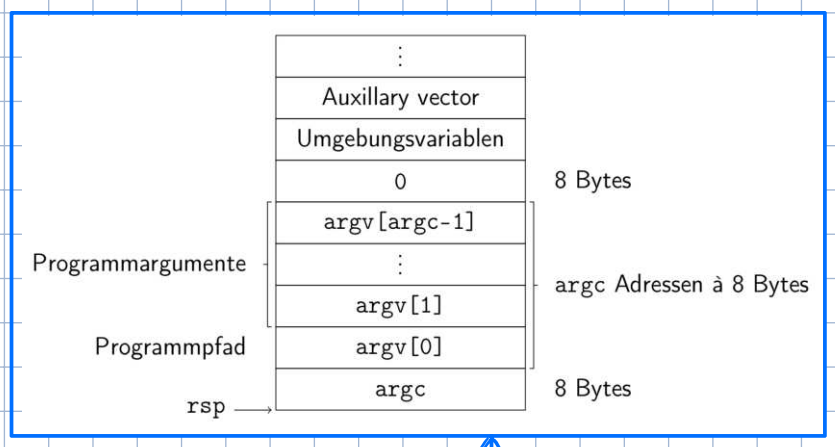
```
mov rax, 0xaaaabbbbccccdddd     erlaubt
add rax, 0xaaaabbbbccccdddd     nicht erlaubt
```

ABER sub rcx, 0xffffffffc0000000 erlaubt

System Calls

- Zugriff geregelt über Anfragen an das Betriebssystem – *System Calls*
  - Interface zwischen Programm und Betriebssystem
  - System Call ≙ Funktionsaufruf
  - Werden eindeutig über System Call Numbers identifiziert





Mit der Instruktion `syscall` wird ein Systemaufruf ausgeführt

Um den bestimmten Systemaufruf auszuführren, wird die Nummer dieses Systemaufrufs in Register `rax` geschrieben

In Register `rax` steht das Ergebnis des Systemaufrufs

Fehler werden dadurch angezeigt, dass in dem Rückgaberegister ein negativer Wert im Bereich von -1 bis -4095 steht

System Call Argumente: stehen in Rehenfolge `rdi, rsi, rdx, r10, r8, r9` . Ein Systemaufruf nimmt nicht mehr als sechs Argumente

Register `rax` (System Call Ergebnis), `rcx` (rip) und `r11` (rflags) können durch System Call verändert werden. Alle andere verändern nicht

Adressen an welche Segmente geladen werden sollen -> Program-Header

Startadresse der Programmausführung (_start) -> Datei-Header

Start eines Programms mittels System Call `execve` - kein Rückgabewert bei erfolgreicher Ausführung

Konsole Output

System Call `write` erwartet 3 Parameter `ssize_t write(int fd, const void *buf, size_t count);`

System Call `stdout` erwartet 3 Parameter `stdin, stdout, stderr - standard I/O streams`

Beenden eines Programms -> nicht einfach mit `ret` , sondern System Call `_exit (Wert zwischen 0-255)` , kein Rückgabewert

Vor dem Kompilieren: Preprocessing



GDB-Debug im Terminal

Schritt 1: Öffnen Sie das Terminal

Schritt 2: Verbinden Sie sich mit Rechnerahlle per `ssh`

username: shq@lxhalle.in.tum.de

Passwort: rHG4YmC8xk

Schritt 3: Mit `wget Link` runterladen

z.B. `wget https://gra.caps.in.tum.de/m/gcd.tar`

Schritt 4: Dateien aus Tar-Archiv extrahieren z.B. `tar xf gcd.tar`

Schritt 5: in neu angelegten Ordner gcd zu wechseln z.B. `cd gcd`

Schritt 6: `ls` ausführen und Dateien finden

Schritt 7: `make` ausführen um es zu kompilieren

Schritt 8: `gdb gcd` um zu debuggen

- `r` run
- `p` print
- `s` step
- `f` frame
- `n` next
- `l` list
- `b` breakpoint
- `bt` backtrace
- `c` continue
- `q` quit
- `info b`
- `info locals`

```
info gdb                    //Manual
info locals                     //Vars in local scope
info variables              //Vars declared outside current scope
info functions              //Names and datatypes of all defined functions
info b                          //List all breakpoints
break funcName                  //Set breakpoint at function funcName (short: b funcName)
break file::line            //Set breakpoint at line in file
layout next                     //Cycle through the layouts of gdb
p var                           //Print the value of variable var
p var = value               //Force set value to a var
run                             //Start the program
start                           //Synonymous to (b main && run). Puts temporary b at main
next                        //Execute the current line in source (short: n)
step                            //Step into function call at current line (short: s)
finish                          //Finish the execution of current function (short: fin)
continue                    //Resume execution (After a breakpoint) (short: c)
refresh                         //Repaint the interface (To fix corrupted interface)
shell cmd                       //Run shell command cmd from gdb prompt
python gdb.execute(cmd)         //Run a gdb command cmd from python prompt
set print pretty on         //Enable pretty printing
                                  (Put in ~/.gdbinit)
$ gdb -c core.num           //Examine the dumped core file from a SIGSEGV(shell command)
bt                              //Print backtrace
break _exit                     //Breakpoint at exit of program
whatis expr                     //Print datatype of expr
ptype expr                      //Detailed print of datatype of expr
watch var                       //Stop when var is modified
watch -l foo                    //Watch foo loaction
rwatch foo                      //Stop when foo is read
watch foo if foo>10         //Watch foo conditionally
delete                          //Delete all breakpoints
delete breakpoint_no            //Delete breakpoint breakpoint_no
command breakpoint_no           //Run user listed commands when breakpoint is hit
                                  (End list of commands with 'end')
file executable                 //Load the executable for debugging from inside gdb
quit                            //Quit (short: q)
```

Programmiersprache C

Datentypen

Datentypen standardmäßig vorzeichenbehaftet - außer `char` und `_Bool`

Overflows nur für vorzeichenlose Zahlen definiert, für vorzeichenbehaftete Zahlen undefiniert

`_Bool` b = 42 = 1 (für `_Bool`: ein Wert ungleich 0 -> automatisch 1)

Datentyp	Wertebereich	übliche Größe	Platzhalter
_Bool	0 (false) und 1 (true)	8 Bit (1 Bit nutzbar)	%u
signed char (int8_t)	-128 bis 127	1 Byte = 8 Bit	%hhd (für Dezimal) %c (für Zeichen)
unsigned char (uint8_t)	0 bis 255	1 Byte = 8 Bit	%hhu (für Dezimal) %c (für Zeichen)
(signed) short (int16_t)	-2^{15} bis $2^{15}-1$	2 Byte = 16 Bit	%hd
unsigned short (uint16_t)	0 bis $2^{16}-1$	2 Byte = 16 Bit	%hu
(signed) int (int32_t)	-2^{31} bis $2^{31}-1$	4 Byte = 32 Bit	%d
unsigned int (uint32_t)	0 bis $2^{32}-1$	4 Byte = 32 Bit	%u
(signed) long (int64_t)	-2^{63} bis $2^{63}-1$	8 Byte = 64 Bit	%ld
unsigned long (uint64_t)	0 bis $2^{64}-1$	8 Byte = 64 Bit	%lu
(signed) long long (int64_t)	-2^{63} bis $2^{63}-1$	8 Byte = 64 Bit	%lld
unsigned long long (uint64_t)	0 bis $2^{64}-1$	8 Byte = 64 Bit	%llu
float (7 Nachkomma, 1×Genauigkeit)	10^{-38} bis 10^{38}	4 Byte = 32 Bit	%f
double (16 Nachkormma, 2×Genauigkeit)	10^{-308} bis 10^{308}	8 Byte = 64 Bit	%f (für `scanf` %lf)

andere Platzhalter

Specifier	Argumenttyp	Ausgabe
% d	Signed Integer	Dezimaldarstellung
% u	Unsigned Integer	Dezimaldarstellung
% x oder X	Unsigned Integer	Hexadezimaldarstellung
% c	Signed Integer	Als ASCII-Zeichen
% s	const char*	Als String

% p **display a pointer**

Escape Sequences

- **`\a` - Bell**
- **`\n` - New line**
- **`\r` - Carriage return**
- **`\b` - Backspace**
- **`\f` - Formfeed**
- **`\t` - Horizontal tab**
- **`\"` - Quotation mark**
- **`\v` - Vertical tab**
- **`\'` - Apostrophe**
- **`\\` - Backslash**
- **`\?` - Question mark**
- **`\0` - Null**

Operator Preference

Operator	Description	Associativity
highest		
() [ ] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right
lowest		

irgendwelche Wert

Wert 1 oder 0 (Typ stets int)

Deklaration von Funktionen ohne Parameter: `void` als Parameterliste

```c
int foo(void); // <-- RICHTIG: akzeptiert keine Parameter

int bar();     // <-- FALSCH: kann mit beliebigen Parametern
               //             definiert/aufgerufen werden
```

Globale und statische Variablen in C werden automatisch mit 0 initialisiert

lokale Variablen in C haben ohne Initialisierung einen *undefinierten* Wert — **Verwendung kann zum Segmentation Fault kommen**

lokale Variablen in Java haben ohne Initialisierung einfach *keinen* Wert

für konstante Variable - Wert-Zuweisung nur gleichzeitig mit der Deklaration möglich, sonst Compilerfehler

```c
TYPE NAME [= VALUE];       // Deklaration [und Zuweisung]

const TYPE NAME = VALUE;   // Deklaration und Zuweisung einer
                           // konstanten Variable
```

Vorsicht bei const in Kombination mit Pointern:

```c
const TYPE* PTR [= ADDR];      // Pointer auf konstante Daten

TYPE* const PTR = ADDR;        // Konstanter Pointer auf
                               // variable Daten

const TYPE* const PTR = ADDR;  // Konstanter Pointer auf
                               // konstante Daten
```

Kontrollfluss-Strukturen

```c
if(expression1)
    statement_1;
else if(expression2)
    statement_2;
else if(expression3)
    statement_3;
else
    statement_n;
```

```c
switch (expression)
{
    case constant-expression1:
        statement_1;
        break;
    case constant-expression2:
        statement_2;
        break;
    // break; springt hinter der Schleife
    // continue; springt am Anfang der Schleife
    ...
    default:
        statement_default;
}
```

```c
do
{
    statements;
    aktuakisiere loop_counter;
} while (condition with loop_counter);
```

```c
while (condition with loop_counter)
{
    statements;
    aktuakisiere loop_counter;
}
```

```c
for (initialization; test condition; run every time command)
{
    block of statement
}
```

Standard-Header

```c
// system_header.h
#include <stdio.h>    // Eingabe und Ausgabe
#include <stdint.h>   // plattformunabhängige Definition von Ganzzahltypen
#include <float.h>    // Wertbereichen von Gleitkommazahlen
#include <stdbool.h>  // Unterstützung für Boolesche Variablen
#include <string.h>   // Funktionen zur String-Manipulation
#include <stdlib.h>   // vermischte Standardfunktionen (z.B. Speicherverwaltung)
#include <stddef.h>   // zusätzliche Typdefinitionen (unsigned, max.Größe)
#include <math.h>     // mathematische Funktion

// lokal_header.h
#include "myheader.h"
```

mit abschließender Newline `\n` - Ausgabe von `printf` sofort erfolgt

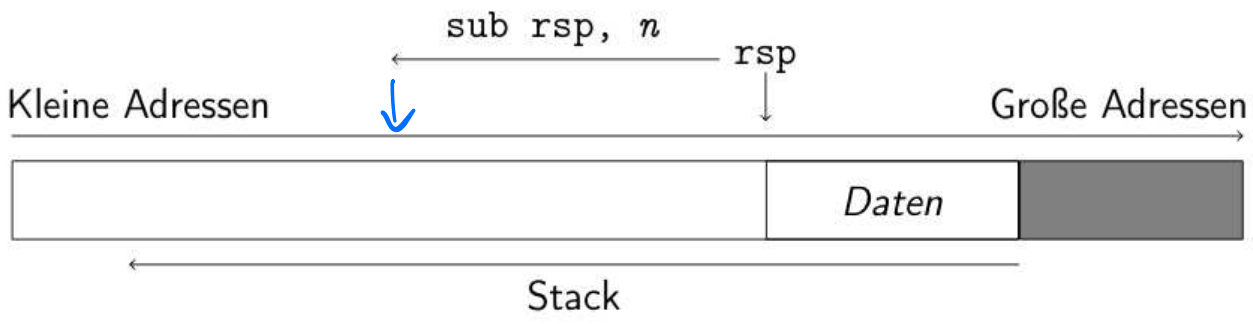
ohne abschließende Newline `\n` - Ausgabe von `printf` kommt erst einmal in einen Zwischenspeicher (Buffer)

Integer Promotion

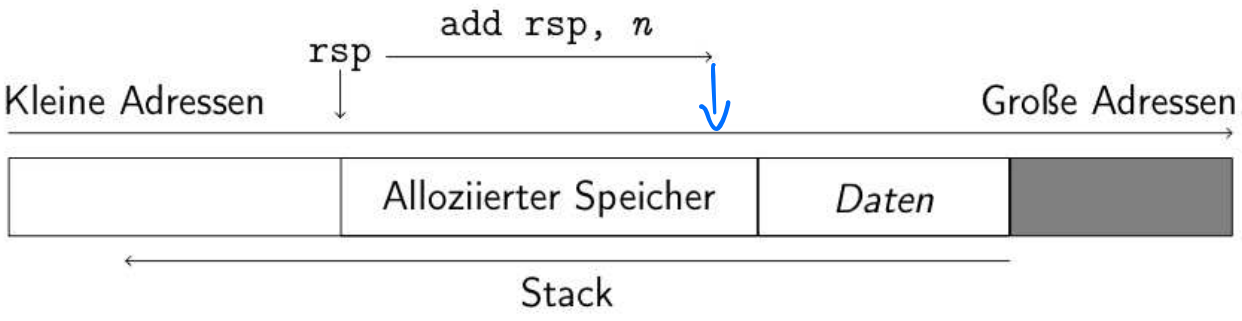
"If an int can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an int; otherwise, it is converted to an unsigned int."

Stack-Manipulation

Push



Pop



Optimierungen

Optimierungsstufen

- Compiler-Optimierungen nur valide, wenn Verhalten unverändert
  - Viele Optimierungen nicht vollautomatisch durchführbar!

`-O0`	Keine Optimierungen (Default)
`-O1` `-Og`	Optimierungen mit wenig Compile-Zeit `-O1` mit Fokus auf debugbaren Code
`-O2` `-Os`	Alle Optimierungen ohne Space–Speed Trade-off `-O2` mit Fokus auf minimaler Code-Größe
`-O3` `-Ofast`	Alle Optimierungen `-O3` mit Floating Point Optimierungen ("Disregard strict standards compliance")

Optimierungsziele -> Laufzeit schneller und Speicherplatz weiniger

bei Optimierung soll ZUERST betrachtet -> Verbesserung der Laufzeitklassen

Optimierter Code hat bessere Performance, ABER ist meist

- aufwändiger zu schreiben

- schwewer zu lesen/ warten

- komplizierter zu testen/ debuggen

K(n)	Laufzeitklasse
2^n	exponentiell
n^2	quadratisch
n	linear
$\log n$	logarithmisch
1	konstant

Optmierung mittels LUT - Vorberechnung und Speichern der Zahlen in Lookuptabelle

PRO: konstante Berechnungszeit, geeignet für häufig auftretende identische Berechnungen

CONTRA: 1) nur verwendbar, wenn alle gewünschte Werte bereits vorberechnet sind

2) potentiell hohe Speicherplatzverbrauch, nicht für sehr große Werte möglich

kleine Lookuptabelle (kleine LUT) - Lookuptabelle in Abschnitte unterteilen

1) Erste zwei Werte jeden Abschnitts speichern; 2) Restlichen Werte dynamisch zur Laufzeit berechnet

z.B. 6 Abschnitte mit je 16 (14 im letzen Abschnitt) Zahlen -> Lookuptabelle schrumpft von 94 Einträgen auf 12

Rekursiv -> exponentiellen Laufzeitkomplexität und Speicherkomplexität wegen rekursive Funktionsaufruf

Lineare Schleife -> am besten auf Speicherplatz optimiert

LUT -> am besten auf Laufzeit optimiert

Kleine LUT -> am besten auf Laufzeit und Speicherplatz allgemein optimiert

`sizeof`-Operator -> ermittelt die Größe des als Argument übergebenen Datentyps in Byte

Alignment -> Speicheradresse muss ein Vielfaches vom Alignment sein

Pointer -> Adresse eines Speicherobjektes, "zeigt" auf das Objekt

Operator `&` -> nimmt Adresse eines Objekts

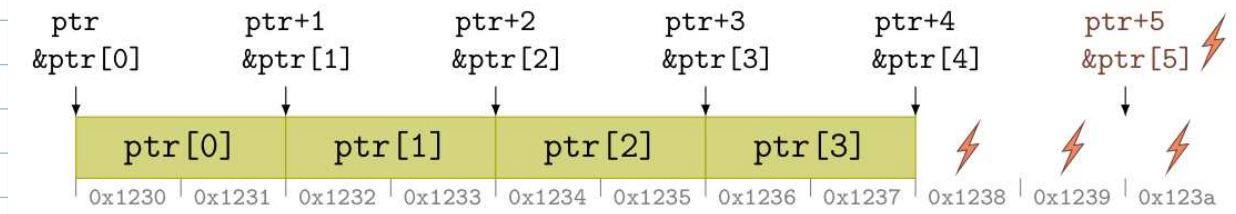
Operator `*` -> Dereferenzieren eines Pointers (Speicherzugriff)

```c
int i = 0;
int* i_ptr = &i;
int i_new = *i_ptr;    // i_new == i
```

Pointer-Arithmetik

- `ptr[0]` ≙ `*ptr`
- `ptr[3]` ≙ `*(ptr + 3)`

  = * (&ptr[0] + 3)



Pointer-Casting

```c
// implizite Casting zwischen void-Pointers und anderem Pointer-Typ erlaubt
int* i_ptr =  ... ;
void* v_ptr = i_ptr; // implizite Pointer-Casting von int zum void
int* i_ptr2 = v_ptr; // implizite Pointer-Casting von void zum int
int i = *v_ptr; // Dereferenzierung eines void-Pointers ist nicht möglich
v_ptr++/--; // In-/Dekrementieren eines void-Pointers - undefiniert behavior

// Derenferenzierung von expliziten Pointer-Casting: undefined behavior
int* i_ptr =  ... ;
char* c_ptr = (char*) i_ptr; // Zugriff möglich - (Einzige Ausnahme: char-Pointer)
short* s_ptr = (short*) i_ptr; // Zugriff nicht möglich - undefined behavior
long* l_ptr = (long*) i_ptr; // undefined behavior (int hat strengere Alignmentanforderungen als char)
```

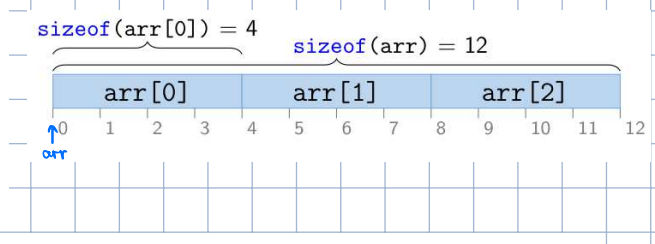
Datenstruktur: Array

Repräsentation als Speicherbereich mit Datenobjekten gleichen Typs direkt aufeinanderfolgend

Zugriff: syntaktisch analog zu Pointern z.B. `arr[0]` oder `*arr`

Ein Array lässt sich immer wie ein Pointer verwenden, aber nicht umgekehrt

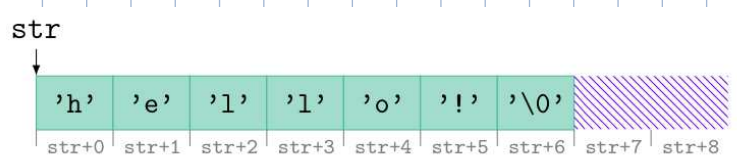
`sizeof`-Operator ist sowohl für Arrays als auch für Pointer definiert (mit verschiedener Semantik)




```c
// Definitionsmöglichkeiten
int arr1[3];  // 1) Deklaration mit impliziter Größenangabe
arr1 = (int [3]) {1,2,3}; // nach Deklaration - neue Zuweisung verboten, nur Modifikation einzelnes Elements erlaubt
int arr2[] = {1, 2, 3};  // 2) Verzicht auf explizite Größenangabe
int arr[3] = {1, 2, 3}; // 3) Deklaration und Initialisierung gleichzeitig
size_t arr_len = sizeof(arr) / sizeof(arr[0]); // Array-Größe = gesamte Größe/ Elementgröße
```

```c
// Mehrdimensionale Array
unsigned matrix [2][3] = {{1, 2, 3}, {4, 5, 6}};
unsigned matrix [a][3] = {{1, 2, 3}, {4, 5, 6}}; // explizite Größenangabe der "oberste Ebene a" fehlt - kein problem
unsigned matrix [2][b] = {{1, 2, 3}, {4, 5, 6}}; // explizite Größenangabe der "unterste Ebene b" fehlt - Compilerfehler
```

Datenstruktur: String



Repräsentation als char-Array (implizit mit 0-Byte terminiert)

```c
// Definitionsmöglichkeiten
char str[4] = {'t', 'u', 'x', '\0'}; // 1) Array mit explizitem null-Byte (darf weiter modifiziert werden)
char str[] = "tux"; // 2) mittels eines String-Literals mit implizitem null-Byte (darf weiter modifiziert werden)
const char* str = "tux"; // 3) über einen Pointer statt einem Array (Modifikation von String Literals - undefined behavior)
```

```c
// sizeof(str) = strlen(str) + 1 (Platz für terminierende null-Byte)
char str[] = "tux";
size_t len = strlen(str); // strlen(str) = 3  -> wie viele chars hat der String (ohne null-Byte)
size_t size = sizeof(str);// sizeof(str) = 4  -> wie viel Speicherplatz dafür allokiert sind (mit null-Byte)
```

struct

Repräsentation als zusammengesetzter Datentyp mit Elementen verschiedenen Typs

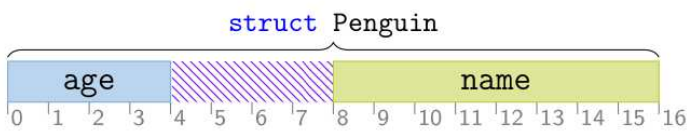
Elemente liegen aufeinanderfolgend (ggf. mit Padding) im Speicher

Zugriff direkt `penguin.age = 0;` oder über einen Pointer `penguin_ptr->age = 0;`

Direkte Rückgabe von `struct` ist möglich, Rückgabe über einen Pointer mit dynamischer Speicherdauer ist keine gute Idee

Übergabe von Argumenten in C erfolgt immer by-value -> nur eine Kopie wird verändert, aber nicht auf dem originalen operiert

```c
struct Penguin {
  int age;
  char* name;
};
```

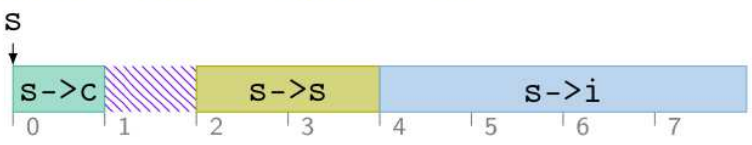



```c
// Initialisierungsmöglichkeiten
struct Penguin penguin1 = { .age = 0, .name = "Tux" }; // 1) normale Initialisierung
struct Penguin penguin2 = { .age = 0 }; // 2) implizite Initialisierung: penguin2.name = NULL
struct Penguin penguin3 = { 0, "Tux" }; // 3) Initialisierung mit festgelegter Reihenfolge
struct Penguin penguin4 = { 0 }; // 4) Initialisierung aller Elemente zu 0
```

struct vs union

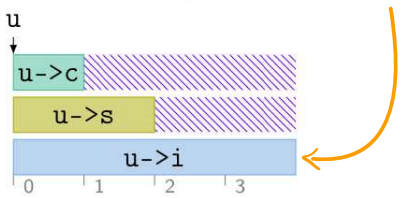
- `struct` speichert immer alle Elemente

```c
struct { char c; short s; int i; }* s;
```

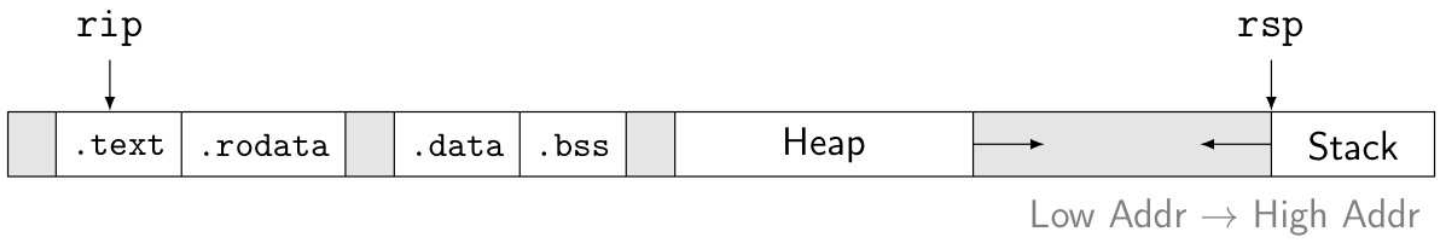


- `union` speichert genau ein Element – das zuletzt geschriebene

```c
union { char c; short s; int i; }* u;
```



Speicherbereiche



- .text: Beinhaltet den Programmcode (readonly, executable)
- .rodata: Beinhaltet globale konstante initialisierte Variablen (readonly)
  - Beispiel: const int i = 42; (global)
- .data: Beinhaltet globale initialisierte Variablen (read-write)
  - Beispiel: int i = 42; (global)
- .bss: Beinhaltet globale Variablen, die mit 0 initialisiert sind
  - Beispiel: int i; (global)

Stack (LIFO Prinzip)

- **für kleine Datenmenge**
- **wächst von oben nach unten**
- **enthält lokale Variablen**
- **automatische Speicherfreigabe**

Heap

- **für große Datenmenge**
- **wächst von unten nach oben**
- **Allokationen global verwendbar**
- **dynamische Allokation & Freigabe**

Speicherverwaltung auf dem Heap

Speicherallokation:

- Funktionen aus `stdlib.h`:
  - `void* malloc(size_t size);`
  - `void* calloc(size_t nmemb, size_t size);`
- Im Falle eines Fehlers: `NULL`-Pointer Rückgabewert – *Immer überprüfen!*

Speicherfreigabe

- Kein Garbage Collector
- Funktion `void free(void* ptr)` aus `stdlib.h`
  - Nur *original* `ptr` von `malloc`, `calloc` etc. `free`n!
  - Nach `free`, `ptr` nicht mehr für Speicherzugriffe und -verwaltung verwenden!

void* alloca(size_t size) reserviert size Bytes Speicher auf dem Stack

`void* realloc(void* ptr, size_t size);`

- Alter Speicher bei Erfolg automatisch freigegeben
- Vergrößerung: Neue Daten uninitialisiert
- `realloc(NULL, size)` ≙ `malloc(size)`
- `NULL`-Pointer Rückgabewert – Fehler!
  - Alter Speicherbereich wird nicht freigegeben

`void* aligned_alloc(size_t alignment, size_t size);`

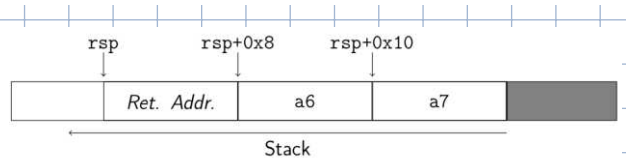
- `alignment` muss Zweierpotenz sein
- `size` muss Vielfaches des `alignment`s sein

Calling Convention

Funktionsparameter

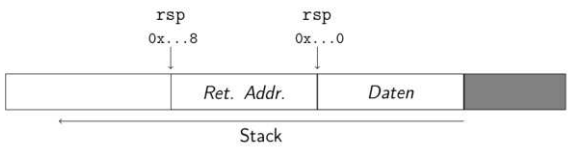
ab 7. Parameter

- Parameter: `rdi, rsi, rdx, rcx, r8, r9,` Stack
  Parameter 1.~6.
- Rückgabewert: `rax, rdx`



Stack-Alignment

- 16-Byte-Alignment vor einem Funktionsaufruf `rsp % 16 == 0`
- `call` legt weitere 8 Byte auf den Stack ⇝ `rsp % 16 == 8`



Registertypen

Register gehört dem caller

- Callee-saved: `rbx, rbp, rsp, r12–r15` (→ aufrufende Funktion)
  - Gleicher Wert nach Funktionsende, ggf. sichern und wiederherstellen

Register gehört dem callee (→ aufgerufene Funktion)

- Temporär/caller-saved: `rax, rcx, rdx, rsi, rdi, r8–r11`
  - Frei verwendbar, Wert nach Funktionsende unbekannt

struct Layout

-- Alignment des structs ist das seines Felds mit dem größten Alignment

-- Größe des structs ist ein Vielfaches davon → Padding am Ende

struct als Funktionsparameter übergeben

a) `struct` kleiner als 16 Byte: Aufteilen in max. 2 Teile mit jeweils 64 Bit (als Int-Parameter behandelt)

-> Mehrere benachbarte Felder können zusammengefasst werden

b) `struct` größer als 16 Byte oder Felder die nicht aligned sind: Übergabe via Stack

- 2 64-Bit Integer

```
1 void func(struct { uint64_t a; uint64_t b; } param);
2 // a -> rdi, b -> rsi
```

- 2 32-Bit Integer

```
1 void func(struct { uint32_t a; uint32_t b; } param);
2 // a -> rdi[31:0], b -> rdi[63:32]
```

- mehrere zusammengefasste Felder

```
1 void func(struct { uint16_t a; uint16_t b; uint8_t c; } param);
2 // a -> rdi[15:0], b -> rdi[31:16], c -> rdi[39:32]
```

- struct größer als 16 Byte

```
1 void func(struct { uint64_t a; uint64_t b; uint64_t c; } param);
2 // a -> [rsp], b -> [rsp + 8], c -> [rsp + 16]
```

falls mehrere struct nacheinander als Parameterübergabe gibt, betrachtet man jede struct getrennt voneinander

rdi[0,31] rdi[63,32]

```
struct bar { int a; int b; };
struct baz { long c; int d; };
          rsi    edx

void foo(struct bar bar, struct baz baz);
```

Buffer Overflow: Lesende bzw. schreibende Zugriffe über die Grenzen des Buffers hinaus

- Im Codebeispiel: `scanf("%40s", buf)` ist Off-By-One Buffer Overflow
  - `scanf("%40s", buf)` liest bis zu 40 Zeichen und hängt ein Nullbyte an
  - ⇒ genau einen Wert über den Buffer hinaus geschrieben

- Um den Fehler zu vermeiden kann man:
  - Ein Byte weniger als die Länge des Buffers in an `scanf()` übergeben
  - Oder deutlich besser: `fgets()` verwenden

Stack Overflow = Buffer Overflow auf Stack

Heap Overflow = Buffer Overflow auf Heap

Segmentation Faults: Verletzung des Speicherschutzes

Vermeidung durch

-> Schreibenden Zugriff auf read-only Daten

-> Zugriff auf Daten mit fehlenden Berechtigungen →

-> Null-Pointer Dereferenzierung

-> Pointer vor Benutzung überprüfen

-> Fehler bei Einhaltung der Array-Grenzen überprüfen

-> Beachtung des Nullterminals bei C-Strings überprüfen

-> Fehlerhafte Speicherzugriffe durch Verletzung der Calling Convention

`malloc` gibt im Fehler-Fall einen NULL-pointer zurück

-> immer überprüfen, ob `malloc` NULL zurückgibt

mehrere `printf`-Statements auf einer Zeile sind erlaubt, aber nicht empfehlenswert

Ein `\n` am Ende von Strings ist nicht zwingend nötig bei `printf`-Ausgabe

Speicher im Heap bleibt reserviert, auch wenn das Programm ihn nicht mehr verwendet, aber nicht explizit freigegeben

Memory Leak -> Vermeidung durch Freigabe von unbenötigtem Speicher

(muss nicht innerhalb dieser Funktion free(), kann auch erst in andere Funktion free())

Vor `return`-Statements immer alle durch `malloc`, `calloc` etc. belegten Speicherbereiche freigeben!

Use after Free: `buf` wird erst durch `free()` freigegeben und dann in `printf` benutzt -> undefined behavior

Double Free: `buf` wird erneut durch `free()` freigegeben -> undefined behavior

Undefined behavior

- Dereferenzierung von Nullpointer
- Double Free
- Use after free
- Lesen uninitialisierter Variablen
- Signed Integer Overflow
- Shift um Länge eines Integerwerts (oder mehr oder negativ)
- Flushen eines Inputstreams, z.B. `fflush(stdin)`
- Fehler bei Pointercasts (meist unnötig/obsolet)

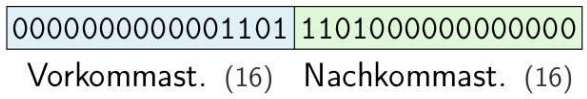
Vermeidung von Fehler - Sanitizer

- Verschiedene Sanitizer können über Compilerflags aktiviert werden:
  - `-fsanitize=address` für Buffer Overflows und Dangling Pointer
  - `-fsanitize=leak` für Memory Leaks
  - `-fsanitize=undefined` für Undefined Behavior

- Nachteile der Verwendung der Sanitizer
  - Erschwert Debugging mit anderen Tools
  - Performanz des Programms wird deutlich verringert
  - Erkennen bei Weitem nicht alle Fehler

Fix/ Festkommazahl

Aufteilung der Bits in Vor- und Nachkommastellen



Arithmetik

Addition/ Subtraktion: Komma muss an gleicher Stelle sein; ansonsten keine weiteren Operationen notwendig

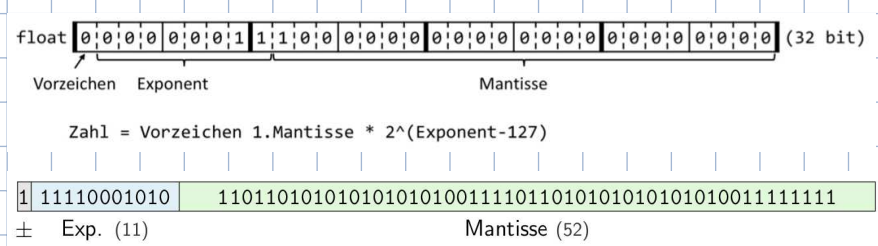
Multiplikation: zweier Festkommazahlen addieren sich die Kommapositionen und dann Shifts notwendig

Division: Multiplikation mit Kehrwert

Vor- und Nachteile

- Schnelle Arithmetik und sehr verständlich
- Gleichbleibende Genauigkeit im gesamten Wertebereich
- Kommaposition fest -> Hoher Speicherplatzverbrauch bei großen Wertebereichen

Gleit-/ Fließkommazahl



größte darstellbare Zahl

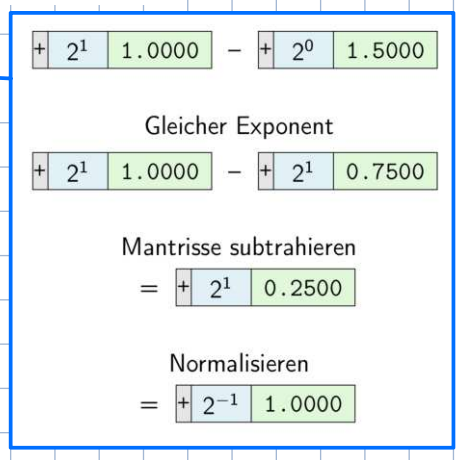
float 2^24

double 2^53

Arithmetik

Addition/ Subtraktion

1. Kleineren Wert auf selben Exponenten bringen wie großen Wert (denormalisieren)
2. Mantissen addieren bzw. subtrahieren (entsprechend der Genauigkeit runden)
3. Ergebnis normalisieren



Multiplikation/ Division

1. Exponenten addieren bzw. subtrahieren
2. Mantissen multiplizieren bzw. dividieren (Führende 1 beachten) und dann runden
3. Ergebnis normalisieren

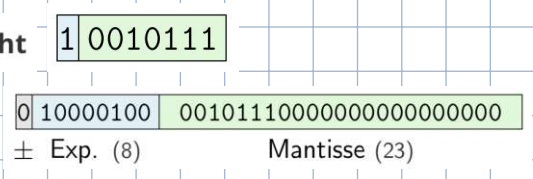
Vor- und Nachteile

- Großer Wertebereich bei weniger Speicherplatzverbrauch
- Ungenauigkeiten durch Rundung (Absorption, Auslöschung)
- Arithmetik nicht trivial

- Rundung: Ergebnis muss wieder in FP-Darstellung gespeichert werden
  - Verschiedene Rundungsmodi, Standard: *round to nearest, ties to even*
- Absorption: Addition/Sub. von sehr großer und sehr kleiner Zahl
  - Keine Veränderung der großen Zahl wg. Rundung
  - Beispiel: $1000000.00f + 0.01f = 1000000.00f$
- Auslöschung: Subtraktion großer ähnlicher Zahlen
  - Subtraktion verstärkt Rundungsfehler
  - Beispiel: $1000000.1f - 1000000.0f = 0.125f \neq .1f$
  - Grund: $1000000.1f$ tatsächlich dargestellt als 1000000.125

Konvertierung von 37.75 zur Fließkommazahl

1. Konvertierung zu einer Fixkommazahl — 100101 | 11
2. Komma so weit verschieben, sodass nur noch eine 1 vor dem Komma steht — 1 | 0010111
3. Nachkommastellen = Mantisse; #Rechtsshifts + Bias(127) = Exponent



Nicht assoziativ $(x + y) + z \neq x + (y + z) \quad (x \times y) \times z \neq x \times (y \times z)$

Nicht distributiv $x(y + z) \neq (xy) + (xz)$

Besondere Fließkommazahlen

Denormale Zahlen / Subnormale Zahlen: Exponent hat alle Bits 0

- Zahlen deren Exponent kleiner ist, als eine normalisierte Darstellung zulassen würde

Null mit Vorzeichen +/-0 : Sign-Bit egal; Exponent und Mantisse alle Bits 0

Unendlich / Infinity / ∞ : Alle Bits in Exponent = 1; Alle Bits in Mantisse = 0; je nach Sign-Bit: +/− Unendlich (-Infinity < +Infinity $:=$ true)

- Zähler u. Nenner haben gleichen Vorzeichen -> Ergebnis positiv (Positive Zahl geteilt durch +0 -> ergibt +Infinity)

- Zähler u. Nenner haben unterschiedlichen Vorzeichen -> Ergebnis negativ (negative Zahl geteilt durch +0 -> ergibt -Infinity)

Not a Number / NaN : Alle Bits in Exponent = 1; Mantisse $\neq 0$ -> Beispiel $0/0$ und $\infty - \infty$

- Jede Operation mit NaN ergibt wieder NaN; Jeder Vergleich mit NaN wird false 0 (nur Ausnahme $\neq$ wird true 1)

SSE-Register und Instruktionen

Register `xmm0` bis `xmm15` - verwendbar für Floating-Point Berechnungen

`pxor dst, src` XOR für xmm-Register

`movss dst, src` FP-Konstanten können aus dem Speicher geladen werden z.B. `movss xmm0, [rip + .Lconstx]`

- Wenn der zweite Operand ein Speicherzugriff ist, werden die Bits [32;127] des xmm-Registers auf Null gesetzt

- Wenn der zweite Operand ein Register ist, dann werden nur die Bits [0;31] des Ziel-Registers geändert

nur untere 32 bits für Addition und Subtraktion

`addss dest, src` Addition zweier Floating-Point Werte

`subss dest, src` Subtraktion zweier Floating-Point Werte

`divss dest, src` Division mit zwei Floating-Point Werten

`mulss dest, src` Multiplikation mit zwei Floating-Point Werten

`ucomiss op1, op2` Skalarer Vergleich zweier Floating-Point Werte

Sprünge mittels JCC

```
ucomiss xmm1 , xmm0
jp .Lunordered   // xmm0 or xmm1 NaN
jb .Llesser      // xmm1 < xmm0
ja .Lgreater     // xmm1 > xmm0
je .Lequal       // xmm1 == xmm0
```

`ss` -> Scalar Single (Precision) `sd` -> Scalar Double (Precision)

`cvtsi2ss` konvertiert eine Single Integer (`si`) in einen Single-Precision (`ss`) Floating-Point Wert

`cvtss2si` konvertiert einen Single-Precision (`ss`) Floating-Point Wert in eine Single Integer (`si`)

Erweiterte Calling Convention für FP

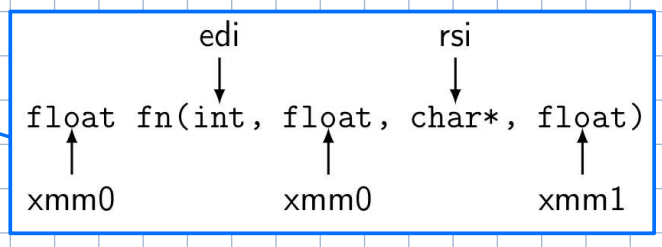
- Floating-Point Rückgabewert: `xmm0`

- Floating-Point Argumente: `xmm0` bis `xmm7` (weitere auf Stack)

- Alle Register sind caller-saved/ temporär

Kombinationen von FP und Int/Ptr Argumenten

-> Seperate Durchnummerierung der Register



SIMD mit SSE

SSE-Instruktionen für SIMD (Single Instruction Multiple Data)

-- Neue eigenständige Instruktionen für parallele Datenverarbeitung

-- Instruktionen arbeiten auf ganzem 128-Bits XMM-Register zur gleichen Zeit

-- Gesonderte Befehle für Integer und Floating-Point Berechnung - Schlüsselwort: `Packed (P)`

BYTE = 1 Byte (8 Bits)

WORD = 2 Bytes (16 Bits)

DWORD = 4 Bytes (32 Bits)

QWORD = 8 Bytes (64 Bits)

Integer Instruktionen

`PADDB xmm1, xmm2/m128` ADD Packed Integer BYTE - Addiert 16 BYTE (8 Bits) Integer auf andere 16 BYTE (8 Bits) Integer

`PADDW xmm1, xmm2/m128` ADD Packed Integer WORD - Addiert 8 WORD (16 Bits) Integer auf andere 8 WORD (16 Bits) Integer

`PADDD xmm1, xmm2/m128` ADD Packed Integer DWORD - Addiert 4 DWORD (32 Bits) Integer auf andere 4 DWORD (32 Bits) Integer

`PADDQ xmm1, xmm2/m128` ADD Packed Integer QWORD - Addiert 2 QWORD (64 Bits) Integer auf andere 2 QWORD (64 Bits) Integer

`PSUBQ` `__m128i _mm_sub_epi64 (__m128i a, __m128i b)` - Subtrahiert 2 QWORD (64 Bits) Integern auf andere 2 QWORD Integern

`movdqu xmm1, [rdi]` lädt 4 Integer aus dem Speicher ab `rdi` in `xmm1`

`movdqu [rdi], xmm1` schreibt das Resultat (4 Integer aus `xmm1`) in den Speicher ab `rdi`

Fließkommazahl Instruktionen

`ADDSS xmm1, xmm2/m32` ADD Scalar Single-Precision Floating-Point Values - Addiert einen 32 Bits Float auf einen anderen 32 Bits Float

`ADDPS xmm1, xmm2/m128` ADD Packed Single-Precision Floating-Point - Addiert 4 Float (32 Bits) auf andere 4 Float (32 Bits)

`ADDPD xmm1, xmm2/m128` ADD Packed Double-Precision Floating-Point - Addiert 2 Double (64 Bits) auf andere 2 Double (64 Bits)

Data Alignment

Die meisten SIMD-Instruktionen erfordern ein 16Byte Alignment (außer Skalare SSE-Instruktionen wie `ADDSS`)

Addresse muss 16Byte aligned (durch 16 teilbar) sein -> Die niederwertigsten 4 Ziffern in Binär-Form sind 0

Alignment von Variable und Structure-Fields in C mittels: `__attribute__((aligned (16)))`

Exceptions bei Zugriff auf non-aligned Speicherbereich

Aligned Zugriff & Unaligned Zugriff

`movaps xmm/m128, xmm/m128` Move Aligned Packed Single-Precision - Speicher-Operanden müssen aligned sein

`movups xmm/m128, xmm/m128` Move Unaligned Packed Single-Precision - Speicher-Operanden müssen nicht aligned sein

-- Kopiert 4 Floats (32 Bits) vom Ursprung ins Ziel (32 Bits) -> Im Zweifelsfall `movups` verwenden, bei Kenntnis aber `movaps`

Load `__m128 _mm_load_ps(float const* mem_addr)` -> `movaps xmm0, [rdi]` lädt 4 Floats ab Speicheradresse `rdi` ins Register `xmm0`

Store `void _mm_store_ps(float* mem_addr, __m128 a)` -> `movaps [rdi], xmm0` schreibt 4 Floats vom `xmm0` an Speicheradresse `rdi`

`movaps` löst mit `xmm`-Zielregister eine general-protection Exception (#GP) aus, wenn die Adresse kein Vielfaches von 16 ist

wenn `loadu_ps` (non-aligned Adressen) durch `load_ps` (aligned Adressen) im Programm ersetzt -> Programm-Absturz (Segmentation Fault)

Optimierungen in GNU C

Optimierung von Berechnungen

- Constant Folding
  - Berechnung von Konstanten zur Compilezeit
- Constant Propagation
  - Variablen werden mit ihren Werten ersetzt
  - Bei Funktionen: Ergebnis des Funktionsaufrufs wird bereits berechnet
- **C**ommon **S**ubexpression **E**limination

```
1 int x = a * b * 24;
2 int y = a * b * c;
```

→

```
1 int tmp = a * b;
2 int x = tmp * 24;
3 int y = tmp * c;
```

Interprozedurale Optimierungen

- Entfernen unnötiger Funktionsparameter
  - Nur bei `static` Funktionen möglich
- Funktions-spezifische Calling Convention
  - Z.B. mehr callee-saved Register, andere Argumentregister
  - Nur bei `static` Funktionen ohne externe Nutzung möglich
- Spezialisierung von Funktionen bei mehreren verschiedenen Aufrufen
  - Duplikation und Optimierung für verschiedene Parameterwerte

Low-Level Optimierungen

- Instruction Selection: Statement → Instruktionen
  - Z.B. Ersetzen von Multiplikation mit `lea`
- Instruction Scheduling: Reihenfolge der Instruktionen
  - Verringern von Abhängigkeiten zwischen Instruktionen
  - Bessere Ausnutzung von Instruction-Level Parallelism im Prozessor
- Register Allocation: Variablen → Register/Stack
  - Verringern/vermeiden von Stack-Zugriffen

Optimierung von Schleifen

Loop Unrolling

PRO: Erhöhte Geschwindigkeit (Loop Conditions werden weniger/gar nicht getestet)

CONTRA: Executable Größe wächst (Mehr Instruktionen nötig)

Unoptimiert

```
1 for(int i = 0; i < 6; i++) {
2     arr[i] = 2*arr[i];
3 }
```

Optimiert[1]

```
1 for(int i = 0; i < 6; i += 2) {
      arr[i] = 2*arr[i];
      arr[i+1] = 2*arr[i+1];
```

Jamming/ Loop Fusion: Vermeidung von doppeltem Schleifen-Overhead

Loop-Invariant Code Motion: Vermeidung redundanter Berechnungen

Vertauschung: Verbessert das Cacheverhalten (weniger Cache-Misses aufgrund der Vertauschung)

Optimierung von Funktionsaufrufen

Inlining

PRO - Kein Overhead durch Funktionsaufruf

CONTRA - kann Code massiv vergrößern und damit verlangsamen

Unoptimiert

```
1 static int square(int x) {
2     return x*x; }
3 ...
4 for (int i = 0; i < n; i++)
5     arr[i] = square(i);
```

Optimiert[1]

```
1 for (int i = 0; i < n; i++)
2     arr[i] = i*i;
```

Tail Call Optimization

Bedingung - Funktionsaufruf ist letzte Operation vor `return`

Ersetzte Funktionsaufruf (`call+ret`) durch jmp (`goto`) zur Funktion

Unoptimiert

```
1 int fac(int k, unsigned n) {
2   if (n <= 0) return k;
3   return fac(k * n, n - 1);
4 }
```

Optimiert[1]

```
1 int fac(int k, unsigned n) {
2 fac:
3   if(n <= 0) return k;
4   k *= n;
5   n--;
6   goto fac;
7 }
```

Optimierte Funktionen

`libc` stellt häufig benutzte Funktionen hochoptimiert bereit

Beste Funktion wird zur Laufzeit ausgewählt -> `libc` vor eigener Implementierung bevorzugen

Builtins - Funktionen für bestimmte Anwendungen (nicht von Standardbibliothek!)

`__builtin_clz(unsigned int x` Count Leading Zeros

`__builtin_expect(long exp, long c)` falls `exp == c` gilt - liefert `true`

Funktionsattribute `__attribute__((etwas))`

`always_inline` : Inlining wird erzwungen

`noinline` : Inlining wird verhindert

`const` : Ausgabe nur durch Eingabe bestimmt, unabhängig vom Programm-Zustand, `const` -Fkt. darf nur `const` -Fkt. aufrufen

`pure` : Rückgabewert darf von Dereferenzierung übergebener Pointer abhängen, `pure` -Fkt. dürfen `pure` -Fkt. und `const` -Fkt. aufrufen

`hot` : für besonders oft aufgerufene Funktionen - höhere Optimierung auf Geschwindigkeit, größerer Code, eigener Speicherbereich

`cold` : für besonders selten aufgerufene Funktionen, langsamer, kleinerer Code,eigener Speicherbereich

Layout von Datenstrukturen

-- Größe der verwendeten Datentpyen - So groß wie nötig & So klein wie möglich

-- Verwendung von fixed-width Integern sinnvoll - `uint32_t` statt `unsigned int`

Pointer: Alias & restrict

`U* ptr2` zeigt auf gleichen Speicherbereich wie `T* ptr1` -> `ptr2` ist Alias von `ptr1`

-- Nicht jeder Pointer kann Alias für jeden anderen sein, Nur gültige Aliase sollten auf gleichen Speichebereich zeigen

`restrict` : Die Pointer zeigen nicht auf gleichen Speicherbereiche -> keine (gültigen) Aliase

-- z.B. `void foo2(unsigned* restrict ptr_a, int* restrict ptr_b)`

SIMD Vergleichsbefehl

Vergleichsbefehle - Ergebnis als Bitmaske in `xmm`-Register

-- Falls Bedingung erfüllt, alle Bits in Element auf 1 gesetzt `0xffffffff`

-- Falls Bedingung nicht erfüllt, alle Bits in Element auf 0 gesetzt `0`

Prüfen auf Gleichheit `=` `__m128i _mm_cmpeq_epi_zahl (__m128i a, __m128i b)`

`PCMPEQB xmm1, xmm2/m128` Vergleicht BYTE (8 Bits) von `xmm1` mit `xmm2/m128` auf Gleichheit, Ergebnis in `xmm1`

`PCMPEQW xmm1, xmm2/m128` Vergleicht WORD (16 Bits) von `xmm1` mit `xmm2/m128` auf Gleichheit, Ergebnis in `xmm1`

`PCMPEQD xmm1, xmm2/m128` Vergleicht DWORD (32 Bits) von `xmm1` mit `xmm2/m128` auf Gleichheit, Ergebnis in `xmm1`

`PCMPEQQ xmm1, xmm2/m128` Vergleicht QWORD (64 Bits) von `xmm1` mit `xmm2/m128` auf Gleichheit, Ergebnis in `xmm1`

Prüfen auf Greater-Than `>` `__m128i _mm_cmpgt_epi_zahl (__m128i a, __m128i b)`

`PCMPGTB xmm1, xmm2/m128` Vergleicht BYTE (8 Bits) von `xmm1` größer als BYTE (8 Bits) von `xmm2/m128`, Ergebnis in `xmm1`

`PCMPGTW xmm1, xmm2/m128` Vergleicht WORD (16 Bits) von `xmm1` größer als WORD (16 Bits) von `xmm2/m128`, Ergebnis in `xmm1`

`PCMPGTD xmm1, xmm2/m128` Vergleicht DWORD (32 Bits) von `xmm1` größer als DWORD (32 Bits) von `xmm2/m128`, Ergebnis in `xmm1`

`PCMPGTQ xmm1, xmm2/m128` Vergleicht QWORD (64 Bits) von `xmm1` größer als QWORD (64 Bits) von `xmm2/m128`, Ergebnis in `xmm1`

Es gibt keine `pcmpltw` Instruktion (Less-Than) -- `xmm0 < xmm1` ist identisch zu `xmm1 > xmm0`

Es gibt keine `pcmpged` Instruktion (Greater-Equal) -- `xmm0 <= xmm1` ist identisch wie `(xmm0 < xmm1) OR (xmm0 = xmm1)`

Vergleich um die vorzeichenlosen `n`-Bit Elemente in zwei Register

1) 2^{n-1} auf alle Elemente beider Register addieren/ subtrahieren

2) dann Vergleichsbefehl ausführen

Beispiel

prüft ob die vorzeichenlosen 8-Bit Elemente von `xmm0` kleiner sind als die von `xmm1` ($2^7 = 128$)

1a) die kleinste unsigned Zahl 0 um 128 subtrahieren zur kleinsten signed Zahl -128

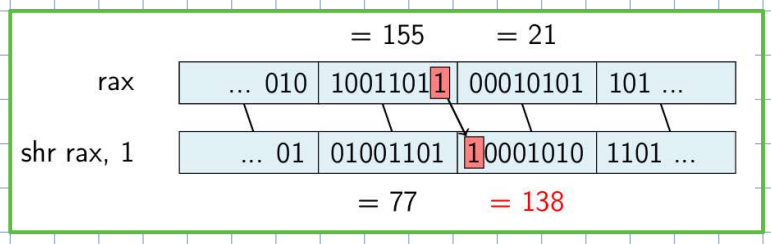
1b) die größte unsigned Zahl 255 um 128 subtrahieren zur größten signed Zahl 127

2) dann Vergleichsbefehl `pcmpgtb xmm1, xmm0` ausführen

SIMD mit General Purpose Registern

Beispiel: Elemente von `uint8_t` Array in `rax` durch 2 teilen

Problem: Niedrigstes Bit von einem Element wird ins nächste geschoben



Lösung

Division durch 2 für einen Vorzeichenbehafteten int8_t Array

1) Elemente jeweils in `eax` und `edx` laden `mov eax, [rdi]` `mov edx, [rdi]`

2) Division durch 2 mit Bitshift `shr eax, 1`

3a) die höchsten Bits mit einer Bitmaske extrahieren `and eax, 0x80808080`

3b) die vorzeichenlosen Teil mit einer Bitmaske extrahieren `and eax, 0x7f7f7f7f`

4) die höchsten Bits und die vorzeichenlosen Teil kombinieren `or eax, edx`

Wann ist SIMD sinnvoll ?

- Große Datenmengen
  - Bildbearbeitung
  - Matrixoperationen
- Gleiche oder ähnliche Instruktionen auf Elementen
- Daten müssen geschickt im Speicher liegen
  - z.B. Linked Lists schlechter als Arrays
- Keine anderen Funktionen in Schleife aufrufen
  - z.B. `a[i] = foo(i);`
- Auf loop carried dependeces aufpassen
  - z.B. `a[i] += a[i-1];`

SIMD mit AVX

AVX-Register: 256 Bit -> Platz für 8 single-precision (32 Bits) oder 4 double-precision (64 Bits) Gleitkommazahlen

Erweiterung der `xmm` Register: `ymm0` bis `ymm15` (256 Bits) beieinhalten `xmm0` bis `xmm15` (128 Bits) -- wie `rax` und `eax`

Erweiterte SSE-Instruktionen in AVX (mehr Flexibilität & V Präfix)

Drei-Operanden Format $c = a + b$

`VADDPS xmm/ymm, xmm/ymm, xmm/m128/ymm/m256` Vektoraddition von Gleitkommazahlen `Op1 = Op2 + Op3`

`VMOVSD xmm, xmm, xmm` Merged zwei 64 Bit Gleitkommazahlen in ein `xmm` Zielregister

`VBROADCASTSS xmm/ymm, xmm/m32` Kopiert die Gleitkommazahl an den unteren 32 Bit (Quelloperand) in alle 32 Bit Blöcke (Zielregister)

Shift-Instruktionen

`VPSLLVD xmm/ymm, xmm/ymm, xmm3/m128/ymm/m256` Shift Logical Left von 32 Bit Blöcken (vorzeichenlos)

`VPSRLVD xmm/ymm, xmm/ymm, xmm3/m128/ymm/m256` Shift Logical Right von 32 Bit Blöcken (vorzeichenlos)

`VPSRAVD xmm/ymm, xmm/ymm, xmm3/m128/ymm/m256` Shift Arithmetric Right von 32 Bit Blöcken (vorzeichenbehaftet)

Alignment mit AVX

Alignment nicht mehr verpflichtend für die meisten SSE-Instruktionen - Außer bei expliziter Forderung (z.B. `movapd`)

Trotzdem sollte man Alignment nutzen (falls möglich) -> Höhere Performance

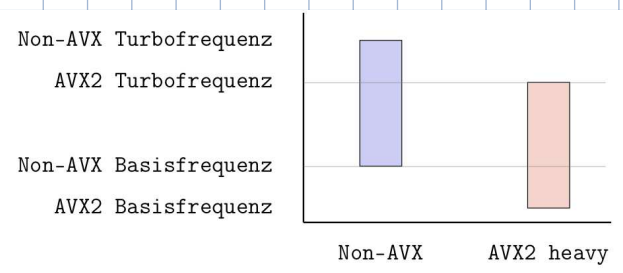
16-Byte Alignment für 128-Bit Daten

32-Byte Alignment für 256-Bit Daten

Verschiedene Instruktionsklasse

Non-AVX: Reguläre- und SSE-Instruktionen -> Normale Basis- und Turbofrequenz

AVX2 heavy: AVX-Instruktionen -> AVX2 Basis- und Turbofrequenz



Zeitmessung

Grundidee: Dauer `time` = Messpunkt `end` - Messpunkt `start`

Vermeidung unnötiger Operationen im Messbereich (z.B. I/O Funktionen `printf()` `scanf()`)

mindestens eine Sekunde Abstand zwischen Messpunkten (-> genügend Workload durch z.B. Schleifen)

mindestens Optimierungsstufe 2

mindestens drei Wiederholungen der Messung

Bereitstellung maximaler Ressourcen

- Eingabe mit `time cmd`, z.B. `time ./matr` oder `time ls -a`
- `time`-Ausgabe der Form:

```
real    0m.024s ⟵ Abstand zwischen call und finish
user    0m.016s ⟵ CPU-Zeit im User-Mode
sys     0m.016s ⟵ CPU-Zeit im Kernel-Mode
```

Umgebungsbedingungen

- Überprüfung auf Korrektheit
- mindestens drei Wiederholungen der Messung
- exklusive Nutzung der Ressourcen
- Messung auf echter Hardware ohne CPU-Features

Dokumentation

- Hintergrund: Reproduzierbarkeit der Ergebnisse
- Setup des Systems (CPU, Speicher, Betriebssystem)
- Kompiliervorgang (Compiler, Version, Optionen)
- Testvorgang (Eingaben, Wiederholungen)
- Aufbereitung der Ergebnisse in Diagramm

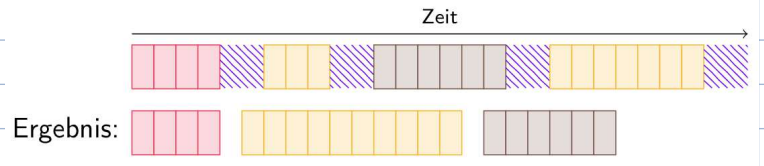
MethodenKopf

- `int clock_gettime(clockid_t clk_id, struct timespec *tp)`
  - `@return`: 0 falls erfolgreich, -1 sonst
  - `clk_id`: `clockid_t CLOCK_MONOTONIC`
  - `*tp`: `struct timespec{time_t tv_sec; long tv_nsec;}`
  - $\Longrightarrow$ Sekunden mit: `double sec = tv_sec + 1e-9 * tv_nsec`

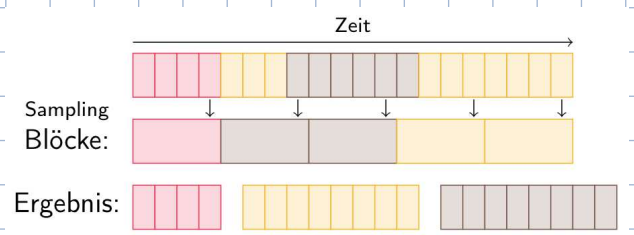
Beispiel-Implementierung für Zeitmessung

```
1   #include <stdio.h>
2   #include <time.h>
3
4   int main(int argc, char **argv){
5       struct timespec start;
6       clock_gettime(CLOCK_MONOTONIC, &start);
7       foo();
8       struct timespec end;
9       clock_gettime(CLOCK_MONOTONIC, &end);
10      double time = end.tv_sec - start.tv_sec + 1e-9 *
            (end.tv_nsec - start.tv_nsec);
11      printf("done after %f seconds", time);
12      return 0;
13  }
```

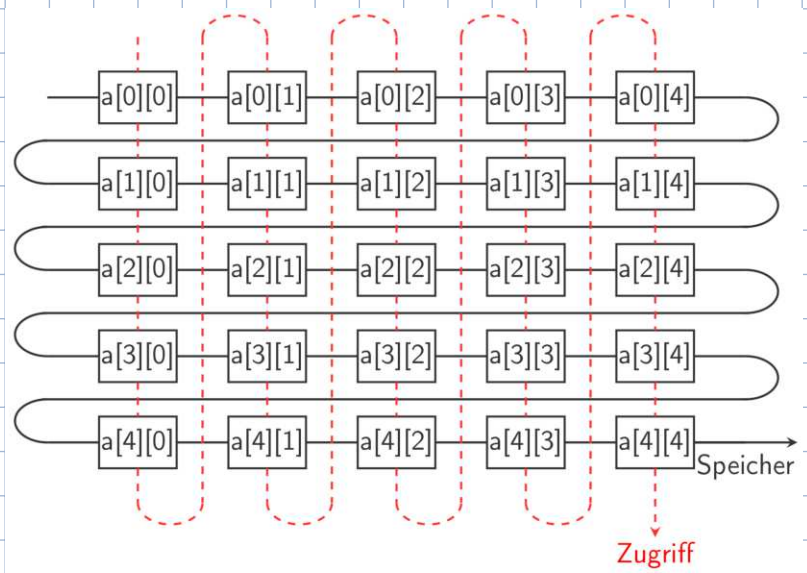
Tracing Profiling



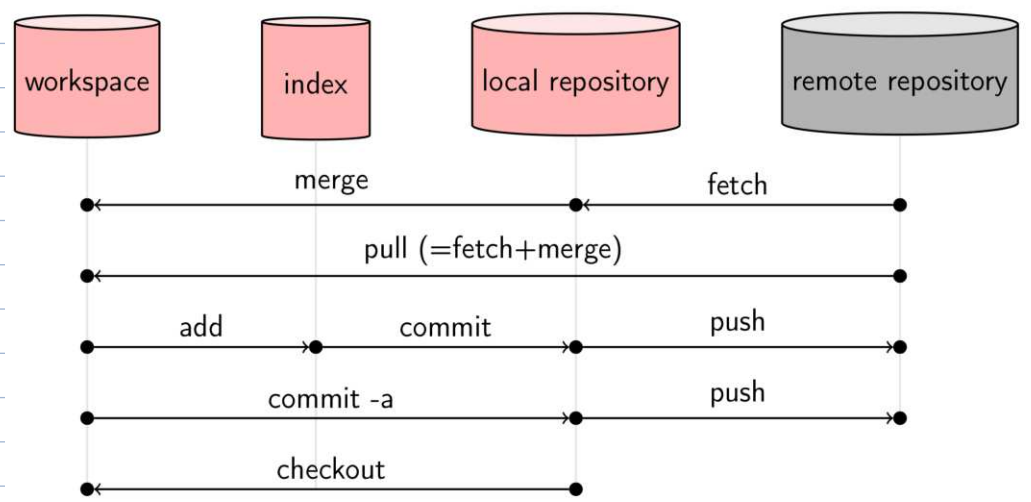
Sampling Profiling



Reihenfolge der Speicherzugriffe



Git (Remote) Repositories



Hinweise bei Git

Git ist ein verteiltes Versionsverwaltungssystem anstatt eines Backupsystems

Konflikte bei Branching möglich - werden nicht automatisch gelöst

Vor Beginn der Arbeit git pull ausführen

Branches regelmäßig mergen / nicht zu lange offen lassen

Immer eine funktionierende Version im master-branch

Aussagekräftige commit-messages schreiben

Übersicht bei Git

`git merge` -- Zusammenführen von zwei Branches auf dem lokalen Repository

`git add` -- Zufügen von Dateien und Verzeichnissen zum Index (staging area)

`git commit` -- Erzeugt eine neue Version im lokalen Repository

`git push` -- Änderungen im lokalen Repository werden auf den Server geladen

`git pull` (= `git fetch` + `git merge`) -- Änderungen vom Server holen + merge des aktuellen branches

`git checkout` -- zwischen Branches wechseln

`git branch` -- u.A. Erzeugen, Auflisten, Löschen von Branches