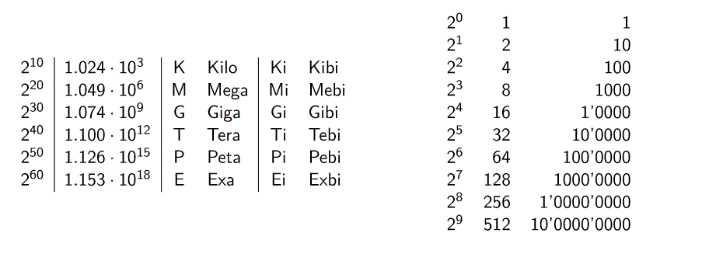
**Binär**

***Begriffe***

**Bit:** Eine Stelle einer Binärzahl  
**Bit = 1**: Gesetztes Bit(*set bit*)  
**Bit = 0:** Gelöschtes Bit(*cleared bit*)  
**LSB**: Least Significant Bit, niederwertigstes Bit, Bit 0  
**MSB**: Most Significant Bit, höchstwertiges Bit, Bit n-1  
**Nibble**: Binärzahl mit *vier* Bit.   
**Byte / Oktett**: Binärzahl mit *acht* Bit.

**Carry Bit**: Übertragsbit, wird bei einem Übertrag gesetzt  
***Formeln***

**Grösste Darstellbare Zahl**(unsigned):   
**Anzahl Zahlen**:   
**Bereich(unsigned)**: 0 bis

****Grösste Darstellbare Zahl**(signed):

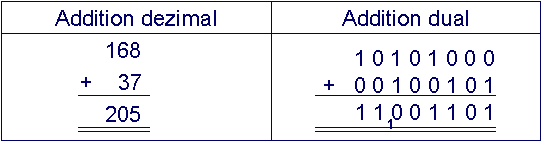
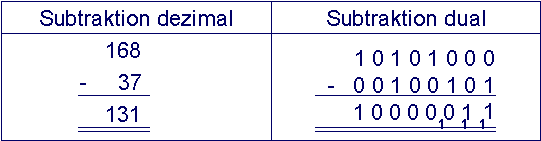
**Kleinste Negative Zahl**(signed):

**Bereich(signed)**: bis

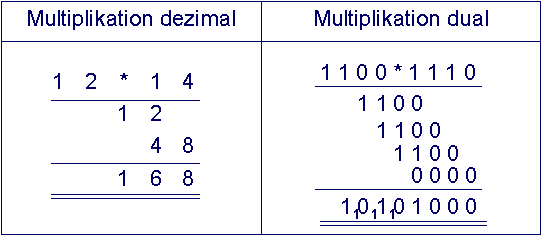
**MSB = 0**: Dient als +

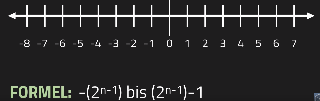
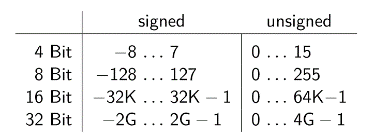
**MSB = 1**: Dient als -

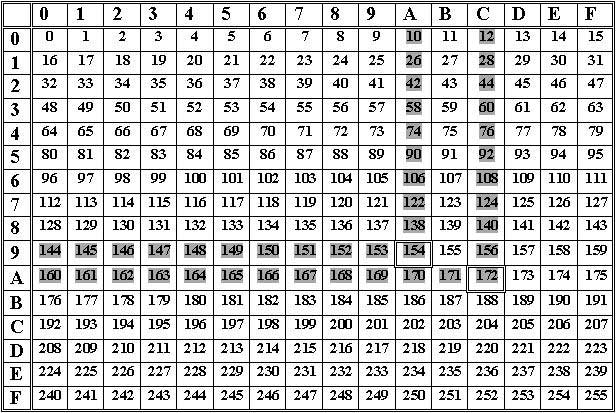
***Rechnen***

***Addition / Subtraktion:***

***Multiplikation:***

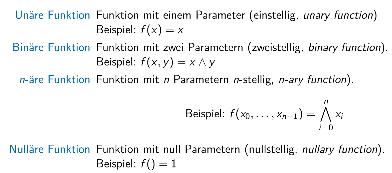
******

***Zweierkomplement(übliche Darstellung negativer Binärzahlen)***

**Hexadezimal(übliche kompakte Darstellung des Binärsystems)**

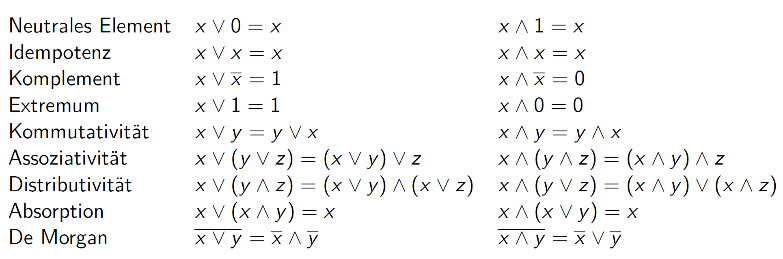
***Beispiel wie viele Bits benötigt für:***

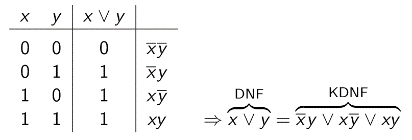
**0-32T -1** 🡪 2^5 \* 2^40 = 45 bit  
**0-32T🡪**46 bit  
**0-1012d** 🡪 40 bit (weil 2^39 = 0.550 \* 10^12 was nicht reicht)

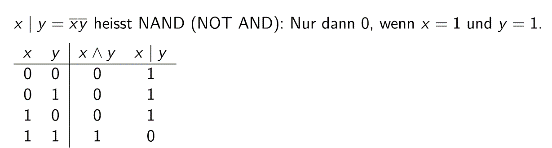
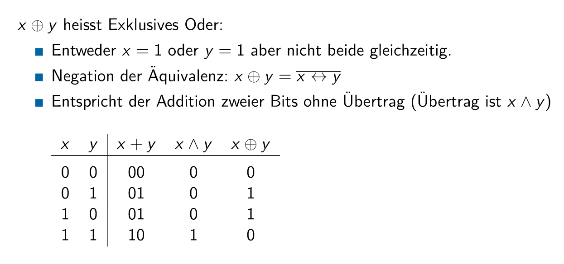
**Logik**

#### **Nulläre Funktion**

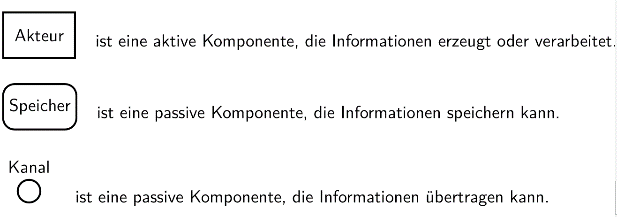
1 Möglichkeit: keine Argumente  
**Unäre Funktion**2 mögliche Argumente: x = 0 oder =1   
**Binäre Funktion**  
4 mögliche Argumente(Kartesisches Produkt): (0,0),(0,1),(1,0),(1,1)   
**n-äre Funktion**2n Argumentkombinationen.

***Regeln für Disjunktion und Konjunktion***

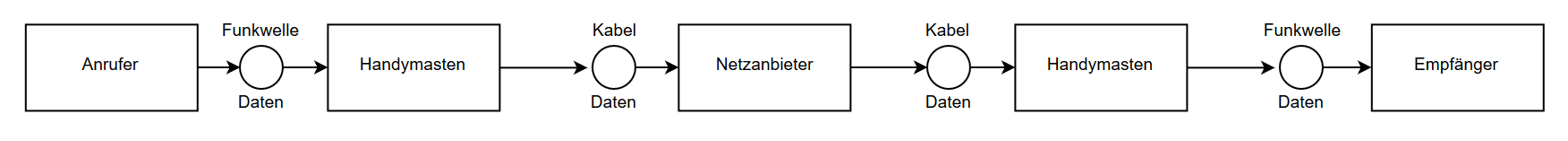
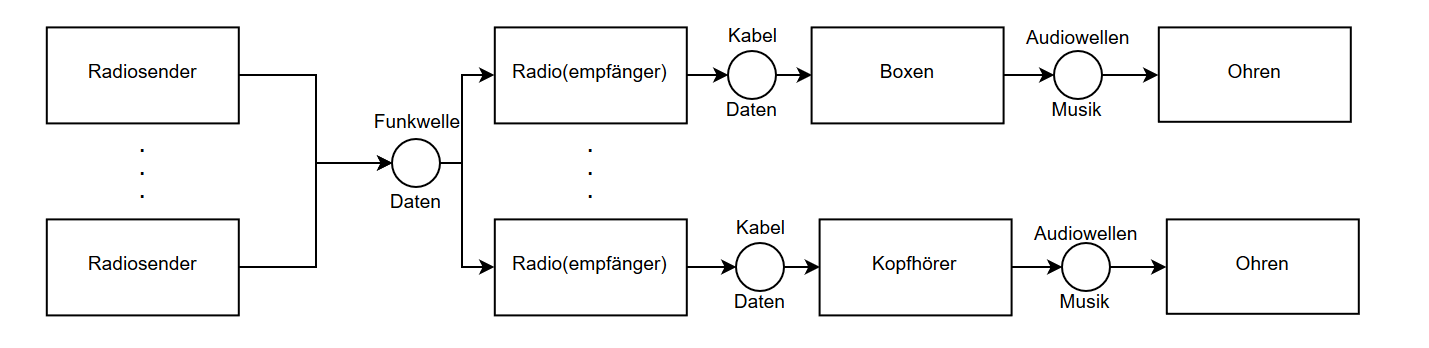
KDNF beinhaltet jedes Element 1x



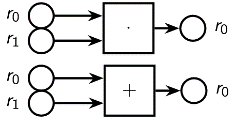
Pro logische Funktion Rückgabewert ein Bit 🡪 Viele Bit = Viele Funktionen

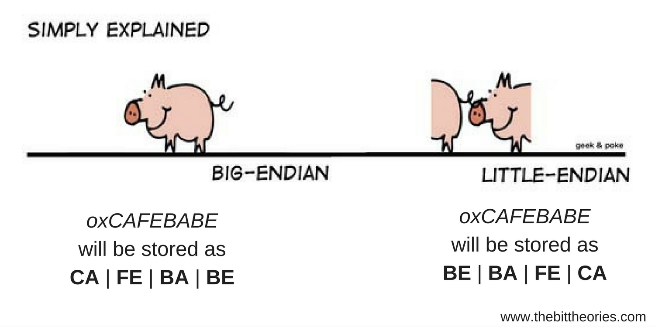
**Modellierung**

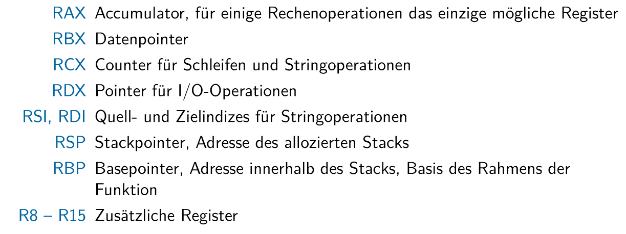
Merke: Nie folgt auf ein Element ein gleiches Element.

**Beispiel Modellierung**

Wie will man am Modell alle Anrufe in einem Netz abbilden? Es geht schlicht und einfach nicht!(Alle Modelle falsch, aber nützlich)

**Prozessor**

Register von Prozessor sind mit Bausteinen verknüpft, welche dieser ansprechen kann und somit rechnen kann.

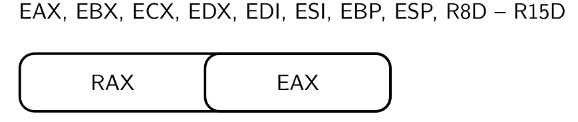
**Zyklus**

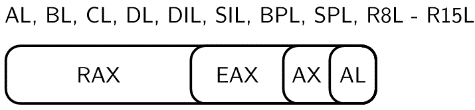
1. Prozessor wählt aktiven Baustein aus
2. Aktiver Baustein liest aus Registern
3. Aktiver Baustein führt Berechnung aus
4. Aktiver Baustein schreibt in Register

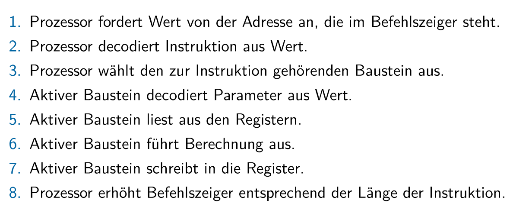
**Maschinencode**

Darstellung von Operationen als Zahlen

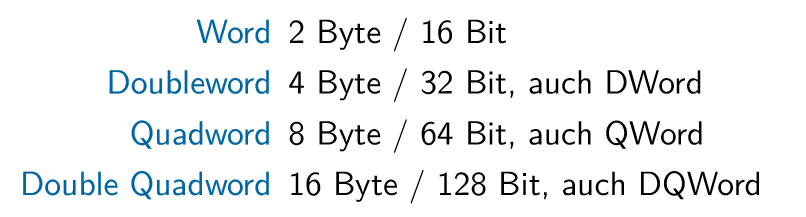
**Befehlssatz**

-Menge aller Maschinencodes, welcher ein Prozessor kennt.

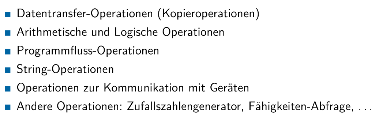
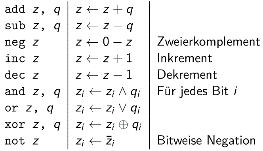
Bildschirmausschnitt-Menge aller Befehle ist von Hersteller zu Hersteller oder sogar von Prozessorfamilie zu Prozessorfamilie unterschiedlich, deshalb ist es enorm wichtig, dass man weiss, mit welchem Prozessor man arbeitet/arbeiten will.  
- Maschinencodes können je nach Befehlssatz gleich oder unterschiedlich lang sein

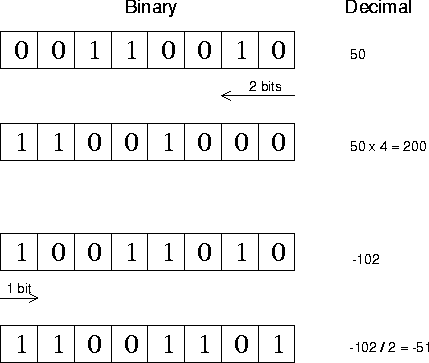
Prozessor-Zyklus

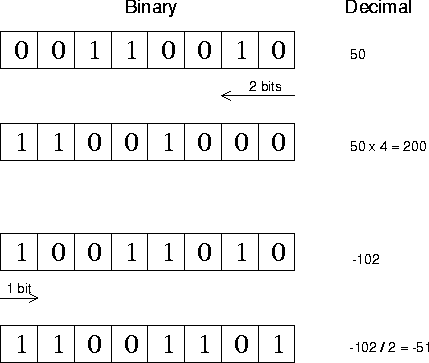
**Assembler**

* Assembler ein Programm, welches textuelle Befehle in Maschinencode übersetzt.
* Die Assemblersprache(assembly) ist die entsprechende Sprache
* Konvention für assembly hängt von Hersteller des Assemblers ab

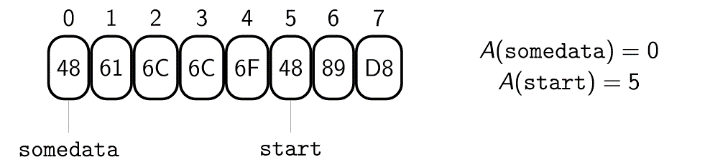
Merke: Beim Schreiben der unteren 32-bit(EAX, EBX, etc.) werden die oberen 32-bit auf 0 gesetzt.   
Werden Änderungen im ersten 16-bit Block vorgenommen( bsp AX/AL) ändert das die oberen bits nicht.

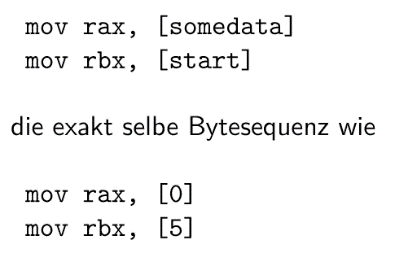
***Operationen***

**Linksshift Rechtshift**



**Label**

Am Anfang jeder Zeile kann ein sogenanntes Label stehen.

Das Label wird nicht in Bytecode übersetzt und nicht ausgegeben.

Intern assoziiert der Assembler Speicher für das Label.

Ab dann ersetzt der Assembler alle Verwendungen mit dem Offset des Labels(erstes Adresse des reservierten Speicherbereichs)

**Sprünge**

Ändern des RIP(RIP + x). JMP, JNZ, usw. Wir springen in Abhängigkeit von Flags!

|  |  |
| --- | --- |
| cmp a, b jg label ; Jump if A > B | cmp a, b jl label ; Jump if A < B |
| Cmp a,b  Jnz label; Jump if A ≠ |  |

**Flags**

**CF(Carry Flag) 🡪** 1 wenn Add./Sub. Übertrag erfordert(unsigned)

**OF(Overflow Flag) 🡪** Gibt Überlauf bei signed Operatioen

CF und OF immer gesetzt, da Prozessor k.P ob signed/unsigned

**ZF(Zero Flag) 🡪** 1 wenn Resultat 0

**SF(Sign Flag)** 🡪 SF = MSB vom Resultat

**PF(Parity Flag) 🡪** Gesetzt wenn unterstes Byte gerade Anzahl von gesetzten Bits(Einsen).

**Vergleich (Bei signed SF ≠ OF)**

BildschirmausschnittGrundoperation a -b a = b 🡪 ZF = 1   
a<b 🡪 CF = 1 a>b ZF = 0 & CF = 0

**CMP**

**Vergleichsoperation** 🡪 Setzt nur Flags ohne Subtraktion

**Programm Start**

Das Label \_start repräsentiert bei uns der EntryPoint. Im Executable steht eine Adresse(für start) ab wo das Programm anfängt. Wieso nicht erstes Byte? Weil dort evtl. Data/Befehle

**Programm Ende**Das Programm muss sagen, wann es beendet ist. Es macht den Systemcall exit(auf Linux rax = 60, rdi = 0 dann syscall)

**Privilege Levels**

0 – 3: 0 Alle Befehle, Prg 3 🡪 Programme «keinen» Schaden

**OS-Handler**

Handler für Systemaufrufe. Was in Rax, welche Argumente in anderen Register(bsp rdi, rsi). Adresse des Handlers wird bei Bootvorgang in IA32\_LSTAR geschrieben(nur bei PL 0 veräderbar)

**System Aufruf**

1. Programm Funktionscode in RAX(Argumente in andere Register)
2. Programm SYSCALL
3. RIP 🡪 RCX, RFLAGS 🡪 R11, PL 0, RIP🡪IA31\_LSTAR
4. Ausführung OS-Handler(Syscall)

Beachte! Rückgabewert exit(RDI) ist nur unterstes Byte  
Beispiel mov rdi, 300 🡪 echo $? 🡪 44 weil 300-256 = 44

|  |  |
| --- | --- |
| **Beispiel Hello World**  bits 64  msg: db 'Hallo World!'  len: equ $-msg  global \_start  \_start:  mov rax, 1 ; sys\_write  mov rdi, 1 ; stdout  mov rsi, msg ;address of the String  mov rdx, len ; length of the String  syscall ;system call--> OS do what  mov rax, 60 ;sys\_exit  mov rdi, 0 ; with code 0  syscall | **Beispiel Summe 0 + 1 … + n**  bits 64  global \_start  \_start:  mov rdi, 6  mov rcx, 0  \_sum:  add rcx, rdi  dec rdi  jnz \_sum  mov rax, 60  mov rdi,rcx  syscall |
| **Beispiel Anzahl Operationen bis gleichgross**  bits 64  global \_start  \_start:  mov rdi, 20  mov rcx, 8  mov rbx, 0  \_operate:  dec rdi  inc rcx  inc rbx  cmp rdi, rcx  jg \_operate  mov rax, 60  mov rdi,rbx  syscall |  |

**Stack**

Stapel von 8 Byte grossen Elemente, der zur Laufzeit Arraygrösse ändert.

**Push/ Pop**

Push legt ein 8-Byte-Wert oben auf den Stapel.

Mit pop kann man einen Wert von Stack nehmen.

Das oberste Element liegt an der niedrigsten Adresse!(Wächst von hohen zu niedrigen Adressen)

**RSP**

Der Stack Pointer(RSP) zeigt auf das aktuelle Element des Stacks. Wird nun Push/Pop ausgeführt, wird dieser RSP verändert.

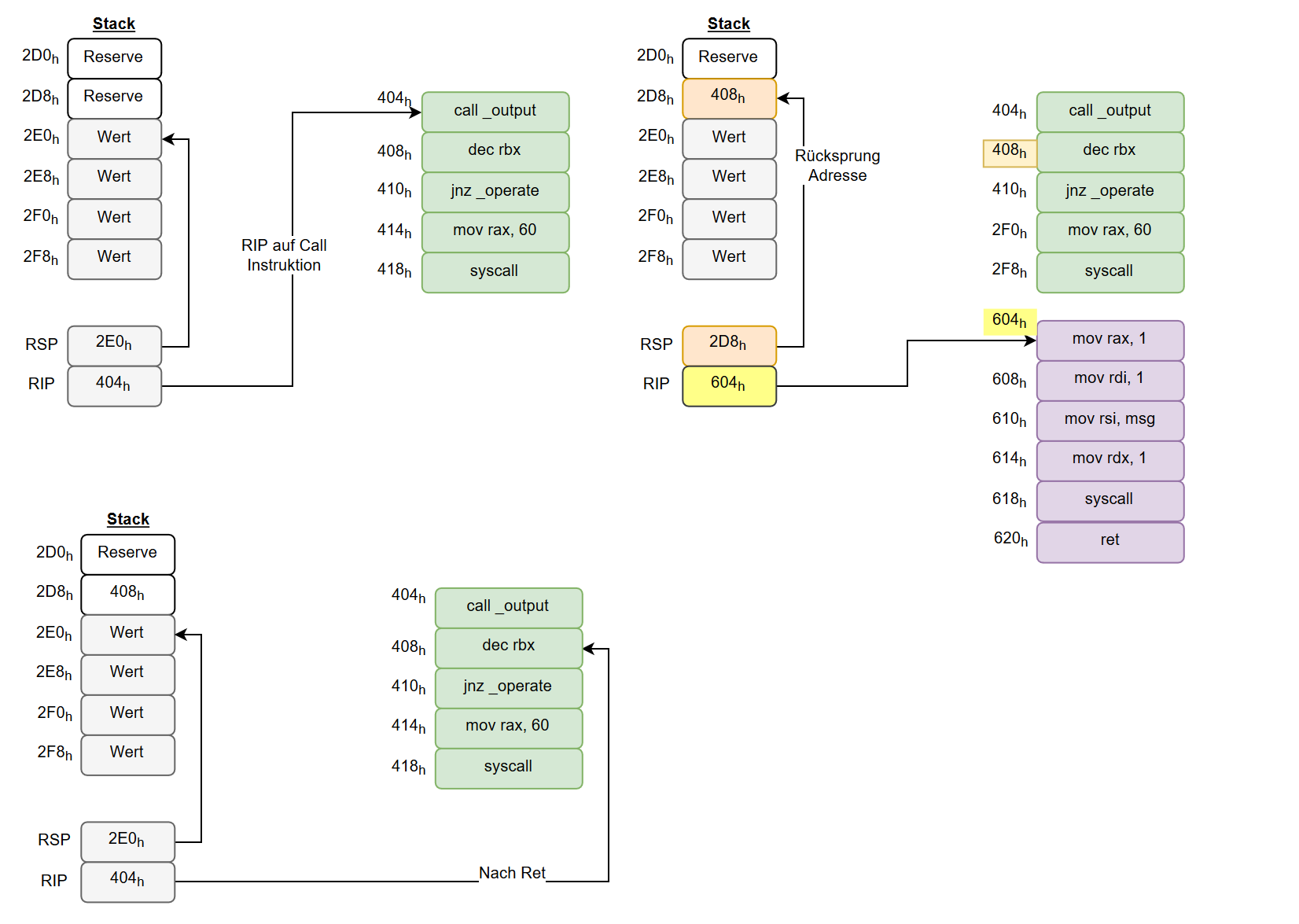
Push rax = sub rsp, 8 & mov [rsp], rax

Pop rax, mov rax, [rsp] & add rsp, 8 // erhöhen 🡪Stack kleiner

**Call**

Call Speichert die Rückspring-Adresse auf dem Stack und Springt dann. Das bedeutet die Adresse der nächsten Instruktion wird auf dem Stack abgelegt und dann wird «die Funktion» aufgerufen.

**Ret**

Die Gegenteilige Funktion von Call heisst Ret(return). Diese kann man in der «aufgerufenen Funktion» aufrufen. Dann springt man zur Adresse zurück, welche man vorhin mit Call auf den Stack gepusht hat. Dann verkleinert man den RSP wieder.

|  |  |
| --- | --- |
| **Beispiel Print Stars**  bits 64  global \_start  msg: db '\*'  \_start:  mov rbx, 6  \_operate:  call \_output  dec rbx  jnz \_operate  mov rax, 60  mov rdi,0  syscall | \_output:  mov rax, 1 ; sys\_write  mov rdi, 1 ; stdout  mov rsi, msg ;address of the String  mov rdx, 1 ; length of the String  syscall  ret ;Back to \_operate |

**Calling Convention**

Da Call keine Parameter oder Rückgabewerte hat und keine Register, Stack oder Flags sichert braucht es Vereinbarungen.

Calling Convention sind Vereinbarungen zwichen Aufrufen(Caller) und aufgerufener Funktion(Callee): Wo Argumente, Wo Rückgabewerte, welche Register bearbeitet, etc.

Von OS zu OS, Compiler zu Compiler verschieden  
Hauptgrund warum verschiedene Compiler/OS inkompatibel

Performance des Gesamtsystems abhängig wegen Tradeoff

Mehr Register -> weniger Register für Rechnungen

Mehr Stack -> mehr langsame Speicherzugriffe Kein Optimum

**Lokale Variabeln / Globale Variabeln**

**Lokle Variabeln**: liegen auf dem Stack

**Globale Variabeln**: fixe Adresse im Speicher(Label)

|  |  |
| --- | --- |
| **Beispiel Wörter Ausgabe**  bits 64  global \_start  anzahl: dq 3  strings: db 'String1', 'plz ', 'Dellsperger'  len: dq 7, 4, 11  \_start:  mov r8, strings ;erste Adresse von s  mov r9, 0 ;String counter | \_iterieren:  call \_ausgabe  inc r9  cmp r9, [anzahl]  jl \_iterieren  mov rax, 60  mov rdi, 0  syscall  \_ausgabe:  mov rax, 1  mov rdi, 1  mov rsi, r8  mov rdx, [len + r9 \* 8]  add r8, rdx  syscall  ret |

**Die Sprache C**

Weniger Aufwand, Register nicht selber verwalten, einfach Schleifen, Verzeigungen, Sprünge, Nachvollzierbarkeit, keine Unterscheidung signed oder unsigned🡪 Evolution Assembler

**Toolchain(logisch gesehen)**

**Assembler:** Macht aus (mehreren) Assembler Dateien (mehrere) Objekt Dateien.

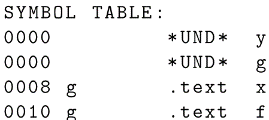
**Linker:** Macht aus mehren Objekt-Dateien eine Executable.

**Präprozessor:**  Entfernt Kommentare, Tokenization, Direktiven

**Compiler**: Kompiliert reinen C-Code in Assembler Dateien

**Objekt Dateien(Binärrepresentation einer Assembler-Datei)**

Enthält Binärsequenzen für Daten und Maschinenbefehle

**Bildschirmausschnitt**Enthält Symbole(Labels) in der Symboltabelle

Bei externen Labels stehen Nullen

Symboltabelle zusammengewürfelt aus verschiedenen Objektdateien

**Executable(Ausführbare Datei)**

Linken weist jedem Symbol eindeutige Adresse zu. Er erzeugt aus mehreren Objekt-Dateien eine einzige Binärsequenz, die Executable. Dann nicht mehr unterscheidbar, aus welcher Objekt-Datei welches Symbol.

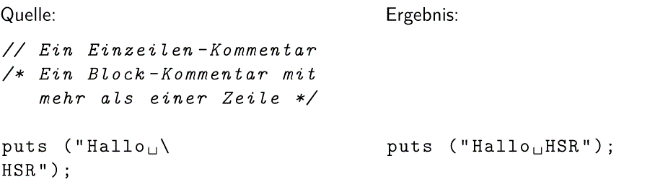
**C Sprachebenen**

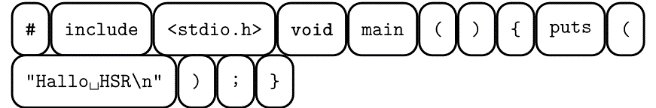
Präprozessor 🡪 Bearbeitet includes und defines

Basiskonstrukte 🡪 Grundgerüst vom Programm, Variabeln, Schleifen, etc

Standardbibliotheken 🡪 stellen Funktionen und Typen zur Verfügung, welche Basis-Funktionalität beinhalten.

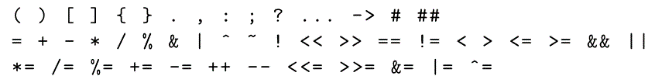
***Präprozessor***

**1.Durchlauf:** Entfernen von Kommentaren und Zeilenumbrüchen bei fortgesetzten Zeilen:

**2.Durchlauf: Zusammenführen Header + Tokenbildung**: Leerschlag, Zeilenumbruch, Tabulatorem trennen Token.

**BildschirmausschnittBezeichner**: Beginnt mit Buchstaben oder \_ gefolgt von Buchstaben, \_ oder Ziffern.   
**Zahl**: Beginnt mit Ziffer(oder Punkt Ziffer) gefolt von Ziffern, Buchstaben, \_ oder . oder Exponenten.

Bildschirmausschnitt

**BildschirmausschnittString und Character**: alles zwischen «» und ‘ ’   
**Punctuator**(achtung greedy 🡪 grösstmögliche Token):

**3.Durchlauf** Direktiven bearbeiten(include, define, if, else, endif)

**#include** 🡪 Kopiert tokenisierten Inhalt von Include-Datei

Präprozessor macht aus mehreren Dateien eine Datei(Translation Unit) danach geht diese abgearbeitete Datei zum Compiler

**#define** 🡪 Ersetzt jeden passenden Token mit neuem Wert

Der Präprozerssor arbeitet ein define nach dem anderen ab, bis alle durch sind: BSP. #define XYZ 123, #define 123 700

1. 🡪 int y = XYZ; 🡪 int y = 123;
2. 🡪 int y = 123 🡪 int y = 700

Nach Ersetzungen geht er nochmals defines von oben durch

|  |  |
| --- | --- |
| datei.c  #include "x.h"  #define z w  int y = x; | x.h  #define x z |

Lösung y = w 🡪 Zuerst y=z 🡪 dann y = w

***Globale Variabeln(Typ + Bezeichner [Optional initalisiert])***

**Werden Exportiert🡪** wie globale Elemente in Assembler

**Extern 🡪** Compiler nimmt an, dass Variable in anderer Datei ist

**Ohne Extern:** Variabel wird in Objekt-Datei erstellt

***Objekte & Wert***

**Objekt:** zusammenhängender Speicherbereich

**Wert**: Bedeutung des Inhalts eines Objektes

Jede globale Variabel Objekt, aber nicht jedes Objekt Variabel

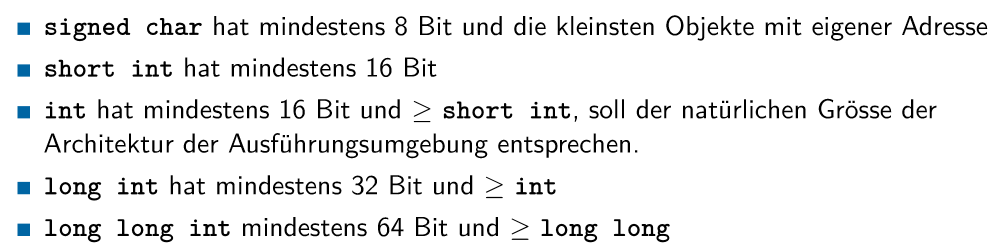
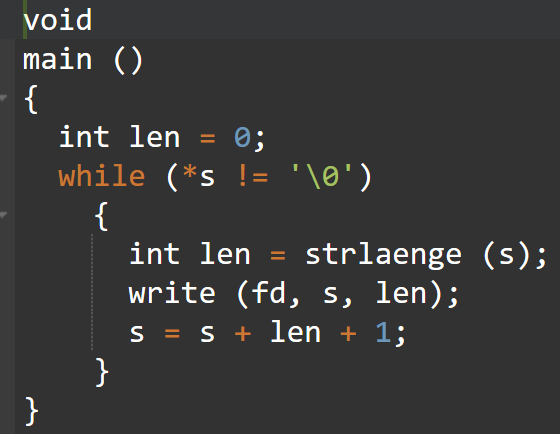
***Typen***

Basistypen(int, float), Abgeleitete Typen(arrays, etc), ENUMS

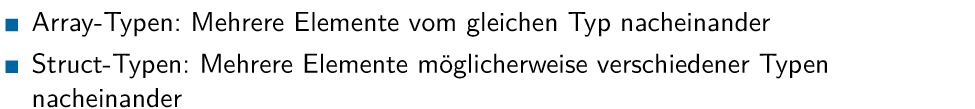
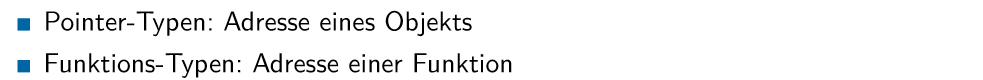
Typen beschreiben Objekt bzw. Eigenschaften, bsp sizeof(T)

Vorteil: auto. Grösse, richtige Operationen(signed / unsigned), Fehler melden(beispiel Typen nicht kompatibel)

Auf Maschinencode-Ebene nur Adressen, keine Variabeln

*********Basistypen***

Ebenfalls unsigned char, unsigned bool und void

***Abgeleitete Typen***

***Pointer***

Adresse, welche auf ein Objekt zeigt.

Definition: void\* oder int\* oder auch int\*\*

***Ausdrücke***

Ausdrücke sind Kombination von Operanden und Operatoren

Besteht aus verschiedenen Teil ausdrücken und hat einen Typ

Jeder Ausdruck ausser void hat einen Wert

***Conditionals(if)***

Ausdrücke, welche interpretiert werden, dass 0 falsch ist und jeder andere Wert wahr.

***Referenz- und Dereferenzoperator***

Der Operator «&» erzeugt die Adresse eines Audruckes

Pointer(\*)werden jeweils Adressen(&) zugewiesen, sprich der eigentliche Wert von T\* ist &a

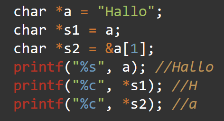
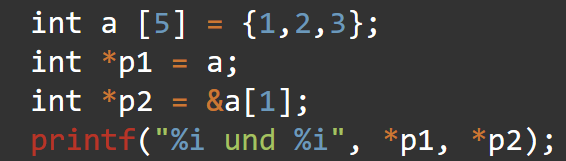
\*& heben sich gegeneinander auf» 🡪 \*&a = a = &\*a

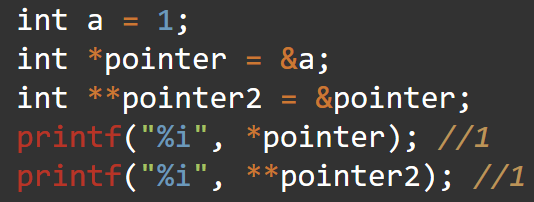
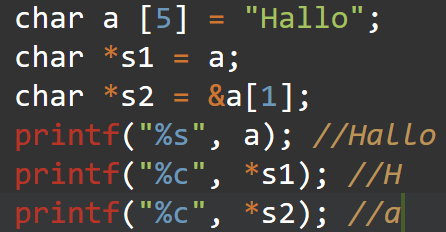
Achtung bei &\*a muss a Pointer sein, weil Adresse verlangt

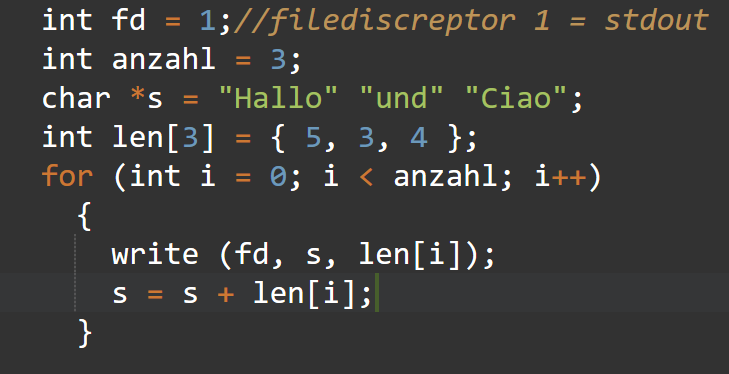
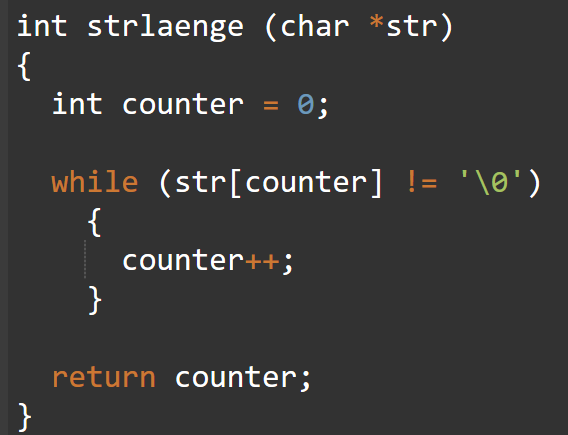
***Arrays***

Int b [5] [= {0,1,2}], verbleibende Elemente mit 0 initialisiert

Wird ein Array mit einem Bezeichner angesprochen, ist immer die erste Adresse des Arrays gemeint.

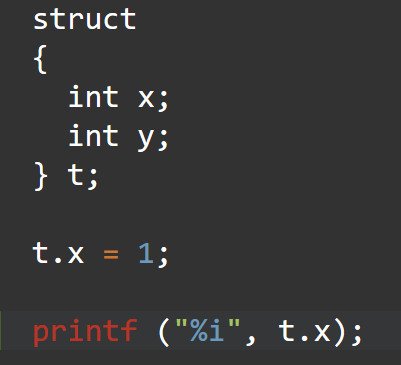
* Es können auch Pointer verwendet werden:





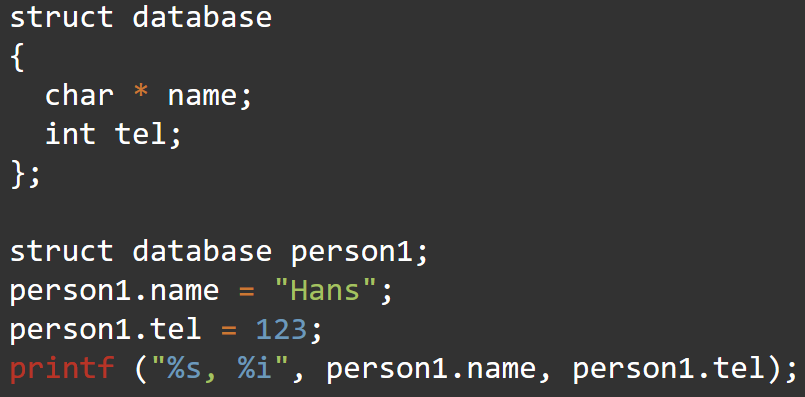
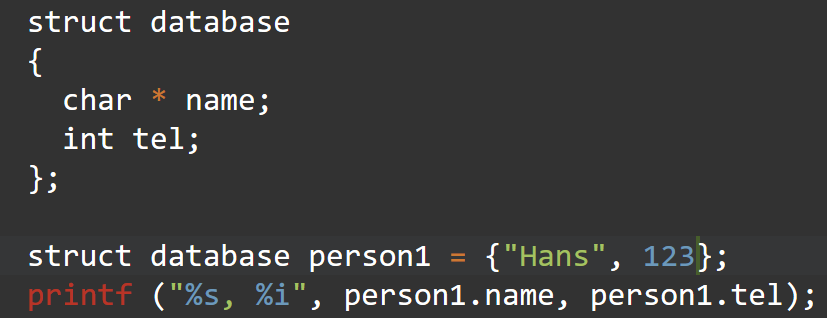
Oder Funktion ausgelagert.

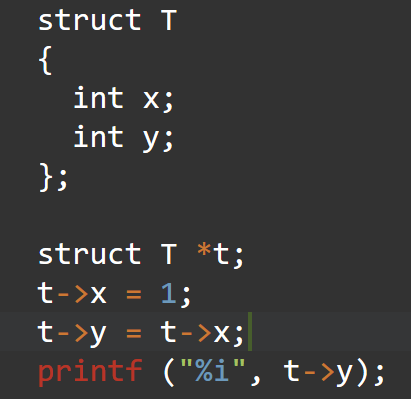
***Strukturierte Variabeln / Structs***

Eine Strukturierte Variabel beinhaltet verschiedene Member, welche eizeln angesprochen werden können:

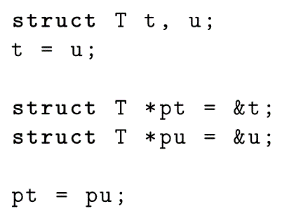
Ein Struct braucht gleich viel Speicher, wie beide einzelnen Variabeln.

Jedes Member hat eigene Adresse.

Structs können auch Variabeln sein.

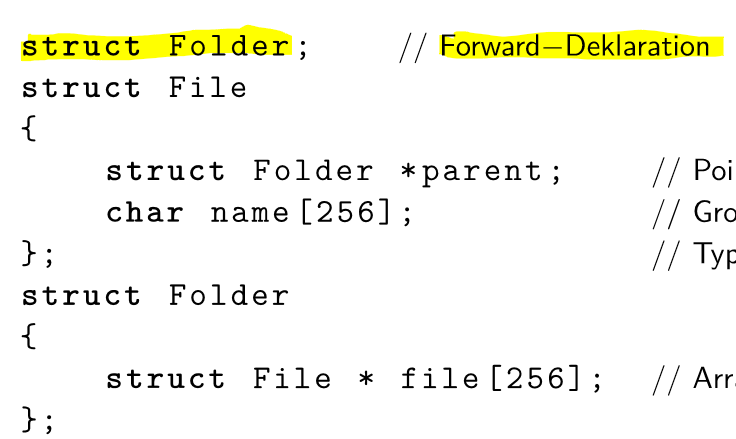
Pointer auf Struct:

Mit dem «->» Operator können auf die Adressen der Member zugegriffen werden.

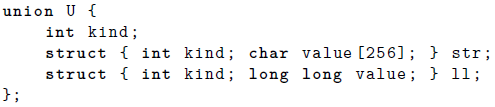
Memberwise Copy

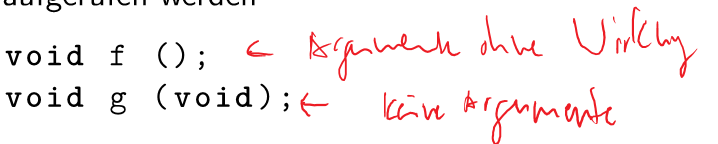
Complete / Incomplete

Vollständig(complete) wenn Compiler genug Infos, um Grösse dieses Typs zu bestimmen, andernfalls unvollständig

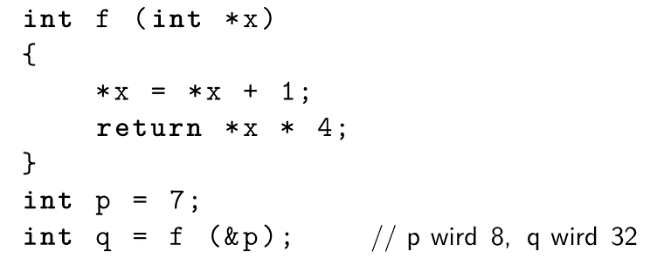
Forward-Deklarationen

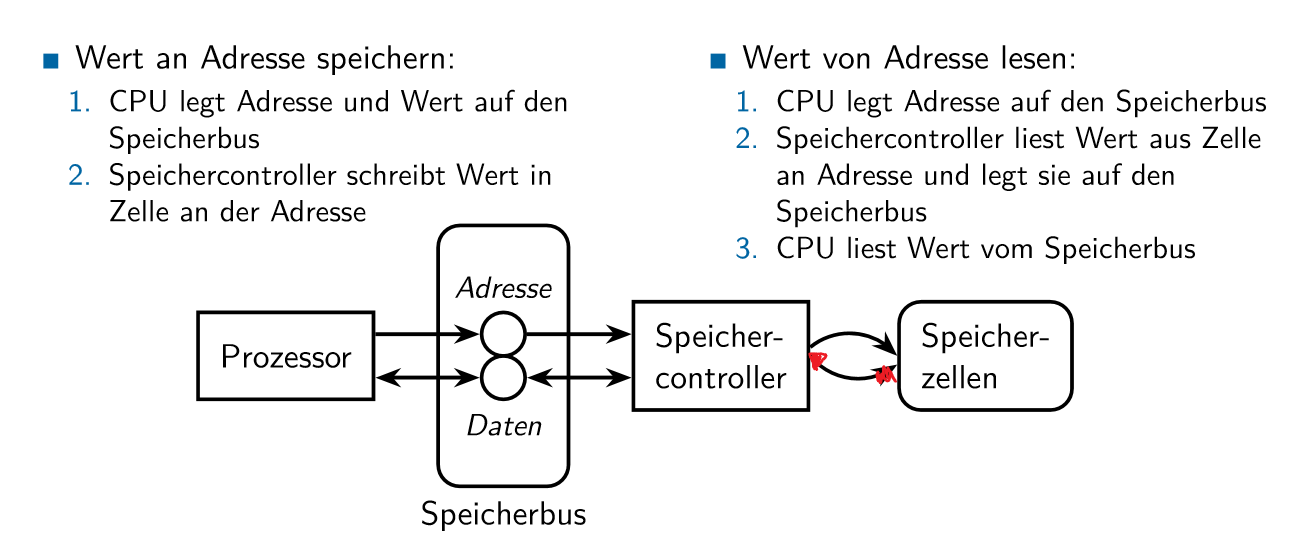
***Unions***

Unions werden in Structs definiert. Ihre Member beginnen alle mit Adresse der Union. In einer Union hat es mehrere Structs.  
Mit Unions lassen sich Casts vermeiden  


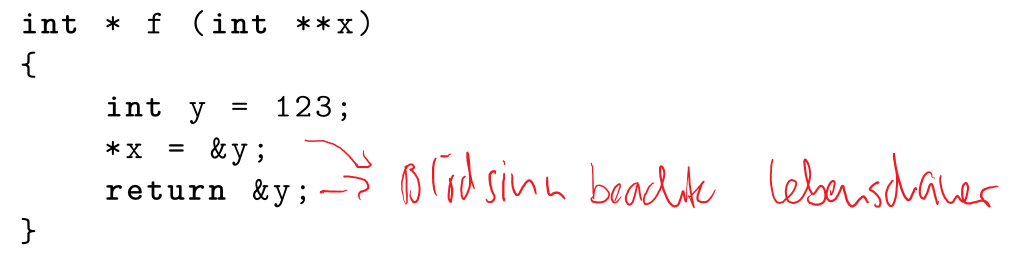
***Funktionen(Call by Value)***



Die Main Funktion hat ein Argument Counter und ein Char-Array von Argumenten, welche der Funktion mitgegeben werden können. Da ein Array immer nur als Pointer übergeben werden kann, muss die Anzahl(argc) einzeln übergeben werden.

1. OS übergibt Programm Kommandozeilenargumente
2. Argv ist ein Pointer auf das erste Element eines Arrays von String Pointern
3. Agrv[0] beinhaltet den Pointer auf den Programmnamen
4. Agrv[1] – [argc-1] enthlt Pointer auf Argumente

***Lokale Variabeln***

Liegen auf dem Stack, innerhalb von Funktion definiert.

***Printf***

%s 🡪 String %i 🡪 int

%p 🡪 pointer %c 🡪 Char

**Speicher**

***Grundsatz***

Es gibt kein System, das optimal für alle Anwendungszwecke eingesetzt werden kann. Jede Optimierung geschieht auf einen oder mehrere Anwendungsfällem hin. Ein perfektes System gibt es nicht!

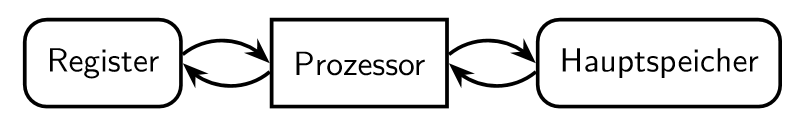
***Software-Basis-Modell***

CPU lädt Befehlssequenzen aus Hauptspeicher und führt die aus

Ein Befehl kann, Daten zwischen Hauptspeicher und Registern verschieben, Auf Registern rechnen oder den Befehlspointer verändern

Compiler generiert Befehlssequenzen, CPU führt diese aus

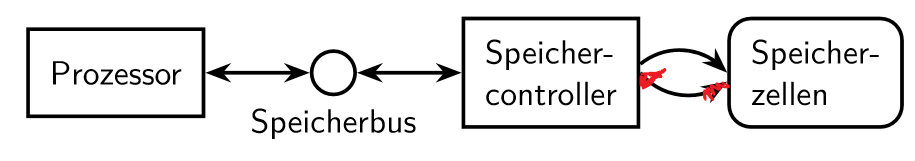
***Speicher-Programmier-Modell***

Früher einfachen Aufbau(weil Proz gleichschnell wie Hauptspeicher):

Aus diesem einfachen Prinzip haben sich Compiler und OS gebildet. Compiler🡪 Generiere Code, um Werte zwischen Hauptspeicher und Prozessort zu Transportieren

OS: Weise den Programmen Speicher(Bereiche) zu

***Hauptspeicher(RAM / ROM)***

******Speicher mit Speicherzellen, welche Wert speichern können. Zugriff über Speichercontroller, Kommunikation zwischen Speichercontroller und Prozessor geht über den Speicherbus

***Speicherhiearchie***

Moderne Computer brauchen aus physikalischen, logischen und ökonomischen Gründen eine Speicherhierarchie, in der verschiedene Sorten Speicher eingesetzt ist.

Physikalisch: Licht, Hitze(mehr Transistoren desto heisser)

Logisch(Adressierung ist logarithmisch, mehr = mehr Zeit)

Ökonomisch: Grundsatz desto CPUnäher Speicher desto teurer, komplexer, schneller, weniger fehleranfällig

* Flüchtiger Speicher(ohne Strom weg)
  + CPU-Register
  + CPU Cache(Level)
  + Hauptspeicher(RAM / ROM)
* Nichtflüchtiger Speicher(behält ohne Strom)
  + Festplatte(HDD/SSD)
  + Externe Datenträger(CD, Flashdrive, etc)
  + Netzwerkspeicher/Cloudspeicher
  + Archiv/Backup(Tape)

Vorteile höherwertiger Speicher

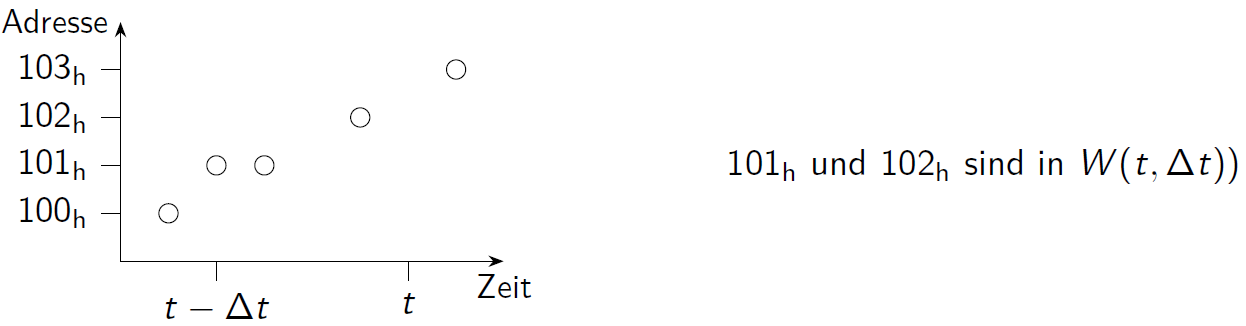
* Näher an CPU
* Höhere Zugriffs und Transfergeschwindigkeit
* Weniger Fehler(anfälligkeit)
* Geringere Sicherheitsanforderungen(bei Zugriff verloren)

Vorteile niederwertiger Speicher

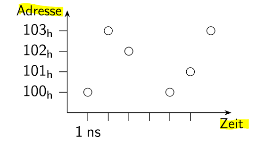
* Niederiger Preis pro Byte
* Höhere Kapazität
* Grössere Transfereinheiten

***Lokalitätsprinzip***

Arbeitsbereich eines Programmes(Zeitraum Delta t vor Zeitpunkt t):

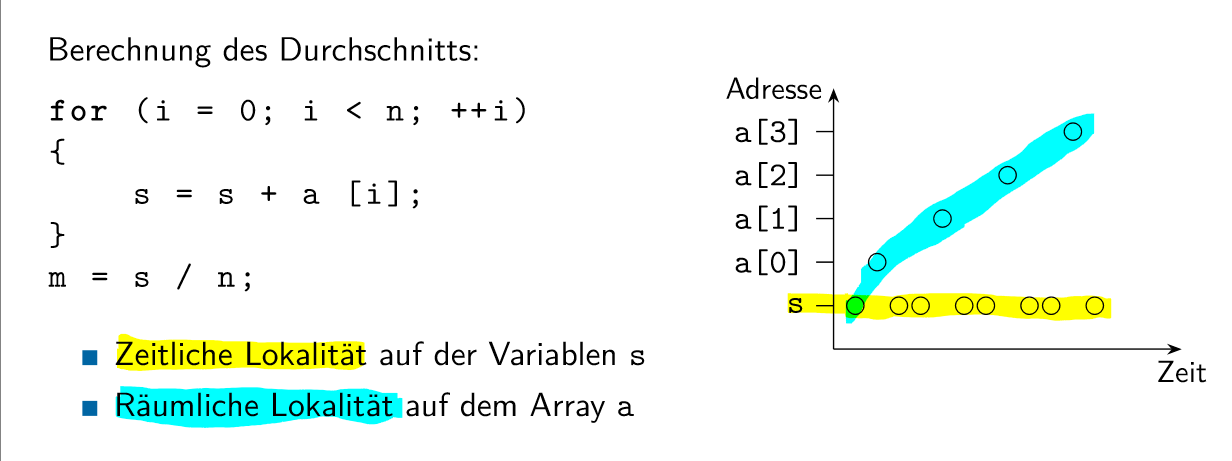


Lokalitätsdiagramm

Im Lokalitätsdiagramm werden die Zugriffe aus Speicherstellen im zeitlichen verlauf aufgezeigt.

**Räumliche Lokalität**: Wird auf eine bestimmte Adressse im Hauptspeicher zugegriffen, so ist die Wahrscheinlichkeit recht hoch, dass der folgende Zugriffe auf eine Adresse in der Nachbarschaft erfolgt. Im Speicher wird dies genutzt in dem man immer Datenblöcke verschiebt.

**Zeitliche Lokalität**: Wird auf eine bestimmte Adresse im Hauptspeicher zugegriffen, so ist die Wahrscheinlichkeit hoch, dass in naher Ukunft wieder darauf zugegriffen wird. Im Speichersystem will man also die zuletzt zugegriffenen Daten auf der schnellsten Stufe der Hierarchie halten.

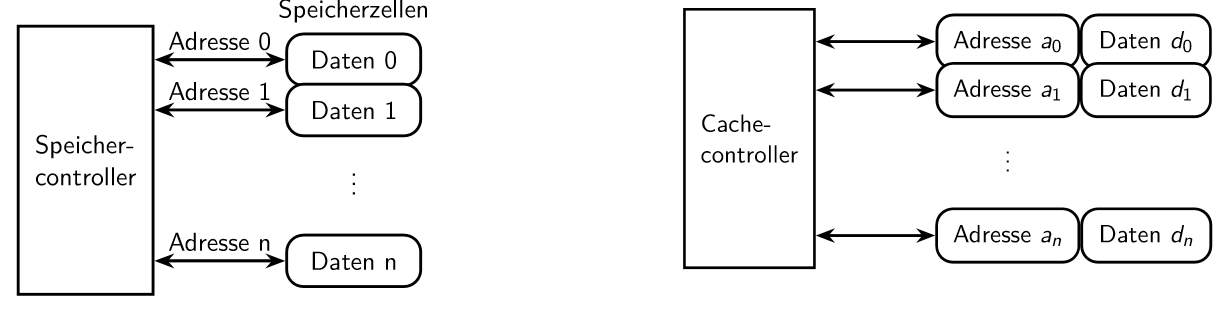
* Wenn ich den Arbeitsbereich kenne, kann ich auf zukünftigen Arbeisbereich schliessen
* Ohne dieses Prinzip, bräuchte man nur die langsamste Speicherstufe

***Cache***

Zwischenspeicher, welcher kleiner als Hautpspeicher ist und näher an CPU🡪schnellere Zugriffszeiten.

Neben Daten auch Teil der Adresse der Zellen speichern

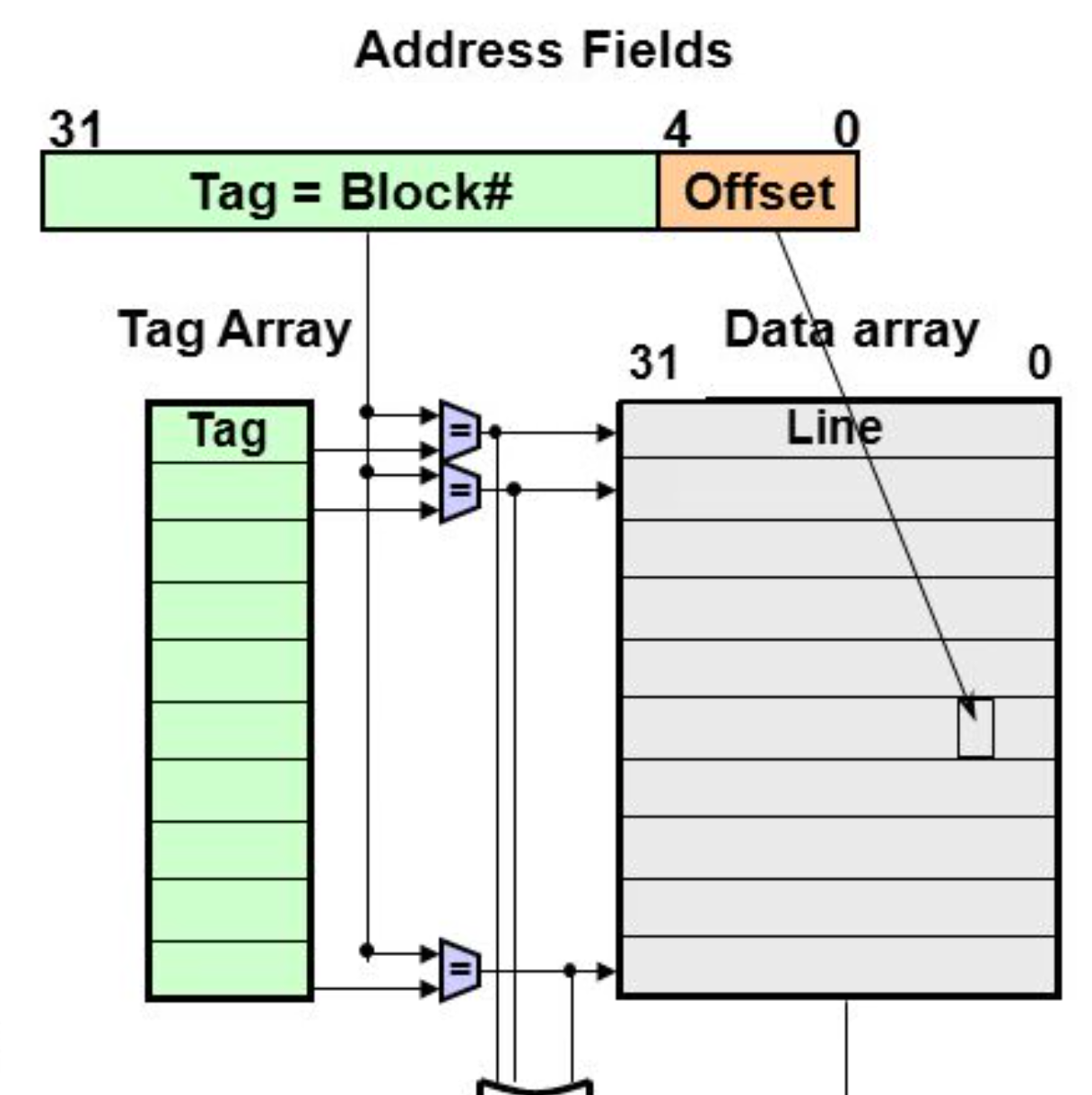
Lokälitätsprinzip🡪 Daten in Cache(schnelle, viele Zugriffe)

Cache zwischen Prozessor und Hauptspeicher: Ohne Sie wären Computersysteme praktisch unbrauchbar(weil langsam).

* Cache aufwendig und teuer

Cache Hit: Gesuchte Adresse ist im Cache.  
Chache Miss: Gesuchte Adresse ist nicht im Cache.  
Tc Zugriffszeit auf den Cache.  
Tm Zugriffszeit auf den Hauptspeicher.  
pc Wahrscheinlichkeit eines Cache Hits.  
Mittlere Zugriffszeit: Erwartungswert E(T) der Zugriffszeit T:  


***Cachearten***

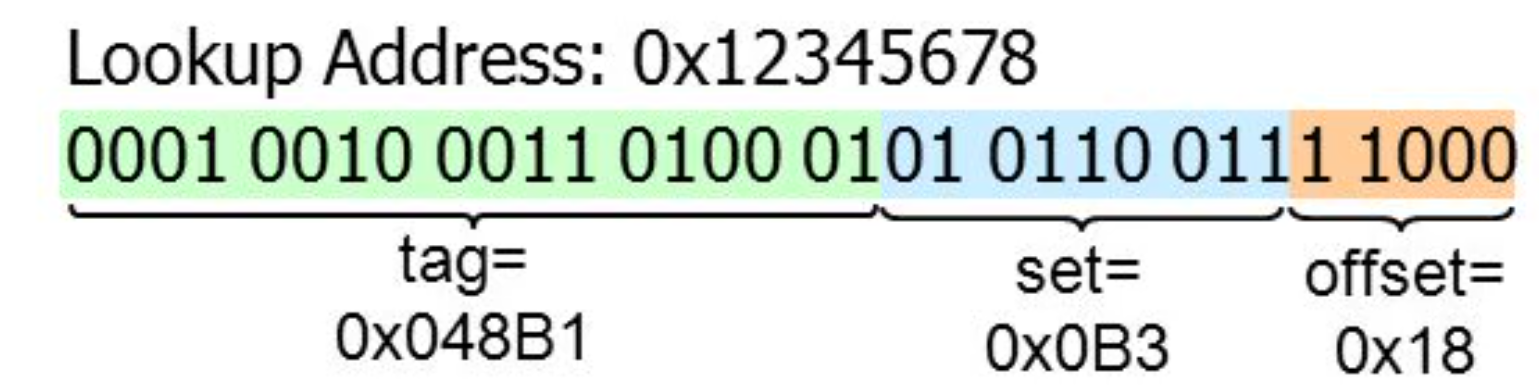
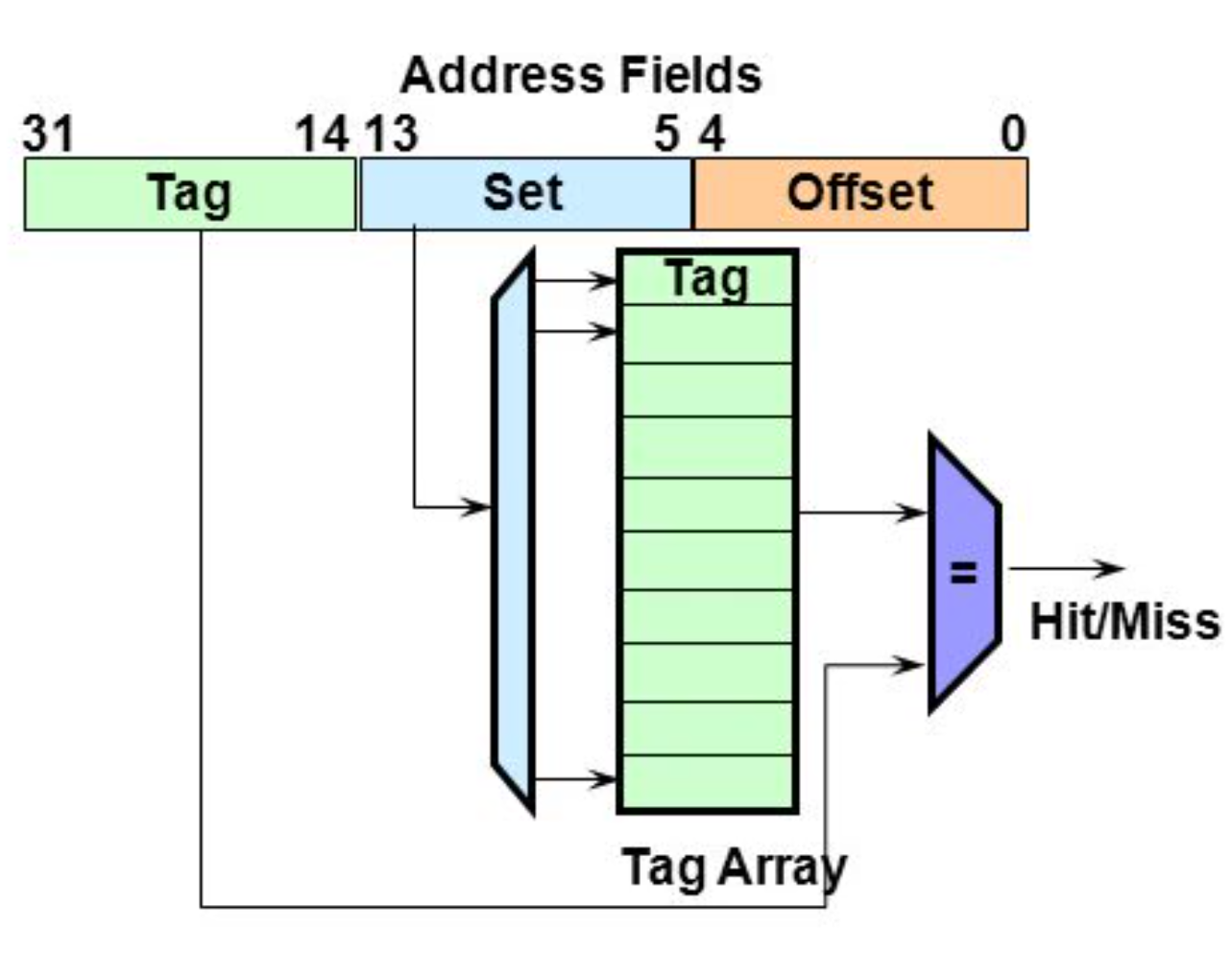
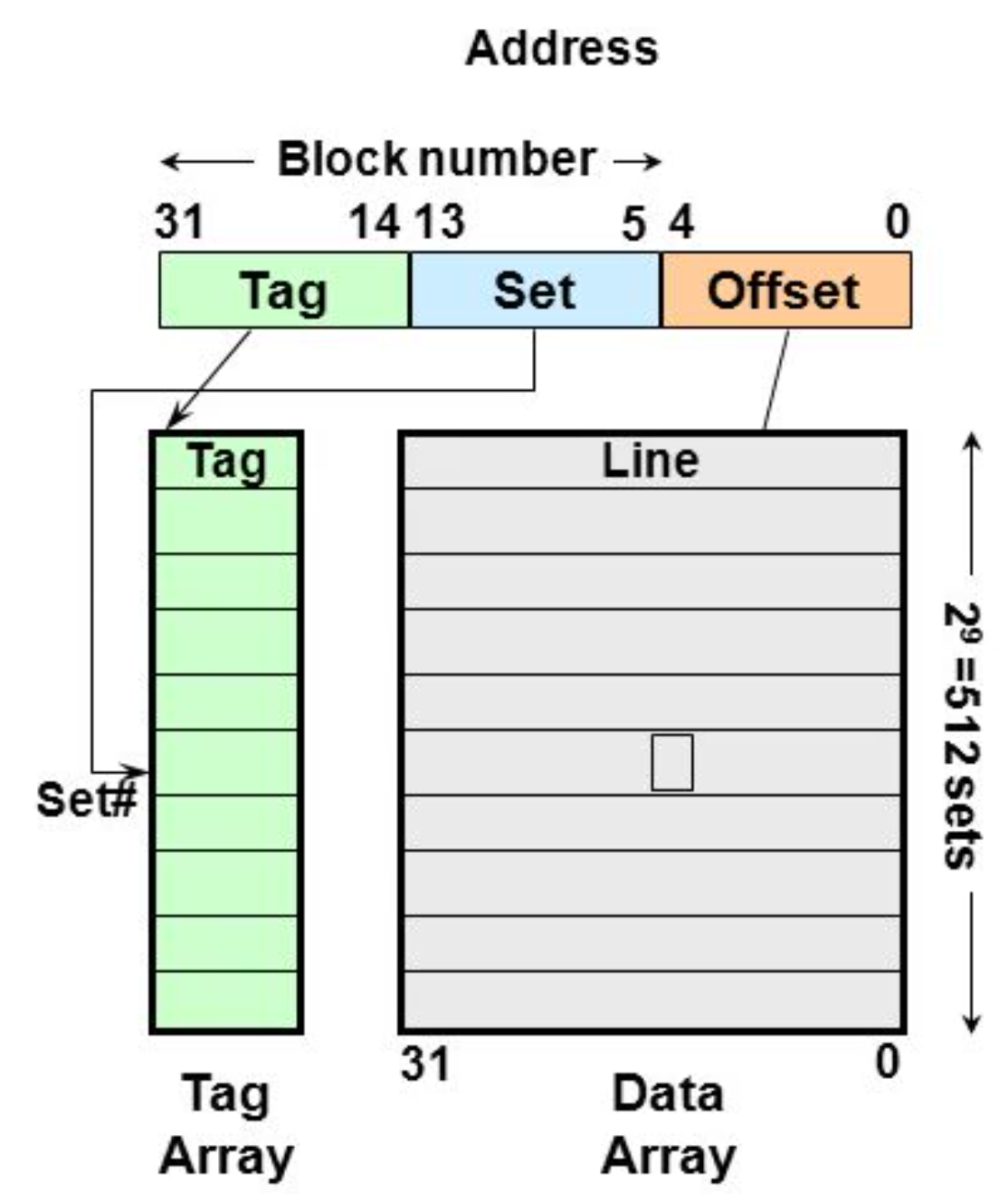
***Fully Associative Cache(FAC)***

Adressen sind aufgeteilt in Tags und Offset.

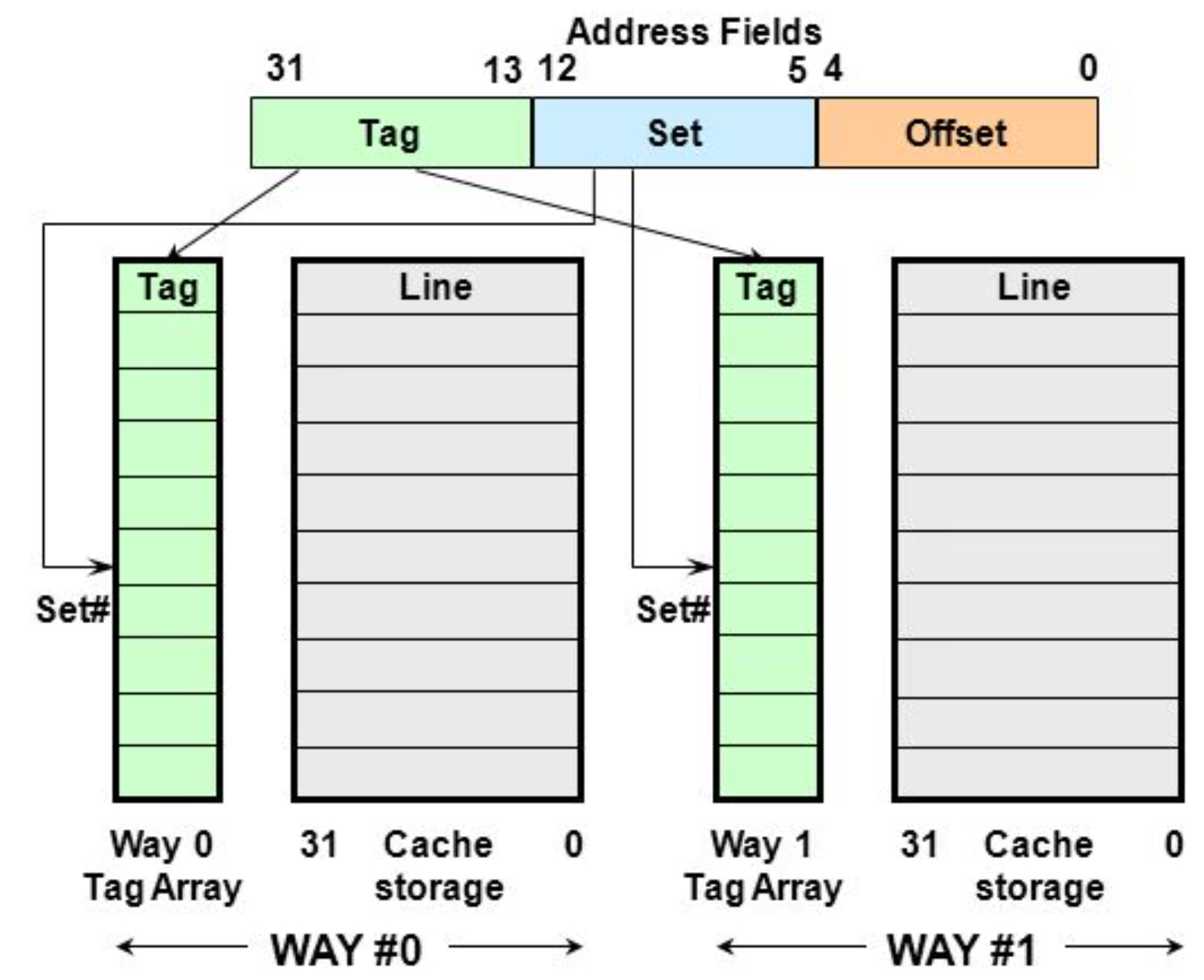
Man sucht die Cachezeile mit dem richtigen Tag und in diesem dann den richtigen Offset.

Beste Cache-Leistung, aber aufwendige Hardware(teuer).

Viel Vergleichsbausteine

***Direct Mapped Cache***

Einfachere Implementierung, viele Kollisionen, schnell

***k-Way Set Associative Cache:***

**k-verschiedene DMCs, Kompromiss zwischen FAC und DMC🡪 Weniger komplex als FAC, weniger Kollisionen als DMC aber genau so schnell wie DMC**

***Mehrstufige Caches***

In der Regel sind heute die Caches im gleichen Chip wie der Prozessor. Ebenfalls sind es mehrstufige Caches. L1-L3.

***Berechnungen***

Gegeben 2 Level Cache:

1. 64KB 4-way set associative
2. 4 MB FAC

Cache: 64Byte Cachezeilenlänge und 64Bit-Adressen. 16 GB Hauptspeicher

1. Wie viele Cachezeilen Pro Stufe und Pro Way
   1. DMC
      1. (pro Stufe 1)
   2. FAC
2. Wie viele Bits benötigt ein Tag in jeder Stufe:
   1. DMC
      1. 🡪**64 – 6 – 8 = 50 Bit**
   2. FAC
      1. 🡪 **64 – 6 = 58Bit**
      2. Im FAC wird des gesamte Tag gespeichert, weil jede Zeile in jedem Eintrag gespeichert werden kann
3. Overhead/zusätzlicher Speicher für Tags im Cache nötig?
   1. DMC
   2. FAC
4. Speicherstellen des Hauptspeichers auf dieselben Cacheeinträge?
   1. DMC
      1. 2 MB Hauptspeicher teilen sich 4 Stellen(anzahl ways) im Cache
   2. FAC
      1. Keine Stellen geteilt!

**Gegeben 8 MB Hauptspeicher, 8KB FAC, 64B Zeilenlänge**

1 Anzahl Cachezeilen = 8KB / 64 = 128 Cachezeilen

2 minimale Anzahl an Bits je Tag:

Hauptspeicher ergibt Grösse Adresse: 8 MB 🡪 23 Bit Adressen

Bit pro Tag: Adresse – Offset = 23 – 6 = 17 Bit

3 Zusätzlicher Speicher für Tags

17 Bit \* 128 = 2176 Bit = 272 Byte

***Heap***

* Speicherbereich für dynamischen Speicher
* Vom OS verwaltet
* Beliebige Grösse(der Speicherblöcke)
* Beliebiger Zeitpunkt Reservation/Freigabe

***Speicherfreigabe***

Explizit: Programmiert, wann Speicher freigegeben wird

Implizit: Speicher auto. Freigegeben, wenn nicht benötigt

OS nur explizite Speicherverwaltung, implizit App(JVM, Py, …)

Malloc(s): allozier Speicherblock grösse S

Free(\*p): gibt Speicherblock frei, der an Adresse p beginnt

Problem expliziter Freigabe: Speicherlacks wenn nicht mehr gebrauchter Speicher nie freigegeben wird.

🡪malloc & free gehören zusammen wie Klammern, sodass es «keine» Speicherlacks gibt

🡪Programmierer zuständig für Freigabe/kein OS Compiler Hilfe

***Interne Fragmentierung***

Heap reserviert grösseren Speicherblock als angefragt wurde, der zusätzliche Speicherbereich wird vom Programm nicht verwendet(Verschwendung)

***Externe Fragmentierung***

Programm reserviert immer wieder Speicher und gibt ihn unregelmässig wieder frei.

* Über Längere Zeit entstehen kleine Löcher, grössere Speicherreservierungen können nicht bedient werden obwohl die Summe der Löcher genug gross wäre

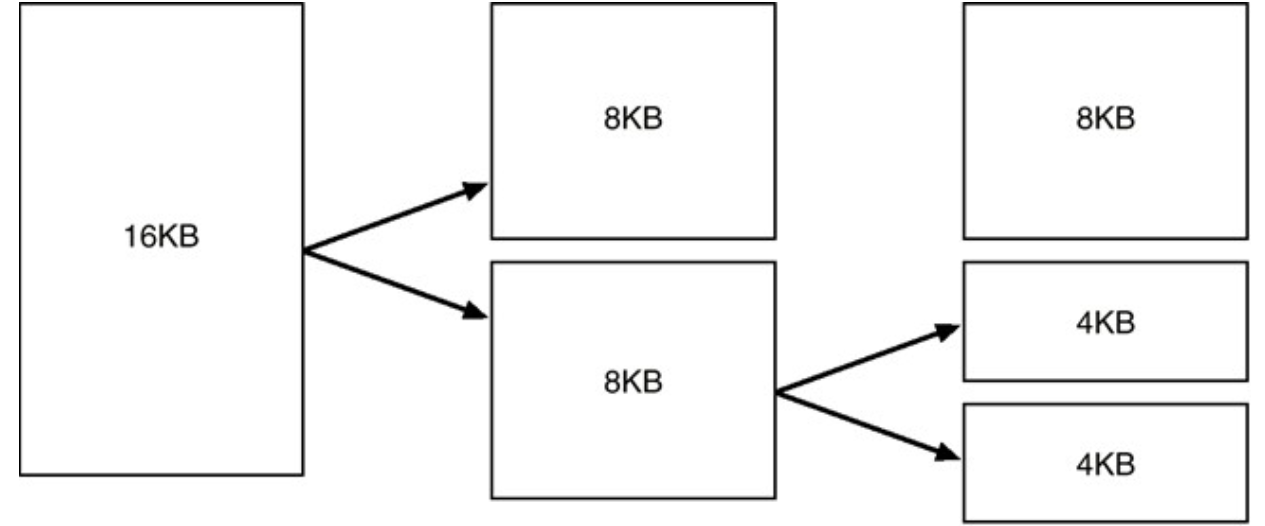
***Heap Implementierungen(geheim, Einsatzgebiet, Forschung)***- Variable Zuordnungsgrösse(Blockgrösse = 1)  
- Mehrfache fester Blockgrösse  
-Grössenklassen(verschiedene Listen von 2x freien Blöcken)  
- Buddy-System  
- Kombinationen der gennanten Basis-Varianten

**Mehrfache fester Blockgrösse**

**Blockgrösse x und Mehrfache von x** 🡪 je kleiner, desto weniger interne Fragmentierung aber mehr Aufwand/Metadaten  
**Metadaten dezentral**( bei Speicherblöcken, Überlaufschutz mit magic Numbers, kein Schutz vor Programmierfehler)  
Metadaten zentral(jenseits von Speicherblöcken) Kein Überlauf, aber zusätzlicher Speicherplatz für Metadaten

**Speicherblock-Verwaltung**Bitliste( Ein Bit pro Block 0 = Block frei, 1 Block verwendet), suche linear und langsam, he weiter hinten Platz frei ist, desto länger  
Verkettete Liste(frei,32 -> besetzt, 32 -> frei 64), sehr schnelles zusammenführen, wenn a(32) frei und b(32) frei -> ab(64)

**Suchalgorithmen(eingefügte Blöcke werden aufgeteilt!)  
First-Fit 🡪** erste passende Lücke am Anfang  
**Next Fit**: 🡪 erste passene Lücke nach zuletzt reserviertem Bereich  
**Best-Fit**🡪Durchsucht alle Lücken, wählt kleinste passende aus  
**Worst-Fit** 🡪 Durchsucht alle Lücken, nimmmt grösste Lücke

**Buddy System**Speicheranfragen 🡪 nächste 2er Potenz, wenn zu klein 🡪 Buddy

Bsp: Gesamt 16 KB

Braucht 5 KB 🡪 zwei Buddys mit jeweils 8 KBs

Dann nochmal 3 KB

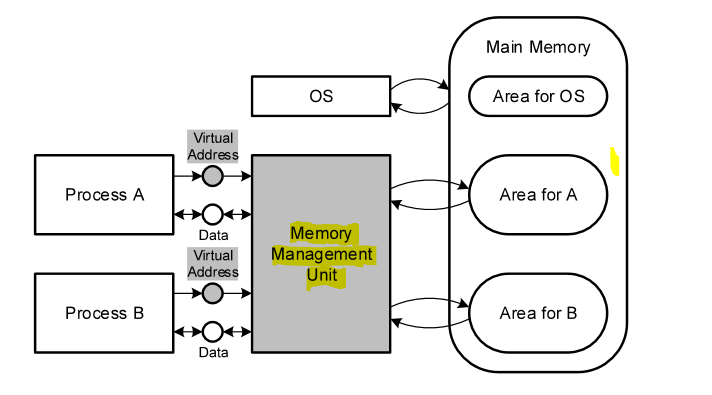
* Nochmals 2 mit 4 KB. Freigabe umgekehrt!

**Virtueller Speicher**

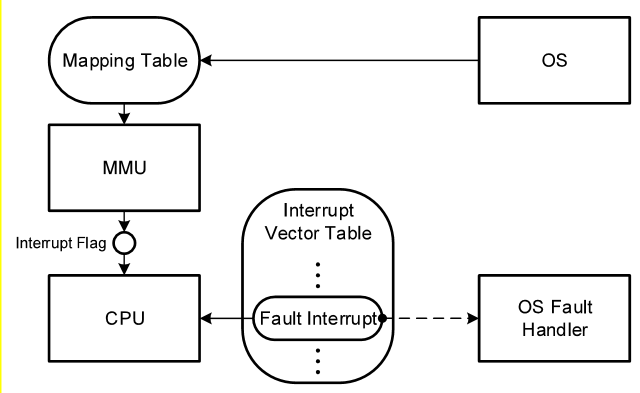
***Prozesse***

Prozesse bekommen von OS Speicher zur Verfügung¨

OS zuständig, dass Prozesse sich nicht stören, Daten austauschen, etc.

Lösung: Prozess kennt nur virtuelle Adresse, MMU übersetzt virtuelle Adresse in physische Adresse, OS konfiguriert MMU pro Prozess:

***Ungültiger Zugriff***

OS stellt Mapping Tabble pro Prozess  
(Übersetzungstabelle von virtuellen zu physischen Adressen)

Bei Fehler 🡪 OS übernimmt

***Pages***

Virtueller Adressraum besteht aus Pages, Page jedoch kein Speicher, benötigt dazu Speicherort(Hauptspeicher/Sekundärspeicher)

***(Page)Frames***

Hauptspeicher besteht aus Frames. In ein Frame passt genau eine Page. Daraus folgt ein Frame = Page.

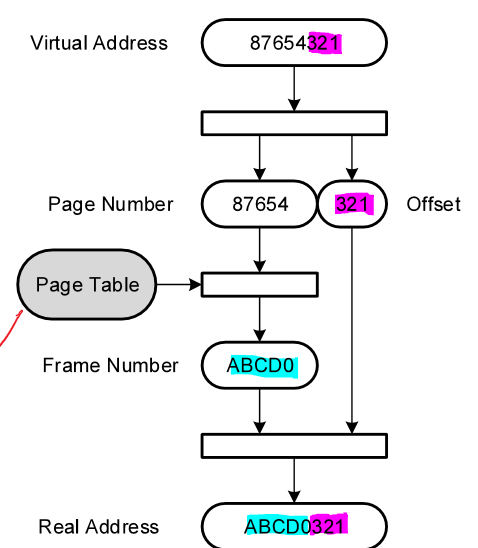
***Virtueller*** Adressraum ***/ Page Table***

Pro Prozess ein virtueller Adressraum

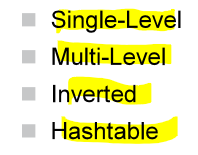
Pro Prozess ein Pagetable(Mapping Tabelle)

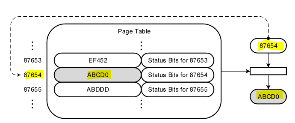
OS verwaltet, welche Pages wann wo liegen müssen, MMU kennt nur Hauptspeicher🡪 MMU kann nur sagen ob Page, resp. Zu Page gehörendes Frame im Hauptspeicher

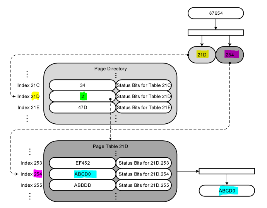
***MMU***

****Generelle Aufgabe**: Übersetzten von virtueller in physikalische Adresse. Page Number 🡪 Frame Number, Offset bleibt gleich.

Umwandlung in Page Table zu finden.

Implementierungen:

**Single-Level**: Für jede mögliche Page einen Eintrag.(Grösse hängt von virtuellem Adressraum ab)

Kann sehr schnell sehr gross werden, Lookup sehr schnell

**Two** **Level**: Page Number wird

aufgeteilt: in Directory Index &

Page Table Index

1)Page Table in Directory finden

2) Table Index in Page Table finden

Viele Page Tables, Page Directory zeigt auf Page Tables

**Status-Bit Used(P Bit)**

Gibt an, ob Page überhautp benutzt(Used = 1 oder 0)

***Interprozesskommunikation(IPC)***

**Shared Memory**: Prozesse teilen Speicher, kein Schutz

**Message Passing**: OS kopiert Daten, Sicher, Overhead

**Berechnungen**

Speicher Two Level :(t + 1) \*4KB 1 Table pro 1KPages

Speicher Single Level (Adressbereich : Pagegrösse \* Zeilengrösse)

***Paging***

Paging nennt man die Verwaltung von pageorniertierem Speicher(ist der zugehörige Frame im HS oder SS), Laden von Pages, etc.

***Dreschen***

Dreschen nennt man den Vorgang, wenn die ganze Zeit Pages aus dem Sekundärspeicher geholt werden müssen. Gründe dafür sind i.d.R Hauptspeicher zu klein oder zu viele Prozesse parallel laufen

***Paging-Strategien***

Da das OS beim Paging ständig Pages in der Speicherhierarchie verschiebt gibt es verschiedene Strategien:

**Ladestrategie:** Wann und welche Pages laden

**Entladestrategie:** Wann modefizierte Pages zurückschreiben

**Veränderungsstrat:** Welche Page entfernen, wenn HS voll

***Ladestrategien***

**Demand Paging**: Laden auf Anfrage(Minimaler Aufwand, Lange Wartezeit)

**Prepaging**: vor Verwendung geladen(«vorhersehen» 🡪Analyse)

**Demand Paging mit Prepaging**: Pages + Nachbarn geladen

***Entladestrategien***

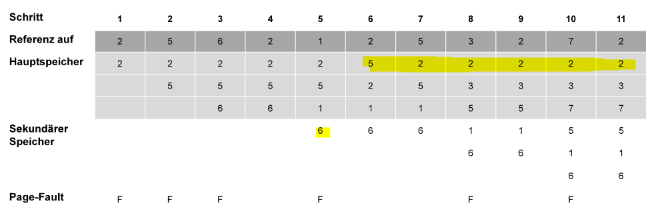
**Demand Cleaning**: Page zurückgeschrieben, wenn Frame wiederverwendet werden soll. Minimaler Aufwand, Lange Wartezeit(Page muss vor Gebrauch noch geschrieben werden)

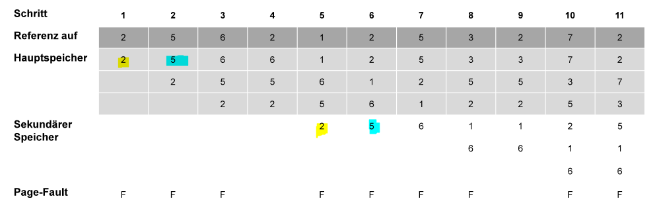
**Precleaning**: Modifizierte Pages frühzeitig in sekundär Speicher🡪 Schreiben evtl. sinnlos, wenn wiederverwendet, kurze Wartezeit

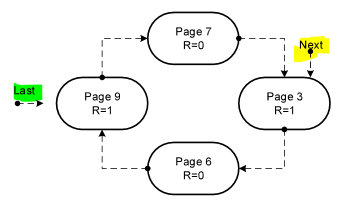
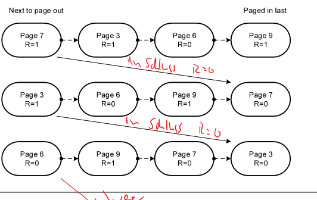
Page Buffering: Zwischenspeichern von veränderten und unveränderten P. (+/- wie Precleaning, +schelle Auswahl)

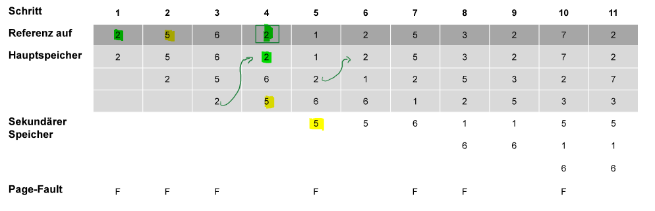
***Verdrängungstrategien(massiver Einfluss auf Performance)***

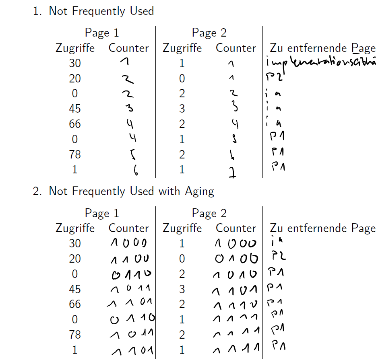
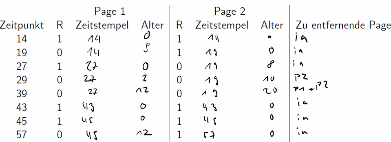
Page Fault, wenn Page nicht in Hauptspeicher

**Optimal**: ersetzte Pages, die am längsten nicht kommt

**FIFO(First in First Out)**

**Second Chance(2. Chance[siehe R]) Clock(Next Spring)**

**LRU(Last Recently Used) Längstes unbenutze Page[fast Optima]**

**NFU(Not Frequently Used)**

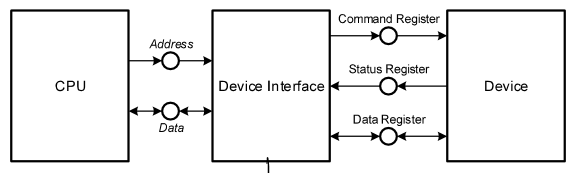
**Ein- & Ausgabe**

***Eingabegerät***: physikalische Sachverhalten🡪 Daten

***Ausgabegerät***: Daten🡪 physikalische Vorgänge

Einige Geräte auch beides(SSD, Festplatten, Touchscreen)

**Datenkommunikation**

Zwischen CPU und Device wird eine Schnittstelle eingefügt, welches Register in Adressen oder Daten übersetzt.

***Memory*-mapped I/O**

Pro Gerät wird ein eigenes Adressbereich reseviert, welcher nur für das Gerät verwendet werden kann.

+ Einfach(CPU weiss nicht, was Geräte, keine I/O-Logik)

– Adressraum des Speichers nicht für Speicher ausgenutzt

***Ported*-mapped IO**

Geräte haben eigenen IO Bus, zwei Adressräume, Speicher und Geräte

+ Speicher kann gesamten Adressraum verwenden

- Komplexität 🡪 CPU zusätzliche Logik, System zusätzlichen Bus

***Ported*-mapped I/O via Speicherbus¨**

Bus hat zusätzlicher Bitleitung(Selector Bit), welche mitteilt, ob Speicher oder Device. Verbindet Vorteile beider Verfahren.

***Pollingsmechanismen***(Wann/Wie werden Daten geholt)

**Programmgesteuert**: Ständige(oder Zyklische) Abfragung, ob Status Register gesetzt.

+keine Verzögerung -beansprucht CPU

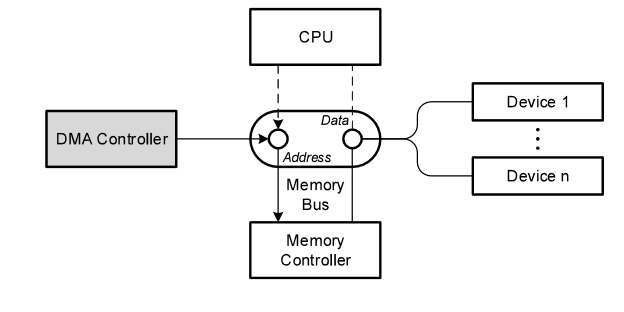
**Interruptgesteuert:** Gerät sendet Unterbruchssignal wenn bereit

***DMA(Direct Memory Access)***

Eine weitere Möglichkeit, um Wartezeiten auf die Peripherie nutzungsbringend einzusetzten. Ziel ist es, dass der Datentransfer nicht über die CPU sondern direkt abläuft. Dabei gibt es einen weiteren Baustein, den DMA-Controller, welcher die Übertragung zwischen Daten aus dem Speicher und dem Device handelt:

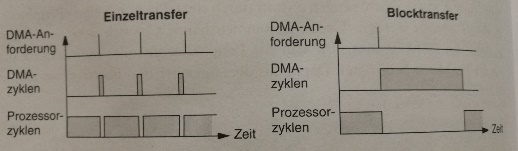
**Vorgehensweise(vereinfacht)**

1. CPU programmiert DMA für Datentransfer
   1. CPU sieht DMA als eigentliches Gerät
   2. Quelle, Ziel, Menge, Betriebsart gesetzt
2. CPU gibt Speicherbus an DMA frei
   1. DMA gibt Angaben am Gerät, sodass dieses direkt den Transfer ausführen kann
3. Nach Beendigung sendet DMA an CPU einen Interrupt

****

**Betriebsarten**

**Einzeltransfer**: CPU gibt Speicherbus für einzelnen Transfer für DMA frei(wenige, periodische Daten Modem, Diskette)

**Blockfransfer:** CPU tritt Speicherbus an DMA ab, bis alle Daten übertragen sind(Fesplatte, Netzwerkkarte, etc)