# Prozessor

## speicherbus

Speicherzellen nicht direkt an Prozessor angebungen, stattdessen hat jede Speicherzelle eine Adresse. Prozessor kennt nur Speicherbus.

**Adressbus:** Enthält Adresse, auf die zugegriffen werden soll

**Datenbus:** Enthält Daten, die aus Speicherzelle stammen oder in diese geschrieben werden soll

**Steuersignale:** Angabe, ob gelesen oder geschrieben wird

Icon

Description automatically generated

Nur der Prozessor steuert Speicherbus und legt eine Adresse auf den Adressbus. Aus Sicht des Prozessors gibt es nur 2 Operationen (Lesen und Schreiben).

**Register:** Kleine Menge an Speicher im Prozessor

**Operation:** Baustein im Prozessor, welche eine bestimmte Arbeit signalisieren

**Instruktion:** Kombination einer Operation mit Operanden

**Beispiel:** Kopiere Wert aus Register x in Register y und addiere Wert in Register y zu Wert in Register x und speichere in Register x

## sequenz

Sequenz ist hardcoded im Prozessor und kann nicht geändert werden. Sequenzen von Instruktionen liegen im Hauptspeicher und müssen über den Speicherbus abgefragt werden. Operationen/Register sind durchnummeriert. Gemeinsam mit Daten- und/oder Adressoperanden ergibt sich eine Bytesequenz.

**Beispiel:** Addiere 12345678h zu Register 1 (Intel 64 Architektur)

Diagram

Description automatically generated

Diagram, schematic

Description automatically generatedGesamtheit aller möglichen Instruktionscodes eines Prozessors ist sein Maschinencode und wird vom Hersteller festgelegt. Sequenzen von Maschinencode sind im Hauptspeicher abgelegt. Prozessor hat ein spezielles Register, das die Adresse des nächsten Befehls enthält.

# assembler

## byte order/endianness

**Intel Terminologie:**

|  |  |
| --- | --- |
| Byte | 8 Bit |
| Word | 2 Byte |
| Doubleword (DWord) | 4 Byte |
| Quadword (QWord) | 8 Byte |
| Double Quadword (DQWord) | 16 Byte |

Intel Prozessoren verwenden als kleinste Einheit 1 Byte. Einzelne Bits sind nicht zugänglich, da diese keine eigene Adresse besitzen. Ein einzelnes Bit muss immer mit dem gesamten Byte verarbeitet werden. Ein Byte hat 2 hexadezimale Stellen und . Die höherwertige Ziffer steht immer links: .

**Byte Order:** Ein Word hat 4 hexadezimale Stellen bis . Üblich die Stellen von rechts nach links zu schreiben . 2 aufeinanderfolgende Bytes im Speicher und haben Adressen und . Üblich Werte im Speicher von links nach rechts zu schreiben .

A picture containing text, whiteboard

Description automatically generated**Endianness:** Double Word 87654321h

A picture containing text

Description automatically generated

**Vorteil Little-Endian:** Selbe Adresse für höherwertige Stellen

## assembler

Assembler ist ein Programm, das textuelle Befehle in Maschinencode übersetzt. Es gibt keine einheitliche Assemblersprache. Konventionen hängen massgeblich vom Hersteller des Assemblers ab, selbst für denselben Prozessor.

NASM (Netwide Assembler) übersetzt jede Anweisung in Binärzahlen und schreibt in Zieldatei

Diagram

Description automatically generatedDiagram

Description automatically generated

**Bsp:**

|  |  |
| --- | --- |
| **Quellcode** | **Binär-Output** |
| db 0x35, 0h21, 049h | Die 3 Byte: 35h 21h 49h |
| db Hallo | ASCII von H, a, l, l und o: db 0x48, 0x61, 0x6c, 0x6c, 0x6f |
| dw 0x2135 | db 0x35, 0x21 |
| dd 0x2135 | db 0x35, 0x21, 0x00, 0x00 |

**Spezifikation von Befehlen:** NASM übersetzt fast jede Anweisung direkt in Binärzahlen und schreibt diese als Sequenz von Bytes in die Zieldatei. Befehle beginen mit der Operation gefolgt von einer Komma-separierten Liste der Argumente.

|  |  |
| --- | --- |
| nop | 90h |
| not rax | 48h F7h D0h |
| mov rax, rbx | 48h 89h D8h |

**Offset:** Nummeriert man die Bytes in der Ausgabe-Sequenz fortlaufend, erhält jedes Byte einen Offset (Adresse, Index).

A picture containing chart

Description automatically generated

**Label:** Intern assoziiert der Assembler das Label w mit dem Offset A(w) des nachfolgenden Befehls in der Ausgabedatei. Den aktuellen Offset kann man mit $ abfragen.

A picture containing table

Description automatically generated

Diagram

Description automatically generated

**Variables:**

SECTION .data ; necessary !!

varname dq 45 ; 8 byte int

**Assembler Instruktionen:**

|  |  |
| --- | --- |
| loop: | Definiert Symbol als Pointer |
| mov rax, rbx | Kopiere Wert von rbx an rax |
| mov rax, [rbx] | Kopiere Byte von rbx an rax |
| inc rax | Inkrementiere rax um 1 |
| sub rax, rbx | Subtrahiere rbx von rax und schreibe in rax |
| cmp rax, 1 | Vergleicht rax mit 1 und setzt Z-Flag falls gleich |
| ret | Hole obersten Wert vom Stack und setze Befehlszeiger darauf |

## ausgabedateien

Per Default erzeugt NASM sogenannte Flat-Form Binaries (raw). Die reine Bytesequenz analog zum Quelltext ohne Zusatzinformationen. Objekt Dateien enthalten darüber hinaus noch weitere Informationen, die später benötigt werden.

Diagram

Description automatically generated

**Objekt-Datei:** Enhält u.a die Symboltabelle. Global deklarierte Label werden exportiert. Extern deklarierte Label haben keinen Wert innerhalt der Objekt-Datei.

Diagram

Description automatically generated

**Symboltabelle:**

Text

Description automatically generated with low confidenceText

Description automatically generated

* Die erste Spalte gibt den Offset an
* Die zweite Spalte zeigt Symbolattribute: g für global und l für lokal
* Die dritte Spalte verweist auf die Sektion (für unsere Fällte .text)
* Die vierte Spalte gibt den Namen am
* Das als extern deklarierte Label z erschant als \*UND\* (undefined)

# variablen

Ursprünglich hatten Intel-Prozessoren 16-Bit-Register, die in zwei 8-Bit-Register unterteilt wurden. Mit der Zeit wurde das Register mit namenlose Bits erweitert.

Text, letter

Description automatically generatedDiagram

Description automatically generated

## instruktionen

**Operationen:** Datentransfer-Operation, Arithmetische/Logische Operation, Programmfluss-Operation, String-Operation, Operation zur Kommunikation, etc.

**Ausführungszeit:** Operationen benötigen unterschiedlich lange (1 Prozessorzyklus bis mehrere 100). Operationen die auf Speicher zugreifen, müssen auf Speicher warten.

**Länge der Instruktion:** Instruktionen sind Binärzahlen, die Operationen/Operanden codieren. Auf Intel 64 können diese 1 bis 15 Byte lang sein. Grösse hängt von Operation ab. Länge einer Instruktion nicht in der Sequenz enthalten, d.h Sequenz muss von Anfang an Instruktion für Instruktion durchgehen, um jede richtig zu decodieren.

**Spezifikation von Instruktionen:** NASM übersetzt Anweisung direkt in Binärzahl und schreibt diese als Sequenz in Bytes in die Zielddatei. todo: maybe add example of mov

## address. vom hauptspeicher

Speicherstellen können auf verschiedene Weisen spezifiziert werden (Displacement, Base (Register), Scaled Index, Jede Summer der drei vorherigen).

|  |  |  |
| --- | --- | --- |
| **Displacement [a]:** Adresse a der Speicherstelle folgt unmittelbar | **Base [r]:** Adresse der Speicherstelle steht in Register | **Scaled Index [i \* s]** |
|  |  |  |

## linker

Programme werden üblicherweise aus mehreren Assemblerdateien generiert. Dadurch kann man Dateien einzeln übersetzen, iterativ übersetzen weil nur geänderte Dateien neu übersetzt werden müssen, die Entwicklung auf Personen verteilen, VSC effizient einsetzen. Die einzelnen Dateien müssen zu einem Programm (Executable) zusammengesetzt werden.

Graphical user interface, text, application, chat or text message

Description automatically generated

Assembler erzeugt aus einer Assemblerdatei eine Objekt-Datei. Der Linker erstellt aus Objekt-Dateien ein Executable. Jedes Symbol erhält einen eigenen Platz im gelinkten Objekt.

**Einsprungspunkt:** In jedem .exe muss der Einsprungspunkt definiert werden. Default ist \_start

**OS-Syscalls:** Instruktion syscall übergibt die Ausführung an das OS. In rax übergibt man den Code für die OS-Funktion. Für allfällige Parameter werden .

**Ende des Programms:** Programme müssen explizit beendet werden. OS weiss nicht wann das Ende ist. Dazu gibt es OS-Syscall , üblicherweise Code 60. (mov rax, 60)

# C-toolchain

Diagram

Description automatically generatedAssembler-Programmierung ist mühselig. Mathematische Formeln erfordern viel Aufwand, Programmierer muss Verwendung der Register selbst verwalten. Schleifen und Verzweigungen werden druch Sprünge dargestellt und sind schwierig nachzuvollziehen. C stellt eine direkte Evolution aus Assembler dar, um obige Probleme zu adressieren. Viele Konzepte sind direkt auf Assembler abbildbar.

**Präprozessor:** Definiert Direktiven, die im Programm vor dem eigentlichen Übersetzen als Textersetzung durchgeführt werden.

**Basiskonstrukte:** Bestimmen das Grundgerüst eines Programms, z.B Variablen, Schleifen, Verzweigung

**Standardbibliotheken:** Stellen Funktionen und Typen bereit, die die Basis-Funktionalität enthalten

## c-Preprocessor

Verarbeitet die Input-Datei in mehreren Durchläufen. In jeden Durchlauf verarbeitet er die gesamte Datei einmal.

**Durchläufe:**

1. Entfernen aller Kommentare und unwandeln fortgesetzter Zeilen, die mit \ enden in eine Zeile
2. Teilt den gesamten Text in Tokens ein (Tokenization). Es gibt 5 Klassen von Tokens: Bezeichner, Präprozessor-Zahlen, String-/Charliterale, Operationen, Sonstige  
   Whitespae trennt Token voneinander, aber nicht alle Token müssen durch Whitespace getrennt werden, z.B a+b
3. Durchläuft nun die Token-Liste und führt Präprozessor-Direktiven aus und ersetzt Makros durch ihre Expansion

Diagram, text

Description automatically generatedDiagram

Description automatically generated with medium confidence

**Bezeichner:** Ein Bezeichner beginnt mit einem Buchstaben oder \_, gefolt von einer Sequenz aus Buchstaben, \_ oder Ziffern. Bsp. a, \_, \_a, a0, \_0, \_a0, \_0\_0, \_\_\_\_\_, int, for, size\_t

**Präprozessor-Zahl:** Beginnt mit Ziffer, gefolgt von einer Sequenz aus Ziffern, Buchstaben, \_, ., oder Exponente. Vor der ersten Ziffer kann auch ein «.» stehen. Bsp. 0, 1, 0123, 0x1234, .05

**String- und Characterliterale:** String-/Characterliterale beginnen/enden mit “/’. Alle Zeichen dazwischen gehören zum Literal und werden nicht weiter in Token unterteilt. “/’ wird in Stringliteralen durch \" und \' dargestellt

**Operatoren und Satzzeichen:** Der Präprozessor ist greedy, d.h er versucht immer das grösstmögliche Token zu bilden. Z.b (), [], {}, ., :, ; etc.

**Präprozessor-Direktiven:** Ist das erste Token auf einer Zeile #, wird das nächste Token als Direktive interpretiert. Beide Token werden entfernt und die entsprechende Direktive ausgeführt. Wichtige Direktive: include, define, if, else, endif

Diagram

Description automatically generated**#include:** Präprozessor öffnet die Datei anhand des nächsten Tokens und durchläuft 1 bis 3 für neue Datei druch. Präprozessor setzt Arbeit nach der Direktive in Originaldatei fort. Präprozessor kann dadurch mehrere Dateien zu einer Translation Unit zusammenführen.

**Objektartige Makros:** Es gibt objektartige und funktionsartige Makros. Objektartige Makros haben keine Parameterliste. Der Präprozessor ersetzt im Programmtext nach der Definition des Makros jedes Token, das dem Makronamen entspricht, durch die Tokenliste

Aus #define XYZ 123 und int y = XYZ; wird also int y = 123; Taucht der eigene Makroname in der Ersetzung auf, wird er nicht ersetzt, um infinite Rekursion zu verhindern.

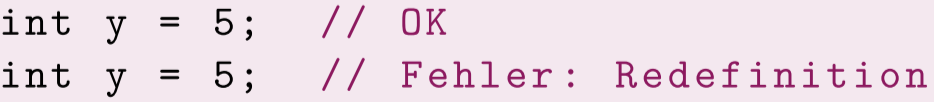
## c-Compiler

Der Compiler übersetzt eine C-Datei in eine Assembler-Datei. Die C-Datei enthält keine Präprozessor-Direktiven, sondern nur C-Sprachkonstrukte. Eine C Translation Unit ist eine Folge von Deklarationen und Definitionen von globalen Variablen, Funktionen und Typen.

**Bezeichner:** Das Gegenstück zu Labels in C sind Bezeichner. Ein Bezeichner ist eine Zeichenkette aus alphanumerischen Zeichen, die nicht mit einer Ziffer beginnt.

**Deklaration:** Eine Deklaration verknüpft einen Bezeichner compiler-intern mit einem Typen. Deklarationen haben keinen direkten Einfluss auf den erzeugten Byte-Stream (ähnlich wie Label in Assembler). Innerhalb einer Translation Unit darf jeder Bezeichner beliebig oft deklariert werden, solange die Deklaration gleich ist. Unterschiedliche Translation Units können Bezeichner unterschiedlich deklarieren, da der Compiler Translation Units unabhängig voneinander betrachtet.

**Definition:** Erzeugt den Inhalt der deklarierten Entität. Eine Entität darf in einer Translation Unit nicht mehrfach definiert werden, auch nicht wenn die Definition exakt gleich ist.



Eine Entität kann in verschiedenen Translation Units mehrfach und auch unterschiedlich definiert werden. Je nach Art der Entität erzeugt der Linker eine Fehlermeldung.

**Globale Variablen in C:** Globale Variablen haben immer einen Typ und einen Bezeichner. Die Deklaration erfolgt durch ein Typ-Token gefolgt von einem Bezeichner-Token: Für jede globale Variable wird im Programm Speicher fix reserviert. Die Grösse des reservierten Speichers hängt ausschliesslich vom Typ: alle Variablen eines Typs brauchen exakt gleich viel Speicher.

**Initialisierte Globale Variablen:** Optional kann jede globale Variable initialisiert werden. Dazu folgt auf das Bezeichner-Token das Token = und dann der Initialwert.

**Export globaler Variablen:** Globale Variablen werden standardmässig exportiert

**Import globaler Variablen:** Globale Variablen, die aus anderen Objekt-Dateien verwendet werden sollen, werden mit extern deklariert

**Static Variablen:** Variablen, die nicht exportiert werden sollen, werden mit static bezeichnet

Table

Description automatically generated

**Header:** Mit einem Header stellt man sicher, dass Variablen in verschiedenen Dateien jeweils diesselbe Deklaration sehen.

**sizeof(t):** Note the return type is size\_t which is big enough to hold any type

printf("&d",sizeof(a)); // 16 for intel x86

# Variablen/ausdrücke

Der Inhalt eines Objekts kann verschieden interpretiert werden, z.B. als signed oder als unsigned. Benachbarte Speicherbereiche können als ein grosses oder mehrere kleine Objekte und Werte betrachtet werden.

**Objekt:** Ein zusammenhändender Bereich, sein Inhalt kann als Wert interpretiert werden

**Wert:** Genaue Bedeutung des Inhalts eines Objekts, wenn dieses mit einem Typ versehen ist

Jede globale Variable ist ein Objekt, aber nicht jedes Objekt muss einer Variablen entsprechen. Rückgabewerte von Funktionen werden in Objekten zurückgegeben (kein Bezeichner). Es gibt Objekte, die nur über Pointer erreichbar sind. Man kann eine globale Variable als benanntes Objekt ansehen.

**Typen:** Typen beschreiben Objekte bzw. deren Eigenschaften eine feste Objektgrösse sizeof(T) hat, bei Operationen die Typgleichheit der Operanden erzwungen wird, Typinformationen nur auf dem Sprachlevel C existieren. Nach der Übersetzung in Assembler gibt es keine Typen mehr, sondern nur noch Speicher.

Damit kann der Compiler automatisch die richtige Grösse für bestimmte Objekte allozieren, die richtige Operation auswählen (signed/unsigned Multiplikation) und Fehler melden, wenn Typen nicht kompatibel sind (Minimaler Vorteil, da C schwach typisiert)

Nachteil ist, dass C Plattform-unabhängig ist und dadurch auf einer Plattform gut läuft und fehlerhaft auf einer einer anderen.

Zu jedem Signed-Integer-Basistypen gibt es das Unsigned Gegenstück, das die gleiche Grösse hat. void hat keine Werte und wird verwendet, wenn keine Werte zulässig sind.

**Pointer-Typen:** Auf Maschinencode-Ebene gibt es keine Variablen, sondern nur Adressen. Adressen sind Binärzahlen und können somit selbst auch gespeichert werden. Die Adresse eines Objekts, dessen Typ nicht bekannt ist, ist vom Typ void . Man nennt solche Adressen void-Pointer.

A picture containing text

Description automatically generated**Beispiel:** Adresse eines Objekts vom Typ T ist vom Typ T\*. Adresse einer Adresse vom Typ T\* ist vom Typ T\*\*

Icon

Description automatically generated

## ausdrücke

C übernimmt die Verwaltung der Register für uns. Stattdessen schreiben wir formel-ähnliche Ausdrücke. Jeder Ausdruck hat einen Typen, den der Compiler aus den Operanden ableitet. Primäre Ausdrücke sind Konstanten und Literale sowie Bezeichner. Alle anderen Ausdrücke werden durch Operatoren und Casts erzeugt.

**Integer-Konstanten in C:**

* Dezimalzahl: Folge von Ziffern (0..9), die nicht mit 0 beginnt
* Oktalzahl: Folge von Ziffern (0...7) die mit 0 beginnt
* Hexadezimalzahl: Folge von Ziffern (0...F), die mit 0x beginnt
* Es kann ein Typen-Suffix angehängt werden: l, ll, u, ul, ull

Typ ist der kleinste, in den die Konstante passt, aber mindestens int.

**Zuweisung:** Zuweisungen erfolgen über den Zuweisungsoperator «=». Der Ausdruck auf der linken Seite muss eine Speicherstelle bezeichnen. Der Ausdruck auf der rechten Seite muss einen Wert haben. Die Typen beider Ausdrücke müssen gleich sein.

**Vergleich Zuweisung :**

|  |  |
| --- | --- |
| C | Assembler |
| x = y; | mov rax, [y] (nicht mov rax, y)  mov [x], rax (nicht mov x, rax) |

**Referenzoperator:** Der Referenzoperator & erzeugt die Adresse eines Ausdrucks. &a ergibt die Adresse der Variable a. Ist T der Typ von a, dann ist T\* der Typ &a

**Dereferenzoperator:** Der Dereferenzoperator \* wird auf eine Adresse angewandt. Er wandelt die Adresse in einen Ausdruck um, der für den Inhalt steht. Ist damit das Gegenstück zum Referenzoperator.

## arithmetische/logische oper.

**Logische Funktion:** Funktion f von n Bits auf 1 Bit

**Parameter:** Variable zur Übergabe von Werten an eine Funktion

**Argument:** Wert eines Parameters bei einer konkreten Verwendung der Funktion

**NAND:**  heisst NAND (NOT AND,«nicht alle wahr»). Das Symbol | heisst in diesem Kontext «Shefferscher Strich». Basisoperationen können ausschliesslich aus | dargestellt werden. NAND-Bausteine sind technisch leicht als Transistoren zu realiseren.

**Logische Operationen: Bitwise Operatoren:** Gleich wie Assembler-Op.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Assembler** | **C** |  | **Assembler** | **C** |
| not z | z = !q |  | not | z = ~ q |
| and z, q | z = q && p |  | and | z = q & p |
| or z, q | z = q || p |  | or | z = q | p |
| xor z, q |  |  | xor | z = q ^ p |

**Basis der Addition:** XOR bildet die Addition zweier Bits ab, AND den Übertrag

**A picture containing text, clock

Description automatically generated**

**Sign Extension:** Wird eine n-Bit-Zahl auf eine n+m-Bit-Speicherstelle kopiert, werden die m oberen Bits auf 0 gesetzt. Bei einer Zahl im Zweierkomplement ändert sich dadurch aber die Bedeutung. Z.b 4 Bit auf 8 Bit kopieren:



Statt 0 muss überall eine 1 kopiert werden.Das Vorzeichen wird auf alle zusätzlichen Bits ausgedehnt. Sign-Extension wird nicht in allen Befehlen automatisch ausgeführt (z.B. mov).

**Shift und Rotate in Assembler:** Shifts können ähnlich performant wie die elementaren Operationen implementiert werden. Rotates füllen statt mit 0 oder 1 mit den ursprünglichen Bits auf.

**Beispiel:** Assembler links, C rechts

Text, letter

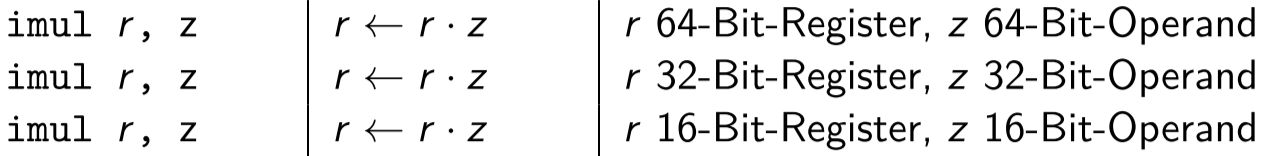
Description automatically generated

**Multiplikation:** Unsigned Multiplikation hat nur einen Operanden, der andere ist implizit: mul z. z kann ein beliebiger Operand sein. Je nach Bit-Grösse von z sind Ergebnis und der andere Operand. Bspw. bedeutet mul rbx, dass RDX: RAX ← RAX · RBX

Table

Description automatically generated

imul ist das signed Äquivalent zu mul. Zusätzlich zu den fixen Registern wie bei mul gibt es drei flexiblere Versionen:



**Division in Assembler:** Umgekehrt gibt es die Division mit fixen Registern: div und idiv. Der Dividend d verwendet dieselben Register wie das Produkt bei mul bzw. imul, z.B. RDX:RAX

Der Divisor z ist ein frei wählbarer Operand. Schreibt den Quotienten in die LSBs von d, z.B. RAX. Schreibt den Rest in die MSBs von d, z.B. RDX. Division sollte gemieden werden, da sehr langsame Operation (80x so lange wie Multiplikation in 64 Bit).

Text

Description automatically generated

# kontrollfluss

**Flags:** Einzelne Bits, die eine eigenständige Bedeutung haben. Liegen gemeinsam in Register RFLAGS, das nicht direkt verwendet werden kann. Operationen verändern Flags in unterschiedlichem Masse.

**Carry Flag CF:** Gibt Überlauf bei unsigned Arithmetik an. Entspricht dem Übertrag bei den MSB. 0001 + 1111 = (1)0000 1 + 15 = 0 CF = 1

**Overflow Flag OF:** Gibt Überlauf bei signed Arithmetik an. Wird gesetzt, wenn die Vorzeichen nicht mehr stimmen. 0111 + 0001 = 1000 7 + 1 = -8

**CF oder OF:** Prozessor weiss nicht, ob mit signed oder unsigned gearbeitet wird. Beide Flags werden immer bestimmt und das Programm muss später das richtige Flag verwenden.

**Weitere Flags:** Keine abschliessende Liste

|  |  |
| --- | --- |
| ZF | Zero Flag wird immer gesetzt, wenn das Resultat 0 ist |
| SF | Sign Flag entspricht immer dem höchstwertigen Bit des Resultats |
| PF | Parity Flag wird immer gesetzt, wenn das niederwertigste Byte des Resultats eine gerade Anzahl an gesetzten Bits enthält |

**Condition Codes:** Einige Zustände von Flags oder deren Kombinationen werden als Condition Codes (CC) bezeichnet. Zu jedem Flag gibt es einen CC mit einem Buchstaben

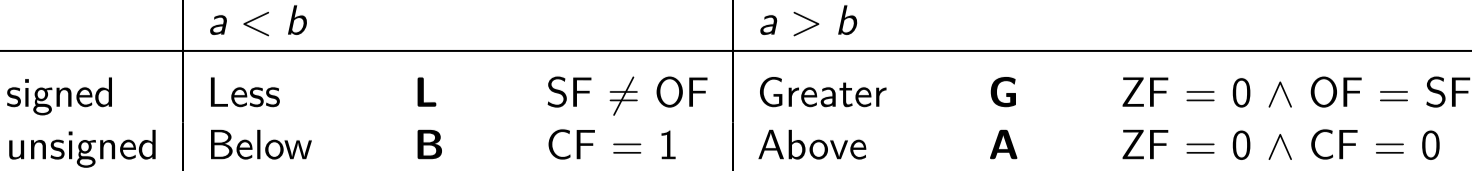
(C = CF, Z = ZF, etc.). Zu vielen CCs gibt es den negierten CC mit vorangestelltem N.

CC werden in bestimmten Befehlen dazu verwendet, eine Bedingung anzugeben. Der Befehl wird nur dann ausgeführt, wenn in diesem Moment die Flags genau so gesetzt sind, wie der Condition Code angibt.

**Beispiel:** Welche Flags werden gesetzt, wenn berechnet wird.

|  |  |  |  |
| --- | --- | --- | --- |
| Fall | Flags | Begründung |  |
|  | ZF = 1 |  | unsigned |
|  | CF = 1 | einen (unsigned) Überlauf |
|  | CF = 0 | heisst |
|  | ZF = 1 CF = 1 | heisst oder |
|  | ZF = 0 CF = 0 |  |  |
| Fall | Flags | Begründung |  |
|  | ZF = 1 |  | signed |
|  | OF SF |  |
|  | OF = SF | heisst |
|  | ZF = 1 OF SF | heisst oder |
|  | ZF = 0 OF = SF | heisst |  |

**Intel Condition Codes:**

****

**Cmp:** Entspricht der Operation sub, verändert aber den Operanden nicht sondern setzt nur die Flags.

**Test:** Entspricht der and Operation, welche den Ziel-Operanden nicht schreiben, sondern nur die Flags setzt

**Bedingte Anweisung:** Auf Intel-CPU gibt es 3 Familien von bedingten Operationen. todo

**Relative Sprünge:** Der Befehl JMP d setzt RIP RIP + d bezogen auf RIP nach Befehl selbst. In Assebler kann man ein Label verwenden und der Assembler berechnet die Adresse.

**Bedingte Sprünge:** Bedingte Sprünge benötigen einen Condition Code und sprint nur wenn der CC erfüllt ist. Ansonsten fährt der Prozessor mit nächsten Instruktion fort.

Graphical user interface

Description automatically generated with low confidence

**While-Schleife:** Vor jedem Ausführen des Body wird Bedingung geprüft. Ist Bedingung nicht 0, wird der Body ausgeführt.

A picture containing graphical user interface

Description automatically generated

**For-Schleife:** Vor der Schleife wird initial ausgeführt. Vor jeder Ausführung wird condition geprüft. Nach jedem Ausführen wird iteration ausgeführt.

**Pointer-Addition:** Die Addition eines Integers zu einem T\*-Pointer berücksichtigt die Grösse sizeof(T). Bei int \*p = 0x20; ist p+1 == 0x24 und p+2 == 0x28, etc.

Diagram

Description automatically generated

In C iteriert man gern mit Adressen. Ein T\*-Pointer wird mit ++ um sizeof(T) erhöht.

Diagram

Description automatically generated

# funktionen

**Stack:** Umfasst einen Stackpointer und Operationen push und pop. Der Stackpointer zeigt immer auf das zuletzt abgelegte Element. Elemente auf dem Stack sind je 8 Byte gross. Push und Pop müssen immer gleichmässig verwendet werden, sonst Stack durcheinander.

**Push:** Kopiert das Element auf den Stack und passt Stackpointer an

Diagram

Description automatically generated with medium confidence

**Pop:** Kopiert das oberste Element und passt Stackpointer an. Stackelement wird nicht entfernt.

A picture containing diagram

Description automatically generated

**Call a:** Nachfolgende Adresse r auf den Stack und Sprung an a. Kennt keine Parameter oder Rückgabewerte. Sichert weder Register noch Stack oder Flags. Funktion findet den Prozessor in dem Zusatnd vor, in dem der Aufrufer hinterlässt.

**Ret:** Adresse r vom Stack und Spring an r

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**Argumente:** Werden nacheinander auf den Stack gepusht. Rücksprungadresse kommt zuletzt auf den Stack durch CALL. Argumente liegen unter der Rücksprungadresse und können nicht gepoppt werden. Stattdessen bleiben Argumente bis zum RET auf dem Stack

**Frame Pointer:** Durch konsekutive CALL Operationen springt der Stackpointer ständig hin und her. Wir benötigen einen stabilen Pointer: den Frame Pointer.

A picture containing circle

Description automatically generated A picture containing text, gauge

Description automatically generated

**Calling Convention:** Vereinbarung zwischen Caller und Callee wo Argumente und Rückgabewerte übergeben werden und welche Register die Funktion ändern darf.

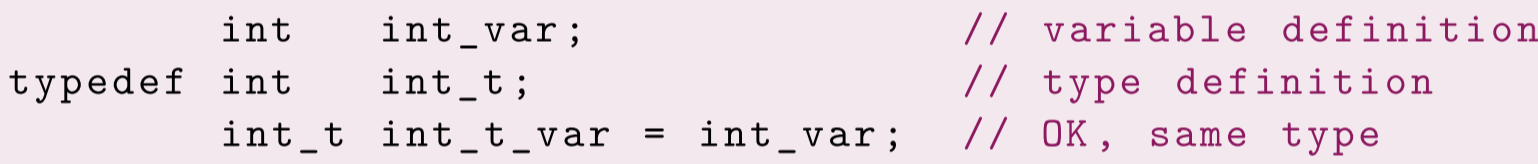
**Zwischenergebnisse:** Fall man mehr Zwischenergebnisse benötigt als man Register hat, kann man diese im Stack speichern. Man alloziert den benötigten Speicher und speichert Zwischenergebnisse über dem Framepointer.

A picture containing text, device, gauge, meter

Description automatically generated 

# datentypen

**Typen-Alias:** Mit dem Schlüsselwort typedef kann ein bestehender Typ einen weiteren Namen (Alias) erhalten. Compiler behandelt beide Aliase gleich



**Interpretation von Adressen:** In Assembler sind Adressen immer untypisiert. In C-äquivalent dazu ist void\*.

A picture containing table

Description automatically generated**Pointer-Arithmetik:** Bei Addition/Subtraktion von T\*-Adressen mit Integern wird die Grösse von T berücksichtigt.

A picture containing text, clock, gauge

Description automatically generated

**Arrays:** T a[n] reserviert Speicher für n Elemente vom Typ T und assoziiert das Label a mit der Adresse des ersten Bytes. Können mit {...} initialisiert werden.

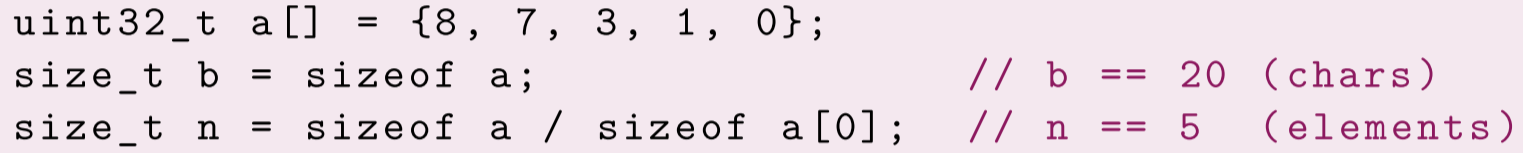
A picture containing diagram

Description automatically generated

Text

Description automatically generated

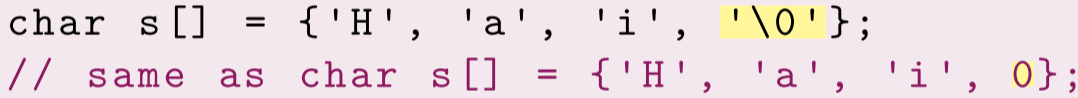
Sizeof gibt die Anzahl Maschinenbytes eines Arrays zurück. Die Anzahl Elemente erhält man durch Division durch Elementgrösse.



**size\_t:** Unsigned Integer-Datentyp, der gross genug ist, die Grösse beliebiger Objekte zu halten. Rückgabetyp von sizeof. Sicherer Datentyp fürs Iterieren über Arrays



**Null-terminierte Strings:** Char-Arrays, deren letztes Element '\0' (ASCII-Code 0) ist.



**Array als Parameter:** Arrays können nicht als ganzes (by value) übergeben werden. Stattdessen werden immer Adressen übergeben, auch wenn der Parameter als Array-Typ deklariert wird. Arraygrenzen werden ignoriert. Arrays enthalten keine Information über ihre Grösse,

A picture containing text, clock, gauge

Description automatically generated

**Strings als Parameter:** Da Strings null-terminiert sind, braucht es keinen zusätzlichen Grössen-Parameter. Strings werden immer als char \* übergeben

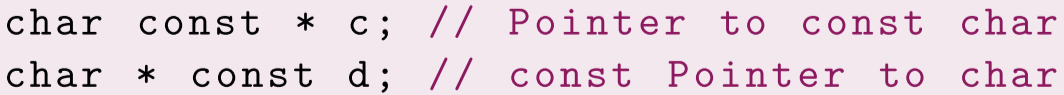
Text

Description automatically generated with low confidence

**const:** const bedeutet, dass der Wert nicht geändert werden darf. Der Wert kann sich aber (durch äussere Einflüsse) ändern. const bezieht sich immer auf den Typ links davon. Steht const ganz links, bezieht es sich auf den Typ rechts davon

Text

Description automatically generated





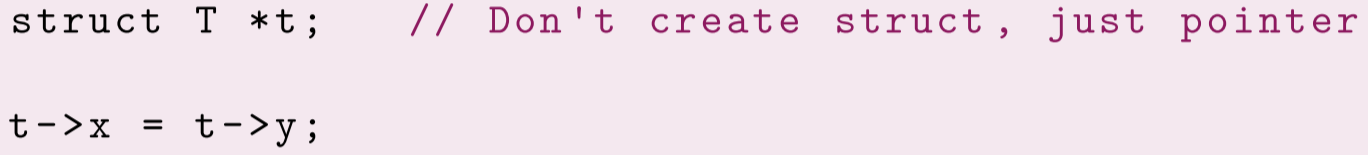
**String-Funktionen:** Strings werden in den meisten Fällen als char const\* übergeben. Potentiell gefährlich, falls kein '\0' kommt.

**Structs:** Globale Variablen können strukturiert werden und belegen den selben Speicherplatz wie wenn diese einzeln definiert wären. Der erste Member eines Structs hat immer dieselbe Adresse wie der Struct. Alle anderen haben grössere Adressen als ihr vorhergehender Member. Member müssen nicht dicht liegen, der Compiler darf Padding einfügen

Text

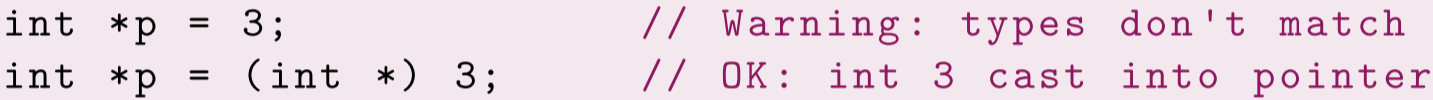
Description automatically generated

Ein Struct kann ein Tag T erhalten. Der Typ der Variable heisst dann struct T. Für Pointer, die die Adresse eines Structs enthalten, gibt eis einen eigenen Member Opeartor ->



**Implizite Konvertierung:** Integers werden immer implizit ineinander umgewandelt. Implizite Konvertierung sollte vermieden werden, ist manchmal aber nötig, z.B. zwischen char und int. Die Verwendung verschiedener Operandentypen (z.B. signed und unsigned) in einer Operation sollte generell vermieden werden.

**Cast:** Ein Cast (T) x ändert den Typ des Ausdrucks x auf T.

****

Casts sind z.B. bei der Multiplikation für hinreichende Präzision nötig:

# hauptspeicher und cache

Software-Basis-Modell: Der Prozessor lädt Befehlssequenzen aus dem Hauptspeicher und führt diese nacheinander aus. Befehle können Daten zwischen Hauptspeicher und Registern verschieben, auf Registern rechnen oder den Instruction Pointer verändern.

Das Software-Basis-Modell ist die Schnittstelle zwischen Compiler und Prozessor. Compiler generiert Befehlssequenzen. Prozessor führt die Befehlssequenzen aus. Beide Seiten optimieren für sich, das Modell bleibt dasselbe.

**Hauptspeicher:** Speicher ist eine feste Menge an Speicherzellen, die jeweils einen Wert speichern können. Zugriff auf den Speicher über den Speichercontroller. Kommunikation zwischen Speichercontroller und Prozessor über den Speicherbus

Text

Description automatically generated with medium confidence Diagram, schematic

Description automatically generated

**Wert an Adresse speichern:** CPU legt Adresse und Wert auf den Speicherbus und der Speichercontroller schreibt Wert in Zelle an der Adresse

**Wert von Adresse lesen:** CPU legt Adresse auf den Speicherbus. Speichercontroller liest Wert aus Zelle an Adresse und legt sie auf Speicherbus. CPU liest Wert vom Speicherbus.

**Physikalische Grenzen in der Computertechnik:** In der Praxis propagiert elektrische Energie nur mit 2/3 Lichtgeschwindigkeit. Je schneller Transistoren schalten, desto mehr Strom wird in Hitze umgesetzt. Grosse Teile der CPU sind daher meist ausgeschaltet.

|  |  |
| --- | --- |
| **Lokalitätsdiagramm:** Zeigt Speicher-Zugriffe im zeitlichen Verlauf. Jeder Zugriff auf eine Speicherstelle s zum Zeitpunkt t wird an der Koordinate (t, s) ausgezeichnet | **Arbeitsbereich eines Programms:** Der Arbeitsbereich W(t, ∆t) eines Programms sind alle Speicherstellen, die im Zeitraum ∆t vor dem Zeitpunkt t verwendet wurden, also im Zeit-Intervall [t - ∆t, t] |
| Chart, scatter chart  Description automatically generated | Chart, scatter chart, bubble chart  Description automatically generated |

**Lokalitätsprinzip:** In realen Programmen ist W(t,∆t) auch für grössere ∆t konstant. Dieses ∆t kann aber für verschiedene Progamme sehr unterschiedlich ausfallen.

**Konsequenz:** Rückblick auf W(t,∆t) ermöglicht Vorhersage über W(t + n, ∆t)

* Voraussetzung für den Nutzen einer Speicherhierarchie: Wären alle Speicherzugriffe unvorhersehbar, brächte Zwischenspeichern geringen Vorteil. Man bräuchte nur langsamste Speicherstufe. Systeme wären um Grössenordnungen langsamer
* Programmierer können ∆t aktiv beeinflussen – möglichst gross halten, z.B.: Arrays systematisch durcharbeiten, Modular programmieren (mit grosser Kohärenz)

|  |  |
| --- | --- |
| **Zeitliche Lokalität** | **Räumliche Lokalität** |
| Dieselbe Speicherstelle bleibt längere Zeit im Arbeitsbereich. Wenn Speicherstelle A jetzt benötigt wird, dann wahrscheinlich auch demnächst. | Arbeitsbereich enthält nahe beieinander liegende Speicherstellen. Wenn Speicherstelle A jetzt benötigt wird, dann demnächst wahrscheinlich auch A+1, etc. |
| Shape  Description automatically generated with medium confidence | A picture containing shape  Description automatically generated |

## cache

Cache ist ein Zwischenspeicher, der kleiner und schneller ist. Nutzen ergibt sich aus dem Lokalitätsprinzip.

|  |  |
| --- | --- |
| **Cache-Grösse** | Anzahl Nutz-Bytes (Payload) |
| **Cache-Hit** | Gesuchte Adresse ist im Cache |
| **Cache-Miss** | Gesuchte Adresse ist nicht im Cache |
|  | Zugriffszeit auf den Cache |
|  | Zugriffszeit auf den Hauptspeicher |
|  | Wahrscheinlichkeit eines Cache-Hits (wegen Lokalitätseffekt oft > 0.9) |

**Durchschnittliche Zugriffszeit:**

**Fully-Associative Cache mit 64-Byte Cachezeilen:** Jeder der Cacheeinträge enthält: Tag (obere n-6 Bits), Datenbytes die im HS abgelegt sind

Diagram

Description automatically generated

**Lookup Beispiel im FAC:** Bits je Adresse sei n = 32, Tag umfasst n – 6 = 26 Bit. Gesucht sei Wert an Adresse a\* = 84480801h. Tag von a\* ist t\* =

Offset j\* = 1.

Diagram

Description automatically generated

Fully-Associative Caches nutzen den Lokalitätseffekt best-möglich aus, da alle arbeiten parallel, dadurch sehr hohe Geschwindigeit. Aufwändig zu implementieren, da viel Hardware

**Direct-Mapped Cache:** Der DMC ist sehr einfach zu implementierender Cache mit einem  
schnellen Lookup. Hat aber viele Kollisionen. Die Adresse besteht aus Tag, Index, Offset.

Diagram

Description automatically generated

**k-way Set-Associative Cache:** Eine parallele Verwendung von k Direct-Mapped Caches mit je-  
weils P / k Einträgen. Jede Chachezeile kann in k verschiedenen Cacheeinträgen gespeichert werden (Set). Jeder DMC in diesem SAC heisst Way.Der SAC ist ein Kompromiss zwischen FAC und DMC: weniger komplex als FAC, weniger Kollisionen als DMC, genauso schnell wie FAC und DMC. Eine Stelle teilt sich (HS ∗ ways / Cachegrösse mit WAY-stellen den Platz).

**Formeln:**

|  |  |
| --- | --- |
| Adresse | Bits für Adresse |
| Cachezeilen | Cache / Byte Cachezeile = Cachezeilen |
| Index | Index Bits (Nur für DMC)  Index Bits (Nur für SAC, wobei k = Anzahl Ways) |
| Offset | Offset Bits |

**Fully-Associative Cache:** Da FAC kein Index/Eintrag hat, ist der Offset 6 Bit und Tag 26 Bit lang.

**Direct-Mapped Cache:** 6 Bits für Offset, 13 Bits für Index und 13 Bits für Tag

**2-way Set-Associative Cache:** 6 Bits für Offset, 12 Bits für Index und 14 Bits für Tag

# dynamischer speicher

**Heap:** Vollständig dynamischer Speicher, welcher vom OS verwaltet wird. Beliebiger Zeitpunkt für Reservation/Freigabe/Lebensdauer. Wünschenswert sind beliebige Grösse der Speicherblöcke, Minimale Metadaten/Zeit-Overhead/Daten-Overhead.

**Speicherfreigabe:** Reservierter Speicher muss freigegeben werden, sonst Memory Leaks. Explizit, wenn Programmierer definiert, wann Speicher freigegeben wird. OS stellt nur explizite zur Verfügung. Interne muss von Applikation eingebracht werden.

**Implizite Speicherfreigabe:** Implizit wird automatisch freigegeben, wenn sicher nicht mehr benötigt wird. Garbage Collector kann Objekt entfernen, wenn kein Pointer darauf weist. Keine direkten Leaks, aber keine Kontrolle über Freigabe und Zeitverhalten indeterminist.

**Explizite Speicherfreigabe:** Programmierer bestimmt Zeitpunkte von Reservation und Freigabe explizit. In C malloc und free. (void\* malloc(size\_t s), void free(void \*p))

**Einfache Implementierung:** Grösse direct vor dem Block size\_t \*s = p - sizeof (size\_t). Free(p) gibt Daten/Metadaten frei. P wird nicht verändert. Bei doppelter Freigabe enthält \*s einen beliebigen Wert und gibt damit irgendetwas frei.

Diagram

Description automatically generated

Diagram

Description automatically generated with medium confidence**Interne Fragmentierung:** Heap-Implementierung reserviert grösseren Speicherblock als nötig, bei dem der Verschnitt unverwendet bleibt. Mögliche Lösung ist die Verwaltung selber übernehmen (Java, .NET Runtime)

**Externe Fragmentierung:** Programm reserviert Speicher und gibt ihn unregelmässig frei. Über längere Zeit entsteht ein Speicherbild, indem es nur noch kleine Löcher gibt. Programmierer können das Problem im Programm umgehen durch eine Vorallozierung vieler Objekte am Anfang des Programms (Object Pool).

A picture containing diagram

Description automatically generated**Mehrfache fester Blockgrösse:** Implementierung definiert grundlegende Blockgrösse b. Speicher wird nur in Bereich reserviert. Blöcke liegen dicht ohne Zwischenraum. Anfangsadressen von Blöcken können wie Elemente in Array schnell bestimmt werden. Grösse des Blocks ist ein Tradeoff, der massgeblich die Performance bestimmt.

**Speicherblock-Verwaltung mit Bitlisten:** In Bitliste ein Bit je Speicherblock (0=frei, 1=verwendet). Anzahl der benötigten Bits ist Blockgrösse. Um freien Speicherbereichs zu finden, muss man eine Sequenz von ausreichend vielen Nullen finden (= angefragte Speichergrösse / Blockgrösse). Je weiter hinten der freie Platz, desto längere Suche.

Diagram

Description automatically generated**Speicherblock-Verwaltung mit verketteten Listen:** Sequentielles Durchsuchen nach ausreichendem freien Bereich. Schnelles Zusammenführen freier Bereiche. Keine aufeinanderfolgende freie Bereiche, aber aufeinanderfolgende verwendete Bereiche.

**Rekombination freier Bereiche:** Es gibt 4 Fälle bei Freigabe eines Bereichs

|  |  |
| --- | --- |
| Kein Nachbar frei | Keine Rekombination |
| Vorheriger Nachbar frei | Erweitere vorheriges Element, entferne dieses Element |
| Nachfolgend Nachbar frei | Erweitere dieses Element, entferne nachfolgendes |
| Beide Nachbarn frei | Erweitere vorheriges Element, entferne dieses und nachfolgendes |

**Suchalgorithmen für freie Bereiche:**

|  |  |
| --- | --- |
| First Fit | Wählt erste passende Lücke am Anfang.  Ansammlung vieler kleiner Lücken am Anfang |
| Next Fit | Wählt erste passende Lücke nach zuletzt reserviertem Bereich.  Kleine Lücken überall, schlechter als First Fit |
| Best Fit | Durchsucht alle Lücken und wählt die kleinste passende aus.  Grösster Speicherverschnitt, sehr langsam |
| Worst Fit | Durchsucht alle Lücken und wählt die grösste aus.  Ebenso langsam und auch keine guten Ergebnisse |

**Verfahren Grössenklassen:**  Bereiche werden nur in bestimmten Grössen zur Verfügung gestellt (z.B Alle Zahlen von 1 bis m oder Zweierpotenzen von bis )

**Quick Fit:** Schnelle Reservation da erstes Element aus der Liste der kleinsten passenden Grösse. Nachbarn dafür nicht leicht zu finden und somit Rekombination aufwendig

Diagram

Description automatically generated**Buddy System:** Grössenklasse mit Zweierpotenz von bis . ist kleinste Speichergrösse, ist gesamter Speicher. Ein freier -Bereich wird immer zuerst aus der entsprechenden Liste zugewiesen, sonst wird Bereich rekursiv halbiert.Benachbarte freie Bereiche werden rekursiv zusammengefügt, falls diese zusammengehören. Minimale Grösse: 4KB, Speichergrösse: 32KB, Allokation von 7KB

# virtueller speicher (hardware)

Programme werden als Prozesse ausgeführt. Jeder Prozess benötigt Hauptspeicher. OS weist Prozessen Hauptspeicher zu und schützt den Hauptspeicher der Prozesse gegeneinander. Kein Schutz innerhalb eines Prozesses.

**Privilage Levels (PL):** Intel 64 hat PL0 bis PL3. Programe laufen auf PL3.

**Virtuelle Adressen:** OS gibt Prozesse virtuelle Adresse. Prozess kennt nur virtuelle Adresse. Memory Management Unit (MMU) übersetzt in reale Adresse. OS konfiguriert MMU per Prozess. Prozess kann Adressübersetzung nicht erraten (Kein Zugriff auf Mapping)

**Gültiger Zugriff:** Prozess will auf gültige Adresse zugreifen, MMU findet Mapping und legt reale Adresse auf Speicherbus, Prozessor liest/schreibt Daten von/auf Speicherbus.

**Ungültiger Zugriff:** Prozess greift auf ungültige Adresse zu. MMU stellt fehlendes Mapping fest und signalisiert Fault-Interrupt. CPU ruft OS-Interrupt-Handler auf und OS übernimmt.

**Flexible Adressräume:** Prozesse sollen beliebig im realen Speicher verschoben/ausgelagert werden können. Prozess merkt davon nichts. Mehr Speicher pro Prozess.

Shape, rectangle

Description automatically generated**Seitenbasierter Virtueller Speicher:** Page (typisch 4KB und 12 Bit Offset). Hauptspeicher besteht aus Frames mit gleicher Grösse wie Page. Frame Number = Startadresse des Frames ohne Offset-Bits. Virtueller Adressraum besteht aus Pages. Eine Page repräsentiert die Daten, ist kein Speicher, sondern benötigt einen Frame. Page Number = Startadresse der Page ohne Offset

**MMU:** Virtueller Speicher nur mit MMU. Nur MMU kann Speicherzugriffe überwachen. CPU legt Adressen nicht direkt auf den Speicherbus. MMU erzeugt reale Adressen. Virtuelle Adresse wird aufgeteilt in: Offset: Adresse innerhalb der Page, 12 LSBs. Page Number: Restliche Bits. Nur die Page Number wird auf Frame Number übersetzt.

Diagram

Description automatically generated

**Page Table:** Dient der MMU. Jede virtuelle Adresse wird via Page-Table in reale Adresse übersetzt. Wird vom OS konfiguriert.

Diagram

Description automatically generated with medium confidence**Single-Level Page Table:** Array mit einem Eintrag pro Page. Lookup sehr schnell. Grösse hängt von Grösse des virtuellen Adressraums ab. Prozess greift auf virtuelle Adresse 87654321 zu. MMU ermittelt Page Number 87654. MMU schaut im Array und findet ABDC0. MMU bestimmt reale Adresse ABCD0321

Diagram

Description automatically generated**Two-Level page Table:** Page Number wird in Directory/Page Table Idx. aufgeteilt. Es gibt viele Page Tables. Alle Pages mit gleichem Directory Index sind in derselben Page Table. Prozess greift auf virtuelle Adresse 87654321 zu. MMU ermittelt Directory = 21D. MMU schaut in Top-Level-Page-Table an Index 21Dh nach. Dort Pointer auf reale Adresse der Second-Level-Page-Table (SLPT) (= 2000h). MMU ermittelt relative Page Number = 254.

**Formeln Single-Level:**

|  |  |
| --- | --- |
| Page Offset | Page Bit Offset |
| Pages | virtueller Adressraum / Page = Pages |
| Page Frames | RAM / Page = Page Frames |
| Page Frame Nr | Bits für alle Page Frame Nr |
| Virtuelle Adresse | Bits (Kleinste Adr. = 0, Grösste Adr. = ) |
| Reale Adresse | Bits (Kleinste Adr. = 0, Grösste Adr. = ) |
| Page Table | Pages \* Page Frame Nr = Page Table Size in Bits |

**Formeln Multi-Level:**

|  |  |
| --- | --- |
| Page Offset | Page Bit Offset |
| Page Nummer | Virtuelle Adresse Bits – Page Bit Offset = Page-Nummer Bits |
| Pages | Pages |
| Page Frames | RAM / Pages = Page Frames |
| Page Table Entry | Virtuelle Adresse / 8 Bits/Byte = Page Table Entry Bytes |
| Einträge pro Page | Page / Page Table Entry = Einträge pro Page |
| Eintrag adressieren | Bits zum Eintrag adressieren |

Directory ist genau 1 Page gross und verwendet 32-Bit breite Einträge

|  |  |
| --- | --- |
| Page Table pro Prozess | Je Eintrag im Directory kann es eine Page-Table geben, also 1K. |
| Pages pro Prozess | Page-Table hat 1K Einträge und für jeden Eintrag kann es eine Page geben. Demzufolge kann es insgesamt 1K · 1K = 1M Pages geben. |
| Virtuelle Adressen pro Prozess | Jede Page umfasst 4K virtuelle Adressen, über alle Pages (den gesamten virtuellen Adressraum eines Prozesses) gibt es somit  4K · 1M = 4G virtuelle Adressen. |

**Beispiel:**



Icon

Description automatically generated

Table Idx. Directory Idx

# virtueller speicher (software)

**Page-Table (x86):** Jeder Page-Table-Eintrag hat 32 Bit. Unterstes Bit = P-Bit («Present»)



MMU setzt A-Bit (Accessed) bei jedem Zugriff auf Page und D-Bit (Dirty) bei jedem Schreibzugriff auf Page. Nur das OS kann beide Bits löschen.

**P:** Falls P=1, dann ist Page im Hauptspeicher. Falls 0, ist Page nicht im Hauptspeicher, MMU wirft Interrupt und schaut andere Bits nicht an. OS kann frei bestimmen, was es speichert (Informationen, um Page im sekundären Speicher zu finden). Wenn diese Info ungültig ist, dann Segmentation Fault.

**Paging:** Ablauf im Interrupt: OS prüft, ob Lokalisierungsinformationen gültig sind (wenn nicht: Seg. Fault). Wenn Hauptspeicher voll: OS entscheidet, welche andere Page entfernt wird. OS initiiert Page-Transfer von sekundärem Speicher. In der Zwischenzeit können andere Prozesse laufen. Sobald die Page geladen wurde: OS setzt Prozess mit der Instruktion fort, wegen der die MMU den Interrupt warf. Prozess wiederholt Zugriff. MMU kann Zugriff nun bedienen, da Present-Bit gesetzt.

**Dreschen:** Häufiges Pagen, wenn Hauptspeicher viel zu klein ist, zu viele Prozesse gibt oder System mehr und mehr mit Pagen beschäftigt ist. Verminderung durch mehr Hauptspeicher, Laststeuerung oder Paging-Strategien

**Paging-Teilstrategien:** Ladestrategien (fetching policies), Entladestrategien (cleaning policies)

und Verdrängungsstrategien (page replacement policies)

# ladestrategie

**Demand Paging:** Laden auf Anfrage: Pages werden nur dann geladen, wenn sie benötigt werden. Erfordert jedes mal einen Interrupt. Minimaler Aufwand, jedoch lange Wartezeiten.

**Prepaging:** Pages frühzeitig vor Verwendung laden. System versucht zu ermitteln, welche Pages benötigt werden. In Praxis in Reinform kaum anzutreffen.

Diagram

Description automatically generated**Demand Paging mit Prepaging:** Wie Demand Paging aber benachbarte Pages werden mitgeladen (Nach Lokalitätsprinzip). Vorteil sind weniger Page-Faults und Blocktransfer jedoch werden möglicherweise nicht benötigte Pages mitgeladen. In Praxis überwiegen Vorteile.

## entladestrategien

**Demand Cleaning:** Entladen auf Nachfrage. Page wird zurückgeschrieben, wenn Frame wiederverwendet wird. Minimaler Aufwand, jedoch erhöhte Wartezeiten.

**Precleaning:** Vorrausschreibendes Schreiben. Modifizierte Pages werden frühzeitig in den sekundären Speicher geschrieben. Reduzierte Wartezeit, jedoch mehr Aufwand.

Diagram

Description automatically generated**Beispiel:** Eine Liste mit Page Nr der unveränderten Pages (U) und eine mit veränderten Pages (M). MMU setzt D-Bit beim Schreiben auf die Page. OS läuft periodisch im Hintergrund. «Verschiebt» Pages mit Dirty Bit von U nach M. Schreibt Pages von M in sekundären Speicher und «verschiebt» sie von M nach U.

## Verdrängungsstrategien

**Grundlagen:** MMU setzt Statusbits, OS löscht sie. Dirty-Bit (Page im Hauptspeicher anders als im sekundären Speicher) wird von MMU bei jedem Schreibzugriff gesetzt. Accessed-Bit (Page wurde kürzlich im Prozess verwendet) wird von MMU bei jedem Lese-/Schreibzugriff gesetzt.

|  |  |  |
| --- | --- | --- |
| A = 0 | D = 0 | Es gab seit längerer Zeit keinen Zugriff |
| A = 0 | D = 1 | Vor kurzer Zeit gelesen |
| A = 1 | D = 0 | Vor kurzer Zeit geschrieben |
| A = 1 | D = 1 | Vor längerer Zeit geschrieben |

**Optimal:** Ersetze Seite, die am spätesten in der Zukunft gebraucht werden wird. In der Praxis nicht umsetzbar, da Zukunft nicht völlig bekannt.

**FIFO:** Ersetze jeweils die älteste Seite. Benötigt keine Statusbits. Aber alte, häufig benutzte Seiten Werden entfernt und gleich wieder geladen.

Calendar

Description automatically generated

**Second Chance:** Erweiterung von FIFO. Prüft A-Bit der ältesten Page. Wenn 0: Page alt und nicht benutzt, löschen. Wenn 1: OS löscht A-Bit und schiebt Page ans Ende der Linked-List, dann weiter mit nächster Page. Entfernt älteste nicht verwendete Page

**LRU:** Ersetze am längsten unbenutzte Page. Bei jedem Zugriff: MMU notiert Zeitpunkt T in Page-Eintrag in Page-Table. Page mit kleinstem T wird ersetzt. Nah am Optimum, jedoch grosser Aufwand in Hardware, da Page-Einträge grösser werden.

Calendar

Description automatically generated

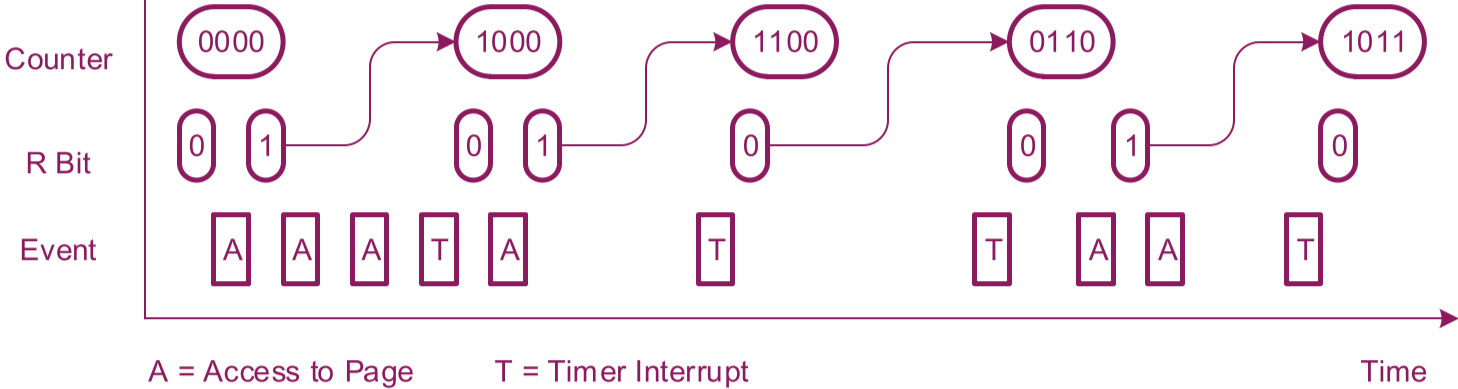
**Verdrängungsstrategien mit Interrupt:** Wenn die HW die Zeit nicht messen kann, was kann man tun? OS kann die Zeit messen: OS registriert Timer-Interrupt (z.B. alle 20 ms). OS kann im Timer-Interrupt R-Bits aller Pages auswerten und löschen

**NFU:** Benötigt zusätzliche Counter Table: Pro Page-Table-Eintrag ein Counter am selben Index. Zählt Intervalle, in denen es Zugriff gab. Wenn es einen Zugriff gab, Counter erhöht

A picture containing text, clock

Description automatically generated

**NFU mit Aging:** Counter gewichtet nach Zeit. Pro Page ein n-Bit Counter. Im Timer-Interrupt: Füge A oben an und verliere ältestes Bit.



**Working Set:** Behalte Pages vom Working Set (mit Intervall T). Pro Page-Table-Eintrag ein Zeitstempel t. Statt Timer-Interrupt: Scanne alle Pages im Fault-Interrupt

A picture containing diagram

Description automatically generated

# smol hack

**Potenzen:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 512 | 4 | 16 |  | 10 | K |
| 8 | 256 | 3 | 8 |  | 20 | M |
| 7 | 128 | 2 | 4 |  | 30 | G |
| 6 | 64 | 1 | 2 |  | 40 | T |
| 5 | 32 | 0 | 1 |  |  |  |