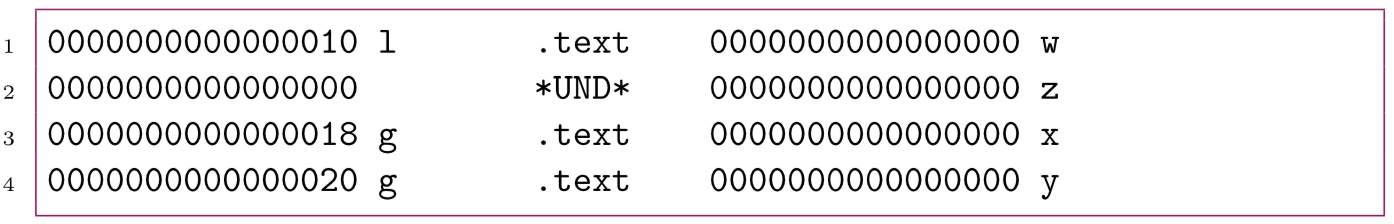
Adressbus, Datenbus, Steuersignale. Eine Instruktion ist die Kombination einer Operation mit Operanden. «Kopiere Wert von Adresse 4000h in Register 2»

## Assembler

Maschinencode

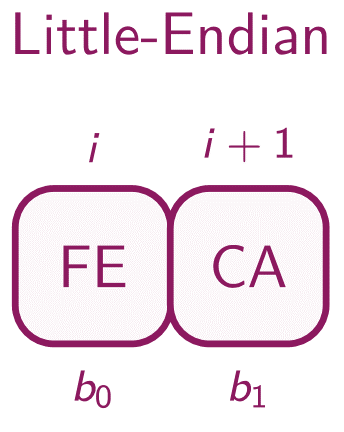
Die Gesamtheit aller möglichen Instruktionscodes eines Prozessors (Maschine) ist sein Maschinencode.   
Sequenzen von Maschinencode sind im Hauptspeicher abgelegt. Der Prozessor hat ein spezielles Register, dass die Adresse des nächsten Befehls enthält (Befehlszeiger, instruction pointer IP, program counter PC)

Intel Terminologie

|  |  |
| --- | --- |
| Byte | 8 Bit (kleinste Einheit) |
| Word | 2 Byte / 16 Bit |
| Doubleword (Dword) | 4 Byte / 32 Bit |
| Quadword (QWord) | 8 Byte / 64 Bit |
| Double Quadword (DQWord) | 16 Byte / 128 Bit |

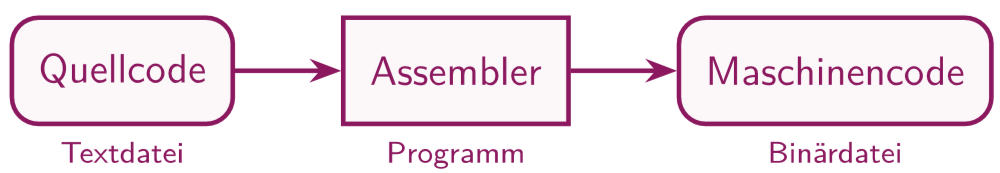
Darstellung einzelner Bytes

Ein Byte hat 2 hexadezimale Stellen und : Eine Stelle ist nicht einzeln zugänglich. Die höherwertige Ziffer steht immer links: . zB. die Zahl bedeutet immer

Byte Order bei Words (Little-Endian bei Intel)

* Ein Word (16 Bit) hat 4 hexadezimale Stellen: bis .
* 2 aufeinanderfolgende Bytes im Speicher und haben Adressen und .
* Die Stellen innerhalb eines einzelnen Bytes werden nie vertauscht.

-----------------------------------------

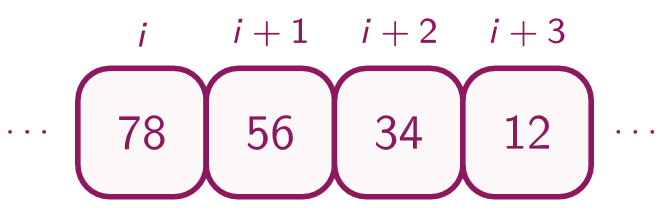
Assembler

**Der Assembler:**   
Programm, das textuelle Befehle in Maschinencode übersetzt.  
**Assembler (Sprache):** Ist die formale (nicht einheitliche) Sprache.   
**Spezifikation von Daten:** Die einfachste Anweisung ist die direkte Spezifikation von Datenbytes (db/dd/dq = define Byte / DWord / QWord)

|  |  |
| --- | --- |
| Quellcode | Resultat im Binär-Output |
| db 48 | Das Byte |
| db 0x35,0h21,049h | Die drei Bytes |
| db ‘a’ | Das Byte mit dem ASCII-Code von a = |
| db ‘Hallo’ | Die ASCII-Codes von H, a, l, l und o |

-----------------------------------------

Referenzen in Assembler

Nummeriert man die Bytes in der Ausgabe-Sequenz fortlaufen, erhält jedes Byte einen Offset (Adresse, Index)

dd 0x12345678

**$** bezeichnet den aktuellen Offset, bevor die von der aktuellen Zeile generierten Bytes in die Ausgabe-Sequenz geschrieben werden. Ein Label wird mit Label: erstellt

my\_text: db ‘BSys 1?!’

current: dq $

erzeugt, weil my\_text 8 Bytes lang ist:

42 53 79 73 20 31 3f 21 | **08** 00 00 00 00 00 00 00

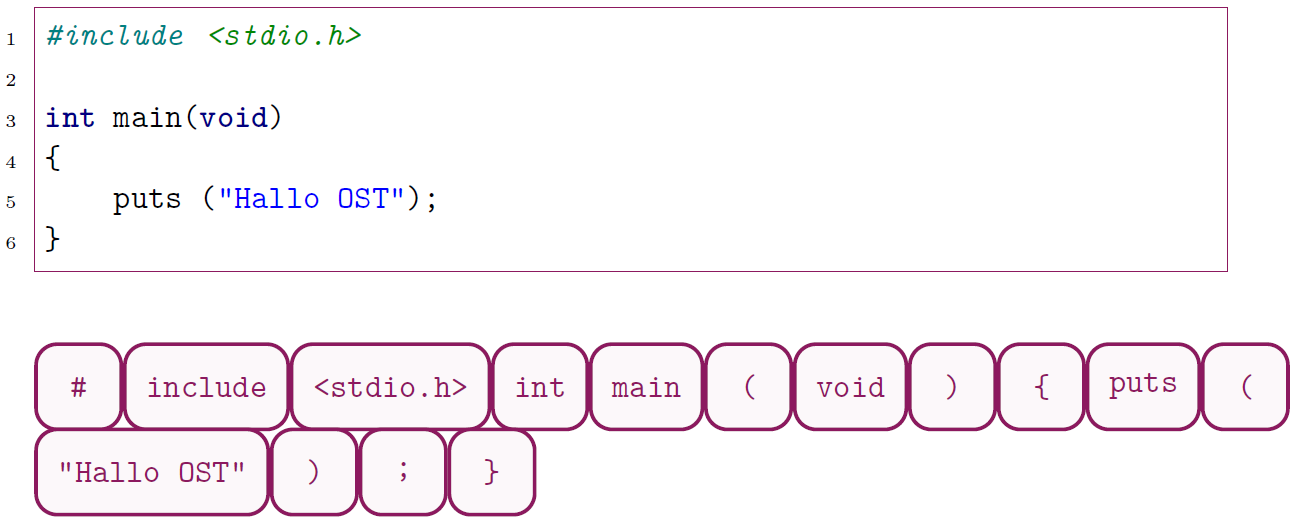
Intern assoziiert der Assembler das Label mit dem Offset des nachfolgenden Befehls in der Ausgabedatei.   
A(my\_text) = 0 und A(current) = 8

**Label-Arithmetik:** NASM kann während der Übersetzung einfache Arithmetik ausführen (da Labels vor der Operation definiert werden).

length: dq after\_my\_text – my\_text

my\_text: db ‘BSys 1?!’

after\_my\_text:

erzeugt:

08 00 00 00 00 00 00 00 | 42 53 79 73 20 31 3f 21

## Variablen

**CPU Register:** Prozessoren besitzen kleine interne Speicherzellen, genannt Register. Operationen können diese als In/Output verwenden.   
**General Purpose Register:** Kann nach Belieben verwendet werden.

**Special Purpose Register:** Wird von der CPU für einen spezifischen Zweck verwendet, nur gewisse Operationen können darauf zugreifen.

* **RIP:** Instruction Pointer, zeigt auf nächste Instruktion
* **RFLAGS:** Flags, die spezifische Situationen markieren

-----------------------------------------

Intel 64 Instruktionen

**Länge der Instruktion:** Binärzahlen, können 1 bis 15 Byte lang sein. Die Länge einer Instruktion ist nicht in der Sequenz enthalten. Eine Sequenz muss von Anfang an Instruktion für Instruktion durchgegangen werden, um jede Instruktion richtig decodieren zu können.

**Datentransfer-Operationen**

**mov Ziel, Quelle:** Kopiert in das Ziel aus der Quelle.

|  |  |
| --- | --- |
| mov rax, rbx | Setze Inhalt von rax gleich rbx |
| mov rax, 0x8000 | Setze rax gleich |
| mov rax, [0x8000] | Setze Inhalt von rax gleich Inhalt von |
| mov qword [0x8000], 5 | Setze Inhalt von gleich 5 (Angabe von Operandengrösse ist optional) |

Zu beachten: Man kann nicht direkt von Speicher in Speicher kopieren und die Operanden müssen gleich gross sein.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Nr.** | **63 … 0** | **31 … 0** | **15 … 0** | **15 … 8** | **7 … 0** |
| **0** | RAX (Accumulator) | EAX | AX | AH | AL |
| **1** | RCX (Counter zb für Schleifen) | ECX | CX | CH | CL |
| **2** | RDX (Pointer für I/O) | EDX | DX | DH | DL |
| **3** | RBX (Generischer Pointer) | EBX | BX | BH | BL |
| **4** | RSP (Stackpointer) | ESP | SP | - | SPL |
| **5** | RBP (Basepointer) | EBP | BP | - | BPL |
| **6** | RSI (Quellindex Stringops) | ESI | SI | - | SIL |
| **7** | RDI (Zielingex Stringops) | EDI | DI | - | DIL |
| **8-15 (n)** | Rn (Zusätzliche Register) | RnD | RnW | - | RnL |

-----------------------------------------

Adressierung von Hauptspeicher

Speicherstellen können auf verschiedene Weisen spezifiziert werden. Die Adressierungsmodi können auch beliebig addiert werden.

|  |  |
| --- | --- |
| mov rax, [0x8000] | ***Displacement:*** Die Adresse der Speicherstelle folgt unmittelbar. |
| mov rbx, 0x8000 mov rax, [rbx] | ***Base:*** Die Adresse der Speicherstelle steht in einem Register r |
| mov rcx, 0x1000 mov rax, [rcx \* 8] | ***Scaled Index [i\*s]:*** i ist ein Register, s eine Konstante 1,2,4 oder 8 (skalierte Adresse) |

Mit lea (load effective address) kann auch eine Adresse durch Kombination von Addressierungsmodi berechnet werden.

-----------------------------------------

Linker

-Asm-Dateien Assembler -Objekt-Dateien Linker ExecutableDie Symboltabelle der Objekt-Datei enthält einen Eintrag pro Symbol. Erste Spalte: Offset, zweite Spalte: Symbolattribut (g für global, l für lokal), dritte Spalte: Sektion, letzte Spalte: Namen. Im gelinkten Objekt erhält jedes Symbol einen eigenen Platz. **Einsprungspunkt:** Muss in jedem .exe 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 Parameter werden rdi, rsi, rdx, r10, r8 und r9 verwendet.  
**Ende des Programms:** Programme müssen explizit beendet werden. Dazu gibt es den OS-Syscall exit (Code 60), der im Parameter den Exit Code übergibt.

**Rahmen eines Programms:**

extern \_start:

\_start:

…

mov rax, 60 ; syscall exit

mov rdi, 0 ; exit code (0 = successful, no error)

syscall

## C-Toolchain

Die Bestandteile der C-Toolchain sind: C Präprozessor, C Compiler, Assembler und Linker. C-Quelle Präprozessor Bereinigte C-Quelle Compiler Assembler-Datei Assembler Objekt-Datei Linker Executable

-----------------------------------------

C-Präprozessor

Der Präprozessor verarbeitet die Input-Datei in mehreren Durchläufen. In jedem Durchlauf verarbeitet er die gesamte Datei einmal.

***1. Durchlauf:*** Entfernen aller Kommentare und umwandeln fortgesetzter Zeilen, die mit \ enden, in eine einzige Zeile.  
***2. Durchlauf:*** Teilt den gesamten Text in Tokens ein. Es gibt 5 Klassen von Tokens: Bezeichner (identifier), Präprozessor-Zahlen, String- und Character-Literale, Operatoren und Satzzeichen (punctuators), sonstige.  
Ein Whitespace trennt Tokens voneinander, aber nicht alle Token müssen durch Whitespace getrennt werden.

**Bezeichner (z.B. Variablennamen):** Beginnt mit einem Buchstaben (a-zA-Z) oder \_ gefolgt von einer Sequenz aus Buchstaben, \_ oder Ziffern (0-9) a, \_, \_a, a0, \_0, \_a0, \_0\_0, \_\_\_\_\_, int, for, size\_t  
**Zahl:** Beginnt mit einer Ziffer, gefolgt von einer Sequenz aus Ziffern, Buchstaben, \_, ., oder Exponenten. Vor der ersten Ziffer kann auch ein Punkt stehen. 0, 1, 0123, 0x1234, .05, 0xE+12, 0\_.0 (für Präprozessor eine Zahl, für Compiler nicht)  
**String- und Character-Literale:** String-Literale beginnen und enden mit ”, Character-Literale mit ‘. Alle Zeichen dazwischen werden nicht weiter unterteilt. *“Hallo OST“, “Hallo \“OST\““, ‘0’, ‘\’’*  
**Operatoren und Satzzeichen:** Jede der folgenden Zeichen:

() [] {} . , : ; ? ... -> # ## = + - \* / % & | ^ ~ ! << >> == != < > <= >= && || \*= /= %= += -= ++ -- <<= >>= &= |= ^=

Der Präprozessor ist ***greedy***, d.h. er versucht immer das grösstmögliche Token zu bilden. Z.B. ist a+++++b nicht a++ + ++b sondern a++ ++ +b, obwohl das in C kein gültiger Ausdruck mehr ist.

3. Durchlauf: Der Präprozessor durchläuft die Token-Liste und führt Präprozessor-Direktiven aus und ersetzt Makros durch ihre Expansion. Präprozessor-Direktiven: Ist das erste Token auf einer Zeile #, wird das nächste Token als Direktive interpretiert. Beide Tokens werden entfernt und die entsprechende Direktiven ausgeführt. Die wichtigsten Direktiven sind: include (fügt einen Header ein), define (Definiert ein Makro), if, else, endif. #include: Der Präprozessor öffnet die Datei anhand des nächsten Tokens. #include <file.h> sucht nur in den Systemverzeichnissen, #include “file.h” sucht zusätzlich zuerst im aktuellen Verzeichnis. Präprozessor führt anschliessend Durchläufe 1 bis 3 für file.h durch. Danach setzt er die Arbeit in der Originaldatei fort. Der Präprozessor kann dadurch mehrere Dateien zu einer Translation Unit (TU) zusammenführen.  
Objektartige Makros: Haben keine Parameterliste. Der Präprozessor ersetzt im Programmtext nach der Definition des Makros jedes Token, das dem Makronamen entspricht, durch die Tokenliste. Infinite Recursion wird automatisch vermieden, da der eigene Makroname nicht weiter ersetzt wird. #define XYZ 123: Aus int y = XYZ; wird int y = 123;

-----------------------------------------

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 TU ist eine Folge von Deklarationen und Definitionen von Globalen Variablen, Funktionen und Typen.

**Bezeichner** (Variabelnamen)**:** Das Gegenstück zu Labels in C. Ein Bezeichner ist eine alphanumerische Zeichenkette, die nicht mit einer Ziffer beginnt und kein Schlüsselwort ist (*int*, *extern*, *static*, etc.)  
**Deklaration:** Verknüpft einen Bezeichner compiler-intern mit einem Typen. Haben keinen Einfluss auf den Byte-Stream. Innerhalb einer TU darf Bezeichner mehrfach definiert werden, solange identisch. Unterschiedliche TU’s können Bezeichner unterschiedlich deklarieren.  
extern int y;  
**Definition:** Erzeugt den Inhalt der deklarierten Entität. Darf nicht mehrfach definiert werden in TU, ausserhalb aber schon. Kann Fehlermeldung verursachen. int y = 5;  
**Globale Variablen in C:** Haben einen Typ und einen Bezeichner. Deklaration erfolgt durch Typ-Token gefolgt von Bezeichner-Token: int x; Für jede globale Variable wird Speicher fix reserviert. Alle Variablen eines Typs brauchen gleich viel Speicher. Bezeichner entspricht dem Label auf Assembler-Ebene. Globale Variablen werden standardmässig exportiert. Um eine Variable zu importieren, kann sie mit ***extern*** gekennzeichnet werden.  
**Static Variablen:** Variabel, die nicht exportiert werden soll. Wird mit static gekennzeichnet. static int c = 5;

extern-Variablen werden beim Import mit globalen Variablen zusammengeführt, bei static-Variablen geht das nicht, da kann nur auf die static-Variable zugegriffen werden. Es können auch Variablen mit unterschiedlichen Typen zusammengeführt werden, dies ist aber gefährlich (undefined behaviour). **Header:** stellt sicher, dass a.c und b.c jeweils dieselbe Deklaration sehen.  
Vergleich Bezeichnung Variablen in Objekt-Datei und C

|  |  |  |  |
| --- | --- | --- | --- |
| **Bez. In Obj.-Datei** | **Deklaration in ASM** | **Bez. In C** | **Deklaration in C** |
| lokal | - | global, internal Linkage | static |
| global | global | global, external Linkage | - |
| - | extern | extern | extern |

Variablen und Ausdrücke

**Objekt:** Zusammenhängender Speicherbereich, sein Inhalt kann als Wert interpretiert werden. Kann unterschiedlich interpretiert werden, z.B: signed / unsigned. Rückgabewerte von Funktionen werden ebenfalls in Objekten zurückgegeben. Jede globale Variable ist ein (benanntes) Objekt, aber nicht jedes Objekt ist eine Variabel (Rückgabewerte, Objekte hinter Pointern).  
**Wert:** Genaue Bedeutung des Inhalts eines Objekts, wenn dieses mit einem Typ versehen ist.  
**Typen:** Beschreiben Objekte bzw. deren Eigenschaften. Jeder Typ T hat eine feste Objektgrösse sizeof(T). C erzwingt Typgleichheit bei Operationen. Nach Übersetzung in Assembler gibt es keine Typen mehr, nur noch Speicher.   
**char:** ist identisch zu signed char oder unsigned char, aber eigener Typ (es gibt alle 3, aber haben dieselbe Funktion). Min. 8 Bit gross.  
**sizeof(T):** wird in Vielfachen von char angegeben.   
sizeof (char) = 1 (8 bit), sizeof(int) = 4 (32 bit)  
**Signed-Integer-Basistypen:** signed char 8 bit ; short (int) 16 Bit ; int 16 Bit und short ; long (int) 32 Bit und int ; long long (int) 64 Bit und long. Bei den letzten 4 Typen kann signed davor geschrieben werden und int kann auch weggelassen werden.   
Zu jedem Signed-Integer-Basistypen gibt es das Unsigned-Gegenstück, welches die gleiche Grösse hat. Statt signed wird dann unsigned geschrieben.   
**Abgeleitete Typen:** Array-Typen (mehrere Elemente vom gleichen Typ), Struct-Typen (Mehrere Elemente beliebiger Typen nacheinander), Union-Typen (Mehrere Elemente beliebiger Typen im selben Speicherbereich), Pointer-Typen (Adresse eines Objekts), Funktions-Typen (Adresse einer Funktion).  
**Pointer-Typen:** Auf Maschinencode-Ebene gibt es keine Variablen, sondern nur Adressen. Sind Binärzahlen und können selbst auch gespeichert werden. Die Adresse eines Objekts, dessen Typ nicht bekannt ist, ist vom Typ void \* (Void-Pointer). Die Adresse eines Objekts von Typ T ist vom Typ T \* (T-Pointer). Die Adresse einer Adresse von Typ T \* ist vom Typ T \*\*.

-----------------------------------------

Ausdrücke (Statements)

**Primäre Ausdrücke:** Konstanten, Literale & Bezeichner

Integer-Konstanten in C: Dezimalzahl (beginnt nicht mit 0), Oktalzahl (beginnt mit 0), Hexzahl (0x…) mit oder ohne Typ-Suffix *(l,ll,u,ul,ull)*

**Typ einer Integer-Konstante in C:** Typ ist jeweils der kleinste, in den die Konstante passt, aber mindestens int. 1 hat Typ int, 0x100000000 hat Typ long int (wenn long int 32 Bit und int 32 Bit). l verwendet mindestens long int, ll verwendet mindestens long long int. u wechselt auf den entsprechenden unsigned Typ.

int x = 14; int y = 27, int z = 82;

z = (y = x) + 1;   
// schreibe Inhalt von x in y und Inhalt von y + 1 in z

Verwendung von Zuweisungen als Ausdruck, Typkonversionen und casts sind unsauber und sollten vermieden werden.

Zuweisung C vs Assembler

Bei Verwendung eines Labels in Assembler wird die damit assoziierte Adresse verwendet. Ein Variablenname in einem C-Ausdruck meint den Inhalt der Variable. Auch der vom C-Compiler generierte Code verwendet die Adresse der Variable.

x = y;

mov rax, [y] ; [y] = Inhalt der Adresse, y = Adresse

mov [x], rax ; NICHT mov x, rax

Referenzoperator &

Erzeugt die Adresse eines Ausdrucks. &a ergibt die Adresse der Variable a. Ist T der Typ von a, dann ist T\* der Typ von &a. Wert Adresse

int x = 5; int \*px = &x;

Dereferenzoperator \*

Wandelt die Adresse in einen Ausdruck um, der für den Inhalt steht. Gegenstück zum Referenzoperator. Adresse \* Wert, ausser \* vor Typ heisst «ist Pointer auf Typ»

int x = 5;

int \*px = &x;//&x = Adresse des ints, \* = Pointer-Bezeichner

int y = \*px; // \*px = Wert einer int-Adresse, y = 5, \* = Dereferenzoper.

-----------------------------------------

Arithmetische und logische Operationen

Logische Funktion: Funktion von Bits (Inputs) auf 1 Bit (true/false)  
Parameter: Variable zur Übergabe von Werten an eine Funktion.   
**Argument:** Wert eines Parameters bei einer Verwendung der Funktion

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | AND (&&) | x && !y | !x && y | XOR (^) | OR (||) | !(x OR y) | EQU (==) | !y | X || !y | !x | !x OR y | **!(x && y)** |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | **1** |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | **1** |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | **1** |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | **0** |

: NAND (NOT AND) Nur dann 0, wenn x = 1 und y = 1

Bitweise Operatoren

Die bitweisen Operatoren entsprechen direkt den Assembler-Operationen. Das Resultat wird in z gespeichert. Bitwise NAND in C: *z = ~(x & y)*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ASM** | **not z** | **and z, q** | **or z, q** | **xor z, q** |
| **C** | z = ~ q | z = q & p | z = q | p | z = q ^ p |
| **Bsp** | not 101 = 010 | 111 and 101 = 101 | 111 or 101 = 111 | 111 xor 101 = 010 |

Logische Operatoren in C

Bedeuten etwas anderes als auf Assembler-Ebene. Der Typ dieser Ausdrücke ist immer int, der Wert immer 0 (false) oder 1 (true) z=!q, z=q&&p, z=q||p Logisches NAND in C: **z = !(x && y)**

Arithmetische Operationen in Assembler

Für Ziel z und Quelle q gelten die gleichen Regeln wie bei mov. c ist das Carry-Flag. adc = addition with carry, sbb = subtraction with carry

-----------------------------------------

Rechnen mit Binärzahlen

Ein Bild, das Schrift, Zahl, Reihe, Diagramm enthält.

Automatisch generierte BeschreibungAddition funktioniert genauso wie bei Dezimalzahlen.  
Überträge der MSBs gehen verloren (carry bit) oder erfordern eine -stellige Binärzahl.

Multiplikation einer Binärzahl mit einer Zweierpotenz ergibt eine Verschiebung nach links um Stellen (links-shift). Das grösstmögliche Produkt zweier -Bit-Zahlen ist

Links-Shift um drei Stellen:

Rechts-Shift um drei Stellen:  
   
Die hintersten 3 Stellen gehen dabei verloren.

*Zweierkomplement*

Sign-Extension

Wird eine -Bit-Zahl auf eine -Bit-Speicherstelle kopiert, werden die oberen Bits auf 0 gesetzt. Bei einer negativen Zahl ändert sich dadurch aber die Bedeutung. z.B. 4 Bit auf 8 Bit kopieren:

Statt einer 0 muss überall eine 1 kopiert werden. Dieser Vorgang heisst sign extension: das Vorzeichen wird auf alle zusätzlichen Bits ausgedehnt. Wird nicht in allen Befehlen automatisch ausgeführt (z.B. mov).

Arithmetischer Rechts-Shift ist ein Rechts-Shift mit Sign-Extension.

Shift und Rotate in Assembler

Shifts sind ähnlich performant wie die elementaren Operationen.  
Rotates füllen statt mit 0 oder 1 mit den ursprünglichen Bits auf. Es gibt folgende Befehle: ist eine Konstante oder das Register CL: shl, shr, sar, rol, ror

Shift in C

Zweierpotenz als Shift: entspricht . Beim Rechts-Shift wählt der Compiler die Instruktion abhängig vom Typ aus.   
Unsigned: shr, signed: sar. Rotates können in C nicht direkt ausgedrückt werden.

Unsigned Multiplikation in Assembler: mul z

z kann ein beliebiger Operand sein. Je nach Bit-Grösse von z sind Ergebnis und der andere Operand in unterschiedlichen Registern: 64: RDX & RAX, 32: EDX & EAX, 16: DX & AX, 8: AH & AL

Signed Multiplikation in Assembler: imul r, z

Zusätzlich zu den fixen Registern wie bei mul gibt es drei flexiblere Versionen. Diese verwenden aber nur die LSBs des Ergebnisses.

## Kontrollfluss

Operationen ohne Unterscheidung

* Bitweise Operationen: and, or, xor, not
* Aufgrund des Zweierkomplements: add, adc, sub, sbb, inc, dec
* Shift-Left (shl) (kann aber Vorzeichen bei zu grossem Shift ändern)

Operationen mit U.: mul (u) / imul (s), div (u) / idiv (s), shr **(u) /** sar **(s)**

Die binären Operatoren in C unterscheiden wo nötig selbst zwischen signed und unsigned. Der Compiler darf immer andere Instruktionen verwenden, solange diese funktional äquivalent sind. GCC verwendet z.B. imul und nicht mul für ux = ux \* uy, weil imul schneller sein kann als mul und in gewissen Operationen äquivalent ist. Der GCC ersetzt automatisch die Division durch Zweierpotenzen durch die richtige Shiftoperation.

-----------------------------------------

Flags

Flags sind einzelne Bits, die eine eigenständige Bedeutung haben. Liegen in einem gemeinsamen Register RFLAGS, das nicht direkt verwendet werden kann. Operationen verändern Flags in unterschiedlichem Masse.

* Carry Flag **(CF)**: Gibt Überlauf bei **unsigned** Arithmetik an.   
  1 = Überlauf vorhanden.
* Overflow Flag **(OF)**: Gibt Überlauf bei **signed** Arithmetik an. (Wenn erste Zahl & das Resultat unterschiedliche Vorzeichen haben)
* Zero Flag **(ZF):** Wird immer gesetzt, wenn das Resultat 0 ist
* Sign Flag **(SF):** höchstwertigstes Bit des Resultat (immer gesetzt)
* Parity Flag **(PF):** Wird gesetzt, wenn das niederwertigste Byte (unterste 8 Bits) des Resultates eine gerade Anzahl an gesetzten Bits enthält (für serielle Kommunikation)

Der Prozessor weiss nicht, ob mit signed oder unsigned gearbeitet wird. Deshalb werden immer CF und OF gesetzt. Das Programm muss später die richtige Flag verwenden.

Condition Codes (CC)

Zustände von Flags oder deren Kombinationen. Zu jedem Flag gibt es einen CC mit einem Buchstaben. usw. Zu vielen CCs gibt es den negierten CC mit vorangestelltem N.

|  |  |  |  |
| --- | --- | --- | --- |
|  | CC | Name | Flags |
| unsigned | A | Above | CF = 0 und ZF = 0 |
| AE | Above or Equal | CF = 0 |
| B | Below | CF = 1 |
| BE | Below or Equal | CF = 1 oder ZF = 1 |
|  | E | Equal | ZF = 1 |
| signed | G | Greater | ZF = 0 und SF = OF |
| GE | Greater or Equal | SF = OF |
| L | Less | SF ≠ OF |
| LE | Less or Equal | ZF = 1 und SF ≠ OF |
|  | PE | Parity Even | PF = 1 |
|  | PO | Parity Odd | PF = 0 |

Verwendung von Condition Codes

Werden in bestimmten Befehlen dazu verwendet, eine Bedingung anzugeben. Der Befehl wird nur dann ausgeführt, wenn in diesem Moment die Flags genauso gesetzt sind, wie der Condition Code angibt.   
Beispiel: Conditional Move:

dec rax ; sets ZF = 1 if rax == 0

cmovz rax, [q] ; sets rax = [q] if rax == 0

Vergleich zweier unsigned Zahlen

Die Differenz zeigt an, ob , oder ist.

|  |  |  |  |
| --- | --- | --- | --- |
| Fall | Flags | Begründung | CC |
|  | ZF = 1 |  | E |
|  | CF = 1 | ergibt (unsigned) Überlauf | B |
|  | CF = 0 | heisst | AE |
|  | ZF = 1 CF = 1 | heisst oder | BE |
|  | ZF = 0 CF = 0 | heisst | A |

Vergleich zweier signed Zahlen

|  |  |  |  |
| --- | --- | --- | --- |
| Fall | Flags | Begründung | CC |
|  | ZF = 1 |  | E |
|  | OF ≠ SF | SF = 0, OF = 1 oder umgekehrt | L |
|  | OF = SF | heisst | GE |
|  | ZF = 1 OF ≠ SF | heisst oder | LE |
|  | ZF = 0 OF = SF | heisst | G |

Beispiel: ;

-----------------------------------------

Bedingte Anweisungen

**cmp:** Für Vergleiche kann die Operation cmp eingesetzt werden. Funktioniert gleich wie sub, ausser dass die vergleichenden Operanden nicht verändert werden.

mov rax, [ux]; mov rbx, [uy]; cmp rax, rbx;

cmove rcx, 5 ; conditional move equal, only if rax == rbx

**test:** Ist ein and, welches den Ziel-Operanden nicht schreibt. Wird dafür verwendet, ein Register auf 0 zu testen.

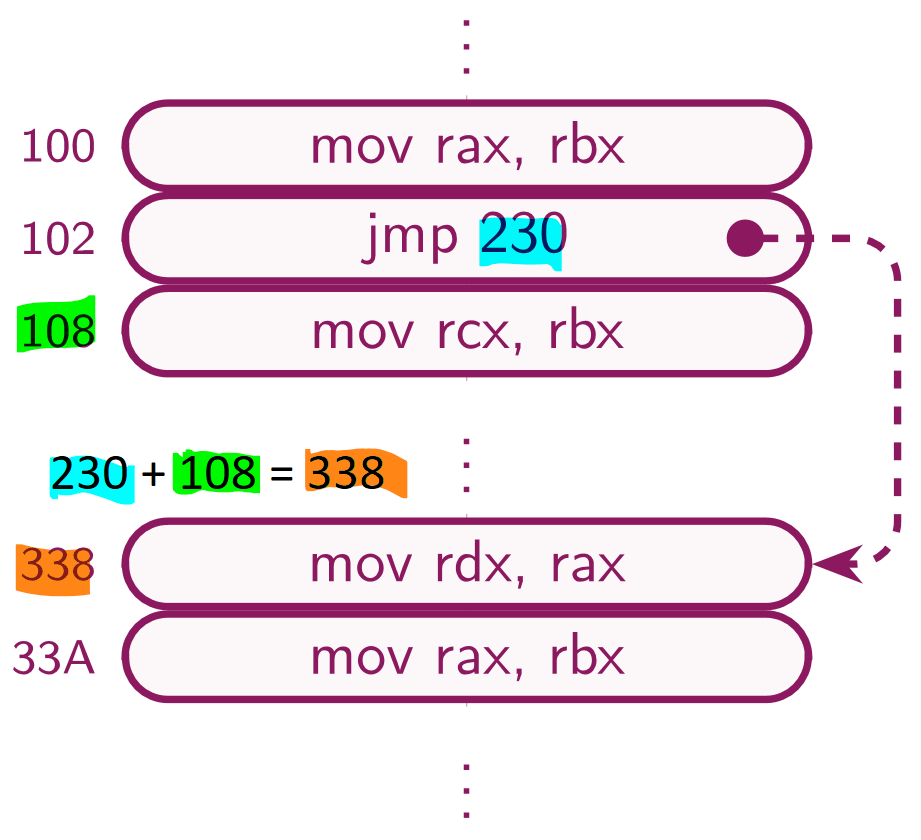
test rax, rax

cmovz rax, rbx ; conditional move zero, only if result == 0

Es gibt auf Intel 3 Familien von bedingten Operationen:

**CMOVcc:** Conditional MOV, **Jcc:** Conditional JMP (Jump), **SETcc:** Schreibt 1 in das 8-Bit

grosse Ziel, wenn CC (Bedingung) erfüllt, sonst 0

Relative Sprünge (Jump)

Der Befehl JMP d springt im Programmfluss um d, Offset ist relativ zum Index der Instruktion, die eigentlich ausgeführt werden würde. Um nicht mit dem Index arbeiten zu müssen, können Labels verwendet werden: jmp target springt bis zum Label target. Dies ist flexibler und stabiler als die Verwendung eines Offsets.

Bedingte Sprünge

Benötigen einen Condition Code. Springen nur dann, wenn der CC erfüllt ist. Ansonsten fährt der Prozessor mit der nächsten Instruktion fort.  
 mov eax, [x]

 cmp eax, 3 ; is eax == 3?

 jne if\_not\_3 ; jump if not equal

 mov eax, [y] ; wird ausgeführt, wenn eax == 3

if\_not\_3:

 inc eax ; wird ausgeführt, wenn eax != 3

Verzweigungen (if)

Wird in C mit **if** implementiert. Der Body wird ausgeführt, wenn die Bedingung nicht 0 ist. Der Compiler verwendet direkt die Condition Codes, wenn möglich.

Verzweigung mit Alternative (if … else)

Der Else-Body wird genau dann ausgeführt, wenn die Bedingung 0 ist. Die Reihenfolge der Bodies kann im Assembler vertauscht sein. Am Ende des ersten Bodies kommt ein unbedingter Sprung.

if (ux < 3) { /\* if-body \*/ } else { /\* else-body \*/ }

wird zu:  
 mov eax, [ux]; cmp eax, 2

 ja else\_body ; jump if above 2

  ; if-body

 jmp after\_if

else\_body: ; else-body

after\_if:

Schleifen (Loops)

Können mit einem Rückwärtssprung implementiert werden.

Do-While-Schleife: Nach jedem Ausführen des Bodys wird die Bedingung überprüft. Wird mindestens einmal ausgeführt.

|  |  |
| --- | --- |
| int i = 23;  do {      /\* body \*/  } while (--i); | mov rcx, 23  loop:    ; body    dec rcx; jnz loop ; jump if not zero |

While-Schleife: Vor jedem Ausführen des Bodys wird die Bedingung überprüft. Bedingung wird einmal öfters ausgeführt als der Body.

|  |  |
| --- | --- |
| int i = 23;  // == while (--i != 0)  while (--i) {      /\* body \*/  } | mov rcx, 23    jmp condition ; check condition  loop:    ; body  condition:    dec rcx; jnz loop ; jump if not zero |

Pointer-Addition

Die Addition eines Integers zu einem T\*-Pointer ***berücksichtigt*** die Grösse **sizeof(T). Beispiel:** sizeof(int) = 4 bytes  
Bei int \*p = 0x20; ist p+1 == 0x24 und p+2 == 0x28, usw.

Iterieren mit Adressen: Ein T\*-Pointer wird mit ++ um sizeof(T) erhöht. int \*p = 0x20; wird nach ++p zu p == 0x24.

***Beispiel Summieren über Speicherbereich***

extern int \*begin; // points to first element

extern int \*end;   // points \*after\* last element

int sum = 0;       // total sum of all elements

for (

    int \*p = begin; // init with address of first element

    p != end;       // execute loop while not after end

    ++p             // set to address of next element

){ sum += \*p; }     // add content of p

## Funktionen

Verwendung gleicher Sequenz mit unterschiedlichen Variablen. Adressen stehen nicht in den Instruktionen, sondern in Register oder globalen Variablen. Variablen sind etwas langsamer, jedoch kein Problem mehr mit zu wenig Register. Jedoch: Für jede Sequenz muss Speicher statisch reserviert werden, für jede Sequenz gibt es jeden Parameter genau einmal. Keine Rekursion (gibt Endlosschleife) und keine Parallelität.

-----------------------------------------

Stack

**push** **q** kopiert das Element aus q oben auf den Stack und dekrementiert Stackpointer. **pop** **z** kopiert das oberste Element vom Stack nach z und inkrementiert Stackpointer (Wert auf Stack wird aber nicht gelöscht).

Register Basepointer (**rbp**):Beginn des Stacks (an höchster Adresse)  
Register Stackpointer (**rsp**):Aktuelle Adresse im Stack. Wird eine Funktion aufgerufen, wird der rbp auf den Stack gepusht und der aktuelle rsp zum rbp. So können Adressen relativ zum rbp angegeben werden. Beim Verlassen der Funktion wird das Ganze wieder rückgängig gemacht. Achtung: Stack wächst «nach unten», Das «oberste» Element liegt an niedrigster Adresse. Stack wächst in Richtung der niedrigen Adressen.

|  |  |
| --- | --- |
| ***push rax*** entspricht: | ***pop rax*** entspricht: |
| sub rsp, 8 ;decrement Pointer  mov [rsp], rax ;insert Value | mov rax, [rsp] ;get Value  add rsp, 8 ;increment Pointer |

Auf dem Stack können Zwischenwerte, Argumente, Rückgabewerte, Rücksprungadresse gespeichert werden. push und pop müssen immer gleichmässig verwendet werden.

call s: Nachfolgende Adresse r auf den Stack und Sprung an S (Funktionsaufruf). Ret: Adresse von r von Stack entfernen und Sprung nach r (Return).

|  |  |
| --- | --- |
| ***call S*** entspricht: | und das entspricht: |
| push cont  mov rip, S ; gleich wie jmp  cont: | sub rsp, 8  mov [rsp], cont  mov rip, S  cont: |

Stack Allokation

Stack kann für Zwischenergebnisse und lokale Variablen verwendet werden. Jedes Zwischenergebnis einzeln zu pushen, wäre umständlich, deshalb (de)alloziert man den benötigten Speicher mit einer einzigen Instruktion. sub rsp, 0x20 ; allocation add rsp, 0x20 ; deallocation. Zwischenergebnisse liegen über dem Basepointer mov rcx, [rbp – 0x8] ; access memory for results

-----------------------------------------

Funktionen in C

Funktionen ohne Parameter können beliebig viele Argumente annehmen. Funktionen können auch über Pointer aufgerufen werden. Argumente werden by value übergeben, also in die Parameter kopiert. Können von der Funktion verändert werden. Mit einer Übergabe des Pointers als Parameter kann ein call by reference emuliert werden.

int f (int \*x) { \*x = \*x + 1; return \*x \* 4; }

int p = 7; int q = f (&p); //p wird 8, q wird 32

Lokale Variablen vs Globale Variablen: G haben fixe Adresse im Speicher, L werden auf Stack abgelegt. Beide können für Zwischenergebnisse verwendet werden. Adressen von L sollten nicht returnt werden, da nicht mehr gültig.

int xx = 0xCAFE;

printf ("%s @ %p: %X\n", "xx", &xx, xx);

// xx @ 0x7ffe1a2b61f4: CAFE

**%d, %i:** decimal integer or signed integer; **%u:** Short unsigned integer; **%li:** signed decimal (long); **%lli:** signed decimal (long long); **%f:** signed float; **%c:** signed char; **%s:** string or sequence of chars; **%lf:** double; **%Lf:** Long double; **%ld:** Long decimal integer; **%x:** Hexadecimal integer; **%p:** Print pointer memory address in hex

## Datentypen

Einfache Datentypen in C: char (Maschinen-Byte), int (signed Menge an Bits), void \* (Adresse eines Maschinen-Worts), int \* (Adresse eines ints)  
Typen-Alias: Mit dem Schlüsselwort typedef kann ein bestehender Typ einen weiteren Namen (alias) erhalten. Compiler behandelt beide Aliasse gleich. typedef int\* int\_pointer;  
Interpretation von Adressen: In Assembler untypisiert, C-äquivalent ist void \*. T \* a in C umfasst die Maschinen-Bytes an den Adressen a, a+1, … a + sizeof(T)-1.  
Pointer-Arithmetik: Bei Addition/Subtraktion von T\*-Adressen mit Integers wird die Grösse von T berücksichtigt.  
Index-Operator: a[b] ist definiert als \*(a+b). Der eine Operand muss eine typisierte Adresse sein, der andere ein Int.  
Arrays: Die Definition eines Arrays T a[n]; reserviert Speicher für n Elemente von Typ T und assoziiert das Label a mit der Adresse des ersten Bytes. int32\_t a[10] // 10 int32\_t: 40 Byte

int a[4] = {0x10,2,17} // a[3] initialized to 0  
int32\_t a [10];     // 10 ints

\*a = 0x42;         // 0x42 into 4 bytes starting at a

a[0] = 0x42;       // same

a = 100;           // Error: a is constant pointer

int32\_t \*p = a;     // assign address of a to p

int32\_t \*q = &a[0] // same

Grösse einer Variable / eines Typs: sizeof a bezeichnet die Anzahl Maschinenbytes des Objekts a. sizeof(T) das gleiche für einen Typ T. Wird immer zur Compilezeit ausgewertet. Bei einem Array erhält man die Anzahl Elemente durch Division durch Elementgrösse.  
uint32\_t a[] = {8,7,3,1,0};

size\_t b = sizeof a; // b == 20 (chars)

size\_t n = sizeof a / sizeof a[0]; // b == 5 (elements)

size\_t: Unsigned Integer-Datentyp, gross genug, die Grösse beliebiger Objekte zu halten. Rückgabetyp von sizeof. Für foreach gut.  
for (size\_t i = 0; i < n; ++i) { /\* body \*/}

Null-terminierte Strings: Char-Arrays, deren letztes Element «\0» ist. Vorteil: Kein Grössen-Parameter nötig.

char s[] = {'H', 'a', 'i', '\0'}; // or char \* s = "Hai";

char \*pc = s;   
while (\*pc != '\0') {/\* body \*/; ++pc;}

const: Wert darf nicht geändert werden, kann sich aber durch äussere Einflüsse ändern. (Pointer darf geändert werden, Inhalt nicht)

const bezieht sich immer auf den Typen links davon. Steht const ganz links, bezieht es sich auf den Typen rechts davon. Am besten liest man Typen von rechts nach links.  
char const \* c; // c is a pointer to const char

char \* const d; // d is a const pointer to char

++c; // OK: c is pointer

++d; // Error: d is const pointer

\*c = 'a'; //Error: \*c is const char

\*d = 'a'; // OK: \*c is char

String-Funktionen: Strings werden in den meisten Fällen als char const \* übergeben. Potenziell gefährlich, falls kein '\0' kommt. Dafür verwendet man beschränkte String-Funktionen mit einem Max-Wert: size\_t strnlen (char const \*str, size\_t max);  
strcmp() für Stringvergleich nutzen.

Structs: Globale Variablen können strukturiert werden und belegen den gleichen 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. Data Member müssen nicht dicht liegen, der Compiler darf Padding einfügen.

struct {

    char c; // offset 0, Zugriff via t.c

    int x; // Offset 4 -> Padding

    char d; // Offset 5, Adresse via int \*0 = &t.d

} t; // sizeof t == 12, Vielfaches des grössten Member

Ein Struct kann einen Tag T erhalten. Der Typ der Variable heisst dann struct T. Für Pointer, welche die Adresse eines Structs enthalten, gibt es einen eigenen Member Operator. Pointer auf Structs: struct T \*t; t->x; t->c; //Pfeil statt Punkt

Vollständige und unvollständige Typen: Ein Typ heisst complete, wenn der Compiler genug Informationen hat, um die Grösse eines Objekts dieses Typs zu bestimmen. Sonst incomplete Forward deklaration.

## Hauptspeicher und Cache

Ein Bild, das Text, Diagramm, Screenshot, Design enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Diagramm, Quittung, Reihe enthält.

Automatisch generierte BeschreibungLokalitätsdiagramme: Zeigen Speicher-Zugriffe im zeitlichen Verlauf. «Zum Zeitpunkt x wird auf die Adresse y zugegriffen».  
Arbeitsbereich eines Programms : Alle Speicherstellen, die im Zeitraum vor dem Zeitpunkt verwendet wurden.   
Lokalitätsprinzip: Programme greifen meistens auf den etwa gleich bleibenden Speicherbereich zu. Deshalb ermöglicht der Rückblick auf eine Vorhersage über . Dieses Prinzip kann für Geschwindigkeitsoptimierungen genutzt werden.

Ein Bild, das Text, Screenshot, Diagramm, Zahl enthält.

Automatisch generierte BeschreibungZeitliche Lokalität: Die gleiche Speicher-stelle bleibt längere Zeit im Arbeits-bereich. «Wenn Speicherstelle A jetzt benötigt wird, dann wahrscheinlich auch demnächst». *(Punkte horizontal nebeneinander)*

Räumliche Lokalität: Arbeitsbereich ent-hält nahe beieinander liegende Speicher-stellen. «Wenn A jetzt benötigt wird, dann wahrscheinlich auch demnächst A+1, A+2 etc». (Punkte diagonal neben/obereinander)  
Beispiel im Bild: Einfachen Schleife, die alle Werte eines Integer-Arrays aufaddiert.

-----------------------------------------

Cache (zwischen Prozessor und Hauptspeicher)

Cache ist ein Zwischenspeicher, der kleiner und schneller ist. Der Cache muss die (Hauptspeicher-)Adresse explizit mit den Daten speichern.

Cache-Grösse: Anzahl Nutz-Bytes (Payload). Platz für Adressen kommt hinzu, wird aber nicht explizit angegeben. 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 Lokalität oft > 0.9)  
**Erwartungswert der Zugriffszeit:**

Fully-Associative Cache (FAC) mit 64-Byte-Cachezeilen

Cachezeilen: Statt einzelner Einträge werden Cache-Zeilen verwendet, die benachbarte Bytes beinhalten. Cache lädt Cachezeilen immer als Ganzes. 64 Byte lange Cachezeile beginnt immer an durch 64 teilbarere Adresse . Die unteren 6 Bits der Adresse sind immer 0.   
P=Anz. Cacheeinträge (M/k) = , M=Hauptspeichergrösse, n=Adressgrösse in b,   
k = 64 (länge der Cachezeile)

Jeder der Cacheeinträge beinhaltet den Tag die oberen Bits und die Datenbytes (untere 6 Bits). Lookup: Die Adresse wird in Tag und Offset eingeteilt. Alle Cachezeilen vergleichen gleichzeitig, ob sie den Tag beinhalten. Falls ja, werden die Datenbytes zurückgegeben. FAC nutzt den Lokalitätseffekt optimal aus, aber aufwendig und teuer.

**Direct-Mapped Cache (DMC) mit reduzierten Tags**Einfach zu implementieren, schneller Lookup. Jedoch viele Kollisionen.   
 Einträge, die mit Bits durchnummeriert werden. Adresse besteht aus Reduziertem Tag , Eintrag und Offset .Lookup: Gegebene Adresse wird in und aufgeteilt. kann sich nur am Eintrag befinden. Man benötigt nur einen einzigen Vergleichsbaustein.  
Beispiel:   
Gegebene Adresse   
 und . Vergleiche, ob . Wenn ja, liegt das gesuchte Byte an

k-Way Set-Associative Cache (Multiple DMCs)  
Parallele Verwendung von k DMCs mit jeweils vielen Einträgen. Jede Cachezeile kann in verschiedenen Cacheeinträgen gespeichert werden. Set: Einträge in Cachezeile. Way: Jeder DMC im SAC. Die Set-Nummer ist die Nummer des Eintrags in jedem Way und wird genauso aus der Adresse bestimmt. Jeder Way arbeitet autonom, aber gemeinsame Logik zur Adressaufteilung. Guter Kompromiss zwischen FAC und DMC.

Mehrstufiger Cache

Normalerweise wird Cache in verschiedene Levels aufgeteilt: Level 1 (schnellster und kleinster) wird für Befehls-Cache und Daten-Cache verwendet. Dazu gibt es noch einen Level 2 und Level 3 Cache, und je nachdem einen Side Cache.

Formeln

* Adressgrösse n: *(M = 1MB = Adresse ist 20 Bit gross)*
* Cachezeilen: Cache / Bytes pro Cachezeile *(Cache = 16kb, BpC = 16b 1k Zeilen im Cache)*
* Cachezeilen pro Way: Cachezeilen / k *(Cachezeilen = 1k, k = 4 250 256 Zeilen je Way)*
* Bits pro Entry FAC: gibt es nicht, da voll assoziativ.
* Bits pro Entry DMC: (Cachezeilen) *Index Bits ( Index = 10)*
* Bits pro Entry SAC: Cachezeilenk Index Bits, k = Anzahl Ways   
  *( =8)*
* Offset: Bytes pro Cachezeile = Offset Bits *(BpC = 16b 4 bits Offset)*
* Overhead: ((Adressgrösse = 20 Bit) \* )
* Bits benötigt für Tags FAC: Adressgrösse Offset
* Bits benötigt für Tags DMC & SAC: Adressgrösse Offset Bits pro Entry

## Dynamischer Speicher

Normalerweise ist nicht im Voraus bekannt, wie viel Speicher gebraucht wird. Deshalb muss Speicher dynamisch alloziert werden.  
Heap: Speicherbereich für vollständig dynamischen Speicher. Wird vom OS verwaltet, beliebiger Zeitpunkt für Reservation, Freigabe und Lebensdauer. Wünschenswert sind beliebige Grösse der Speicherblöcke, Minimale Metadaten / Zeit-Overhead / Daten-Overhead. In der Praxis nicht möglich, alles zu optimieren.  
Speicherfreigabe: Reservierter Speicher muss wieder freigegeben werden, sonst Speicherlecks. Legt OS & andere Programme lahm.  
Implizit: Speicher wird automatisch freigegeben (Java, Python, Applikation) – Garbage Collector. Keine direkten Speicherleaks, jedoch keine Kontrolle über Freigabe und indeterministisches Zeitverhalten. Pointer sind immer gültig & alle Pointer auf Objekt sind bekannt.  
Explizit: Programmierer definiert, wann Speicher freigegeben wird (Assembler, C, Betriebssystem). C-Funktionen aus stdlib.h, immer zusammen verw.:  
void \* malloc(size\_t s), void free(void \*p). Pointer selber nullen  
Interne Fragmentierung: Heap-Implementierung reserviert grösseren Speicherblock als benötigt, bei dem der Verschnitt unverwendet bleibt. Beispiel: Heap-Implementierung kann nur Vielfache von 8 Byte allozieren, Programm fordert aber 12 Byte an. OS kann nur mind. 16 Byte allozieren, was zu 4 Byte Verschnitt führt.  
Externe Fragmentierung: Programm reserviert immer wieder Speicher und gibt ihn unregelmässig wieder frei. Im Speicher viele kleine Löcher mit Platz – jedoch kein grosser Bereich mehr.

Heap-Optimierungen

Mehrfache fester Blockgrösse: Implementierung definiert grundlegende Blockgrösse . Speicher wird nur in Vielfachen von reserviert. Grösse des Blocks ist ein Tradeoff, der die Performance bestimmt (klein: langsamer. gross: höhere interne Fragmentierung). **Dezentrale Speicherung Metadaten:** Metadaten liegen bei den Speicherblöcken. Schnelle Rekombination von freigegebenen Blöcken. **Zentrale Speicherung Metadaten:** Metadaten separat und geschützt vor Überläufen. Aufwändiger  
Speicherblock-Verwaltung mit Bitlisten: In Bitliste ein Bit je Speicherblock (0=frei, 1=verwendet). Anzahl der benötigten Bits ist die Speichergrösse / Blockgrösse (klein: mehr Bits, gross: höhere interne Fragmentierung). Um freien Speicherbereich zu finden, muss man eine Sequenz von ausreichend vielen Nullen finden. Suche ist linear und kann lange dauern.  
Speicherblock-Verwaltung mit verketteten Listen: Ein Listen-Element je Speicherbereich: Status (1=benutzt, 0=frei), Start (Adresse des ersten Blocks), Size (Anzahl Blöcke) und Next (Adresse des nächsten Listenelements). Elemente bilden eine verkettete Liste sortiert nach Adressen. Keine aufeinanderfolgende freie Bereiche, aber aufeinanderfolgende verwendete Bereiche.  
Rekombination freier Bereiche

Kein Nachbar frei: keine Rekombination. Vorheriger Nachbar frei: Erweitere vorheriges Element, entferne dieses Element. Nachfolgender Nachbar frei: Erweitere dieses Element, entferne nachfolgendes. Beide Nachbarn frei: Erweitere vorheriges Element, entferne beide.

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 keine guten Ergebnisse

Verfahren Grössenklassen  
Bereiche werden nur in bestimmten Grössen zur Verfügung gestellt. Alle freien Bereiche einer Grössenklasse werden in jeweils einer Liste vorgehalten.   
Quick Fit: Schnelle Reservation, erstes Element aus der Liste der kleinsten passenden Grösse. Nachteil: Nachbarn sind nicht leicht zu finden, Rekombination freier Bereiche aufwendig.  
Buddy System: Variante des Verfahrens Grössenklassen mit Zweierpotenzen von (kleinste Speicherbereichsgrösse) bis (gesamter Speicher). Am Anfang gibt es nur einen Bereich, der den gesamten Speicher umfasst. Alle Listen freier Bereiche sind leer.  
**Allokation:** Falls kein freier -Bereich in Liste, wird ein freier -Bereich gesucht und in zwei -Bereiche halbiert. Der eine wird verwendet, der andere kommt in die Freiliste.  
**Freigabe:** Wird ein -Bereich freigegeben, wird überprüft, ob sein Buddy in der Freiliste ist. Wenn ja, entferne Buddy aus der Liste & kombiniere Bereiche. Rekursiv mit Grösserem fortfahren.  
**Aligned Buddys**: Zwei Bereiche sind genau dann Buddys, wenn sie die gleiche Grösse haben und sich ihre Startadressen genau im Bit unterscheiden. (8kb = 213 → Unterscheidung im 13. Bit)

## Virtueller Speicher (Hardware)

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

Zugriff nur über OS

Prozess entkoppelt vom Speicher, kann nichts zerstören. OS hat volle Kontrolle. Verschiedene Prozessor-Modi für Prozesse und OS  
Privilege Levels (PL): PL0 bis PL3. Programme laufen auf PL3, OS auf PL1.  
Virtuelle Adressen: OS gibt dem Prozess nicht die reale Adresse, sondern eine virtuelle. Aufgeteilt in Offset und Page Number. Nur Page Number wird auf Frame Number übersetzt, Offset nicht. MMU verwaltet.  
Gültiger Zugriff: Prozess will auf Adresse zugreifen, MMU findet Mapping in Mappingtabelle, MMU legt reale Adresse auf Speicherbus, Prozess liest/schreibt Daten von/auf Speicherbus.  
Ungültiger Zugriff: Prozess will auf ungültige Adresse zugreifen, MMU stellt fehlendes Mapping fest, signalisiert Fault-Interrupt. CPU ruft OS-Interrupt-Handler auf. OS übernimmt.  
Flexible Adressräume: Prozesse beliebig im realen Speicher verschieben und auslagern. Ergibt mehr Speicher pro Prozess. Prozess merkt nichts.  
Wenn verschoben: OS-Interrupt-Handler triggert OS Memory Manager, dieser passt Speicher und Mapping Table an. Danach kann der Prozess fortgesetzt werden.

Seitenbasierter virtueller Speicher (Pages)

Hauptspeicher besteht aus Page Frames, in ein Page Frame passt eine Page (4KB, 12 Bits Offset). Page Frame Number = Startadresse des Page Frames ohne Offset-Bits. Beispiel im 32-Bit System: Adresse: , Page Frame Number: . Der virtuelle Adressraum besteht aus Pages. Eine Page repräsentiert Daten, ist kein Speicher, sondern benötigt einen Page Frame. Pro Prozess gibt es ein virtueller Adressraum und eine Page-Table (Mapping).   
Memory Management Unit (MMU): kann Speicherzugriffe überwachen, übernimmt Erzeugen der realen Adressen. Wenn MMU Page nicht findet, ist sie entweder im Sekundärspeicher oder existiert nicht.   
Beispiel: Prozess greift auf virtuelle Adresse zu. MMU ermittelt Page Number . In Page Table ist die dazugehörende Page Frame Number . MMU bestimmt reale Adresse

**Page-Table**

Dient der MMU. Jede Adresse wird via Page-Table übersetzt. Wird vom OS konfiguriert.  
Single-Level: Array mit einem Eintrag pro Page (Page Frame Number, Page-Status-Bits). Lookup sehr schnell, (Index = Page Number).   
Die Grösse hängt vom virtuellen Adressraum ab. Für jede mögliche Page gibt es einen Eintrag. Kann sehr schnell sehr gross werden.  
Prozess greift auf virtuelle Adresse zu. MMU ermittelt Page Number . MMU schaut im Array an Index nach (. Page Frame Number dort ist . Reale Adresse ist .

Formeln:

* Page Offset: Page (Resultat in Bits)
* Pages: virtueller Adressraum / Pagegrösse
* Page Frames: Hauptspeicher / Pagegrösse
* Page Frame NR: Page Frames
* Virtuelle Adresse: virtueller Adressraum = Bits (Kleinste Adr. = 0, grösste Adr. = )
* Reale Adresse: Hauptspeicher = Bits (Kleinste Adr. = 0, Grösste Adr. = )
* Page Table Grösse: Pages Page Frame Nr (Resultat in Bits)

Two-Level Page-Table: Page Number = Directory Idx + Page Table Idx. Alle Pages mit gleichem Directorx Idx sind in derselben Page Table. Page Directory zeigt auf Page Tables. Lookup funktioniert zweistufig: Erst Page Table in Directory bestimmen, dann weiter wie bei Single-Level Page-Table, aber mit Page Table Idx statt Page Number. Status-Bit «Used»: Gibt an, ob virtuelle Seite überhaupt benutzt. Vorteil: Nur benötigte Page-Tables existieren.  
Prozess greift auf virtuelle Adresse zu. MMU schaut in Top-Level-Page-Table an Index nach. Dort befindet sich Pointer auf reale Adresse der Second-Level-Page-Table (SLPT) . MMU ermittelt relative Page Number = , schaut in SLPT bei diesem Index nach. Page Frame Number dort ist , reale Adresse ist .

Formeln:

* Page Offset: Pagegrösse (Resultat in Bits)
* Page Nummer: Virtuelle Adresse Bits Page Bit Offset (Resultat in Bits)
* Anzahl Pages: Virtueller Adressraum / Pagegrösse
* Pages:
* Page Frames: Haupspeicher / Pagegrösse
* Page Table Entry  (Grösse des Eintrags): Virtuelle Adresse / 8 Bits bzw 1 Byte
* Einträge pro Page: Pagegrösse / Page Table Entry
* Eintrag adressieren bzw. Bits für Directory Index / Table Index: Einträge pro Page

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.
* Minimale Anzahl Page-Tables: ein Page-Directory und ein Page-Table
* Maximale Anzahl Page-Directories: N Ebenen, Ebene hat E Einträge:

Multi-Level Page-Table: Besteht aus Page Directory Pointer Table, Page Directory, Page Table. Translation Lookaside Buffer: Cache für häufig benötigte Mappings. Inverted Page-Table: Nur eine Tabelle für alle Prozesse. Suche dauert dafür sehr lange, nicht praktikabel. Hashed Page-Table: Hashable mit Page Number als Key. Suche kann lange dauern, bei Kollision muss Linked List iteriert werden. Performance aber praktikabel genug, da TLB benutzt.

## Virtueller Speicher (Software)

Page Table (x86): Jeder Page-Table-Eintrag hat 32 Bit. Unterstes Bit = P-Bit («Present»). Entspricht dem Used-Bit. P = 1: Page ist im Hauptspeicher. MMU setzt A-Bit («Accessed») bei jedem Zugriff und D-Bit («Dirty») bei jedem Schreibzugriff. Nur das OS kann beide Bits löschen.  
P=0: MMU wirft Interrupt und schaut andere Bits nicht an. OS überprüft, ob Page Table Eintrag gültig ist, wenn nein: Page Fault, wenn ja: Entferne eine alte Page und lade fehlende Page in Hauptspeicher.  
P = 1: MMU übersetzt virtuelle in reale Adresse.  
Dreschen: Zu häufiges Pagen kann System überladen. Kann durch mehr Hauptspeicher, Laststeuerung, Paging-Strategien vermieden werden.

Ladestrategien (fetching policies)

Welche Pages aufs Mal Laden und wann?  
Demand Paging: Pages werden nur dann geladen, wenn sie benötigt werden. Minimaler Aufwand, lange Wartezeiten.   
Prepaging: Seiten werden frühzeitig geladen, vor Verwendung. System muss dafür «in die Zukunft sehen», schwierig umzusetzen.  
Demand Paging mit Prepaging: Demand Paging, benachbarte Pages werden jedoch mitgeladen. Weniger Page-Faults, Blocktransfer.

Entladestrategien (cleaning policies)

Wann modifizierte Pages zurückschreiben?  
Demand Cleaning: Entladen auf Nachfrage (Wenn Platz benötigt wird). Minimaler Aufwand, lange Wartezeiten.  
Precleaning: Modifizierte Pages werden frühzeitig in den sekundären Speicher geschrieben. Reduzierte Wartezeit, mehr Aufwand.  
Precleaning mit Page Buffering: Zwei Listen, Liste mit Page Numbers der unveränderten Pages (), Liste mit Page Numbers veränderter Pages (). MMU setzt D-Bit beim Schreiben auf die Page. OS läuft periodisch im Hintergrund: «Verschiebt» Pages mit Dirty Bit von nach , Schreibt Pages von in sekundärem Speicher & «verschiebt» sie von nach .

Verdrängungsstrategien (page replacement policies)

Welche Page entfernen, wenn kein Platz mehr im Hauptspeicher?  
A&D=0 Seit längerer Zeit keinen Zugriff A=1&D=0 Vor kurzer Zeit gelesen A&D=1 Vor kurzer Zeit geschrieben A=0&D=1 Vor längerer Zeit geschrieben

Pages mit D = 0 werden bevorzugt ersetzt.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte BeschreibungOptimal: Ersetze Seite, die am spätesten in der Zukunft gebraucht wird. Quasi nicht umsetzbar. Gut als theoretische Vergleichsbasis.

Page-Fault: Jedesmal, wenn neue Page in Hauptspeicher geladen wird.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte BeschreibungFIFO: Ersetze jeweils die älteste Seite. Alte, häufig benutzte Seiten werden entfernt und gleich wieder geladen. Béládis Anomalie: Grösserer Hauptspeicher kann zu mehr Page Faults führen.

Second Chance: Prüft A-Bit der ältesten Page. 0 = Page wird ersetzt. 1 = A-Bit wird gelöscht und Page ans Ende der Liste geschoben, nächste Page wird überprüft. Dies entfernt die älteste nicht verwendete Page.

Ein Bild, das Entwurf, Diagramm, Zeichnung enthält.

Automatisch generierte BeschreibungClock: Effiziente Implementierung von Second-Chance. Linked-List wird zum Kreis («Clock»). Elemente werden nicht verschoben, sondern Next-Pointer umgesetzt. (Pointer geht reihum und prüft A=0, dann entfernen)

Least Recently Used (LRU): Ersetzt am längsten unbenutzte Page. MMU notiert bei jedem Zugriff den Zeitpunkt . Page mit kleinstem wird ersetzt. Sehr nah am Optimum, erfordert aber grossen Aufwand in HW.

Verdrängungsstrategien mit Interrupt:

Wenn die HW die Zeit nicht messen kann, muss das das OS tun.

Ein Bild, das Text, Schrift, Screenshot, Zahl enthält.

Automatisch generierte BeschreibungNot Frequently Used (NFU): Benötigt Counter Table. Pro Page-Table-Eintrag ein Counter am selben Index. Zugriff erhöht Counter. Page mit niedrigstem Counter wird ersetzt. Problem: Alte Pages können lange im Speicher bleiben, weil sie bereits einen höheren Counter haben.  
NFU with Aging: Counter wird nach Zeit gewichtet. Pro Page gibt es einen n-Bit Counter. Im Timer Interrupt: Füge A oben an und verliere das älteste Bit, also: wird zu bzw .

Ein Bild, das Text, Schrift, Zahl, Screenshot enthält.

Automatisch generierte BeschreibungWorking Set: Pro Page-Table-Eintrag ein Zeitstempel , Intervall = . Statt Timer-Interrupt: Scanne alle Pages im Fault-Interrupt: Wenn : Setzte now und . Wenn : Wenn now : Page bleibt. Wenn now : Page wird entfernt. Wenn keine Page gefunden: Entferne älteste Page (bevorzugt saubere Page).

WSClock: LinkedList («Clock») wie bei «Clock»-Verdrängungsstrategie. Statt Timer-Interrupt: Iteriere über Clock im Fault-Interrupt. Ansonsten gleich wie Working Set.

## Ein- und Ausgabe

Memory-mapped I/O: Geräte am Speicherbus. Pro Gerät muss eigener Adressbereich reserviert werden. Einfach, jedoch beschränkter Speicher.  
Port-mapped I/O: Geräte haben separaten Bus für Adressen und Data. Speicher kann gesamten Adressraum belegen, jedoch komplex.  
Port-mapped I/O via Speicherbus: Bus hat zusätzliche Bitleitung, die angibt, ob Bus aktuell für Speicher oder I/O genutzt wird. Nur ein Bus nötig, nur ein zusätzliches Bit im Adressraum nötig. Beste Option.

Polling: CPU fragt regelmässig das Gerät. Wenn Status OK, kann Programm auf Gerät schreiben oder lesen.  
Polling mit Busy Wait: Software pollt hintereinander weg. Keine Verzögerung, jedoch legt das die CPU quasi lahm. Unprofessionell.  
Polling ohne Busy Wait: Software pollt in regelmässigen Abständen. Wenn Gerät nicht bereit, arbeitet an anderen Aufgaben. Erfordert genaue zeitliche Analyse.  
Interruptgesteuert: Gerät unterbricht Software, sobald bereit. Interrupt Controller ist mit Interrupt Pin mit der CPU verbunden und gibt der CPU die Interrupt-Nummer an. CPU hat Interrupt-Vektor-Tabelle, um entsprechende Funktion passend zur Nummer aufrufen zu können. CPU prüft nach (fast) jeder Instruktion, ob Interrupt Pin gesetzt. Interrupts können verschachtelt auftreten. NMI = Nicht abschaltbarer Interrupt.

Direct Memory Access (DMA): Gerät greift direkt auf Speicher zu. Zusätzlicher Baustein: DMA-Controller. CPU programmiert DMA für Transfer: Quelle, Ziel, Menge, Betriebsart. CPU gibt Speicherbus an DMA frei, DMA lässt Gerät direkt in Speicher kopieren. Nach Beendigung setzt DMA Interrupt.  
DMA für Memory-mapped I/O (IOMMU): Analog zu MMU. Erlaubt Speichervirtualisierung für den für Geräte reservierten Bereich. Nur CPU kann IOMMU konfigurieren. Erlaube es einer VM, OS DMA einzusetzen. Host OS konfiguriert IOMMU, Gerät DMA wird in Speicherbereich Gast OS umgeleitet. Gast OS weiss nichts von Virtualisierung.  
DMA Einzeltranfer: CPU stösst DMA nur für einen einzelnen Transfer an, gibt Bus gleich wieder frei (Für periodische Geräte wie Tastatur).  
DMA Blocktransfer: DMA belegt Bus bis Transfer vollständig. Für Geräte mit eigenem Datenpuffer (viele Daten aufs Mal liefern) wie Festplatte.

Ohne Treiberarchitektur: Benutzerprogramm verwaltet Hardware, komplex, nicht sicher, nicht stabil, nicht gut.  
Mit Treiberarchitektur: OS verwaltet Hardware. Benutzerprogramme können nur über OS-API (mittels Treiber) auf HW zugreifen.  
Treiberarchitektur: Software-Schichten-Modell: Programme Laufzeitumgebung (Dateioperationen) Geräte-unabhängige OS-Komponenten (Dateisystem-Manager) Treiber (zb für Datenträger) Interrupt-Handler. Treiber können auf anderen Treibern aufsetzten.  
Microkernel-Treiberarchitektur: Vollständige Separation aller Treiber. Stabil, aber eventuell schlechtere Performance.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4096 | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |