

Binärdaten I

Binär / Hex

Bit = 1 Gesetztes Bit (set bit) — Bit = 0  
Gelöschtes Bit (cleared bit)  
LSB Least Significant Bit, niederwertigstes Bit  
MSB Most Sifziant Bit, höchstwertiges Bit  
Nibble Binärzahl mit 4 Bit  
Oktett Binärzahl mit 8 Bit (== Byte)

Binär	Hex	Dez	Dütsch
0000	0	0	2 <sup>0</sup> = 1
0001	1	1	2 <sup>1</sup> = 2
0010	2	2	2 <sup>2</sup> = 4
0011	3	3	2 <sup>3</sup> = 8
0100	4	4	2 <sup>4</sup> = 16
0101	5	5	2 <sup>5</sup> = 32
0110	6	6	2 <sup>6</sup> = 64
0111	7	7	2 <sup>7</sup> = 128
1000	8	8	2 <sup>8</sup> = 256
1001	9	9	2 <sup>9</sup> = 512
1010	A	10	2 <sup>10</sup> = K ≈ 1 Tausend
1011	B	11	2 <sup>20</sup> = M ≈ 1 Million
1100	C	12	2 <sup>30</sup> = G ≈ 1 Milliarde
1101	D	13	2 <sup>40</sup> = T ≈ 1 Billion
1110	E	14	2 <sup>50</sup> = P ≈ 1 Billiarde
1111	F	15	

Prozessor

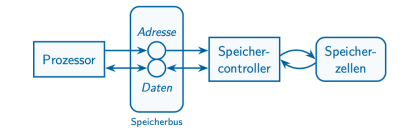
Der Prozessor ist mit dem Speicherbus verbunden und der Speicherbus besteht aus Adressbus, Datenbus und Steuersignale. Der Adressbus enthält Adresse der Speicherzelle, auf die zugegriffen werden soll. Der Datenbus enthält Daten, die aus der Speicherzelle gelesen wurde, bzw. geschriebe werden soll. Jeder Prozessor hat eine kleine Menge an Speicher Register. Mit seinen Bausteinen kann er Operationen durchführen. Sequenzen können Hart-codiert im Prozessor oder auf dem Hauptspeicher sein.

Word	2 Byte	16 Bit
Doubleword	4 Byte	32 Bit
Quadword	8 Byte	64 Bit
Double Qword	16 Byte	128 Bit

Hauptspeicher

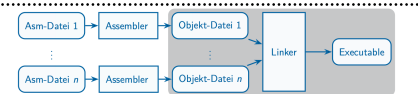
Wert an einer Adresse im Hauptspeicher speichern:

CPU legt Adresse und Wert auf den Speicherbus  
Speichercontroller liest Wert aus und legt sie auf den Speicherbus  
Speichercontroller schreibt Wert in Zelle an der Adresse



Assembler und Linker

Big-Endian: CAFEB<sub>h</sub> Little-Endian: FECA<sub>h</sub>



Programme werden üblicherweise aus mehreren Assemblerdateien generiert. Der Assembler erzeugt aus einer Assemblerdatei eine Objekt-Datei. Der Linker erstellt aus einer oder mehreren Objekt-Dateien ein Executable.

Intel 64 Architektur

Nr	64Bit	32Bit	16Bit	8Bit	Beschreibung
0	RAX	EAX	AX	AL	Rechenoperat.
1	RCX	ECX	CX	CL	Counter für Schleifen
2	RDX	EDX	DX	DL	Pointer
3	RBX	EBX	BX	BL	Datenpointer
4	RSP	ESP	SP	SPL	Stackpointer
5	RBP	EBP	BP	BPL	Adresse innerhalb des Stacks
6	RSI	ESI	SI	SIL	Zielindizes für String-Operationen
7	RDI	EDI	DI	DIL	Zielindizes für String-Operationen

Datentransfer-Operationen sind Instruktionen. In ein Register kann man kopieren:

- von einem anderen Register  
mov rax, rbx → Kopiere Inhalt von rax nach rbx
- einer Konstante  
mov rax, 0x8000 → Kopiere rax nach 8000<sub>h</sub>
- vom Speicher  
mov rax, [0x8000] → Kopiere Inhalt von rax gleich Inhalt von 8000<sub>h</sub> ... 8007<sub>h</sub>

Bei Displacement folgt die Adresse unmittelbar (mov rax, [0x800]). Bei Base steht die Adresse in einem anderen Register. (mov rax, [rbx]) Scaled Index (i \* s): Index i ist ein Register, Scale s ist eine Konstante (1,2,4 oder 8), die Ausgangsadresse steht in einem Register: → mov rax, [rcx \* 8]

C-Toolchain

Die Bestandteile der C-Toolchain sind: C-Präprozessor, C-Compiler, Assembler, Linker Es lassen sich 3 Sprachebenen unterscheiden:

- Präprozessor definiert Direktiven, die im Programm vor dem eigentlichen Übersetzen als Textersetzung durchgeführt werden.
- Basiskonstrukte bestimmen das Grundgerüst eines Programms, z.B. Variablen, Schleifen, Verzweigungen.
- Standardbibliotheken stellen Funktionen und Typen bereit, die die Basis-Funktionalität enthalten.

Der Präprozessor verarbeitet die Datei in mehreren Durchläufen. In jedem Durchlauf wird die ganze Datei bearbeitet.

- Durchlauf: Entfernt alle Kommentare oder Zeilenumbrüche.
- Durchlauf: Teilt den gesamten Text in Tokens ein. Es gibt 5 Klassen von Tokens:

- Bezeichner (size\_t)
- Präprozessor-Zahlen (0x1234)
- String- und Character-Literale (text nach " oder ')
- Operatoren und Satzzeichen (a+++++b → a++ ++ ++b)
- Sonstige

- Durchlauf: Durchläuft Token-Liste, führt Präprozessor-Direktiven aus und ersetzt Makros durch Expansion. Ist das erste Token auf einer Zeile #, wird das nächste Token als Direktive (include, define, if, else, endif) interpretiert.

Es gibt objektartige und funktionsartige Makros. Objektartige Makros haben keine Parameterliste. Der Präprozessor ersetzt im Programm nach der Definition jedes Token, das das dem Makronamen entspricht, durch die Tokenliste. Tritt der eigene Makroname in der Ersetzung auf, wird er nicht ersetzt, um infinite Rekursionen zu verhindern.

Variablen und Zuweisungen

Deklaration in C:

extern int y; //Es gibt ein int namens y  
Definition in C;

int y = 5; //Reserviert Speicher von int  
//namens y und initialisiert den Wert 5

Es kann mehrmals Deklariert werden (sofern immer gleich), jedoch nur einmal definiert!

Globale Variablen haben immer einen Typ und Bezeichner. Optional kann jede globale Variabel Initialisiert werden. Sie werden standardmässig exportiert:

int c = 5; //in C  
global c //in Assembler  
c: dd 5  
Soll eine Globale Variabel aus anderer Objekt-Datei importiert werden:  
extern int c; //import  
static int c = 5; //wird nicht exportiert!

Bezeichner	Deklarat.	Bezeichner	Deklarat.
Obj-Dat.	Assem.	in C	in C
lokal	-	global	static
global	global	global	-
-	extern	extern	extern

Linker setzt die Dateien auch bei verschiedenen Typen zusammen, da Typen nicht in .o-Dateien gespeichert werden!

Lösung: Deklaration oder Header erstellen → Compiler gibt Fehlermeldung.

Pointer: Die Adresse eines Objekts vom Typ T ist vom Typ T\*

Der Referenzoperator: & erzeugt die Adresse eines Ausdrucks. Der Dereferenzator: \* dereferenziert eine Adresse.

Arithmetische und Logische Operationen

logische Operatoren:

AND (∧): Konjunktion  
OR (∨): Disjunktion  
NOT: Gegenteil  
XOR (⊕): ==1, wenn a oder b ==1  
NAND(⋈):==0, wenn a und b ==1

Zweierkomplement:

Binärzahl b + N(b) = 0!  
Vorgehen: b invertieren und + 1 Bit  
Spezialfall: (0) = 0 und  
N(2<sup>n-1</sup>) = 2<sup>n-1</sup> (für 2<sup>n-1</sup> < 0)  
Operationen in C vs. Assembler

Assembler	C	Kommentar
add z, q	z = z + q	Addition
sub z, q	z = z - q	Subtration
adc z, q	z=z+q+c(CarryFlag)	
sbb z, q	z=z-q-c	
neg z	z=-z	0-z
inc z	z = ++ z	z+1
dec z	z = -- z	z-1

Shiften

Linksshift (um 3 Stellen):  
101<sub>b</sub> \* 2<sup>3</sup> = 10'1000<sub>b</sub> (5→40)  
Rechtsshift (um 3 Stellen):  
10'1011<sub>b</sub>/2<sup>3</sup> = 101<sub>b</sub> (43→5)

Multiplikation/Division

Es dürfen nur signed \* signed und unsigned \* unsigned gerechnet werden!

Assembler	C	Kommentar
imul z, q	z = z * q	signed Multipl.
mul z, q	z = z * q	unsigned Multipl.
idiv z, q	z = z / q	signed Division
div z, q	z = z / q	unsigned Division

Kontrollfluss

Signd Zahlen → Hälfte pos. und neg. Zahlen  
Unsigned Zahlen sind nur positiv.

Folgende Operationen sind bei beiden gleich:

- Bitoperationen (and, or, xor, not)
- Aufgrund des Zweierkomplements (add, adc, sub, sbb, inc, etc.)

Folgende Operationen sind unterschiedlich:

- Multiplikationen (mul (u) / imul (s))
- Division (div (u) / idiv (s))
- Rechtsschift (shr (u) / sar (s))

Division durch Zweierpotenzen C → Assembler

unsigned ux; ux = ux / 8; // wird zu: mov eax, [ux] shr eax, 3 mov [ux], eax	int sx; sx = sx / 4; // wird zu: mov eax, [sx] sar eax, 2 mov [sx], eax
---	--

Flags

Liegen im Register RFLAGS, das nicht direkt verwendet werden kann.  
CF - Carry Flag wird gesetzt, wenn ein Überlauf bei unsigned Arithmetik stattfindet.

OF - Overflow Flag wird gesetzt wenn die Vorzeichen nicht mehr stimmen.

ZF - Zero Flag wird gesetzt, wenn das Resultat 0  
SF - Sign Flag entspricht 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 bezeichnet.

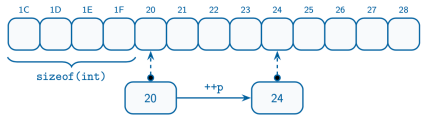
CC	Flags	Info
AC	CF = 0 and ZF = 0	unsig.: a>b
AE	CF = 0	sign.: a≥b
B	CF = 1	unsig.: a<b
BE	CF = 1 oder ZF = 1	sign.: a≤b
E	ZF = 1	a = b
G	ZF = 0 and SF = OF	sign.: a>b
GE	SF = OF	a≥b
L	SF ≠ OF	sign.: a<b
LE	ZF = 1 and SF ≠ OF	unsig.: a≤b

Sprünge auf Assembler-Ebene

Der Befehl JMP d setzt RIP = RIP + d nach dem Befehl. (d ist eine 8- oder 32-Bit-Zahl, die auf 64-Bit-signed extended wird). Anstatt der Zahl d kann auch ein Label verwendet werden.

Iterieren mit Adressen

In C iteriert man gern direkt mit Adressen. Ein T\*-Pointer wird mit ++ um sizeof(T) erhöht. int \*p = 0x20; wird nach ++p zu p == 0x24



Funktionen

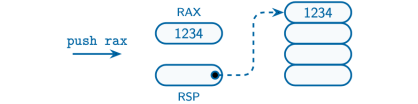
Rücksprungmechanismus auf Assembler

Wird mit jmp rsi umgesetzt. rsi muss jedoch vor dem Springen zum (Label) and512 definiert werden.

Stackframes und Call-Stack

Das Konzept «Stack» umfasst einen Stackpointer und Operationen push und pop. rsp zeigt immer auf das zuletzt abgelegte Element.

push rax kopiert das Element aus rax auf den Stack und passt den Stackpointer an.



pop rax kopiert das oberste Element vom Stack in rax und passt den Stackpointer an.



CALL

call x legt die Rücksprungadresse auf den Stack und springt dann zu x

RET

ret nimmt die Rücksprungadresse vom Stack und springt dahin

Calling Convention

- Die Calling Convention ist die Vereinbarung zwischen Caller und Callee darüber,
- wo Argumente / Rückgabewerte übergeben werden: z.B. Argument 1 liegt in RAX"
  - welche Register die Funktion verändern darf
  - wer den Stackframe für lokale Variablen und Argumente ab- und aufbaut

Funktionskonzept in C

Eine Funktionsdeklaration besteht aus 3 Teilen:

- Typ des Rückgabewert (kein Array!)
- Bezeichner
- Parameterliste in runden Klammern

Lokale und Globale Variablen in C

Jedes Objekt hat in C eine Lebensdauer, während derer Speicherplatz garantiert ist.

- Globale Variablen leben so lange, wie das Programm läuft.
- Argumente leben bis zum Ende der Ausführung der Funktion
- Rückgabewert lebt vom Ende der Funktion bis zur Auswertung des Aufrufers
- Variablen leben vom Zeitpunkt der Definition bis zum Ende des Blocks {}, in dem sie definiert wurde.

- Globale Variablen leben so lange, wie das Programm läuft.
- Argumente leben bis zum Ende der Ausführung der Funktion
- Rückgabewert lebt vom Ende der Funktion bis zur Auswertung des Aufrufers
- Variablen leben vom Zeitpunkt der Definition bis zum Ende des Blocks {}, in dem sie definiert wurde.

Lokale Variablen

Variablen, die innerhalb einer Funktion definiert werden heissen lokal. Sie werden bei jedem Aufruf auf dem Stack oder Register neu angelegt. Werden sie nicht definiert, so ist ihr Wert (im Gegensatz zu globalen Variablen) nicht definiert. Adressen lokaler Variablen liegen immer auf dem Stack. Sie können wie jede andere Adresse der Rückgabewert einer Funktion sein, Sie können auch in einen Pointer geschrieben werden.

Global vs. Lokale Variablen

- Globale Variablen haben eine fixe Adresse im Speicher
- Lokale Variablen werden auf dem Stack angelegt, die Adresse ist nicht fix
- Globale Variablen verhindern rekursive Funktionen.
- Grundsätzlich können globale sowie lokale Variablen für Zwischenergebnisse verwendet werden

Datentypen

In **Assembler** existieren keine Datentypen.  
In **C** gibt es folgende:

- **char**: Maschinen-Byte
- **int**: "natürliche" Menge an Bits, als signed
- **void \***: Maschinen-Wort, interpretiert als Addr
- **int \***: Maschinen-Wort, interpretiert als Adresse eines int

Kann nur von gleichen Typen gebildet werden  
`int32_t y = 100; int32_t *x = 100;`  
`ptrdiff_t z = x - y; // z ==5`

Array

Die Definition eines Arrays `T a[n];` reserviert Speicher für `n` Elemente vom Typ `T` und assoziiert das Label `a` mit der Adresse des ersten Bytes. Arrays werden über eine Liste von Werten in geschweiften Klammern initialisiert.

```
int a[4] = {0x30, 0x10}; //b[2] & b[3]
        initialized to 0
```

Wird nichts in die eckigen Klammern geschrieben, schaut das Array selber, wie gross es sein soll.

sizeof(T)

`sizeof a` bezeichnet die Anzahl Maschinenbytes des Objekts `a`. `sizeof (T)` das gleiche für einen Typ `T`.

Null-terminierte Strings

Char-Array, deren letztes Element `'\0'` ist.

String-Literale

String Literale stehen zwischen `"`", entsprechen einer Sequenz von `char` und enden mit einer impliziten `'\0'`

```
char * s = "Hai" // gleich wie:
char c[] = {'H','a','i','\0'};char * s = c;
const
```

`const` bedeutet, dass der Wert nicht geändert werden darf. Der Wert kann sich aber durch äussere Einflüsse ändern. Am besten von rechts nach links lesen!

```
char const * c; // Pointer to const char
char * const d; // 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: *d is char
```

String-Funktionen

Strings werden in den meisten Fällen als `char const *` übergeben.

Strukturierte Variablen

Globale Variablen können strukturiert werden:

```
struct{ int x; int y; } t;
t ist eine strukturierte globale Variable mit den beiden Members x und y, jeweils vom Typ t. Der Zugriff erfolgt über die Variable t.
t.x =5;
```

Das erste Member eines Structs hat immer die Adresse wie der Struct selbst. Member müssen nicht dicht liegen, der Compiler darf Padding (Abstände) einfügen.


Null-Pointer

Der Int-Wert 0 wird in Verwendung mit Pointer als NullPointer interpretiert.

```
if (px == 0) ... //testet ob Null-Pointer
```

Typ-Casting

```
int *p = (int *)3; //OK: int 3 cast into pointer
```

Created with 

Cache

**Lokalitätsprinzip**: werden bestimmte Daten in kleinen Abständen häufig verwendet, zählt es sich aus, diese Daten schnell zugreifen zu können. Er funktioniert aber autonom und kann nicht vom Programmierer angesteuert werden. In realen Programmen ist  $W(t, \Delta t)$  auch für grössere  $\Delta t$  konstant.

Adressen und Daten im Hauptspeicher

Hauptspeicher enthält die Adresse implizit als Ort der Speicherzelle.

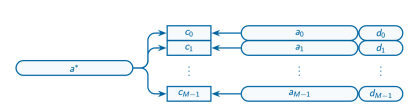
Adressen und Daten im Cache

Cache muss die (Hauptspeicher-) Adresse explizit mit den Daten speichern

- **Cache-Grösse** Anzahl Nutz-Bytes (ohne Addr.)
- **Cache-Hit** Gesuchte Adresse ist im Cache
- **Cache-Miss** Gesuchte Adresse nicht im Cache
- $T_c$  Zugriffszeit auf Cache
- $T_M$  Zugriffszeit auf Hauptspeicher
- $p_c$  Wahrscheinlichkeit eines Hit (oft >0.9)

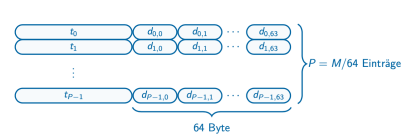
Fully-Associative Cache (FAC)

Der M-Byte grosse Cache wird in M Einträge aufgeteilt. Eintrag `i` besteht aus der Adresse `a-i` und Datenbyte `di`. Im Lookup soll überprüft werde, ob der Cache die Adresse `a*` enthält. Dies wird mit dem Hardwarebaustein `c` überprüft.



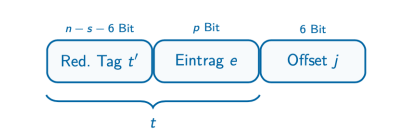
FAC mit 64-Byte-Cachezeilen

Der Cache lädt die Cachezeilen immer als Ganzes. Jeder Cacheeintrag enthält einen **Tag** `tk` und **Datenbytes** `dk,0...dk,63`. Die unteren 6 Bits der Adressen in `k` sind die Zahlen von 0 bis 63. Die oberen `n - 6` Bits ( $\leftarrow$  Tag) sind gleich (Adressen sind n Bit gross).



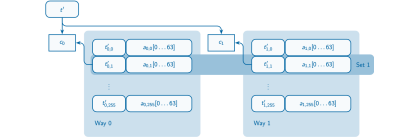
Direct-Mapped Cache (DMC)

$2^p$  Einträge, die mit `p` Bits durchnummeriert werden. Eine Cachezeile mit Tag `t` kann nur einen einzigen Eintrag speichern. Der Teil von `t` ohne `e` heisst reduziertes Tag `t'` =  $t/2^p$



k-Way Set-Associative Cache (SAC)

Ein SAC ist die parallele Verwendung von DMCs mit jeweiligen `P/k` vielen Einträgen.



Dynamischer Speicher

Der **Heap**:

- ist ein Speicherbereich für vollständig dynamischen Speicher
  - wird vom OS verwaltet
  - beliebiger Zeitpunkt für Reservation / Freigabe
- Explizite Speicherfreigabe**: Programmierer definiert, wann Speicher freigegeben wird  
**Implizite Speicherfreigabe**: Speicher wird automatisch freigegeben, wenn nicht mehr benötigt

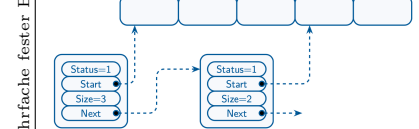
`void * malloc (size_t s) in <stdlib.h>` alloziert ein Speicherblock der Grösse `s`, gibt kleinste verwendbare Adresse zurück und `size_t` ist ein Unsigned-integer für Speichergrösse  
`void free (void *p)` gibt den Speicherblock frei, der an der Adresse `p` beginnt.

**Interne Fragmentierung**: Heap-Implementierung reserviert grösseren Speicherblock als benötigt. Zusätzliches Speicherbereich wird vom reservierenden Programm nicht verwendet, kann aber auch für kein anderes Programm verwendet werden. **Externe Fragmentierung**: Programm reserviert immer wieder Speicher und gibt ihn unregelmässig wieder frei → Über längere Zeit entsteht ein Speicherbild, in dem es nur noch kleine Löcher gibt.

Varianten von Heap-Implementierungen:

Implementierung definiert **grundlegende Blockgrösse**. Blöcke liegen im Speicher dicht, **ohne Zwischenräume**. Grösse des Blocks bestimmt massgeblich die Performance  
Kleinere Blöcke: mehr Metadaten, langsamere Grössere Blöcke: höhere interne Fragmentierung

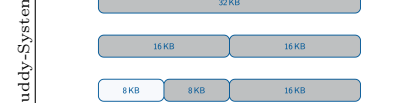
In **Bitliste** ein Bit pro Speicherblock: 0 → Block frei / 1 → Block verwendet  
Oder in **verketteter Liste** auch möglich:



**First Fit**: Wählt erste passende Lücke am Anfang  
**Next Fit**: Wählt erste passende Lücke nach zuletzt reserviertem Bereich  
**Best Fit**: Durchsucht alle Lücken und wählt die kleinste passende aus  
**Worst Fit**: Durchsucht alle Lücken und wählt die grösste aus

Bereiche werden nur in **bestimmten Grössen** zur Verfügung gestellt, z.B.: Alle Zahlen von 1 bis `m`. Alle freien Bereiche jeweils einer Grössenklasse werden in jeweils einer Liste vorgehalten.  
**Schnelle Reservation**: Erstes Element aus der Liste der kleinsten passenden Grösse  
**Nachteil**: Nachbarn sind nicht leicht zu finden

Variante des **Verfahrens Grössenklassen** mit Zweierpotenzen von  $2^m$  bis  $2^n$ .  $2^m$  ist die kleinste Speicherbereichsgrösse.  $2^n$  ist der gesamte Speicher.



Wird ein  $2^k$ -Bereich frei, wird **überprüft**, ob sein **Buddy** in der **Freiliste** ist.  
Falls Nein: Bereich in Freiliste einfügen.  
Falls Ja: Entferne Buddy aus Freiliste und Bereich Freigeben.

Virtueller Speicher

V-Adresse wird aufgeteilt: PageNr. + Offset

FrameNr. + Offset = Reale Adresse

Hauptspeicher besteht aus Frames (4KB)

V-Adressraum besteht aus Pages (4KB)

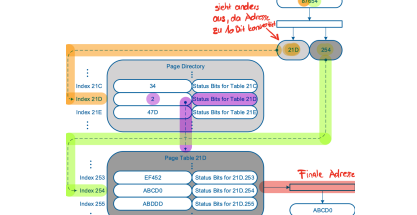


Bei ungültiger Adresse → MMU stellt fehlendes Mapping fest → Fault-Interrupt → OS-Interrupt-Handler → OS kopiert Pages in RAM und aktualisiert MMU.

Singel Level Page



Two Level Page



Multi Level Page

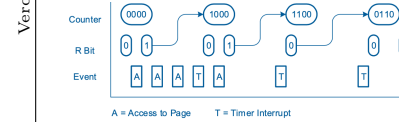
Gleich wie Two, nur mit mehreren Hierarchischen Tables (Pointer auf Directory)

In einem Page Eintrag gibt es ein **Dirty-Bit**, **Access-Bit** und **P-Bit** (ob im RAM oder nicht)

**Demand Paging** (Laden auf Anfrage) **Minimaler Aufwand** / **lange Wartezeiten**  
**Prepaging** (versucht frühzeitig zu laden) **In Praxis kaum anzutreffen**  
**Demand Paging mit Prepaging** (Wie D-Paging + benachbarte Page mitladen → nach Lokalitätsprinzip) **Weniger Page-Faults** / **Blocktransfer** / **nicht benötigte Pages werden mitgeladen**

**Demand Cleaning** Entladen auf Anfrage **nur geschrieben, wenn nötig** / **Erhöhte Wartezeit**  
**Precleaning** Vorausschauend Schreiben **Reduzierte Wartezeit** / **Mehr Aufwand**  
**Page Buffering** (Zwei Listen: Page Numbers der unveränderten Pages / veränderten Pages) MMU setzt D-Bit beim schreiben → OS verschiebt dann zu veränderten Pages **± wie Precleaning**

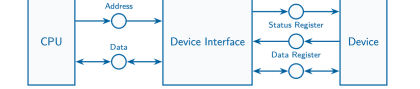
**FIFO** (Ersetzt jeweils älteste Seite) **Häufig benutze gelöscht/geladen/gelö/.**  
**Second Chance** (Prüft A-Bit der Ältesten Seite) ==0 → weg / ==1 → OS löscht A-Bit und schreibt an ende der Liste  
**Clock** (Wie SC, nur Kreis mit Pointer)  
**Least Recently Used** (Page wird grösser → HW misst Zeit) **MMU kann das oft nicht mit Interrupt** (OS misst Zeit → R-Bit wird gelöscht nach Timer)  
**Not Frequently Used** (zusätzliche Counter Table) Pro Eintrag ein Counter. Zugriff: Counter++/Löschen: Tiefster Counter  
**NFU mit Aging**



Ein- und Ausgabe

Viele Geräte sind Ein- sowie Ausgabegeräte. Deshalb ist die Kommunikation konzeptionell über Register.

- Gerät stellt Register bereit
  - CPU kann Register lesen oder schreiben
  - Gerät hat Adress-Pins und Daten-Pins
  - System hat Adress- und Datenbus
- Geräte werden mit einer Schnittstelle verbunden, da direktes verdrahten mit der CPU zu aufwändig wäre.



**Memory-mapped I/O**: Jedes Gerät ist am Speicherbus und bekommt einen reservierten Adressbereich. Dieser Bereich kann nicht für Speicher verwendet werden. **CPU** muss **keine Logik** haben / nicht wissen was Geräte sind. **Adressraum des Speichers** kann nicht für Speicher verwendet werden.

**Port-mapped I/O**: Geräte haben einen separaten Bus und Adressräume. Die CPU hat dann zwei Adressräume und pro Raum eigene Instruktionen. **Speicherverwendung** / **Komplexität**

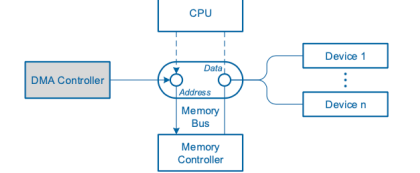
**Port-mapped I/O via Speicherbus**: Geräte sind am Speicherbus, der aber eine zusätzliche Bitleitung hat. Diese gibt an, ob für Speicher oder I/O. (Etwas von beidem der anderen 2)

Kommunikationsmechanismen

**Polling** (Programmgesteuert): Dies ist ein einfacher Mechanismus: CPU fragt regelmässig das Gerät. Programm pollt Statusregister. Wenn OK → Programm kann schreiben/lesen **Polling mit Busy Wait**: Software pollt hintereinander weg, **keine Verzögerung** / **legt CPU lahm sehr unprofessionell**

**Polling ohne Busy Wait**: Software pollt in regelmässigen Abständen gemäss einem vordefiniertem Zeitintervall. **Arbeiten an anderen Aufgaben möglich** / **erfordert genaue zeitliche Analyse**

**Interruptgesteuert**: Die Idee ist es, dass das Gerät die Software unterbricht (sendet Interrupt), sobald es bereit ist. Die CPU prüft nach (fast) jeder Instruktion, ob Interrupt Pin gesetzt ist. Falls ja: Unterbricht normale Ausführung → Sichert Instruktor-Pointer und Kontext (auf dem Stack) → holt sich die Nummer n des Interrupts → springt zum Interrupt-Handler, der an Index n in der Interrupt Vector Table notiert ist → CPU fährt mit der Aufgabe wieder fort. **Direct Memory Access (DMA)**: Die Idee ist es, dass das Gerät direkt auf den Speicher zugreift. Dafür benötigt es zusätzlich einen DMA-Controller. Der Controller steuert den Speicherbus anstelle der CPU. Aus CPU-Sicht ist es nur ein weiteres Gerät.



1. CPU programmiert DMA für Transfer: Quelle, Ziel, Menge, Betriebsart
2. CPU gibt Speicherbus an DMA frei
3. DMA lässt Gerät direkt in Speicher kopieren
4. DMA legt Adresse auf Speicherbus
5. Gerät legt Daten auf Speicherbus
6. Nach Beendigung setzt DMA Interrupt