Friedrich-Alexander Universität Erlangen-Nürnberg
Technische Fakultät
Department Informatik
Lehrstuhl für Informatik 3 (Rechnerarchitektur)

Klausurthemenzusammenfassung der Veranstaltung

Grundlagen der Rechnerarchitektur und -organisation (GRa)

gehalten im Sommersemester 2017
von Prof. Dr. Dietmar Fey

Inhaltsverzeichnis

Seite

1 Grundprinzipien einer Rechnerarchitektur **3**
1.1 URA . . . 3
1.2 Befehlszyklus . . . 3
1.3 Abweichungen vom URA-Prinzip . . . 4
Alternative 1: Neuronale Rechner . . . 4
Alternative 2: Datenflussrechner . . . 5
Alternative 3: Systolische Rechner . . . 5
Modifikationen am URA-Prinzip . . . 5

2 Hardwarenahe Programmierung in Assembler **6**
2.1 Quellprogramm ↦ Prozess . . . 6
2.2 Assemblerprogrammierung am Beispiel des MIPS-Assemblers . . . 7
Registerstruktur im MIPS . . . 7
Adressierungsarten . . . 8
Behandlung von Variablen . . . 8
Behandlung von Kontrollstrukturen . . . 8
Behandlung von Unterprogrammen . . . 11

3 Architektur von Rechnern und Prozessoren **16**
3.1 Leitwerk: Mikroprogrammierung . . . 16
Warum braucht man Mikroprogramme? – Vorteile der Mikroprogrammierung . . . 16
Horizontale Mikroprogrammierung (siehe Abb. 3.2c) . . . 18
Vertikale Mikroprogrammierung (siehe Abb. 3.2b) . . . 18
Organisation Leitwerksspeicher . . . 19
3.2 Leitwerk: Behandlungen von Unterbrechungen . . . 19
Behandlung von Interrupts . . . 20
Behandlung von mehreren Interrupts . . . 20
3.3 Befehlssatzarchitekturen . . . 21
Beispielprogramme für die Architekturen in Abbildung 3.6 . . . 22
3.4 Endianness . . . 23
3.5 Alignment . . . 23
3.6 Speicherwerk: Speicherhierarchie . . . 25
Charakteristika eines Speichers . . . 25
3.7 Speicherwerk: Grundlegende Cache-Techniken . . . 26
Zeitliche und Räumliche Lokalität . . . 26
Zugriff auf Daten . . . 26
Organisationsformen . . . 26
Wirtschaftlichkeit . . . 28
Aktualisierungsstrategie . . . 28
Ersetzungsstrategie . . . 28
Klassifikation von Fehlzugriffen — Die drei Cs . . . 29

3.8 Aufbau des Arbeitsspeichers . . . 29
SRAM – Static Random Access Memory . . . 29
DRAM – Dynamic Random Access Memory . . . 30
Speicherverschränkung . . . 30
Synchroner DRAM . . . 31
DDR-RAM . . . 32
3.9 Kopplung des Speichers mit der Ein- und Ausgabe . . . 32
Adressierungsmodi der Peripherie . . . 32
Programmierte Ein-/Ausgabe . . . 33
Unterbrechungsgesteuerte Ein-/Ausgabe . . . 33
Direkter Speicherzugriff . . . 33

4 Architekturen moderner Prozessoren 35
4.1 CISC-Architekturen . . . 35
4.2 RISC-Architekturen . . . 35
Pipelining . . . 37
Superskalare Architekturen . . . 39
4.3 VLIW – Very Long Instruction Word . . . 39
Beispiel . . . 40
4.4 Multi-Threading . . . 41
Zeitscheiben Multithreading . . . 42
Ereignisgesteuertes Multithreading . . . 42
Simultanes Multithreading . . . 42

5 Schnittstelle zum Betriebssystem 43
5.1 Anbindung zum Betriebssystem . . . 43
Aufgaben des Betriebssystems . . . 43
5.2 Memory Management . . . 43
Virtueller Adressraum – Virtueller Speicher . . . 43
Paging . . . 44
Segmentierung . . . 49
Segmentierung + Paging . . . 50
Vergleich der Segmentierung zur Seitenumlagerung . . . 51

Gefunden in den Klausuren SS2009, WS2009[1], sowie ab WS13[2] — Gefunden in Präsenzübung 0.3

[1]: *als eigenständige Aufgabe*
[2]: *in Teilaufgaben verstreut*

Grundprinzipien einer Rechnerarchitektur

1.1 URA

Definition(*Klassischer Universalrechenautomat*)

Der klassische Universalrechenautomat (URA) ist an sieben Prinzipien definiert:

1. Der Rechner besteht aus 4 Werken
   Leitwerk, Speicherwerk, Rechenwerk, Ein-/Ausgabewerk
2. Die Struktur des Rechners ist unabhängig vom Problem — **programmgesteuert**
3. Programme und Daten stehen in **demselben** Speicher, sind prinzipiell **modifizierbar**
4. Hauptspeicher ist in Zellen gleicher Größe eingeteilt, die durch fortlaufende Nummern (Adresse) benannt werden
   über die Adresse werden Daten und Programmbefehle angesprochen
5. Programm besteht aus einer Folge von Befehlen, die im Allgemeinen nacheinander ausgeführt werden (**Prinzip der Sequentialität als Fortschaltungsregel**)
6. Abweichungen von der sequentiellen Ausführung der Befehle sind **nur** durch **bedingte** und **unbedingte** Sprungbefehle möglich.
7. Der URA ist **binär** kodiert

1.2 Befehlszyklus

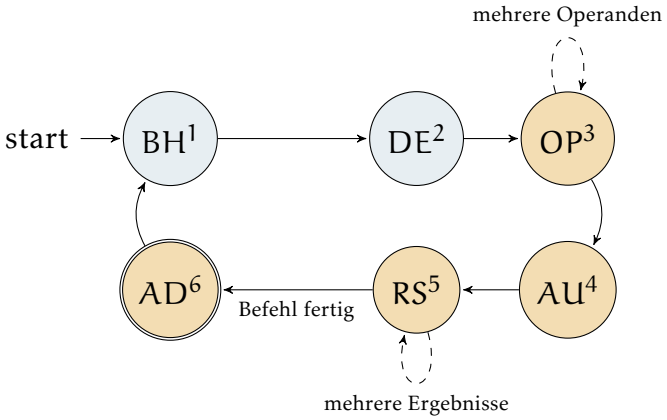
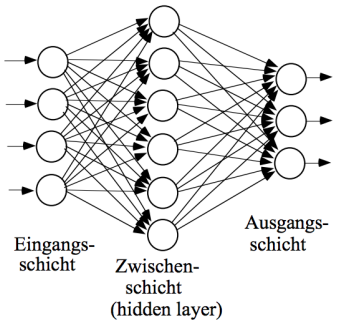
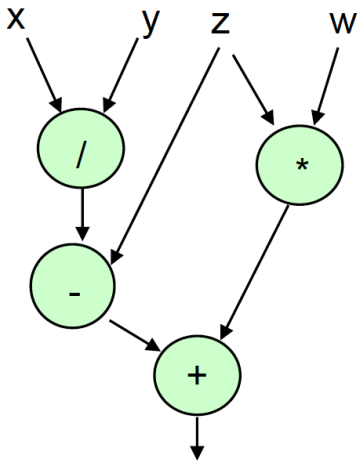


Abbildung 1.1: Erweiterter Befehlszyklus eines Universalrechenautomats

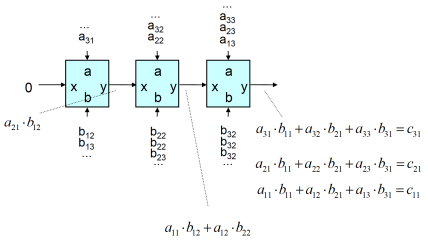
Der klassische IAS-Rechner verzichtet auf eine so spezifische Unterteilung in Einzelschritte. Hier gibt es nur 2 Unterzyklen (BH) und (EX).



(a) Schematische Zeichnung eines neuronalen Netzes



(b) Schematische Zeichnung eines Datenflussrechners für die Berechnung von $\frac{x}{y} - z + z \cdot w$



(c) Schematische Zeichnung eines systolischen Rechners nach Takt 5 für eine Matrix-Matrix-Multiplikation, Prozessorknoten berechnen $y = a \cdot b + x$

Abbildung 1.2: Alternativen im URA-Prinzip

Erläuterung 1 (*BH — Befehlsholphase*)

Auf Basis des **Befehlszählers** wird der nächste zu bearbeitende Befehl aus dem Speicher in das Instruktionsregister geladen.

Erläuterung 2 (*DE — Dekodierungsphase*)

Aus dem eben gelesenen, nun dekodierten Operationskode werden die Steuersignale generiert.

Erläuterung 3 (*OP — Operandenholphase*)

Stellt der ALU die im Adressteil des Maschinenbefehls spezifizierten Operanden zur Verfügung

Erläuterung 4 (*AU — Ausführungsphase*)

Verknüpft in den Registern des Rechenwerks die zuvor geholten Operanden.

Erläuterung 5 (*RS — Rückschreibphase*)

Die während der Ausführungsphase produzierten Ergebnisse werden in die vorgesehenen Speicherstellen (Speicher, Register) zurückgeschrieben.

Erläuterung 6 (*AD — Adressierungsphase*)

Die Adresse des nächsten Befehls wird bestimmt, der Befehlszähler entsprechend gesetzt.

1.3 Abweichungen vom URA-Prinzip

Alternative 1: Neuronale Rechner

Vorbild: Menschliches Gehirn
Das künstliche neuronale Netz besteht aus mehreren Neuronenschichten mit verschiedenen künstlichen Neuronen. Diese bekommen Eingaben anhand derer dann entschieden wird, ob ein Neuron zündet oder nicht. Eine solche Funktion wird nun beispielsweise anhand der Neuronen des `Hidden Layers`[1] beschrieben:

[1] Mittlere Schicht in Abb. 1.2a

- Die Neuronen bekommen als Eingaben alle Ausgaben der Neuronen der vorherigen Schicht $x_{0,j}^{out}$
- Jede dieser Eingaben wird mit einem Gewicht $w_{0,j}$ multipliziert und danach aufsummiert:

$$S_g = \sum_j \left(x_{0,j}^{out} \cdot w_{0,j}\right)$$

- Ist die Summe S_g größer als eine Grenzschwelle ϑ_G so zündet das Neuron, andernfalls nicht.

Den Lernprozess kann man sich als rießige Lookup-Tabelle vorstellen. Das Trainieren eines neuronalen Netzes ist das Lösen eines Gleichungssystems, in dem die Gewichte $w_{i,j}$ bestimmt werden, also nichts anderes als eine Matrix-Vektor-Multiplikation.
Als Deep Learning bezeichnet man eine Konkatenation von Neuronalen Netzen.

Alternative 2: Datenflussrechner

Hier lautet die Devise: Bring die Operanden zu den Operatoren. Hochzeit in den 50er Jahren, heute wieder ein bisschen aktuell in systolischen Rechnern. Wenig klausurrelevant ...

Alternative 3: Systolische Rechner

Kombination aus Datenfluss- und SIMD-Prinzip. Die blauen Kästen (Prozessorknoten) machen nichts anderes als eine Operation zu berechnen und am Ausgang auszugehen. Der Rechner ist ein synchron getaktetes System. Die Ein-/Ausgabe über am Rand angeordnete Prozessorknoten.
Anwendung: Digitale Signalverarbeitung

Modifikationen am URA-Prinzip

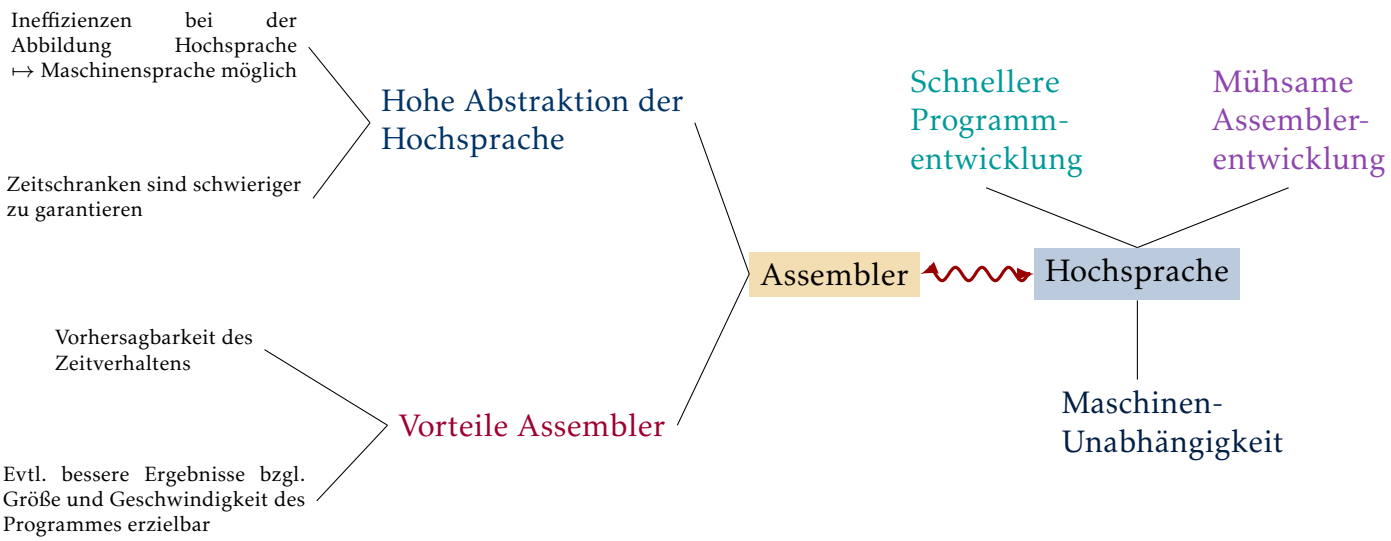
- Verfielfachung einer oder mehrerer Teilwerke
  - Mehrere E-/A-Werke, um Ein-/Ausgabe zu beschleunigen bzw. den Datendurchsatz zu erhöhen
  - Mehrere Leit- und Rechenwerke, um mehrere Befehle gleichzeitig zu bearbeiten
- Anstelle der zweistufigen Speicherhierarchite $\rightarrow$ mehrstufige Hierarchie
  - Besseres Preis/Leistungsverhältnis $\Rightarrow$ Mehrstufige Hintergrundspeicher
  - „von Neumannscher Flaschenhals“ $\rightsquigarrow$ Cache
- Getrennte Speicher und Busse für Daten und Befehle
- Prinzip der Selbstmodifikation aus Sicherheitsgründen aufgegeben ($\rightarrow$ ist für rekonfigurierbare Hardware wieder aktuell geworden)

*Gefunden in **allen** Klausuren*
Assembler-Sprache ist die symbolische Darstellung der Maschinensprache eines Rechners

Patterson/Hennessy; Computer Organization & Design

KATEGORIE 2

Hardwarenahe Programmierung in Assembler



2.1 Quellprogramm ↦ Prozess

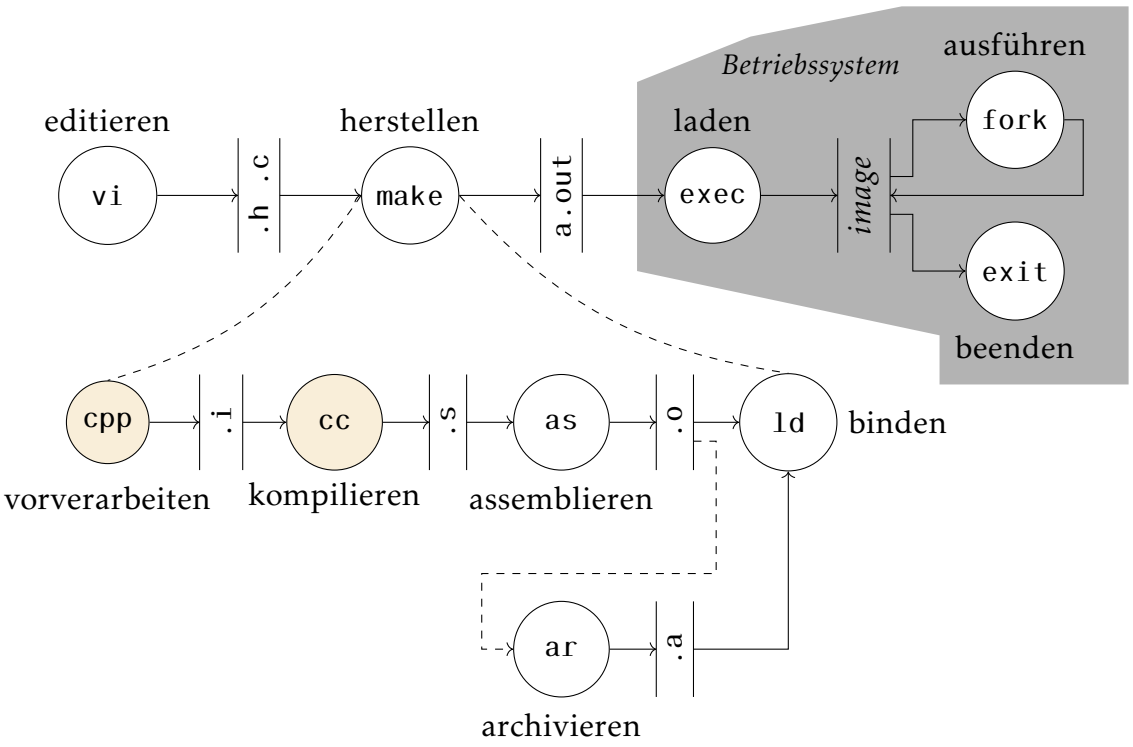


Abbildung 2.1: Die sogenannte „Toolchain“: Beim Assemblerprogramm werden die gelb markierten Knoten übersprungen

2.2 Assemblerprogrammierung am Beispiel des MIPS-Assemblers

Registerstruktur im MIPS

Bezeichnung		Bedeutung des Registers
Numerisch	Symbolisch	
`$0`	`$zero`	**Nullregister** Liefert beim Lesen immer 0 und ignoriert Schreibzugriffe
`$1`	`$at`	**Assembler Temporary** Wird vom Assembler als temporäres Register benutzt, um Pseudobefehle zu implementieren.
`$2-$3`	`$v0-$v1`	**Ergebniswertregister** Diese Register enthalten die Rückgabewerte von Funktionen
`$4-$7`	`$a0-$a3`	**Argumentregister** Diese Register dienen der Übergabe von Argumenten an Funktionen.
`$8-$15`	`$t0-$t7`	**Temporäre Register** Beliebig verwendbare **Caller**-Save-Register.
`$16-$24`	`$s0-$s7`	**Preserved Register** Beliebig verwendbare **Callee**-Save-Register.
`$24-$25`	`$t8-$t9`	**Temporäre Register** Beliebig verwendbare **Caller**-Save-Register.
`$26-$27`	`$k0-$k1`	**Kernel Register** Werden vom Betriebssystemkernel – u.a. während Traps oder Interrupts – verwendet. In normalen Programmen daher auf **gar keinen Fall** verwenden.
`$28`	`$gp`	**Global Pointer** Bei Verwendung von Position-Independent Kode wird dieses Register dazu verwendet um die Differenz zwischen Text- und Datensegment mitzuteilen.
`$29`	`$sp`	**Stack Pointer** Zeigt auf das momentan letzte Element des Stacks. Der Stack wächst zu kleineren Speicheradressen hin.
`$30`	`$fp`	**Frame Pointer** Zeigt auf den gesicherten, alten Framepointer im Stack. Dient als Alternative zum Stackpointer zur Parameter- und Variablenadressierung
`$31`	`$ra`	**Return Adress** Speichert die Rücksprungadresse bei Funktionsaufrufen. `jal` speichert darin automatisch die Adresse der nächsten Instruktion nach dem Rücksprung.

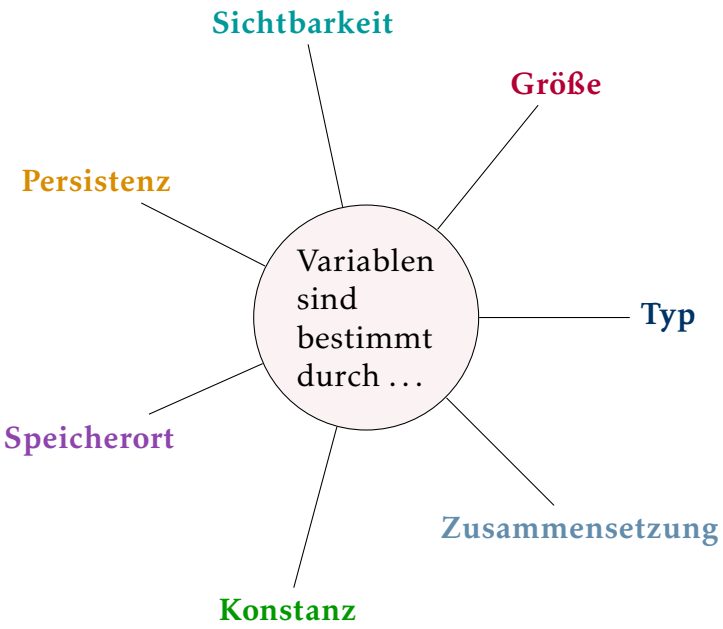
Tabelle 2.1: Registerstruktur des MIPS-Assemblers nach SysV-ABI

Adressierungsarten

Addresierungsart \ Assembler	MIPS
Register	$t0
Unmittelbar	123
Register indirekt	($t0)
Register displacement	12 ($t0) um wie viel wird verschoben

Tabelle 2.2: Adressierungsarten beim MIPS-Assembler

Behandlung von Variablen



Der Speicherort einer Variablen wird von drei Dingen bestimmt: Ihrer **Persistenz**, **Konstanz** und **Sichtbarkeit**.
Persistente Variablen erhalten eine Speicherstelle im Datensegment, nicht persistente Variablen auf dem Stack (→ Stackpointer).
Zusammengesetzte Variablen können auf primitive Variablen abgebildet werden:

$$\texttt{int x[4]} \Longrightarrow \texttt{int } x_0, x_1, x_2, x_3$$

Die Adressierung der Elemente 1 mit 3 erfolgt dann über Displacement:

$$\begin{aligned} x[0] &= x \\ x[1] &= x + 1 \cdot Size_{int} \\ x[i] &= x + i \cdot Size_{int} \end{aligned}$$

Beispiel

Gegeben sei folgendes Programm in C:

```
1  int i;
2  int x[4] = {5, 6, 7, 8};
3
4  int some_func(void) {
5      i = 42;
6      x[2] = 42;
7  }
```

In MIPS sieht das ganze dann wie folgt aus:

```
1  .data
2  i:
3      .byte 00, 00, 00, 00
4  x:
5      .byte 05, 00, 00, 00
6      .byte 06, 00, 00, 00
7      .byte 07, 00, 00, 00
8      .byte 08, 00, 00, 00
9  .text
10 some_func:
11     li $t1, 42
12     sw $t1, i
13     la $t0, x
14     addi $t0, $t0, 8
15     sw $t1, ($t0)
16     jr $ra
```

$8 = 2 \cdot \underbrace{Size_{Int}}_{=4}$

Behandlung von Kontrollstrukturen

Neben den Variablen erscheint beim Umsetzen von einer Hochsprache in Maschinensprache noch ein weiteres Problem, das der Kontrollstrukturen. In Maschinensprachen gibt es nur Sprünge, welche von

Gruppe	Befehl	Bedeutung
Sprunganweisungen	`j immediate`	Springt zur Adresse `immediate`
	`jr $reg`	Springt zur im Register `$reg` gegebenen Adresse
Spring und Binde	`jal imm`	Im Grunde ähnlich zu `j` mit dem Unterschied, dass die Rücksprungadresse ins Register `$ra` geladen wird
	`jalr $reg [$raAlt]`	Im Grunde ähnlich zu `jr` mit dem Unterschied, dass die Rücksprungadresse ins Register `$raAlt`[1]oder `$ra` geladen wird
Zweiganweisungen	`beq $r1, $r2, add`	Springt genau dann zu `add`, wenn das Prädikat `$r1 = $r2` erfüllt ist.
	`bne $r1, $r2, add`	Springt genau dann zu `add`, wenn das Prädikat `$r1` $\neq$ `$r2` erfüllt ist.
	`bgtz $r1, add`	Springt genau dann zu `add`, wenn das Prädikat `$r1` > 0 erfüllt ist.
	`bgez $r1, add`	Springt genau dann zu `add`, wenn das Prädikat `$r1` $\geqslant 0$ erfüllt ist.
	`bltz $r1, add`	Springt genau dann zu `add`, wenn das Prädikat `$r1` < 0 erfüllt ist.
	`blez $r1, add`	Springt genau dann zu `add`, wenn das Prädikat `$r1` $\leqslant 0$ erfüllt ist.

Tabelle 2.3: Sprunganweisungen in MIPS

einer Bedingung abhängen können. Ein Analogon dazu bildet die `goto`-Anweisung in Hochsprachen. `for`, `while`, `do ... while` und `if-then-else` müssen also irgendwie durch das `goto` dargestellt werden.

if-then-else

Hochsprache:

```
1 if(praedikat) {
2     ...1
3 } else {
4     ...2
5 }
```

`if-goto`:

```
1 if(praedikat) goto Lthen;
2     ...2
3     goto LEnd;
4 Lthen:
5     ...1
6 LEnd:
7     ...
```

switch-case

Hochsprache:

```
1 switch(var) {
2     case v1: ...1 break;
3     case v2: ...2
4     default: ...d break;
5 }
```

`if-goto`:

```
1     if(var == v1) goto L1;
2     if(var == v2) goto L2;
3     goto LDefault;
4 L1:
5     ...1 goto LEnd;
6 L2:
7     ...2
8 LDefault:
9     ...d
10 LEnd:
11    ...
```

[1]wenn angegeben

while

Hochsprache:

```
1 while(praedikat1) {
2     if(praedikat2) continue;
3     if(praedikat3) break;
4     ...
5 }
6 ...
```

if-goto:

```
1 LBody:
2     if(praedikat2) goto LCond;
3     if(praedikat3) goto LEnd;
4     ...
5 LCond:
6     if(praedikat1) goto LBody
7 LEnd:
8     ...
```

for

Hochsprache:

```
1 for(...ini; praed; ...inc) {
2     ...b
3 }
4 ...
```

if-goto:

```
1     ...ini
2     goto LCond;
3 LBody:
4     ...b
5     ...inc
6 LCond:
7     if (praed) goto LBody;
8 LEnd:
9     ...
```

do ... while

Hochsprache:

```
1 do {
2     ...
3 } while(praedikat);
4 ...
```

if-goto:

```
1 LBody:
2     ...
3     if(praedikat) goto LBody
4 LEnd:
5     ...
```

AND-Prädikat

Hochsprache:

```
1 if(praedikat1 && praedikat2) {
2     ...1
3 } else {
4     ...2
5 }
```

if-goto:

```
1 if(!praedikat1) goto Lelse;
2 if(praedikat2) goto Lthen;
3 Lelse:
4     ...2
5     goto LEnd;
6 Lthen:
7     ...1
8 LEnd:
9     ...
```

OR-Prädikat

Hochsprache:

```
1 if(praedikat1 || praedikat2) {
2     ...1
3 } else {
4     ...2
5 }
```

if-goto:

```
1 if(praedikat1) goto Lthen;
2 if(praedikat2) goto Lthen;
3     ...2
4     goto LEnd;
5 Lthen:
6     ...1
7 LEnd:
8     ...
```

Behandlung von Unterprogrammen

```c
int readInt(void) {
    int r;
    r = scanf("%d", r);
    return r;
}

int readAndAdd(int i) {
    int j = readInt();
    return j + i;
}

void main(void) {
    int k = readAndAdd(42);
}
```

Generell sind Unterprogramme durch die folgenden Eigenschaften definiert:

- **Parameter**übergabe
- **Rückgabewerte**
- Auftreten von **lokalen** Variablen

→ verschiedene Möglichkeiten dies in Assembler umzusetzen

Parameterübergabe und Rückgabewerte

Rückgabewerte unterscheiden sich nicht wirklich von Parametern, als dass sie als Parameter beschrieben durch das Unterprogramm aufgefasst werden. Wir unterscheiden drei Möglichkeiten:

Möglichkeit 1: Speicher

```
.data
read_and_add_i:
    .byte 00, 00, 00, 00
read_and_add_erg:
    .byte 00, 00, 00, 00
read_and_add_j:
    .byte 00, 00, 00, 00
read_and_add_ra:
    .byte 00, 00, 00, 00
k:
    .byte 00, 00, 00, 00
.text
readAndAdd:
    sw $ra, read_and_add_ra
    jal read_int
    sw $v0, read_and_add_j
    lw $ra, read_and_add_ra
    lw $t0, read_and_add_j
    lw $t1, read_and_add_i
    add $t2, $t0, $t1
    sw $t2, read_and_add_erg
    jr $ra
...
main:
    li $t0, 42
    sw $t0, read_and_add_i
    jal readAndAdd
    lw $t1, read_and_add_erg
    sw $t1, k
    ...
```

Stellen im Speicher für die Parameter und die Ergebnisse werden zu Beginn festgelegt. Ein Beispiel der Umsetzung des obigen C-Programmes ist links gegeben.

Achtung

Man beachte die Notwendigkeit des Speicherplatzes read_and_add_ra aufgrund des zweiten Unterprogrammaufrufs von `readInt`.

Man stößt schnell auf verschiedene Probleme mit diesem Ansatz:

1. **Speicherverwendung ist langsam**:
   Im Vergleich zu Registern ist das Speichern und Laden von Werten aus dem Speicher extrem langsam. Vor allem in echtzeitkritischen Systemen sind diese Zugriffe damit nur beschränkt möglich.

2. **Keine Rekursion möglich**:
   Man betrachte das folgende Programm:

```c
long fak(int i) {
    if (i == 0) return 1;
    return i * fak(i - 1);
}
```

   Wir sehen schnell, dass hier Parameter nicht über den Speicher übergeben werden können.[2]

Möglichkeit 2: Register

```
1  .data
2  k:
3      .byte 00, 00, 00, 00
4  .text
5  readAndAdd:
6      move $s0, $ra
7      jal read_int
8      move $t0, $v0
9      move $ra, $s0
10     add $t1, $t0, $a0
11     move $v0, $t1
12     jr $ra
13 ...
14 main:
15     li $a0, 42
16     jal readAndAdd
17     sw $v0, k
18     ...
```

Anstelle vorher definierter Speicherstellen werden verschiedene Register für die Übergabe von Parametern und Rückgabewerten verwendet. Welche Register wie zu verwenden sind, findet sich in der Registerübersicht[3]. Ein Beispiel der Umsetzung des obigen C-Programmes ist oben gegeben.

Achtung

Man beachte die Notwendigkeit des Speicherns des Rücksprungregisters $ra aufgrund des zweiten Unterprogrammaufrufs von `readInt`.

Möglichkeit 3: Stack

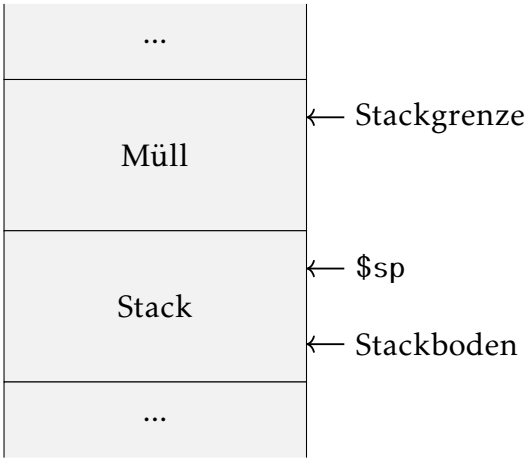


Abbildung 2.2: Schaubild eines exemplarischen Stackaufbaus

Man stößt auch bei diesem Ansatz schnell auf Probleme:

1. **Anzahl an Registern stark begrenzt**
2. **Keine Garantie auf Registerinhalte möglich**:
   Man stelle sich nun folgende Situation vor:

```
1  readInt:
2      li $a0, 213
3      li $v0, 5
4      syscall
5      jr $ra
```

   In diesem Fall überschreibt das Unterprogramm das Register, dessen Inhalt vom Aufrufer noch benötigt wird, und verfälscht damit das Ergebnis. ↯(⇝ Lösung über Stack)
3. **Keine Rekursion möglich**:
   Man betrachte das folgende Programm:

```
1  long fak(int i) {
2      if (i == 0) return 1;
3      return i * fak(i - 1);
4  }
```

   Wir sehen schnell, dass hier Parameter nicht über Register übergeben werden können. Es ist eine viel zu kleine Anzahl an Registern verfügbar und Werte werden dann immer wieder überschrieben, obwohl sie noch gebraucht werden.

Definition(*Stack*)

Bezeichnung eines Datentyps fur eine Sammlung von Daten, auf der zwei grundlegende Operationen definiert sind: *push* für (dt.) vordrängen, um ein Datum oben auf dem Stoß abzulegen und *pop* für (dt.) herausholen, um ein Datum oben von dem Stoß zu entfernen. Eine spezielle Art von dynamischer Datenstruktur, bei der die Elemente umgekehrt zur Einreihungsfolge entfernt werden (LIFO).

[2]Eine Rekursionstiefe ist vorher nicht bekannt und die Parameter werden **nach** dem Rekursionsschritt nocheinmal gebraucht.

[3]siehe Tabelle 2.1

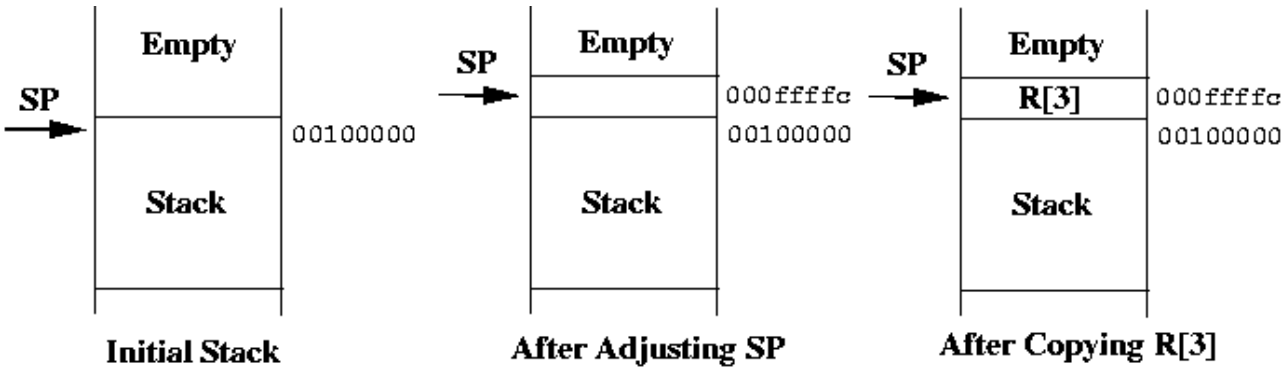


Abbildung 2.3: Beispiel für die Push-Operation auf einem Beispielstack

Erläuterung 7 (*Stackpointer*)

Der Stackpointer sp bezeichnet ein Register, welches auf das obere Ende des Stacks zeigt. Es gilt: $addr(sp) = \min\{x \in Addr_{valid} : \forall y \in Addr_{valid} : y \geqslant x \Rightarrow y \text{ enthält Müll}\}$

Erläuterung 8 (*Stackgrenze, Stackboden*)

Der Stackboden ist die größt mögliche Adresse im Stack. Wenn ein Stack initialisiert wird, so zeigt der Stackpointer auf den Stackboden.
Die Stackgrenze stellt die minimal möglichste Adresse im Stack dar. Sollte der Stackpointer auf eine Adresse kleiner als die Grenze zeigen, kommt es zum *stack overflow*.

Manche Systeme verfügen über bereits integrierte Befehle für die **push**- und **pop**-Operationen, MIPS nicht. Hier muss dies manuell über das Verändern des Stackpointers geschehen.
Per Konvention ist in MIPS das Register 29 das Register für den Stackpointer.

Push-Operation

```
1 push:
2     addi $sp, $sp, -4
3     sw $r3, 0($sp)
```

Die Push-Operation in MIPS ist eigentlich relativ simpel. Man **erniedrigt** den Stackpointer um 4[4] und speichert das Register an die Stelle des Stackpointers. Eine Veranschaulichtung des Vorgangs findet sich in Abbildung 2.3.

Pop-Operation

```
1 pop:
2     lw $r3, 0($sp)
3     addi $sp, $sp, 4
```

Die Pop-Operation in MIPS ist ebenfalls relativ simpel. Man holt das Register von der Stelle des Stackpointers und **erhöht anschließend** den Stackpointer um 4[5]. Eine Veranschaulichtung des Vorgangs findet sich in Abbildung 2.4.

Achtung

Die Veränderung des Stackpointers ist wichtig, um gleiche Vorraussetzungen zu schaffen. So ist nach dem Hinzufügen des Registers zum Stack bei nebenstehendem Kode die Bedeutung des Stackpointers gewahrt. Der Kode bleibt wiederverwendbar ohne wichtige Daten zu überschreiben.

[4]Der Stack wächst ja von groß nach klein → Erniedrigung
[5]Der Stack wächst ja von groß nach klein → Erhöhung

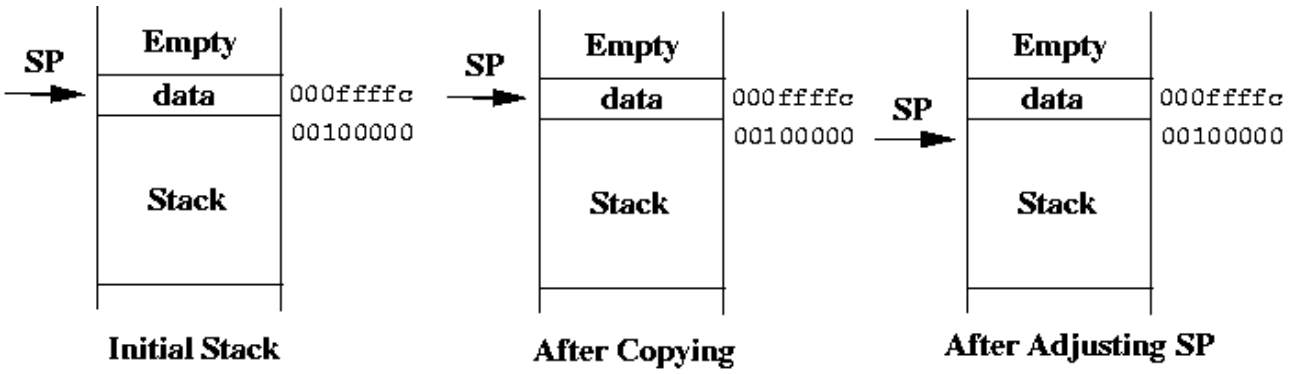


Abbildung 2.4: Beispiel für die Pop-Operation auf einem Beispielstack

```
1  .data
2  k:
3      .byte 00, 00, 00, 00
4  .text
5  read_and_add:
6      addi $sp, $sp, -4
7      sw   $ra, 0($sp)
8      addi $sp, $sp, -4
9      sw   $a0, 0($sp)
10     jal read_int
11     move $t0, $v0
12     lw $a0, 0($sp)
13     addi $sp, $sp, 4
14     add $t0, $t0, $a0
15     move $v0, $t0
16     lw $ra, 0($sp)
17     addi $sp, $sp, 4
18     jr $ra
19 main:
20     li $a0, 24
21     jal read_and_add
22     sw $v0, k
```

Hier stößt man nur auf ein großes Problem, die **Geschwindigkeit** der Anweisungen. Ähnlich wie bei Möglichkeit 1 festgestellt sind Speicherzugriffe (auch auf den Stack) langsam und teuer. MIPS verfügt über viele Register, welcher unteranderem auch für die Parameterübergabe und Wertrückgabe vorgesehen sind. Wir verwenden diese deshalb, vor Unterprogrammaufrufen, welche die Registerinhalte verändern können, speichern wir **diese** auf dem **Stack**.
Zurück zu unserem Beispielproblem vom Beginn. In MIPS sieht das ganze dann wie links dargestellt aus.

Stack-Frame und Framepointer

Definition(*Stack-Frame*)

> Der Bereich eines Unterprogramms bestehend aus **Parametern**, der **Rücksprungadressse**, den **lokalen Variablen** und dem Platz zum Sichern von Registern wird als **Stack-Frame** bezeichnet.

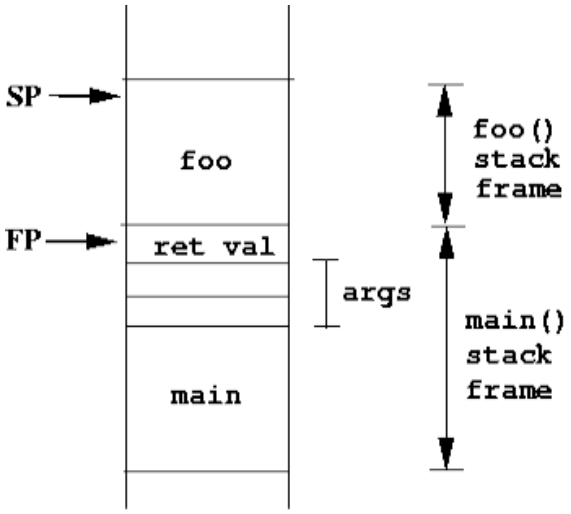


Abbildung 2.5: Beispielausartung eines Stacks mit mehreren Stackframes

Indem wir Werte auf dem Stack speichern, vergrößern wir den Stack-Frame unseres Unterprogramms. In Abbildung 2.5 sind die Stack-Frames zweier Methoden eingezeichnet. Zudem ist nebem dem Stackpointer (**SP**) auch noch ein Framepointer (**FP**) eingezeichnet.
Die Konvention besagt, dass der Framepointer an die Stelle zeigt, an der der Stackpointer **vor** dem Aufruf des Unterprogramms war. Damit ist es leicht den Stackpointer am Ende der Prozedur wieder zurückzusetzen, ohne, dass man mitzählen muss wie oft der Stackpointer verschoben wurde.

Behandlung von Registerinhalten
Wie in den Problemen zu Möglichkeit 2 bereits erwähnt, kann ein Unterprogramm Register für eigene Zwecke verwenden, und damit noch benötigte Werte überschreiben. Die Lösung zu dem Problem war eine Sicherung der Werte auf den Stack. Wir unterscheiden hier zwischen **caller**-save- und **callee**-save-Registern.

Definition(*caller-save-Register*)

Register, welche per Konvention als caller-save-Register ausgezeichnet sind, sind stets vom **Aufrufer** des Unterprogramms zu sichern, sofern Werte daraus noch benötigt werden.

Definition(*callee-save-Register*)

Bei Registern, welche per Konvention als callee-save-Register ausgezeichnet sind, kann der Aufrufer davon ausgehen, dass nach dem Funktionsaufruf die Werte immer noch vorhanden sind.
Der Entwickler des Unterprogramms muss also die alten Werte wiederherstellen (und zwischenspeichern), sollte eines dieser Register verwendet werden.

Architektur von Rechnern und Prozessoren

3.1 Leitwerk: Mikroprogrammierung

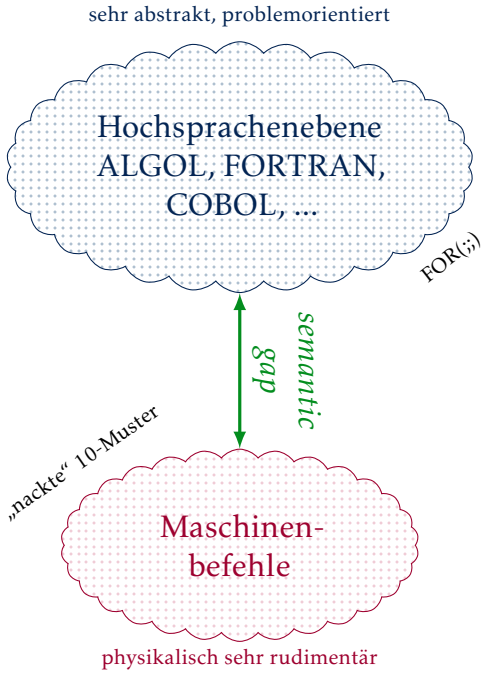
Als ursprünglich akademisches Konzept nach Wilkes[1] mit dem IBM System in der 360er Reihe zu kommerziellem Erfolg gekommen.

Definition(*Mikroprogrammierung*)

Als **Mikroprogrammierung** wird die Methode und das Vorgehen verstanden, Schalt-, Rechen- und Steuerabläufe in einem Rechenwerk, insbesondere einem **integrierten Rechenwerk** (Mikrocontroller, Mikroprozessor), unterhalb der Ausführungsebene von Maschinenbefehlen auf Mikrobefehlsebene festzulegen und zu regeln. Die Programme beschreiben grundlegende architektonische Merkmale, die der Prozessor, in dem es als Firmware – für gewöhnlich nicht flüchtig, aber durchaus änderbar – gespeichert vorliegt, aufweisen soll. Das Programm legt dann wesentliche funktionale Eigenschaften des Prozessors fest und schafft damit die Grundlage für die Ausprägung als **CISC** (siehe Kategorie 4), ein RISC ist meistens **nicht** mikroprogrammiert; die einfachen Befehle sprechen direkt die Funktionseinheiten des Prozessors an und brauchen keine algorithmische Form.

Warum braucht man Mikroprogramme? – Vorteile der Mikroprogrammierung

1. **Es gibt eine semantische Lücke zwischen Hochsprachen und Maschinenbefehlen, die überwunden werden muss.**



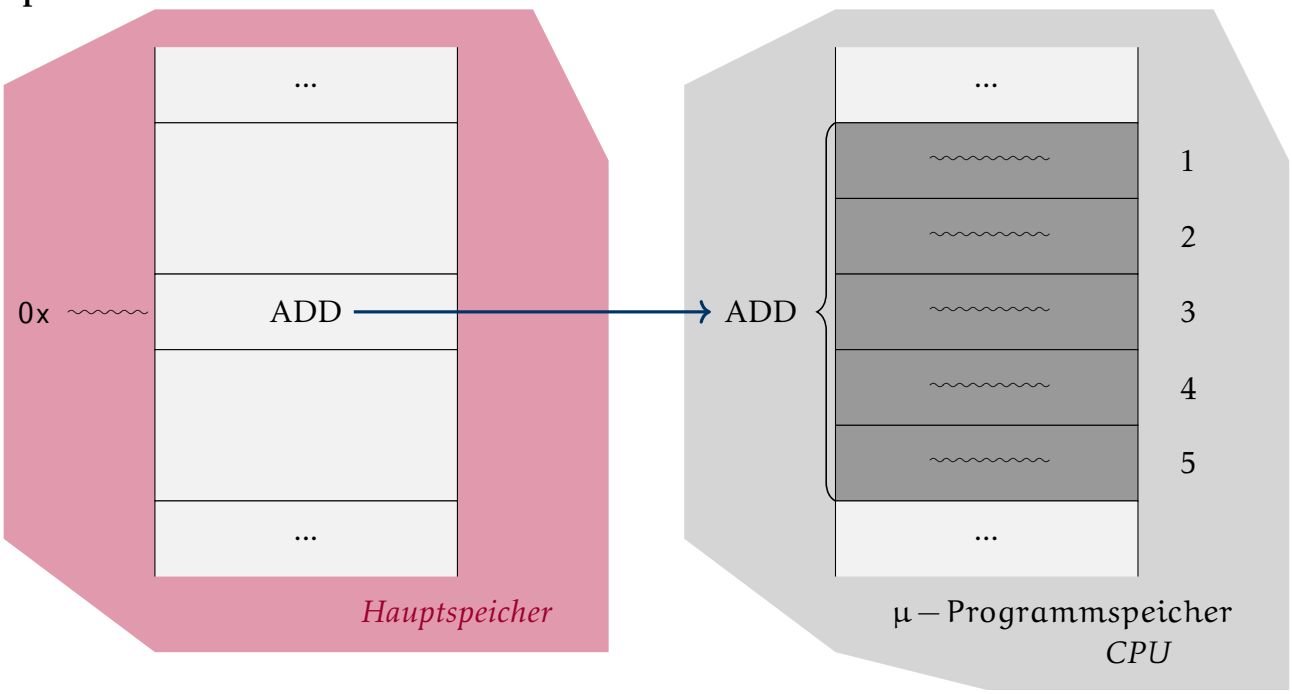
Definition(*Semantische Lücke*)

Bedeutungsbezogener Unterschied zwischen den Beschreibungen von zwei Sprachebenen in einem mehrschichtig organisierten Rechensystem, für das eine bestimmte hierarchische Struktur definiert ist. Ursprünglich begründet in der Diskrepanz zwischen den komplexen Operationen der Konstrukte einer höherer Programmiersprache und den einfachen Operationen einer CPU

Es gibt quasi eine Lücke zwischen dem Problem und der Umsetzung des Problems auf Maschinenebene. Nackte 10-Muster wie 100101011101000010 sind für den Menschen schwer zu verstehen, problemorientierte Lösungen sind so für die CPU schwer zu verstehen. Diese Lücke ist von semantischer Natur, deswegen auch semantische Lücke, die Mikroprogrammierung versucht und schafft es diese Lücke zu schließen.

[1] erstmals 1951 in seinem Artikel "The Best Way to Design an Automated Calculating Machine,"Manchester University Computer Inaugural Conf., 1951, pp. 16-18. erwähnt

2. **Speicher ist und war ein kostbares Gut**



An dem Beispiel sieht man: Statt einer Vielzahl an Instruktionen (hier: 5) braucht es nur **eine** Instruktion im kostbaren Hauptspeicher!

3. **Flexibilität in der Programmierung**
Man erreicht hohe Flexibilität unter anderem durch einen veränderbaren Mikroprogrammspeicher. Dadurch, dass die Operationen hier nicht fest verdrahtet sind, sondern auf Mikroinstruktionen aufbauen, die in der Firmware sind (bspw. BIOS auf Flash-Speicher), sind diese per se aktualisierbar. Bei einem Fehler in einer Subroutine – überzogen: ADD addiert nicht sondern subtrahiert — kann diese ausgetauscht werden **ohne** dass sich etwas für die darüberliegenden Schichten verändert oder die CPU ausgetauscht werden muss. Ebenso können neue Befehle hinzugefügt werden ohne dass die Hardware groß verändert werden müsste.

Zusammenhang Makrobefehl – Mikrobefehl

Makrobefehle (wie zum Beispiel Assemblerbefehle) bilden den Einstieg in ein Mikroprogramm. Das *Instruction Register* ist ein Makrobefehl, der mit dem *Dekoder 1* dekodiert wird zur Einstiegsadresse, an der die Mikrobefehele im *Control Memory* abgelegt sind. Der *Dekoder 2* dekodiert dann die Befehle für die CPU[2]. Die *Sequencing Logic* bestimmt abhängig von den ALU Flags und dem Ergebnis des *Dekoder 2* die kommende Adresse.
Das *Control Address Register* enthält die Adresse der **nächsten** Instruktion, das *Control Buffer Register* nimmt den Inhalt aus dem *Control Memory* auf, enthält quasi das Wort der Adresse des *Control Address Register* einen Takt zuvor.

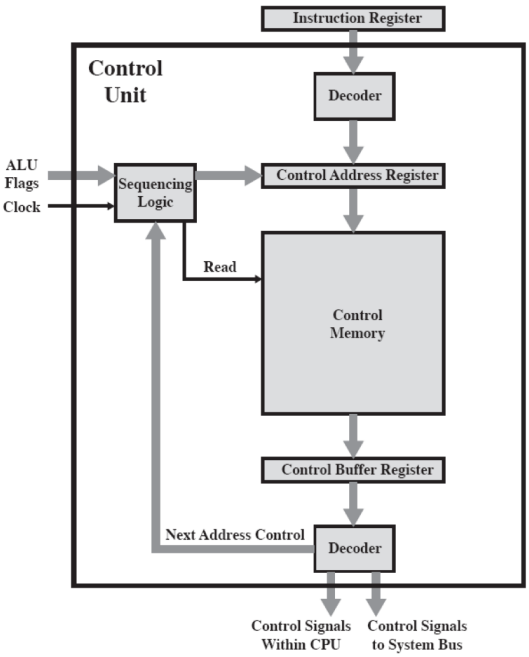
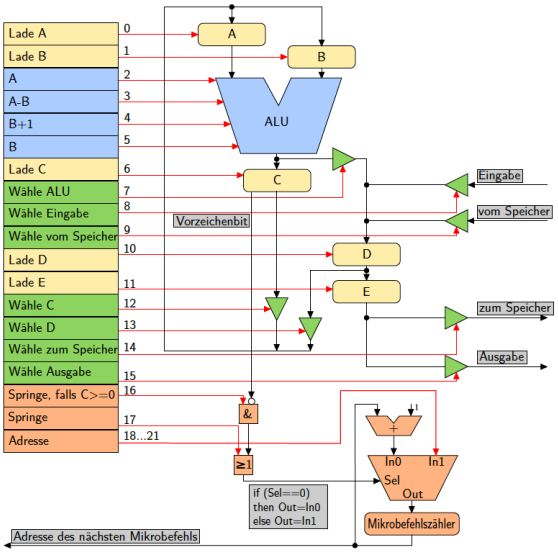


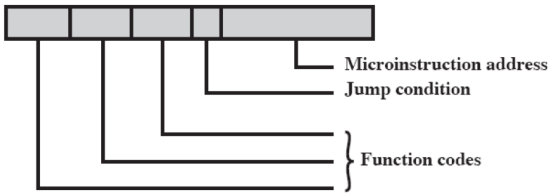
Abbildung 3.1: Ablaufsteuerung

[2]Ein Hinweis auf vertikale Mikroprogrammierung

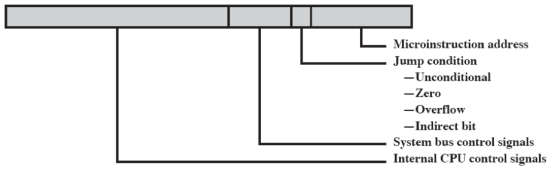
Abbildung 3.2: Mikroprogrammwerk



a: Mikroprogrammierte Architektur mit Sprüngen



b: Vertikales Mikroprogrammwort mit Sprungadresse



c: Horizontales Mikroprogrammwort mit Sprungadresse

Mit einem Rechen- und einem Steuerwerk gelingt es Mikroprogramme umzusetzen (siehe Abbildung 3.2a). Mit einem Algorithmus im Kopf überlegt man sich, welche Steuerleitungen in welchem Takt aktiviert werden müssen, und setzt diese dann auf HI oder logisch 1, alle anderen auf LO oder logisch 0. Sprünge sind in Mikroprogrammen ebenfalls erlaubt. In Abbildung 3.2a betrachte man die Leitungen 16 mit 21, un erkennt, dass die gegebene Architektur Sprünge unterstützt.
Man unterscheidet zwischen horizontaler und vertikaler Mikroprogrammierung:

Horizontale Mikroprogrammierung (siehe Abb. 3.2c)

Horizontale Mikroprogrammierung beschreibt die eben erwähnte Methodik. Für jede Kontrollleitung in unserem Mikroprogrammwerk (Abbildung 3.2a) gibt es ein Bit im Kontrollwort (*Internal CPU control signals* und *System bus control signals*[3] in Abbildung 3.2c). Jede Operation liegt nun im Speicher. Um Sprünge zu erlauben, fügen wir zu jedem Kontrollwort ein Adressfeld und eine Sprungbedingung hinzu, die das nächste Kontrollwort addresiert, wenn die Bedingung wahr ist. Eine solche Mikroinstruktion wird wie folgt ausgeführt:

1. Um die Mikroinstruktion auszuführen, aktiviere alle Signalleitungen, deren Kontrollbit auf 1 gesetzt ist und deaktiviere alle, deren Kontrollbit auf 0 gesetzt ist.

2. Wenn die Bedingung indiziert durch die Bedingungsbits falsch ist, führe sequentiell fort.

3. Wenn die Bedingung indiziert durch die Bedingungsbits wahr ist, führe mit der Adresse spezifiziert durch die Adressbits fort.

Vertikale Mikroprogrammierung (siehe Abb. 3.2b)

Anstelle direkter Kontrollsignale ist es auch möglich einen kodierten Funktionskode zu verwenden, über den die Mikroinstruktion mit Dekodern erzeugt wird. Das resultierende Mikrokontrollwort (siehe Abbildung 3.2b) ist damit kürzer als bei der horizontalen Mikroprogrammierung, was den Hauptvorteil dieses Verfahrens bildet. Das geht aber nur auf Kosten zusätzlicher Dekodierlogik.

[3]Bündel an Leitungen, das alle Komponenten miteinander verbindet

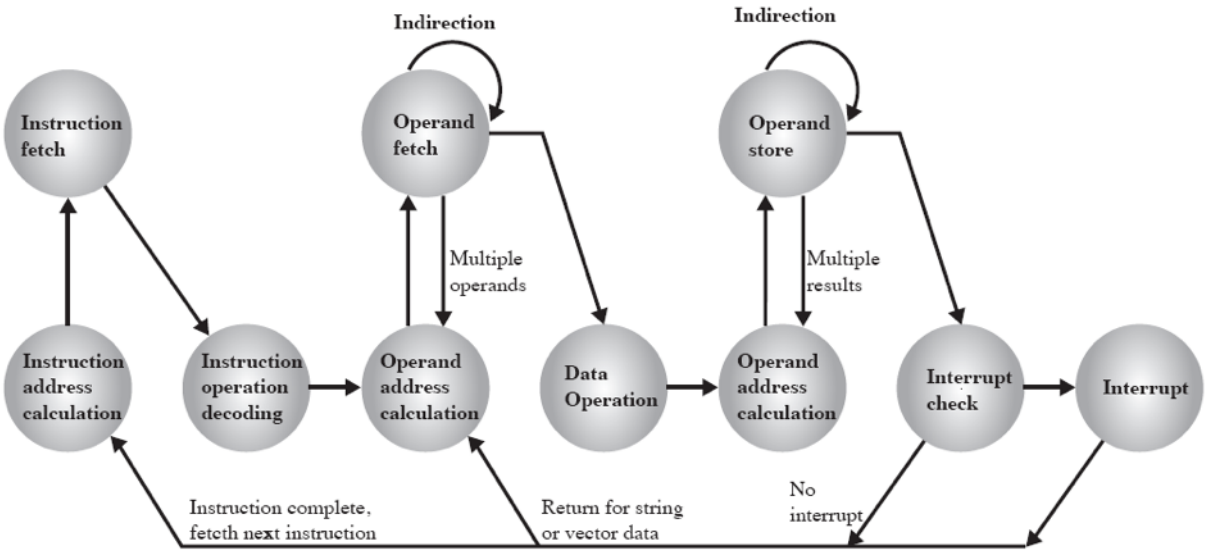


Abbildung 3.4: Modifizierter Instruktionszyklus mit Behandlung von Interrupts und Berücksichtigung indirekter Adressierung sowie Vektordaten

Organisation Leitwerksspeicher

Die Frage die sich jetzt noch stellt ist, wie man den Befehlszyklus auf einem mikroprogrammierten Leitwerk darstellt. Man bildet deswegen jeden Befehlszyklusschritt auf eine feste Folge von Mikroinstruktionen ab, die den vorher gezeigten Ablauf zeitlich nacheinander umsetzen (siehe rechts). So werden zum Beispiel die Aufgaben „Befehl holen“, „indirekte Adressierung“ oder auch die „Interrupt Routine“ selbst als **nicht veränderbare** Mikroprogramme umgesetzt. In der Abbildung rechts beginnt danach dann der veränderliche, **variable** Teil. Somit ist es auch klar, wie neue Mikroprogramme hinzugefügt werden können.

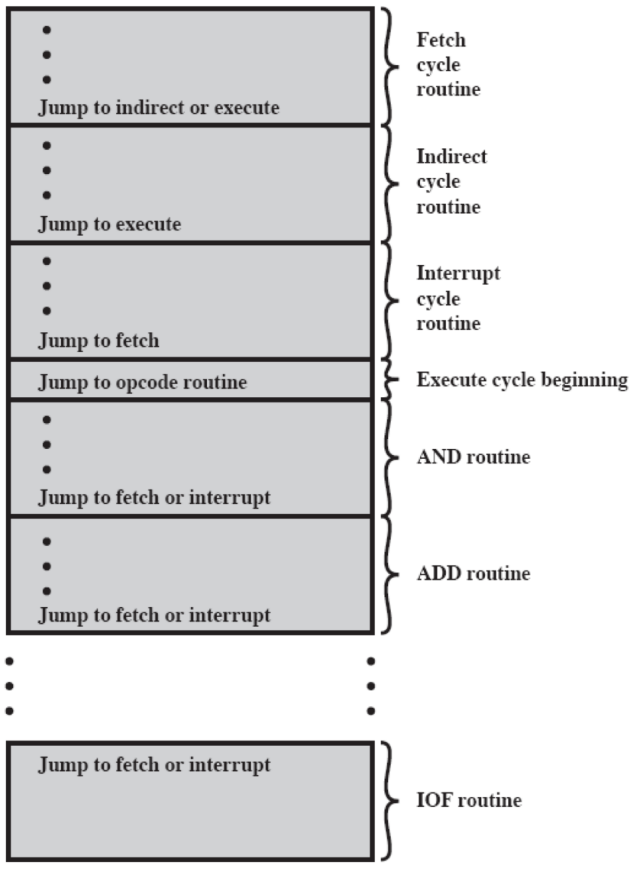


Abbildung 3.3: Leitwerksspeicher

3.2 Leitwerk: Behandlungen von Unterbrechungen

Fast alle Prozessoren bieten Mechanismen mit denen andere Module (E/A, Speicher) den „normalen“ Prozessablauf der CPU unterbrechen können. Dies ist durchaus sinnvoll, da man meist nicht auf

Ereignisse mit dem Prozess warten möchte, wenn man gleichzeitig sinnvoll weiterarbeiten könnte (bspw. bei langen E/A-Prozessen). Wenn die Unterbrechung dann eintrifft, wird eine Unterbrechungsroutine ausgeführt. Hier kommt das Konzept der Mikroprogrammierung wieder zum Vorschein (siehe Abbildung 3.3). Man sieht: Unterbrechungsroutinen sind auch nur „normale“ Unterprogramme.

Traps und Interrupts

Trap — synchron, vorhersagbar, reproduzierbar
Ein in die Falle gelaufenes Programm, das unverändert wiederholt und jedesmal mit den selben Eingabedaten versorgt auf ein und dem selben Prozessor zur Ausführung gebracht wird, wird auch immer wieder an der selben Stelle in die selbe Falle tappen.
Beispiele:

- unbekannter Befehl
- falsche Adressierungsart oder Rechenoperation
- Adressraumverletzung

Interrupt — asynchron, unvorhersagbar, nicht reproduzierbar
Ein „externer Prozess“ (z.B. ein Gerät) signalisiert einen Interrupt unabhängig vom Arbeitszustand des gegenwärtig sich in Ausführung befindlichen Programmes. Ob und ggf. an welcher Stelle die Ausführung des betreffenden Programmes unterbrochen wird, ist **nicht** vorhersagbar.
Beispiele:

- Signalisierung „externer“ Ereignisse (z.B. Drucker ist fertig o.ä.)
- Beendigung einer E/A-Operation

Behandlung von Interrupts

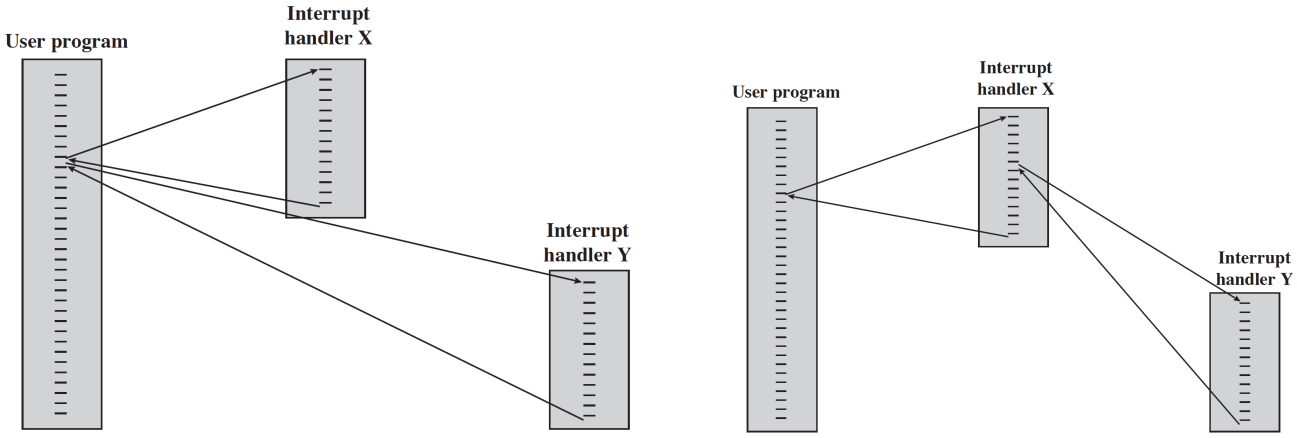
Die Befehlskette hier ist eigentlich die selbe wie bei Unterprogrammaufrufen in Assembler. Der Unterschied: Die folgenden Schritte werden von der CPU ausgeführt.

1. Der Interrupt trifft ein
2. Aktuellen Befehlszähler auf den Stack speichern
3. Befehlszähler mit Adresse des Interrupt-Handlers[4] laden
4. Speichern der Inhalte der CPU-Register auf dem Stack
5. Interrupt wie „gewöhnliches“ Unterprogramm abarbeiten
6. Am Ende der Interrupt-Routine die Inhalte der CPU-Register wiederherstellen, durch zurückholen vom Stack in umgekehrter Reihenfolge, Befehlszähler wiederherstellen → Zustand vor Eintreffen des Interrupts ist wiederhergestellt
7. Befehlszähler erhöhen und weiterarbeiten

Behandlung von mehreren Interrupts

Man unterscheidet grundsätzlich zwei verschiedene Ansätze. Sequentielle Abarbeitung der Interrupts oder die Möglichkeit der Hierarchischen Abstufung bei Interrupts.

[4] Diese kann fest vorgegeben sein, oder mit dem Interrupt eintreffen



(a) Sequentielles Verhalten bei mehreren Interrupts (b) Hierarchisches Verhalten bei mehreren Interrupts

Abbildung 3.5: Verschiedene Möglichkeiten für das Verhalten einer CPU mit mehreren Interrupts

Sequentielle Abarbeitung (siehe Abbildung 3.5a)

In diesem Fall ist die Interrupt-Routine eine unteilbare Prozedur in der Interrupts kurzfristig abgeschaltet werden, das heißt ein Interrupt kann hier zwar eintreffen, wird aber dann von der CPU ignoriert. Er bleibt solange weiter aktiv, bis die CPU den ursprünglichen Interrupt abgearbeitet hat, Interrupts wieder aktiviert und überprüft, ob weitere Interrupts eingetroffen sind. Der Ansatz wird dargestellt in Abbildung 3.5a. Nachteil dieses Verfahrens ist, dass es die relative Priorität und zeitkritische Verfahren nicht berücksichtigt und damit Datenverluste riskiert.

Hierarchische Abarbeitung (siehe Abbildung 3.5b)

Der zweite Ansatz definiert Prioritäten für Interrupts und erlaubt einem Interrupt von höherer Priorität einen niederpriorisierten Interrupt-Handler selbst wieder zu unterbrechen. Dieser Ansatz ist in Abbildung 3.5b dargestellt.

Systemcalls und CPU-Exceptions

Systemcalls sind auch Unterbrechungen im „normalen“ Programmfluss, da hier das Betriebssystem aufgerufen wird. Sie sind ähnlich wie Traps vorhersagbar, jedoch ist die Realisierungsebene eine andere, da hier meist andere Berechtigungen gefordert sind. Systemcalls sind somit **synchrone** Unterbrechungen, welche durch bestimmte Befehle **absichtlich** ausgelöst werden. Sie ermöglichen eine Ausführung von Betriebssystemprogrammen als **nicht-priviligierter** Nutzer, indem sie den Modus für diesen Befehl ändern (*user mode* $\mapsto$ *system mode*) — Beispiele hierfür sind der Zugriff auf externe Geräte, Ein- und Ausgaben, ...

CPU-Exceptions sind **synchrone** Programmunterbrechungen **innerhalb** der CPU, ähnlich den Traps. Die Behandlung wird durch das Betriebssystem festgelegt (bspw. Division durch 0, Seitenfehler, ...) — meistens ist das dann die Terminierung des Prozesses.

3.3 Befehlssatzarchitekturen

Wir unterscheiden hier grundsätzlich zwischen vier Befehlssatzarchitekturen: Register-Register, Register-Memory, Akkumulator und Stackarchitekturen.

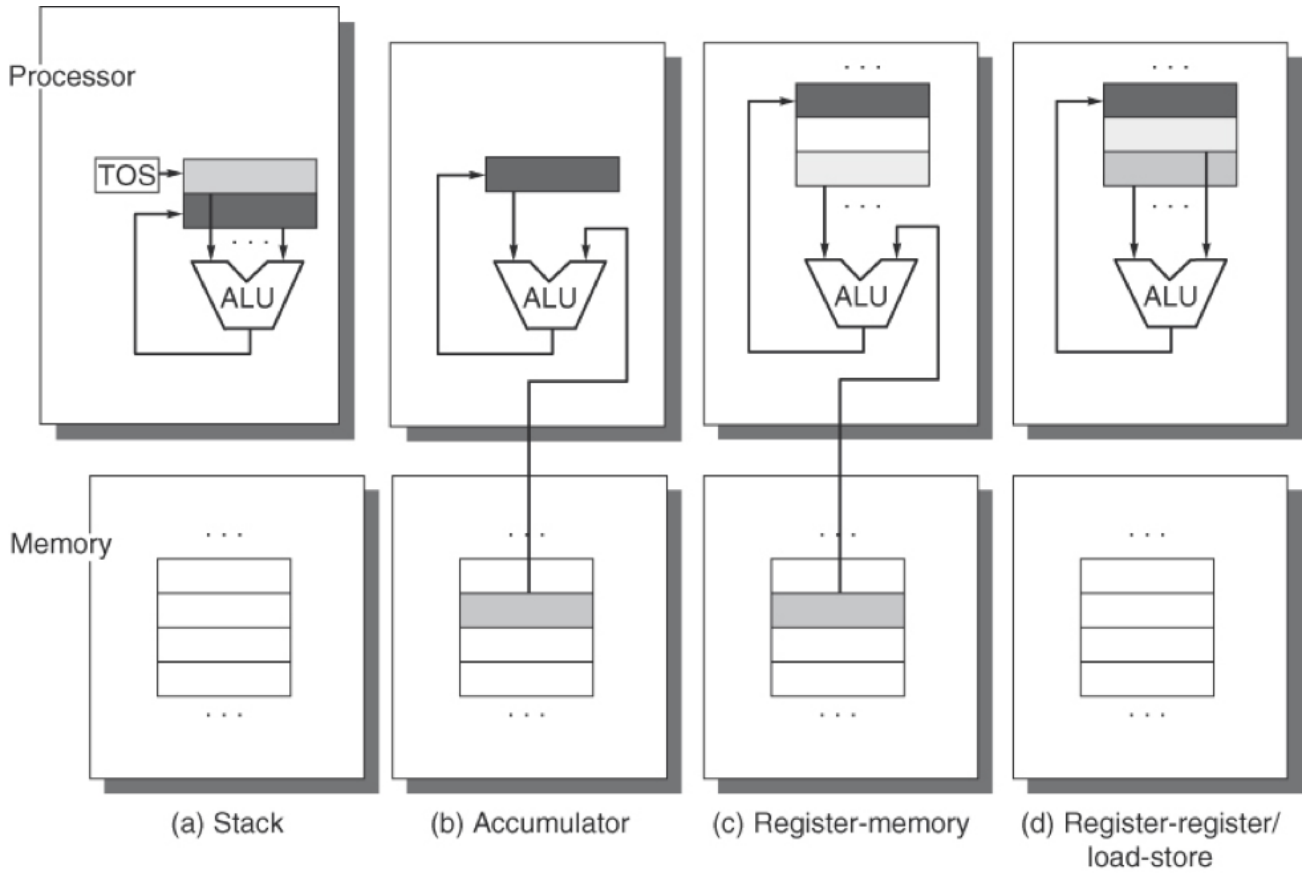


Abbildung 3.6: **Speicherplätze der Operanden bei vier verschiedenen Befehlssatzarchitekturklassen.** Die Pfeile geben an, ob ein Operand eine Eingabe oder das Ergebnis einer ALU-Operation ist. Hellere Grautöne stehen für Eingaben, dunklere für Ergebnisse. In (a) zeigt ein „Beginn des Stacks"-Register (TOS) auf den „obersten" Eingabeoperanden, welcher mit dem nachfolgenden Operanden verknüpft wird. Der erste Operand wird vom Stack „gelöscht" und das Ergebnis nimmt den Platz des zweiten Operanden ein; der TOS zeigt danach auf das Ergebnis. Alle Operanden sind somit impliziert und müssen nicht mit angegeben werden. In (b) stellt der Akkumulator sowohl einen impliziten Eingabeoperanden als auch ein Ergebnis dar. Es gibt damit nur **einen** Operanden pro Operator, meistens auch **keine** Register. In (c) ist ein Eingabeoperand ein Register, der andere hingegen kann auch im Speicher stehen. Das Ergebnis wird in einem Register gespeichert. Ein Stack ist hier meist nicht ausgeschlossen. Dahingegen sind sowohl bei (a) als auch bei (d) alle Operanden Register und können nur mit bestimmten Befehlen übertragen werden: `pop` und `push` für (a) und `load`/`store` für (d)

Beispielprogramme für die Architekturen in Abbildung 3.6

Im Folgenden wird die ALU-Instruktion $C \leftarrow A + B$ beispielhaft für die vier Instruktionen vorgestellt in Abbildung 3.6 implementiert.

Stack

```
push A
push B
add
pop C
```

Akkumulator

```
load A
add B
store C
```

Register-Memory

```
load R1, A
add R1, B
move C, R1
```

Register-Register

```
load R1, A
load R2, B
add R3, R1, R2
store R3, C
```

3.4 Endianness

Das untenstehende Beispiel bezieht sich auf die willkürlich gewählte Zahl

$$240707093511855_{\text{dec}} = \Big(\underbrace{\text{DA}}_{\text{Byte 6}}\ \underbrace{\text{EB}}_{\text{Byte 5}}\ \underbrace{\text{FC}}_{\text{Byte 4}}\ \underbrace{\text{CD}}_{\text{Byte 3}}\ \underbrace{\text{BE}}_{\text{Byte 2}}\ \underbrace{\text{AF}}_{\text{Byte 1}}\Big)_{\text{hex}}$$

Definition(*Big Endian*)

Das **Most**-Significant-Byte (**M**SB) steht an der **niedrigsten** Adresse. → so wie wir Zahlen lesen.

Definition(*Little Endian*)

Das **Least**-Significant-Byte (**L**SB) steht an der **niedrigsten** Adresse. Beispiel: Deutsche Datumsschreibweise (DD.MM.JJJJ)

Typ	relative Byteadressen					
	+0	+1	+2	+3	+4	+5
Big Endian	DA	EB	FC	CD	BE	AF
Little Endian	AF	BE	CD	FC	EB	DA

Interessantes zur Ethymologie der Begriffe

Im satirischen Roman Gullivers Reisen von Jonathan Swift geht es um zwei verfeindete Gruppen der Bewohner des Landes Liliput. Der Streit entbrach, weil man sich nicht darauf einigen konnte, wie man die Eier richtig aufschlägt. Die Gruppe der „Big Ender“ schlagen ihre Eier am großen Ende auf, während die „Little Ender“ sie am spitzen, „kleinen“ Ende aufschlagen. Es brauchte schließlich Gulliver um den Streit zu schlichten. Danny Cohen stellte eine Analogie zur Byteordnung in seinem 1980 veröffentlichten Artikel „On Holy Wars and a Plea for Peace“ her und gab den beiden Varianten ihre Namen, big endian und little endian.

3.5 Alignment

Definition(*Alignment*)

Ein Objekt bestehend aus s Bytes, im Speicher abgelegt unter der Adresse $\texttt{addr}$ mit der Datenbusbreite von w_{db} ist exakt ausgerichtet (auch: **aligned**) gdw. $\texttt{addr} \mod \min\{s, w_{\text{db}}\} = 0$ gilt.

Warum braucht man Alignment?

Der Speicherzugriff auf nicht ausgerichtete Anordnungen verkomplizieren ihn ungemein und kann sogar unter Umständen zu Fehlern führen. Speicher an sich ist häufig in Bänken organisiert, so dass der Zugriff entlang den Zeilen einer Bank in einem Speicherzugriff erledigt wird. Nicht ausgerichtete Objekte erfordern nun mehrfache Speicherzugriffe.

Problem: Wort wird aufgeteilt und ist dummerweise in verschiedenen Speicherseiten. Es kann während das eine Teilwort gelesen wird beim zweiten zu einem TLB-Miss oder page fault kommen, oder da die Operation nicht mehr atomar ist zu einer Lese-Schreibe-Ändere-Wettlaufsituation kommen.

Typ	Vorteile	Nachteile
Register-Register	• Befehle fester Länge sind einfach zu kodieren • Einfaches Kodeerzeugungsschema • Ungefähr gleiche Ausführungsdauer bei unterschiedlichen Befehlen	• Mehr Befehle als bei Architekturen, die einen direkten Speicherzugriff in den Befehlen ermöglichen • Zusammen mit der **geringeren Dichte** der Befehle führt dies zu längeren Programmen.
Register-Memory	• Auf Daten aus dem Speicher kann direkt zugegriffen werden • Ein einfach zu kodierendes Befehlsformat erreicht eine gute Dichte	• Operanden sind nicht gleichwertig, da bspw. bei einer 2-Operanden-Maschine einer der beiden Quelloperanden bei einer binären Operation überschrieben wird. • Die nötige Taktzahl eines Befehls variiert abhängig vom Speicherort des Operanden
Memory-Memory	• Am kompaktesten; Es werden keine Register für Zwischenergebnisse „verschwendet“	• Starke Unterschiede bzgl. der Länge der Befehle, vor allem bei Befehlen mit 3 Operanden • Folglich starke Unterschiede bei der Abarbeitung der Befehle → schlecht für Pipelining (siehe Kapitel 4) • Speicherzugriffe erzeugen einen „Speicherflaschenhals“

Tabelle 3.1: **Vergleich der Vor- und Nachteile der verschiedenen Befehlsatzarchitekturen**: Speicher-Speicher-Adressierung wird in der Praxis nicht mehr eingesetzt

3.6 Speicherwerk: Speicherhierarchie

Das Speichersystem eines Rechners ist selbstverständlich entscheidend für dessen Leistungsfähigkeit und Kosten. In einem idealen System besitzt dieses dann ausreichend Kapazität, sowie eine Zugriffszeit, welche mit den Verarbeitungsgeschwindigkeiten des Prozessors mithalten kann. Das ist aus rein wirtschaftlichen (Kosten für Zugriffszeiten $\nearrow\nearrow$) und technischen Gründen aber nicht möglich.
Man schafft sich deswegen über eine mehrstufige Speicherhierarchie Abhilfe. Hierbei gilt: Mit Voranschreiten jeder Stufe verringert sich die Kapazität und es erhöhen sich die Geschwindigkeit und die Kosten pro Byte.

Definition(*Inklusionsbedingung*)

> Jeder Speicher einer Hierarchiestufe enthält einen Ausschnitt des Speichers der nächst höheren Hierarchiestufe.

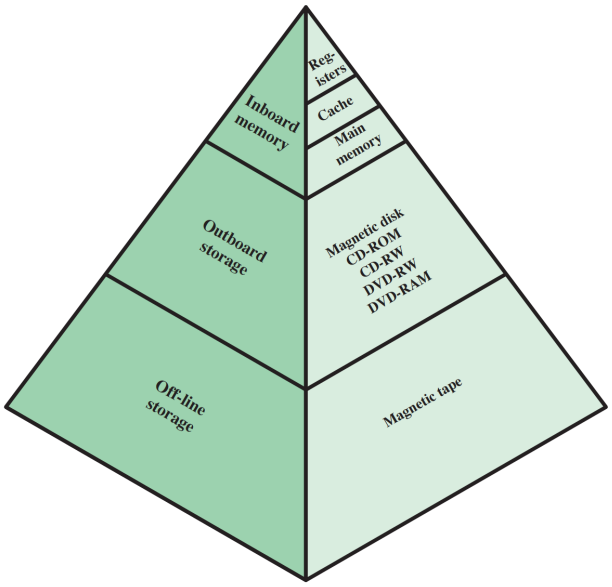


Abbildung 3.7: Speicherhierarchie

Eine typische Speicherhierarchie ist in Abbildung 3.7 gegeben. Wenn man an der Pyramide absteigt, so ...

- ... sinken die Kosten pro Bit
- ... steigt die Kapazität
- ... **sinkt** die Zugriffs**geschwindigkeit**, damit *steigt* die Zugriffs*zeit*
- ... sinkt die Speicherzugriffsfrequenz des Prozessors

Charakteristika eines Speichers

- **Ort**
  Prozessor, Primärspeicher (innerhalb der CPU), Sekundärspeicher (außerhalb)
- **Kapazität**
  Wortgröße, Anzahl an Worten
- **Speicherübertragung**
  ... per Block, ... per Wort
- **Speicherzugriffsmethode**
  Sequenziell, Direkt, Zufällig, Assoziativ
- **Performanz**
  Zugriffszeit, Transferrate, Zyklusdauer
- **Typ des Speichers**
  Halbleiter, Magnetisch, Optisch, Magneto-optical
- **Physikalische Charakteristiken**
  Flüchtig?, Wiederbeschreibbar?
- **Organisation**

3.7 Speicherwerk: Grundlegende Cache-Techniken

Definition(*Cache*)

Ein schneller Pufferspeicher **zur Verbergung der Latenzzeit für Zugriffe auf den im Vergleich zur Zykluszeit der CPU langsameren Arbeitsspeicher**, der zwischen Registern und Hauptspeicher liegt. Die Puffergröße ist ein ganzzahliges Vielfaches der Größe einer Zwischenspeicherzeile (*cache line*) oder eines Zwischenspeicherblocks (*cache block*). Damit ist der Aufbau eines Caches wie im Hauptspeicher in Blöcken unterteilt.

Zeitliche und Räumliche Lokalität

Definition(*Räumliche Lokalität*)

Nach einem Zugriff auf eine Speicheradresse wird mit hoher Wahrscheinlichkeit ein nächster Zugriff in naher Zukunft auf eine benachbarte Speicheradresse stattfinden.
Geschaffen durch: Sequentialität, Felder
Cache nutzt dies mit den Cacheblöcken und -zeilen aus.

Definition(*Zeitliche Lokalität*)

Ein Zugriff auf eine Speicheradresse wird mit hoher Wahrscheinlichkeit in naher Zukunft nochmal stattfinden.
Geschaffen durch: Schleifen
Cache nutzt dies durch sich selbst aus, da er ja dafür geschaffen wurde.

Zugriff auf Daten

Ohne Cache wurden die Daten jedes Mal aus dem Hauptspeicher geladen, mit Cache überprüft das Steuerwerk allerdings zuerst, ob das Datum im Cachespeicher vorhanden ist. Es gibt nun zwei Möglichkeiten: **HIT** oder **MISS**.

1. Es kommt zum *cache hit*: Das Datum wird aus dem Cache gelesen; man spart Zeit
2. Es kommt zum *cache miss*: Der Cache liest eine *cache line* – in der die Adresse liegt – aus dem Hauptspeicher ein und speichert sie im Cache. Sollte im Cache kein Platz mehr, der Organisationsform geschuldet, sein, so wird ein Datum gemäß einer a priori bestimmten Ersetzungsstrategie ersetzt. Wurde ein in dieser Zelle liegendes Speicherwort verändert, so muss bei manchen Aktualisierungsstrategie das geänderte Datum zuerst rückgeschrieben werden.

Organisationsformen

Platzierungsproblem In welchem Cacheblock wird der Hauptspeicherblock abgelegt?

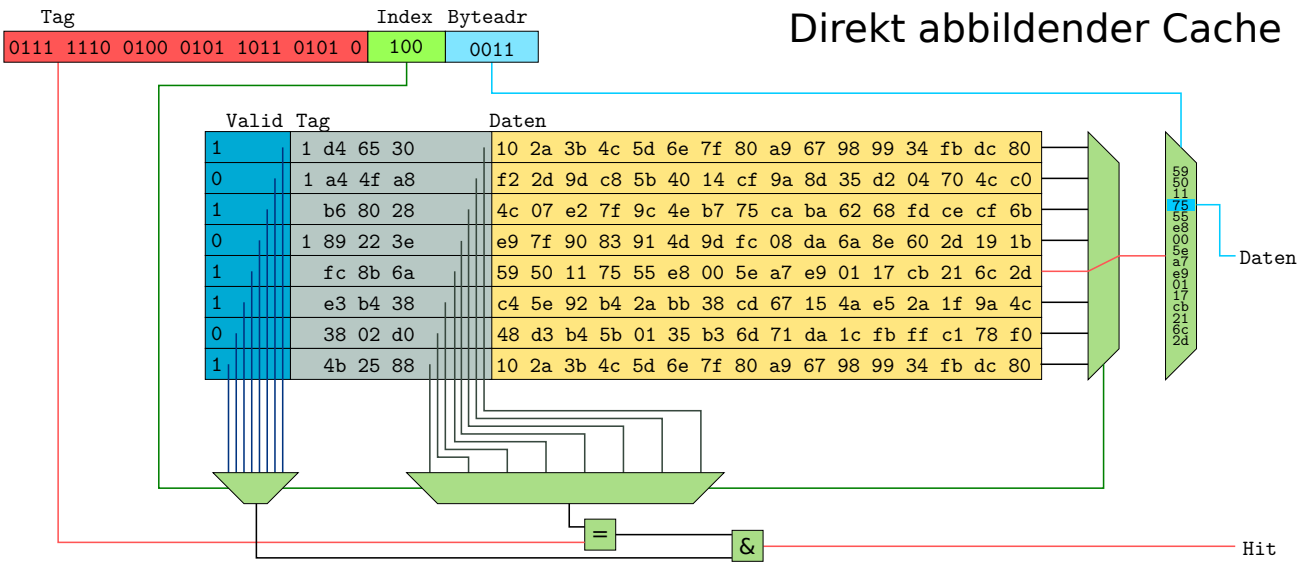
Identifikationsproblem Wie finde ich ein gewünschtes Datum resp. einen Block im Cache wieder auf?

Direktabbildender Cache

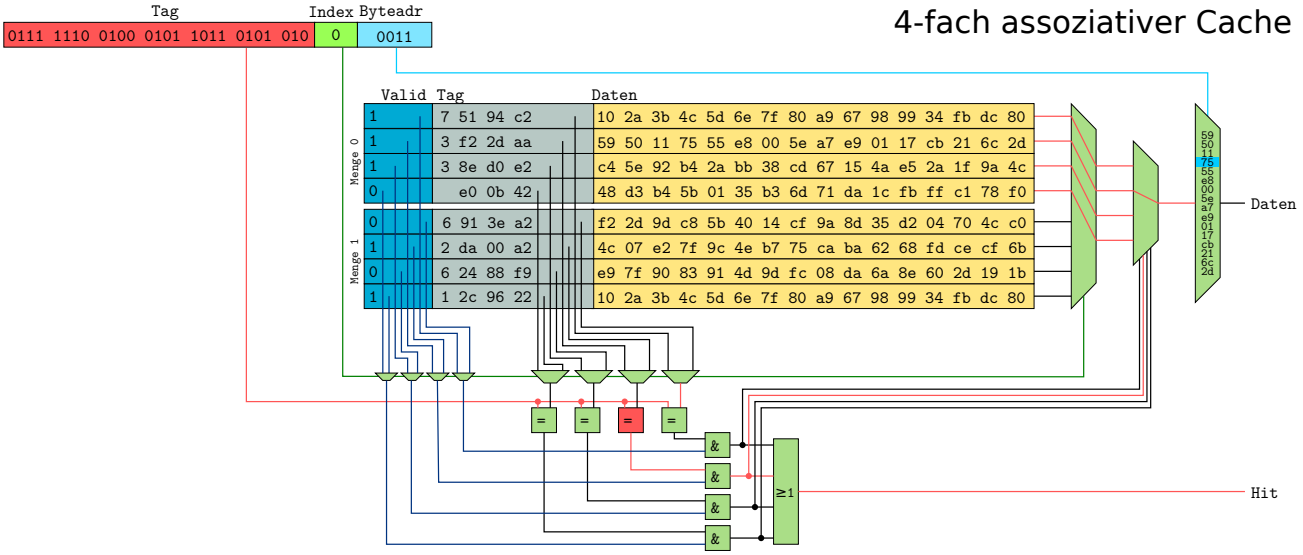
Ein Block m von insgesamt N Cacheblöcken wird einer jeden Adresse B über eine feststehende Abbildung $f : B \times N \mapsto m$ zugewiesen. Beispiel: $m = f(B, N) := B \mod N$

Vollassoziativer Cache

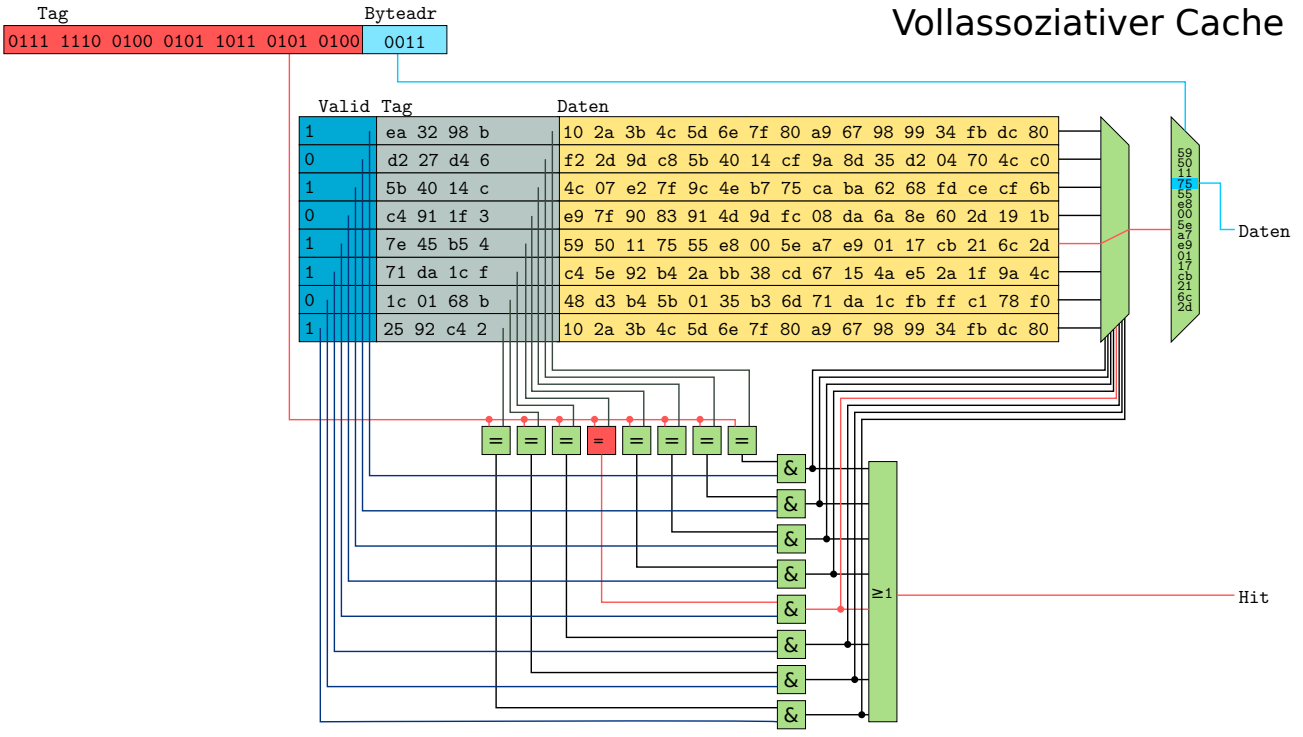
Jede Adresse eines Hauptspeicherblocks kann in jedem beliebigen Cache-Block abgeblidet werden.



(a) Direktabbildender Cache



(b) 4-fach assoziativer Cache



(c) Vollassoziativer Cache

Abbildung 3.8: Drei typische Organisationsformen für einen Cache

n-fach assoziativer Cache

Der Cache wird in s Mengen mit jeweils n Blöcken unterteilt; es gilt: $s = \frac{N}{n}$. Der Hauptspeicherblock wird nach der Methode der direkten Abbildung in die Cacheblockmenge, innerhalb der Mende dann nach der vollassoziativen Methode plaziert.

Fehlende Injektivität — Tag-Feld

Die Abbildung Hauptspeicherblock $\mapsto$ Cacheblock ist nur surjektiv, aber nicht injektiv. Damit fehlt die Umkehrbarkeit der Abbildung und die Lösung zu unserem Indentifikationsproblem. Wir führen deswegen zusätzliche Daten zum Speichern für den Cache ein: das **Tag-Feld**.
Das Tag-Feld ist ein Teil der Hauptspeicheradresse des Wortes, welches im Cache gespeichert für die **eindeutige** und vollständige Identifizierung des Wortes vorhanden ist. Ein Teil der Adresse ergibt sich aus der Adresse des Cache-Blocks (zumindest bei dem n-fach assoziativen Cache). Als Kennung der Tag-Information wird für gewöhnlich der restliche Teil der Hauptspeicheradresse verwendet.

Zusätzliche Bits

valid bit — Dieses Bit gibt an, ob die *cache line* mit gültigen Daten bestückt ist (1) oder nicht (0).

dirty bit — Dieses Bit gibt an, ob die zugehörige *cache line* nach dem Lesen bereits geändert wurde („dreckig“) und zurückgeschrieben werden muss.

Wirtschaftlichkeit

Wie man der Abbildung 3.8 entnehmen kann, brauchen die unterschiedlichen Organisationsformen eine unterschiedliche Anzahl an Komparatoren. So braucht ein n-fach assoziativer Cache n Komparatoren, wohingegen ein direkt abbildender Cache nur einen Komparator braucht, da die *cache line* eindeutig feststeht.
Dabei sinkt jedoch die Wirksamkeit des Caches, da mehrere Adressen auf den gleichen Cacheblock abbilden $\rightarrow$ viele Konfliktsituationen und häufiges Umladen erforderlich.
Man verwendet heute deswegen meistens Caches vom Grad 2,4,8 oder 16.

Aktualisierungsstrategie

Durchschreiben (*write through*)

Bei dieser Aktualisierungsstrategie werden sämtliche Änderungen umgehend an den übergeordneten Speicher weitergegeben. Somit wird sichergestellt, dass der Hauptspeicher immer gültige Daten enthält. Die Konsistenz ist damit zwar gegeben, jedoch kommt es zu einer hohen Belastung des Prozessor-/Speicherbusses und es könnte einen Flaschenhals hervorrufen. $\rightarrow$ Primärcaches arbeiten nach diesem Prinzip

Zurückschreiben (*write back*)

Bei dieser Aktualisierungsstrategie sind sämtliche Änderungen erstmal *cache-lokal*. Bei diesen Änderungen wird ein **dirty bit** oder **use bit** gesetzt. Bei einer Verdrängung wird anhand des **dirty bits** entschieden, ob ein Rückschreiben erforderlich ist.

Ersetzungsstrategie

LSF – Least Frequently Used

Die Zeile, die am wenigsten häufig benutzt wurde, wird ausgetauscht

LRU – Least Recently Used

Die Zeile, die am längsten nicht verwendet wurde („... least recently used ...“), wird ausgetauscht

FIFO – First-In First-Out

Die Zeile, die am **längsten** bereits gespeichert ist, wird ausgetauscht

Random – Zufällig

Eine zufällige Zeile wird ausgetauscht

Klassifikation von Fehlzugriffen — Die drei Cs

C wie Compulsory

Der erste Zugriff auf einen Block trifft nicht den Cache, Block muss erstmals geladen werden (Kaltstartmiss oder First-Reference-Miss)

C wie Capacity

Der Cache hat nicht genug Platz, um alle Blöcke der aktuell zu bearbeitenden Befehlsfolge aufzunehmen

C wie Conflict

In nicht vollassoziativen Caches werden Blöcke aufgrund von Adresskonflikten überschrieben und ggf. später zurückgeladen.
Diese Art von Miss liegt dann vor, wenn folgende Frage mit JA beantwortet werden kann: *Wäre der Cache vollassoziativ, träte dieser Miss dann immer noch auf?*

3.8 Aufbau des Arbeitsspeichers

Definition(*RAM – Random Access Memory*)

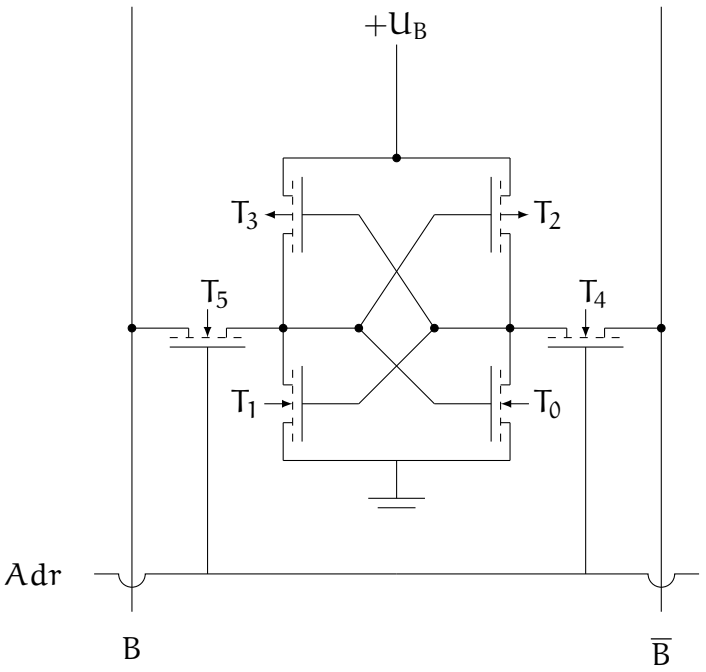
> Als **Speicher mit wahlfreiem Zugriff** (*Random Access Memory*) wird ein Datenspeicher – in der gängigsten Form ein Halbleiterspeicher – bezeichnet, der besonders bei Computern als Arbeitsspeicher Verwendung findet.

SRAM – Static Random Access Memory

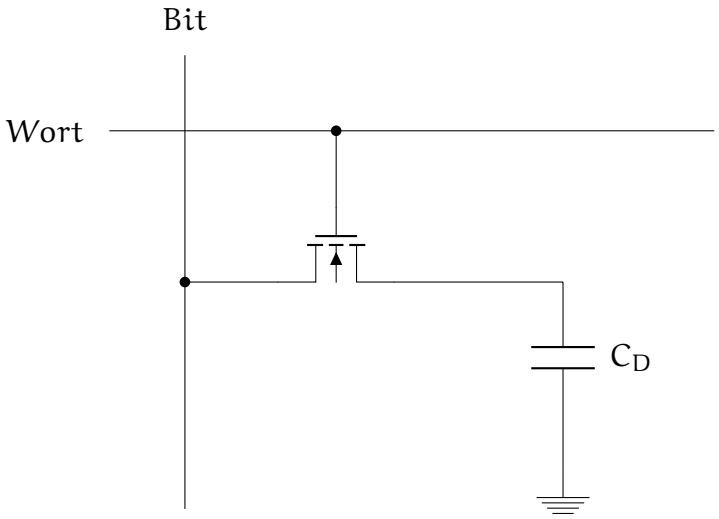
Definition(*Statischer Speicher mit wahlfreiem Zugriff*)

> Ein **statischer Speicher mit wahlfreiem Zugriff** (*Static RAM*, *SRAM*) ist ein elektronischer Speicherbaustein. Zusammen mit dem dynamischen Speicher mit wahlfreiem Zugriff bildet es die Gruppe der flüchtigen Speicher. Anders als dynamischer Speicher mit wahlfreiem Zugriff muss der Speicherinhalt jedoch nicht periodisch erneuert werden, als dass durch die Verwendung von bspw. Flip-Flops Speichererhaltung bis zur Stromabschaltung garantiert werden kann. In Abbildung 3.9a ist ein beispielhaftes Schaubild einer Zelle, bestehend aus 6 CMOS-Transistoren gezeichnet.

SRAM-Zellen benötigen **mindestens vier**, haben *meistens* aber zwischen ***sechs und acht*** Transistoren. Sie sind um ungefähr den Faktor 8 schneller, haben dafür aber auch eine um denselben Faktor geringere Kapazität.
Die **Anwendung** findet sich häufig in Caches und zum Teil in Hochleistungsrechnern.



(a) Schaubild einer 6T(*ransistor*)-SRAM Zelle



(b) Schaubild einer 1T(*ransistor*)-DRAM Zelle

Abbildung 3.9: Typischer Aufbau verschiedener RAM-Zellen

DRAM – Dynamic Random Access Memory

Definition(*Dynamischer Speicher mit wahlfreiem Zugriff*)

Ein **dynamischer Speicher mit wahlfreiem Zugriff** (*Dynamic RAM*, *DRAM*) bezeichnet einen elektronischen Speicherbaustein . Es gehört zusammen mit dem statischen Speicher mit wahlfreiem Zugriff zu der Gruppe der flüchtigen Speicher. Anders als der statische Speicher besteht der dynamische Speicher nur aus einem Kondensator, der entweder ent- oder geladen ist. Über einen Schalttransistor wird der zugänglich und entweder ausgelesen oder mit neuem Inhalt beschrieben. Charakteristisch für diese Speicherart ist das zerstörende Lesen und das zyklische Wiederbeschreiben der Dateninhalte aufgrund von auftretenden Leckströmen. In Abbildung 3.9b ist ein beispielhaftes Schaubild einer DRAM-Zelle gezeichnet.

DRAM-Zellen benötigen einen Kondensator und einen zusätzlichen Transistor, sind also im Gegensatz zu den SRAM-Speicherzellen sehr kompakt. Daher kann ein DRAM-Chip auch – bei gegebener, fester Chipgröße – mehr Bits speichern. Um mehr als ein bit zu adressieren, setzt man mehrere Bänke nebeneinander. So ist beispielsweise für 256 KiB Wörter eine 18-bit Adresse notwendig, die dann 8 256 K $\times$ 1-bit Chips präsentiert wird, welche eine 1-bit Ein- und Ausgabe bereitstellen.

Definition(*Speicherkontroller*)

Der **Speicherkontroller** (*memory controller*) sorgt für die Adressinterpretation, die Wortadressierung sowie die Auswahl einer oder mehrerer Byte-Blöcke. Er enthält Funktionen um dynamischen Speicher (*DRAM*) zu lesen, beschreiben und zu aktualisieren.

Speicherverschränkung

Definition(*Speicherbank*)

Eine **Speicherbank** (*memory bank*) besteht aus mehreren parallel angeordneten Speicherbausteinen und einem Speicherkontroller. Es ist ein eindeutig, unabhängig adressierbarer N bit breiter Bereich

eines Speichermoduls. Jede Bank verhält sich wie ein separates Speichermodul.

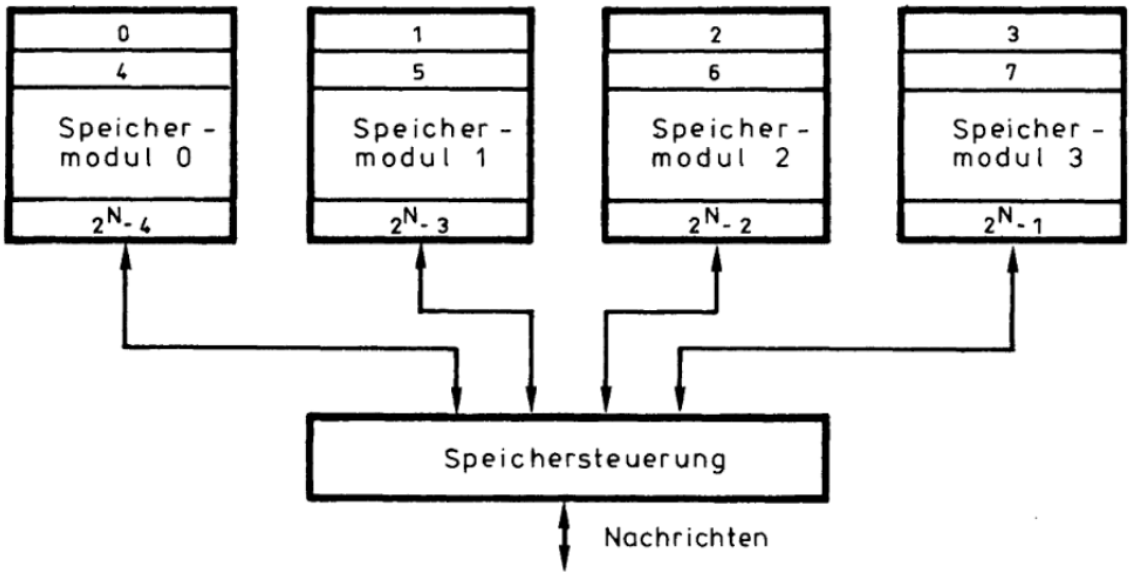


Abbildung 3.10: Prinzip der Speicherverschränkung dargestellt für 4 Speicherbänke

Definition(*Speicherverschränkung*)

Bei einer **Speicherverschränkung** (*interleaved memory*) wird der Speicher in gleich große, voneinander unabhängige Speicherbänke unterteilt, welche zeitlich verschränkt gelesen oder beschrieben werden können, da sie alle eigene Eingänge zum Speicherkontroller haben. Die Adressen werden nach folgendem Schema den einzelnen Modulen zugeteilt:
Sei 2^N die Größe des gesamten Speichers und k die Anzahl der Module, dann liegen im i-ten Modul (mit $0 \leqslant i \leqslant k-1$) alle Adressen m ($0 \leqslant m \leqslant 2^N - 1$) für die gilt: $m \mod k = i$.
Das Beispiel für den Fall $k = 4$ ist in Abbildung 3.10 dargestellt.
Mit der Speicherverschränkung ist es möglich die Wartezeiten auf Speicherzugriffe zu minimieren.

Prozessor-Speicherlücke

In den letzten Jahren hat sich die Latenzzeit von Speichern pro Jahr durch die Technologie nur um etwa zehn Prozent verringert. Dies führte zu erheblichen Differenzen zwischen der Prozessorleistung und der Speicherleistung.

Synchroner DRAM

Definition(*Synchroner dynamischer RAM*)

Als **synchronen dynamischen Speicher** (*SDRAW*, *synchronous DRAM*) bezeichnet man eine getaktete Art des DRAMs. Der Takt wird dabei durch den Systembus – gegebenenfalls auch durch einen separaten Speicherbus – vorgegeben. Die Taktung erfolgt über die Verwendung von Registern für Adresseingänge, Steuerinformationen sowie die Ein-/Ausgabedaten, indem Wertänderungen in den Registern nur mit den Taktflanken durchgeführt werden. Dabei entfällt die – bei asynchronen Verfahren notwendige – Komunikation.

Definition(*Stoßmodus*)

Als einen Stoßmodus (*burst mode*) bezeichnet man einen Übertragungsmodus zur Beschleunigung von Lest-oder Schreibvorgängen bei Speichereinheiten. Im Gegensatz zum Wortmodus (*word-at-a-time mode*) werden beim Stoßmodus größere Datenblöcke als ununterbrochenes Bündel kleinerer

Dateneinheiten übertragen, ohne dass gewisse Initialisierungen für jede dieser Untereinheiten neu erfolgen muss.

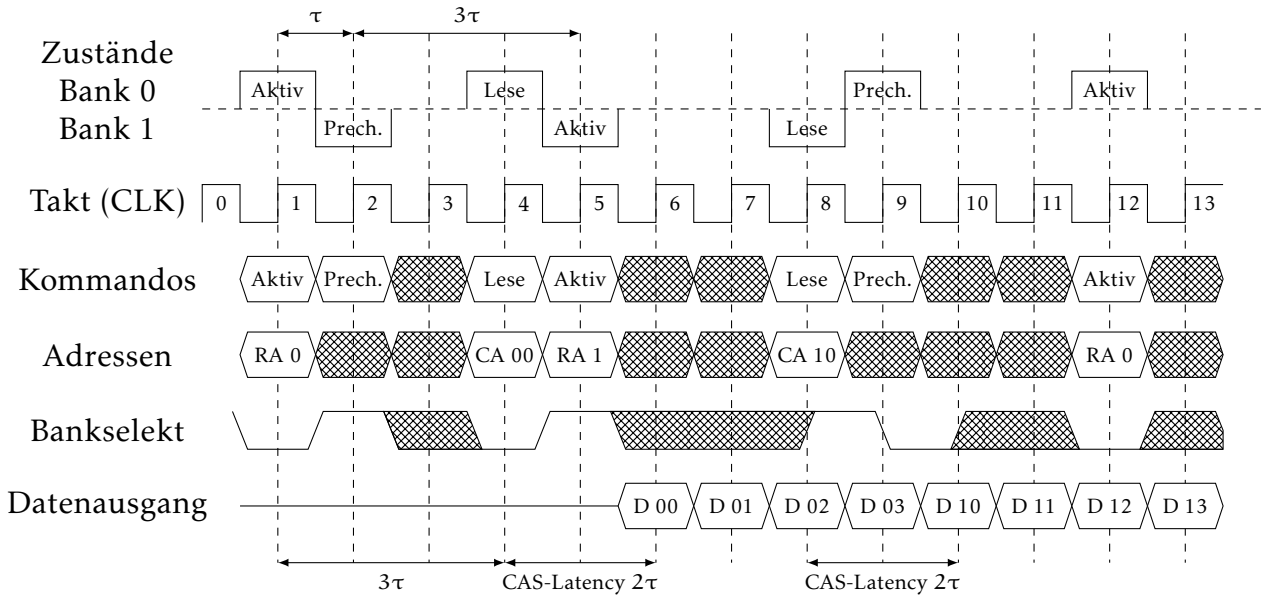


Abbildung 3.11: Beispiel für einen abwechselnden Burst-Zugriff auf verschiedene Speicherbänke

In Abbildung 3.11 wird ein Beispiel für den Betrieb eines SDRAMs dargestllt. Es wird zunächst aus Bank 0 und anschließend aus Bank 1 jeweils ein Burst der Länge 4 gelesen, so dass sich ein fortlaufender Datenstrom von 100 MByte pro Sekunde ergibt. Im gezeigten Beispiel ist eine Taktperiodenzahl, nach der bei READ die Daten am Ausgang stehen – die CAS-Latency – von 2 Taktzyklen eingestellt. Ein Precharge ist nur dann notwendig, wenn nach einer Stoßleseoperation, die neuen Daten in einer anderen Speicherzelle liegen.

DDR-RAM

Definition(*DDR-RAM*)

Als **Speicher mit wahlfreiem Zugriff und doppelter Datenrate** (*double data-rate DRAM*) bezeichnen wir einen Speicher, dessen Datenübertragung zweimal pro Taktzyklus – sowohl bei steigender als auch bei fallender Taktflanke – stattfindet.

3.9 Kopplung des Speichers mit der Ein- und Ausgabe

Adressierungsmodi der Peripherie

Definition(*Speicherabgebildete Ein-/Ausgabe*)

Ein Ein- und Ausgabeschema, wobei Teile des Adressraums den verschiedenen Ein-/Ausgabegeräten zugeordnet sind. Lese- und Schreiboperationen für diese Adressen werden als Befehle für das jeweilige Ein-/Ausgabegerät interpretiert.

Definition(*Isolierte Ein-/Ausgabe*)

Ein Ein- und Ausgabeschema, wobei Ein- und Ausgabegeräte eigenständige Adressräume verwenden. Es isoliert die Ein-/Ausgabegeräte von dem restlichen Speicher, bietet einen eigenständigen Bus und verwendet separate Instruktionen um Ein-/Ausgabegeräte anzusprechen.

Programmierte Ein-/Ausgabe

Definition(*Programmierte Ein-/Ausgabe*)

Die programmierte Ein-/Ausgabe (*programmed input-output*, *PIO*) ist ein Ein-/Ausgabeschema, welches den Prozessor Polling verwenden lässt, um den Status des Ein-/Ausgabegeräts abzufragen. Im Prinzp dasselbe Verfahren wie bei der unterbrechungsgesteuerten Ein-/Ausgabe mit dem Unterschied, dass nach Aufgeben des E/A-Befehls **aktiv** auf Fertigstellung **gewartet** werden muss.

Unterbrechungsgesteuerte Ein-/Ausgabe

Definition(*Unterbrechungsgesteuerte Ein-/Ausgabe*)

Die unterbrechungsgesteuerte Ein-/Ausgabe (*interrupt-driven input-output*, *IDIO*) ist ein Ein-/Ausgabeschema, das Interrupts verwendet, um dem Prozessor anzuzeigen, dass ein Ein-/Ausgabegerät Aufmerksamkeit benötigt. Der Prozessor beauftragt das E/A-Gerät mit einer Benachrichtigung, sobald die Daten verfügbar sind. Während der Abarbeitung des E/A-Befehls kann sich der Prozessor anderweitig beschäftigen. Sobald die Daten vorhanden sind schickt das E/A-Gerät dem Prozessor eine asynchrone Unterbrechung (*interrupt*). In der Zwischenzeit können andere Prozesse bearbeitet werden. **Die Daten können nun mittels PIO oder DMA übertragen werden**.

Direkter Speicherzugriff

Definition(*Direkter Speicherzugriff*)

Der direkte Speicherzugriff (*direct memory access*, *DMA*) ist ein Mechanismus, der einem Kontroller (DMA-Kontroller) die Möglichkeit gibt, Daten direkt zum oder vom Speicher zu übertragen, **ohne** dass der Prozessor daran beteiligt ist.
Der Prozessor muss den DMA-Kontroller anweisen, von wo nach wo Daten zu übertragen sind. Der eigentliche Transfer wird durch den DMA-Kontroller abgewickelt. Der Prozessor wird durch eine asynchrone Unterbrechung (*interrupt*) benachrichtigt, wenn die Übertragung abgeschlossen wurde. Sie kann in der Zwischenzeit andere Prozesse abarbeiten, allerdings nicht auf den Bus zugreifen.

Anmerkung zu den drei Ein-/Ausgabeschemata

Auch diese drei Varianten des Datentransfers schliessen sich nicht aus. Unterbrechungsgesteuerte Ein-/Ausgabe legt fest, wie der Prozessor über den Zustand des Geräts informiert wird. Dies kann allerdings auch mittels PIO erreicht werden, nämlich indem der Prozessor zyklisch den Status des Geräts liest und auswertet (Polling). Ist der Zustand bekannt und es stehen Daten zum Transfer bereit, kann dieser entweder mittels PIO oder DMA erfolgen, je nach Größe der zu übertragenden Daten. Lediglich DMA und PIO beschreiben also die eigentliche Datenübertragung.

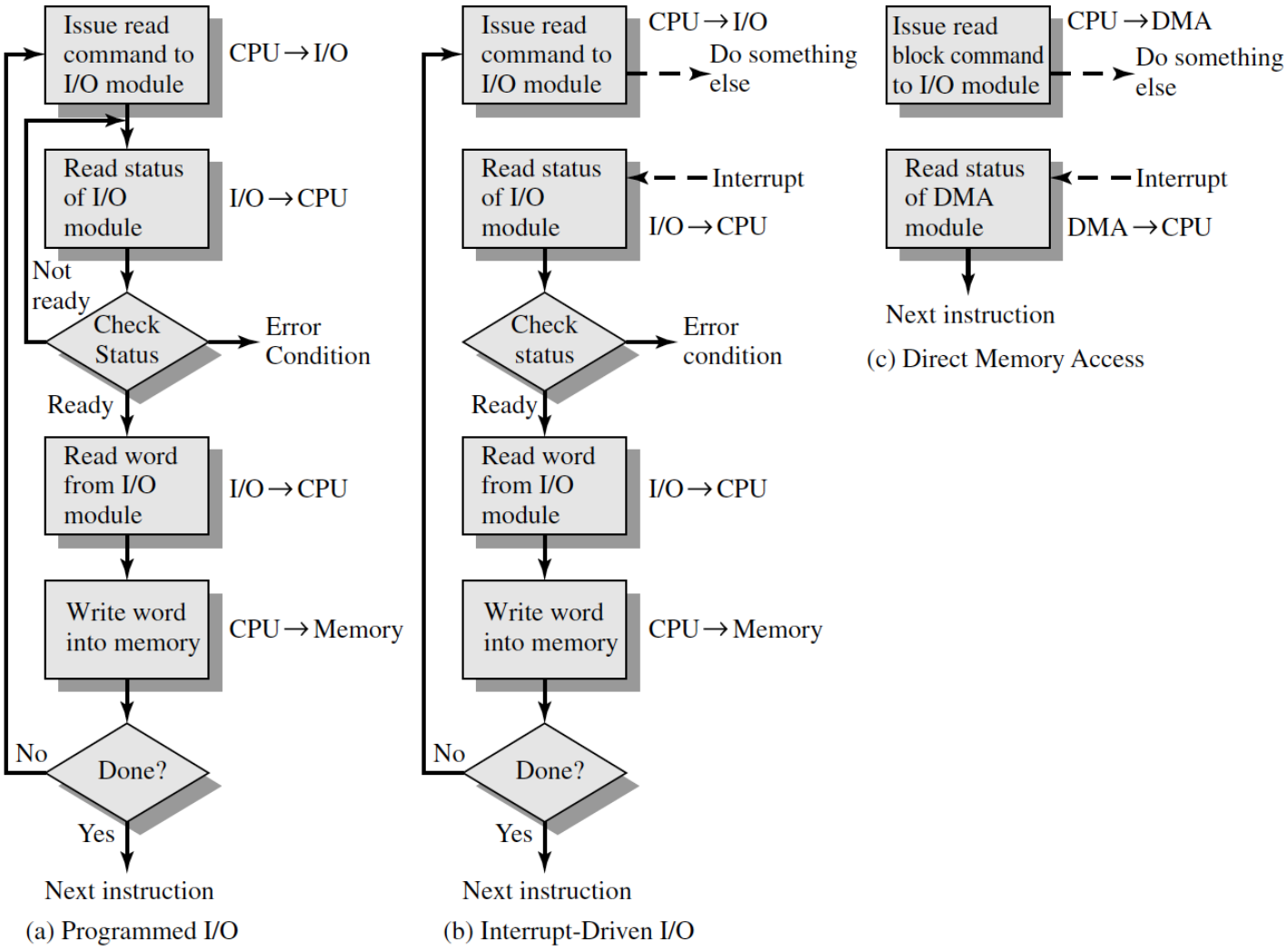


Abbildung 3.12: Eingabe eines Datenblocks auf drei verschiedene Weisen

*Gefunden in **allen** Klausuren*

Architekturen moderner Prozessoren

4.1 CISC-Architekturen

Definition(*CISC*)

Abkürzung für *complex instruction set computer*, also für einen Rechner mit einem vielschichtigem Befehlssatz. Der Prozessor in solch einem Rechner kann einen einzelnen Maschinenbefehl als mehrere Einzeloperationen direkt ausführen oder ist meistens mikroprogrammiert. Für gewöhnlich variieren die Befehlslängen, einerseits wegen verschieden langer Befehlkodes und andererseits wegen der für einen Befehl erlaubten Adressierungsarten. Entsprechend variiert die Anzahl der bei Ausführung benötigten Taktzyklen von Befehl zu Befehl.
Anders als beim RISC ist der Grundgedanke hinter der Rechnerarchitektur, die semantische Lücke auch durch Maßnahmen in Hardware zu verkleinern und dafür in Bezug auf Performanz eher einen Kompromiss einzugehen (→ Mikroprogrammierung).

Vorteile:

- Erforderliche Speicherkapazität geringer — Dies geschieht durch Mikroprogrammierung
- Semantische Lücke ist überbrückbar

Probleme:

- Zwang zur Abwärtskompatibilität, Erweiterung der Befehlssätze
- immer größere und undurchschaubarere Befehlssätze (Komplizierte Adressierungsarten, Befehle mit sehr unterschiedlichen Längen)
- Patterson-Studie → Kompilerbauer verwendeten nur noch eine kleine Teilmenge des Befehlssatzes

4.2 RISC-Architekturen

Definition(*RISC*)

Abkürzung für *reduced instruction set computer*, also für einen Rechner mit vermindertem Befehlssatz. Der Prozessor in solch einem Rechner verfügt über vergleichsweise wenige, dafür aber **hoch optimierte** Befehle. Ein weiteres typisches Merkmal ist, dass RISC-Architekturen meistens *load/store-Architekturen* sind. Für gewöhnlich sind Befehlslängen gleich, so dass jeder Maschinenbefehl gleich viel Platz im Arbeitsspeicher belegt. Darüber hinaus ist bei dieser Rechnerarchitektur angestrebt, jeden Befehl in gleicher Anzahl von Taktzyklen auszuführen (→ Grundvorraussetzung für das Pipelining).
Anders als beim CISC ist der Grundgedanke hinter der Rechnerarchitektur, Performanz vor allem durch Schlichtheit zu steigern und die semantische Lücke nur durch Maßnahmen in Software zu verkleinern.

	Complex Instruction Set (CISC) Computer		Reduced Instruction Set (RISC) Computer		Superskalar	
Name	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	MIPS R10000
Erscheinungsjahr	1973	1989	1987	1991	1993	1996
Anzahl an Instruktionen	303	235	69	94	225	
Instruktionsgröße (in bytes)	2-57	1-11	4	4	4	4
Adressierungsarten	22	11	1	1	2	1
Anzahl an Universal-registern (GPR)	16	8	40-520	32	32	32
Control memory size (Kbits)	480	246	—	—	—	—
Cachegröße (KBytes)	64	8	32	128	16-32	64

Tabelle 4.1: Charakteristiken einiger CISC-, RISC- und superskalaren Architekturen

Kennzeichen:

- Elementare, kleine, einheitliche Maschinenbefehlssätze
- Adressrechnungen werden durch explizite Befehle ausgeführt
- Register-Register-Architektur (*load-store*)
- große, universelle Registersätze
- festverdrahtete Leitwerke (→ kein Mikroprogramm)
- konsequentes Ausnutzen von Fließbandverarbeitung

  → Für ein effizientes Ausnutzen der Fließbandverarbeitung (*pipelining*) braucht es ein Zusammenspiel des **optimierenden** Kompilers und der RISC-Prozessor-Architektur

Abbildung 4.1: Befehlsabarbeitung ohne Pipelining

Pipelining

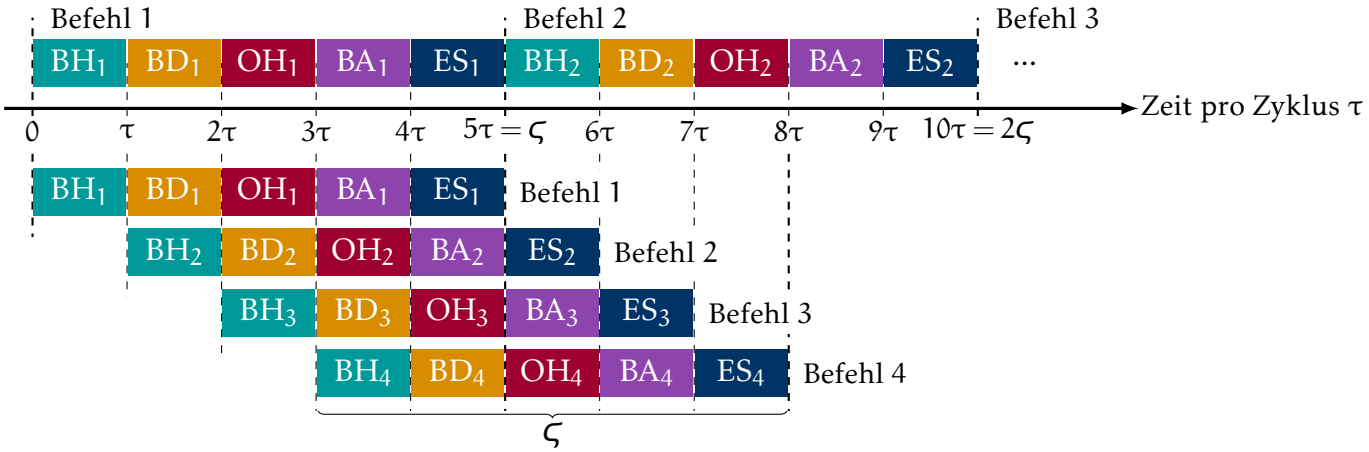
Definition(*Pipelining*)

> Die **Pipeline** bezeichnet bei Mikroprozessoren eine Art „Fließband“, mit dem die Abarbeitung der Maschinenbefehle in **Teilaufgaben** zerlegt wird, die für mehrere Befehle **parallel** durchgeführt werden. Dieses Prinzip nennt man auch **Pipelining**.

Beispielsweise nehmen wir im Folgenden an, dass unsere Pipeline die folgenden fünf elementaren Phasen hat:

- **BH** — *Befehl hohlen, was das Inkrementieren des Befehlszählers einschließt*
- **BD** — *Befehl dekodieren*
- **OH** — *Operanden holen*
- **BA** — *Befehl ausführen*
- **ES** — *Ergebnisse zurückschreiben, dies kann das Inkrementieren des Befehlszählers einschließen*[1]

Wir nehmen vier Befehle, lassen sie *ohne Pipelining* – ähnlich zu Abbildung 4.1, *über* dem Zeitstrahl – sowie **mit Pipelining** – **unter** dem Zeitstrahl – ausführen und stellen die Phasen graphisch dar:



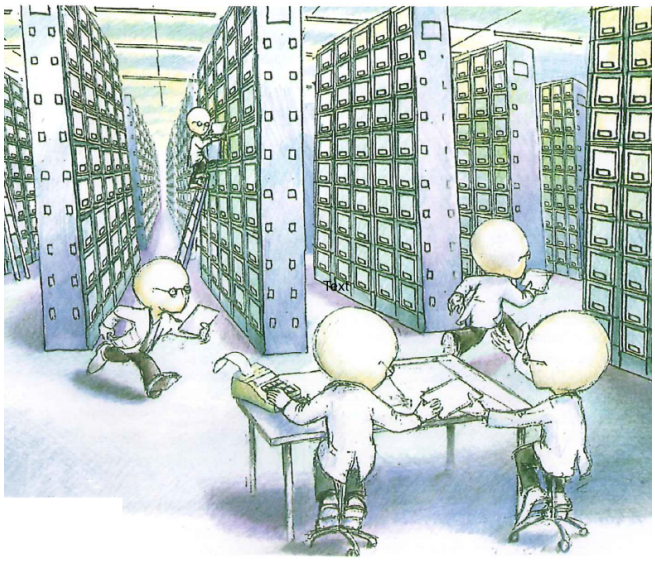


Abbildung 4.2: Pipelining mit „Eierköpfen“

Dabei stellen wir fest: Die Latenzzeit – die Dauer eines einzelnen Befehls (ς) – sinkt nicht, dafür aber der Durchsatz – der Durchschnitt aller Befehle, hier von 20 auf 8 Zeiteinheiten. Eine Illustration wie in Abbildung 4.1 findet sich in Abbildung 4.2.

Leistungssteigerung

Die Zykluszeit τ einer Pipeline kann bestimmt werden durch

$$\tau = \max_i [\tau_i] + d = \tau_m + d \quad \text{mit } 1 \leqslant i \leqslant k$$

mit τ_i der Verzögerung der i-ten Stufe, τ_m der maximalen Stufenverzögerung, k der Anzahl an Stufen und d der Zeitverzögerung, die gebraucht wird um Signale und Daten von einer Stufe zur nächsten weiterzuleiten. Generell bestimmt d sich durch einen Takt und $\tau_m >> d$.

Nehmen wir an, dass n Instruktionen sequentiell bearbeitet werden. Sei $T_{k,n}$ die Gesamtzeit bei einer

[1] bei Sprüngen

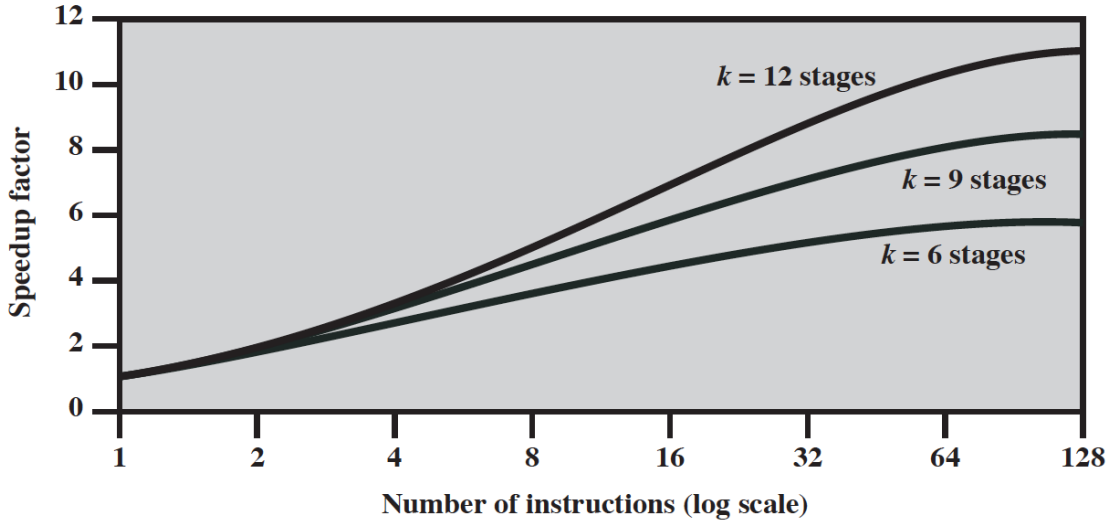


Abbildung 4.3: Kurvenverlauf des Speed-Up

Pipeline mit k Stufen zu n Instruktionen. Dann gilt:

$$T_{k,n} = [k + (n-1)] \cdot \tau$$

Anfangs ist die Pipeline leer und wird in $k \cdot \tau$ Schritten gefüllt. Nach jeder Stufe wird ein neuer Befehl in die Pipeline geladen und ein anderer Befehl wird fertiggestellt[2]. Die restlichen Befehle werden daher in $(n-1) \cdot \tau$ Schritten fertiggestellt.
Man bildet nun den Speed-Up S_k als Verhältnis von Ausführungszeit T_1 *ohne* Pipeline zu Ausführungszeit T_k **mit** k-stufiger Pipeline und erhält:

$$S_k = \frac{T_1}{T_k} = \frac{n \cdot k \cdot \cancel{\tau}}{[k + (n-1)] \cdot \cancel{\tau}} = \frac{n \cdot k}{k + (n-1)}$$

Für den maximalen Speed-Up einer solchen k-stufigen Pipeline ergibt sich dann:

$$\lim_{n \to \infty} S_k = \lim_{n} \frac{n \cdot k}{k + (n-1)} \overset{\text{l'H}}{=} \lim_{n} \frac{k}{1} = k$$

Wie in Abbildung 4.3 zu sehen ist, gilt: Desto höher die Stufenanzahl, desto später „sättigt“ sich die Kurve. Naive Lösung: $k \to \infty$ $\to$ damit ist ein höherer Chiptakt möglich aufgrund kürzerer Laufwege von einer Stufe zur nächsten.
Probleme mit dem Ansatz:

- **Anstieg an Hardwarekosten**
  Je mehr Stufen desto mehr Register werden benötigt — Denn: Zwischen Pipelinestufen müssen die Daten ja „gefangen“ werden ← Entkopplung
- **Einschwingphase dauert länger**
  Damit: Anstieg der Latenzzeit
- **Energieverbrauch nimmt zu**
  Kein Beitrag zu GreenIT
- **Wahrscheinlichkeit leerer Pipelinezyklen steigt**
  Vor allem bei Datenabhängigkeiten und Verzweigungen

[2]Dies setzt keine Hazards vorraus → siehe später

Konflikte — Hazards

Man spricht von einer **Abhängigkeit**, wenn es für die Bearbeitung eines Befehls in einer Stufe einer Pipeline **notwendig** ist, dass ein **vorheriger** Befehl bereits ausgeführt ist. Aus einer Abhängigkeit kann ein Konflikt – Hazard – entstehen.

Ressourcenkonflikte Eine Stufe der Pipeline benötigt Zugriff auf eine Ressource, welche bereits von einer anderen Stufe belegt ist.

Datenkonflikte Eine Stufe der Pipeline benötigt Daten oder Registerinhalte, welche nicht zur Verfügung stehen, oder in einem anderen Befehl verändert werden. Wir unterscheiden zwischen drei Typen:

1. **Read-After-Write (RAW)**
   *Eine Instruktion modifiziert ein Register oder eine Speicherstelle und eine nachfolgende Instruktion liest die Daten des Registers oder der Speicherstelle. Ein Konflikt tritt auf, wenn das Lesen **vor** der Schreiboperation stattfindet.*

2. **Write-After-Read (WAR)**
   *Eine Instruktion liest ein Register oder eine Speicherstelle und eine nachfolgende Instruktion verändert die Daten des Registers oder der Speicherstelle. Ein Konflikt tritt auf, wenn das Schreiben **vor** dem Lesen stattfindet.*

3. **Write-After-Write (WAW)**
   *Zwei Instruktionen modifizierten ein und dasselbe Register oder ein und dieselbe Speicherstelle. Ein Konflikt tritt auf, wenn die Schreiboperationen in **umgekehrter** Reihenfolge stattfinden.*

Kontrollflusskonflikte Ein Kontrollflusskonflikt tritt auf, wenn die Pipeline auf die Ausführung eines **bedingten** Sprungs warten muss.

Superskalare Architekturen

Definition(*Superskalarität*)

Unter **Superskalarität** versteht man die Eigenschaft eines Prozessors, mehrere Befehle aus einem Befehlsstrom gleichzeitig mit mehreren **parallel** arbeitenden Funktionseinheiten zu verarbeiten. Es handelt sich um eine Nebenläufigkeit auf Befehlsebene, bei der die feinkörnige Nebenläufigkeit zwischen den einzelnen Befehlen ausgenutzt wird.

Achtung

Die Befehle bei einer superskalaren Architektur müssen **nicht synchron** abgearbeitet werden.

Das eigentliche Grundprinzip der Superskalarität ist von Supercomputern übernommen, die bereits 1964 mehrere parallel arbeitende Funktionseinheiten für unterschiedliche mathematische Operationen – beispielsweise die Vektorrechnung – bereitstellten.
Die Anwendung auf Befehle benötigt einen Befehlsgruppierer. Zudem werden die sequentiell einlaufenden Befehle zur Laufzeit umgeordnet, um sie parallel auszuführen (→ dynamische Parallelisierung). Damit treten auch häufiger WAR und WAW Hazards auf.

4.3 VLIW – Very Long Instruction Word

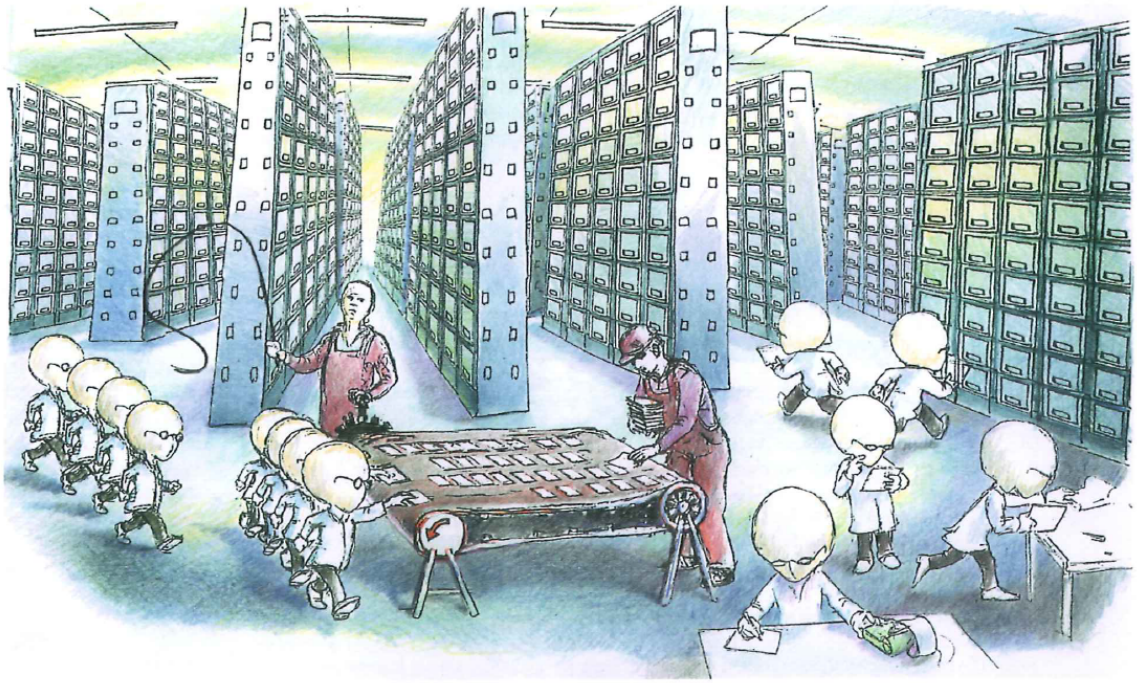


Abbildung 4.4: Superskalare Architektur mit Befehlsgrupperier (mit Peitsche) und mehreren Rechenwerken (mehrere Gruppen an „Eierköpfen“)

Definition(*VLIW*)

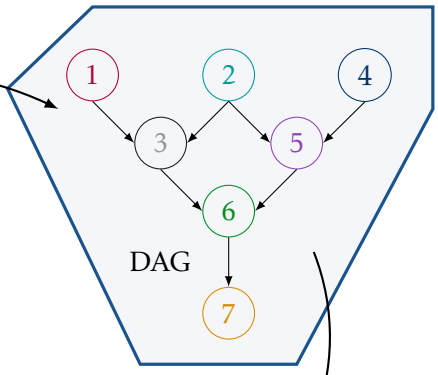
Very Long Instruction Word (VLIW) bezeichnet eine Eigenschaft einer Befehlssatzarchitektur. Ziel ist die Beschleunigung der Abarbeitung von sequentiellen Programmen durch Ausnutzung von Parallelität auf **Befehlsebene**. Jedoch wird – im Gegensatz zu *superskalaren Prozessoren* – nicht dynamisch parallelisiert, sondern **statisch** vom **Kompiler**, der parallel ausführbare Befehle gruppiert.

Zum Gruppieren fasst der Kompiler die Befehle zu einem Datenabhängigkeitsgraphen zusammen, um möglichen Konflikten vorzubeugen.

Beispiel

```
1  ADD R2, R3 → R1
2  MUL R4, R5 → R6
3  SUB R6, R1 → R7
4  ADD R8, R9 → R10
5  DIV R10, R6 → R11
6  ADD R7, R11 → R12
7  STORE R12 → X
```

Kompiler erzeugt Datenabhängigkeitsgraphen (DAG) aus Quellkode



Mit dem DAG werden die Befehle dann auf die einzelnen ALUs verteilt

ALU 1	ALU 2	ALU 3
1: ADD R2, R3 → R1	2: MUL R4, R5 → R6	4: ADD R8, R9 → R10
3: SUB R6, R1 → R7	5: DIV R10, R6 → R11	
6: ADD R7, R11 → R12		
7: STORE R12 → X		

4.4 Multi-Threading

Definition(*Thread*)

Als einen Faden (*thread*) – eher Aktivitätsträger oder leichtgewichtiger Prozess – bezeichnen wir einen Ausführungsstrang, der einen eigenen Laufzeitkontext besitzt und typischerweise mitlaufender Ablaufplanung unterliegt, in der Abarbeitung eines Programms. Ein Thread ist immer Teil eines Prozesses. Man differenziert in Anwendungsfaden (*user thread*) und Systemkernfaden (*kernel thread*).

Definition(*Anwendungsfaden*)

Bezeichnung für einen Faden im Maschinenprogramm, der daselbst implementiert ist und nicht erst durch einen Faden im Betriebssystemkern entsteht. Sämtliche für solch einen Faden erforderlichen Betriebsmittel sowie für seine Verwaltung nötigen Funktionen stellt das als ↑nichtsequentielles Programm ausgelegte Maschinenprogramm. Dies betrifft insbesondere den Laufzeitkontext eines Fadens und die Funktionen zur Ablaufplanung, Einlastung und Synchronisation.
Ein Prozesswechsel, um innerhalb des Maschinenprogramms vom aktuellen zu einem anderen Faden umzuschalten, verläuft im lokalen Adressraum und auch unter Speicherschutz ohne Systemaufruf. Jeder dieser Fäden ist vom Bautyp her ein federgewichtiger Prozess, für den zur Steuerung und Überwachung keine kostspielige Systemfunktion erforderlich ist und er selbst dazu auch keine benutzt. Für das Betriebssystem ist ein solcher Faden kein Objekt erster Klasse, das heißt, der durch den Faden konkretisierte Prozess ist dem Betriebssystem gänzlich unbekannt. Eine vom Maschinenprogramm aus beanspruchte Systemfunktion läuft damit nicht im Namen des effektiv beanspruchenden Fadens, sondern nur im Namen derjenigen Entität ab, die im Betriebssystem das jeweils in Ausführung befindliche Maschinenprogrammen repräsentiert. Ist dies beispielsweise ein schwergewichtiger Prozess oder ein leichtgewichtiger Prozess und wird dieser dann durch die Systemfunktion blockiert (*Prozesszustand*), blockieren implizit alle Fäden, die unbekannterweise eben auf diese eine Prozessinkarnation durch eine Aktion im Maschinenprogramm abgebildet wurden.

Definition(*Systemkernfaden*)

Bezeichnung für einen Faden im Maschinenprogramm, der durch eine eigene Prozessinkarnation im Betriebssystemkern implementiert ist. Sämtliche für solch einen Faden erforderlichen Betriebsmittel sowie für seine Verwaltung nötigen Funktionen stellt das Betriebssystem. Dies betrifft den Laufzeitkontext des Fadens und die Funktionen zur Ablaufplanung, Einlastung und Synchronisation: All diese Funktionen, insbesondere auch der Prozesswechsel, verlaufen unter Kontrolle des Betriebssystemkerns und erfordern daher einen Systemaufruf, wenn innerhalb des Maschinenprogramms vom aktuellen zu einem anderen Faden umgeschaltet werden soll. Damit ist jeder dieser Fäden aus Sichtweise des Maschinenprogramms vom Bautyp her ein leichtgewichtiger Prozess, da – im Gegensatz zum Anwendungsfaden – zwar vergleichsweise aufwändige Systemaufrufe zu seiner Kontrolle erforderlich sind, jedoch bei Fadenwechsel im selben Maschinenprogramm derselbe gegebenenfalls unter Speicherschutz stehende Adressraum aktiv bleibt. Für das Betriebssystem ist ein solcher Faden ein Objekt erster Klasse, das heißt, der durch den Faden konkretisierte Prozess ist insbesondere auch dem Betriebssystemkern bekannt. Eine vom Maschinenprogramm aus beanspruchte ↑Systemfunktion läuft damit immer im Namen dieses Fadens ab. Wird dieser dann durch die Systemfunktion blockiert (*Prozesszustand*), kann der Betriebssystemkern von selbst zu einen anderen Faden dieser Art im selben Maschinenprogramm umschalten.

Definition(*Multithreadingarchitekturen*)

In vielfädigen Architekturen werden die Rechenwerke nicht mit einzelnen Befehlen gefüttert, sondern mit Befehlen aus Threads. Die Zuteilung „Threads $\mapsto$ Rechenwerke“ (*Sheduling*) geschieht in der Hardware.

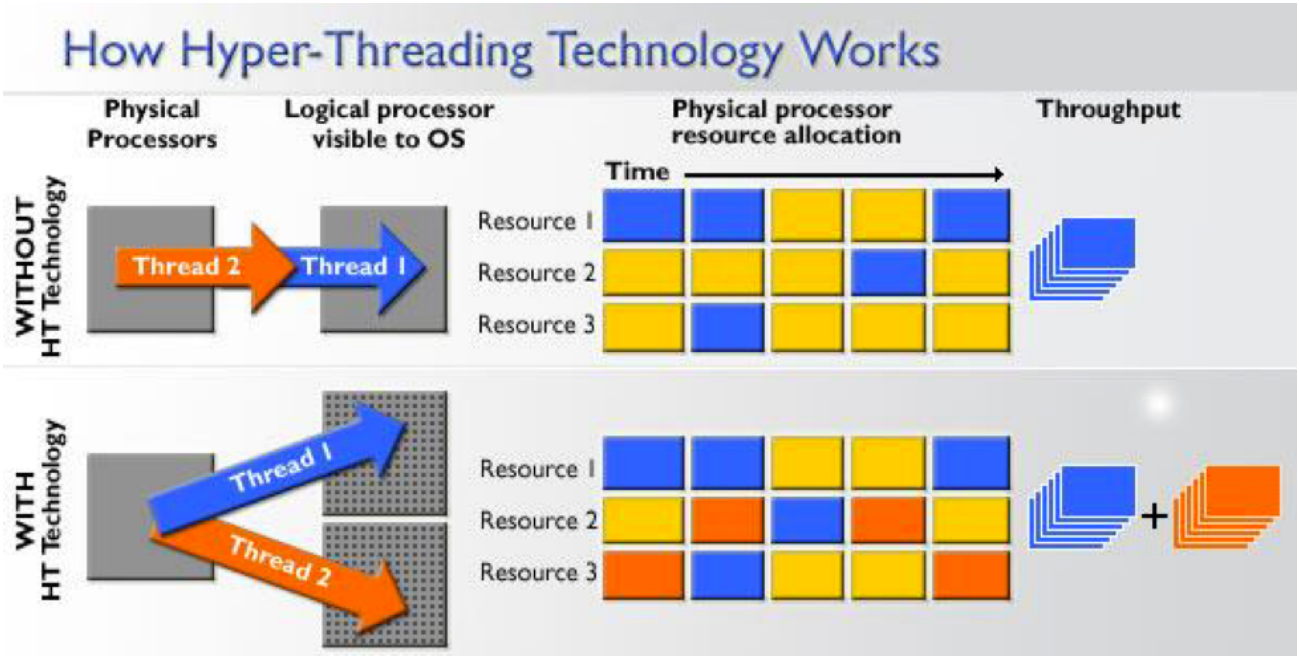
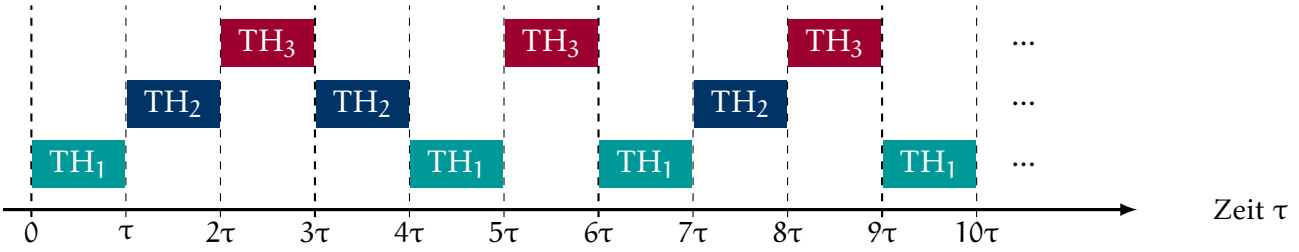


Abbildung 4.5: Wie Hyperthreading funktioniert

Zeitscheiben Multithreading



Jeder Thread bekommt bei dieser Variante ein festes zeitliches Raster (hier: τ) zugeteilt und ist in diesem Raster aktiv. Nach Ablauf des Rasters wird automatisch auf einen anderen Thread umgeschaltet.

Ereignisgesteuertes Multithreading

Definition(*Switch on Event Multithreading (SoEMT)*)

SoEMT ist eine Art des Multithreading, bei der die Threadumschaltung durch Ereignisse ausgelöst wird. Das Ereignis kann die Ausführung einer Operation sein, die eine lange Latenz mit sich führt – wie zum Beispiel der Zugriff auf I/O, ein Cache-Miss, ein TLB-Miss, ... – oder auch der Rücksprung aus einer solchen Operation.

Simultanes Multithreading

Definition(*Simultanes Multithreading (SMT)*)

SMT bezeichnet die Fähigkeit eines Prozessors, mittels getrennter Pipelines und/oder zusätzlicher Registersätze, mehrere Fäden **gleichzeitig** auszuführen. Damit stellt SMT eine Form des hardwareseitigen Multithreadings dar. Man nennt eine dieser „Unterprozeduren“ dann auch **logischer Prozessor**.

Angetrieben wurde die Entwicklung durch den überproportionalen Anstieg an elektrischer Leistung und Chip-Fläche gegenüber dem Zuwachs an Rechenleistung.

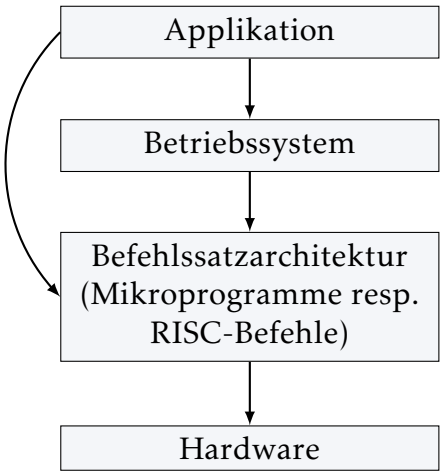
*Gefunden in **allen** Klausuren*

Schnittstelle zum Betriebssystem

5.1 Anbindung zum Betriebssystem

Aufgaben des Betriebssystems

- Das Betriebssystem ist die Schnittstelle zur Hardware
- Das Betriebssystem stellt den Betrieb mit der Peripherie sicher
- Das Betriebssystem erfüllt Dienste für die Benutzung des Rechners durch die Verwaltung der Ressourcen
  - *Dateiverwaltung (Kopieren, Löschen, ...)*
  - *Prozessverwaltung*
  - *Speicherverwaltung*
- Das Betriebssystem stellt Schutzmechanismen – zum Beispiel für den Mehrbenutzerbetrieb – bereit



Das Betriebssystem führt neue Befehle und Merkmale oberhalb der Ebene der Befehlssatzarchitektur aus. Die Ebene des Betriebssystems enthält die Schnittstelle zum Anwendungsprogrammierer und ist in Software implementiert.

Achtung

Mikroprogramme sind aber über den Assembler nach wie vor direkt aufrufbar. Jedoch erzeugen manche Befehle Ausnahmen, zum Beispiel wenn versucht wird, priveligierte Bereiche zu adressieren.

Systemaufrufe (system calls)

Systemaufrufe aktivieren einen vordefinierten Betriebssystemdienst wie beispielsweise die Ein- oder Ausgabe. Systemaufrufe können über Mikroprogramme oder Assemblerprogramme implementiert werden.

5.2 Memory Management

Virtueller Adressraum – Virtueller Speicher

Definition(*Realer Adressraum*)

Der durch einen Prozessor definerte Wertevorrat $A_r = [0,\ 2^n - 1]$ von Adressen, normalerweise mit $16 \leqslant e \leqslant n \leqslant 64$. Nicht jede Adresse in A_r ist gültig, d.h. A_r kann Lücken aufweisen.

Definition(*Logischer Adressraum*)

Der in Programm P definierte Wertevorrat $A_l = [n,\ m]$ von Adressen, mit $A_l \subset A_r$, der einem Prozess von P zugebilligt wird. Jede Adresse in A_l ist gültig, das heißt A_l enthält *konzeptionell* keine Lücken

Definition(*Virtueller Adressraum*)

$A_v = A_l$: A_v übernimmt alle Eigenschaften von A_l. Jedoch nicht jede Adresse in A_v bildet auf ein im Hauptspeicher liegendes Datum ab.

Virtueller Adressraum

Die Benutzung einer solchen – nicht abgebildeten – Adresse in A_v resultiert in einem **Zugriffsfehler** und dann in einer synchronen Programmunterbrechung (*trap*). Das Betriebssystem sorgt dann für die Einlagerung des adressierten Datums in den Hauptspeicher und der Prozess wird zur Wiederholung der gescheiterten Aktion gebracht.

Die Idee kam mit dem Auftreten der Speicherüberlagerungsprogramme (*Overlays*) auf den Sekundärspeicher. Jedoch war der Programmierer **eigenverantwortlich** – nicht das Betriebssystem – für die Verwaltung der Overlays ($\rightarrow$ Seitenumlagerung).
Damit kam dann die Idee auf, sich einen großen Speicher A_v mit $v_i \in A_v$ vorzustellen, der dann mit einer Abbildung $v_i \mapsto r_i$ mit $r_i \in A_r$ „übergelegt“ wird.

Paging

Definition(*Seitenumlagerung – Paging*)

Funktion in einem Betriebssystem, durch die eine bei der Ausführung von einem Programm referenzierte, aber nicht im Hauptspeicher vorliegende Seite vom Hintergrundspeicher automatisch eingelagert wird. Gegebenenfalls wird dazu, wenn nämlich noch Platz für die Einlagerung zu schaffen ist, eine im Hauptspeicher vorliegende Seite vorher auf den Hintergrundspeicher ausgelagert. Grundlage dafür bildet seitennummerierter virtueller Speicher. Für den anfallenden Transfer aller betroffenen Seiten im Vorder- und Hintergrundspeicher sorgt das Betriebssystem.

Definition(*Seite*)

Speicherstück fester Größe, die immer eine Zweierpotenz der Größe von einem Speicherwort ist. Die effektive Größe hängt ab von verschiedenen Faktoren: Umfang der Seitentabelle, verfügbarer Platz im Übersetzungspuffer, interner Verschnitt und der Zugriffszeit auf den Hintergrundspeicher.

Das Ziel der Seitenumlagerung ist es eine – für den Programmierer – völlig unsichtbare Abbildung zu schaffen. Es soll quasi die Illusion eines im Prinzip beliebig großen, aber linear adressierbaren Hauptspeichers entstehen, bei dem die Seitenumlagerung im Hintergrund geschieht.
Der virtuelle Adressraum wird hierzu dann eingeteilt in Seiten gleicher Größe N. Es gilt meistens $512\ \text{Byte} \leqslant N \leqslant 4\ \text{MB}$ und immer $\exists k \in \mathbb{N} : N = 2^k$.
Der vorgesehene Platz im Hauptspeicher zur Aufnahme von Seiten heißt Seitenrahmen.

Definition(*Seitenrahmen*)

Ein Abschnitt fester Größe im Hauptspeicher zur Aufnahme von exakt einer Seite: die effektive Rahmengröße ist definiert durch die Seitengröße. Die Nummer eines solchen Rahmens bildet die reale Adresse einer für gewöhnlich nur durch eine logische Adresse oder virtuelle Adresse

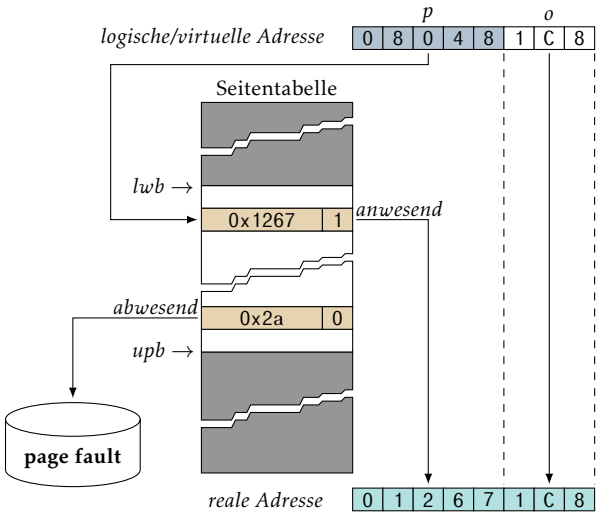
(a) Schaubild für den internen Verschnitt (rot dargestellt)

(b) Schaubild für den externen Verschnitt (Löcher)

Abbildung 5.1: Schaubilder für Verschnittarten

erreichbaren Seite. Im Hintergrund steht ein seitennummerierter Adressraum, dessen einzelne Seiten auf korrespondierende Seitenrahmen abzubilden sind. Die hierfür nötige Adressumsetzung leistet eine Speicherverwaltungseinheit (*memory management unit* (MMU)), die dazu vom Betriebssystem vorher entsprechend programmiert werden muss.

Beim Paging wird jede Adresse als Tupel $A_p = (p, o)$ interpretiert, wobei o die Oktettnummer (also die Verschiebung in einer Seite) und p die Seitennummer (also die Speicherstelle der Seite in der Seitentabelle) darstellt.
Über die virtuelle Seitennummer wird in der Seitentabelle der zugehörige Eintrag geholt. Das `valid`-Bit zeigt an, ob die virtuelle Seite im Hauptspeicher liegt oder nicht (siehe rechts → an-/abwesend). Der Eintrag enthält dann die zugehörige Nummer der physikalischen Seite – sofern sie im Hauptspeicher ist – oder enthält einen Verweis aus dem hervorgeht, an welcher Stelle die Seite im Sekundärspeicher zu finden ist – sofern sie abwesend ist.



Die Adresse besteht aus der Seitennummer und dem Offset, welcher auf die Adresse innerhalb der Seite verweist. Der Offset der virtruellen und realen Adresse ist identisch, wohingegen die Seitennummer mit der Seitentabelle übersetzt werden muss. Meistens gilt: #Seitenrahmen < #virtuelle Seiten. Damit kann es Situationen geben, in denen eine virtuelle Seite angefordert wird, die nicht in einem Seitenrahmen vorliegt. Es kommt zum *trap* → **page fault**.
Ein solcher Seitenfehler heißt **gültig**, wenn $lwb \leqslant p \leqslant upb$, dann ist p abwesend oder ungenutzt. Eine ungenutzte Seite ist gültig, sie wurde nur noch nicht abgebildet; er heißt **ungültig**, wenn $lwb \leqslant p \leqslant upb$ nicht gilt, dann gehört die betreffende Seite nicht zum Prozessadressraum.

Nachteil – Interner Verschnitt (Fragmentierung)

Definition(*Interner Verschnitt*)

Verschnitt innerhalb von einem Speicherstück. Normalerweise bezogen auf eine Seite, betrifft aber jeden Typ Speicherbereich, dessen Größe ein echtes Vielfaches der Größe eines Byte ist und am Ende ein Loch aufweist. Typisch ist ein solcher Verschnitt, wenn ein seitennummerierter Adressraum die Grundlage für die Verwaltung von Arbeitsspeicher bildet. Es gilt: Kleinere Seitenrahmen verringern einen internen Verschnitt, da die Speichervergabe genauer erfolgen kann. Dargestellt in Abbildung 5.1a

Problem

Der Prozess erhält mehr Speicher als angefordert und dürfte in logischer Hinsicht nicht auf den dem Loch entsprechenden Seitenabschnitt zugreifen. Physisch kann er an einem solchen Zugriff allerdings nicht gehindert werden — obwohl er sich in dem Fall fehlerhaft verhalten würde, da er das Loch nicht kennen dürfte. Die MMU kann aber diesen Zugriffsfehler nicht feststellen.

Beispiel

Das Betriebssystem verwaltet einen seitennummerierten Adressraum mit einer Seitenrahmengröße von 4 KiB. Ein Prozess beantragt 5 KiB. Das Betriebssystem reserviert daraufhin 2 Seiten, also 8 KiB. Es bleibt ein interner Verschnitt von 3 KiB. Mit kleineren Seitenrahmengrößen, zum Beispiel 2 statt 4 KiB, wäre der interne Verschnitt von 3 auf 1 KiB gesunken. Problem: Es wird mehr Speicherplatz für die Seitenrahmen gebraucht ...

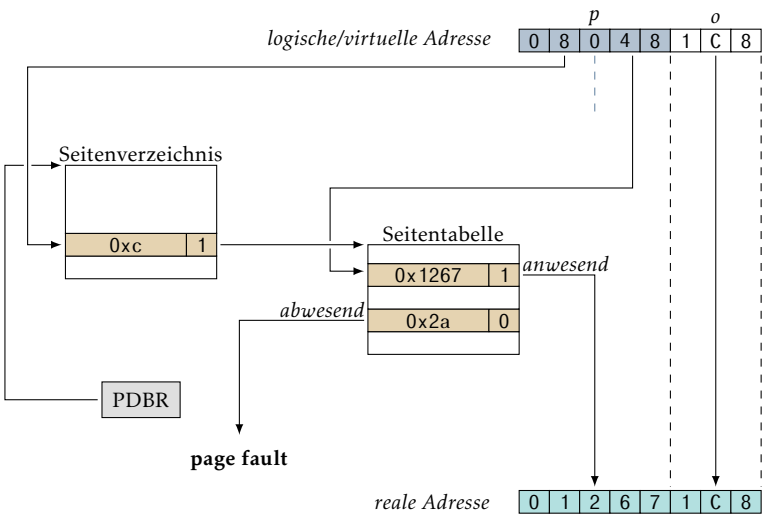
Wie kann man interne Fragmentierung vermeiden?

Interne Fragmentierung lässt sich durch Relozierung von Speicherbereichen oder einer Heap-Kompaktifizierung abmildern (!), kleinere Seitengrößen zeugen dabei auch von Vorteil.

Optimierung bei der Speicherverwaltung – Speicherplatzreduktion via mehrstufiger Seitenverwaltung

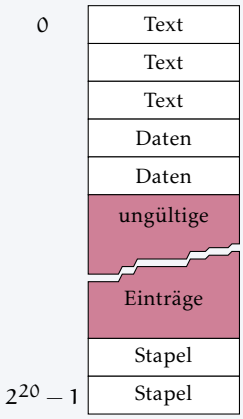
Das Problem bei einstufiger Seitenverwaltung ist, dass man immer so viele Einträge brauch, wie es virtuelle Seiten geben **könnte**. Für mehrere Prozesse ist dies ein unakzeptabler Aufwand. Man beruft sich deshalb auf die Idee, dass jeder Prozess ein **Seitenverzeichnis** erhält, dessen Einträge auf eine Seitentabelle verweisen. Ein jeder dieser Einträge kann leer sein, wenn der Prozess den Adressraum nicht braucht und spart dadurch Speicher. Die Einträge in den Seitentabellen zeigen dann – wie üblich – auf die Seitenrahmen.

Alle Seitentabellen und Seitenverzeichnisse sind gleich groß für alle Prozessexemplare. Die Seitennummer wird aufgeteilt in Seitenverzeichnissoffset und Seitentabellenadresse. Das *page directory base register* (PDBR) verweist auf das erste Seitenverzeichnis. Mit dem Seitenverzeichnissoffset wird dann die richtige Seitentabelle nachgeschlagen und dann mit der Seitentabellenadresse die richtige Seite bestimmt. Es kommt hierbei nur dann zu einem Trap, wenn auf einen ungültigen oder leeren Seitendeskriptor verwiesen wird.



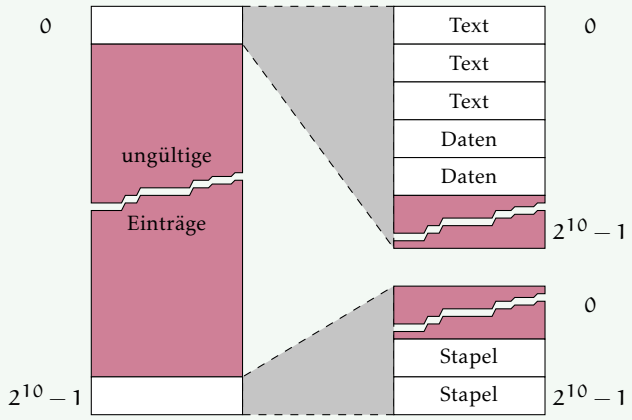
Rechenbeispiel Ein Prozess belege 12 KiB Text, 8 KiB Daten und 8 KiB Stapel. Es kommt ein seitennummerierter Adressraum zum Einsatz mit 4 KiB Seiten, 32-Bit Seitendeskriptoren und 32-Bit Adressen.

einstufige Seitenverwaltung



$2^{20}-7$ ungültige Einträge
→ Trotz einer Tabelle pro Prozessexemplar **schon 4 MiB** Speicherplatzbedarf

zweistufige Seitenverwaltung



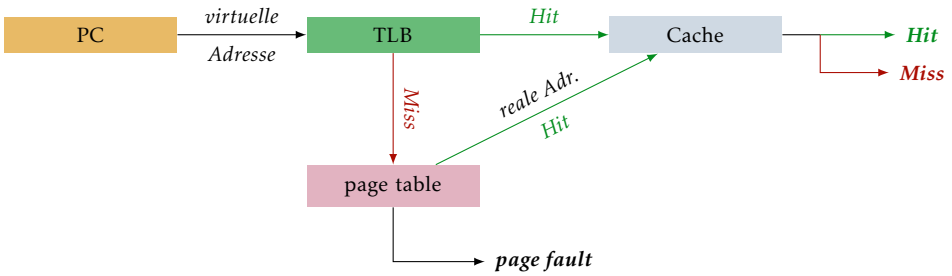
$3 \cdot 2^{10} - 9 = 3063$ ungültige Einträge
→ Trotz drei Tabelle pro Prozessexemplar **nur 12 KiB** Speicherplatzbedarf

Optimierung bei der Speicherverwaltung – Zugriffsgeschwindigkeitserhöhung via TLB

Definition(*Übersetzungspuffer – translation look-aside buffer*)

> Auch kurz als TLB bezeichnete Funktionseinheit zur schnellen Adressabbildung, integriert in einer MMU oder CPU. Beim Zugriff auf den Arbeitsspeicher ermöglicht diese Einheit mit einem kurzen „Blick zur Seite“ (*look aside*) festzustellen, ob eine von der CPU verwendete logische Adresse direkt in eine reale Adresse übersetzt werden kann. Die für die Übersetzung erforderlichen Hinweise werden entweder von der MMU oder dem Betriebssystem in dem Puffer zwischengespeichert.

Der TLB stellt damit eine Art Cache für die Einträge in den Seitentabellen dar. Unter der Annahme, dass ein Cache physikalische Adressen speichert, ergibt sich folgendes Schaubild:



Vorteile der Seitenumlagerung

- Adressrechnung ist bei der Seitenumlagerung einfacher – sie besteht aus einer Konkatenation der Seitenanfangsadresse (→ ermittelt über die Seitentabelle) und dem Versatz.
  im Allgemeinen sind die Seitentabellen größer als die Segmenttabellen, wodurch letztere leichter in schnellen Registern zu halten sind. Dafür gibt es jedoch bei der Seitenumlagerung die Abhilfe mit dem Übersetzungspuffer (TLB)

- Es kommt zu keinem externen Verschnitt

- Die Adressrechnung der Segmentierung erfordert einen Addierer (→ zusätzlicher Hardwareaufwand)

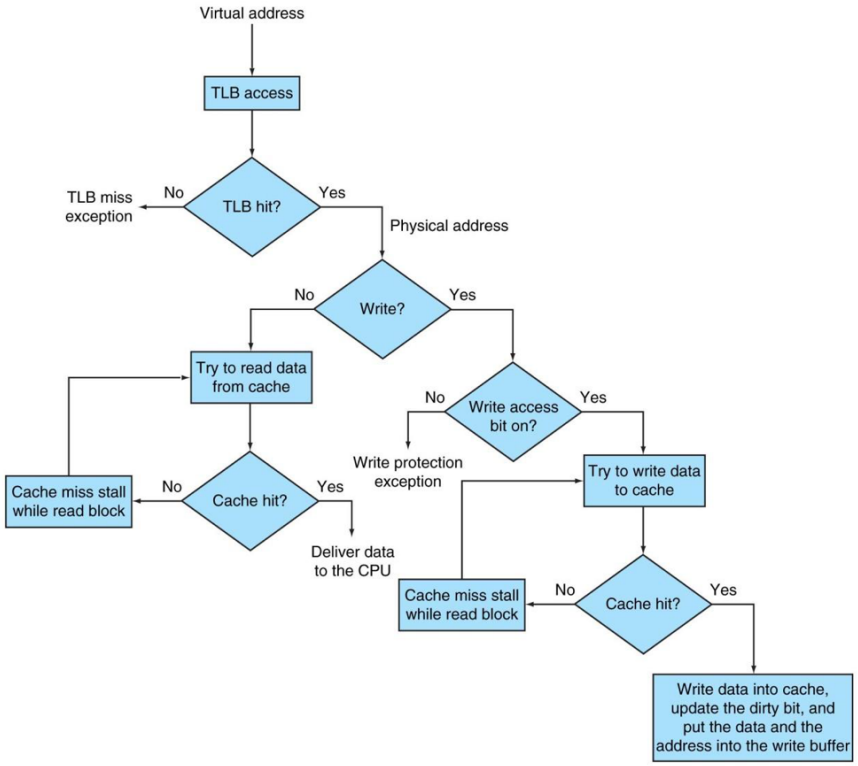


Abbildung 5.2: Zugriffsablauf TLB und Cache algorithmisch dargestellt — Dabei wurde eine *write-through* Strategie angenommen

TLB	**Seitentabelle**	**Puffer**	*Ist das möglich? Wenn ja wie?*
Hit	Hit	Hit	trivial
Hit	Hit	Miss	Möglich, wenn gleich unwahrscheinlich, da die Seitentabelle nicht wirklich kontrolliert wird bei einem TLB-Hit
Miss	Hit	Hit	TLB trifft nicht, der Eintrag wird in der Seitentabelle gefunden; Nach erneutem Versuch wird das Datum im Puffer gefunden
Miss	Hit	Miss	TLB trifft nicht, der Eintrag wird in der Seitentabelle gefunden; Nach erneutem Versuch wird das Datum im Puffer **nicht** gefunden
Miss	Miss	Miss	TLB trifft nicht, gefolgt von einem ***page fault***; Nach erneutem Versuch **muss** das Datum im Puffer **nicht** gefunden werden
Hit	Miss	Miss	**Unmöglich**: Es kann keine Übersetzung im TLB geben, wenn die Seite nicht im Speicher *anwesend* ist.
Hit	Miss	Hit	**Unmöglich**: Es kann keine Übersetzung im TLB geben, wenn die Seite nicht im Speicher *anwesend* ist.
Miss	Miss	Hit	**Unmöglich**: Das Datum darf nicht in den Cache, wenn die Seite nicht im Speicher *anwesend* ist.

Tabelle 5.1: Mögliche und nicht mögliche Kombination von TLB, Seitentabelle und Cache

Segmentierung

Definition(*Segmentierung*)

Zerlegung eines komplexen Ganzen in einzelne Abschnitte. Jeder Abschnitt bildet ein eingenständiges Segment, das Text oder Daten enthält. Das komplexe Ganze entspricht einem Adressraum.

Definition(*Segment*)

Unterteilung im Prozessadressraum, wobei der Adressbereich dieser Unterteilung auf den Speicher eines Rechensystems abgebildet ist, genauer: dem Vordergrundspeicher und Hintergrundspeicher. In diesem Bereich liegt Programmtext oder -daten, auch bezeichnet als Textsegment und Datensegment. Ein solcher Adressraum kann mehrere dieser Bereiche umfassen, die sich nicht überlappen und von fester oder variabler Größe sind. Beispiele für Segmente sind grobkörnig Bereiche wie ganze Dateien oder gesamte Text- oder Datenbereich von einem Programm oder feinkörnige Gebilde wie ein einzelnes Unterprogramm, Datenstrukturen, Objekte oder Variablen.

Wir definieren uns damit mehrere – vollkommen unabhängig voneinander vorhandene – Adressräume, unsere **Segmente**.

Vorteil Segmente

- Segmente können unabhängig voneinander wachsen und schrumpfen
- Segmente unterstützen die Modularisierung bei der Programmierung
  - Jede Prozedur hat ein eigenes Segment mit Anfangsadresse 0
  - Aufrufe einer anderen Prozedur über Angabe der Segmentnummer und Adresse 0
  - Nachträgliche Änderungen an einer Prozedur sind leichter umzusetzen
  - Einbinden von Bibliotheken leichter → *dynamisches Binden*
- Schutzmechanismen der Segmentierung sind einfacher
  Grenzschutzregister → Löst beim Überschreiten einen trap aus.

Achtung

Das Segment ist eine **logische** Einheit, von dessen **Existenz** der Programmierer wissen **muss**, da er das Segment explizit verwenden muss.

Übersetzung

Aus der Segmenttabelle wird über die Segmentnummer die zugehörige Anfangsadresse ermittelt.

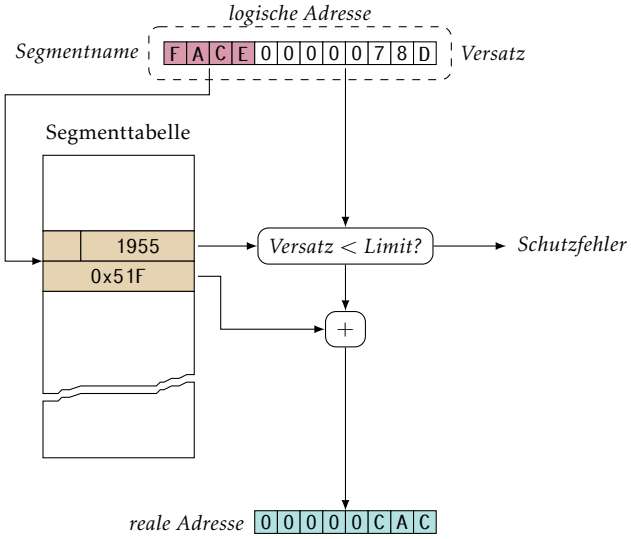
Übersetzung der Adressen

Definition(*Segmentdeskriptor*)

Bezeichnung für einen Deskriptor von einem Segment. Datenstruktur einer MMU, um eine logische in eine reale Adresse umzusetzen. Enthält typischerweise folgende Informationen:

- **Basis**
- **Limit**
- **Attribute** (wie Typ, Zugriffsrechte, Expansionsrichtung, Präsenzbit, ...)

Beispiel Angenommen, die CPU dereferenziert die Adresse `0x78D` im Segment `0xFACE` ⇝ zweikomponentige Adresse.



Auch hierbei kann es zu einer Ausnahme wegen Abwesenheit (*swap-out*) kommen. Ein referenziertes – im Speicher jedoch nicht vorhandenes – Segment wird nachgeladen. Sollte kein Platz vorhanden sein, müssen ein oder mehrere Segmente auf den Sekundärspeicher geschrieben werden.
Andere Ausnahmen sind Schutz- und Zugriffsverletzungsfehler. Zum Schutzfehler (*segmentation fault*) kommt es, wenn die Adresse größer als die Segmentlänge ist oder außerhalb des Segments liegt. Zu einer Zugriffsverletzung kommt es, wenn der Fordernde die falschen Rechte besitzt.

Nachteil – Externer Verschnitt (Fragmentierung)

Definition(*Externer Verschnitt*)

Verschnitt im Hauptspeicher. Der Grad der Fragmentierung ist so hoch, dass kein einziger freier Abschnitt für die Speicherzuteilung nutzbar ist. Oft sind in der Situation zwar genügend viele Byte im Hauptspeicher frei, sie liegen jedoch zu stark verstreut vor und bilden damit keinen zusammenhängenden Abschnitt angeforderter Größe. Der Verschnitt lässt sich durch Defragmentierung oder aber auch der Nutzung von Paging (→ führt aber u.U. zu internem Verschnitt) auflösen. Dargestellt in Abbildung 5.1b.

Definition(*Defragmentierung*)

Verfahren zur Auflösung von externem Verschnitt. Die im Hauptspeicher belegten Abschnitte werden geeignet verschoben, um einen einzigen zusammenhängenden freien Abschnitt zu schaffen. Voraussetzung dafür ist ein logischer Adressraum für jeden Prozess, dessen im Hauptspeicher liegende Anteile von Text und Daten von der Verschiebung betroffen sind: die reale Adresse eines jeden verschobenen Abschnitts ändert sich, dessen logische Adresse jedoch **nicht**.

Segmentierung + Paging

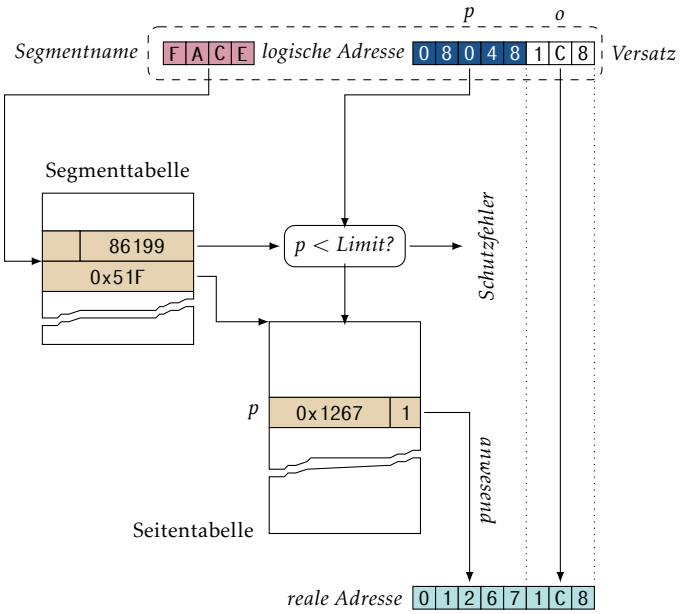
Definition(*Seitennummerierte Segmentierung*)

Art der Segmentierung von einem Adressraum, in dem ein Segment als seitennummerierter Adressbereich aufgestellt ist. Ein Segmentdeskriptor beschreibt dabei die für das Segment benötigte Seitentabelle (insbesondere die Anfangsadresse und Länge). Je nach MMU wird die globale Segmenttabelle ebenfalls als Segment erfasst (das sogenannte Wurzelsegment). Kommt zusätzlich seitennummerierter virtueller Speicher zum Einsatz, können damit für sehr große Segmente wie auch Segment- resp. Seitentabellen nur die jeweils benötigten Abschnitte im Hauptspeicher gehalten werden.

Die logische Adresse besteht damit aus einem Tupel $A = (s,(p,o))$, also aus dem Segmentnamen s, dem Seitenrahmenversatz p und dem Versatz in der Seite o. **Ein** Segment erfasst letztlich **eine** Seitentabelle **bestimmter** Größe. Die Segmentanfangsadresse lokalisiert die Seitentabelle im Hauptspeicher, die Segmentlänge definiert die Größe der Seitentabelle.

Achtung

Es kann jetzt **sowohl** zu **externem** als auch zu **internem** Verschnitt kommen.



Vergleich der Segmentierung zur Seitenumlagerung

Wegen des Vorteils, dass die Seitenumlagerung im Hintergrund bleibt, wird heutzutage meist die Seitenumlagerung verwendet. Das Problem mit dem Aufwand für die Seitentabellen ist immer noch vorhanden, da jeder Prozess seine eigene Seitentabelle besitzt.

Consideration	Paging	Segmentation
Need the programmer be aware of it?	No	Yes
How many linear addresses spaces are there?	1	Many
Can virtual address space exceed memory size?	Yes	Yes
Can variable-sized tables be handled easily?	No	Yes
Why was the technique invented?	To simulate large memories	To provide multiple address spaces

Abbildung 5.3: Vergleich zwischen Seitenumlagerung und Segmentierung in verschiedenen Punkten