

- Moore's Law: Number of transistors doubles every 2 years

- IC = Instruction Count
CPI = Cycles Per Instruction

- $\text{CPU_Zeit} = \text{IC} * \text{CPI} * \text{Taktdauer} = (\text{IC} * \text{CPI}) / \text{Taktfrequenz}$

- $\text{CPI} = \text{Anzahl Taktzyklen} / \text{IC}$

CISC

Complex Instruction Set Computer

- Komplexe Maschinenbefehle
- Mikroprogrammierter Steuerwerk
- Variables Befehlsformat und Befehlslänge

Instruction Set Architecture

- Ziel: Reduzierung von Instruction Count
 - Großer Befehlssatz + viele Adressierungsarten und Datentypen
= sehr komplexes Steuerwerk
 - Schnellere Hauptspeicher + Cache-Speicher
 - Unterschiedliche Komplexität der Befehle = differierte Ausführungszeiten
 - Schwieriger Compilerbau
 - Nut wenige Befehle häufig benutzt
- **Z.B. Systemprogramme in XPL auf IBM/360 (1964)**
 - Insg. 143 Assemblerbefehle
 - 90 % aller ausgeführten Befehle: 10 verschiedene Befehle
 - 95 % aller ausgeführten Befehle: 21 verschiedene Befehle
 - 99 % aller ausgeführten Befehle: 30 verschiedene Befehle

RISC

Reduced Instruction Set Computer

- Viele Einfache Maschinenbefehle
- Befehle arbeiten auf Registeroperanden
- Lade- und Speicherbefehle greifen auf Speicher zu
- Effizientes Pipelining
- Einheitliches Zeitverhalten (außer Lade-, Speicher- und Verzweigungsbefehle)

Instruction Set Architecture

- Ziel: Reduzierung von Cycles Per Instruction
- Oft benutzte Befehle so schnell wie möglich ausführen (möglichst in einer Taktphase, keine Mikroprogrammierung)
- Operanden eher in Register gespeichert (schneller Zugriff -> schnelle Verarbeitung)
- Load/Store Architektur
- Optimierende Compiler (weniger Befehle -> reduzierte Laufzeit)

CISC vs RISC

CISC	RISC
Komplexe Befehle, Ausführung in mehreren Taktzyklen	Einfache Befehle, Ausführung in einem Taktzyklus
Jeder Befehl kann auf den Speicher zugreifen	Nur Lade- und Speicherbefehle greifen auf den Speicher zu
Wenig Pipelining (früher)	Intensives Pipelining
Befehle werden von einem Mikroprogramm interpretiert	Befehle werden durch festverdrahtete Hardware ausgeführt
Befehlsformat variabler Länge	Alle Befehle mit fester Länge
Die Komplexität liegt im Mikroprogramm	Die Komplexität liegt im Compiler
Einfacher Registersatz	Große / Mehrere Registersätze

	IBM 370/168	CISC VAX 11/780	Xerox Dorado	IBM 801	RISC Berkeley RISC I	Stanford MIPS
Fertigstellungsjahr	1973	1978	1978	1980	1981	1983
Instruktionen	208	303	270	120	31	55
Mikrocodegröße	54k	61k	17k	0	0	0
Instruktionsgröße	2-6 Bytes	2-57 Bytes	1-3 Bytes	4 Bytes	4 Bytes	4 Bytes
Operationsmodell	Reg-Reg Reg-Mem Mem-Mem	Reg-Reg Reg-Mem Mem-Mem	Stack	Reg-Reg	Reg-Reg	Reg-Reg

C

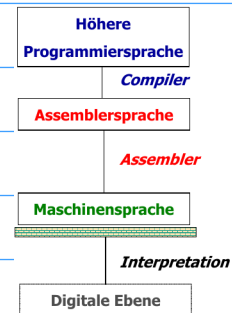
Datentypen:

char = ASCII (8 bit)
 int = (un)signed (16/32 bit)
 float = Einfache Genauigkeit (32 bit)
 double = Zweifache Genauigkeit (64 bit)

- Pass by value

- Größen sind rechnerabhängig

$\text{sizeof(char)} \leq \text{sizeof(short int)} \leq \text{sizeof(int)} \leq \text{sizeof(long int)} \leq \text{sizeof(long long int)}$



```

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

lw $t5, 0($2)
lw $t6, 4($2)
sw $t6, 0($2)
sw $t5, 4($2)
  
```

```

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
  
```

ALUOP[0:3] ← InstReg[9:11] & MASK

Operator Symbol	Operation	Beispiel
~	Bitweise NOT	~x
<<	links Schieben	x << y
>>	rechts Schieben	x >> y
&	Bitweise AND	x & y
^	Bitweise XOR	x ^ y
	Bitweise OR	x y

- Compiler converts source (*.c) to object (*.o) files

- Linker converts object and all includes and creates an executable

- Bibliothek einbinden: #include <stdio.h>

- Modul einbinden: #include "modul.h"

- Globale Textersetzung: #define EYE_COLOR blau

- Variable existiert extern: extern int i;

- Funktionen werden global definiert

- Call by Value: Kopie des Wertes übergeben

- Call by Reference: die Funktion erhält die Speicheradresse der Variablen

Speicherklassen

- auto = default

- register = Hinweis für Compiler, dass es im Register zu finden ist

- static = like global, but local to file; can store state between function calls

- extern = globale Variable in einem anderen Modul

Zeiger und Vektoren

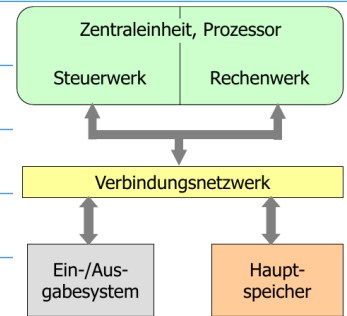
int *p

- p = int Pointer
- *p = int (Dereferenzierung)
- &a = Adresse von a
- (int*) 0x8010 = Type cast: hex int -> int pointer

von Neumann

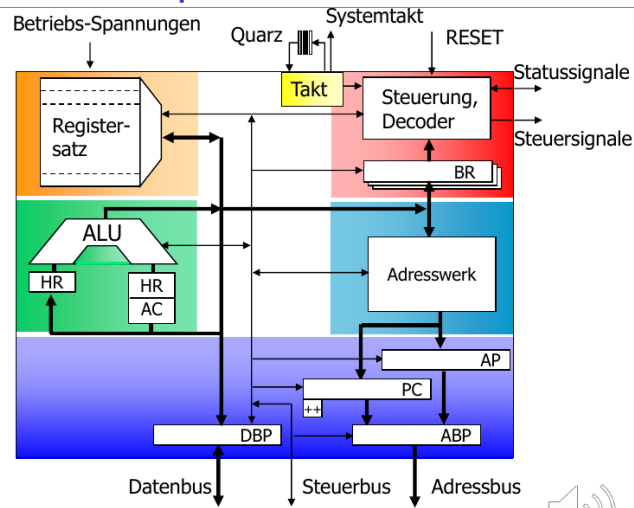
Zentraleinheit (CPU)

- Besteht aus Steuerwerk und Rechenwerk
- Verarbeitet Daten gemäß eines Programms



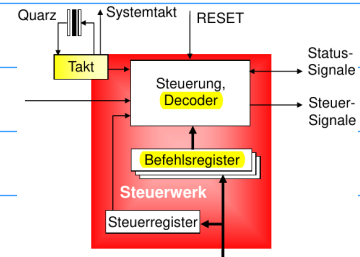
Aufbau eines Mikroprozessors

- **HR:** Hilfsregister
- **AC:** Accumulator
- **BR:** Befehlsregister
- **AP:** Adress Puffer
- **PC:** Program Counter
- **ABP:** Adressbus Puffer
- **DBP:** Datenbus Puffer



Steuerwerk (CU)

- Holt Befehle aus Speicher
- Dekodiert sie
- Steuert ihre Ausführung durch Signale
- CISC = Für jeden Befehl ein Mikroprogramm im Speicher
- RISC = festverdrahtetes Steuerwerk

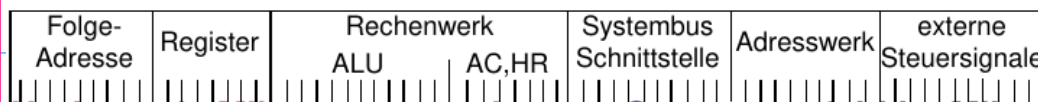


Phasen der Befehlsausführung:

- Holphase: Nächsten Befehl in Befehlsregister laden
- Dekodierphase: Ermitteln der Startadresse des Mikroprogramms
- Ausführungsphase: Befehl wird ausgeführt

Mikroprogrammsteuerwerk

- Mikroprogramm -> Mikrobefehl -> Mikrooperation (einzelne Bits)
- Mikrooperationen = Steuern welche hardware Komponente benötigt werden

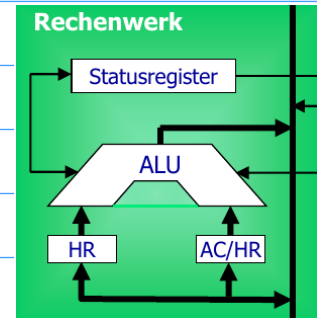


Bits des Steuerregisters:

- Interrupt Enable Bit = ob auf Interrupts reagiert wird
- User/System Bit = ob im Benutzer- oder im Systemmodus ausgeführt wird
- Trace Bit (Single Step Mode) = Debugging
- Decimal Bit = Dual oder BCD

Rechenwerk (ALU)

- Führt Rechneroperationen aus
- Schickt Signale zurück an das Steuerwerk



Register:

- Hilfsregister (HR) = Zwischenspeicher von Operanden
- Akkumulator = ALU Ergebnisse
- Statusregister = Statusinfo bzgl. der Berechnung

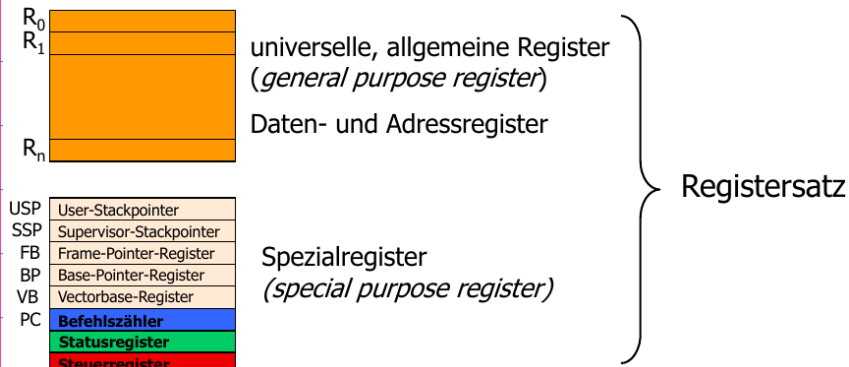
Statusregister Bits (Flags):

- CF (Carry Flag) = Übertrag aus dem höchstwertigsten Bit
- AF (Aux Carry) = Übertrag von Bit 3 un Bit 4 (BCD-Arithmetik)
- ZF (Zero Flag) = Ob das Ergebnis 0 ist
- SF (Sign Flag) = Ob das Ergebnis negativ ist
- Overflow Flag = Overflow im Zweierkomplement
- EV (Even Flag) = Ob das Ergebnis gerade ist
- PF (Parity Flag) = Uneven Parity des Ergebnisses

PSW (Prozessorstatusword) = Statusregister + Steuerregister

Registersatz

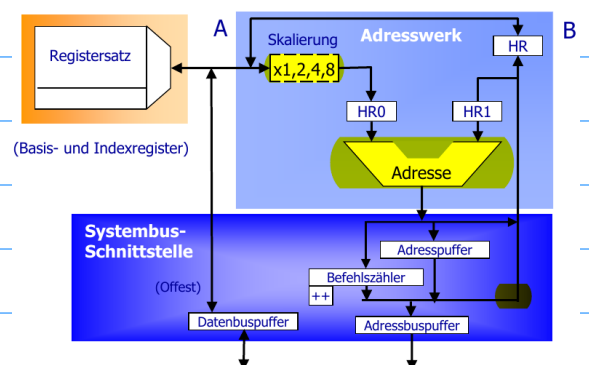
- Zwischenspeicher für häufig benutzte Daten
- Schnellere Zugriff als Hauptspeicher



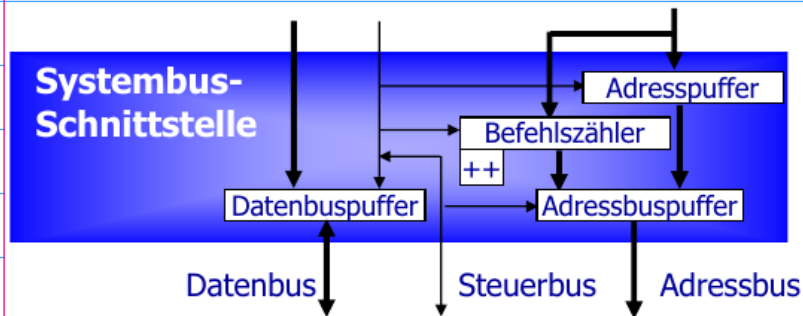
- Heutzutage werden mehrere Datenregister als Akkumulator benutzt

Adresswerk

- Berechnet Adressen (inkl. ALU)
- Skalierung für unterschiedliche Word Größen



Systembusschnittstelle



Verbindungsnetzwerk (Bus)

- Verbindet die Komponenten für den Austausch von Daten
- Transportiert Adressen, Daten und Steuersignale

Hauptspeicher

- Jede Speicherzelle eindeutig durch ihre Adresse identifizierbar
- von Neumann: Speichert Daten und Programme
- Harvard: Programm- und Datenspeicher getrennt

Ein-/Ausgabesystem

- Eingabe von Daten und Programm / Ausgabe der verarbeiteten Daten

Weitere Komponente:

- MMU (Memory Management Unit)
- FPU (Gleitkommaverarbeitung)
- Cache-Speicher

MIMA

- Mikroprogrammierte Minimalmaschine

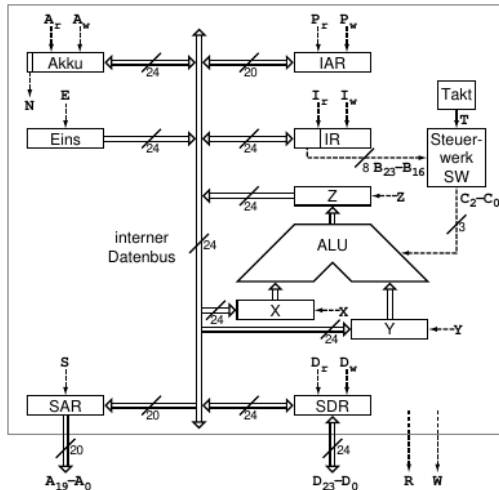
Fetch-Phase:

Mikrobefehlsformat

A _r	A _w	X	Y	Z	E	P _r	P _w	I _r	I _w	D _r	D _w	S	C ₂	C ₁	C ₀	R	W	00	Folgeadresse F	
27			24				20				16					12		9	8	0

- Lesen und schreiben dauert je 3 Takte (deswegen 3 Takte zwischen SAR und SDR)

Architektur der MIMA



Register

Akku: Akkumulator
X: 1. ALU Operand
Y: 2. ALU Operand
Z: ALU Ergebnis
Eins: Konstante 1
IAR: Instruktionsadreßregister
IR: Instruktionsregister
SAR: Speicheradreßregister
SDR: Speicherdatenregister

Steuersignale vom SW

– für den internen Datenbus

A_r: Akku liest
A_w: Akku schreibt
X: X-Register liest
Y: Y-Register liest
Z: Z-Register schreibt
E: Eins-Register schreibt
P_r: IAR liest
P_w: IAR schreibt
I_r: IR liest
I_w: IR schreibt
D_r: SDR liest
D_w: SDR schreibt
S: SAR liest

– für die ALU

C₂-C₀: Operation auswählen

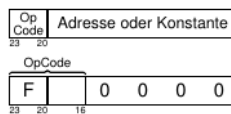
– für den Speicher

R: Leseanforderung
W: Schreib Anforderung

Meladesignale zum SW

T: Takteingang
N: Vorzeichen des Akku
B₂₃-B₁₆: OpCode-Feld im IR

Befehlsformate



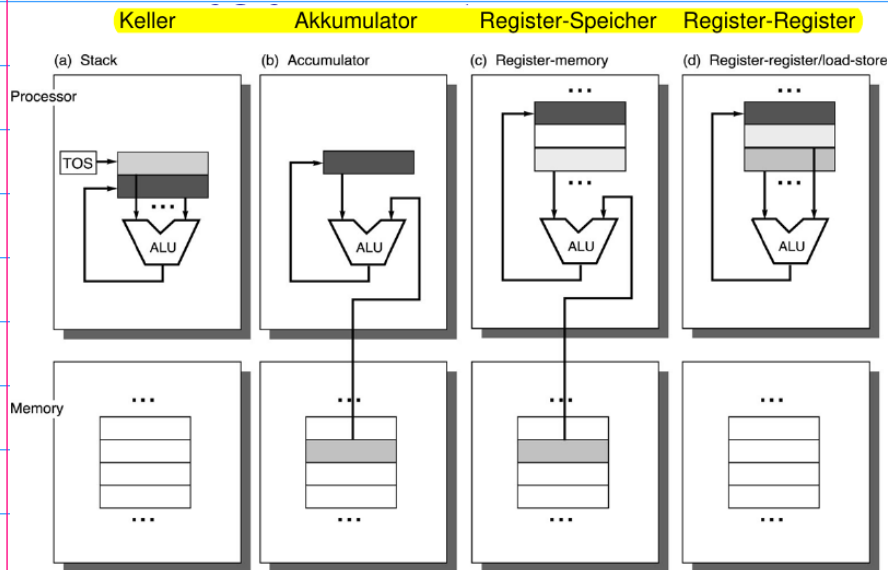
C ₂ C ₁ C ₀	ALU Operation
0 0 0	tue nichts (d.h. Z -> Z)
0 0 1	X + Y -> Z
0 1 0	rotiere X nach rechts -> Z
0 1 1	X AND Y -> Z
1 0 0	X OR Y -> Z
1 0 1	X XOR Y -> Z
1 1 0	Eins-Komplement von X -> Z
1 1 1	falls X = Y, -1 -> Z, sonst 0 -> Z

OpCode	Mnemonic	Beschreibung
0	LDC c	c -> Akku
1	LDV a	<a> -> Akku
2	STV a	Akku -> <a>
3	ADD a	Akku + <a> -> Akku
4	AND a	Akku AND <a> -> Akku
5	OR a	Akku OR <a> -> Akku
6	XOR a	Akku XOR <a> -> Akku
7	EQL a	falls Akku = <a>: -1 -> Akku sonst: 0 -> Akku
8	JMP a	a -> IAR
9	JMN a	falls Akku < 0 : a -> IAR
F0	HALT	stoppt die MIMA
F1	NOT	bilde Eins-Komplement von Akku -> Akku
F2	RAR	rotiere Akku eins nach rechts -> Akku

Befehlssatzarchitektur

- Beschreibt die Attribute und das funktionale Verhalten eines Prozessors
- Spezifikation:
Befehlssatz, Befehlsformat, Datentypen und Datenformate,
Adressierungsarten, Register-/Speichermodelle, Unterbrechungssystem

Architekturklassen (Ausführungsmodelle):



Register-Register-Modell (add R1, R2, R3):

- **Vorteil:**
 - Einfaches und festes Befehlsformat
 - Einfaches Code-Generierungsmodell
 - Etwa gleiche Ausführungszeit der Befehle
- **Nachteil:**
 - Höhere Anzahl von Befehlen im Vergleich zu Architekturen mit Speicherreferenzen
 - Mehr Instruktionen und geringere Befehlsdichte führen zu längeren Programmen
- **Beispiele**
 - ARM, MIPS, PowerPC, SPARC, ...

Register-Speicher-Modell (add R1, Adresse | add Adresse, R1):

- **Vorteile:**
 - Auf die Daten kann ohne vorherige Lade-Operation zugegriffen werden
 - Kodierung im Befehlsformat führt zu höherer Code-Dichte
- **Nachteile:**
 - Operanden können nicht gleich behandelt werden, wenn eine Überdeckung vorliegt
 - Anzahl der Taktzyklen pro Instruktion variiert in Abhängigkeit der Adressrechnung
- **Beispiele:**
 - IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x

Akkumulator-Modell (add Adresse | add R1):

- Ein spezielles Akkumulator-Register wird als Quelle und Ziel verwendet


Keller-Modell (add):

- Operanden befinden sich auf dem Stack

Speicher-Speicher-Modell (add Addr1, Addr2, Addr2):

- Operanden und Ergebnis befinden sich im Speicher

Beispiel ($C=A+B$, $D=C-B$):

Register-Register	Register-Speicher	Akkumulator	Keller
<pre>load Reg1,A load Reg2,B add Reg3,Reg1,Reg2 store C,Reg3 load Reg1,C load Reg2,B sub Reg3,Reg1,Reg2 store D,Reg3</pre>	<pre>load Reg1,A add Reg1,B store C,Reg1 load Reg1,C sub Reg1,B store D,Reg1</pre>	<pre>load A add B store C load C sub B store D</pre>	<pre>push B push A add pop C push B push C sub pop D</pre> 

Datentypen und Datenformate

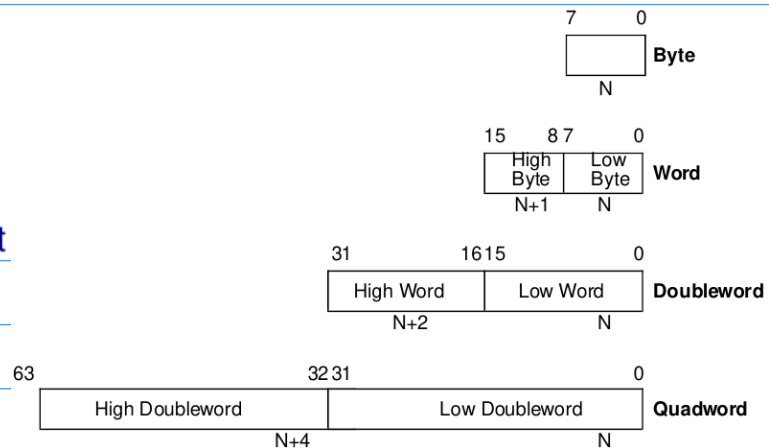
Datentypen:

- Datentypen, die nicht von der Hardware unterstützt werden, müssen auf elementare Datentypen zurückgeführt werden

Datenformate:

■ Standardformate

- Byte: 8 Bit
- Halbwort: 16 Bit
- Wort: 32 Bit
- Doppelwort: 64 Bit

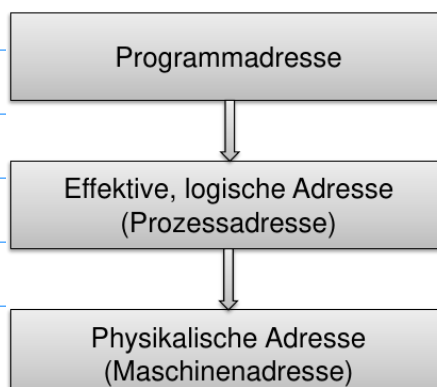


Speicheradressierung

- Byte-adressierbar: direkter Zugriff auf Byte, Halbwort oder Wort. Adressen beziehen sich auf Bytegrenzen

Adressierungsarten

Ablauf der Adressberechnung



Pipelining

Aufbau des DLX- (MIPS-) Prozessors

■ Typ R: Register-Register-Befehle (z.B. add, sub, ...)

opcode	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

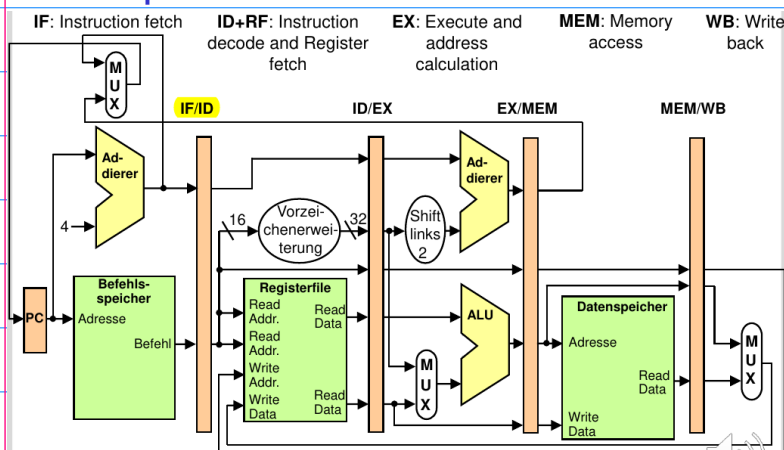
■ Typ I: Immediate-Register Befehle (z.B. addi, lw, beq, ...)

opcode	rs	rt	immediate
31-26	25-21	20-16	15-0

■ Typ J: Jump (z.B. j, jal, ...)

opcode	target
31-26	25-0

DLX-Pipeline:



- IF = Holt das nächste Befehl aus dem Speicher

- ID+RF = Opcode -> CPU Signale, Operanden aus Speicher geholt 2. Takthälfte

- EX/MEM = Operation ausgeführt (Lade-/Speicher- Adresse berechnet)

- MEM = Zugriff auf Speicher
- WB = Ergebnis in Registerfile 1. Takthälfte

- Die Blöcke speichern das Ergebnis jeder Stufe (geholtes/dekodierte Befehl)

- Ergebnisse werden taktsynchron weitergegeben

Leistungsbetrachtung:

- n = Anzahl Befehle, k = Ausführungsstufen

- Sequentiell: $n * k$ Taktzyklen

- Pipeline: $n + (k-1)$ Taktzyklen

- Speedup: $S = \text{Sequentiell} / \text{Pipeline}$

- Pipelineregister zwischen Stufen => Verzögerung:

Länge des Taktzyklus: $t = \max\{t_1, t_2, \dots, t_k\} + t_{reg}$

Datenabhängigkeiten:

```

S1:  add  r1, r2, 2      #   r1 := r2 + 2
S2:  add  r1, r1, r3     #   r1 := r1 + r3
S3:  muli  r3, r1, 3     #   r3 := r1 * 3
S4:  muli  r3, r4, 3     #   r3 := r4 * 3
    
```

Echte Datenabhängigkeit d!: S1→S2, S2→S3 (aber NICHT: S1→S3)

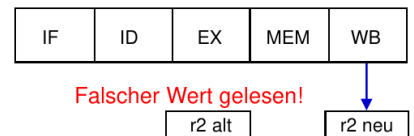
Gegenabhängigkeit d^a: S2→S3, S2→S4

Ausgabeabhängigkeit d^o: S1→S2, S3→S4

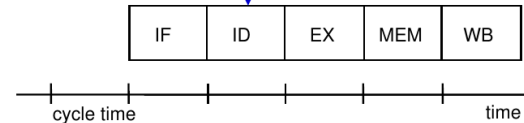
Datenkonflikte:

- Read after Write (durch echte Abhängigkeiten) ->
- Write after Read (durch Gegenabhängigkeit) nur falls Schreibstufe vor Lesestufe
- Write after Write (durch Ausgabeabhängigkeit) nur falls in mehrere Stufen geschrieben wird

add r2, r1, r2

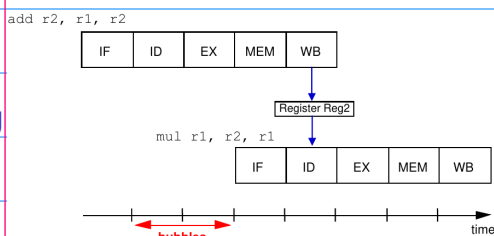


mul r1, r2, r1



Lösungen für Konflikte:

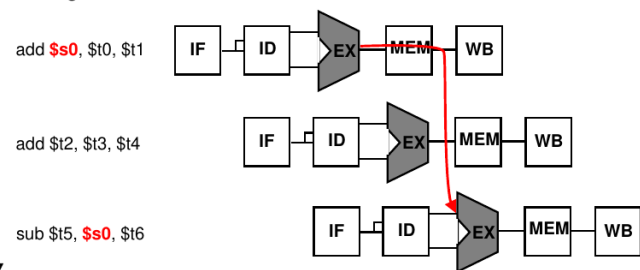
- Leeroperationen (nops) zwischen Befehle mit Konflikte (Software)
- Instruction Scheduling: umordnen der Befehle (Software)
- Pipeline-Sperrung (interlocking) / Pipeline-Leerlauf (stalling): Decode/Fetch Stufe für 2 Takte anhalten (Hardware)



Zyklus 6				
Befehl 4	Befehl 3	mul	-	-

Load Forwarding

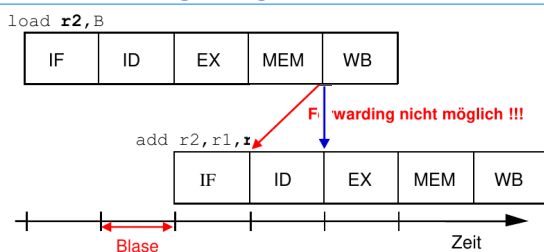
Programm-ausführung



- Forwarding: Ergebnis frühern verfügbar machen (Hardware)

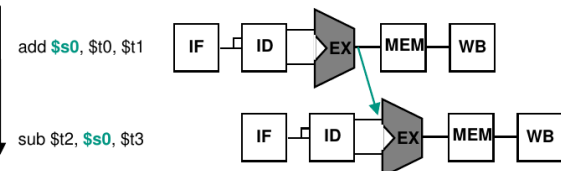
Beides

-->

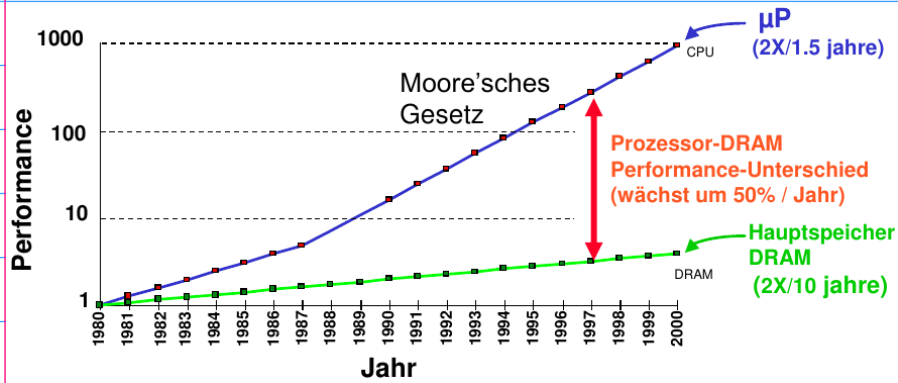


Result forwarding

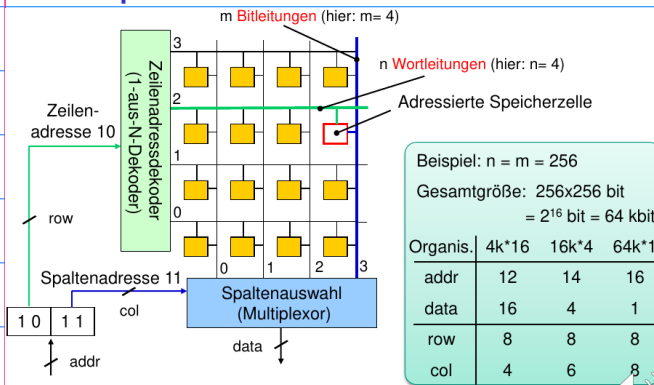
Programm-ausführung



Speicher



Prinzipieller Aufbau:



256x256 = Eigentliche Speicher Matrix

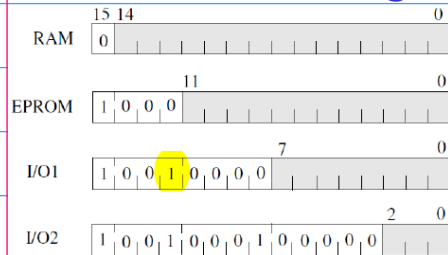
Organisation = adressierbare Zeilen und deren Größe

Speicher Aufgaben

Adressen im Speicherraum:

Anfang	End	—
\$0000	\$7FFF	für den RAM-Bereich (32 Kbyte)
\$8000	\$8FFF	für den EPROM-Bereich (4 Kbyte)
\$9000	\$90FF	für den (I/O)1-Bereich 1 (256 Byte),
\$9100	\$9107	für den (I/O)2-Bereich 2 (8 Byte).

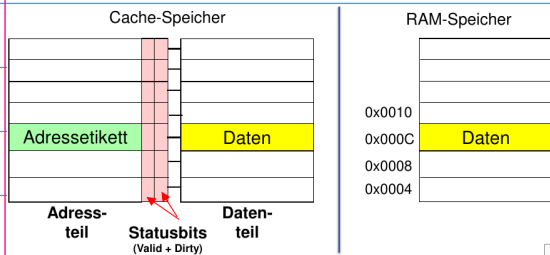
Adressunterteilung:



Speicherorganisation:

1. 256x8-Organisation: 256 Speicherstellen mit 8-Bit Wörter \Rightarrow 256 Speicherstellen müssen adressiert werden. Dazu sind 8 Adressleitungen erforderlich.
2. Es sind 8 RAM-Bausteine der Organisation 4kx4 notwendig, um einen Speicher mit einer Kapazität von 16k Wörter und einer Wortbreite von 8 Bit zu realisieren.
3. ROM-Baustein der Speicherkapazität von 8192 Bits und 7 Adressleitungen
 \Rightarrow Es können 128 Speicherstellen adressiert werden
 \Rightarrow 128x64-Organisation

Cache-Speicher



gegebene Informationen

- Speicherkapazität
- Blockgröße
- Länge der Adresse
- Cache-Typ

indirekte Informationen

- **Cachegröße:** Speicherkapazität / Blockgröße
- Anzahl Sätze: Cachegröße / N
- **Blockauswahl:** $\log_2(\text{Blockgröße})$ Bit
- **Satzauswahl:** $\log_2(\text{Sätze})$ Bit
- **Tag:** Adresse - Blockauswahl - Satzauswahl Bit

Aktualisierungsstrategien:

Cache-Zugriff	Write-Through	Write-Back
Read-Hit	Cache-Datum → CPU	Cache-Datum → CPU
Read-Miss	Ggf. muss im Cache eine Zeile verdrängt werden: einfach invalidieren; HS-Datenblock & Tag → Cache; HS-Datum → CPU; 1 → V	Ggf. muss eine Cache-Zeile verdrängt werden. Falls Dirty: Cache-Zeile → HS ; HS-Datenblock & Tag → Cache; HS-Datum → CPU; 1 → V, 0 → D
Write-Hit	CPU-Datum → Cache & HS	CPU-Datum → Cache 1 → D
Write-Miss	CPU-Datum → HS (ggf. auch in Cache)	Ggf. muss eine Cache-Zeile verdrängt werden. Falls Dirty: Cache-Zeile → HS ; HS-Datenblock & Tag → Cache; 1 → V; CPU-Datum → Cache; 1 → D

- Hit-Rate = Treffer / Zugriffe

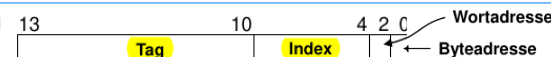
- Zugriffszeit =
 $(\text{Hit-Rate} * \text{Hit-Zeit}) +$
 $(1 - \text{Hit-Rate}) * \text{Miss-Zeit}$

- Hit-Zeit = Zugriff auf Cache

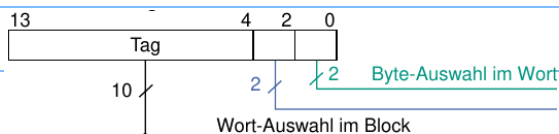
- Miss-Zeit = Zugriff auf Speicher

Cache-Organisationsformen:

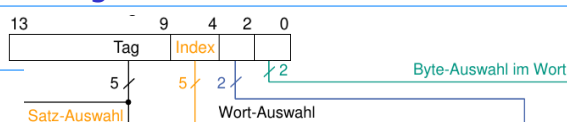
- Direct-Mapped Cache: Block im Cache -> eindeutiger & fester Platz im Cache:
 $\text{Cache-Zeile} = \text{Blockadresse} \% \text{Anzahl Cache-Zeilen}$



- Voll-Assoziativer Cache: Ein Block kann an beliebiger Stelle abgelegt werden:



- N-weg Satzassoziativer Cache: mehrere Zeilen -> ein Satz (Wege innerhalb Sätze):

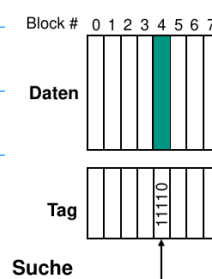


- **Beispiel: Cache mit 8 Cache Zeilen mit jeweils 16 Byte**

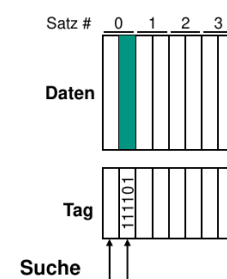
- Zugriff auf Adresse 0xF40 = 0b1111 0100 0000

Wort Addr. → Byte Addr.

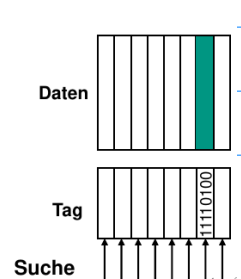
Direct-mapped



2-Wege Satz-Assoziativ



Voll-Assoziativ



Cache Aufgaben

Einfacher Beispiel

Aufgabe

Gegeben seien ein direkt-abgebildeter Cache (**direct-mapped**), ein 2-fach satzassoziativer Cache (**2-way- set-associativ**) und ein vollassoziativer Cache (fully-associativ). Die drei Cachespeicher haben jeweils eine **Speicherkapazität von 64 Byte** und werden in **Blöcken von je 8 Byte** geladen. Die **Hauptspeicher- adresse umfasst 32 Bits**. Falls notwendig, wird die **Least Resently Used** -Ersetzungsstrategie LRU verwendet.

Länge des Tag-Feldes und Anzahl der Vergleicher:

Cache	Länge des Tag-Feldes	Anzahl der Vergleicher
AV	29	8
DM	26	1
A2	27	2

„-“ für Cache-Miss und „x“ für Cache-Hit:

Adresse:	\$12	\$8A	\$9A	\$6C	\$34	\$54	\$68	\$FE	\$17
AV	-	-	-	-	-	-	x	-	x
DM	-	-	-	-	-	-	x	-	-
A2	-	-	-	-	-	-	x	-	-

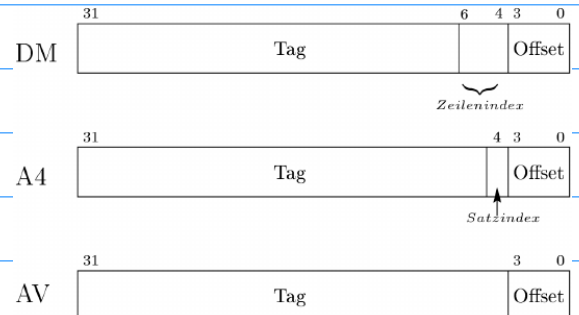
- Adressen in binary konvertieren,
Cache aufmalen, tag/index/usw.

Noch ein Beispiel: Tut08, Klausur/SS14

Cache + Speicherverbrauch:

Gegeben seien ein direkt-abgebildeter Cache (direct mapped, Abkürzung: **DM**), ein 4-fach satzassoziativer Cache (4-way-set-associativ, Abkürzung: **A4**) und ein vollassoziativer Cache (fully-associativ, Abkürzung: **AV**). Die drei Cache-Speicher haben jeweils eine **Speicherkapazität von 128 Byte** und werden in **Blöcken von je 16 Byte** geladen. Die **Hauptspeicheradresse umfasst 32 Bits**. Falls notwendig, wird die **Least Resently Used**- Ersetzungsstrategie verwendet.

Welche Bits der 32-Bit-Adresse bilden Offset, Tag und Index? Skizzieren Sie hierzu die Unterteilung der Hauptspeicheradresse für die drei Cache-Speicher.



Cache	Benötigter Speicherplatz
DM	$(32-7+2) \cdot 8 \text{ Bits} = 27 \text{ Byte}$
A4	$(32-5+2) \cdot 8 \text{ Bits} = 29 \text{ Byte}$
AV	$(32-4+2) \cdot 8 \text{ Bits} = 30 \text{ Byte}$

Der Zustand eines Cacheblocks wird durch zwei Statusbits (**Valid-Bit** und **Dirty-Bit**) gekennzeichnet. Wieviel Speicherplatz wird insgesamt für die Realisierung des Tag-Speichers der einzelnen Cache- Speicher benötigt?

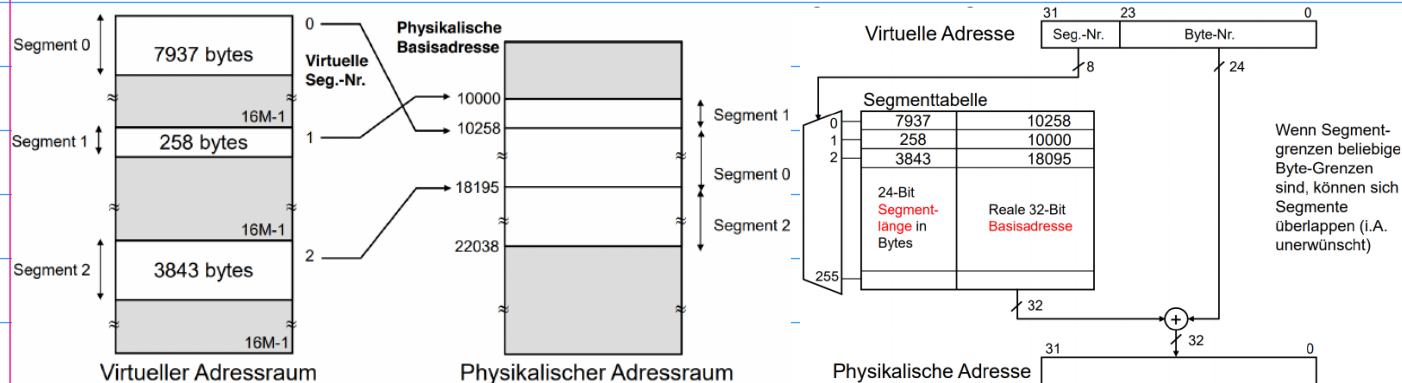
Adresse:	\$2D	\$38	\$9E	\$D4	\$19	\$29	\$3E	\$9D	\$CA
DM	-	-	-	-	-	x	x	-	-
A4	-	-	-	-	-	x	x	x	-
AV	-	-	-	-	-	x	x	x	-

Noch ein Beispiel: Tut09, Aufgabe 2.1

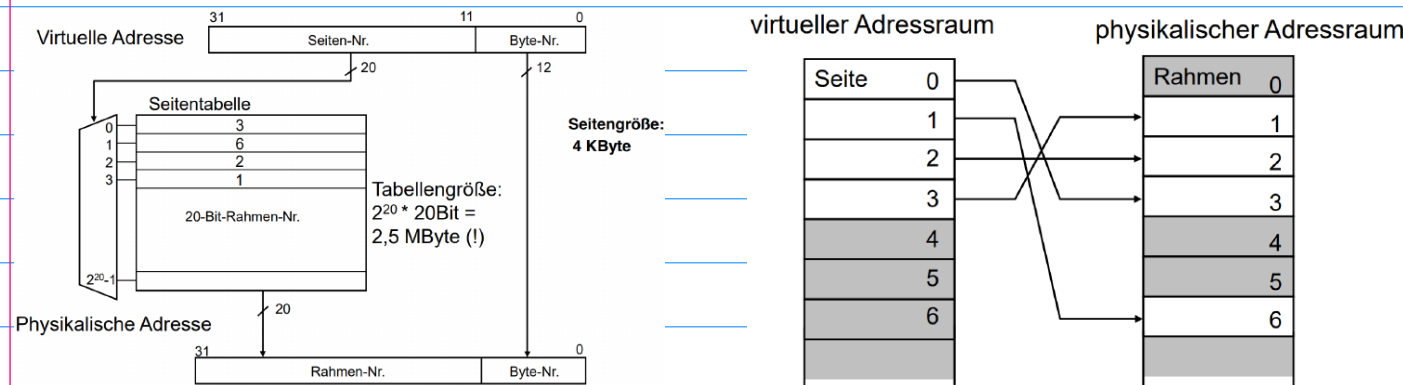
Weitere Beispiele: Tut10, Alle Aufgaben

Speicherverwaltung

Segmentierungsverfahren:



Seitenwechselverfahren:



Informationen im Segment-Deskriptor:

- Informationen darüber, ob das Segment gültig ist und wieviele Seiten es umfasst.
- Welche Seitentabellen im Hauptspeicher präsent sind.
- Zeiger auf eine Seitentabelle, welche die Deskriptoren aller zu diesem Segment gehörenden Seiten und die Rahmennummer der Seiten enthält.

Vorteile bzw. Nachteile einer zweistufigen Adressumsetzung:

Vorteil: Bei einer reinen Seitenverwaltung hat man eine einzige, sehr große Seitentabelle, die im Hauptspeicher entsprechend viel Platz beansprucht. Im Gegensatz dazu gibt es bei einer zweistufigen Adressumsetzung viele, jedoch kleinere Seitentabellen, von denen nur die aktuelle im Hauptspeicher (neben der Segmenttabelle) gehalten werden muss. Die anderen Seitentabellen können in einem Hintergrundspeicher stehen und müssen dann bei Bedarf geladen werden.

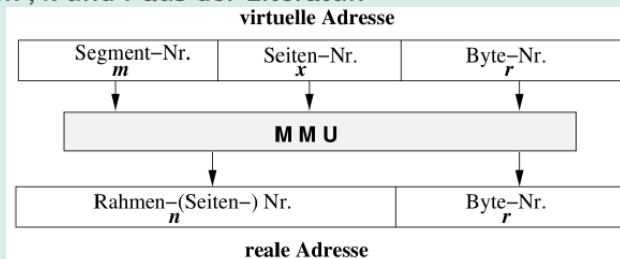
Nachteil: Zweistufige Adressbildung.

Virt. Speicher Aufgaben

Seiten und Segmente:

Die Speicherverwaltung in einem Rechnersystem geschieht zweistufig über eine Segmenttabelle und eine Seitentabelle. Die Unterteilung der virtuellen und der physikalischen Adresse ist unten dargestellt.

Hinweis: Zur Lösung dieser Aufgabe verwenden Sie bitte gängige Werte für m , x und r aus der Literatur.



$m=10$ Bit; $x=10$ Bit; $r=12$ Bit
 \Rightarrow Adresse = 32 Bit

Anzahl der Seiten pro Segment:

$2^{10} = 1024$ Seiten 0...1023

Größe einer Seite in Byte:

2^{12} Byte = 4 Kbyte

Größe des maximal verfügbaren virtuellen Adressraums in Byte:

2^{32} Byte = 4 GByte 0...(2³² - 1)

Anzahl der Segmente im virtuellen Adressraum:

$2^{10} = 1024$ Segmente 0...1023

Memory Tabellen und Adressen:

Gegeben sei eine Speicherverwaltungseinheit (MMU). Der virtuelle Speicher ist in 8 Seiten mit je 1 KByte unterteilt. Der physikalische Speicher hat eine Kapazität von 4 KByte. Der aktuelle Ausschnitt der Seitentabelle ist in Tabelle 1 angegeben.

Virtuelle Seitennummer	Physikalische Seitennummer
0	-
1	-
2	1
3	3
4	-
5	0
6	2
7	-

Seitengröße: **1 KByte**

virtuelle Seiten: **8**

physikalische Speicherkapazität: **4 KByte**

Unterteilung der virtuellen Adresse:

31	9	0
Virtuelle Seitennummer	Bytenummer	

Virtuelle Adresse	Physikalische Adresse
2100	1076 (0x434)
4095	4095 (0xFFFF)
5620	500 (0x1F4)
6200	2104 (0x838)
1023	page fault

For 2100:

bin = 10 11111111

vPage = 2 (first 2 bits) \Rightarrow pPage = 1

Byte = 52 (last 10 bits)

pAdress = 1 * 1024 + 52

Eine Beschleunigung der Adressumsetzung durch den TLB wird beim **zweiten Zugriff** auf eine Seite und solange die entsprechenden Einträge aus dem Seitentabellen-Verzeichnis und der Seitentabelle aus dem TLB **nicht verdrängt** wurden.

Weiteres Beispiel: Tut11, Aufgabe 3