

I. GRUNDLAGEN

Aufgaben der Hardware:

Ein- und Ausgabe von Daten
Verarbeiten von Daten
Speichern von Daten

Klassische Hardwarekomponenten:

Ein- und Ausgabe
Hauptspeicher
Rechenwerk
Leitwerk

II. ANFORDERUNGEN HÖHERER PROGRAMMIERSPRACHEN

Begriffe:

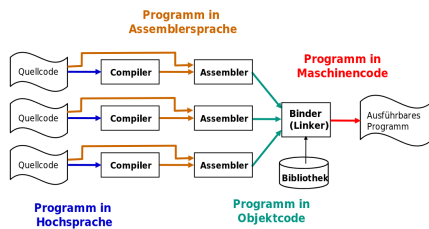
Maschinensprache: Für Prozessor verständliche Anweisungsrepräsentation, z.B. 00101101001110101

Assemblersprache: Für Menschen verständliche Maschinensprache, z.B. add \$s2, \$s1, \$s0

Assembler: Übersetzt Assemblersprache eindeutig in Maschinensprache

Objektcode: Maschinenprogramm mit ungelösten externen Referenzen

Binder/Linker: Löst ungelöste Referenzen auf, verbindet alles zu einem ausführbaren Maschinenprogramm



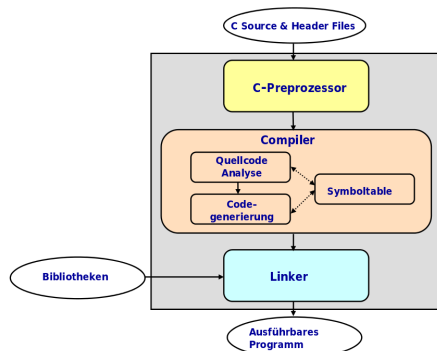
Programmiersprache C:

Zwischenstellung zwischen Assembler und Hochsprache
hohe Portabilität trotz guter Architekturanpassung
einfache Programmierung

Datentypen: char, int, float, double

Kontrollstrukturen: Entscheidungen, Schleifen, Blöcke, Unterprogramme

Zeiger als Parameter möglich



C - Datentypen:

char: Ein Zeichen, meist 1 Byte

int: Integerzahl, 2 oder 4 Byte

float: Gleitkommazahl, meist 4 Byte

double: Gleitkommazahl, meist 8 Byte

C - Operatoren:

*: Multiplikation ($x*y$)

/: Division (x/y)

%: Modulo ($x\%y$)

+: Addition ($x+y$)

-: Subtraktion ($x-y$)

+ und - auch als Prä- und Postfix, alle auch als assign (= anhängen)

C - Bit-Operatoren:

~: Bitweise NOT ($\sim x$)

<<: links schieben ($x<<y$)

>>: rechts schieben ($x>>y$)

&: bitweise AND ($x\&y$)

^: bitweise XOR ($x\^y$)

|: bitweise OR ($x|y$)

alle auch als Assign (= anhängen)

C - Vergleichsoperatoren:

>, <: größer, kleiner als ($x>y$, $x<y$)

>=, <=: größergleich, kleinergleich als ($x>=y$, $x<=y$)

==, !=: gleich, ungleich ($x==y$, $x!=y$)

C - Spezialoperatoren:

Auswahloperator: $z = (a < b) ? a : b$ ($z=a$, falls $a<b$, sonst $z=b$)

C - Operatoren-Priorität

Operator Type	Operator	Associativity
Primary Expression Operators	() [] . -> $expr++$ $expr--$	left-to-right
Unary Operators	* & + - ! ~ ++ $expr$ -- $expr$ (typecast) sizeof	right-to-left
Binary Operators	* / %	left-to-right
	+ -	
	>> <<	
	< > <= >=	
	== !=	
	&	
	^	
	&&	
Ternary Operator	?:	right-to-left
Assignment Operators	= += -= *= /= %= >>= <<= &= ^= =	right-to-left
Comma	,	left-to-right

C - Kontrollstrukturen

```
if (Bedingung) { Aktionen_if } else { Aktionen_else }
switch (var) { case a: ... break; ... default: ... break; }
while (Bedingung) { ... }
for (init; Bedingung; reinit) { ... }
do { ... } while (Bedingung)
```

C - Programmaufbau

1. Präprozessor-Anweisungen:

- (a) `#include <stdio.h>` (Bibliotheken einbinden)
- (b) `#include "modul.h"` (Module einbinden)
- (c) `#define COLOR blau` (Globale Textersetzung)

2. Globale Deklarationen/Definitionen:

- (a) `int i;` (Deklaration)
- (b) `int j = 13;` (Definition)
- (c) `int fakultaet (int n);` (Funktionsprototyp)

3. Funktionen/Programmstruktur

```
int fakultaet (int n) { ... }
jedes Programm enthält Funktion void main(...) { ... }
Unterprogramm = Funktion
Programmstart: main wird aufgerufen
Rekursion ist zulässig
```

C - Parameterübergabe

1. Call by Value: Normalfall, Kopie des Parameters wird an Funktion übergeben, bei Änderung keine Auswirkung beim Aufrufer
2. Call by Reference: Mit Zeigern umsetzbar, selbe Speicheradresse wie Aufrufer

C - globale und lokale Variablen

Global: Sind gesamtem Programm bekannt (zu vermeiden)

Lokal: Nur in Block deklariert

C - Speicherklassen

auto: lokale Variablen

register: wird in CPU-Register gespeichert, nur für zeitkritische Variablen zu verwenden

static: statischer Speicherplatz

extern: globale Variable

C - Zeiger und Vektoren

Pointer: Enthält Adresse, die auf Daten verweist

`int* p` (p ist Zeiger auf int)

`a = 3; p = &a` (p enthält Adresse von a)

`int b = *p + 1` (=4)

	Adresse	Inhalt
p	...	0x8004
a	0x8004	3
b	...	4
q	...	0x8010
	0x8010	

```
int *p;
int *q;
int a = 3;
int b;

p = &a;
b = *p + 1;
q = (int*) 0x8010
```

III. ZAHLENDARSTELLUNG

Zahlensysteme – Stellenwertsystem

Darstellung einer Zahl durch Ziffern z_i – Stellenwert i te Position:
ite Potenz der Basis b

Wert $X_b = \sum_{i=-m}^n z_i b^i$

Wichtige Zahlensysteme: Dual-, Oktal-, Dezimal-, Hexadezimalsystem

	Dual	Oktal	Dezimal	Sedezimal
0	0	0	0	0
1	1	1	1	1
2		2	2	2
3		3	3	3
4		4	4	4
5		5	5	5
6		6	6	6
7		7	7	7
8			8	8
9			9	9
10				A
11				B
12				C
13				D
14				E
15				F

Umwandlung von Dezimal zu Basis b

1. euklidischer Algorithmus:

- (a) Berechne p mit $b^p \leq Z < b^{p+1}$, setze $i = p$
- (b) Berechne $y_i = Z_i \text{ div } b^i$, $R_i = Z_i \text{ mod } b^i$
- (c) Wiederhole (b)H für $i = p - 1, \dots$, ersetze dabei Z durch R_i , bis $R_i = 0$ oder b^i klein genug ist

$$2^3 \leq 13 < 2^4$$

$$13 : 2^3 = 1 \text{ Rest } 5$$

$$5 : 2^2 = 1 \text{ Rest } 1$$

$$1 : 2^1 = 0 \text{ Rest } 1$$

$$1 : 2^0 = 1 \text{ Rest } 0$$

$$\rightsquigarrow Z = 13_{10} = 1101_2$$

2. Horner-Schema:

- (a) ganzzahliger Teil: 15741_{10} in Hexadezimal:

$$15741_{10} : 16 = 983 \text{ Rest } 13 (= D_{16})$$

$$983_{10} : 16 = 61 \text{ Rest } 7 (= 7_{16})$$

$$61_{10} : 16 = 3 \text{ Rest } 13 (= D_{16})$$

$$3_{10} : 16 = 0 \text{ Rest } 3 (= 3_{16})$$

$$\rightsquigarrow Z = 15741_{10} = 3D7D_{16}$$

- (b) Nachkommateil: $0,233_{10}$ in Hexadezimal:

$$0,233_{10} * 16 = 3,728$$

$$0,728_{10} * 16 = 11,648$$

$$0,648_{10} * 16 = 10,368$$

$$0,368_{10} * 16 = 5,888$$

$$\rightsquigarrow Z = 0,233_{10} \approx 0,3BA5_{16}$$

Umwandlung Basis b zu Dezimal

Einzelne Stellen nach Stellenwertgleichung addieren

$$101101, 1101_2 =$$

$$2^{-4} + 2^{-2} + 2^{-1} + 2^0 + 2^2 + 2^3 + 2^5$$

$$= 45,8125_{10}$$

Umwandlung Basis b_1 zu Basis b_2

1. Umwandlung über Dezimalsystem
2. Ist eine Basis Potenz der anderen, so können mehrere Stellen zu einer Ziffer zusammengefasst werden

$$0110100, 110101_2 = 0011 \ 0100, 1101 \ 0100 = 34, D4_{16}$$

Darstellung negativer Zahlen

1. Betrag und Vorzeichen: Erstes Bit von Links ist Vorzeichen, Rest ist Betrag (0001 0010 = 18, 1001 0010 = -18)

Vorteile: Symmetrischer Zahlenbereich

Nachteile: Darstellungsänderung bei Bereichserweiterung, gesonderte Vorzeichenbehandlung bei Addition und Subtraktion, doppelte Darstellung der Null

2. Einerkomplement: Negative Zahl = NOT(positive Zahl)

0000 = 0 1111 = -0
0001 = 1 1110 = -1
0010 = 2 1101 = -2
0011 = 3 1100 = -3
...

Vorteile: Symmetrischer Zahlenbereich, keine gesonderte Betrachtung des ersten Bits

Nachteile: doppelte Darstellung der Null

3. Zweierkomplement: = Einerkomplement + 1

0000 = 0 1111 = -1
0001 = 1 1110 = -2
0010 = 2 1101 = -3
0011 = 3 1100 = -4
...

Vorteile: Wie Einerkomplement, eindeutige Null

Nachteile: Asymmetrischer Zahlenbereich (eine negative Zahl mehr)

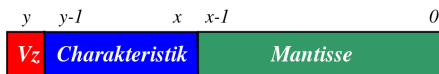
4. Exzess-Darstellung: Verschiebung nach oben derart, dass kleinste negative Zahl die Darstellung 0...0 hat

Darstellung von Kommazahlen

1. Festkommazahlen: Komma sitzt an einer festen Stelle
2. Gleitkommazahlen: $X = \pm \text{Mantisse} * b^{\text{Exponent}}$ (b fest)

$$X = (-1)^{\text{Vorzeichen}} * (0, \text{Mantisse}) * b^{\text{Exponent}}$$

$$\text{Exponent} = \text{Charakteristik} - b^{(y-1)-x}$$

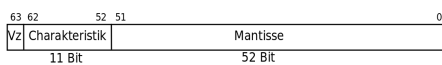


3. IEEE-Standard:

(a) 32-Bit:



(b) 64-Bit:



Codierungen

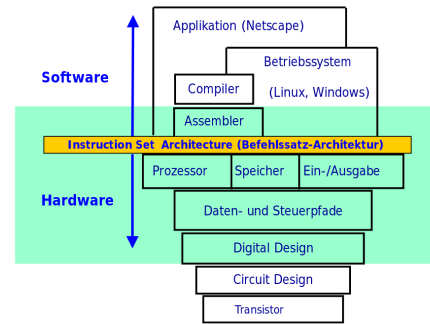
1. BCD: Dezimalzahl ziffernweise als Binärzahl (= *Tetrad*) codieren:

$$8127 = 1000\ 0001\ 0010\ 0111$$

Nachteil: Verbraucht viel Speicher, ungeschickt zum Rechnen

2. ASCII: 7-Bit-Codierung zur Textdarstellung
3. Unicode: Weltweit genormte Codierung aller Zeichen (wegen der vielen inkompatiblen ASCII-Derivaten)

IV. BEFEHLSATZARCHITEKTUR



ISA – Aufgaben

Wie werden Daten repräsentiert?

Wo werden Daten gespeichert?

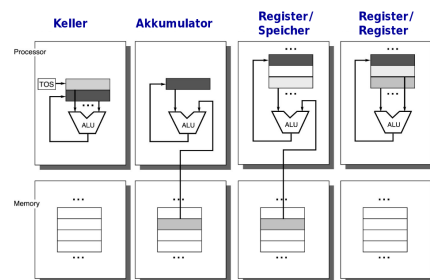
Welche Operationen können auf den Daten ausgeführt werden?

Wie werden die Befehle codiert?

Wie wird auf die Daten zugegriffen?

~ abstrahiert Hardware für den Maschinenprogrammierer

Ausführungsmodelle



Ausführungsmodell – Register-Register

Alle Operanden und Ergebnis stehen in Allzweckregistern

Load/Store: Bestimmte Befehle holen Operanden aus Hauptspeicher, schreiben Inhalte von Registern in Speicher

Dreiadressformat

```
load R2,A   R2<-mem[A]
load R3,B   R3<-mem[B]
add R1,R2,R3 R1<-R2+R3
store C,R1  mem[C]<-R1
```

Vorteile: Einfaches und festes Befehlsformat, einfaches Code-Generierungsmodell, etwa gleiche Ausführungszeit der Befehle

Nachteile: Höhere Anzahl von Befehlen im Vergleich zu Architekturen mit Speicherreferenzen, längere Programme

Ausführungsmodell – Register-Speicher

Ein Operand im Speicher, ein Operand im Register, Ergebnis in Speicher oder Register

Explizite Adressierung mit/ohne Überdeckung

Zweiadressformat

```
add A,R1  mem[A]<-mem[A]+R1
add R1,A  R1<-R1+mem[A]
```

Vorteile: Zugriff auf Daten ohne vorherige Ladeoperationen, Befehlsformat-Kodierung ~ höhere Code-Dichte

Nachteile: Keine gleiche Operanden-Behandlung bei Überdeckungen, Taktzyklen pro Instruktion von Adressrechnung abhängig

Ausführungsmodell – Akkumulator-Register

Akkumulator: Ausgezeichnetes Register, dient als Quelle eines Operanden und als Ziel für das Resultat (zweistellige Operationen)

Implizite und überdeckte Adressierung

Spezielle Befehle ermöglichen Operanden-Transport

Einadressformat

```
add A  acc<-acc+mem[A]
addx A acc<-acc+mem[A+x]
add R1 acc<-acc+R1
```

Ausführungsmodell – Keller

Operanden einer zweistelligen Operation stehen auf den obersten zwei Kellerelementen

Ergebnis wird auf Keller abgelegt

Implizite Adressierung über Kellerzeiger (tos)

Überdeckung

Nulladressformat

```
add tos<-tos+next
```

Ausführungsmodelle – Übersicht

C=A+B; D=C-B

Register-Register	Register-Speicher	Akkumulator	Keller
load Reg1,A load Reg2,B Add Reg1,Reg1,Reg2 store C,Reg1 load Reg1,C load Reg1,C sub Reg2,B sub Reg3,Reg1,Reg2 store D,Reg3	load Reg1,A add Reg1,B store C,Reg1 load Reg1,C sub Reg1,B store D,Reg1	load A add B store C load C sub B store D	push B push A add pop C push B push C sub pop D

Architektur – Datentypen

= Datenformat + inhaltliche Interpretation

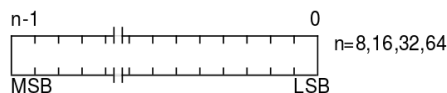
Alternative: Datentyparchitektur (Daten führen Typinformation mit sich)

Datentyp nicht von Hardware unterstützt \leadsto Programm muss Datentyp auf elementare Datentypen zurückführen

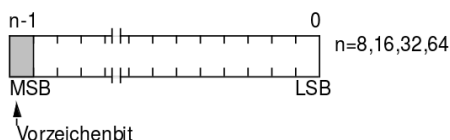
Standardformate:

1. Byte: 8 Bit
2. Halbwort: 16 Bit
3. Wort: 32 Bit
4. Doppelwort: 64 Bit

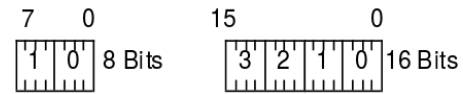
Vorzeichenlose Dualzahl:



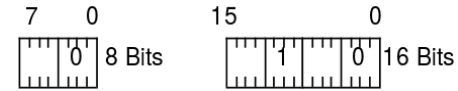
2er-Komplement (signed Integer):



BCD (gepackt): ein Halbbyte codiert eine Zahl

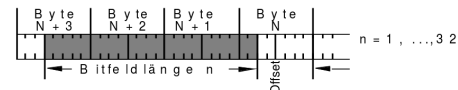


BCD (ungepackt): ein Byte codiert eine Zahl



Gleitkommazahlen: Siehe IEEE-Standard oben

Bitfeld: Darstellung/Verarbeitung von Bitvektoren, vorzeichenlosen Dualzahlen, Zweierkomplementzahlen - Darstellung durch Byte-Adresse und Bitfeld-Offset



String: Aufeinanderfolgend gespeicherte Bytes, enthalten meist ASCII-Zeichen

Speicheradressierung – Datenzugriff

1. Byte-adressierbarer Speicher: Jedes Byte ist über eine bestimmte Adresse adressierbar
2. Wort-organisierter Speicher: Zugriffsbreite = Datenbusbreite (32/64/... Bit)

Speicheradressierung – Alignment

Data Alignment: Datum (s Bytes) ist ausgerichtet abgelegt \Leftrightarrow seine Adresse A ist derart, dass $A \bmod s = 0$

Data Misalignment: Daten an beliebigen Adressen gespeichert

Vorteile: Lückenlose Speichernutzung

Nachteile: zusätzliche Speicherzugriffe nötig

Little Endian: Niedrigstwertigstes Byte an der niedrigsten Adresse

Big Endian: Niedrigstwertigstes Byte an der höchsten Adresse

Speicheradressierung – Adressierungsarten

1. Programmadresse: Im Programm vorliegende Adressen (Prozessor erzeugt aus Programmadressen Prozessadressen mittels Indexmodifikation/Substitution/relativer Adressierung/offener Basisadressierung)
2. Prozessadresse (effektive Adresse): Vom Prozessor verwendet (Prozessor erzeugt nach OS-Angaben aus Prozessadressen Maschinenadressen mittels verdeckter Basisadressierung/Seitenadressierung) - Grund: beliebige Lage des Programms und seiner Werte, partielle Lagerung im Speicher
3. Maschinenadresse: Vom Prozessor gegenüber Hauptspeicher verwendet

Instruction Set

legt Grundoperationen eines Prozessors fest

Befehlsarten:

1. Transport
2. Arithmetik/Logik
3. Schieben/Rotieren
4. Multimedia
5. Gleitkomma
6. Programmsteuerung
7. Systemsteuerung
8. Synchronisation

Instruction Set – Formate

Befehlsformat legt Befehlscodierung fest

Befehlscodierung: [opcode] [parameter1] ...

Adressformate: vier Befehlssatzklassen:

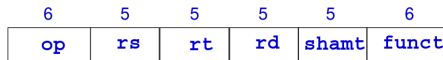
1. Dreiadressformat: [opcode] [dest] [src1] [src2]
2. Zweiadressformat: [opcode] [dest/src1] [src2]
3. Einadressformat: [opcode] [src]
4. Nulladressformat: [opcode]

Instruction Set – MIPS-Prozessor

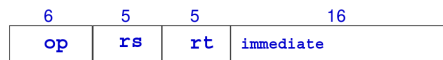
Alle Befehle 32 Bit lang

Befehlstypen:

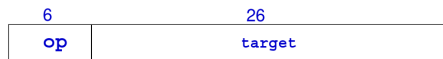
1. **Typ R**: Register-Register-Befehle



2. **Typ I**: Lade-/Speicher-Befehle



3. **Typ J**: Sprungbefehle



Abkürzungen:

I	Immediate (direkt)
J	Jump (Sprung)
R	Register
op	6 Bit, Opcode des Befehls
rs	5 Bit, Kodierung eines Quellenregisters/Zielregisters
immediate	16 Bit, unmittelbarer Wert/Adressverschiebung
target	26 Bit, Sprungadresse
rd	5 Bit, Kodierung des Zielregisters
shamt	5 Bit, Größe einer Verschiebung (shift amount)
funct	6 Bit, Codierung der Funktion (function)

Adressierung – Berechnung

Adressierungsarten: Verschiedene Möglichkeiten, die Adresse eines Operanden/Sprungziels zu berechnen

Früher: Adressen in Befehlen absolut vorgegeben \rightsquigarrow Programm Lageabhängig

Heute: *dynamische Adressberechnung*:

Programmadresse \rightsquigarrow logische Adresse \rightsquigarrow physikalische Adresse

Adressierungsarten

1. Register-Adressierung

- (a) implizit: Flag
- (b) explizit

2. einstufige Speicher-Adressierung

- (a) unmittelbar
- (b) direkt: absolut, Zero-Page, Seiten(-Register)
- (c) Register-indirekt
- (d) indiziert: Speicher-relativ, Register-relativ, Register-relativ mit Index
- (e) Programmzähler-relativ

3. zweistufige Speicher-Adressierung

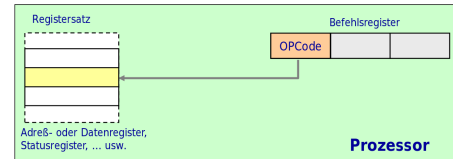
- (a) indirekt absolut
- (b) indirekt Register-absolut
- (c) indirekt indiziert: Speicher-relativ, Register-relativ, Register-relativ mit Index
- (d) indiziert indirekt
- (e) indirekt Programmzähler-relativ

Adressierungsarten – Register-Adressierung

Operand steht bereits im Register \rightsquigarrow kein Speicherzugriff nötig

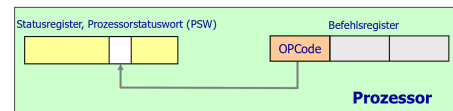
Implizite Adressierung: Nummer des Registers ist im Opcode-Feld codiert enthalten

Beispiel: LSRA (Verschiebe Akkumulatorinhalt eine Bitposition nach rechts)



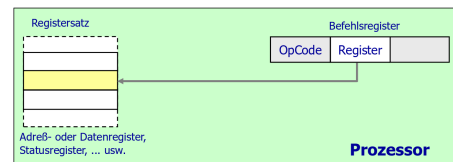
Flag-Adressierung: Spezialfall der impliziten Adressierung: Nur ein Bit (=Flag) wird im Register angesprochen

Beispiel: SEI/CLI (set/clear interrupt flag)



Explizite Adressierung: Registernummer wird im Operandenfeld angegeben

Beispiel: DEC R0 (Dekrementiere Register R0)

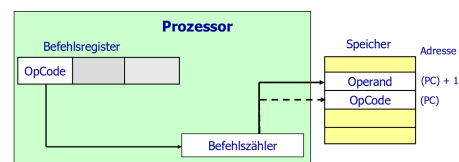


Adressierungsarten – Einstufige Speicher-Adressierung

Eine Adressberechnung ist zur Ermittlung der effektiven Adresse nötig \rightsquigarrow keine mehrfachen Speicherzugriffe zur Adressermittlung

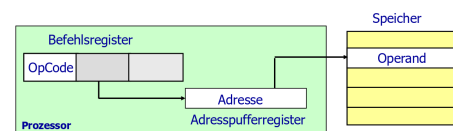
Unmittelbare Adressierung: Befehl enthält nicht Adresse des Operanden, sondern Operand selbst (Opcode und Operand stehen hintereinander im Speicher)

Beispiel: LDA #A3 (Lade Akkumulator mit Sedezimalwert \$A3)



Direkte Adressierung: Befehl enthält nach Opcode logische Adresse des Operanden, aber keine Vorschriften zur Manipulation

1. **Absolute Adressierung**: Speicherwort enthält vollständige (absolute) Operandenadresse
Beispiel: JMP \$07FE (Springe zur Adresse \$07FE)



2. **Seitenadressierung**: Im Befehl nur Kurz-Adresse (niederwertige Teil der Adresse).
Zero-Page: Höherwertiger Teil: 0-Bits
Seiten-Register: Höherwertiger Adressteil wird in Prozessorregister bereitgestellt

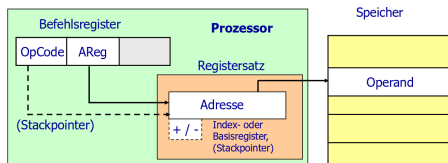
Register-indirekte Adressierung: Im Opcode angegebenes Adressregister enthält Adresse des Operanden (= *Pointer*)

Beispiel: LD R1, (A0) (Lade Register R1 mit Inhalt des in A0 angegebenen Speicherwortes)

Im Register stehende Adresse oft Anfang/Ende eines Tabellenbereiches, deswegen Hilfsmethoden:

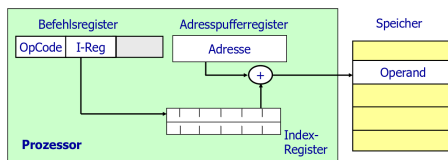
Postinkrement: Nach Befehlsausführung Registerinhalt inkrementieren und auf nächste Speicherzelle zeigen (z.B. INC (R0)+: Inkrementiere Speicherwortinhalt, das von R0 adressiert wird, danach Inhalt von R0)

Predekrement: Vor Befehlsausführung Registerinhalt erniedrigen und auf vorhergehende Speicherzelle zeigen (z.B. CLR -(R0): Dekrementiere Inhalt von R0, lösche dann durch R0 adressiertes Speicherwort)

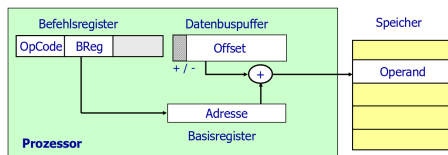


Indizierte Adressierung: Effektive Adresse wird durch Addition eines Registerinhalts zu angegebenem Basiswert berechnet

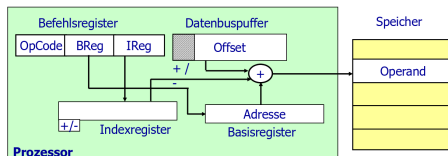
1. **Speicher-relative Adressierung:** Basiswert wird als absolute Adresse im Befehl vorgegeben
Beispiel: ST R1, \$A704 (R0) (Speichere Inhalt von R1 in Speicherwort, dessen Adresse man durch Addition des Inhalts von R0 zu Basis \$A704 erhält)



2. **Register-relative Adressierung:** Basiswert befindet sich in Basisregister, verwiesen im BReg-Feld des Opcodes
Beispiel: CLR \$A7 (B0) (Lösche Speicherwort, dass man durch Addition von \$A7 zu B0 erhält)



3. **Register-relative Adressierung mit Index:** Basiswert in Basisregister, Addition des Indexregister-Inhalts (hat ggf. Autoinkrement/Autodekrement), ggf. Angabe eines zusätzlichen Offsets im Befehl (wird hinzuaddiert)
Beispiel: DEC \$A7 (B0) (I0)+ (Dekrementiere Speicherwort, dessen Adresse I0+B0+\$A7 ist, inkrementiere danach den Inhalt von I0)

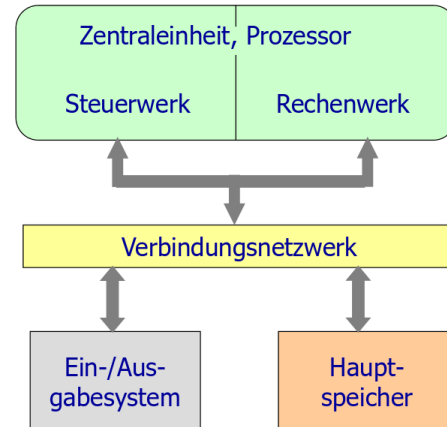


Programmzähler-relative Adressierung: Effektive Adresse = Befehlszählerstand + Offset (im Befehl angegeben) - erlaubt Programme im Hauptspeicher zu verschieben

Beispiel: LBRA \$7FFF (verzweige *unbedingt* zu Speicherzelle, deren Adressdistanz zu Programmzähler \$7FFF ist)

V. RECHNERMODELL

Rechnermodell – von-Neumann-Rechner



Von-Neumann-Rechner – Komponenten

Zentraleinheit: Verarbeitet Daten gemäß eines Programms

1. **Leitwerk/Steuerwerk:** Holt Programmbefehle aus Speicher, entschlüsselt sie, steuert Ausführung
2. **Rechenwerk/ALU:** Führt arithmetische/logische Operationen aus, wird beeinflusst durch Steuersignale, liefert Meldesignale an Steuerwerk

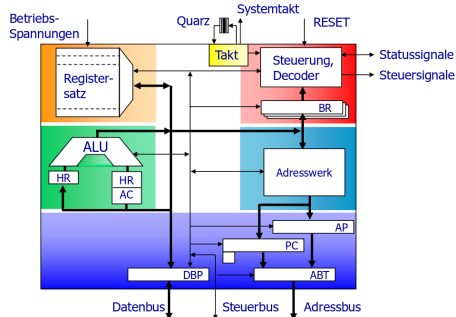
Hauptspeicher: Besteht aus eindeutig adressierbaren Speicherzellen, bewahrt Programme und Daten auf (im Gegensatz zur *Harvard-Architektur*)

Bussystem:

1. **Adressleitungen:** Transportieren unidirektional Adressinformationen
2. **Datenleitungen:** Transportieren bidirektional Daten und Befehle (von/zum Prozessor)
3. **Steuerleitungen:** Transportieren uni- oder bidirektional Steuerinformationen (von/zum Prozessor)

Ein-/Ausgabesystem: Geräte zur Eingabe von Daten/Programmen bzw. zur Ausgabe von Daten (angebunden durch Bussysteme)

Rechnermodell – einfacher Mikroprozessor



Steuerwerk: Steuert die Systemkomponenten (meist *dynamisches Schaltwerk* \leadsto Zustandsinformation in Kondensatoren gespeichert \leadsto Refresh-Taktfrequenz erforderlich, sonst Informationsverlust)

1. **Befehlsregister:** enthält den gerade ausgeführten Befehl (besteht aus mehreren Registern: unterschiedlich lange Befehlsformate bzw. Vorabladen von Befehlen)
2. **Decoder:** dekodiert Befehlsworte (= mikroprogrammiertes/festverdrahtetes Schaltwerk)
3. **Taktgenerator:** erzeugt Systemtakt (durch externen Quarz), erzeugt ein mit Prozessortakt synchronisiertes Rücksetzsignal (startet Initialisierungsroutine des Steuerwerks)
4. **Steuerregister:** Ermöglicht Beeinflussung der aktuellen Arbeitsweise des Steuerwerks (Interrupt enable Bit: wird auf Unterbrechungsanforderung am INT-Eingang reagiert?, User/System Bit: Sind alle Befehle nutzbar oder nur bestimmte?, Trace Bit: Unterbrechungsroutine nach jeder Befehlsausführung starten?, Decimal Bit: Dual oder BCD rechnen?)

Befehlsausführung:

1. Holphase: nächster Befehl wird in Befehlsregister geladen
2. Dekodierphase: Decoder ermittelt Startadresse des Mikroprogramms, welches Befehl ausführt
3. Ausführungsphase: Mikroprogramm führt Befehl aus, indem Signalfolgen an andere Komponenten übermittelt und Meldesignale ausgewertet werden

Rechenwerk: Führt vom Steuerwerk verlangte logische/arithmetische Operationen aus (reines Schaltnetz)

1. **Statusregister:** informiert Steuerwerk über Ablauf des Ergebnisses (Übertrag CF, Hilfsübertrag AF, Null ZF, Vorzeichen SF, Überlauf OF, Even EF, Parität PF)
2. **Hilfsregister/Akkumulatoren:** Zwischenspeichern von Operanden und Ergebnissen (meist zweigeteilt: Akkumulator-Register AC + nachgeschaltetes Hilfsregister HR (Latch))
3. **Busse:** 2 Eingangsbusse (Operanden), 1 Ausgangsbuss (Ergebnis)
4. **Ergebnisse:** Werden entweder in Prozessor-Registern gespeichert, zu ALU-Eingangsregistern zurückgeführt oder über externen Datenbus an andere Systemkomponenten übergeben

Varianten:

1. **Variante A:** Hilfsregister des Akkumulators hinter der ALU (Vorteil: ALU-Operationen ohne Akkumulatorveränderung möglich. Nur bei alten 8 Bit-Prozessoren)
2. **Variante B:** Rechenwerk ohne Akkumulator - Akkumulator in Prozessor-Registersatz verlegt (Vorteil: Mehrere Register können Akkumulatorfunktionen übernehmen. Bei allen modernen 16/32 Bit-Prozessoren)

Operationen:

1. Arithmetische Operationen (Addieren/Subtrahieren ohne/-mit Übertrag, Inkrementieren/Dekrementieren, Multiplizieren/Dividieren ohne/mit Vorzeichen, Komplement)
2. Logische Verknüpfungen (Negation, Und, Oder, Antivalenz)
3. Schiebe-/Rotationsoperationen (Links-/Rechtsverschieben, Links/Rechts ohne/mit Übertragsbit rotieren)
4. Transportoperationen (Transferieren: move, load, store, ...)

Registersatz: Zwischenspeicherung von häufig benutzten Operanden \leadsto schnellerer Zugriff als auf Hauptspeicher

1. **Dual-Port-Speicher** zwischen Ein- und Ausgangsbuss \leadsto ein Register lesen und anderes schreiben gleichzeitig möglich
2. **Zusatzfunktionen:** Inkrementieren, Dekrementieren, Nullsetzen, Inhaltsverschiebung
3. **Datenregister:** Zwischenspeichern von Operanden, bei modernen Prozessoren mehrere Datenregister als Akkumulator nutzbar
4. **Adressregister:** Speichern von Adressen/-teilen eines Operanden, Basisregister (enthält Anfangsadresse eines Speicherbereichs, unverändert während dessen Bearbeitung), Indexregister (enthält Offset zu Basisadresse zur Auswahl eines bestimmten Datums im Speicherbereich)
5. **Spezielle Register:** Instruction pointer, Steuerregister, Statusregister, Interrupt-Behandlungsregister, Stackregister
6. **Registerskalierung:** Je nach aktueller Datenlänge (1 Byte, 2 Byte,...) wird Indexregister vor Auswertung mit 1,2,... multipliziert (Vorteil: bessere Registerbreitenausnutzung, da Register nur um 1 de-/inkrementiert werden muss)
7. **Laufzeitstack:** LIFP-Speicher im Hauptspeicher, speichert Prozessorstatus und Programmzähler bei Unterprogrammaufruf/Unterbrechungsroutinen, Parameterübergabe, kurzzeitige Datenlagerung bei Ausführung, Datenübertragungsbefehle PUSH und POP/PULL



Adresswerk: Berechnet nach Steuerwerkvorschriften Adresse eines Befehls/Operanden (früher Bestandteil des Rechenwerks, heute sehr komplex, da es viele verschiedene komplexe Adressierungsarten gibt)

Aufbau und Funktionsweise:

Adress-Addierer - Eingang A: Registersatz (Adresse aus Basis-/Indexregister), Datenbuspuffer (absolute Adresse im Befehl oder absolute Distanz zu Basisregister)

Adress-Addierer - Eingang B: Befehlszähler (Adresse des aktuellen Befehls), Adresspuffer (Adresse des aktuellen Operanden)

Adressberechnung parallel zu Rechenwerkaktivitäten durch Pipelining

Tätigt Adressrechnung zur virtuellen Speicherverwaltung Früher separat, heute in Prozessor integriert

Komplexe Adresswerke zur virtuellen Speicherverwaltung

Systembus-Schnittstelle: Enthält Zwischenspeicherregister zur kurzfristigen Daten-/Adressaufbewahrung (Befehlszähler, Adresspuffer, Datenbuspuffer) und Ein-/Ausgangstreiber

Spezielle Funktionseinheiten: Moderne Mikroprozessoren haben Speicherverwaltungseinheit (MMU), Arithmetik-Einheiten, Cache-Speicher,...

Rechnermodell – CISC

= *complex instruction set computer*

Vorteile: Mikroprogrammierung begünstigt komplexe Befehle, komplexe Befehle \leadsto kurze Programme, Unterstützung höherer Programmiersprachen durch komplexe Befehle (Abbildung Sprachkonstrukt \rightarrow Befehl direkter)

Fazit: Entwicklung von Hardware, Programmiersprachen und Einsatzgebieten begünstigt *komplexe* Befehle

Nachteile: Umfangreiche Mikroprogramme, verlängerte Entwurfszeit, komplexe Steuerwerke, nur kleine Teile des Befehlssatzes häufig benutzt, größere Fehleranfälligkeit auf Mikroprogrammebene, schwieriger Compilerbau

80/20-Regel: Nur 20% der Befehle werden häufig verwendet

Cycles per instruction (CPI): Bei meisten heutigen CISC-Architekturen: $CPI \gg 2$

Rechnermodell – RISC

= *reduced instruction set computer*

Grundprinzipien:

1. Vielbenutzte Befehle so schnell wie möglich machen (keine Mikroprogrammierung, Befehlspipeline)
2. Hauptarbeit durch optimierende Compiler
3. Operanden in großen Registersätzen halten (schneller Zugriff, schnelle Verarbeitung)
4. einheitliche Befehlsformate (schnelle Decodierung)
5. viel Pipelining

Ziele:

1. Jeder Befehl ein Taktzyklus
2. alle Befehle gleich lang
3. nur Load-Store und Register-Register-Befehle (wenige Adressierungsarten \leadsto schnelle Ausführung)
4. Koprozessorarchitektur für komplexe Befehle

keine Ziele: Unterstützung von Gleitkomma-Arithmetik oder Betriebssystemfunktionen

Forderungen an RISC-Systeme:

1. Mindestens 75% Ein-Zyklus-Befehle
2. Länge aller Befehle = Datenbusbreite
3. Nicht mehr als 128 Befehle
4. Nicht mehr als 4 Adressierungsarten
5. Load-Store-Architektur
6. Festverdrahtete Steuereinheit (keine Mikroprogramme)
7. Mindestens 32 allgemein verwendbare Register

RISC – Prozessoraufbau

Harvard-Architektur: Programm- und Datenspeicher getrennt \leadsto paralleles Holen von Operanden und Instruktionen

Varianten:

1. **zwei** getrennte Bussysteme bis zu Cachespeichern, nur ein Arbeitsspeicher (niedrigere Kosten)
2. **ein** Bussystem (wie Standard-Mikroprozessoren)

Systembusschnittstelle: enthält Registerblocks für Daten und Adressen (parallel Lesen von Daten und Zwischenspeichern von Ergebnis)

Befehlszähler: manchmal als Hardware-Stack ausgebildet (schnellere Unterprogrammaufrufe)

Steuerwerk: festverdrahtet, Befehlsregister als Warteschlange (FIFO), für jede Pipeline-Stufe ein Register, Opcodes jeder Stufe können von Steuerwerk-Schaltznetz ausgewertet werden

Registersatz: große Registeranzahl, erlaubt gleichzeitige Auswahl von 3 bis 4 Registern

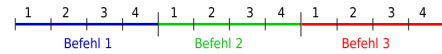
Rechenwerk: Load/Store-Architektur, Operanden via 2 Operandenbusse aus Registersatz, Ergebnis (im selben Taktzyklus) über Ergebnisbus in Registersatz, keine direkte Verbindung zwischen ALU und Systembus - Transfer über Register

Superskalarität: Schafft RISC-Prozessor eine Befehlsausführung pro Takt, so heißt er *superskalar* - pro Takt können mehrere Befehle den Ausführungseinheiten zugeordnet und so viele Befehlsausführungen beendet werden

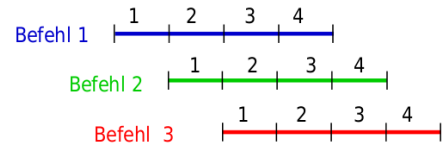
VI. PIPELINE-VERARBEITUNG

Pipeline vs Seriell

Seriell:

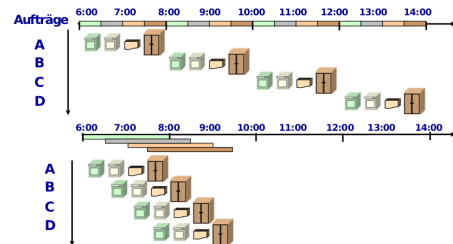


Pipelining:



Beispiel: Wäsche waschen

1. Schmutzige Wäsche in Waschmaschine
2. Nasse Wäsche in Trockner
3. Falten/Bügeln
4. Kleider in Schrank legen

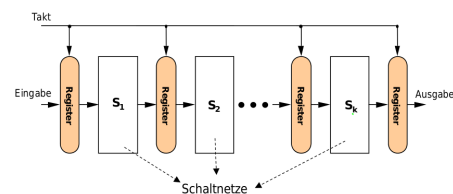


Definition *Pipelining:* Operationszerlegung in mehrere Phasen/-Suboperationen, die von hintereinandergeschalteten Verarbeitungseinheiten takt synchron bearbeitet werden (jede Verarbeitungseinheit führt eine Teiloperation aus).

\leadsto Gesamtheit Verarbeitungseinheiten = Pipeline

Begrifflichkeiten

Stufen, Register:



Verzögerungszeiten: Schaltnetze: τ_i ($i = 1, \dots, k$), Register: τ_{reg}
Pipeline-Maschinentakt: Benötigte Zeit, um Befehl eine Stufe weiterzuschieben (ideal: k -stufige Pipeline führt Befehl in k Takten mit k Stufen aus)

Latenz: Zeit, die ein Befehl braucht, um alle k Stufen zu durchlaufen

Durchsatz: Anzahl Befehle, die Pipeline pro Takt verlassen können: $D = \frac{n}{(k+n-1) \cdot \tau}$ ($\tau : \max\{\tau_1, \dots, \tau_k\} + \tau_{\text{reg}}$) - $\lim_{n \rightarrow \infty} D = D_{\text{max}} = \frac{1}{\tau}$

Speedup: n Befehle mit k -stufiger Pipeline ausführen:

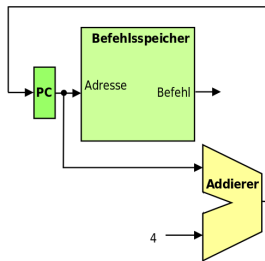
1. sequentiell: $n \cdot k$ Taktzyklen
2. mit pipelining: $k + (n - 1)$ Taktzyklen (Latenz k , Durchsatz $1 - k$ Takte zum Füllen, $(n - 1)$ für die restlichen Befehle)

\leadsto Speedup $S = \frac{n \cdot k}{k + n - 1}$ (Leistungssteigerung - $\lim_{n \rightarrow \infty} S = k$)

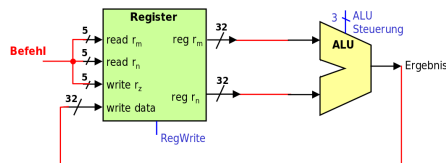
MIPS – Befehlsabarbeitung

Befehl holen: Wird bei jedem Befehlstyp benötigt

1. Befehl im Befehlsspeicher adressieren
2. Befehlszähler (PC) um 4 inkrementieren

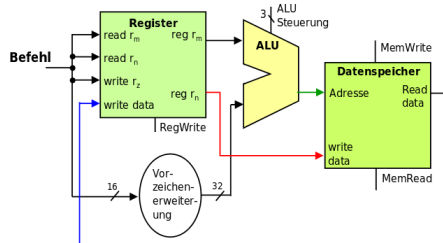


Typ R: Lesezugriff auf beide Operandenregister, Schreibzugriff auf Zielregister

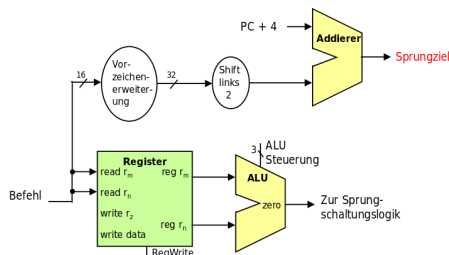


Typ I: Bestimmung der Speicher-/Ladeadresse durch Addieren von 16-Bit-Offset zu Basisadresse in r_m

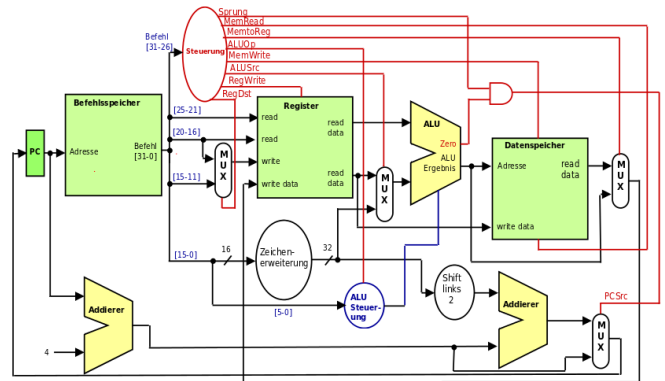
1. Laden (z.B. `lw r_z , offset(r_m)`): Wort an Ladeadresse wird in r_z geladen
2. Speichern (z.B. `sw r_n , offset(r_m)`): Wort in r_n wird in Speicheradresse gespeichert



Typ J: 16-Bit Offset \rightsquigarrow bis zu $2^{15} - 1$ Befehle vorwärts/ 2^{15} rückwärts (Basisadresse zur Sprungberechnung = Befehlsadresse hinter Verzweigungsbefehl (PC+4), z.B. `beq r_n , r_m , offset`: Vergleich von r_n , r_m entscheidet ob Sprung genommen wird)

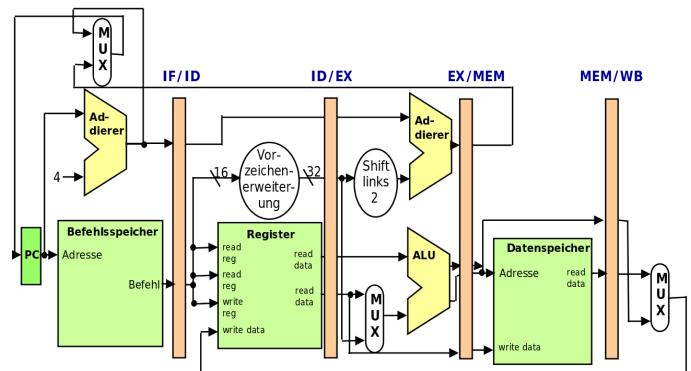


MIPS-Datenpfad: Zusammenführung der drei Modelle



MIPS – Pipelining: DLX-Pipeline (DLX = deluxe)

Pipelinestufen, Buffering (siehe Grafik „Begrifflichkeiten: Stufen, Register“)



1. Phase 1 (IF-Phase): Instruction Fetch. Der durch Befehlszähler adressierte Befehl wird aus Cache/Speicher in Befehlsbuffer geladen, Befehlszähler wird inkrementiert
2. Phase 2 (ID/RF-Phase): Instruktion Decode, Register Fetch. Aus Opcode werden prozessorinterne Steuersignale erzeugt, Operanden werden aus (Universal-)Register bereitgestellt
3. Phase 3 (EX-Phase): Execution, Effective Address Calculation. Operation wird auf Operanden ausgeführt, bei Lade-/Speicherbefehl oder Verzweigung berechnet ALU effektive Adresse
4. Phase 4 (MEM-Phase): Memory access. Speicherzugriff (bei Lade-/Speicherbefehl) wird durchgeführt
5. Phase 5 (WB-Phase): Write back. Ergebnis wird in (Universal-)Register geschrieben (ergebnislose Befehle in dieser Phase passiv)

MIPS – Pipeline-Konflikte

Datenkonflikte: Benötigter Operand ist in der Pipeline (noch) nicht verfügbar (Datenabhängigkeiten im Befehlsstrom)

Struktur-/Ressourcenkonflikte: Zwei Pipeline-Stufen benötigen selbe Ressource, aber nur einfacher Zugriff möglich

Steuerflusskonflikte: Bei Programmsteuerbefehlen

1. Zieladresse des nächsten Befehls in Befehlsbereitstellungsphase noch nicht berechnet
2. Bedingter Sprung; noch unklar, ob gesprungen werden muss

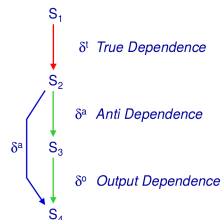
Datenabhängigkeiten/-konflikte

Datenabhängigkeit: Zwischen zwei aufeinanderfolgenden Befehlen $Inst_1, Inst_2$ kann eine *Datenabhängigkeit* bestehen:

1. **Echte Datenabhängigkeit** (true dependence, δ^t): $Inst_1$ schreibt Ausgabe in Register, dass die Eingabe von $Inst_2$ ist
2. **Gegenabhängigkeit** (antidependence, δ^a): $Inst_1$ liest von einer Stelle, die von $Inst_2$ überschrieben wird
3. **Ausgabeabhängigkeit** (output dependence, δ^o): $Inst_1$ und $Inst_2$ schreiben Ergebnis in selbes Register, aber $Inst_2$ nach $Inst_1$

Beispiel:

```
S1 : add r1,r2,2 # r1 := r2+2
S2 : add r4,r1,r3 # r4 := r1+r3
S3 : mul r3,r5,3 # r3 := r5*3
S4 : mul r3,r6,3 # r3 := r6*3
```



Datenkonflikte:

1. **Lese-nach-Schreibe-Konflikt** (read after write, RAW): Verursacht durch echte Abhängigkeit
2. **Schreibe-nach-Lese-Konflikt** (write after read, WAR): Verursacht durch Gegenabhängigkeit, tritt auf, wenn Schreib- vor Lesestufe in Pipeline
3. **Schreibe-nach-Schreibe-Konflikt** (write after write, WAW): Verursacht durch Ausgabeabhängigkeit, tritt auf, wenn Pipeline in mehr als einer Stufe schreibt oder ein Befehl fortgesetzt werden darf, wenn vorhergehender angehalten wurde

Kein WAR-/WAW-Konflikt in fünfstufiger DLX-Pipeline, da nicht überholt werden kann und immer in Stufe 2 gelesen wird (bzw. in Stufe 5 geschrieben)

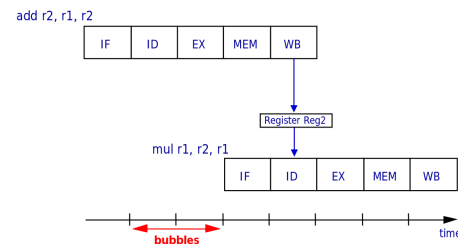
Lösungen:

1. Software-Lösung:

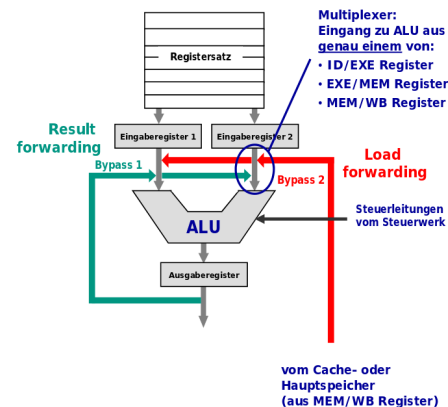
- (a) Compiler erkennt Datenkonflikte, fügt nach konfliktverursachenden Befehlen Leeroperation (**noop**) ein
- (b) Statische Befehlsanordnung: Befehle werden umgeordnet (Optimierung), um Leeroperationen zu minimieren (Instruction Scheduling, Pipeline Scheduling)

2. Hardware-Lösung:

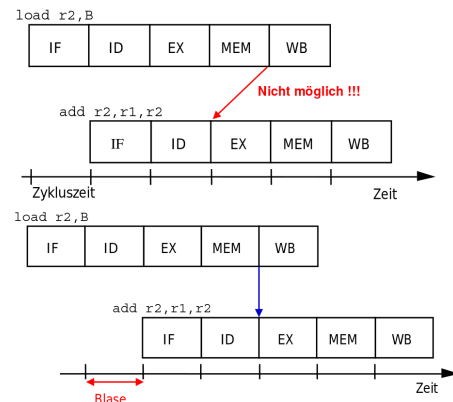
- (a) Pipeline leerlaufen lassen: Pipeline-Sperrung (*interlocking*) bzw. Pipeline-Leerlauf (*stalling*)



- (b) *Forwarding*: Wird Datenkonflikt erkannt, so wird Operand nicht aus Universalregister, sondern direkt aus ALU-Ausgaberegister der vorherigen Operation in ALU-Eingaberegister übertragen



- (c) *Forwarding with interlocking*: Forwarding löst nicht alle möglichen Datenkonflikte



Ressourcenkonflikte

Treten bei einfachen Modellen wie einer DLX-Pipeline nicht auf

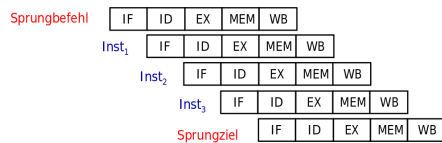
Lösungen:

1. **Arbitrierung mit Interlocking**: Arbitrierungslogik hält konfliktverursachenden Befehl an \rightsquigarrow Verzögerung
2. **Übertaktung**: Konfliktverursachende Ressource wird schneller getaktet als übrige Pipeline-Stufen
3. **Ressourcenreplizierung**: Ressourcen werden vervielfacht (z.B. Registersatz mit mehreren Schreibkanälen)

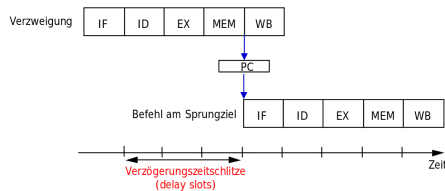
Steuerflusskonflikte

Programmsteuerbefehle: bedingte und unbedingte Sprünge, Unterprogrammaufruf- und -rückkehrbefehle, Unterbrechungsbefehle

Befehlszähler wird erst in MEM-Stufe ersetzt \leadsto Befehle in Pipeline, die wieder gelöscht werden müssen



\leadsto nach genommenem Sprung müssen Wartezyklen eingefügt werden (die ggf. durch Umstrukturierung der Pipeline verringert werden können)



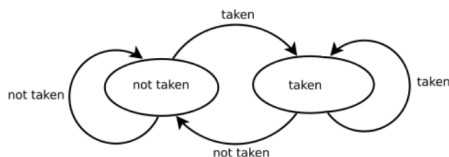
Lösungen:

1. Software-Lösung:

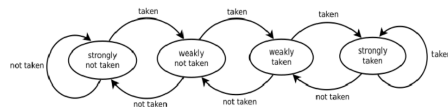
- Einfügen von `noop` in die Wartezyklen
- Statische Befehlsanordnung: Compiler optimiert Code und fügt nicht sprunghafte Befehle in Wartezyklen ein, `noop`, falls es keine gibt \leadsto Code Pipeline-abhängig

2. Hardware-Lösung:

- Hardware Erkennt Verzweigungsbefehle in ID-Stufe und lädt danach bis zur Berechnung der Zieladresse keine weiteren Befehle \leadsto bereits in IF-Stufe geladener Befehl muss gelöscht werden
- Spekulation auf nicht genommene bedingte Sprünge - wird Sprung doch genommen Pipeline leeren (*Pipeline flushing*)
- 1-Bit-Prädiktor zur Sprungvorhersage



(d) 2-Bit-Prädiktor zur Sprungvorhersage

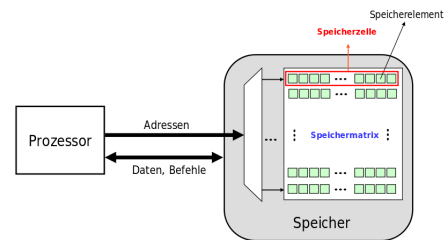


- Moderne Sprungvorhersagetechniken: In mehr als 90% der Fälle richtig

VII. SPEICHER

Begriffe:

Hauptspeicher: „Gedächtnis“ des Rechners. Beinhaltet Programme und Daten, die jederzeit und sofort (*random access*) zur Verfügung stehen müssen



Speicherelement: 1 Bit Speicher

Speicherzelle: feste Anzahl von Speicherelementen, auswählbar durch eindeutige Adresse. 8, 16, 32, ... Bit

Speicherwort: maximale Anzahl an Speicherelementen, die in einem Buszyklus zwischen Mikroprozessor und Speicher übertragen werden können \leadsto Speicherwortbreite = *Datenbusbreite*

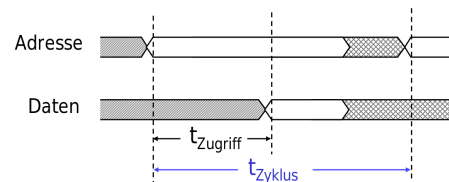
Wahlfreier Zugriff: Jede Speicherzelle kann direkt angesprochen werden (ohne andere Zellen ansprechen zu müssen), Selektion über Adressdecoder

Speicherorganisation: Definition über Anzahl n der Zeilen und Anzahl m der Speicherelemente pro Zeile, z.B. 16-MBit-DRAM mit Organisation 4Mx4/2Mx8/1Mx16

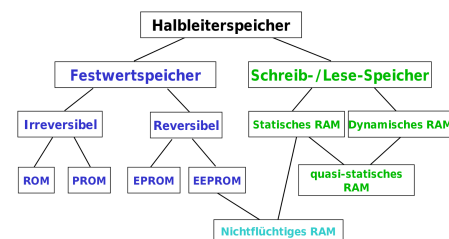
Kapazität: Informationsmenge, die im Speicher untergebracht werden kann ($n * m$ Bit)

Arbeitsgeschwindigkeit:

- Zugriffszeit** (access time): maximale Zeit zwischen Anlegen einer Speicheradresse und Ausgabe der gewünschten Daten
- Zykluszeit** (cycle time): minimale nötige Zeit zwischen zwei hintereinanderfolgenden Adressenaufschaltungen an den Speicher



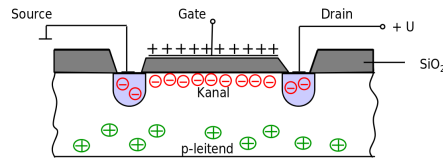
Speicherklassifizierung:



Transistor (MOSFET):

Eine Spannung an einem *Gate* regelt, ob Strom zwischen *Source* und *Drain* fließt.

n-MOS-MOSFET: Kanal sperrt, wenn keine Spannung anliegt (*selbstsperrend*)



Speicherzelle – statisch (SRAM):

Aufgebaut aus zwei kreuzweise rückgekoppelten Invertoren und zwei Transistoren zur Ankopplung an Bitleitungen \leadsto 6-Transistor-Zelle

Vorteile: Strom fließt nur zum Umschaltzeitpunkt \leadsto kein Refresh nötig

Nachteile: Hoher Platzverbrauch

Speicherzelle – dynamisch (DRAM):

Aufgebaut aus einer Transistorzelle und einem Kondensator (vergrößerte Drain-Zone, von Drain-Kontakt durch dünne Isolierschicht getrennt) \leadsto Platzverbrauch viermal kleiner als bei SRAM

Vorteile: Geringer Platzverbrauch

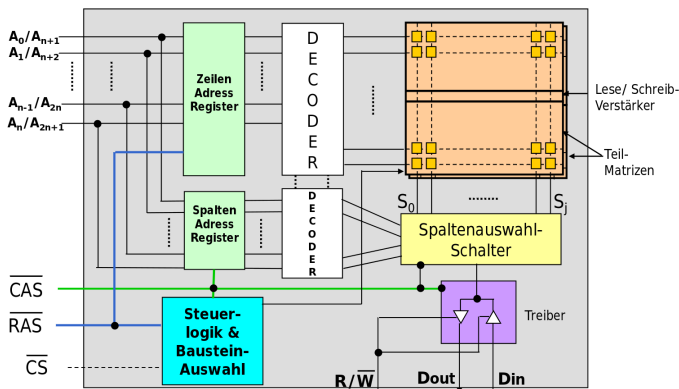
Nachteile: Information geht beim Lesen verloren und muss neu gespeichert werden (*destructive read*), Ladung geht nach einiger Zeit durch Leckströme verloren \leadsto periodische Auffrischung (*refresh*) nötig

Lesen:

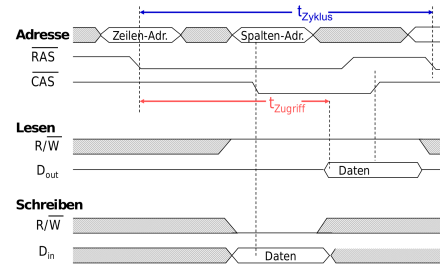
1. Leistungskapazität wird vorgeladen (*precharge*)
2. Positive Spannung wird an Gate des Speichertransistors angelegt
3. Leseverstärker misst Strom am Ende der Bitleitung

Schreiben:

1. Speichertransistor wird durch Spannung U_{GS} leitend
2. Bitleitung auf Masse \leadsto Elektronen werden auf Drain-Zone aufgebracht, Kondensator lädt
3. Bitleitung auf U_B \leadsto Elektronen von Drain-Zone abgesaugt, Kondensator entlädt



DRAM – Adressierung



DRAM – Auffrischung

Zeilenweise, jede Zeile alle 2 msec

Nur Zeilenadresse wird an Baustein angelegt, RAS=0, CAS=1

Zugriffsbeschleunigung – Techniken

Prinzip: Übertragung von benachbarten Bytes (Blöcken) anstatt einzelnen Bytes

\leadsto **beschleunigter Zugriff** auf Speicherbaustein, falls alle zu lesenden/schreibenden Speicherzellen in einer Zeile liegen
Zeilenadresse wird auch bei wiederholtem Zugriff auf Zeile (auch *page* genannt) nur einmal angelegt (und im Register gespeichert). Dann werden in schneller Folge die Spaltenadressen angelegt (*fast page mode*: **FPM-DRAM**)
 \leadsto **erheblich beschleunigter Zugriff**

FPM-DRAM

aufeinanderfolgende Speicherzugriffe oft in selber Zeile
 \leadsto ausnutzen

Initialisieren: Wie normaler DRAM

Nach 1. Lesezyklus: Speichersteuerung RAS-Signal bleibt aktiv
 \leadsto Zeile bleibt aktiv

Bei folgenden Lesezugriffen: Speichersteuerung übergibt nur noch jeweils eine neue Spaltenadresse an DRAM

\leadsto RAS-precharge-Zeit und RAS-CAS-Delay entfallen bei Folgezugriffen

EDO-RAM

= *extended data output* RAM

Datenausgabe wird bei Lesen von CAS-Signal durch **interne Pufferung** entkoppelt

\leadsto Daten stehen länger am Ausgang bereit

\leadsto bessere Verschachtelungsmöglichkeiten beim Lesen

Prozessor kann Daten auslesen, während Speichersteuerung neue Spaltenadresse an DRAM übergibt

SDRAM

= *synchronous* dynamische RAMs

beherrscht heute Speichermarkt

Alle Ein-/Ausgangssignale synchron zum Systemtakt

Prozessor, Chipsatz, Speicher kommunizieren über ein Bussystem (mit einer Frequenz getaktet)

Intern 2 bis 4 Speicherbänke

Nach Anlegen von Zeilen-/Spaltenadresse:

1. Speichersteuerung generiert nachfolgende Adressen
2. Speichersteuerung führt alternierenden, überlappenden Zugriff auf die Speicherbänke aus

DDR-SDRAM

Nächste Stufe SDRAM (SRAM II)

Vier Speicherbänke, die parallel arbeiten

Prinzip:

- Bandbreitenerweiterung durch Nutzung beider Taktflanken
- Daten werden bei steigender + fallender Taktflanke übertragen
- \leadsto doppelter Datendurchsatz

Laufzeitverzögerungen sehr kritisch

\leadsto Verwendung von bidirektionalem Strobe-Signal (**DQS**) zusätzlich zu Systemtakt

SDDR

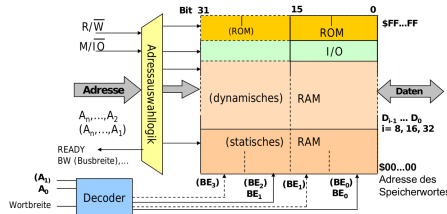
- = *sync link* SDRAM
- Weiterentwicklung SDRAM
- Höhere erlaubte Busfrequenzen \rightsquigarrow höhere Leistung

Organisation – Hauptspeicher

- lineare Liste von Speicherworten
- Aufbau: Speicherbausteine
- Zugriffszeit: Abhängig von verwendeten Speicherbausteinen
- Breite: IdR Breite des Datenbus
- Maximale Kapazität: Gegeben durch Breite des Adressbus

Memory Map

- = Speicher-Belegungsplan
- Gibt an, welche Speicherbausteine auf welchen Bereichen des Hauptspeichers liegen



VIII. CACHE-SPEICHER

Hintergrund

Lücke zwischen Verarbeitungs- (CPU) und Zugriffsgeschwindigkeit (DRAM) immer größer

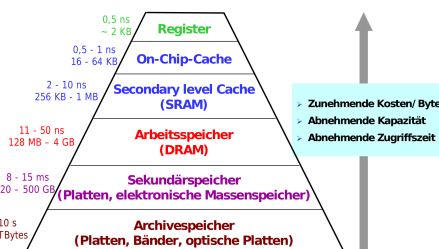
Lösung: Hierarchische Anordnung verschiedener Speicher

\rightsquigarrow Ausgleich der unterschiedlichen Zugriffszeiten

Strategien:

1. **Cache-Speicher**: Kurze Zugriffszeiten
 \rightsquigarrow Beschleunigung Prozessorzugriff
2. **Virtueller Speicher**: Vergrößerung des tatsächlich vorhandenen Hauptspeichers (z.B. bei gleichzeitiger Bearbeitung mehrerer Prozesse)

Leistung abhängig von Eigenschaften der Speichertechnologien, Adressierung und Organisation



CPU-Cache

- kleiner, schneller Pufferspeicher
- speichert *Kopien* von Hauptspeicherteilen, auf die mit hoher Wahrscheinlichkeit als nächstes zugegriffen wird
- Effizienz durch *Lokalitätseigenschaft* von Programmen:

1. **zeitliche Lokalität**: zukünftig angesprochene Information mit hoher Wahrscheinlichkeit schon einmal angesprochen worden (z.B. Schleifen)
2. **örtliche Lokalität**: zukünftig angesprochene Information mit hoher Wahrscheinlichkeit in Nähe des bisherigen Zugriffs (z.B. Arrays)

Cache-Controller lädt alle Daten in Cache, auf die Prozessor zugegriffen

Daten werden aus Cache verdrängt, wenn sie nicht mehr benötigt werden

Cache – Funktionsweise

Lesezugriff: μP überprüft davor ob Datum in Cache steht

- **read hit**: Datum wird ohne Wartezyklen aus Cache geladen
- **read miss**: Datum wird mit Wartezyklen aus Arbeitsspeicher geladen und in Cache eingefügt

Schreibzugriff:

- **write miss**: Datum wird in DRAM und Cache geschrieben
- **write hit**: verschiedene Verfahren möglich

Schreibzugriff – Durchschreibverfahren

Datum wird von CPU immer gleichzeitig in Cache- und Arbeitsspeicher geschrieben

Vorteil: Konsistenzgarantie zwischen Cache und DRAM

Nachteil: Schreibzugriffe benötigen immer langsame Zykluszeit von Hauptspeicher, belasten Systembus

Schreibzugriff – gepuffertes Durchschreibverfahren

Verwendung von *Schreib-Puffer*, der zu schreibende Daten temporär aufnimmt

Daten werden dann automatisch von Cache-Controller in Hauptspeicher übertragen

Schreibzugriff – Rückschreibverfahren

Datum wird von CPU nur in Cachespeicher geschrieben und durch spezielles Bit (*dirty bit*) gekennzeichnet

Arbeitsspeicher wird nur geändert, wenn *dirty*-Datum aus Cache verdrängt wird

Vorteil: Alle Schreibzugriffe mit schneller Cache-Zykluszeit abwickelbar

Nachteil: Konsistenzprobleme zwischen Cache- und Hauptspeicher

Konsistenzprobleme

Andere Systemkomponenten (z.B. DMA-Controller) finden ggf. „veraltete Daten“ in DRAM vor, die von CPU längst geändert wurden

Andere Systemkomponenten können Daten in Hauptspeicher ändern, während CPU noch mit alten Daten aus Cache arbeitet

\rightsquigarrow aufwendige Verfahren bei Cache-Steuerung zur Inkonsistenzvermeidung erforderlich

Begriffe

Hit-Rate: = Anzahl Treffer pro Anzahl Zugriffe

Mittlere Zugriffszeit:

$$t_{\text{access}} = (\text{Hit-Rate})t_{\text{hit}} + (1 - \text{Hit-Rate})t_{\text{miss}}$$

Cache-Speicher – Aufbau

Besteht aus drei Teilen:

1. **Datenspeicher**: im Cache abgelegte Daten
2. **Adressspeicher**: Adresse dieses Datums im RAM
3. **Statusbits**: Geben an, ob Informationen gültig sind

Zusammen: *cache-line*

Komparator: ermittelt, ob zu Adressbus gehörendes Datum in Cache abgelegt ist

Cache-Speicher – Organisation

Problem: Wie feststellen, ob Daten im Cache sind? Wie können diese gefunden werden?

Techniken für Adressvergleich:

1. voll-assoziativer Cache
2. direct mapped cache
3. n-way set associative cache

Cache-Speicher – voll-assoziativer Cache

Gesamter Hauptspeicher wird auf gesamten Cache abgebildet

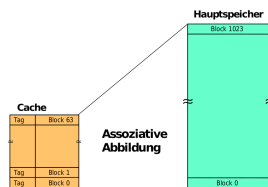
~> Jeder aus dem Hauptspeicher gecachte Wert kann an jeder Stelle im Cache stehen

Vorteile:

- beliebiges Ablegen
- optimale Cache-Ausnutzung

Nachteile:

- hoher HW-Aufwand (ein Vergleich pro Cachezeile)
- ~> nur für sehr kleine Caches realisierbar
- große Flexibilität erfordert weitere HW, die Ersetzungsstrategie realisiert



Cache-Speicher – direct mapped-Cache

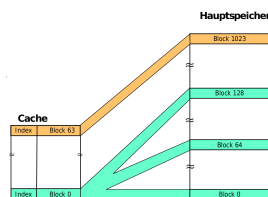
jede Hauptspeicherstelle erhält eindeutigen, festen Cacheplatz

Vorteile:

- Geringer HW-Aufwand (ein Vergleich für alle Tags)
- schneller Zugriff, da Tag-Feld parallel zu zugehörigem Block gelesen werden kann
- keine Ersetzungsstrategie erforderlich

Nachteile:

- ständige Konkurrenz der Blöcke, obwohl andere Blöcke frei sein können
- bei abwechselndem Zugriff auf zwei Speicherblöcke aus gleichem Index-Teil entsteht laufendes Überschreiben (aka *flattern*, *trashing*)



Cache-Speicher – N-way set associative-Cache

Mehrere Cache-Zeilen (z.B. 2) werden zu Sets zusammengefasst
Kompromiss zwischen direct mapped- und vollassoziativem Cache
Bessere Trefferrate durch Auswahlmöglichkeit

Ersetzungsstrategie:

- **zyklisch:** erster gespeicherter Eintrag wird als erster verdrängt (FIFO)
- **LRU** (*last recently used*): am längsten unbenutzter Eintrag wird verdrängt
- **zufällig**

Cache-Speicher – Ersetzungsstrategien

gibt an, welcher Cachespeicherteil nach cache miss durch neu geladene Speicheroption überschrieben werden soll

ist nur bei voll/n-fach-satzassoziativer Cachespeicherorganisation nötig

LRU (*last recently used*): am längsten nicht benutzte Speicherportion wird ersetzt (schwer in HW umzusetzen, deswegen meist nur Pseudo-LRU)

Ursachen für Fehlzugriffe

Erstzugriff (*compulsory*): Bei Erstzugriff auf Cache-Block ist dieser noch nicht im Cache und muss geladen werden

- Kaltstartfehlzugriffe (*cold start misses*)
- Erstbelegungsfehlzugriffe (*first reference misses*)

Kapazität (*capacity*): Falls Cache-Speicher nicht alle benötigten Cache-Blöcke aufnehmen kann müssen Cache-Blöcke verdrängt (und ggf später neu geladen) werden

Konflikt (*conflict*): Cache-Block wird verdrängt und später neu geladen, falls zu viele Cache-Blöcke auf selben Satz abgebildet werden

- Kollisionsfehlzugriffe (*collision misses*)
- Interferenzfehlzugriffe (*interference misses*)

Trefferquotenmaximierung

Sehr kleine Cachespeicher (32-128 Einträge): Voll-assoziativ

Größere Cachespeicher: Trend zu Direct Mapped oder 2-8-fach-assoziativ

Systembusanbindung

Cache-Controller: Tag-RAM, Steuerung, Tag-Komparator

- muss sehr schnell sein ~> auf Chip integriert
- übernimmt Steuerung der Treiber zum Systembus (Systembuszugriff nur bei Cache-Miss, ansonsten ist Systembus für andere Komponenten frei)
- sendet Systembussignale zur Wartezykleneinführung bei Cache-Miss (READY, HOLD, HOLDA, ...)

Cachespeicher: separat aus SRAM-Bausteinen aufgebaut

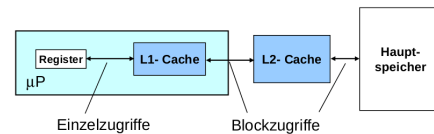
Cache – Hierarchien

First-Level Cache (*on-chip cache*): Auf Prozessor-Chip

- oft in **Befehlscache** und **Datencache** unterteilt

Secondary-Level Cache (*on-board cache*)

- kann parallel zu Hauptspeicher an Bus angeschlossen werden (*look-aside cache*) ~> schnelles Nachladen bei first-level-miss



Cache – Kohärenzproblem

Kohärenz: korrektes Voranschreiten des Systemzustands durch abgestimmtes Zusammenwirken der Einzelzustände

Konsistenz: alle Kopien eines Hauptspeicherdats in verschiedenen Cachespeichern identisch

~> **Kohärenz-Sicherstellung**

Bus-Schnüffeln (Snooping)

verwendet bei Mehrprozessorsystemen mit lokalem Cache pro Prozessor und gemeinsamen Hauptspeicher

Schnüffel-Logik: jeder Prozessor hört an Bus mit, wenn andere Prozessoren Adressen auf Bus legen

~> Vergleich mit den im lokalen Cache gespeicherten Daten

Snooping – Verfahren

Schreibzugriff auf dieselbe Adresse:

- Cache-Block für ungültig erklären (*write-invalidate*)
- Cache-Block aktualisieren (*write-update*)

Lesezugriff auf dieselbe Adresse mit modifiziertem Datum

~> Cache-Controller legt *snoop status signal* (SSTAT) auf Bus

1. Prozessor, der Adresse auf Bus gelegt hat, unterbricht Bustransaktion
2. schnüffelnder Cache-Controller übernimmt Bus und schreibt Cacheblock in Hauptspeicher
3. erneute Ausführung der Bustransaktion

IX. VIRTUELLE SPEICHERVERWALTUNG

Notwendigkeit

Immer größere Programme

Immer mehr Programme „gleichzeitig“

↪ verfügbarer Arbeitsspeicher schnell zu klein

Lösung: Nur gerade benötigte Teile der aktiven Programme im Arbeitsspeicher, Rest bei Bedarf aus Hauptspeicher nachladen (*swapping, paging*)

Umsetzung: MMU (*memory management unit*) setzt virtuelle Adressen in physikalische um

Virtueller Speicher

Speicherkapazität größer als effektive Hauptspeicherkapazität

Betriebssystem lagert nach Bedarf Speicherbereiche ein/aus

MMU-Adressberechnung hardwaremäßig eindeutig

Abbildungsinformation in Übersetzungstabellen gespeichert

↪ Abbildungsinformation für zusammenhängende Adressbereiche, um Übersetzungstabellen klein zu halten

Virtueller Speicher – Verwaltung (Segmentierung)

Virtueller Adressraum wird in Segmente verschiedener Länge zerteilt

Mehrere Segmente pro Programm (zB für Programmcode, Daten)

Segmente enthalten logisch zusammenhängende Informationen, relativ groß

Vorteile:

- spiegelt logische Programmstruktur wieder
- große Segmente ↪ relativ seltener Datentransfer

Nachteile:

- Datentransfer umfangreich falls notwendig
- Programm aus nur einem Code- und Datensegment
- ↪ muss vollständig eingelagert werden

Virtueller Speicher – Verwaltung (Seiten)

logischer und physikalischer Adressraum in Teile fester Länge (Pages) zerteilt

Pages relativ klein (256-4k Byte)

Viele Seiten pro Prozess, keine logischen Zusammenhänge

Vorteile:

- kleine Seiten ↪ nur wirklich benötigter Programmteil wird eingelagert
- geringerer Verwaltungsaufwand als Segmentierung

Nachteile:

- häufigerer Datentransfer als bei Segmentierung

Segmentbasierte Speicherverwaltung – Implementierung

virtuelle Adresse wird in **Segmentnummer** (n höherwertigste Bits der virtuellen Adresse) als Segmentkennung und in **Byte-nummer** (verbleibende m Bits der virtuellen Adresse) als Abstand zum Segmentanfang unterteilt

max. virtuelle Segmentanzahl 2^n , max. Segmentgröße 2^m

Adressabbildung über Segmenttabelle (im MMU-Registerspeicher)

reale Adresse = Segmentbasisadresse + virtuelle Byte-Nummer

Segmentlängenangaben um segmentüberschreitende Zugriffe feststellen/verhindern zu können

Verschnitt: ungenutzter Raum bei Segmenten kleiner als 2^m

↪ gute Hauptspeicherausnutzung, wenn Segmentgrenzen an jeder Byteadresse zulässig sind

Realität: Segmentgrenzen an Vielfachen von Blöcken (hier 256 Bytes)

Segmente im virtuellen physikalischen Adressraum in 256-Byte-Blöcke unterteilt

↪ m -Bit-Bytenummer wird aufgeteilt in kürzere Bytenummer für Byteadressierung in Block und Blocknummer

Adressumsetzung: virtuelle Segmentnummer wird auf reale 24-Bit-Blocknummer als Segmentbasis abgebildet

Virtuelle Bytenummer für Adressierung innerhalb des Blocks wird unverändert übernommen

Virtuelle Speicherverwaltung – Probleme

Einlagerungszeitpunkt: Wann werden Segmente/Seiten in Hauptspeicher eingelagert?

Zuweisungsproblem: Wo in Hauptspeicher werden Seiten/Segmente eingelagert?

Ersetzungsproblem: Welche Segmente/Seiten auslagern um Platz für neu benötigte Daten zu schaffen?

Probleme – Einlagerungszeitpunkt

Gängiges Verfahren: Einlagerung auf Anforderung (*demand paging* bei Seitenverfahren)

Daten werden eingelagert, sobald auf sie zugegriffen wird, aber nicht in Hauptspeicher liegen

Segment-/Seitenfehler (*segment/page fault*): Zugriff auf nicht in Hauptspeicher vorhandene(s) Segment/Seite

Probleme – Zuweisungsproblem (Segmentierung)

ausreichend große Lücke in Hauptspeicher muss gefunden werden
Strategien:

1. **first-fit:** erste passende Lücke wird genommen
2. **best-fit:** kleinste passende Lücke wird genommen
3. **worst-fit:** größte passende Lücke wird genommen

Problem: Speicher zerfällt in belegte und unbelegte Speicherbereiche

↪ **externe Fragmentierung**

unbelegte Speicherbereiche oft zu klein um weitere Segmente aufnehmen zu können

Probleme – Zuweisungsproblem (Seiten)

Problem taucht nicht auf, da alle Seiten gleich groß

↪ es entstehen immer passende Lücken

↪ **keine externe Fragmentierung**

Interne Fragmentierung: Einheitliche Seitengröße ↪ auf letzter Seite des Programm-Moduls wahrscheinlich ungenutzter Leer-raum

Probleme – Ersetzungsproblem (Segmentierung)

Limitierung der Anzahl gleichzeitig von einem Prozess benutzbaren Segmente

↪ bei Einlagerung eines neuen Segments wird anderes Segment des Prozesses ausgelagert

Probleme – Ersetzungsproblem (Seiten)

Seitenersetzungsstrategien:

1. **FIFO** (*first in first out*): älteste Seite wird ersetzt
2. **LIFO** (*last in first out*): jüngste Seite wird ersetzt
3. **LRU** (*last recently used*): am längsten unbenutzte Seite wird ersetzt
4. **LFU** (*least frequently used*): am seltensten benutzte Seite wird ersetzt
5. **LRD** (*least reference density*): Seite mit geringster Zugriffs-dichte wird ersetzt

Virtueller vs. physikalischer Cache

virtueller Cache: zwischen CPU und MMU, Tags = höherwertigste Bits der logischen Adressen

Vorteile:

- bei Treffer keine MMU nötig

physikalischer Cache: zwischen MMU und Speicher, Tags = höherwertigste Bits der physikalischen Adressen

Vorteile:

- kleinerer Tag, da physikalische Adresse idR viel kleiner als logische Adresse ist

Segmentorientierte Speicherverwaltung – Beispiel x86

Segment-Deskriptor: beschreibt sein Segment durch Attribute:

- **Segment-Basisadresse** (*base address*)
- **Segment-Größe** (*limit*) in Bytes
- **Zugriffsrechte** (*access rights*) auf das Segment

Speicherwort wird durch **Segment-Selektor** (kennzeichnet Segmentanfang) und **Offset** innerhalb des Segments adressiert

Segment-Selektoren wechseln selten, da durch Lokalitätseigenschaften von Programmen nicht bei jedem Hauptspeicherzugriff neues Segment benutzt wird

Segment-Selektoren in **Segment-Registern** gespeichert (wegen Geschwindigkeit)

Segment-Register:

- **Code-Segment** (CS)
- **Stack-Segment** (SS)
- **Daten-Segment** (DS)
- **Extra-Segment** (ES, FS, GS,...)

↪ Speicherwortadressierung durch Segment-Register + Offset

Segment-Deskriptor-Cache: Speichert wichtige Informationen über ausgewähltes Segment

↪ keine Speicherzugriffe beim Lesen von Segmenteigenschaften

Jedes Segment mit Anfangsadresse im Segmentregister ist physikalisch im Hauptspeicher

Arbeitsmenge (*working set*): durch Segmentregister spezifizierte Segmente eines Prozesses

Adressierungsmodi – Beispiel x86

Verschiedene Adressierungsmodi möglich (für Kompatibilität)

Real (Address) Mode (Kompatibilitätsmode zu 8086):

- physikalische Adressen: 20 Bit
- max. adressierbare Hauptspeicherkapazität: 1 MB
- max. Segmentlänge: 64 KB
- Abbildung virtueller Adressraum (mehrere TB) auf physikalischen Adressraum (4 GB)