

Grundlagen

AUFGABEN DER HARDWARE

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

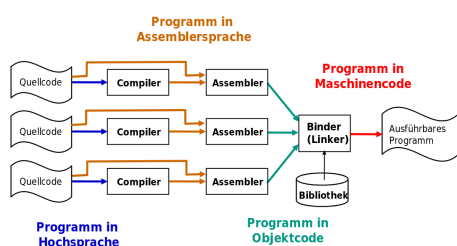
KLASSISCHE HARDWAREKOMponentEN

- Ein- und Ausgabe
- Hauptspeicher
- Rechenwerk
- Leitwerk

Anforderungen höherer Programmiersprachen

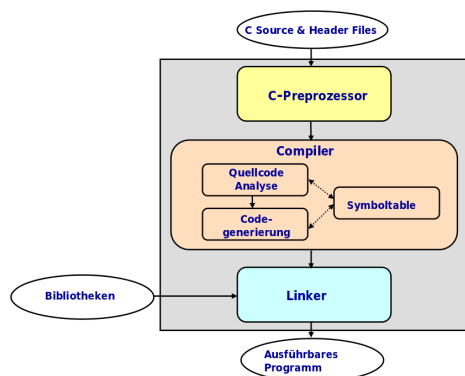
BEGRIFFE

- **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 ($x \cdot y$)
- **/**: 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 ($x \ll y$)
- **>>**: rechts schieben ($x \gg y$)
- **&**: bitweise AND ($x \& y$)
- **^**: bitweise XOR ($x \wedge y$)
- **|**: bitweise OR ($x | y$)
- alle auch als Assign (= anhängen)

C — VERGLEICHSOPERATOREN

- **>**, **<**: größer, kleiner als ($x > y$, $x < y$)
- **>=**, **<=**: größergleich, kleinergleich als ($x \geq y$, $x \leq y$)
- **==**, **!=**: 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	<code>() [] . -> expr++ expr--</code>	left-to-right
Unary Operators	<code>* & + - ! ~ ++expr --expr (typecast) sizeof</code>	right-to-left
Binary Operators	<code>* / %</code>	left-to-right
	<code>+ -</code>	
	<code>>> <<</code>	
	<code>< > <= >=</code>	
	<code>== !=</code>	
	<code>&</code>	
	<code>^</code>	
	<code> </code>	
	<code>&&</code>	
	<code> </code>	
Ternary Operator	<code>?:</code>	right-to-left
Assignment Operators	<code>= += -= *= /= %= >>= <<= &= ^= =</code>	right-to-left
Comma	<code>,</code>	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
<code>int *p;</code>	p	...
<code>int *q;</code>	a	0x8004
<code>int a = 3;</code>	b	3
<code>int b;</code>	b	...
	q	4
		...
<code>p = &a;</code>		0x8010
<code>b = *p + 1;</code>		0x8010
<code>q = (int*) 0x8010</code>		

Zahlendarstellung

ZAHLENSYSTEME — STELLENWERTSYSTEM

- Darstellung einer Zahl durch Ziffern z_i — Stellenwert i te Position: i te 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 $y_i = Z_i \div b^i, R_i = Z_i \bmod b^i$
 - (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
 $2^3 \leq 13 < 2^4$
 $13 : 2^3 = 1 \text{ Rest } 5$
 $5 : 2^2 = 1 \text{ Rest } 1$
 $1 : 2^1 = 0 \text{ Rest } 1$
 $1 : 2^0 = 1 \text{ Rest } 0$
 $\leadsto Z = 13_{10} = 1101_2$
2. **Horner-Schema:**

- (a) ganzzahliger Teil: 15741_{10} in Hexadezimal:
 $15741_{10} : 16 = 983 \text{ Rest } 13 (= D_{16})$
 $983_{10} : 16 = 61 \text{ Rest } 7 (= 7_{16})$
 $61_{10} : 16 = 3 \text{ Rest } 13 (= D_{16})$
 $3_{10} : 16 = 0 \text{ Rest } 3 (= 3_{16})$
 $\leadsto Z = 15741_{10} = 3D7D_{16}$
- (b) Nachkommateil: $0, 233_{10}$ in Hexadezimal:
 $0, 233_{10} * 16 = \underline{3}, 728$
 $0, 728_{10} * 16 = \underline{11}, 648$
 $0, 648_{10} * 16 = \underline{10}, 368$
 $0, 368_{10} * 16 = \underline{5}, 888$
 $\leadsto Z = 0, 233_{10} \approx 0, 3BA5_{16}$

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 b1 ZU BASIS b2

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
...			

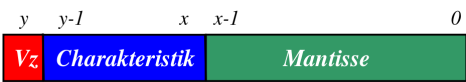
- **Vorteile:** Symmetrischer Zahlenbereich, keine gesonderte Betrachtung des ersten Bits
 - **Nachteile:** doppelte Darstellung der Null
3. **Zweierkomplement:** = Einerkomplement + 1

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

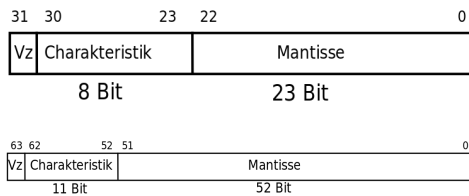
- **Vorteile:** Wie Einerkomplement, eindeutige Null
 - **Nachteile:** Asymmetrischer Zahlenbereich (eine negative Zahl mehr)
4. **Exzess-Darstellung:** Verschiebung nach oben derart, dass kleinste negative Zahl die Darstellung $0 \dots 0$ hat

DARSTELLUNG VON KOMMAZAHLEN

1. **Festkommazahlen:** Komma sitzt an einer festen Stelle
2. **Gleitkommazahlen:** $X = \pm \text{Mantisse} * b^{\text{Exponent}}$ (b fest)
 - $X = (-1)^{\text{Vorzeichen}} * (0, \text{Mantisse}) * b^{\text{Exponent}}$
 - Exponent = Charakteristik - $b^{(y-1)-x}$



3. **IEEE-Standard:**
 - (a) 32-Bit:
 - (b) 64-Bit:



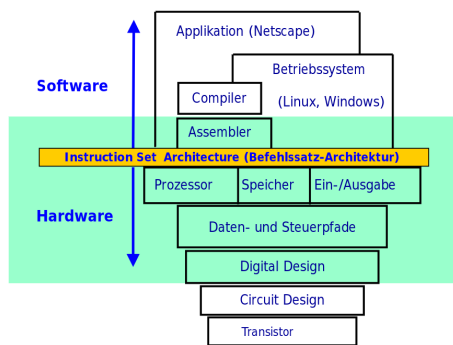
CODIERUNGEN

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

Befehlssatzarchitektur



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?
- \leadsto abstrahiert Hardware für den Maschinenprogrammierer

AUSFÜHRUNGSMODELLE

AUSFÜHRUNGSMODELL — REGISTER-REGISTER

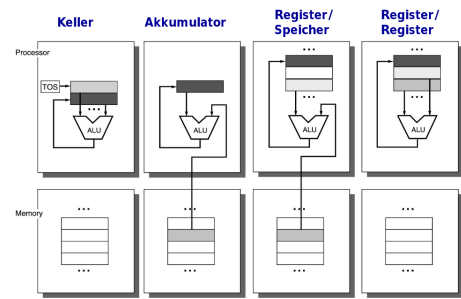
- Alle Operanden und Ergebnis stehen in Allzweckregistern
- **Load/Store**: Bestimmte Befehle holen Operanden aus Hauptspeicher, schreiben Inhalte von Registern in Speicher
- **Dreiadressformat**

```
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 \leadsto 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
```

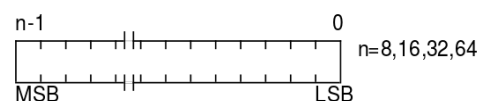
AUSFÜHRUNGSMODELLE — ÜBERSICHT

- $C=A+B$; $D=C-B$

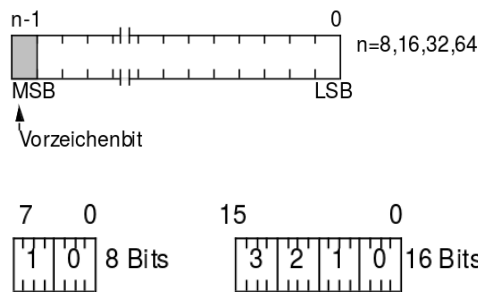
Register-Register	Register-Speicher	Akkumulator	Keller
<pre>load Reg1,A load Reg2,B Add Reg3,Reg1,Reg2 store C,Reg3 load Reg1,C load Reg2,B sub Reg3,Reg1,Reg2 store D,Reg3</pre>	<pre>load Reg1,A add Reg1,B store C,Reg1 load Reg1,C sub Reg1,B store D,Reg1</pre>	<pre>load A add B store C load C sub B store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

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:** Niegigstwertigstes 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

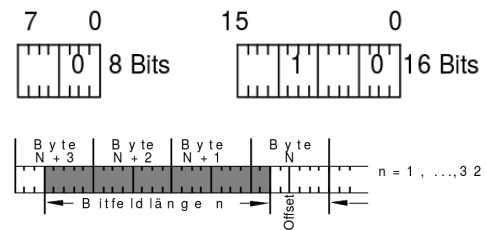
- legt Grundoperationen eines Prozessors fest
- Befehlsarten:
 1. Transport
 2. Arithmetik/Logik
 3. Schieben/Rotieren
 4. Multimedia
 5. Gleitkomma
 6. Programmsteuerung
 7. Systemsteuerung
 8. Synchronisation

INSTRUCTION SET — FORMATE

- **Befehlsformat** legt Befehlskodierung fest
- **Befehlskodierung:** [opcode] [parameter1] ...
- **Adressformate:** vier Befehlssatzklassen:
 1. Dreiaadressformat: [opcode] [dest] [src1] [src2]
 2. Zweiaadressformat: [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 \leadsto Programme lageabhängig
- **Heute:** *dynamische Adressberechnung:*
Programmadresse \leadsto logische Adresse \leadsto physikalische Adresse

ADRESSIERUNGSARTEN

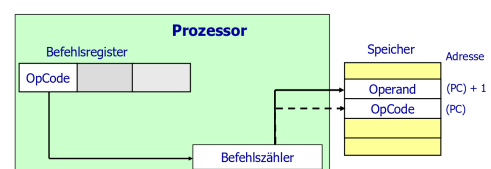
1. **Register-Adressierung**
 - (a) implizit: Flag
 - (b) explizit
2. **einstufige Speicher-Adressierung**
 - (a) unmittelbar
 - (b) direkt: absolut, Zero-Page, Seiten(-Register)
 - (c) Register-indirekt
 - (d) indiziert: Speicher-relativ, Register-relativ, Register-relativ mit Index
 - (e) Programmzähler-relativ
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

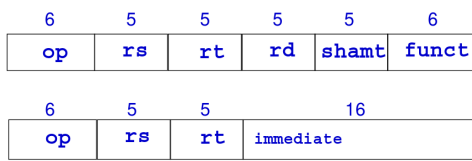
ADRESSIERUNGSARTEN — REGISTER-ADRESSIERUNG

- Operand steht bereits im Register \leadsto 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 ADRESSIERUNG

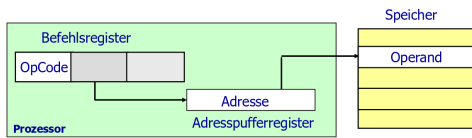
- Eine Adressberechnung ist zur Ermittlung der effektiven Adresse nötig \leadsto keine mehrfachen Speicherzugriffe zur Adressermittlung
- **Unmittelbare Adressierung:** Befehl enthält nicht Adresse des Operanden, sondern Operand selbst (Opcode und Operand stehen hintereinander im Speicher)
Beispiel: `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 (niederwertiger 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*)

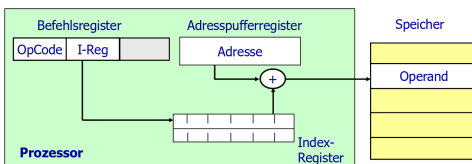
Beispiel: `LD R1, (A0)` (Lade Register `R1` mit Inhalt des in `A0` angegebenen Speicherwortes)

- 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 gegebenem Basiswert berechnet

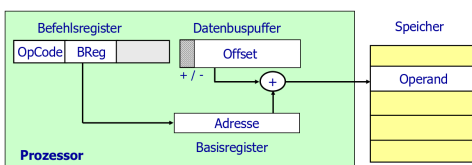
1. **Speicher-relative Adressierung:** Basiswert wird als absolute Adresse im Befehl vorgegeben

Beispiel: `ST R1, $A704 (R0)` (Speichere Inhalt von `R1` in Speicherwort, dessen Adresse man durch Addition des Inhalts von `R0` zu Basis `$A704` erhält)

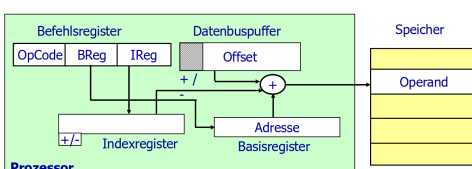


2. **Register-relative Adressierung:** Basiswert befindet sich in Basisregister, verwiesen im BReg-Feld des Opcodes

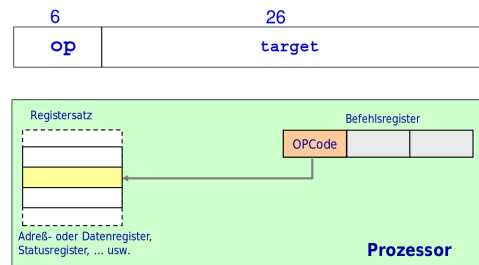
Beispiel: `CLR $A7 (B0)` (Lösche Speicherwort, dass man durch Addition von `$A7` zu `B0` erhält)



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 Hauptspe-



icher zu verschieben

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

Rechnermodell

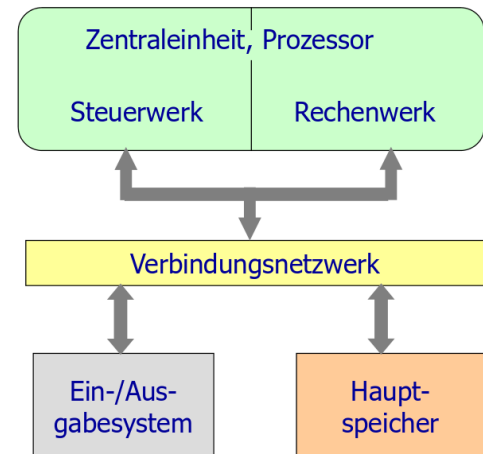
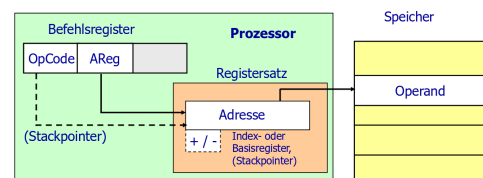
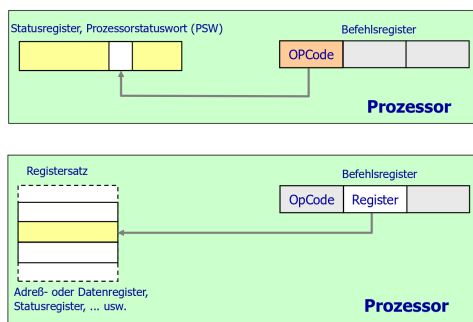
RECHNERMODELL — VON-NEUMANN-RECHNER

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* → Zustandsinformation in Kondensatoren gespeichert → 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 Ausgangsbus (Ergebnis)
 4. **Ergebnisse:** Werden entweder in Prozessor-Registern gespeichert, zu ALU-Eingangregistern 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

RECHNERMODELL — RISC

- = *reduced instruction set computer*
- Grundprinzipien:**
 - Vielbenutzte Befehle so schnell wie möglich machen (keine Mikroprogrammierung, Befehlspipeline)
 - Hauptarbeit durch optimierende Compiler
 - Operanden in großen Registersätzen halten (schneller Zugriff, schnelle Verarbeitung)
 - einheitliche Befehlsformate (schnelle Decodierung)
 - viel Pipelining
- Ziele:**
 - Jeder Befehl ein Taktzyklus
 - alle Befehle gleich lang
 - nur Load-Store und Register-Register-Befehle (wenige Adressierungsarten → schnelle Ausführung)
 - Koprozessorarchitektur für komplexe Befehle
- keine Ziele:** Unterstützung von Gleitkomma-Arithmetik oder Betriebssystemfunktionen
- Forderungen an RISC-Systeme:**
 - Mindestens 75% Ein-Zyklus-Befehle
 - Länge aller Befehle = Datenbusbreite
 - Nicht mehr als 128 Befehle
 - Nicht mehr als 4 Adressierungsarten
 - Load-Store-Architektur
 - Festverdrahtete Steuereinheit (keine Mikroprogramme)
 - Mindestens 32 allgemein verwendbare Register

RISC — PROZESSORAUFBAU

- Harvard-Architektur:** Programm- und Datenspeicher getrennt → paralleles Holen von Operanden und Instruktionen
- Varianten:**
 - zwei** getrennte Bussysteme bis zu Cachespeichern, nur ein Arbeitsspeicher (niedrigere Kosten)
 - 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-Schaltkreis 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

- Variante B:** Rechenwerk ohne Akkumulator - Akkumulator in Prozessor-Registersatz verlegt (Vorteil: Mehrere Register können Akkumulatorfunktionen übernehmen. Bei allen modernen 16/32 Bit-Prozessoren)

Operationen:

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

- Registersatz:** Zwischenspeicherung von häufig benutzten Operanden → schnellerer Zugriff als auf Hauptspeicher

- Dual-Port-Speicher** zwischen Ein- und Ausgangsbuss
→ ein Register lesen und anderes schreiben gleichzeitig möglich
- Zusatzfunktionen:** Inkrementieren, Dekrementieren, Nullsetzen, Inhaltsverschiebung
- Datenregister:** Zwischenspeichern von Operanden, bei modernen Prozessoren mehrere Datenregister als Akkumulator nutzbar
- 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)
- Spezielle Register:** Instruction pointer, Steuerregister, Statusregister, Interrupt-Behandlungsregister, Stackregister
- 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)
- Laufzeitstack:** LIFP-Speicher im Hauptspeicher, speichert Prozessorstatus und Programmzähler bei Unterprogrammaufruf/Unterbrechungsroutinen, Parameterübergabe, kurzzeitige Datenlagerung bei Ausführung, Datenübertragungsbefehle **PUSH** und **POP/PULL**

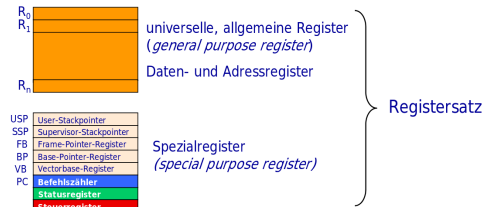
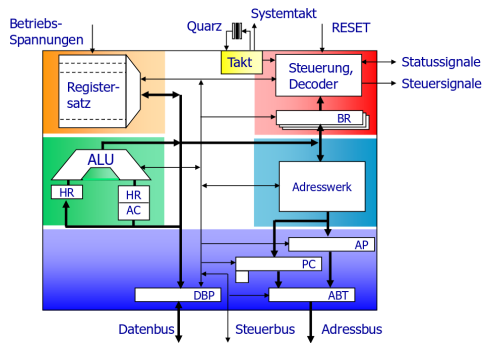
- Adresswerk:** Berechnet nach Steuerwerkvorschriften Adresse eines Befehls-/Operanden (früher Bestandteil des Rechenwerks, heute sehr komplex, da es viele verschiedene komplexe Adressierungsarten gibt)

Aufbau und Funktionsweise:

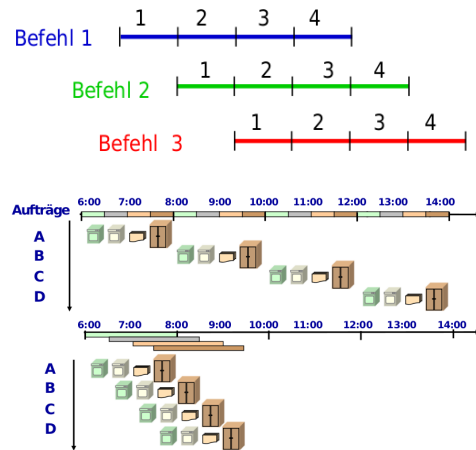
- Adress-Addierer - Eingang A:** Registersatz (Adresse aus Basis-/Indexregister), Datenbuspuffer (absolute Adresse im Befehl oder absolute Distanz zu Basisregister)
- Adress-Addierer - Eingang B:** Befehlszähler (Adresse des aktuellen Befehls), Adresspuffer (Adresse des aktuellen Operanden)
- Adressberechnung 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 → kurze Programme, Unterstützung höherer Programmiersprachen durch komplexe Befehle (Abbildung Sprachkonstrukt → 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 Mikroprogrammzebene, schwieriger Compilerbau
- 80/20-Regel:** Nur 20% der Befehle werden häufig verwendet
- Cycles per instruction (CPI):** Bei meisten heutigen CISC-Architekturen: CPI

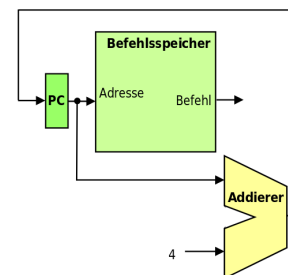


heißt er *superskalar* - pro Takt können mehrere Befehle den Ausführungseinheiten zugeordnet und so viele Befehlsausführungen beendet werden

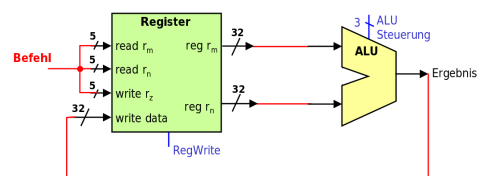


MIPS — BEFEHLSABARBEITUNG

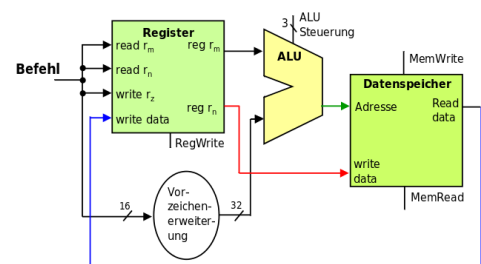
- **Befehl holen:** Wird bei jedem Befehlstyp benötigt
 1. Befehl im Befehlsspeicher adressieren
 2. Befehlszähler (PC) um 4 inkrementieren



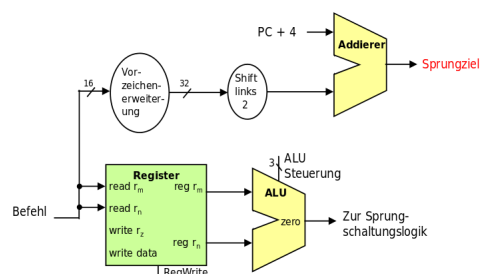
- **Typ R:** Lesezugriff auf beide Operandenregister, Schreibzugriff auf Zielregister



- **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



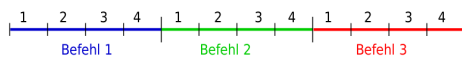
- **Typ J:** 16-Bit Offset \leadsto bis zu $2^{15} - 1$ Befehle vorwärts/ 2^{15} rückwärts (Basisadresse zur Sprungberechnung = Befehlsadresse hinter Verzweigungsbefehl (PC+4), z.B. `beq r_n, r_m, offset`: Vergleich von r_n, r_m entscheidet ob Sprung genommen wird)



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

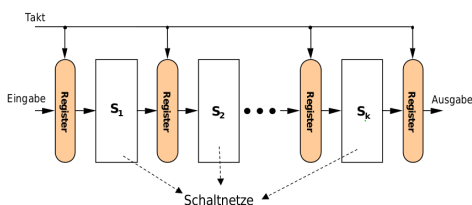
DEFINITION

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

\leadsto Gesamtheit Verarbeitungseinheiten = Pipeline

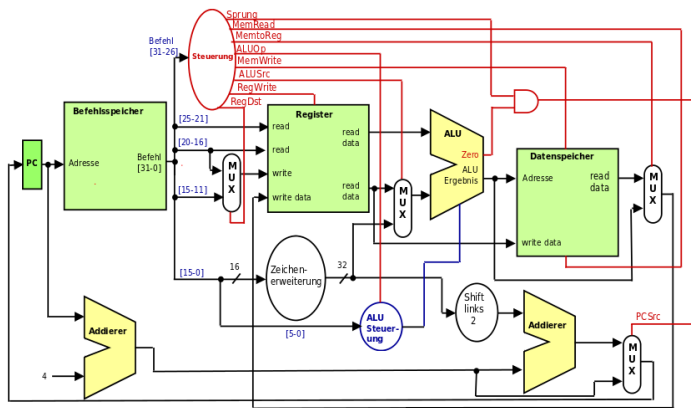
BEGRIFFLICHKEITEN

- **Stufen, Register:**



- **Verzögerungszeiten:** Schaltnetze: τ_i ($i = 1, \dots, k$), Register: τ_{reg}
- **Pipeline-Maschinentakt:** Benötigte Zeit, um Befehl eine Stufe weit-erzuschieben (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: $D = \frac{n}{(k+n-1) \cdot \tau}$ ($\tau : \max\{\tau_1, \dots, \tau_k\} + \tau_{reg}$) - $\lim_{n \rightarrow \infty} D = D_{max} = \frac{1}{\tau}$
- **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 - $\lim_{n \rightarrow \infty} S = k$)

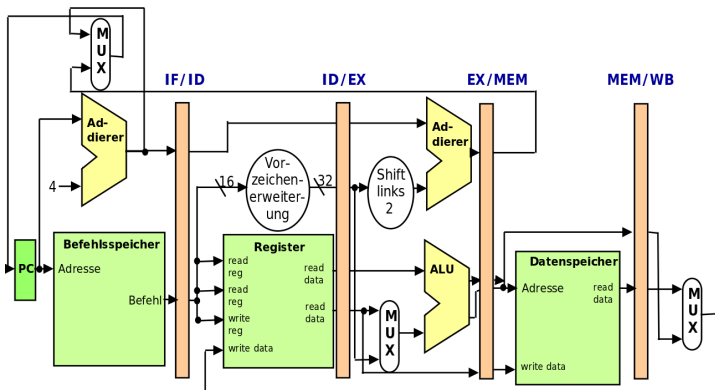
MIPS-Datenpfad: Zusammenführung der drei Modelle



MIPS — PIPELINING: DLX-PIPELINE

(DLX = deluxe)

- Pipelinstufen, Buffering** (siehe Grafik "Begrifflichkeiten: Stufen, Register")



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

MIPS — PIPELINE-KONFLIKTE

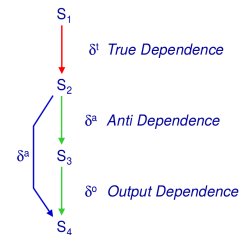
- Datenkonflikte:** Benötigter Operand ist in der Pipeline (noch) nicht verfügbar (Datenabhängigkeiten im Befehlsstrom)
- Struktur-/Ressourcenkonflikte:** Zwei Pipeline-Stufen benötigen selbe Ressource, aber nur einfacher Zugriff möglich
- Steuerflusskonflikte:** Bei Programmsteuerbefehlen
 - Zieladresse des nächsten Befehls in Befehlsbereitstellungsphase noch nicht berechnet
 - Bedingter Sprung: noch unklar, ob gesprungen werden muss

DATENABHÄNGIGKEITEN/-KONFLIKTE

- Datenabhängigkeit:** Zwischen zwei aufeinanderfolgenden Befehlen $Inst_1$, $Inst_2$ kann eine *Datenabhängigkeit* bestehen:
 - Echte Datenabhängigkeit** (true dependence, δ^t): $Inst_1$ schreibt Ausgabe in Register, dass die Eingabe von $Inst_2$ ist
 - Gegenabhängigkeit** (antidependence, δ^a): $Inst_1$ liest von einer Stelle, die von $Inst_2$ überschrieben wird
 - Ausgabeabhängigkeit** (output dependence, δ^o): $Inst_1$ und $Inst_2$ schreiben Ergebnis in selbes Register, aber $Inst_2$ nach $Inst_1$

Beispiel:

$S_1: \text{add } r1, r2, 2 \quad \#r1 := r2+2$
 $S_2: \text{add } r4, r1, r3 \quad \#r4 := r1+r3$
 $S_3: \text{mul } r3, r5, 3 \quad \#r3 := r5*3$
 $S_4: \text{mul } r3, r6, 3 \quad \#r3 := r6*3$

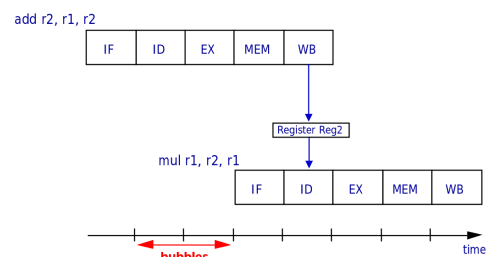


Datenkonflikte:

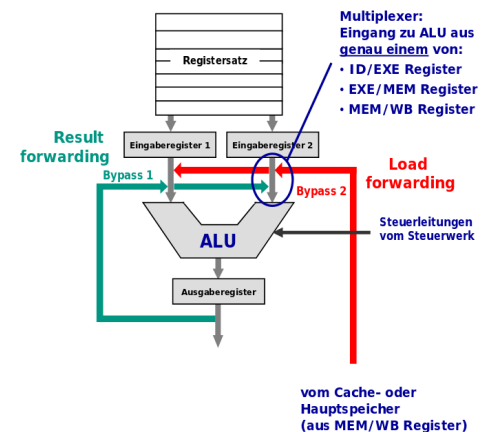
- Lesenach-Schreibe-Konflikt** (read after write, RAW): Verursacht durch echte Abhängigkeit
 - Schreibnach-Lese-Konflikt** (write after read, WAR): Verursacht durch Gegenabhängigkeit, tritt auf, wenn Schreib- vor Lesestufe in Pipeline
 - Schreibnach-Schreibe-Konflikt** (write after write, WAW): Verursacht 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:

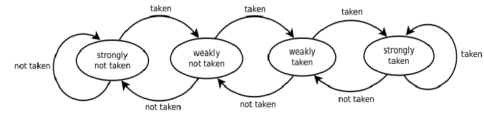
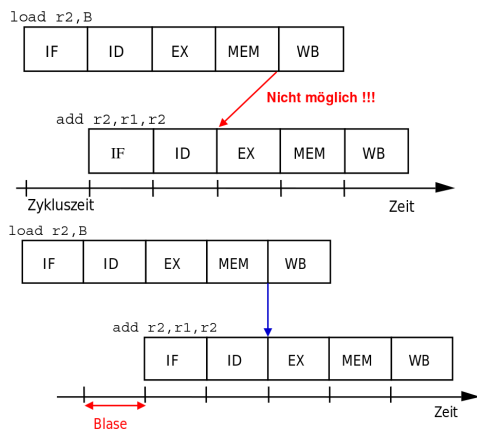
- Software-Lösung:**
 - Compiler erkennt Datenkonflikte, fügt nach konfliktverursachenden Befehlen Leeroperation (`noop`) ein
 - Statische Befehlsanordnung: Befehle werden umgeordnet (Optimierung), um Leeroperationen zu minimieren (Instruction Scheduling, Pipeline Scheduling)
- Hardware-Lösung:**
 - Pipeline leerlaufen lassen: Pipeline-Sperrung (*interlocking*) bzw. Pipeline-Leerlauf (*stalling*)



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



- Forwarding with interlocking:** Forwarding löst nicht alle möglichen Datenkonflikte

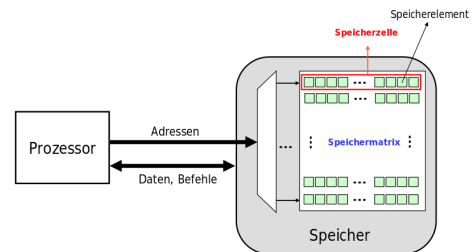


(e) Moderne Sprungvorhersagetechniken: In mehr als 90% der Fälle richtig

Speicher

BEGRIFFE

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

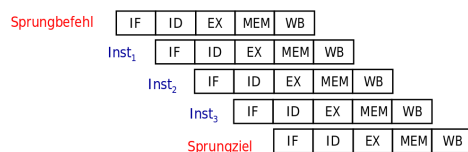


RESSOURCENKONFLIKTE

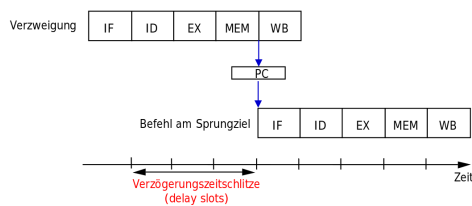
- Treten bei einfachen Modellen wie einer DLX-Pipeline nicht auf
- **Lösungen:**
 1. **Arbitrierung mit Interlocking:** Arbitrierungslogik hält konfliktverursachenden Befehl an \leadsto Verzögerung
 2. **Übertaktung:** Konfliktverursachende Ressource wird schneller getaktet als übrige Pipeline-Stufen
 3. **Ressourcenreplikierung:** 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

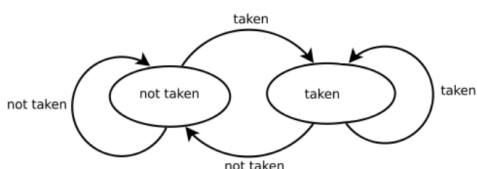


- \leadsto nach genommenem Sprung müssen Wartezyklen eingefügt werden (die ggf. durch Umstrukturierung der Pipeline verringert werden können)



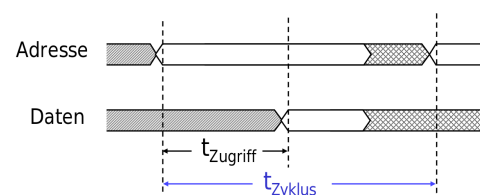
Lösungen:

1. **Software-Lösung:**
 - (a) Einfügen von **noop** in die Wartezyklen
 - (b) 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:**
 - (a) 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
 - (b) Spekulation auf nicht genommene bedingte Sprünge - wird Sprung doch genommen Pipeline leeren (*Pipeline flushing*)
 - (c) 1-Bit-Prädiktor zur Sprungvorhersage

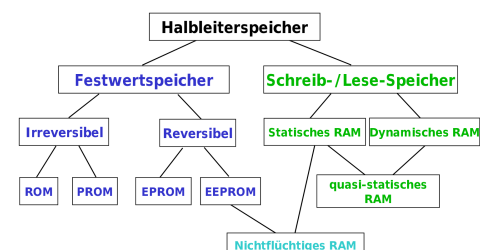


(d) 2-Bit-Prädiktor zur Sprungvorhersage

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

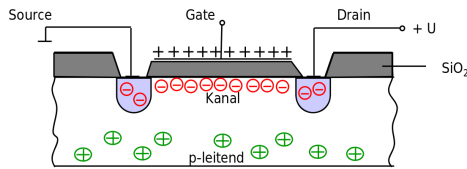


SPEICHERKLASSIFIZIERUNG



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*)

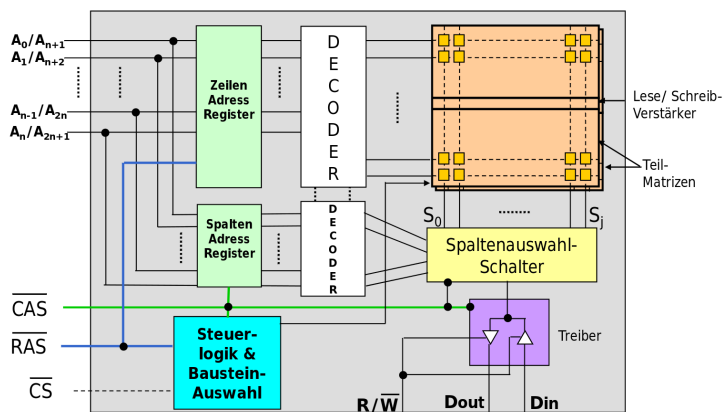


SPEICHERZELLE — STATISCH (SRAM)

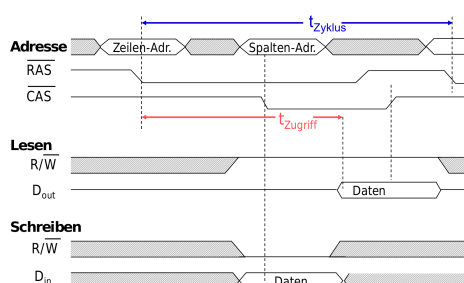
- Aufgebaut aus zwei kreuzweise rückgekoppelten Invertern und zwei Transistoren zur Ankopplung an Bitleitungen \leadsto 6-Transistor-Zelle
- **Vorteile:** Strom fließt nur zum Umschaltzeitpunkt \leadsto kein Refresh nötig
- **Nachteile:** Hoher Platzverbrauch

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

- = *synchrone 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

SDRAM

- = *sync link* SDRAM
- Weiterentwicklung SDRAM
- Höhere erlaubte Busfrequenzen \leadsto 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

