Rechnerarchitektur 2006

Prof. Dr. Konrad Froitzheim

Cables and Connectors - that is what computer architecture is about
[C.Gordon Bell]

Thema

- Rechnen
  - addieren, subtrahieren, multiplizieren, dividieren, …
  - konvertieren (umrechnen), bewegen, speichern, …
- Instruktionen
  - vom Compiler erzeugt, von der Hardware verstanden
  - Assemblerprogrammierung
  - Hardwaremodell
- Instruktionen ausführen
  - Speicher, ALU, …
  - dekodieren
  - Ablauf steuern
- Archtekturbeispiele
  - CISC, RISC
  - VLIW, superskalar, Pipeline
  - Signalprozessor
  - PIC, Controller, …

Formales

Termine:

Vorlesung: Mittwoch 14:00 – 15:30, WEI-1051

Dramatis Personae:

Prof. Dr. Konrad Froitzheim

Professur Betriebssysteme und Kommunikationstechnologien

frz@informatik.tu-freiberg.de

Helge Bahmann

hcb@chaoticmind.net

Vorlesungsunterlagen (kein Scriptum):

http://ara.informatik.tu-freiberg.de/Vorlesungen/comparch2006.doc

Prüfung: Klausur in der vorlesungsfreien Zeit

Literatur

50 Years of Computing; IEEE Computer, 10/96

Brinkschulte, U., Ungerer, T.: Mikrocontroller und Microprozessoren; 2002.

Foster, C.: Computer Architecture; Academic Press, New York

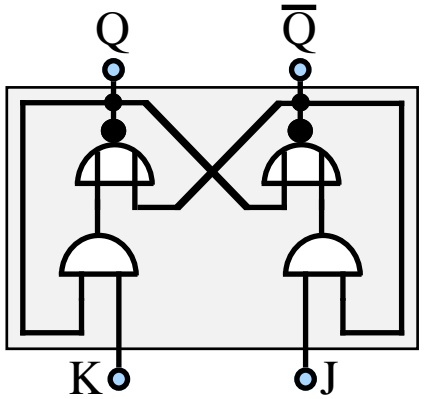
Patterson, D., Hennessy, J.: Computer Organization and Design, 1998.

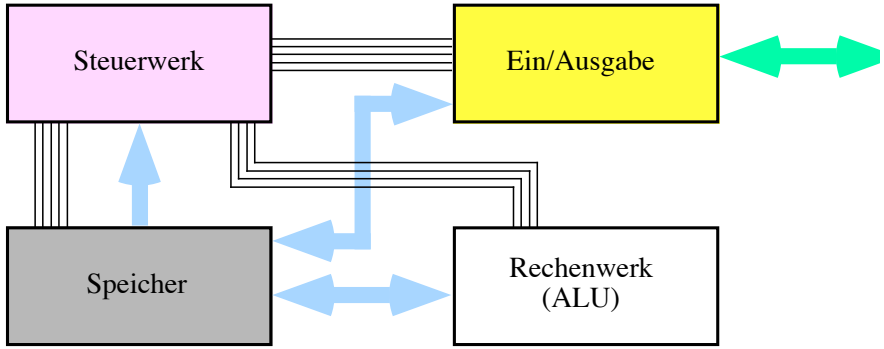
Patterson, D., Hennessy, J.: Computer Architecture: A Quantitative Approach; 2003.

Silc, J., Robic, B., Ungerer, T.: Processor Architecture - From Dataflow to Superscalar and Beyond; Springer-Verlag 1999.

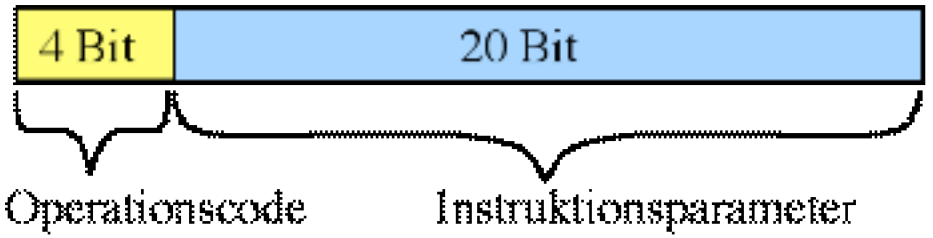
tecCHANNEL: Prozessortechnologie; München, 2004.

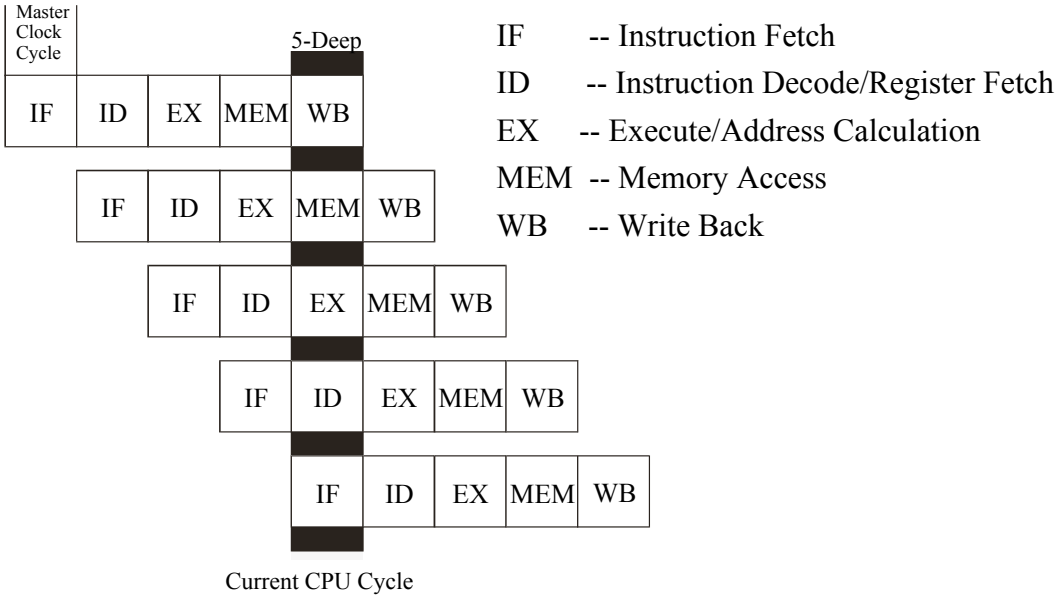
Inhaltsübersicht





2. Instruktionssätze
   2.1 Adressierung und Operanden
   2.2 Operationen
   2.3 CISC
   2.4 RISC
3. Pipelining
   3.1 Instruktionen laden
   3.2 Branch prediction und EPIC
   3.2 Ausführung
   3.4 Superscalare Architecturen
4. Speicherarchitektur
   4.1 Hauptspeicher
   4.2 Virtueller Speicher
   4.3 Caches





0. Transistoren und Instruktionen

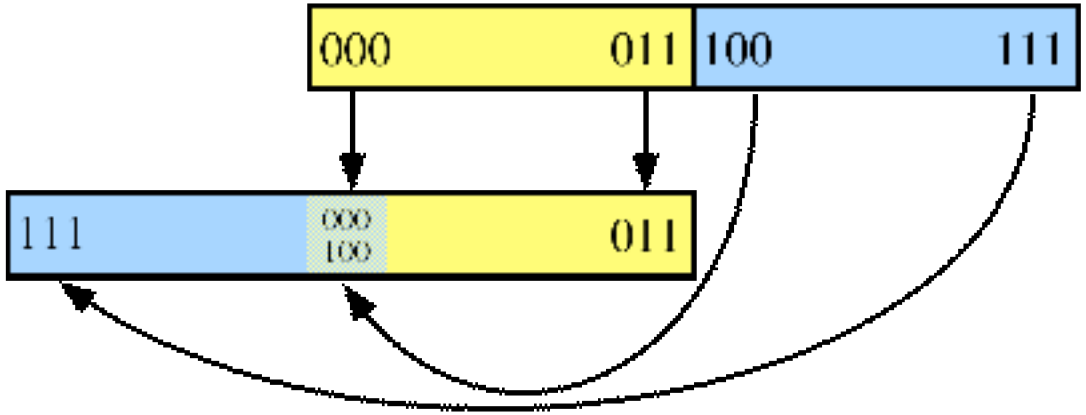
0.1 Zahldarstellung

0.1.1 Ganze Zahlen

- Polyadisches Zahlsystem
  - $z = \sum_{i=0}^{n-1} a_i B^i$
  - $0 \le a_i < B$
  - Basis 10: $1492 = 2*10^0 + 9*10 + 4*100 + 1*1000$
  - Basis 2: $1492 = 1*1024+0*512+1*256+1*128+1*64$ $+0*32+1*16+0*8+1*4+0*2+0*1 = 10111010100$
  - Basis 16: $1492 = \$5D4 = 5*256 + 13 * 16 + 4 * 1$
- Zahlenbereich
  - 10 Bit => 0..1023 = 1 'Kilo' -1
  - 20 Bit => 0..1024*1024-1 = 1 'Mega' -1
  - 32 Bit => 0..4 294 967 295 (2^{32}-1) = 4 'Giga' - 1
- negative Zahlen?

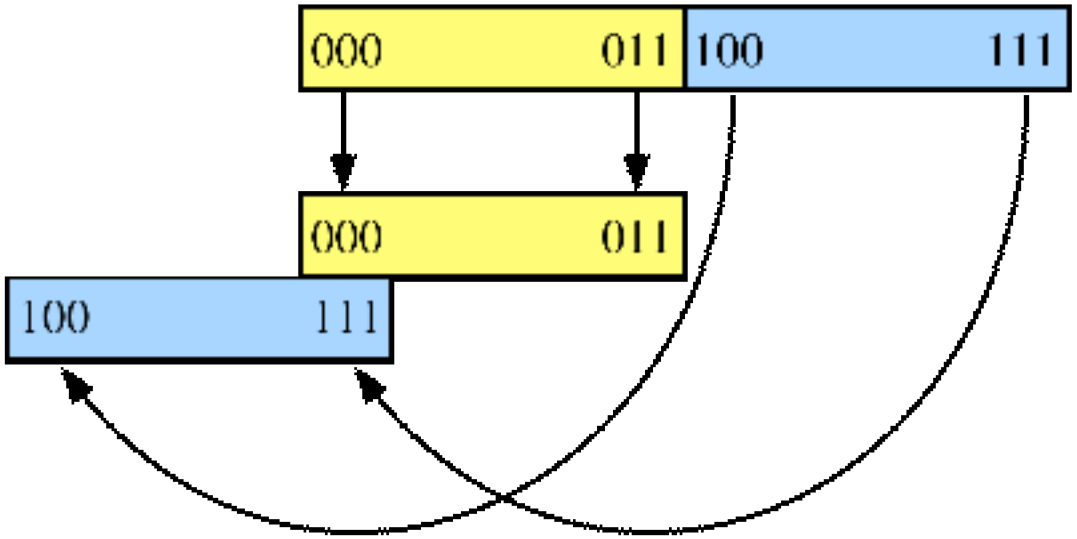
- Vorzeichen-Betrag (sign-magnitude)
  - 1 Bit Vorzeichen
  - höchstes Bit

  - 8 Bit (256 Werte): -127..127



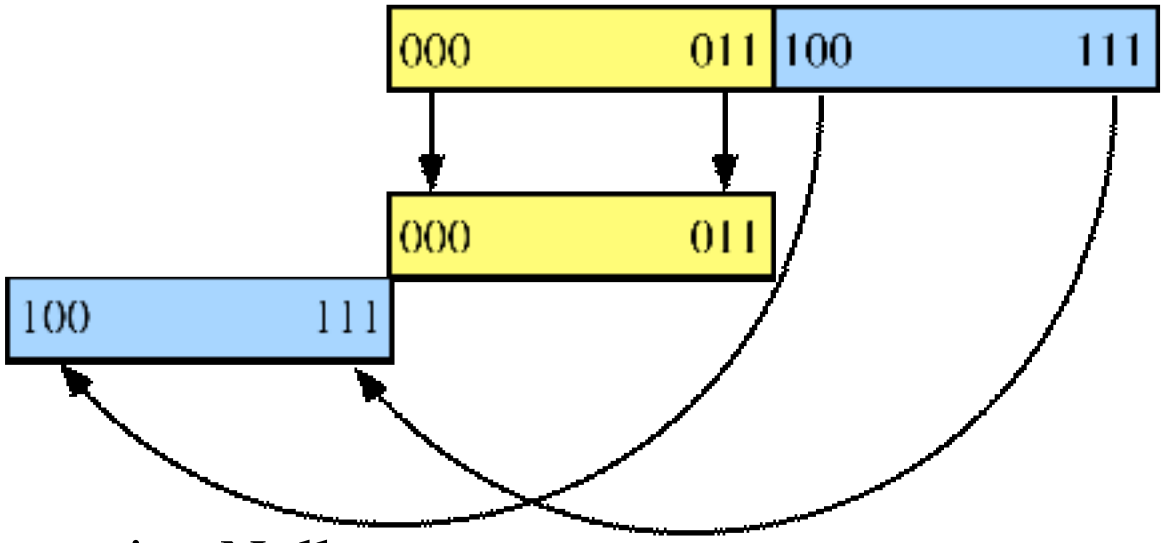
  - komplexe Rechenregeln

- Stellenkomplement
  - jede Stelle 'negieren': 0->1, 1->0
  - 00111111 = 63
  - 11000000 = -63



  - negative Null: 00000000 und 11111111
  - -127..0,0..127
  - besondere Rechenregeln

- Basiskomplement
  - jede Stelle negieren, 1 addieren
  - 00111111 = 63
  - 11000000+1= 11000001= -63



  - nur eine Null
  - Umrechnung macht mehr Arbeit
  - Rechenregeln einfach

0.1.2 Rationale und Reelle Zahlen

- Rationale Zahlen
  - Brüche: 1/2, 1/3, …
  - Dezimalschreibweise ggg,ddd: 0,5; 12,625
  - evtl unendlich viele Stellen: 16/3
  - ggggg,ddddd… : 1,33333333333333333333...
  - ggg,ddd = ggg + 0,ddd
  - ganzzahliger Teil + Bruchteil

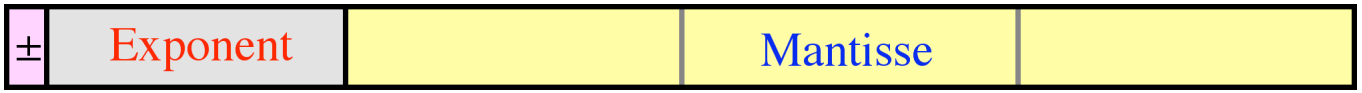
±	64	32	16	8	4	2	1

$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$

- Näherungsdarstellung von reellen Zahlen
  - $\pi \approx 3{,}1415926$
  - $\sqrt{2} \approx 1{,}414213562$
  - wann sind diese Fehler nicht katastrophal?

  => numerische Mathematik

- Normalisieren
  - ggg,ddd = 0,gggddd * 10^3
  - 1,34567 = 0,134567 * 10^1
  - 0,012345 = 0,12345 * 10^{-1}
  - 0,0000001 = 0,1 * 10^{-6}
- Floating-Point Zahlen
  - 0 ≤ Mantisse < 1
  - 10^{exp} "Exponent"
  - Vorzeichen



- Trick
  - normalisieren => 0,10111010101010 *2^{exp}
  - erstes Bit immer = 1 => weglassen
- typische Formate
  - ANSI/IEEE 754-1985
  - single: 32 bit - 23 bit Mantisse, 8 bit Exponent
  - double: 64 bit - 52 bit Mantisse, 11 bit Exponent
  - extended: 80 bit

0.2 Rechnen

- Addieren im Zehnersystem
  - stellenweise addieren

```
   1513
+  2112
   3625
```

  - Übertrag

```
   1523
+  2192
   3715
```

  - Rechenregeln für Übertrag 7+8 = 5 + Übertrag 1
- Binärer Fall
  - stellenweise addieren

```
   01001010
+  00101111
   011110001
```

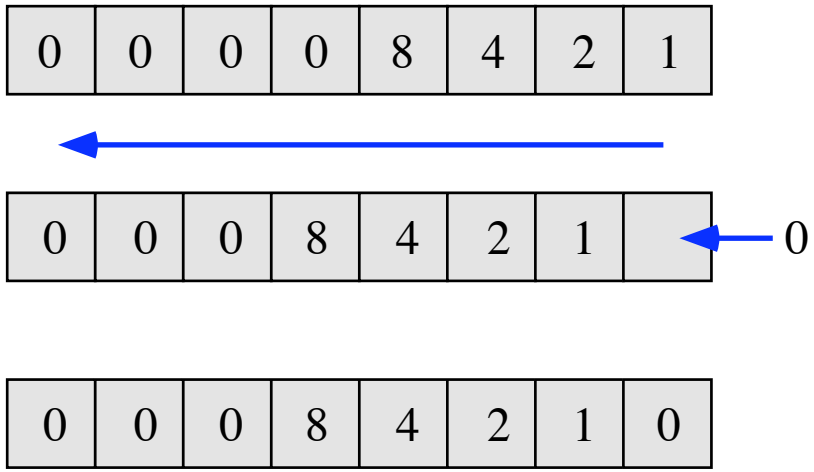
  - Rechenregeln einfach: 0+0=0, 0+1=1, 1+0=1, 1+1=0 Übertrag 1

- Beschränkte Stellenzahl
  - größte darstellbare Zahl
  - Ergebnis grösser: Überlauf

    11001010
    + 10101111
    01111001
  - eventuell negative Zahl

    01001010
    + 01101111
    10111001
- Subtrahieren
  - Komplement addieren: a-b = a+(-b)
  - besondere Regeln zur Ergebnisinterpretation

- Multiplikation
- Primitives Verfahren
  - Multiplikand * Multiplikator
  1. erg auf Null setzen
  2. erg = erg + Multiplikand
  3. Multiplikator um 1 herunterzählen
  4. falls Multiplikator > 0: weiter mit 2
- Sonderfall
  - Verschieben um eine Stelle, Null nachziehen
  => Multiplikation mit Stellenwert



- Multiplizieren
  - Multiplikand * Multiplikator
    1. i auf Eins setzen
    2. Addieren von (Multiplikand * Stelle i des Multiplikators)
    3. Verschieben des Multiplikanden (mit Stellenwert multiplizieren)
    4. i hochzählen
    5. weiter mit 2, falls i ≤ Stellenzahl Multiplikator
  - im Zehnersystem:

```
1214  *    211
------------
   1214
  12140
 242800
------------
 256154
```

- Trick:
  - Multiplikator in jedem Schritt eine Stelle nach rechts schieben
  - letzte Stelle in Schritt 2 verwenden

- binär Multiplizieren
  - ShiftLeft = Multiplikation mit 2
  - Stellen nur 0 oder 1 => Multiplikand addieren oder nicht
- Verfahren
  1. Ergebnis = 0; i = 0; a = Multiplikand; b = Multiplikator
  2. falls letzte Stelle von b = 1: Ergebnis = Ergebnis + a
  3. ShiftLeft(a)
  4. ShiftRight(b)
  5. Falls b>0 : weiter mit 2
- Beispiel: 12 * 5

Iteration	a	b	erg
0	0000 1100	0000 0101	0000 0000
			+ 0000 1100
1	0001 1000	0000 0010	0000 1100
			nix tun
2	0011 0000	0000 0001	0000 1100
			+ 0011 0000
3	0110 0000	0000 0000	0011 1100

0.3 Transistoren, Gates

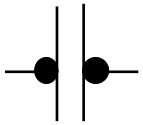
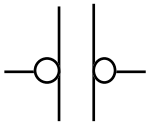
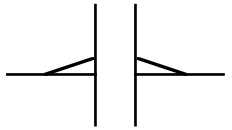
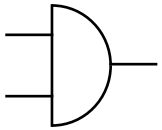
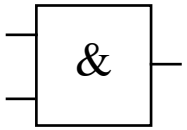
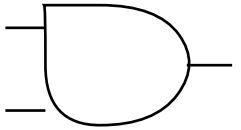
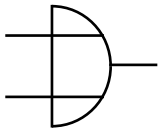
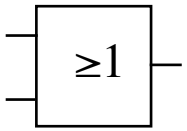
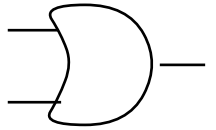
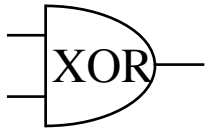
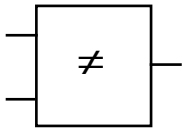
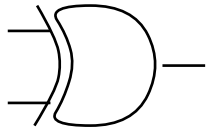
- Schaltfunktionen
  - 1..n Eingänge, 1..m Ausgänge
  - 2^n Argumentkombinationen
  - 2^{2^n} mögliche Funktionen

V_1	V_2	V_3	…	V_n	f_1	f_2	…	f_{max}
1	1	1		1	0	0		1
0	1	1		1	0	0		1
0	0	1		1	0	0		1
0	0	0		0	0	1		1

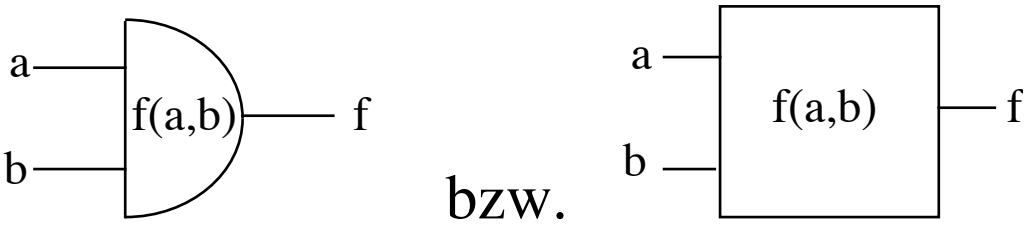
- 2-stellige Schaltfunktion

		a=1, b=1	a=1, b=0	a=0, b=1	a=0, b=0
f_0		0	0	0	0
f_1		0	0	0	1
…		…	…	…	…
f_8	AND	1	0	0	0
f_{14}	OR	1	1	1	0
f_{15}		1	1	1	1

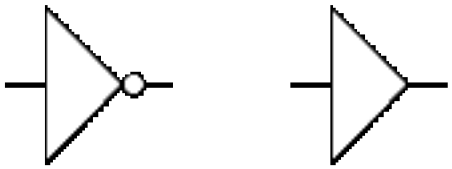
- Symbolische Darstellung von Schaltkreisen
  - Schaltplan
  - DIN 40700, IEEE/ANSI Y32.14, IEC

Funktion	Operator	graphisches Symbol		
		DIN	IEEE	IEC
Negation	$\neg$, $\overline{\ }$			
Und	·,∧, *			
Oder	+,∨			
Exklusiv Oder	⊕,≠			

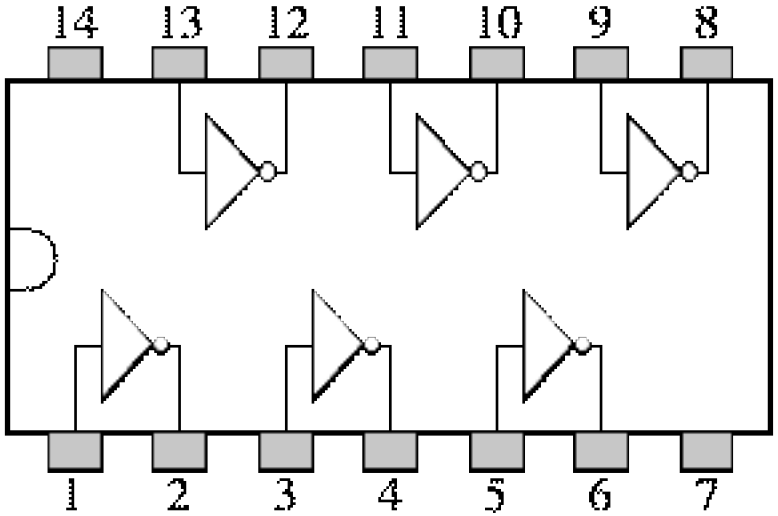
- Allgemeine Schaltfunktionen:



- Inverter, Verstärker:

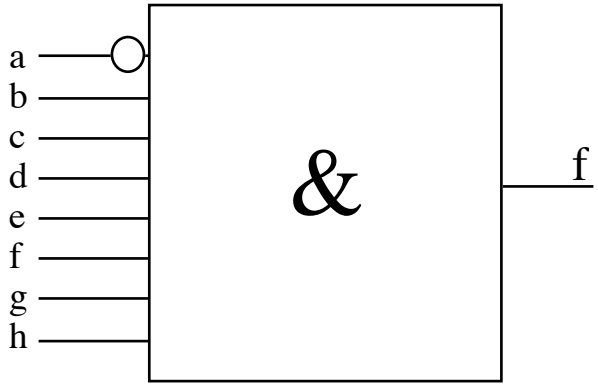


- Hex Inverter 7404 als Chip:
  - Stift #7: 0 Volt, Erde, GND
  - Stift #14: z.B. +5 Volt:



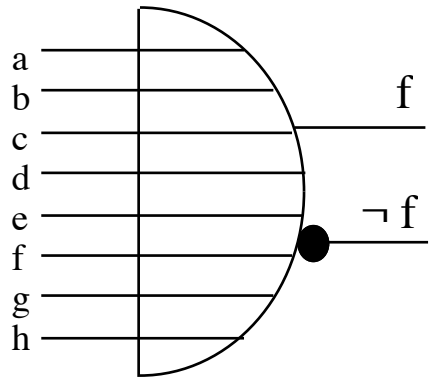
- Komplexere Beispiele:

f = $\bar{a}$ ∧ b ∧ c ∧ d ∧ e ∧ f ∧ g ∧ h :

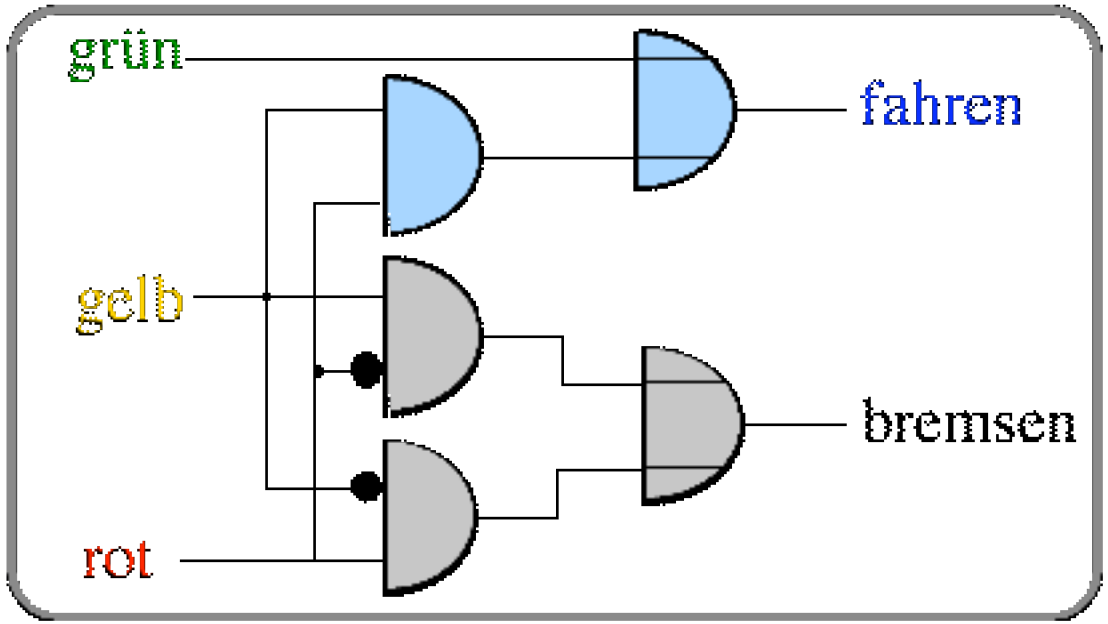


$$\bar{f} = \overline{a \vee b \vee c \vee d \vee e \vee f \vee g \vee h}$$

f = a ∨ b ∨ c ∨ d ∨ e ∨ f ∨ g ∨ h:

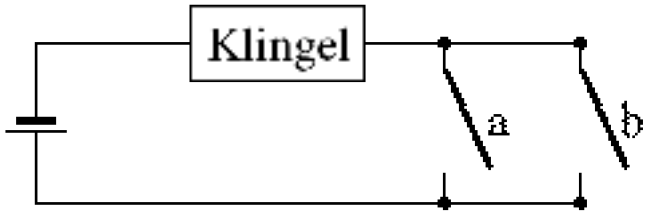


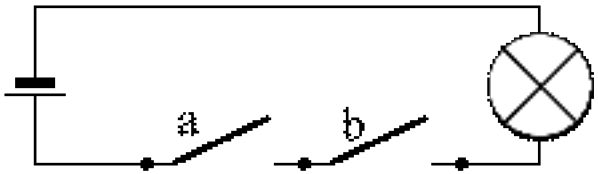
- Ampel-Reaktion

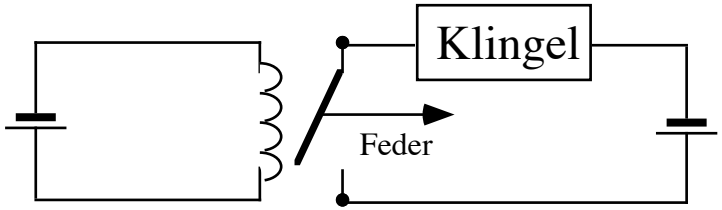


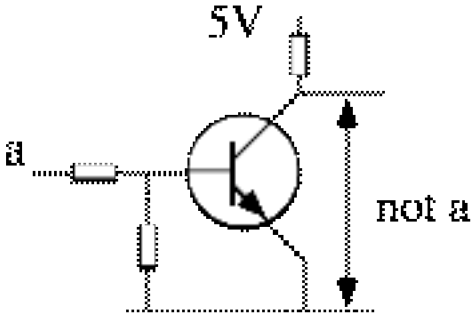
- Schaltalgebra
  - Schaltfunktionen
  - Rechenregeln
  - Normalform
  - Minimierung

- Diode:
  - Diode lässt elektrischen Strom nur in einer Richtung durch.
  - Positive Ladungsträger dürfen in Pfeilrichtung fließen.
- ODER (OR) Schaltung
  - Disjunktion, logische Summe, Vereinigung
  - a ∪ b
- UND (AND) Schaltung
  - a ∩ b
  - Konjunktion, logisches Produkt, Durchschnitt
- Negation (NOT)



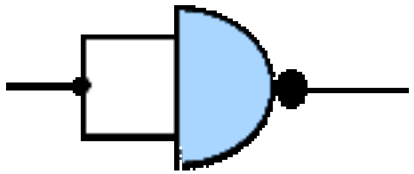




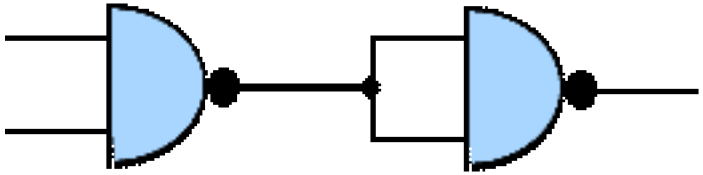


- Boolsche Algebra
- Kombination von NICHT mit UND / ODER
  - $\overline{A \wedge B} = \overline{A} \vee \overline{B}$
  - $A \vee B = \overline{\overline{A} \wedge \overline{B}}$
- NAND und NOR einfacher zu bauen
  - Feldeffekt-Transistoren (FET)
  - CMOS: complementary Metal-Oxide-Silicon
  - 4 Transitoren in CMOS-Technik

Inverter aus 1 NAND



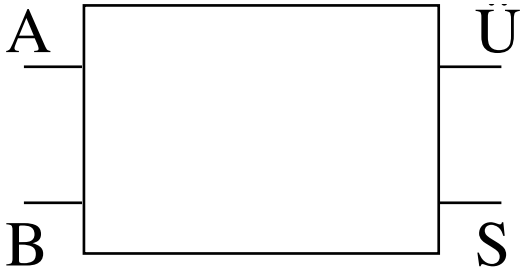
UND aus 2 NANDs



- 74LSxx, 74HCTxx, …
  - Einfache Logik-Chips
  - 7404: 6 Inverter
  - 7400: 4 NAND-Gatter
  - 7402: 4 NOR-Gatter
  - 7420: 2 NAND-Gatter mit je 4 Eingängen

0.4 Rechnen und Speichern

- Addition von zwei einstelligen Binärzahlen
  - Eingangsvariablen
  - Summenausgang
  - Übertrag zur nächsten Stufe

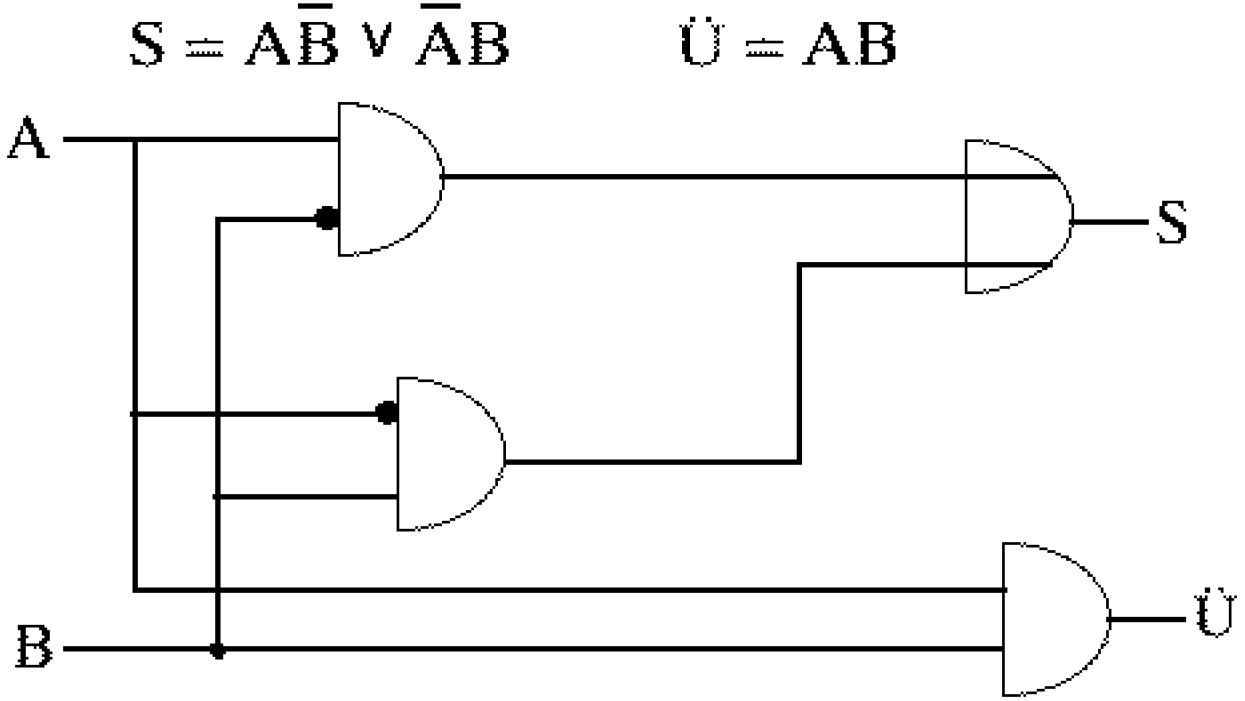


- Wahrheitstafel

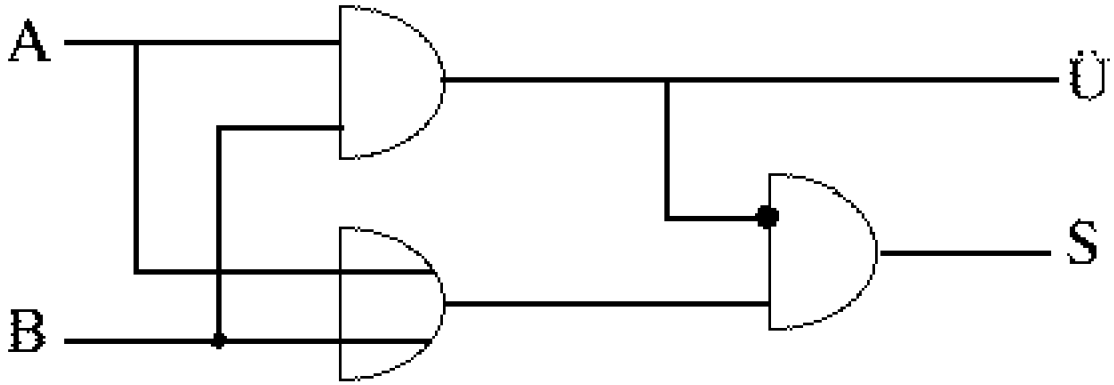
A	B	S	Ü
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- Addierer kann aus sogenannten Halbaddieren zusammengesetzt werden
- Halbaddierer berücksichtigt nicht den Übertrag der früheren Stufe

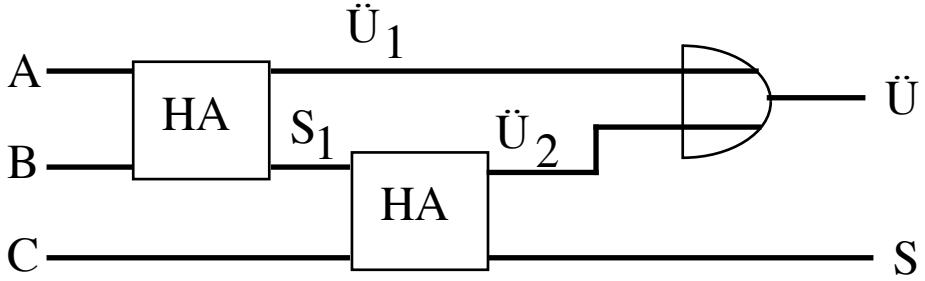
- Halbaddierer



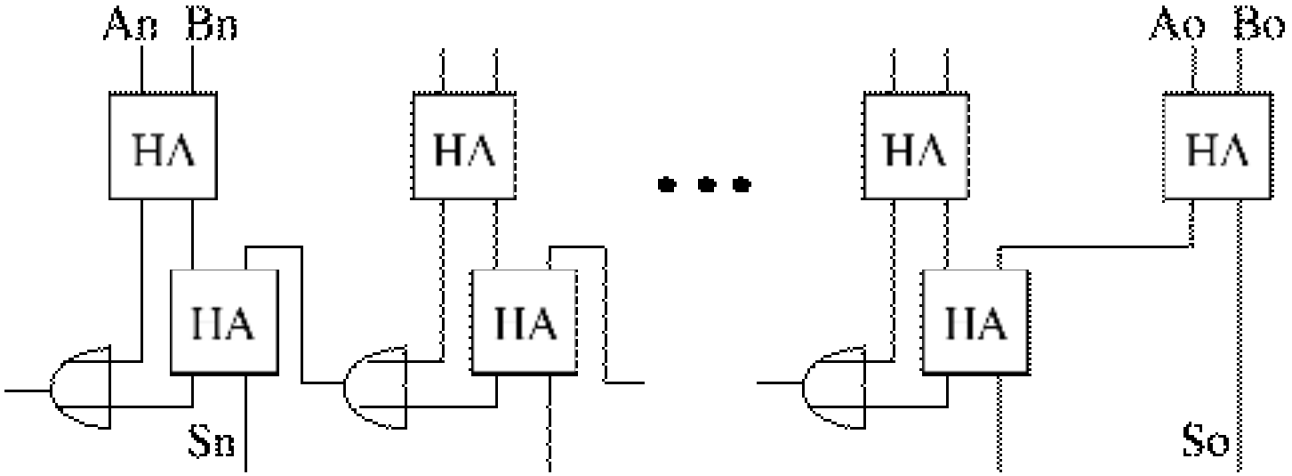
- Vereinfacht



- Volladdierer
  - für eine Binärstelle
  - berücksichtigt auch den Übertrag einer früheren Stufe



- Paralleladdierer
  - für die Addition eines Binärwortes
  - die Summen der jeweiligen Binärstellen parallel bilden
  - Übertrag durch die Stufen fortpflanzen lassen (Delay!)

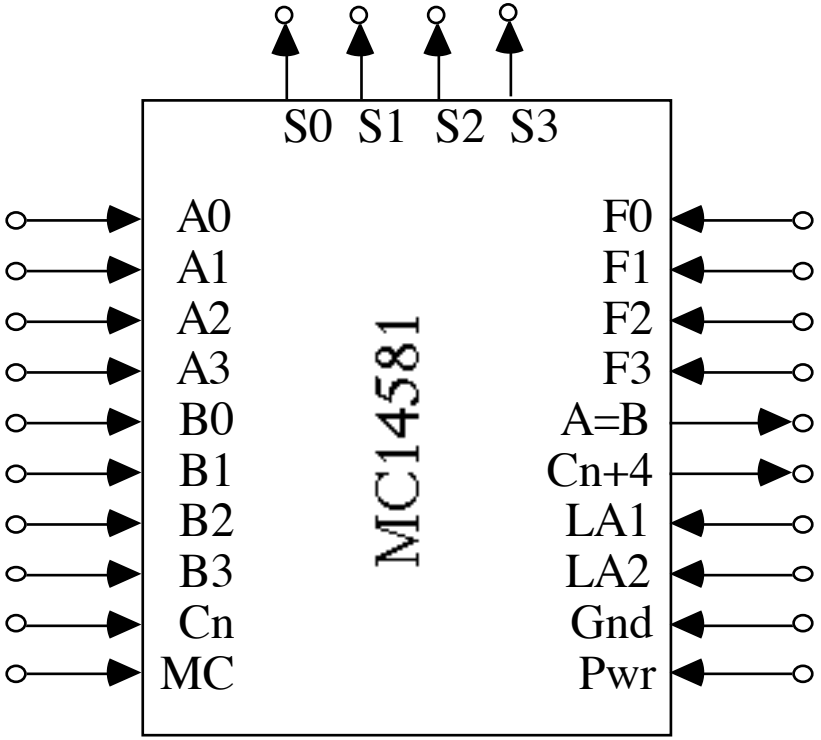


  -> Carry Lookahead

- CMOS MC14581 4-Bit ALU
  - 4 bit parallel
  - Steuerung über F0..F3, LA1, LA2
- Logische Funktionen:

$A \quad A \wedge B \quad A \vee B \quad A \neq B \quad True \quad False$

$\overline{A} \quad \overline{A} \wedge B \quad \overline{A} \vee B \quad \overline{A \neq B} \quad \ldots$



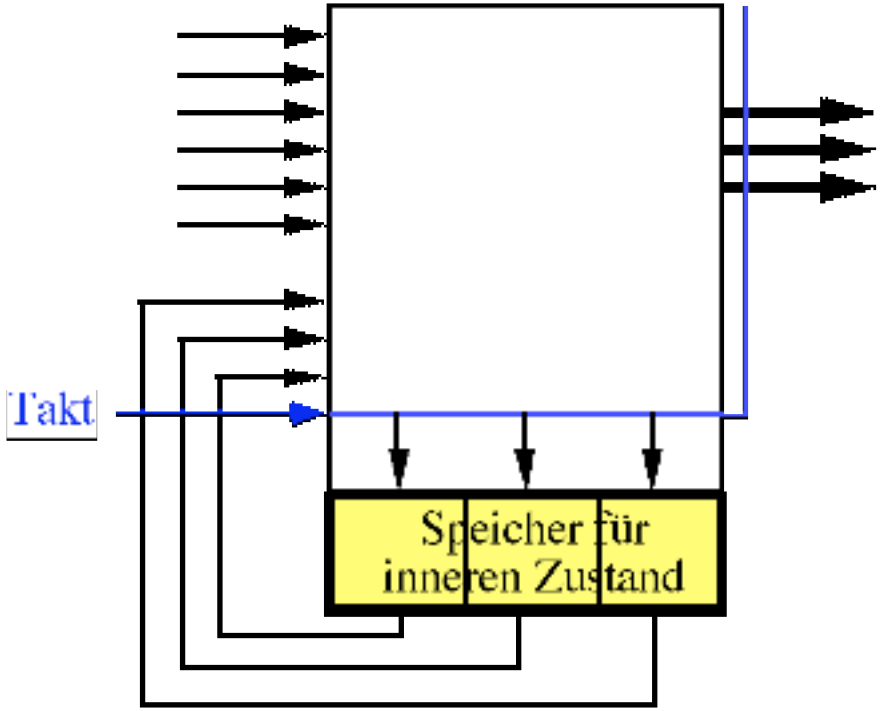
- Arithmetische Funktionen:

A-1 A-B-1 A+B -1

$A \wedge B - 1 \quad A \wedge \overline{B} - 1 \quad A + (A \vee B) \quad A = B$

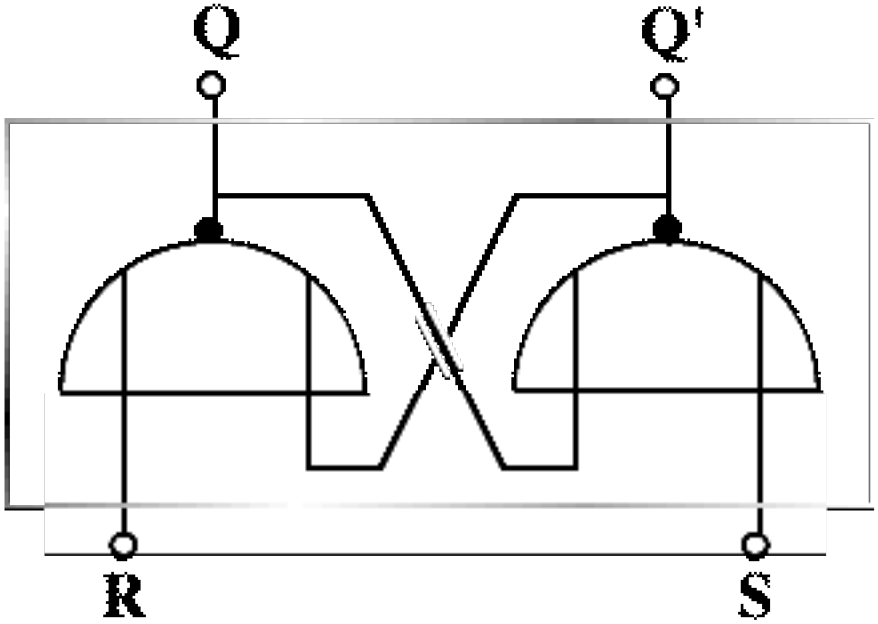
...

- Sequentielle Schaltungen
  - mit inneren Zuständen
  - beeinflussen Ausgangsfunktionen

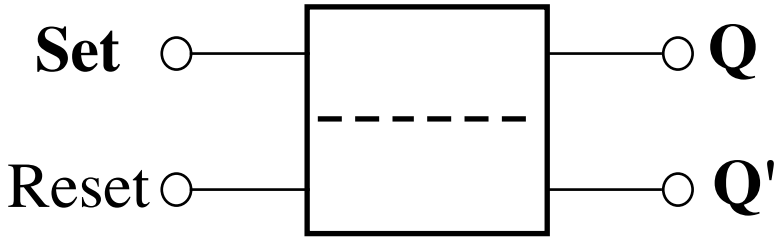


- Takt sorgt für die Sequenzierung
  - Eingänge und Zustand -> Ausgänge und neuen Zustand
  - Neuen Zustand übernehmen und für nächste Taktperiode speichern

- Einfache Speicherzelle: Flip-Flop
  - Speichern einer Binärstelle
  - die beiden Hälften halten sich gegenseitig



- sog. RS-Flip-Flop:
  - **S**etzen ("Set"),
  - **R**ücksetzen ("Reset")

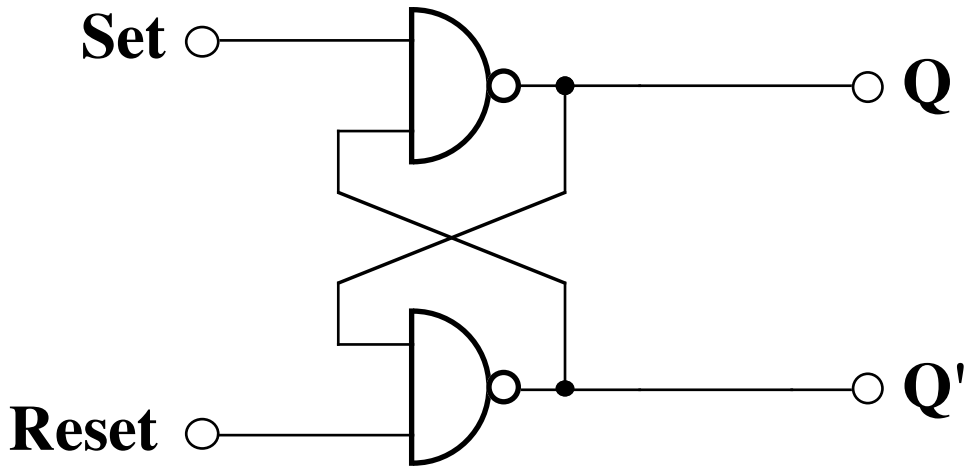


- Schaltfunktion für RS-Flip-Flop:
  - $Q = \overline{R \vee Q'} = \overline{R \vee \overline{(S \vee Q)}} = \overline{R} \wedge (S \vee Q)$
  - $Q' = \overline{(S \vee Q)} = \overline{S} \wedge (R \vee Q')$
- Wahrheitstafel:

R	S	Q_n	Q'_n
0	0	Q_{n-1}	Q'_{n-1}
0	1	1	0
1	0	0	1
1	1	0	0

- Undefinierter Folgezustand für **R=1; S=1**
- Zwei Speicherungs-Zustände mit **R=0; S=0**:
  - Q = 0; Q' = 1
  - Q = 1; Q' = 0
  - fast immer $Q \neq Q'$!
- Zwischenzustände während Umschaltung

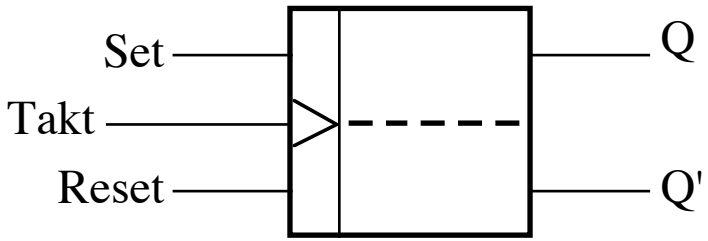
• RS-FF aus Nand-Gattern:



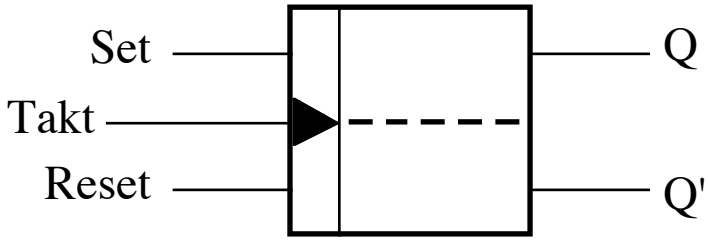
=> modifizierte Wahrheitstabelle:

S	R	Q	Q'	
1	1	Q_{alt}	Q'_{alt}	Speichern
0	1	1	0	Setzen
1	0	0	1	Rücksetzen
0	0	1	1	Unzulässig

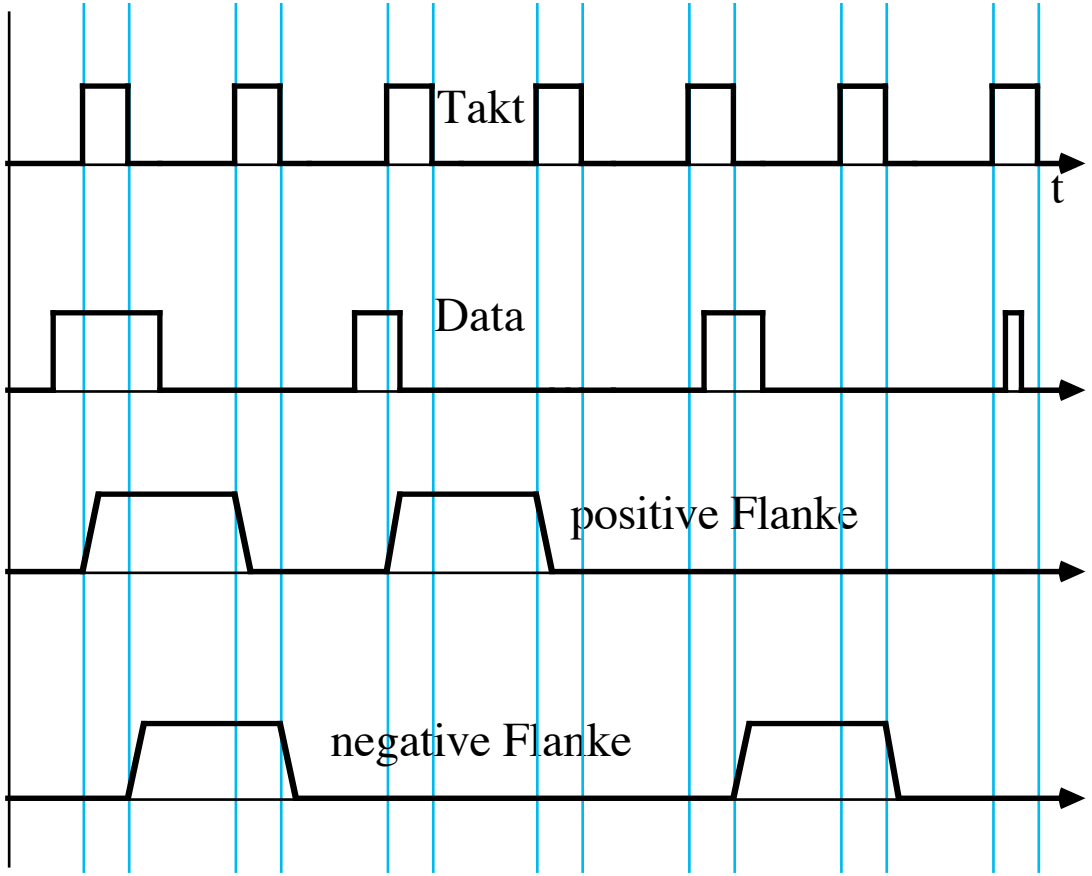
- Taktsteuerung
  - Eingangswerte stabilisieren lassen
  - dann Übernahmepuls
  - Eingangswerte werden nur zum Taktzeitpunkt berücksichtigt
- Flankengesteuertes Flipflop (abfallend):



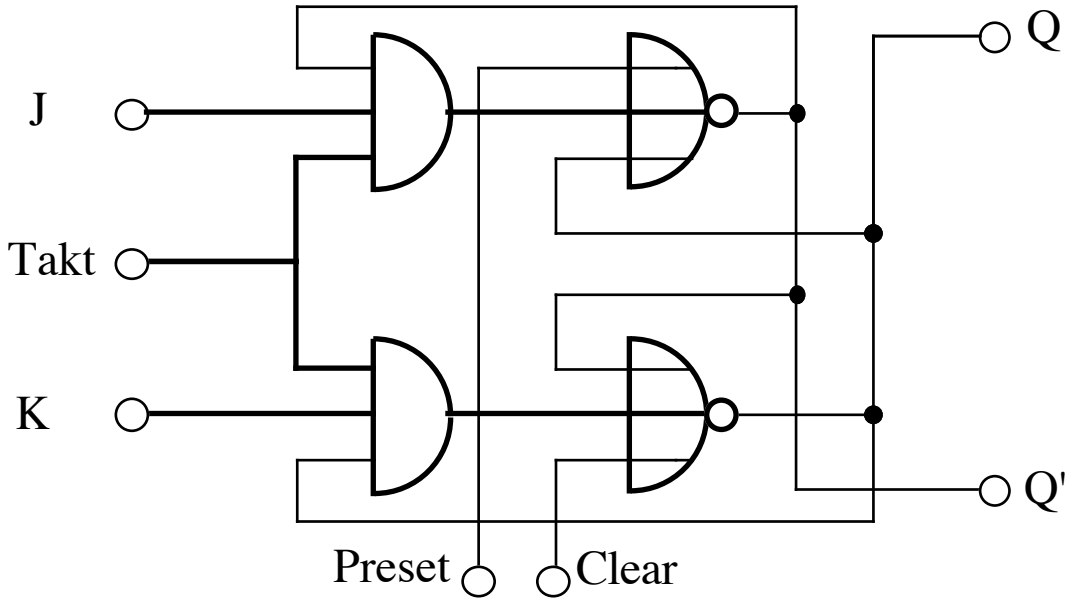
- Flankengesteuertes Flipflop (ansteigend):



- Signalverlauf
  - erst bei entsprechender Taktflanke Eingabe einlesen
  - evtl. unterschiedliche Ergebnisse



- JK-Flip-Flop
  - Eingänge vom Ausgang her verriegeln
  - Preset- & Clear-Eingänge möglich



- Ablauf

a) J=x, K=x, T=0, Q=1, Q'=0 {Anfangskonfiguration}

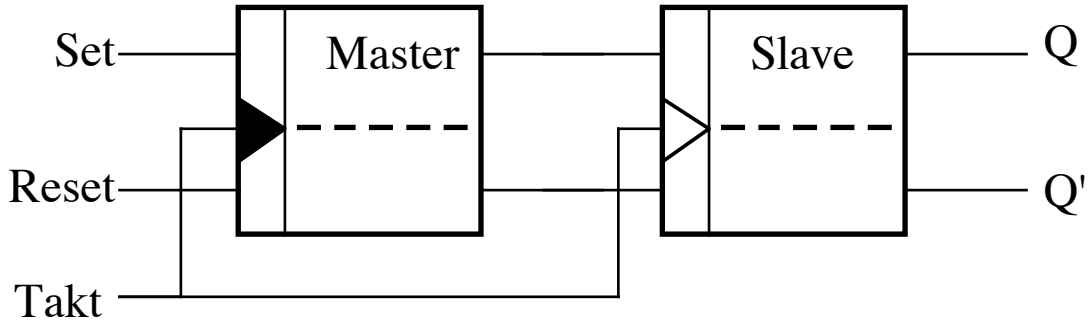
b) J=1, K=0, T=1 -> Q=1, Q'=0 {Eingang J unwirksam}.

c) J=0, K=0, T=0 -> Q=1, Q'=0 {Takt aus, J&K unwirksam }

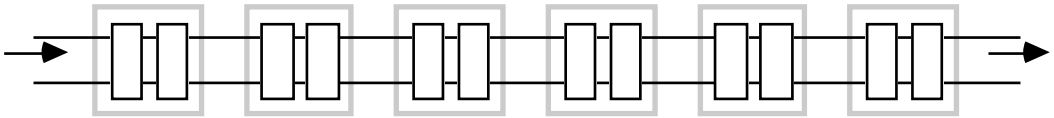
d) J=0, K=1, T=1 -> Q=0, Q'=1 {Umschalten nur mit Takt }

J=1, K=1, T=1 -> Q=1, Q'=0

- Master-Slave Flip-Flop
  - Zwei FF-Stufen arbeiten phasenverschoben
  - Master übernimmt Eingangswerte mit ansteigender Flanke
  - Slave gibt mit abfallenden Flanke Werte an Ausgang

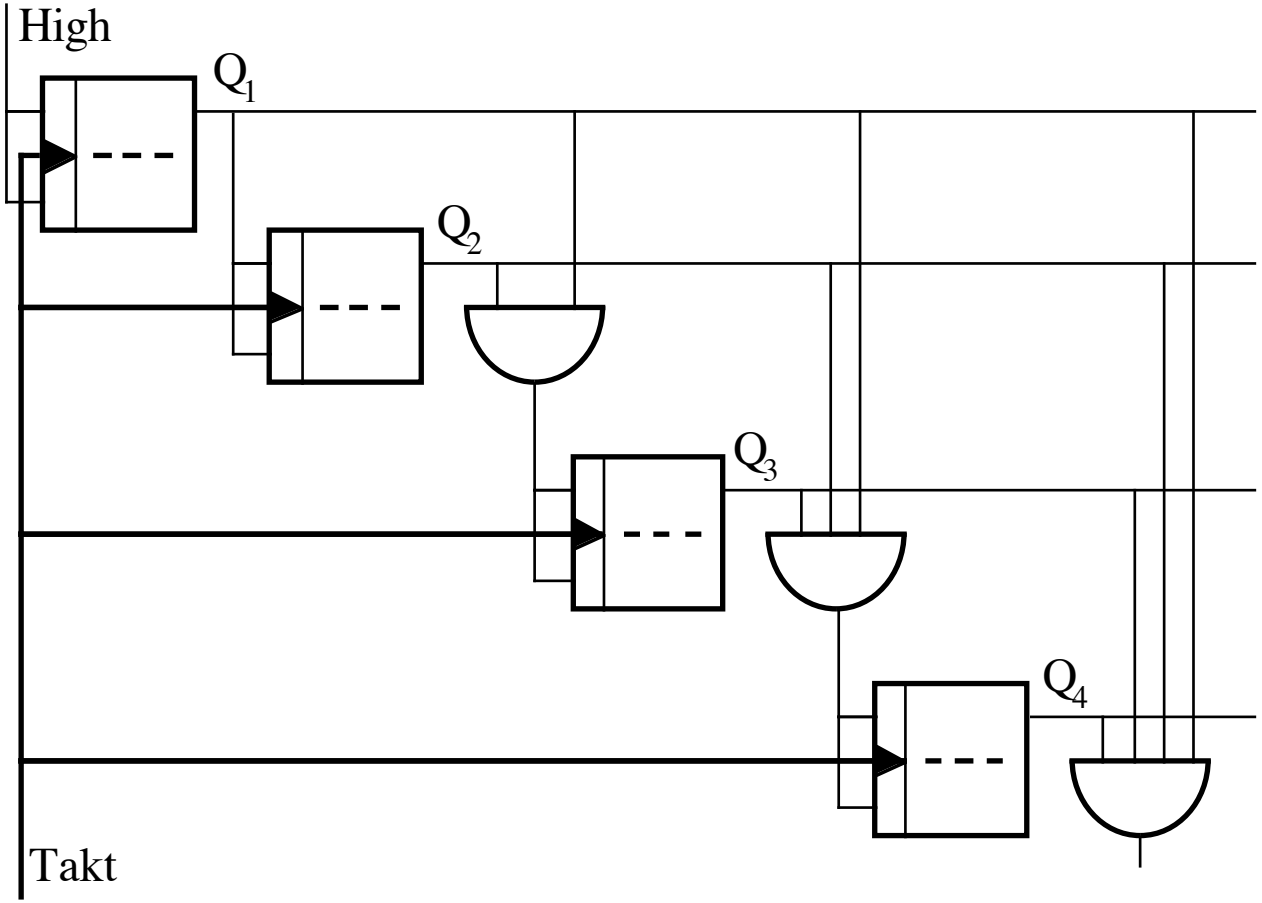


- Für Schaltungen mit heiklem Timing
  - Registertransfers
  - Pufferregistern mit MS-FFs
- MS-Schieberegister

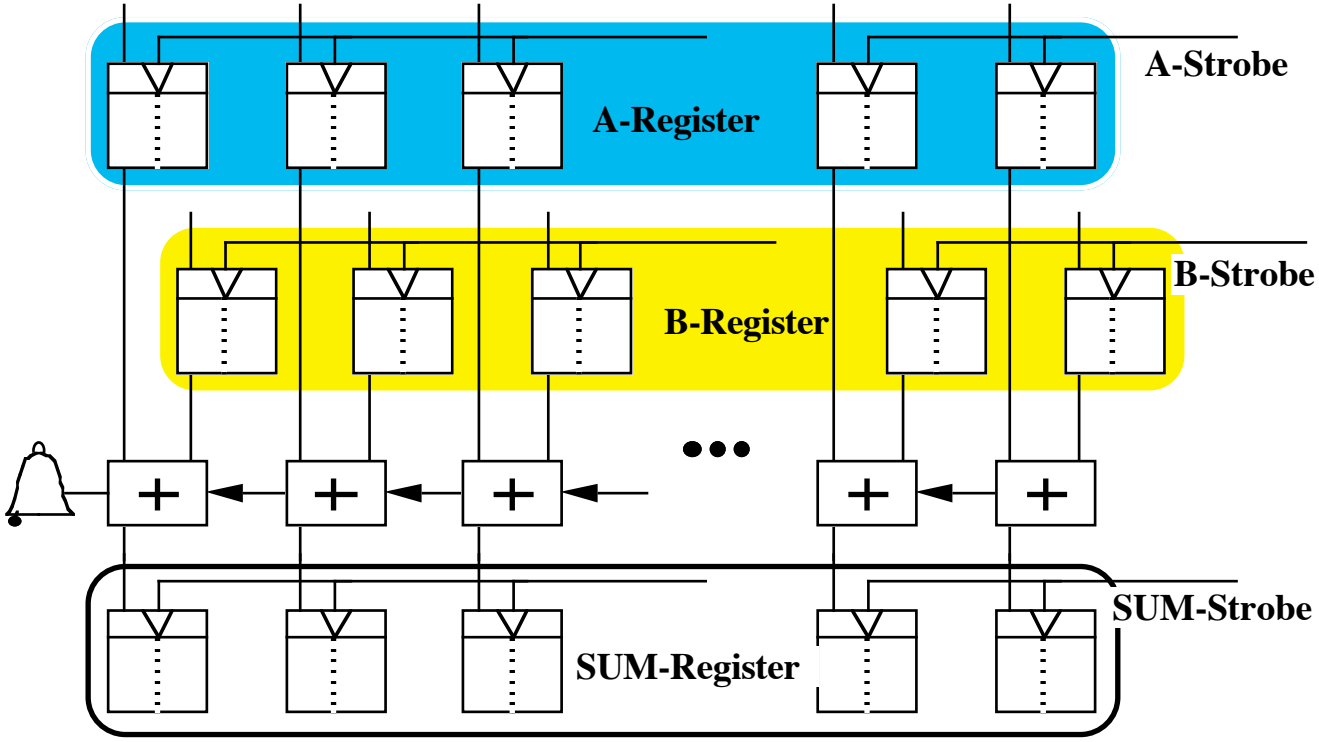


  - als digitale Verzögerungsleitung
  - für digitale Filter
  - als Rechenoperation

- Zähler
  - Jede Stufe erhält Takt
  - schaltet in Abhängigkeit vom Zustand aller vorherigen Stufen



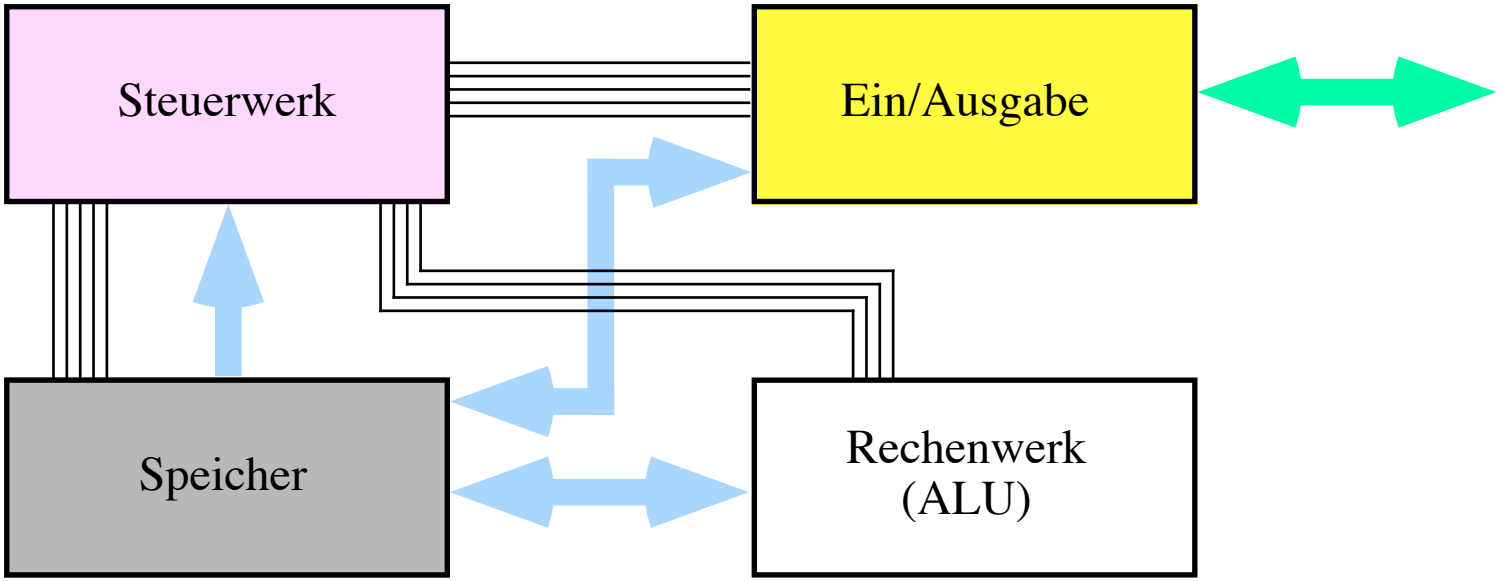
- Register speichert Zahlen
- Register in Verbindung mit Addierschaltung bzw. ALU



- Ablauf
  - A-Strobe zum Füllen des Registers A
  - B-Strobe zum Füllen des Registers B
  - Addition bzw. Übertrag abwarten,
  - Summe abholen mit SUM-Strobe
  - Überlauf?

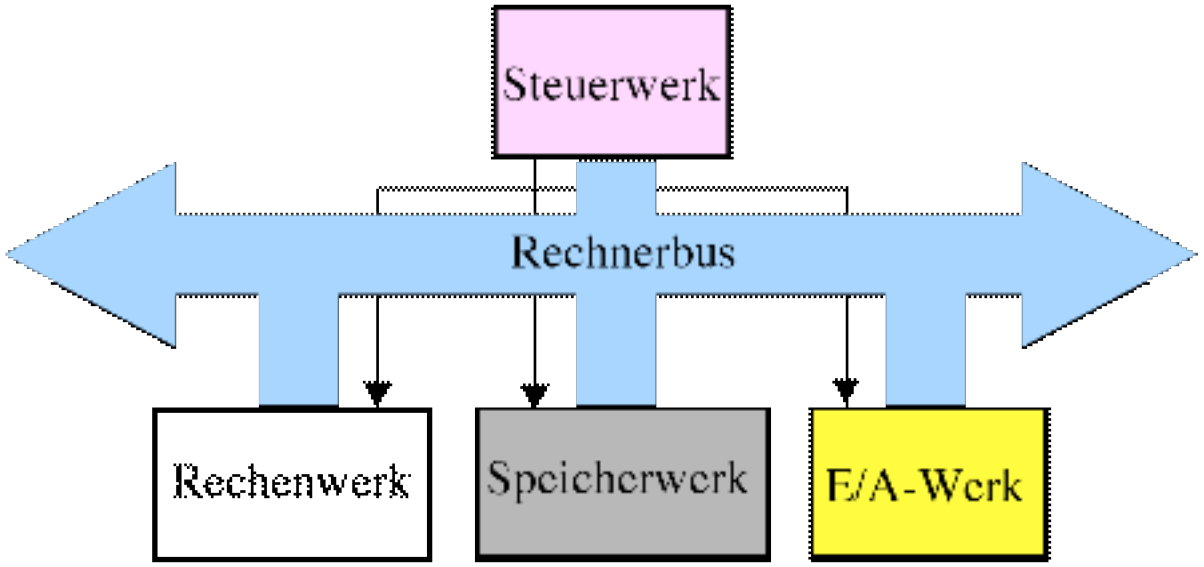
0.6 Funktionseinheiten: Speicher, Prozessor, Bus, …

- klassischer Rechner nach J. v. Neumann
  - Rechnerwerk für arithmetische und logische Verknüpfungen
  - Universalspeicher für Daten und Programme

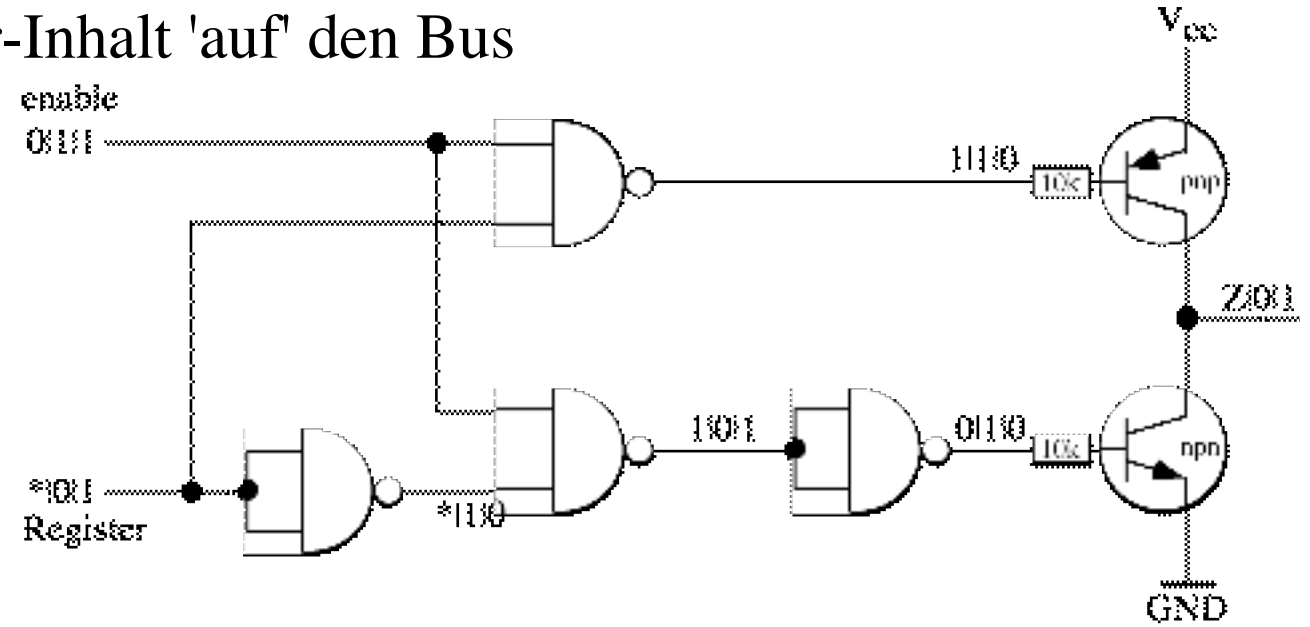


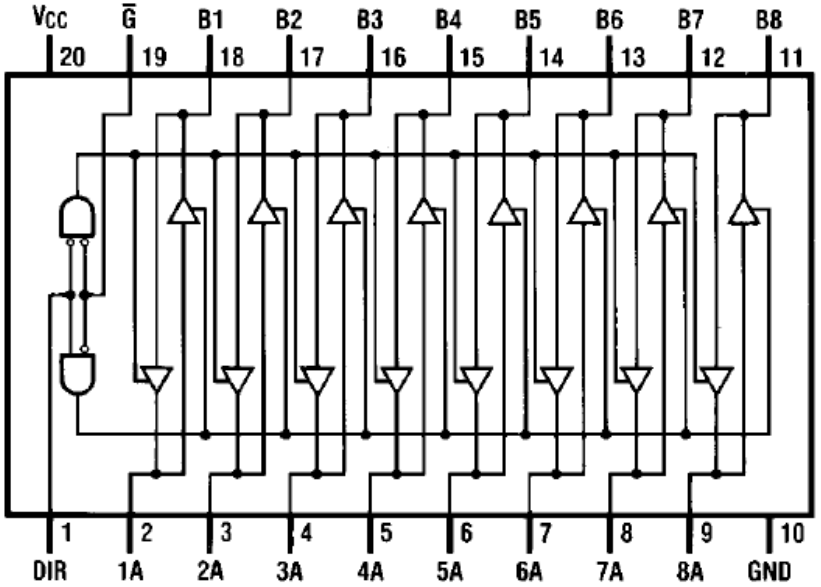
- Steuerwerk
  - sequentieller Ablauf des Programmes
  - Interpretation der Instruktionen => Steuerbefehle

- Busorientierter Rechner
  - universell verbunden
  - Schwerpunkt Transport und Verteilung
  - weniger Verbindungen

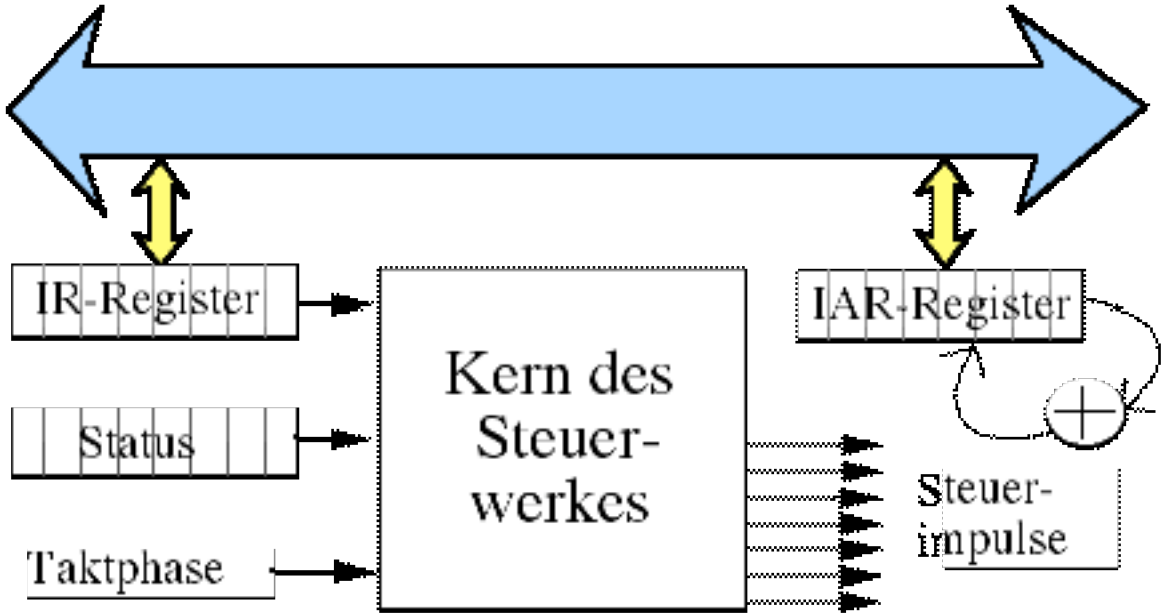


- Bustreiber
  - Register liegt nicht 'direkt' am Bus
  - Steuersignal legt Register-Inhalt 'auf' den Bus
- Tri-state-Schaltung
  - 3 Zustände
  - hochohmig
  - 0, 1
- 74xx251
  - 8 Bit
  - Transceiver
  - bidirektional
  - Richtungseingang
  - Strobe



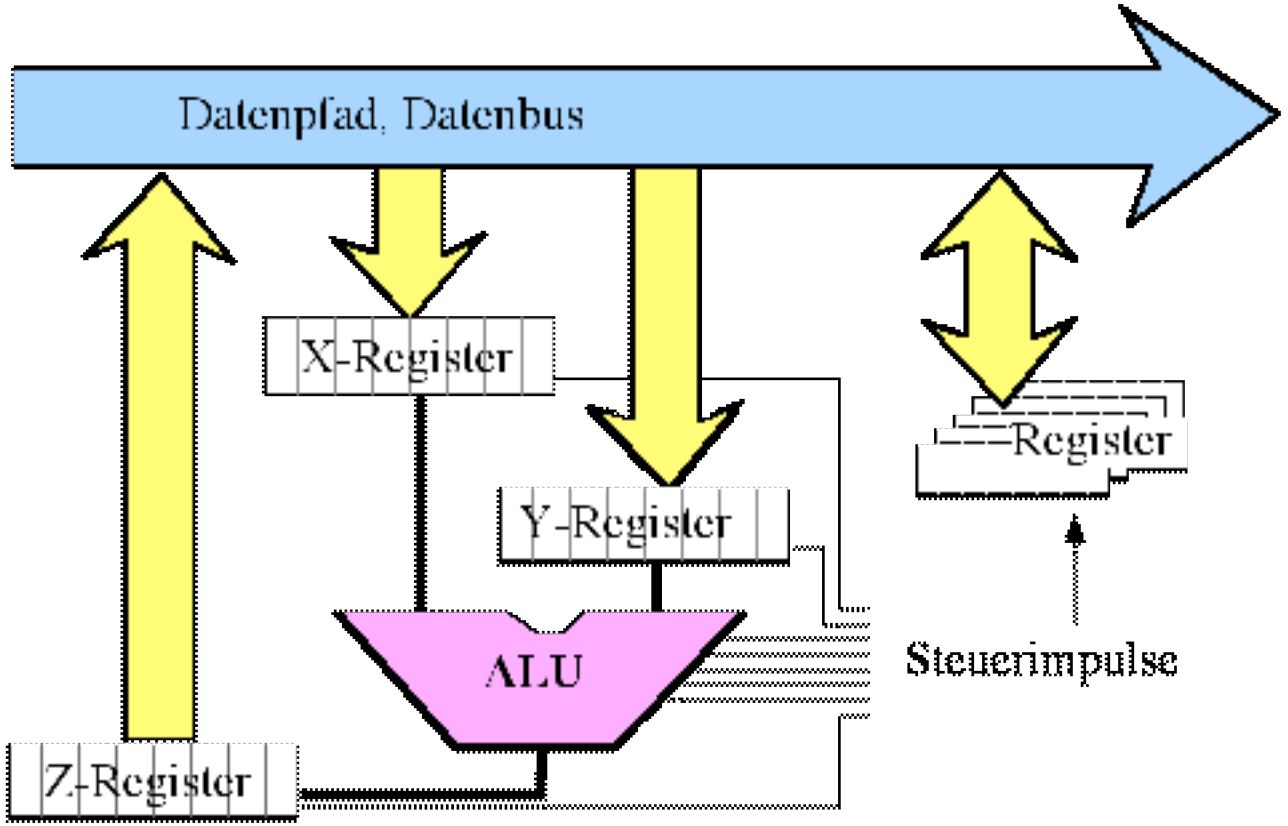


- Steuerwerk erzeugt Signale, die
  - Datentransfer auslösen
  - ALU-Operation auswählen
  - Speicheroperation auslöst



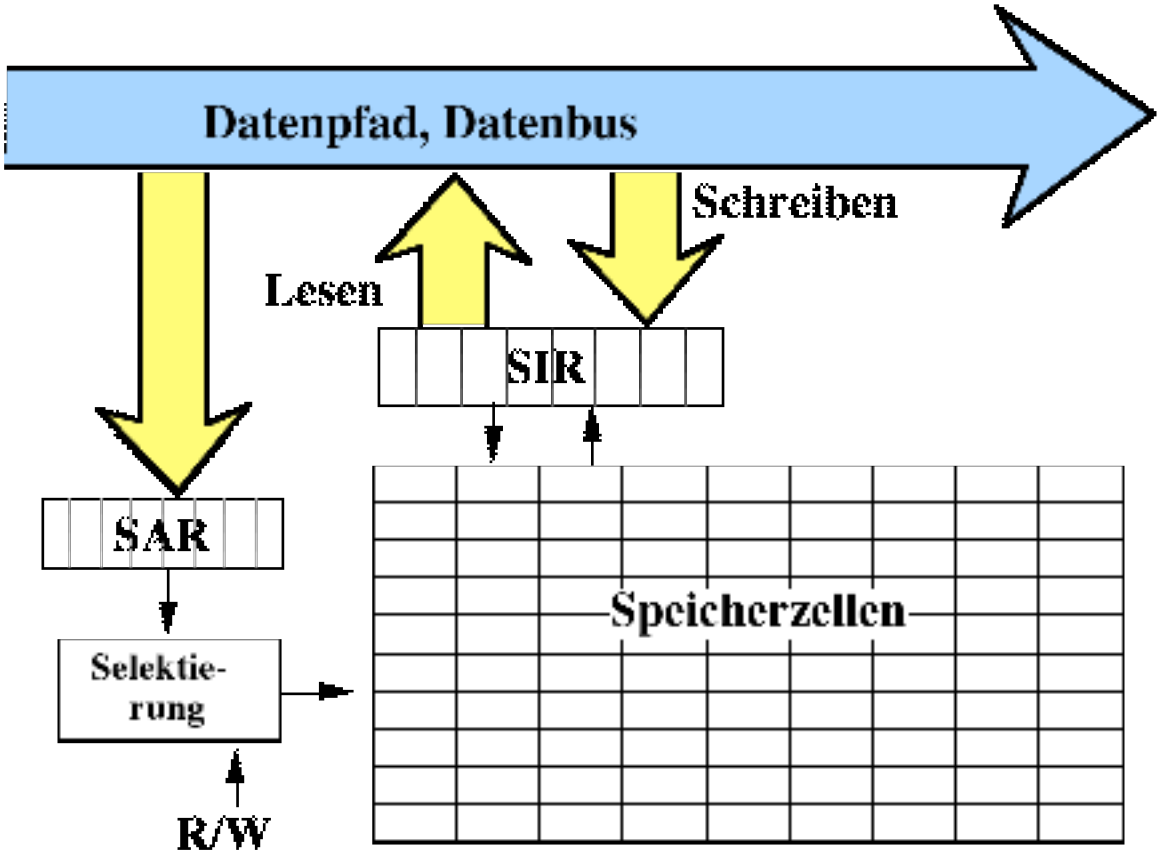
- Befehl wird in Sequenz von Kernzuständen umgesetzt
  - Kernzustand bestimmt Steuersignale
- Register
  - Instruktionsregister (IR)
  - Instruktionsadressregister (IAR) bzw. "Program Counter" (PC)
  - eventuell eigener Addierer für IAR

- Rechenwerk
  - kombinatorisches Schaltnetz
  - Addieren, Multiplizieren, UND, ODER, Vergleich, Shift, …
  - Ausgang der ALU liegt dauernd am Z-Register an
  - X-Register und Y-Register liegen dauernd am ALU-Eingang an



  - Zwischenresultate in Zusatzregistern
  - Steuerleitungen bewirken Datenübernahme = Transport

- Speicher

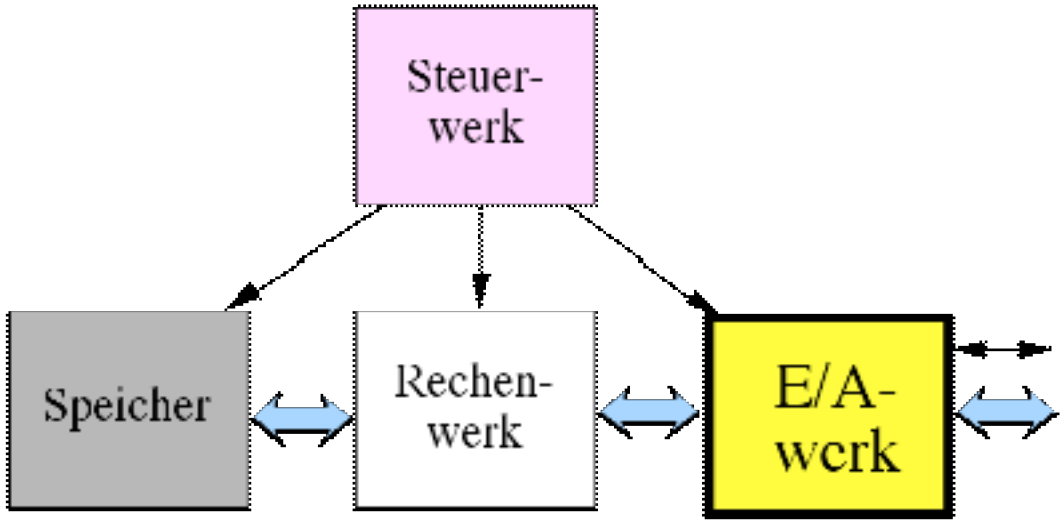


SAR = Speicheradressregister, SIR = Speicherinhaltsregister

- Random Access Memory (RAM)
  - Halbleiterspeicher halten Inhalt nur wenn sie "unter Strom" stehen
  - dynamische Speicher (DRAM) müssen periodisch aufgefrischt werden
- ROM: Festwertspeicher

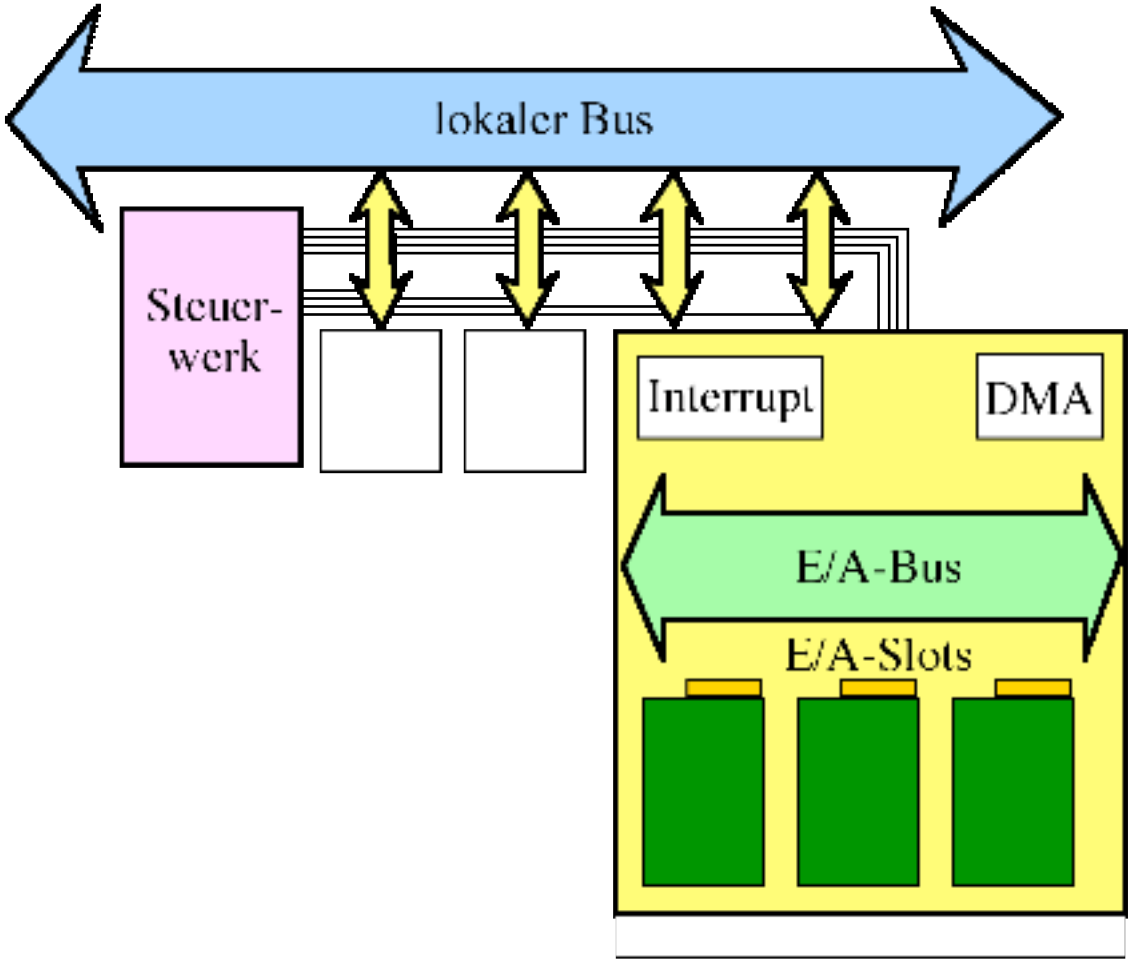
- Speicherhierarchie aus Kostengründen
- Register
  - 4-256 Wörter,< 1 Nanosekunde
  - kein separater Zyklus für die Adresse.
- Cache, Prozessorpufferspeicher
  - 8 KWörter bis 1024 KBytes, < 10 Nanosekunden
  - häufig gebrauchte Daten und Codeteile
  - für Programmierer unsichtbar
- Hauptspeicher
  - 64 - 2000 Megabytes, ~ 7 - 10 Nanosekunden
  - evtl. inklusive Adressübersetzung
  - separater Zyklus für die Speicheradresse
- Hintergrundspeicher
  - Festplatte, Netz,
  - 10 Gbytes -100 Gbytes, ~ 10 Millisekunden
  - Zugriff über Betriebssystem
  - blockweise übertragen

- Ein-/Ausgabewerk
  - kontrolliert durch Steuerwerk
  - Transport via Rechenwerk in den Speicher
  - Bedienungsleitungen zum Peripheriegerät
  - Keine Parallelarbeit von Rechenwerk & E/A
  - Missbrauch der Rechenregister
  - Wartezyklen der CPU auf Peripheriegeräte



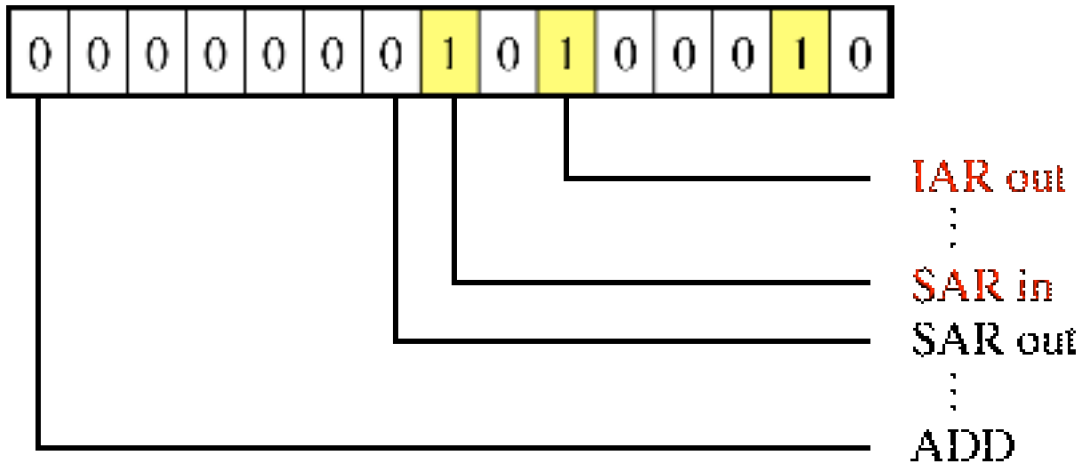
- Pfad in den Speicher als Flaschenhals
  - Resultate von Berechnungen
  - E/A-Übertragungen
  - Instruktionen

- Moderne Rechner
  - besonderer, standardisierter E/A-Bus
  - getrennt vom Prozessorbus
- autonomere E/A-Schnittstelle
  - externer Bus,
  - Abläufe parallel zum Rechenwerk
  - Interrupt-Funktion
  - DMA-Kanäle (direct memory access)
  - Display-Kontroller
  - Adapterplatinen
  - VLSI-Chips
- Grafik evtl. Extrabus
  - AGP

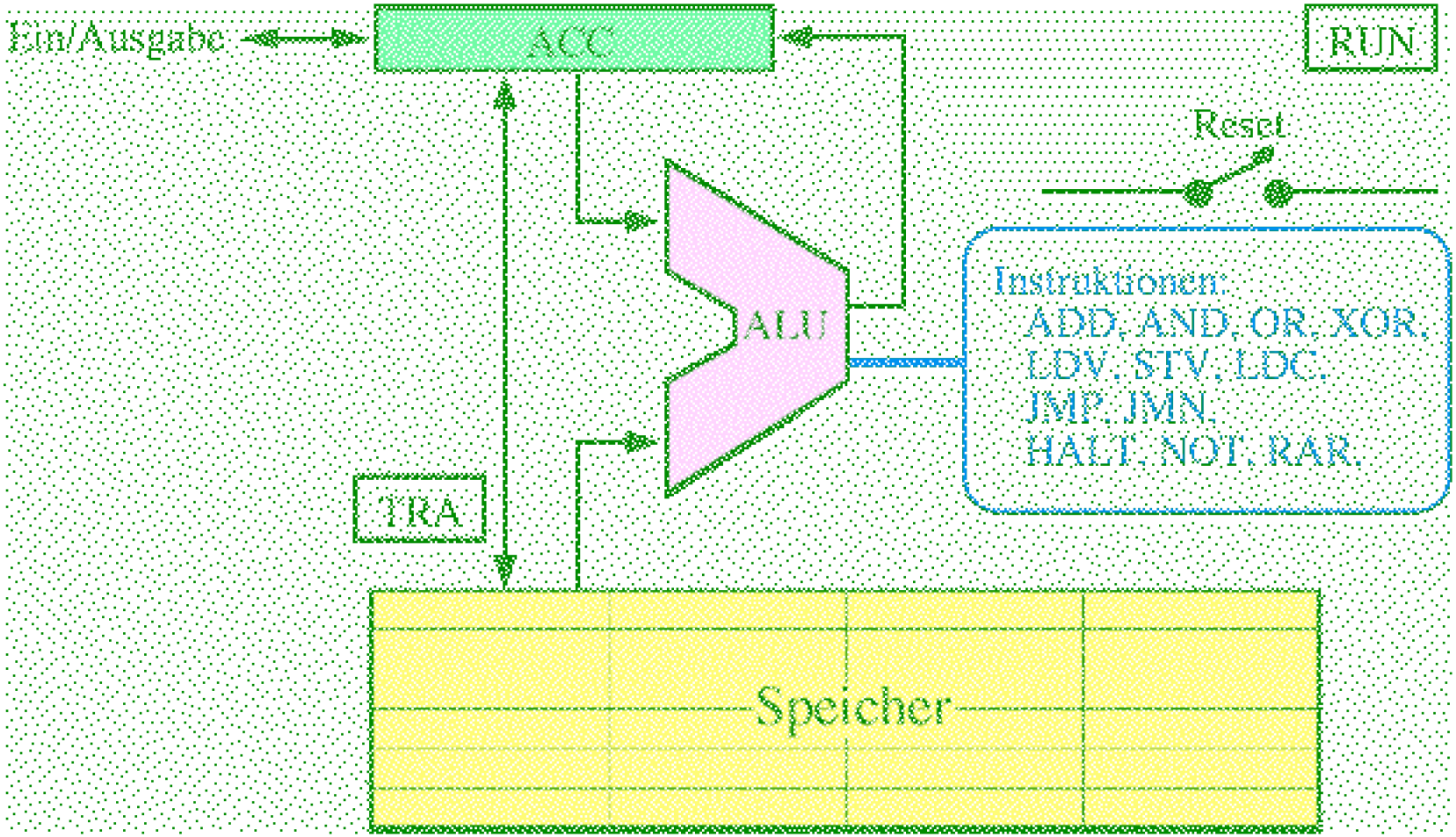


0.6 Maschinenbefehle

- Befehle liegen im Speicher
  - werden geholt wie Daten
  - kommen in Befehlsregister
  - Befehlszähler
- Befehl ⊗ Status => Sequenz von Steuerworten
  - Bits des Steuerwortes an Steuerleitungen anlegen
  - Speicher, E/A, ALU, … reagieren auf Steuerleitungen
  - in jedem Takt ein Steuerwort
  - Befehle können mehrere Steuerworte enthalten
  - Mikrobefehle



0.6.1 Minimalmaschine



- ein Register
- nur wenige, einfache Instruktionen
- TRA: Warten auf E/A
- 32-Bit Architektur

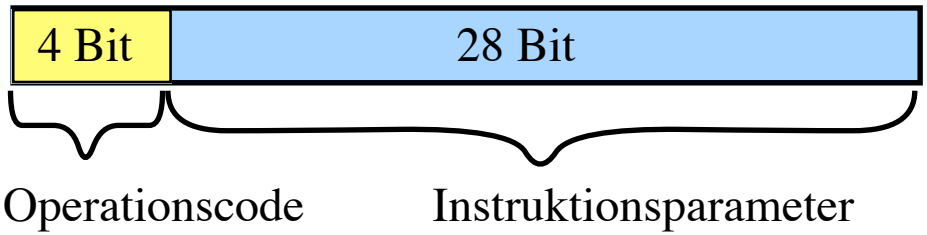
- Befehle zur Datenmanipulation

NOT		Komplementiere den Akkumulator, B-1 Komplement
RAR		Rotiere den Akkumulator um 1 Bit nach rechts
ADD	a	Addiere den Inhalt der Speicherzelle a zum Akkumulator
		ACC ← ACC + Speicher[a]
AND	a	ACC ← ACC ∧ Speicher[a]
OR	a	ACC ← ACC ∨ Speicher[a]
XOR	a	ACC ← ACC ⊕ Speicher[a]
EQL	a	ACC<>Speicher[a]: ACC← 0
		ACC=Speicher[a]: ACC← 11..1
LDV	a	ACC ← Speicher[a]
STV	a	Speicher[a] ← ACC
LDC	c	ACC ← c

- Kontrollfluß

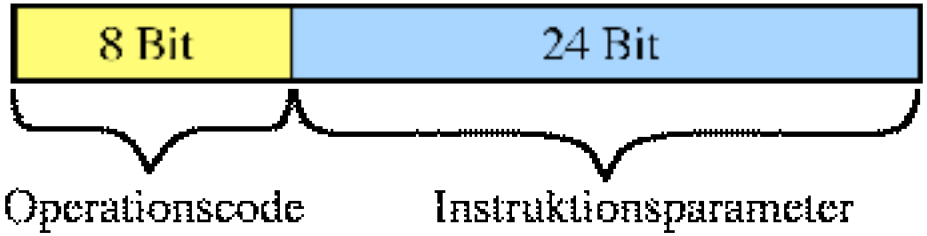
JMP	p	Nächste Instruktion in Speicher[p]
JMN	p	Sprung nach p falls ACC negativ
HALT		RUN ← 0 Instruktionsausführung hält an
		Später Restart nur aus Speicher[0] möglich

- Basisformat



  - 0,1,2,3 ADD, AND, OR, XOR
  - 4, 5, 6 LDV, STV, LDC
  - 7, 8, 9 JMP, JMN, EQL
  - A .. E reserviert

- Erweitertes Format
  - mehr als 16 Instruktionen möglich,
  - 24 Bit Parameter & nicht immer benützt

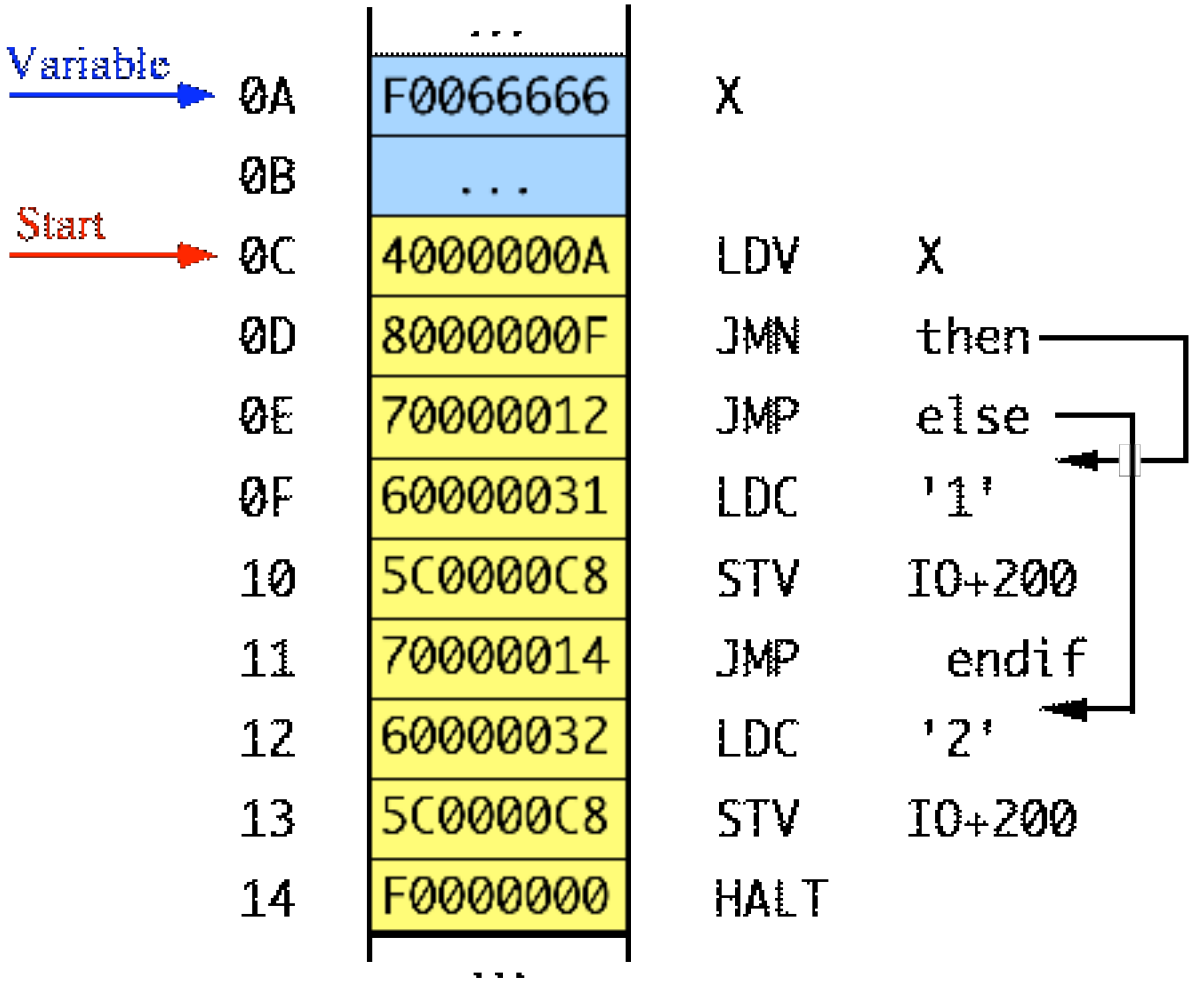


  - F0, F1, F2 HALT, NOT, RAR
  - F3, F4,… OUT, INP, …

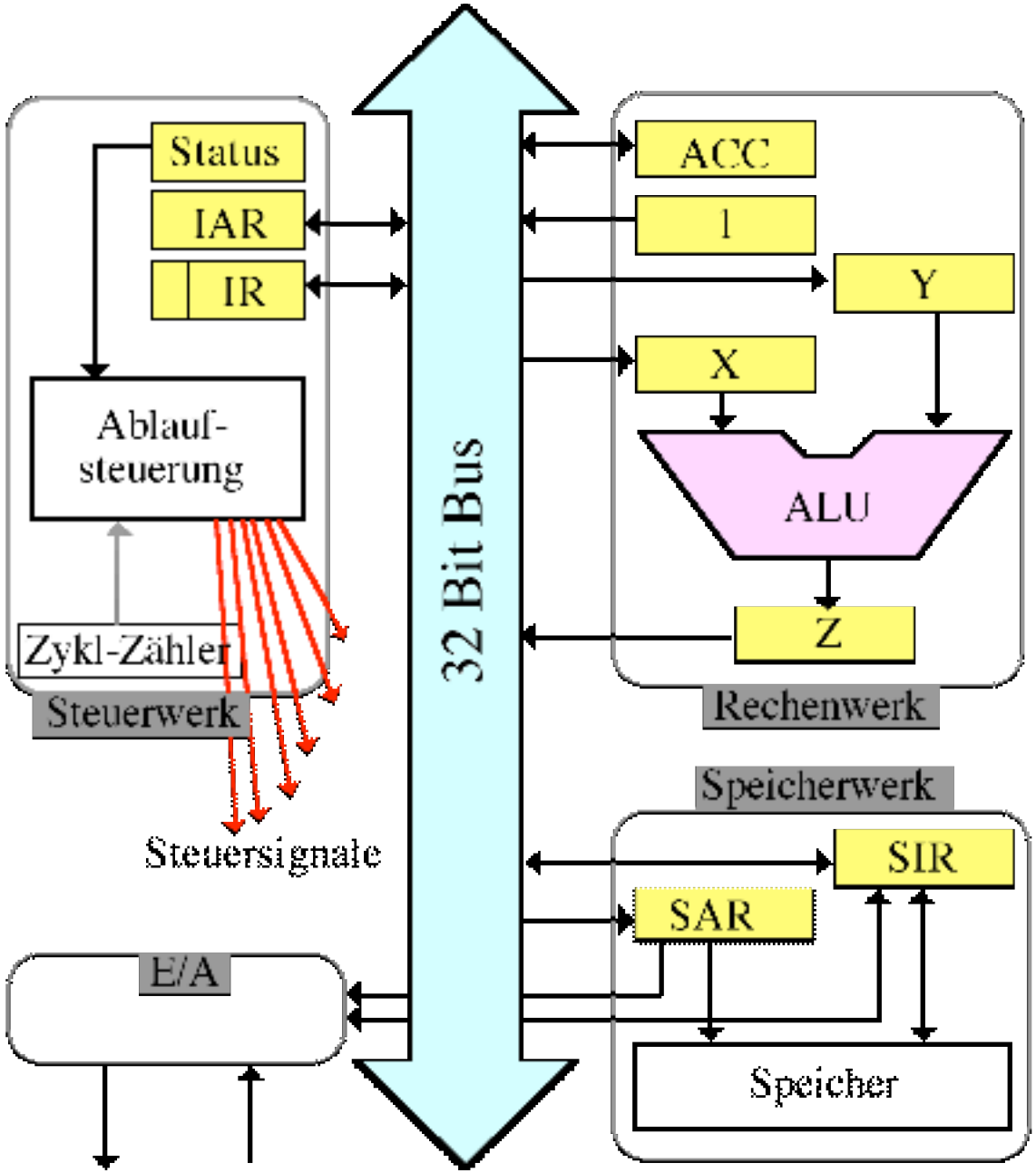
- Assemblerprogrammieren im Schnelldurchgang

```
if (X < 0)     printf("1")
               else printf("2");
```

- Hexadezimaler MiMa-Speicherauszug

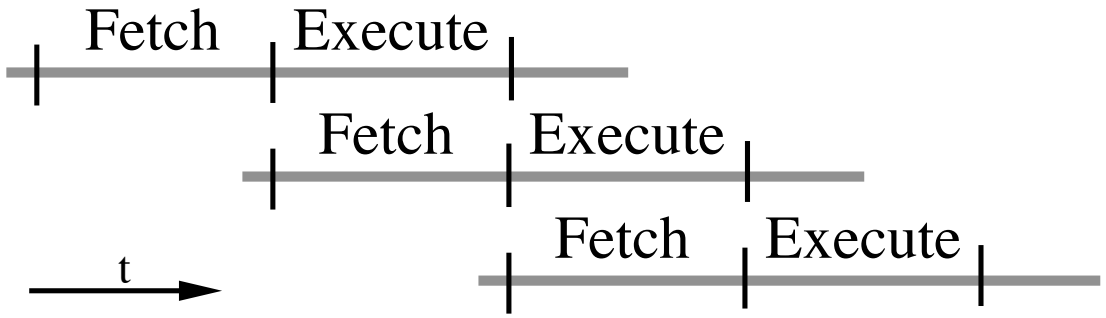


- Speicher
  - statisch
  - 32 Bit/Wort
- 1-Register
  - z.B. Inkrement
- Steuerwerkregister
  - Instruktionen
  - Instruktionsadresse (PC)
  - Status
- ALU
  - Eingang X,Y
  - Ergebnis Z
  - ADD, AND, XOR, OR
- Registertransfer
  - Quell-Register an Bus
  - Zielregister übernimmt
  - Steuerleitungen



0.6.2 Ablauf der Instruktionen

- Zweiteilung der Befehle
  - Fetch-Zyklus holt Befehl
  - Execute Zyklus führt Befehl aus
  - eventuell überlappend



- Unterzyklen im Steuerwerk
  - Mikrozyklus, "Minor Cycle", Taktphase
  - andere Steuerimpulse in jedem Unterzyklus
  - Registertransferebene
- Mima-Instruktionen
  - 12 Unterzyklen
  - 5 Unterzyklen Fetch
  - 7 Unterzyklen Execute
  - evtl. flexibel

- Ablauf der Lade-Instruktion

  LDV a ; laden von Inhalt der Speicherzelle a in ACC

  ; ACC <= Speicher[a]

```
    ;Fetch-Zyklus
1.          IAR -> (SAR, X),Leseimpuls an Speicher
2.          1   -> Y,       ALU auf Addition schalten,
                            Warten auf Speicher
3.                          Warten auf ALU, Warten auf Speicher
4.          Z   ->  IAR,    Warten auf Speicher
5.          SIR -> IR
    ;Fetch-Ende, jetzt Execute-Zyklus
6.          IR  -> SAR,     Leseimpuls an Speicher
7.,8.,9.                    Warten auf Speicher
10.         SIR -> ACC
11.,12.                     leere Unterzyklen
    ;Execute-Ende, nächste Instruktion
```

- Ablauf der Add-Instruktion

  ADD a ; addieren von Inhalt der Speicherzelle a zum ACC

```
    ;Fetch-Zyklus
1.        IAR -> (SAR, X),Leseimpuls an Speicher
2.        1   -> Y,       ALU auf Addition schalten,
                          Warten auf Speicher
3.                        Warten auf ALU, Warten auf Speicher
4.        Z   ->  IAR,    Warten auf Speicher
5.        SIR -> IR
    ;Fetch-Ende, jetzt Execute-Zyklus
6.        IR  -> SAR,  Leseimpuls an Speicher
7.        ACC  -> X,   Warten auf Speicher
8., 9.                 Warten auf Speicher
10.       SIR -> Y,    ALU auf Addition schalten
11.                    warten auf ALU
12.       Z   -> ACC
    ;Execute-Ende, nächste Instruktion
```

- JMN p - Jump if Negative
  - Bedingter Sprung zur Instruktion im Speicherwort[p]
  - negativ bedeutet, das oberste Bit im Akkumulator ist 1
  - X, Y-Register und ALU werden ignoriert
  - kein Datenzugriff auf Speicher

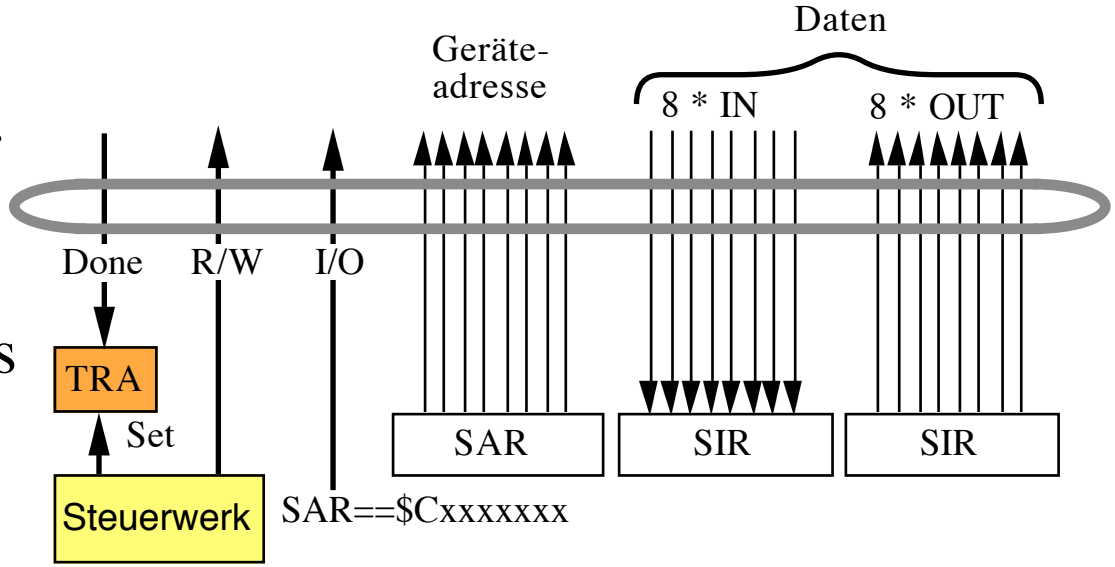
```
    ;Fetch-Zyklus
1.        IAR -> (SAR, X),Leseimpuls an Speicher
2.        1   -> Y,       ALU auf Addition schalten,
                          Warten auf Speicher
3.                        Warten auf ALU, Warten auf Speicher
4.        Z   ->  IAR,    Warten auf Speicher
5.        SIR -> IR
    ;Fetch-Ende, jetzt Execute-Zyklus
6a.    falls vorderstes Bit in ACC = 1:
              IR  -> IAR
6b.    falls vorderstes Bit in ACC = 0:
              leerer Unterzyklus
7. … 12.  leere Unterzyklen
    ;Execute-Ende, nächste Instruktion
```

- Start der Mima
  - Drücken der Reset-Taste

```
0 -> IAR
0 -> TRA
1 -> RUN
```

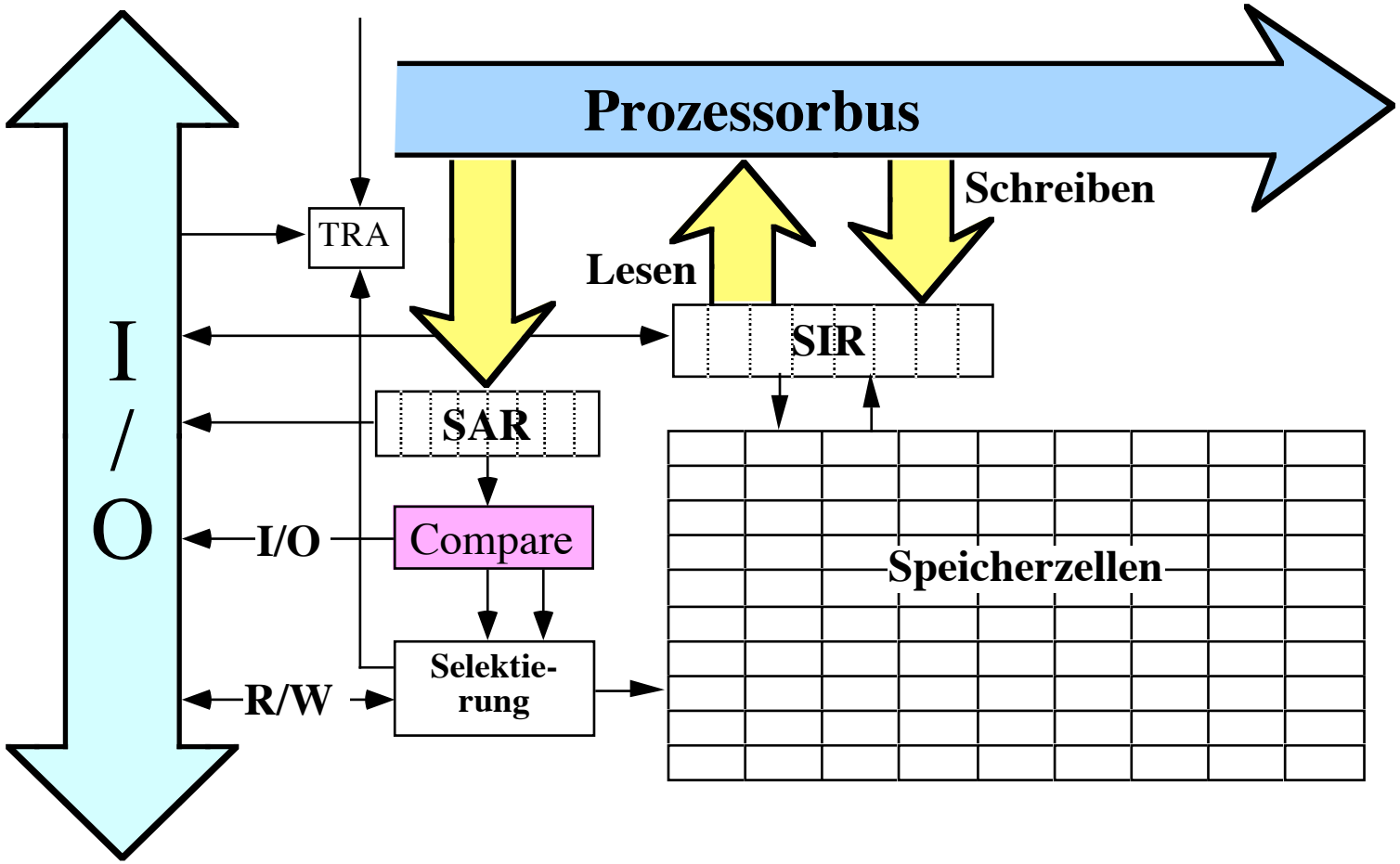
- Maschine beginnt ab Adresse 0 Instruktionen auszuführen
  - evtl. andere, festgelegte Adresse
- Urlader
  - einige Speicherzellen ab Adresse 0 sind Festwertspeicher
  - enthalten einen Urlader ("Boot-Strap Loader")
  - liest Programm von einem bestimmten Gerät in den Speicher
  - beginnt mit dessen Ausführung
- Schaltpult mit Schaltern und Lämpchen zum
  - Modifizieren und Inspizieren von Registern und Speicherinhalten
  - bei älteren Rechnern

- Integration der E/A in den normalen Adressbereich
  - memory mapped I/O
  - In - LDV, Out - STV
  - Geräte langsamer als Speicher
  - variable Geschwindigkeit
  - => Handshake
- Speicherwerk wertet Adresse aus
  - Adr == $Cxxxxx => E/A
  - Adr <> $Cxxxxx => Speicher



- TRA-Bit
  - vom Steuerwerk gesetzt
  - von Speichercontroller oder Peripherie zurückgesetzt
  - Steuerwerk wartet auf TRA=0
- Geräte beobachten die Geräteadresse im EA-Werk
- Als Bus ausgeführte E/A-Schnittstelle

- Speicher mit Memory-mapped I/O
  - TRA Bit wird vom Prozessor gesetzt
  - von der Selektionsschaltung zurückgesetzt
  - oder vom Gerätebus zurückgesetzt



SAR = Speicheradressregister, SIR = Speicherinhaltsregister

- Instruktionen modifizieren

LDV a ; laden von Inhalt der Speicherzelle a in ACC
; ACC <= Speicher[a] bzw. Input/Output

```
    ;Fetch-Zyklus
1.          IAR -> (SAR, X),Leseimpuls an Speicher
2.          1   -> Y,       ALU auf Addition schalten, Speicher aktiv
3.                          Warten auf ALU, Speicher aktiv
4.          Z   ->  IAR,    Speicher aktiv
5.,…                    Alle Steuerleitungen 0, Wiederholen bis TRA==0
n.          SIR -> IR
    ;Fetch-Ende, jetzt Execute-Zyklus
n+2.        IR  -> SAR,     Leseimpuls an Speicher
n+3.                    Alle Steuerleitungen 0, Wiederholen bis TRA==0
m.          SIR -> ACC
    ;Execute-Ende, nächste Instruktion
```

0.6.3 Erzeugung der Steuersignale

- Eingangssignale zum Steuerwerk:
  - Operationscode aus IR-Register (4/8 Bit)
  - Vorzeichenbit im Akkumulator
  - Mikrozyklus aus externem Zähler
  - Status-Register = (RUN, TRA)

=> 8+1+4+2 = 15 Eingangsleitungen

- Ausgangssignale vom Steuerwerk
  - Rechteckige Steuer-Impulse
  - evtl. statische Steuerung (z.B. für ALU)
  - Takteingang für Flip-Flops
  - Out-Enable für Tristate-Ausgang am Bus

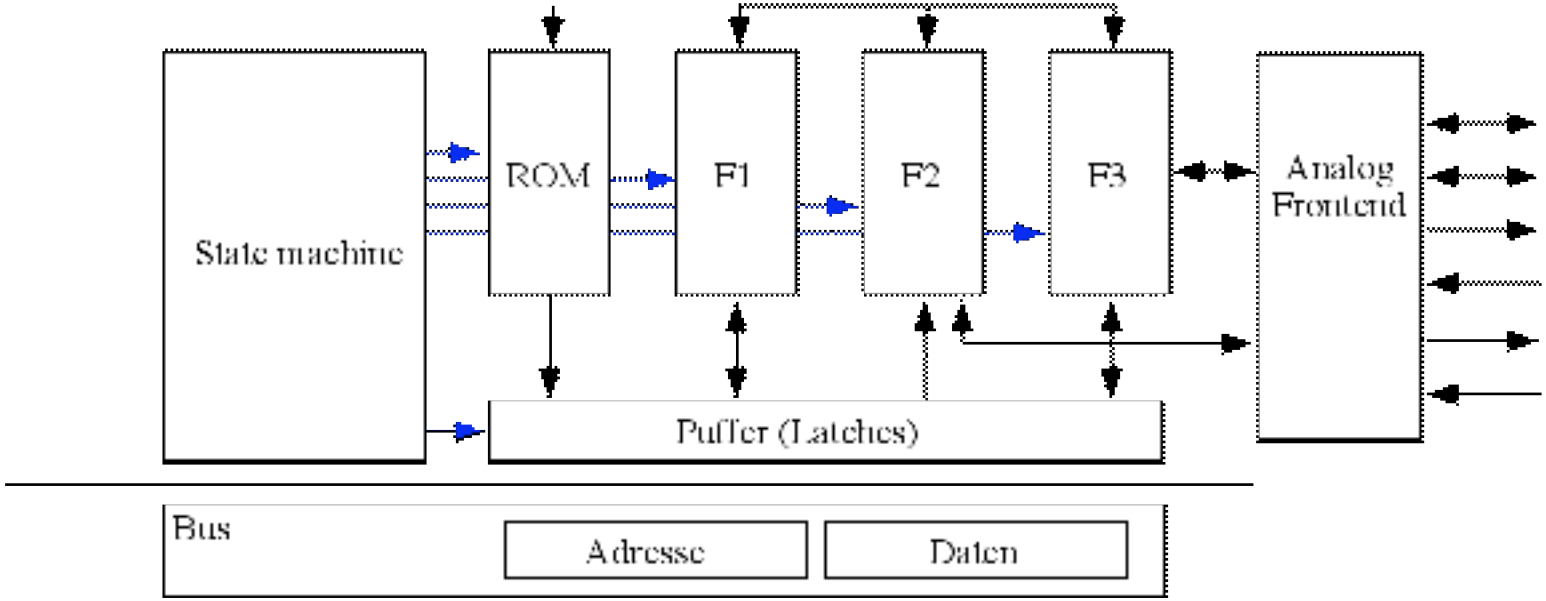
- RUN: 1 Leitung zum Clear-Eingang
- TRA: 1 Leitung zum Preset-Eingang
- ALU: 3 statische Pegel für die unterschiedlichen ALU-Funktionen
- Speicher: 2 Leitungen für Read & Write
- je 1 Input-Enable für Register
  - X, Y, ACC
  - IR, IAR, SIR, SAR
- zusätzlich Output-Enable für Register
  - ACC, 1, Z
  - IAR, SIR, IR (nur Bit [0..27])

=> 1+1+3+2+7+6 = 20 Steuersignale

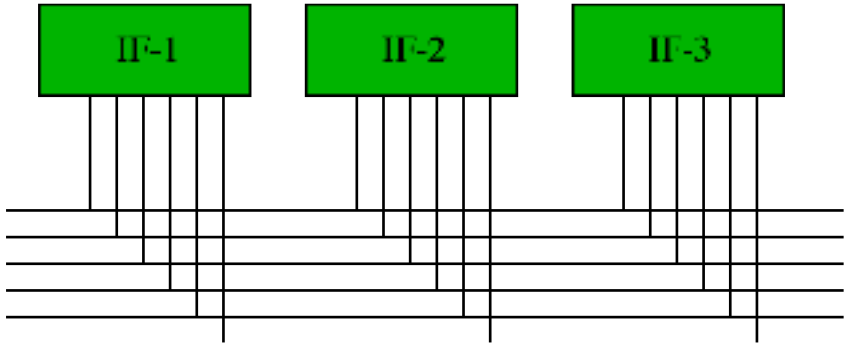
- Steuerwerk kann realisiert werden:
  - als 20 Schaltfunktionen mit 15 Variablen
  - als ROM mit 2^{15} Wörtern à 20 Bit

0.7 Adressierung, Timing, Hardware/Software-Interface

- Funktionseinheiten (Chips) an Bus anschliessen
  - Prozessorbus ('memory-mapped') oder I/O Bus
  - Adressen und Daten
  - Steuerleitungen: Read/Write, Strobes, …
- Businterface
  - Busphasen, Adressdekodierung (state machine)
  - Puffer
  - Register
  - Identifizierung



- Bus
  - Adressbus und Datenbus oft 'gemultiplext'
  - Steuerleitungen: Adress-Strobe, DataStrobe
  - Takt
  - Card-Select



- Puffer
  - entkoppeln Bustakt und Adapter-Takt
  - evtl. bidirektional
  - Bustreiber: active-high, active-low, offen (tri-state)
- State-Machine
  - Chipselect (C/S) erzeugen
  - Adress-Dekodierung für Datenpuffer
  - Adress-Dekodierung für Chip
  - niedrige Adressbits zur Registerauswahl im Chip
- Chip Select: CS
  - aus Adresse abgeleitet
  - z.B. mit kombinatorischer Schaltung

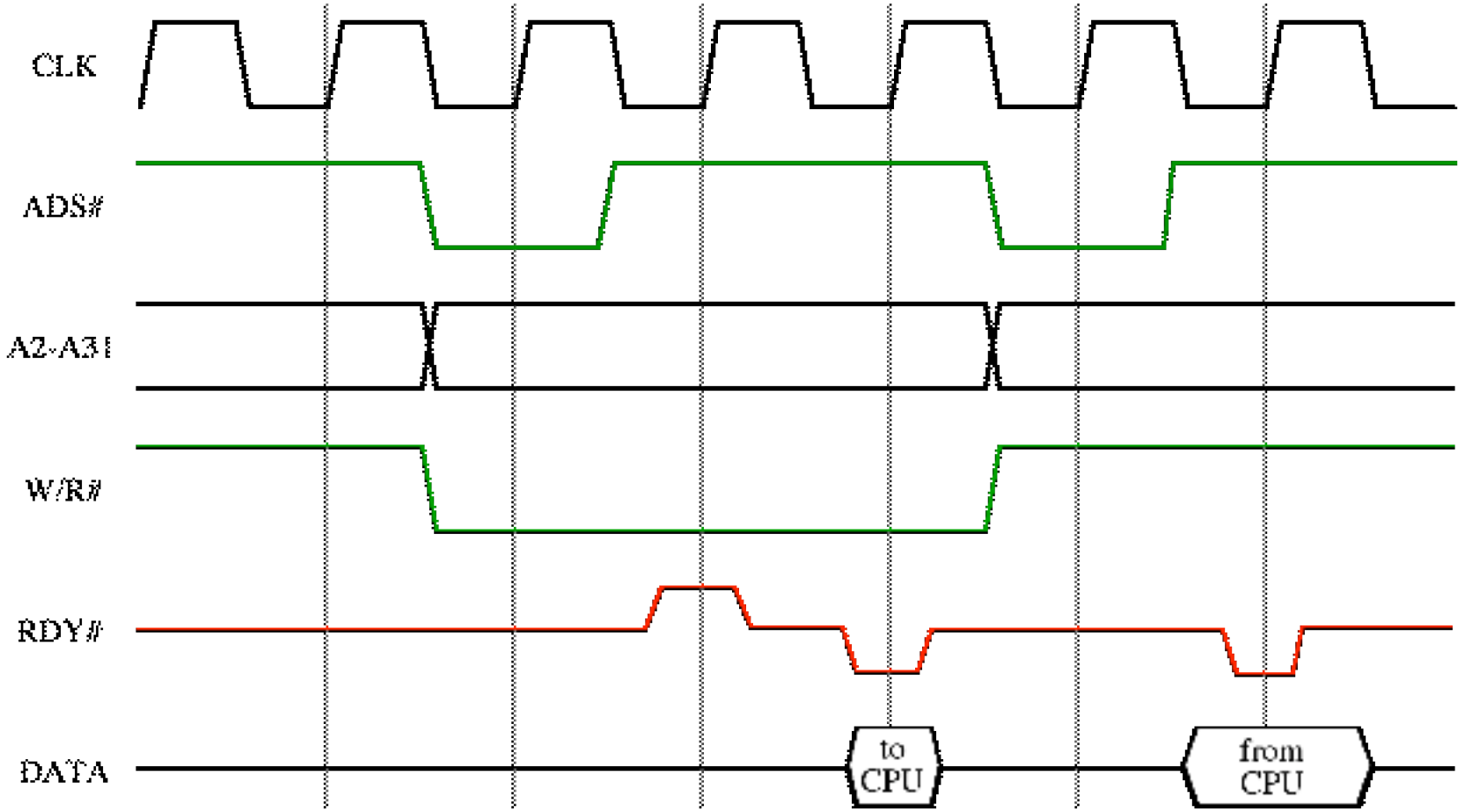
```
if (((unsigned) theAddress>>24) == 0x0FC)
```

- Programmable Logic
  - z.B. GAL 16V8, 22V10, …
  - Eingabe-Pins und Ausgabe-Pins
  - manche mit Flip-Flops
  - Und, Oder, Not
  - Enwicklungssysteme (Abel, …)

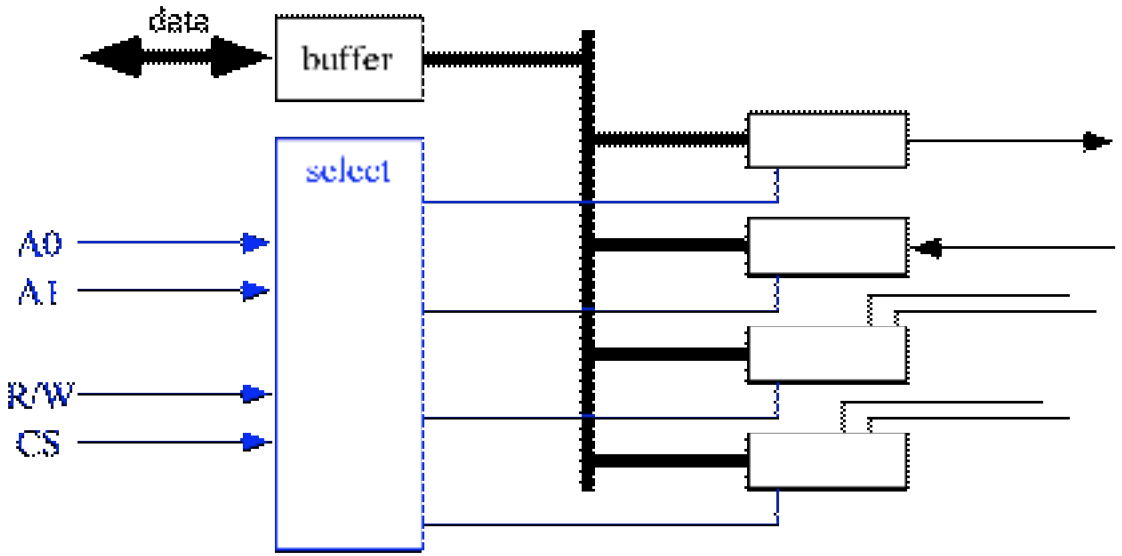
```
"/**  Inputs  **/
A24       Pin 1          ;      "/* address bus                    */
A25       Pin 2          ;      "/*                                */
A26       Pin 3          ;      "/*                                */
A27       Pin 4          ;      "/*                                */
A28       Pin 5          ;      "/*                                */
A29       Pin 6          ;      "/*                                */
A30       Pin 7          ;      "/*                                */
A31       Pin 8          ;      "/*                                */
"/**  Outputs  **/
MYCHIP    Pin 13         ;

Equations;
MYCHIP = !A24 & !A25 & A26 & A27 & A28 & A29 & A30 & A31
                  "/* (Adresse = FC XX XX XX) */
```

- Timing-Diagramm
  - Buszyklen von Intel 486
  - mit eingefügten Wait-State beim Read
  - # für invertierte Logik



- Instruktion zum Ansprechen von HW
  - Abstraktion: Register
  - LDV/STV mit Adressen
  - Adresse wählt Chip und Register (Chip-Select siehe oben)
  - evtl. A0 und A1 nicht auf dem Bus -> Registernummer*4



- Register für Kommandos
  - Bitketten
  - Bitkette lesen: Statusinformation
  - Bitkette schreiben: Kommando
  - Registerauswahl aus Subadresse im Chip

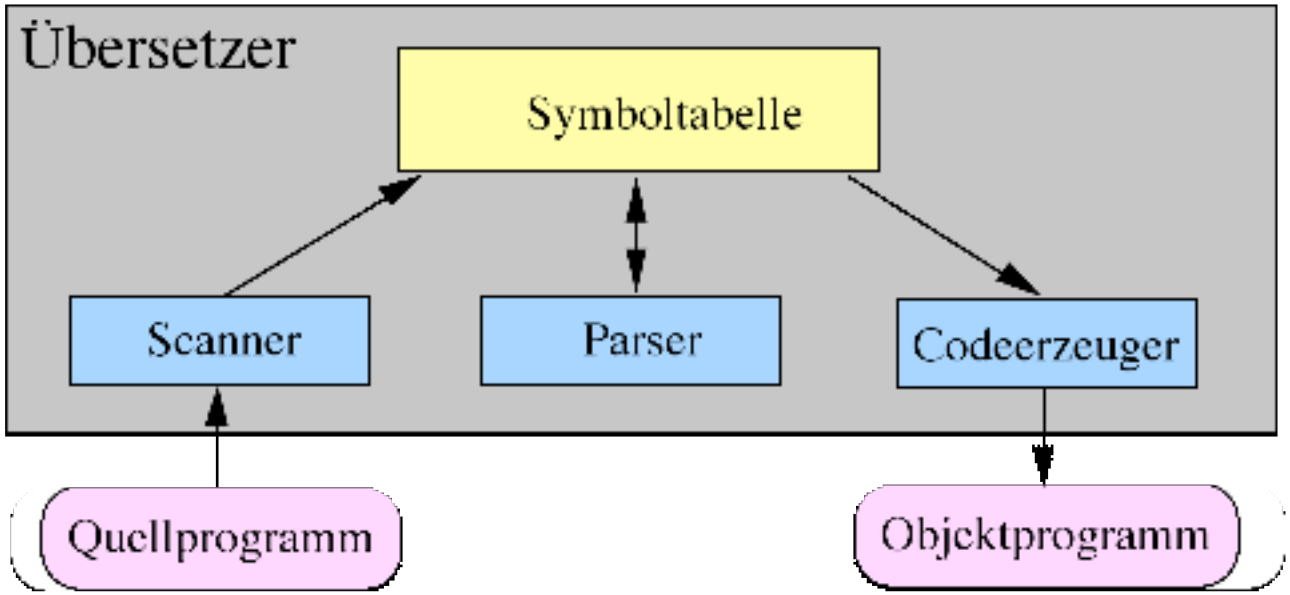
- Datenübergabe
  - Byte bzw. Wort
  - evtl. mehrere Byte lesen

```
int i;
volatile unsigned char *status = 0xFFA00;
volatile unsigned char *datareg = 0xFFA04;
unsigned char packet[1500]
…
i = 0;
while ((*status & $04) && i < 1500)
   packet[i++]= *chipdatareg;
```

- Kontrollblock
  - Kommando/Status und Daten als Record im Speicher
  - verkettete Liste, Ringpuffer , etc.

0.8 Compiler

- Brücke Programmiersprache - Objektcode
  - C, Pascal, Modula, Fortran,
  - IA, 68000, PowerPC, 8051, Z80, DSPs, …
  - Name => Adresse
  - Statement => Instruktionen
  - Prozeduraufruf => Sprung und Rücksprung



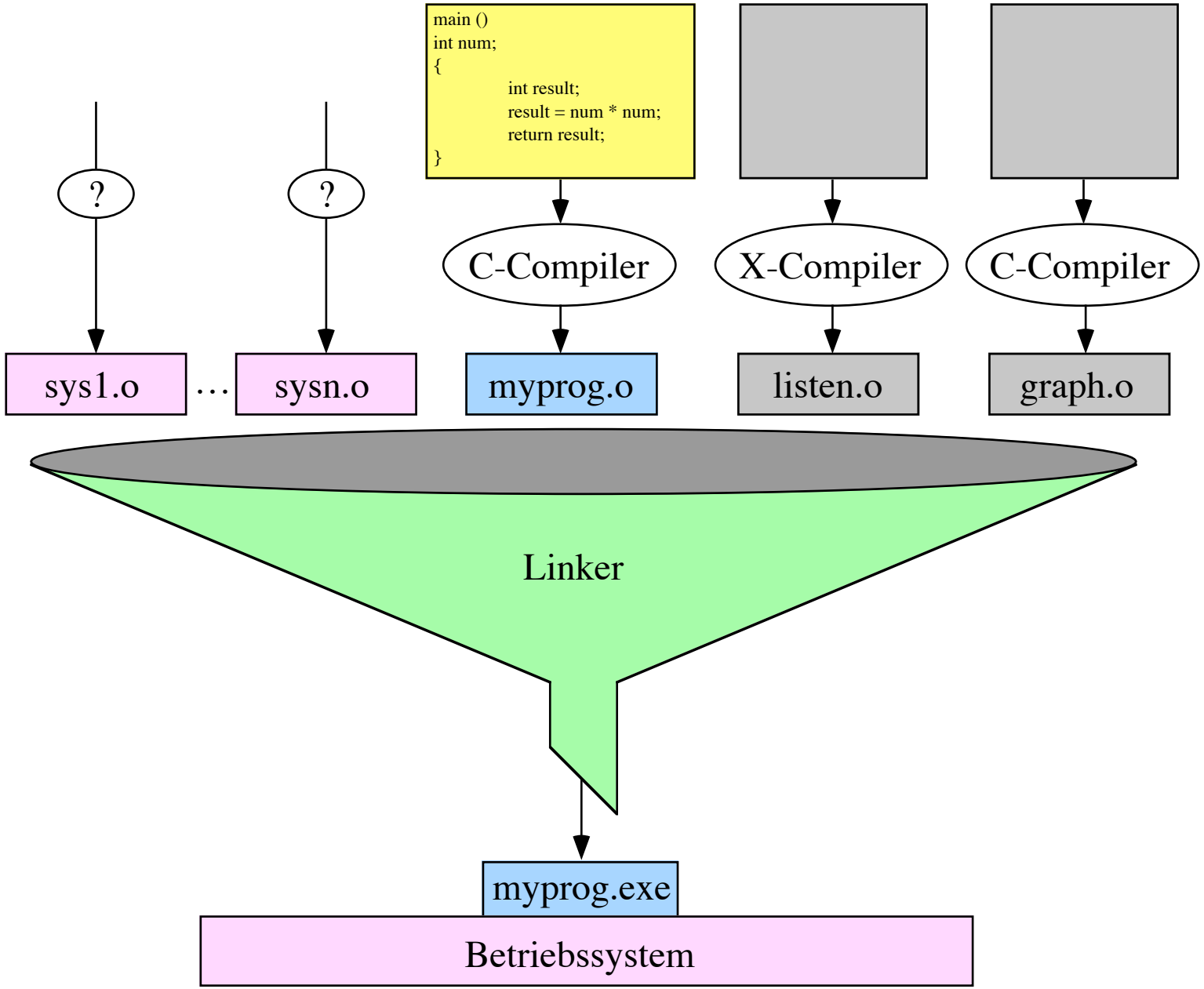
- Einlesen des Programmes (Scanner)
  - findet Symbole
  - Identifier, Konstanten, …
- Syntaktische Analyse
  - zulässige Symbole werden verarbeitet ("Parsing")
  - für unzulässige Symbole Fehlermeldungen erzeugen
  - über "Look-Ahead" entschieden, welcher Pfad gewählt werden soll
  - bei schwierigen Programmiersprachen sehr weit vorausschauen
  - LL1 Programmiersprachen => maximal 1 Symbol Look-Ahead.
- Erzeugen der Maschinenbefehle (Codegenerierung)
  - syntaktische Prozeduren können auch die Instruktionen erzeugen
- Strategien
  - rekursive decent
  - bottom-up
  - top-down
  - Übersetzung für virtuelle Maschine besonders einfach
  - zeilenweise Übersetzung

- Beispiel: Ausdruck übersetzen

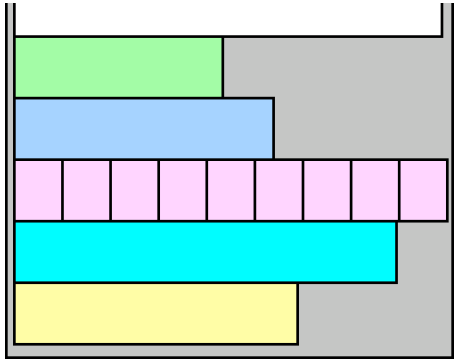
a:=b-(c-((d*e)-g/h))

```
LDV     g
DIV     h           ; g/h
STV     hilf        ; optimiert wird nicht
LDV     d
MUL     e           ; (d*e)
SUB     hilf        ; -
STV     hilf
LDV     c
SUB     hilf
STV     hilf
LDV     b
SUB     hilf
STV        a           ;   a:= ...
```

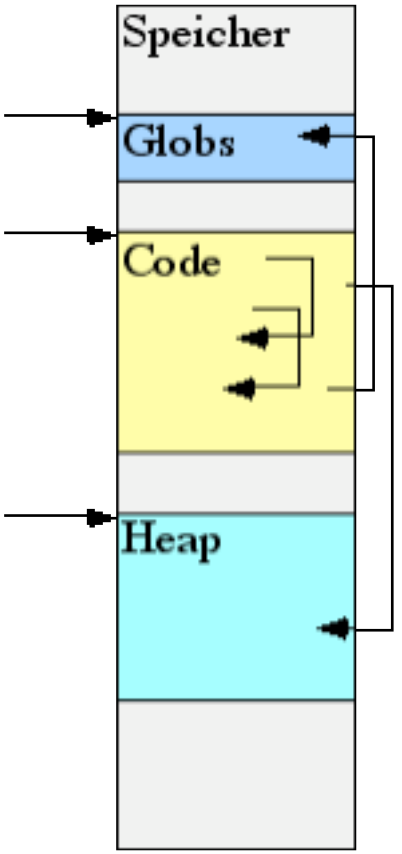
- Kompletter Programmierablauf



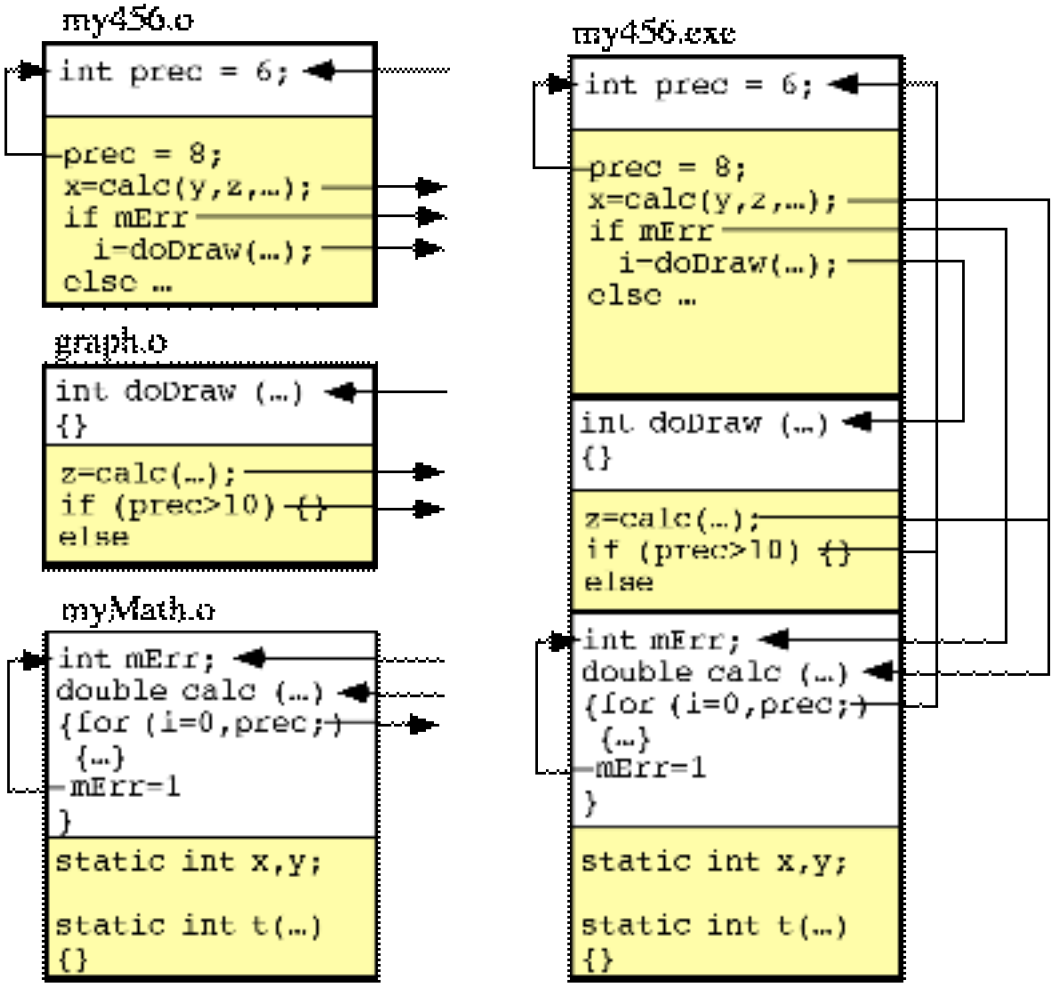
- Daten im Speicher
  - globale Variablen
  - lokale Variablen
  - Datenstrukturen
  - dynamische Datenstrukturen
  - Instanzvariablen => dynamische Datenstrukturen



- Adressierung
  - Befehle brauchen Adressen
  - LDV, STV, …
  - JMP, JMN
- Lage der Ziele zur Laufzeit?
  - Ladepunkt im Speicher
  - relative Adressierung der Sprünge
  - relative Adressierung der Daten mit Basisregister
  - besonderer Code für dynamische Bibliotheken (DLLs))



- Adressen in anderen Modulen?
  - Auflösung durch Linker
  - Auflösung durch Lader



0.9 Assembler

- Assemblersprache
  - menschenverständliche Form der Maschinenbefehle
  - Mnemonics: LDV, ADD, JMP, …
  - Variablen, Datenstrukturen
  - Makros
- Programm zum assemblieren
  - transformiert Assembler-Programme in Objektcode
  - Modularisierung
  - Speicheraufteilung
  - Sprünge berechnen
- Cross-Assembler
  - läuft auf Maschine A
  - erzeugt Code für Maschine B
  - für neue Computer oder kleine Architekturen
- Vorteile
  - Geschwindigkeit
  - kompakter Code
  - Zugang zu Spezialbefehlen

0.9.1 Elemente der Assemblersprache

- Vordefinierte Namen für die Instruktionen
  - aussagekräftig
  - instruction mnemonics"
- Namen für Speicheradressen:
  - frei wählbar
  - Instruktions- und Datenadressen
  - Sprungmarken
  - Variablen
- Literalkonstanten:
  - Dezimalzahlen
  - Hexadezimalzahlen
  - Zeichenkonstanten in Hochkommas
  - bedeuten Werte oder Adressen
- Kommentar nach Strichpunkt
- gegenwärtige Speicheradresse : *

0.9.2 Anweisungen der Assemblersprache

- Konstantenvereinbarung:

```
< name >      =    < wert >                    [; < kommentar >]
; acht       =    8
```

- Ladepunktanweisung:

```
*             =    < Adresse >                 [; < Kommentar >]
; *          =    start                       ; Anweisung für Assembler
```

- Speicherdeklarationsanweisung:

```
[< Name >]    DS  [< Wert >]                   [; Kommentar]
; index       DS  0                           ; ein Wort, nicht typisiert
```

  - Datentypen z.B. DL, DW, …

- Instruktionsanweisung:

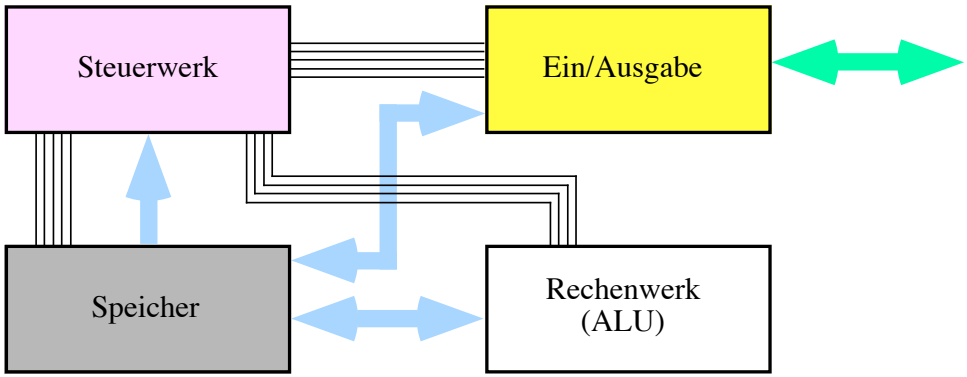
```
[Name]        Instruktionsname    [Operand [± Wert] ]  [; Kommentar]
; start       LDV                 index
```

  - **[± Wert]** Modifikation des Operandenwertes (-adresse) durch Assembler

- Bessere Assembler bieten auch Module, Scopes etc.

1. Taxonmie von Rechnerarchitekturen

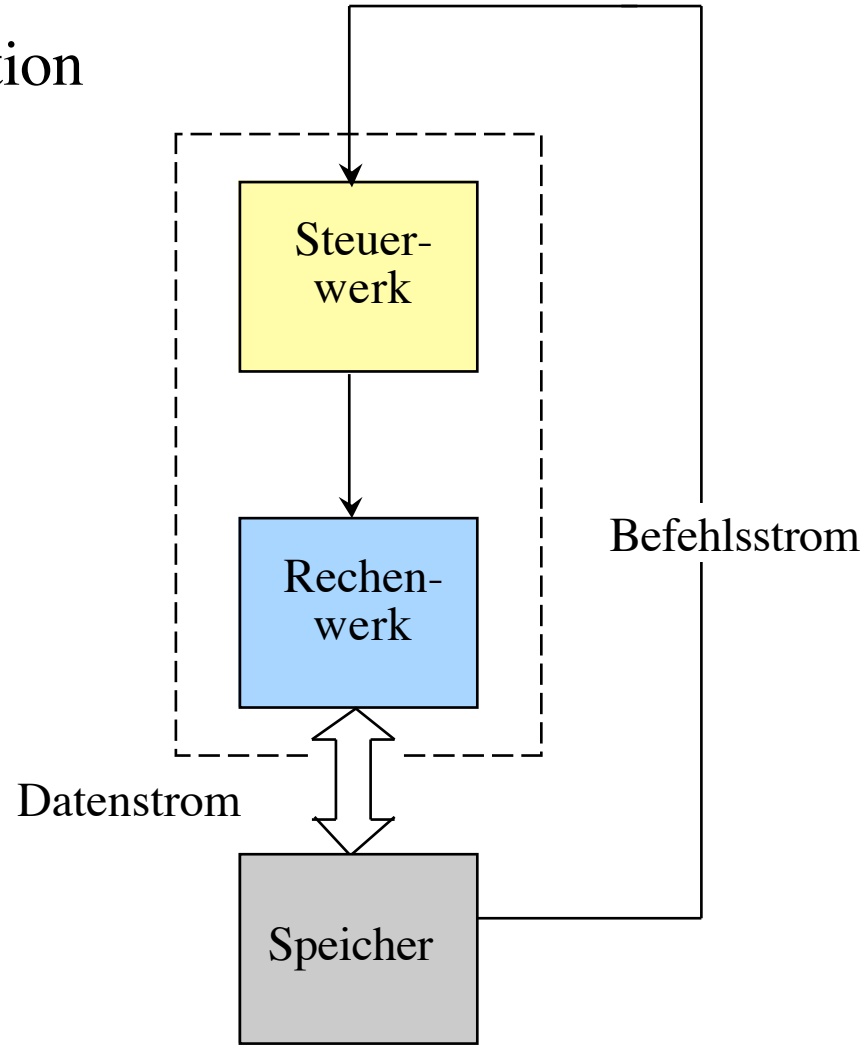
1.1 von Neumann



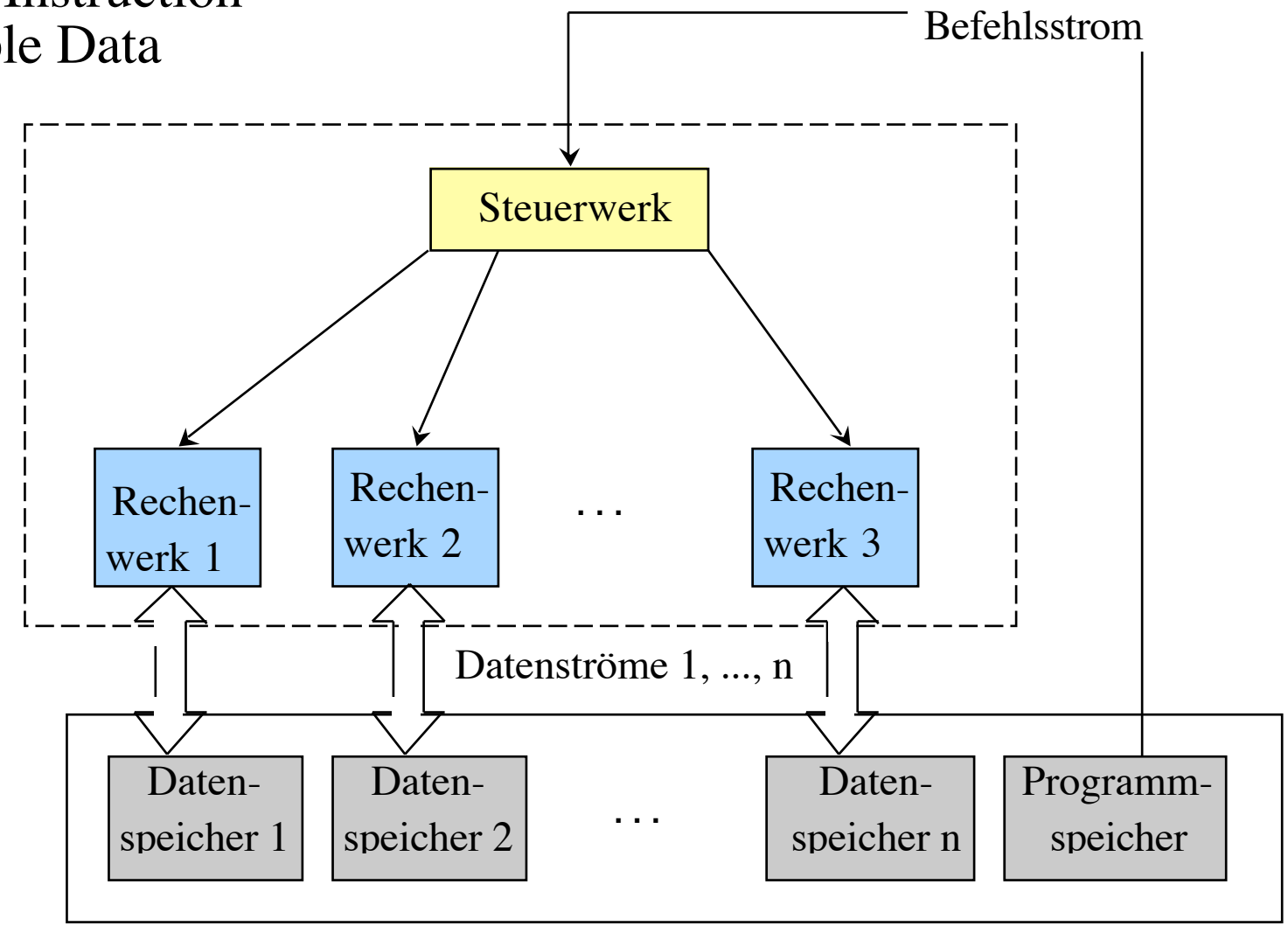
- Komponenten-Modell
- Befehle
  - ein Datenwert
  - seriell abgearbeitet
- Speicher
  - Code und Daten ohne Unterschied
  - kein eingebauter Zugriffsschutz
  - unstrukturiert und nicht typisiert: Semantik im Programm
  - linear adressiert
- Verbindung Speicher-CPU: von-Neumann Flaschenhals

1.2 Klassifikation nach Flynn

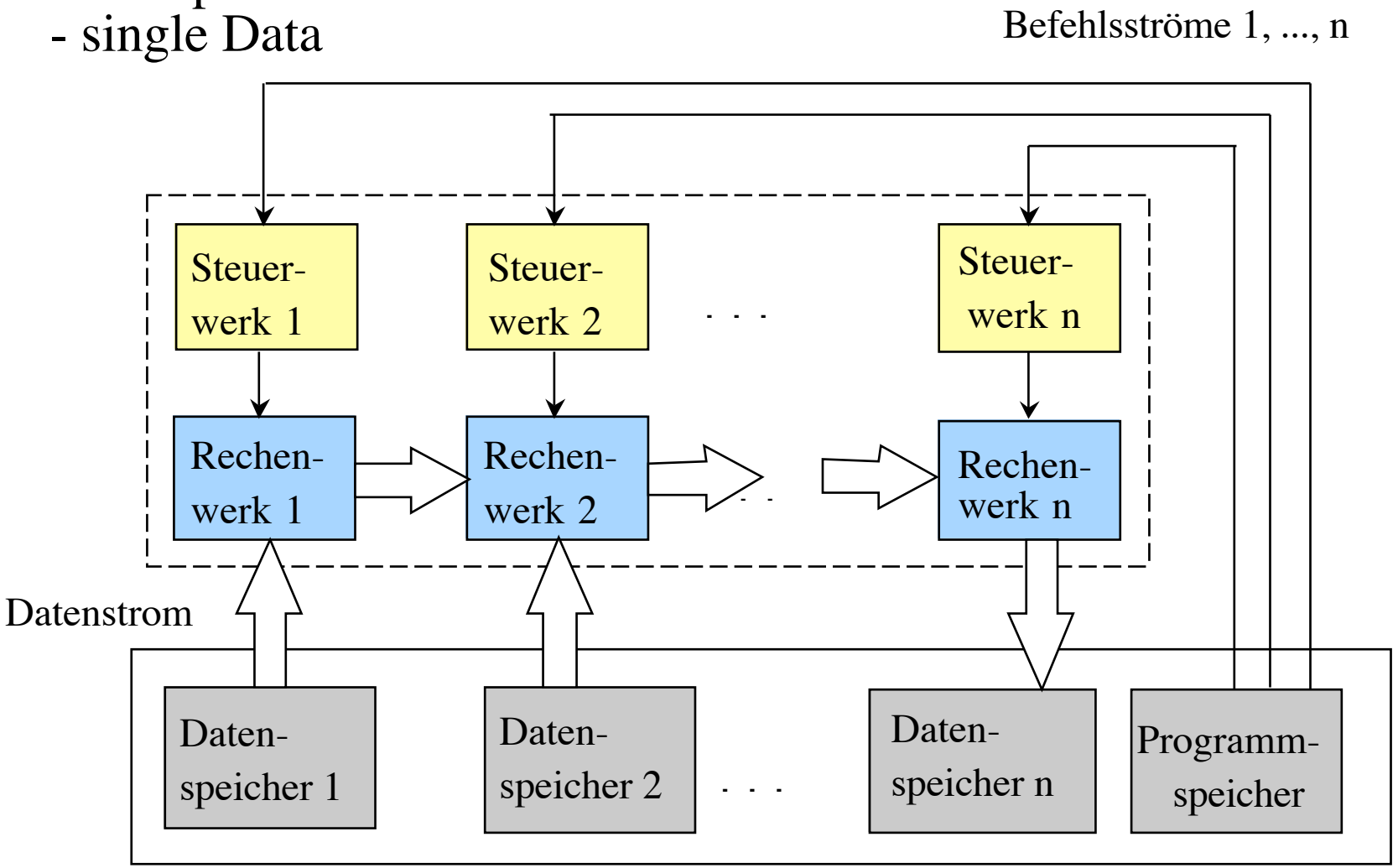
- SISD
  - single Instruction
  - single Data



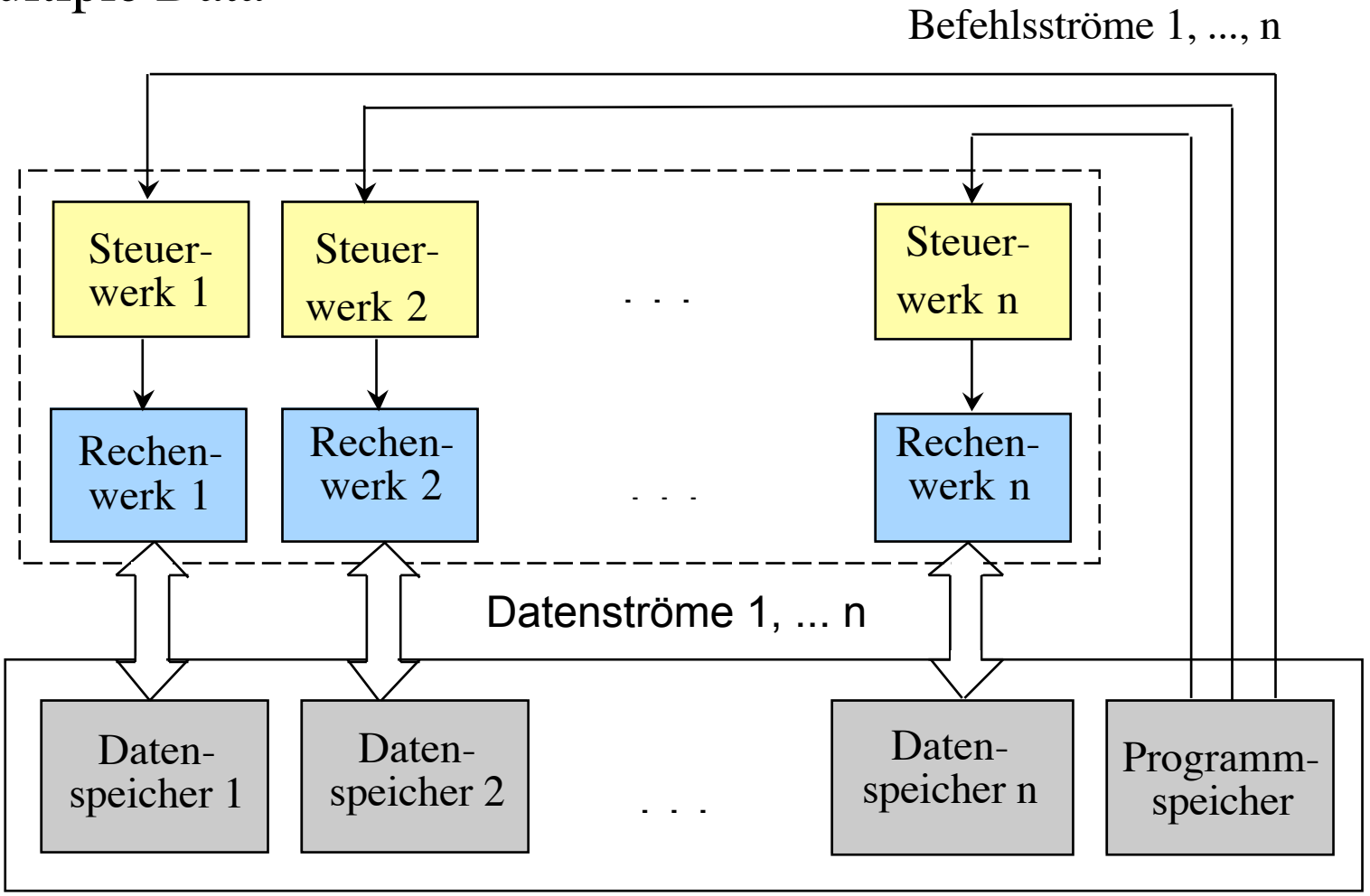
- SIMD
  - single Instruction
  - multiple Data



- MISD
  - multiple Instruction
  - single Data



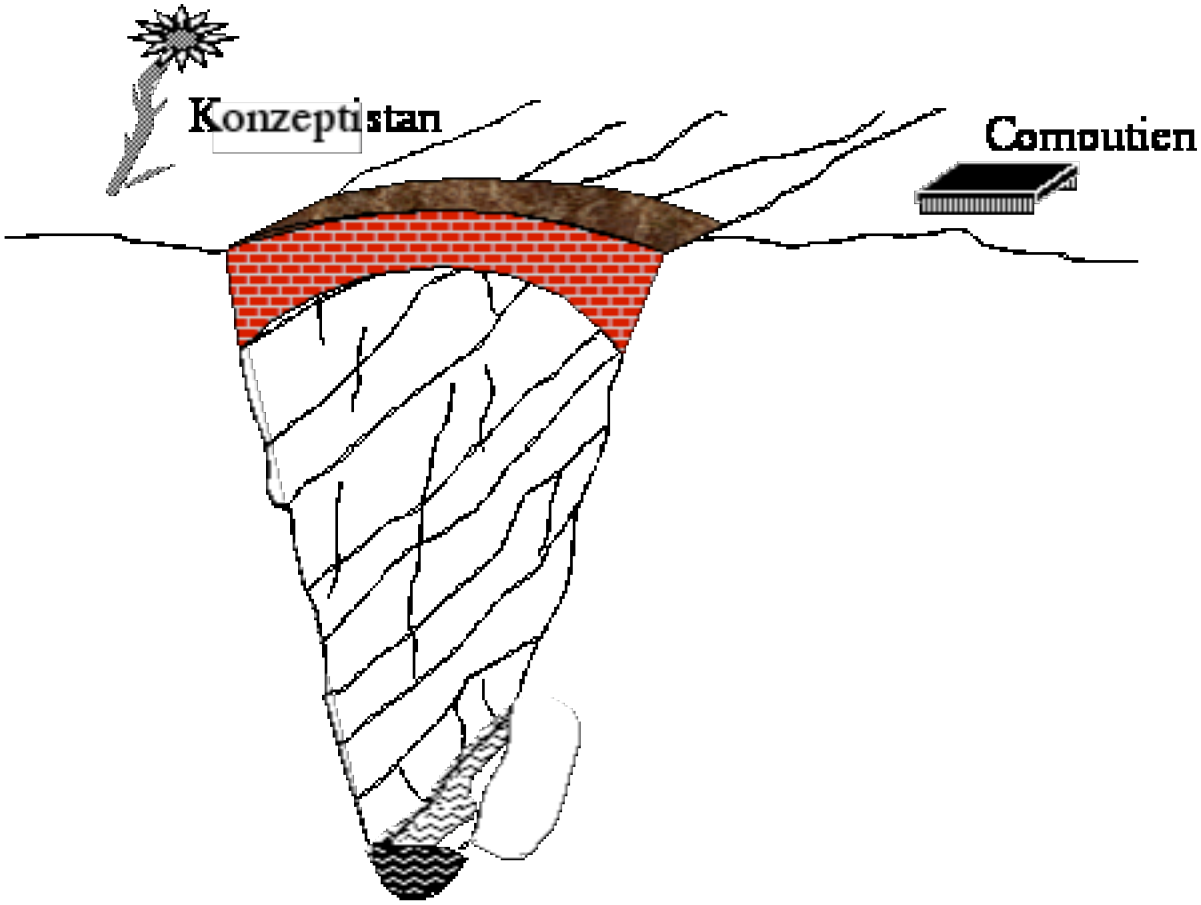
- MIMD
  - multiple Instruction
  - multiple Data



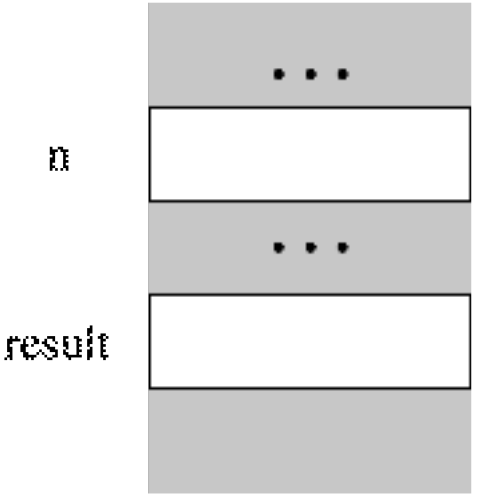
2. Instruktionssätze

- Programmiermodell
- Semantische Lücke
  - Datentypen
  - Objekte
  - Prozeduren
  - Formeln

  - RAM
  - Register
  - Operationen
- Datenfluss-Maschinen
- Mengen von Befehlen

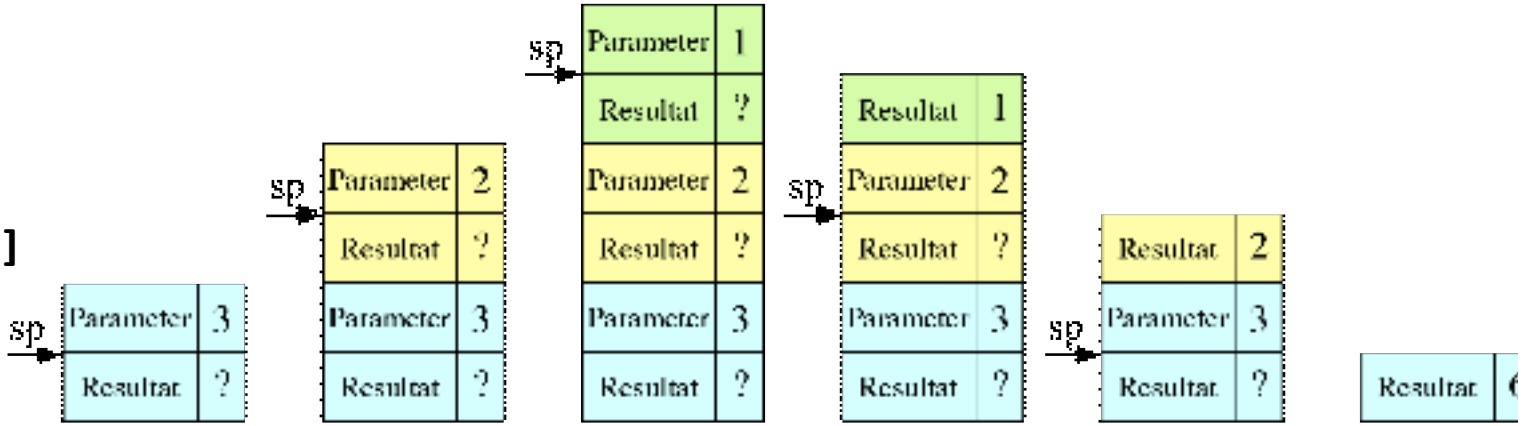


- Prozeduren und Parameter
  - Aufrufender: Platz für Parameter und Resultate
  - Gerufener: Platz für lokale Variablen
  - naiv: statische Allokation durch Compiler
  - Beispiel: fac(3)=1?

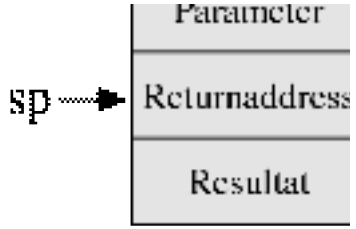



```
int fac(int n)
{ if (n==1) return 1;
  else return fac(n-1)*n;
}
…
fac(3);
```

```
if (mem[sp]==1)
{   mem[sp-1]=1
    "return" }
mem[sp+2]=mem[sp]
sp=sp+2
"call"
sp=sp-2
mem[sp-1]=mem[sp+1]*mem[sp]
"return"
```

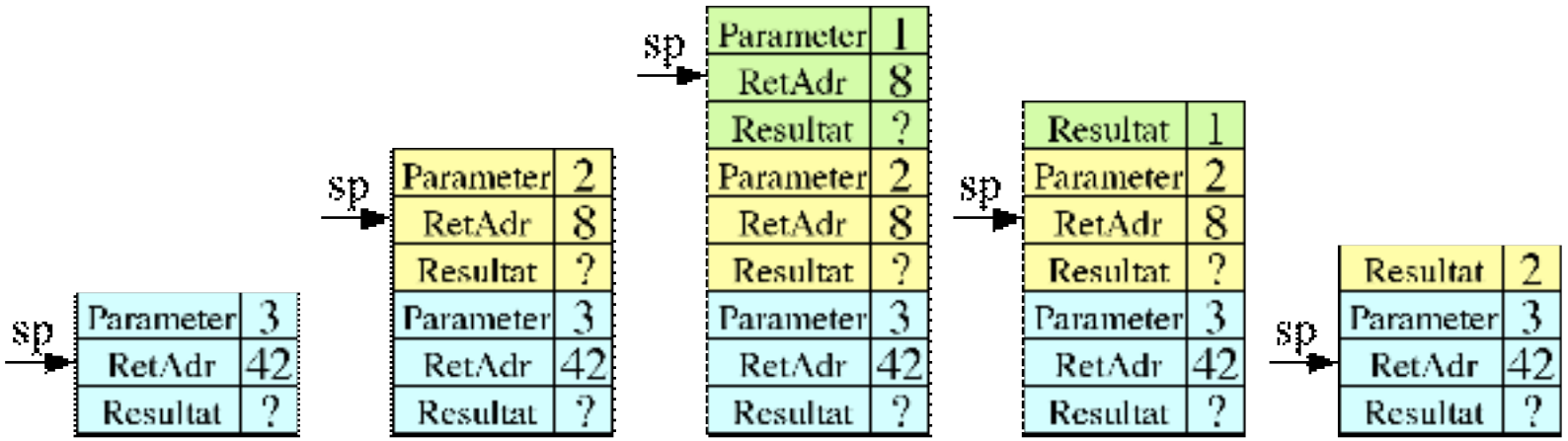


- Integration der Return-Adresse
  - Stackframe
  - z.B. Aufrufer verwaltet Stackframe

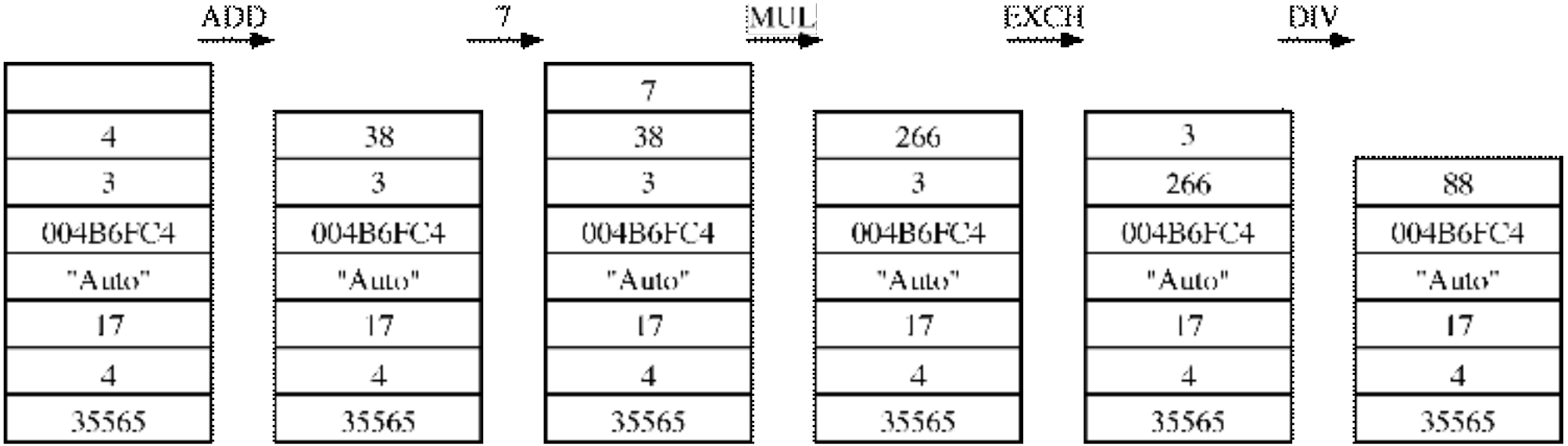



```
01 if (mem[sp+1]>1) jump 04
02 mem[sp-1] = 1
03 jump 10
04 mem[sp+4] = mem[sp+1] - 1
05 sp=sp+3
06 mem[sp] = 08
07 jump 01
08 sp=sp-3
09 mem[sp-1] = mem[sp+1] * mem[sp+2]
10 jump mem[sp]
…
38 mem[sp+4]=3
39 sp=sp+3
40 mem[sp]=42
41 jump 1
42 sp=sp-3
```

```
int fac(int n)
{ if (n<=1)
    return 1;
  return fac(n-1)*n;
}
```



- Stack als Maschinenmodell
  - Java Virtual Machine
  - p-code, Interpreter
  - PostScript



- Speicher-Speicher
- Register-Modelle
  - Akkumulator (siehe Mima)
  - GPR: General Purpose Register
- GPRs
  - Register-Speicher
  - Load-Store
  - viele Register -> viele Bits in der Instruktion (Länge!)
  - wenige Register -> viele Speicherzugriffe
- Vergleich für `c = a+b;`

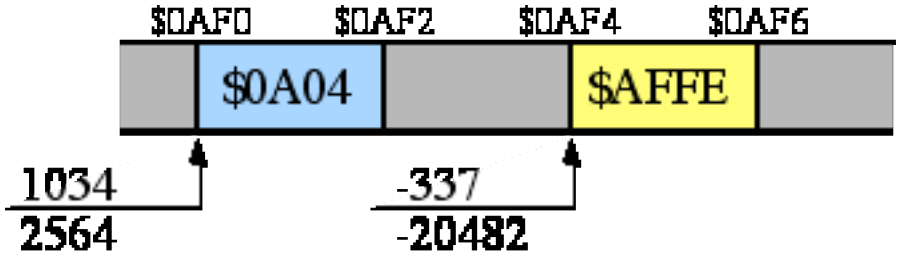
Stack	Akkumulator	Register-Speicher	load-store
`Push a`	`LDV a`	`Move a,R1`	`Load R1,a`
`Push b`	`ADD b`	`Add b,R1`	`Load R2,b`
`Add`	`STV c`	`Move R1,c`	`Add R3,R1,R2`
`Pop c`			`Store c, R3`

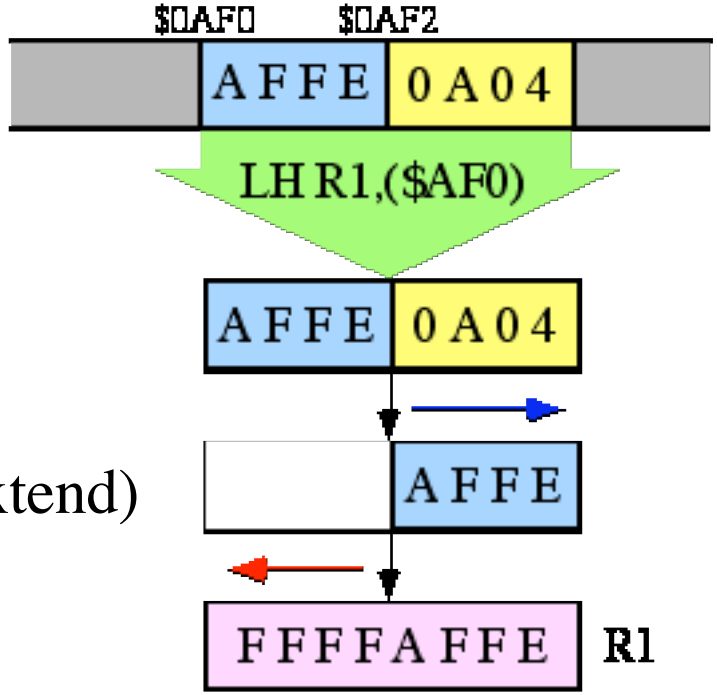
- Konventionen für GPRs
  - Pointer: Stack, Procedure-Frame, Globals, With, …
  - Procedure-Parameter
  - Temporäre Variablen, Expression-Eval, …
- Anzahl Operanden
  - 2: Operation Op1, Op2/Ziel
  - 3: Operation Ziel, Op1, Op2
  - alle Register/Memory?
- GPR-Maschinen

Typ	Beispiel	Vorteile	Nachteile
Reg-Reg (0,3)	SPARC, PPC, Mips, Alpha, HP-PA	Feste Befehlslänge, einfache Code-Generierung, feste Taktlänge	Mehr Instruktionen, evtl. Bits verschwendet
Reg-mem (1,2)	80x86, 68000	gute Codedichte, einfach zu codieren	unterschiedliche Befehlslänge, Operand verloren (Erg.)
Mem-Mem (3,3)	VAX	sehr kompakt	Befehle unterschiedlich lang, Speicherflaschenhals, unterschiedliche Taktanzahl

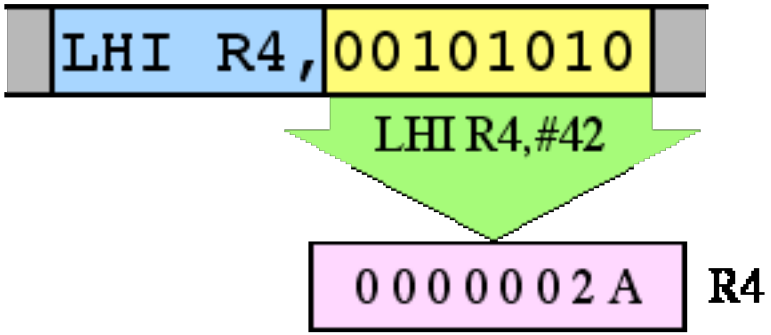
2.1 Adressierung und Operanden

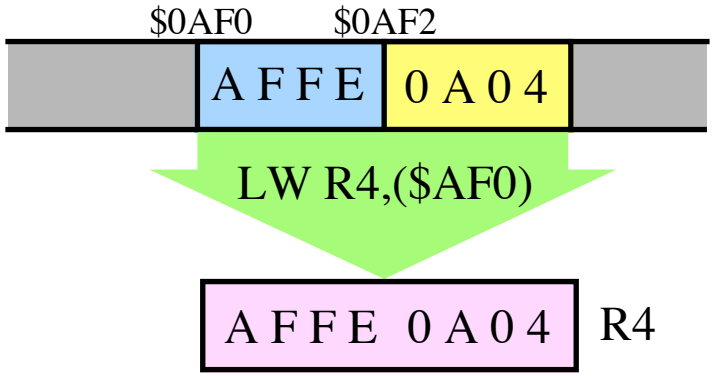
- Operandenlänge
  - 8, 16, 32, 64 Bit
  - Bitzugriff meist Bytezugriff
- Little-Endian
  - niederwertigstes Bit/Byte an der Adresse
  - Intel, Alpha
- Big Endian
  - höchstwertigstes Bit/Byte an der Adresse
  - 68000, PPC, …
- Alignment
  - Halbwort-Adressen durch 2 teilbar
  - Wortadressen durch 4 teilbar
  - 8, …
  - Gesamtes Speicherwort wird geholt
  - Schaltung zum Ausrichten (Shift + Sign-Extend)
  - Auswirkung auf Rest-Register?



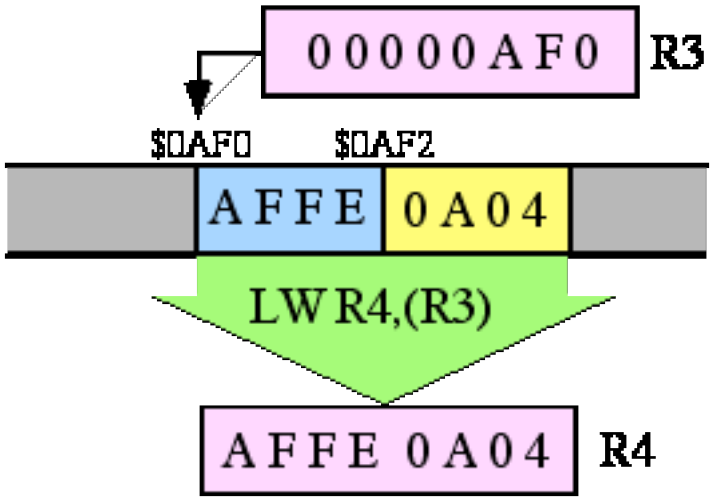


- Adressierungsmodi
  - Mima: Adresse in der Instruktion (einfache Variable)
  - Mima: Operand in der Instruktion (Konstente)
  - weitere Modi: indirekte Adressierung (Pointer, …)
- Immediate
  - Operand im Befehl
  - `LHI R4,#42`
  - Konstante laden
  - Shift, ALU-Ops, Compare, …
- Direkt (absolut)
  - EffektiveAdresse = Instruktion AND 007FFFFF
  - `LW R4,($0AF0)`
  - Adresselänge beschränkt
  - oder Instruktion 2 Worte

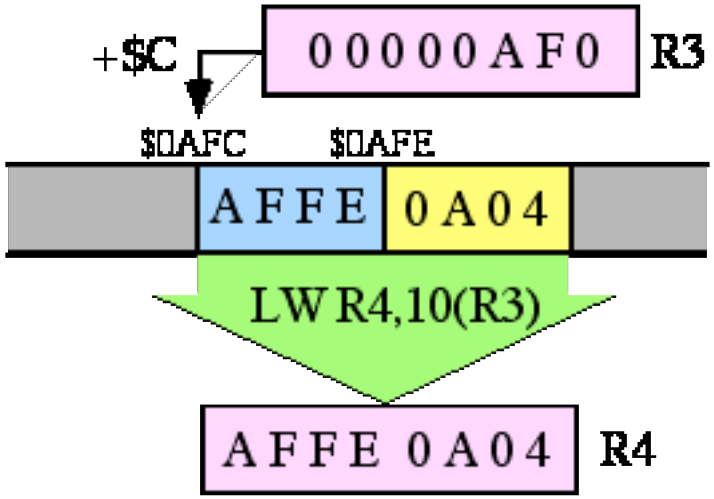




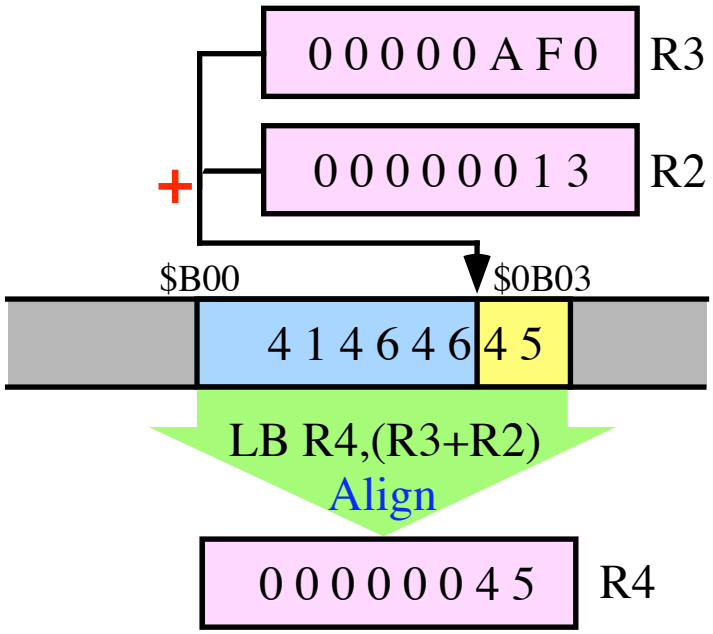
- Register Indirekt
  - EA = Rn
  - `LW R4,(R3)`
  - Pointer
  - Adressrechnung
  - komplexe Arrays



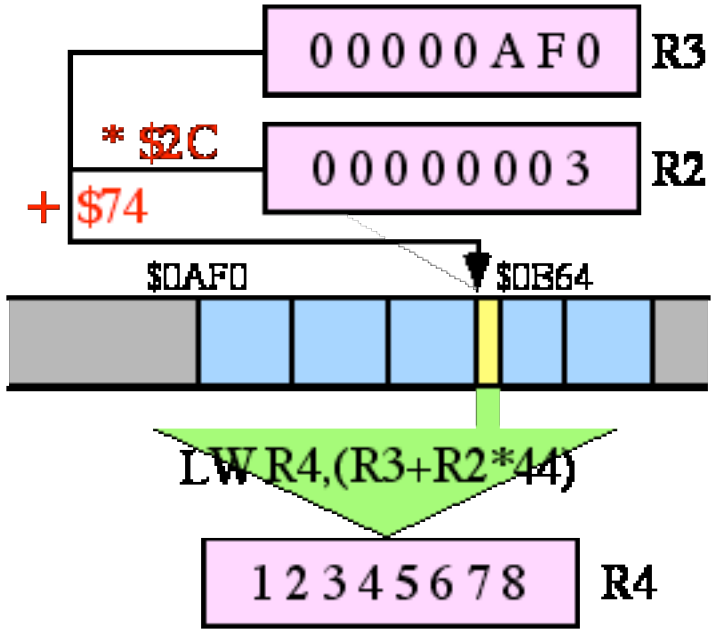
- Register Indirekt mit Displacement
  - EA = Rn+Disp
  - `LW R4,12(R3)`
  - Feld im Record
  - lokale Variablen (Stackframe), …



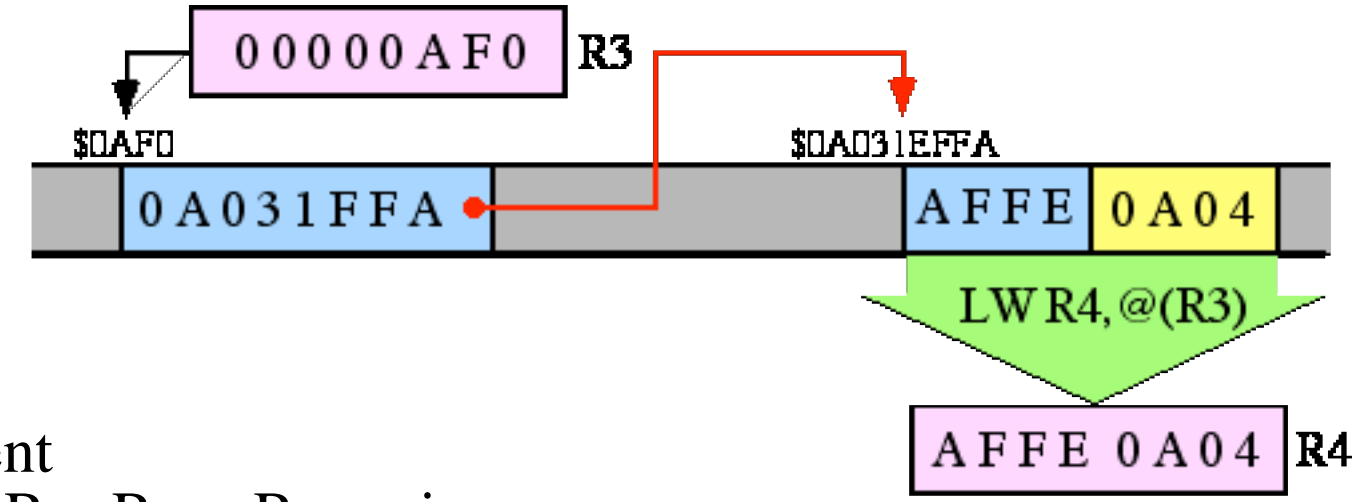
- Register Indirekt mit Index
  - EA = Rn + Ri
  - `LB R4,(R3+R2)`
  - z.B. Char-Array
  - R3 Zeiger auf Vektor
  - R2 z.B. Schleifenindex



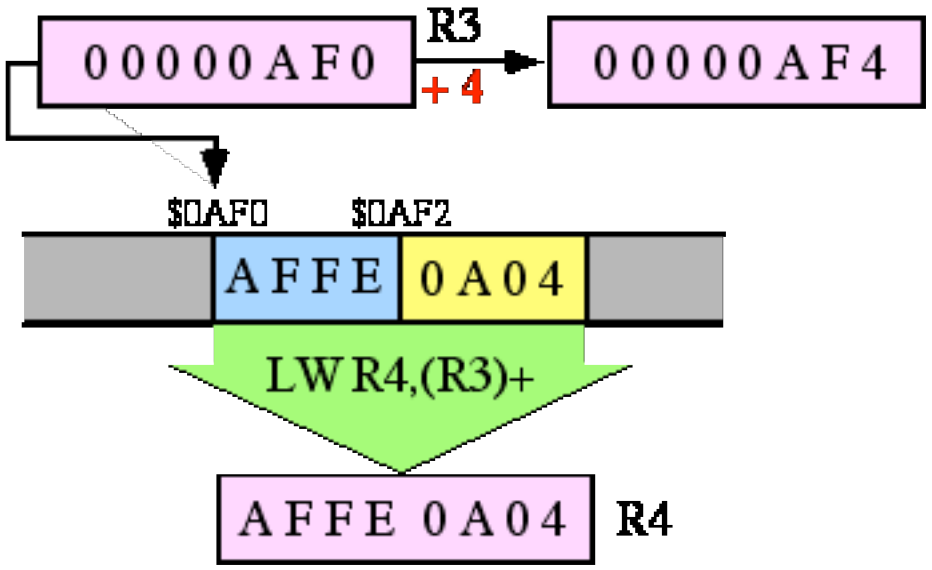
- Register Indirekt mit Index*Scale
  - EA = Rn + Ri*scale
  - `LW R4,(R3+R2*44)`
  - Vektor mit Elementen > Byte



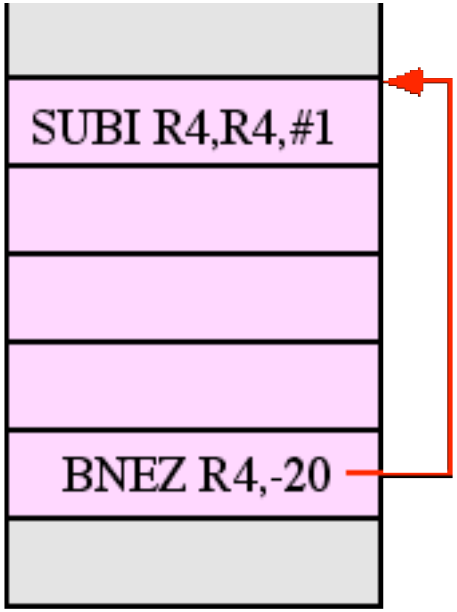
- Speicher Indirekt
  - EA = Speicher[Rn]
  - `LW R4,@(R3)`
  - Handles dereferenzieren



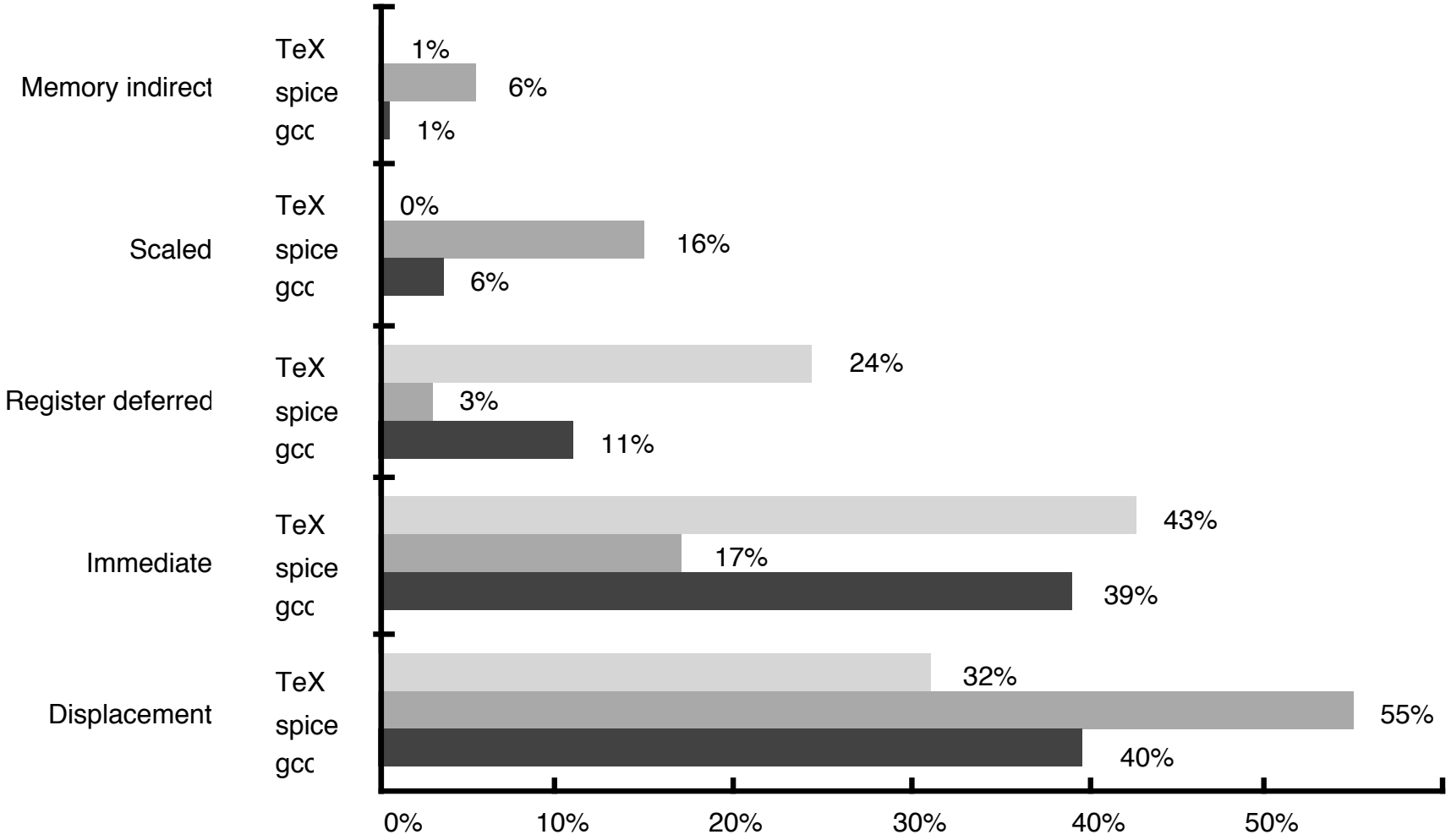
- Increment und Dekrement
  - post-operation: EA = Rn; Rn = Rn ± size
  - pre-operation: Rn = Rn ± size; EA = Rn;
  - Operandentyp ergibt Increment-Wert
  - `LW R4,(R2)+`
- Architekturspezialitäten
  - segmentbasiert (Datensegment beim 8086, …)
  - PC-relativ (strings von Prozeduren)



- Sprungadressen: PC-relativ
  - Ziel = PC+displacement
  - `if-then-else` etc.
  - `for, while, …`
  - positionsunabhängiger Code
- Sprungadressen: Register-indirekt
  - Ziel = Rn, Rn+Ri…
  - ähnlich Datenadressmodi
  - case/switch, DLLs, virtuelle Funktionen, Sprungtabellen, …
- Absolute Sprünge
  - relozieren beim Laden
  - Standard-Adressraum?



- Häufigkeit der Adressierungsarten (VAX)



=> `LW R4,@(R3)` ersetzen durch: `LW R2,(R3)` und `LW R4,(R2)`

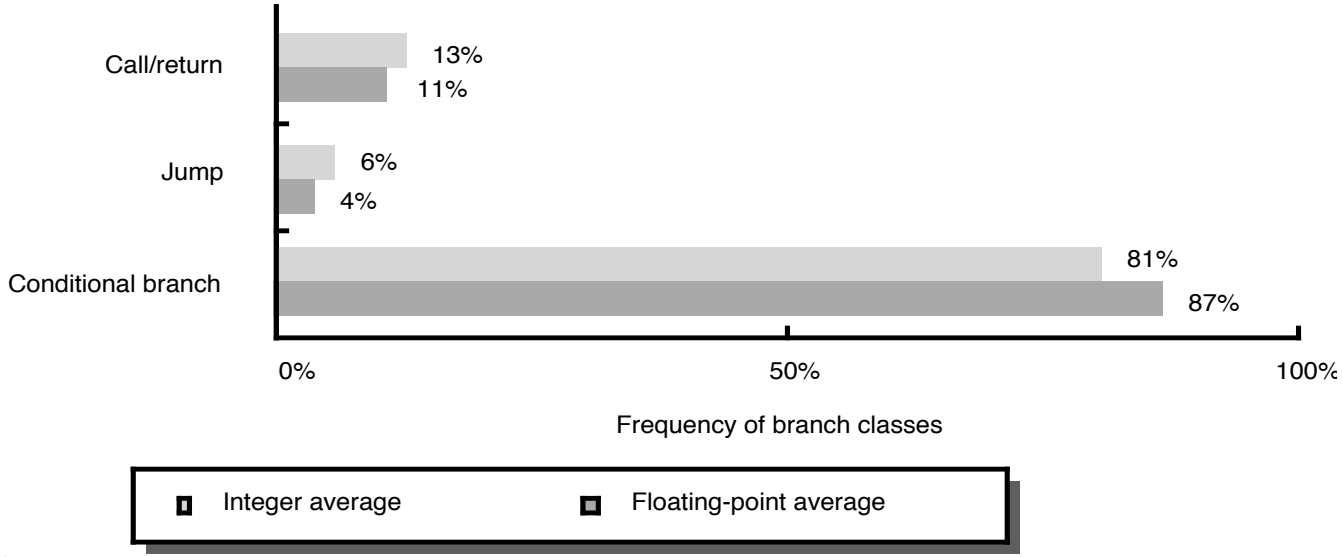
- Instruktionssatzentwurf
  - nur compilererzeugbare Adressmodi
  - seltene Adressierungsarten ersetzen
- Wichtige Adressierungsarten
  - Indirekt mit und ohne Displacement
  - Immediate
  - mehr als 75%
- Displacement-Feldgröße
  - 12 - 16 Bit
  - mehr als 75%
  - Displacement zu groß => Adressrechnung
- Immediate-Feldgröße
  - 8 - 16 Bit
  - 50%-80%
  - Immediate zu groß => Konstante im Speicher

2.2 Operationen

- Instruktionsgruppen
  - Kontrollfluß
  - Datentransfer
  - Arithmetische und logische Operationen
  - Vektorbefehle (SIMD), Multiply-Accumulate (MISD), …
  - Floating Point
  - Systemaufrufe
  - Dezimal, String
- Lokale Verzweigung
  - aus Programmiersprachenkonstrukten
  - J, JR; B, …
  - geringes Displacement: -128 .. 127
- Sprung
  - größere Distanz
  - J, JR; JMP, J, …
  - evtl. tabellisiert
  - typisch register- oder speicherindirekt

80x86 Befehlssatz

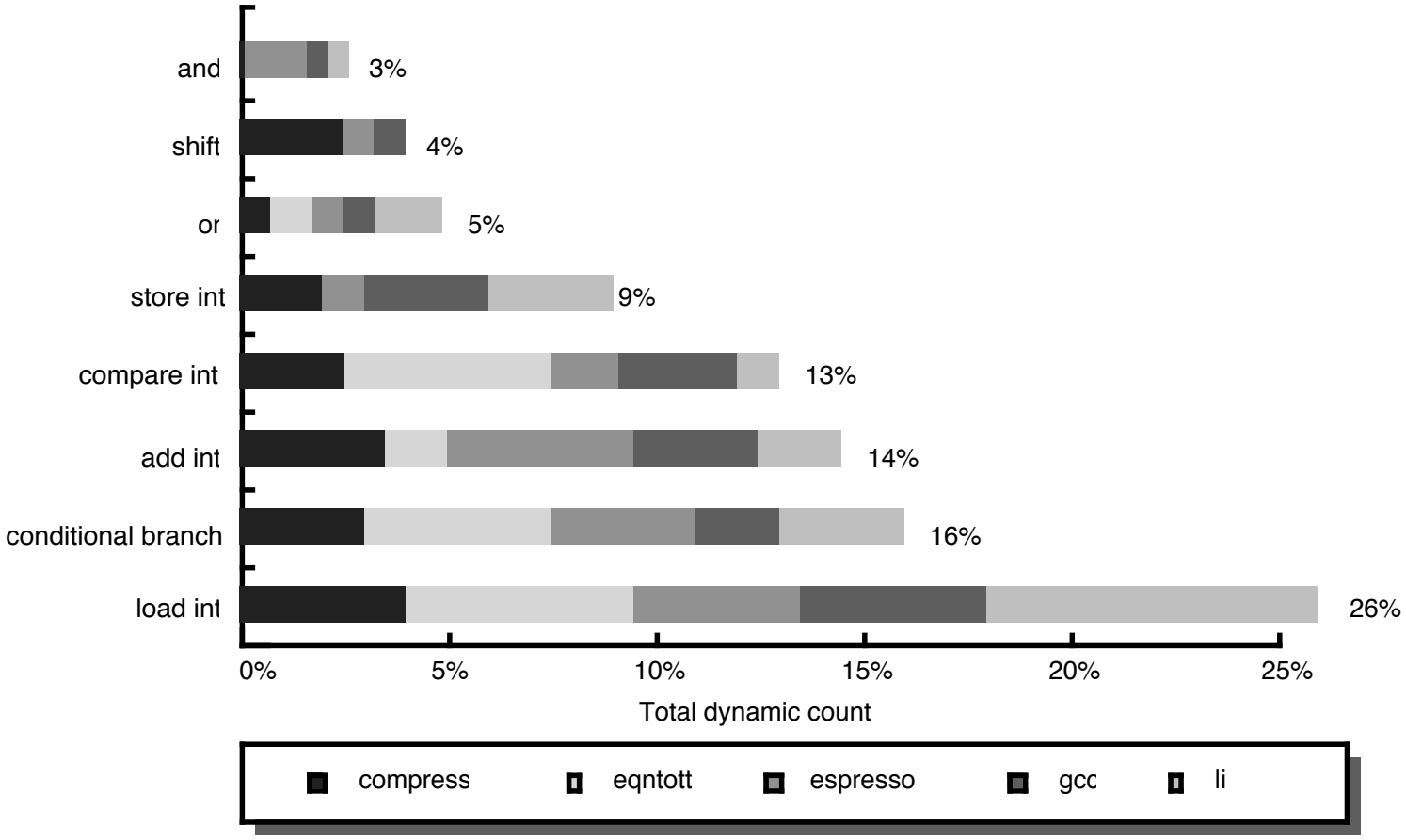
load	22%
conditional branch	20%
compare	16%
store	12%
add	8%
and	6%
sub	5%
move register-register	4%
call	1%
return	1%
(SpecInt92)	95%



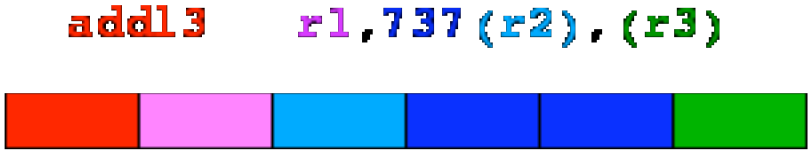
- Bedingte Verzweigung
  - Condition Code
  - von ALU gesetzt
  - evtl. beim Laden gesetzt
  - Zero, Negative, Carry, Borrow, …
  - Compare-Befehl
- Prozeduraufrufe
  - Hin-Sprung (JAL, JALR)
  - lokal: mittleres Displacement
  - auch entfernt
  - evtl. tabellisiert (DLL, Interruptvektor,)
  - Rücksprung (JR, ; RET)
  - Register retten

- Arithmetische und logische Operationen
  - Integerarithmetik: Add, Sub, Multiplikation, Division
  - And, Or, …
  - Compare
  - Bitfeldoperationen
- Floating Point
  - Koprozessor-Konzept
  - Hardware und evtl. Software
  - eigene Register, Transferbefehle
  - CPU muss trotzdem eff. Adresse generieren
  - evtl. nur Transfer aus GP-Registern
  - Datenformate: IEEE 754
- Konfiguration und Management
  - Interruptsperren
  - Version
  - Prozessorstatuswort
  - Test-and-Set
  - Interrupt (TRAP, RFE; IRET)

- Datentransfer
  - Load und Store bzw. Move
  - Schwerpunkt Adressierung
  - Register-Spec
  - Speicher-Adresse
- Relative Häufigkeit (DLX, SpecInt92)



- Effiziente Kodierung der Befehle
  - Operation und Register
  - Adressen und Immediates
  - kompakt: Kodedichte und schnelles Laden
  - Platz für Adressen und Immediates
  - einfach dekodierbar
  - leicht generierbar
- Opcode
- Adress-Spezifikation
  - Adressierungsmodus
  - eigenes Feld
  - Register
- Instruktion soll in 1 Wort passen
  - 32 Bit, 64 Bit
  - Verschnitt bei vielen Befehlen
- Pipeline-Effizienz



- Amdahls Gesetz
- Den häufigen Fall beschleunigen
  - tatsächlich ausgeführte Instruktionen

$$Speedup = \frac{1}{(1 - schnellerer\_Anteil) + \frac{schnellerer\_Anteil}{Beschleunigung}}$$

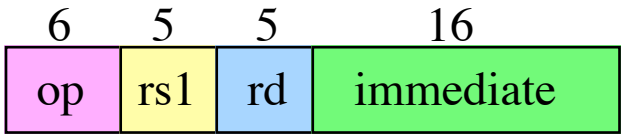
  - Bsp: 90% Anteil 10% Beschleunigung

$$Speedup = \frac{1}{0,1 + \frac{0,9}{10}} = \frac{1}{0,19} = 5,2631$$

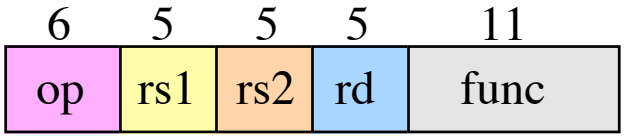
- Beispielapplet
  - http://www.cs.iastate.edu/~prabhu/Tutorial/CACHE/amdahl.html

$$CPU - Zeit = Anzahl\_Inst * \frac{Zeit}{Zyklus} * \frac{Zyklen}{Befehl}$$

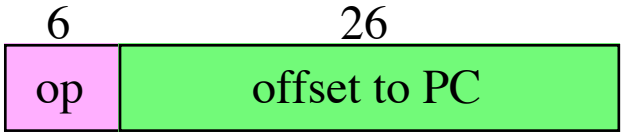
- DLX-Architektur
  - Load/Store, 32 GPRs, R0=0
  - 32 Bit, Big Endian
  - 8, 16, 32 Bit Integer
  - Adressierung indirekt mit 16 Bit Displacement
  - Adressierung immediate mit 16 Bit
  - 32 Floating Point Register (32 Bit) = 16 * 64 Bit
- DLX-Instruktionskodierung
  - 32 Bit Befehle
  - 6 Bit Opcode inkl. Adressierung
  - ALU-Funktion 11 Bit
- I(mmediate)-Typ
  - load, store, …
  - conditional branch, jump register, …



- R(egister)-Typ
  - ALU rd := rs1 func rs2



- J(ump)-Typ
  - Jump, Jump&Link, Trap, RTE



- Datentransfer
  - zwischen Registern und Speicher
  - zwischen Registern und FP-Registern
  - Adressierung 16-bit displacement + contents of a GPR

LB, LBU, SB	Load byte, load byte unsigned, store byte
LH, LHU, SH	Load halfword, load halfword unsigned, store halfword
LW, SW	Load word, store word (to/from integer regs)
LF, LD, SF, SD	Load SP float, load DP float, store SP float, store DP float
MOVI2S, MOVS2I	Move from/to GPR to/from a special register
MOVF, MOVD	Copy FP reg or a DP pair to another reg or pair
MOVFP2I, MOVI2FP	Move 32 bits from/to FP reg to/from integer reg

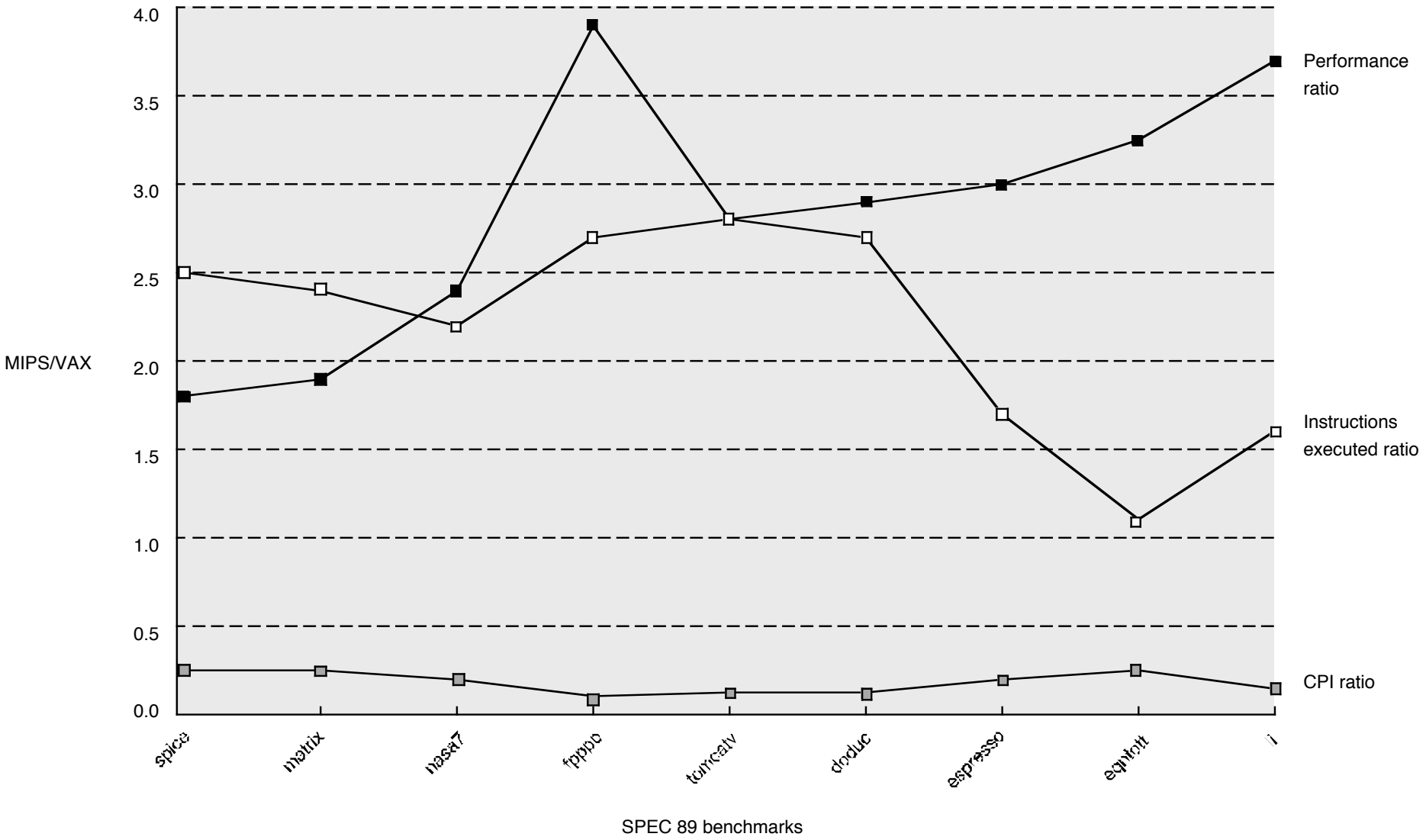
- Arithmetic / Logical
  - Integer Bzw Bitketten in Registern
  - Overflow => Trap

ADD, ADDI, ADDU, ADDUI	signed and unsigned Add, add immediate (all immediates are 16-bits);
SUB, SUBI, SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT, MULTU, DIV, DIVU	* und /, signed bzw.unsigned Operanden floating-point register
AND, ANDI	AND, AND immediate
OR, ORI, XOP, XOPI	OR, OR immediate, Xor, XOR immediate
LHI	Load high immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts
Sxx, SxxI	Set conditional: xx: LT, GT, LE, GE, EQ, NE

- Control
  - Conditional branches and jumps
  - PC-relativ oder Register-indirekt
  - 16-bit offset from PC

BEQZ, BNEZ	Branch GPR =/≠ zero
BFPT, BFPF	Test compare bit in FP status register and branch;
J, JR	26-bit Offset vom PC(J) oder Ziel in Register (JR)
JAL, JALR	Jump and link: R31:= PC+4;
	Ziel PC-relative (JAL) oder Register (JALR)
TRAP	Transfer to operating system at a vectored address
RFE	Return to user code from an exception; restore user code

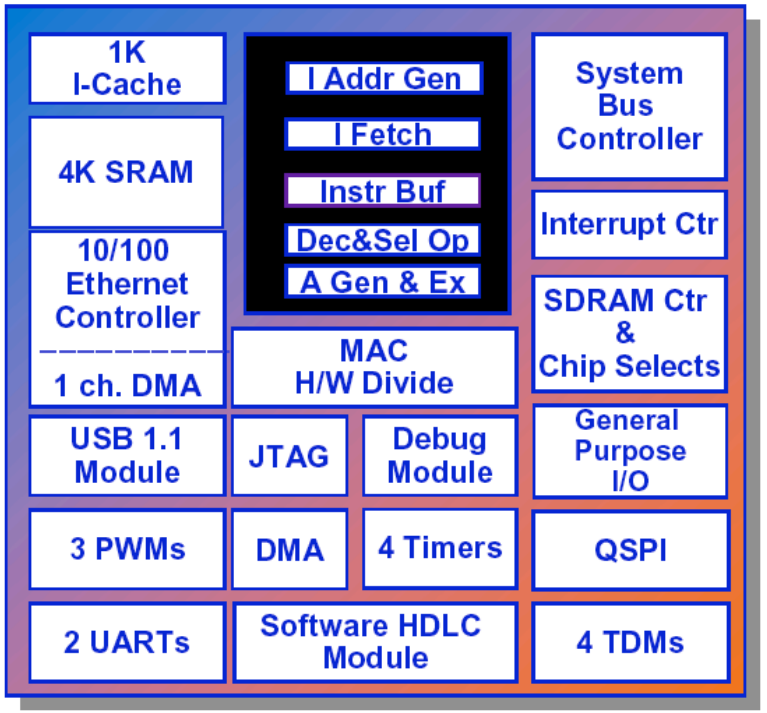
- Pro Befehl ein Maschinenzyklus
  - Vergleich VAX- MIPS



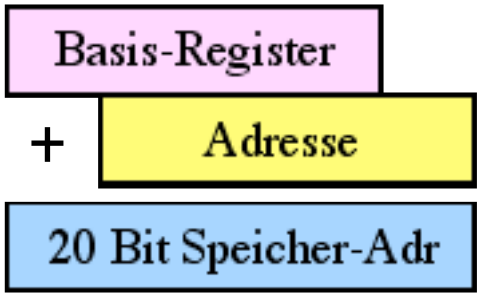
2.3 Stichwort: CISC

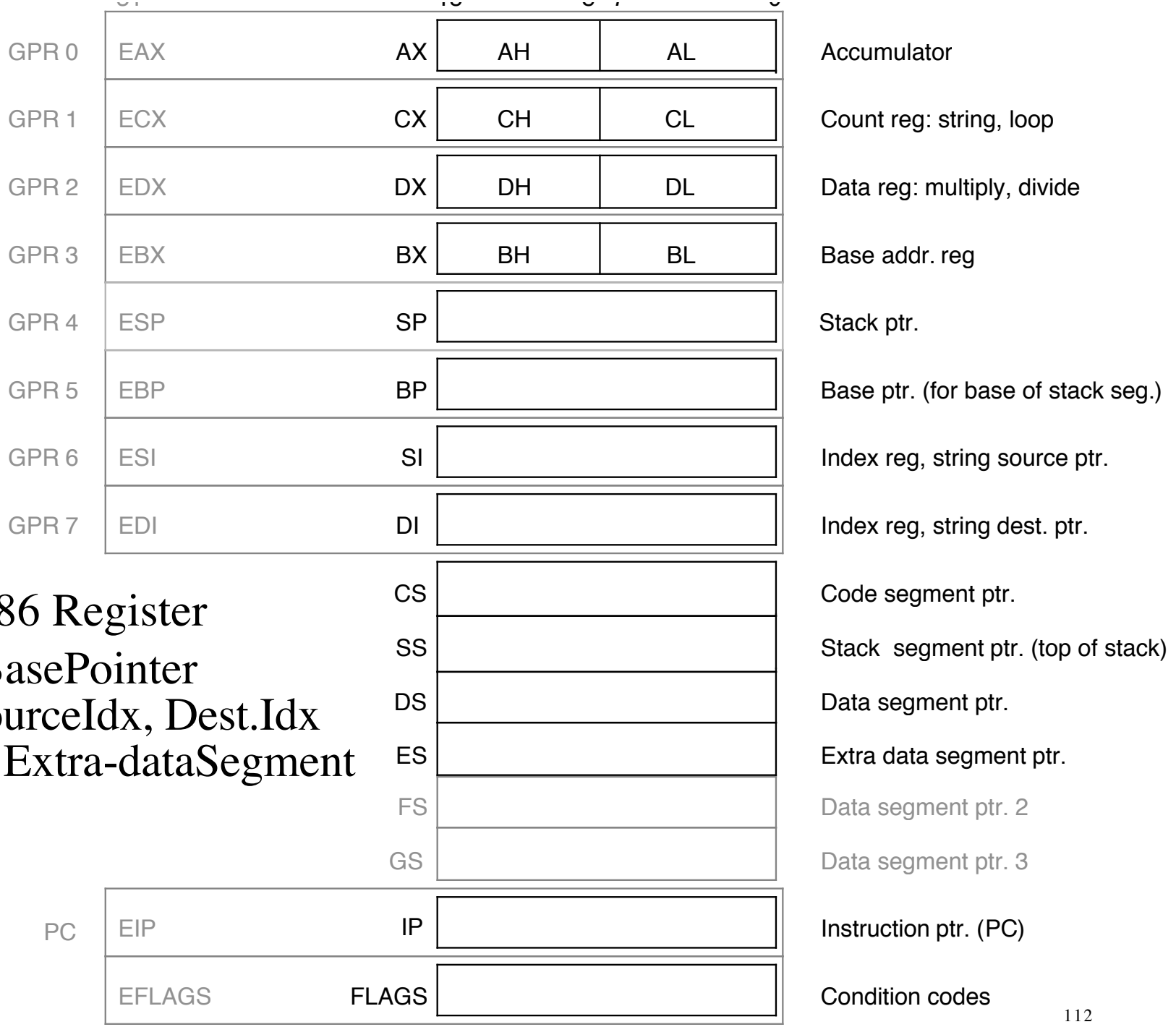
- Complex Instruktion Set Computer
  - individuelle Instruktionslänge: Speicher und Takte
  - evtl. dynamisch veränderliche Anzahl Takte
  - aufwendige Dekodierung
- Befehle mit viel Funktionalität
  - Anzahl Operanden
  - MISD-Instruktionen (Multiply-Accumulate, etc.)
- Mythos Programmiersprachen-Konstrukte
  - Schleifen => Schleifenbefehle (decrement & branch)
  - Funktionsaufrufbefehle mit Registerrettung
  - Case-Befehle
- Codedichte
  - viele Längen für Adress-Felder
  - Selektoren für Operandenzahl

- VAX - die ultimative CISC
  - 32 Bit Architektur, 32 Bit Bus
  - 16 Register, 12-15: AP, FP, SP, PC
  - leistungsstarke Befehle: CALL inkl. Register retten
  - INDEX-Instruktion: 45% schneller mit anderen VAX-Inst.
  - MicroVAX: 175 von 304 Instruktionen+Emulation
  - Nachfolger: Alpha
- Motorola 680x0 [1979]
  - orthogonaler Befehlssatz
  - 16 Register: A0-A7, D0-D7
  - 32 Bit Architektur, linearer Speicher
  - viele Adressmodi
  - 24 Bit externe Adressen
  - 68008: 8 Bit externer Daten-Bus
  - Multiplex von Adressen und Daten
  - 68020 [1984]: 3-stufige Pipeline,
    256 byte cache, 32 Bit extern
  - Coldfire ohne komplizierte 680x0 Instruktionen
  - Konfigurationen mit RAM, ROM und Interfaces



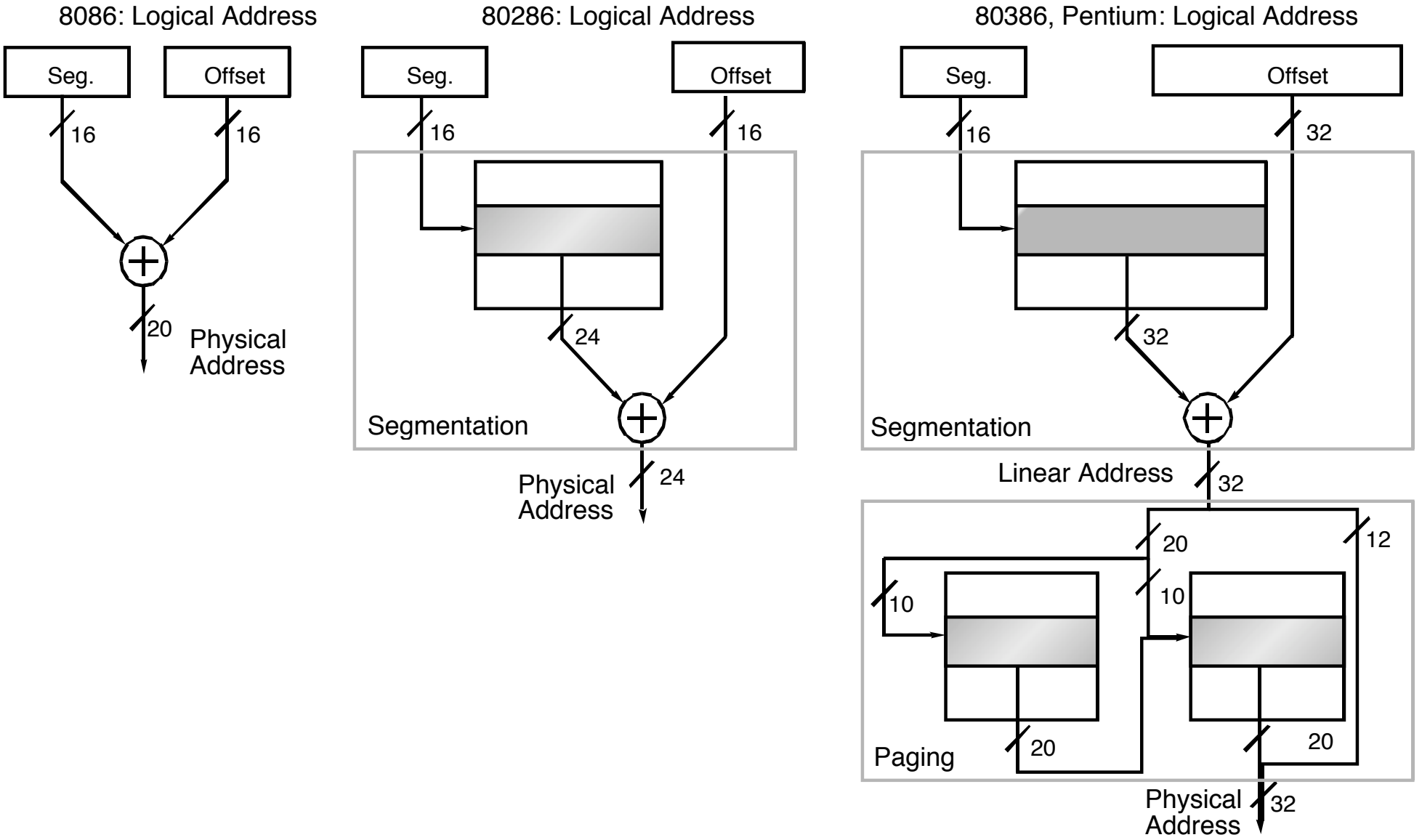
- IA: Intel Architecture
- Intel 8086
  - wenige Register: AX, BX, CX, DX, …
  - komplizierte Architektur
  - basiert auf 4004, 8008, 8080
  - segmentierter Speicher: 16 Bit Adressen + 16 Bit Basisregister
  - near/far Pointer und Jumps
- Evolution des 80x86
  - 8088, 8086, 80188, 80186, 80286
  - 80386DX, 80386SX
  - 80486DX, 80486DX2
  - Pentium
- Pentium Pro, PII, PIII, P4
  - RISC-Kern mit Übersetzung für 80x86-Befehle
  - Pipeline 14 und mehr Stufen
- 'Megahertz sells'
  - extreme Pipeline => extreme Taktzahlen
  - hohes Pipelinerisiko => EPIC
  - Befehle evtl. mehrere Takte lang



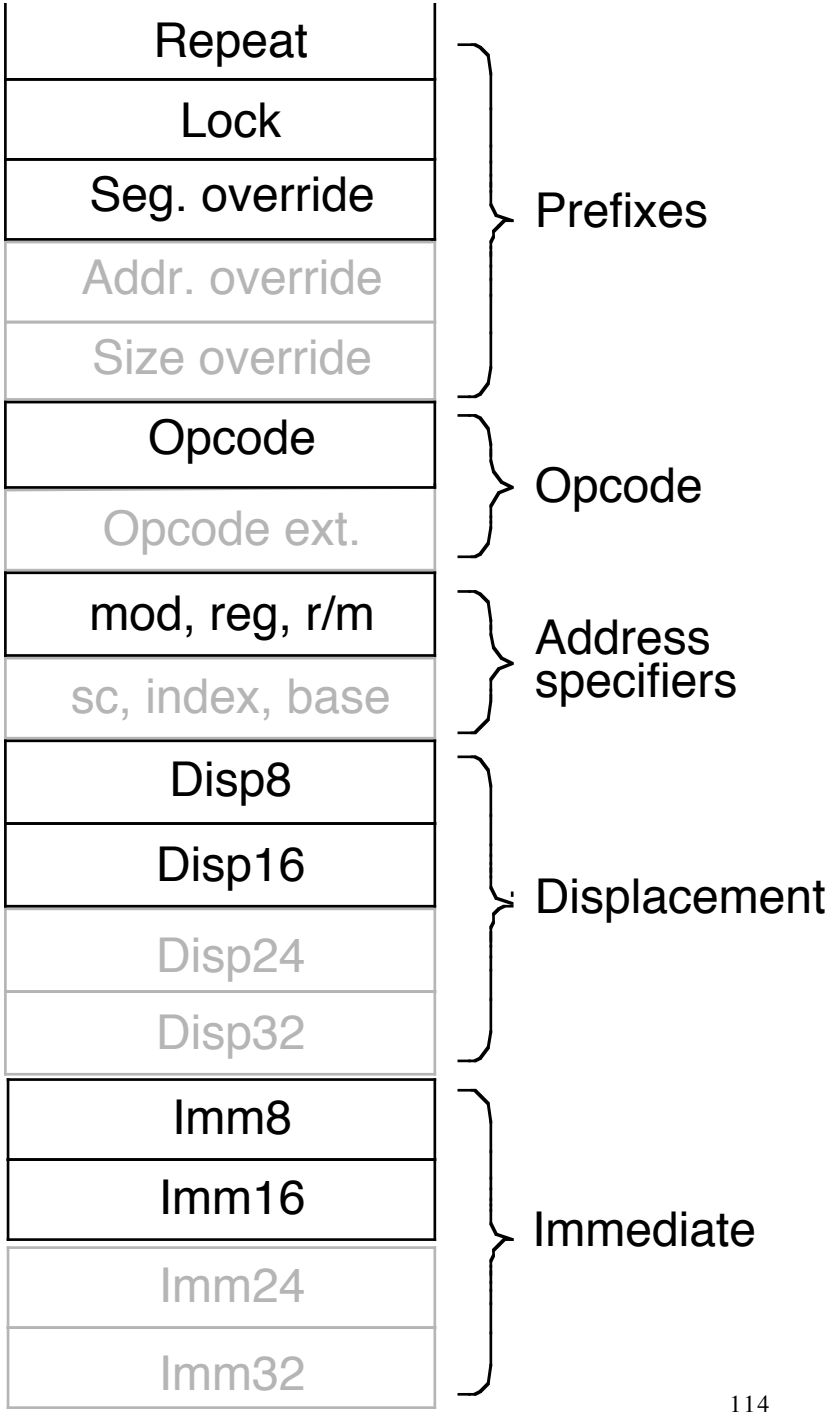


- Besondere 80x86 Register
- StackPointer, BasePointer
  - Stringops SourceIdx, Dest.Idx
  - CS, DS, SS, Extra-dataSegment

• 80x86 Adressgenerierung



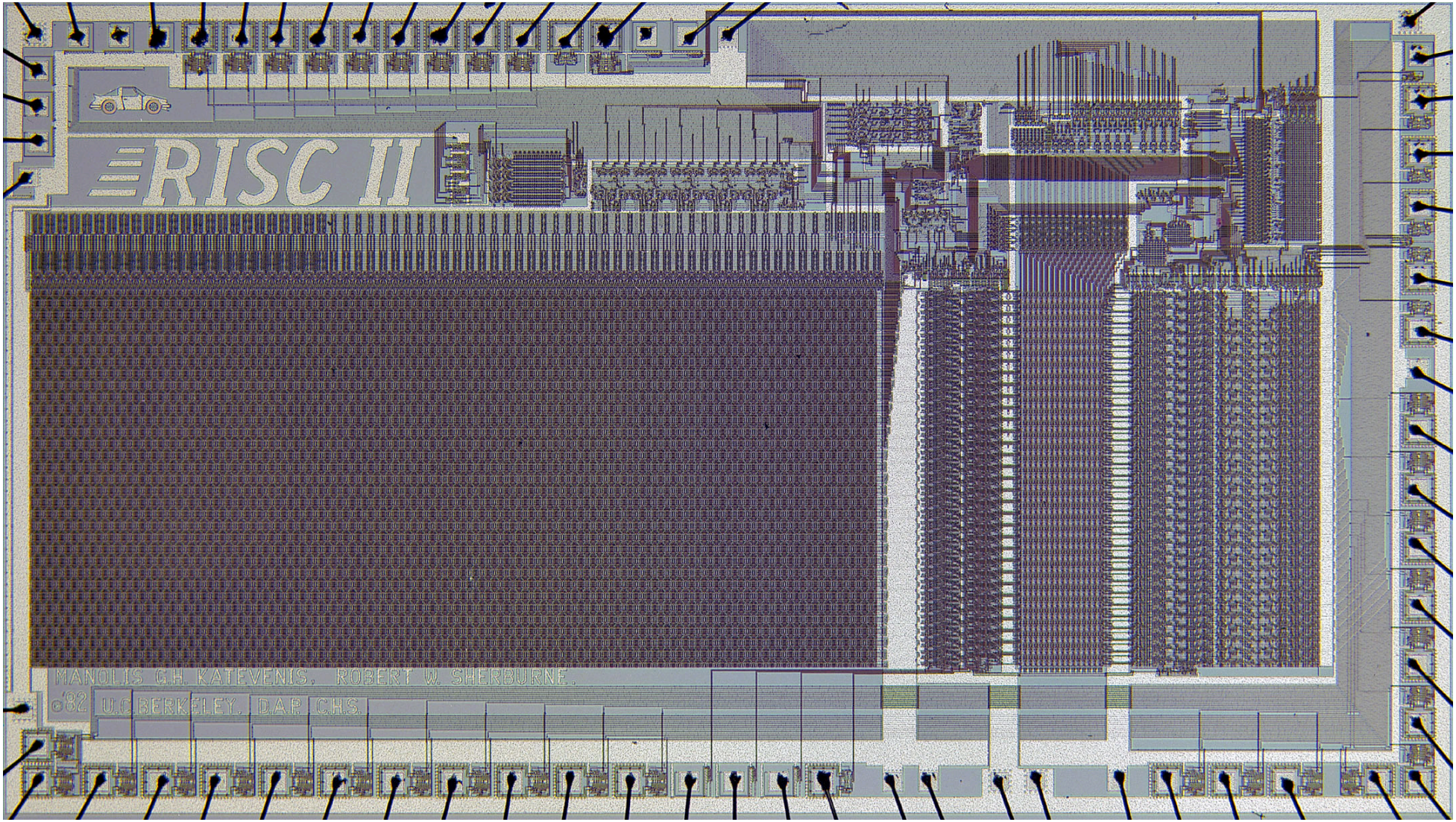
- Sprünge near und far
  - JMP, JMPF, JNZ, JZ
  - CALL, CALLF
  - RET, RETF
  - LOOP: CX--, JNZ 8 Bit Displacement
- Datentransfer
  - MOV reg-reg, reg-mem
  - PUSH, POP
- Arithmetik
  - ADD; SUB, CMP, …
  - SHx, RCR, INC, DEC, …
- Stringoperationen
  - MOVS
  - LODS
- Floating Point
  - Stackbasiert
  - 8 Stack-'Register'



2.4 Stichwort: RISC

- IBM 801
- A Case for the Reduced Instruktion Set Computer
  - David Patterson, UCB, 1980
  - pro Takt ein Befehl
  - Kompromisse bei den Befehlen
  - Load-Store Architektur
- Make the general case fast
  - Instruktionssatz-Optimierung
  - ein Befehl - ein Zyklus (≠Takt)
  - Pipeline
  - Branch-Delay-Slot(s) (siehe Pipeline)
- Einfache Konstruktion
  - reguläre Struktur
  - Registerbänke, Cache, ALUs, Multiplexer
  - wenig Verwaltung
  - leichte Migration auf neue Verfahren

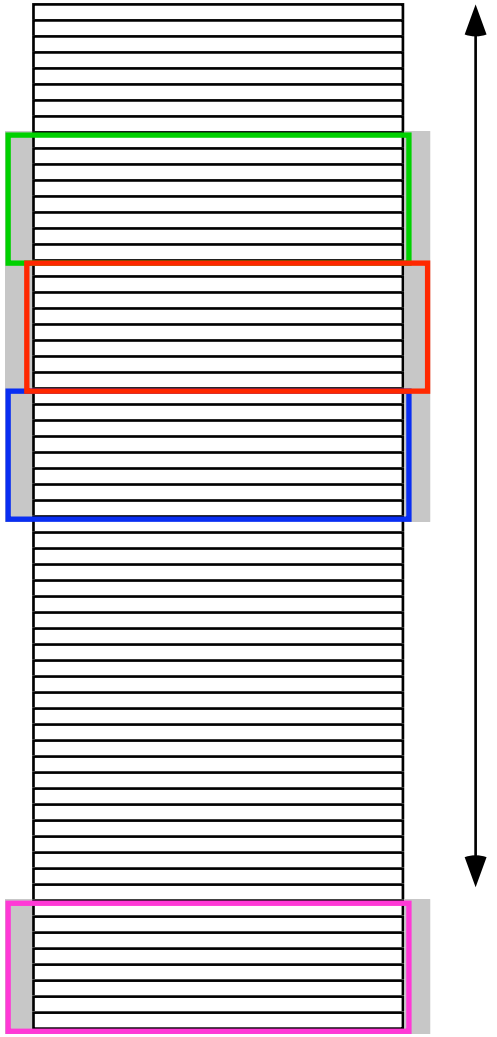
- RISC I/II [1982, 1983, Patterson et al, UCB]
  - 40,760 Transistoren, 3 micron NMOS, 60 mm^2, 3 MHz
  - Basis für Sun's SPARC



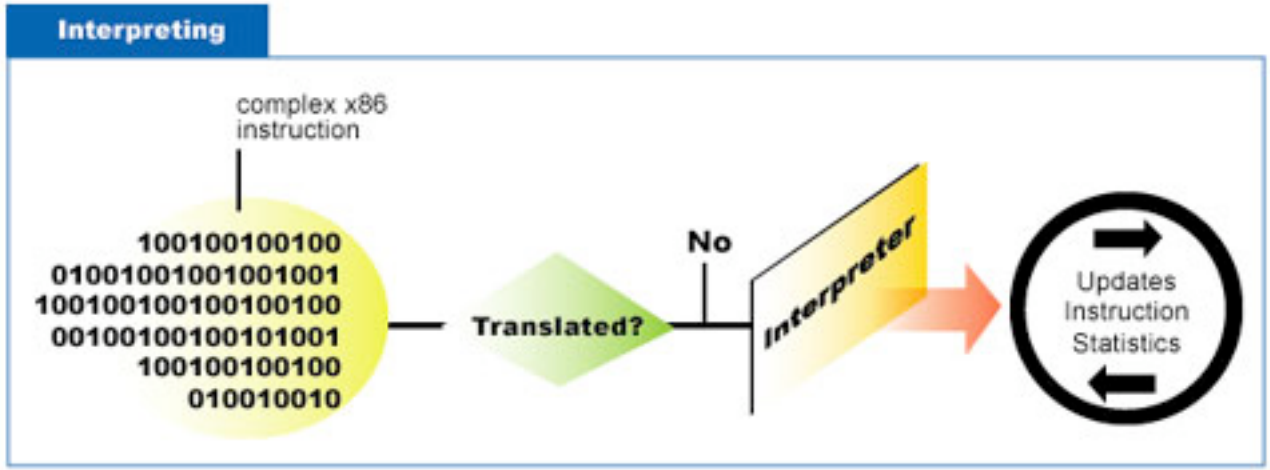
- MIPS [John Hennessy]
  - Universitätsprojekt, Stanford U
  - Microprocessor without Interlocked Pipeline Stages
  - später Mips Inc, dann SGI
  - R2000, …
  - SGI, Nintendo 64, Playstation, PS2
  - Drucker, Cisco-Router : http://www.mips.com/coolApps/s3p3.html
- MIPS Kennzahlen
  - 32 Bit: Adresseraum, Register, Instruktionen
  - 31 GPRs und R0
  - 32*32 FPRegs (oder 16*64)
- MIPS IV Vergleich zu DLX
  - Indizierte Adressierung
  - Instruktionen für nicht-alignierte Daten
  - Multiply-Add
  - Unteilbares SWAP
  - 64 Bit Datentransfer und Arithmetik

- PowerPC
  - Motorola, IBM, Apple
  - Ideen der 801: Power 1, Power 2
  - PPC 601 (G1), 603 (G2), 604, 620 (64 Bit, MP)
  - G3, G4 < 1 GHz; G4 mit Vektoreinheit AltiVec
  - G5 > 1GHz
  - IBM-Gekko: Nintendo GameCube (Dolphin)
- PowerPC Kennzahlen
  - 32 Bit: Adresseraum, Register, Instruktionen
  - 32 GPRs
  - 32*32 FPRegs (oder 32*64)
- PowerPC Vergleich zu DLX
  - Fast alle Adressierungsarten
  - nicht-alignierte Daten möglich
  - Multiply-Add
  - Unteilbares SWAP
  - 8*4 Bit Condition-Codes (ähnlich Registerkonzept)
  - 64 Bit Instruktionen für 64 Bit Chips

- HP-Precision Architecture
  - erste RISC mit SIMD Multimediaerweiterungen
  - ShiftAdd statt Multiplikation, DS für Division
  - viele Conditional Branches
  - 32/48 Bit Adressen mit Segmentierung
- SPARC
  - max. 32*16 Register, klassisch 128
  - Register Windows für Prozeduren
  - Window-Größe 8+8+8
  - 8 für lokale Variable
  - 16 für Parameter: 8 In, 8 Out
  - 8 für globale Variablen
  - typisch 8 Register für Variablen und Parameter
  - Save/Restore um Registerfenster zu verschieben
  - V8: Tagged-Instruktionen für LISP und Smalltalk

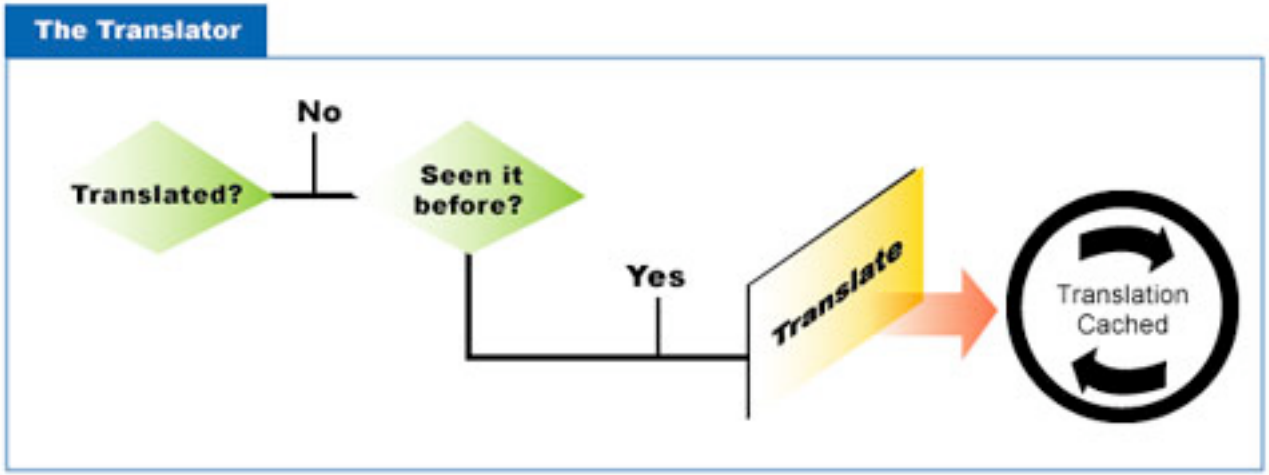


- Transmeta
- Apples Emulatorprojekt
  - Interpretation der 68LC040-Befehle
  - 40 Mhz 68040 auf 66 MHz PPC601
  - Übersetzung häufiger Instruktionssequenzen
- IBM-Projekt zur Emulation des 80386

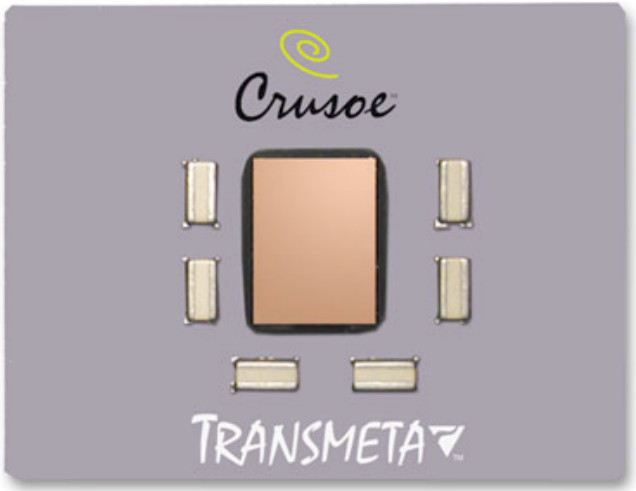


- VLIW: Very Large Instruktion Word
  - mehrere Instruktionen pro Instruktion Fetch
  - Unabhängigkeit?

- Häufig ausgeführte Sequenzen besonders behandeln
  - Instruktions-Statistik
  - Code Profiling => Optimierung
  - dynamischer Übersetzer (code morphing)
  - übersetzte Blöcke cachen



- Crusoe
  - 64 Register a 32 Bit
  - 4 Funktionseinheiten zur parallelen Befehlsausführung
  - Java Mode



- Intel-Code

```
addl  %eax,(%esp) // load data from stack, add to %eax
addl  %ebx,(%esp) // ditto, for %ebx
movl  %esi,(%ebp) // load %esi from memory
subl  %ecx,5      // subtract 5 from %ecx register
```

- 1. Übersetzungslauf

```
ld    %r30,[%esp]    // load from stack, into temporary
add.c %eax,%eax,%r30 // add to %eax, set condition codes.
ld    %r31,[%esp]
add.c %ebx,%ebx,%r31
ld    %esi,[%ebp]
sub.c %ecx,%ecx,5
```

- 2. Übersetzungslauf (klassische Compilertechnik)
  - gemeinsame Teilausdrücke
  - Schleifeninvarianten
  - unbenutzer Code

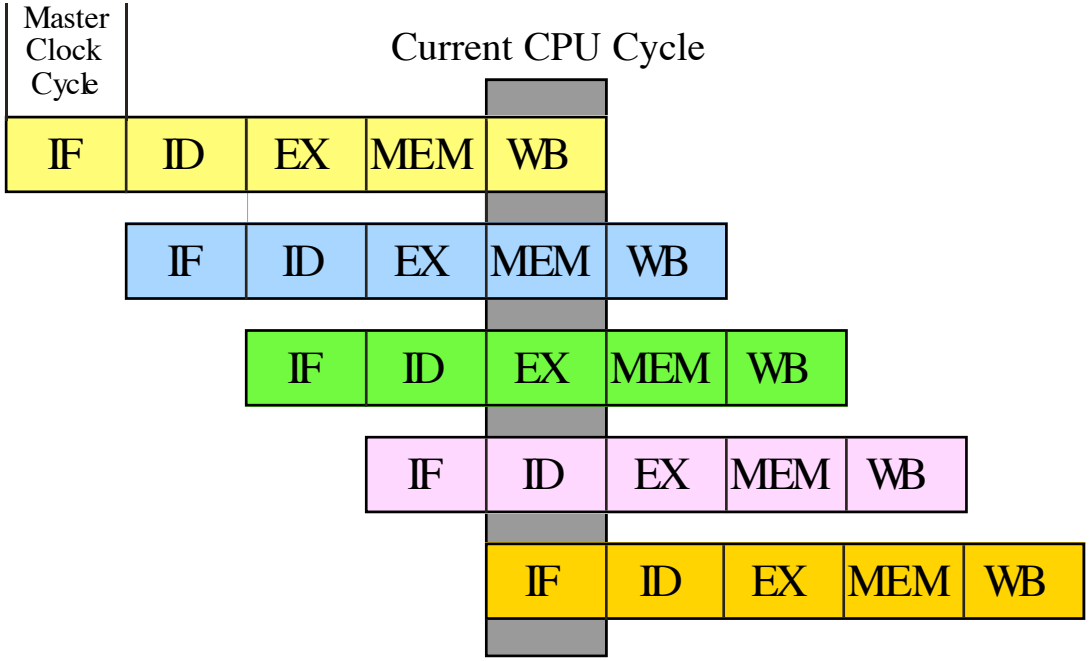
```
ld    %r30,[%esp]    // load from stack only once
add   %eax,%eax,%r30
add   %ebx,%ebx,%r30 // reuse data loaded earlier
ld    %esi,[%ebp]
sub.c %ecx,%ecx,5    // only this last condition code needed
```

- 3. Übersetzungslauf
  - Befehle umordnen
  - Befehle auf parallele Funktionseinheiten verteilen
  - Software

```
1. ld %r30,[%esp];   sub.c %ecx,%ecx,5
2. ld %esi,[%ebp];   add %eax,%eax,%r30;  add %ebx,%ebx,%r30
```

3. Pipelining

- Fließbandarbeit
  - Instruktionen haben Teilschritte
  - parallele Ausführung der Teilschritte verschiedener Instruktionen
  - Weitergabe der Zwischenergebniss an nächste Stufe (≠MIMA)

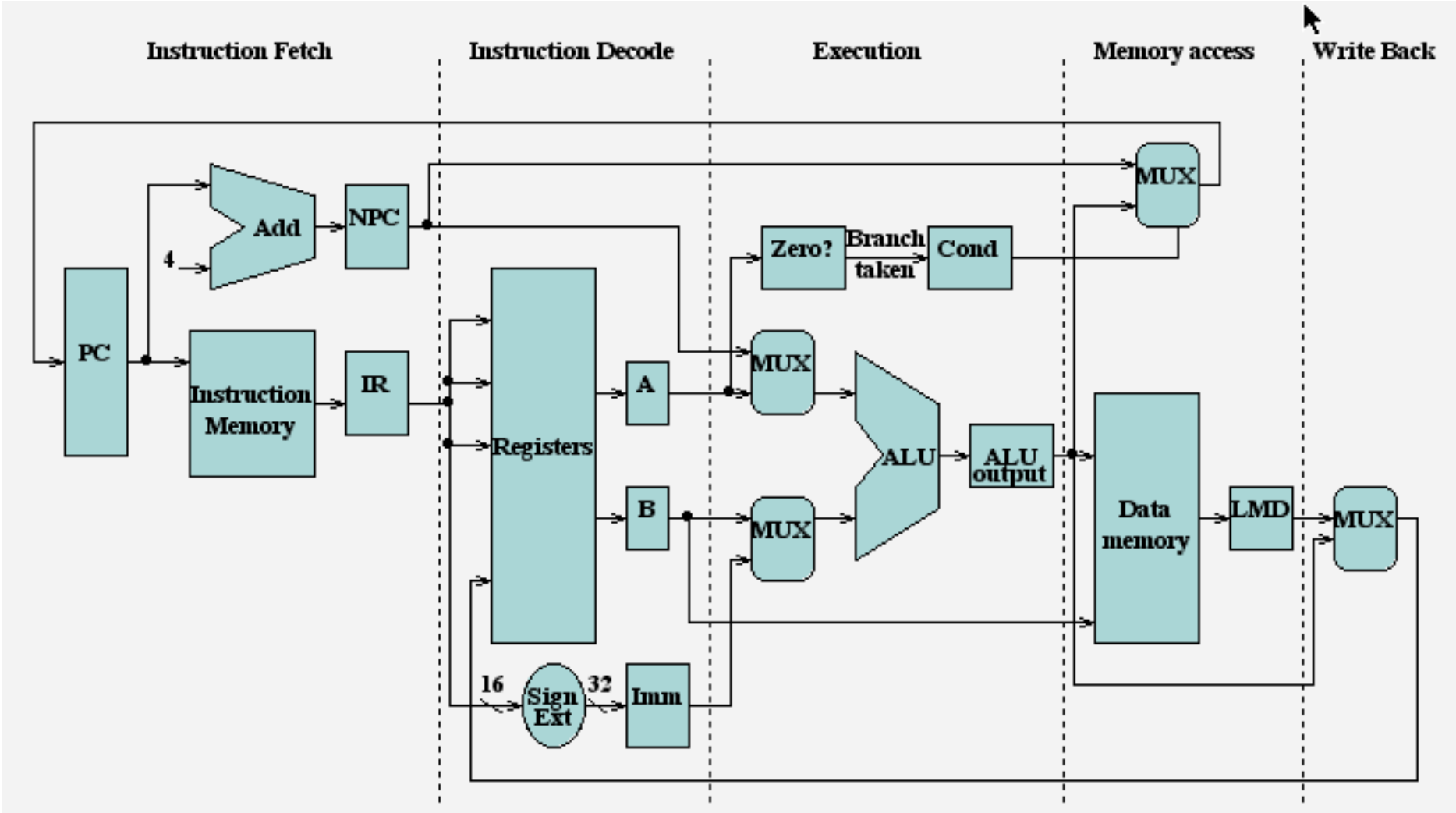


- 1 Instruktion
  - n Maschinen-Zyklen insgesamt
  - pro Maschinen-Zyklus 1 Instruktion fertig
  - m Takte = 1 Maschinen-Zyklen; m<=4

- Unabhängigkeit der Einzelschritte in der Pipeline
  - Entkopplung der Stufen
  - genug Verarbeitungseinheiten (ALU + 1 Addierer für Adressen)
- Pipelinetakt
  - bestimmt durch längsten Teilschritt
  - z.B. Speicherzugriff oder Alu-Operation
  - Länge der Einzelschritte ausbalancieren
- Pipeline-Register entkoppeln Stufen
  - Pipeline-Latches
  - Stufe n-1 schreibt Resultat spät im Zyklus
  - Stufe n nimmt Input früh im Zyklus
- Speicher
  - Resultate in einem Zyklus
  - getrennte Speicher/Caches für Instruktion und Daten?
- Pipeline-Risiken
  - Verzweigung, Sprung
  - Operandenbezüge
  - Resource-Mangel (z.B. ALU, Registerbank)

  => Pipeline-Stall

- DLX-Pipeline



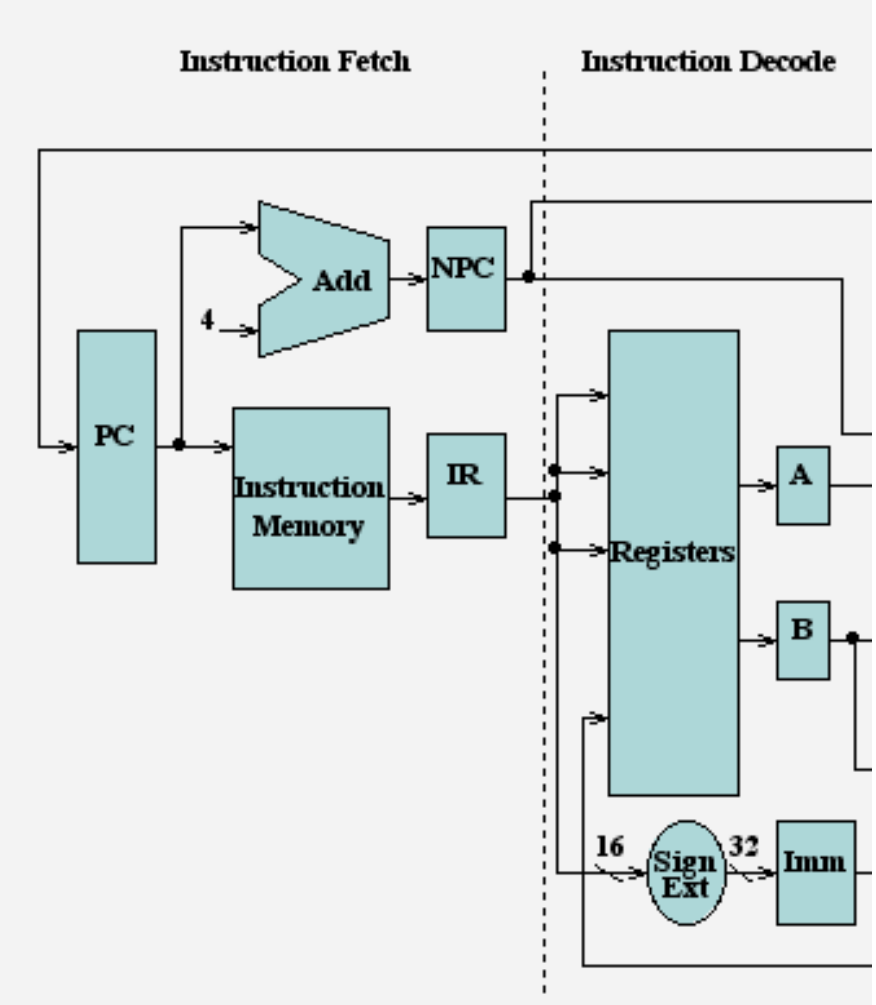
- Steuerung sorgt für unabhängige Ausführung
  - 'Durchreichen' des Opcodes und der Operanden

3.1 Instruktionen bearbeiten und ausführen

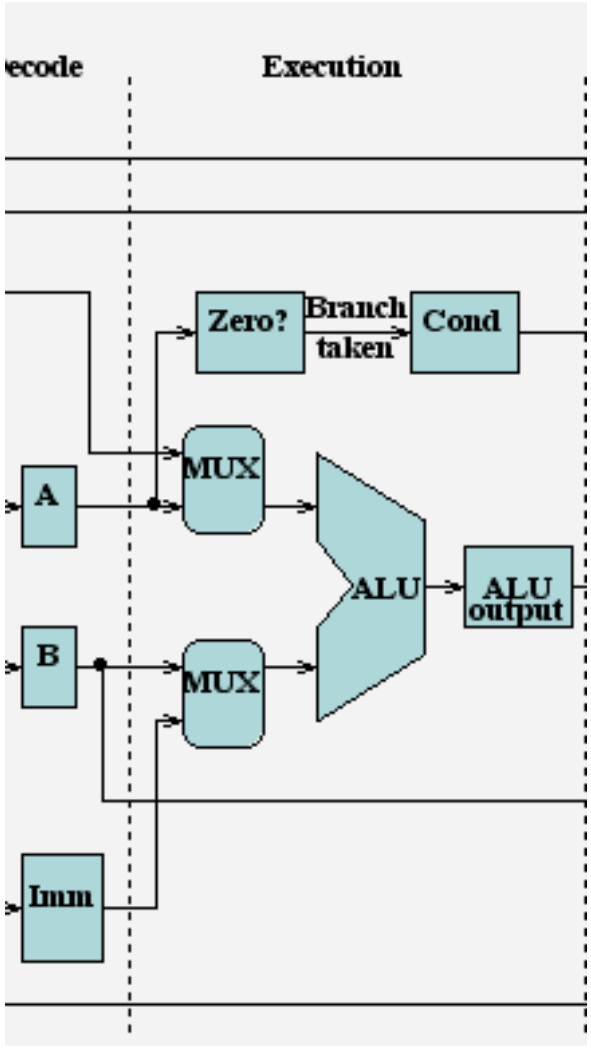
- Schönes Java-Applet

  http://www.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/DLXimplem.html

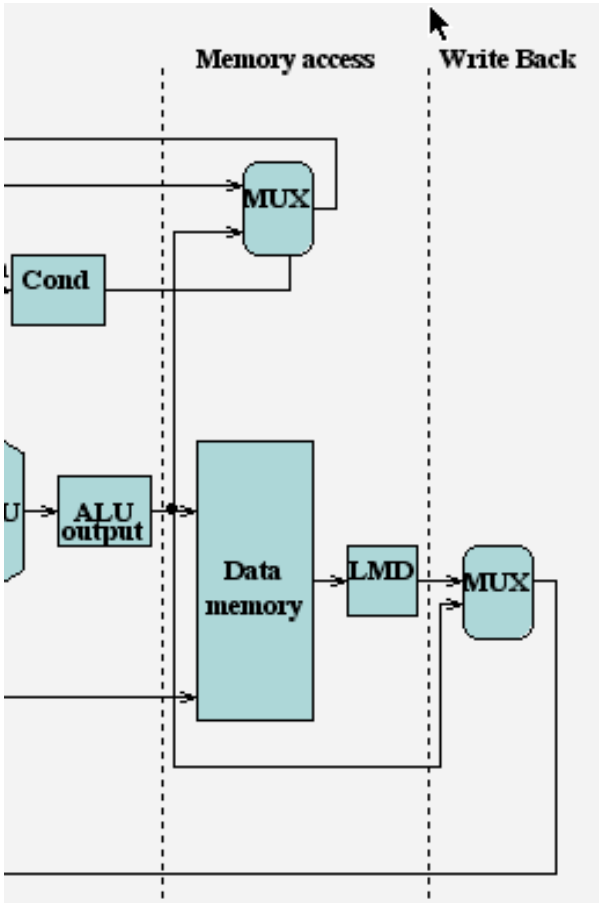
- Instruction fetch cycle (**IF**)
  - `IR := Mem[PC]`
  - `NewPC := PC+4`
- Instruction decode/register fetch (**ID**)
  - Dekodierung: welcher Befehl
  - Operanden aus der Register-Bank in die Eingangsregister der ALU
  - $A := Regs[IR_{6..10}]$
  - $B := Regs[IR_{11..15}]$
  - $Imm := (IR_{16})^{16} \#\# IR_{16..31}$
  - Festes Instruktionsformat: Parallelität



- Execution/Effective address cycle (**EX**)
  - Berechnung
  - Ausgangsregister der ALU setzen (+ CC)
  - Load/Store: EA im ALU-Output-Reg
  - `ALUOut := A + Imm`
  - Register-Register: Resultat der Operation
  - `ALUOut := A` *`func`* `B`
  - Register-Immediate: Resultat der Operation
  - `ALUOut := A` *`func`* `Imm`
  - Control-Xfer: ALU berechnet Sprungziel
  - `ALUOut := NPC + Imm`
  - `Cond := A` *`op`* `0`



- Memory access/branch completion cycle (**MEM**)
  - nur bei Load/Store ausgeführt
  - Load: Daten von MEM[ALU-Output] holen
  - `LMD := MEM[ALUOut]`
  - Store: Daten nach MEM[ALU-Output] schreiben
  - `MEM[ALUOut]:= B`
  - Control-Xfer: bei ausgeführten Sprüngen PC schreiben
  - `if Cond then PC:=ALUOut`
    `else PC:=NPC`
- Write-back cycle (**WB**)
  - leer falls Control-Xfer
  - Resultat in die Registerbank übertragen
  - Load
  - $\texttt{Regs[IR}_{11..15}\texttt{]} := \texttt{LMD}$
  - Register-Immediate
  - $\texttt{Regs[IR}_{11..15}\texttt{]} := \texttt{ALUOut}$
  - Register-Register
  - $\texttt{Regs[IR}_{16..20}\texttt{]} := \texttt{ALUOut}$



3.2 Pipeline Hazards

- Konflikte in Architektur oder Abhängigkeiten
- Cache-Miss
  - Pipeline anhalten
  - Warten bis Daten angekommen sind
- Hazard
  - konkurrierender Zugriff auf Ressourcen oder Daten
  - keine neuen Instruktionen starten
- Pipeline-Bubble (oder Stall)
  - Beispiel nur ein Instruktions/Datencache

	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				
Instr i+2			IF	ID	EX	MEM	WB			
Stall				bubble	bubble	bubble	bubble	bubble		
Instr i+3					IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

- Alternative Darstellung

Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				
Instr i+2			IF	ID	EX	MEM	WB			
Instr i+3				stall	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

- Anderer Fall

Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	stall	EX	MEM	WB			
Instr i+2			IF	stall	ID	EX	MEM	WB		
Instr i+3				stall	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

3.2.1 Structural Hazards

- Nur ein Schreibeingang in das Register-file
  - 2 Schreibzugriffe in einem Zyklus
- Nur ein Cache (Bus) für Daten und Befehle
  - Load.MEM kollidiert mit Inst3.IF

Instr	1	2	3	4	5	6	7	8
Load	IF	ID	EX	MEM	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				IF	ID	EX	MEM	WB

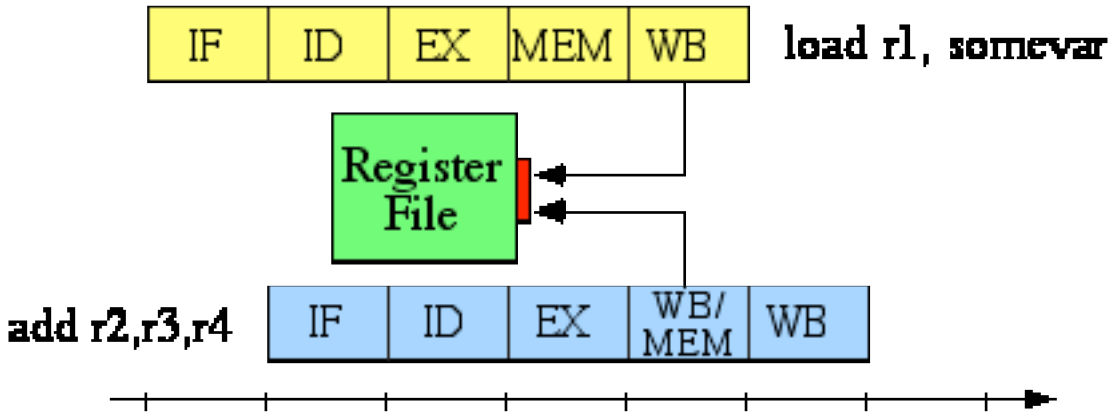
  - Inst3.IF verschieben

Instr	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	**MEM**	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Stall				bubble	bubble	bubble	bubble	bubble	
Instr 3					**IF**	ID	EX	MEM	WB

- Annahme:
  - MEM-Stufe kann ALU-Resultat ins Register-File schreiben
  - nur ein Write-Port

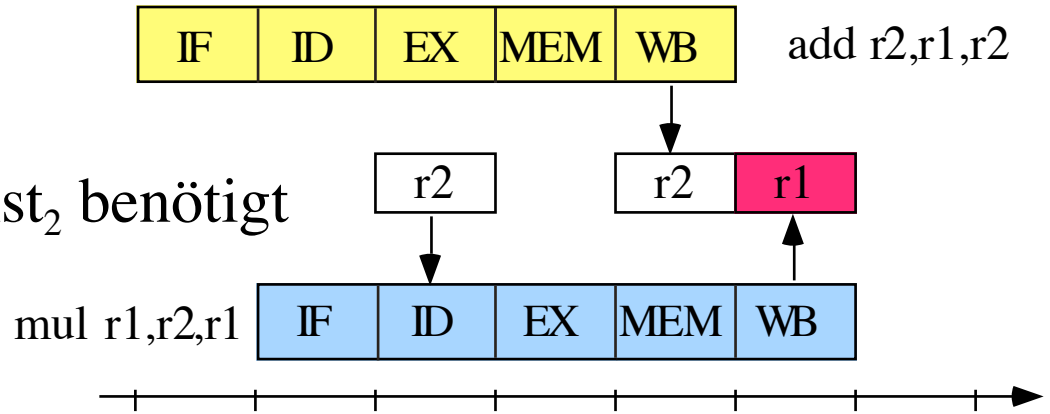
```
load    r1, somevar
add     r2,r3,r4
```

  - Daten für r1 und r2 kommen gleichzeitig am RegisterFile an



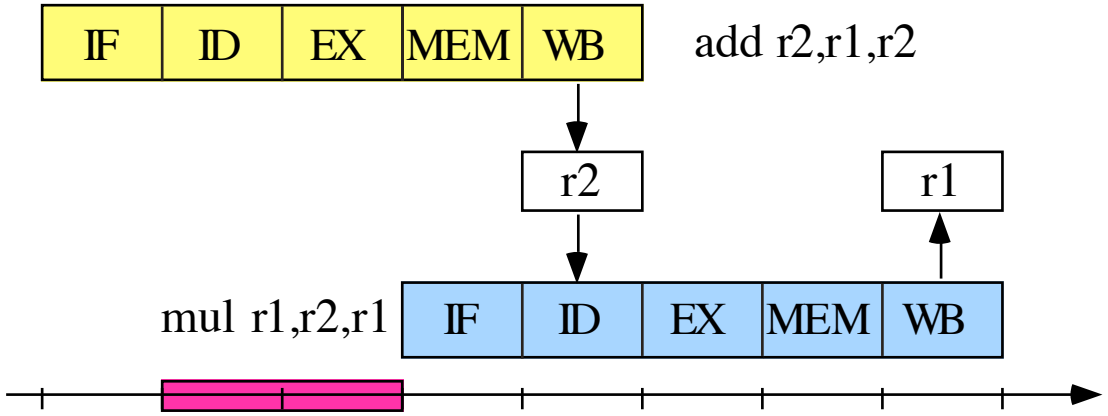
- Hardware entdeckt Konflikt und fügt bubble ein
- Resource Replication: 2 write ports
  - weitere Abhängigkeiten erzeugt?
  - weitere Überwachung und Interlocking

3.2.2 Data Hazards

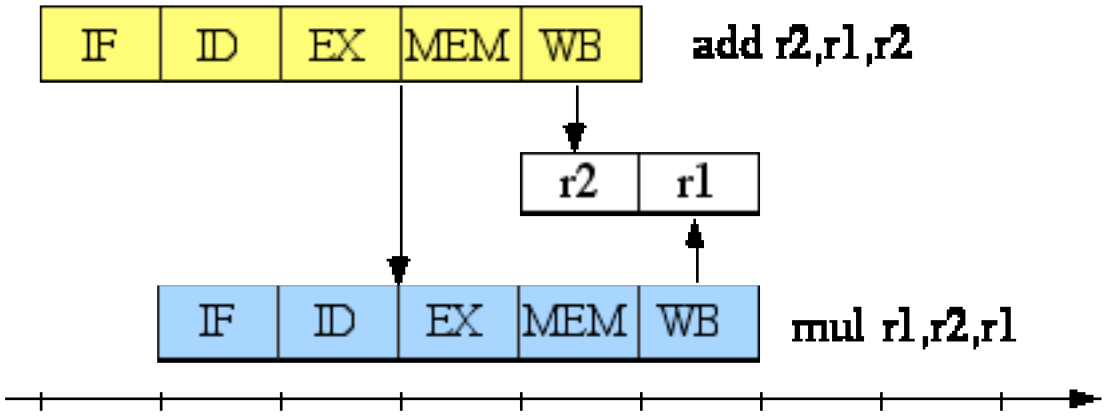


- Echte Abhängigkeit
  - $Inst_1$ schreibt in das Register, das $Inst_2$ benötigt
  - auch im Speicher!
- Read After Write (RAW)
  - $Instr_2$ liest Operand, bevor $Instr_1$ ihn schreibt
- Write After Read (WAR)
  - $Instr_2$ schreibt Operand, bevor $Inst_1$ ihn liest
  - nur bei superskalaren Architekturen
- Write After Write (WAW)
  - $Instr_2$ schreibt Operand, bevor $Inst_1$ ihn schreibt
  - in komplexeren Pipelines als DLX
- Namens-Abhängigkeit
  - anti-abhängig: $Inst_1$ liest Register, das $Inst_2$ später überschreibt
  - Ausgabe-abhängig: $Inst_1$ schreibt Reg., das $Inst_2$ danach überschreibt
  - keine Daten von $Inst_1$ and $Inst_2$ übergeben
  - Implementierungsproblem

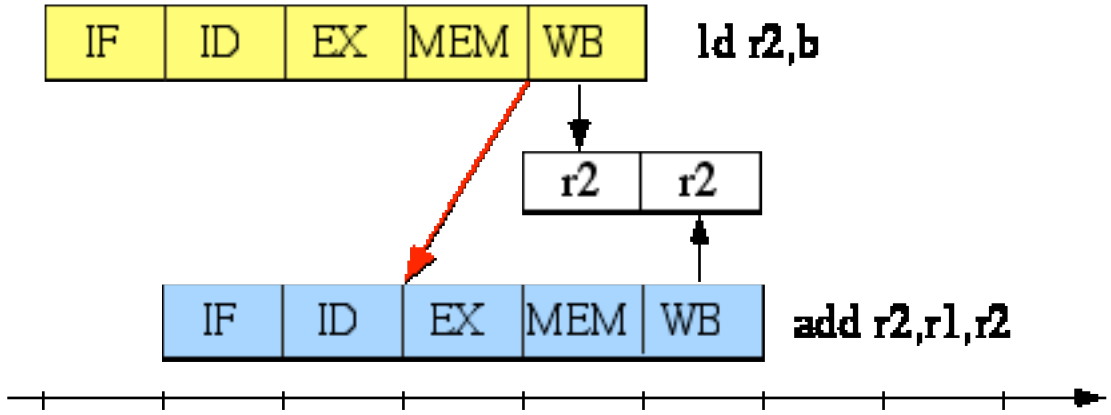
- Lösungen
- Compiler erzeugt geeigneten Code
  - NOP(s) einfügen (Software bubble)
  - Instruktionen umordnen
- Hardware
  - Schaltung um Konflikt zu erkennen
  - Pipelinesteuerung
- Interlocking
  - Pipeline anhalten (stall)
  - evtl. mehrere Zyklen



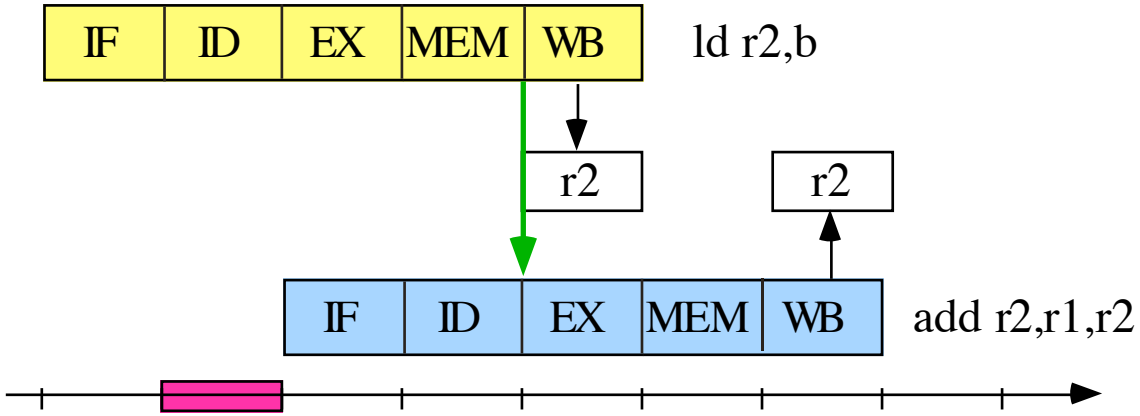
- Forwarding
  - Abkürzungen in der Pipeline
  - ALU-Resultat von $Inst_{EX}$ direkt an ALU-Input für $Inst_{ID}$ kopieren



  - LMDR von $Inst_{MEM}$ auch an ALU-Input für $Inst_{ID}$ kopieren
- Forwarding nicht immer möglich



- Forwarding mit Interlocking
  - $Inst_2$ datenabhängig (RAW) von $Inst_1$



  - bubble kann nicht ganz aufgelöst werden
- Scoreboard entdeckt Abhängigkeiten

3.2.3 Control Hazards

- Sprung-Instruktionen (bedingt und fest)
  - Spungziel muss berechnet werden
  - Bubbles einfügen bis Sprung MEM ausgeführt hat
  - Sprung erst in ID festgestellt
  - IF während Sprung-ID oft wertlos

IF	ID:Sprung	EX	MEM	WB					
	IF->stall	stall	stall	IF	ID	EX	MEM	WB	
					ID	EX	EX	MEM	WB

- 3 Zyklen pro Spung verloren
  - Verzweigungen ca 20%

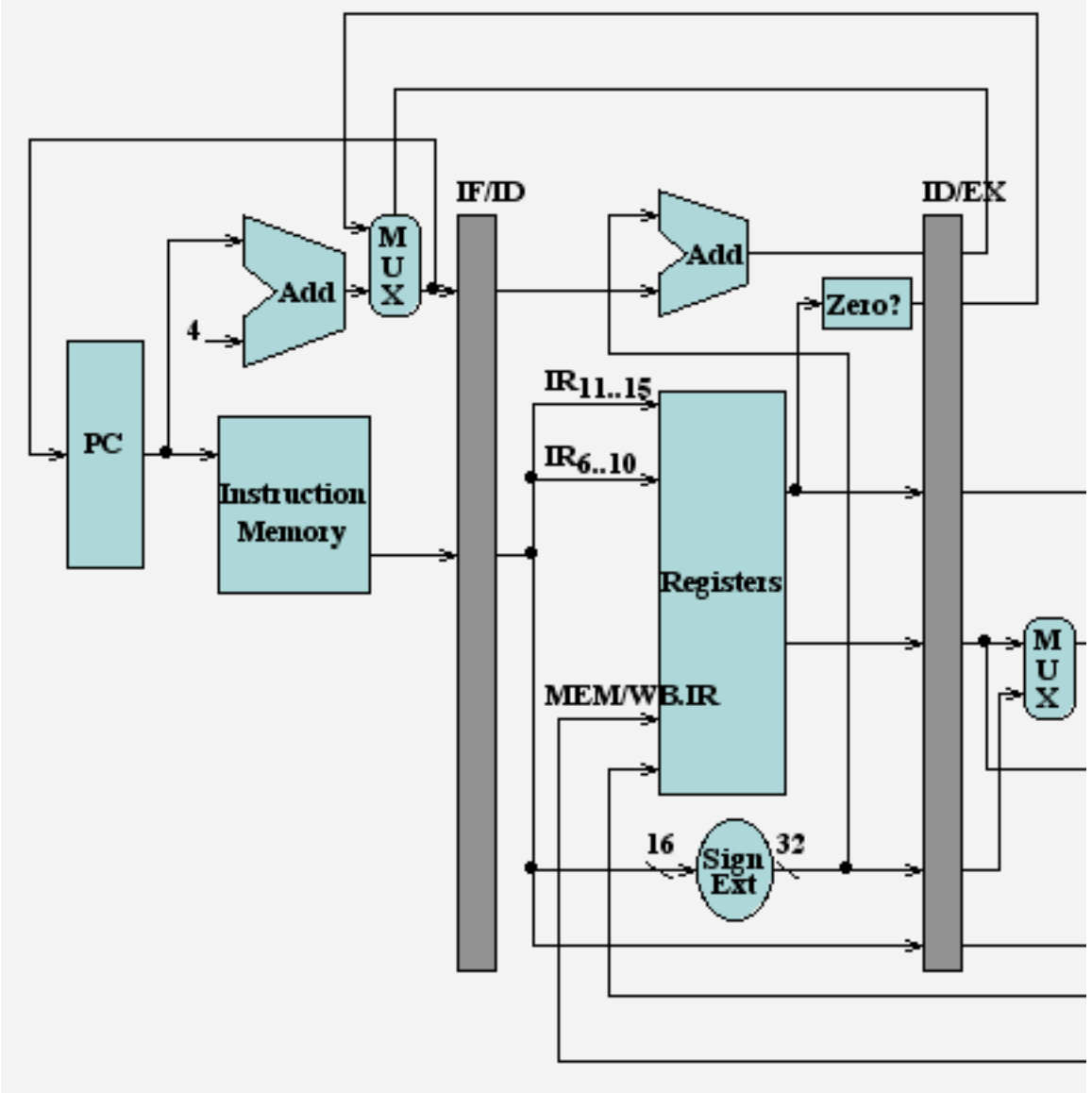
  => 1,6 CPI (cycles per instruction)
- Abhilfe
  - schon in IF feststellen ob gesprungen wird
  - neuen PC früher berechnen

- Pipeline umbauen
  - Addierer für Branch in ID
  - Zugriff auf Reg früh
  - Addieren spät
  - Condition auswerten spät
  - nur ein Stall

IF	ID	EX
	IF->stall	IF

- Bedingte Sprünge
  - einfacher Test auf 0
  - einfach ausführbar
  - Data hazard?

```
SUBI   r3,r3,1
BNEZ   r3,loop1
```



- Wie oft werden Sprünge ausgeführt?
  - 17% bedingte Sprünge
  - 85% der Rückwärts-Sprünge ausgeführt (Schleifen)
  - 60% der Vorwärts-Sprünge ausgeführt (if/then/else)
  - 67% aller Sprünge werden ausgeführt
- Triviale Vorhersage: Sprung wird nicht genommen
  - predict-not-taken
  - einfach weitermachen, als ob nicht gesprungen wird
  - permanenten Zustand (Register!) nicht verändern
  - falls doch gesprungen wird: IF/ID latches löschen
  - immerhin 33% Gewinn

IF	ID:Sprung	EX	MEM	WB					
	IF	ID/leer	EX/leer	MEM/leer	WB/leer				
		IF/IF	ID	EX	MEM	WB			
			IF	ID	EX	MEM	WB		
				IF	ID	EX	MEM	WB	
					ID	EX	EX	MEM	WB

3.2.4 Exceptions

- I/O, OS-Call, …
- Fehler, Page-Fault, …
- Breakpoints
- Beispiel: Page Fault bemerkt in der MEM-Stufe
  - Betriebssystem tauscht Seite
  - Wiederaufnahme
  - was passiert mit der Instruktion in der WB-Stufe?
- Unterbrechung in DLX
  - TRAP-Instruktion in IF/ID latches schreiben
  - Schreiboperationen der (Folge-)Instruktion unterbinden
  - Unterbrechung rettet PC
  - PC restaurieren mit RFE
- Problem: Delay Slot

3.3.1 Softwarelösungen

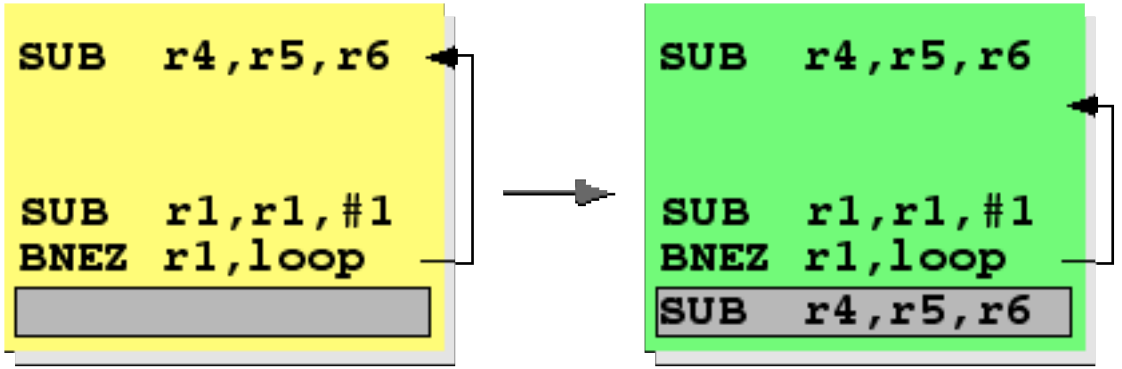
3.3.1.1 Branch Delay Slot

- m Instruktionen nach dem Branch werden immer ausgeführt
  - kann der Compiler oft sinnvoll füllen
  - Programmierer finden fast immer etwas

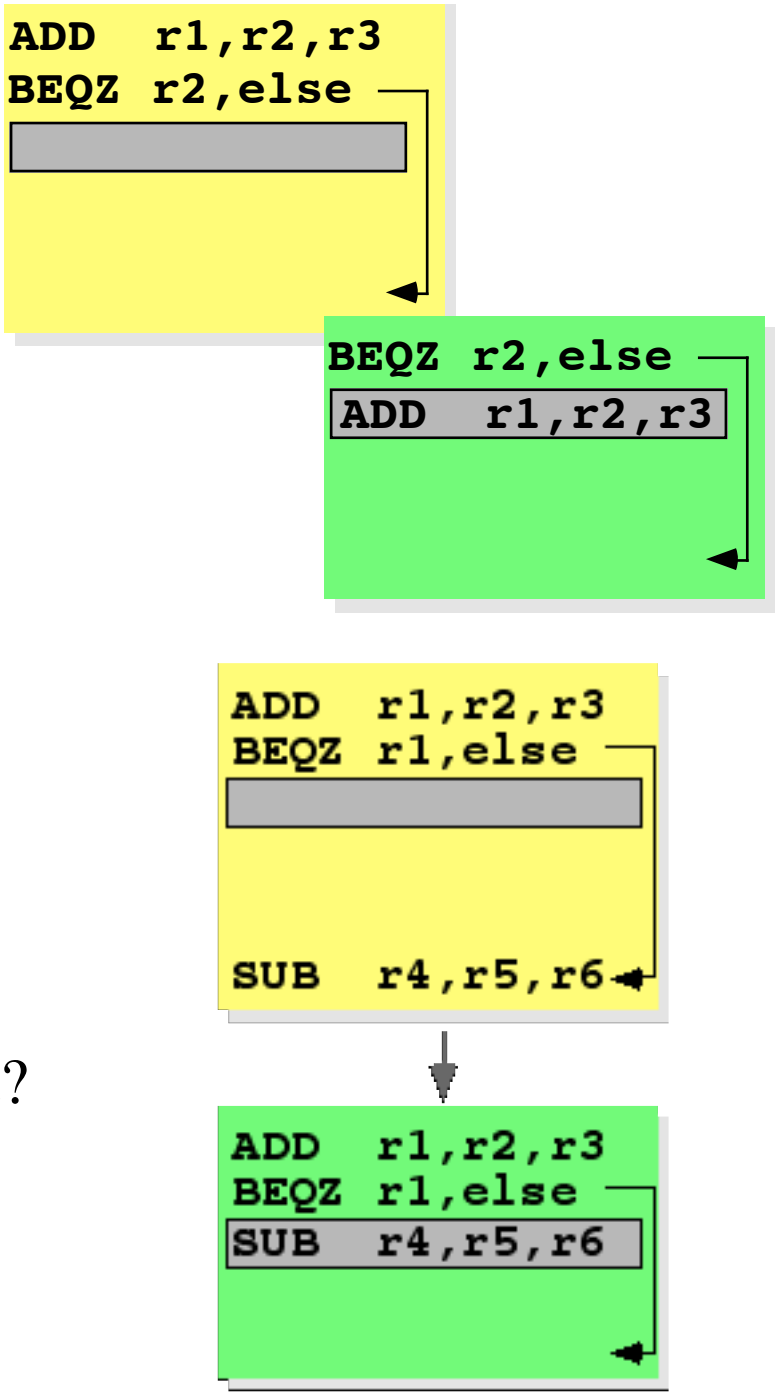
Inst i	IF	ID	EX	MEM	WB				
Inst i+1 (delay-slot)		IF	ID	EX	MEM	WB			
Inst i+2			IF	ID	EX	MEM	WB		
Inst i+3				IF	ID	EX	MEM	WB	
Inst i+4					IF	ID	EX	MEM	WB

Inst i	IF	ID	EX	MEM	WB				
Inst i+1 (delay-slot)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target+1				IF	ID	EX	MEM	WB	
Branch target+2					IF	ID	EX	MEM	WB

- Compiler plant für den Delay-Slot
- Instruktion von vorn in den slot schieben
  - Sprung unabhängig von der Instruktion
- Instruktion vom Ziel vorziehen
  - muss evtl. kopiert werden



  - einmal zuviel berechnet
  - Ergebnis korrigieren?
  - Ergebnis hinter der Schleife nicht benutzen
- Nachfolge-Instruktion vorziehen
  - Register in der Sprung-Region unverändert?
- Auch dynamisch in Hardware



3.3.1.2 Loop Unrolling

- Spünge sind Pipeline-Feinde
  - häufig am Ende von Schleifen
  - Anzahl Sprünge pro Durchlauf reduzieren

```
for (i=0;i<1000;i++)
x[i]=x[i] + y [i];
```

```
i=0;
while (i<1000) {
x[i]=x[i] + y [i];
x[i+1]=x[i+1] + y [i+1];
x[i+2]=x[i+2] + y [i+2];
x[i+3]=x[i+3] + y [i+3];
i+=4;}
```

  - reduziert auch Branch-Delay-Slots
- Schleifenzähler
  - n Schleifendurchläufe
  - Ausrollfaktor k
  - Hauptschleife n DIV k
  - Aufräumschleife n MOD k
  - Autoinkrement?

3.3.1.3 Pipeline Scheduling im Compiler

- Code wird für Prozessor optimiert
  - Eigenschaften der Prozessor-Pipeline bekannt (exposed)
  - delay-slot(s)
  - Resource Hazards
  - eigentlich schlecht: spätere Verbesserungen nicht nützlich
- Instruktionen umsortieren
  - besser, je mehr Instruktionen zur Auswahl
  - Schleifen ausrollen
  - datenabhängig Instruktionen auseinanderziehen
  - Branch-Slots füllen
- Datenabhänigigkeit
  - Datenfluß zwischen Instruktionen: RAW
  - statische Abhängigkeit
- Namensabhängigkeit
  - Antiabhängigkeit kann zu WAR führen
  - Ausgabe-Abhängigkeit kann zu WAW führen
  - Register-Renaming (andere Register verwenden)

- Kontroll-Abhängigkeit
  - Codeblock durch IF bewacht
  - welche Instruktionen können herausgezogen werden?
  - Instruktionen in den Block hineinschieben?
- Exception-Verhalten

```
      BEQZ    R2, L1
      LW      R1, 0(R2)
L1:
```

  - Exception ignorieren falls Sprung gemacht wird
- Datenfluß
  - dynamische Abhängigkeit
  - Spekulation

Datenfluß-Abhängigkeit (R1)	R4 später nicht verwendet	R4 spekulativ früh berechnen
`ADD R1,R2,R3`	`ADD R1,R2,R3`	`SUB R4,R5,R6`
`BEQZ R4,L`	`BEQZ R12,L`	`ADD R1,R2,R3`
`SUB R1,R5,R6`	`SUB R4,R5,R6`	`BEQZ R12,L`
`;stall`	`;stall`	`ADD R5,R4,R9`
`L: OR R7,R1,R8`	`ADD R5,R4,R9`	`L: OR R7,R8,R9`
	`L: OR R7,R8,R9`	

- Beispiel: for (i=0;i<1000;i++) x[i]=x[i] +s;
- DLX-Code einfach

```
loop:   LD     F0,0(R1)
        ADDD   F4,F0,F2      ; s in F2
        SD     0(R1),F4
        SUBI   R1,R1,#8      ; double word
        BNEZ   R1,loop
```

- Ausführung mit DLX-Pipeline

```
loop:   LD     F0,0(R1)
        stall
        ADDD   F4,F0,F2      ; 3 Zyklen!
        stall
        stall
        SD     0(R1),F4
        SUBI   R1,R1,#8      ; double word
        stall
        BNEZ   R1,loop
        stall
```

- Instruktionsreihenfolge ändern

```
loop:  LD     F0,0(R1)
       SUBI   R1,R1,#8      ; double word
       ADDD   F4,F0,F2      ; 2 Zyklen
       stall
       BNEZ   R1,loop
       SD     8(R1),F4      ; Instruktion verändert
```

  - SD eigentlich abhängig von SUBI
  - SD-Instruktion verändern
  - 6 Zyklen pro Durchlauf statt 10

- Loop unrolling

```
loop:  LD     F0,0(R1)
       ADDD   F4,F0,F2
       SD     0(R1),F4
       LD     F0,-8(R1)
       ADDD   F4,F0,F2
       SD     -8(R1),F4
       LD     F0,-16(R1)
       ADDD   F4,F0,F2
       SD     -16(R1),F4
       LD     F0,-24(R1)
       ADDD   F4,F0,F2
       SD     -24(R1),F4
       SUBI   R1,R1,#32
       BNEZ   R1,loop
```

  - Adressen in Instruktionen verändert
  - 28 Zyklen = 7 Zyklen pro Element
  - 14 Inst + 4 LD-stall + 1 SUBI -stall, 8 ADDD-stalls + branch-delay

- Schleife planen (scheduling)

```
loop:  LD     F0,0(R1)
       LD     F6,-8(R1)
       LD     F10,-16(R1)
       LD     F14,-24(R1)
       ADDD   F4,F0,F2
       ADDD   F8,F6,F2
       ADDD   F12,F10,F2
       ADDD   F16,F14,F2
       SD     0(R1),F4
       SD     -8(R1),F8
       SUBI   R1,R1,#32
       SD     -16(R1),F12  ;R1 noch nicht geändert
       BNEZ   R1,loop
       SD     8(R1),F16    ;R1 fertig: 8=32-24
```

  - 14 Zyklen = 3.5 Zyklen pro Element
  - ohne stalls

3.3.2 Hardwarelösungen

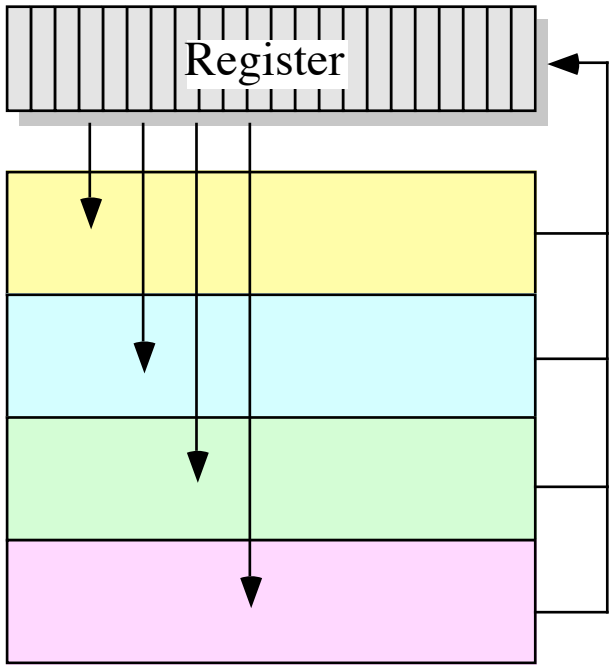
- Mehrere Ausführungseinheiten
- Pipeline-Eigenschaften nicht in das Programm einbauen
  - Prozessor kann verbessert werden
  - schnelle Ausführung alten Codes
  - CISC auf RISC
- Beispiel Pentium Pro und später
  - 80x86-Instruktion auf eine oder mehrere µOperationen aufteilen

1 µOperation	`mov eax,[es:0x10]`	**Wert laden**
2 µOperationen	`mov [es:0x10],eax`	**berechne Zieladresse**
		speichern
4 µOperationen	`add [es:0x10],eax`	**Wert laden**
		addieren
		berechne Zieladresse
		speichern

  - µOperationen im Instruktionspool abgelegt
  - dynamisch planen

3.3.2.1 Dynamic Scheduling

- Instruction-Decode entscheidet über Ausführung
  - Warteschlange mit Befehlen
  - structural Hazards auswerten
  - nächsten möglichen Befehl starten
  - auf Functional-Units verteilen
  - evtl. dynamisch umordnen
- Instruktions-Resultat-Reihenfolge geändert
  - WAR, WAW können entstehen
  - Unprecise Exceptions?
  - Planung abhängig von Registern
  - Instuktionen können lange in FU hängen
- Modifikation der DLX-Pipeline
  - DLX-ID prüft structural H. und data H.
  - Neu: ID aufteilen in *Issue* und *Read Operands*
  - Warteschlange zwischen Fetch und Issue
  - DLX: Nur bei Floating Point Unit
  - 1 Integer, 2 FP-MULs, 1 FP-DIV, 1 FP-ADD



- Beispielproblem

```
DIVD   F0,F2,F4      ; viele Zyklen
ADDD   F10,F0,F8
SUBD   F12,F8,F14    ; WAW: SUBD  F10,F8,F14
                     ; WAR: SUBD  F8,F8,F14
```

  - SUBD vorziehen
- Registerverwaltung
  - Auflösung der data hazards
  - Warten auf Daten
  - Warten beim Speichern in Register
- Scoreboard
  - startet Instruktionen (issue) in EX[i]
  - gibt Register als Operanden frei (RAW)
  - stellt fest, dass EX[i] ist fertig
  - überprüft ob WB Register schreiben kann (WAR, WAW)
- Scoreboard-Datenstrukturen
  - Instruktions-Status: issued, read, execution complete, write result
  - Funktionseinheit-Status
  - Register Resultat Status

- Scoreboard-Tabellen [CDC 6600, 1964]
  - Schnappschuss für DLX-Pipeline

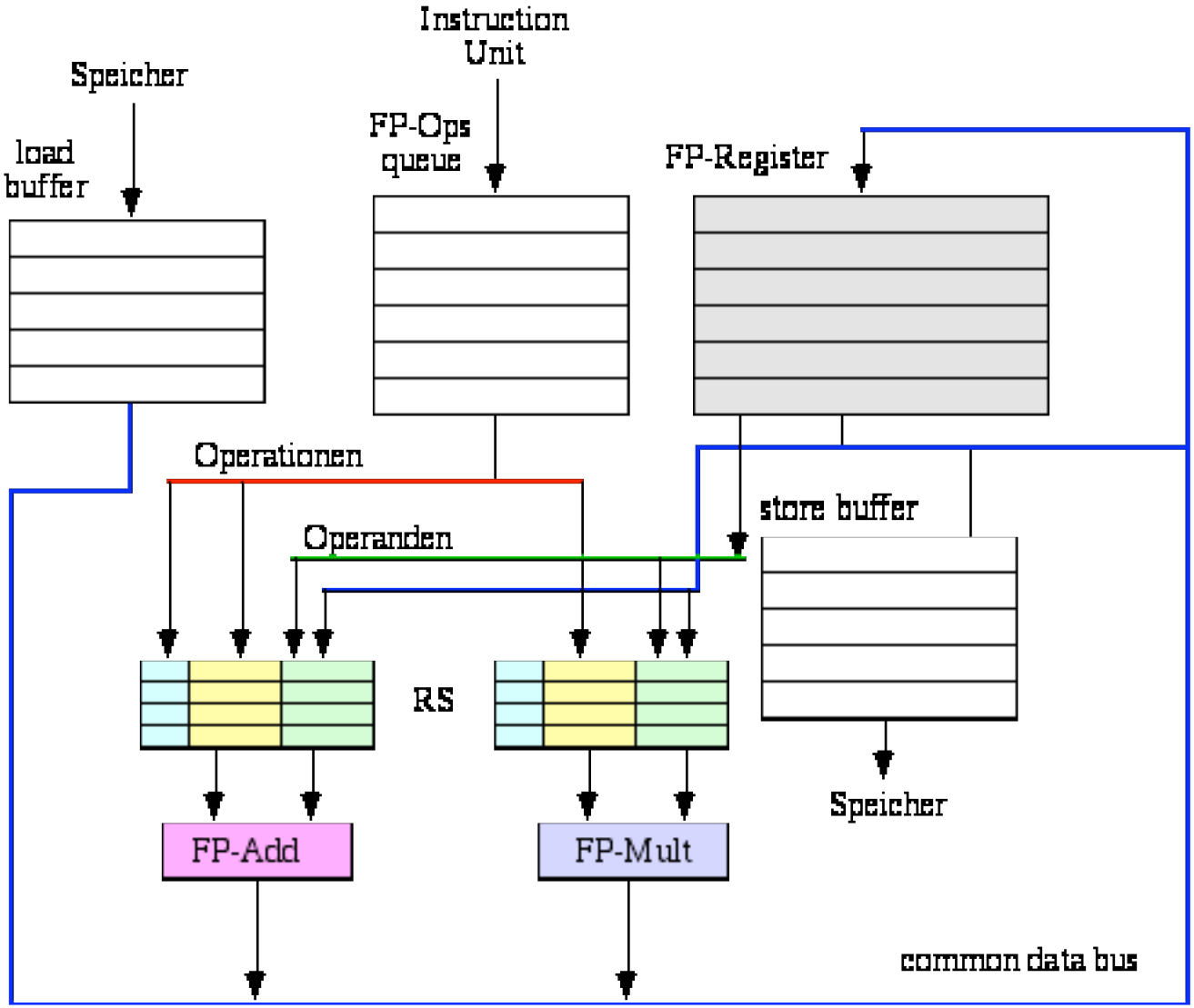
Instruktion		issued	read ops	exec. comp.	write back
`LD`	`F6,32(R2)`	true	true	true	true
`LD`	`F2,48(R3)`	true	true	true	
`MULTD`	`F0,F2,F4`	true			
`SUBD`	`F8,F6,F2`	true			
`DIVD`	`F10,F0,F6`	true			
`ADDD`	`F6,F8,F2`				

Funktionseinheit	busy	Op	RegD	RegS1	RegS2	FU1	FU2	rdy1	rdy2
Integer	true	Load (2)	F2	R3				false	
FMult1	true	Mult	F0	F2	F4	Integer		false	true
FMult2	false								
FAdd	true	Sub	F8	F6	F2		Integer	true	false
FDiv	true	Div	F10	F0	F6	FMult1		false	true

Register	F0	F2	F4	F6	F8	F10	F12	…	F30
Funktionseinheit	FMult1	Integer			FAdd	FDiv			

- Register Renaming
  - eigentlich: weitere Register verwenden
  - Register nicht unbedingt im Programmiermodell
  - vermeidet 'unechte Hazards': WAR und WAW
  - wirkt nicht bei Datenfluss-Hazards
- Tomasulu [Robert Tomasulu, 1967]
  - FPU für IBM 360/91
  - 'verteiltes' Scoreboard
  - Common Data Bus (CDB)
  - implizites Register-Renaming mit 'Reservation Stations'
  - aktive Register
  - Load und Store Puffer
- Reservation Station (RS)
  - Puffer vor Funktionseinheit
  - Operation und Operanden
  - Register beim Eintrag in RS mit anderer RS verbinden
  - verhindert WAR und WAW
  - aktiver Puffer: besorgt Operanden vom CDB (ähnl. forwarding)
  - ohne Register-'Umweg'

- Tomasulu-DLX-FPU



3.3.2.2 Spekulative Ausführung

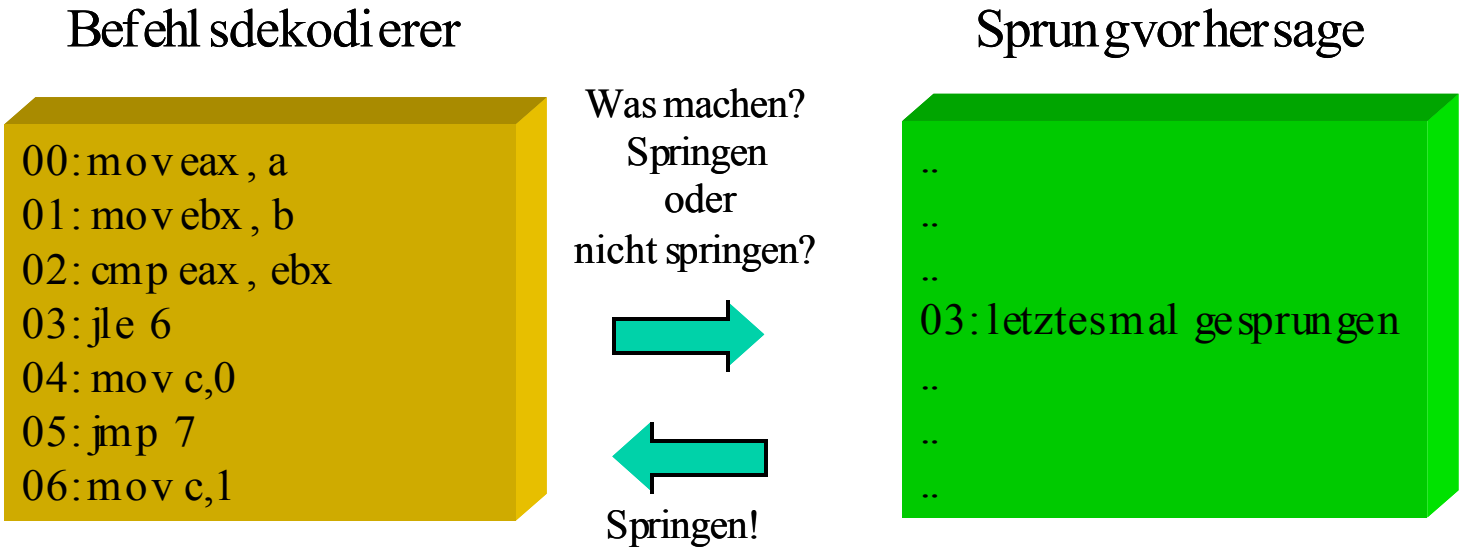
- Kontroll-Abhängigkeiten
  - Schleifen, Verzweigungen, Assertions, …
  - Branch-Delay
  - ILP: Anzahl-Instruktionen in Verarbeitung steigt
- Branch Prediction
  - Sprungentscheidung dynamisch vorhersagen
  - ausgewählte Alternative starten
  - rückgängig machen falls Fehlentscheidung
- Branch Prediction Buffer (branch history table)
  - Speicher, adressiert mit niederwertigen Adressbits
  - enthält letzte Entscheidung für diese Adresse (nicht Befehl!)
  - Verbesserung: n-Zähler
  - Entscheidung > n/2 > Sprung
  - Sprung inkrementiert, nicht springen dekrementiert
- Beispiel 4096 Puffereinträge á 2 Bit
  - 82% - 99% richtige Vorhersage bei SPEC89
- Korrelierte Prädiktoren für abhängige Sprünge

- Instruktionspool sollte immer gefüllt sein

```
        cmp a,b
        jle @else
        mov ecx, 0
        jmp @end
@else:  mov ecx, 1
@end:
```

- Problem
  - welcher Befehl soll als nächstes dekodiert werden?
  - mov ecx,0 oder mov ecx,1
  - Prozessor führt Code spekulativ aus
- Realisierung im PII & PIII
  - Puffer mit 512 Einträgen
  - Adresse des Sprungbefehls
  - Sprung ausgeführt: Ja oder Nein
  - jeder Sprung (ausgeführt oder nicht) wird vermerkt

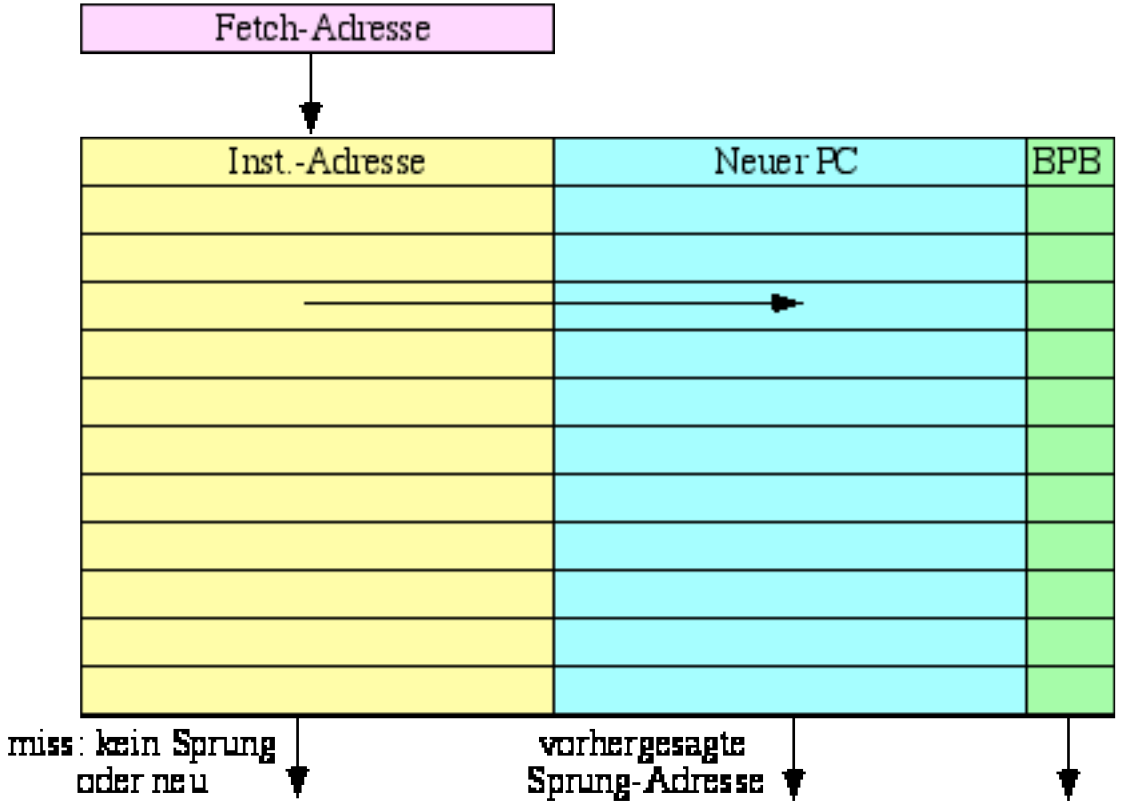
- Beispiel der dynamischen Sprungvorhersage



- Statische Sprungvorhersage
  - falls Sprung der Sprungvorhersage nicht bekannt
  - bedingte Rückwärtssprünge: Sprung ausführen
  - bedingte Vorwärtssprünge: Sprung nicht ausführen
- Strafzyklen (Penalties)
  - keine: korrekt vorhergesagt, dass der Sprung nicht ausgeführt wird
  - 1 Zyklus: korrekt vorhergesagt, dass Sprung ausgeführt wird
  - 9 - 26 Zyklen bei falscher Vorhersage

- Branch Target Buffer
  - nicht Entscheidung vorhersagen, sonder Ziel
  - Test beim Laden *jeder* Instruktion
  - neue Sprünge eintragen
  - evtl. Ziel ändern

```
if (PC in BTB) nextPC = item[PC].predictedPC
```



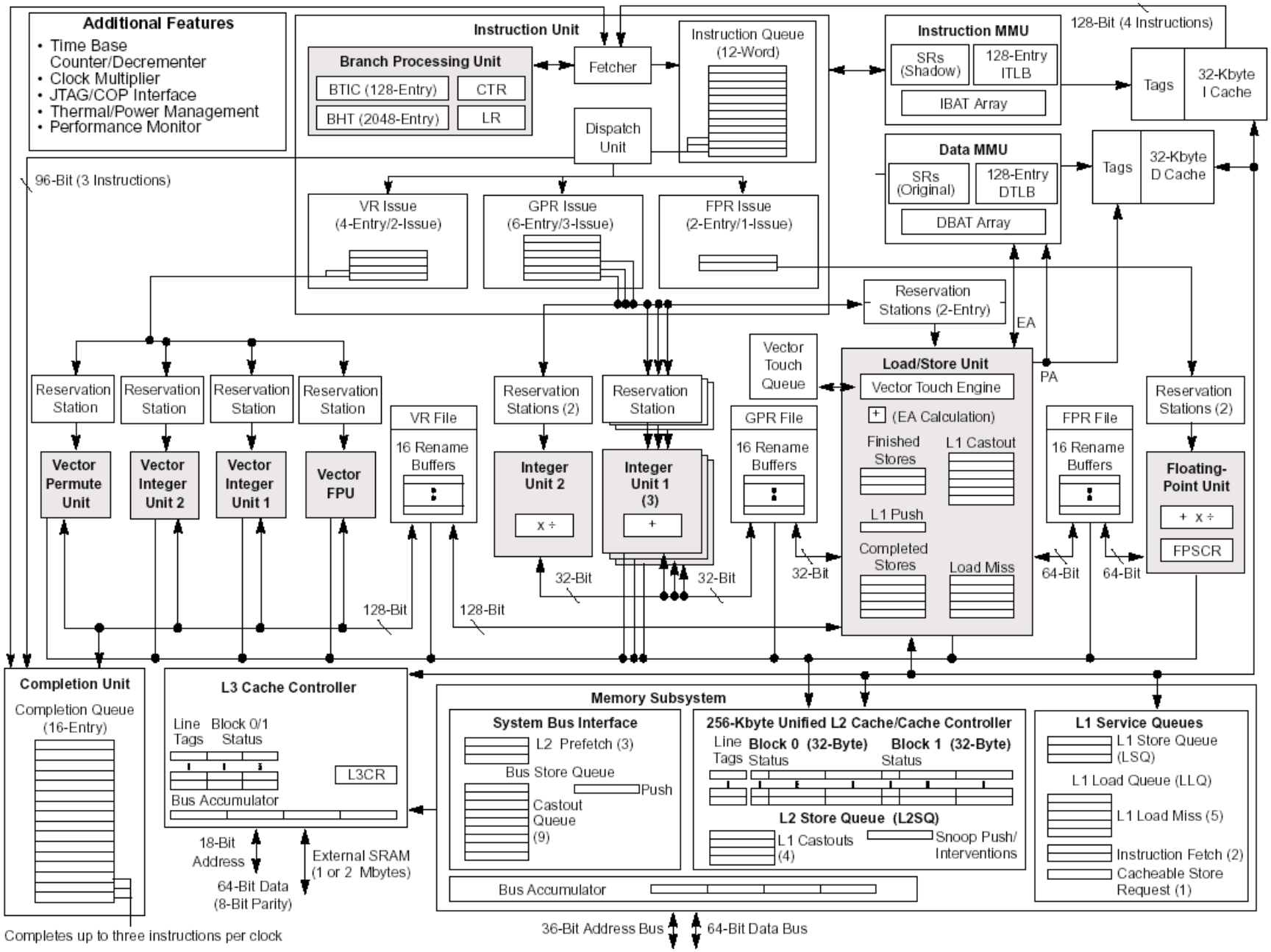
3.3.2.3 Stichwort: Superskalare Architektur

- Multiple Instruction Issue
  - an mehreren Instruktionen parallel arbeiten
  - parallele Pipelines
  - aber evtl. Abhängigkeiten
- Very Long Instruction Word (VLIW)
  - mehrere Befehle pro Instruktions-Adresse
  - statisch Funktionseinheiten zugeordnet
  - siehe Crusoe (oben) und Signalprozessoren
- EPIC: Explicitly Parallel Instruction Computing [IA64]
  - jede Instruktion trägt Branch-Zweig-Prädikat
  - spekulative Ausführung aller Zweige
- Superskalare Ausführung
  - 2 bis 8 Befehle gleichzeitig starten
  - abhängig von Hazards
  - mehrere Befehle gleichzeitig holen: 64/128 Bit
  - einfachste Variante: 1* Integer + 1 * FP
  - dynamisch geplant: Scoreboard oder Tomasulu

Zusammenfassung Instruktionsausführung

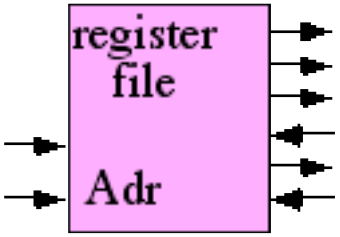
- Pipeline steigert Durchsatz
  - Phasen der Verarbeitung durch Puffer entkoppelt
- Probleme: Hazards
  - echte Datenabhängigkeit und Namensabhängigkeit
  - Ressource-Engpässe
  - Control-Hazards
- Statische Techniken zur Optimierung
  - Delay-Slot füllen
  - Schleifen ausrollen
  -> Instruktionsplanung
- Dynamische Techniken
  - Instruktionen bedingt starten: Hazard-Entdeckung
- Leistungssteigerung: Instruction Level Parallelism (ILP)
  - parallele Funktionseinheiten
  - Scoreboard, Tomasulu
  - completion unit bewertet spekulative Ausführung
  - VLIW oder multiple Issue

- PowerPC 7450

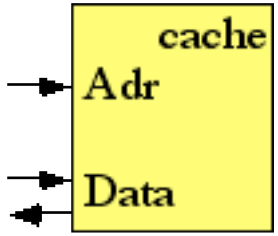


4. Speicherarchitektur

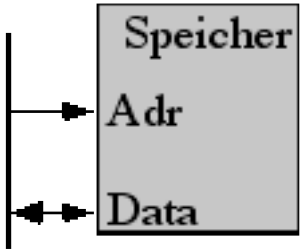
- Read/Write Speichertypen
  - statisches RAM
  - dynamisches RAM
  - EEPROM: Schreiben sehr langsam
  - Flash: Schreiben langsam
  - Magnetspeicher: langsamer Zugriff



- Hierarchie
  - Register (Flip-Flops u.ä.)
  - Level 1 Cache integriert in Prozessor (8-64 K, *2?)
  - on-chip Cache auf dem Prozessor-Die (L2, z.B. 128-1024 K)
  - externer/Backside Cache (L3, z.B. 2/4 MB)
  - Hauptspeicher
  - Virtueller Speicher: Auslagern von Teilen auf Magnetspeicher

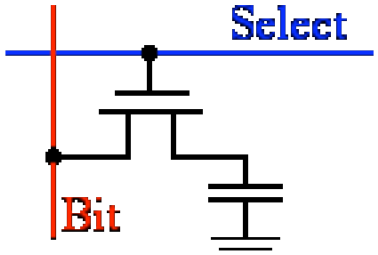


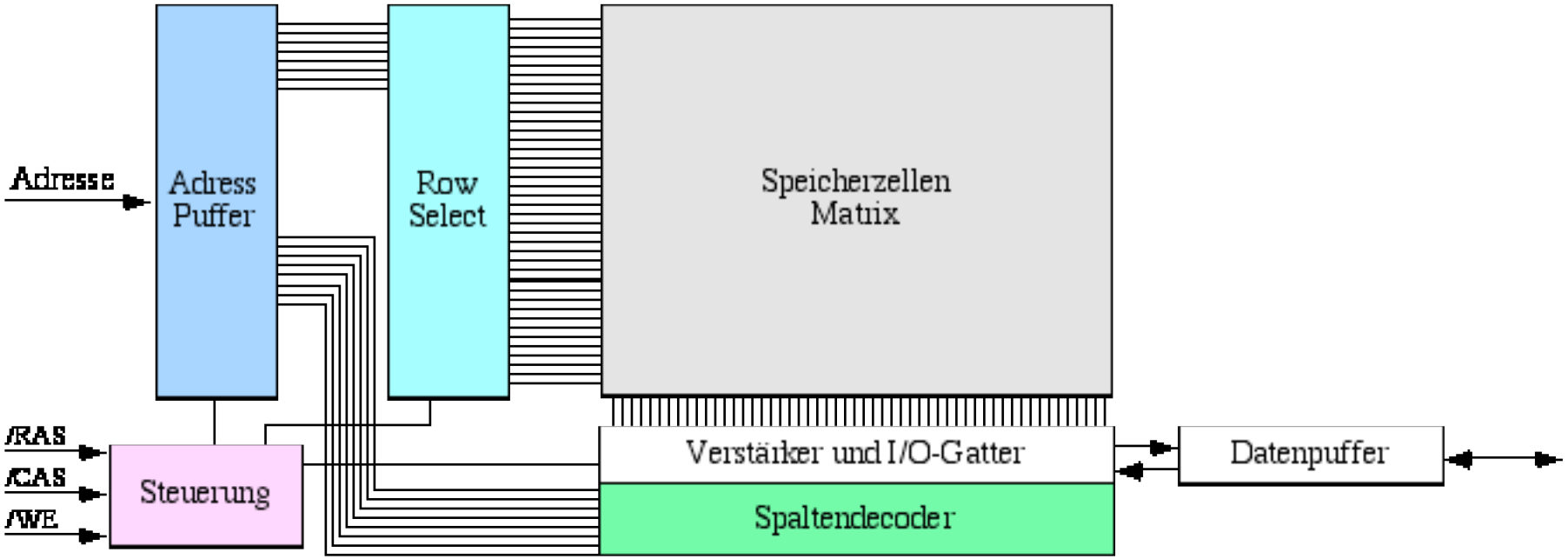
- (Flash-)ROM
  - Urlader
  - Parameter
  - Systemkomponenten: BIOS, Toolbox, etc.
  - Programme für Embedded Systems



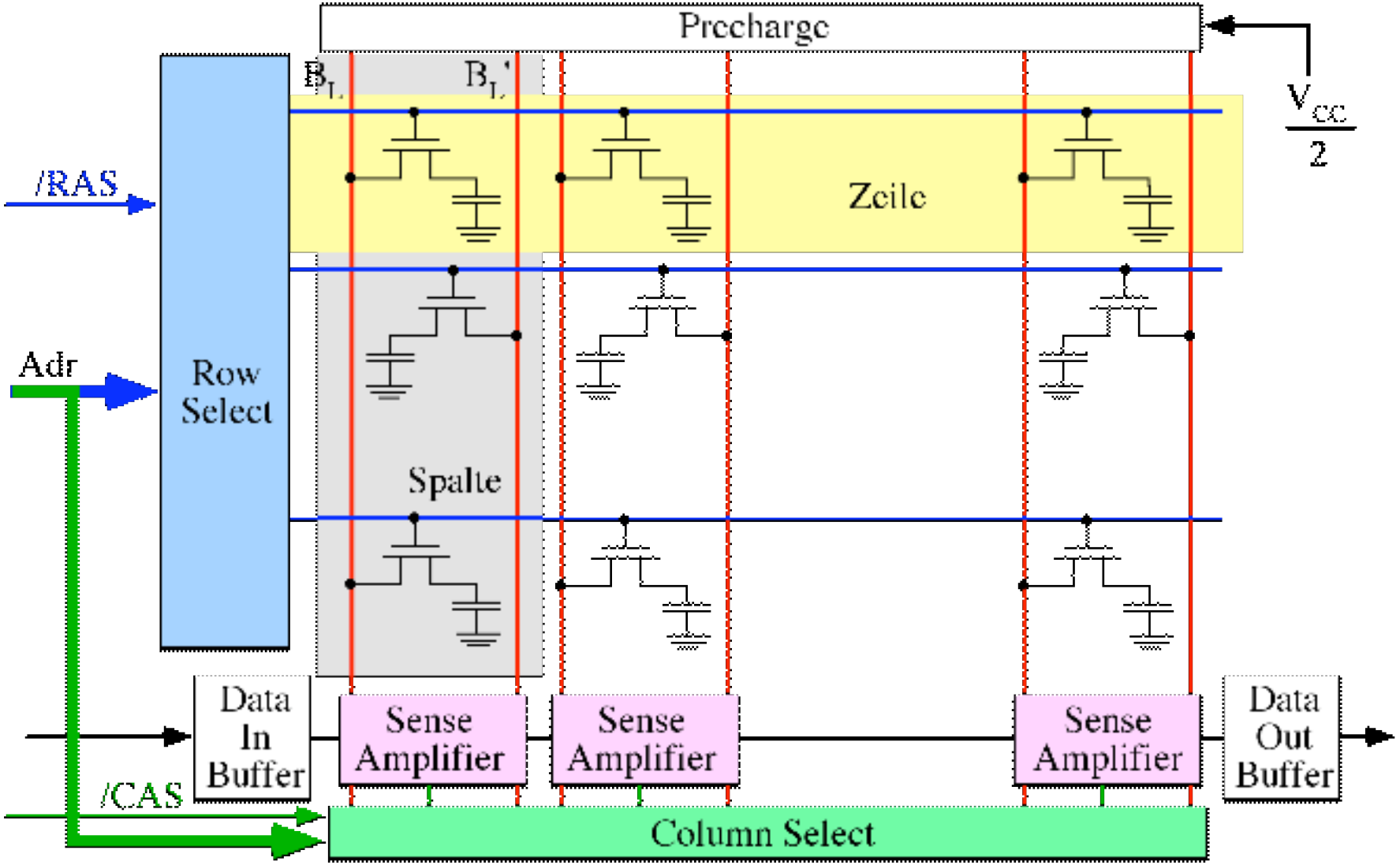
4.1 Hauptspeicher

- Dynamisches RAM (DRAM)
  - 1 Kondensator + 1 Transistor pro Speicherzelle (Bit)
  - Refresh nach 8 - 64 msec
- DRAM-Zugriff
  - Adressleitungen sparen: Multiplex
  - Anordnung als Matrix in Zeilen und Spalten
  - Row Address, danach Column Address
  - Strobes wenn gültiger Teil anliegt: RAS und CAS

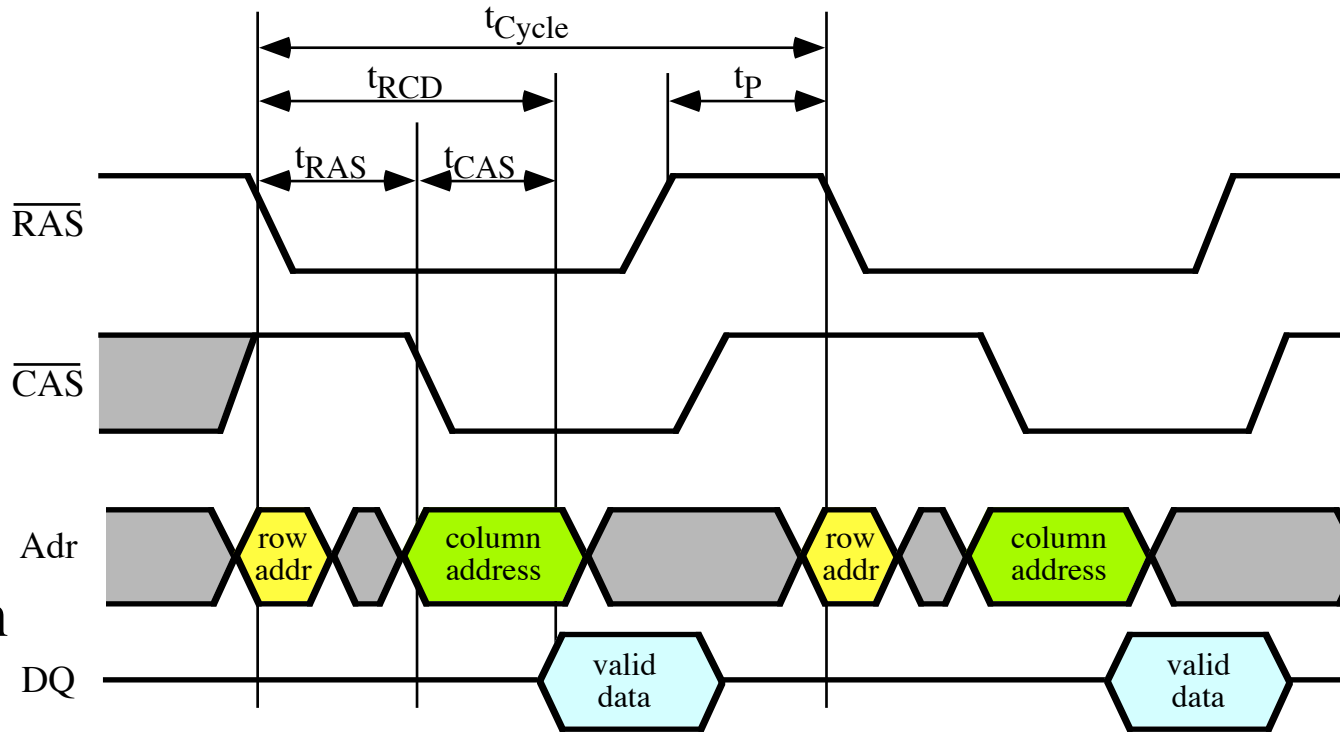




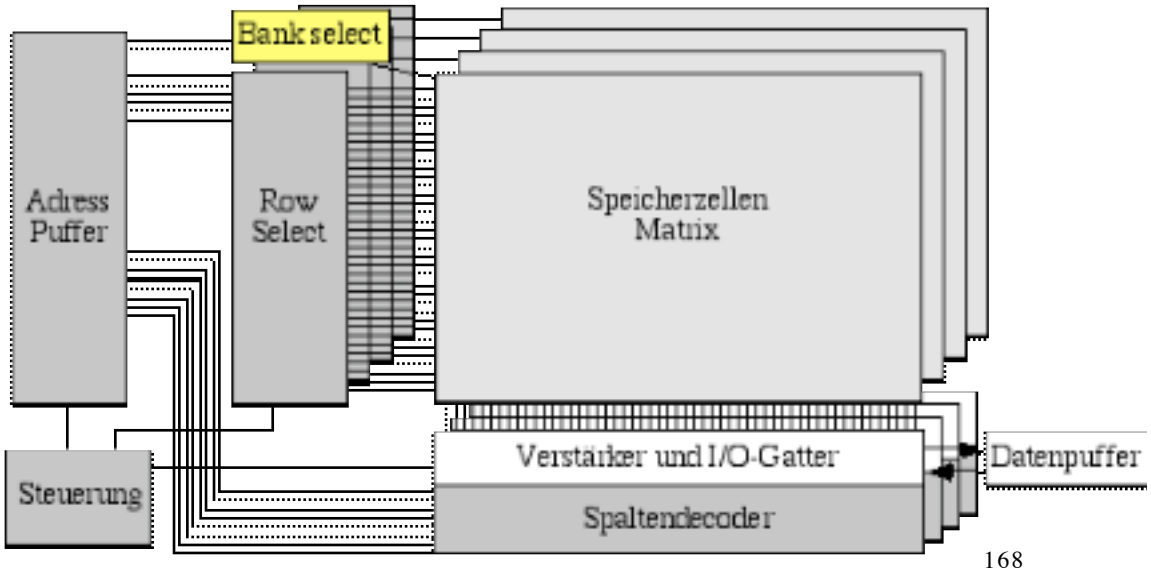
- Dynamisch?
  - Lesen zerstört Bit => Zurückschreiben (Refresh)
  - Auswerten geringster Ladungen (pF Kondensatoren)



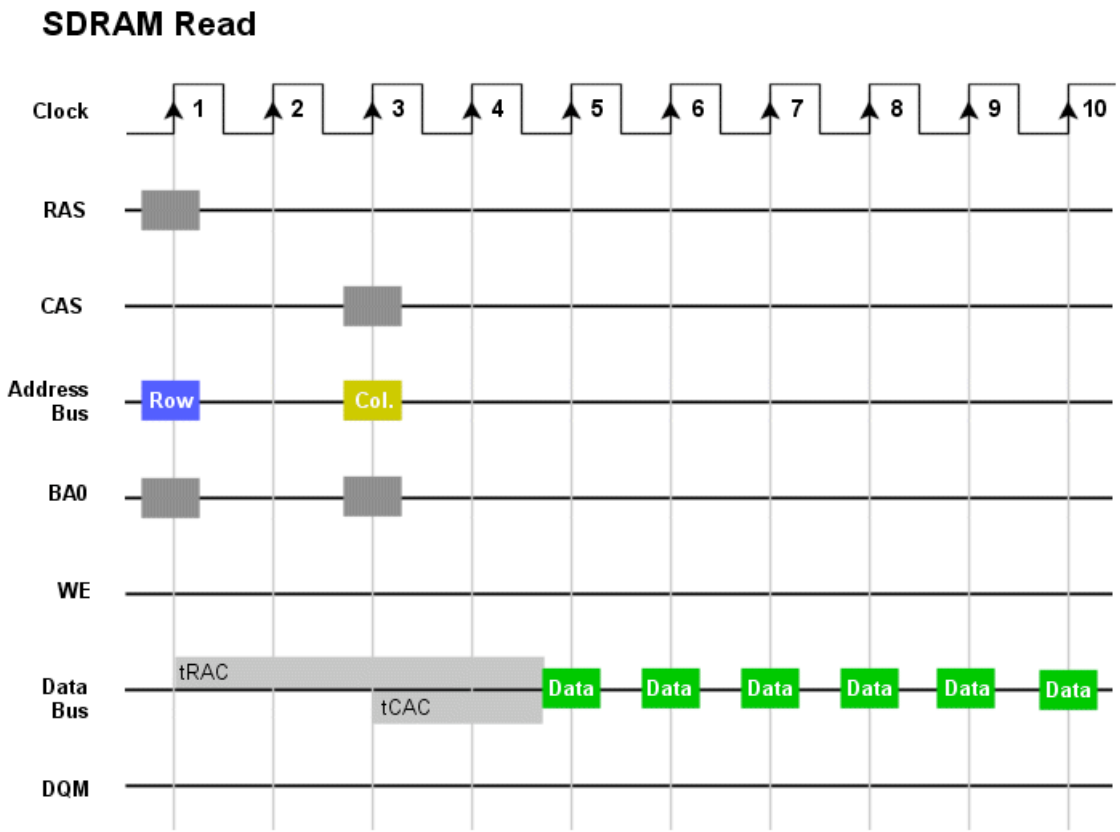
- Ablauf
  - R/W anlegen
  - MSB Adresse anlegen
  - /RAS
  - R-Adr. dekodieren
  - Zeile ansteuern
  - Zeile verstärken
  - LSB Adresse anlegen
  - /CAS
  - Bits auswählen
  - Daten liegen an
  - Kondensatoren aufladen
  - Precharge auf Vcc/2
- Zugriffszeit 50 ns (SDRAM)
- Zykluszeit 70 ns (SDRAM)
  - vor nächstem Lesen
  - Ladung wiederherstellen
  - Precharge abwarten



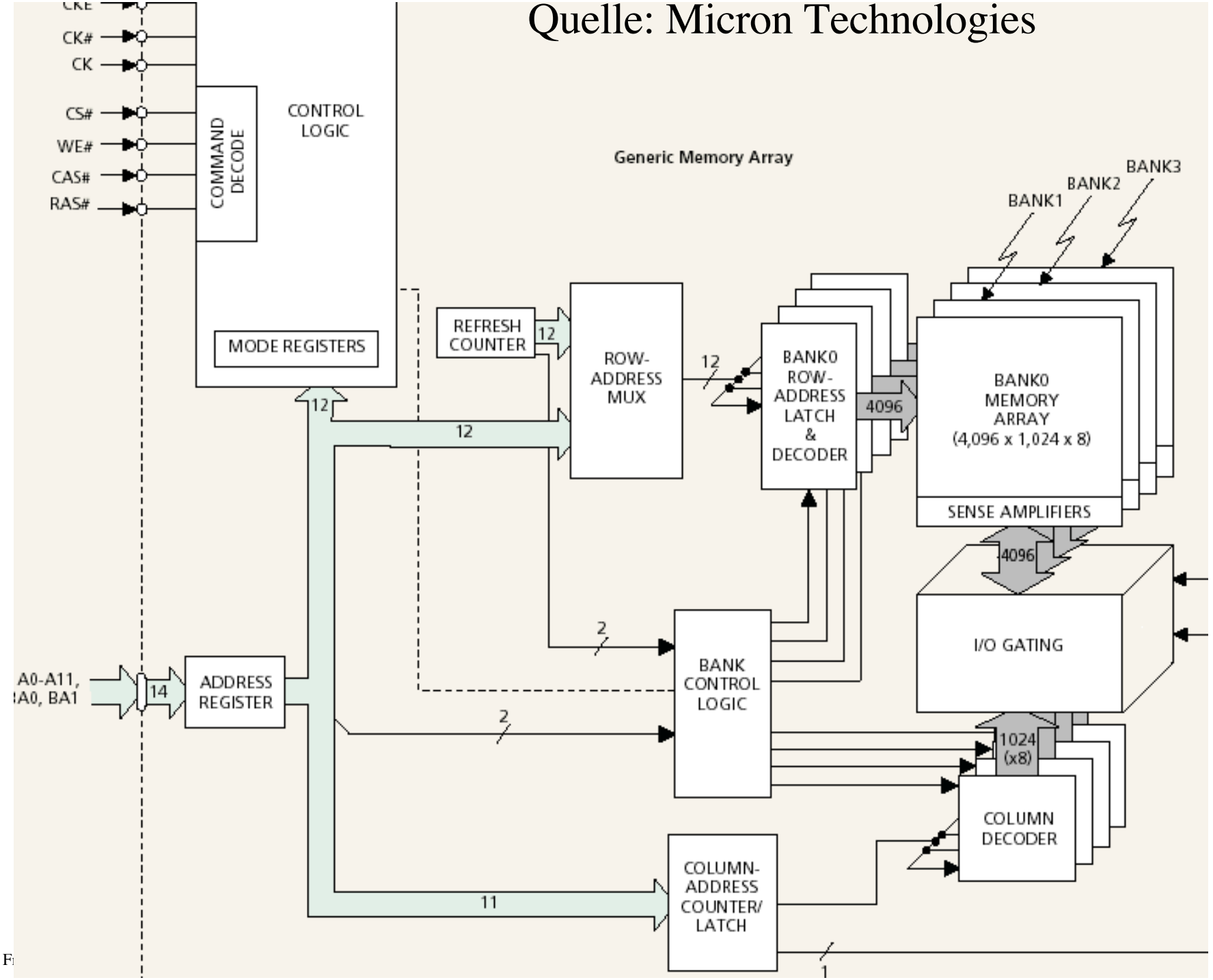
- Refresh
  - über Zeit geht Ladung verloren, falls kein R/W-Zugriff
  - 8-64 msec
  - Steuerung erzeugt Lese-Zyklen für Zeilen
  - extern durch 'Zeilenleser'
  - evtl. transparent für Prozessor (hidden refresh)
- Refresh zeilenweise
  - ~ 20 / sec
  - 2k, 4k, 8k Zeilen
  - pro Zeile T_{RC}-T_{CAC} => 40 ns
  - 20 * 4096 * 40 ns = 3.3 msec
  - Anzahl Zeilen beschränkt Zugreifbarkeit
- Anordnung
  - Bit-parallel: nMx8 etc.
  - 2 Bänke reduzieren Precharge-Wahrscheinlichkeit
  - entsprechend Zugriffsmuster optimieren



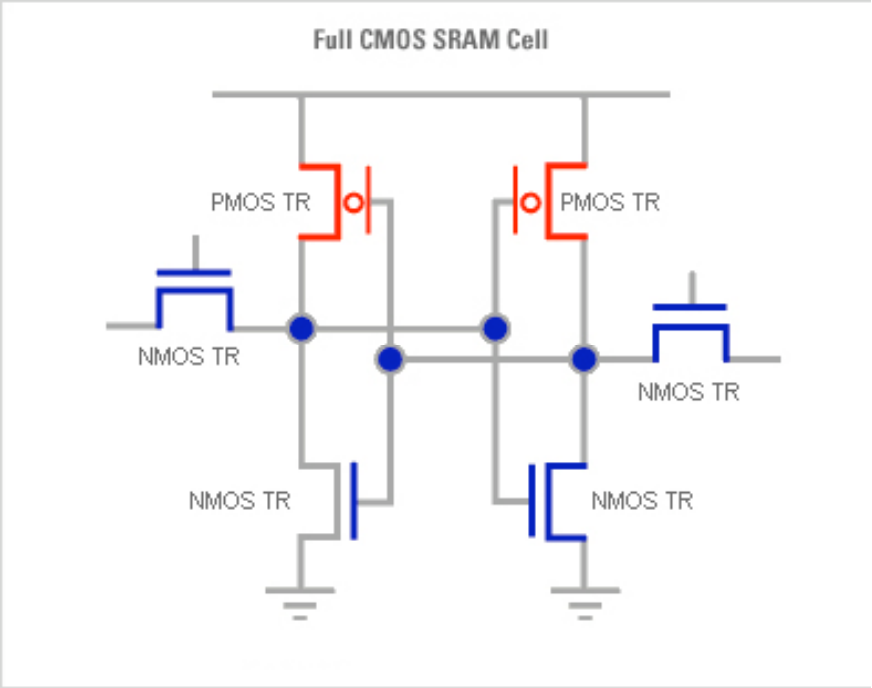
- DRAM asynchron
  - Adresse anlegen
  - RAS, CAS
  - CPU wartet bis Daten bereit
- FPM: (RAS - CAS-Daten) - CAS-Daten - CAS-Daten - CAS-Daten
  - 6-3-3-3 Bus-Zyklen
- EDO 5-2-2-2@66 MHz
- SDRAM
  - synchron zum Speicherbus
  - 100, 133, 266 MHz
  - CPU kann anderes tun
  - timing z.B. 3-2-2
  - CAS-Latenz
  - RAS-CAS Delay
  - RAS precharge
- DDR-RAM
  - Datenaustausch an beiden Flanken



Quelle: Micron Technologies

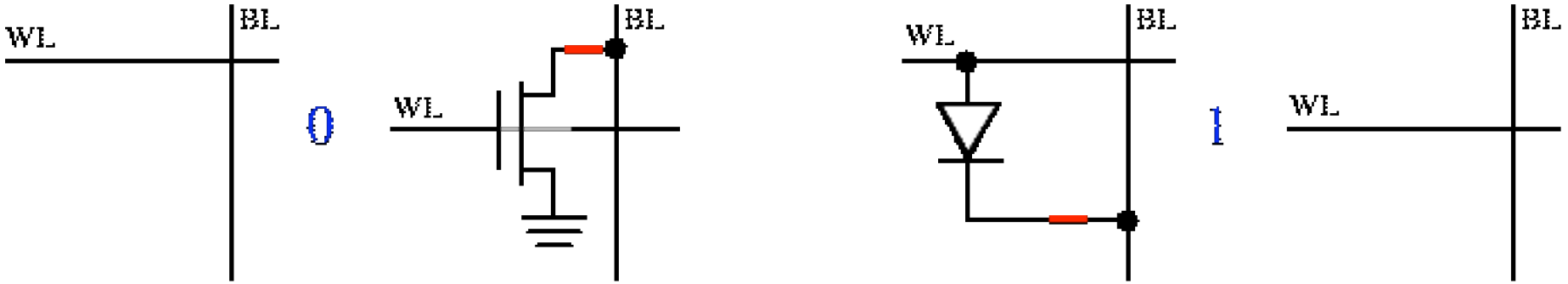


- DRAM-Probleme
  - Adressmultiplex (Row-Column) kostet Zeit
  - Precharge bestraft Mehrfachzugriff auf Chip hintereinander
  - FP-RAM, EDO, SD-RAM, … beschleunigen sequentiellen Zugriff
  - 50 ns = 50 Prozessorzyklen bei moderner CPU
- Speichersystem-Optionen
  - n*Wortbreite => n*Speicherdurchsatz (Alpha AXP 21064: 256 bit)
  - Interleaving
  - Unabhängige Speicherbänke
- Statisches RAM (SRAM)
  - 4 - 6 Transistoren pro Speicherzelle (Bit)
  - alle Adressleitungen zum Chip
  - kein Refresh
  - Zugriffszeit = Zykluszeit
  - Kapazität ca. 1/4 - 1/8 von DRAM
  - 8 mal teurer
  - 8 - 16 mal so schnell



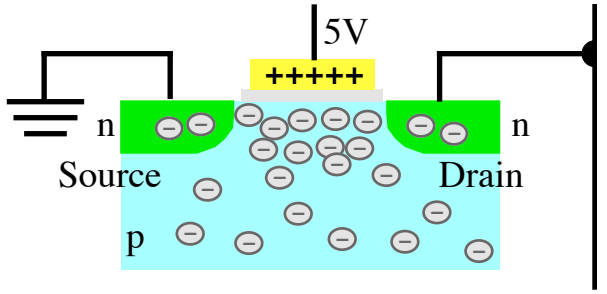
Quelle: Samsung

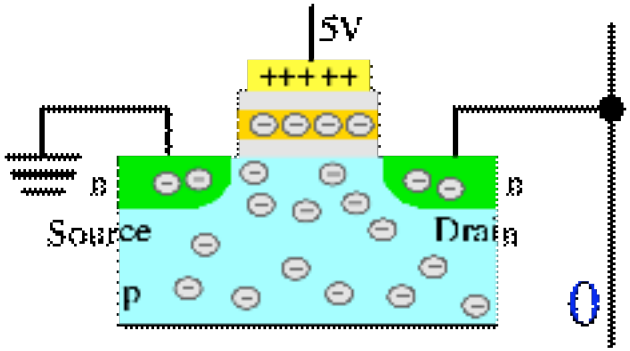
- ROM - Read Only Memory
  - Dioden-Matrix
  - Zeilenleitung aktivieren
  - Diode -> 1; fehlende Diode -> 0
  - CMOS: Transistoren an den verbundenen Kreuzungen
  - maskenprogrammiert

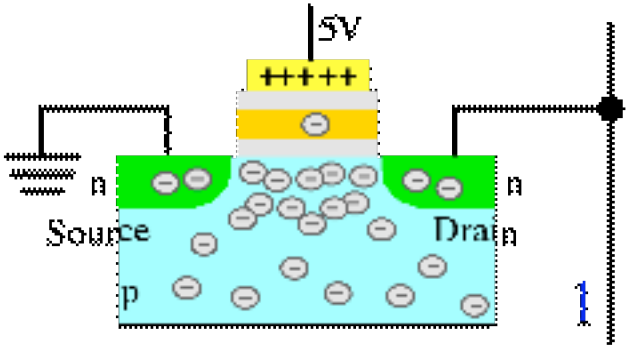


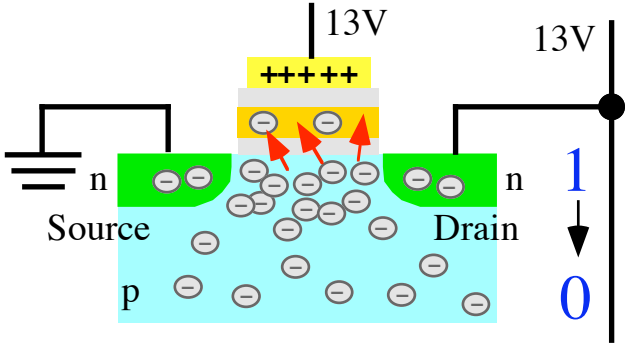
- PROM - Programmable ROM
  - alle Kreuzungen verbunden
  - Fuse an jeder Verbindung
  - Fuse mit Programmierspannung 'durchschmelzen'

- EPROM - Erasable PROM
  - besonderer Transistor mit 'Floating Gate': FAMOS
  - Fowler-Nordheim Tunneleffekt
  - Löschen mit UV-Licht

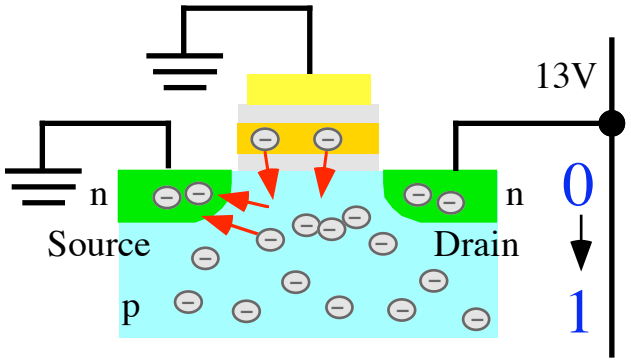




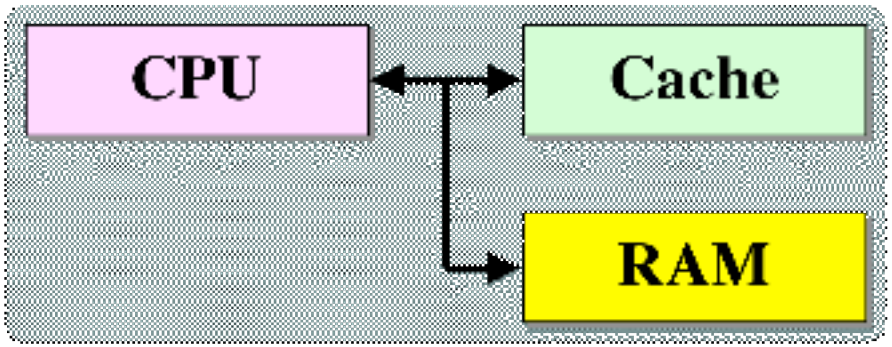




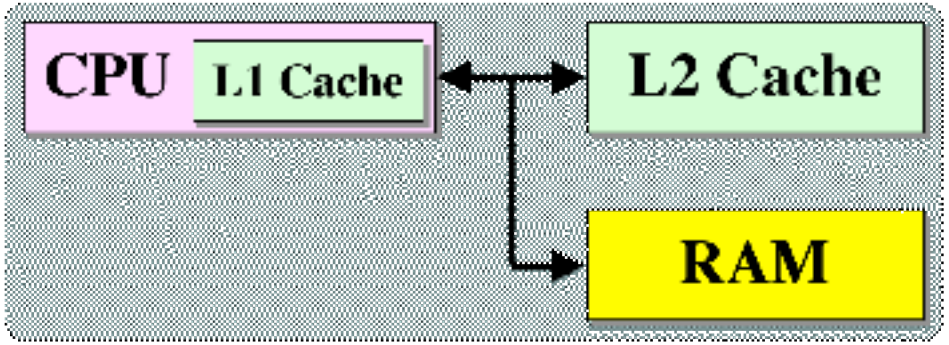
- EEPROM - Electrically EPROM
  - floating gate Transistoren elektrisch löschbar
  - Zellen 'einzeln' löschen
- Flash Memory
  - ganze Bänke löschen
  - modern: mehrwertige Logik



4.2 Cache



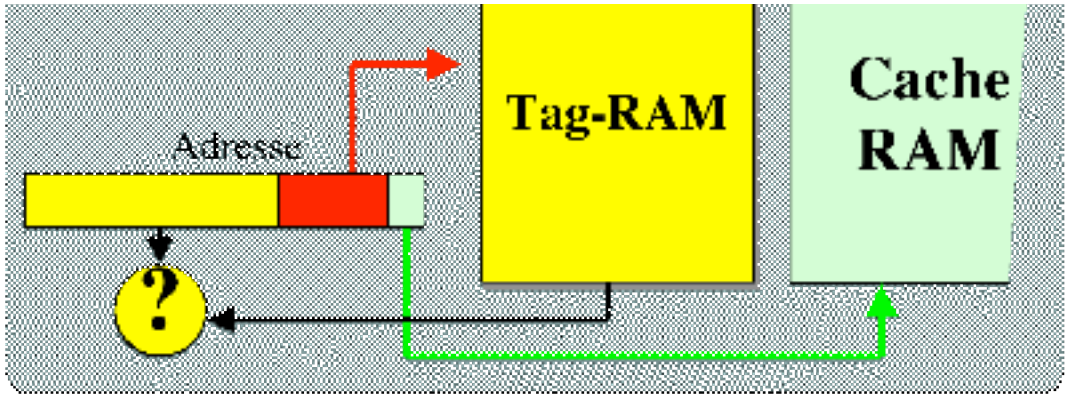
- Problem
  - Pipelinestufe MEM
  - Daten in 1 Zyklus, sonst stall(s)
  - DRAM zu langsam
  - SD-RAM 100 MHz => 50 Stalls!
- Schneller Pufferspeicher
  - zwischen Pipeline und Speicher
  - hält bereits benutzte Speicherinhalte
  - Befehle und Daten
  - eventuell getrennt für Befehle

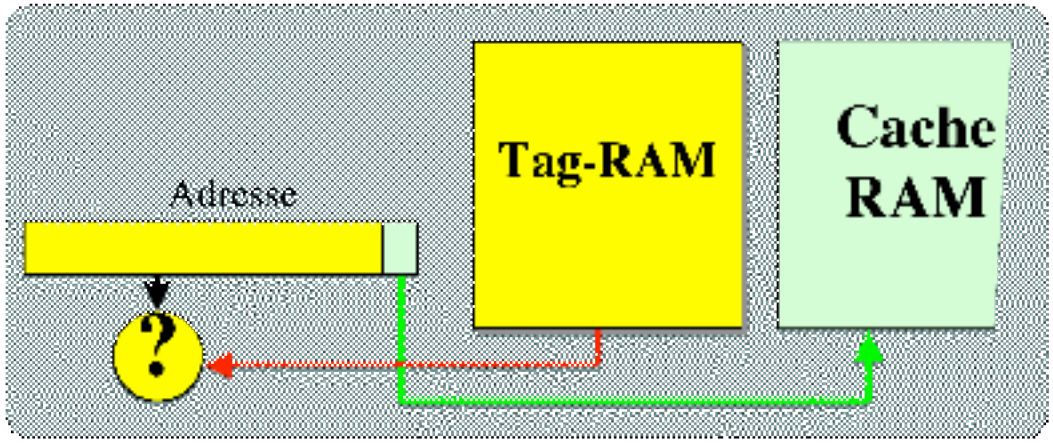


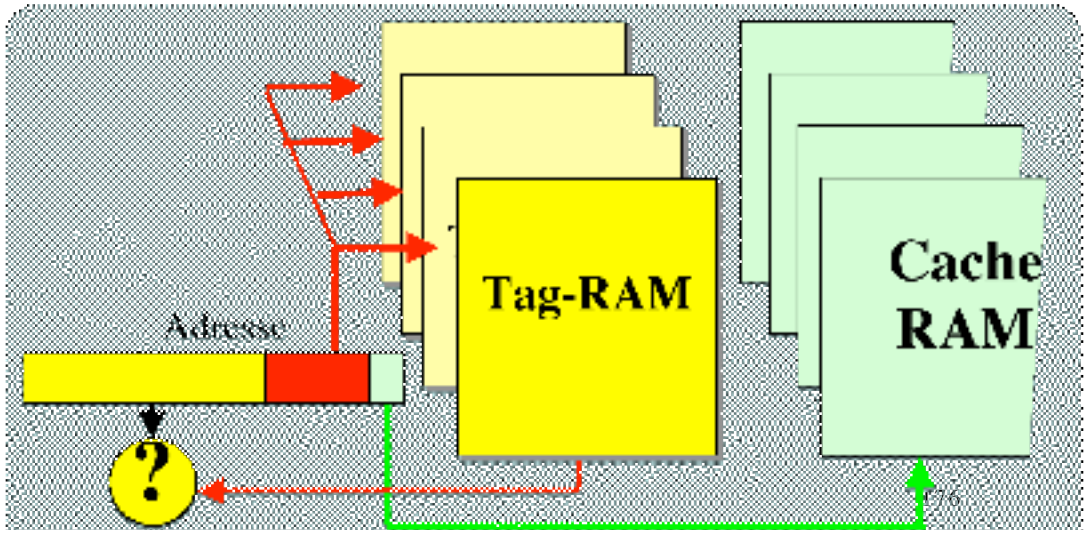
- Beschleunigung?
  - beim *ersten* Lesen einer Adresse Speicherwort puffern (langsam)
  - bei weiteren Lesevorgängen von Adresse Wort sofort liefern
  - beim Schreiben puffern und evtl. asynchron schreiben
  - Durchschreibestrategie
- Blockstrukturiert (Cachelines)
  - Cache speichert größere Speicherstücke
  -> Lokalitätsprinzip

- Schreibzugriffe 7% der DLX-Speicherzugriffe
  - 25% der datenorientierten Speicherzugriffe
- Write-Through (leicht implementierbar)
  - Cache-Inhalt sofort in unterer Hierarchiestufe nachführen
  - erhebliche Verzögerungen über den Speicherbus
  - ->Block-Write-Buffer reduziert Write-Stall
- Write-Back
  - Modifikationen im Cache durchführen
  - Cache-Inhalt erst beim Ersetzen der Zeile zurückschreiben
  - Dirty-Bit markiert beschriebene Cache-Zeile
  - Konsistenzproblem zwischen Cache & Hauptspeicher
  - Multiprozessor-Architekturen?
- Sonderfall:Write-Miss
  - Write-allocate: Zeile in den Cache laden und modifizieren
  - No-write-allocate: nur auf niedriger Stufe modifizieren

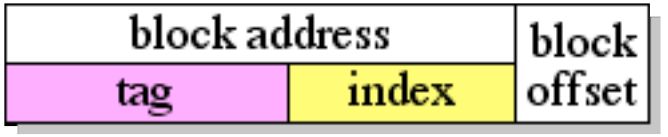
- Cachezugriff
  - Speicheradresse
  - Cacheadresse
  - ist das Wort im Cache?
  - wo liegt das Wort?
- Direct Mapped Cache
  - Abb. Adresse->Cacheeintrag
  - Vergleich ob Treffer
- Voll assoziativ
  - Vergleich Adresse - Tag-RAM
  - kostet evtl. Zeit
  - Hardware aufwendig
- Weg assoziativ
  - Untermenge (Set, Blockgruppe)
  - Set-Auswahl direct mapped
  - in der Untermenge assoziativ
  - n Wege: n Blöcke im Set
  - n-way set associative





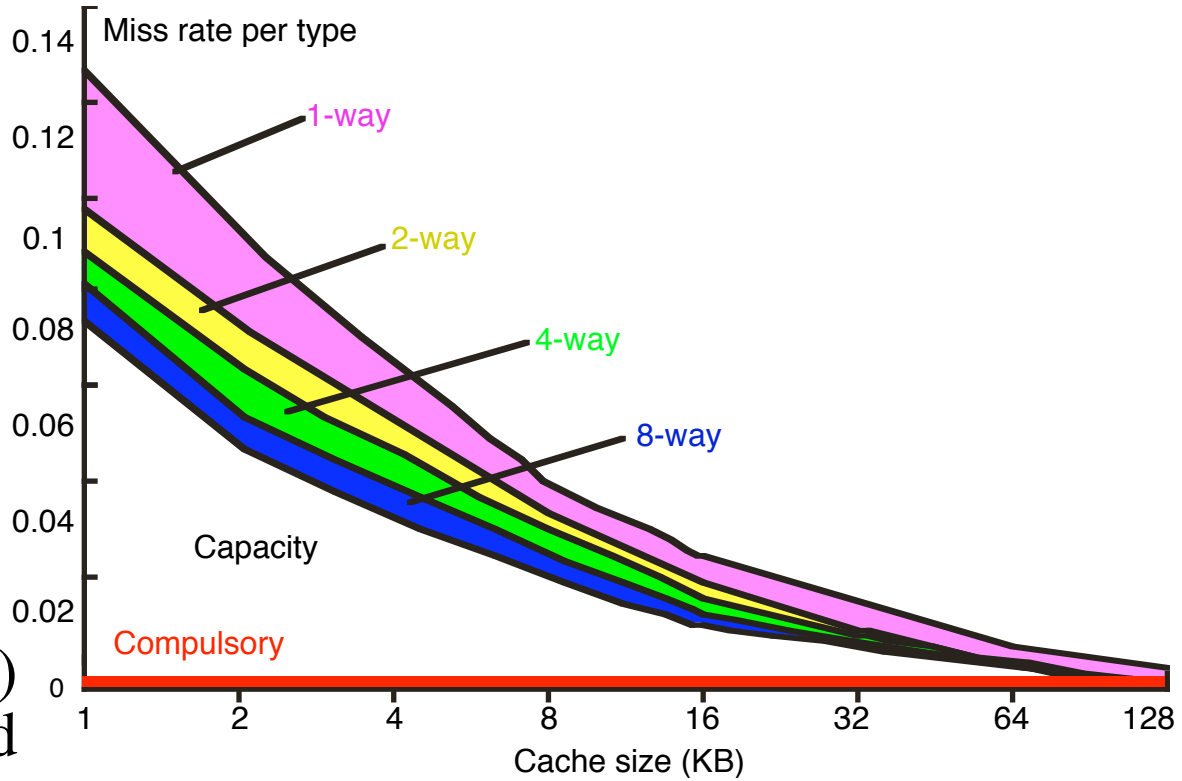


- Zugriffsstruktur Block-Tag
  - stückweise Interpretation der Adresse
  - Test der Blöcke im Set: tag
  - Index des Sets im Cache
  - Block-offset führt zum Datenwort
  - Gültigkeits-Bit (vaild-bit)



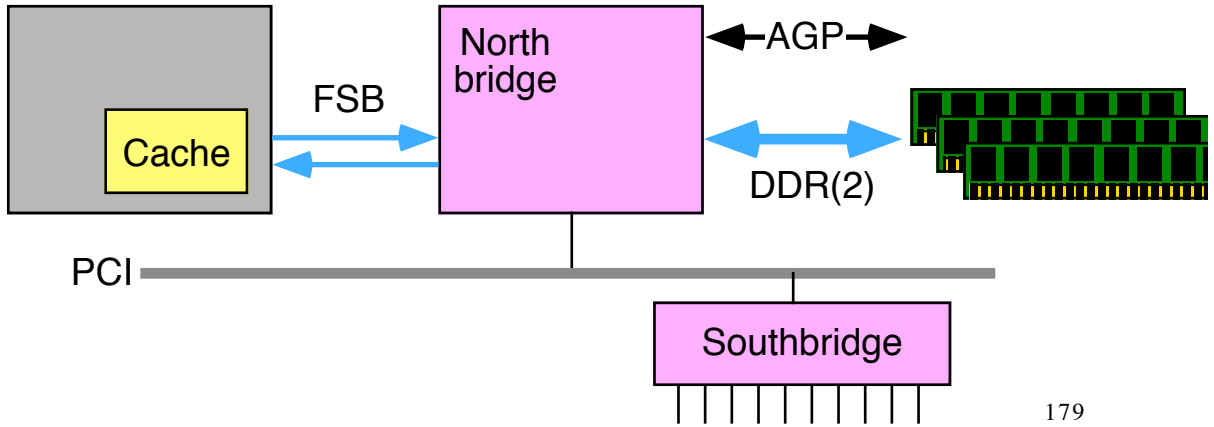
- Tag-Vergleich parallel
  - z.B. wired-XOR
  - parallel Daten aus dem Cache holen
  - n-Wege => n:1 Multiplexer
- Austausch mit dem Hauptspeicher
  - immer ganze Cache-Zeilen (z.B. 16 oder 32 Bytes)
  - als Burst-Zugriff transportiert
  - Cache-Zeile überschreiben: alten Inhalt vorher zurückschreiben?
- Block-Austausch-Strategie
  - Zufall
  - LRU: Least Recently Used

- Ursachen für Cache-Miss
  - unvermeidliche (compulsory)
  - größenbedingt (capacity)
  - konfliktbedingt (conflict)
- Miss-Rate reduzieren
  - größere Blöcke => größere(r) Miss-penalty
  - mehr Assoziativität => Kosten, Zeit für Vergleich?
  - 'Opfer'-Cache (z.B. 4 Einträge)
  - Prefetch bei Miss: Block[i] und Block[i+1]
  - Prefetch-Anweisungen vom Compiler
  - Compiler erhöht Lokalität des Programmes (z.B. Arrays mischen)
- Cache-Miss-Penalty reduzieren
- Cache-Hit-Zeit reduzieren
  - virtual Cache vs. physical Cache (aber: flush beim Prozesswechsel)



4.3 Bridges: Interface CPU/Cache - Speicher - Bus

- DDR/DDR2 Speicher
  - 64 bit pro Modul, Dual Channel: 2 Kanäle á 64 Bit
  - Adressbits nach Kapazität
  - CAS, RAS-CAS, Gesamtzyklus
  - DDR: z.B. 2-2-6 bei 200 MHz
  - DDR2: z.B. 4-4-12 bei 400 MHz
- CPU
  - Datenzugriff zunächst im Cache, evtl. Cache Miss
  - Cache-Line 4 oder 8 Worte = 128/256 bit
  - Speicherzugriff transportiert Zeile
  - Bsp: PPC 970FX: ADIN(0:43), ADOUT(0:43)
  - Multiplex: Adresse-Daten-Daten-Daten-Daten
  - P6-Bus: 64 bit 'split-transaction'
- Test [Bahmann, 2005]
  - sequentiell 75 MTrans/s
  - entspricht 11 CPU-Takten

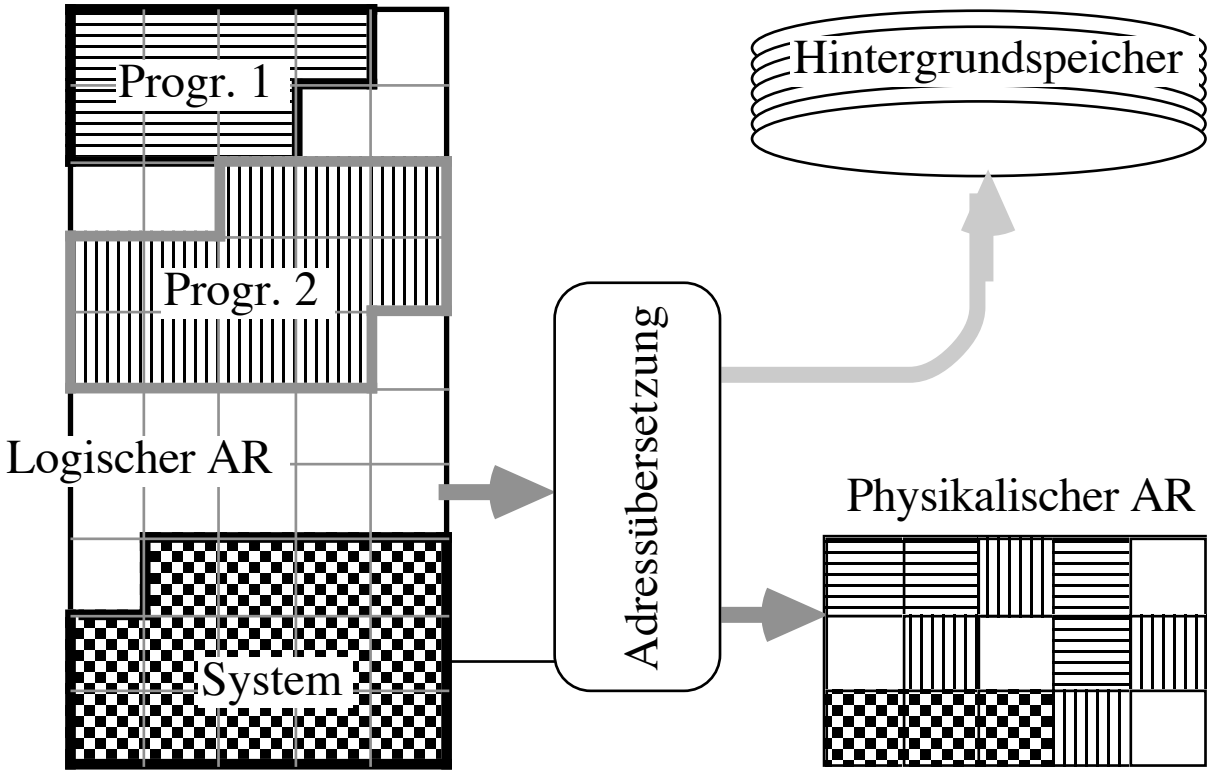


4.4 Virtueller Speicher

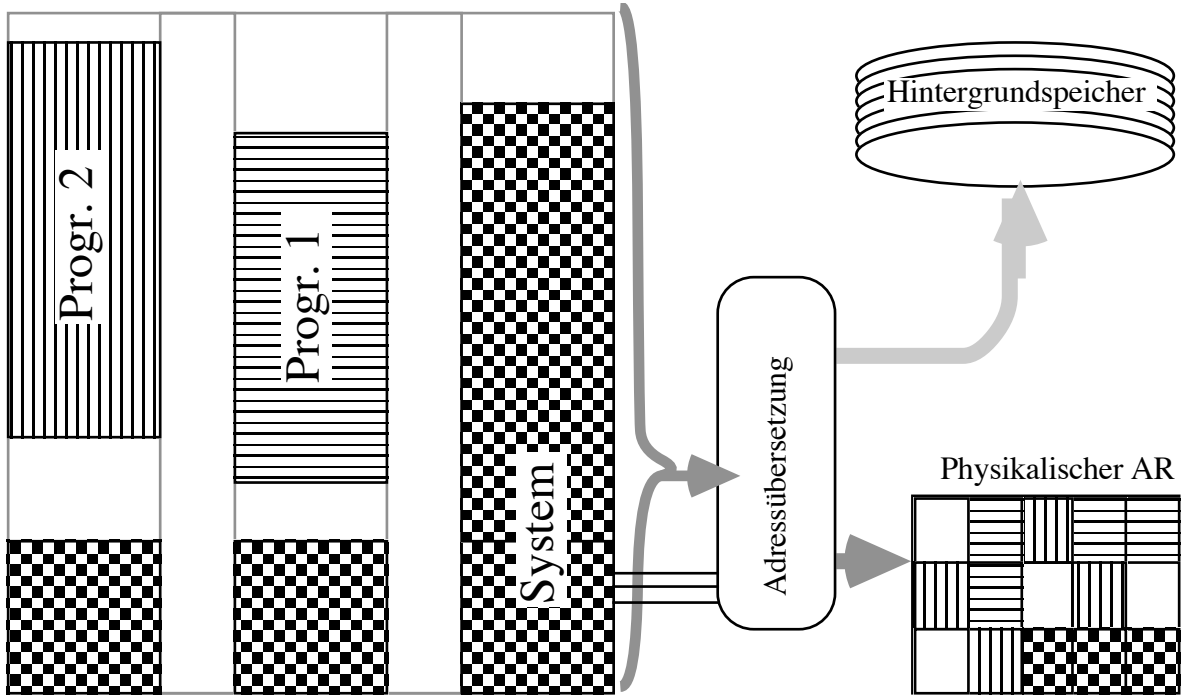
- RAM kleiner als Adressraum
  - 512 MB = 2^{29}
  - Programme wollen konzeptuell den ganzen Adressraum benutzen
  - Programme zu groß => Overlays
- Ladepunktabhängigkeit: Relozieren des Codes beim Laden
- Multitasking
  - mehrere Prozesse laufen im Zeitmultiplex
  - komplettes Auslagern (Swappen) beim Prozesswechsel zu teuer
  => gleichzeitig im RAM
  - Größenproblem verschärft
  - Schutz vor anderen Prozessen?
- Virtueller Speicher
  - Prozess (Programm) sieht nur logische Adressen
  - flacher Adressraum so groß wie Adressbits erlauben
  - Prozessor sieht virtuelle Adresse
  - Memory Management übersetzt in physische Adressen

- Physischer Adresseraum
  - RAM beschränkt
  - Teil des Prozessspeichers ausgelagert auf Hintergrundspeicher
  - z.B. Festplatte
- Seiten-Kachel Konzept
  - virtueller Speicher in Seiten eingeteilt
  - physischer Speicher in Kacheln eingeteilt
  - Verbindungs-Tabelle kann sehr groß werden
- Memory Management Unit
  - übersetzt virtuelle Adresse in physische Adresse
  - überprüft Präsenz der Seite in Kachel
- Page Fault
  - Seite liegt nicht im physischen Speicher => Interrupt
  - Betriebssystem speichert eine Kachel
  - lädt Kachel mit benötigter Seite vom Hintergrundspeicher
  - ändert Seiten/Kacheltabelle
- Auslagerungsstrategie LRU

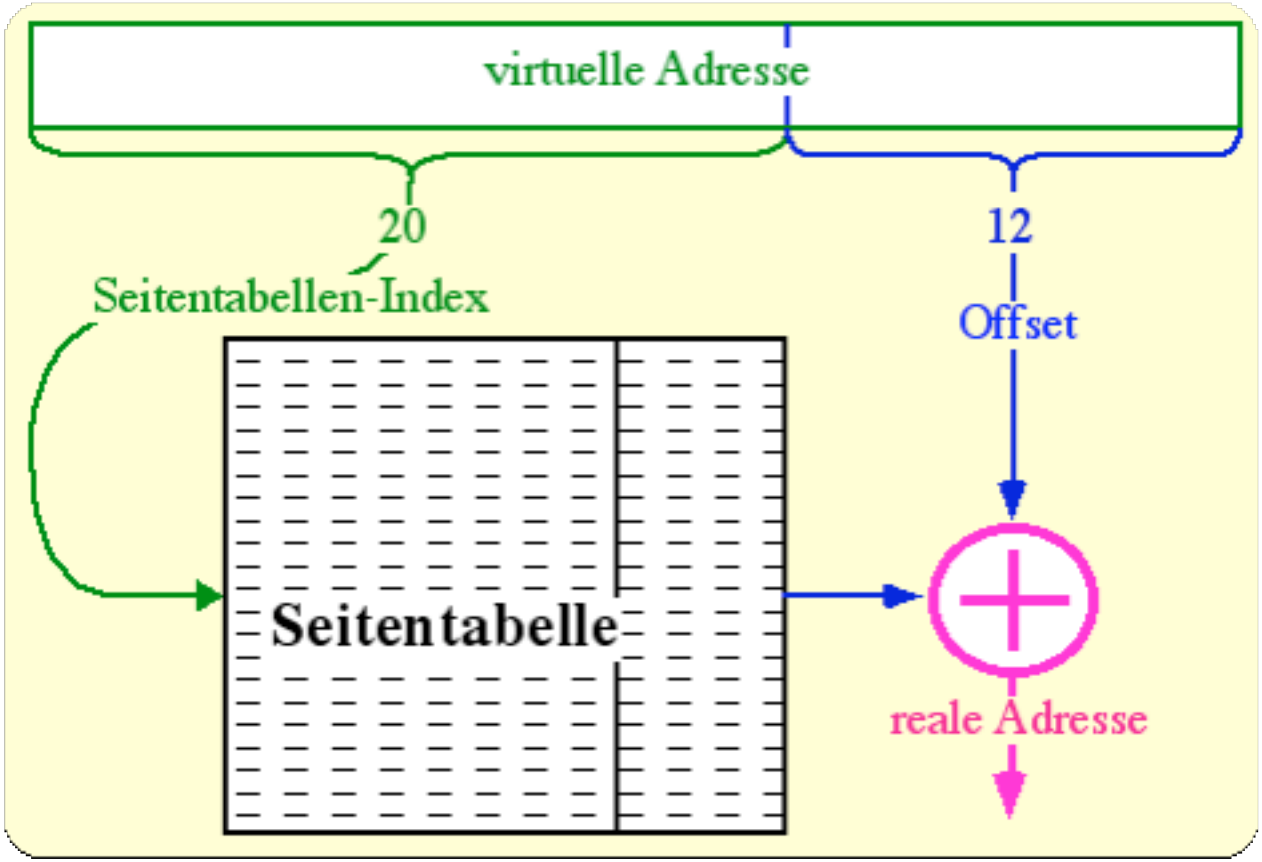
- Einfach virtualisierter Speicher
  - mehrere Programme teilen sich einen virtuellen Adressraum



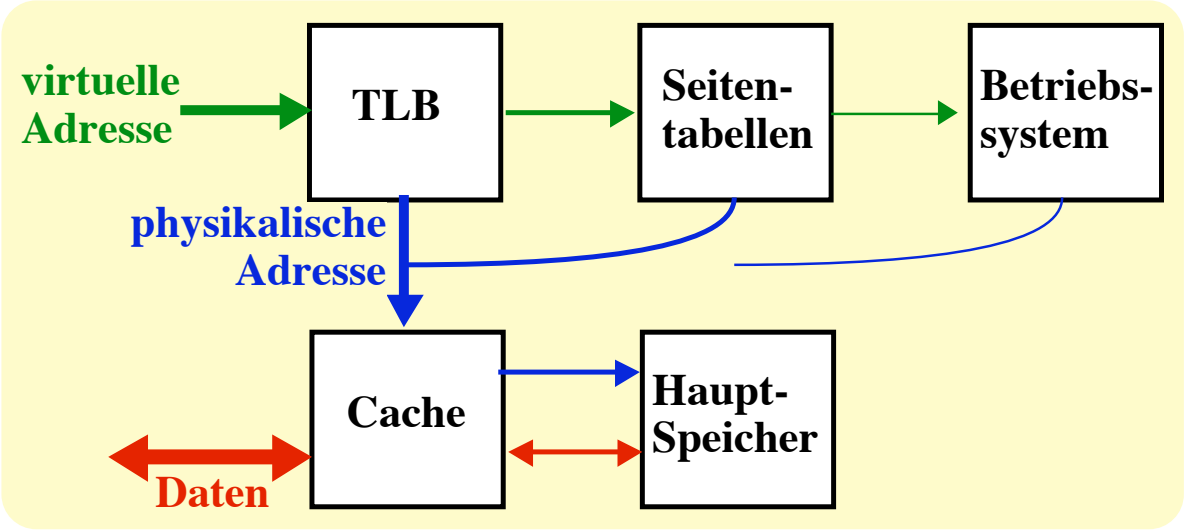
- Mehrfach virtualisierter Speicher
  - jedes Programm hat eigenen virtuellen Adressraum
  - Adressübersetzungstabelle beim Prozesswechsel umgeschalten
  - Teile des Betriebssystemadressraumes auch für Anwendungsprogramme zugreifbar
  - Hardwareeinrichtung zur Adressübersetzung.



- Abbildung virtuelle Adresse -> physische Adresse
  - virtuell: Seitenadresse + Offset
  - Seiten/Kacheltabelle: Seitenadresse -> Kacheladresse
  - physische Adresse:= Tabelle.Kacheladresse + virtuelle_Adresse.Offset
  - Translation Lookaside Buffer: Cache für Adressübersetzung

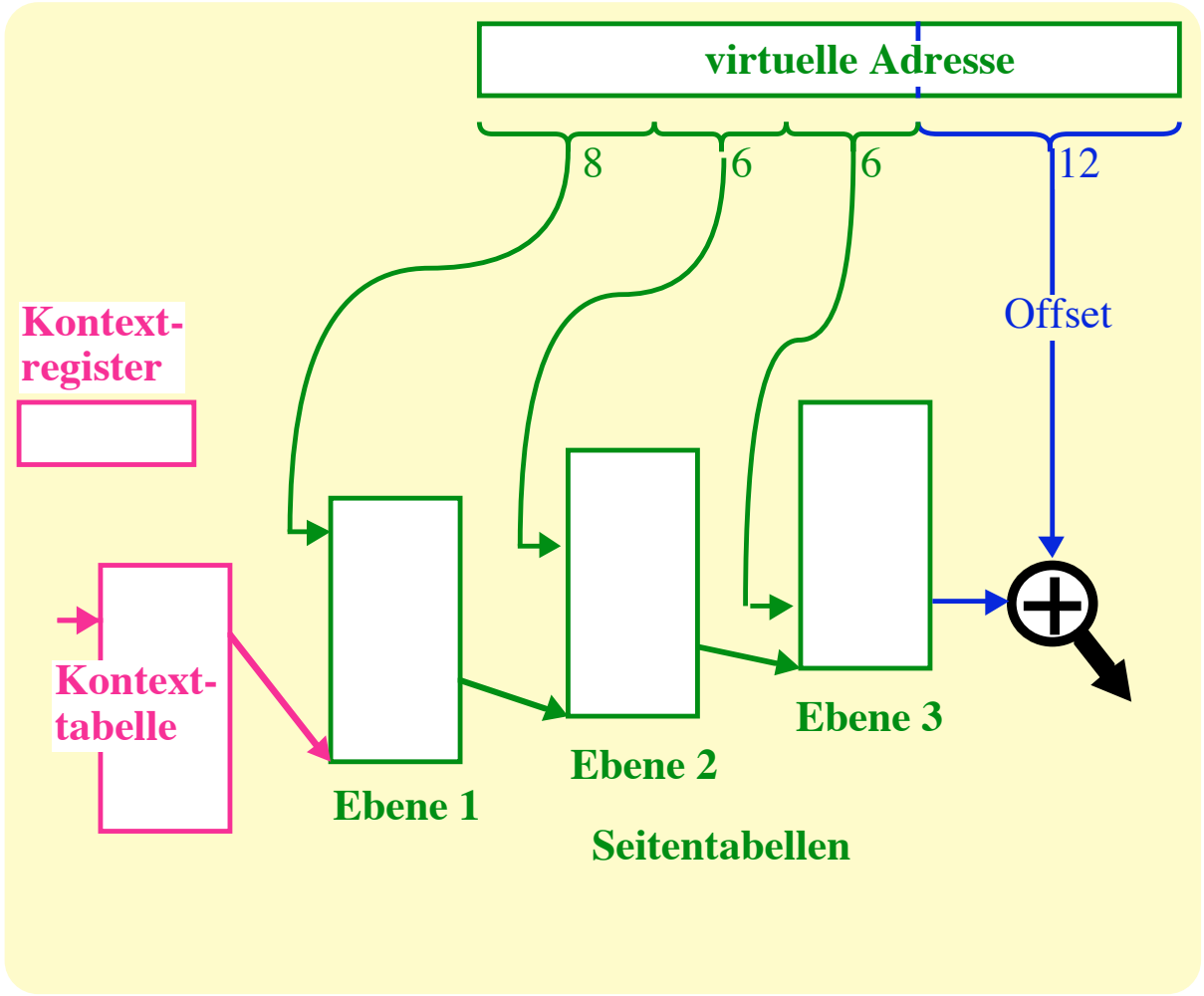


- Übersetzungspuffer (TLB)
  - TLB = "Translation Lookaside Buffer"
  - puffert schon früher übersetzte Adressen
  - kein Treffer im TLB => hardwaremässig auf die Seitentabellen im Hauptspeicher zugreifen
  - Hohe Trefferrate (Hit ratio) wichtig



- Cache benutzt evtl. physische Adressen
  - nicht sichtbar für die Software
  - physische Adresse -> Eindeutigkeit

- Mehrstufige Adressübersetzung (IA)
  - eine Übersetzungstabelle pro Prozeß
  - Adresskontext umschalten



- Beispiel Seitentabelleneintrag Intel Architektur