Rechnerstrukturen

4. CPU

4.1

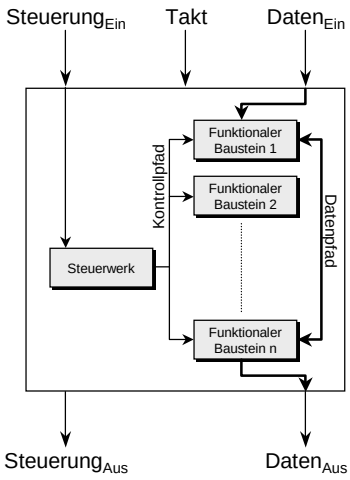
Inhalt

- Einfacher Beispielprozessor
  - Konzeptionell mit TVMUL vergleichbar
  - A. S. Tanenbaum, Structured Computer Organization, 3. Auflage, Prentice-Hall, 1990, pp. 170-209
- Grundkonzepte
  - Von Neumann-Architektur
  - Instruktionszyklus
  - Piplining
  - Superskalar
  - Instruktionssätze
  - Adressierungsarten
- Gängige CPUs
  - Intel-Line (80x86, Pentium, Pentium Pro, ...)
  - DEC Alpha
  - Motorola 680x0
  - PowerPC

4.2

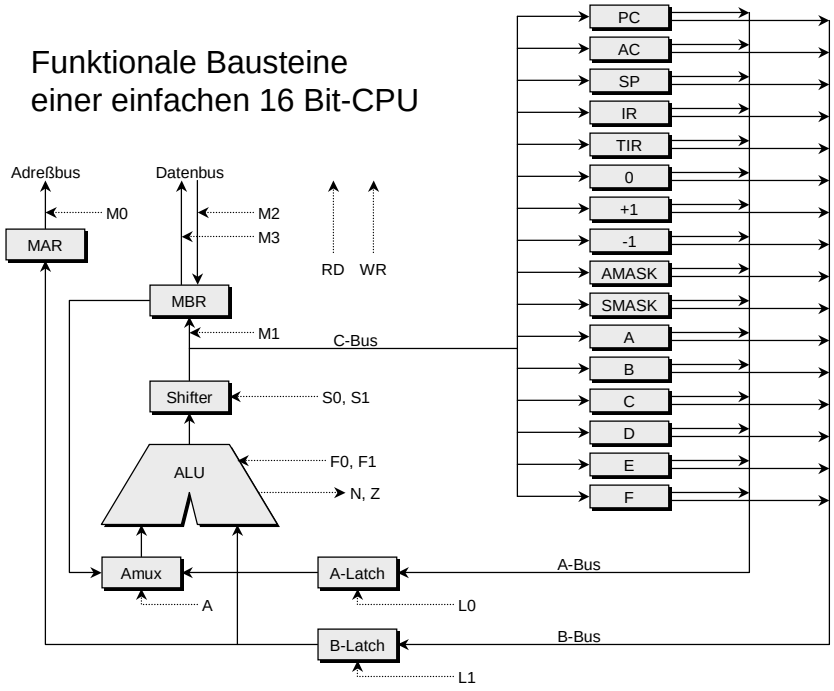
Rückblick

- Schaltnetze
  - Normalformen
  - Minimierung
- Schaltwerke
  - Latch, Flip-Flop
- Bausteine
  - Dekoder (1 aus n, ...)
  - Multiplexer / Demultiplexer
  - Register
  - Einfache Speicher
  - Shiften / Rotieren
  - Serialisieren / Parallelisieren
  - Arithmetik, Komparator
  - Endliche Automaten
- TVMUL



4.3

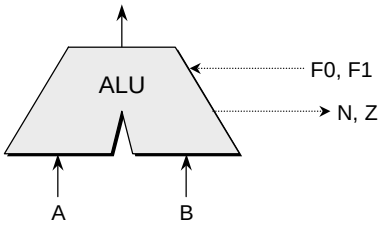
Funktionale Bausteine einer einfachen 16 Bit-CPU

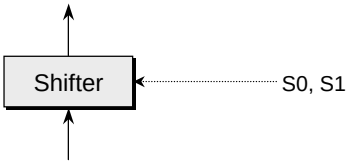


4.4

ALU und Shifter

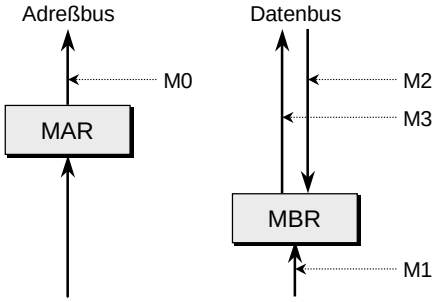
- 16 Bit-ALU
  - 4 Grundoperationen
    - A+B
    - A AND B
    - A
    - NOT A
  - Ansteuerung über F0 und F1
  - Kontrollausgänge
    - N = Negativ
    - Z = Null
- Shifter
  - 1 Bit Linksshift
  - 1 Bit Rechtsshift
  - Kein Shift





MAR und MBR

- MAR =
  Memory Address Register
  - Ansteuerung des Adreßbusses
- MBR =
  Memory Buffer Register
  - Speichert Wert, der
    - aus Speicher geladen wurde (Read)
    - in Speicher geschrieben werden soll (Write)



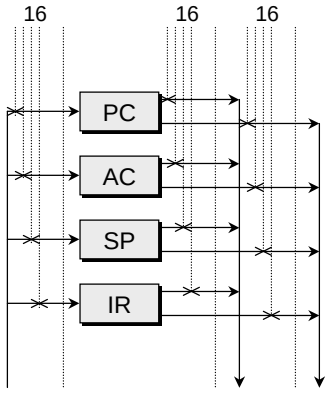
Die wesentlichen Register

- PC = Progammzähler
  - Adresse der nächsten auszuführenden Instruktion
- AC = Akkumulator
  - Zentrale Arbeitsregister
  - Übernimmt Resultat einer ALU-Operation
- SP = Stackzeiger
  - Adresse eines Datenkellers im Hauptspeicher
- Konstanten
  - 0
  - Inkrement +1
  - Dekrement -1

4.7

Kontrollpfade

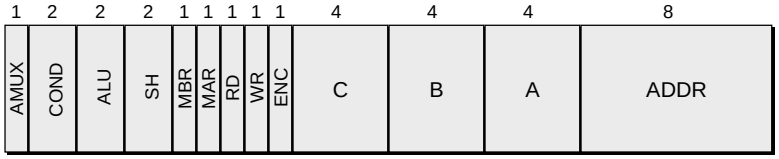
- Insgesamt 61 Leitungen
  - 16 Auswahl A-Bus
  - 16 Auswahl B-Bus
  - 16 Auswahl C-Bus
  - 2 A- und B-Latch
  - 2 ALU
  - 2 Shifter
  - 4 MAR und MBR
  - 2 Speicher lesen oder schreiben
  - 1 Amux
- Möglichkeiten der Leitungsreduktion?



4.8

Mikroinstruktionen

- Vektor aller notwendigen Kontrollpfade



- COND: Steuerung des Mikrokontrollflusses
  - 0: Sequentiell nächste Mikroinstruktion ausführen
  - 1: Sprung an angegebene Adresse, wenn N=1
  - 2: Sprung an angegebene Adresse, wenn Z=1
  - 3: Immer Sprung an angegebene Adresse

4.9

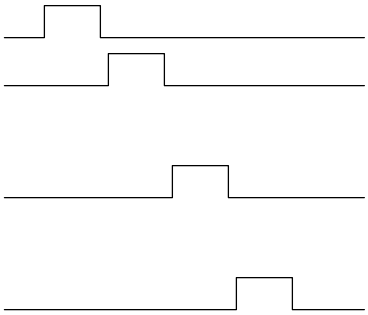
Bedeutung der einzelnen Felder

- AMUX
  - 0 = A-Latch, 1 = MBR
- ALU
  - 0 = A+B
  - 1 = A AND B
  - 2 = A
  - 3 = NOT A
- SH
  - 0 = Kein Shift
  - 1 = Shift rechts 1 Bit
  - 2 = Shift links 1 Bit
  - 3 = unbenutzt
- MBR, MAR
  - 1 = Laden
- RD
  - 1 = MBR von Datenbus laden
- WR
  - 1 = MBR auf Datenbus schreiben
- ENC
  - 0 = C-Bus nicht aktiv (z.B. nur Bestimmung von N und Z)
  - 1 = Register über C-Bus laden
- C
  - x = angegebenes Register (nur wenn ENC=1)
- B
  - x = ausgewähltes Register
- A
  - x = ausgewähltes Register

4.10

Ausführung einer Mikroinstruktion

- 4 Phasen
- Φ1: Mikroinstruktion laden
- Φ2: Operanden laden
  - Freigabe A-Latch
  - Freigabe B-Latch
- Φ3: Instruktion ausführen
  - Laden von MAR über B-Latch
- Φ4: Ergebnis speichern
  - ALU und Shifter fertig
  - Übernahme Resultat
    - ENC und C-Bus
    - MBR
  - Folgeinstruktion bestimmen

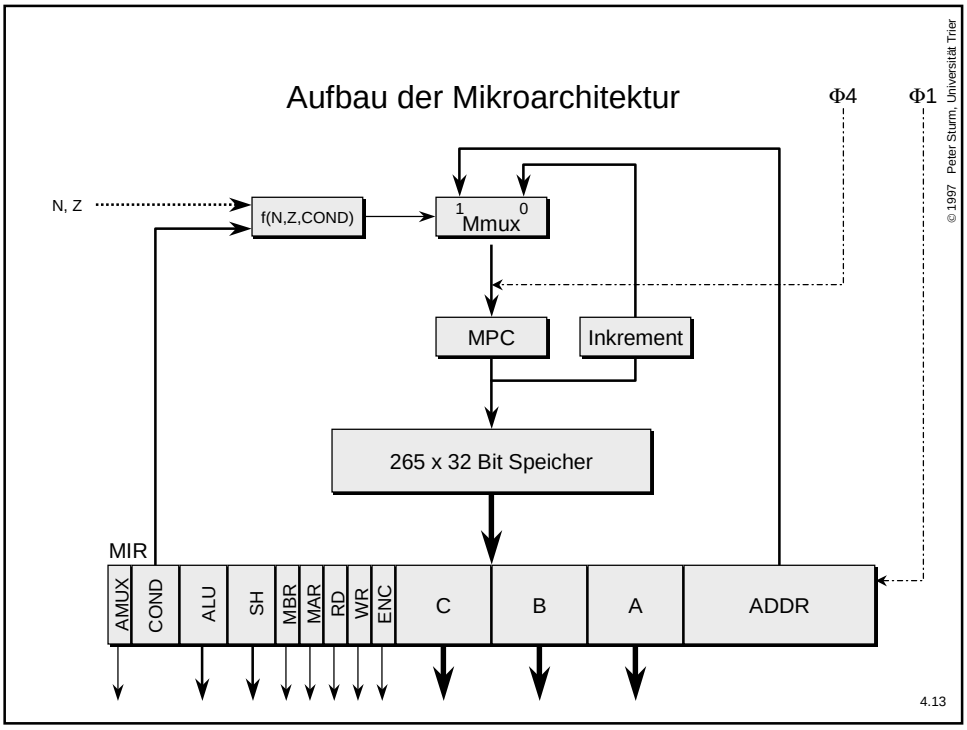


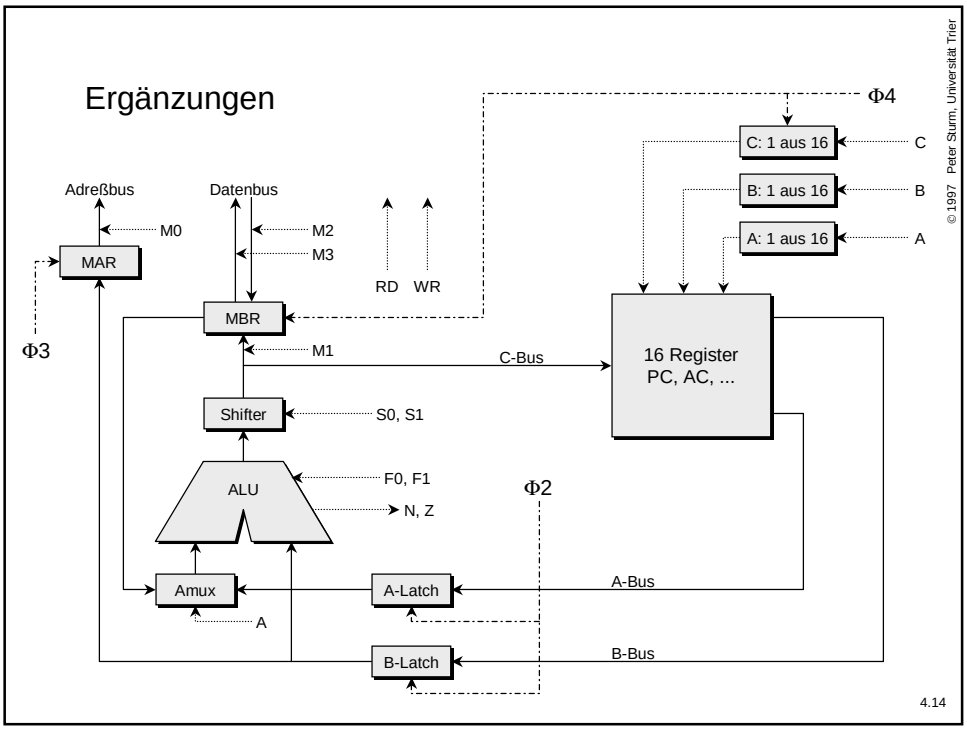
4.11

Aufbau der Mikroarchitektur?

- Welche Elemente sind notwendig?
  - Mikroinstruktion laden: Φ1
  - Operanden laden: Φ2
  - Instruktion ausführen: Φ3
  - Ergebnis speichern: Φ4

4.12


Aufbau der Mikroarchitektur
Φ4
Φ1
© 1997 Peter Sturm, Universität Trier
N, Z
f(N,Z,COND)
1
0
Mmux
MPC
Inkrement
265 x 32 Bit Speicher
MIR
AMUX
COND
ALU
SH
MBR
MAR
RD
WR
ENC
C
B
A
ADDR
4.13


Ergänzungen
Φ4
© 1997 Peter Sturm, Universität Trier
C: 1 aus 16
C
B: 1 aus 16
B
A: 1 aus 16
A
Adreßbus
Datenbus
M0
M2
M3
RD
WR
MAR
MBR
Φ3
M1
C-Bus
16 Register
PC, AC, ...
Shifter
S0, S1
ALU
F0, F1
N, Z
Φ2
Amux
A
A-Latch
A-Bus
B-Latch
B-Bus
4.14

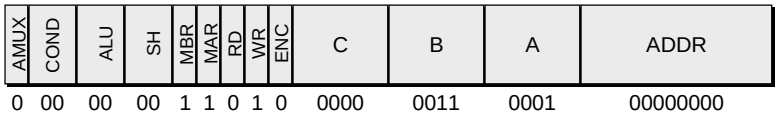
Beschreibungsmethoden

- Bitvektor = Mikroinstruktion
  - 00000001101000000011000100000000
  - 00100100000110101011011001101011
- Pseudo-Hochsprache
  - Beispiele
    - MAR := x
    - x := lshift(y+z)
    - if n then goto 27
  - 1 Zeile = 1 Zyklus?
    - Was kann maximal parallel durchgeführt werden?
  - Nur auf N und Z testen?
- Umwandlung in Bitvektor
  - 0x01a03100 = ?
  - 0x241ab66b = ?

4.15

Umwandlung

- Mikroinstruktion
  - 00000001101000000011000100000000
- Zuordnung

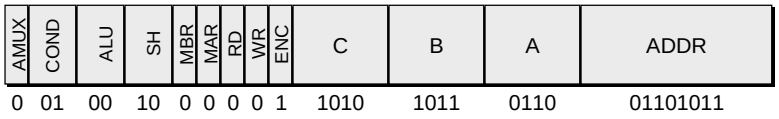


- mar := ir; mbr := ac + ir; wr

4.16

Umwandlung

- Mikroinstruktion
  - 00100100000110101011011001101011
- Zuordnung

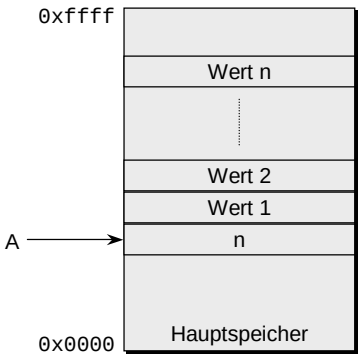


- a := LSHIFT(b+1); if N then goto 107

4.17

Mikroprogramm

- Aufgabe: Aufsummieren von Werten
  - A zeigt auf Wertetabelle im Speicher
  - Berechne Summe aller Werte in AC



4.18

Aufsummieren von n Werten

- Annahmen
  - Ausführung beginnt nach dem Einschalten ab Adresse 0 im Mikrospeicher (MPC = 0)
  - Externer Speicher ist schnell genug (nicht realistisch)
- Programm:

```
0: mar := a, rd
1: b := mbr
2: ac := 0
3: <none> := b; if Z then goto 8
4: b := b+(-1)
5: a := a+1
6: mar := a; rd
7: ac := ac+mbr; goto 3
8: // Fertig
```

4.19

Sinnvolle Abstraktionsebene?

- Zu gering
- Längere Programme?
  - Größer als Mikrospeicher
- Was ist parallel ausführbar, was nicht
  - Mit wachsender Anzahl an funktionalen Bausteinen schwieriger
  - Hohes Fehlerpotential
- Vorteil:
  - Maximale Auslastung aller Bausteine potentiell möglich
  - Hohe Effizienz
- Lösung: Automaten realisieren, der höhere Sprache versteht

4.20

Makroinstruktionen

- Nur 12 Bit-Adressen genutzt = 4096 Speicherzellen á 16 Bit
  - Vereinfachung: nur 16 Bit lange Instruktionen
- Direkte Adressierung: Ziel oder Quelle im Speicher
  - `0000xxxxxxxxxxxx` LODD ac := m[x]
  - `0001xxxxxxxxxxxx` STOD m[x] := ac
  - `0010xxxxxxxxxxxx` ADDD ac := ac + m[x]
  - `0011xxxxxxxxxxxx` SUBD ac := ac - m[x]
- Stack-relative Adressierung
  - `1000xxxxxxxxxxxx` LODL ac := m[sp+x]
  - `1001xxxxxxxxxxxx` STOL m[sp+x] := ac
  - `1010xxxxxxxxxxxx` ADDL ac := ac + m[sp+x]
  - `1011xxxxxxxxxxxx` SUBL ac := ac - m[sp+x]
- Immediate Adressierung: Konstante in Instruktion
  - `0111xxxxxxxxxxxx` LOCO ac := x

4.21

Makroinstruktionen (2)

- Kelleradressierung
  - `1111010000000000` PUSH sp--; m[sp] := ac
  - `1111011000000000` POP ac := m[sp]; sp++
  - Push Indirect
    `1111000000000000` PSHI sp--; m[sp] := m[ac]
  - Pop Indirect
    `1111001000000000` POPI m[ac] := m[sp]; sp++
- Bedingte und unbedingte Sprünge
  - `0100xxxxxxxxxxxx` JPOS if ac >= 0 then pc := x
  - `0101xxxxxxxxxxxx` JZER if ac = 0 then pc := x
  - `1100xxxxxxxxxxxx` JNEG if ac < 0 then pc := x
  - `1101xxxxxxxxxxxx` JNZE if ac != 0 then pc := x
  - `0110xxxxxxxxxxxx` JUMP pc := x

4.22

Makroinstruktionen (3)

- Unterprogramme
  - `1110xxxxxxxxxxxx` CALL — sp--; m[sp] := pc; pc := x
  - `1111100000000000` RETN — pc := m[sp]; sp++
- Diverses
  - AC und SP vertauschen
    `1111101000000000` SWAP — tmp := ac; ac := sp; sp := tmp
  - SP erhöhen
    `11111100yyyyyyyy` INSP — sp := sp + y
  - SP erniedrigen
    `11111110yyyyyyyy` DESP — sp := sp - y

Offene Fragen

- Größere Konstanten
  - LOCO kann nur Kostanten bis 4095 laden
  - Wie könnten wir 16 Bit-Konstanten laden?
- Unterprogrammsprünge
  - Bedeutung der stack-relativen Instruktionen?

Beispiel: Berechnung Fibonacci-Zahlen

- Berechnet wird Fib(n)
  - n steht an Adresse 100 im Speicher
  - Resultat Fib(n) an Adresse 101
- Rekursive Realisierung
- Unterprogrammkonvention
  - Argumente kommen auf den Keller
  - Lokale Prozedurvariablen ebenfalls auf dem Keller
  - Stack-Korrekturen vor und nach dem Unterprogrammrücksprung
- Rückgabewert einer Funktion in AC

Lösung

```
 0: LOCO 0        // Stack initialisieren
 1: SWAP
 2: LODD 100
 3: PSHI          // n kommt auf den Stack
 4: CALL 8
 5: STOD 101      // Fib(n) an Adresse 101 speichern
 6: INSP 1        // Argument n vom Keller nehmen
 7:               // Programmende

 8: DESP 1        // Speicher für lokale Variable f anlegen
 9: LOCO -2
10: ADDL 2        // AC := n-2
11: JPOS          // if n>1 then goto
12: LOCO 1
13: STOL 0        // f := 1
14: JUMP 25       // Rücksprung vorbereiten
15: LOCO -1
16: ADDL 2        // AC := n-1
17: CALL 8        // AC := Fib(n-1)
18: INSP 1
19: STOL 0        // f := AC
```

```
20: LOCO -2
21: ADDL 2        // AC := n-2
22: CALL 8        // AC := Fib(n-2)
23: INSP 1
24: ADDL 0        // AC := Fib(n-2) + Fib(n-1)
25: INSP 1        // Lokale Variable f freigeben
26: RETN
```


Realisierung

- Annahmen
  - MPC beginnt nach dem Einschalten mit Adresse 0
  - Konstanten sind initialisiert
  - Erste ausgeführte Makroinstruktion an Speicheradresse 0
  - Zugriff zum Speicher kostet 2 Zyklen
- Zusatzregister
  - IR = Instruction Register
    Speichert die aktuell ausgeführte Makroinstruktion
  - TIR = Temporäre Kopie von IR (falls notwendig)
  - AMASK = Address Mask
    := `0x0fff` (Maskieren des Opcodes)
  - SMASK = Stack Mask
    := `0x00ff` (Maskieren des Opcodes bei INSP DESP)
  - Register A-F stehen dem Mikroprogrammierer zur Verfügung
    (keine sichtbaren Register)

Opcodes

- **CCCCxxxxxxxxxxxx**
  - 0000 LODD
  - 0001 STOD
  - 0010 ADDD
  - 0011 SUBD
  - 0100 JPOS
  - 0101 JZER
  - 0110 JUMP
  - 0111 LOCO
  - 1000 LODL
  - 1001 STOL
  - 1010 ADDL
  - 1011 SUBL
  - 1100 JNEG
  - 1101 JNZE
  - 1110 CALL
- **1111CCC000000000**
  - 1111 000 PSHI
  - 1111 001 POPI
  - 1111 010 PUSH
  - 1111 011 POP
  - 1111 100 RET
  - 1111 101 SWAP
- **1111CCC0yyyyyyyy**
  - 1111 110 INSP
  - 1111 111 DESP

Lösung Tanenbaum, pp. 190f

```
LOOP  0: mar := pc; rd
      1: pc := pc+1; rd
      2: ir := mbr; if n then goto 28
      3: tir := lshift(ir+ir); if n then goto 19
      4: tir := lshift(tir); if n then goto 11
      5: alu := tir; if n then goto 9

LODD  6: mar := ir; rd
      7: rd
      8: ac := mbr; goto LOOP

STOD  9: mar := ir; mbr := ac; wr
     10: wr; goto LOOP

     11: alu := tir; if n then goto 15

ADDD 12: mar := ir; rd
     13: rd
     14: ac := mbr+ac; goto LOOP
```

```
SUBD 15: mar := ir; rd
     16: ac := ac+1; rd              // x-y = x+1+inv(y)
     17: a := inv(mbr)
     18: ac := ac+a; goto LOOP

     19: tir := lshift(tir); if n then goto 25
     20: alu := tir; if n then goto 23

JPOS 21: alu := ac; if n then goto LOOP
     22: pc := band(ir,amask); goto LOOP

JZER 23: alu := ac; if z then goto 22
     24: goto LOOP

     25: alu := tir; if n then goto 27

JUMP 26: pc := band(ir,amask); goto LOOP

LOCO 27: ac := band(ir,amask); goto LOOP

     28: tir := lshift(ir+ir); if n then goto 40
     29: tir := lshift(tir); if n then goto 35
     30: alu := tir; if n then goto 33
```


4.31

```
LODL 31: a := ir+sp
     32: mar := a; rd; goto 7

STOL 33: a := ir+sp
     34: mar := a; mbr := ac; wr; goto 10

     35: alu := tir; if n then goto 38

ADDL 36: a := ir+sp
     37: mar := a; rd; goto 13

SUBL 38: a := ir+sp
     39: mar := a; rd; goto 16

     40: tir := lshift(tir); if n then goto 46
     41: alu := tir; if n then goto 44

JNEG 42: alu := ac; if n then goto 22
     43: goto LOOP

JNZE 44: alu := ac; if z then goto LOOP
     45: pc := band(ir,amask); goto LOOP
```


4.32

```
     46: tir := lshift(tir); if n then goto 50

CALL 47: sp := sp+(-1)
     48: mar := sp; mbr := pc; wr
     49: pc := band(ir,amask); wr; goto LOOP

     50: tir := lshift(tir); if n then goto 65
     51: tir := lshift(tir); if n then goto 59
     52: alu := tir; if n then goto 56

PSHI 53: mar := ac; rd
     54: sp := sp+(-1); rd
     55: mar := sp; wr; goto 10

POPI 56: mar := sp; sp := sp+1; rd
     57: rd
     58: mar := ac; wr; goto 10

     59: alu := tir; if n then goto 62

PUSH 60: sp := sp+(-1)
     61: mar := sp; mbr := ac; wr; goto 10
```


4.33

```
POP  62: mar := sp; sp := sp+1; rd
     63: rd
     64: ac := mbr; goto LOOP

     65: tir := lshift(tir); if n then goto 73
     66: alu := tir; if n then goto 70

RETN 67: mar := sp; sp := sp+1; rd
     68: rd
     69: pc := mbr; goto LOOP

SWAP 70: a := ac
     71: ac := sp
     72: sp := a; goto LOOP

     73: alu := tir; if n then goto 76

INSP 74: a := band(ir,smask)
     75: sp := sp+a; goto LOOP

DESP 76: a := band(ir,smask)
     77: a := inv(a)
     78: a := a+1; goto 75
```


4.34

Problem

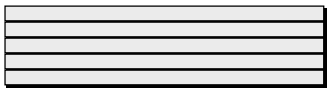
- Code-Fragment Fibonacci:

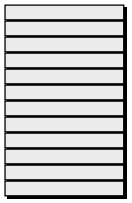
```
    ...
 8: DESP 1
 9: LOCO -2
10: ADDL 2
11: JPOS
12: LOCO 1
13: STOL 0
14: JUMP 25
15: LOCO -1
16: ADDL 2
    ...
```

- Zwei Befehle funktionieren nicht wie gewünscht?

4.35

Horizontale vs. vertikale Mikroprogrammierung





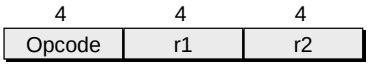
- Breite Mikroinstruktionen
- Vorteile
  - Alle funktionalen Bausteine ansprechbar
  - Maximale Parallelität möglich
  - Günstiges Verhältnis Mikro- zu Makroinstruktionen
- Nachteil
  - Großer Flächenbedarf

- Schmale Mikroinstruktionen
- Vorteile
  - Einschränkungen in der möglichen Parallelarbeit
  - Ungünstiges Verhältnis Mikro- zu Makroinstruktionen
  - Kleiner Flächenbedarf
- Nachteil
  - Langsame Makroinstruktionen

4.36

Beispiel vertikale Programmierung

- Verdrahtete Mikroinstruktionen




```
0000  ADD    r1 := r1+r2
0001  AND    r1 := r1 AND r2
0010  MOVE   r1 := r2
0011  COMPL  r1 := inv(r2)
0100  LSHIFT r1 := lshift(r2)
0101  RSHIFT r1 := rshift(r2)
0110  GETMBR r1 := mbr
0111  TEST   if r2<0 then n:=true; if r2=0 then z:=true
1000  BEGRD  mar := r1; rd                  // Begin Read
1001  BEGWR  mar := r1; mbr := r2; wr       // Begin Write
1010  CONRD  rd                             // Continue RD
1011  CONWR  wr                             // Continue WR
1100
1101  NJUMP  if n then goto r
1110  ZJUMP  if z then goto r
1111  UJUMP  goto r
```


4.37

Leistungssteigernde Maßnahmen im Prozessor

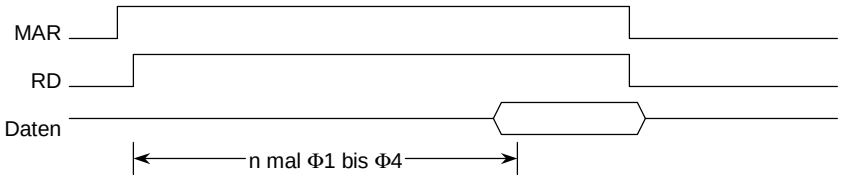
- Taktdauer von $\Phi 1$ bis $\Phi 4$ besser anpassen
  - Feinere Grundtakte
  - Ziel: ausgewogene Architektur
- Pipelining
  - Instruction Pipelining
    - Verschränkte Ausführung der vier Phasen
  - Komplexe Operationen wie z.B. Addition, Multiplikation, ...
    - Unterteilung der Langläufer
    - Mehrere gleiche Instruktionen taktversetzt ausführen
- Funktionale Bausteine mehrfach vorhanden
  - Superskalar
    (mehrere aufeinanderfolgende Instruktionen gleichzeitig fertig)
- Komplexes Steuerwerk
  - Datenabhängigkeiten beachten

4.38

Probleme

- Instruction Pipelining
  - Setzt stark sequentielle Programmausführung voraus
  - Sprungbefehle kritisch (30% der Instruktionen)
  - Lösungsansätze?
- Umordnen erhöht Parallelisierungspotential
- Parallele Instruktionsausführung
  - Erkennen von Ressourcenkonflikten
    - Nur 2 Addierer aber 3 Additionen
    - Belegung von internen Bussen
  - Erkennen von Datenabhängigkeiten
    - Beispiel:
      - STOD 700
      - LOCO 2
      - ADDD 700

Zykluszeit vs. Speicherzugriffszeit



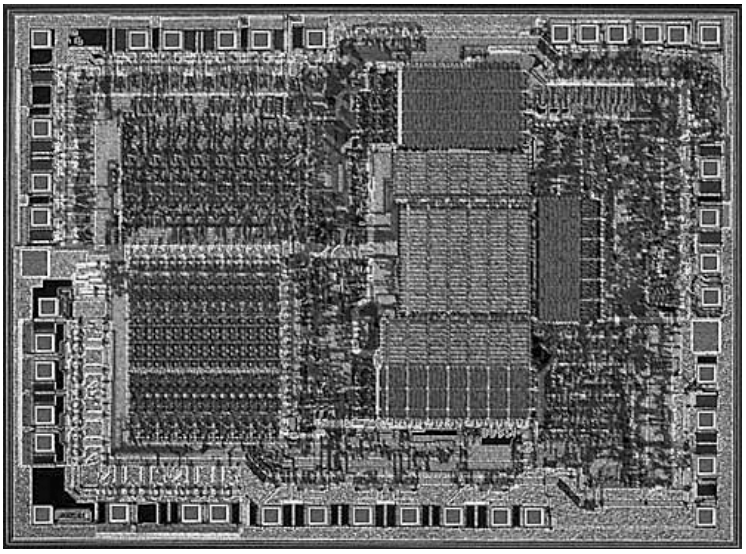
- Wie groß ist n?
- Ansatzpunkte
  - Prozessor:
    - Genügend Arbeit zur Überbrückung der Wartezeiten beschaffen
    - Durch Pipelining und Parallelarbeit wächst Flaschenhals
  - Speicher:
    - Zugriffszeit verkürzen, d.h. n reduzieren
    - Mehr Register
    - Schnelle Zwischenspeicher (Cache)

Die Intel-Linie

- Erster Mikroprozessor 4004 (4 Bit)
  - Steuerung von elektronischen Kassen
- Durchbruch 8 Bit-Prozessoren 8080 und 8085
- Schnelle Generationswechsel 8-, 16- und 32-Bit
- Aktuell
  - 32 Bit-CISC-Prozessoren
    - Pentium (80586)
    - Pentium Pro
    - Pentium II
  - 64 Bit-RISC-Prozessor
    - Merced (98/99)

4.41

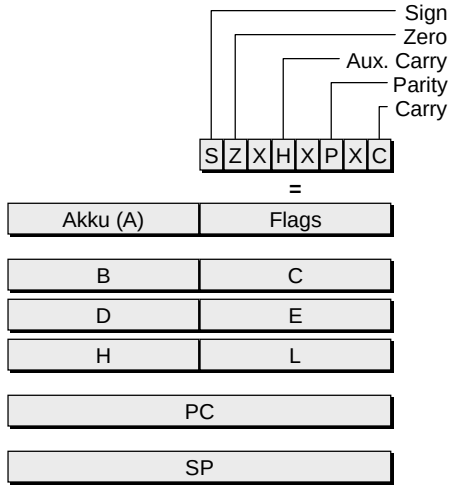
Intel 8085



4.42

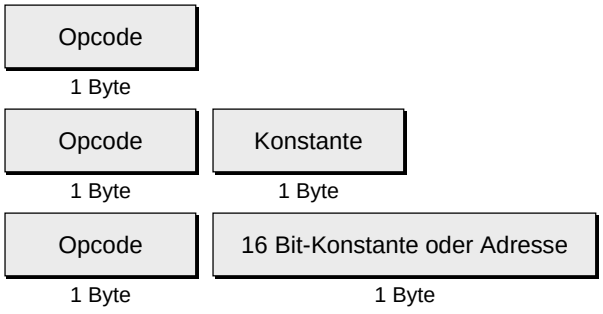
Sichtbare Registerstruktur

- 16 Bit Adressen
  - 64 Kbyte Speicher adressierbar
- 8 Bit-Register
  - Akkumulator
  - Flags
  - B, C, D, E, H, L
- 16 Bit-Register
  - Programmzähler (PC)
  - Stackpointer (SP)
  - PSW = Akku + Flags
  - BC, DE, HL
- Viele Spezialinstruktionen
  - Nur bestimmte Register verwendbar
  - typisch CISC



4.43

Instruktionsformate



- Little Endian
  - Reihenfolge der einzelnen Bytes bei Mehr-Byte-Größen
  - Beispiel `0x34af` ab Adresse `0x2000`
    - `0x2000`: `0xaf`
    - `0x2001`: `0x34`

4.44

Befehlsgruppen

- Datentransfer
  - Register-Register
  - Register-Speicher
  - Speicher-Register
  - Immediate
- Arithmetisch-logische Befehle
  - Addition mit und ohne Carry
  - Subtraktion mit und ohne Borrow
  - Inkrement und Dekrement
  - Logisch Und, Oder, Exklusiv-Oder
  - Vergleich
  - Shiften und Rotieren
- Sprungbefehle
- Unterprogrammbefehle
- Kelleroperationen
- Spezialbefehle

Datentransfer

- 8 Bit Register-Register
  - **`MOV r1, r2`**
    - r1 := r2, mit r1, r2 aus { A, B, C, D, E, H, L }
- 16 Bit Register-Register
  - **`XCHG`**
    - tmp16 := DE
DE = HL
HL = tmp16
  - **`PCHL`**
    - PC := HL
  - **`SPHL`**
    - SP := HL
- 8 Bit Register-Speicher
  - **`MOV r, M`**
    - r := m[HL], mit r aus { A, B, C, D, E, H, L }
  - **`LDAX rr`**
    - A := m[rr], mit rr aus { BC, DE, HL }
  - **`STA adr16`**
    - m[adr16] := A
- 16 Bit Register-Speicher
  - **`LHLD adr16`**
    - L := m[adr16]
H := m[adr16+1]

Datentransfer (2)

- 8 Bit Speicher-Register
  - **`MOV M, r`**
    - m[HL] := r, mit r aus { A, B, C, D, E, H, L }
  - **`STAX rr`**
    - m[rr] := A, mit rr aus { BC, DE, HL }
  - **`STA adr16`**
    - m[adr16] := A
- 16 Bit Speicher-Register
  - **`SHLD adr16`**
    - m[adr16] := L
m[adr16+1] := H
- 8 Bit Immediate
  - **`MVI r, byte`**
    - r := byte, mit r aus { A, B, C, D, E, H, L, M }
- 16 Bit Immediate
  - **`LXI rr, dbyte`**
    - rr := dbyte, mit r aus { BC, DE, HL, SP }

Arithmetik

- 8 Bit Addition
  - **`ADD r`**
    - A := A + r, r aus { A, B, C, D, E, H, L, M }
    - Flags: C, Z, S, P, AC
  - **`ADC r`**
    - A := A + r + Carry
    - Flags: C, Z, S, P, AC
  - **`ADI byte`**
    - A := A + byte
    - Flags: C, Z, S, P, AC
  - **`ACI byte`**
    - A := A + byte + Carry
    - Flags: C, Z, S, P, AC
- Analog Subtraktion
  - SUB, SBB, SUI, SBI
- 16 Bit Addition
  - **`DAD rr`**
    - HL := HL + rr, rr aus { BC, DE, HL, SP }
    - Flags: C
- 8 Bit Inkrement / Dekrement
  - **`INR r`**
    - r := r+1, r aus { A, B, C, D, E, H, L, M }
    - Flags: Z, S, P, AC
  - **`DCR r`**
    - r := r-1, r aus { A, B, C, D, E, H, L, M }
    - Flags: Z, S, P, AC

Arithmetik (2)

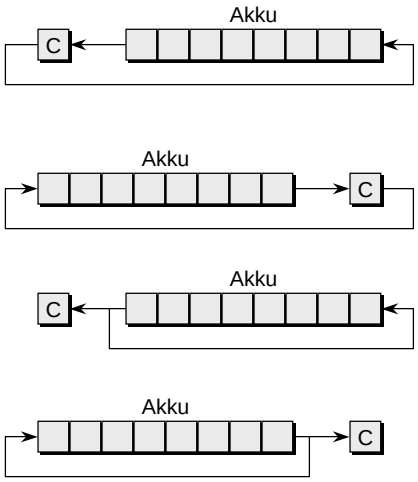
- 16 Bit Inkrement / Dekrement
  - **INX rr**
    - rr := rr+1, rr aus { BC, DE, HL, SP }
  - **DCX rr**
    - rr := rr-1, rr aus
    - { BC, DE, HL, SP }
- Vergleichen
  - **CMP r**
    - tmp8 := A - r
    - Flags: C, Z, S, P, AC
  - **CPI byte**
    - tmp8 := A - byte
    - Flags: C, Z, S, P, AC
- Carry setzen
  - **STC**
    - Flag C := 1
- Carry invertieren
  - **CMC**
    - Flag C := inv(Flag C)
- BCD-Korrektur
  - **DAA**
- Akku invertieren
  - **CMA**
    - A := inv(A)

Logik

- 8 Bit Logisches UND
  - **ANA r**
    - A := A AND r, r aus { A, B, C, D, E, H, L, M }
    - Flags: C:=0, Z, S, P, AC
  - **ANI byte**
    - A := A AND byte
    - Flags: C:=0, Z, S, P, AC
- Analog Logisches ODER
  - **ORA r**
    ORI byte
- Analog Logisches Exklusiv-ODER
  - **XRA r**
    XRI byte

Rotieren

- RAL
  - tmp1 := Carry
    Carry := A7
    A7 := A6, ..., A1 := A0
    A0 := tmp1
- RAR
  - tmp1 := Carry
    Carry := A0
    A0 := A1, ..., A6 := A7
    A7 := tmp1
- RLC
  - Carry := A7
    A7 := A6, ..., A1 := A0
    A0 := Carry
- RRC
  - Carry := A0
    A0 := A1, ..., A6 := A7
    A7 := Carry



4.51

Sprünge und Unterprogramme

- Unbedingter Sprung
  - **`JMP adr16`**
- Bedingter Sprung
  - **`Jcc adr16`**
  - cc =
    - NZ: Not Zero
    - Z: Zero
    - NC: No Carry
    - C: Carry
    - PO: Parity Odd
    - PE: Parity Even
    - P: Positive
    - M: Minus (Negative)
- Indirekter Sprung
  - **`PCHL`**
    - PC := HL
- Unterprogrammsprung
  - **`CALL adr16`**
  - **`Ccc adr16`**
  - SP := SP-1
    m[SP] := PC_{MSB}
    SP := SP-1
    m[SP] := PC_{LSB}
    PC := adr16
- Unterprogrammrücksprung
  - **`RET`**
  - **`Rcc`**
  - PC_{LSB} := m[SP]
    SP := SP+1
    PC_{MSB} := m[SP]
    SP := SP+1

4.52

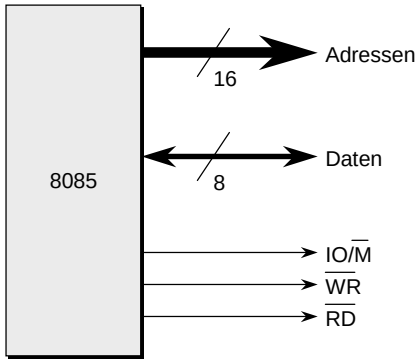
Kellerbefehle

- Push
  - **PUSH rr**
    - rr aus { BC, DE, HL, PSW }
    - SP := SP-1, m[SP] := rr_{MSB}, SP := SP-1, m[SP] := rr_{LSB}
- Pop
  - **POP rr**
    - rr aus { BC, DE, HL, PSW }
    - rr_{LSB} := m[SP], SP := SP+1, rr_{MSB} := m[SP], SP := SP+1
- Spezialbefehle
  - **XTHL**
    - tmp8 := L, L := m[SP], m[SP] := tmp8
      tmp8 := H, H := m[SP+1], m[SP+1] := tmp8
  - **SPHL**
    - SP := HL // Der Vollständigkeit halber

4.53

Ein- und Ausgabe

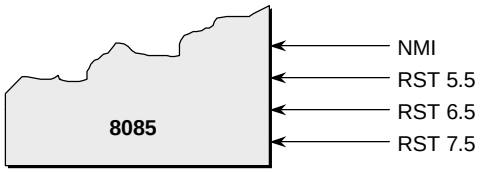
- 256 E/A-Adressen
- Ausgabe
  - **OUT adr8**
    - EA[adr8] := A
- Eingabe
  - **IN adr8**
    - A := EA[adr8]



4.54

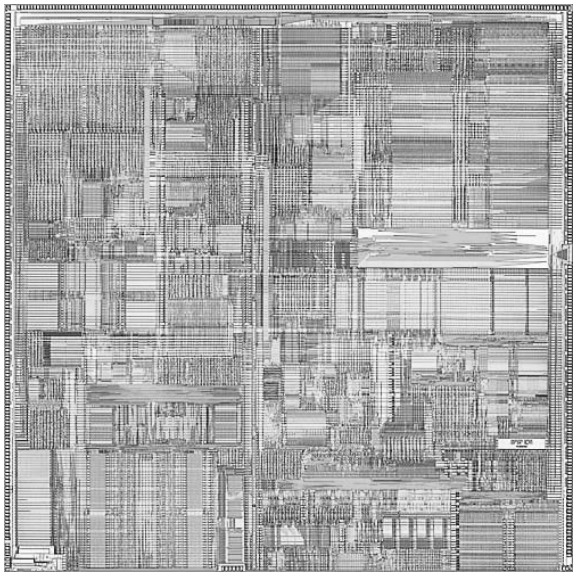
Interrupt-Behandlung

- Software-Interrupt
  - **RST x**
    - SP := SP-1
      m[SP] := PC_{MSB}
      SP := SP-1
      m[SP] := PC_{LSB}
      PC := m[x*8]
- Hardware-Interrupts
  - 4 Stück
  - RST 4.5 (NMI), 5.5, 6.5, 7.5
- Interruptmaske
  - **RIM**
    - A := IM
  - **SIM**
    - IM := A
- Interrupts zulassen
  - **EI**
- Interrupts sperren
  - **DI**
- Misc
  - **HLT**
  - **NOP**



4.55

Die Pentium-Familie (80586)



4.56

Beispiele
Pentium, MMX, Pro
Power PC 750
Alpha 21164
...
siehe Ordner CPUs

Übersicht (aus iX 10/97, pp. 158)

	Intel				AMD		Cyrix
	8086	Pentium	P MMX	P Pro	K5	K6	M2
Typ	CISC	CISC	CISC	CISC	CISC	CISC	CISC
Jahr	78	95	97	96	97	97	97
Bits	16	32	32	32	32	32	32
Takt[1)]	10	133	133	200	117	233	
Units/Issue[2)]	1/1	3/2	5/2	7/3	6/4	7/6	3/2
Pipestate[3)]	1/1/na	5/na/8	6/na/8	14/14/16	5/5/5	6/7/7	7/7/na
Cache[4)]	-	8/8	16/16	8/8	16/8	32/32	64

1) Maximale Taktrate zum Zeitpunkt der Einführung
2) Anzahl Verarbeitungseinheiten / Davon parallelisierbar
3) Stufen der Integer- / Load- and Store- / Floating Point-Pipeline
4) Cache bzw. Befehlscache / Datencache

Übersicht (cont.)

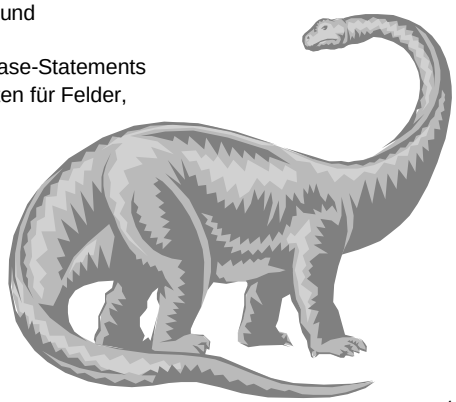
	Motorola			SUN	DEC	
	68040	PPC 604e	PPC 620	Ultra II	21164a	21264
Typ	CISC	RISC	RISC	RISC	RISC	RISC
Jahr	89	97	96	97	96	97
Bits	32	32	64	64	64	64
Takt[1)]	25	350	200	300	600	600
Units/Issue[2)]	2/1	6/4	6/4	9/4	4/4	6/4
Pipestate[3)]	6/na/6	4/5/6	4/5/6	9/9/9	7/7/9	7/7/10
Cache[4)]	4/4	32/32	64/64	16/16	8/8	64/64

1) Maximale Taktrate zum Zeitpunkt der Einführung
2) Anzahl Verarbeitungseinheiten / Davon parallelisierbar
3) Stufen der Integer- / Load- and Store- / Floating Point-Pipeline
4) Cache bzw. Befehlscache / Datencache

4.59

CISC

- Complex Instruction Set Computer
  - Viele, z.T über 200, Instruktionen
  - Instruktionen unterschiedlicher Länge
- Gründe
  - "Semantic Gap" zwischen Hoch- und Maschinensprache
    - Spezielle Instruktionen für Case-Statements
    - Besondere Adressierungsarten für Felder, Strukturen, Listen, ...
  - Langsame Speicher
  - Hardware teuer, Software billig



4.60

Die wilden 70er

- Alles wurde anders:
  - Hardware wurde billig
  - Software extrem teuer
- Steigende Integrationsdichte
  - Was tun mit dem vielen Platz?
- Großer Leistungshunger neuer Anwendungen
  - Mikroprogramme brauchen Zeit
- Beobachtung: Prozessoren viel zu komplex
  - Compiler nutzen nur einen Bruchteil der vorhandenen Instruktionen und Adressierungsarten
- Die RISC-Bewegung
  - **“Make the Common Case Fast”**

4.61

CISC vs. RISC

Komplexe Instruktionen, die viele Zyklen benötigen	Einfache Instruktionen, die alle nur 1 Zyklus benötigen
(Fast) jede Instruktion kann auf Speicheroperanden zugreifen	Nur spezielle Instruktionen referenzieren Speicher
Kein oder wenig Pipelining	Viel Pipelining
Instruktionen werden von Mikro-programm interpretiert	Instruktionen werden von HW direkt ausgeführt
Variables Instruktionsformat	Festes Instruktionsformat
Viele Instruktionen und Modi	Wenig Instruktionen und Modi
Komplexes Mikroprogramm	Komplexer Compiler
Ein Registersatz	Mehrere Registersätze

4.62

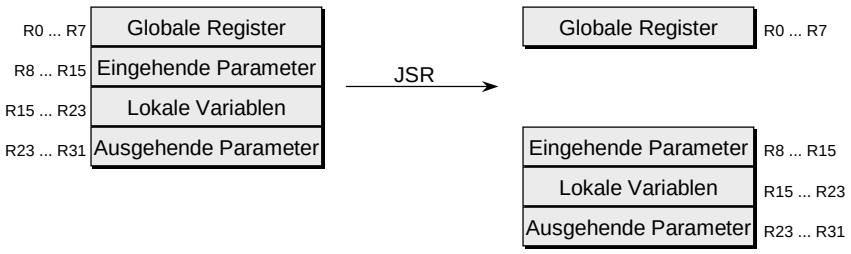
Bemerkungen

- 1 Instruktion pro Zyklus
  - RISC-Instruktion vergleichbar Mikroinstruktion
  - Keine RISC-Instruktion für längere Operationen (z.B. Multiplikation)
- Spezielle LOAD/STORE-Instruktionen für Speicherzugriff
  - Allgemein 1 Instruktion pro Zyklus nicht machbar
  - Problem, LOAD und STORE nicht in einem Zyklus fertig
- Pipelining
  - Abschwächung von Regel 1: 1 Instruktion pro Zyklus starten
  - Delayed Load (Store)
  - Delayed Branch
- Kein Mikroprogramm
  - Nutzung wachsender Chipflächen
  - Maximale Performanz
  - => Fester Instruktionsformat notwendig

4.63

Überlappende Registerfenster

- Besondere Unterstützung für Unterprogramme
- Empirische Daten
  - Wenige Argumente pro Prozeduraufruf
  - Wenige lokale Variablen
  - Selten tiefere Verschachtelung
- Ein sehr großer Registersatz (128 und mehr Register)
  - Besondere Sichtbarkeitsregeln



4.64