Rechnerstrukturen

4. CPU

4.1

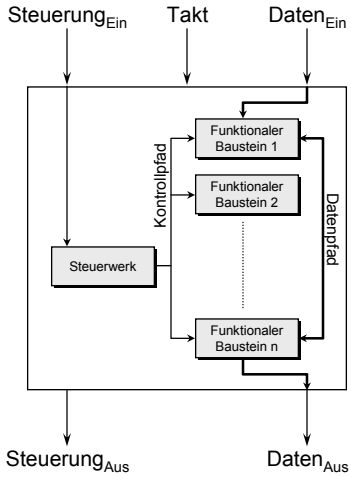
Inhalt

- 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

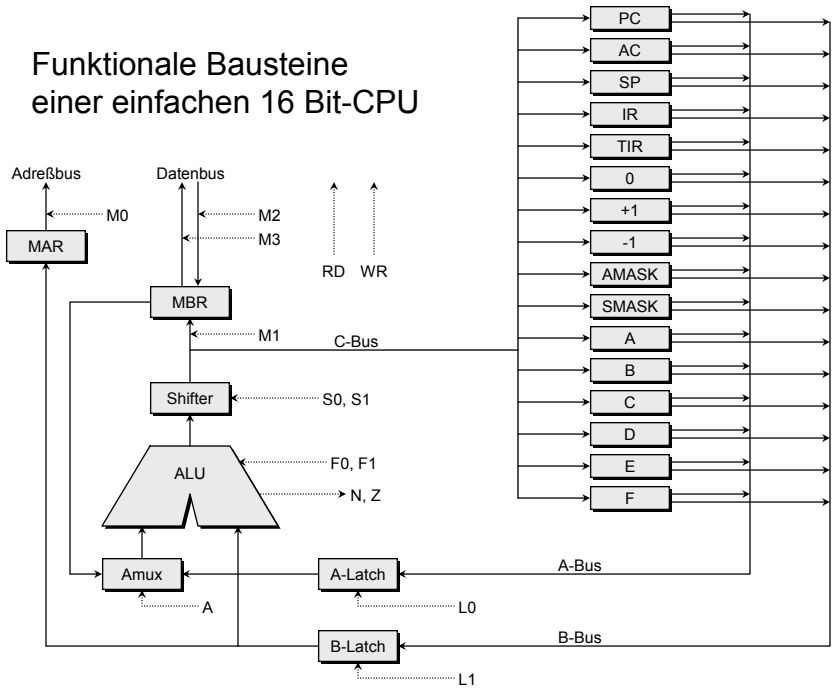
Rückblick

- 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

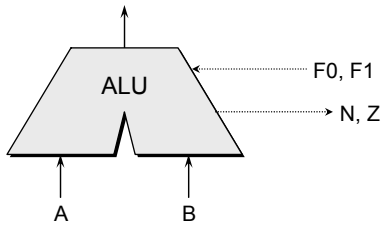
Funktionale Bausteine einer einfachen 16 Bit-CPU

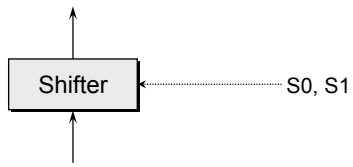


4.4

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

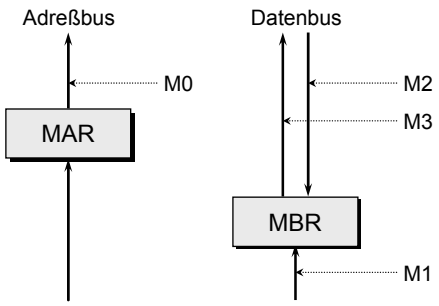




4.5

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)



4.6

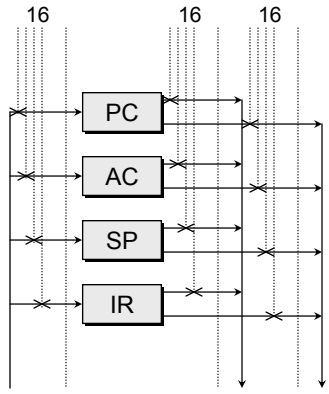
Die wesentlichen Register

- 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

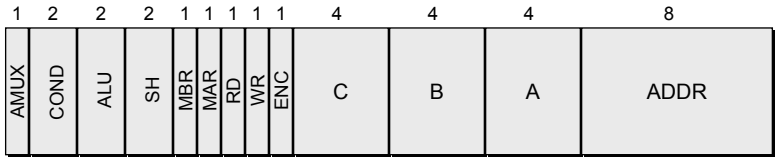
- Insgesamt 61 Leitungen
  - 16 Auswahl A-Bus
  - 16 Auswahl B-Bus
  - 16 Auswahl C-Bus
  - 2 A- und B-Latch
  - 2 ALU
  - 2 Shifter
  - 4 MAR und MBR
  - 2 Speicher lesen oder schreiben
  - 1 Amux
- Möglichkeiten der Leitungsreduktion?



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

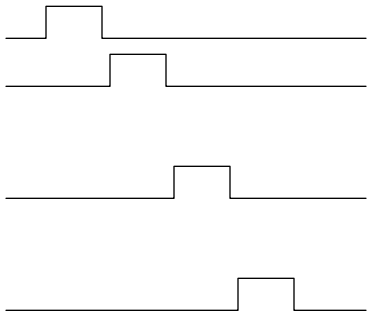
Bedeutung der einzelnen Felder

- 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

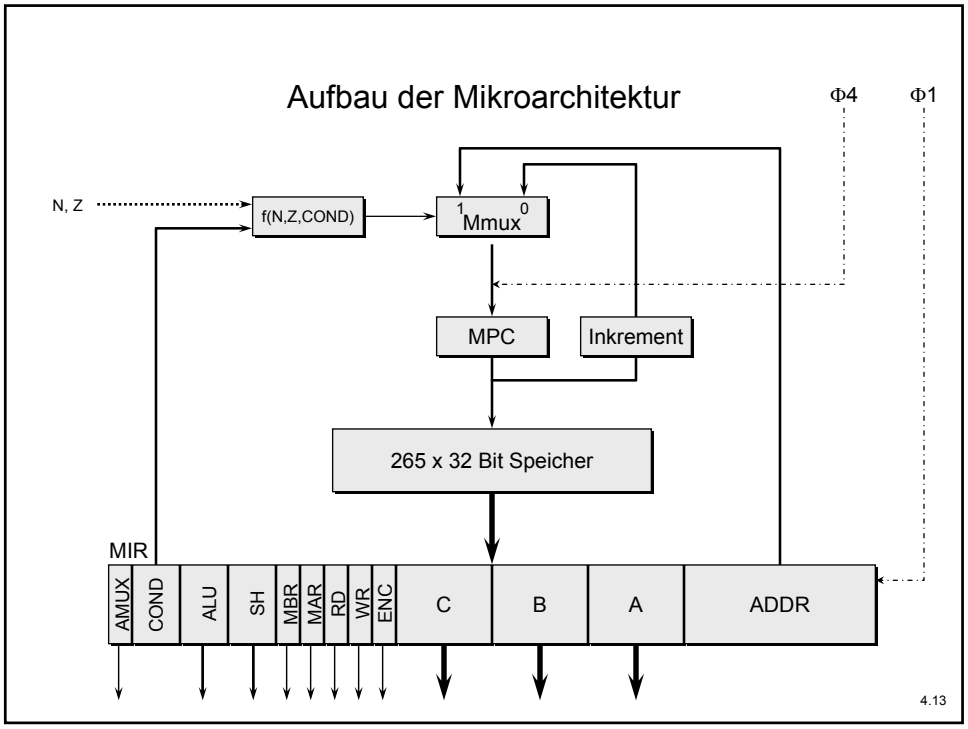


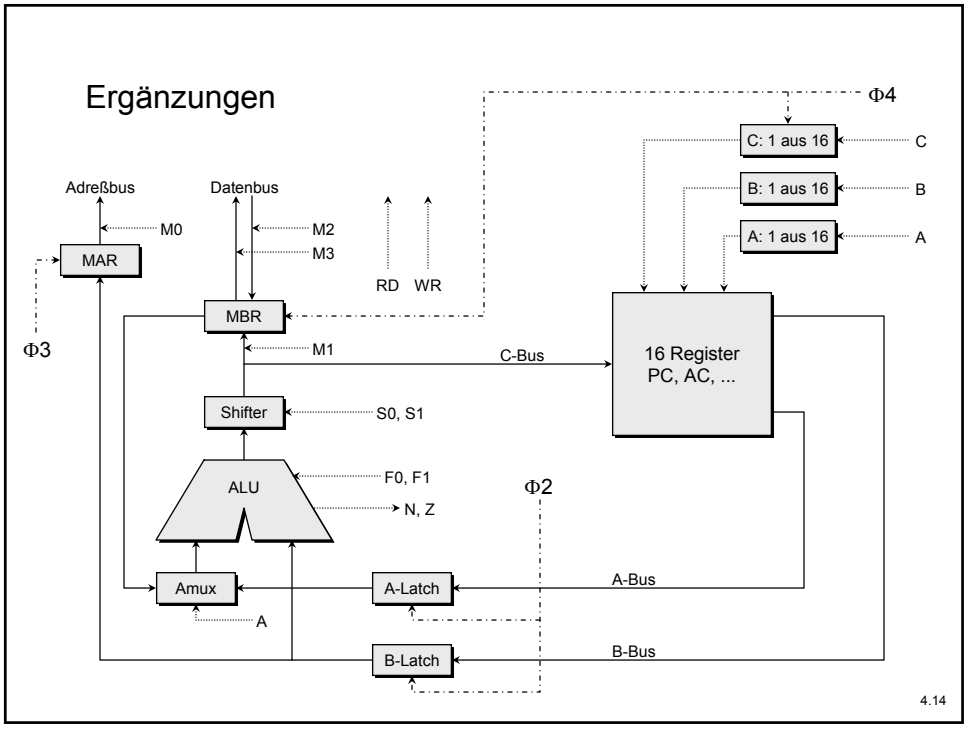
4.11

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
N, Z
f(N,Z,COND)
1 Mmux 0
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
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-Latch
A-Bus
A
B-Latch
B-Bus
4.14

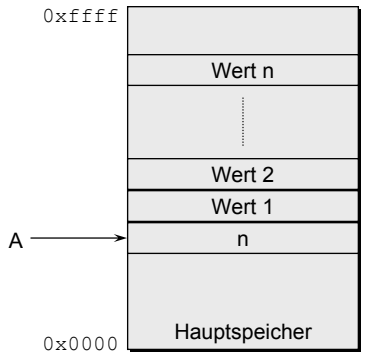
Beschreibungsmethoden

- 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

Mikroprogramm

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



4.18

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

4.23

Offene Fragen

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

4.24

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

4.25

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
```

4.26

```
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
```

4.27

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)

4.28

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

4.29

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
```

4.30

```
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

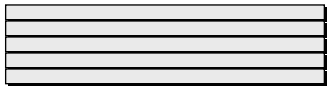
- Code-Fragment Fibonacci:

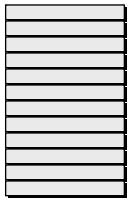
```
    ...
 8: DESP 1
 9: LOCO -2
10: ADDL 2
11: JPOS
12: LOCO 1
13: STOL 0
14: JUMP 25
15: LOCO -1
16: ADDL 2
    ...
```

- Zwei Befehle funktionieren nicht wie gewünscht?

4.35

Horizontale vs. vertikale Mikroprogrammierung





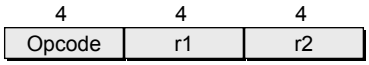
- Breite Mikroinstruktionen
- Vorteile
  - Alle funktionalen Bausteine ansprechbar
  - Maximale Parallelität möglich
  - Günstiges Verhältnis Mikro- zu Makroinstruktionen
- Nachteil
  - Großer Flächenbedarf

- 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

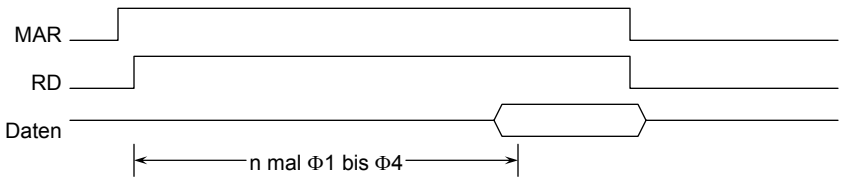
4.38

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

4.39

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)

4.40