Vorlesung Rechnerstrukturen

Dies ist eine Mitschrift der Vorlesung Rechnerstrukturen aus dem SS 2000 (ab Ende Kapitel 4).

Ich habe diese Mitschrift für mich selbst angefertigt, und stelle sie allen, die auch noch die Klausur schreiben müssen, „as is" zur Verfügung.

Diese Mitschrift stellt keinen Anspruch auf Vollständigkeit oder Fehlerfreiheit; es sind im Gegenteil eigentlich nur Stichwörter; vor allem das, was in der Vorlesung gesagt wurde und nicht auf den Folien steht.
Sie stellt eine Ergänzung zu den Folien dar, und setzt diese Voraus.

Wenn jemand Fehler entdeckt (außer natürlich der zahlreichen Rechtschreibfehler), würde ich mich über eine e-mail freuen.

Wenn sonst noch etwas zu Rechnerstrukturen in digitaler Form hat (und seien es nur Teile der Vorlesung oder von Übungen) kann mir das gern auch senden, ich freue mich!

Viel Erfolg bei RS!

Ach ja: da ich auch am 2.März 2001 RS schreibe, such ich noch Lernpartner. Wer also Lust hat, mit eine kleine Lerngruppe zu bilden, schreib mir bitte eine e-mail!

Klaus Ridder (klaus.ridder@gmx.de)

17.02.2001

Inhalt

4. SPEICHERZELLEN 4

D-LATCHES 4
D-FLIP-FLOP: 4

5. DARSTELLUNG VON ZAHLEN / ZEICHEN IM RECHNER 6

EXCESS-DARSTELLUNG FÜR D: 8
DARSTELLUNG DER 0: 8
MULTIPLIKATION 8
VERFAHREN FÜR OPERANDEN MIT HALBER WORTLÄNGE M = N/2: 8
MULTIPLIKATION BEI GLEITKOMMA-ZAHLEN: 9
DARSTELLUNG ALPHANUMERISCHER DATEN 9

6. PROGRAMMIEBARE LOGISCHE ARRAYS (PLA'S) 10

PUNKT-ORIENTIERTE DARSTELLUNG EINS PLA'S: 11
FALTUNG VON PLA'S 11
FALTUNG MIT NEBENBEDINGUNGEN 12
READ ONLY MEMORY: (DURCH PLA REALISIERT) 12
PLA'S MIT ZUSTAND 12
GRAY CODE 13

VON-NEUMANN-RECHNER 14

CPU 14
CHARAKTERISTIKA DES VON-NEUMANN-RECHNERS: 14
FETCH-EXECUTE-ZYKULS 15
SPEICHERHIERARCHIE: 15
CACHE-HIERARCHIE 16
I/O-ORGANISATION 16
PROGRAMMIERTER I/O 16
INTERRUPT 16
DMA 16
BUSSE 16
CISC-PROZESSOREN 17
RISC-PROZESSOREN 17
BEFEHLSPHASEN-PIPELINING 17
SUPERSKALAR-ARCHITEKTUR 17
ALTERNATIVEN ZUM VON-NEUMANN-KONZEPT 17

DER POWER-PC 601 18

DIE BPU 18
DIE INSTRUCTION UNIT 18
DIE INTEGER UNIT 18
DIE FLOATING POINT UNIT (FPU) 18
PROGRAMMIERMODELL DES PPC 601 19
SUPERISOR-MODUS 19
KLASSEN VON BEFEHLEN 19
INTEGER-BEFEHLE 19
MULTIPLIKATION 20

LOGISCHE OPERATIONEN 20
SHIFT -OPERATIONEN 20
VERGLEICHE 21
MOVE-BEFEHLE 21
FLOATING-POINT-BEFEHLE: 21
ADDRESSIEUNGSARTEN FÜR LOAD- UND STORE-BEFEHLE 21
BEISPIELE FÜR LOAD: 21
BEISPIELE FÜR STORE: 22
SPRUNGBEFEHLE 22
SPRÜNGE MIT SEITENEFFEKTEN 22
PROGRAMM-KONTOLL-BEFEHLE 22

ASSEMBLER 23

FESTSTELLEN EINER ORDNUNG (COMPARE) 23
FIBONACCI-ZAHLEN 23
SPEICHERMANAGEMENT IM POWER-PC 24
STRUKTURIERUNG DES SPEICHERS: 24
ADRESSÜBERSETZUNG 24
BAT-REGISTER 25
CACHE (PPC 601) 25
MEMORY UNIT 25
SNOOPING 26
ANDERE ARCHITEKTUREN 26

17. Mai 00

4. Speicherzellen

D-Latches

Problem war: das **D-Latch** reagiert während der gesamten „up-time" des Taktsignals auf Veränderungen.

Um einen Flip-Flop zu realisieren, fügen wir einen Pulsgenerator hinzu. Nur während des sehr kurzen Pulses (am Anfang der „up-time") reagiert dann das Latch auf Schaltungen.

Folie 4.19 linke Abb.:
Das Signal erreicht b nach ca. 10 psec (Verzögerung durch den Inverter).
Wenn Leitung a-c ca. 2 Üm lang ist, dann erreicht das Signal c nach ca. 0,01 psec. (vernachlässigbar)
Ç = 10 psec.

Auf deutsch: der Inverter braucht Zeit zum schalten, in der Zwischenzeit liegt das alte Signal an. Also haben wir für einen kurzen Zeitpunkt das gleiche Signal an beiden Eingängen des Und - Gatters, und somit einen kurzen Impuls am Ausgang.

Folie 4.20:

D-Flip-Flop:

D-Latch mit vorgeschaltetem Pulsgenerator: Zustandsänderungen nur noch während des kurzen Impulses bei aufsteigender Flanke (nicht bei absteigender Flanke, da wir dort am UND-Gatter kurz 00 anliegen haben, und das ergibt auch 0.)

ALSO: Latches: schalten während der gesamten Laufzeit eines Taktes, Flip-Flops schalten nur während einer aufsteigenden Flanke. Hierfür wird ein „Pulsgenerator" vorgeschaltet, der aus einem UND-Gatter besteht, in der das Signal und sein inverses reingeht. Beim umschalten auf 1 hat der Inverter eine kurze Verzögerung, es liegt kurz „1" auf beiden Eingängen an und wir haben einen kurzen Impuls.

Folie 4.21:
A + B: Latches: a schaltet bei hohem Taktsignal, b bei niedrigem.
C + D: Flip-Flops: C schaltet bei hohem Taktsignal, D bei niedrigem
CLR: (Clear): setzt Q auf 0. **PR** (Preset): setzt Q auf 1.

Folie 4.22:

Chip mit 2 unabhängigen D-Flip-Flops:
V_{CC} = Spannungsquelle
GND = Masse, Erde, Ground.

6 Eingänge:
CK: Clock (das Taktsignal)
D: Set bei 1, Reset bei 0.
PR: überschreibt D und setzt Q auf 1.
CLR: überschreibt D und setzt Q auf 0.
Q: Ausgang
$\bar{Q}$: invertierter Ausgang.

8-Bit-Register: 2 dieser Chips parallel geschaltet ergeben ein 16 Bit Register (wobei die 1er- und 11er-Pins verbunden werden).

SSI: small scale integration: 1-10 Gatter
MSI: medium scale integration: 10-100 Gatter
LSI: large scale integration: 100-100.000 Gatter
VLSI: very large scale integration: > 100.000 Gatter (vgl. Intel P II: 33 Mio. Gatter)

Folie 4.23: 4x3-Speicher

Ij; Eingang (auf D)
Oj: Ausgang (in O)
A, A_0: Adressen
CS: Chip Select
RD: Read (Lesen / Schreiben)
OE: Output Enable

Es wird jeweils 1 3-Bit-Wort gelesen oder geschrieben.
Lesen: CS=1, RD=1, OE=1
→ CK-Eingänge sind niedrig (auf 0), damit sind keine Änderungen möglich.
Die von A_1, A_0 adressierten Flip-Flops (Bits) senden ihre Q-Signale in die Ausgänge Q_1, Q_2, Q_3.

Schreiben: CS=1, RD=0, OE=0, durch OE=0 werden Ausgänge blockiert.
RD auf 1 = Schreibschutz aufgehoben, indem wir kurz die CK (Clock) auf 1 setzen.

Ij wird in das j-te FlipFlop des adressierten Wortes gespeichert.
(bei aufsteigender Flanke des Takts)

Folie 4.24: Invertierender Schalter / Puffer
Data In und Data Out des Schalters/Puffers werden bei
gesetztem Control (also gesetztem Output Enable Bit auf Folie 4.23)
verbunden

Speicherdesign von 4.23 beliebig erweiterbar. 2^m D n - Speicher.
Regelmäßige Schaltwerkmuster sind ideal geeignet für Chip-Realisierung.

Moore's Law: Speicherdichte auf 1 Chip verdoppelt sich ca. alle 18 Monate (seit 1965)
(allerdings gehen Investitionen in die Millionen $ für Verdopplung.)

17.05.2000

CS = Chip select
WE = Write enable
OE = Output Enable

Schreibweise: "CS mit Strich drüber" bedeutet, dass der Chip ausgewählt wird, wenn das Signal auf der Leitung niedrig (0) ist.

512K = 2^{19} , 1K = 1024 = 2^{10}, 1M = 2^{20}

Folie 4.25
Linke Figur: direkte Auswahl des Bits.
Rechte Figur: Speicherzugriff (auf ein Bit) in 2 Zyklen: 1.) Zeilenauswahl, 2.) Spaltenauswahl.
(RAS = Row Address Stroke)
(CAS = Column Address Stroke)
→ Vorteil: weniger Pins.
→ Nachteil: langsamer.
Typisch für DRAM (Hauptspeicher)

Folie 4.26
SRAM : static RAM: Speicherung in FLIP-FLOPS (5-6x teurer)
DRAM: Dynamic RAM: Speicherung in Kondensatoren; muss aufgefrischt werden. (langsam!)
EDO-RAM: Extended Data Output – Random Access Memory:
Während man 1 Spalte ausliest, kann man schon die nächste Zeile auslesen (verschränken der beiden Zugriffszyklen)

SDRAM: Synchrounous DRAM. (heute typisch für Hauptspeicher):
Kombination aus SRAM und DRAM (hybrid)

5. Darstellung von Zahlen / Zeichen im Rechner

Darstellung von INTEGER, REAL; Multiplikation, alphanumerischer Daten.

Darstellung negativer Zahlen:

- Zusatzbit für + oder - : ungeeignet, da z.B. 2 Darstellungen für 0 und Extra Subtrahierwerk nötig.
- Lösung: KOMPLEMENTDARSTELLUNG:
  - **Einer-Komplement:** einfach invertieren.
  - **ZweierKomplement:** +1.

Beachte: Komplementdarstellung immer relativ zu gegebener Stellenzahl.

Komplement kann für jedes b-adische System (b>1) angegeben werden, als b- bzw. b-1 – Komplement.
(Komplement = einfach im Zählwerk rückwärts zählen, wie z.B. bei „Kilometerzähler" mit 2 Zahlen.)

Beispiel: Dezimalsystem mit 2 Stellen:
- 23 im 10er-Komplement: 77
- 23 im 9er-Komplement: 76

zurück zum Dualsystem:
[Negative Zahlen werden als Komplement des Betrags der jeweiligen Zahl dargestellt:
1.) Positive Zahl invertieren
2.) 1 dazu

Beispiel: 4 = 0100, 1er-Komplement: 1011, 2er-Komplement: 1100 (=1011 + 1)

Wertebereich bei n Stellen:
1er-Komplement: $-(2^{n-1} - 1)$ bis $2^{n-1} - 1$
2er-Komplement: -2^{n-1} bis $2^{n-1} - 1$
Beachte:
Sei $x \geq 0$,
$-x = \mathbf{(2^n-1)-x}$ im 1er-Komplement (n=4 -6 = **15**-6 = $(1001)_2$
$-x = \mathbf{2^n} \; \mathbf{-x}$ im 2er-Komplement (n=4 -6 = **16**-6 = $(1001)_2$

analog im **10er-Komplement** für Dezimalzahlen: $-x = 10^n - x$
Bsp.: N02 $-23 = 10^2 - 23 = 77$

Vorteil der Komplementdarstellung: **Subtraktion** ist auf **Addition** reduzierbar.

Beispiel: 6-7, 7-4 n=4

a) 1er-Komplement b) 2er-Komplement:

6 - 7:		7 - 4:		6 - 7:		7 - 4:	
6	0110	7	0111	6	0110	7	0111
$+K_1(7)$	1000	+K1(4)	1011	$+K_2(7)$	1001	$+K_2(4)$	1100
$=K_1(1)$	1110	=K1(3)	0010	$=K_2(1)$	1111	= +3	0011
		+ Übertrag	0001			(Übertrag hier ignorieren!)	
		+3	0011 korrekt				

→ 1er-K.: berechne x-y durch $x+K_1(y)$. (ggf. Übertrag draufaddieren)
→ 2er-K.: berechne x-y durch $x+K_2(y)$. (ggf. Übertrag ignorieren)

x –y: berechne $x + K2(y) = x + 2^n - y = 2^n + (x - y)$
^ eventueller Übertrag

→ Overflow zu beachten !! (z.B. –7-2), da wir den Übertrag wegschmeißen.

24.05.2000 leider kam ich zu spät, wer kann das ergänzen?

Dezimalzahl → Dualzahl:

Immer mit 2 multiplizieren, wenn links eine 1 steht, diese in die Binärdarstellung schreiben (und her löschen), sonst eine 0 schreiben. Dann weitermultiplizieren, bis rechts vom Komma nur noch Nullen sind. → fertig.

Nachtrag von Martin Niedernolte:
Das Verfahren "Immer mit 2 multiplizieren..." ist für Nachkommastellen gedacht. Z.B. 0,375 im Dezimalsystem ist
0,375 * 2 = 0,75
0,75 * 2 = 1,5 (die 1 weg)
0,5 * 2 = 1,0 (Ende)
-> 0,375 dec = 0,011 bin
Für die Vorkommastellen rechnet man hingegen denke ich die Potenzen von 2 aus und versucht sie abzuziehen, was nach dem Komma (für z.B. 1/128) zu kompliziert wäre.

Verfahren für Reelle Zahlen:

Umwandlung Dezimalsystem → Dualsystem

- fortgesetzte Multiplikation mit 2, wobei die Stelle vor dem Komma ignoriert wird.
- Terminierung, wann nach dem Komma nur 0en stehen
- Die erzeugten Ziffern vor dem Komma ergeben in der Reihenfolge der Erzeugung die Dualdarstellung des Bruchs (nach dem Komma)

Gleitkommadarstellung:

Beispiel: $12{,}288E10^{2}$:
M d
= $0{,}12288E10^{4}$
= $12288E10^{-1}$

verschiedene Darstellungen derselben Zahl eher ungünstig. (z.B. für Vergleiche)
Daher wählt man die normierte Darstellung

Für b=2 heißt das: das höchste Bit muss 1 sein.

$0{,}0001101E10^{20}$ (unnormalisiert) →
$0{,}1101000E2^{17}$ (normalisiert) Darstellung der Matisse m

Beispiel für interne Darstellung
Wortlänge n=32, b=2
m Matisse: 23 Bits + 1 Bit für Vorzeichen
d Exponent: 8 Bits (negative Exponenten im 2er-Komplement)

Bsp.:
0 | 10011101001110011000000 | 00001101
VZ m d
$=(5031{,}1875)_{10}$

darstellbarer Bereich:
pos. Zahlen: $0{,}5E2^{-128} \leq x \leq (1-2^{-23})E2^{127}$
neg. Zahlen: $(-1-2^{-23})E2^{127} \leq x \leq -0{,}5E2^{-128}$
→ 0 nicht darstellbar (Lösung siehe gleich)

(die erste Stelle nach dem Komma in der Binärdarstellung ist immer 1! → dieses Bit können wir uns sogar sparen. „hidden bit“) → 1 Dualstelle mehr. (24 statt 23))
Vorsicht: m= 000...0000 mit hidden bit steht für $(0{,}1)_2 = (0{,}5)_{10}$.

Excess-Darstellung für d:

(als Ersatz für reguläre Komplementdarstellung)

Beispiel: g = 8 bits für Exponenten d
Im 2er-Komplement: $-128 \leq d \leq 127$

Excess-128-Darstellung: verwendet **d' = d+128** → d' zwischen 0 und 255.
Vorteil: einfache Größenvergleich von Exponenten über Dualzahlinterpretation.

$d_1 = -1$ $d_2 = +1$
$d_1' = 127$ $d_2' = 129$

$(d_1')_2 = 01111111 < 10000001 = (d_2')_2$ dagegen:
$K_2(1) = 11111111 \not< 00000001$

Allgemein: bei g bits ergibt sich für Exponenten d die Excess-$2^{(g-1)}$-Darstellung $d' = d+2^{(g-1)}$

> *Auf deutsch: statt –128 ... 127 betrachten wir einfach 0 ... 255 (Verschiebung). 128 ist also unsere neue Null.*

Darstellung der 0:

Exponent bei Excess-Darstellung=0 → 0. (Matisse wird ignoriert)
(mit anderen Worten Für g=8 wird der Exponent –128 für die reserviert.)
(oft werden noch andere Exponenten reserviert für unendlich und „not a number")

1.) Gleitkommazahlen: großen Wertebereich, aber **geringe Genauigkeit**.
Beispiel: Bei 23 Bits für m ergeben sich 10 signifikante Dezimalstellen.
Abhilfe: Double precision (Verwendung von 2 Worten für Gleitkommadarstellung)

2.) dargestellte Zahlen sind nicht gleich verteilt: (nahe 0 kann man in einem festgelegten Bereich viel mehr Zahlen darstellen als weit draußen.)

Multiplikation

Beispiel: 13 x 9: „Schulmethode":

```
1101 x 1001
   1101
    0000
     0000
      1101
   1110101
```

> *mit 2 multiplizieren ist ja ein links-shift um 1.*
> *Jedes mal wenn wir eine 1 haben, addieren wir die Zahl drauf, danach shiften wir sie um 1 nach links. So machen wir weiter, bei 0 addieren wir halt nicht drauf.*

5. Mai 2000

Verfahren für Operanden mit halber Wortlänge m = n/2:

1. Akku:=0
   Lade Multiplikanden in die unteren (rechten) Bits von X. Entsprechend für den Multiplikator.
2. **FOR** i:=1 **TO** m **DO**:
   IF niedrigstes Bit von Y = 1 **THEN**
   Akku := Akku + X,
   Rechtsshift von Y um 1 Bit,
   Linksshift von X um 1 Bit („Nullen nachschieben")
   END FOR.

CSA(Carry Save)-Methode: Bei längeren Worten Wallace-Trees verwenden. (Tiefe = $\log_2 n$)
(n = Wortlänge der Operanden)

- Verfahren zur Multiplikation direkt anwendbar auf Festkommazahlen
- Bei ganzen Zahlen Verwendung der Vorzeichendarstellung mit gesonderter Berechnung des Vorzeichens.
  Bsp.: -9E8
  → Betrag: 9E8 = 72
  → Vorzeichen: - , da VZ der Operanden verschieden.
  → Ergebnis: -72

Multiplikation bei Gleitkomma-Zahlen:

Folie 5.13:

xEy: : Ergebnisse ggf. normalisieren!
Einfach Matissen multiplizieren und Exponenten addieren.

Beachte. Bei +, - wird der Operand mit dem kleineren Exponenten denormalisiert.

Problem: Rundungsfehler!

Bsp.: Matisse mit 4 Bits:
$0{,}1000E2^{2} + 0{,}1E2^{6}$
$= (0{,}1000E2^{2-6} + 0{,}1)E2^{6}$
$= (0{,}0000 + 0{,}1)E2^{6}$
$= 0{,}1E2^{6}$

FEHLER: 2^{-4} heißt Rechtsshift um 4 Bits (4 Stellen gehen verloren, und unsere 1 ist eine davon...)

Wegen **Rundungsfehlern** gilt z.B. das **Assoziativgesetz** nicht für Gleitkommadarstellungen, d.h. im Allgemeinen sind **x+y+z** und **x+(y+z)** nicht identisch.

Abhilfe durch Spezialhardware („FPU", Floating Point Unit) mit sehr langen Registern.

Darstellung alphanumerischer Daten

Folie 5.14

ASCII-Code: (American Standard Code for Information Interchange)

Sonderzeichen: z.B. CR=Carriage Return, LF = Line Feed, ect.

7 Bits pro Zeichen, +1 Parity Bit (P) zur Fehlererkennung.
(Parity = 1 ɜ Anzahl der 1en in den restlichen 7 Bits ~~un~~gerade)

H	e	l	p	!
11001000	**1**1100101	**1**1101100	**0**1110000	**1**0100001

Parity Bit erlaubt einfache Fehler zu erkennen. (invertieren *eines* Bits wird erkannt.)

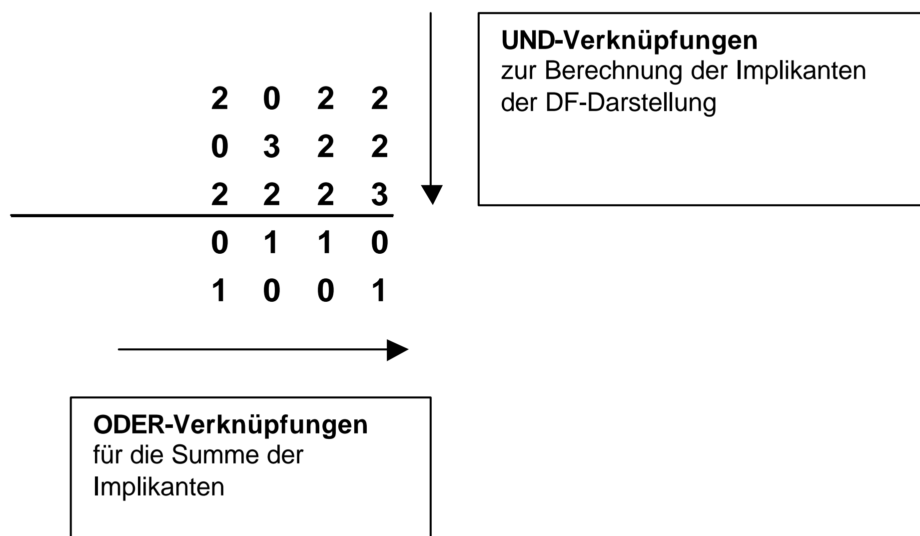
6. Programmierbare logische Arrays (PLA's)

Zur Erinnerung: Speicher wie SDRAM's eignen sich wegen der einfachen Layouts besonders für Chip-Realisierungen (erlaubt sehr hohe Packungsdichten: > 10^5 Gatter pro 4 mm^2

Jetzt: Realisierung von Schaltfunktionen beliebiger Art mit Hilfe **regelmäßiger Gitternetze**, an deren Knoten Standardbausteine angebracht sind.

4 Bausteintypen:

	VERTIKAL	HORIZONTAL
0	x → x	y → y
1	x → x	y → x OR y
2	x → x AND y	y → y
3	x → x NAND y	y → y



Für eine Schaltfunktion F: $B^n \rightarrow B^m$ benötigt man für die PLA - Realisierung mindestens
n UND-Zeilen und
m ODER - Zeilen. Wenn F ein DF-Darstellung mit k Produkttermen (Implikanten) hat, benötigt man ≥ k Spalten.

Satz 6.9: Für jede Schaltfunktion F:$B^n \rightarrow B^m$ gibt es eine Realisierung durch ein (entsprechend dimensioniertes) PLAs. (Der Satz folgt unmittelbar aus der Darstellbarkeit von boolschen Funktionen durch disjunktive Formen.

13.05.2000

Jede Funktion kann man also durch ein PLA darstellen.

Beachte: Optimierung der Schaltfunktion ist bei PLA-Realisierung oft nicht notwendig, da sie zu keiner Verkleinerung des Arrays führt.

Für das Beispiel: $u = (x + \underline{x})\underline{y}z + xyz = x\underline{y}z + \underline{xy}z + xyz$
$V = x(y + \underline{y})z + xy\underline{z} = xyz + x\underline{y}z + xy\underline{z}$
→ immer noch 4 verschiedene Summanden, also keine Änderung der Zeilen-/Spaltenzahl.

PLA-Matrix:

2	**2**	**2**	**3**
2	**2**	**3**	**3**
2	**3**	**2**	**2**
1	**0**	**1**	**1**
1	**1**	**1**	**0**

zu Folie 6.10:
Wir legen nun an jeden Baustein noch Leitungen s,t, die festlegen, welcher Baustein 0-3 es nun ist.

Nun haben wir ein PLA: m+n Zeilen , k Spalten → (m+n)Ek Gitterpunkte. (=U)
Wir brauchen also **2xU** bits. → meist im ROM.

Punkt-orientierte Darstellung eins PLA's:

Zu Folie 6.11:
Punkte links: Inverter
Punkte auf den Gitterpunkten: Und-Ebene: Punkt=2, sonst 0.
Oder-Ebene: Punkt=1, sonst 0.

Dadurch dass wir alle Signale doppelt anlegen (einmal original, einmal invertiert), sparen wir uns den 3. Baustein. (der wird ja nur in der UND-Ebene gebraucht.

Faltung von PLA's

Denken wir uns um alle 1en in jeder Zeile einen Block.
Sind die Blöcke von 2 Zeilen disjunkt, können wir diese zusammenlegen: die Zahl vom linken Block bleibt links angelegt, den Eingang vom rechten Block legen wir rechts an.

Beachte: Beim Verschmelzen von Zeilen reicht die Disjunktheit der 1-en nicht aus. Die 1-en müssen jeweils zusammenhängende Blöcke ergeben, die sich nicht überlappen.
→ Sollte dies nicht von vornherein möglich sein, kann es oft durch Vertauschung von Spalten erreichen.

Faltung mit Nebenbedingungen

a) Blockfaltung:

Bedingung: eine Spalte darf keine Punkte A,B erhalten, wobei A sein Signal nur von links und B nur von rechts erhält.
d.h.: wir können alle Zeilen an der gleichen Stelle durchschneiden.

BEDINGTE FALTUNG

Vorgabe, welche Signale von rechts und welche von links anliegen.
Die Vorgabe, welche Signale links bzw. rechts anliegen, können beliebig sein.

b) bedingte Blockfaltung:

Kombination aus a) und b).

Beachte: um eine bedingte Faltung zu erreichen, ist es manchmal nötig, Zeilen vorzusehen, wo nur auf einer Seite ein Signal anliegt.

07.06.00

Folie 6.19

Read Only Memory: (durch PLA realisiert)

m: Wortlänge, 2^n Worte

ODER-Ebene entspicht dem eigentlichen Speicher, d.h. die i-te Spalte der ODER-Ebene entspricht dem i-ten Speicherwort . (i zwischen 0 und 2^n-1)

UND-Ebene: n D 2^n – Matrix.
j-te Spalte der UND-Ebene entspricht der dualen Darstellung von j, wobei für 0 der Baustein 3 und für 1 der Baustein 2 eingesetzt wird.

(merke: 2 AND, 3 NAND ; 1 OR , 0 gar nichts)

Folie 6.22

PLA's mit Zustand

Neu: ein Teil der Aufgaben wird zwischengespeichert (Register) und sind Teil der Eingaben beim nächsten Taktzyklus.
Der Registerinhalt kann als interner Zustand des Schaltwerks aufgefasst werden.
Formal entsprechen PLA's mit Zustand einem endlichen Automaten (siehe Vorlesung theoretische Informatik).

Beispiel 6.3

Gray Code

Ringzähler[1] für 3-stelligen Gray-Code und extra Ausgaben y_1, y_2

Gray-Code: spezielle Codierung von Zahlen (hier: 0-7): Im Gegensatz zur Dualdarstellung wird die Darstellung so gewählt, dass sich zwei (zyklisch) benachbarte Bitfolgen an genau einer Stelle (Bit) unterscheiden.

Folie 6.24

Die Ausgabe c0, c1, c2 ist ja die folgende Zahl. Genau die wird nun wieder in den Eingang (getaktet!) zurückgeführt, und wir zählen dann alle Zahlen durch.

Folie 6.25

Ein PLA mit Zustand kann als Rechenwerk für +,-,E,/ dienen:

Die beiden Eingangssignale S,T legen fest, welche Operation ausgeführt werden soll:

S	T	Funtion des PLA
0	0	Addierer
0	1	Subtraktion
1	0	Multiplikation
1	1	Division

Aufgabe: Berechne X:=X+Y
Dazu braucht man eine zusätzliche Kontrolleinheit. Was gibt es zu tun? („Mikroprogramm“)

1. Erkenne, um **welchen Befehl** es sich handelt
   („Addiere den Inhalt der Speicherzellen X und Y, und speichere das Ergebnis in X“)
2. Lade **X** in den **Akkumulator**
3. Lade **Y** in den **Puffer** des Rechenwerks
4. Setze **S** und **T** auf **0** (=Addition) (aktiviere das Rechenwerk)
5. Speichere den **Akkuinhalt** in **X**.

Dieses „Mikroprogramm“ wird im Steuerwerk-PLA abgespeichert.

[1] Ringzähler = die Ausgabe ist immer der Nachfolger des aktuellen Wertes.

Von-Neumann-Rechner

Idee:
- Programm + Daten werden gemeinsam im Speicher abgelegt.
- **C**entral **P**rocessing **U**nit führt die **I**nstruktionen nacheinander aus.
- Ein-Ausgabeeinheit **I**/O dient zur Kommunikation nach außen (Menschen oder Rechner)
- Verbindungen über Datenbus und Addressbus

Prinzip des **Universalrechners**:
- Jede berechenbare Funktion ist durch ein Programm beschreibbar und damit durch einen Von-Neumann-Rechner realisierbar.

CPU

1. Datenprozessor: Rechenwerk für +,-,*,/ und Vergleiche von Resgisterinhalten:

Arithmetic **L**ogic **U**nit mit 3 Registern:

- **A**kku (A) und **L**ink (L) für Überträge
- **M**ultiplikator**R**egister (MR)
- **M**emory **B**uffer **R**egister (MBR)

Beachte: Moderne Rechner haben oft viel mehr Register. Z.B. SPARC-II hat 138 Register, inkl. mehrerer Akkus.

2. Befehlsprozessor: Befehle entschlüsseln und Ausführung steuern.

Ebenfalls 3 Register:
- **I**nstruction **R**egister (IR) = Befehlsregister: zur Aufnahme es aktuellen Befehls.
- **P**rogram **C**ounter (PC) = Befehlszähler: enthält die Adresse des nächsten Befehls.
- **M**emory **A**ddress **R**egister (MAR) = Speicheradressregister: enthält die Adresse der als nächstes verwendeten Speicherzelle.

08.06.00

BEACHTE:
DEM PROGRAMMIERER (ETWA ASSEMBLER) BLEIBEN **MAR, MBR** + **IR** IN DER REGEL VERBORGEN.
Diese spielen natürlich eine Rolle bei der Mikroprogrammierung.

Der **PC** hat eine „Zwitterstellung“: Er wird sowohl intern verändert als auch explizit gesetzt durch Sprungbefehle.

Folie 8.3:

Charakteristika des von-Neumann-Rechners:

1. 1 Befehl pro Takt (**SISD** = **S**ingle **I**nstruction **S**ingle **D**ata)
2. **Befehl**, **Daten**, **Adressen** im gleichen Speicher!
3. → Daten nicht schützbar

Folie 8.4

Maschinenbefehle sind „nur“ Folgen von Bits. Um für Menschen lesbar zu sein, haben Maschinenbefehle mnemonische Darstellung wie „ADD X,Y“ („**Assemblersprache**“)

BEACHTE: JEDER BEFEHL VERWENDET DEN AKKU (DAHER BRAUCHT MAN NUR DAS 2. ARGUMENT EXTRA ANZUGEBEN)

Es reichen 1-Adressbefehle

Bsp.: **SUB X** bedeutet: **Akku := Akku – X** (das 1. Argument ist immer der Akku, das 2. Argument ist der angegebene Parameter. Gespeichert wird immer wieder im Akku.)

Es gibt auch Befehle ohne Adresse (also ohne Parameter, wo wir also nur mit dem Akku arbeiten) (z.B. **INC** bedeutet **Akku:=Akku+1**)

Neben 0- und 1-Adressbefehle werden in der Regel auch 2-, 3- oder 4-Adressbefehle zur Verfügung gestellt.
(zur Erleichterung der Programmierung und Effizienzsteigerung, falls viele Register zur Verfügung stehen.)

Fetch-Execute-Zykuls

FETCH-PHASE:
1. Adresse des neuen Befehls: **P**rogram **C**ounter → **M**emory **A**ddress **R**egister
2. Inhalt dieser Adresse (Befehl): → **M**emory **B**uffer **R**egister → **I**nstruction **R**egister
3. **Decodierer** ekennt Befehl und holt benötigte Daten aus Speicher.

EXECUTION-PHASE:
Ausführung des Befehls und Initialisierung neuer **Fetch-Phase**

Beachte: Die weitaus meiste Zeit wird verbraucht durch Speicherzugriffe, d.h. Kommunikation CPU 3 Speicher stellt (vor allem heute) einen **Flaschenhals** für von-Neumann-Architektur dar.

Beachte: Die Größe des MAR bestimmt die maximal adressierbare Speichergröße.

Bsp.: Intel-Prozessoren

8088	20 Bit = 2^{20} = 1 MB Speicherworte
80286	24 Bit = 16 MB
80386 und Pentium **III**	32 Bit = 4 Gbit. (auch noch zu wenig)
Titanium	64 Bit = 2.097.152 Terabyte → Endlich genug ?

Die Größe des MBR entspricht der eines Speicherworts (Pentium: 32 Bit)

Folie 8.7:

Speicherhierarchie:

CPU
3 Register
3 Cache
3 Hauptspeicher (z.B. SD-RAM)
3 Hintergrundspeicher (Festplatte, Diskette, DVD, Magnetband,...)

Beachte Unterschiede in Zugriffszeiten:

Cache:	~10 Nanosekunden (schneller SRAM-Speicher: enthält Ausschnitte des Hauptspeichers, auf die gerade zugegriffen wird. Sinnvoll wegen „90:10-Regel“: 90% der Zugriffe erfolgen auf 10% der Daten, die in der Regel lokal zusammenhängend sind.)
DRAM:	~ 50 Nanosekunden
Festplatte:	~ 10 Millisekunden

Folie 8.8

Cache-Hierarchie

1. Level-1-Cache: (16-64 KB) direkt in der CPU. (Befehls-Cache / Daten-Cache getrennt)
2. Level-2-Cache: (~512 KB): 512 KB – 1 MB
3. Level-3-Cache (optional; ca. 1 MB)

→ Dient zur Abmilderung des von-Neumannschen Flaschenhalses.

I/O-Organisation

Problem: Endgeräte (Maus, Keyboard, Festplatte etc.) benötigen Zugriff auf den Hauptspeicher im Konkurrent zur CPU und sind dabei oft viel langsamer als die CPU.

Zunächst: Jedes Endgerät besitzt einen I/O-Controller als Schnittstelle zum Speicher und zur CPU

- Steuerleitungen zur CPU
- Datenleitungen zum Hauptspeicher
- Puffer, um Daten bei der Übertragung zwischenzuspeichern
- Statusregister (z.B. Flag „bin lesebereit“)

3 Arten der Datenübertragung:

Programmierter I/O

[Versprecher des Monats: „Die CDU ist für alles verantwortlich – äh – die CPU meine ich.“]

CPU wartet aktiv darauf, dass eine ein Endgerät lesen / schreiben will (durch zyklisches inspizieren des Statusregisters), und steuert den eigentlichen Lese-/Schreibvorgang.

→ bei „billigen“ Realzeit-Prozessoren (die eh nix anderes zu tu haben, als auf Daten zu warten)

Interrupt

Endgerät sendet ein **Interruptsignal** an die CPU.
Erst bei Eintreffen eines Interrupts **unterbricht** die CPU die Programmausführung.
Nachteil: Verarbeitung eines Interrupts immer noch **zeitaufwendig** für die CPU.

DMA

Ein spezieller DMA-Controller übernimmt den eigentlichen Datentransfer.

Beispiel: Schreiben von 32 Speicherworten ab Adresse 100 zum Terminal (Device #4)
Beachte: DMA hat Vorrang vor der CPU beim Speicherzugriff! (→ „Cycle Stealing“)

21.6.2000

Busse

- In der Regel parallele Datenübertragung (z.B. 32 Leitungen für Datenbus bei Speicherwotlänge 32)
- Spezielle Controller „Bus Arbiter“ regeln den Bus-Zugriff
- Beispiele:
- ISA-Bus: 32 Bit, 8.33 Mhz, max. 333 MB / sec. (ISA = Industry Standard Architecture)
- PCI-Bus: 64 Bit, 66 Mhz, 528 MB/sec. (PCI = Peripheral Component Interconnect Bus)

CISC-Prozessoren

CISC = **C**omplex **I**nstruction **S**et **C**omputer (z.B. Pentium)

- Sehr umfangreicher Befehlssatz
- Abarbeitung komplexer Befehle durch Mirkoprogramme gesteuert

Diese Nachteilung führten in den 80er Jahren zur Entwicklung der RISC-Prozessoren.

RISC-Prozessoren

RISC = **R**educed **I**nstruction **S**et **C**omputer (z.B. Sun)

Unterschied: nur ganz simple Befehle werden unterstützt, da die Compiler höherer Programmiersprachen eh nur weniger verschiedene Maschinenbefehle benutzen.

Vorteil: keine Mirkoprogrammierung, sondern feste Verdrahtung.

- Extra Befehle zum Zugriff auf den Speicher
- Sehr viele Register!

Befehlsphasen - Pipelinining

(Folie 8.14): Nachdem eine Instruktion geholt wurde, und hierfür die Daten geholt werden, wird auch **schon die nächste Instruktion** geholt.

Instruction fetch → data fetch → execute → result write laufen also **verschränkt** parallel.

Heute verwenden **alle modernen Rechner** dieses Prinzip.

Manchmal geht das aber nicht, z.B. :

- Datenabhängigkeit→„**Data Hazard**“: (wenn der nächste Befehl das Resultat des vorherigen benötigt)
- Kontrollfluss → „**Control Hazard**: (wenn ein Sprung vom Ergebnis abhängt)

Dann muss die Pipeline angehalten werden.

Viele Rechner „**raten**“ schon mal und rechnet schon mal (eit wäre ja eh verloren). Wenn sich am Ende rausstellt, dass wir die Richtige Operation ausgeführt haben, werden die Befehle DANN zurückgeschrieben, sonst werden sie verworfen und die Instruktionen nochmals ausgefüht.

Superskalar-Architektur

Parallel mehrere Instruktionen ausführen, z.B. auf mehreren Prozessoren.

Alternativen zum von-Neumann-Konzept

Single **Instrultion** - Single Data: S **I** S D
Multiple Multiple M M

Der Power-PC 601

Die BPU

BPU: Behandlung von **Sprungbefehlen**. Verwendet 3 Register (32-bit):

I. **Link-Register** (**LR**): Enthält z.B. Rücksprung-Adresse von Unterprogrammen (wird unterstützt durch Compiler höherer Sprachen speziell für diese Architektur)

II. **Count-Register** (**CTR**): Zähler für Schleifenimplementierung (Vorteil für Kompilierung von Schleifen höherer Spachen)

III. **Condition Register**: (**CR**): unterteilt in 8 4-Bit-Felder, in denen gewisse Ereignisse oder das Ergebnis von Vergleichen festgehalten werden.

Beispiel: Bit 0-3: (CR0): bedeuten bei Integer-Operationen:

- **Bit 0**: **Negativ-Flag** (**LT**) ist 1, wenn Ergebnis negativ.
- **Bit 1**: **Positive Flag**
- **Bit 2**: **Zero Flag**
- **Bit 3**: **Summary-Overflow-Flag** (**SO**): ist 1, wenn Summe größer MaxInt (=Overflow)

Bemerkung: Bei PPC wird die „Big-Endian-Notation" verwendet: d.h. die Bits eines Registers werden **von links nach rechts** durchnummeriert (0-31)
(dagegen „Little-Endian-Notation": Zählung von rechts nach links.)

Die 31 Fehler werden in 7 Blocks unterteilt (CR0 bis CR7)

Die Instruction Unit

Holt gleichzeitig maximal 8 Befehle aus dem Cache (1 Cache-Block oder Sektor)
Q_4-Q_7 dienen als Puffer, um Cache-Zugriff zu begrenzen.
Q_0-Q_3 werden nach Sprung- und Gleitkommabefehlen durchsucht und an BPU + FPU weitergeleitet (Q0 der einzige für Integer-Berechnung!) (?)
Außerdem gibt es den „Q0 Hold", wenn ein Befehl „zurückgestellt" werden muss (s.o.)

2000-06-27

Die Integer Unit

- benötigt in der Regel 1 Takt pro Befehl (keine Mikroprogramme, sondern feste Verdrahtung)
- erledigt Speicherzugriffe (LOAD / STORE)
- verfügt über 32 **G**ereral **P**uropse **R**egister (**GPR**s)
- Im Integer Exception Register (**XER**) werden Flags gesetzt:
  - BIT 1: Summary Overflow Bit (**SO**)
  - BIT 2: overflow Bit (**OV**)
  - BIT 3: Carry Bit (**CA**) (Übetrag)
  - → Overflow bei add → OV und SO auf 1; SO bleibt jedoch dauerhaft auf 1, bis es zurückgesetzt wird!

Die Floating Point Unit (FPU)

- 32 64-Bit-Register
- 1 **F**loating **P**oint **S**tatus and **C**ontrol **R**egister (**FPSCR**)
- enthält Ausnahmeinformationen (z.B. Overflow), wie im IEEE-754-Standard festgelegt.

Programmiermodell des PpC 601

weitere Register:
MQ: **M**ultiplikationsregister (64 Bit)
RTCU: **R**EAL **T**IME **C**LOCK **U**PPER
RTCL REAL **T**IME **C**LOCK **L**OWER

Auf Register in der „Virtual Environment" (die beiden Clocks) nur lesender Zugriff .
IM 601 NOCH REALZEIT-UHR, IN NEUEN PROZESSOREN NUR NOCH ZÄHLER:
TBU, **TBL**: **T**ime **B**ase **U**pper/ **L**ower: 2*32 Bit – Zähler, der über die interne Uhr getaktet wird.

Superisor-Modus

Register, auf die nur das Betriebssystem (bzw. die Hardware) Zugriff hat.
Beispiel: **M**achine **S**tate **R**egister (**MSR**): stellt momentanen Prozessorzustand dar.

Bit 17: **0** = Supervisor-Modus, **1** = nur User-Instruktionen.

Klassen von Befehlen

Integer / Floationg Point-Befehle
Load- / Store- Befehle
Sprung-Befehle
Prozessor-Kontroll-Befehle

alle Befehle sind 32 Bit (**4 Bytes**) lang !!!
Die Adressen einer Instruktion hat also hinten immer zwei Nullen (weil ja durch 4 teilbar)

Integer-Befehle

(‚Arithmetik + logische Befehle, Shift-Befehle,...)

6 Bit: Op-Code: Name der Befehlsgruppe (ADD liegt in der X-Kategorie („31"))
3x **5 Bit**: Adresse des Zielregisters und der 2 Quellregister (es gibt ja 32 Register, also 5 Bit)
10 Bit: Name des genauen Befehls (ADD=266 in X-Kategorie!)
Bit 21: OE-Bit (overflow enable): (in OV- und SO-Bits, s.o.)
Bit 31: RE-Bit: Sollten die CR0-Bits 0-3 gesetzt werden? (zeigen an, ob Register negativ, Null,...)

add rD, rA, rB OE = 0, RE = 0
ADD**.** RD, RA, RB OE = 0, RE = 1
add**o** rD, rA, rB OE = 1, RE = 0
add**o.** rD, rA, rB OE = 1, RE = 1 (o=overflow Bit 21, . = Re Ergebnisbits Bit 31)

Es gibt noch weitere Varianten von ADD:

add**c** (add carrying): wenn Overflow-Enable an ist, wird ein eventueller Übertrag ins CA-Bit im XER gesetzt.)
(analog auch addco, addc., addco.)

add**e** (add extended): addiere auch das CA-Bit dazu, also einen eventuellen Übertrag aus voeriger Operation.

add**i** rD, rA, SIMM (add imediate): hier können wir direkte Werte in die Instruktion schreiben:
wie add, aber Bit 16-31 wird für „SIMM" verwendet: (hat eigenen Gruppencode „14"):
SIMM = **S**igned **Imm**ediate Value: ganze Zahl im 2er-Komplement
SIMM wird zu 32 Bit konvertiert, wobei die oberen 16 Bit mit dem ersten Bit (Vorzeichenbit) aufgefüllt wird.

add**is** rD, rA, SIMM (add immediate shifted): Bei der Konvertierung des SIMM's zu 32 Bit werden die 16 Bit im oberen statt im unteren Bereich gespeichert. (also hinten 16 Nullen anfügen)

add**me**: rD, rA „add minus 1 extended" (rA + CA-Bit – 1)

add**ze**: rD, rA: "add to zero extended" (rA + CA-Bit): 1 addieren, wenn ein Null am Ende ist.

BEISPIEL:
Seien doppelt lange Integer (64 Bit) durch je zwei konsekutive Register dargestellt, etwa (r1,r2) und (r3,r4).
Ziel: (r5,r6) = (r1,r2) + (r3,r4)
Realisierung:
addc r6, r2, r4 (r6 := r2 + r4)
adde r5, r1, r3 (r5 := r1 + 3 + Übertrag)

Substraktion erfolgt über 2er-Komplement-Bildung: subf rD, rA, rB (rD := [rB]– [rA]) (rD := [rB] + [rA] + 1)

Multiplikation

Das Ergebnis der Multiplikation benötigt in der Regel 64 Bit (2 Register).
Daher Multiplikation in 2 Teilen:

mul**hw** rD, rA, rB: „multiply **high** word": berechnet [rA] * [rB] und speichert die **oberen** 32 Bit in rD
mul**lw** rD, rA, rB: „multiply **low** word": berechnet [rA] * [rB] und speichert die **unteren** 32 Bit in rD

außerdem gibt es noch die ganzzahlige Division.

Logische Operationen

and rD, rA, rB: alle Bits von rA und rB werden paarweise parallel durch AND verknüpft, und das Ergebnis in rD gespeichert.

Shift-Operationen

srawi rD, rA, SH: „**s**hift **r**ight **a**lgebraic **w**ord **i**mmediate: Rechtsshift um SH Bits, wobei das Vorzeichenbit (Bit 0) nachgeschoben wird. (rechts fallen Bits raus, links wird das Vorzeichen-Bit, also das Bit ganz links, nachgefüllt.)

BEACHTE: Fällt beim Shift rechts eine 1 heraus, und ist die dargestellte Zahl negativ (also 0-tes Bit ist 1), so wird das CA-Bit auf 1 gesetzt.

2000-06-29

Beispiel: Berechnung des Mittelwertes von 2 ganzen Zahlen: Argumente in r3 und r4, Ergebnis in r3:
add r5,r3,r4 // r5 := r3 + r4
srawi r5,r5,1 // Div 2
addze r3,r5 // 1 dann addieren, falls Summe der beiden Zahlen ungerade und negativ sind.

für –5 ergibt der Rechtsshift eine –3: 011 → 101 (deshalb 1 dazuaddieren → -2.)

Vergleiche

cmpd rA, rB („compare direct"): **LT**-Bit (rA less than rB), **GT** (lA Greater Than rB) oder **EQ** (rA equal rB) – Bit wird gesetzt, die anderen Null.
Außerdem wid XER[SO] noch nach **CR0** kopiert.
Beachte: für den Vergleich werden [rA] und [rB] als ganze Zahlen im **2er-Komplement** intepretiert.

cmpdi rA, SIMM („compae direct immediate"): wir cmpd, jedoch wird statt mit [rB] mit dem auf 32 Bit eweiterten SIMM (sign extension) verglichen.

→ es gibt noch *sehr* viel mehr solche Vergleichsoperationen !

Move-Befehle

nach/ von LR (Linkregister für Sprungadressen), CTR (Counter Register für Schleifen)

mtcr rS (move to Counter Register): CTR ← [rS]
mfcr rD (move from Counter Register): rD ← [CTR]

entsprechend für Link-Register (LR):
milr rS, **mflr** rD

Floating-Point-Befehle:

Operationen gemäß IEEE-754-Standad:
Unterscheidung zwischen Single (32) und Double (64) Expression

SINGLE: Bit 0: Vorzeichen, Bit 1-**8**: Exponent, Bit **8-31**: Matisse
DOUBLE:Bit 0: Vorzeichen, Bit 1-**11**: Exponent, Bit **12-63**: Matisse

Addressieungsarten für Load- und Store-Befehle

„Register Indirect with Immediate Index":Sprungadresse = Wert in Befehl + 1 Registerinhalt
„Register Indirect with Register Index": Sprungadresse = Registerinhalt A + Registeinhalt B
„Register Indirec": Sprungadresse = Registerinhalt A

In allen Fällen gilt: „Register 0" bedeutet Wert 0 !!!

Es können geladen werden:
8-Bit: Byte (b)
16-Bit: Halbworte (h)
32-Bit: Worte (w)
64-Bit: Doppelworte (für Floating Point)

Ablage im Register / Speicherwort erfolgt **rechtsbündig**, wobei der Rest mit Nullen aufgefüllt wird.

Beispiele für Load:

load **b**yte **z**ero rD, d(rA):
Lade 1 Byte von der Adresse, die in **rA+d** steht nach **rD** (bzw. von Adresse **d**, wenn rA = 0). und fülle mit Nullen auf.

load **b**yte **z**ero inde**x**ed rD, rA, rB:
Wie oben, aber mit Adresse [rA] + [rB], (bzw. [rB], wenn [rA] = 0)

load **b**yte **z**ero with **u**pdate rD, d(rA)
Wie lbz, aber zusätzlich wird die errechnete Addresse in rA agelegt.
lbzux anallog (2. Fall, auch mit Update.)

Beispiele für Store:

stw rS, d (rA) = "**s**tore **w**ord":
Speichere den Inhalt von rS in die Adresse, die in rA+d steht (= [rA]+d). (bzw. in d wenn rA =0)

stwx rS, rA, rB = „**s**tore **w**ord inde**x**ed":
Speichere in [rA] + [rB] (bzw. [rB] wenn rA=0).

stwu rS, d(rA) = „**s**tore **w**ord with **u**pdate":
wie **stw**, zusätzlich wird errechnete Adresse nachher in **rA** abgelegt.

Sprungbefehle

Es gibt bedingte und unbedingte Sprungbefehle:

- absolute Adresse
- relative Adresse (= Adresse des aktuellen Befehls + ein angegebener Offset)
- Adresse im Link-Register
- Adresse im Counter-Register

Beispiele für unbedingte Sprünge:

branch (target) (Ziel: aktuelle Adresse + target) (also Relativ!)
branch **a**bsolute (target): spinge nach target (absolut)
branch **l**ink **r**egister: springe zur Adresse aus dem LR

Beispiele für bedingte Sprünge:

branch **t**arget **eq**ual (target): Springe nach aktuelle Adresse + Target, wenn EQ-Bit in CR0 1 ist (d.h. wenn der letzte Vergleich positiv war)
branch **f**alse **a**bsolute: Springe nach target, wenn **G**reater**T**han-Bit in CR0 0 ist.

Bemerkung: es gibt Befehle, um jedes beliebige Bit in CR zu testen.

Sprünge mit Seiteneffekten

branch **t**arget **l**ink **eq**ual , target:
Zusätzlich die Adresse des Befehls, der vor dem Sprung normalerweise als nächstes gekommen wäre, im Link-Register speichern.

branch **d**ecrement **n**ot **z**ero (target):
CTR--, Jump if CTR @ 0.
branch **d**ecrement **z**ero
s.o., aber Jump if CTR = 0. → geeignet für Implementierung von FOR-Schleifen)

Programm-Kontoll-Befehle

system **c**all:
sichert die Adresse des nächsten Befehls in **SRR0** und MSR in **SRR1**.
Danach Abgabe der Kontolle ans Betriebssystem.

Assembler

Location **C**ounter (**LC**):
mit 0 initialisiert. Nach jedem verarbeiteten Befehl wird der LC um 4 Bytes erhöht.

Symboltabelle:
Enthält die Verbindung zwischen symbolischen Adressen und der berechneten Adresse (relativ zum Programmanfang mit Adresse 0. Im physischen Speicher muss der Prgrammblock später nicht bei 0 beginnen.)
PROBLEM: Bei Vorwärtsreferenzen sind 2 Durchläufe des Assemblers nötig.

Assembler-Programmierung in C:
die Argumente werden in den Registern **r3**, **r4**, **r5**, ... gespeichert.
Das Ergebnis wird in **r3** gespeichert.

Beispielprogramm auf der Folie:
1. Zeile = cmpd r3,r4, Ergebnis in CR0 speichern.
2. Zeile = cmpd r4,r5, Ergebnis in CR1 speichern.
3. Zeile = UND-Verknüpfung der CR-Bits 0 und 4 (LT-Bit von CR0 und CR1) und Speicherung in Bit 8 von CR. (LT-Bit von CR2).

li r3,0 = addi r3, 0, 0.

2000-07-05

Feststellen einer Ordnung (compare)

cmp cr0, 0, r3, r4 = cmpd r3,r4

cmp cr1, 0, r4, r5 wie voher, nur merke Ergebnis in cr1

crand 8, 4, 0: UND-Verknüpfung der CR-Bits 0 und 4 (LT-Bit von CD0 und Speicherung in Bit 8 von CR (LT-Bit von CR2)

LT	GT	E0	S0	LT											

Beachte: LR enthält die Rücksprungadresse im Hauptprogramm.

Folie 8.21:
lwzx = load word (with zero) und lwz machen hier dasselbe in diesem Beispiel!

lwzx <Zielregister>, <Summand1>

Fibonacci-Zahlen

1,1,2,3,5,8,13,21,...
Fib(0) Fib (1) = 1
Fib (n) = Fib (n-1) + Fib (n-2)

→ elegante Definition, aber die direkte Umsetzung als rekursive Funktion ist denkbar ineffizient. (exponentielle Anzahl von Additionen.)

geschicktere Definition:
Fib(n) = F(n,1,0) mit
F(0,c,p) = c
F(n+1,c,p) = F(n,p+c,c)

z.B. Fib(2) = F(2,1,0) = F(1,1+0,1) = F(0,1+0+1,1) = 2

Befehle im Fibonacci.Beispiel:

bdnz (branch decrement not zero): subtrahiere 1 von [CTR] und teste ob [CTR] @ 0 und in dem Fall springe.
mtctr: move to counter
load **i**mmediate
compare **d**irect
branmch **o**n **f**alse
move **r**egister
branch **d**ecrement **n**ot **z**ero

Speichermanagement im Power-PC

- mehr Speicher **adressierbar** als physisch vorhanden
- logische Adresse → physische Adresse
- Auch auf Cache achten

Virtueller Speicher: Auslagerung auf Festplatte

virtueller Adressraum: 2^{52} bei 32-Bit-Wortlänge, 2^{80} bei 64-Bit-Wortlänge

Strukturierung des Speichers:

1 Segment = 256 MB
1 Seite = 4 KB (1 Seite ist immer komplett im RAM oder komplett nicht!)

Die Zuordnung, welche Seite im RAM ist (und wo), wird in Seitentabellen verwaltet.

Wird eine Seite adressiert, die nicht im Hauptspeicher steht → „Page Fault“ : Seite wird von Festplatte in den RAM eingelagert. („paging“)

Adressübersetzung

2. Block Translation: zusammenhängender Speicher, der nicht ausgelagert wird.

Im Power-PC gibt es neben der virtuellen Adressierung noch 2 Arten:
<u>direkte Adressierung</u>: effektive = physische Adresse.
<u>Blockadressierung</u>: Es werden Bereiche (Blöcke) der Größe 128K bis 256 MB im Speicher reserviert. Kein Paging für diese Speicherbereiche!! (z.B. sehr große Arrays für numerische Behandlung, oder „memory-mapped display“.

UTLB + ITLB dienen allein der Performancesteigerung, da Adresskonvertierung aufwendig ist.

2000-07-05

BAT-Register

im 601 und BAT 0-4
mit jeweils „upper“ und „lower“ Registe

BAT: BATO (U/L), BAT3 (U/L)
(nach 601 jeweils getrennte BAT’s für Daten und Instruktionen

BAT0:

Bit 0-14: effektive Blockadresse |Blockadresse|
Bit 0-14 physikalische Blockadresse

Beachte: Block + Page Translation parallel. Wenn die Blockadressierung erfolgreich dann wird sie genommen.

Cache (PPC 601)

32 KB Cache für Daten und Instruktionen
(Nachfolger haben getrennte D+I Caches). besteht aus:

8 Sets (Einheiten) mit
- je **64 CacheLines** (Cache-Blöcke) mit
- je **2 Sektoren** mit
- je **8 Worten**

Die beiden Sektoren einer Cache Line entsprechen **konsekutiven Worten** im Hauptspeicher:
Ein-/Auslagerung erfolgt **sektorweise**. Adressierung erfolgt über die berechnete **physische Adresse**.

IM CACHE: Bit 0-19: physische Speicheradresse. Bit 20-25 Zeile, Bit 26-31: Spalte.

Wenn die Seitenadresse in der physischen Adresse mit dem entsprechenden Address Tag der Cache Line übereinstimmt, haben wir einen „Cache hit“ (sonst „Cache Miss“).

Bei „Cache Miss“ wird der gesuchte Sektor aus dem Hauptspeicher geladen. Falls kein Platz im Cache ist, wird nach LU-Prinzip ein Sektor aus dem Cache entfernt.

> *(auf Deutsch: ich mache mir also eine Art „Hasching Table“, Bit 20-25 geben die Zeile, Bit 26-31 die Spalte an. Da lege ich die 32-Bit-Adresse ab und den entsprechenden Speicherinhalt. Will ich ein Wort mit denselben Bit 20-25 und 26-31 ablegen, so muss ich auf ein anderes der 8 Sets).*

Cache Arbitration: Bestimmung der **Reihenfolge** bei mehreren gleichzeitigen Anfragen.
(z.B. Konvention: FP immer vor Integer). danach „**Retry**“ für die „Velierer“.

Memory Unit

Folie 9.36
Beachte: Es wird immer ein Sektor auf einmal rausgeschrieben (1 Zyklus). Gelesen werden Sektoren in 2 Zyklen (2x4 Worte)

Snooping

Situation: mehrere CPU's und / oder I/O-Geräte haben Zugriff auf den Hauptspeicher.
PROBLEM: Cache ist ja pro CPU vorhanden. Möchte nun CPU-2 etwas aus dem Speicher holen, kann es sein, dass CPU-1 die Daten verändert hat, diese jedoch noch im Cache der CPU-1 stehen und noch nicht im RAM.
Lösung: „Snooping" = übeprüfen, welche Adressen von den anderen gelesen bzw. geschrieben werden.: die andere CPU snooped also in *meinem* Cache, was los ist!!

SnoopHit on write: jemand anderes will einen Sekor ändern, der in meinem Cache steht.
Folge: Daten im eigenen Cache ungültig.

SnoopHit on Read: wenn Sektor im Cache steht, und er wurde nur hier geändert, schreibe ich schnell die Daten in den RAM zurück – solange muss der andere warten.
ACHTUNG: man muss auch noch in dieser „Snoop Queue" nachschauen: es kann ja sein, dass die Daten schon „auf dem Weg" sind.

Im Cache wird bei jeder Speicheradresse mitgespeichert, in welchem Zustand sich das entsprechende Speichewort befindet:
Modified: Modifiziert, aber noch nicht im RAM
Exclusive: Sektor steht nur in meinem Cache und ist konsistent mit Hauptspeicher.
Shared: Sektor ist konsistent, aber noch andere haben den Sektor im Cache.
Invalid: Sektor nicht im Cache (bzw. Daten sind ungültig)

> *BEISPIEL: jemand „snooped" in meinem Cache, ob er die Daten lesen kann: danach weiss ich, dass meine Daten „shared" sind. „Snooped" er dann bei mir, dass er schreiben will, so weiss ich, dass meine Daten „invalid" sind.*

Andere Architekturen

SPARC-Architektur: entwickelt von Sun in „Open Source". → „High End Workstations"
(SPARC = **S**calable **P**rocessor **Arc**hitecture)
UltraSparc: 64 Bit Adressen. Alles öhnlich wie bei den anderen RISC-Prozessoren:

- **getrennte Caches** für Instruktionen und Daten
- Memory **Management** Unit
- Units zum Laden der **Instruktionen** im Voraus
- NEU: Schnittstelle zum **2nd-Level-Cache** (S-RAM)
- **getrennte** Floating-Point-Units für Mult, Div, Add
- spezielle Hardware (FP) für **Grafik**-Operationen (a la „MMX")
- → neueste Version: **SPARC 9**.

!!! Wichtige Besonderheit der SPARC-Architektur: **sehr viele Register** !!! (> 500 Register) für die Benutzer. („GPR" = General Purpose Register")

IDEE: jedem **Prozeduraufruf** wird ein **Fenster** zugewiesen (für Parameter und lokale Variablen). Wird eine weitere Prozedur aufgerufen, so wird ein neues Fenster erzeugt.
Die **Parameterübergabe** erfolgt in überlappenden Registern.

→ kein Wegschreiben (sichern) der eigenen Register beim Prozeduraufruf → kein Kopieren der Parameter.

Problem: geht nur begrenzt oft (schwierig handhabbar).

THE END