Rechnerstrukturen

Klausurzusammenfassung

erstellt von Sandip Sar-Dessai

1 Das Dualsystem und Schaltfunktionen

Alphabet (feste Wortlänge n): $\Sigma_b^n = \{0, ..., b-1\}^n$ (b-adisch), Konvention: $10 = A$,..., $B = \{0, 1\}$

Umrechnungen: $b \to 10$: $z_{10} = \sum_0^{n-1} z_i b^i$; $2 \leftrightarrow 8$: Umformung von/zu Dreierblöcken; $2 \to 16$: Umformung von/zu Viererblöcken; $10 \to b$ (Vorkommateil): Iteriertes Teilen durch b (erste Teilung ergibt niedrigste Stelle); $10 \to 2$: Iteriertes Multiplizieren mit 2, nacheinander anhängen (an 0,), wenn Produkt > 1.

Rechnerinterne Darstellung von B: $1 = +5V$ und $0 = -5V$.

Boolesche Algebra: distr., kompl. Verband mit kleinstem & größten Element $(B, \cup, \cap, \text{-})$.

Rechengesetze in der Booleschen Algebra: **(1)** Kommutativgesetze **(2)** Assoziativgesete **(3)** Verschmelzung: $(x \cup y) \cap x = (x \cap y) \cup x$ **(4)** Distributivgesetze **(5)** Komplementgesetze $x \cap \overline{x} = 0$, $x \cup \overline{x} = 1$ **(6)** deMorgan $\overline{x \cup y} = \overline{x} \cap \overline{y}$

Schaltfunktion: $F: B^n \to B^m$, **Boolesche Funktionen**: $f: B^n \to B$. $F(x) = (f_1(x), ..., f_m(x))$

Wichtige **Operatoren**: **(1)** Konjunktion: and,$\cdot$ **(2)** Disjunktion: or, $+$ **(3)** Antivalenz: xor, $\oplus$, $\leftrightarrow$ **(4)**Äquivalenz: $\leftrightarrow$ **(5)** Peircescher Pfeil: nor, $\downarrow$, $1 - max$ **(6)** Schefferscher Strich: nand, $\uparrow$, $1 - min$

Einschlägiger Index: Index mit Dualdarstellung $(x_1..x_n)_2$, so daß $f(x_1, ..., x_n) = 1$.

Minterm (zu Index i): Konj. $m_i = a_1 + ... + a_n$ mit $a_k = x_k \Leftrightarrow (i)_2$ an k-ter Stelle 1, sonst $= \overline{x_k}$

Maxterm (zum Index i): Disjunktion $M_i = a_1 \cdot ... \cdot a_n$, mit $a_k = \overline{x_k} \Leftrightarrow (i)_2$ an k-ter Stelle Eins.

DNF von f: Disjunktion der Minterme der einschlägigen Indizes von f (eindeutig).

KNF von f: Konjunktion der Maxterme der nicht-einschlägigen Indizes von f (eindeutig).

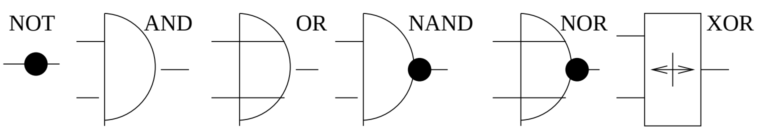
RNF (Ringsummen-NF) von f: Antivalenz der Minterme der einschl. Indizes (eindeutig).

Kodierung eines **Graphen**: $V = \{v_1...v_n\}$. Kanten $K = \{k_1...k_{\binom{n}{2}}\}$ des zugehörigen vollst. Graphen numerieren, Angabe als Bitfeld der vorhandenen Kanten $z \in B^{\binom{n}{2}}$.

Menge von Operatoren **funktional vollst.** $\Leftrightarrow$ alle n-stelligen Fkt. lassen sich damit darstellen.

2 Schaltnetze

Stufenzahl: Anzahl sequentieller Gatterstufen, Gatter: 1, n-Gatter: $n-1$, not: 0



DAG (Directed Acyclic Graph): Graph eines Schaltnetzes (ohne Zykeln), Gatter=Knoten, Leitung=Kante, Knoten ohne hineinführende Kanten=Input, Knoten ohne herausführende Kanten=Output. Bei Knotenbeschriftung "Operator-Schaltznetz", ohne "Verbindungsnetz".

Fan-In/-Out: # in Gatter hinein-/herausführende Leitungen, bei #=0: "kein Fan-In/-Out"

Effizienz: Abhängig von **(1)** Geschwindigkeit∼Gatterstufen **(2)** Größe∼Gatterzahl **(3)** Fan-In/-Out

Komplementfreie Ringsummendarstellung (Reed, Muller): Bei f in RNF (nur XOR-Gatter): [ausgehend von DNF]
(1) + in ⊕ ändern **(2)** $\overline{x}$ in $(x \oplus 1)$ ändern **(3)** Distributiv ausmultiplizieren **(4)** Terme in ungerader Anzahl durch einen ersetzen, Terme in gerader Anzahl streichen

2.1 Multiplexer (MUXe)

Multiplexer (n-MUX, n=Anzahl Steuersignale): besteht aus 2^d Dateninputs x_0..., d Steuersignalen y_1... und einem Output z. y binärkodiert die Datenleitung x, die an z durchgeleitet werden soll. Schaltfunktion (2-MUX): $f = x_0\overline{y_1 y_2} + x_1\overline{y_1}y_2 + x_2 y_1\overline{y_2} + x_3 y_1 y_2$, allg. $z = \sum x_i m_i(y)$.

Realisation:

- Hintereinanderschaltung not, and, or ⇒ Hoher Fan-In bei großen d
- Aus 1-MUXen: $2d$-MUX=$2^d + 1$ d-MUXe ⇒ viele Gatter, aber Fan-In≤ 2; → 44

DeMUX: Ein Input x, d Steuersignale y_i, 2^d Outputs z_i - schaltet x entspr. y auf ein z_i. $z_i = x \cdot m_i(y)$

$n \times m$-**Decoder**: DeMUX mit $x = 1$ ("Enable Signal"). n Steuersignale, $m = 2^n$ Ausgänge, Aktiviert den Ausgang, der durch die Steuersignale binärcodiert wird.

$n \times m$-**Encoder**: MUX ohne z, liefert an y die Binärcodierung des gesetzten x_i (nur ein $x_i = 1$!)

Darstellung Boolescher Funktion $f : (x_1 ... x_n) \to y$ durch (De-)MUXe:

- $(n-1)$-MUX: Steuersignale $x_1..x_{n-1}$, Input $f(x_1...x_{n-1}) = \{0, 1, x_n, \overline{x_n}\}$
- n-MUX ("Hardware-Lookup"): Steuersignale $x_1..x_n$, Input $f(x_1..x_n)$ (1=einschl. Index)
- Decoder/MUX: Decoder mit $\frac{n}{2}$ Eingängen $(x_1..x_{\frac{n}{2}})$, Output weiter an Inputs vom MUX (gem. Schaltfunktion), $x_{\frac{n}{2}+1}..x_n$ als Steuersignale des MUX. z vom MUX liefert f.

2.2 Addiernetze

Halbaddierer (HA): Liefert zu zwei Dualziffern x, y das Ergebnis $R = x \oplus y$ und Übertrag $U = xy$

Volladdierer (VA): Liefert zu x, y, u Ergebnis $R = x \oplus y \oplus u$ und Übertrag $U = xy + (x \oplus y)u$.

Folgende Realisierungen für Addiernetze:

- **Ripple-Carry-Adder** (A4): HA für niedrigste Stelle, danach mehrere VA (u wird durchgereicht). Statt HA ist auch VA mit $u = 0$ möglich. Nachteil: Sehr langsam, da Übertrag nur langsam durchsickert.
- **Carry-Bypass/Carry-Lookahead**: aus mehreren $\widetilde{\mathrm{A4}}$ und einem A_4 (höchste Stelle). $\widetilde{\mathrm{A4}}$ ist A_4 mit zusätzlicher Logik, die sofort Übertrag berechnet und weiterreicht ($U = x_n y_n + \sum\limits_{i<n} \left(x_i y_i \cdot \prod_{j>i} x_j + y_j \right) + u \prod (x_i + y_i)$, dreistufig and/or-and-or)→ 59.
- ***Carry-Select****: Der obere Teil der Ziffern wird 2mal berechnet ($u = 1, 0 \to N, E$), sobald der Übertrag aus der unteren Hälfte bereitsteht, wird zugehöriges selektiert (nach $uE \cdot \overline{u}N$).*

- **Carry-Save** (CSA, für Addition ≥ 3 Zahlen): Der erste CSA besteht aus VA's die für xyu drei Zahlen bekommen. Dabei werden Ergebnisse und Überträge unmittelbar hinausgeleitet. Jeder weitere CSA bekommt dann an der i-ten Stelle: Übertrag u_{i-1} und Ergebnis r_i der letzten Stufe sowie i-te Stelle eines neuen Operanden. In der letzten Stufe werden die CSA-Outputs mit normalem Addierer addiert.$\rightarrow$ 62

3 Vereinfachung von Schaltnetzen, Fehlerdiagnose

Allgemein durch Prinzip der Resolution: $xt + \overline{x}t \equiv t$.

3.1 Karnaugh-Diagramm

Für Funktionen $f : B^n \rightarrow B$, $n \in \{3, 4\}$. **(1)** Ein $4x4$ bzw. $4x2$ Feld wird erstellt, die Spalten werden mit Werten für $x1x2$ beschriftet, die Zeilen mit Werten für $x3$ ($x3x4$), nacheinander: $00 - 01 - 11 - 10$. **(2)** Die Funktionswerte (nur Einsen) werden eingetragen. **(3)** Schließlich werden (möglichst große) $2^s x 2^r$-Blöcke gebildet, die nur Einsen enthalten (auch zyklisch!). **(4)** Für jeden Block wird ein Term gebildet: die Variablen, die konstant bleiben, werden konjugiert (x für $= 1$, $\overline{x}$ für $= 0$) **(5)** Es werden nun Terme so gewählt, daß die Blöcke zusammen alle Einsen überdecken, diese werden disjungiert.

Don't-Cares: Deckt eine Fkt. nicht alle Eingabe ab, können die fehlenden als Don't Cares angemerkt werden (D), sie werden beliebig belegt, so daß möglichst große Blöcke entstehen.

Karnaugh-Ringsummen: Nullen und Einsen werden mit Blöcken überdeckt, Nullen mit gerader Zahl (auch 0), Einsen mit ungerader. Die daraus entstehenden Terme werden mit XOR verknüpft.

3.2 Quine-McCluskey

Sei f in disjunktiver Form $M_1 + ... + M_k$ mit $M_i = m_{i1} \cdot ... \cdot m_{in}$. Die **Kosten** K sind definiert als: $K(M_1 + ... + M_k) = (k - 1) + \sum K(M_i)$ und $K(M_i) = n - 1$. Damit ergibt sich für f in DNF: $K = nk - 1$.

Implikant: M Implikant von f ($M \leq f$) gdw. $M(x) = 1 \Rightarrow f(x) = 1$ (u.a. Minterme sind Impl.)

Primimplikant: M PI $\Leftrightarrow$ keine Verkürzung ist noch Implikant (Terme max. Karnaugh-Blöcke)

Kernimplikant: M KI $\Leftrightarrow$ M ist Bestandteil jeder minimalen disj. Form

Satz: Ist f kostenminimal in DF, dann ist jeder Teilterm ein Primimplikant.

Quine-McCluskey-Verfahren: **(1)** Alle Minterme in Tabelle aufschreiben mit folgenden Spalten: Gruppennummer (=#Negationen), Minterm, Index binär, Index dezimal **(2)** In benachbarten Gruppen Resolution (alle möglichen Paare) anwenden, neue Implikanten hinzfügen (Don't Cares im Binärindex mit * kennzeichnen, alle beteiligten dez. Indizes aufschreiben), nicht benutzte Implikanten übernehmen. **(3)** Iteriere (2) bis keine Änderungen mehr **(4)** Die Primimplikanten werden in Matrix geschrieben: Zeilen=Implikanten, Spalten=beteiligte Minterme; $a_{ij} = 1 \Leftrightarrow$ Implikant i enthält Minterm j. **(5)** Auswahl von Primimplikanten so, daß insgesamt in jeder Spalte eine Eins steht und diese kostenminimal ist.

3.3 Fehlerdiagnose

Typ 1: stuck-at-0, 0-Verklemmung. Schon ein gerissener Draht führt zu Fehler

(1) Ausfalltafel: Für alle Drähte i in einer Funktionstabelle unter f_i eintragen, welches Ergebnis entsteht, wenn nur Draht i gerissen ist **(2)** Ausfall-Matrix: Gleiche Spalten streichen **(3)** Fehlermatrix: f_i ersetzen durch $f_i \oplus f$ **(4)** min. Testmenge: Minimale Anzahl Inputs so finden, daß alle Fehler (Einsen) abgedeckt werden.

Typ 2: schaltungsunabh. Diagnose. Defekt welcher Abhängigkeit der Fkt. f von x_i zerstört

f-Testpaar: Zwei Tests (Tupel) (a, b) mit $a_i \neq b_i \wedge a_j = b_j (i \neq j) \wedge f(a) \not\equiv f(b)$. Minimale Testmenge für f: Menge von Tests, so daß für alle x_i ein $a, b \in T$ existiert mit (a, b) Testpaar, und $T \leq T'$ für alle anderen solchen Mengen T'.

(1) Testpaare aus Fkttabelle ablesen (Paare mit Unterschied nur an 1., 2., ... Stelle & untersch. Fktswert) **(2)** Testmengen: Mengen von Testpaaren, so daß für jede abh. Stelle ein Paar vorhanden ist **(3)** min. Testmenge: Menge mit min. vielen Elementen.

3.4 Hasards

Typ 1: Funktionshasards (bedingt durch f)

Ausgangspunkt: Karnaugh-Diagramm. Für $k = 1..n-2$: Wähle $a_0 = y_1..y_k$ fest für $y_i = 0, 1$, dies entspricht der Auswahl von Bereichen des Diagramms. Suche dort zwei Felder mit: **(1)** die Dualdarstellung differiert nur in einer Stelle [$2x4$: Pferdsprünge, $1x4$: nicht-benachbarte] **(2)** haben gleiche Beschriftung **(3)** ein Feld auf einem Verbindungsweg hat eine andere Beschriftung. Bei Übergang von der ersten in die zweite Stellung kann dann ein Hasard auftreten. Notieren in Tabelle: k, a_0, $f(a_0, a_1) = f(a_0, a_1')$, a_1' (mit anderem Wert), $f(a_0, a_1')$.$\rightarrow$ 85

Typ 2: Schaltungshasards. Entstehen durch verschiedene Signallaufzeiten und spätere Rekonvergenz

Satz: Sind alle Primimplikanten beteiligt, tauchen keine Schaltungshasards auf.

4 Schaltwerke (synchrone Schaltnetze)

Delay: bestet aus Vorspeicher V und Speicher S. In der Arbeitsphase wird Input in V gespeichert, Output auf S gesetzt, in der Setzphase (Taktsignal) wird Inhalt von V nach S übertragen. Realisierung: Latches/FlipFlops.

Register: Zusammenschaltung mehrerer Delays.

Addierwerke: Register: Akku A (Summand,Ergebnis), Puffer P (Summand) und Link L (Übertrag)

- **Parallel-Addierer**: jedem HA/VA ist der entsprechende A- und P-Delay vorgeschaltet, und das A wieder nachgeschaltet. Letzter Übertrag geht in L.
- **Serien-Addierer** (nur ein VA): Die Werte werden an den VA durchgereicht (mit dem niedrigsten zuerst) - das Ergebnis stellt sich hinten an, die anderen rücken vor. Der Übertrag wird in L gespeichert und bei nächster Rechnung wieder eingespeist. Nach n-Zyklen steht die Summe im Akkumulator.

- **von-Neumann-Addierwerk** (aus HAs): zusätzl. Statusdelay S. Im ersten Schritt wird jeweils $A_i + P_i$ berechnet, das Ergebnis geht in A_i, der Übertrag in P_{i+1} (bzw. in L). P_0 wird danach mit Null initialisiert. Iteration, bis kein Übertrag mehr entsteht. Solange Übertrag existiert, wird S mit Eins gefüllt, danach mit Null (=Rechnung fertig)$\rightarrow$ 116. max. n Schritte, durchschn. $ld n$. Welches Signal in AP geschrieben wird, hängt von S ab (I=Input, R=Result$\Rightarrow$ $SR + \overline{S}I$ wird eingespeist).
  Addier-Subtrahierwerk: Komplementierer zw. Addierwerk u. P, eingeschaltet bei Subtr.

4.1 Multiplikation

Ähnlich Addierer: Akkumulator A (Akku), Puffer X (Multiplikand), zusätzl. Y (Multiplikator) und Standardaddierer. X, Y müssen Shift unterstützen, A muß Stellenzahl $|X| + |Y|$ haben.

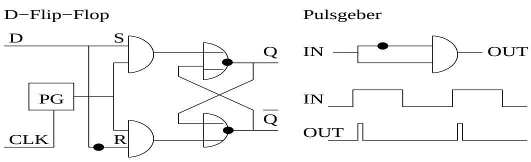
Algorithmus:

Initialisiert wird $A = 0$, X =Multiplikand, Y =Multiplikator. In jedem Schritt wird dann:

1. X hat 1 ganz rechts: Addiere Y zu A (Ergebnis in A speichern)
2. Y um eine Stelle rechtsshiften, X um eine Stelle linksshiften (Nullen nachreichen)
3. Multiplikation ist fertig, wenn $Y = 0$.

Zusätzliche Logik bestimmt das Vorzeichen aus höchstem Bit.

4.2 Latches und Flip-Flops



SR-Latch: Schaltung mit zwei Eingängen S (Set) und R (Reset) sowie zwei Ausgängen Q und $\overline{Q}$. Das Latch hat zwei stabile Zustände $Q = 0$ und $Q = 1$ ($\overline{Q}$ komplementär). S und R müssen ebenfalls komplementär sein, sonst stellt sich am Ausgang ein zufälliger Wert ein. Mit $S = 1$ wird $Q = 1$ gesetzt, mit $R = 1$ wird $Q = 0$ gesetzt; das Latch "merkt" sich den letzten Wert. Zusätzlich sind oft zwei weitere Eingänge CLR (Clear) und PR (Preset) vorhanden, um Q zu löschen bzw. zu setzen (bei Registern zusammengeschaltet$\Rightarrow$löscht/setzt alle).

Getaktetes Latch: Ein Latch, bei dem S und R mit Und-Gattern mit einem Taktgeber verbunden sind. Dadurch ist eine Zustandsänderung nur möglich, wenn das Taktsignal hoch ist.

D-Latch: Latch mit nur einem Eingang D (Data). Dieses Signal wird an S weitergeleitet, und $\overline{D}$ an R; dadurch wird verhindert, daß S und R gleiche Signale erhalten. Als Ergebnis wird Q auf D gesetzt.

Pulsgenerator: Schaltung mit einem Ein- und Ausgang. Das Eingangssignal wird doppelt an ein AND-Gatter geleitet, wobei eine der Leitungen invertiert wird. Durch die Invertierverzögerung kommt es bei beliebig langem hohem Eingangssignal nur anfänglich zu einem kurzen Ausgangssignal (ca. $10psec$).

Flip-Flop: getaktetes Latch, bei dem zwischen Clock und den And-Gattern ein Pulsgenerator zwischengeschaltet wird. Dadurch sind Zustandsänderungen nur während der Taktflanken möglich.

Register: Zusammenschaltung mehrerer D-FlipFlops. Ein-/Ausgänge: V_{cc} (Spannungsquelle), GND (Erdung) und CLR (Clear), die parallel auf alle FlipFlops geschaltet werden, sowie für jedes Flip-Flop ein Q und ein D. Register schalten auf absteigender Flanke, da dem Takt ein Inverter zur Verstärkung nachgeschaltet wird.

$2^m \times n$**-Speicher**: Speicher aus 2^m Zeilen mit je n FlipFlops. Kontakte:

- I_j (Data In, $j = 0..n-1$): Für jede Spalte ein Eingang, der an alle D-Kontakte der Spalte weitergegeben werden. Die Taktung wird nur geschaltet, wenn $CS = 1$ und $RD = 0$ ist (nur dann ist Speicherung möglich).
- Q_j (Data Out, $j = 0..n-1$): Für jede Spalte ein Ausgang. Alle Q-Kontakte der Spalte werden mit OR zusammengeschaltet und an den Ausgang gegeben. Der Ausgang wird nur geschaltet (mit Schaltern), wenn $CS = RD = OE = 1$.
- A_i (Adress, $i = 0..m-1$): Signale für Adressierung einer Zeile (im Binärcode), wird mit der Taktung verknüpft, so daß nur eine Zeile angesprochen werden kann.
- CS (Chip Select): Schreiben/Lesen nur möglich, wenn $CS = 1$.
- RD (Read): Selektiert Schreiben ($RD = 0$) oder Lesen ($RD = 1$).
- OE (Output Enable): Lesen nur möglich, wenn $OE = 1$ (mit $RD = CS = 1$).

Alternativ dazu kann statt RD auch WE (Write Enable) mit zu RD komplementärer Bedeutung verwendet werden. Außerdem wird statt X oft $\overline{X}$ angegeben, um anzuzeigen, daß bei absteigender (und nicht aufsteigender) Taktflanke geschaltet wird.

RAM: Random Access Memory; Speicher mit wahlfreiem Zugriff, d.h. jede Adresse kann einzeln adressiert werden.

SRAM (Static RAM): Zeilen/Spalten-Speicher (wie oben). Sehr schnell ($< 10ns$).

DRAM (Dynamic RAM): Ähnlich SRAM, verfügt aber über nur einen Ausgang D, so daß nur eine Speicherzelle geschaltet werden kann. Dies geschieht über zwei zusätzliche Eingangssignale CAS (Column Adress Stroke) und RAS (Row Adress Stroke); ein Signal auf einer der Leitungen adressiert die an den Adresseingängen spezifizierte Zeile bzw. Spalte (zwei Zyklen). Hohe Speicherdichte, geringe Kosten; langsamer als SRAM (ca. $50ns$), außerdem müssen die Speicherinhalte aufgefrischt werden.

EDORAM (Extended Data Output RAM): Verschränken der beiden Zugriffszyklen beim DRAM.

SDRAM (Synchronous DRAM): Kombination aus SRAM und DRAM.

5 Rechnerarithmetik

5.1 Integers

Darstellungen im Rechner (H=linkes Bit, L=Rest der Dualdarstellung):

- **Vorzeichen/Betrag**: [V][Betrag] mit $- : 1, + : 0$. Bereich $-(2^{n-1}-1)..2^{n-1}-1$. Nachteile: $+0 \neq -0$
  $\rightarrow 2$: $H = 1$ bei negativer Zahl, sonst $= 0$; $L =$Dualdarstellung von n
  $\rightarrow 10$: $H = 1 \Rightarrow$Faktor -1 (negative Zahl); $(L)_{10}$ ist Betrag.
- **Einerkomplement** K_1: Faktor -1 ist Komplementierung. Bei negativen Zahlen ist ersten Bit Eins, Bereich wie bei Vorzeichen/Betrag (Umordnung der neg. Zahlen); $+0 \neq -0$ bleibt.
  $\rightarrow 2$: Bei negativer Zahl dualen Betrag komplementieren, sonst $(n)_2$
  $\rightarrow 10$: $H = 1 \Rightarrow$Faktor -1, Komplementieren; $(L)_{10}$ ist Betrag.

- **Zweierkomplement** K_2: Wie Einerkompl, aber Bereich $-2^{n-1}..2^{n-1}-1$. Eine Null!
$\rightarrow$ 2: negative Zahl: dualen Betrag kompl. und Eins addieren (Überträge ign.), sonst $(n)_2$.
$\rightarrow$ 10: $H = 1 \Rightarrow$Komplementieren, in Dezzahl, Eins addieren, Faktor -1; sonst $(L)_{10}$.

- **BCD** (Binary Coded Decimal): Jede Dezimalziffer wird in einen 4er-Binärblock umgewandelt. 0..9 in 0000..1001, +: 1010, −: 1011. Die restlichen Kombinationen bleiben ungenutzt.

Arithmetik

- **V/B**: Getrenntes Subtrahierwerk nötig. U.u. Termumformung ($-a-b=-(a+b)$).

- **Komplementdarstellungen**: **(1)** Negative Zahlen entsprechend komplementieren, Form $a+b$ schaffen **(2)** Zahlen wie gewohnt addieren **(3)** Einerkomplement: bei Übertrag ist eine Eins zu addieren. Zweierkomplement: Übertrag=Overflow, bei der Berechnung wird Übertrag ignoriert.

- **BCD** (Addition): Es wird (von rechts beginnend) normal addiert. Taucht ein Übertrag auf, muß 0110 addiert werden. Gleiches gilt, falls eine Zahl ungültig wird. Vorzeichen wird nicht mitberechnet.$\rightarrow$ 140.

5.2 Floats

Zwei Darstellungen:

- **Festpunkt**-Darst: Es wird 2^k für das höchste Bit definiert, der Stellenwert der weiteren Stellen nimmt um Faktor 2 ab. Nachteil: signifikante Stellen gehen verloren, da nach Vorkommastellen/Mantisse getrennt wird.

- **Gleitkomma**-Darst: Höchstes Bit ist Vorzeichen, danach folgt Mantisse, schließlich Exponent im Zweierkomplement (jeweils feste Größe). Die Exponentbasis wird festgelegt auf b (Angabe des Exponenten IMMER zur Basis b)
$\rightarrow$ 2: Ausgehend vom Betrag den Vorkommateil wie gewohnt übersetzen. Danach Mantisse: Zunächst $r = Mantisse$. Dann nacheinander $a = 2^{-i}$ ($i = 1, ...$) bilden. Falls $a < r$, $r = r - a$ und 1 der Dualdarstellung (rechts) anhängen, ansonsten 0 anhängen. Iterieren, bis insg. soviele Stellen wie Mantisse (Vorkomma+Nachkomma) entstanden sind oder $r = 0$. Danach solange rechtsshiften, bis die Zahl $0, n...$ lautet. Dann ist Vorkomma=1 wenn negative Zahl, Exponent=geshiftete Stellen, Mantisse=n....
$\rightarrow$ 10: Vorzeichen merken. Danach Komma um soviel Stellen nach rechts verschieben, wie Exponent angibt. Berechnung wie normal (erste Stelle vor dem Komma zählt b^0), Vorzeichen anmultiplizieren.

Zur Gleitkommadarstellung:

- Bei Basis 8, 16 beachten, daß in Dualdarstellung eine Stelle drei/vier Dualstellen entspricht!

- Vor Addition/Subtr. wird Exponent angeglichen (kleinerer erhöht), danach Rechnung normal, ggf. Normalisierung (d.h. auf $0, n...$ bringen). U.U. Rundungsfehler!

- Mul/Div: Vorzeichen wie gewohnt, Exponenten addieren/subtr., Mantisse multiplizieren

- **Hidden Bit**: Bei normalisierter Darstellung ist erstes Bit der Mantisse immer Eins und kann daher weggelassen werden für erhöhte Genauigkeit. Zu beachten ist dann, daß $0,0$ NICHT 0 ist (sondern $0,10$).
Nulldarstellung: Per Konvention ist die Null $V = 0, E = 0$ bei beliebiger Mantisse.

- **Biased/Excess-2^{g-1}**: Bei g Bits für Exp. wird der Exponent nicht in K_2 dargestellt sondern als $d' = d + 2^{g-1}$ (Addition auf alle Zahlen; Exponenten-Vergl. ist nun einfacher).

Die FP-Arithmetik (Exponentenvergleich, ggf. Mantissenshift des Operanden mit kleinerem Exponenten und dadurch Exponentenangleich, Ausführung der Operation und Normalisierung des Ergebnisses) wird von der **FPU** übernommen. Sie versucht auch durch sehr lange Register, Rundungsfehler zu vermeiden.

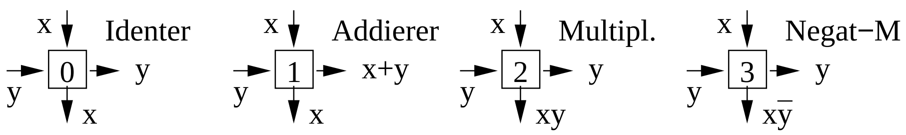
5.3 Alphanumerische Zeichen

Die Zeichen werden entsprechend einer Codetabelle als Zahlen dargestellt. Varianten:

- **6-Bit-Code**: 6 Bits (26 Buchstaben, 10 Ziffern, 28 weitere): wird selten genutzt, weil zuwenig Zeichen (insb. keine Unterscheidung groß/klein), Fehlererkennung bei Datenübertragung schwer, da Tabelle ausgenutzt wird.
- **ASCII** (Am. Standard Code for Information Interchange): 8 Bit (höchstes ist Paritätsbit P) mit 128 Zeichen. P ist Eins, wenn Anzahl restlicher Einsen gerade, sonst Null (gerade Par.).
- **EBCDIC** (Extended Binary Coded Decimal Interchange Code): 8 Bit, von IBM entwickelt. Ziffern haben die Darstellung $1111BCD$.

6 Programmierbare Logische Arrays (PLA)

Gittergeflecht aus Leitungen, an deren Kreuzungen folgende Bausteine sitzen:



Das Gittergeflecht wird unterteilt in **zwei Ebenen**: UND-Ebene (oben), die nur die Bausteine [0],[2],[3] enthält, und die ODER-Ebene (unten), die nur die Bausteine [0],[1] enthält.

Realisierung Boolescher Fkt. von $f : B^n \to B^m, f(x_1..x_n) = (y_1...y_m)$ (f in DF):

- Von oben wird Eins eingespeist
- Von links werden x_1 bis x_n in die UND-Ebene eingespeist, in die ODER-Ebene Nullen
- Rechts aus der ODER-Ebene kommen $y_1...y_m$
- Jede Spalte entspricht einem Teilterm der DF; an Kreuzung mit x wird [0] gesetzt, falls x nicht beteiligt ist im Term, [2], falls x beteiligt ist, [3], falls $\overline{x}$ beteiligt ist.
- Jede ODER-Zeile entspricht einem Funktionswert: an Kreuzung mit Term M wird [0] gesetzt, falls M nicht beteiligt ist, sonst [1].

Besteht die DF aus t Termen, ist die PLA-Dimension $(m + n) \times t$.

PLA-Matrix: Matrix, die nur die verwendeten Bausteine enthält.

Satz: Speist man neben x auch $\overline{x}$ ein, kommt man ohne [3] aus.

Punktorientierte Darstellung: Wird nur [0],[1],[2] verwendet, trägt man in das Schaltdiagramm nicht die Bausteine ein, sondern nur noch einen Punkt, falls an der Kreuzung [1] bzw. [2] verwendet wurde.

PAL (Programmable AND-Logic): ODER-Ebene ist fest verdrahtet (bspw. je 8 Spalten pro Outputzeile).

programmierbarer PLA: Für jede Kreuzung kann durch zwei Steuersignale der Bausteintyp bestimmt werden.

6.1 Faltung

Reduktion der Zeilenzahl, indem sich zwei Variablen eine Zeile teilen (eine von links, eine von rechts hineingeführt), zwischen den beiden Bereichen wird die Zeile durchgetrennt. Die Variablen, die sich eine Zeile teilen können, dürfen nicht gemeinsam in einem Term vorkommen. Bessere Ergebnisse oft durch Spaltenpermutation. Spezialvarianten:

- **Blockfaltung**: Es gibt keine Spalte, so daß in einer Zeile der Input von rechts und in einer anderen von links kommt. Vorteil: Trennung an einer Stelle möglich über alle Zeilen.
- **bedingte Faltung**: Vorgabe, welche Symbole rechts und links anliegen. Ggf. mehr Zeilen nötig.
- **bedingte Blockfaltung**: Kombination aus beiden anderen Varianten (ggf. Zeilen mit nur einem Signal nötig).

6.2 Anwendungen der PLAs

ROM: Fest verdrahtete UND-Ebene (Adreßdecodierer), der die zum Input gehörende Spalte aktiviert [x_1 Highbit]. In der ODER-Ebene (ROM) sind dann die Daten vermerkt, die zu dieser Adresse gehören (werden über y_i ausgegeben).

Mikroprogrammierung: Ein Teil der (oder alle) Outputs werden als ein Teil der (bzw. für alle) Inputs wieder neu eingespeist (mit zwischengeschalteten Delays). Die zurückgeführten Outputs bestimmen die Next-State-Logic, die restlichen die Output-Logic.

PLAs mit Delays heißen "**integrierte PLA**"

7 Von Neumann - Rechner

Grundbausteine: **(1) CPU** (Central Processing Unit) besteht aus Daten- und Befehlsprozessor und verarbeitet Instruktionen **(2) Speicher** bestehend aus RAM und ROM für Programm und Daten gemeinsam **(3) I/O-Einheit** als Schnittstelle mit Peripherie **(4) Busse** für Verbindung zwischen den Einheiten

Klassifikation:

- Personal Computer / Workstation / Mainframe
- Durchsatz in **MIPS** (Million Instr. per Second), **FLOPS** (Floating Pt Ops per Second)
- **CISC** (Complex Instruction Set Computer), **RISC** (Reduced ~)
- **Pipelining** (Unterteilung eines Befehlszyklus, paralleles Bearbeiten), **Superpipelining** (tiefere Pipes)

- Superskalararchitektur (Mehrere Pipes)

RISC-Eigenschaften: **(1)** wenige Instruktionen **(2)** Befehlsdecoder/Steuerwerk fest verdrahtet **(3)** möglichst ein Takt pro Instr., CPI (Clocks/Instruction)=1 **(4)** Speicherzugriff nur load,store **(5)** viele Register **(6)** festes Instruktionsformat **(7)** optimierende Software, Compiler

7.1 CPU

Besteht aus Datenprozessor und Befehlsprozessor.

Datenprozessor: besteht aus **ALU** (Arithmetic Logic Unit, Recheneinheit) und folgenden Registern: **A** (Akkumulator, auch General Purpose Reg. GPR), **MR** (Multiplikator-Reg.), **L** (Link-Register) für Überträge, **MBR** (Memory Buffer Reg.) als Puffer für Speicherkommunikation. Moderne Rechner haben oft mehr Register.

Befehlsprozessor: Befehlsentschlüsselung und Ausführung, besteht aus: **IR** (Instruction Register, Befehlsregister) für den aktuellen Befehl, **MAR** (Memory Adress Reg., Speicheradreßreg.) mit der Adresse der nächsten zu verwendenden Adresse (Operand), **PC** (Program Counter, Befehlszähler) mit Adresse des nächsten Befehls.

Register werden nach zeitlichen Kontext interpretiert, grundsätzlich zwei **Phasen**: **(1) Fetch**: Inhalt von PC wird nach MAR geschrieben (muß Befehl sein), Adresse wird über MBR nach IR geholt, Befehl wird dekodiert, ggf. werden Operanden bereitgestellt und PC wird aktualisiert (Sprungbefehl: neu laden, sonst PC=PC+1) **(2) Execution**: Befehlsausführung und Initialisierung des nächsten Fetch.

Arbeitet nach dem SISD-Prinzip (Single Instruction Single Data), Befehlsfolge ist Binärzahl in festem Code (Maschinencode). Programmiert wird in Assembler, der Mnem-Code in Befehlscode übersetzt. Die Register MBR, MAR und IR bleiben dem Programmierer verborgen. Da jeder Befehl den Akku nutzt, muß dieser nicht unbedingt angegeben werden.

Zeitproblem entsteht durch Kommunikation mit dem Speicher (vN-Flaschenhals, **bottleneck**)

7.2 Speicher und Busse

CPU-nah befindet sich schneller Cache-Speicher (SRAM) für Ausschnitte des Hauptspeichers RAM. Sinnvoll weil 90% der Zugriffe auf 10% der Daten erfolgen. Oft in mehrere Levels gegliedert.

RAM: Folge einzeln adressierbarer Zellen. Kenngrößen: Breite (Länge der kleinsten adressierbaren Einheit in Bit) und Länge (Gesamtspeichergröße in Byte). Maximum hängt von MBR (Länge=Speicherzellengröße) und MAR (Länge=duale Adressenlänge) ab.

ROM: auch PROM (Programmable ROM), EPROM (Erasable PROM), EEPROM (Electrically EPROM).

Bustypen: seriell (ein Bit pro Takt), parallel (mehrere Bits pro Takt), laufen mit zentraler Synchronisation. Normal separater Datenbus (Breite wie MBR) und Adreßbus (Breite wie MAR), da Breite i.d.R. verscheiden. Häufig noch mehr Busse.

Spezielle Busse: ISA (Industry Standard Architecture) 32Bit 8,33MHz und PCI (Peripheral Component Interconnect) 64Bit 66MHz

7.3 I/O-Organisation

Controller-Typen: **(1) SIO**: seriell, **(2) PIO**: parallel zu Speicher und I/O-Devs, seriell zur CPU **(3) UART** (Universal Asynchronous Receiver and Transmitter): pararallel zu Speicher, seriell zu Endgeräten und I/O-Devs.

Controller verfügen über **Statuswort** (lesbar von CPU), das angibt, was benötigt wird.

I/O-Typen:

- **programmierter I/O**: CPU liest Controller-Statuswort in regelmäßigen Abständen, Flag zeigt an, ob CPU benötigt wird. CPU steuert Kommunikation.
- **Interrupt-gesteuerter I/O**: Controller sendet Interrupt, daß Statuswort gelesen werden soll. Phasen: **(1)** Endgerät bereit zur Datenübertragung an Speicher **(2)** Controller sendet Interrupt **(3)** CPU sendet Startsignal **(4)** Controller empfängt Daten und speichert sie in internem Puffer **(5)** Interrupt **(6)** CPU überträgt Daten in Speicher **(7)** Wiederholen bis Datenübertragung vollständig.
- **DMA** (Direct Memory Access)-gesteuerter I/O: wie Interrupt-gesteuert, aber Controller überträgt mittels DMA-Controller ohne CPU-Hilfe Daten zum Speicher. Controller hat Vorrang (bei Inkonsistenz macht CPU Wartezyklen, cycle stealing)

Interrupt-Einteilung: intern/extern (in der CPU bspw. div/0 oder außerhalb bspw. I/O), maskierbar (falls vorübergehend abschaltbar), Priorität

Reaktion auf Interrupt: CPU unterbricht Arbeit, Interrupthandler wird gestartet, Weiterführung

8 RISC-Rechner (PowerPC 601)

Register im Big-Endian-Modus (Bit 0 ist links)

CPU-Einheiten: **(1) BPU** (Branch Processing Unit) interpretiert Befehle und führt Sprünge aus **(2) IU** (Integer Unit) **(3) FPU** (Floating Point - Unit)

Branch Processing Unit: Kann u.U. parallel zur IU/FPU Sprungbefehle abarbeiten, gibt sonst an IU/FPU weiter. Register: **(1) LR** (Link-Register), 32Bit, bspw. für Rücksprungadressen, **(2) CTR** (Count Register), 32 Bit, Zähler für Schleifenimplementierungen, **(3) CR** (Condition Register), 4x8Bit (CR0..7), CR0 für Integerops, CR1 für FP-Ops, CR0/1-Flags: [0] Negative Flag LT [1] Positive Flag GT [2] Zero Flag EQ [3] Summary Overflow SO

8.1 PowerPC

zusätzliche CPU-Einheiten: **Instruction Unit** (enthält BPU), **MMU** (Memory Management Unit), **MU** (Memory Unit)

zusätzliche Register: Multiplikatorregister **MQ** (64Bit), Realtime Clock Lower/Upper **RTCL/RTCU** oder (ab 602) Time Base **TBUL/TBUH**. Auf Uhrzeit nur lesender Zugriff.

Supervisor-Register (Betriebssystem/Hardware): 16 **SR** (Segment-Reg), 8 **BAT** (Block Adress Translation Reg, Numerierung 0U,0L...3U,3L), 1 **TSDR** (Table Search Description Tag) sowie ein **MSR** (Machine State Register). MSR: [16] EE: External Exception Enable, [17] PR: Priviledge Level (0=Usr/Supervisor, 1=User) [18] FP: FP Enable.

Instruction Unit: Holt gleichzeitig bis zu 8 Befehle aus dem Cache (MU) in die Instruction Queue. Queue-Register (8x32Bit): Q0..Q3 werden nach Sprung- und Rechenbefehlen durchsucht,

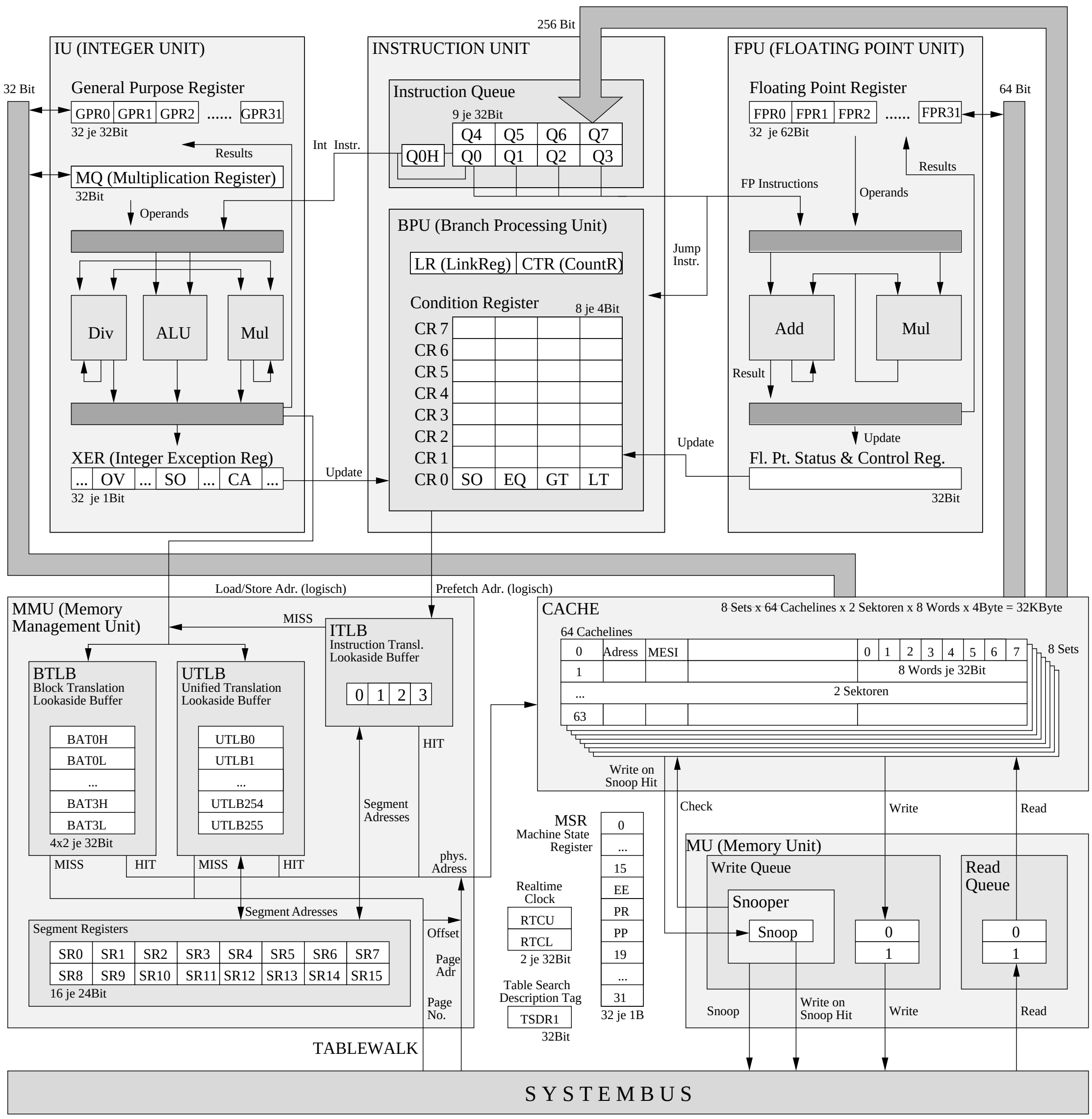


Abbildung 1: CPU des PowerPC 601 (schematisch)

Weitergabe an BPU,FPU,IU; Q4..Q7 dienen als Puffer. Zusätzlich Q0Hold, um Int-Befehle zu speichern, falls IU nicht bereit. Pipeline-Stufen: Fetch (Befehl holen), Decode (Befehl decodieren, Operanden holen), Execute (Ausführen), Writeback (Resultat speichern)

IU: i.d.R. CPI=1, erledigt auch Ber. der Speicherzugriffe, hat ALU und Mul,Div-Einheiten. 32 **GPR** sowie **XER** (Integer Exception Register) mit folgenden Bits (u.a.): [0] SO [1] OV (Overflow) [2] CA (Carrybit). Im Gegensatz zu OV muß SO explizit auf Null gesetzt werden. CA trägt Übertrag.

FPU: 32 64-Bit Register (**FPR0..31**) sowie ein FPSCR (FP Status and Control Reg) für Ausnahmeinformationen. mehrere FP-Formate.

8.2 601-Befehle

Befehlsformat (31 Bit): [0..5] Primary Opcode [6..10] rD (Destination) [11..15] rA (Source A) [16..20] rB (Source B) [21] OE-Flag [22..30] Ext. Opcode [31] Rc-Flag

OE (Overflow Enable): Falls Eins, werden OV und SO in XER gesetzt (mnemo)
Rc: Falls Eins, wird CR0 aktualisiert (mnem.)

SIMM (Signed Immediate): ganze Zahl wird zusammen mit Befehl angegeben (steht nicht im Register), Zahl (16Bit) steht in [16..31].

Formate: mnem[o.] rD, rA, rB (normal) oder mnem[o.] rD, rA, SIMM (bei SIMM)

8.2.1 Load- und Store-und Move-Befehle

Adressierungsarten für Load/Store: **(1)** Reg Indirect with Immediate Index: Adresse=rA+SIMM **(2)** Reg Indirect with Reg Index: Adresse=rA+rB **(3)** *Reg Indirect: Adresse=rA*

Kopiergrößen Load/Store: Byte (b), Halbwort (h), Wort (w). Ablage erfolgt rechtsbündig in den Registern, wobei fehlende Stellen mit Nullen gefüllt werden.

Bei FP-Load/Store: Unterscheidung zwischen Speicher-Formaten single precision ([0]V [1..8]E [9..31]M) und extended precision ([0]Vz [1..11]E [12..63]M).

- `lbz rD, SIMM` (load byte with zero, RI/II): Ein Byte von Adresse rA+SIMM nach rD laden
- `lbzx rD, rA, rB` (lbwz indexed, RI/RI): Ein Byte von Adresse rA+rB nach rD laden
- `lbzu rD, SIMM` (lbz with update, RI/II): Ein Byte von Adresse rA+SIMM nach rD laden, danach rA auf rA+SIMM setzen
- `lbzux rD, rA, rB` (lbzu indexed, RI/RI): lbzu mit Quelladresse rA+rB
- `stw rS, SIMM` (store word, RI/II): Ein Word von rS an Adresse rA+SIMM speichern
- `stwx rS, rA, rB` (stw indexed, RI/RI): Ein Word von rS an Adresse rA+rB speichern
- `stwu rS, SIMM` (stw with update, RI/II): Nach stw Register rA aktualisieren
- *`lfps, lfpd` ...: FP-Register laden (Speicher: Format single/double)*
- *`sfps, sfpd` ...: FP-Register speichern (Speicher: Format single/double)*
- `mtctr rS` (move to CTR): Inhalt von rS nach CTR kopieren
- `mfctr rD` (move from CTR): Inhalt von CTR nach rD
- `mtlr rS` und `mflr` (move to/from LR): analog für LR

8.2.2 Arithmetik

- `add` (31, 266): rD=rA+rB
- `addc` (31, 10), Add Carrying: rD=rA+rB, falls OE=1, wird XER[CA] ggf. Eins, sonst CA=0
- `adde` (31, 139), Add Extended: rD=rA+rB+XER[CA]
- `addi` (14), Add Immediate: rD=rA+SIMM, bei 32Bit-Konv. SIMM in [16..31]
- `addis` (15), addi Shifted: rD=rA+SIMM, bei 32Bit-Konv SIMM in [0..15]
- *`addic` (12), addi Carrying: rD=rA+SIMM+CA.*
- `addme rD, rA`; Add Minus One Extended: rD=rA+XER[CA]-1
- `addze rD,rA`; Add to Zero Extended: rD=rA+XER[CA]
- `subf`, Subtract From: rD=rB+rA' mit rA'=2er-Komplement von rA
- `mull`, Multiply Low: rD=rA*rB, rD enthält untere 32Bit des Ergebnisses
- `mulh`, Multiply High: rD=rA*rB, rD enthält obere 32Bit des Ergebnisses
- *`and` (`or`, `xor`, `nand`, `nor`, `eqv`): Logische Operationen*
- `srawi`, Shift Right Algebraic Word Immediate (SIMM-Op): Rechtsshift um SIMM Bits, wobei Vorzeichenbit nachgeschoben wird. Fällt Rechts eine 1 heraus und ist die dargestellte Zahl negativ, wird CA auf Eins gesetzt.
- `cmpd rA,rB` (compare direct): CR0[LT],[GT],[EQ] werden entsprechend gesetzt. XER[SO] wird nach CR0[SO] kopiert. rA und rB werden als ganze Zahlen im 2er-Komplement interpretiert.
- `cmpdi rA, SIMM`: statt rB wird SIMM benutzt.
- *`fadd`, `fsub`, `fmul`, `fdiv`, `fneg`, `fabs`, `fnabs` (negative absolute): FP-Arithm.*

Mit `addc`, `adde` kann 64-Bit Addition (2x 32Bit) implementiert werden.

8.2.3 Sprungbefehle

- `b target` (branch): Unbedingter Sprung um target von akt. Adresse
- `ba target` (branch to adress): Unbedingter Sprung zu target
- `blr` (branch to LR): Unbedingter Sprung zur Adresse in LR
- `bt eq, target` (branch if true): b target falls CR[EQ]=1
- `bfa gt, target` (branch if false): ba target falls CR[GT]=0
- `btl eq, target` (branch if true, update LR): bt eq, target und in LR die Adresse des nächsten Befehls (Sprungadresse+4 Bytes) ablegen
- `bdnz target` (decrement, branch if non-zero): Dekrementiere CTR um Eins, führe dann b target aus, wenn CTR$\neq 0$
- `bdz target` (decrement, branch if zero): wie bdnz, aber springe nur, wenn CTR$= 0$

8.2.4 Prozessor-Kontroll-Befehle

- `sc` (system call): Sichert die Adresse des nächsten Befehls in SPR0 und MSR in SPR1, danach Abgabe der Kontrolle ans OS

8.3 Assembler

Arbeitet mit mnem-Codes (statt mit Opcodes). Besitzt einen LC (Location Counter), der Maschinenbyte-Codes zählt. Namen und Adressen werden bei Compilation durch den Wert des LC an dieser Stelle ersetzt; dazu werden alle Symbole intern in der Symboltabelle gespeichert.

8.4 Speichermanagement

Man unterscheidet drei **Speichertypen**:

- **Hintergrundspeicher**, eingeteilt in Seiten (pages) fester Länge.
- **Hauptspeicher**: Enthält Ausschnitte des Hintergrundspeichers, ebenfalls eingeteillt in Seiten (Frames), wobei die Seitentabelle (Page Table) vermerkt, welche Hintergrundspeicherseiten sich im Hauptspeicher befinden und ggf. an welcher Adresse.
- **Cache**: Speichert die zuletzt vom Hauptspeicher zur CPU übertragenen Daten (s.u.)

Adress Translation (unter Rückgriff auf Seitentabelle): Befindet sich die gewünschte Seite im Hauptspeicher, so wird aus der in der Seitentabelle vermerkten Adresse und dem Offset die reale Adresse gebildet. Falls nicht, wird ein **Page Fault** ausgelöst; dies bewirkt, daß die entsprechende Seite in den Hauptspeicher gelesen wird. Ist dort kein Platz mehr, wird die älteste Seite im Hauptspeicher (Least Recently Used, **LRU**) u.U. vorher zurückgeschrieben (falls nötig) und die neue an diese Stelle geladen.

8.4.1 MMU (Memory Management Unit)

Übersetzt logische in physikalische Adressen. Man unterscheidet folgende **Adreßübersetzungen**:

- **Block Adress Translation** (für größere Speicherbereiche): In der **BTLB** (Block Translation Lookaside Buffer) werden die zuletzt verwendeten vier Adressen gespeichert; Für jede dieser Adressen wird in BATiU die log. Blockadresse [0..14] und die Blocklänge [15..31] gespeichert, in BATiL die physische Adresse [0..31]. Schritte bei Übersetzung:
  1. PA[0:14] aus BTLB (suche LA[0:14] = Seg.Nr. und 11 Bit Page Index)
  2. PA[15:31] aus LA[15:31] (5 Bits des Page Index und Offset = 128K)
- **Page Adress Translation**: Im **UTLB** (Unified Translation Lookaside Buffer) werden die letzten 256 genutzten Adressen gespeichert, zusätzlich speziell für die Instruction Unit die vier letzten dort gebrauchten in der **ITLB** (Instruction ~). Schritte:
  1. VA wird ermittelt; VA[0:23] aus Segmentregister SR(LA[0:3]), VA[24:51] ist LA[4:31] (Page Index + Offset)
  2. PA wird ermittelt; PA[0:19] aus UTLB/ITLB/Page Table, PA[20:31] aus LA[20:31] (Offset)

Adresstypen: (1) Logische Adresse (LA, 32Bit): [0:4] Segment-Nr. [5:19] Page Index [20:31] Offset, **(2)** Virtuelle Adresse (VA, 52Bit): [0:23] VS-ID, [24:39] Page Index, [40:51] Offset, [0:39] Virtual Page Number (VPN), **(3)** Physikalische Adresse (PA, 32Bit): [0:19] Physikalische Seitennummer, [20:31] Offset

Die MMU versucht, unter Rückgriff auf die TLBs Rückgriffe auf die Seitentabelle zu vermeiden. Dazu wird zunächst im ITLB (nur bei Anfragen von der Instruction Unit), dann parallel im UTLB und BTLB gesucht. Ein **Tablewalk** (Rückgriff auf Seitentabelle) wird nur durchgeführt, falls die Suche dort erfolglos war. Die reale Adresse geht dann an Cache und MU.

8.4.2 Cache

Der PPC601-Cache besteht aus 8 **Sets** mit je 64 **Cachelines** mit je 2 **Sektoren** mit je 8 Worten (=32KByte). Die Sektoren einer Cacheline entsprechen konsekutiven Wortfolgen im Speicher (Ein-/Auslesen erfolgt sektorweise). An jeder Cacheline wird dann ein Status (MESI, s.u.) und die Adresse vermerkt, die sie enthält. Ist eine Adresse dort nicht vorhanden (**Cache Miss**), so wird sie geladen (ist der Cache voll, wird entspr. LRU eine Cacheline zurückgeschrieben), ansonsten (**Cache Hit**) benutzt. Das Nachladen übernimmt die MU.

MESI: **(M)** Modified: Sektor modifiziert, noch nicht im Hauptspeicher **(E)** Exclusive: Nur in diesem Cache und konsistent mit Hauptspeicher **(S)** Shared: Konsistent, aber mehrere Caches haben diesen Sektor **(I)** Invalid: Daten ungültig bzw. Stelle frei

8.4.3 MU (Memory Unit)

Lädt Daten aus dem Hauptspeicher in den Cache. Dazu wird eine je 4stellige Write- bzw. Readqueue verwendet. Ein Platz der Writequeue ist für das Snooping reserviert.

Snooping: Bei Multiprozessorsystemen muß das Cachen kohärent ablaufen. Dazu überwacht die MU den Bus, auf dem die CPU mit den anderen CPUs Daten zum Hauptspeicher überträgt. Falls dort Daten übertragen werden, die nicht im Cache oder in den Queues liegen (**Snoop Miss**) ist alles in Ordnung. Falls nicht (**Snoop Hit**) wird wie folgt vorgegangen: Liest die CPU, wird der Vorgang unterbrochen, der Cache-Eintrag in den Hauptspeicher geschrieben, und der Vorgang fortgesetzt. Schreibt sie, wird der Eintrag für ungültig erklärt.

8.5 SPARC (Scalable Processor Architecture)

Besonderheit: Hat sehr viele Register (40..520 GPR). Für jeden Prozeduraufruf wird ein Fenster (insg. 2..32) von 16 Registern für die Verwaltung der Parameter und lokalen Variablen angelegt. Parameterübergabe wird mittels überlappender Fenster (gemeinsame Register) gelöst.

Vorteile: kein Sichern der Register bei Prozeduraufrufen nötig, kein Kopieren der Parameter

Problem: geht nur begrenzt oft, schwierig handhabbar bei rekursiven Prozeduren

Spezielle Register in der IU:

- **PC/nPC**: Program und next-Program Counter
- **PSR** (Processor State Register) mit NZVC-Flags (Neg,Zero,Overflow,Carry) und CWP (Current Window Pointer)
- **WIM** (Window Invalid Mask), 32Bit, Bit $i = 1 \Rightarrow$ Fenster i nicht belegt

Der SPARC rechnet in **Delayed Branch** - Technik: Da der Sprungadresse bekannt ist, bevor berechnet wurde, ob gesprungen werden soll, wird angenommen, daß der Sprung ausgeführt wird. War die Annahme falsch, wird ab dort neu berechnet.