Skript zum Kurs
“Rechnerarchitektur”, Teil 1

Peter Marwedel
Informatik 12
TU Dortmund

14. April 2012

Inhaltsverzeichnis

1 Einleitung **4**

1.1 Begleitmaterial 5

1.2 VHDL-Notation 5

1.2.1 Datentypen integer und natural 6

1.2.2 Datentyp boolean 6

1.2.3 Datentypen bit und bit_vector 6

2 Die Befehlssatz-Architektur (*instruction set architecture*) **8**

2.1 RISC und CISC 8

2.2 DSP-Befehlssätze 8

2.3 Netzwerk-Prozessoren und CRC-Zeichen 9

2.3.1 CRC-Zeichen 9

2.4 EPIC- und VLIW-Befehlssätze 13

2.5 SIMD-Befehle (Vektor- und Multimedia-Befehle) 16

2.6 Graphikprozessoren 17

2.7 ASIPs 18

2.8 Abstrakte Maschinen 18

2.8.1 Java Virtual Machine 18

2.9 Beurteilung von Befehlssätzen 19

2.10 Nicht-Von-Neumann-Maschinen 20

2.10.1 Reduktionsmaschinen 20

2.10.2 PROLOG-Maschinen 21

2.10.3 Datenflussmaschinen 21

2.10.4 Weitere Nicht-Von-Neumann-Maschinen 23

3 Mikroarchitektur **24**

3.1 Elementare Datentypen und deren Darstellung 24

3.1.1 Bitvektoren 24

3.1.2 Natürliche Zahlen 26
3.1.3 Ganze Zahlen 31
3.1.4 Gleitkomma-Zahlen 35
3.2 Dynamisches Scheduling 37
3.2.1 Einführung 37
3.2.2 *Scoreboarding* 38
3.2.3 Verfahren von Tomasulo 39
3.3 Sprung-Vorhersage 40
3.4 *Multiple Instruction Issue* 41

4 Speicherarchitektur **44**
4.1 Speicherhardware 44
4.2 Austauschverfahren 44

5 Der CO_2-Fußabdruck von PCs **47**

Literaturverzeichnis

Indexvereichnis

Kapitel 1

Einleitung

1. Vorles.

Die Vorlesung „Rechnerarchitektur” (RA) ist eine Stammvorlesung im Studiengang Informatik und eine Wahlpflichtvorlesung im Studiengang „Angewandte Informatik”. „Rechnerarchitektur” und die ältere Bezeichnung „Rechensysteme” sind gemäß der Prüfungsordnungen als äquivalent anzusehen. Die Vorlesung ist eine von relativ wenigen Vorlesungen aus dem Bereich der Technischen Informatik an der Universität Dortmund. Aufgrund der typischen Berufsfelder heutiger Informatiker [CEF+95] ist es in der Tat nicht (mehr) erforderlich, **jeden** Informatiker **umfangreich** in Technischer Informatik auszubilden. Fundierte **Grundkenntnisse** sind dennoch dringend erforderlich, damit Informatiker eine Vorstellung und ein **Grundverständnis** von den Geräten haben, mit denen sie umgehen. Diesem Ziel dient die Vorlesung „Rechnerarchitektur”.

Warum sollte sich nun **jeder** Informatiker und **jede** Informatikerin mit dem Stoff dieser Vorlesung beschäftigen? Nun, die Bedeutung des Grundverständnisses der Geräte wurde bereits erwähnt. Dieses Grundverständnis wird u.a. bei folgenden Tätigkeiten eines Informatikers bzw. einer Informatikerin benötigt:

- bei der Geräteauswahl,
- bei der Fehlersuche,
- bei der Leistungsoptimierung,
- bei Zuverlässigkeitsanalysen,
- beim Neuentwurf von Systemen,
- beim *accounting*,
- bei der Codeoptimierung im Compilerbau,
- bei Benchmarkentwürfen,
- bei Untersuchung des Einflusses der Paging-Hardware,
- bei Sicherheitsfragen.

Letzlich sollten sich Informatiker bzw. Informatikerinnen auch nicht durch grobe Wissenslücken in zentralen Bereichen der Datenverarbeitung blamieren.

In der Vorlesung werden wir zunächst auf die äußere Rechnerarchitektur (Definition siehe Kurs Rechnerstrukturen) eingehen. Als nächstes werden wir uns dem inneren Aufbau widmen. Speziell werden wir den Aufbau der Prozessoren (CPUs) und des Speichers behandeln. In der zweiten Hälfte des Kurses werden wir uns mit Mehrprozessorsystemen beschäftigen.

1.1 Begleitmaterial

Weltweit sehr weit verbreitet ist die Orientierung von RA-Vorlesungen an dem hervorragenden Buch „Computer Architecture – A Quantitative Approach" von Hennessy und Patterson [HP96]. Dieses Buch ist in der Lehrbuchsammlung enthalten. Seit 2002 existiert zu diesem Buch eine stark erweiterte dritte Auflage [HP02] und inzwischen auch eine vierte Auflage. Für das Buch gibt es zahlreiche ergänzende Quellen, die über die Web-Seite des Verlages zu erreichen sind (www.mkp.com). Teilweise wird sich die Vorlesung an das Buch anlehnen. Eine vollständige Orientierung soll allerdings unterbleiben, da die Bücher sehr stark auf die Fähigkeit zielen, selbst Prozessoren zu entwickeln. Diese Fähigkeit wird im Silicon Valley, im dem die Autoren arbeiten, auch zur Weiterentwicklung beispielsweise der Produkte der Firmen Sun, MIPS und Intel benötigt. Zu diesem Zweck wird in den Büchern ein großes Schwergewicht auf den Entwurf einer sehr starken Fließbandverarbeitung in modernen Prozessoren gelegt. Erst seit der dritten Auflage werden die Prozessoren eingebetteter Systeme, die für europäische Informatiker besonders wichtig sind, mit berücksichtigt. Die zahlreichen Diagramme zu Leistungsaussagen vermitteln allerdings wenig über die Prinzipien von Rechensystemen, sind relativ schnell überholt und auch kaum als Grundlage von Klausuren oder Prüfungen geeignet. Auch ist in Dortmund ein großer Teil des Buches den Studierenden bereits aus dem Kurs "Rechnerstrukturen" bekannt. Dieser Teil soll hier natürlich nicht wiederholt werden.

Ein didaktisch gutes deutsches Buch ist das Buch „Mikrorechnersysteme" von H. Bähring [Bae94]. Von diesem Buch besitzt die Lehrbuchsammlung zahlreiche Exemplare.

Andere Bücher enthalten v.a. eine ausführlichere Behandlung von Parallelrechnern. Genannt werden sollen hier die Bücher von Culler und Singh [CS99] sowie von Silc, Robic und Ungerer [SRU99].

In dieser Vorlesung werden wir den Büchern jeweils geeigneten Stoff entnehmen. Zusätzlich werden wir Stoff zu speziellen Themen aus anderen Büchern verwenden. Diese Bücher werden jeweils angegeben werden. Schließlich werden wir im Interesse der Aktualität gelegentlich auf Zeitschriftenartikel zurückgreifen.

1.2 VHDL-Notation

Die Beschreibung der Arbeitsweise von Rechnern erfordert die Verwendung einer klar definierten Sprache. Zu diesem Zweck sind verschiedene Hardware-Beschreibungssprachen (engl. *hardware description languages*, HDLs) entwickelt worden. Die Motivation für die Benutzung von HDLs liegt in

- der präzisen Spezifikation,
- der Möglichkeit der Simulation,
- der Kommunikation zwischen Entwicklern,
- der automatisierten Erzeugung und der Überprüfung des Entwicklungsergebnisses,
- der Dokumentation des Arbeitsergebnisses,
- der Beschleunigung des Entwurfsprozesses.

Sehr verbreitet ist die Sprache VHDL (VHSIC Hardware Description Language) [IEE88, IEE92]. VHSIC wiederum steht für *very high speed integrated circuit*. Das VHSIC-Progamm geht auf eine Initiative des amerikanischen Verteidigungsministeriums (DoD) zurück.

VHDL ist für unsere Zwecke geeignet. Ein wesentlicher Grund für die Verwendung von VHDL in diesem Text ist, dass VHDL eine streng getypte Hardwarebeschreibungs-Sprache ist. Dadurch führt VHDL zu dem nützlichen Zwang, zwischen Folgen von Bits und deren Interpretion als Wert in einem anderen Bereich (wie z.B. dem der natürlichen Zahlen) immer streng zu unterscheiden.

VHDL ist allerdings eine sehr komplexe, auf der Sprache ADA basierende Sprache. Wir werden daher nur eine kleine Teilmenge der Sprache vorstellen. Speziell werden wir die zentralen Datentypen verwenden.

1.2.1 Datentypen integer und natural

Als Datentyp ist in VHDL der Datentyp `integer` vordefiniert. Die oberen Grenzen des mit diesem Typ darstellbaren Zahlenbereichs sind Implementations-abhängig. Leider unterstützen viele VHDL-Implementierungen nur die mit 32 Bit darstellbaren Zahlen. Mit Hilfe des Ausdrucks `integer'high` kann man auf die obere Grenze des jeweils darstellbaren Integer-Zahlenbereichs Bezug nehmen.

Auf dem Datentyp `integer` sind in VHDL die üblichen arithmetischen Operationen `+, <` usw. definiert.

Aus dem Datentyp `integer` können in VHDL mit Hilfe von `subtype`-Definitionen weitere Sub-Typen abgeleitet werden. Beispiel:

```
subtype natural  is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Variablen von derartig abgeleiteten Typen sind zuweisungskompatibel.

1.2.2 Datentyp boolean

Als weiterer Datentyp ist der Typ `boolean` vordefiniert:

```
type boolean is (False, True);
```

`boolean` wird also durch Aufzählung der möglichen Werte definiert.

1.2.3 Datentypen bit und bit_vector

Weiterhin ist der Datentyp `bit` vordefiniert. Literale des Typs `bit` werden durch einfache Anführungszeichen bezeichnet. Beispiel:

```
'0'
```

Der Datentyp `bit` ist in VHDL als Aufzählungstyp definiert durch

```
type bit is ('0','1');
```

Als weiterer Datentyp ist der Typ `bit_vector` vordefiniert als

```
type bit_vector is array (natural range <>) of bit;
```

Dabei kennzeichnen die Klammern <> den ausgelassenen Index eines sogenannten *unconstrained array*, eines Arrays, dessen Indexgrenzen durch einen Teilbereich der natürlichen Zahlen später festgelegt werden können. Dies geschieht beispielsweise in einer Definition der Art

```
variable instruction : bit_vector (31 downto 0);
```

Das Schlüsselwort `downto` zeigt dabei einen **absteigenden** Indexbereich an. VHDL erlaubt zusätzlich auch **aufsteigende** Indexbereiche. **Zur Vermeidung von Konfusion werden wir stets absteigende Indexbereiche mit der unteren Indexgrenze 0 benutzen**. Solche Indexbereiche werden auch in den meisten auf VHDL aufbauenden Standards, wie z.B. dem *VHDL math package* eingesetzt.

Literale des Typs `bit_vector` werden in doppelten Anführungszeichen eingeschlossen. Beispiel:

```
"01010101"
```

In VHDL kann auch explizit angegeben werden, dass es sich bei einem Literal um ein Binärliteral handelt. Dies geschieht durch Voranstellen von `B`:

```
B"01010101"
```

Auf den Daten `bit` und `bit_vector` sind in VHDL die üblichen logischen Operationen sowie die Konkatenation vordefiniert. Letztere wird durch ein `&`-Zeichen dargestellt. Beispiele:

```
"01010101" & '0'
"01010101" & "01010101"
'1' & '0'
```

In Kapitel 2 werden wir beschreiben, wie Bitvektoren mittels der Funktionen `nat` und `int` als natürliche bzw. ganze Zahlen interpretiert werden können.

Kapitel 2

Die Befehlssatz-Architektur (*instruction set architecture*)

In den folgenden beiden Kapiteln werden wir Prozessoren behandeln, und zwar zunächst die externe Sicht. Diese umfasst eine **logische** Sicht auf das Gesamtsystem eines Prozessors.

Die logische Sicht von Prozessoren enthält

- die unterstützten elementaren Datentypen (wie z.B. `Integer` und `Real`-Typen),
- die durch die Maschinenbefehle ansprechbare logische Speicherorganisation,
- den Maschinenbefehlssatz des Prozessors sowie
- das logische Verhalten der Interruptorganisation.

Im folgenden werden wir uns mit verschiedenen Klassen von Befehlssätzen beschäftigen.

2.1 RISC und CISC

Bezüglich der RISC- und CISC-Befehlssätze verweisen wir auf den Rechnerstruktur-Kurs.

2.2 DSP-Befehlssätze

2. Vorles.

Prozessoren werden heute nicht nur in PCs und Workstations, sondern vielfach auch in so genannten **eingebetteten Systemen** eingesetzt. Dies sind Systeme, in denen die Informationsverarbeitung in die (meist physikalische) Umgebung vollständig integriert ist und in denen (physikalische) Größen direkt von den Ergebnissen der Informationsverarbeitung aus beeinflußt werden. Diese Systeme kommen meist ohne Bildschirm und Tastatur aus. Das Benutzer-Interface läßt in der Regel nicht erkennen, dass Eingaben an Rechensysteme gemacht werden. Beispiele hierfür sind informationsverarbeitende Systeme in der Fahrzeug-Elektronik, in der Telekommunikation und im Audio/Video-Bereich. Charakteristisch für fast alle dieser Bereiche ist weiterhin die hohe Bedeutung der **Effizienz** der technischen Lösung, bei portablen Anwendungen v.a. hinsichtlich des Stromverbrauchs.

Eine wichtige Teilaufgabe von eingebetteten Systemen besteht in der digitalen Signalverarbeitung (engl. it digital signal processing (DSP)). Für diese Aufgaben sind spezielle, DSP-Aufgaben effizient verarbeitende Prozessoren entwickelt worden. DSP-Prozessoren besitzen die folgenden spezifischen Eigenschaften:

- *saturating arithmetic*

- **spezielle Adressierungsarten:**
  Aufgrund der Anwendung in üblichen DSP-Algorithmen sehen DSP-Prozessoren häufig spezielle Adressierungsarten vor. Die Modulo-Adressierung beispielsweise erlaubt die effiziente Realisierung von verzögerten Signalen mit Hilfe von Ringpuffern.
- **Eingeschränkte Parallelität:**
  Die Rechenwerke der meisten DSP-Prozessoren erlauben es, in einem Takt Zuweisungen zu mehreren Registern gleichzeitig auszuführen. Diese Prozessoren stellen diese Form der begrenzten Parallelität dann meist auch an der Befehlsschnittstelle zur Verfügung. Gängig sind z.B. *parallel moves* genannte Befehle, die gleichzeitig eine Arithmetik-Operation und einen Datentransport veranlassen.
- **Heterogene Registersätze**
- **multiply/accumulate-Befehle**

  Eine wichtige Aufgabe der digitalen Signalverarbeitung ist die digitale Filterung. Bei der Filterung werden die Elemente $y(j)$ einer Ausgangsfolge y mit der Formel

  $$y(j) = \sum_{k=0}^{m} a(k) * x(j-k) \qquad (2.1)$$

  aus den Elementen einer Eingangsfolge x berechnet. Der Vektor a stellt dabei einen Gewichtsvektor dar, der die Charakteristik des Filters bestimmt. m bestimmt die Ordnung des Filters. Die Summe kann iterativ aus Partialsummen $y_K(j)$ der Summation bis zu einem gewissen K berechnet werden:

  $$y_{-1}(j) = 0 \qquad (2.2)$$
  $$y_k(j) = y_{k-1}(j) + a(k) * x(j-k) \qquad (2.3)$$
  $$y(j) = y_m(j) \qquad (2.4)$$

  Multiply/accumulate-Befehle erlauben es, mit einem einzigen Befehl einen Iterationsschritt der Gleichung 2.3 auszuführen. Dazu wird ein Akkumulatorregister `ACC` mit 0 initialisiert, zwei Datenregister `X` und `Y` mit $a(0)$ bzw. $x(j)$ geladen und und zwei Adressregister `A1` und `A2` zeigen auf das nächste relevante Element der Arrays a bzw. x. Anschließend wird mittels eines einzigen multiply/accumulate-Befehles in einer Schleife folgende Zuweisungen ausgeführt:

  ```
  ACC := ACC + X * Y, X := Speicher[A1], Y := Speicher[A2], A1++, A2--
  ```

  Mit jeder Befehlsausführung wird die nächste Partialsumme berechnet.
- **Realzeit-Fähigkeit:**
  Es ist wichtig, die maximale Laufzeit eines Programms möglichst exakt und nicht nur im Mittel angeben zu können. Deshalb wird z.B. vielfach auf Caches, die eine datenabhängige Laufzeit bewirken würden, verzichtet.

DSP-Prozessoren bieten v.a. für DSP-Applikationen mehr Rechenleistung pro Watt Leistungsverbrauch, weshalb sie v.a. in portablen Geräten vielfältig Einsatz finden.

2.3 Netzwerk-Prozessoren und CRC-Zeichen

3. Vorles.

Netzwerk-Prozessoren: siehe Folien

2.3.1 CRC-Zeichen

2.3.1.1 Prinzip

Zur Absicherung von zu übertragenden oder abzuspeichernden Informationen werden meist bestimmte Codes eingesetzt, bei denen man eine bestimmte Menge von Fehlern erkennen kann [PB61, HQ89]. Wir werden diese hier zunächst allgemein, unabhängig von Plattenspeichersystemen, betrachten.

Zu übertragen sei eine Nachricht, die in k Bits kodiert sei. Durch Übertragung mit einem Code von mehr als k Bits ist ein Schutz gegen fehlerhafte Übertragung möglich. Wir wollen annehmen, dass insgesamt n Bits übertragen werden. Falls der gewählte Code in den k Bits mit der Ausgangsnachricht übereinstimmt, sprechen wir von einem **systematischen Code**:

Nachricht	Prüfbits
$\leftarrow - - - - - \rightarrow$	$\leftarrow - - - - - \rightarrow$
k Bits	$n-k$ Bits

Für die weitere Theorie betrachten wir die einzelnen Bits a_i der Nachricht und der Prüfbits als Koeffizienten eines Polynoms G bzw. R. Wir betrachten also das **Nachrichtenpolynom**:

$$G = \sum_{i=0}^{k-1} a_i * 2^i \tag{2.5}$$

Statt der speziellen Basis 2 betrachten wir im folgenden das Polynom zur allgemeinen Basis x:

$$G(x) = \sum_{i=0}^{k-1} a_i * x^i \tag{2.6}$$

Den übertragenen Code können wir, falls es ein systematischer Code ist, durch ein **Codepolynom** der Form

$$F(x) = G(x) * x^{n-k} + R(x) \tag{2.7}$$

darstellen. Darin stellt $R(x)$ die Prüfbits dar.

Wir fordern nunmehr, dass $F(x)$ für alle benutzten Codierungen der Nachricht durch ein **Generator-Polynom** $P(x)$ teilbar sein soll, d.h. es soll gelten:

$$F(x) = P(x) * Q(x) = G(x) * x^{n-k} + R(x) \tag{2.8}$$

Wir nennen derartige Codes **zyklische Codes**.

Wir wollen im folgenden weiter voraussetzen, daß wir alle Operationen modulo 2 durchführen. Als Folge davon brauchen wir zwischen + und − nicht zu unterscheiden. Aus der letzten Gleichung folgt also:

$$R(x) = G(x) * x^{n-k} - P(x) * Q(x) \tag{2.9}$$

Damit sind die Prüfbits $R(x)$ gleich dem Rest der Division von $G(x)$ multipliziert mit x^{n-k} durch $P(x)$[1].

Wir wollen den Rechengang an einem praktischen Beispiel deutlich machen. Dabei gehen wir davon aus, dass unsere Nachricht ursprünglich durch ein Polynom vom Grad 4 darstellt wird und dass dieses aufgrund eines Generatorpolynoms vom Grad $n-k=2$ mit x^2 multipliziert wird.

[1] Der Rest r der Division von g durch p ist allgemein definiert durch $r = g - p * q$ mit $r < p$

$$
\begin{array}{l}
(1x^6 + 0x^5 + 1x^4 + 1x^3 + 1x^2 + 0x^1 + 0x^0) : (1x^2 + 1) = 1x^4 + 0x^3 + 0x^2 + 1x^1 + 1x^0 \\
\;1x^6 + 0x^5 + 1x^4 \\
\;- - - - - - - - - - - \\
\;0x^6 + 0x^5 + 0x^4 + 1x^3 \\
\qquad\quad 0x^5 + 0x^4 + 0x^3 \\
\qquad\quad - - - - - - - - - - - \\
\qquad\quad 0x^5 + 0x^4 + 1x^3 + 1x^2 \\
\qquad\qquad\qquad 0x^4 + 0x^3 + 0x^2 \\
\qquad\qquad\qquad - - - - - - - - - - - - \\
\qquad\qquad\qquad 0x^4 + 1x^3 + 1x^2 + 0x^1 \\
\qquad\qquad\qquad\qquad\quad 1x^3 + 0x^2 + 1x^1 \\
\qquad\qquad\qquad\qquad\quad - - - - - - - - - - - - \\
\qquad\qquad\qquad\qquad\quad 0x^3 + 1x^2 + 1x^1 + 0x^0 \\
\qquad\qquad\qquad\qquad\qquad\qquad 1x^2 + 0x^1 + 1x^0 \\
\qquad\qquad\qquad\qquad\qquad\qquad - - - - - - - - - - \\
\qquad\qquad\qquad\qquad\qquad\qquad 0x^2 + 1x^1 + 1x^0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad 1x^1 + 1x^0 \; (Rest)
\end{array}
$$

Auf die Koeffizienten reduziert, sieht der Rechengang wie folgt aus:

```
(1 + 0 + 1 + 1 + 1 + 0 + 0) : (1 + 0 + 1) = 1 + 0 + 0 + 1 + 1
 1 + 0 + 1
 ---------
 0 + 0 + 0 + 1
     0 + 0 + 0
     ---------
     0 + 0 + 1 + 1
         0 + 0 + 0
         ---------
         0 + 1 + 1 + 0
             1 + 0 + 1
             ---------
             0 + 1 + 1 + 0
                 1 + 0 + 1
                 ---------
                 0 + 1 + 1
                     1 + 1  (Rest)
```

Diese Operationen können wir durch die Schaltung nach Abb. 2.1 realisieren.

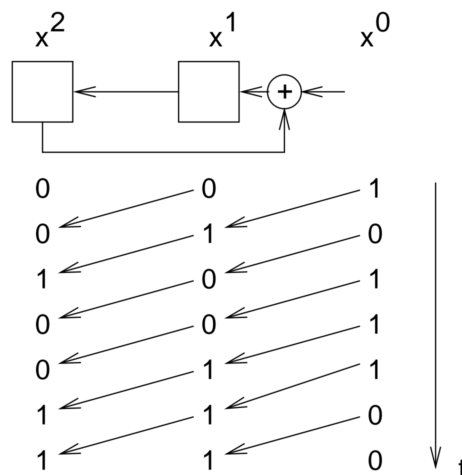

Abbildung 2.1: Division durch ein Polynom (x^2+1) mit Registern und XOR-Gatter

Die Positionen der Modulo-2-Addierer (XOR-Gatter) sind dabei durch das Polynom bestimmt, durch das wir teilen. Für jeden Term des Generatorpolymoms, mit Ausnahme des Terms mit der höchsten Potenz von x, benötigen wir ein XOR-Gatter. Abb. 2.2 zeigt die Schaltung bei einem etwas komplexeren Polynom.

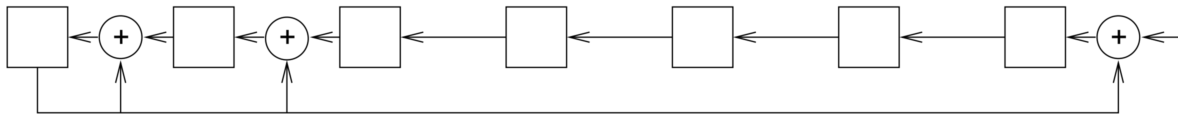

Abbildung 2.2: Schaltung für das Polynom $x^8 + x^7 + x^6 + 1$

2.3.1.2 Eigenschaften von zyklischen Codes

Vom Empfänger wird im allgemeinen nicht die Nachricht $F(x)$, sondern irgendeine, möglicherweise davon verschiedene Nachricht $H(x)$ empfangen werden. Sei $E(x)$ der durch Störungen verursachte Unterschied zwischen beiden, sei also

$$H(x) = F(x) + E(x).$$

$E(x)$ heißt **Fehlerpolynom** Jede 1 dieses Polynoms kennzeichnet eine Fehlerstelle. Für die Erkennung von Fehlern gilt das folgende:

- Ist $H(x)$ nicht durch $P(x)$ teilbar, so wird ein Fehler erkannt
- Ist $H(x)$ durch $P(x)$ teilbar, so ist die Übertragung fehlerfrei oder es liegt ein nicht erkennbarer Fehler vor.

Satz 1: Ein zyklischer Code, der durch ein Polynom mit mehr als einem Term erzeugt wird, entdeckt alle Einzelfehler.

Bew.: Einzelfehler besitzen ein Fehlerpolynom der Form $E(x) = x^i$. Hat $P(x)$ mehr als einen Term, so teilt es x^i nicht.

Satz 2: Jedes durch $1 + x$ teilbare Polynom hat eine gerade Zahl von Termen.

Bew.-Idee: Vielfache von $(1 + x)$ enthalten Paare.

$\Rightarrow$: Mit $(1 + x)$ als Faktor findet man eine ungerade Anzahl von Fehlern (Parity-Prüfung).

Def.: Ein Burstfehler der Länge b ist ein Fehler, bei dem die falschen Symbole den Abstand b haben.

Beispiel: $E(x) = x^7 + x^4 + x^3 = 0010011000$. Per Definition zählt man dies als Abstand 5, d.h. man zählt die Randsymbole mit.

Satz 3: Ein durch ein Polynom vom Grad $n - k$ erzeugter zyklischer Code entdeckt alle Burstfehler der Länge $b \leq n - k$, wenn $P(x)$ x nicht als Faktor enthält.

Bew.: Sei x^i der Term mit dem kleinsten Exponenten, der in $E(x)$ vorkommt. Sei zunächst einmal $i > 0$. Wir können dann $E(x)$ darstellen als $E(x) = x^i * E_1(x)$. Das Produkt ist nicht durch $P(x)$ teilbar, wenn die beiden Terme nicht durch $P(x)$ teilbar sind.

1. x^i ist nicht durch $P(x)$ teilbar, wenn $P(x)$ nicht x als Faktor enthält.
2. $E_1(x)$ ist höchstens vom Grad $b - 1$, d.h. vom Grad $n - k - 1$. Daraus folgt: $P(x)$ teilt $E_1(x)$ nicht.

Sei nun $i = 0$. Dnn kann kein x^i mit $i > 0$ ausgeklammert werden. Die o.a. Argumente bezüglich $E_1(x)$ gelten dann aber direkt bezüglich $E(x)$.

Satz 4: Die Anzahl der nicht erkannten Burstfehler der Länge $b > n - k$ ist, bezogen auf die Anzahl möglicher Burstfehler:

$$2^{-(n-k-1)} \quad , \quad \text{wenn } b = n - k + 1 \tag{2.10}$$

$$2^{-(n-k)} \quad , \quad \text{wenn } b > n - k + 1 \tag{2.11}$$

Beispiel: $P(x) = (1 + x^2 + x^{11})(1 + x^{11})$ erkennt:

- 1 Burst der Länge ≤ 22
- 2 Bursts der $\sum$-Länge ≤ 12 wenn $k \leq 22495$
- Jede ungerade Anzahl von Fehlern
- 99,99996 % der Bursts der Länge 23
- 99,99998 % der Bursts der Länge > 23

Burst-Fehler sind gerade deshalb so interessant, weil sie bei einer kurzzeitigen Störung der Übertragung oder bei einem Staubkorn auf dem Band oder der Platte auftreten.

2.3.1.3 Standard-Generatorpolynome

Die Prüfbits $R(x)$ werden bei der Übertragung oder Speicherung von Nachrichten als sog. **CRC-Zeichen** (von engl. *cyclic redundancy check*) zugesetzt. Generatorpolynome sind laut DIN:

Magnetband 800bpi	$1 + x^3 + x^5 + x^6 + x^9$
Magnetbandkassette	$1 + x^2 + x^{15} + x^{16}$
Floppy disc	$1 + x^2 + x^{15} + x^{16}$
GCR ECC-Zeichen	$1 + x^2 + x^{15} + x^{16}$
HDLC	$1 + x^5 + x^{12} + x^{16}$

Dabei wird das CRC-Zeichen aus dem bitseriellen Datenstrom erzeugt. Bei byteparalleler Übertragung müssen die Daten also zusätzlich in einen bitseriellen Strom gewandelt werden.

Im Falle eines erkannten Lesefehlers wird in der Regel zunächst eine Wiederholung des Lesens durchgeführt. Erst nachdem eine gewisse Anzahl von Wiederholungen erfolglos blieb, wird auf einen permanten Fehler geschlossen.

2.4 EPIC- und VLIW-Befehlssätze

4. Vorles.

Die begrenzte Form der Parallelität von DSP-Prozessoren wird bei *very large instruction word* (VLIW) Architekturen weiter ausgebaut. Diese Architekturen besitzen eine größere Anzahl an funktionellen Einheiten sowie meist auch Speicher mit mehreren Ports. Das Befehlswort ist bei diesen Architekturen breit genug, um alle Einheiten gleichzeitig anzusteuern. VLIW-Architekturen beziehen ihre Leistungsfähigkeit aus der expliziten Parallelität in den langen VLIW-Befehlen, können sich daher den Aufwand zur Erkennung möglicher Parallelität in Hardware sparen.

Beispiel:

Abb. 2.3 zeigt skizzenhaft ein Beispiel einer einfachen, hypothetischen Architektur (Multiplexer und Bus-Treiber sind hierbei nicht gezeigt). In den ALUs können hier 3 Operationen gleichzeitig ausgeführt werden.

Prozessoren mit expliziter Parallelität werden neuerdings auch als *explicit parallelism instruction computers* (EPIC) bezeichnet. Sie beinhalten als spezielle Klasse die VLIW-Architekturen, können aber die explizite Parallelität evtl. auch anders als gemäß Abb. 2.3 ausnutzen. Kernidee der EPIC-Architekturen ist, die Erkennung der Parallelität in den Compiler zu verlagern und so mehr Parallelität bei gleichzeitiger Einsparung an Hardware zu gewinnen.

Weitere Beispiele:

1. Der TMS320C62xx Signalprozessor ist die erste „richtige" kommerzielle VLIW-Maschine, angekündigt 1997.

   Eigenschaften:

   - Zwei Rechenwerke mit je

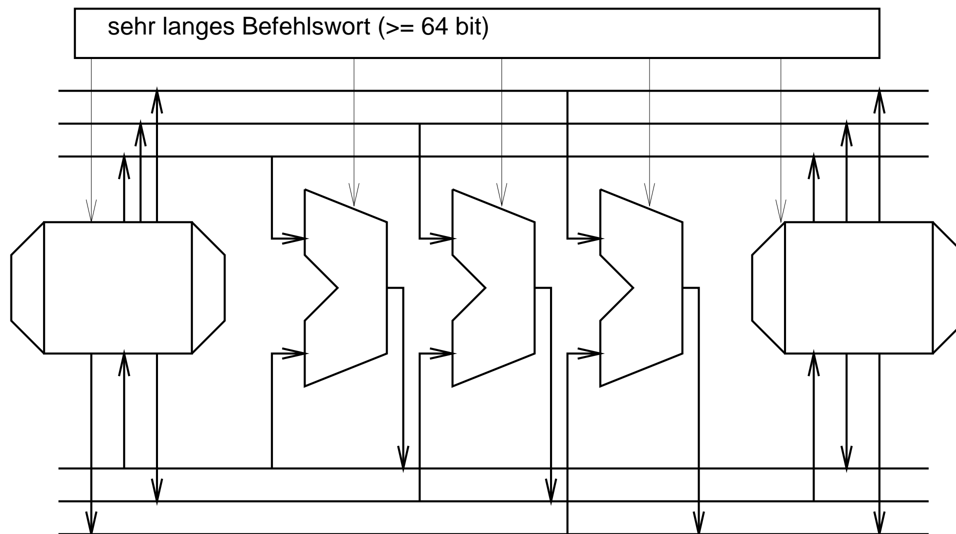

Abbildung 2.3: VLIW-Architektur (Beispiel)

  - 16 x 16 Bit Multiplizierer
  - 1 Addierer
  - 1 Addierer/Shifter
  - 1 Addierer/Shifter/Normalisierer
  - Registerspeicher mit 16 32-Bit-Registern
  - 1 Speicherlese- + 1 Speicherschreib-Pfad
  - 1 Pfad zum anderen Rechenwerk

- Vorgesehen, bei 200 MHz bis zu 8 32-Bit-Befehle pro Takt (5 ns) zu starten (max. 1.6 „GIPS")
- Jeder 32-Bit-Befehl spricht eine funktionelle Einheit an; der Compiler muss die Zuordnung zur Compile-Zeit vornehmen.

  Jeder 32-Bit-Befehl besitzt 1 Bit, welches entscheidet, ob der nächste Befehl noch im gleichen Takt ausgeführt wird. Dadurch Holen von max. 256 Befehlsbits (*instruction packet* genannt) / Zyklus.

1	Befehl-A	1	Befehl-B	1	Befehl-C	1	Befehl-D	0	Befehl-E	0	Befehl-F

Ausführung:

Schritt 1	Befehle A, B, C, D, E
Schritt 2	Befehl F

Abbildung 2.4: Befehlspakete des TMS320C62xx

  De facto variable Befehlslänge von 32 bis 256 Bit. Falls in einem Zyklus nicht alle funktionellen Einheiten benutzt werden können, gehen dadurch keine Befehlsbits verloren.

  Man kann „mitten in ein Befehlspaket" hinein springen.

- Übliche DSP-Funktionalität: Sättigungsarithmetik (*saturating arithmetic*), Modulo-Adressierung (*modulo addressing*), Normalisierungsoperation, integrierte E/A-Leitungen, kein virtueller Speicher.

2. Der NXP TriMedia-Prozessor für Multimedia-Applikationen mit bis zu 5 Befehlen/Zyklus.

3. Der Intel i860 ist ein Prozessor mit einer Befehlswortbreite von 64 Bit. Davon spricht die eine Hälfte Gleitkomma-Einheiten und die andere Integer-Einheiten an. Dieser Prozessor enthält damit auch schon VLIW-Konzepte.

4. Die in der PG PRIPS entwickelte PROLOG-Maschine [ABM+93].
5. Die als Nachfolge der Pentium-Architektur u.a. von der Fa. Intel entwickelte IA-64 Architektur. Sein Befehlsformat gehört zur EPIC-Klasse (siehe Abb. 2.5).

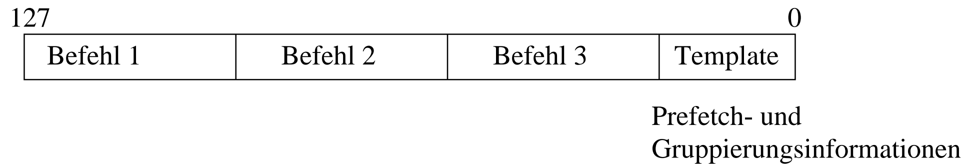

Abbildung 2.5: Befehlsformat des IA-64

Potentielle Probleme der VLIW-Architekturen liegen in folgenden Bereichen:

- Skalierbarkeitsprobleme

  Die explizite Parallelität in VLIW-Befehlen hat zur Folge, dass Architekturen mit mehr funktionellen Einheiten auch ein anderes Befehlsformat benötigen. Das heißt, selbst innerhalb einer Rechnerfamilie gibt es keine Binärcode-Kompatibilität.

  Einen Ausweg bietet hier die Benutzung anderer Formate zur Abspeicherung ausführbarer Programme: ausführbare Programme werden in Form von auszuführenden Operationen (abstrakte Opcodes) gespeichert, und die Erzeugung der Binärcodes erfolgt erst während des Ladevorgangs.

- Timing-Probleme

  VLIW-Architekturen sind aufgrund ihrer Parallelität auf Speicher angewiesen, die eine größere Anzahl von gleichzeitigen Zugriffen erlauben. Bei allen bekannten Realisierungsformen solcher Speicher hängt die Zugriffszeit davon ab, ob die Adressen dieselben Bereiche innerhalb der Speicher ansprechen. Da die Adressen nun vor der Ausführungszeit nicht bekannt sind, ergeben sich Probleme bei der Wahl der Zykluszeit. Bei einer festen Zykluszeit müßte man den schlechtesten Fall (alle Adressen beziehen sich auf denselben Speicherblock) annehmen und würde damit kaum über die Geschwindigkeit klassicher Befehle hinauskommen. Es müssen also Verfahren untersucht werden, mit denen die Zykluszeit von den tatsächlich vorkommenden Speicherzugriffskonflikten abhängig gemacht werden kann. Diese Techniken werden aber eigentlich auch bei modernen RISC-Maschinen benötigt.

- Kompatibiltätsprobleme

  Es ist nicht einfach, mit VLIW-Architekturen den Befehlssatz des 8086 zu verarbeiten. In PCs konnten sie daher bislang nicht eingesetzt werden. Für eingebettete Systeme, die nicht zum 8086 codekompatibel sein müssen, kommen sie aber durchaus in Frage.

  Der IA-64 soll allerdings x86-Code ausführen können.

Gelegentlich wird gegen diese Architekturen eingewandt, die Wortbreite wäre unsinnig groß. Dies trifft aber v.a. auf die ältere Implementierung von Fisher zu. Die oben angegebenen Maschinen sind durchaus effizient.

Bei gegenwärtigen RISC-Architekturen ist man bei derartig vielen Fließband-Stufen angelangt, dass die Hardware-Logik zur Behandlung von Datenabhängigkeiten bereits einen großen Teil der Logik überhaupt ausmacht. Gleichzeitig ist dieser Teil der Hardware sehr schwer korrekt zu entwerfen. Auch nach Beseitigung des bekannten Pentium-Gleitkomma-Fehlers enthalten gegenwärtige Pentium-Chips eine Vielzahl von Fehlern, die zum gutem Teil durch die Komplexität der Fließbandverarbeitung erklärbar sind. Selbst Hennessy und Patterson mussten zugeben, dass die erste Auflage ihres Buches [HP96] auch nach Erprobung an vielen amerikanischen Universitäten noch einen Fehler in der Fließbandkonzeption enthielt. Es sei in diesem Zusammenhang Bob Rau (hp Labs) zitiert: *the only ones in favor of superscalar (RISC) machines are those who haven't built one yet.*

Mit der Ankündigung des VLIW-Prozessors durch die Fa. Texas Instruments (1997) und des IA-64 wurde der Trend zu solchen Prozessoren gestartet.

2.5 SIMD-Befehle (Vektor- und Multimedia-Befehle)

5. Vorlesung

Der erste Teil der Vorlesung behandelt Vektorbefehle. Die betreffenden Folien entsprechen dem Abschnitt über Vektorprozessoren im Kapitel 4 in der fünften Auflage des Buches von Hennessy/Patterson.

Im zweiten Teil behandeln wir Multimedia-Befehle, auch *short vector extensions* genannt. Vektor- und Multimedia-Befehle führen aufgrund eines Befehls Operationen auf vielen Daten aus, gehören damit zur Klasse der SIMD-Befehle (*single instruction, multiple data*).

Multimedia-Befehle lassen sich durch die hohen Leistungsanforderungen im Multimediabereich motivieren. Durch den Einsatz von DSP-Algorithmen zur Audio- und Videosignalverarbeitung, entstand der Bedarf zur Leistungssteigerung bei diesen Applikationen. Aus diesem Grund wurden viele RISC- und CISC-Prozessoren um DSP-spezifische Eigenschaften erweitert[PWW97]. So werden beispielsweise meist auch multiply/accumulate-Befehle angeboten. Dazu kommt die Ausnutzung der ohnehin vorhandenen parallelen Rechenwerke für die gleichzeitige Verarbeitung verschiedener Farbwerte oder verschiedener Bildpunkte.

Beispiele:

1. Die Hewlett-Packard *precision architecture* (hp PA) unterstützt den so genannten Halbwort-Additionsbefehl `HADD`, der die Inhalte von 32-Bit-Registern als zwei in 16 Bit kodierte Zahlen interpretiert. Mit einem `HADD`-Befehl werden so zwei in 16 Bit kodierte Zahlen addiert (siehe Abb. 2.6).

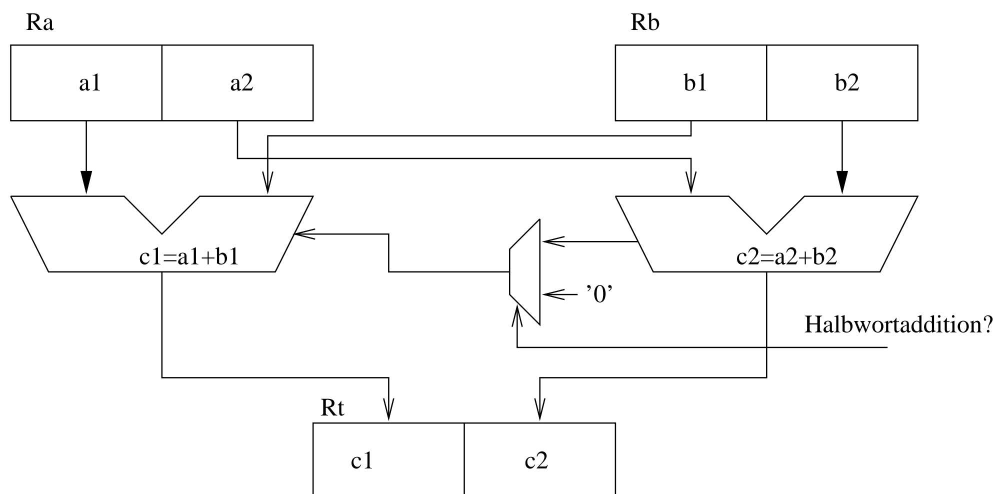

Abbildung 2.6: 16-Bit-Addition von Werten in 32-Bit-Registern Ra und Rb

   Im 32-Bit-Addierer muss man dafür nur die internen Überträge an den 16-Bit-Grenzen unterbrechen.

   Weiterhin können beide Additionen optional mit Sättigungsarithmetik erfolgen. Dies entspricht bei jeder Addition jeweils einem zusätzlichen Test auf Überlauf oder Unterlauf. Beachtet man, dass man in manchen RISC-Befehlssätzen die 16-Bit-Werte durch Schiebeoperationen aus 32-Bit-Worten extrahieren und ggf. zu solchen Worten wieder kombinieren muss, so ist verständlich, dass bis zu 10 Befehle durch einen `HADD`-Befehl ersetzt werden können.

2. Die Pentium MMX-Architektur benutzt für die ohnehin vorhandenen 64-Bit Datenwege einen ähnlichen Ansatz. 64-Bit-Vektoren können als 8 bytecodierte, 4 wortcodierte oder 2 doppelwortcodierte Zahlen betrachtet werden. Für die Arithmetik sind *wrap around*- *saturating*-Optionen wählbar (siehe Tabelle 2.1) [PWW97]. Es stehen separate Multimediaregister `mm0` bis `mm7` zur Verfügung, die aber stets konsistent mit den Gleitkommaregistern gehalten werden, damit sie beim Umschalten auf andere Prozesse nicht separat gerettet werden müssen, also den Kontext des laufenden Prozesses nicht vergrößern (siehe Ende diese Kapitels).

   Abb. 2.7 zeigt, wie die MMX-Befehle benutzt werden können, um zwischen zwei Bildern skaliert zu interpolieren. Die größte Beschleunigung ergibt sich, wenn die Bildinformation einer Farbe in benachbarten Bytes des Speichers abgelegt ist. Dann können, wie in dem Beispiel, die Werte einer Farbe für vier benachbarte Bildpunkte in einem Schritt bearbeitet werden und es ergibt sich im Idealfall eine Beschleunigung um den Faktor 4.

Befehl	Optionen	Beschreibung
Padd[b/w/d] PSub[b/w/d]	*wrap around,* *saturating*	Addition/Subtraktion von Bytes, Worten, Doppelworten
Pcmpeq[b/w/d] Pcmpgt[b/w/d]		Ergebnis = "111..11" falls wahr, "000..00" sonst Ergebnis = "111..11" falls wahr, "000..00" sonst
Pmullw Pmulhw		Multiplikation von 4*16-Bit, unteres Wort Multiplikation von 4*16-Bit, oberes Wort
Psra[w/d] Psll[w/d/q] Psrl[w/d/q]	Stellenzahl in Register oder in Befehl	Paralleles Schieben von Worten, Doppelworten und 64 Bit-Quad-Worten
Punpckl[bw/wd/dq] Punpckh[bw/wd/dq]		Paralleles Entpacken Paralleles Entpacken
Packss[wb/dw]	*saturating*	Paralleles Packen
Pand, Pandn Por, Pxor		Logische Operationen auf 64-Bit-Werten
Mov[d/q]		Datentransport Speicher/Register

Tabelle 2.1: Befehle der x86-MMX-Erweiterung

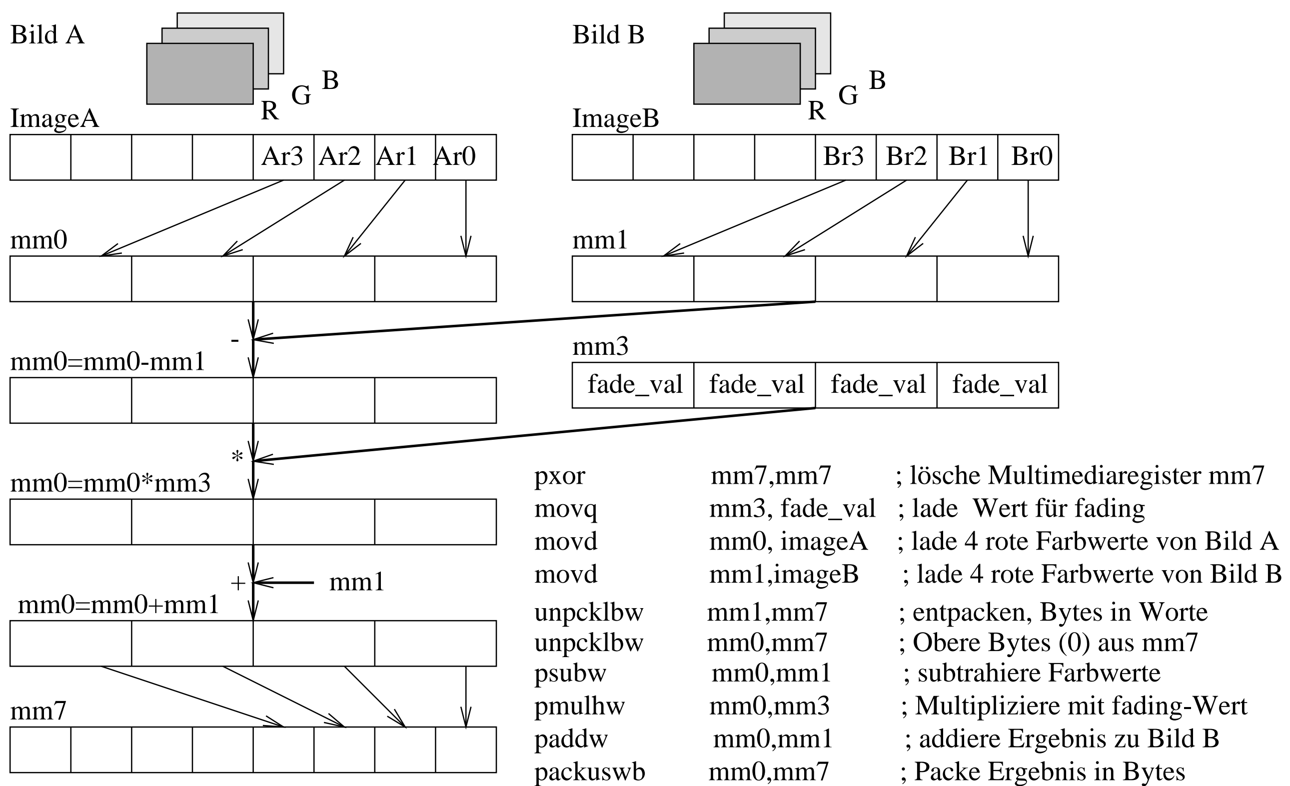

Abbildung 2.7: Skalierte Interpolation (fading) mit MMX-Befehlen

3. Der Ultra-SPARC Prozessor besitzt zur Multimedia-Untersützung den so genannten *visual instruction set* (VIS). Dieser enthält einen Befehl zur Bewegungsschätzung, der bei 8-Bit Daten 8 Subtraktionen, 8 Absolutwerte und 8 Additionen in einem Zyklus ausführt. Auf diese Weise können bei einer Bewegungsschätzung 1.500 konventionelle Befehle durch 32 solcher Befehle ersetzt werden [BK95].

2.6 Graphikprozessoren

6. Vorlesung

Bezüglich der Befehlssätze von Graphikprozessoren verweisen wir auf die Folien. Der erste Teil der Folien entspricht dabei dem Abschnitt über Graphikprozessoren im Kapitel 4 in der fünften Auflage des Buches

von Hennessy/Patterson.

2.7 ASIPs

In dem Bestreben, für eingebettete Prozessoren effiziente Hardware anbieten zu können, ist die Verwendung von Applikations-spezifischen Prozessoren (ASIPs) vorgeschlagen worden [ANH+93]. Für diese wird der Befehlssatz aufgrund der Anwendung festgelegt.

2.8 Abstrakte Maschinen

Eine weitere Besonderheit stellen abstrakte Maschinen dar, die jeweils einen abstrakten Befehlssatz interpretieren. Solche abstrakten Befehlssätze sind insbesondere für die Programmiersprachen (UCSD-) Pascal, PROLOG, LISP, FORTH, Smalltalk und Java vorgeschlagen worden. Die Interpretation dieser abstrakten Befehlssätze vermeidet es, Programme jeweils in die Befehlssätze verschiedener Maschinen übersetzen zu müssen. Lediglich der Interpreter muss für verschiedene Maschinen jeweils neu erzeugt werden. Nachteilig ist allerdings die niedrigere Ausführungsgeschwindigkeit interpretierter Programme.

2.8.1 Java Virtual Machine

Sehr aktuell ist die abstrakte Java-Maschine, *Java Virtual Machine* (JVM) genannt. Folgende Stichworte kennzeichnen die Sprache Java:

- Objektorientierte Programmiersprache
- Datentypen: byte, short, int, long, float, double, char
- Unterstützt Netzwerk-Programmierung
- Im Hinblick auf Sicherheit entworfen:
  - keine Pointer-Manipulation wie in C, C++.
  - beschränkte Möglichkeit, Informationen über die momentane Umgebung (wie z.B. Benutzernamen) zu erfahren
- Automatische Freispeicherverwaltung
- Multithreading

Java soll über Netze auf alle relevanten Maschinen geladen und dort ausgefürhrt werden können. Aus diesem Grund ist für Java eine abstrakte Maschine (JAM) definiert worden. Java-Programme können überall dort ausgeführt werden, wo diese abstrakte Maschine realisiert ist. Diese abstrakte Maschine besitzt die folgenden Eigenschaften [Dal97]:

- Die JAM ist eine Kellermaschine, die in einem Byte Operationscodes kodiert.
- Typische Befehle der JAM sind: lade integer auf den Keller, addiere die obersten Kellerelemente, starte nebenläufige Ausführung, synchronisiere nebenläufige Ausführung, Befehle zur Realisierung der Objektorientierung
- Der Byte-Code ist sehr kompakt (ca. 1/3 des Speicherbedarfs von RISC-Code). Dadurch ist sie besonders für **eingebettete Systeme** interessant, bei denen Programme zusammen mit den Prozessor auf einem Chip gespeichert werden müssen.
- Die JAM verzichtet weitgehend auf Alignment-Beschränkungen.

Es gibt drei Methoden der Realisierung von JAMs:

- Durch **Interpretation** von JAM-Befehlen in Software.
- Durch Übersetzung in den Maschinencode der aufrufenden Maschine unmittelbar vor der Ausführung *just-in-time compilation*.
- Durch Realisierung einer JAM als „echte" Maschine.

Zur Vermeidung der Performance-Verluste durch Interpretation des Byte-Codes ist die PicoJava-Hardware entwickelt worden. Die PicoJava-Maschine ist die Realisierung einer JAM als echte Maschine. Sie führt den Bytecode als hardwareunterstützten Befehlssatz ausführt. Komplexe Byte-Code werden dabei durch ein Mikroprogramm realisiert [Sun97].

PicoJava besitzt die folgenden Eigenschaften:

- Die obersten Kellerelemente werden im Prozessor in einem schnellen Speicher gehalten (64 Einträge bei der PicoJava I). Durch einen *dribbling*-Mechanismus wird versucht, diesen *stack cache* nie zu voll und nie zu leer laufen zu lassen.
- Die häufigsten Befehle werden in Hardware ausgeführt. Von diesen Befehlen kann pro Takt ein Befehl gestartet werden.
- Etwas komplexere Befehle werden mit einem Mikroprogramm ausgeführt.
- Ganz komplexe Befehle (z.B. Thread-Synchronisation) erzeugen einen Interrupt und werden dann in Software ausgeführt.
- Aufgrund der häufigen Folge eines Arithmetik-Befehls auf einen Lade-Befehl wird diese Kombination dynamisch in einen einzigen Befehl umgewandelt. Auf diese Weise soll die Kellermaschinen inhärente Ineffizienz abgemildert werden.
- Der Keller ist so organisiert, dass aufgerufene Methoden die übergebenen Parameter direkt auf dem Keller verarbeiten können und ein Kopieren in den Speicher nicht notwendig ist (Weiterentwicklung der überlappenden Registerfenster der SPARC).
- PicoJava I soll Sun Microelectronics zufolge fünfmal schneller sein als ein gleich schnell getakteter Pentium mit *just-in-time* Compiler.

2.9 Beurteilung von Befehlssätzen

Die folgenden Regeln (z. Tl. aus [HP96] entnommen) helfen bei der Beurteilung von Befehlssätzen:

- Eine variable Befehlswortlänge (wie bei CISC-Rechnern) bzw. kurze Befehlsworte (wie bei DSP-Prozessoren) sind vorteilhaft, wenn es auf eine geringe Codegröße ankommt.
- Eine feste Befehlswortlänge ist vorteilhaft, wenn es v.a. auf die Ausführungsgeschwindigkeit ankommt.
- Condition-Codes können die Codegröße reduzieren, erschweren aber die Fließbandverarbeitung.
- Bei mehreren Speicheroperanden pro Befehl ist es günstig, die Adressierungsart in einem separaten Feld zu kodieren (siehe Motorola 68000).
- Bei höchstens einem Speicheroperanden ist es günstig, die Adressierungsart im Befehlscode zu kodieren.
- Wenn die Effizienz der Implementierung bei hohen Leistungsanforderungen wichtiger ist als die Codekompatibilität (also bei anspruchsvollen eingebetteten Systemen), dann sind VLIW-Maschinen vorteilhaft.
- Wenn die Codekompatibilität höchstes Ziel ist, dann sind abstrakte Maschinen angesagt.

2.10 Nicht-Von-Neumann-Maschinen

Alle bislang besprochenen Befehlssätze basieren auf der Definition des von-Neumann-Rechners (siehe Kap. 1). Implizit wurde stets angenommen. dass Befehle sequentiell aufgrund des **Kontrollflusses** abgearbeitet werden. Eine allgemeinere Unterscheidung als die Unterscheidung von Maschinen anhand des Befehlssatzes ist die Unterscheidung anhand des so genannten **Operationsprinzips** [Gil81]. Das Operationsprinzip legt fest, welcher Mechanismus der Auslösung der elementaren Operationen zugrunde liegt.

Auch moderne Prozessoren enthalten folgende wesentliche Elemente des von-Neumann-Modells:

1. Speicherzellen als **beliebig wiederverwendbare Container** für Werte. Variable in PASCAL sind letztlich auch nur eine etwas abstraktere Notation für Speicherzellen. Die Mathematik benötigt zur Beschreibung von Funktionen keine Speicherzellen.
2. Der explizite, durch Befehle bestimmte **Kontrollfluß** (diese Rechner heißen daher auch *control flow driven*).
3. Bei den meisten Sprachen kommt noch die explizite Sichtbarkeit der Adressen und ggf. eine explizite Speicherverwaltung hinzu. Diese werden zur Definition mathematischer Funktionen ebenfalls nicht benötigt.

Die folgenden Abschnitt beschreiben einige alternative Maschinenmodelle.

2.10.1 Reduktionsmaschinen

Bei Reduktionsmaschinen [Tea82, SK83, Veg84, Kog91] handelt es sich um eine Realisierung einer Maschine, welche die Auswertung funktionaler Programme direkt unterstützt. Reduktionsmaschinen akzeptieren geschachtelte Ausdrücke einer funktionalen Sprache (und keine Sequenzen von Maschinenbefehlen). Sie werten diese Ausdrücke mittels **Baumtransformationen** aus, welche die Ausdrücke reduzieren, bis zum Schluß ein **Wert** erhalten wird. Die Bedeutung eines Programms ist damit durch den abgelieferten Wert gegeben. Durch das Fehlen eines explizit sichtbaren Speichers ist die Verifikation von Programmen vereinfacht. Den theoretischen Hintergrund einer deratigen Programmierung bildet der λ-Kalkül.

Es gibt zwei unterschiedliche Ansätze der Speicherorganisation:

1. *string reduction*

   Jede Operation, die auf eine bestimmte Definition (auf einen bestimmten Teilbaum) zugreift, erhält und bearbeitet eine Kopie des Teilbaums.

   → erleichterte Realisierung, schnelle Bearbeitung skalarer Werte, ineffiziente Behandlung gemeinsamer Teilausdrücke (Beispiel: Der Aufwand für die Berechnung der Fibonacci-Zahlen steigt von linear auf exponentiell).
2. *graph reduction*

   Jede Operation, die auf eine bestimmte Definition zugreift, arbeitet über Verweise auf der Original-Definition.

   → schwieriger zu realisieren, falls parallel bearbeitet wird; effizient auch für strukturierte Werte und gemeinsame Teilausdrücke; komplizierte Haldenverwaltung (*garbage collection*).

Mögliche Ansätze hinsichtlich der Steuerung der Berechnungs-Reihenfolge:

1. Die *outermost*- oder *demand driven*-Strategie:

   Operationen werden selektiert, wenn der Wert, den sie produzieren, von einer bereits selektierten Operation benötigt wird (Aufrufe von der Wurzel zu den Blättern).

   Erlaubt *lazy evaluation*, d.h. redundante Argumente, z.B. bei bedingten Ausdrücken, brauchen nicht berechnet zu werden.

   Erlaubt konzeptuell unendliche Listen, solange nur endl. Teilmengen referenziert werden (vgl. Streams).

2. Die *innermost*- oder *data-driven*-Strategie:
   Operationen werden selektiert, wenn alle Argumente verfügbar sind.

Vorteile (im wesentlichen Vorteile der funktionalen Programmierung im allgemeinen):

- Programmierung auf höherer Ebene
- kompakte Programme
- keine Seiteneffekte
- kein Aliasing, keine unerwartete Speichermodifikation
- keine Unterscheidung zwischen call-by-name, call-by-value und call-by-reference (siehe PSÜ) notwendig
- einfachere Verifikation, da nur Funktionen benutzt werden
- das von-Neumann-Modell von Speicherzellen und Programmzählern ist überflüssig
- beliebige Berechnungsreihenfolge für Argumente (mit Ausnahme von Problemen bei nicht-terminierenden Berechnungen)
- vereinfachte Parallelverarbeitung
- Debugging einfach: Trace des aktuellen Ausdrucks (dies gilt für die „normale" funktionale Programmierung nicht)

2.10.2 PROLOG-Maschinen

Zur direkten Realisierung von logischen Programmiersprachen gibt es v.a. PROLOG-Maschinen.

Hierbei erfolgt meist Übersetzung von PROLOG in den Code der WAM (Warren Abstract Machine). Die Befehle der WAM werden sodann entweder von Maschinenprogrammen interpretiert, als Maschinenbefehle realisiert oder das komplette PROLOG-Programm wird in Maschinenbefehle übersetzt.

Man kann zwischen verschiedenen Klassen von PROLOG-Maschinen unterscheiden:

- Sequentielle Maschinen mit strenger Wahrung der PROLOG-Semantik
- Parallele Maschinen mit unterschiedlicher Semantik

Beispiele von Maschinen:

- Viele Entwürfe innerhalb des japanischen *5th Generation Projekts*.
- PRIPs = Entwurf der Projektgruppe PRIPs (Entwurf eines PROLOG-Chips) an der Uni Dortmund, Informatik XII. 1993 gefertigt und getestet.

2.10.3 Datenflussmaschinen

Bei diesen veranlaßt die Verfügbarkeit von Daten die Ausführung von Operationen [Tea82, Tre84, Den80].

Beispiel:

z = (x + y) * (x - y); Darin sind x,y und z lediglich Benennungen für Werte. Siehe Abb. 2.8.

Modellierung mit Marken:

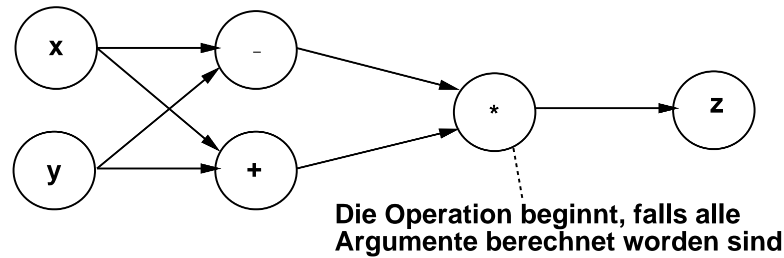

Abbildung 2.8: Datenflußgraph

> Gültige Daten werden als Marken dargestellt. Eine Operation kann ausgeführt werden, falls alle ihre Argumente markiert sind. Potenziell viele Operationen gleichzeitig!

Darstellung der Befehle in der Maschine (Dennis):

> Als Tupel (Opcode, Plätze für Argumente, Ziel-Liste). Die Ziel-Liste ist die Liste der Tupel, die das Ergebnis als Argument benötigen, siehe Abb. 2.9 bis Abb. 2.10.

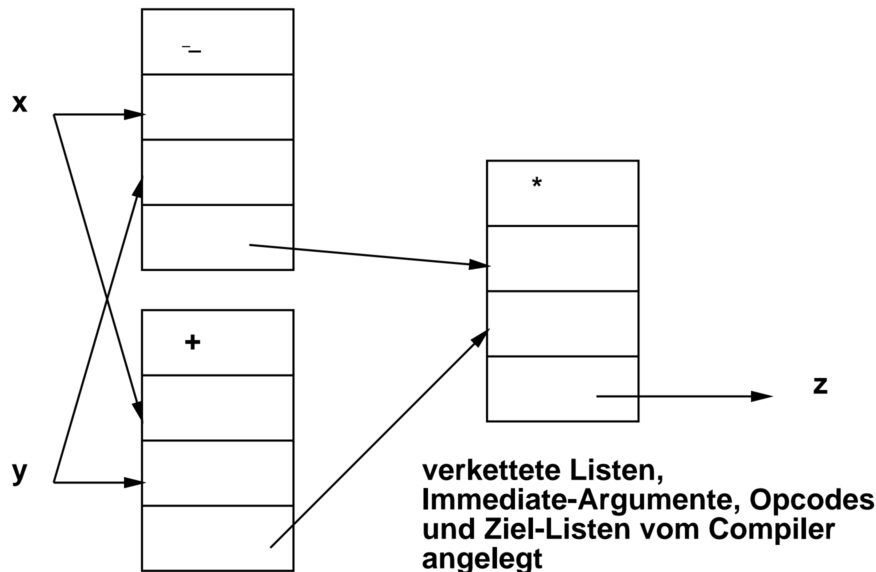

Abbildung 2.9: Implementierung nach Dennis für einfache Ausdrücke

Vorteile der Datenflußrechner sind:

- eingebaute Parallelität
- eingebaute Synchronisation
- Adressen sind nach außen nicht sichtbar
- vorteilhaft bei gemeinsamen Teilausdrücken
- beliebige Reihenfolge der Abarbeitung ausführbereiter Befehle

Während vieler Jahre sah es so aus, als wäre die Entwicklung von Datenflussrechnern in einer Sackgasse gemündet. Ein Grund dafür war sicherlich der zusätzliche Aufwand durch die Verwaltung der Marken usw. Inzwischen sind die Fließbänder normaler von-Neumann-Rechner wegen des Leistungsdrucks so komplex geworden, dass ein Aufwandsvergleich nicht mehr unbedingt zum Nachteil der Datenflussrechner ausfällt. Sie können relativ leicht asynchron realisiert werden und kommen dadurch mit wenig Strom aus. Aus diesem Grund ist offenbar geplant, eine Weiterentwicklung [] dieser Rechner in Videokameras der Fa. Sharp zur Datenreduktion einzusetzen.

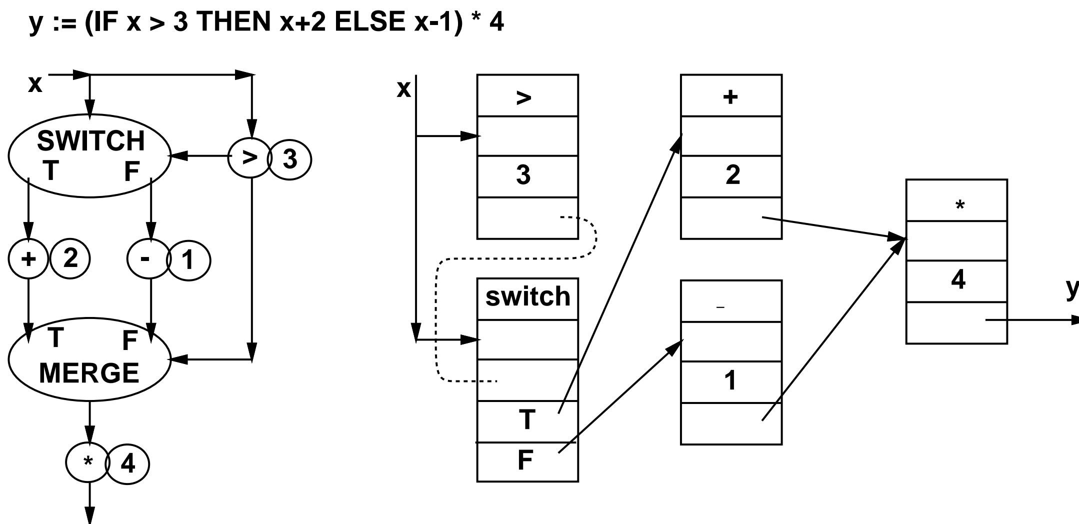

Abbildung 2.10: Implementierung nach Dennis für bedingte Ausdrücke

2.10.4 Weitere Nicht-Von-Neumann-Maschinen

Einige in der Regel auf der Basis elektrischer Prinzipien realisierter Nicht-Von-Neumann-Maschinen, wie z.B. Neuronale Netze, können wir in dieser Vorlesung aus Zeitgründen nicht behandeln.

2.10.4.1 DNA/RNA- und Quantenrechner

DNA/RNA- und Quantenrechner basieren auf anderen Wirkungsprinzipien als die bislang erwähnten elektronischen Rechner. Beide Formen von Rechnern leisten eine Parallelverarbeitung weit jenseits der z.Z. parallel benutzten Prozessorzahlen.

Im Falle von RNA/DNA-Rechnern werden Informationen, beispielsweise mögliche Lösungen, in RNA- bzw. DNA-Strängen kodiert und mit mikrobiologischen Methoden gefiltert. Wegen der großen Zahl von gleichzeitig bearbeitbaren Strängen können riesige Lösungsmengen gleichzeitig auf zulässige Lösungen untersucht werden [Pet00]. Ein einfacher 10-Bit RNA-Rechner ist bereits in Betrieb [Joh00].

Im Falle von Quantenrechnern können sich Nullen und Einsen verschiedener Lösungen in einem Register überlagern. Durch die Form der Parallelarbeit gerät die klassische Komplexitätstheorie an ihre Grenzen. Es wurde nachgewiesen, dass die RSA-Kodierung mit Quantenrechnern effizient geknackt werden kann. Es wird erwartet, dass ein 10-Bit-Quantenrechner in zwei Jahren gebaut werden kann [SR00].

Kapitel 3

Mikroarchitektur

Als nächstes werden wir uns mit der Realisierung von Befehlssätzen mittels Mikroarchitekturen beschäftigen.

3.1 Elementare Datentypen und deren Darstellung

7. Vorles.

Rechner stellen in der Regel gewisse elementare Datentypen zur Verfügung, wobei sie von einer standardisierten Informationsdarstellung (d.h.: Interpretation von Bitvektoren) ausgehen und Grund-Operationen (wie z.B. die Addition) anbieten. Diese Datentypen stellen im weitesten Sinne **abstrakte Datentypen** dar [Gil81], wie sie aus Grundvorlesungen bekannt sind.

Eine besondere Charakteristik des von-Neumann-Rechners ist es, dass auf gespeicherte Werte nicht nur mit den eigentlich dazugehörigen Operationen zugegriffen werden kann, sondern dass beliebige Operationen auf beliebigen Speicherwerten zulässig sind. So verhindert es die Hardware in der Regel nicht, dass Gleitkomma-Zahlen mit Integer-Operationen verarbeitet werden.

Wir werden in separaten Abschnitten Operationen auf Bitvektoren sowie die Operationen für natürliche, für ganze und für Gleitkomma-Zahlen besprechen. Rechnerarithmetik ist an unserer Universität bereits Gegenstand der Vorlesung „Rechnerstrukturen" vor dem Vordiplom. In der Vorlesung „Rechnerarchitektur" wollen wir ausgewählte ergänzende (z.Tl. auch wiederholende) Themen besprechen. Rechnerarithmetik ist bei vielen zwar ein unbeliebtes Thema. Gewisse Grundkenntnisse in diesem Bereich sind aber erforderlich, z.B. wenn Pakete zur mehrfachen Genauigkeit erstellt werden müssen, wenn die vorhandene Arithmetik zur Simulation anderer Datentypen genutzt werden soll und wenn die Genauigkeit von Gleitkomma-Rechnungen beurteilt werden soll. Eine detailliertere Beschreibung von arithmetischen Operationen enthält z.B. das Buch von Spaniol [Spa76].

3.1.1 Bitvektoren

Die meisten Befehlssätze unterstützen Operationen auf Bitvektoren begrenzter Länge (häufig: ein Wort). Wir wollen hier mit der Behandlung von Schiebeoperationen beginnen.

3.1.1.1 Funktionalität

Bei den Schiebeoperationen ist zwischen den so genannten logischen und den arithmetischen Schiebeoperationen zu unterscheiden. Die folgende Liste zeigt die Definition der vier von uns benutzten Schiebeoperationen *shift right logical, shift left logical, shift right arithmetical, shift left arithmetical* jeweils eine Stelle:

(3.1) srl(a) = `'0' & a (a'left downto 1)`

(3.2) sll(a) = `a (a'left-1 downto 0) & '0'`

(3.3) sra(a) = `a (a'left) & a (a'left downto 1)`

(3.4) sla(a) = `a (a'left) & a (a'left-2 downto 0) & '0'`

Schiebeoperationen um n können durch die n-malige Anwendung der Schiebeoperation um eine Stelle erklärt werden.

Neben den Schiebe-Operationen unterstützen Rechner häufig auch Einzelbit-Operationen wie z.B. „Test Bit i", „Set Bit i", usw. Weiterhin können Bitvektoren häufig auch kopiert, nach Mustern durchsucht und gelöscht werden.

3.1.1.2 Realisierung

Realisiert werden Schiebeoperationen auf Bitvektoren um n Stellen häufig durch **Barrelschifter**. Abbildung 3.1 zeigt den Elementarbaustein für das Schieben mit Barrelshiftern um 0-3 Bit bei 4 Ausgängen.

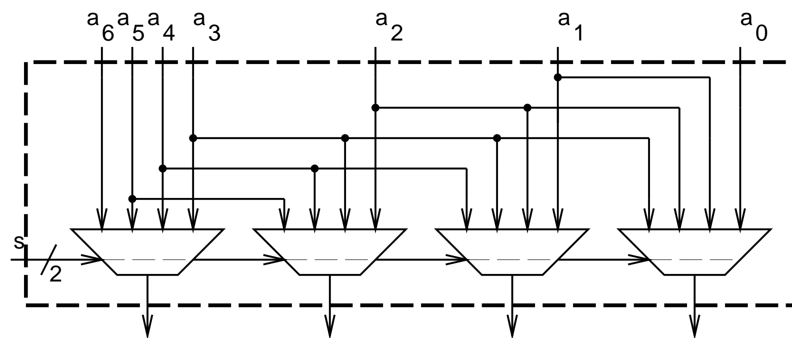

Abbildung 3.1: Elementarbausein, 4 Ausgänge, Schieben um 0-3 Stellen

Durch Kombination dieser Elementarbausteine kann man die Anzahl der Ausgänge erhöhen, siehe Abb. 3.2.

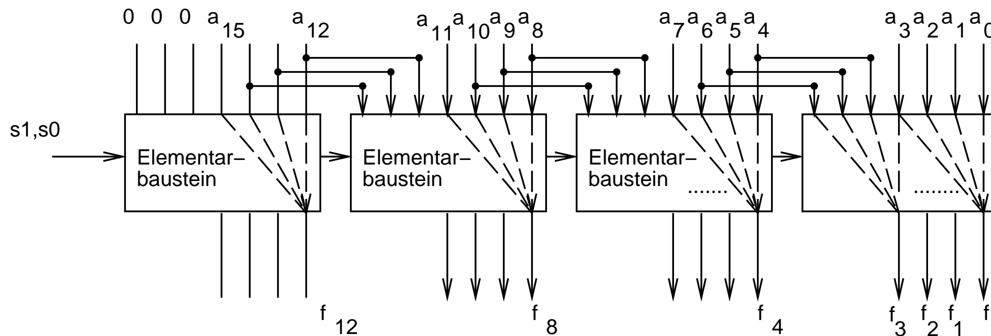

Abbildung 3.2: Barrelshifter mit 16 Ausgängen

Schließlich lässt sich durch geschickte Verschaltung auch die Anzahl der Stellen erhöhen, um die geschoben wird, siehe Abb. 3.3.

Mit diesem kombinatorischen Schaltungselement kann ein Bitvektor um 0-15 Stellen verschoben werden.

Anhand dieser Schaltungen sollte deutlich werden, dass die Operation „Schieben um n Stellen" durch eine kombinatorische Schaltung realisiert werden kann, also nicht auch n Schritte einer sequentiellen Schaltung benötigt. Dies ist eigentlich selbstverständlich, wird aber aufgrund von Einschränkungen in manchen Maschinenbefehlssätzen leicht vergessen.

Barrel-Shifter sind (in Kombination mit Prioritätsencodern) sehr gut zur Beschleunigung des Booth-Algorithmus (s.u.) sowie zur Normalisierung von Gleitkommazahlen geeignet.

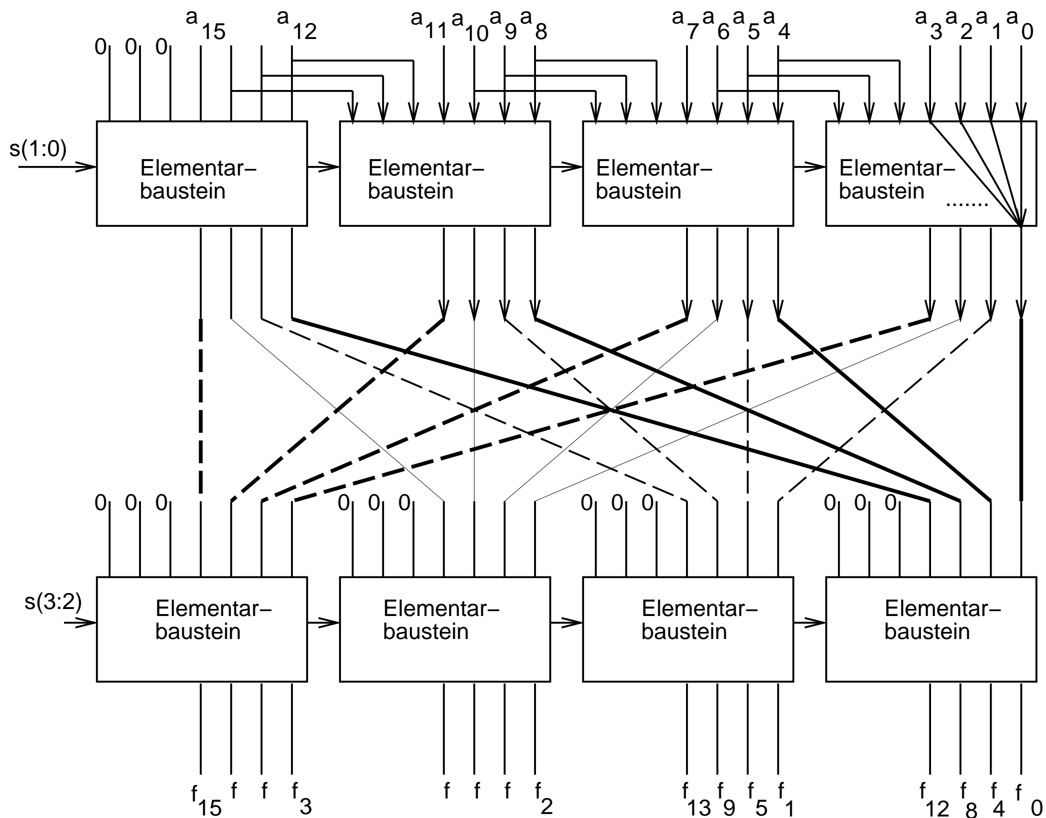

Abbildung 3.3: Barrelshifter für das Schieben um 0 bis 15 Stellen

3.1.2 Natürliche Zahlen

Als ersten Zahlendatentyp wollen wir die natürlichen Zahlen behandeln.

3.1.2.1 Funktionalität

Die folgende Funktion beschreibt die übliche Interpretation von Bitvektoren als natürliche Zahlen:

$$\texttt{nat(a)} \quad = \quad \sum_{i=0}^{a'left} a_i * 2^i \tag{3.5}$$

Dementsprechend stellt `"1000"` also beispielsweise eine 8 dar.

Übungsaufgaben:

- Zeigen Sie, dass die folgende Aussage gilt: `nat(srl(a))` $\rightarrow$ `nat(a) / 2` für alle Bitvektoren `a`. Dabei sei '/' die ganzzahlige Division.
- Unter welcher Vorraussetzung gilt `nat(sll(a))` = `nat(a) * 2`?

<u>Addition</u>

Die Addition liefert die Summe natürlicher Zahlen, soweit dies aufgrund der üblicherweise festen Datenwortlänge möglich ist.

Im folgenden wollen wir annehmen, dass die Argumente der Grundrechenoperation `+` durch Bitvektoren $a = (a_{n-1}, .., a_0)$ und $b = (b_{n-1}, .., b_0)$ repräsentiert sind.

Wir wollen untersuchen, wie man erkennen kann, ob das Ergebnis einer Addition außerhalb des darstellbaren Zahlenbereichs liegt, also ob das Ergebnis der Operation mit einem Bitvektor $f = (f_{n-1}, .., f_0)$ der Länge n dargestellt werden kann.

Wir möchten gerne wissen: wann ist

$$nat(a) + nat(b) \quad \geq \quad 2^n \tag{3.6}$$

(wobei 2^n die erste nicht mehr durch f darstellbare Zahl ist) ? Wir möchten also wissen, ob die folgende Ungleichung gilt:

$$\sum_{i=0}^{n-1} a_i * 2^i + \sum_{i=0}^{n-1} b_i * 2^i \quad \geq \quad 2^n \tag{3.7}$$

Dies ist gerade dann der Fall, wenn bei der Addition ein Übertrag in die Stelle n hinein entsteht, also der Übertrag $c_n =' 1'$ ist. Bei der Addition **natürlicher** Zahlen ist der Übertrag c_n in die nächste Stelle also gleich dem Überlauf, den wir mit cf_+ bezeichen wollen.

Für c_n gilt:

$$\begin{aligned} cf_+ = c_n \quad &= \quad (a_{n-1}b_{n-1}) \vee ((a_{n-1} \text{ xor } b_{n-1})c_{n-1}) \\ &= \quad (a_{n-1}b_{n-1}) \vee ((a_{n-1} \vee b_{n-1})c_{n-1}) \\ &= \quad (a_{n-1}b_{n-1}) \vee (a_{n-1}c_{n-1}) \vee (b_{n-1}c_{n-1}) \end{aligned}$$

Dabei haben wir davon Gebrauch gemacht, dass wir *xor* durch ein normales „oder" ersetzen können, weil der erste Term für $a_{n-1} = b_{n-1} = 1$ ohnehin eine '1' liefert.

Diese Gleichung ist jedoch in der Praxis vielfach nicht anwendbar, weil der interne Übertrag c_{n-1} meist nur innerhalb des Addierers verfügbar ist und auch auf Maschinen-Sprachebene und in höheren Programmiersprachen ist der Wert von c_{n-1} nicht abfragbar. Wir versuchen daher, ihn durch einen verfügbaren Wert zu ersetzen. Abb. 3.4 (links) zeigt die Realisierung von c_n aus den drei Termen gemäß Gleichung 3.8. Abb. 3.4 (mitte) zeigt die Funktion $\overline{f_{n-1}}$.

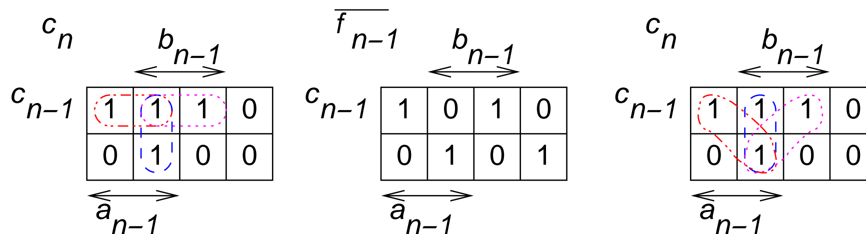

Abbildung 3.4: c_n und f_{n-1}

Durch Vergleich folgt, dass c_n auch durch die drei Terme der Abb. 3.4 (rechts) dargestellt werden kann, dass also gilt:

$$c_n \quad = \quad (a_{n-1}b_{n-1}) \vee (a_{n-1}\overline{f_{n-1}}) \vee (b_{n-1}\overline{f_{n-1}})$$

Der Wert von $\overline{f_{n-1}}$ kann meist abgefragt werden, sowohl in Schaltungen als auch in Programmiersprachen. Bei letzteren genügt ein Test „< 0" bei Interpretation des Ergebnisses als Zweierkomplementzahl.

Im Falle eines Überlaufs liefern Rechner meist ein Ergebnis nach der so genannten *wrap around*-Arithmetik. Dies bedeutet, dass im Ergebnis nicht darstellbare Überträge einfach weggelassen werden. Eine Addition `a+b = "1000" + "1000"` liefert also den Wert `"0000"` (siehe Abb. 3.1).

	wrap-around-Arithmetik	Sättigungsarithmetik
`a`	`"1000"`	`"1000"`
`b`	`"1000"`	`"1000"`
`a+b`	`"0000"`	`"1111"`

Tabelle 3.1: Ergebnisse der Addition natürlicher Zahlen bei Bereichsüberlauf

Würden diese Vektoren z.B. Helligkeiten darstellen, so ergäbe die Addition halbheller Werte bei *wrap-around*-Arithmetik einen schwarzen Wert. Bei Audio- und Videoanwendungen ist es bei Bereichsüberschreitungen meist besser, den größten darstellbaren Wert als Ergebnis abzuliefern. In dem o.a. Fall würde also das Ergebnis `"1111"` abgeliefert werden, der Mittelwert wäre `"0111"`, also nicht ‘ganz so falsch’ wie `"0000"` . Gewisse Fehler werden bei der (verlustbehafteten) Datenreduktion von Audio- und Videodaten ohnehin zugelassen.

Entsprechend ist es besser, bei Bereichsunterschreitungen den kleinsten darstellbaren Wert abzuliefern, bei natürlichen Zahlen also `"0000"` . Eine Arithmetik mit diesem Verhalten heißt **Sättigungsarithmetik**. Diese Arithmetik verhält sich auch bei den übrigen Rechenoperationen entsprechend. Auf eine solche Arithmetik kann bei vielen Prozessoren für die digitale Signalverarbeitung (DSPs) umgeschaltet werden.

Subtraktion

Diese Operation liefert die Differenz natürlicher Zahlen, soweit dies aufgrund der üblicherweise festen Datenwortlänge möglich ist.

Bei der Subtraktion entsteht ein Überlauf cf_-, falls $a_{n-1} = 0$ ist und $b_{n-1} = 1$ oder $c_{n-1} = 1$ sind, sowie im Fall $a_{n-1} = 1$ und sowohl b_{n-1} als auch c_{n-1} sind gleich ’1’. Das KV-Diagramm nach Abb. 3.5 (links) zeigt die Situation.

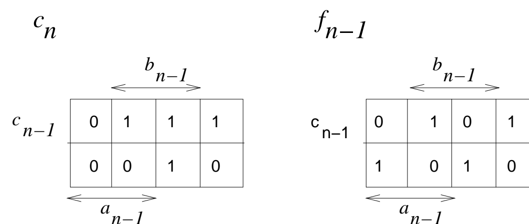

Abbildung 3.5: c_n und f_{n-1}

Gemäß linkem Teil der Tabelle gilt also für cf_-:

$$cf_- := (\overline{a_{n-1}} b_{n-1}) \vee (b_{n-1} c_{n-1}) \vee (\overline{a_{n-1}} c_{n-1})$$

Wieder entsteht das Problem, dass c_{n-1} meist nicht zugreifbar ist. Wir versuchen daher, statt c_{n-1} wieder f_{n-1} zu verwenden. f_{n-1} ist gleich ’1’, falls die drei beteiligten Variablen insgesamt eine ungerade Anzahl von Einsen besitzen (siehe rechten Teil des obigen KV-Diagramms). Unter Ausnutzung von f_{n-1} können wir cf_- schreiben als:

$$cf_- := (\overline{a_{n-1}} b_{n-1}) \vee (b_{n-1} f_{n-1}) \vee (\overline{a_{n-1}} f_{n-1})$$

Damit ist das Ziel wieder erreicht.

Viele Rechner hinterlegen als Ergebnis einer Subtraktion im so genannten Condition-Code Register eine Anzeige des Ergebnisses der Subtraktion. Insbesondere wird meist angezeigt, ob das Ergebnis Null und ob es negativ war. Dies leisten die Anzeigen *zf* und *sf*, die wie folgt definiert sind:

Def.: $zf =' 1' \iff \forall i \in [0..n-1] : f_i =' 0'$

Def.: $sf =' 1' \iff f_{n-1} =' 1'$

Für Vergleichoperationen mit natürlichen Zahlen ergeben die in Tabelle 3.2 gezeigten einfachen Umformungen die Formeln zur Berechnung von des Ergebnisses aus den Werten von zf und dem oben definierten cf_-.

$$\begin{array}{lllll}
a < b & \iff & (a-b) < 0 & \iff & cf_- = 1 \\
a \geq b & \iff & \neg((a-b) < 0) & \iff & cf_- = 0 \\
a > b & \iff & ((a-b \geq 0) \wedge (a-b) \neq 0) & \iff & (cf_- = 0) \wedge (zf = 0) \\
a \leq b & \iff & \neg(a > b) & \iff & (cf_- = 1) \vee (zf = 1)
\end{array}$$

Tabelle 3.2: Berechnung der Vergleichsergebnisse aus den Condition-Codes

Die Multiplikation liefert das Produkt natürlicher Zahlen, soweit dies aufgrund der üblicherweise festen Datenwortlänge möglich ist. Auf Maschinensprachebene kann das Produkt meist durch einen Bitvektor dargestellt werden, welcher die doppelte Länge der Bitvektoren der Argumente hat. Der entsprechend vergrößerte Zahlenbereich steht in höheren Programmiersprachen meist nicht zur Verfügung.

3.1.2.2 Realisierung

Addition und Subtraktion werden in der Praxis in sog. arithmetisch/logischen Einheiten (ALUs) realisiert. Eine bekannte ALU ist die „74181”, die in vielen Technologien realisiert wird. Mit Hilfe von Steuersignalen S3,S2,S1,S0 und M können die in Abb. 3.6 aufgeführten Funktionen ausgewählt werden.

Steuerkode				$M =' 1'$ (logische Op.)	$M =' 0'$ (arithm. Op.)	$M =' 0'$
S3	S2	S1	S0		$c_0 =' 0'$	$c_0 =' 1'$
'0'	'0'	'0'	'0'	$F := \overline{A}$	$F := A$	$F := A + 1$
'0'	'0'	'0'	'1'	$F := \overline{A \vee B}$	$F := A \vee B$	$F := (A \vee B) + 1$
'0'	'0'	'1'	'0'	$F := \overline{A} \wedge B$	$F := A \vee \overline{B}$	$F := (A \vee \overline{B}) + 1$
'0'	'0'	'1'	'1'	$F := 0$	$F := -1_{10}$	$F := 0$
.	.	.	.	...	$F := ...$	$F := ...$
'0'	'1'	'1'	'0'	$F := A \neq B$	$F := A - B - 1$	$F := A - B$
.	.	.	.	...	$F := ...$	$F := ...$
'1'	'0'	'0'	'1'	$F := A = B$	$F := A + B$	$F := A + B + 1$
.	.	.	.	...	$F := ...$	$F := ...$
'1'	'1'	'1'	'0'	$F := A \vee B$	$F := (A \vee \overline{B}) + A$	$F := (A \vee \overline{B}) + A + 1$
'1'	'1'	'1'	'1'	$F := A$	$F := A - 1$	$F := A$

Abbildung 3.6: Von der ALU 74181 angebotene Operationen

Die Multiplikation kann für natürliche Zahlen nach der für die Basis 2 modifizierten Schulmethode erfolgen. Anhand eines Beispiels sei das Vorgehen erläutert:

```
10 *        1010   A
13          1101   B
        --------
        00000000   P_0 = 0
            1010
        --------
        00001010   P_1 = P_0 + B_0 * 2^0 * A
           0000
        --------
        00001010   P_2 = P_1 + B_1 * 2^1 * A
          1010
        --------
        00110010   P_3 = P_2 + B_2 * 2^2 * A
         1010
        --------
= 130   10000010   P_4 = P_3 + B_3 * 2^3 * A
```

Das Partialprodukt P_{i+1} kann also $\forall i = 0..(n-1)$ wie folgt gebildet werden:

$P_{i+1} := P_i + B_i * 2^i * A$ mit $P_0 = 0$.

Auch Multiplikationen können bei entsprechendem Hardware-Aufwand kombinatorisch ausgeführt werden. Hierfür verweisen wir wieder auf den Kurs "Rechnerstrukturen". Stichwort "Wallace-Trees".

3.1.2.3 Natürliche Zahlen in VHDL

Falls der Zahlenbereich des VHDL-Datentyps `natural` ausreicht, kann `nat` mit der folgenden VHDL-Funktion berechnet werden :

```
function nat (a: bit_vector) return natural is
 variable s : natural;
 begin
  s:= 0;
  for i in a'range loop -- absteigender
   s := s + s;          -- Schleifenindex
   if a(i) = '1' then s := s + 1;
   end if;
  end loop;
  return s;
 end nat;
```

Diese Funktion kann mit Bitvektoren mit unterschiedlichem Indexbereich aufgerufen werden. Mit Hilfe des Attributs `'range` kann man im Rumpf der Funktion auf den Indexbereich des aktuellen Parameters Bezug nehmen, wie dies hier in der `for`-Schleife ausgenutzt wurde.

Die Funktion `nat` wird häufig benötigt, z.B. wenn Arrays indiziert werden. Beispiel:

`a (nat (b) )`

Dabei sei `a` ein beliebiges Array und `b` ein Bitvektor.

Um den Zugriff auf ein Array deutlich von einem Funktionsaufruf zu unterscheiden, werden wir gelegentlich eckige Klammern benutzen:

`a [nat(b)]`

Der Kürze halber werden wir gelegentlich den Aufruf der Funktion `nat` bei der Indizierung von Arrays fortlassen. Die Semantik kann jedoch stets als durch die ausführlichere Schreibweise definiert gelten.

3.1.3 Ganze Zahlen

8. Vorles.

3.1.3.1 Funktionalität

Bezüglich der ganzen Zahlen nehmen wir stets die Darstellung im **Zweierkomplement** an, weil dies heute in Rechnern praktisch ausschließlich verwendet wird. Bitvektoren $a = (a_n, ...a_0)$ werden bei dieser Darstellung gemäß der folgenden Formel als Zahlen interpretiert:

$$\texttt{int(a)} = \sum_{i=0}^{a'left-1} a_i * 2^i - 2^{a'left} * a_{a'left} \qquad (3.8)$$

Dementsprechend stellt "1000" also beispielsweise eine -8 und "1001" eine -7 dar.

Übungsaufgaben:

- Zeigen Sie, dass die folgende Aussage gilt: `b=int(sra(a))` → `b=int(a) / 2` für alle Bitvektoren `a`. Dabei sei '/' die ganzzahlige Division.
- Unter welcher Vorraussetzung gilt `int(sla(a))` = `int(a) * 2`?

Bei der Interpretation von Bitvektoren $a_{n-1}..a_0$ bzw. $b_{n-1}..b_0$ gemäß `int` als ganze Zahlen ist der in Abb. 3.7 gezeigte Zahlenbereich darstellbar.

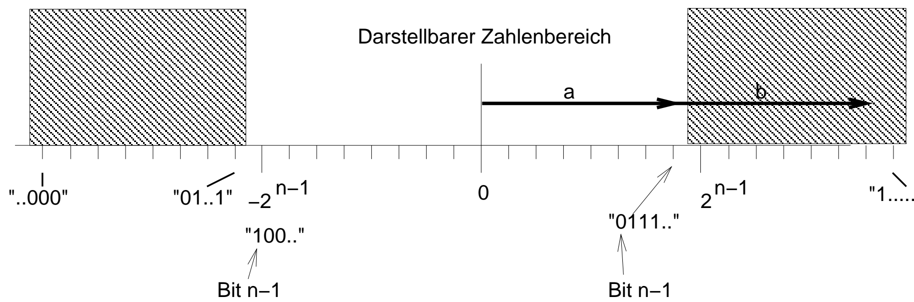

Abbildung 3.7: Darstellung ganzer Zahlen, Überlauf bei zwei positiven Zahlen

Addition

Bei der Addition tritt ein Überlauf auf, wenn beide Operanden das gleiche Vorzeichen haben, das Ergebnis aber das entgegengesetzte Vorzeichen hat (siehe Tabelle 3.3).

a_{n-1}	b_{n-1}	Überlauf unter der Bedingung
0	0	$f_{n-1} = 1$
0	1	nicht möglich
1	0	nicht möglich
1	1	$f_{n-1} = 0$

Tabelle 3.3: Bedingungen des Überlaufs bei der Addition von ganzen Zahlen

Dies gilt auch für die Fälle $int(a) = -2^{n-1}$ bzw. $int(b) = -2^{n-1}$, die häufig eine Sonderbehandlung erfordern. Es gilt also (siehe auch: Bähring, Anhang.):

$$overflow_add(a, b) = (a_{n-1} \equiv b_{n-1}) \wedge (a_{n-1} \text{ xor } f_{n-1})$$

Rechner speichern das Ergebnis dieser Berechnung nach Additionen meist in einem Bit des so genannten Condition Code Registers. In einem folgenden Befehl kann dann mittels eines *branch if overflow*-Befehls

bei einem Überlauf eine Verzweigung ausgelöst werden. Man beachte, dass der Rechner üblicherweise nicht 'weiß', ob er natürliche oder ganze Zahlen addiert und dass erst die 'richtige' Wahl des Verzweigungsbefehls eine bestimmte Interpretation zugrunde legt.

Etwas anders sind die Verhältnisse bei Sättigungsarithmetik: bei ganzen Zahlen sind die größten und die kleinsten Zahlen anders darzustellen als bei natürlichen Zahlen. Ist Sättigungsarithmetik sowohl für natürliche wie auch für ganze Zahlen realisiert, so werden separate Maschinenbefehle für die Grundoperationen beider Datentypen benötigt.

Subtraktion

Die Bedingungen bei der Subtraktion zeigt die Tabelle 3.4.

a_{n-1}	b_{n-1}	Überlauf unter der Bedingung
0	0	nicht möglich
0	1	$f_{n-1} = 1$
1	0	$f_{n-1} = 0$
1	1	nicht möglich

Tabelle 3.4: Bedingungen des Überlaufs bei der Subtraktion von ganzen Zahlen

Bei der Subtraktion tritt also ein Überlauf auf, wenn beide Operanden entgegengesetztes Vorzeichen haben und das Ergebnis $f_{n-1}..f_0$ ein anderes Vorzeichen als der erste Operand hat:

$$overflow_sub(a,b) = (a_{n-1} \text{ xor } b_{n-1}) \wedge (a_{n-1} \text{ xor } f_{n-1})$$

Größenvergleich

Für Zahlen in Zweierkomplementdarstellung gilt: wegen

$$overflow_sub(a,b) = (a_{n-1} \text{ xor } b_{n-1}) \wedge (a_{n-1} \text{ xor } sf) = (\overline{a_{n-1}} \wedge b_{n-1} \wedge sf) \vee (a_{n-1} \wedge \overline{b_{n-1}} \wedge \overline{sf})$$

folgt:

$$\begin{aligned} sf =' 1' &\rightarrow overflow_sub = \overline{a_{n-1}} \wedge b_{n-1} \\ sf =' 0' &\rightarrow overflow_sub = a_{n-1} \wedge \overline{b_{n-1}} \end{aligned}$$

Daraus ergibt sich die folgende Tabelle 3.5.

overflow_sub	sf	Kommentar	Ergebnis
'0'	'0'	kein Überlauf, F positiv, $0 \leq a - b \leq 2^{n-1} - 1$	$a \geq b$
'0'	'1'	kein Überlauf, F negativ, $-2^{n-1} \leq a - b < 0$	$a < b$
'1'	'0'	$(a_{n-1} = 1) \wedge (b_{n-1} = 0)$: a negativ, b positiv	$a < b$
'1'	'1'	$(a_{n-1} = 0) \wedge (b_{n-1} = 1)$: a positiv, b negativ	$a > b$

Tabelle 3.5: Bedingungen für Überläufe

Daraus ergeben sich folgende Beziehungen:

$$\begin{aligned} a < b &\iff (overflow_sub \text{ xor } sf) & & \\ a \leq b &\iff (a < b) \vee (a = b) &&\iff (overflow_sub \text{ xor } sf) \vee zf \\ a > b &\iff \neg(a \leq b) &&\iff \neg((overflow_sub \text{ xor } sf) \vee zf) \\ a \geq b &\iff \neg(a < b) &&\iff overflow_sub \equiv sf \end{aligned}$$

Diese Beziehungen werden vielfach benutzt, um anhand der Werte im Condition-Code Register Verzweigungsbedingungen abzutesten. Insbesondere ist zu beachten, dass die Größenvergleiche bei vorzeichenbehafteten

Zahlen wegen der möglichen Überläufe nicht einfach durch Subtraktion und Vorzeichentest des Ergebnisses erfolgen dürfen!

Multiplikation

Diese Operation liefert das Produkt ganzer Zahlen, soweit dies aufgrund der üblicherweise festen Datenwortlänge möglich ist. Auf Maschinensprachebene kann das Produkt meist durch einen Bitvektor dargestellt werden, welcher die doppelte Länge der Bitvektoren der Argumente hat. Der entsprechend vergrößerte Zahlenbereich steht in höheren Programmiersprachen meist nicht zur Verfügung.

3.1.3.2 Integer in VHDL

Falls der Zahlenbereich des VHDL-Datentyps `integer` ausreicht, kann `int` mit der folgenden VHDL-Funktion berechnet werden :

```
function int (a: bit_vector) return integer is    -- Annahme: a'left > 0
constant t : natural := (2 ** (a'left));
 begin
  if a(a'left) = '0'     - - positive Zahl
  then return nat (a)
  else return (nat (a(a'left-1 downto 0)) -t)
  end if;
 end int;
```

In diesem Fall wurde mit dem Attribut `a'left` die „linke” Grenze des Indexbereichs des aktuellen Parameters abgefragt.

Man beachte, dass 2^n eventuell nicht mehr als VHDL-Zahlentyp darstellbar ist. Sofern 2^n gerade die erste nicht mehr darstellbare Zahl ist, kann es helfen, den Wert $-(2)^{n-1} + nat(a(a'left - 1 \text{ downto } 0)) - (2)^{n-1}$ zu berechnen. Im allgemeinen Fall wird man jedoch nicht jeden (langen) Bitstring in die Integer-Darstellung eines bestimmten VHDL-Simulationsystems konvertieren können.

3.1.3.3 Realisierung

Für die Subtraktion kann ähnlich wie für die Addition dieselbe Schaltung wie für natürliche Zahlen benutzt werden.

Booth-Algorithmus

Ganze Zahlen können analog zu dem Verfahren für natürliche Zahlen multipliziert werden, indem ggf. vor und nach der Multiplikation das Zweierkomplement gebildet wird. Diese (zeitraubende) Sonderbehandlung des Vorzeichens kann beim **Booth–Algorithmus** vermieden werden. Die Grundidee besteht darin, dass für eine Kette von Einsen nur 2 Additionen erforderlich sind. In dem folgenden Beispiel sei i die Position der am weitesten rechts und j die Position der am weitesten links stehenden 1:

$$A * \texttt{int}(\texttt{"}0001111000\texttt{"})$$

Ein Bitstring, der ab der Position j rechts nur noch Einsen enthielte, hätte den Wert $2^{j+1} - 1$. Durch die Nullen am rechten Rand wird der Wert um $2^i - 1$ kleiner. Es gilt also:

$$A * \texttt{int}(\texttt{"}0001111000\texttt{"}) = A * (2^{j+1} - 1 - 2^i + 1) = A * (2^{j+1} - 2^i)$$

Auf der stellengerechten Addition bzw. Subtraktion von A basiert der folgende `Booth`-Algorithmus:

```
FUNCTION Booth(A,B: IN bit_vector) RETURN bit_vector IS
```

```
    CONSTANT n : natural := A'LENGTH; -- Vor.: A'LENGTH = B'LENGTH
    VARIABLE P : bit_vector(n-1 DOWNTO 0) := (OTHERS => '0');
    VARIABLE Q : bit_vector(n DOWNTO 0)   := (OTHERS => '0');
  BEGIN
   Q(n DOWNTO 1) := B;
   FOR i IN 0 TO n-1 LOOP
     CASE Q(1 DOWNTO 0) IS
       WHEN "10" => P := P - A;
       WHEN "01" => P := P + A;
       WHEN OTHERS =>          -- keine Aktion
     END CASE;
     -- arithmetisches Schieben (1 Bit nach rechts)
     P & Q := sra (P & Q);
   END LOOP;
   RETURN P(n-2 DOWNTO 0) & Q(n DOWNTO 1);
  END Booth;
```

Beispiel:

Sei:

$$A = int(\texttt{"0011"}) = 3; B = int(\texttt{"0110"}) = 6$$

Es gilt also:

$$-A = int(\texttt{"1101"}); n = 4$$

Die Multiplikation mit 6 wird hier zur Multiplikation mit 8 und zur anschliessenden Subtraktion von $A * 2$:

$$A * 6 = A * int(\texttt{"0110"}) = A * (2^3 - 2^1)$$

Die Einzelschritte der Prozedur `Booth` sind (sra = Schiebeoperation):

```
Operation    P      Q    Kommentar
            0000  00000
            0000  01100  Q(1 DOWNTO 0) = ''00''
sra         0000  00110  Q(1 DOWNTO 0) = ''10''
-A          1101  00110  Subtraktion 1 Bit vom späteren LSB entfernt = -2* A
sra         1110  10011  Q(1 DOWNTO 0) = ''11''
sra         1111  01001
+A          0010  01001  ''1111''+''0011''=''0010'';3 Bits vom späteren LSB=+8*A
sra         0001  00100
                     ↑ LSB des Ergebnisses
             ↑ MSB des Ergebnisses
```

Das Ergebnis ist $int(\texttt{"0010010"}) = 18$.

Der „Trick" hinsichtlich der Behandlung des Vorzeichens von A besteht darin, dass P und Q zusammen eine Zahl speichern, die als Zweierkomplement-Zahl interpretiert wird, da beim Schieben das Vorzeichen erhalten bleibt. Wegen der Subtraktion von A muss ohnehin mit Vorzeichen gearbeitet werden. Allerdings funktioniert der Algorithmus nicht für die kleinste negative Zahl mit der Kodierung A="100...00" da für diese Zahl $-A$ nicht in derselben Anzahl von Bits darstellbar ist.

Wir wollen nun zeigen, dass der Booth-Algorithmus tatsächlich Zahlen gemäß der Integer-Interpretation der Bitvektoren A und B multipliziert. Wir zeigen dies für den oben verwendeten Fall von *B'length=4*. Die Verallgemeinerung auf Bitvektoren belieber Länge wird offensichtlich sein. Seien im folgenden die Elemente von `B` mit b_0 bis b_3 bezeichnet.

Diese Wirkung eines einzelnen Schrittes des Booth-Algorithmus besteht dann in einer stellengerechten Multiplikation von A mit dem Ausdruck $(b_{i-1} - b_i)$, der die Werte 0, -1 und +1 annimmt. Insgesamt berechnet der Booth-Algorithmus also den Ausdruck

$$\begin{array}{lcl} A & * & (b_{-1} - b_0) * 2^0 + \\ A & * & (b_0 - b_1) * 2^1 + \\ A & * & (b_1 - b_2) * 2^2 + \\ A & * & (b_2 - b_3) * 2^3 \end{array}$$

mit $b_{-1} = 0$.

Mit $-b_i * 2^i + b_i * 2^{i+1} = b_i * 2^i$ folgt, dass der folgende Ausdruck berechnet wird:

$$A * (-b_3 * 2^3 + b_2 * 2^2 + b_1 * 2^1 + b_0 * 2^0) \quad = \quad A * int(B)$$

Der Ausdruck in der Klammer entspricht genau der Integer-Interpretation des Bitvektors B.

Der Trick der korrekten Behandlung von B als Integer-Zahl besteht darin, dass „vergessen" wird, bei einer '1' im Vorzeichen von B die eigentlich übliche Behandlung am linken Rand einer Folge von Einsen vorzunehmen. Der „Fehler" bewirkt gerade, dass b_3 mit negativem Vorzeichen in den o.a. Ausdruck eingeht und B damit gemäß `int` und nicht gemäß `nat` interpretiert wird.

Verbesserungen des Booth-Algorithmus:

- Einzelne Nullen bzw. Einsen sollen wie beim Standardverfahren nur eine Operation erzeugen.
  - ⇒ Bei isolierter 1 im Bit i wird $2^{i+1} - 2^i = 2^i$ addiert.
  - ⇒ Bei isolierter 0 im Bit i wird $2^i - 2^{i+1} = -2^i$ addiert.
  - ⇒ Übergang auf die Betrachtung eines Fensters von 3 Bit und Verschiebung um jeweils 2 Bit; abhängig vom Muster im Fenster Addition von $\pm 2 * A, \pm A$ (siehe Hayes)
- Ignorieren von Folgen gleicher Ziffern

  Statt jeweils um ein Bit zu schieben, kann man auch gleich in einem Schritt bis zum nächsten Rand einer Folge von Einsen schieben. Die bereits vorgestellten Prioritätsencoder sind geeignet, den Abstand bis zur nächsten '1' effizient zu finden. Mit zwei geeignet verschalteten Prioritätsencodern können sowohl linke als auch rechte Ränder der nächsten Folge von Einsen erkannt werden. In Kombination mit dem ebenfalls vorgestellten Barrelshifter kann so ein Schieben bis zum nächsten Rand einer Folge von Einsen vorgenommen werden.

  Sofern diese Kombination von Barrelshifter und Prioritätsencoder keinen Rand mehr finden kann, sofern also `B` nach einer gewissen Anzahl von Schritten **nur** noch Nullen oder Einsen enthält, kann das Verfahren abgebrochen werden. Wichtig ist dies für die häufig vorkommende Multiplikation von kleinen Zahlen. Bemerkenswert ist, dass dies auch für (betragsmäßig) kleine negative Zahlen funktioniert. So kann die Multiplikation von 32-Bit-Zahlen z.B. in 2 oder 3 Schritten abgeschlossen sein!

3.1.4 Gleitkomma-Zahlen

Die Funktionalität von Gleitkomma-Zahlen wurde im Kurs "Rechnerstrukturen" ausführlich besprochen. Uns bleibt nur noch nachzutragen, wie mit diesen Datentypen auf der Ebene höherer Programmiersprachen eigentlich umzugehen ist.

Generell gibt es das Problem, dass der IEEE-Standard keinerlei Aussagen über die Unterstützung in höheren Programmiersprachen macht und auch die mögliche Unterstützung bei Verabschiedung des Standards nicht bedacht worden ist. Der Grund dafür: unter den Gleitkomma-Experten, die den Standard vorbereitet haben, gab es keinen Compiler-Spezialisten!

Wir betrachten hier vor allem die Frage, mit welcher Genauigkeit Zwischenrechnungen ausgeführt werden sollen. Entsprechende Überlegungen müssen v.a. beim Bau von Compilern bedacht werden.

Es gibt zunächst zwei intuitive Ansätze, die aber zu unakzeptablen Ergebnissen führen:

1. Man nehme für alle Zwischenrechnungen die maximal verfügbare Genauigkeit.

   Dies führt zu unerwarteten Ergebnissen. Beispiel:

   Sei `q` eine Variable einfacher Genauigkeit (32 Bit). Dann führt die Sequenz

   ```
   q :=3.0/7.0; print(q=(3.0/7.0))
   ```

   zum Ausdruck von `false`, denn bei der Zuweisung von `3.0/7.0` zu `q` gehen Mantissenstellen verloren. Wird der Vergleich (als „Zwischenrechnung") mit doppelter Genauigkeit ausgeführt, so müssen Mantissenstellen von `q` mit Nullen oder Einsen aufgefüllt werden.

2. Man nehme für alle Zwischenrechnungen das Maximum der Genauigkeiten der Argumente.

   Dies führt zu unnötigem Verlust von im Prinzip bekannter Information. Beispiel:

   Seien `x,y` Variablen einfacher und `dx` eine Variable doppelter Genauigkeit. Dann würde die Subtraktion in

   ```
   dx := x - y
   ```

   mit einfacher Genauigkeit erfolgen und `dx` könnten im Prinzip bekannte Mantissenstellen nicht zugewiesen werden.

Zur Lösung des Problems werden in den meisten Compilern zwei Durchläufe durch den Ausdrucksbaum benutzt. Die Aktionen sind dabei die folgenden:

1. Durchlauf von den Blättern zur Wurzel des Ausdrucksbaums. Für jede arithmetische Operation wird das Maximum der Genauigkeiten der Argumente gebildet. Die Genauigkeit bei einer Zuweisung ergibt sich aus der Genauigkeit der Zielvariablen. Für die Genauigkeit von Vergleichen empfiehlt sich in der Regel das Minimum der Genauigkeit der Argumente.
2. Aufgrund von Durchlauf 1 kann der Ausdrucksbaum inkonsistent sein. Im zweiten Durchlauf von der Wurzel zu den Blättern wird die Konsistenz hergestellt. Die Genauigkeit einer Operation wird reduziert, wenn ihr Ergebnis nicht in der bislang vorgesehenen Genauigkeit benötigt wird. Die Genauigkeit einer Operation wird erhöht, wenn ihr Ergebnis in größerer Genauigkeit als bislang vorgesehen benötigt wird.

Beispiel:

Abb. 3.8 zeigt, wie die Genauigkeit der Subtraktion an die der Zielvariablen angepaßt wird. `s` und `d` stehen dabei für die einfache und die doppelte Genauigkeit. Es werden zwei Konvertierungen von einfacher in doppelte Genauigkeit benötigt.

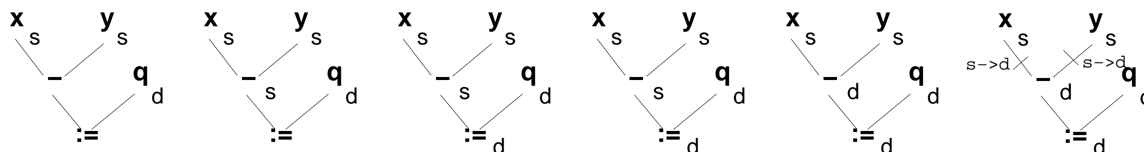

Abbildung 3.8: Erhöhung der Genauigkeit der Subtraktion

Folgende Probleme verbleiben auch bei dem angegebenen Verfahren:

- Die Genauigkeit ist nur im Kontext zu ermitteln. Ein Debugger, dem man die rechte Seite einer Anweisung übergibt, berechnet möglicherweise einen anderen Wert als der Compiler, der die Genauigkeit der Zielvariablen kennt.

- Der mögliche Fehler eines Ergebnisses ist unbekannt. Als Alternative wird von Kulisch (Univ. Karlsruhe) die Verwendung der **Intervallarithmetik** propagiert. Bei der Intervallarithmetik werden für alle Berechnungen die Intervallgrenzen der möglichen Werte betrachtet. Hierfür ist spezielle Hard- und Software entwickelt worden.

Unzulässige Optimierungen

Aufgrund der Ungenauigkeiten der Gleitkommaarithmetik sind viele zunächst korrekt erscheinende Compiler-„Optimierungen" falsch. Dazu einige Beispiele:

Ausdruck	unzulässige Optim.	Problem
$x/10.0$	$0.1 * x$	0.1 $\neg$ exakt darstellbar
$x * y - x * z$	$x * (y - z)$	falls $y \approx z$
$x + (y + z)$	$(x + y) + z$	Rundungsfehler
konstanter Ausdruck	result. Konstante	Flags werden $\neg$ gesetzt
gemeinsamer Ausdruck	Ref.auf 1.Berechn.	Rundungsmodus geändert ?
$eps := 1$; WHILE $eps > 0$ DO BEGIN $h := eps$; $eps := eps * 0.5$; END	$eps := 1$; WHILE $(eps + 1) > 1$ DO BEGIN $h := eps$; $eps := eps * 0.5$; END	

Zunächst scheint es zulässig zu sein, von beiden Argumenten des Vergleichs in der WHILE-Schleife jeweils 1 abzuziehen. Dennoch brechen die beiden Versionen der WHILE-Schleife an verschiedenen Stellen ab. In der linken Version wird *eps* bei 1 beginnend solange halbiert, bis es auf 0 abgerundet wird. Die gestrichelte Linie in Abb. 3.9 zeigt den Verlauf der Werte des Vergleichsoperanden *eps*. h speichert in diesem Fall den letzten von 0 verschiedenen Wert, 1/16. In der rechten Version wird *eps* solange halbiert, bis seine Addition zu 1 kein von 1 unterscheidbares Ergebnis liefert. Im Falle der Gleitkomma-Genauigkeit der Abb. 3.9 erhält man für *eps* die Folge 1, 1/2, 1/4, 1/8. Die durchgezogene Linie der Abbildung zeigt die Folge der Werte für den linken Vergleichsoperanden, $eps + 1$. Die Addition von 1/8 zu 1 verändert die 1 nicht mehr und h speichert den vorletzten Wert von *eps*, 1/4.

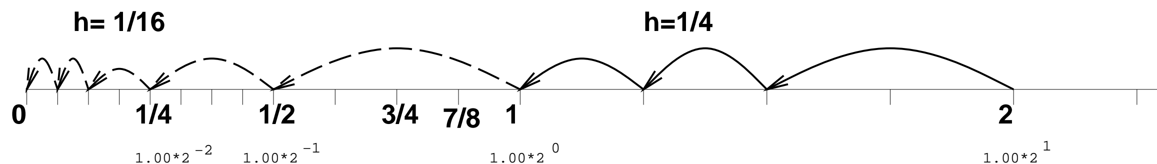

Abbildung 3.9: Werte der Vergleichsoperanden

Lediglich einige sehr einfache Optimierungen sind zulässig:

Ausdruck	optimierte Fassung
$x + y$	$y + x$
$2 * x$	$x + x$
$1 * x$	x
$x/2.0$	$x * 0.5$

3.2 Dynamisches Scheduling

9. Vorles.

3.2.1 Einführung

Eine der Techniken zur Realisierung von Parallelarbeit ist das dynamische Scheduling. Das dynamische Scheduling überwindet eine Einschränkung der bisherigen Techniken der Beschleunigung, nämlich die Tatsache,

dass Befehle in der Reihenfolge gestartet werden, in der sie im Programmcode enthalten sind. Im folgenden werden wir Möglichkeiten betrachten, die Reihenfolge der Befehlsbearbeitung erst zur Laufzeit festzulegen. Techniken dazu bezeichnet man als dynamisches Scheduling (engl. *dynamic scheduling*). Vom dynamischen Scheduling zu unterscheiden ist die statische Festlegung der Befehlsreihenfolge zur Übersetzungszeit. Statisches Scheduling hatten wir als eine Möglichkeit kennengelernt, möglichst wenige Zyklen zu vergeuden, z.B. indem wir *branch delay slots* mit nützlichen Operationen füllen.

Der Aufwand für dynamisches Scheduling ist erheblich höher, bietet aber auch einige Vorteile.

Zunächst ist statisches Scheduling immer auf ein bestimmtes Prozessormodell beschränkt. Dies macht die Codegüte modellabhängig. Vom Compilerstandpunkt aus gesehen ist es besser, guten Code für eine ganze Prozessorfamilie generieren zu können, ohne sich um die modellabhängigen Eigenschaften von Fließbändern zu kümmern. Dynamisches Scheduling kann mit statischem Scheduling kombiniert werden. So kann der Compiler bereits versuchen, datenabhängige Befehle möglichst weit auseinander zu ziehen. Verbleibende modellabhängige Ressourceabhängigkeiten könnten dann durch dynamisches Scheduling aufgelöst werden.

Als erstes Beispiel betrachte man die folgende Sequenz doppelt genauer MIPS-Gleitkommabefehle, die sich auf die speziellen Gleitkomma-Register $Fx beziehen:

```
div.d $F0,$F2,$F4   ; F0  := F2/F4
add.d $F10,$F0,$F8  ; F10 := F0+F8
sub.d $F12,$F8,$F14
```

Der `sub.d`-Befehl kann bei *in order execution* nicht sogleich ausgeführt werden. Für *out of order execution* ist zu beachten, dass der `add.d`-Befehl vom `div.d`-Befehl abhängt, der seinerseits das Fließband belegt. Andererseits ist der `sub.d`-Befehl aber von den übrigen der angegebenen Befehle nicht abhängig und er könnte vor dem `add.d`-Befehl gestartet werden. Um eine derartige *out of order execution* zu ermöglichen, muss in der Befehlsdekodierung zwischen der Erkennung von Datenabhängigkeiten und der Erkennung von Ressourcenabhängigkeiten getrennt werden.

In einem ersten Schritt wird der Befehl dekodiert und es wird geprüft, ob Ressourcenabhängigkeiten vorliegen. Falls solche nicht vorliegen, wird der betreffende Befehl in einer Befehlsqueue eingetragen. Für Befehle in der Befehlsqueue wird überprüft, ob ihre Datenabhängkeiten mit den vorhandenen Hardware-Maßnahmen berücksichtigt werden können. Falls ja, so wird ihre Bearbeitung gestartet.

Eine Möglichkeit der Berücksichtigung von Datenabhängigkeiten besteht im *score-boarding* (score board = Trefferbrett). Scoreboarding wurde zuerst im Hochleistungsrechner CDC 6600 benutzt. Score-boarding muss in der Lage sein, auch Antidatenabhängigkeiten und Ausgabeabhängigkeiten zu berücksichtigen, denn beide Formen der Abhängigkeit kommen bei *out of order execution* zum Tragen, auch wenn sie bei *in order execution* noch keine Probleme bereiteten. Man betrachte dazu die folgende modifizierte Befehlssequenz:

```
div.d $F0,$F2,$F4     ; F0  := F2/F4
add.d $F10,$F0,$F8    ; F10 := F0+F8
sub.d $F8,$F8,$F14    ; F8  := F8-F14
```

Falls jetzt der `sub.d`-Befehl vor dem `add.d`-Befehl ausgeführt wird, so liest der `add.d`-Befehl aufgrund der Antidatenabhängigkeit zwischen beiden Befehlen den falschen Wert für `$F8`. Da in der bislang besprochenen Architektur stets in der letzten Fließbandstufe abgespeichert wurde und da sich Befehle nicht überholen konnten, traten derartige Probleme bislang nicht auf.

Auch die Ausgabeabhängigkeit zwischen Befehlen kann sich jetzt auswirken. Würde man im `sub.d`-Befehl das Zielregister `$F10` verwenden, könnte bei *out of order execution* in `$F10` der falsche Wert verbleiben.

3.2.2 *Scoreboarding*

Die Abhängigkeiten werden im Scoreboard wie folgt berücksichtigt:

Jeder Befehl, der von der *instruction fetch*-Einheit kommt, durchläuft das Scoreboard. Dieses führt über die Abhängigkeiten aller ihm bekannten Befehle Buch. Wenn es feststellt, dass für einen Befehl alle Ope-

randen bekannt sind und wenn auch eine Ausführungseinheit frei ist, weist es den Befehl der Ausführungseinheit zu. Wenn es Datenabhängigkeiten zwischen noch nicht abgeschlossenen und von der fetch-Einheit empfangenen Befehlen gibt, werden die empfangenen Befehle zunächst einmal zwischengespeichert. Alle Ausführungseinheiten müssen abgeschlossene Berechnungen dem Scoreboard melden. Das Scoreboard erteilt der Ausführungseinheit die Berechtigung zum Abspeichern des Ergebnisses, sofern die Speichereinheit frei ist. Danach prüft es, ob für wartende Befehle nun alle Operanden bekannt sind und startet diese, sofern dies der Fall ist. *Scoreboarding* überwindet Grenzen der Reorganisation des Codes, die sich ansonsten durch Datenabhängigkeiten ergeben würden. Trotz der Datenabhängigkeit zwischen obigem `div.d` und `add.d`-Befehlen kann die Ausführung des `sub.d`-Befehls vorgezogen werden und schon starten, während sich der langsame `div.d`-Befehl noch in der Bearbeitung befindet. Nicht überwinden kann *scoreboarding* Grenzen der Reorganisation aufgrund der Ausgabe- und Antidatenabhängigkeiten.

3.2.3 Verfahren von Tomasulo

Diese Beschränkung wird mit dem Algorithmus von Tomasulo aufgehoben, der für die IBM 360/91 entwickelt wurde. Ziel war es, trotz der geringen Anzahl an Gleitkommaregistern und der langsamen Gleitkommaeinheiten möglichst viel Parallelarbeit zu ermöglichen.

Für den Algorithmus von Tomasulo ist der Begriff der *reservation stations* zentral. Jede funktionelle Einheit besitzt derartige *reservation stations* als eine Art Eingabewarteschlange.

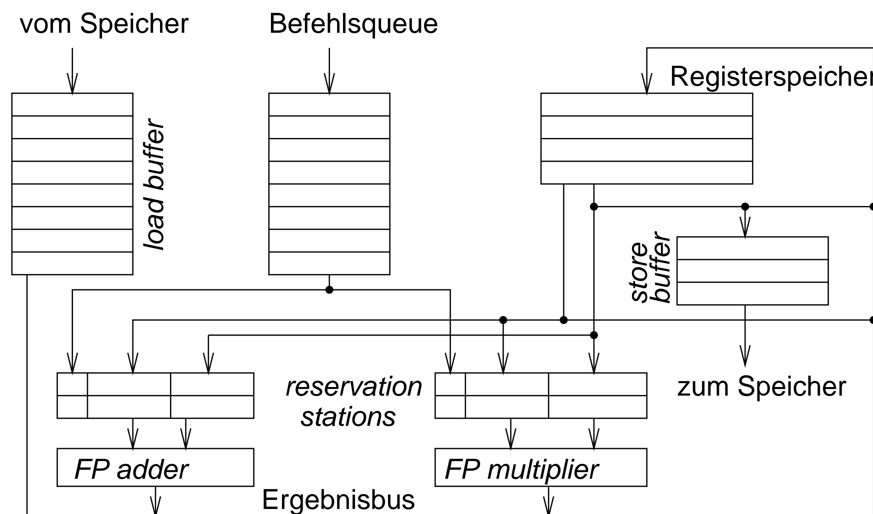

Abbildung 3.10: Architektur für Tomasulo's Algorithmus

Reservation stations enthalten auszuführende Operationen und -soweit bereits bekannt- die zugehörigen Operanden. Befehle werden gestartet, indem sie einem Befehlspuffer entnommen und einer *reservation station* zugewiesen werden. Sind alle Operanden bekannt und ist die zugeordnete funktionelle Einheit frei, so kann die Bearbeitung in der funktionellen Einheit beginnen. Am Ende der Bearbeitung wird das Ergebnis über einen gemeinsamen Bus von allen Einheiten übernommen, die das Ergebnis benötigen. Zu den Einheiten zählen insbesondere alle *reservation stations*, die auf das Ergebnis warten. Das Verteilen der Daten kann erfolgen, solbald die Daten bekannt sind. Es muss nicht erst auf die *write back*-Phase gewartet werden. Bei datenabhängigen Operationen ist es nicht notwendig, die Daten in einen Registerspeicher zu übernehmen und aus diesem wieder zu lesen. Auch das *forwarding* wird durch den Algorithmus von Tomasulo mit erledigt.

Jeder Operandenbereich einer *reservation station* enthält *tag bits*, welche den gewünschten Operanden bezeichnen. Aus den *tag bits* geht hervor, von welcher *reservation station* oder aus welchem *load buffer* der Operand kommen muss. Diese *tag bits* ersetzen die Registeradressen. Die Speicherbereiche für Operanden in den *reservation stations* realisieren damit de facto mehr Register als im Befehlssatz nach aussen sichtbar sind. Implizit wird damit ein *register renaming* durchgeführt. Das bedeutet, die Register des Befehlssatzes werden dynamisch auf eine größere Anzahl von Speicherplätzen in den *reservation stations* abgebildet. Durch diese Umbennenung können Ausgabe- und Antidatenabhängigkeiten beseitigt und so die Parallelität gegenüber dem *scoreboarding* erhöht werden.

Ein weiterer Unterschied zum *scoreboarding* ist die dezentrale Kontrolle des Algorithmus von Tomasulo.

Für weitere Details sei hier auf die Folien und das Buch von Hennessy/Patterson verwiesen.

3.3 Sprung-Vorhersage

10. Vorles.

Bei sehr komplexen Fließbändern können die Performance-Verluste durch bedingte Sprünge erheblich sein. Wenn vier oder fünf Befehle pro Zyklus gestartet werden können, ist bereits ein einzelner durch Sprünge verlorener Zyklus ein erheblicher Verlust. Man versucht daher, die Richtung von Sprüngen möglichst gut vorherzusehen und in dieser Richtung weiterzuarbeiten, bevor die Sprungbedingung tatsächlich bekannt ist.

Zu diesem Zweck realisiert man sogannte *branch prediction buffer*. Derartige Puffer werden mit einigen der unteren Adressbits (den sog. Index-Bits) des bedingten Sprungs indiziert. Ob die so angesprochene Zelle des Puffers tatsächlich zu dem gerade betrachteten Sprungbefehl gehört, ist nicht sicher, denn es könnte mehrere Sprünge geben, die sich nur in den oberen Adressbits (den Tag-Bits) unterscheiden. Um sicher zu sein, dass der Eintrag zum aktuellen Programmzähler-Stand gehört, muss man auch die Tag-Bits abspeichern und die aktuellen Tags mit den gespeicherten vergleichen (siehe Abb. 3.11). Solange man aber **im Mittel** fast immer den Eintrag zum aktuellen Sprungbefehl erhält, kann man auf diesen Tag-Vergleich evtl. auch verzichten.

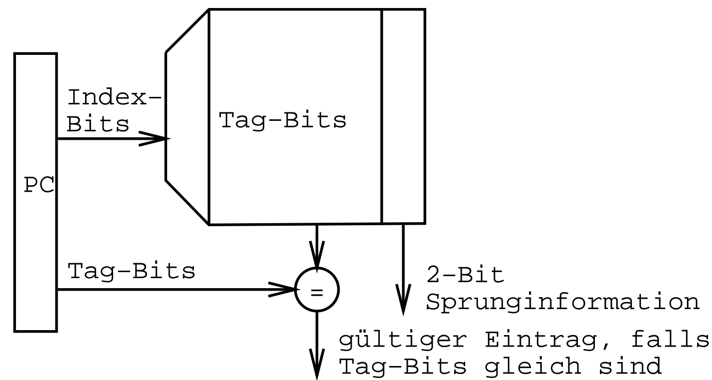

Abbildung 3.11: *Branch prediction buffer*

Der Puffer enthält in der so angesprochenen Zelle mindestens zwei Bits, welche Auskunft geben, in welche Richtung der Sprung bei seiner letzten Ausführung ging. Je eine Codierung ("00" und "11" in Abb. 3.12) besagt, dass der Sprung mindestens zweimal in dieselbe Richtung verzweigte. Die beiden übrigen Codierungen besagen, dass es eine Ausnahme von der bisherigen Sprungrichtung gab. Es wird angenommen, dass ein Sprung in eine Richtung verzweigt, solange nicht zweimal nacheinander in die andere Richtung verzweigt wurde.

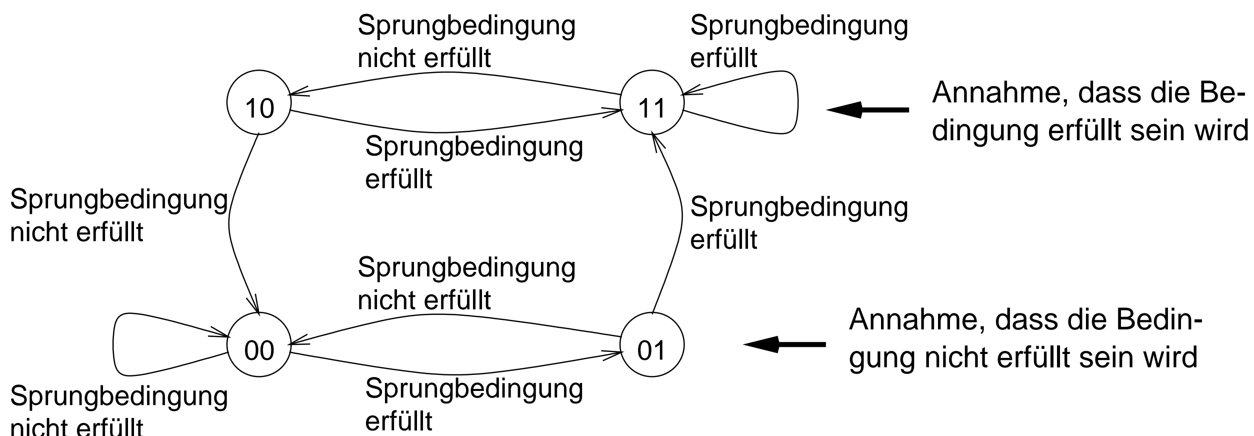

Abbildung 3.12: 2-Bit Sprungvorhersage

Die 2-Bit Sprungvorhersage bewirkt im Falle der MIPS-Architektur keine Beschleunigung. Ein *branch delay*

slot bleibt auch bei einer 2-Bit Sprungvorhersage erhalten.

Eine Beseitigung des *branch delay slots* gelingt mit der Einführung eines *branch target buffers*, der auch die Adresse des Sprungziels enthält. So braucht bei komplexen Adressberechnungen nicht erst das Ende der Adressberechnung abgewartet zu werden.

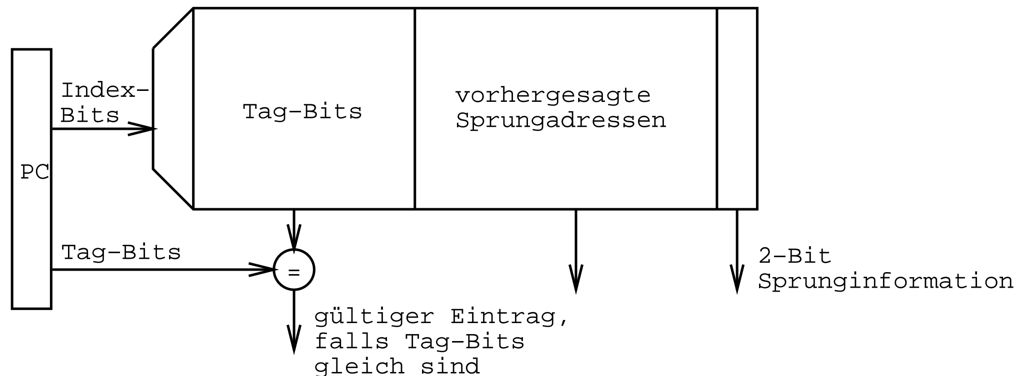

Abbildung 3.13: *Branch target buffer*

Sobald ein Eintrag im *branch target buffer* gefunden wird, werden die weiteren Befehle ab der dort angegebenen Adresse geholt. Im gleichen Puffer kann man auch die 2-Bit Sprungvorhersage-Information speichern.

Man kann *branch target buffer* und *branch prediction buffer* auch getrennt realisieren, denn beim *branch target buffer* muss zur Vermeidung eines Leistungsverlusts durch falsche Vorhersagen auf jeden Fall ein Tag-Vergleich durchgeführt werden. Die entsprechenden Einträge großer Wortbreite werden aber nur für erfüllte Bedingungen benötigt, während der *branch prediction buffer* für alle bedingten Sprünge benötigt wird. Aus diesem Grund werden *branch prediction buffer* und *branch target buffer* im PowerPC getrennt realisiert.

Durch die Sprung-Vorhersage ist es möglich, Befehle so schnell zu holen, dass das Fließband stets mit anderen Befehlen gefüllt bleibt, dass also die Sprünge selbst praktisch keine Zeit kosten.

Sofern man in einer der Verzweigungsrichtungen bereits Befehle holt und sie verarbeitet, bevor überhaupt der Sprungbefehl komplett abgearbeitet wurde, spricht man von *spekulativer Ausführung*. Bei spekulativer Ausführung muss man natürlich stets in der Lage sein, die Wirkung der spekulativ ausgeführten Befehle zu annulieren, falls man in der falschen Richtung vorgearbeitet hat.

Für weitere Details sei hier auf die Folien und das Buch von Hennessy/Patterson verwiesen.

3.4 *Multiple Instruction Issue*

11. Vorles.

Um die Rechenleistung weiter zu steigern, ist man vor einigen Jahren dazu übergegangen, auch Architekturen mit CPI-Werten <1 zu entwickeln. Zu diesem Zweck besitzen Maschinen mehrere funktionelle Einheiten wie z.B. Integer- und Gleitkomma-Rechenwerke. Da mehrere dieser Einheiten gleichzeitig arbeiten können, kann bei geigneter Befehlsdekodierung mehr als ein Maschinenbefehl pro Zyklus ausgeführt werden.

Zur Realisierung solcher Maschinen muss die *instruction fetch*-Stufe pro Takt mehrere Befehle zur Ausführung bereitstellen. Man spricht in diesem Zusammenhang auch von *multiple instruction issue*.

Es gibt im Wesentlichen zwei Methoden, mehrere Befehle pro Takt bereitzustellen:

- **VLIW-Architekturen**

  Bei VLIW-Prozessoren ist allein der Compiler dafür verantwortlich, mehrere Befehle zu finden, die während eines Taktes bereitgestellt werden können.

- **superskalare Architekturen**

Superskalare Architekturen enthalten Hardwaremechanismen, welche zur Laufzeit pro Takt mehrere Befehle bereitstellen können. Bei ausreichend vielen funktionellen Einheiten benötigt ein Befehl im Mittel dann weniger als einen Takt zur Ausführung.

Def.: Architekturen mit CPI-Werten < 1 heißen **superskalare Architekturen**.

Beispielsweise stellen der Prozessor MIPS R8000 pro Zyklus 4, der Pentium Pro 5 Befehle pro Zyklus zur Verfügung. Es kann allerdings Einschränkungen hinsichtlich der gleichzeitigen Ausführung aller bereitgestellten Befehle geben.

Eine ganze Reihe von leistungsverbessernden Maßnahmen ergibt sich, wenn man den Compiler mit in die Überlegungen einbezieht. Das vielleicht bekannteste Beispiel ist das Abrollen von Schleifen (engl. *loop unrolling*). Zur Einführung betrachten wir das folgende kleine C-Programm.

```
for (i=1; i<=20; i++)
   x[i] = x[i] + s;
```

welches einen skalaren Wert zu Array-Elementen hinzu addiert. Eine geschickte Übersetzung in MIPS-Assemblercode könnte wie folgt aussehen:

```
      Li   $5,80          -- R5:=20*4 (4 Byte / Element; load lower immediate)
      Lw   $2,s           -- R2:=s;
Loop: Lw   $3,&x($5)      -- $3 = Array-Element; &x: Adr. erstes Elem.
      add  $4,$3,$2       -- addiere in R2 befindliches s
      sub  &x($5),$4      -- speichere Ergebnis
      subi $5,$5,4        -- dekrementiere pointer um 4 Bytes
      bne  $5,$0,Loop     -- springe, wenn $5<>$0 (=0)
```

Zusätzlich zu den 5 nützlichen Schleifen-Zyklen würde ein solches Programm möglicherweise noch einige Zyklen mit dem Warten verbringen. Eine drastische Beschleunigung dieser Schleife ist möglich, sofern der Programmcode nicht sehr kompakt sein muss und sofern die Schleifengrenzen fest sind. In diesem Fall kann man nämlich den Schleifenrumpf kopieren. Man spricht in diesem Zusammenhang vom **Abrollen der Schleife**. Bei einem zweimaligen Abrollen kommt man zu der folgenden Version:

```
for (i=1; i<=20; i++)
  { x[i] = x[i] + s; i++;
    x[i] = x[i] + s; }
```

Die Ausführung der beiden Kopien des Rumpfes kann sich zeitlich überlappen und so das Fließband besser füllen. Der Vorteil wird noch deutlicher, wenn -wie beim Pentium- zwei oder mehr Fließbänder vorhanden sind.

Im Extremfall, dem sog. **vollständigen Abrollen**, wird die Schleife komplett aufgelöst:

```
x[1] = x[1] + s;
x[2] = x[2] + s;
x[3] = x[3] + s;
x[4] = x[4] + s;
x[5] = x[5] + s;
...
```

In dieser Lösung ist kein Overhead für die Verwaltung der Schleife erforderlich.

Interne Struktur von Pentium-Prozessoren

Die vorgestellte Fließbandverarbeitung kann man mit überschaubarer Logik (wenn überhaupt) nur bei einfachen Befehlen und v.a. bei Befehlen fester Länge realisieren. Dies ist eine wesentliche Motivation für die Benutzung von RISC-Befehlssätzen. Aus diesem Grund konnte man Ende der 80er-Jahre eigentlich erwarten, dass CISC-Befehlssätze aussterben würden. Dennoch hat sich der x86er-Befehlssatz erstaunlich gut gehalten, besser als die meisten RISC-Befehlssätze. Der Grund liegt in dem Zwang zur Befehlssatz-Kompatibilität für PC-Applikationen und der Tatsache, dass man bei der Realisierung

neuer Pentium-Architekturen die Grundideen der RISC-Architekturen aufgegriffen hat. Die Architekturen sind zwar extern weiter CISC-Architekturen. Intern werden jedoch x86-Befehle in RISC-Befehle zerlegt und diese werden in mehreren parallelen Fließbändern abgearbeitet (siehe Abb. 3.14). Dies ist zwar keine sehr effiziente Verwendung des Siliziums, aber darauf kommt es bei PC-Applikationen nicht an.

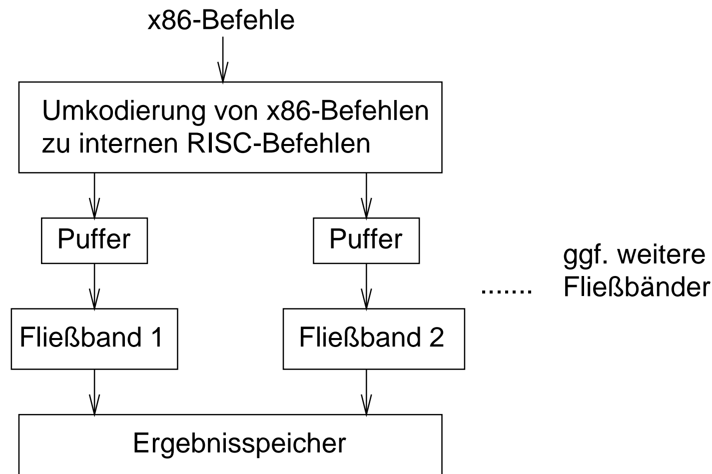

Abbildung 3.14: Interne Struktur von Pentium-Prozessoren

Für weitere Details sei hier auf die Folien und das Buch von Hennessy/Patterson verwiesen.

Kapitel 4

Speicherarchitektur

4.1 Speicherhardware

12. Vorlesung

Über den grundsätzlichen Aufbau von Speicherhierarchien haben wir bereits in dem Kurs "Rechnerstrukturen" gesprochen. Im laufenden Kurs bleibt uns nur übrig, einige Themen zu vertiefen. Dies betrifft insbesondere die Organisation von Caches.

Zur Organisation von Caches sei zunächst wieder auf die Folien verwiesen.

Ergänzen können wir hier das Thema "Austauschverfahren".

4.2 Austauschverfahren

Innerhalb der Speicherhierarchie ist es auf jeder Stufe erforderlich, Informationen in der schnelleren Stufe der Hierarchie auszutauschen, um häufig benötigter Information Platz zu machen. Es gibt sehr viele Verfahren zur Wahl der Information im schnellen Speicher, die verdrängt wird. Im folgenden werden drei anhand der Hierarchiestufe Cache/Hauptspeicher besprochen[1]:

1. **Random-**, **bzw. Zufallsverfahren**
   - Zufällige Auswahl der auszulagernden Cache-Zeile
   - lokal oder global, fest oder variabel
   - keine zusätzliche Hardware erforderlich
   - sinnvoll nur dann, wenn ohnehin keine Regularität im Zugriffsverhalten erkennbar ist; sonst ungeeignet

2. ***NRU, Not Recently Used***

   Dieses Verfahren basiert auf der Analyse von zwei Einträgen, die üblicherweise existieren: dem *used bit* und dem *modified bit*. Aufgrund der Werte dieser beiden Bits kann man zwischen vier Klassen von Seiten unterscheiden:

   (a) Nicht benutzte, nicht modifizierte Zeilen
   (b) Nicht benutzte, modifizierte Zeilen
   (c) benutzte, nicht modifizierte Zeilen
   (d) benutzte, modifizierte Zeilen

[1] Beschreibungen weiterer Verfahren findet man bei Baer [Bae80] sowie in vielen Büchern über Betriebssysteme (siehe z.B. Tanenbaum [Tan76]).

Die Klasse 2 ist deshalb möglich, weil die Used-Bits periodisch mit Hilfe eines Timers zurückgesetzt werden und daher Zeilen der Klasse 4 zu Zeilen der Klasse 2 werden können. Das Modified-Bit kann nicht periodisch zurückgesetzt werden, weil modifizierte Zeilen sonst nicht zurückgeschrieben werden würden.

Der NRU-Algorithmus überschreibt jetzt eine zufällig ausgewählte Zeile der niedrigsten, nicht-leeren Klasse.

3. ***LRU = least recently used***

   Es wird stets die Kachel überschrieben, die am längsten nicht benutzt wurde.

   Lösungen zur Realisierung:

   (a) verkettete Liste von Kachelnummern (Bei Baer als *stack* bezeichnet)

   Es wird ständig eine verkettete Liste von Kachelnummern gehalten, die nach dem Alter des letzten Zugriffs sortiert ist.

   Operationen auf dieser Datenstruktur:

   - Zugriff auf eine Kachel i:
     i wird als Element am Anfang der Liste eingefügt und, falls in der Liste sonst noch vorhanden, dort entfernt.
   - Seitenfehler:
     Die Kachel, die zu dem letzten Element der Liste gehört, wird dort entfernt

   Beispiel:

Zugriff auf Seite	1	2	3	4	5	3	5	4	6
und damit auf Kachel	1	2	3	4	1	3	1	4	2
verkettete Liste von	1	2	3	4	1	3	1	4	2
Kacheln (Anzahl der		1	2	3	4	1	3	1	4
Kacheln = 4)			1	2	3	4	4	3	1
				1	2	2	2	2	3

   Einfügen am Anfang und Entfernen am Ende einer Liste lässt sich relativ einfach in $O(1)$ Zeiteinheiten realisieren, kann also durch die Hardware relativ leicht während der Zugriffe auf Seiten durchgeführt werden. Das Entfernen eines Elements ist nur dann hinreichend schnell zu realisieren, wenn es durch eine geeignete Datenstruktur unterstützt wird. Möglich ist z.B. das Führen eines *bucket arrays*. Zu jeder Kachel i könnte dieses Array einen Verweis auf den Platz der Kachel i innerhalb der Liste enthalten. Bei doppelt-verketteter Liste kann dann auch das Entfernen des Listenelements in $O(1)$ erfolgen.

   (b) Dreiecksmatrix[2]

   Für alle Paare (i, j) von Kacheln wird in einer Matrix die Relation „Der letzte Zugriff auf i ist älter als der letzte Zugriff auf j" gespeichert, also z.B.:

$$f[i,j] = \begin{cases} 1, & \text{falls der letzte Zugriff auf } i \text{ älter ist als der auf } j \\ 0, & \text{sonst} \end{cases}$$

   Wegen der Antisymmetrie braucht lediglich eine Δ-Matrix realisiert zu werden.

   Operationen auf dieser Datenstruktur:

   - Zugriff auf eine Kachel i: i wird jüngste Kachel, d.h. es ist zu setzen:

$$\forall j \neq i : f[i,j] := 0$$
$$\forall j \neq i : f[j,i] := 1$$

   - Seitenfehler:
     gesucht ist die älteste Kachel, d.h. die Kachel i, für die gilt:

$$\forall j \neq i : f[i,j] = 1 \wedge f[j,i] = 0$$

   Abbildungen 4.1 bis 4.3 zeigen ein Beispiel (1. Index=Spalte, 2. Index=Zeile).

   Das zeilen- bzw. spaltenweise Setzen und Suchen nach 0 bzw. 1 kann durch eine geeignet Hardware schnell ausgeführt werden. Für eine größere Zahl von Kacheln ergibt sich aber ein erheblicher Aufwand.

[2] Quelle: Liebig [Lie80]

Zugriff auf Seite 1
-> Seitenfehler
-> Überschreiben
von Kachel 1

Zugriff auf Seite 2
-> Seitenfehler
-> Überschreiben
von Kachel 2

jüngste Kachel jeweils gestrichelt

Kachel	1	2	3	4
1		1	1	1
2			1	1
3				1
Seite	6	7	8	9

	1	2	3	4
1		0	0	0
2			1	1
3				1
	1	7	8	9

	1	2	3	4
1		1	0	0
2			0	0
3				1
	1	2	8	9

Abbildung 4.1: Benutzung einer Dreicksmatrix

Zugriff auf Seite 3
-> Seitenfehler
-> Überschreiben
von Kachel 3

Zugriff auf Seite 4
-> Seitenfehler
-> Überschreiben
von Kachel 4

Zugriff auf Seite 5
-> Seitenfehler
-> Überschreiben
von Kachel 1

	1	2	3	4
1		1	1	0
2			1	0
3				0
	1	2	3	9

	1	2	3	4
1		1	1	1
2			1	1
3				1
	1	2	3	4

	1	2	3	4
1		0	0	0
2			1	1
3				1
	5	2	3	4

Abbildung 4.2: Benutzung einer Dreicksmatrix

Zugriff auf Seite 3
Kachel 3 wird
jüngste Kachel

Zugriff auf Seite 5
Kachel 1 wird
jüngste Kachel

Zugriff auf Seite 4
Kachel 4 wird
jüngste Kachel

	1	2	3	4
1		0	1	0
2			1	1
3				0
	5	2	3	4

	1	2	3	4
1		0	0	0
2			1	1
3				0
	5	2	3	4

	1	2	3	4
1		0	0	1
2			1	1
3				1
	5	2	3	4

Abbildung 4.3: Benutzung einer Dreicksmatrix

Benutzt wird LRU in der reinen Form nur für wenige Einträge im lokalen Speicher, z.B. bei Caches oder TLBs nach dem *set associative*-Prinzip, dort vor allem für den Fall einer zweielementigen Menge (Δ-Matrix entartet zu einem Bit).

Kapitel 5

Der CO_2-Fußabdruck von PCs

14. Vorles.

Siehe Folien

Literaturverzeichnis

[ABM+93] C. Albrecht, S. Bashford, P. Marwedel, A. Neumann, and W. Schenk. The design of the PRIPS microprocessor. *4th EUROCHIP-Workshop on VLSI Training*, 1993.

[ANH+93] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. An ASIP instruction set optimization algorithm with functional module sharing constraint. *Int. Conf. on Computer-Aided Design (ICCAD)*, pages 526–532, 1993.

[Bae80] J.L. Baer. *Computer Systems Architecture*. Pitmen, 1980.

[Bae94] H. Baehring. *Mikrorechnersysteme*. Springer, 1994.

[BH80a] A. Bode and W. Händler. *Rechnerarchitektur*. Springer, 1980.

[BH80b] A. Bode and W. Händler. *Rechnerarchitektur II*. Springer, 1980.

[BK95] V. Bhaskaran and K. Konstantinides. Multimedia enhancements for RISC processors. *in: Image and Video Compression Standards, Kluwer Acad. Publishers*, 1995.

[BN71] Bell and Newell. *Computer Structures*. McGraw-Hill, 1971.

[Bod] A. Bode. *Risc-Architekturen*. Bibliographisches Institut, Mannheim.

[CEF+95] H. P. Cinka, J. Ebert, B. Fischer, J. Freytag, and R. Gunzenhäuser. Empfehlung des Fachbereichs 7 (Ausbildung und Beruf) der Gesellschaft für informatik zur weiterbildung für Informatiker durch die Hochschulen. *Informatik-Spektrum*, pages 106–109, 1995.

[Coy92] W. Coy. *Aufbau und Arbeitsweise von Rechenanlagen*. Vieweg, 1992.

[CS99] D. Culler and J. P. Singh. *Parallel Computer Architecture - A hardware / software approach*. Morgan Kaufman Publishers, 1999.

[Dal97] M.K. Dalheimer. *Java Virtual Machine - Sprache, Konzept, Architektur*. O'Reilly, 1997.

[Den80] J. Dennis. Data flow supercomputers. *IEEE Computer*, page 48, 1980.

[Fis93] W. Fischer. Datenkommunikation mittels ATM – Architekturen, Protokolle, betriebsmittelverwaltung. **it + ti** *(Informationstechnik und Technische Informatik)*, pages 3–11, 1993.

[Gil81] W. Giloi. *Rechnerarchitektur*. Springer, 1981.

[Hay79] J.P. Hayes. *Computer Architecture and Organization*. McGraw-Hill, 1979.

[Hof77] R. Hoffmann. *Rechenwerke und Mikroprogrammierung*. Oldenbourg, 1977.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantative Approach*. Morgan Kaufmann Publishers Inc., 2. Aufl., 1996.

[HP02] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantative Approach*. Morgan Kaufmann Publishers Inc., 3. Aufl., 2002.

[HQ89] W. Heise and P. Quattrocchi. *Informations- und Codierungstheorie*. Springer, 2. Auflage, 1989.

[HT93] W.D. Hillis and L. W. Tucker. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, pages 31–49, 1993.

[IEE88] Design Automation Standards Subcommittee of the IEEE. IEEE standard VHDL language reference manual (IEEE Std. 1076-87). *IEEE Inc., New York*, 1988.

[IEE92] Design Automation Standards Subcommittee of the IEEE. Draft standard VHDL language reference manual. *IEEE Standards Department, 1992*, 1992.

[Jes75] E. Jessen. *Architektur digitaler Rechenanlagen*. Springer, 1975.

[Joh00] R. Johnson. RNA computer clears 10-bit hurdle. *EETimes (www.eet.com), 1.2.2000*, 2000.

[JS92] D. Jungmann and H. Stange. *Einführung in die Rechnerarchitektur*. Hanser, 1992.

[Kai89a] R. Y. Kain. *Computer Architecture Vol. I*. Prentice-Hall, 1989.

[Kai89b] R. Y. Kain. *Computer Architecture Vol. II*. Prentice-Hall, 1989.

[Kog91] Peter M. Kogge. *The architecture of symbolic computing*. McGraw-Hill, 1991.

[Kuc78] D.J. Kuck. *The Structure of Computers and Computations, Vol. I*. Wiley, 1978.

[Lie80] H. Liebig. *Rechnerorganisation*. Springer, 1980.

[MSS91] Müller-Schloer and Schmitter. *RISC-Workstation-Architekturen*. Springer, 1991.

[PB61] W.W. Peterson and D.T. Brown. Cyclic codes for error correction. *Proc. of the IRE*, pages 228–235, 1961.

[Pet00] C. Peterson. Taking technology to the molecular level. *IEEE Computer, Jan. 2000*, pages 46–53, 2000.

[PWW97] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, pages 25–38, 1997.

[Sch73] H. Schecher. *Funktioneller Aufbau digitaler Rechenanlagen*. Springer, 1973.

[SK83] C. Schmittgen and W. Kluge. A system architecture for the concurrent evaluation of applicative program expressions. *10th Int. Symp. on Computer Arch*, 1983.

[Spa76] O. Spaniol. *Arithmetik in Rechenanlagen*. Teubner, 1976.

[SR00] A.M. Steane and E.G. Rieffel. Beyond bits: The future of information processing. *IEEE Computer, Jan. 2000*, pages 38–45, 2000.

[SRU99] J. Silc, B. Robic, and T. Ungerer. *Processor Architecture - From Dataflow to Superscalar and Beyond*. Springer Verlag, 1999.

[Sta94] W. Stallings. *Data and Computer Communications*. MacMillan, 1994.

[Sta95] W. Stallings. *Operating Systems*. Prentice Hall, 1995.

[Sto80] H.S. Stone. *Introduction to Computer Architecture*. 1980.

[Sun97] Sun Microsystems. picojava^{TM} i microprocessor core architectur. *http://www.sun.com/sparc/-whitepapers/wpr-0014-01/index.html*, 1997.

[Tan76] A.S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 1976.

[Tea82] P. Treleaven and et al. Data-driven and demand-driven computer architectures. *acm computing surveys*, pages 93–144, 1982.

[Tre84] P. Treleaven. Future computers: Logic,...,data flow, control flow. *IEEE Computer*, pages 47–55, 1984.

[Veg84] S. Vegdahl. A survey of proposed architectures for the execution of functional languages. *EEEE Trans. Comp.*, page 1050, 1984.

Index

ALU, 29
Architektur
  Speicher-, 44
  superskalare, 41
Arithmetik, 24
  Intervall-, 37
arithmetisches Schieben, 34
ASIP, 17
Attribut, 33
Austauschverfahren, 44

Barrelschifter, 25
Befehl
  multiply/accumulate-, 9
Befehlsatz
  Multimedia-, 16
Befehlssatz
  ASIP-, 17
  EPIC, 13
  MMX-, 16
  VLIW, 13
bit_vector, 6
Bitvektor, 6, 24
boolean, 6
Booth-Algorithmus, 33
branch delay slots, 38
branch prediction buffer, 40
branch target buffer, 41
Burstfehler, 12

Code
  systematischer, 10
CRC-Zeichen, 13

data-driven, 20
Datentypen
  abstrakte, 24
  elementare, 8
demand driven, 20
DNA/RNA-Rechner, 23
DSP, 8, 28

ECC, 13

GCR, 13
Größenvergleich, 32

IA-64, 15
int, 33
integer, 6, 31

JAM, 18
Java, 18

Kontrollfluß, 20

loop unrolling, 42
LRU, 45, 46

Maschine
  abstrakte, 18
  Keller-, 19
Mikroprogramm, 19
MMX, 16
multiple instruction issue, 41
Multiplikation, 33
  nach Schulmethode, 29

nat, 30
natural, 6

out of order execution, 38
overflow, 32

PicoJava, 19
Polynom
  Code-, 10
  Fehler-, 12
  Generator-, 10
  Nachrichten-, 10
positive, 6
Programmiermodell, 8
PROLOG, 21

Quantenrechner, 23

real, 35
Reduktionsmaschinen, 20

Sättigungsarithmetik, 16
saturating arithmetic, 8
Schiebeoperationen, 24, 25
score-boarding, 38
Speicher, 44
Speicherorganisation, 8
Sprung-Vorhersage, 40
string reduction, 20
Subtraktion, 28, 32
Systeme
  eingebettete, 8, 18

TLB, 46
TMS 320C62xx, 13

Tomasulo-Algorithmus, 39
TriMedia, 14
Typisierung, 5

VHDL, 5, 30, 33
VIS, 17

WAM, 21

Zahlen
    Ganze, 31
    Gleitkomma-, 35
    natürliche, 26
Zweierkomplement, 31