C. Rechnerarchitektur

Autor: Markus Möller

- Bezug: Vorlesung bei Herrn Rammig und die dort angegebene Referenzliteratur.
- Dieser Extrakt entstand als Vorbereitung auf meine Diplomprüfung (Teil: Vertiefungsgebiet). Er faßt einige Themen einfach zusammen und mag etwas unorthodox erscheinen. Er ist als Vorlesungsergänzung zur Verständnisverbesserung zu sehen.
- Erstellt auf Apple Macintosh.

1. Einführung

1.1 Sichten und Abstraktionsebenen zur Modellierung von Rechnerhardware

TABELLE 2: Abstraktionsebenen

Nr.	Name	Modellierungskonzept	Zeitmodell	beobachtbare Werte	Struktur/ Geometrie
6	System	kooperierende semi-autonome Komponenten	kausal	beliebige Werte	Komponenten, Verschaltung
5	Algorithmus	nebenläufige Algorithmen	kausal	Bitkette mit Interpretation	
4	Register Transfer	Guarded Commands	Taktung	Bitketten	RT-Struktur/ Floorplanning
3	Gatter	Boolesche Gleichungen	Kontinuum	Bits	Netzliste/ Floorplanning
2	Switch	diskrete Gleichungen	Kontinuum	Wert, Stärke	Netzliste/ Stickdiagramm
1	elektrisch	Differentialgleichungen	Kontinuum	I,R,U,C,…	Netzliste/ Layout

1.2 Historischer Überblick

-1. Generation (1600-1900)

Mechanisch und Komplementdarstellung.

0. Generation (1830-1945)

Programmsteuerung durch Programmkarten. Gleitkomma.

1. Generation (1946-1955)

Der Urvater der heutigen Universalrechner: von Neumann's IAS. Die CPU besteht aus einer Arithmetic-Logic-Unit (ALU) und der Program-Control-Unit (PCU). Die PCU enthält den Programmzähler und ein Instruktionspufferregister. Ferner das Instruktionsregister und das Adreßregister, welches mit dem Hauptspeicher verbunden ist. Das Instruktionsregister steuert als Eingabe über die Schaltkreise den Rechner.

In der ALU ist das Datenregister die Verbindung mit der PCU, allen I/O-Einheiten und dem Hauptspeicher. Nur hiermit wird der Speicher gelesen. Das Adreßregister gibt eine Speicherstelle an und das Datenregister empfängt den Inhalt. Hier finden in den arithmetisch-logischen Schaltkreisen Rechenoperationen statt. Als weitere Register existiert der AC (Akkumulator) und das MQ (Multiplikationsregister).

Der Interpretationszyklus der IAS besteht aus der Holphase und der Ausführungsphase. Wenn die nächste Instruktion schon im Puffer ist, dann werden das Adreßregister und das Instruktionsregister damit geladen. Andernfalls wird der Befehl vom Hauptspeicher geladen: PC in Adreßregister schreiben. Datenregister mit gewünschtem Befehl laden und dann IR (Instructionregister) und AR (Adreßregister) mit DR (Dataregister) laden. Falls die linke Instruktion benötigt wurde, dann wird die rechte im IBR (Instruction-Buffer-Register) gehalten.

In der Ausführungsphase wird dann der Befehl bearbeitet. Z.B. AC mit M(AR) geladen oder Sprung mit PC mit AR laden etc.

Mit der IAS ist keine vernünftige Adreßrechnung möglich. Insbesondere Arrays lassen sich schlecht handhaben. Unterprogramme werden nicht unterstützt. Fließkommarechnung und nicht-numerische Instruktionen sind unmöglich. Kein vernünftige I/O.

2. Generation (1950-1965)

Transistoren statt Röhren. Indexregister, Gleitkomma, Kernspeicher, Magnettrommel, I/O-Prozessoren, Systemsoftware. Während ein I/O-Prozessor einen Befehl für die CPU abarbeitet, kann sich diese mit anderen Aufgaben beschäftigen. Das ist der erste Schritt zur Parallelarbeit.

Erste höhere Programmiersprachen wie Fortran, Algol und Cobol. Erste Supercomputer. Pipelining, Prefetching, Multiprocessing, Timesharing. Stapelbetrieb: Top of Stack etc.

3. Generation (1965-1975)

ICs statt Transistoren. Halbleiterspeicher statt Kernspeicher. Nebenläufigkeit, Betriebssysteme. Pipelining (Fließbandverarbeitung).

Der Auslöser für die Einführung virtueller Speicherverwaltung waren die I/O-Prozessoren. Durch sie wurde Multiprogramming möglich, wodurch für die vielen Programme der Hauptspeicher zu klein wurde. Daher trennt man den logischen vom physikalischen (Hauptspeicher) Adreßraum. Der logische kann wesentlich größer sein als der physikalische.

Multiprogramming, Multiprocessing. Einführung des Programmstatuswort PSW als erweiterter PC (Program Counter, Nummer des abzuarbeitenden Befehls). Darin sind neben dem Programmzähler als Kontext der Prozessorstatus und die Zugriffsberechtigung auf Speicher enthalten. Bei Kontextswitch ist das PSW austauschbar.

4. Generation (1975-1985)

VLSI (Very Large Scale Integration, „Packungsdichte“ auf dem Chip), Adreßraum von 2^{32} Byte = 4 Gigabyte. Mikroprozessoren.

5. Generation (1985-?)

Massive Parallelität: Message Passing Systeme, SIMD-Rechner (Single Instruction Multiple Data), Virtual-Shared-Memory, Workstation-Cluster.

2. Informationsdarstellung

2.1 Festkommadarstellungen

Wenn man verschiedene Zahldarstellungen betrachtet, dann gibt es eine Basis, die den geringsten Aufwand (Stellenanzahl * Basiszahl) hat, um N verschiedene Zahlen darzustellen. Um N verschiedene Zahlen mit der Basis b darzustellen, benötigt man $\log_b N$ Stellen. Der Aufwand dafür ist also $\log_b N \quad b$. Den geringsten Aufwand hat demnach die Basis 2,71828 für jedes N. Die beiden besten ganzzahligen Basen sind 3 und dann 2.

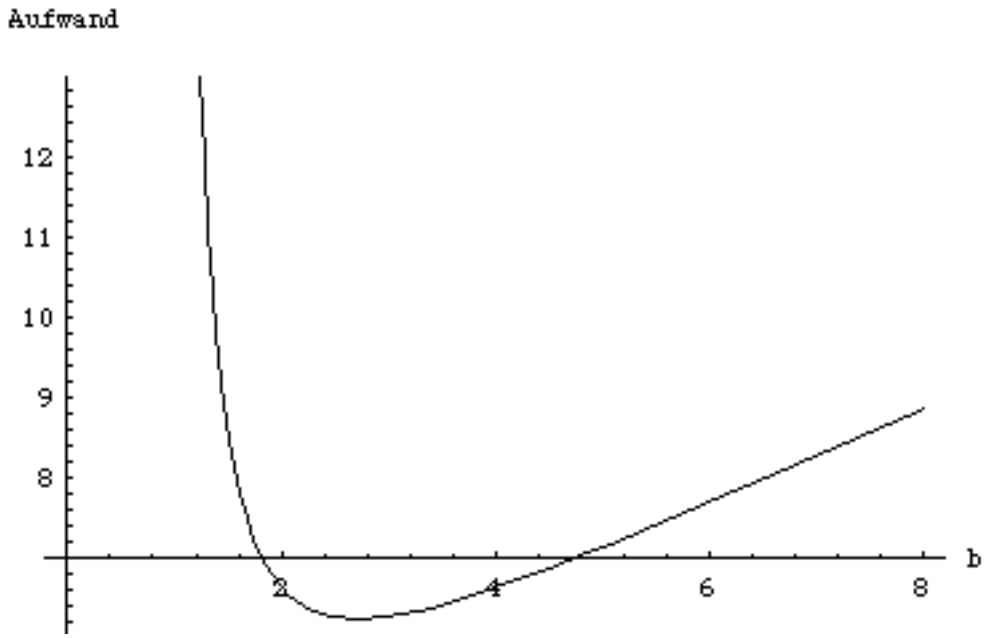

ABBILDUNG 3: Aufwand für Zahldarstellungen bzgl. verschiedener Basen

Wenn man eine Zahl auf eine andere Basis umrechnen will, dann macht man im Prinzip eine fortlaufende Division der Zahl bzw. des ganzzahligen Ergebnisses der vorigen Division durch die neue Basis. Das entspricht einer Kommaverschiebung, bei der man sich den Rest jeweils notiert. Der erste Rest ist die Einerstelle, da hier nur einmal durch die neue Basis dividiert wurde.

2.1.1 Vorzeichen/Betrag

Bei n Bits werden n-1 Bits für die Größe und 1 Bit für das Vorzeichen verwendet. Darstellbarer Bereich: $[-(2^{n-1}-1),(2^{n-1}-1)]$.

Nachteil +0 und -0. Vorteil: einfacher Vorzeichentest und * und / einfach. Verwendung bei Mantissen der Gleitpunktdarstellung.

2.1.2 Excess-Darstellung

Mit n Bit wird eine Zahl dargestellt, die um 2^{n-1} vergrößert wurde. Darstellbarer Bereich: $[(0-2^{n-1}),(2^n-1-2^{n-1})] = [-2^{n-1},2^{n-1}-1]$. Anwendung als Exponent bei Gleitpunktdarstellung.

Vorteil: Eine Null. Nachteil: asymmetrischer Wertebereich, Arithmetik schwierig.

2.1.3 Einerkomplement-Darstellung

Nichtnegative Zahlen im Bereich $[0,2^{n-1}-1]$ werden im normalen Stellensystem dargestellt. Negative Zahlen im Bereich $[-(2^{n-1}-1),0]$ werden bitweise „umgekippt“. Kein explizites Vorzeichenbit. Aber durch das „Umkippen“ der negativen Zahlen werden diese auf Bitfolgen abgebildet die alle mit 1 beginnen. Die negativen Zahlen liegen also in ihrer Darstellung „über“ den positiven. Dabei ist -0 identisch mit der „11…1“ Bitkette.

Vorteil: einfache Berechnung von -x, einfacher Vorzeichentest, einfaches +,-, symmetrischer Wertebereich.

Nachteil: Zwei Nullen.

2.1.4 Zweierkomplement-Darstellung

Berechnung des positiven Bereich wie oben. Bei negativen Zahlen auch wie oben und dann +1. Darstellbarer Bereich folglich: $[-2^{n-1},2^{n-1}-1]$.

Diese Darstellung ist heute in Rechenwerken üblich.

Vorteil: Nur eine Null, einfacher Vorzeichentest, einfaches +,-.

Nachteil: -x etwas schwieriger, asymmetrischer Wertebereich.

2.1.5 Dezimaldarstellung

Dezimalzahlen als Bytefolge. Entweder zwei Dezimalziffern pro Byte oder nur eine. Verschiedene Untermöglichkeiten.

2.1.6 Gleitkommadarstellungen

Wenn die Position des Kommas festliegt, dann ist es „normalisierte Darstellung“. Z.B. links der signifikantesten „1“. Damit lassen sich allerdings keine Zahlen darstellen, die zwischen Null und der kleinsten normalisierten liegen. Hier darf von der Normalisierung abgewichen werden und man kommt zur sog. „denormalisierten Darstellung“.

2.1.7 Nicht numerische Daten

EBCDIC in der IBM-Welt und ASCII für den Rest der Welt. Zeichenketten einfach durch Folge von Zeichen.

Bei der „tagged“ Speicherung werden unterschiedliche Datentypen inklusive ihrer Bedeutung gespeichert.

2.1.8 Befehle

2.1.8.1 Identifikation des Datenbereichs

Es gibt verschiedene **Datenbereiche**: Register, Hauptspeicher, I/O-Bereich, Spezialregister, Stack.

Der jeweilige Datenbereich kann

- *im Opcode* bestimmt werden, z.B. implizit Stack. Oder er kann
- *in der Adresse* festgelegt werden, z.B. implizit durch Anordnung (1. Operand immer Register, 2. Hauptspeicher) oder explizit (häufiger).

2.1.8.2 Spezifikation des Datentyps

Datentypen sind z.B. „integer word“ oder „floating point“.

Der Datentyp kann

- *im Op-Code* spezifiziert werden, was die häufigste Methode ist (typbedingte Befehlsunterarten) oder

- *in der Adresse*, was in keiner kommerziellen Maschine realisiert wurde oder
- *im Datum selbst*, wobei man von „tagged architecture“ spricht. Hier wird jedes Speicherwort um ein Tagfeld erweitert, das den Datentyp des Wortes spezifiziert.

2.1.8.3 Identifikation des Datenwertes

- *Durch Opcode* selten. Beispiel „increment“ mit Wert implizit 1.
- *In Adresse* als unmittelbare Adressierung (Konstanten).
- *In Daten selbst* ist der Normalfall.

Drei Elemente werden bei der Ausführung eines Befehls eingesetzt: die Operation (Op-Codes), die Adresse der Operanden und die Werte der Operanden.

2.1.8.4 Der Opcode-Teil

Eine m-Adreßmaschine nennt pro Instruktion m Adressen explizit. Man kann danach klassifizieren, ob die Instruktionslänge oder die Operandenzahl fest oder variabel sind. Feste Werte ermöglichen höhere Geschwindigkeit. Variable Instruktionslänge oder Operandenanzahl spart z.B. Speicherplatz.

Üblich ist eine feste Länge. Ein kurzer Opcode darf nicht als Präfix eines langen vorkommen. Beim 1,5-Adreßformat ist eine Adresse eine Registeradresse, die kürzer ist als die übliche Hauptspeicheradresse.

Man nimmt meist kurze Opcodes für häufige Befehle.

2.1.8.5 Adreßteil

Man unterscheidet nach

- Referenzierungstiefe:
  0 entspricht *unmittelbarer* Adressierung, der Wert ist also die Adresse selbst. 1 heißt, daß die Adresse des Operanden im Befehl steht, *direkte Adressierung.* 2 bedeutet *indirekte Adressierung*, also Adresse der Adresse steht im Befehl.
- Speicherklasse:
  Die wichtigsten sind Register, Hauptspeicher und E/A-Bereich.
- Adreßbildung:
  Im einfachsten Fall absolut, also komplette Adresse in Instruktion. Komfortabler ist die relative Adressierung über Basis- und Indexregister, bei denen nur ein Adreß*teil* in der Instruktion steht.

3. Informationsspeicher

3.1 Einführung in Speicherverwaltung

Ziele sind:

- großer logischer Adreßraum
- schnelle Antwortzeiten
- verschiebbarer Code (relocation)
- Schutz von Speicherbereichen (protection)

3.2 Cache (L2-Cache)

Wird vom Prozessor nicht bemerkt und ist ungefähr so schnell wie ein internes Prozessorregister.

Nach der Entfernung von der CPU unterscheidet man

- Level-1-Cache, der sich im Prozessorchip befindet.
- **Level-2-Cache**, der zwischen Prozessorchip und Hauptspeicher z.B. als Karte oder Modul gesteckt ist. Diesen Second-Level-Cache meint der Laie gemeinhin, wenn er von „Cache“ in seinen Verkaufsprospekten schreibt.
- Level-3-Cache, wenn z.B. ein Teil des Hauptspeichers als Festplattencache genutzt wird.
- Eine neue Entwicklung ist, den L2-Cache in einer zweiten Schicht direkt auf den Prozessorchip zu pappen.[1] Vorteil dabei: Der Prozessor adressiert den L2-Cache nicht mit dem langsameren Systemtakt, sondern mit seinem schnelleren Prozessortakt (z.B. also 200 MHz anstatt 66 MHz.) Dadurch ändert sich die Numerierung wieder.

Voraussetzung: Lokalität!

- Temporäre Lokalität: Wenn in naher Zukunft nur Instruktionen und Daten benötigt werden, die kurz zuvor referenziert wurden. Gegeben bei Schleifen und Stacks z.B.
- Räumliche Lokalität: Wenn in naher Zukunft nur Instruktionen und Daten benötigt werden, die nahe bei den bisher referenzierten liegen. Gegeben bei sequentiellem Code und Array z.B.

1. Z.B. beim sogenannten “Pentium Pro“.

Die Trefferrate (hit ratio) ist die Wahrscheinlichkeit, daß sich die referenzierte Information im Cache befindet: $\frac{\text{Anzahl der Treffer}}{\text{Anzahl der Speicherzugriffe}}$ = Trefferrate. Die Fehlerrate ist 1-Trefferrate.

3.2.1 Line- und Cachegrößen

Die wichtigen Faktoren, die die Trefferrate bestimmen, sind die Linegröße und die Cachegröße. Erhöhung der Linegröße bewirkt eine bessere Trefferrate, da jede Line mehr der sog. „räumlichen Lokalität“ abdeckt. Die daraus resultierenden Nachteile sind der höhere Zeitaufwand beim Transport einer Line vom Hauptspeicher zum Cache und, eine fixe Cachegröße vorausgesetzt, eine kleinere Anzahl von Lines im Cache. Das bedeutet eine geringere „temporäre Lokalität“. Bei einer gegebenen Linegröße wird die Erhöhung der Cachegröße einen größeren Teil der temporären Lokalität abdecken, während dieselbe Abdeckung der räumlichen beibehalten wird. So haben größere Caches bessere Trefferraten. Eine Linegröße von 16 bis 64 Bytes gilt als gut (anno ‘89, Anmerkung des Autors). Kleinere haben zuwenig räumliche Lokalität und daher zuviele Misses. Größere sind angebracht für größere Caches und Hauptspeicher.

3.2.2 Anordnung im Cache

Ein wichtiger Designaspekt ist die Plazierung von Hauptspeicherblöcken (Lines) im Cache, so daß sie leicht (also schnell) wiedergefunden werden. Eine Line enthält mehrere Byte und ist im Cache von derselben Größe wie im Hauptspeicher. (Man geht von einem Byte als adressierbarer Einheit aus.)

Dafür gibt es drei verschiedene Strategien:

3.2.2.1 Direct Mapping

Die einfachste Methode. Line i des Hauptspeichers wird auf Line (i modulo #Lines im Cache) abgebildet. Wenn eine Line 16 Bytes enthält, bestimmen die untersten 4 Bit ($2^4 = 16$) der Adresse die verschiedenen 16 Bytes innerhalb einer Line. Die nächsten 10 Bit (falls 2^{10} Lines im Cache Platz finden) bestimmen die Adresse im Cache wegen modulo. Die höherwertigsten Bits (Tag), die noch übrig bleiben, dienen dazu festzustellen, ob die gewünschte Line hier gespeichert ist. Kein assoziativer (inhaltsbezogener) Vergleich nötig und triviale Ersetzungsstrategie. Dadurch

einfache Hardwareimplementation. Jedoch kann eine Cacheline nur eine von „2 hoch Tag“ Hauptspeicherlines enthalten, die auf ihn abgebildet werden. Daher kann die Trefferrate erheblich sinken, obwohl der Cache nicht gefüllt ist.

3.2.2.2 Fully associative

Das andere Extremum. Jede Hauptspeicherline kann an beliebiger Stelle im Cache gespeichert sein. Höhere Trefferrate, aber teuerer wegen Assoziativspeicher. (Bei Assoziativspeicher wird nach Speicher*inhalten* gesucht. Gegenteil von Suche nach bestimmten *Adressen* wie bei den Tags des direct mapping.) Eventuell aber langsamer, da assoziative Suche langsamer ist als direct mapping. Hier werden sämtliche Adreßbits als Tag verwendet außer den untersten line-internen 4 Bits.

3.2.2.3 Set-associative

Kombiniert beide Vorteile. (Set-)*Assoziativität* ist die Anzahl der Lines pro Set. Der Cache ist in Sets unterteilt. Wie beim direct mapping wird mit den untersten Bits (über den line-internen 4) das Set bestimmt, in dem die Line liegt. Also „log von #Lines“ in Set. Die restlichen höheren Bits (Tag) bestimmen assoziativ die Stelle im Set. Dafür sind weniger Komparatoren nötig als beim fully associative. Nämlich anstatt für sämtliche Lines im Cache nur für die Anzahl Lines im Set.

Wenn die Set-Assoziativität 1 beträgt, dann erhält man direct mapping. Mit Assoziativität = Lines im Cache erhält man fully associative.

3.2.3 Nachladestrategien

3.2.3.1 Wann wird nachgeladen?

Demand fetch

Eine Line wird nur in den Cache geladen, wenn sie referenziert (heißt: angesprochen, verwendet, aufgerufen, benötigt) wird und noch nicht enthalten ist. Also bei Cache-miss.

Prefetch

Eine Line wird geladen, die unmittelbar auf die referenzierte Line folgt. Nennt sich **one block lookahead (OBL)**. Hierbei wird unterschieden, wann der Prefetch gemacht wird:

- Immer. Beste Trefferrate, aber hoher Speicherverkehr.
- On Miss. Auch gute Rate und weniger Speicherverkehr.
- Tagged prefetch. Wie on miss und zusätzlich, wenn zum ersten Mal auf eine prefetched Line zugegriffen wird. Merkt man sich in einem zusätzlichen Tag in der Cacheline. So gut wie „immer“, jedoch weniger Busverkehr.

Prefetch macht Demand-fetch nicht überflüssig, da keine Prefetch-Strategie die Zukunft voraussagen kann.

3.2.3.2 Was wird verdrängt?

Strategien dafür sind LRU, FIFO und Random. Komplizierte Lösungen sind nicht möglich, weil sie in Hardware schnell realisiert werden müssen.

3.2.4 Hauptspeicher-Updates

Wenn eine Line aus dem Cache verdrängt wird, kann Information verloren gehen, wenn die Line modifiziert wurde und die Cache-Hardware nicht sicherstellt, daß Datenübereinstimmung (**data coherence**) zwischen dem Cache und dem Hauptspeicher besteht. Es gibt zwei Möglichkeiten, dieses Problem zu lösen:

Beim **writethrough** wird bei jeder Schreiboperation der Cache und der Hauptspeicher beschrieben. Von Vorteil ist, daß der Hauptspeicher immer eine korrekte Kopie jeder Cacheline hat. Das vereinfacht die Datenkohärenz in Multiprozessorsystemen mit mehreren Caches. Allerdings ist der Speicherverkehr hoch, weil jeder Schreib-Hit dennoch einen Hauptspeicherzugriff auslöst. Bei einem Schreib-Miss kann die Line, abhängig von der Nachladestrategie, in den Cache kopiert werden oder nicht.

Die zweite Methode ist **copy-back**. Hier wird nur dann eine Cacheline zurückkopiert in den Hauptspeicher, wenn sie aus dem Cache verdrängt wird. Das spart Speicherverkehr. Entweder wird eine Line dabei immer im Hauptspeicher auf den neuesten Stand gebracht, egal ob sie geändert wurde oder nicht (**always copy-back**) oder nur wenn sie tatsächlich geändert wurde (**flagged copy-back**), was durch ein hinzugefügtes „dirty“ flag angezeigt wird. Bei einem Write-Miss wird im Gegensatz zum write-through beim copy-back immer die entsprechende Line im Cache abgelegt. Beim flagged copy-back wird der Transfer zum Hauptspeicher deutlich reduziert. Nachteilig bei den copy-back Strategien ist, daß der Hauptspeicher manchmal eine alte Kopie der Daten hat, so daß, wenn ein Prozessor oder

Cache versagt, nicht immer der Inhalt des Hauptspeichers wiederhergestellt werden kann. Außerdem muß dafür gesorgt werden, daß, wenn ein anderer Cache oder ein I/O-Gerät Daten verlangt, die Daten vom Cache oder vom Hauptspeicher kommen, je nachdem wer eine gültige Kopie hat.

3.3 Virtuelle Speicher

3.3.1 Begriffsbildung

Adressen, die von Programmen (Prozessen) benutzt werden, heißen **virtuelle Adressen**, weil sie sich von Adressen unterscheiden können, die benutzt werden, um auf den Hauptspeicher zuzugreifen. Der gesamte Adreßraum, der von einem Prozeß benutzt werden kann (maximal also der gesamte Adreßraum der CPU), wird **virtueller Adreßraum** genannt. Die Adressen, die für Hauptspeicherzugriffe benutzt werden, nennt man **reale Adressen**, weil für jede solche Adresse eine entsprechende Hauptspeicherstelle real existiert.

Einem Prozeß kann also mehr (virtueller) Speicher zugeteilt werden, als Hauptspeicher zur Verfügung steht. Jedem Prozeß wird eine bestimmte Menge Hauptspeicher zugeteilt (bestenfalls der ganze), auf den der virtuelle Speicherraum abgebildet wird. Wenn die Hauptspeicherpartition kleiner ist, als der virtuelle Adreßraum des Prozesses, dann kann immer nur ein Teil der Seiten dort gehalten werden.

Die Umsetzung von virtuellen in reale Adressen geschieht durch Hardware: die **Memory-Management-Unit (MMU)**. Wenn die Adreßumsetzung transparent (unsichtbar) für den Prozeß geschieht, spricht man von einem **virtuellen Speichersystem**. Die Alternative ist die **Overlay**technik, bei der ein Prozeß in verschiedenen Teile zerlegt ist, von denen einer immer präsent bleibt und die übrigen gegeneinander nach Bedarf austauscht zwischen Hauptspeicher und Sekundärspeicher. Dies geschieht dann natürlich nicht transparent für den Prozeß.

Lage des Cache bzgl. der MMU

Der Cache kann im realen (hinter MMU) oder im virtuellen Adreßraum (vor der oder parallel zur MMU) sein.

Für die erste Möglichkeit spricht ihre Simplizität (Einfachheit, für Nicht-Lateiner). Allerdings ist es möglich, die MMU-Übersetzung parallel zum Cachezugriff zu machen, indem während der Seitenumrechnung das Displacement zum Cache

geschickt wird. Anschließend vergleicht man im Falle eines Cachetreffers die reale Seitenadresse mit der im Cache. Damit erreicht man etwas mehr Geschwindigkeit und den Nachteil, daß der Cache auf Blockgröße (Anzahl möglicher Displacements) beschränkt wird.

Im zweiten Fall spart man bei einem Cachetreffer die MMU-Zeit ein und erhält sein Datum sofort. Der Cache kann hierbei größer sein. Der Cache im virtuellen Adreßraum kann on-chip realisiert werden. Allerdings gibt es drei Probleme dabei:

1. Virtuelle Adressen sind prozeßabhängig. Daher muß beim Prozeßwechsel entweder der Cache invalidiert werden oder eine Prozeß-ID im Tag mitgespeichert werden. Oder man verwendet einen einzigen virtuellen Adreßraum, der für jeden Prozeß gültig ist. Aber auch damit wird der Tag vergrößert also auch die Cachekosten und die Komplexität.
2. Mehrere virtuelle Adressen können auf dieselbe reale abgebildet werden (Synonym). Bei Änderung einer Kopie im Cache stimmen alle anderen nicht mehr. Außerdem könne I/O-Controler z.B. andere virtuelle Adressen verwenden. Auch hier könnte ein gemeinsamer virtueller Adreßraum Abhilfe schaffen, der ein „shared segment“ enthält. Oder man verwendet ein **Reverse Translation Buffer** (RTB, Gegenteil vom TLB s.u.), um nachzusehen, ob bei einem Cachemiss das Datum unter anderer virtueller Adresse vorhanden ist. Diese wird dann auf die aktuelle geändert und an ihre neue Position gerückt, sonst normaler Memoryzugriff.
3. Da I/O-Geräte lieber reale Adressen verwenden, muß Sorge getragen werden, daß ein im Hauptspeicher geändertes Datum im Cache invalidiert wird. Dies kann durch virtuelle Adressierung im I/O erreicht werden. Bleibt aber wie in 2 das Synonymproblem. Oder I/O mit realen Adressen und Verwendung eines RTB zum Invalidieren der Matchingadressen. Oder das BS stellt sicher, daß aktive I/O-Bereiche nicht in den Cache kommen.

3.3.2 Dynamische Zuordnungsverfahren

Beim statischen Zuordnungsverfahren (static relocation) werden die virtuellen Adressen in reale zur Ladezeit übersetzt. Wenn das Programm dann ausgeführt wird, werden reale Adressen generiert.

Wenn der zugeteilte Hauptspeicherbereich zur Ladezeit nicht bekannt ist, benutzt das Programm virtuelle Adressen, die während der Ausführung umgesetzt (relocated) werden müssen. Das ist dynamische Zuordnung (dynamic relocation).

Hier werden nur dynamische Zuordnungsverfahren betrachtet, da statische mit Rechnerarchitektur nichts zu tun haben. Die konventionellen Adreßübersetzungs-

mechanismen für „dynamic relocation“ sind relocation, paging, segmentation und paged segmentation.

3.3.2.1 Relocation

Bei älteren Computern wurde das memory management für dynamic relocation oft durch **relocation register** erfüllt. Ein Prozeß benutzt einen virtuellen Adreßraum, der kleiner oder gleich dem realen Adreßraum ist, startend mit 0. Wenn der Prozeß irgendwohin in den Speicher geladen wird, dann speichert das relocation register diese reale Startadresse. Bei jeder Referenz wird nun die virtuelle Adresse zum relocation register hinzuaddiert, um die reale zu erhalten. Bei einem Kontextswitch (Prozeßwechsel) wird einfach das Register umgeladen.

Um die Speicher und Zeit für die relocation Addition zu sparen, kann man die niederwertigen k Bits des relocation register streichen und erlaubt somit nur Startadressen, die Vielfache von 2^k sind. Es können ein oder mehrere relocation register verwendet werden.

Eine Adreßraumverletzung kann durch **protection register** abgefangen werden. Darin wird entweder die größte reale Adresse des Prozesses abgespeichert oder die Größe des virtuellen Adreßraums (Vorteil, daß der Check bzgl. Overflow parallel zur relocation Addition gemacht werden kann; sonst muß die Addition erst abgewartet werden). Underflow (negative virtuelle Adresse) muß auch gecheckt werden.

Nachteilig: Beschränkung des virtuellen Adreßraums, reale Partition muß zusammenhängend sein.

3.3.2.2 Paging

Paging unterteilt den virtuellen Adreßraum in Blöcke, **pages**, die alle gleich groß sind. Der Hauptspeicher ist in Blöcke derselben Größe eingeteilt, **page frames**. Die Umsetzung der virtuellen in eine reale Adresse wird mit Hilfe einer **page table (PT)** für jeden Prozeß erledigt. Diese tables liegen irgendwo im Hauptspeicher und werden via **page table register (PTR)** adressiert. Bei einem Kontextswitch wird das PTR umgesetzt, so daß es auf die page table des zu aktivierenden Prozesses zeigt.

Die grundsätzliche Adressierung erfolgt über die Seitenadresse konkateniert mit einem Displacement. Bei Referenz wird die *virtuelle Seitenadresse als Index* genommen, um in der page table den zugehörigen Eintrag zu finden. Dieser enthält die Zugriffsrechte (read, write, execute: RWX), ein Präsenzflag und die reale Seitenadresse.

Ein einfaches Pagingsystem lädt einen Prozeß in seiner Gesamtheit. Da die Seiten beliebig in den Hauptspeicher geladen werden können, ist es nicht nötig eine zusammenhängende Partition zu belegen wie bei „relocation“. Durch das komplette Laden entsteht **superfluity**, was bedeutet, daß viele Seiten des Prozesses in nächster Zeit nicht gebraucht werden, also sinnlos Hauptspeicher belegt wird.

Das zweite Problem (begrenzter virtueller Speicher) wird dadurch aufgehoben, daß beim Paging die Größe des realen Speichers überschritten werden kann, wenn man **demand paging** benutzt. Dies basiert auf einem one-level store. Bedeutet, daß der Adreßraum des Hauptspeichers und des Sekundärspeichers einen einzigen Adreßraum bilden. Virtuelle Adressen werden in diesen one-level store Raum übersetzt, so daß es transparent für den Prozeß ist, ob die Seite im Haupt- oder im Sekundärspeicher liegt. Bei einem **page fault** (Referenz auf nicht geladenen Seite) wird diese geladen. Bei solchen sog. inaktiven Seiten kann das Feld in der page table für die Realadresse benutzt werden, um auf die Stelle im Sekundärspeicher zu verweisen. Ein page fault führt wie jede I/O-Operation zur Suspendierung des Prozesses.

Wenn eine extrem große page table im Hauptspeicher liegt, von der das Programm nur wenig nutzt, wird dies **table superfluity** genannt. Das kommt daher, weil (in jeder page table) für jede mögliche virtuelle Adresse ein Eintrag sein muß, der die o.a. Elemente enthält. Auch bei speichergenügsamen Prozessen müssen für wesentlich mehr Speicheradressen Einträge vorhanden sein als vom Prozeß benötigt werden. Die meisten Programme benötigen nämlich weniger als den gesamten virtuellen Adreßraum. Daher ist bei einer page table, die größer als der reale Speicher ist, demand paging auch für diese PT nötig.

Ein weiteres (Verschwendungs-) Problem ist die **interne Fragmentierung**, die dadurch entsteht, daß Prozesse wohl nie Größen haben, die dem Vielfachen einer Seitengröße entsprechen. So ist ein Teil der letzten Seite immer ungenutzt. Wenn man allerdings die Seitengröße verringert, dann bekommt man größere page tables.

3.3.2.3 Segmentation

Um eine logische Organisation des virtuellen Adreßraums zu bekommen, die der Benutzersicht des Speichers entspricht, führt man das Konzept der Segmentierung ein. Es ist unnatürlich, ein Programm in Teile gleicher Größe zu zerlegen. Normale blockstrukturierte Sprachen z.B. Pascal und C erlauben logische Einheiten wie Prozeduren und Module mit unterschiedlichen Größen. Solch eine logische Einheit (Entity) kann durch Speicherung der Information einer Entity in einem Segment gehandhabt werden. Im Gegensatz zur page kann ein Segment wachsen und schrumpfen während seiner Lebenszeit wie das der Fall ist für ein Stacksegement.

Der Unterschied zum paging besteht im Folgenden: Anstelle virtueller Seitenadressen werden virtuelle Segmentadressen verwendet. Hinzu kommt in jedem **segment table entry (STE)** bzw. **segment descriptor** ein Feld, das die Segmentlänge angibt. Auch die gesamte Verwaltung inklusive **segment table register (STR)** und **segment table** geschieht analog zum paging.

Es taucht ein ernsthaftes Problem auf: **externe Fragmentierung**. Das bedeutet, daß ein Segment nicht geladen werden kann, weil der noch ausreichend freie Hauptspeicherplatz zu zersplittert ist, als daß das Segment in eine Lücke geladen werden könnte. Dadurch wird die zeitaufwendige **compaction** des Speichers nötig, die die Segmente an den Anfang zusammenschiebt.

3.3.2.4 Paged Segmentation

Da Segmentation ebenfalls unter superfluity leidet wird es oft mit demand paging kombiniert. Dabei wird jedes Segment in pages unterteilt und jede virtuelle Adresse setzt sich aus virtueller Segment- und virtueller Pageadresse zusammen plus Displacement. Wenn ein Prozeß aktiv ist, muß seine segment table im Speicher präsent sein. Table superfluity wird reduziert, indem nur die page tables des Segments geladen werden, die zum working set des Prozesses gehören.

Die Zugriffsrechte werden hierbei in den STEs kontrolliert. Falls die STEs keine Längenfelder haben und die Zugriffsrechte in den PTEs liegen, ist es nur eine erweiterte Form von Paging. Two-level-paging. Der Vorteil gegenüber dem one-level paging ist, daß nur benötigte page tables im Speicher sind anstelle von einer page table für den ganzen virtuellen Adreßraum.

3.3.3 Speicherverwaltungsverfahren

3.3.3.1 Fetch-Methode

Im Prinzip kann eine Information geladen werden, bevor sie benötigt wird, **prefetching**, oder wenn sie benötigt wird, **demand fetching**. Die gebräuchlichere, weil einfachere Methode ist die demand fetching. Im anderen Fall müßte eine schlechte Vorhersage mit falsch geholter Information und deren Ersetzung bezahlt werden. Bei segmentation wäre ein prefetching nicht sinnvoll, da die Segmente *logisch unabhängig* sind.

3.3.3.2 Plazierungs-Methoden

Beim paging ist eine placement policy trivial, da jede Seite in jedes Loch paßt. Bei segmentation gibt es Löcher verschiedener Größe, die in einer Liste mit Startadresse und Größe eingetragen werden. Methodenabhängig wird die Liste verschieden sortiert:

Best fit

Sortiert aufsteigend nach Lochgröße. Erstes ausreichend große Loch wird gewählt. Kleiner Verschnitt im Einzelfall, aber viele extrem kleine Löcher und daher großer globaler Verschnitt, der eine häufige Kompaktierung nötig macht.

Worst fit

Sortierung nach absteigender Lochgröße. Erstes Loch wird gewählt, falls es groß genug ist, sonst kein Einlagern möglich. Es werden extrem kleine Löcher vermieden. Nachteilig ist, daß die Löcher zu identischer Größe tendieren, dadurch großer Verschnitt und ebenfalls häufige Kompaktierung nötig.

First Fit

Sortiert nach aufsteigender Adresse. Erstes genügend großes Loch wird gewählt. Dadurch viele kleine Löcher bei niedrigen Adressen und deshalb Listendurchlauf unnötig lang.

Rotating first fit

Wie oben, nur daß die Suche ab letzter Plazierung begonnen wird. Damit im wesentlichen keine Nachteile, gleichmäßige Verteilung, gute Ausnutzung, wenig Verschnitt.

Binary buddy

Hier wird die Segmentgröße grundsätzlich auf die nächste Zweierpotenz aufgerundet. Dadurch nur Löcher der Größe 2^k. Es wird für jede Lochgröße eine separate Liste gehalten.

3.3.3.3 Ersetzungsmethoden

Bei Partitionen[2] fester Größe muß immer im Falle eines page- (oder segment-) fault (Partition komplett belegt, Seite nicht dabei) ausgetauscht werden. Bei variablen Partitionen kann eventuell die Größe geändert werden.

Ersetzungsmethoden für fixe Partitionen

- FIFO
  Es wird hierbei leider nicht beachtet, daß die älteste Seite, die am häufigsten referenzierte sein kann. Etwas temporäre Lokalität wird berücksichtigt.
- LRU
  Bessere Methode, um temporäre Lokalität zu beachten.

Ersetzungsmethoden für variable Partitionen

In der Vorlesung wurde nur die working-set-Methode besprochen. Es gibt jedoch noch weitere.

Das **working set** eines Prozessors sind die Seiten, die zum Zeitpunkt t in den letzten T Zeiteinheiten referenziert wurden. Diese Methode ist deshalb wichtig, weil sie Speicherverwaltung und Prozeßverwaltung verbindet. Man spricht vom **Workingsetprinzip**, wenn die Speicherverwaltung so aufgebaut ist, daß

- ein Prozeß nur für lauffähig erklärt wird, wenn sein gesamter Workingset[3] im Hauptspeicher ist,
- keine Seite aus dem Workingset eines laufenden Programms aus dem Hauptspeicher entfernt wird.

2. Partition ist der Hauptspeicherbereich eines Prozesses.
3. Menge der Seiten zur Zeit t, die der Prozeß in den letzten T Referenzen verwendete.

Dadurch kann der Hauptspeicherbereich (Partition) eines Prozesses mit der Zeit variieren.

Seitenaustauschalgorithmen sind im Wesentlichen nicht in der Hardware eines Rechners realisiert, sondern im Betriebssystem.

3.3.4 Hardwareunterstützung zur Adreßumsetzung

3.3.4.1 Translation Lookaside Buffers (TLB)

Grundlage ist hier die temporäre Lokalität. Man speichert einfach die n zuletzt vorgenommenen Adreßumrechnungen in einem Cache. Adressierung der Cacheeinträge erfolgt assoziativ über die Einträge RWX, V_s, V_p.

Probleme dabei: Bedingt durch die beschränkte Größe kommen „misses“ vor. Da nur ein TLB für alle Prozesse aus Kostengründen möglich ist, müssen bei einem Kontextswitch alle TLB-Einträge invalidiert werden. Da das bei jedem page fault passiert, kommt es häufig vor. Abhilfe schafft hier die Addition eines Tagfeldes mit Prozeß-ID.

3.3.5 Inverted Page Table (IPT)

Die normale page table enthält für jede virtuelle Seite einen Eintrag. Daher superfluity. Die Lösung ist eine Tabelle mit nur einem Eintrag pro realer Seite. Alle anderen Einträge würden eh nur auf den Sekundärspeicher verweisen. Man beschränkt sich damit auf die relevanten Einträge. Der Vorteil ist also, daß hier eine beschränkte, feste Menge Speicher belegt wird für die table, unabhängig von der Größe des virtuellen Adreßraums. Daher kann die IPT in schnellem Speicher implementiert werden.

Die TLB ist eine übliche Hashtabelle mit Kollisionslisten. Gehasht wird über die virtuelle Seite. Jeder Eintrag enthält ein Link-Feld, das auf einen weiteren Eintrag eventuell verweist mit derselben Hashwertung, die virtuelle Seitenadresse und die reale Seitenadresse.

IPT und TLB sind kombinierbar. Z.B. im POWER-PC. Die IPT ist billig, weil sie normaler Speicher ist.

4. Informationstransport (Bussysteme)

4.1 Allgemeines

Historisch kommt die Bezeichnung BUS von Bidirectional Universal Switch. Heute definiert man so: Busse dienen dazu, zwei oder mehr digitale Systemelemente zu verbinden. Ein **Bus** ist eine Menge von Leitungen, die von diesen Systemelementen gemeinsam benutzt wird und zur Kommunikation zwischen diesen Systemelementen dient. Der Term „Bus“ impliziert gewöhnlich **parallel** verlaufende Verbindungen, wobei mehrere Signale gemeinsam nahezu die gleiche Route zu nahezu der gleichen Zeit zurücklegen. Dies kann jedoch simuliert werden durch eine Sequenz von Signalen auf einer einzelnen Verbindung, was dann ein **serieller** Bus wäre. Ein **Bus-System** ist eine Menge von Bussen, die benutzt wird, um die verschiedenen Systemelemente (Komponenten) in einem Computersystem zu verbinden.

4.2 Bus-Hierachie

Bus-Systeme bestehen aus einer Hierachie von Bussen auf verschiedenen Ebenen. Die folgenden Bus-Ebenen können unterschieden werden:

Platinen-Ebene (board level):

Board-level-Busse verbinden Systemelemente, die aus Komponenten bestehen (z.B. Chips), auf einer oder mehreren Platinen. Sie werden **local bus** genannt. Diese Busse werden meistens gemeinsam von vielen Komponenten benutzt, um die Anzahl der Verbindungen zwischen den Komponenten zu reduzieren.

Backplane-Ebene:

Backplane-level-Busse dienen zur Kommunikation zwischen Systemelementen, die aus Platinen bestehen (boards). Hierzu gehört der Nubus.

Interface-Ebene:

Interface-level-Busse stellen den gemeinsamen Kommunikationspfad zur Verfügung zwischen I/O-Geräten (Laufwerke, Drucker) und dem Rest des Systems.

Hierzu gehört der SCSI- (Small Computer System Interconnect) Bus. Dieser Bus hat große Ähnlichkeit mit Backplane-Bussen.

Ein local bus verbindet also z.B. CPU, PROM und RAM auf demselben Board. Auf Backplane-Ebene werden Boards verbunden und ein Interface-Bus verbindet den I/O-Controler auf einem Board mit einem I/O-Gerät. Der Controler ist über einen local bus mit der CPU auf dem Board verbunden.

Hier wird auf die Backplane-Ebene fokussiert, weil sie allgemein und meistens Prozessor/Geräte-unabhängig ist und weil ihre Charakteristik auf die anderen Level angewendet werden kann. Diese Charakteristik umschließt mechanische, elektrische und logische Eigenschaften. Die Betonung liegt hier auf den logischen Eigenschaften von Backplane-Bussen.

4.3 Busklassifizierung nach Dedizierung

Busse können entweder dediziert sein oder nicht-dediziert. Ein **dedizierter** Bus leistet nur *eine einzige Funktion* wie z.B. Verbinden von der CPU mit dem I/O-System im Falle eines I/O-Busses. Hierbei erfüllt jedes Board eine spezielle Funktion (Instruction memory, Data memory, I/O system) und ist mit der CPU über einen eigenen Bus verbunden. Wegen deren dedizierter Funktion benötigt ein Computersystem gewöhnlich mehrere verschiedene dedizierte Busse. Der Vorteil dedizierter Busse ist ein hoher Durchsatz; mehrere Funktionen können simultan kommunizieren, falls mehrere dedizierte Busse vorhanden sind. Außerdem ist eine Spezialauslegung jeweils möglich. Der Nachteil ist ein hoher Aufwand wegen vielen dedizierten Bussen systemweit und die Fehleranfälligkeit aufgrund vieler Verbindungen. Daher werden dedizierte Busse nicht oft auf Systemebene in kleineren Rechnern verwendet.

Nicht-dedizierte Busse werden geteilt (shared) von *vielen Funktionen* (eine Funktion kann von einem oder von mehreren Boards geleistet werden). Daher werden sie auch als „shared buses“ bezeichnet. Shared buses resultieren in einfacheren (evtl. nur 1 Bus systemweit), billigeren Systemen verglichen mit dedizierten Bussen. Das allerdings auf Kosten von geringerer Datentransferkapazität (Busbandbreite) und der Notwendigkeit eines Arbitrierungsmechanismus, der Konflikte verhindern muß, die auftreten, wenn zwei oder mehr potentielle Benutzer den Bus gleichzeitig anfordern. Für Rechner mit relativ kleiner Anzahl von low-perfor-

mance-boards ist ein Bussystem bestehend aus einem einzigen, shared bus, ausreichend stark.

4.4 Busklassifizierung nach Partitionierung

Shared buses können weiter klassifiziert werden nach der Art, wie die Funktionen des Systems über die Boards verteilt (partitioniert) sind. Die Partitionierung kann nach dem Resourcentyp oder der geleisteten Funktion erfolgen.

Ein Bus heißt **resourcenpartitioniert**, wenn Ressourcen desselben Typs (z.B. CPU, Speicher, I/O) zusammengepackt werden und die „Pakete“ verbunden sind durch den Bus.

Der VME-Bus gehört dazu. Diese Art wurde ausgedacht, als der Integrationsgrad so war, daß Speicher aus einem oder mehreren Boards (also nicht aus wenigen Chips) bestand und I/O-interfaces nicht intelligent waren (d.h. sie hatten keinen eigenen Prozessor), so daß I/O unter CPU-Kontrolle erledigt wurde. Die Hauptcharakteristika des ressourcenpartitionierten Busses sind:

- Einzel-Prozessor-orientiert: D.h., daß der Bus gewöhnlich einen default-master (die CPU) hat. Zentrale Buszuteilung (Arbitration) dadurch und alle Interrupts haben ein default-Ziel: die CPU.
- Speicherbus-orientiert: CPU-Speicher-Verkehr dominiert die Busnutzung. Deshalb folgende Charakteristika:
  (a) Übertragung von Einzeldaten: Die meisten Speicherzugriffe der CPU betreffen nur ein einzelnes Datenelement (keinen Block von Daten), Byte, Wort.
  (b) Kurze Buszykluszeit: Während die CPU auf den Speicher wartet, kann sie typischerweise nichts berechnen. Der Bus ist deshalb so designed, daß die Zykluszeit kurz ist. Erreicht wird das durch:
    — Getrennte Leitungen für Adresse und Daten, so daß sie parallel verarbeitet werden können.
    — Ein einfaches Busprotokoll (das Busprotokoll ist die Methode, Adressen, Kommandos, Daten und Statusinfos zu signalisieren).
    — Keine Überprüfung (z.B. parity), weil dies den Bus verlangsamen würde.
    — Asynchroner Betrieb, um sich auf verschiedenen CPU-Geschwindigkeiten und verschiedene Speichergeschwindigkeiten einzustellen.

- Optimales Passen von CPU- und Bussignalen: Dies ist wegen der kurzen Zykluszeit, die diese Klasse von Bussen oft abhängig macht von der Prozessorarchitektur.

Ein **funktionspartitionierter** Bus verbindet Boards (Komponenten), die eine halbautonome Funktion ausführen. Jede Funktion (Komponente) hat ihre eigene Prozessorfähigkeit, Speicher und I/O, verbunden durch einen local bus.

Durch technologischen Fortschritt kann ein einziges Board einen Prozessor, Speicher und I/O-Interface-Schaltkreise (falls nötig) enthalten. Das bedeutet, daß der high-speed CPU-Speicher-Verkehr durch einen local bus abgewickelt werden kann, so daß der Systembus nur für die Kommunikation zwischen intelligenten Funktionen benutzt wird. Beispiele: Multibus II und Futurebus. Die Hauptcharakteristika eines funktionspartitionierten Busses sind:

- Multiprozessor-orientiert.
- Nachrichten-orientiert: Kommunikation zwischen den intelligenten Geräten erfolgt per Nachrichten. Übertragung von Nachrichten kann parallel zum lokalen Berechnen erfolgen. Daher hat der Bus folgende Eigenschaften:
  - — Blöcke von Daten werden übertragen.
  - — Hoher Nachrichtendurchsatz: Anfragen für Nachrichtentransfers können in eine Schlange eingereiht und Nachrichten gepuffert werden. Daher ist nur die Busbandbreite von Bedeutung.
- Entkopplung von CPU und Bussignalen: Weil die CPU mit dem Systembus indirekt über einen Nachrichtenmechanismus aus Hardware kommuniziert, können CPU-Eigenschaften in Form von Signalen und deren zeitlichen Beziehungen entkoppelt werden vom Systembus. Funktionspartitionierte Busse sind daher eher prozessorunabhängig.

4.5 Terminologie bei Bussen

Eine an einen Bus angeschlossene Komponente heißt **master**, falls sie einen Buszyklus initiieren kann, sonst **slave**, der auf die Wünsche des master antwortet. Einige Geräte (Komponenten) können als master und slave agieren, aber nicht zur selben Zeit. Alle Komponenten, die als master auftreten können werden potentielle Busmaster genannt. Da nur einer von ihnen jeweils den Bus kontrollieren kann, muß ein **bus arbitration** Mechanismus entscheiden, welcher Busmaster als nächstes die Kontrolle bekommt. Beim Datentransfer heißt die sendende Komponente **source** und die empfangende **destination**. Eine komplette Sequenz von der Anforderung des Busses bis zur Beendigung des Datentransfers heißt **bus transaction**.

Sie kann aus mehreren Operationen bestehen, die jeweils mehrere **bus cycles** benötigen können. Um einen Bustransfer zu machen, finden die folgenden Operationen statt:

- Anforderung (request operation);
- Zuteilung (arbitration operation);
- Adressierung (adressing operation);
- Datenübertragung (data transfer operation);
- Fehlererkennung (error detection und signalling operation).

4.6 Datentransfer

Es gibt verschiedene Typen von Datentransferoperationen:

- Write operation.
- Read operation.
- Read-modify-write operation. Adresse nur einmal.
- Read-after-write operation. Adresse nur einmal.
- Block data transfer operation. Einmal Adresse, n-mal Daten.
- Split data transfers. Im Falle einer Leseoperation evtl. lange Zugriffszeit (bis die Daten kommen). Der Bus kann dann freigegeben werden, um ihn nicht unnötig zu blockieren. Der Transfer kann später wieder initiiert werden, mit dem slave diesmal als master. Dies wird oft **message** oder **packet switching** genannt im Gegensatz zum **circuit switching**, wobei die Verbindung erhalten bleibt.

4.7 Adressierung

Um einen Kontakt herzustellen mit einem oder mehreren slaves, adressiert der master diese. Die Adressierung besteht meist aus zwei Teilen: Boardadressierung und Adressierung des Datenelements auf diesem Board. Das kann mit high- und loworder Bits gleichzeitig geschehen.

Slaveadressenspezifikation

Während einer Busoperation wird die Adresse des betroffenen slave über die Adreßleitungen des Bus spezifiziert. Eine Komponente gilt als **selektiert**, wenn ihre Adresse (vom Board) mit der gewünschten übereinstimmt. Man unterscheidet zwei Methoden zur Anweisung von Boardadressen: logische (ortsunabhängig) Adressierung und geographische (ortsabhängige) Adressierung.

Logische Adressierung bedeutet, daß jedes Board (Komponente) eine eindeutige Adresse oder eine Gruppe von Adressen durch (manuelle) Schalter hat. Beispiel VME-Bus, Unibus.

Geographische Adressierung bedeutet, daß ein Board durch seine physikalische Position adressiert wird. Diese ist die Stelle, in der das Board in die Backplane eingesetzt ist, die **slot number**. Macht nur bei funktionspartitionierten Bussen Sinn oder zur Initialisierung, um logische Adressen zu verteilen.

Anzahl der slaves bei einer Bustransaktion

Read- und Writeoperationen, die mehr als einen slave verwenden, werden **broadcall** bzw. **broadcast** Operationen genannt. Da bei einem broadcall alle gewählten slaves ihre Daten auf den Bus legen, bekommt man ein kollektives AND oder OR. Daher kann broadcall nur für einige Fälle verwendet werden, z.B. um die Quelle eines interrupts zu identifizieren, wobei jeder potentielle interrupter eine single-bit Position erhält, um sich selbst zu identifizieren. Broadcall ist für die Geschwindigkeit wichtig, z.B. um verschiedene Caches simultan upzudaten oder für eine Resetoperation, wobei jedes Board seinen Anfangszustand annehmen soll.

4.8 Bus-Protokolle

Informationsübertragungen über einen Bus sind Gegenstand von Timingregeln, so daß source und destination synchronisiert werden können. Die Zeit, die bis zu einem stabilen Zustand bei der destination führt, wird **bus delay** genannt. Diese

TABELLE 3: Typischer Ablauf einer Schreiboperation

Zeitschritte	Source	Destination
1	Lege Information auf Busleitungen.	
2		Empfangene Information stabilisiert sich.
3	Signalisiere, daß Information stabil ist.	
4		Bemerke, daß Information stabil, übernimm Information.
5		Bestätige, daß Information genommen wurde.
6	Bemerke, daß Information genommen wurde.	
7	Entferne Information.	

TABELLE 3: Typischer Ablauf einer Schreiboperation

Zeitschritte	Source	Destination
8	Bestätige, daß Information entfernt wurde.	
9		Bemerke, daß Information entfernt wurde.
10		Bestätige, daß Transfer beendet.
11	Starte nächsten Transfer.	

Sequenz, oder eine Teilmenge davon, kann implementiert werden gemäß verschiedener Konzepte: synchron, d.h. alle Ereignisse finden zu festen Zeitpunkten statt; asynchron, d.h. die Signale der Sequenz (Ereignisse) können zu beliebigen Zeitpunkten generiert werden; und semi-synchron, d.h. die Signale können nur in fester Beziehung zu einer Clock erzeugt werden (starres Taktraster).

Synchrones Bustiming

Der Augenblick, zu dem die Ereignisse auftreten, ist fest und hängt nicht von der source und/oder destination ab. Synchrone Bustimingprotokolle haben einen theoretisch höheren Durchsatz als asynchrone Protokolle, weil keine Signale zwischen source und destination nötig sind. Die Schritte 3, 5, 8 und 10 sind also nicht nötig. Der langsamste Partner bzw. der längste Weg diktiert die Geschwindigkeit. Wird daher nur in spezialisierten Bussen benutzt.

Asynchrones Bustiming

Das Grundcharakteristikum des asynchronen Protokolls ist das Fehlen einer clock. Dies bedeutet, daß jeder Schritt der Sequenz durch ein spezielles Ereignis angezeigt werden muß, und dieses über den Bus zum Partner transportiert wird. Um aber den Busdelay zu vermeiden, nehmen einige Protokollvarianten an, daß das Timing bestimmter Ereignisse implizit geschieht. D.h. durch Annehmen einer genügend langen festen Delayzeit dafür. Abhängig von der Anzahl implizierter Ereignisse, unterscheidet man non-interlocked, half-interlocked und fully-interlokked Busprotokolle.

Semisynchrones Bustiming

Hier sind die Zeitintervalle zwischen den Ereignissen zwar variabel, jedoch nur Vielfache der Clockperiode. Der Vorteil ist, daß die Rauschempfindlichkeit gerin-

ger ist, weil nur zu den festgelegten Momenten Fehlinterpretationen verursacht werden können.

Asynchrones fully interlocked Protokoll

Dieses Protokoll wird so genannt, weil alle Übergänge bei den Kontrollsignalen nur als Antwort auf Übergänge des Partners passieren. Ein anderer Ausdruck für dieses Ineinanderhaken ist **handshaking**. Die Kontrollsignale haben alle eine variable Länge. Um Hardwarefehlfunktionen nicht den Bus blockieren zu lassen, wird eine **time-out Mechanismus** genutzt. Dieses Zeitlimit nimmt an, daß alle handshakes bis dann erledigt sein müßten.

Dieses Protokoll wird häufig benutzt, weil alle Geräte mit ihrer eigenen Geschwindigkeit arbeiten können: schnelle haben kurze Buszyklen und langsame werden automatisch eine längere Zykluszeit verursachen. **Zyklus** bedeutet, daß ein Datenpaket komplett verschickt wurde mit allem was dazu gehört an Kommunikation. Das handshaking von jedem Zyklus benötigt vier Flanken:

1. Die steigende Flanke des DR data ready, was valide Daten signalisiert.
2. Die steigende Flanke des DA/DE data accepted/data error, was anzeigt, daß destination die Daten übernommen hat.
3. Die fallende Flanke des DR, was signalisiert, daß source das DA/DE der destination gesehen hat und nun die Daten vom Bus nehmen und die nächste Bustransaktion vorbereiten kann.
4. Die fallende Flanke des DA/DE, was den Bus für die nächste Transaktion freigibt.

4.9 Bus arbitration (Buszuteilung)

Wie oben erwähnt muß der bus arbiter dafür sorgen, daß nur ein Sender zu jedem Zeitpunkt möglich ist. Dafür gibt es zwei wesentliche Möglichkeiten: statische und dynamische Zuteilung.

4.9.1 Statische Buszuteilung

Hier werden den Mastern zyklisch Zeitscheiben zugeteilt. Der betreffende Master kann davon Gebrauch machen, muß aber nicht. Vorteilhaft ist die Einfachheit dieses Verfahrens und der für jeden Master garantierte Durchsatz. Der Nachteil ist die Verschwendung an untätige Master. Das kann durch das folgende gelöst werden:

4.9.2 Dynamische Buszuteilung

Diese Methode erlaubt die dynamische Übertragung der Buseignerschaft, d.h. nach Bedarf. Die Zuteilung benutzt Buszuteilung und Busfreigabe.

4.9.2.1 Buszuteilung

Wenn der aktuelle Master den Bus freigibt, muß entschieden werden, welchem der anfragenden Master er zugeteilt werden soll. Drei Möglichkeiten:

- Priorität (Gefahr der Dominierung, **starvation**=Hunger).
- Faire Verteilung (kann für wichtige Master zu langen Wartezeiten führen).
- Kombinierte Verfahren (Höhere Prioritätsanfragen nach Priorität, niedrigere nach Fairneß).

Multiprozessorsysteme nutzen typischerweise die kombinierte Variante; für I/O Subsysteme wird Priorität benutzt; während andere Prozessoren, die User- und Systemprogramme ausführen, Fairneß nutzen.

4.9.2.2 Busfreigabe

Auch wieder drei Möglichkeiten:

- release on request: master behält Bus, solange niemand anders ihn will, auch wenn er nichts sendet (üblich bei Singleprozessorsystemen). Spart die Zuteilungszeit, da meist sowieso die CPU agiert.
- release when done: master behält Bus für seine Transaktion, gibt Bus dann frei.
- pre-emption: master kann während Transaktion durch master höherer Priorität unterbrochen werden. Dies kann bei langen Blocktransfers angewendet werden.

4.9.3 Buszuteiler Hardwaremechanismen

Hiermit wird die dynamische Zuteilung implementiert. Die Hardware kann zentralisiert oder verteilt implementiert werden. **Zentrale Zuteilung** bedeutet, daß die Zuteilungshardware an einem Ort ist. Sie kann dazu in einem der Module stecken oder in einem separaten **bus arbiter**. **Verteilte Zuteilung** heißt, daß die Hardware über die potentiellen Busmaster verteilt ist.

4.9.3.1 Zentrale Buszuteilung

Master, der Bus will, sendet request an arbiter. Dieser wählt eine aus und bestätigt durch grant. Drei Methoden:

Shared request with daisy-chained grant

Jeder potentielle master kann einen request via „bus request“-Leitung erzeugen. Diese Leitungen bilden ein ODER. Wenn der zentrale Arbiter einen request empfängt, sendet er ein bus grant zum ersten potentiellen bus master. Diese Leitung verbindet alle master und bildet eine Kette, die **daisy-chain**. So sendet jeder master das Signal weiter, falls er es nicht benötigt. Priorität durch Anordnung der Prozessoren in der Kette. Die drei Leitungen sind signalmäßig voneinander abhängig. Sozusagen geschachtelt: bus request, bus grant, bus busy, bus not busy, no bus grant, no request.

Zentrale Zuteilung, wegen zentralem arbiter. Die eigentliche Zuteilung ist dezentral innerhalb der daisy-chain.

Independent requests and grants

Hier hat jeder master sein eigenes Paar von bus request und bus grant Leitungen, die zum zentralen arbiter führen. Die Timingsequenz ist wie bei daisy-chain. Die ist ein übliches Verfahren bei Multiprozessorsystemen. Vorteil ist, daß fehlfunktionierende master ignoriert werden können und die kurze Zuteilungszeit. Nachteilig sind die vielen Leitungen.

Kombinierte Lösung

Z.B. mehrere direkt an den arbiter gebundene daisy-chains.

4.9.3.2 Verteilte Zuteilung

Wird benutzt bei Bussen, die optimiert sind für Multiprozessorsysteme. Ein potentieller master gibt seine arbitration priority number an seinen arbiter. Dieser legt die Nummer auf die gemeinsame request/grant-Leitungen. Dies tun auch alle anderen Anforderer. Jeder Anforderer vergleicht in seinem Arbiter seine Nummer mit der auf den gemeinsamen Leitungen. Falls eigene kleiner: Anforderung zurückziehen. Nach gewisser Zeit stabiler Zustand: Nummer auf gemeinsamen Leitungen = Nummer von genau einem Anforderer. Bekommt Bus. Bus busy setzen.

Dieses Prinzip nutzt z.B. der Nubus, Multibus II, Futurebus, wobei die Zuteilung aus einer Kombination von prioritized arbitration und fairness besteht.

5. Informationsverarbeitung: Arithmetik

5.1 Festkomma Addition/Subtraktion

Die „schnellste“ Realisierung geht über die **zweistufige Normalform**; disjunktiv oder konjunktiv, je nachdem wie die Terme verbunden sind in der „äußeren Stufe“.

Man betrachtet die Addition/Subtraktion als Boolesche Funktionen $B^n \rightarrow B$, wobei für jede Ergebnisbitstelle eine einzelne Funktion nötig ist. Für jede Funktion betrachtet man die möglichen Belegungen der n Eingabebits und stellt fest, ob das Ergebnis „1“ sein soll. Wenn ja, dann wird im DNF-Fall eine UND-Verknüpfung der n (evtl. negierten) Variablen gebildet, so daß dieser UND-Term „1“ wird. Das macht man für alle 2^n Möglichkeiten der n Inputvariablen und verknüpft alle UND-Terme mit ODER. Dieser DNF-Ausdruck ist genau dann „1“, wenn die Funktion „1“ ist. Er hat im schlimmsten Fall 2^n Konjunktionen mit jeweils n Variablen. Und das für jede Bitstelle des Ergebnisses. Man sieht, daß hier in Abhängigkeit von den n Inputbits ein 2^n-großer Aufwand entsteht. Der Vorteil ist jedoch, daß alle Ergebnisbits gleichzeitig berechnet werden können: parallele Auswertung der verschiedenen DNF-Formeln für die Ergebnisstellen. Schnell und aufwendig also. Leider zu teuer wegen exponetiellem Aufwand.

Deswegen probiert man eine modulare Kaskadenlösung. Ein Volladdierer wird als Modul pro Binärstelle verwendet. Ein Volladdierer berechnet aus zwei binären Ziffern und einem Übertrag der vorherigen Stelle die Summe und den neuen Übertrag. Ein Halbaddierer berücksichtigt keinen Übertrag der vorherigen Stelle. Beim sog. „**ripple carry**“-Addierer werden n Volladdierer so geschaltet, daß der Übertragsausgang jeweils in den Übertragseingang des nächsten Addierers mündet. Ripple bedeutet kleine Welle. Beim **sequentiellen** Addierer wird nur ein Volladdie-

rer verwendet und der Übertragsausgang taktweise an den Übertragseingang zurückgeleitet.

TABELLE 4: Vergleich der Addierer

	Hardware	**Zeit**
ripple carry	O(n)	n*Verzögerung von Volladdierer
sequentiell	const	n*(Verzögerung von Volladdierer+Taktverzögerung)

Beschleunigung: „**carry-lookahead**“-Addierer (CL-Addierer). Hier werden n Volladdierer an einen Carry-lookahead-circuit angeschlossen. Die Volladdierer für den CL-Addierer unterscheiden sich in ihren Ausgängen vom normalen Volladdierer: Anstelle des Carry-out haben sie einen Generate- und einen Propagate-Ausgang. Identisch sind Carry-in, x, y und der Sum-Ausgang. Der Generate- und der Propagate-Ausgang münden in die carry-lookahead-Schaltung, aus der jeweils ein carry-in einen Volladdierer versorgt. Die Ausgänge des Volladdierers für CL-Addierer sind wie folgt definiert:

$$z_i = x_i \quad y_i \quad c_{i-1} \qquad \text{(sum)}$$

$$g_i = x_i \quad y_i \qquad \text{(generate)}$$

$$p_i = x_i \quad y_i \qquad \text{(propagate)}$$

Die CL-Schaltung realisiert die parallele Berechnung der Überträge mit folgender Formel: $c_i = \sum_{j=0}^{i} g_j \prod_{k=j+1}^{i} p_k$

Das generate bedeutet, daß ein Übertrag an dieser Stelle erzeugt wird. Das propagate bedeutet, daß an dieser Stelle ein eventueller Übertrag von der vorherigen

Stelle weitergereicht wird. Damit läßt sich ein Übertrag für alle Stellen sofort berechnen. Beispiele:

TABELLE 5: carry-lookahead-Berechnung

Carry-stelle	Berechnung von c
c_0	$= g_0$
	$= x_0 \quad y_0$
c_1	$= g_0 p_1 + g_1$
	$= x_0 \quad y_0 \quad x_1 \quad y_1 + x_1 \quad y_1$
c_2	$= g_0 p_1 p_2 + g_1 p_2 + g_2$
	$= x_0 \quad y_0 \quad x_1 \quad y_1 \quad x_2 \quad y_2 + x_1 \quad y_1 \quad x_2 \quad y_2 + x_2 \quad y_2$

Der CL-Addierer hat eine Verzögerung von 4 Gatterschaltzeiten. Unabhängig von der Zahlengröße. Dabei wird vorausgesetzt, daß generate und propagate durch ein zweistufiges Schaltnetz (im Volladdierer) berechnet werden und alle carries c durch ein zweistufiges Schaltnetz (CL-Schaltung) berechnet wird.

Der CL-Addierer hat allerdings einen kubischen Aufwand. Das kommt von n Stellen benötigen n carry-Berechnungen. Diese sind Summen mit n/2 Summanden im Mittel. Pro Summand Produkt mit im Mittel n/4 Faktoren.

Für große Zahlen n macht man einen Kompromiß aus CL und ripple carry mit m ripple-Stufen die jeweils aus einem k-bit-CL-Addierer bestehen mit mk=n.

5.2 Festkomma Multiplikation

5.2.1 Einfacher „shift and add“-Algorithmus

Entspricht im Prinzip der gewohnten dezimalen schriftlichen Multiplikation zweier Zahlen mit anschließender Addition der Teilsummen. Der Unterschied ist nur, daß das Aufsummieren während der Berechnung der Teilsummen geschieht und die Zwischensumme nach jeder Multiplikation verschoben wird um eine Stelle, bevor die nächste Teilsumme draufaddiert wird. Sequentielles Verfahren. Für negative Zahlen muß erst komplementiert werden.

5.2.2 Erweiterter „shift and add“-Algorithmus: vorzeichengerecht

Grundsätzlich wie oben, jedoch wird z.B. beim shift bei Bedarf eine 1 links zugegeben, um negative Zahlen zu signalisieren.

5.2.3 Festkomma Division

Der einfachste Algorithmus dafür ist der sog. restoring-Algorithmus. Wenn man $\frac{x}{y} = q + r$ berechnen will, kann man für jede Binärstelle q_m von q feststellen, ob $q_m 2^m y > x$ ist. Wenn ja, dann muß q_m 0 sein, sonst 1. Bei der Implementierung wird y um i Stellen versetzt jeweils abgezogen und das Ergebnis kontrolliert. Man beginnt mit $i=m_{max}$. D.h. man zieht y erstmal von der höchstwertigen Stelle ab. Ist das Ergebnis negativ, addiert man y wieder dazu. Sonst setzt man q_m auf 1. Dann shiftet man x um eine Stelle nach links und zieht y von den beiden höchstwertigen Stellen ab wie oben. Dann von den drei höchstwertigen usw. bis man alle Stellen durch hat. Dann liegt y vor und der Rest ist der Rest von x.

5.3 Gleitkommaoperationen

5.3.1 Addition/Subtraktion

Wie üblich: Exponentenvergleich. Angleichen durch Mantissenverschieben. Addieren und Normalisieren. Der Exponentenvergleich geschieht am einfachsten durch Subtraktion der Exponenten voneinander. Damit erhält man auch gleich die Anzahl der zu verschiebenden Stellen.

5.3.2 Multiplikation

Auch wie üblich: Addiere die Exponenten (subtrahiere Excess, da einmal zu viel addiert, addiere Mantissenlänge weil rechten n bits des doppelt langen Ergebnisses weggeworfen werden). Multipliziere Mantissen. Normalisiere, falls nötig.

5.3.3 Division

Subtrahiere die Exponenten und führe Justierung durch (Excess-Addition). Dividiere die Mantissen, hebe nur Quotient auf. Normalisiere, falls nötig. Der Quotient kann n+1 bit lang sein, dann Rechtsshift der Mantisse und Exponentenanpassung.

6. Ein-/Ausgabe

6.1 Allgemeines

Hauptsächlich werden nur zwei Komponenten des Computers benutzt: CPU und Hauptspeicher. Die übrigen Komponenten des Rechners können locker als das **I/O-System** bezeichnet werden, weil es ihr Zweck ist, zwischen Hauptspeicher oder CPU und der Außenwelt Informationen zu übertragen. Das Ein-/Ausgabesystem umfaßt IO-Geräte (Peripherie), Kontrolleinheiten für diese Geräte und Software für die Durchführung der I/O-Operationen.

I/O-Systeme können danach unterschieden werden, in welchem Ausmaß die CPU für die Ausführung der I/O-Operationen benötigt wird. Eine **Ein-/Ausgabe** bedeutet einen Datentransfer zwischen einem I/O-Gerät und dem Hauptspeicher oder zwischen einem I/O-Gerät und der CPU. Und zwar gibt es folgende **Grundmethoden**:

- Programmierte E/A (**programmed I/O**): I/O wird komplett von der CPU kontrolliert; d.h. die CPU führt das Programm aus, das die I/O-Operationen initiiert, steuert und terminiert.
- Direkter Speicherzugriff (direct memory access, **DMA**): Die CPU ist immer noch für das Starten jedes Transfers verantwortlich, aber der I/O-Gerätcontroller kann dann allein einen Blocktransfer durchführen. Er hat die Fähigkeit, Speicheradressen zu erzeugen und Daten zum oder vom Systembus zu übertragen, d.h. er ist ein Busmaster. Die CPU und der I/O-Controler haben nur miteinander zu tun, wenn die CPU die Systembuskontrolle an den I/O-Controler auf dessen Anfrage hin abgeben muß.
- E/A-Prozessoren (input-output processor, **IOP**, channel): Ein IOP kann die gesamte Kontrolle für Datenübertragungen zwischen einem oder mehreren Geräten und dem Hauptspeicher übernehmen, ohne Rückgriff auf die CPU. Ein IOP kann I/O-Programme direkt ausführen. Diese Programme können spezielle I/O-Instruktionen, die nicht im Instructionset der CPU vorkommen, anwenden. Gewöhnlich wird ein I/O-Prozessor über einen separaten I/O-Bus (I/O-Interface) mit den Geräten, die er kontrolliert, verbunden.

Ein DMA-Controler und ein IOP können mit einem **Interrupt** die CPU unterbrechen. Das befreit die CPU davon, periodisch einen I/O-Gerätestatus zu testen. Im Gegensatz zu einer DMA-Anfrage (request) wechselt die CPU sofort ihr Programm und führt eine Interruptroutine aus. Danach kann sie das unterbrochene Programm weiter ausführen.

6.2 Programmierte E/A

Typischerweise findet der Transfer zwischen einem CPU-Register (z.B. accumulator) und einem Pufferegister des I/O-Gerätes statt. Das Gerät hat keinen direkten Zugriff zum Hauptspeicher. Ein Datentransfer vom Gerät zum Hauptspeicher benötigt mehrere CPU-Befehle: Inputinstruktion, um ein Wort vom Gerät zur CPU zu übertragen, Speicherbefehl, um das Wort von der CPU zum Hauptspeicher zu bringen. Weitere Befehle sind nötig zur Adreßberechnung und zum Datenwortzählen.

I/O-Adressierung

I/O-Geräte, Hauptspeicher und CPU kommunizieren über den Systembus. Jeder Anschluß des Systembus an ein I/O-Gerät wird **I/O-Port** genannt. Er hat eine eigene Adresse. Die I/O-Geräte werden wie Hauptspeicherstellen über die Adreßlines des Busses adressiert. Der I/O-Port hat ein eigenes Datenpufferregister, wodurch er sich aus CPUsicht vom Hauptspeicher unterscheidet.

Es gibt zwei Adressierungsmethoden:

- Memory-mapped: Hierbei sind die I/O-Ports im Hauptspeicheradreßraum enthalten. Daher keine speziellen I/O-Befehle.
- I/O-mapped (eigener Adreßraum): Separate Instruktionen zum Schreiben und Lesen von I/O-Ports. I/O-Gerät und Hauptspeicherstelle können dieselbe Adresse haben. Der Hauptspeicher- und I/O-Adreßraum sind also getrennt.

I/O-Befehle

Die einfachste Möglichkeit kann durch zwei Befehle implementiert werden: IN X, d.h. Wort von Port X zum Akku, und OUT X, d.h. Wort vom Akku zum Port X. Da die CPU keine Bedeutung diesen Worten gibt, muß der Programmieren unterscheiden zwischen Daten, Steuerinformationen (Lesekopfpositionierung) und Statusinfos (Lesefehler z.B.). Üblicherweise liefert das E/A-Gerät kontinuierlich eine Statusinformation. Die CPU liest diese und testet, ob der Status anzeigt, daß das Gerät bereit ist für einen Transfer. Falls nicht bereit, dann wieder den Status lesen.

6.3 Interrupts

Das Problem ist die Rechenzeitverschwendung der CPU in der Abfrageschleife bzgl. des Gerätestatus. Besser ist es, wenn das Gerät sich selbst meldet, wenn sich sein Status ändert. Darauf muß der Prozessor irgendwie reagieren.

Ein **Interrupt** ist ein (selten und unerwartet) auftretendes Ereignis, daß über spezielle Kontroll-lines vom Gerät zur CPU übertragen wird. Der Prozessor wird dadurch gezwungen, sein aktuell bearbeitetes Programm zu unterbrechen und stattdessen ein spezielles Programm zur Bedienung des Interrupts auszuführen.

Methode: Prozessor besitzt ein von extern setzbares **Interrupt-Flipflop**. Will ein Gerät den Prozessor unterbrechen, so setzt es das FF. Einmal pro Instruktionszyklus (üblicherweise vor der Fetch-Phase) testet der Prozessor das Interrupt-FF. Ist das IFF gesetzt, so führt der Prozessor einen Unterprogrammsprung durch. (Retten des alten Zustandes, Sprung an Start der **Interruptroutine**). Diese Reihenfolge ist zwar typisch, aber nicht notwendig. Es ist auch möglich, daß nur eine einzige Interruptserviceroutine existiert, die ihrerseits auswählt, wohin sie verzweigt.

Maskierung bedeutet, daß z.Z. keine Interrupts möglich sind. Nötig z.B. für zeitkritische Abschnitte. Dafür gibt es eine eigenen CPU-Befehl: disable interrupts. Oder man disabled alle Interrupts geringerer Priorität, um einen wichtigen Interrupt bedienen zu können. Danach muß aber ein interrupt enable erfolgen, um den niederwertigen Interrupts Zugriff zur CPU zu geben.

Identifikation des Unterbrechnungsgrundes (Wer braucht Service?):

- **Single-line** Methode: 1 IFF für alle Geräte gemeinsam. Prozessor fragt alle Geräte in gewisser Reihenfolge ab, ob es Interrupt ausgelöst hat. Erstes solches Gerät wird bedient. Priorität durch Abfragereihenfolge. Geringster Hardwareaufwand.
- Eine Variante davon ist, daß der Prozessor eine acknowledge-Signal (Quittung) generiert, was per daisy-chain durch alle Geräte durchgeschleift wird. Erstes Gerät, das unterbrochen hat und dieses Signal erhält, reagiert (legt etwas auf den Datenbus) und reicht acknowledge nicht weiter. Priorität durch Reihenfolge der Geräte in der daisy chain.
- **Multiple-line** oder **multilevel**-Interrupts: 1 IFF pro Gerät. Dadurch ist die Interruptquelle sofort bekannt für die CPU.
- Mischlösung: Multilevel, aber pro IFF eine daisy chain. Das ist die heutige Standardmethode.

Adresse der benötigten ISR (Was ist zu tun?):

- Eine einzige ISR und daher feste Adresse.
- **Vectored interrupt**: Das Peripheriegerät legt die nötige Sprungadresse auf den Datenbus. Der Vektor ist die Adresse der ISR.

Nach der Identifizierung und dem Erhalt der Adresse Unterprogrammsprung zur richtigen Adresse. Ausführung der ISR und Rücksetzen des IFF. Dann Rücksprung in unterbrochenes Programm.

6.4 DMA (Direct Memory Access)

Grundidee hierbei ist, daß der Prozessor die E/A startet und die DMA-Einheit dann autonom arbeitet. Der Prozessor teilt der DMA-Einheit die Startadresse des Datenbereichs im Hauptspeicher mit und die Anzahl der zu übertragenden Worte. Danach arbeitet die DMA-Einheit autonom, bis der gesamte Block übertragen ist. Anschließend gibt sie per Interrupt eine Rückmeldung an die CPU. Ein IO-Gerät ist über einen DMA-Controler an den Systembus angeschlossen. Der Controler enthält ein Datenpufferregister (wie beim programmierten IO) und Adreßregister und ein Datenzählregister.

Ein DMA-Transfer sieht so aus:

1. Die CPU führt zwei I/O-Instruktionen an DMA aus: Die eine lädt das DMA-Register IOAR mit der Startadresse des Datenbereichs im Hauptspeicher und die andere speichert im DMA-Register DC die Blocklänge.
2. Wenn das Gerät übertragungsbereit ist setzt es die DMA-request line zur CPU. Die CPU gibt bei nächster Möglichkeit Daten- und Adreßbus frei und aktiviert DMA-acknowledge line.
3. Der DMA-Controler überträgt nun die Daten direkt zum oder vom Hauptspeicher und inkrementiert bzw. dekrementiert nach jedem Wort seine beiden Register um 1.
4. Falls DC nicht auf 0 dekrementiert ist, aber das Gerät nicht mehr übertragungsbereit ist übergibt der DMA-Controler die Kontrolle wieder an die CPU durch Freigabe der Busse und Deaktivieren der DMA-request line. Die CPU antwortet durch Deaktivieren der DMA-acknowledge line. Sobald das Gerät wieder bereit ist, weiter bei 2.
5. Wenn DC 0 ist gibt das Gerät eine Rückmeldung an die CPU (interrupt). Die CPU beendet E/A oder startet neuen DMA-Transfer.

Offensichtlich kann immer nur der DMA Controler oder die CPU aktueller Busmaster sein. Der jeweils andere hat dann keine Möglichkeit den Bus für sich zu nutzen:

- **Blocktransfer**: Übertragung an einem Stück hat den höchsten Durchsatz. Jedoch kann dadurch die CPU zum Warten gezwungen sein, weil sie nicht auf den Hauptspeicher zugreifen kann.
- **Cycle stealing**: Hier muß die Buskontrolle nach jedem übertragenem Wort zurückgegeben werden. Die I/O-Datenübertragung läuft dann verzahnt mit CPU-Transaktionen ab. Weniger Performance, dafür wird die CPU weniger gestört in ihren Aktivitäten. Die DMA-Einheit darf jedoch jeweils für ein oder wenige Worte stören.
- **Transparent Mode**: Man kann das DMA-Interface allerdings auch so gestalten, daß Buszyklen nur dann gestohlen werden, wenn die CPU den Systembus gerade nicht braucht.

6.5 E/A-Prozessoren (IOP)

DMA entlastet die CPU vom eigentlichen Datentransfer, jedoch nicht von der Gerätesteuerung. Ein E/A-Prozessor (oft **Kanal** genannt) übernimmt den gesamten E/A-Verkehr.

Die CPU kennt nur noch wenige E/A-Instruktionen wie z.B.: startio, stopio, testio. Der E/A-Prozessor (oft normaler Prozessor) führt Programme aus, die im Hauptspeicher stehen. Die IOP-Programme sind gerätespezifisch (device driver).

Typische IOP-Befehle:

- Transfer (Gerät, Speicheradresse, Blocklänge, Richtung)
- Gerätesteuerung (rewind, seek adress)
- Sprünge
- Arithmetik (zur Berechnung von Adressen und Geräteprioritäten)

Es gibt verschiedene Anschlußkonzepte zwischen IOPs und den Geräten. Meistens sind mehrere Geräte über einen I/O-Bus mit einem bestimmten IOP verbunden. Beim **Kreuzschienenverteiler** (crossbar-switching) kann jedoch jeder IOP mit jedem Gerät in Aktion treten. Das ist effizienter in einigen Fällen.

IOPs sind hauptsächlich Kommunikationsverbindung zwischen IO-Geräten und Hauptspeicher. Die CPU führt eine kleine Anzahl von IO-Befehlen durch, die es ihr erlauben, die Ausführung von IO-Programmen durch den IOP zu starten und zu beenden, ferner, um den Status des IO-Systems zu testen. Die CPU führt keine IO-Datentransferbefehle aus. Solche Befehle stehen in IO-Programmen im Hauptspeicher und werden von IOPs ausgeführt. Befehle, die von IOPs ausgeführt werden, haben primär mit Datentransfer zu tun. Ein typischer ist READ (WRITE) Daten-

block mit n Worten vom (auf) Gerät X in (aus) Hauptspeicherregion Y. Der IOP ist mir DMA ausgestattet.

Wie DMA-Controler hat der IOP die Fähigkeit, mehrere Datentransferoperationen mit verschiedenen Hauptspeicherregionen und verschiedenen IO-Geräten durchzuführen, ohne CPU. Der IOP hat darüber hinaus im Instructionset noch arithmetische, logische und Sprungbefehle, um die Berechnung komplexer Adressen oder von Geräteprioritäten zu ermöglichen. Eine weitere Instruktionskategorie jenseits von DMA-Controlern sind gerätespezifische Kontrollbefehle.

7. Entwurfsprinzipien

7.1 Bauelemente der Gatterebene

7.1.1 Kombinatorische Bauelemente: Boolesche Gatter

Gatter bzw. **gates** bezeichnen boolesche Bauelemente, die Funktionen wie AND oder NOR etc. realisieren. Kombinatorische (Schalt-) Funktionen sind boolesche Funktionen. Sie lassen sich durch Wahrheitstabellen darstellen. Sie beziehen die Zeit nicht ein und stehen folglich im Kontrast zu sequentiellen Schaltfunktionen, in denen die Ein- und Ausgaben Funktionen der Zeit sind.

7.1.2 Elementare Informationsspeicher: Flipflops

Flipflops sind elementare Informationsspeicher, die ein Bit speichern können. Im wesentlichen handelt es sich um RS-Flipflops aus zwei wechselseitig rückgekoppelten NOR-Gattern. Ein JK-Flipflop hat einen Clockeingang, bei dessen positiver Flanke es die Eingänge übernimmt und bei dessen negativer Flanke es seine Ausgänge aktualisiert. JK sind die wesentlichen Eingänge. 10 setzt den Ausgang. 01 negiert ihn, 00 tut nichts und 11 invertiert. Der Ausgang hat noch einen zweiten inversen Ausgang. Ferner gibt es ein Preset (setzt Ausgang) und einen Cleareingang (setzt Ausgang 0).

Es gibt sequentielle (**sequential**) und kombinatorische (**combinational**) **Schaltkreise (circuits)**. Sequentielle können im Gegensatz zu kombinatorischen Infor-

mationen speichern. Ihre Werte sind also zeitabhängig. Ein sequentieller Schaltkreis kann durch das **Huffman Modell** dargestellt werden, was aus einem kombinatorischen Schaltkreis und einem Speicher besteht. Der Schaltkreis darin liefert und bekommt Werte an/vom Speicher. Außerdem hat er externe Ein- und Ausgaben.

Sequentielle Schaltkreise sind logische Schaltkreise, deren aktuellen Outputs von vergangenen Inputs abhängen.

Um unvorhersagbares Verhalten zu vermeiden, wenn die Signale verschieden schnell die sequentielle Schaltung durchlaufen und die schnellsten erneut als Input anliegen, während die langsameren noch nicht durchgelaufen sind, versieht man Flipflops u.ä. (CPU) mit einem Takt. Die **Clock** ist dazu da, dem kombinatorischen der Schaltung (Huffman) zu erlauben, auf eine Inputmenge zu antworten und eine unzweideutige Menge von Outputs und Statuswerten (im Speicher) zu erzeugen. Das high-Signal muß dabei kürzer als die Signallaufzeit im kombinatorischen Teil sein, da sonst erneut Inputs übernommen werden. Das low-Signal muß lang genug sein, daß der komb. Teil eine stabile Ausgabe liefern kann. Das ist der sog. einfache Takt.

Bei einem Master-Slave-Takt werden zwei einfach getaktete Flipflops in Reihe geschaltet, so daß das eine enabled wird während das andere disabled ist. Inverse Taktung. Damit wird sichergestellt, daß sowohl der Eingang als auch der Ausgang nur zu bestimmten Zeiten übernommen/geändert wird.

7.2 Bauelemente der Register Transfer Ebene (RTE)

Der nächsthöhere Level nach dem gatelevel in der Computerdesignhierachie ist der Registerlevel. In Zusammenhang stehende Bits werden in geordnete Mengen gruppiert: Worte oder Vektoren. Die daraus resultierenden komplexeren Elementarbausteine sind kombinatorische oder sequentielle Schaltungen zum Verarbeiten oder Speichern von Worten.

Die entscheidende sequentielle Komponente, von der dieser Komplexitätslevel seinen Namen hat, sind (parallele) Register, das sind Speicher für Worte.

TABELLE 6: Komponenten auf Register Transfer Ebene

Typ	Komponente	Funktionen
Kombinatorisch	Wortgatter	Boolesche Operationen
	Multiplexer	Datenrouter, generelle Funktionen erzeugen
	Decoder/Encoder	Code überprüfen und konvertieren
	Programable Arrays	generelle Funktionen erzeugen
	Arithmetische Elemente (Addierer, ALUs)	Numerische Operationen
Sequentiell	(Parallele) Register	Informationsspeicher
	Shift Register	Informationsspeicher; seriell-parallel Umwandlung
	Zähler	Kontroll/Timingsignal-Erzeugung

7.2.1 Kombinatorische Bauelemente der RT-Ebene

7.2.1.1 Wort-Gatter

Es ist oft nützlich, auf zwei m bit langen Worten Gatteroperationen (AND, OR, etc.) durchzuführen, um einen ebenfalls m bittigen Ergebnisvektor zu bekommen. Die Entsprechenden Gatter haben zweimal m Eingänge und m Ausgänge. Die Verknüpfung erfolgt dann paarweise intern m mal.

7.2.1.2 Multiplexer

Ein k-Eingänge, m-Bit Multiplexer verbindet einen der k-Eingänge (je m bit breit) mit seinem m bit breiten Ausgang. Dazu dienen k Adreßleitungen.

Multiplexer können baumartig kaskadiert werden, so daß mit q Schichten von k-Eingänge Multiplexern ein k^q-Eingänge Multiplexer erreicht wird. Dabei enthält jede höhere Schicht die doppelte Anzahl Multiplexer wie die darunterliegende. Die unterste hat einen einzigen.

Beim **kodierten Multiplexer** werden nicht wie bisher die Adressen im 1 aus k-Code bestimmt sondern man interpretiert die k Adreßleitungen als unsigned integer. Damit hat man einen 2^k-Eingänge Multiplexer.

Mit einem 2^n-Eingänge 1-bit Multiplexer läßt sich jede n-stellige boolesche Funktion realisieren, indem man durch Anlegen geeigneter Adreßwerte die jeweilige Funktion auswählt.

7.2.1.3 Decoder

Ein 1-aus-2^n oder $1/2^n$ Dekoder ist ein kombinatorischer Schaltkreis mit n Inputleitungen und 2^n Outputleitungen, so daß jede der 2^n möglichen Inputkombinationen exakt eine der Outputleitungen aktiviert.

Die primäre Anwendung des Dekoders ist die Adressierung, wobei der n-bit Input als eine Adresse interpretiert wird, um eine der 2^n Outputleitungen auszuwählen.

7.2.1.4 Encoder

Ein Encoder ist ein Schaltkreis, der dazu benutzt wird, Adresse oder Name einer aktivierten Inputleitung zu generieren; daher ist er das Inverse zum Dekoder. Ein typischer Encoder hat 2^n Inputleitungen und n Outputadreßleitungen. Es wird ein zusätzlicher Ausgang benötigt, um anzuzeigen, ob überhaupt kein Eingang aktiv ist, damit man den 0-Fall unterscheiden kann, der entweder heißt, daß Eingang 0 aktiv ist oder kein Eingang aktiv ist.

Ein **priority-encoder** löst das Problem, daß mehr als eine Eingabeleitung aktiv seien könnten. Es wird einfach nur die rechteste aktiver Eingabeleitung betrachtet.

7.2.1.5 PLA (Programable Logic Arrays)

Ein PLA ist ein zweiteilige Matrix aus einfachen Schaltelementen. Es handelt sich um die Implementierung einer DNF-Formel, die beliebig ausgewertet werden kann. Man stellt für jede Outputvariable (boolean) eine DNF-Formel der Inputs auf. Dann sieht man nach, welche UND-Terme benutzt werden insgesamt. Die UND-Terme werden in einer UND-Ebene implementiert, so daß jede Spalte einer Input-Variablen zugeordnet ist und in den Zeilen (eine pro UND-Term) die verschiedenen Variablen durch -1 als negiert enthalten, 0 als nicht enthalten und +1 als enthalten in die Terme „eingesetzt“ werden. In einer OR-Ebene werden dann diese in Zeilen angeordneten UND-Terme mit jeweils den Spalten oder-verknüpft, die einer Outputvariablen zugeordnet sind, deren DNF-Formel den Term enthält.

Diese Methode ist ziemlich effizient, da jeder Term nur einmal implementiert wird. Die Arrayelemente -1, 0 und +1 können durch einfache Gatter erzeugt werden.

In der Vorlesung wurde eine PLA-Implementierung eines Volladdieres gezeigt.

7.2.1.6 Arithmetische Elemente

Einige sehr simple arithmetische Funktionen, wie z.B. Addition und Subtraktion von Festkommazahlen, können durch kombinatorische RT-Level Komponenten implementiert werden. Die meisten Formen von Festkommamultiplikation und -division und alle Fließkommaoperationen sind zu komplex, um durch eine einzelne Komponente auf diesem Designlevel realisiert zu werden.

7.2.2 Sequentielle Bauelemente der RT-Ebene

7.2.2.1 Register, Schieberegister

Ein m-bit Register ist eine geordnete Menge von m Flipflops, um ein m-bit Wort zu speichern. Jedes Bit des Wortes wird in einem separaten Flipflop gespeichert. Die Datenleitungen sind unabhängig. Daten können von oder zu allen Flipflops gleichzeitig übertragen werden; parallel input-output. Da die gespeicherte Information als eine Einheit betrachtet wird, sind die Kontrollsignale (clock, preset, clear) für alle Flipflops im Register gemeinsam. Register können mit einem beliebigen Flipfloptyp konstruiert werden; bevorzugt werden jedoch Master-Slaves.

Ein Register, das links- oder rechts-shift zuläßt wird Schieberegister genannt. Diese werden für das Speichern serieller Daten („Reinschieben“), seriell-parallel-Umwandlung (Rein- oder Rausschieben in Verbindung mit parallelem Lesen/ Schreiben) und für arithmetische Operationen (Verschieben bedeutet Exponentenänderung) genutzt. Dies läßt sich leicht zum Rotieren weiterentwickeln: Verbinden von seriellem input und seriellem output.

7.2.2.2 Zähler

Ein Zähler ist eine einfache sequentielle Maschine, um durch eine vorbestimmte Sequenz von verschiedenen Zuständen zu zirkulieren als Reaktion auf Impulse einer Inputleitung.

Die einfachsten Zähler erhält man durch kleine Veränderungen ordinärer (shift) Register. Sie bestehen aus einer Anzahl JK-Flipflops, deren Ausgänge Binärstellen repräsentieren und mit Eingängen der andern verbunden sein können.

Zähler sind im Grunde genommen seriell-input parallel-output Geräte. Sie haben zwei hauptsächliche Anwendungen im Computerbau:

1. Die Speicherung des Status einer Kontrolleinheit, z.B. einen Programmzähler.
2. Erzeugung von Timingsignalen. Mehrere hintereinander geschaltete JK-Flipflops wie beim modulo-16-Zähler wirken als Frequenzteiler, da jeweils nur jedes zweite Signal „weitergegeben“ wird.

7.2.2.3 Busse

Ein Bus ist eine Menge von Leitungen zum Transportieren aller Bits eines n-bit Wortes von einer Quelle zu einem Ziel, welche typischerweise Register sind. Obwohl Busse keine logischen Funktionen erfüllen, erzeugen sie doch signifikante Kosten, da sie gewöhnlich logische Schaltungen benötigen, um den Zugriff auf sie zu kontrollieren, ebenso wie Signalverstärkungsschaltungen (Bustreiber und Busempfänger).

Ein Bus hat häufig einige Attribute mit Registern gemeinsam. Er kann Daten temporär speichern, möglicherweise in Pufferregistern die ihm beigeordnet sind. Er kann einen Namen bekommen und durch Instruktionen adressiert werden in der selben Weise wie ein Register oder eine Speicherstelle. Der Transfer zwischen zwei Registern über den Bus ist oft in zwei „Register“-Transfers aufgeteilt: Databus <- Register a; Register b <- Databus.

7.3 Steuereinheiten

Ein digitales System wird in Datenmanipulation (gesteuerter Teil) und in Steuerung eingeteilt. Wie wird eine Steuerung realisiert?

7.3.1 Fest verdrahtete Steuerungen

Fest verdrahtete Schaltungen können durch

- einen endlichen Automaten **(FSM)**,
- Verzögerungsglieder oder
- Zähler realisiert werden.

Die Alternativen bei der Realisierung eines FSM nach Huffman Modell: Kombinatorischer Teil entweder aus Gatterlogik oder PLA.

Eine **Steuerung** stellt immer einen **endlichen Automaten** dar. Sequentielle Schaltungen, die beim Rechnerentwurf verwendet werden, haben eine Struktur, die durch das generelle **Huffman** Modell repräsentiert werden kann. Der sequentielle Schaltkreis wird aus zwei Teilen bestehend interpretiert: Einem kombinatorischen Schaltkreis C und einem Speicher M. Die in M gespeicherte Information wird interner Zustand (internal state) genannt. Die Menge aller Signalwerte im Schaltkreis zu einem Zeitpunkt wird Gesamtzustand (total state) genannt. Jeder primäre Output z, generiert von C, ist eine Funktion der primären Inputs x und der sekundären Inputs (Statusvariablen, Inhalt von M) y. Der sekundäre Output Y von C bestimmt die neuen Werte (next states) von y (Inhalt von M) und ermöglicht der Schaltung, Informationen, die von früheren Inputs hergeleitet ist, im Gedächtnis z zu behalten. Daher hängt der primäre Output nicht nur von den aktuellen primären Inputs ab, sondern auch von einer Sequenz früherer Inputs.

Beim **Mealy-Automaten** hängt der Output von den Eingaben von außen und vom alten Zustand ab.

Ein endlicher Automat $A = (X, Y, S, \delta, \lambda)$ nach Mealy ist:

X = Wertemenge auf Testleitungen (primärer Input)
Y = Wertemenge auf Steuerleitungen (primärer Output)
S = Zustandsmenge
$\delta = X \times S \rightarrow S$ = Zustandsüberführung
$\lambda = X \times S \rightarrow Y$ = Ausgabefunktion

Beim **Moore-Automaten** bestimmt sich der Output (Steuerung, Y) nur durch den aktuellen Zustand, d.h. $\lambda = S \rightarrow Y$ = Ausgabefunktion. Der neue Zustand wird jedoch auch bei diesem Automaten aufgrund des alten Zustandes und den Inputs von außen bestimmt.

Beide Automatenmodelle sind gleich mächtig. Ein Moore-Automat muß jedoch evtl. für die Berechnung derselben Ausgabefolge mehr verschiedene Zustände ver-

wenden. Der Unterschied wird in der Huffman-Darstellung sichtbar, wenn man die next-state- und die output-function des kombinatorischen Teils getrennt zeichnet: Bei beiden Automaten ist die next-state-function mit primärem (außen) und sekundärem (alter Zustand) Input verbunden. Die output-function beim Mealy-Automaten erhält den neuen Zustand und den primären Input. Beim Moore-Automaten jedoch erhält die output-function nur den neuen Zustand.

7.3.2 Mikroprogrammierung

D.h. wie wird der kombinatorische Teil realisiert? Es müssen die Zustandsübergangsfunktion und die Outputfunktion berechnet werden. Anstatt einer Berechnung Tabellierung um Speicher. Ersetzt man die AND-Plane eines PLA durch einen Dekoder, der genau (per Def.) eine (ehemalige Produktterm-) Leitung pro Bitkombination der Eingänge auswählt, wird aus dem PLA ein **ROM**. Damit werden die Funktionen (Werte auf Leitungen, die die OR-Plane verlassen) nicht mehr berechnet (UND + ODER), sondern tabelliert, und die Funktionswerte werden einfach durch die Werte der Argumente adressiert. Die Ausgabe ist dann (Dekoder setzt Leitung j) bei Ausgabebit i gleich 1, falls (j,i) + ist, sonst 0.

Ein Wort im **Steuerspeicher**, d.h. der Wert aller „Funktionen“ beim selben Argumentwert, heißt **Mikroinstruktion**. Mikroinstruktion (Speicherwort) enthält

- das Steuerfeld (Ausgabe) und
- das Folgeadreßfeld (Folgezustand d.h. Adresse der nächsten Mikroinstruktion).

Also liefert das Zustandsregister (M in Huffman-Modell, als Eingabe) die Adresse des Speicherworts (Mikroinstruktion). Falls Folgezustand von externer Eingabe abhängt, dann nachträgliche Modifikation vom Folgezustand. Abhängigkeit der primären Ausgabe nicht nötig von primärer Eingabe laut Moore-Automat.

Typischerweise enthält ein Steuerspeicher für jede Instruktion des zu implementierenden Prozessors ein **Mikroprogramm**. Jedes Mikroprogramm beginnt an einer Adresse, die aus dem Opcodefeld abgeleitet werden kann.

Wenn für jedes Bit des Steuerfeldes (1. Teil der Mikroinstruktion) eine Kontrolleitung existiert spricht man von **horizontaler Mikro-Programmierung**. Hierbei ist der Parallelitätsgrad bzgl. der Operationen eines Steuerwerks am größten, da alle möglichen Operationen gleichzeitig ausgeführt werden können.

Nun sind evtl. einige Operationen inkompatibel wie z.B. das Laden eines Busses von verschiedenen Quellen. Um die gleichzeitige Ausführung solcher Operationen zu verhindern und um Platz im Steuerspeicher zu sparen, kodiert man inkompatible Operationen in dieselbe (Klasse) Zone des Steuerfeldes.

Wenn z.B. die ersten vier Steuerleitungen inkompatibel sind, dann braucht man nicht vier Bits dafür wie bisher in der Ausgabe (Steuerfeld), von denen jedes Bit für jeweils eine Operation (ausführen oder nicht) bestimmt ist, sondern nur noch zwei Bits, die die Kodierung der vier Möglichkeiten als Binärzahl darstellen. Ein pro Zone zwischen Steuerfeld und entsprechenden Steuerleitungen gesetzter Dekoder aktiviert dann die entsprechende Leitung. Somit kann nur jeweils eine der inkompatiblen Operationen ausgeführt werden, da sie über denselben Dekoder fließt. Dekoder haben ja bekanntlich die Eigenschaft, daß sie nur einen ihrer Ausgänge wählen. Sogenannte **zonenorientierte Mikro-Programmierung**.

Im extremsten Fall gibt es nur eine einzige Klasse. Hier spricht man dann von **vertikaler Mikro-Programmierung**.

Da die zonenorientierte Variante einen Kompromiß zwischen vertikal und horizontal darstellt, nennt man sie auch **diagonal**.

Die Unterscheidung bzgl. der Struktur des Steuerwortes wird nicht nur bei der Mikroprogrammierung verwendet, sondern auch bei anderen Arten des Steuerwerksentwurfs.

Adressierung von Mikroinstruktionen

Beim **Wilkes** design enthält jede Mikroinstruktion die CM-(Control Memory, Steuerspeicher) Adresse der nächsten auszuführenden Mikroinstruktion, s.o.. Im Fall von Verzweigungsmikroinstruktionen sind zwei mögliche Folgeadressen enthalten. Dies hat den Vorteil, daß keine Zeit verloren wird mit der Adreßgenerierung, ist jedoch CM-verschwenderisch.

Die Adreßfelder können bei allen Instruktionen, *außer Verzweigungsanweisungen*, durch Einführung eines **Mikroprogrammzählers** eliminiert werden, der die primäre Quelle von Mikroinstruktionsadressen bildet. Seine Rolle ist analog zu der des Programmzählers auf der Befehlsebene. Da nur Instruktionen aus dem CM geholt werden müssen, ist der µPC auch als CM-Adreßregister (CMAR) in Gebrauch.

Bedingte Sprünge sind auf verschiedene Weisen implementiert. Die zu testende Bedingung ist generell eine Bedingungsvariable oder Flag, die von der data processing unit (Datenmanipulationsteil) erzeugt wird. Falls mehrere solcher Bedingungen existieren, dann enthält die Mikroinstruktion ein condition select subfield, das anzeigt, welche der möglichen Bedingungsvariablen getestet werden muß. Die Verzweigungsadresse kann

- in der Mikroinstruktion selbst enthalten sein. Dann wird die Adresse ins CM Adreßregister geladen, sobald eine Verzweigungsbedingung erfüllt ist.
- Wenn nicht die ganze Adresse in der Instruktion steht, sondern nur z.B. einige low-order Bits davon und der Rest vom Datenmanipulationsteil gesetzt wird, kann CM-Speicher gespart werden.
- Ein anderer Ansatz besteht darin, daß die Verzweigungsvariablen (gesetzt vom Datenmanipulationsteil) den Inhalt des CM Adreßregisters (µPC) direkt zu modifizieren. Dadurch wird der Adreßteil in Mikroinstruktionen komplett beseitigt. Z.B. kann ein overflow-flag mit dem count-enable input des µPC verbunden sein, dadurch „skip“.

Timing von Mikrooperationen

Bisher wurde angenommen, daß jede Mikroinstruktion Kontrollsignale aktiviert, die für die Dauer des Mikroinstruktionsausführungszyklus aktiv sind. Eine Clock synchronisiert alle Kontrollsignale; die Clockperiode kann dieselbe sein wie die Mikroinstruktionszyklusperiode. Dies wird **Monophase** (Eine Gruppe von Kontrollsignalen pro µInstruktion.) genannt.

Die Anzahl der Mikroinstruktionen, um eine Aufgabe zu erledigen, kann meist durch Teilung des Mikroinstruktionszyklus in verschiedene sequentielle Phasen verringert werden. Jedes Kontrollsignal ist typischerweise nur während einer dieser Phasen aktiv. Dieser Modus, die **Polyphase** (Mehrere sequentielle Gruppen von Kontrollsignalen pro µInstruktion.), erlaubt einer einzelnen µInstruktion, eine Sequenz von Mikrooperationen zu bestimmen. Die Komplexität des µInstruktionsformates steigt natürlich dadurch, weil es nötig ist, die Phasen, während der ein Kontrollsignal aktiviert werden soll, zu spezifizieren.

Z.B. kann eine Registertransferaufgabe, bei der R das Ergebnis einer Funktion f angewendet auf R_1 und R_2 erhalten soll. Dies kann in vier Phasen erledigt werden:

- Inhalt von R_1 und R_2 zu den Inputs der f-Unit übertragen.
- Ergebnis der f-Unit in temporäres Register L speichern.

- Inhalt von L zum Zielregister R übertragen.
- Nächste Mikroinstruktion vom CM holen.

Diese vier Aktionen müßten auf vier einzelne µInstruktionen aufgeteilt werden, falls alle Kontrolleitungen mit der µInstruktion die Clock gemeinsam hätten. Denn dann müßten sie gleichzeitig ausgeführt werden, was nicht geht. Und sequentiell würde mehrere µInstruktionen erfordern. Daher ist die Polyphase günstiger, da die Ladezeit einer Mikroinstruktion aus dem CM hoch ist.

Eine weitere Beschleunigung ist möglich, wenn man eine Registerpipeline anstelle des µIR verwendet, die zwei Register enthält: Das vordere wird schon mit der folgenden Instruktion geladen, während das letzte die aktuell in der Ausführung stekkende µInstruktion enthält.

8. CISC- und RISC-Architekturen

Ein CISC Computer ist darauf optimiert, die Anzahl der auszuführenden Instruktionen zu reduzieren, indem er komplexe Instruktionen zur Verfügung stellt.

Ein RISC Rechner ist darauf optimiert, die Anzahl der Zyklen pro Instruktion (Anzahl der µInstruktionen pro Instruktion) zu reduzieren und die Zykluszeit (Dauer einer µInstruktion) gleichzeitig kurz zu halten.

8.1 Geschichte und Entwicklung CISC

Da damals Speicher eine teure Ressource war, galten die Prozessoren als gut, die kurze Programme ermöglichten. Außerdem sollte durch Einführung der Mikroprogrammierung der zeitaufwendige Speicherverkehr gedrosselt werden, indem anstatt vieler Instruktionen eine einzige geholt wird, die alle umfaßt und dann per Mikroprogramm wieder aufgebröselt wird. Ferner wird durch eine durch CISC „höhere“ Assemblersprache das Schreiben von Compilern erleichtert, da eine High-Level-Language-Instruktion einem CISC-Befehl fast entspricht.

Um mit neueren Prozessoren kompatibel zu bleiben mit den Vorgängern, müssen alte Instruktionen beibehalten werden. Dadurch wächst das Instructionset im Laufe der Entwicklung ziemlich. (Mein liebstes Negativbeispiel: alle 80X86 d.h. auch Pentium und PentiumPro etc. sind kompatibel zum uralten 8088, was ein 8-bit Prozessor war.) Je mehr man berücksichtigen muß, umso schwerer ist ein optimales Design.

Durch verschiedene diverse Instruktionen werden variable Instruktionslängen nötig. Dadurch starten Instruktionen an verschiedenen byte- oder sogar bit-Stellen, was die Leistung von instruction fetch units oder memory data transfers nicht effizienter macht. So kann z.B. ein page fault durch einen Operanden entstehen, nachdem schon irreversible Aktionen durch die Instruktion geschahen. Das ist ‘ne Vollbremsung des Prozessors.

8.2 RISC-Prinzipien

CISC-Rechner haben mit folgenden Restriktionen Probleme:

- Chipfläche
- Designzeit
- Register

Die Auswertung von Programmausführungen hat zum RISC-Ansatz geführt, der Tradeoffs zwischen Architektur und Implementation, Hardware und Software, Compile- und Laufzeit, macht, um eine optimale Leistung zu erreichen. Dies führte zu einer Computerarchitektur mit weniger Befehlen und schnellerer Ausführung (mindestens so schnell wie eine CISC-Mikroinstruktion). Die Architektur hat auch ein besseres Compilerinterface (im Gegensatz zum benutzerfreundlichen Interface). „Reduzierter Instruktionssatz“ beschreibt nur einen Teilaspekt. „Stromlinienförmig“ ist treffender.

Die RISC-Philosophie basiert auf folgenden Beobachtungen:

- Simple Operationen: Komplexe werden interpretiert.
- Simple Adressierungsmodi.
- Kleine Konstanten- Displacementwerte.
- Wenige Datentypen: Fehlende emulieren.
- Prozeduraufrufe häufig.
- Wenige Parameter meistens.
- Lokale Variablen meist wenige.
- Compiler können komplexe Instruktionen schlecht ausnutzen.
- Komplexe Instruktion oft langsamer als mehrere simple als Ersatz.

und Grundsätzen:

- Simple Instruktionen für schnelle Ausführung, ein Zyklus.
- Zykluszeit kurz, d.h. einfacher Datenpfad.
- Kurze Operandenzugriffe durch nur Registerbefehle.

- Viele Register zur Speicherung des Activationrecord.
- Nur load/store-Instruktionen bearbeiten Hauptspeicher.
- Feste Länge für Instruktionen, dadurch schneller fetch.
- Instruktionen mit simplem Layout und wenige Formate, Decodespeed.
- Simple Adreßmodi, schnelle Operandenadreßrechnung.
- Harvard Architektur.
- Pipelining.
- Compiler löst Pipelinekonflikte, Implementation auf Architekturebene also sichtbar.

8.3 RISC-Architektur Charakteristiken

Diese Architektur nutzt Compilertechniken, um die Architektur selbst zu vereinfachen und die Implementation zu beschleunigen. Natürlich können viele der RISC-Beschleunigungsverfahren auch für CISCs verwendet werden, jedoch ist der Vorteil der RISCs, daß sie hinsichtlich dieser Mechanismen entworfen wurden.

8.3.1 Register Windowing

Unterprogrammsprünge kommen häufig vor. Dabei Parameterübergabe. Speicherzugriffe sind teuer, d.h. bei RISC zwei Zyklen anstatt einem. Daher versucht man, Unterprogrammsprünge mit Parameterübergabe so zu realisieren, daß kein Speicherzugriff notwendig ist.

Also muß entfallen:

- Retten von Registersatz bei Unterprogrammsprung
- Parameterübergabe über den Speicher.

Lösung:

- Physikalisch mehrere Registersätze, eingeteilt in Registerfenster. Einem (Unter)programm wird genau ein Fenster zugewiesen. D.h. anstatt die Registerinhalte zu retten, wird der Fensterzeiger umgesetzt.
- Die Parameterübergabe erfolgt über sich überlappende Registerfenster, die als Parameterfelder verwendet werden.

Nun kann natürlich jedes Fenster belegt sein und man benötigt ein weiteres für ein neues Unterprogramm. Dem beugt man vor, indem jeweils ein Fenster in Reserve frei gehalten wird und die Fenster zyklische angeordnet werden.

8.3.2 Load-Store-Architektur

Eine Architektur heißt so, wenn es zwei spezielle Befehle gibt, die ein Register mit dem Hauptspeicher laden bzw. ein Register zum Hauptspeicher schreiben, und alle anderen Befehle nur mit Registern arbeiten.

8.3.3 Delayed Branch

Wenn ein Branchbefehl in der Pipeline ausgeführt wird, dann sie praktisch alle vorherigen Pipelinestufen zu invalidieren. Ein optimierender Compiler füllt die Pipeline nach dem Branchbefehl mit sinnvollen Instruktionen, die nicht ungültig werden. Ohne dieses Verfahren würde ein delay auftreten, bis die Pipeline wieder gefüllt wäre.

Beim delayed branch wird die *Ausführung eines geholten Branchbefehles modifiziert, so daß zusätzliche Befehle ausgeführt werden*, nachdem der Branch geholt ist.

Da dieses Verfahren auf Architekturebene nicht transparent ist, denn die CPU muß den Branchbefehl ja anders als ursprünglich behandeln, kann es nur bei neuen Architekturen mit entsprechendem Compiler, der dieses Funktion berücksichtigt, zur Geschwindigkeitssteigerung verwendet werden. Da keine Kompatibilität mit bestehenden Architekturen nötig war, nutzen nahezu alle RISCs diesen Mechanismus.

Sinnvoll bei Sprüngen und Interrupts. Sog. Steuerkonflikte.

8.3.4 Sichtbares Pipelining

Obiges entspricht im Prinzip dem sichtbaren Pipelining. Beim unsichtbaren Pipelining werden Befehle innerhalb der Pipeline nach dem Branch weggeworfen, um für den Programmierer unsichtbar zu sein. Beim sichtbaren Pipelining wird Befehl nach dem Sprung auch ausgeführt. Dafür ist ein optimierter Compiler nötig s.o.

Zur Erinnerung: beim Pipelining werden die sequentiellen Phasen des Befehlszyklus mit aufeinanderfolgenden Befehlen in jeweils verschiedenem Verarbeitungsstadium parallel genutzt. Während sich ein Befehl in der Holphase befindet werden andere dekodiert, Operanden geholt, Befehl ausgeführt, Ergebnis gespeichert. Superskalares Pipelining hieß, daß mehrere parallele Pipelines existieren.

Effizientes Pipelining durch

- homogenen Befehlssatz (identische Zeit in Ausführungsstufe)
- wenige, homogene Adreßmodi (leichtes Operandenbestimmen)
- seltene Speicherzugriffe (keine Buskonflikte).

Deshalb:

- Load/Store-Architektur
- viele Register
- einfache Befehle

8.3.4.1 Pipelinecharaktere

Eine Pipelinestruktur heißt **unifunktional**, falls alle Befehle durch genau ein Belegungsschema (benötigen pro Pipelinestufe diesselbe Hardware z.B. Speicher, Register Decoder, ALU) repräsentiert werden, **multifunktional** sonst.

Ein Pipelineschema heißt **linear**, falls jede Hardwareeinheit in genau einer Pipelinestufe benötigt wird.

8.3.4.2 Pipelinekonflikte

Falls die Pipeline nicht linear ist, können Lesen und Schreiben von Registern in verschiedenen Stufen zu Problemen führen. Da Zugriff einfach, kann das Schreiben und Lesen auf 2 Halbtakte aufgeteilt werden. Interne Taktverdopplung sozusagen.

Ein Schnittstellenkonflikt, der zwischen Befehlholen und Ergebnisdatenspeichern auftreten kann, kann durch die Harvardarchitektur gelöst werden. Eine Architektur heißt Harvard-Architektur, falls für Befehlsspeicher und Datenspeicher räumlich getrennte Daten- und Adreßbusse zur Verfügung stehen.

8.3.4.3 Datenkonflikte

Z.B. kann ein Register im nachfolgenden Befehl gelesen werden (Operanden holen), während es im vorherigen noch gar nicht geschrieben wurde (Ergebnisdaten speichern).

Lösungen:

- Software: Compiler erkennt Datenkonflikt und fügt NOPs ein oder sortiert die Befehle um. D.h. hier wird zwischendurch ein anderer Befehl eingefädelt.
- Hardware: Scoreboarding, d.h. für jedes Register wird Buch geführt, ob sich ein Befehl, der dieses Register benutzt, in der Pipeline befindet. Will weiterer Befehl auf das Register zugreifen, wird er verzögert. Nachteilig:
- Hardware: Forwarding (Bypass), d.h. durch zusätzliche Verbindungen steht das Ergebnis nicht erst nach der letzten Stufe, sondern direkt nach seiner Berechnung zur Verfügung für nächste Befehlsausführung.

8.4 Sonstige Anmerkungen

Durch die fest verdrahtete Implementation und die Einfachheit des instruction set ist die control area eines RISC 6% der Chipfläche, während bei konventionellen CISC mit 50% zu rechnen ist. Dadurch wurde bei RISC-Prozessoren u.a. auch das register file, welches zum Registerwindowing genutzt wird möglich.

Die Harvard Architektur mag sich auf das CPU-Cache-Interface beschränken, so daß es also zwei Cachespeicher gibt: einen für Daten, einen für Instruktionen.

Aufgrund der einfacheren Instruktionen tendieren RISC-Pipelines dazu weniger Stufen als vergleichbare CISCs zu enthalten, was die Heftigkeit von Pipelineproblemen reduziert.

> Zu Pipelinekonflikten bei zu großen Pipes:
> Die Pipeline des PentiumPro ist mit 12 Stufen die längste mir bekannte. Bei Befehlen, die Ladeoperationen enthalten, verlängert sie sich sogar auf mindestens 18 Takte. Von der Branch Prediction Unit falsch vorhergesagte Verzweigungen werden erst in Stufe 10 erkannt. Damit führt jede falsche Vorhersage zu mindestens 11 Straftakten.

Da ein RISC schneller entwickelt werden kann, ist seine Lebensdauer ungefähr ein Jahr länger am Markt als beim CISC.

RISC ist optimal auf die Sprache C zugeschnitten. Die meisten CISCs versuchen verschiedene Sprachen gleichzeitig zu unterstützen. Daraus folgt, daß ein C-Programm auf RISC an sich schon schneller läuft als auf CISC.

9. Parallelrechnerkonzepte

Wenn Prozessoren ihren eigenen Speicher haben und das Verbindungsnetz nur zur Kommunikation zwischen den Prozessoren genutzt wird, dann nennt man das Ganze **Multicomputer-System**.

Wenn die Prozessoren einen gemeinsamen Speicher benutzen, der aus mehreren Modulen bestehen kann, und das Verbindungsnetz zur Kommunikation zwischen den Prozessoren und dem Speicher dient, was eine hohe Kommunikationsbandbreite benötigt, dann nennt man das Ganze **Multiprozessor-System**.

Der Zweck ist generell die Steigerung von Leistung und Zuverlässigkeit des Gesamtsystems.

Verschiedene Faktoren hindern beide Systemvarianten daran, die lineare Leistungsvervielfachung zu erreichen:

- Synchronisation: Die unabhängig arbeitenden Prozessoren müssen gelegentlich koordiniert werden.
- Streit: Verschiedene Prozessoren können dieselbe gemeinsam genutzte Ressource beanspruchen. U.a. Zeit fürs Routen im Verbindungsnetz nötig.
- Algorithmen: Nicht jeder Algorithmus läßt sich ohne weiteres effizient parallelisieren, so daß einige Prozessoren arbeitslos sind.

9.1 Multicomputersysteme

Multicomputersysteme bestehen aus verschiedenen autonomen Computern, die geographisch verstreut sind. Lose gekoppelte Systeme. Diese Systeme bieten sich für grobkörnigen Parallelismus an, d.h. Programme sind in relativ unabhängige Tasks gesplittet, die ohne viel Kommunikation ausgeführt werden.

Ein Prozessor wird zusammen mit seinem Speicher und I/O-Geräten **Computermodul** oder **Knoten** genannt. Kommunikation geschieht zwischen den Knoten durch Nachrichtenaustausch über das **Verbindungsnetz**, an das sie jeweils mit einem Netzinterface angekoppelt sind.

Mit aktueller VLSI-Technik kann ein gesamter Knoten inklusive Knoteninterface auf einem Chip implementiert werden.

9.2 Multiprozessorsysteme

Die Prozessoren in einem Multiprozessorsystem werden gewöhnlich dazu genutzt, eine einzelne Aufgabe gemeinsam auszuführen. Dazu haben sie einen gemeisamen Speicher mit einem systemweiten Adreßraum, der für alle Prozessoren verfügbar ist. Eng gekoppelte Systeme. Bieten sich für feinkörnige Parallelität an.

Die Prozessoren sind an den gemeisamen Speicher über ein **Prozessor-Speicher-Verbindungsnetz** angeschlossen. Sie kommunizieren darüber miteinander und meistens zusätzlich noch über ein **Prozessor-Prozessor-Netz**. Darüber hinaus sind sie mit den I/O-Geräten durch ein drittes Netz, das **Prozessor-I/O-Verbindungsnetz**, verbunden.

Varianten sind private Speicher für die Prozessoren, Cachespeicher zwischen Prozessoren und Hauptspeicher und evtl. ein einziges Verbindungsnetz für alles. Ferner kann die ganze I/O-Geschichte über einen (I/O-) Prozessor oder einige abgewickelt werden.

9.3 Verbindungsnetze

Ein Verbindungsnetz hat generell vier Unterscheidungskriterien:

Timing

Dieses kann *synchron* oder *asynchron* sein. Synchrone Systeme werden z.B. in Arrayprozessoren benutzt, um Daten zu allen Prozessorknoten zu senden, oder um alle Knoten zur Kommunikation mit einem ihrer Nachbarn zu veranlassen. Asynchrone Systeme haben keine globale Clock und werden bei Systemen genutzt, wo Anforderung nach Kommunikation unabhängig von den Prozessoren ausgelöst wird.

Switching Methode (Verbindungsaufbau)

Hier gibt es *circuit switching* und *packet switching*. Beim circuit switching wird zwischen den Partnern ein physikalischer Pfad aufgebaut, der für den kompletten Datentransfer gehalten wird. Beim packet switching wird die Information in Pakete zerlegt, die individuell durch das Netz gesendet werden, ohne tatsächlich einen physikalischen Pfad aufzubauen. Verschiedene Pakete können verschiedene Routen nehmen.

In der Regel ist packet switching besser für kurze Nachrichten geeignet und circuit switching empfiehlt sich für seltene lange Nachrichten.

Control strategy (Steuerung)

Die Kontrolle kann *zentralisiert* oder *verteilt* sein. Bei zentraler Kontrolle erhält der globale Controler alle Kommunikationswünsche und stellt die Schalter des Netzes entsprechend ein. Bei der verteilten Kontrolle regelt das Netz solche Anforderungen und das Schaltersetzen selbst auf verteilte Weise.

Topologie

Die Topologie bestimmt, welche Quellknoten mit welchen Zielknoten verbunden sind. Sie können in statische und dynamische unterschieden werden. Eine statische Topologie hat dedizierte Verbindungen zwischen Konten, die nicht geändert werden können. Bei einer dynamischen Topologie können diese Verbindungen durch Umsetzen von Schaltern im Netz anders konfiguriert werden.

9.3.1 Statische Verbindungsnetze

Verbindungsnetze für Multicomputersysteme haben für gewöhnlich eine statische Topologie, asynchrone Timing, nutzen packet switching und basieren auf verteilten Kontrollmechanismen.

Um ein statisches Verbindungsnetz zu charakterisieren, werden zwei Kriterien unterschieden: der **Grad eines Knoten** und der **Durchmesser des Netzwerks**. Der Knotengrad ist die Anzahl der Verbindungen an diesen Knoten. Der Durchmesser eines Netzes ist die maximale Anzahl von Verbindungen, die eine Nachricht von einem beliebigen Knoten zu einem anderen passieren muß auf dem kürzesten Weg. Ein kleiner Durchmesser benötigt also einen hohen Grad für jeden Knoten, was die Kosten erhöht.

Das Forschungsziel für Multicomputernetze ist ein System, bei dem die Leistung des Algorithmus gut ausgewogen ist mit der Netzwerktopologie, d.h. dem Grad und Durchmesser.

Ein weiteres Charakteristikum ist die Anzahl der **Dimensionen**, die man braucht, um den Netzgraphen kreuzungsfrei darzustellen. Ein lineares Array ist eindimensional. Ein Ring oder ein Baum oder eine Matrix sind zweidimensional. Allerdings ist der Baum asymmetrisch. Eine Topologie heißt **symmetrisch**, wenn alle Knoten

dieselbe Sicht auf das Netz haben. Sechs voll verbundene Knoten (jeder mit jedem) ist dreidimensional.

Eine interessante Topologie ist der boolesche n-cube oder **Hypercube**. Ein Hypercubenetz ist ein n-dimensionales Netzwerk mit 2^n Knoten. Der Durchmesser ist n, sein Grad auch. Dies ist ein vernünftiger Kompromiß zwischen der linearen und der vollverbundenen Topologie. Ist allerdings für große Knotenanzahlen kostenintensiv. Interessant ist hierbei, daß die Knoten mit binären Adressen numeriert sind und Nachbarn sich in genau einem Bit unterscheiden. Ferner entspricht die Anzahl verschiedener Bits der Entfernung der betrachteten Knoten. Routen einer Nachricht ist einfach: Man leitet sie über eine Verbindung, deren Knoten sich um ein Bit unterscheidet das dem Ziel entspricht. Da jeweils n disjunkte Pfade für eine Nachricht existieren, ist diese Topologie sehr fehlertolerant.

Aus einem n-cube macht man einen n+1-cube, indem man ihn verdoppelt und die gleichnamigen Knoten verbindet. Die Namen werden dann um ein Bit erweitert.

Eine Variation des n-cube ist die **n-cube connected cycles** Topologie. Jeder Knoten des Hypercube wird durch einen Ring aus n Knoten ersetzt. Diese Topologie hat $n \cdot 2^n$ Knoten, einen Grad von 3 und den Durchmesser $n + \lfloor n/2 \rfloor$.

Dem Hypercube ähnlich ist das **binäre De Bruijn-Netzwerk**. Auch hier sind die Knoten binär numeriert. Ein Knoten ist mit einem anderen verbunden, wenn seine Nummer in die andere verwandelt werden kann durch einen links- oder rechts-Shift bzw. durch eine zusätzliche Invertierung des geshifteten Bits. Der maximale Knotengrad ist also 4. Der Durchmesser ist n. De Bruijn Netze sind die asymptotisch kürzesten (bekannten) Netzwerke vom Grad 4.

9.3.2 Dynamische Verbindungsnetze

Netze für Multiprozessorsysteme sind gewöhnlich von dynamischer Topologie. D.h. die Verbindungen zwischen Knoten werden auf Bedarf hergestellt, zur Programmlaufzeit. Diese Netze variieren von einfachen time-shared Bussen bis zu voll verbundenen cross-bar switch.

9.3.2.1 Busorientierte Verbindungsnetze

Time-shared oder common bus (Singulärer Bus)

Das einfachste Verbindungsnetz verknüpft alle Ressourcen (Prozessoren, Speicher, I/O) mit einem allgemeinen Bus. Dies ist wegen seiner Einfachheit sehr billig, jedoch ist die Systemleistung durch die Busbandbreite begrenzt. Dies kann dadurch verbessert werden, daß man jeden Prozessor mit lokalem Speicher oder Cache ausrüstet. Ein Bus ist nicht fehlertolerant, da ein Busfehler alle Ressourcen unerreichbar macht und dadurch ein Systemcrash entsteht.

Multibusorganisation

Eine Lösung dafür kann die Multibusorganisation sein. Hier werden mehr als ein Bus zum selben obigen Zweck eingesetzt. Dann können mehr Prozessoren gleichzeitig Speicher oder I/O-Geräte ansprechen, wobei sichergestellt werden muß, daß Prozessoren nicht dasselbe Speichermodul oder I/O-Gerät ansprechen. Höhere Fehlertoleranz aber auch redundante Pfade.

Cross-bar switch (Kreuzschienenverteiler)

Die Anzahl der Busse kann weiter erhöht werden, bis es für jedes Speichermodul einen Bus gibt. Bei diesem Ausmaß wird das Verbindungsnetz cross-bar switch genannt und es existiert eine vollständige Vernetzung zwischen allen Prozessoren und allen Speichermodulen. Ein cross-bar ermöglicht den simultanen Datentransfer für alle Speichermodule. Um Streit um ein Speichermodul aufzulösen muß jeder Schalter mit extra Hardware ausgestattet sein, was teuer wird. Wegen hoher Pinzahl inkompatibel mit VLSI packaging Praxis.

Multiport memory (Multiport-Speicher)

Wenn die Schaltfunktion und die Logik dazu in die Speichermodule verlegt wird, ist jeder Prozessor direkt mit jedem Speichermodul verbunden, welches dann entsprechend viele Eingänge hat. Im Gegensatz zu einem Bussystem muß ein Prozessor nur warten, wenn er auf eine gerade benutzte Ressource zugreifen will.

Entspricht cross-bar mit Schaltern in den Speichermodulen.

9.3.2.2 Shuffle networks

Diese Netze haben n inputs und n outputs, welche mit Speichern und Prozessoren verbunden werden können. Sie erlauben eine dynamische Topologie, weil Schalter benutzt werden, um inputs und outputs zu verbinden. Diese Schalter sind in Stufen gruppiert und man unterscheidet zwischen single-stage interconnection networks (SSINs) und multi-stage (MSINs). Das Blockierungsproblem kann bei beiden auftauchen, wenn ein Schalter konfliktäre Positionen einnehmen soll.

Single-stage

Jeder Schalter hat zwei inputs und zwei outputs und kann eine gerade oder eine exchange Verbindung intern formen. Die n inputs werden mit den Schaltern **perfect shuffle** verbunden. D.h. input x wird an Schalterstelle $2 \quad x$ modulo $(2^n - 1)$ angeschlossen. D.h. die erste Hälfte der inputs wird auf die jeweils ersten inputs der Schalter verteilt der Reihe nach. Die zweite Hälfte wird mit den jeweils zweiten inputs verbunden.

Es müssen die inputs mit den outputs derselben Nummer verbunden sein, um volle Konnektivität zu erreichen. Dann sind im schlimmsten Fall n Iterationen nötig, bis der gewünschte Ausgang erreicht wird. Nur für packet-switching geeignet.

Multi-stage

Beim **Ω-network** werden 2^n Eingänge mit 2^n Ausgängen über n Stufen mit Schaltern verbunden. Auf jeder Stufe perfect shuffle. Dieses Netz erlaubt, jede Verbindung zu schalten, aber nicht notwendigerweise simultan, wenn sich die Wege zweier Pakete kreuzen.

Lösung des Blockierungsproblems

10% aller Permutationen liefern kein blocking. D.h. bei 90% der Fälle ist die Sequentialisierung der verschiedenen Transfers nötig.

Mit dem sog. Benes-Netzwerk konstruiert man ein Netz, das VLSI-gerechter, weil regelmäßiger ist. Es entstehen identische Subnetze.

9.4 Das Datenkohärenzproblem

Da Multicomputersysteme keinen gemeinsamen Speicher benutzen entsteht das Kohärenzproblem nur bei Multiprozessorsystemen. Wir betrachten hier ein Multiprozessorsystem, das die Prozessoren über einen Bus mit dem Speicher verbindet und jeder einen lokalen Cache besitzt zur Leistungssteigerung.

Ein Speicherschema heißt **kohärent**, wenn

- der Wert, der von einer Loadoperation geliefert wird, immer der Wert der letzten Storeoperation auf dieselbe Adresse ist und
- Operationen mit mehreren Speicherzugriffen atomar, d.h. jede andere Operation auf derselben Adresse ist ausgeschlossen, ausgeführt wird.

Das Kohärenzproblem besteht darin, daß irgendwo eine Kopie besteht, die nicht den aktuellen Wert hat. Ein **write-through** nützt nicht immer etwas, da dadurch eine alte Kopie in einem anderen Cache nicht upgedatet wird. Es gibt jedoch folgende Lösungen:

- Eine **statische Lösung** wird zur Compilezeit ausgeführt und nutzt **Software**, um das Datenkohärenzproblem durch **Vorbeugung** zu lösen.
- **Dynamische Lösungen** finden zur Laufzeit statt und benötigen **Hardware** zur **Entdeckung und Auflösung** des Datenkohärenzproblems. Dynamische Lösungen können
  (a) **zentralisiert** oder
  (b) **verteilt** implementiert werden.

9.4.1 Vorbeugende Maßnahmen

Dem Datenkohärenzproblem kann vorgebeugt werden, wenn das Betriebssystem alle schreibbaren Daten, die gemeinsam benutzt werden, als **non-cacheable** etikettiert.

Um das Problem der erhöhten Speicherzugriffszeit auf alle gemeinsamen Daten zu erleichtern, können gemeinsame Datenstrukturen gecached werden. Der Zugriff auf diese Daten ist nur innerhalb von **kritischen Abschnitten** erlaubt, die durch non-cacheable locks geschützt sind. Innerhalb dieser kritischen Abschnitte ist der Code dafür verantwortlich, daß alle geänderten Daten zum Speicher zurückgeschrieben werden, bevor er den lock wieder aufhebt.

Die Nachteile dieser Lösungen sind die Notwendigkeit spezialisierter Systemsoftware und die Notwendigkeit, daß der User oder der Compiler Daten als shared oder non-shared deklariert. Außerdem fällt die cache-hit-ratio, da Teile der Daten nicht gecached ist.

9.4.2 Zentralisierte Lösungen basierend auf Entdekkung und Auflösung

Der Vorteil hierbei ist die Transparenz für den User, obwohl hier extra Hardware nötig wird, weil Datenkohärenz sichergestellt wird während der Programmausführung.

Censier hat 1978 ein Schema vorgeschlagen, bei dem der Status der Datenline im Cache gespeichert wird:

- in einem Tag neben der Cacheline im Cache und
- in einem Tag neben der entsprechenden Line im Speicher.

Jede Line im Speicher hat ein Tag, das ein Präsenzflag P_i für jeden Cache im System und ein modified-flag M enthält. P_i=1 bedeutet, daß Cache i eine Kopie dieser Line enthält. M=1 bedeutet, daß eine Kopie der Dataline in einem Cache modifiziert wurde.

Ein Prozessor darf eine Line in seinem Cache nur ändern, wenn er die einzige gültige Kopie besitzt. D.h., wenn M=1, dann ist nur ein einziges Präsenzflag gesetzt.

Jede Cacheline hat einen Tag, der die Speicherline angibt, ein Gültigkeitsflag V enthält und ein Privatflag P.

Der Vorteil hierbei ist, daß keine unnötigen Invalidierungsanforderungen versand werden und daß der Verbleib der letzten Kopie einer Dataline leicht vom Memorycontroler bestimmt werden kann.

9.4.3 Verteilte Lösungen basierend auf Entdeckung und Auflösung

Bei diesem Ansatz wird die Verantwortung über die Datenkohärenz an die Cachecontroler gegeben, und nicht wie oben an die Speichersteuerung. Vorteil: reduzierter Busverkehr.

Bei diesem Schema, basierend auf einer verteilten Lösung, muß jeder Cachecontroler kontinuierlich alle Datentransfers überwachen und im Falle eines Adreßmatch etwas unternehmen (z.B. Cacheline invalidieren) in Abhängigkeit vom Typ der Busanfrage. So ein Cachecontroler wird **snoopy cache controler** genannt. Es gibt wiederum viele verteilte Hardware-Ansätze zur Lösung des Datenkohärenzproblems. Hier sollen nur drei Methoden diskutiert werden:

9.4.3.1 Dedicated broadcast Busse

Jeder Cache ist an einen speziellen high-speed broadcast Bus angeschlossen. Jeder wirte auf einen Cache wird an den Speicher weitergerreicht (write-through) und zusätzlich die Adresse über den Spezialbus gesendet, der permanent von jedem Cache überwacht wird. Wenn die gesendete Adresse einen cache-hit auslöst, wird die Kopie dieser Line in diesem Cache invalidiert.

Nachteile sind: Hoher Busverkehr wegen write-through, Cache tut fast nix sonst als Überwachen, Puffer im Cache nötig, um viel Invalidationsverkehr aufzufangen. Dieses Verfahren eignet sich also nur für Multiprozessorsysteme mit wenigen (typischerweise zwei) Prozessoren.

9.4.3.2 Write-once Methode

Ein write-through wird benutzt, um exklusiven Gebrauch einer Datenline zu beanspruchen. Nach diesem ersten write wird der Speicher nur noch nach der copy-back Strategie upgedatet.

Eine Cacheline hat also vier mögliche Zustände:

1. Invalid: Keine Daten im Cache.
2. Valid: Line wurde vom Speicher gelesen und nicht geändert.
3. Reserved: Line wurde lokal exakt einmal geändert, seitdem sie vom Speicher gelesen wurde und die Änderung ist zum Speicher geschickt worden.
4. Dirty: Line wurde mehrmals geändert und Speicher ist nicht auf dem neuesten Stand.

Der Nachteil ist hier, daß Zusatzleitung auf Bus nötig ist, um Speicher daran zu hindern, Daten zu liefern. Dafür muß der entsprechende Cachecontroler sorgen, denn im Speicher ist kein Vermerk darüber, ob es sich um eine gültige Kopie handelt. Vorteil: Keine besondere Speichersteuerung nötig, kein zusätzlicher Speicherplatz.

9.4.3.3 Ownership Methode

Jede Speicherline hat ein 1-bit Tag, welches anzeigt, ob der Speicher oder ein Cache die Daten bei einer Busreadoperation liefern soll. Der Vorteil ist, daß keine speziellen inhibit-Leitungen auf dem Bus nötig sind. Hier muß Eigentum angemeldet werden, bevor auf die Line geschrieben wird. Um Eigentümer zu werden, gibt der Cachecontroler ein private read oder declare private Kommando. Jede Line kann public oder private sein, angezeigt durch ein Flag im Speicher und in den Caches. Wenn eine Line public ist, dann ist der Speicher der Eigentümer und Caches können nur read-only Kopien enthalten (geholt mit einem shared read). Wenn der Status private ist, dann ist der Eigentümer ein Cache und der hat die einzige gültige Kopie und exklusiven Zugriff darauf. Wenn eine Line gelesen wird, die einem Cache privat gehört, dann gibt dieser das Eigentum an den Speicher und liefert die Daten an den anfragenden Cache und den Speicher. Die Line wird dann public. Falls die Line vorher public war, liefert der Speicher die Daten. Wenn ein private read kommt, dann werden Eigentum und Daten direkt an den anfragenden Cache gegeben. Der Speicher ignoriert dies, da er nicht Eigentümer ist.

Im Vergleich zum dedizierten Broadcastbus wird der Speicherverkehr verringert, da nur bei cache-miss und beim modifizieren einer public line Busaktivität nötig ist.

10.Datenfluß- und Reduktionsmaschinen

Dazu bitte unter >Kapitel B. auf Seite 1, das das gleichnamige Buch zusammenfaßt, nachsehen!