Skript zur Vorlesung

Rechnerarchitektur I/II

Version 2.3.0

Prof. Dr. Christian Siemers

Institut für Informatik

TU Clausthal

Inhaltsverzeichnis:

1 Einleitung

Die Technische Informatik befasst sich mit den technischen Grundlagen und den realen Architekturen von digitalen Schaltkreisen, um (meist sequentielle) Programme, die in Form der Software vorliegen, auch tatsächlich auszuführen. Hierzu hat sich in den vergangenen Jahrzehnten eine Dualität der Maschinen herausgebildet, die es nunmehr ermöglicht, zwei voneinander deutlich zu unterscheidende Ausführungsmodelle zur Anwendung kommen zu lassen.

Diese beiden Modelle – mit 'Computing in Time' sowie 'Computing in Space' treffend beschrieben – waren bis vor Kurzem sowohl übergangs- als auch berührungslos in der Praxis anzutreffen. *Computing in Space* bezeichnet die Möglichkeit, jeden Algorithmus in eine (feldprogrammierbare) Hardware zu konfigurieren, um dann in einer speziellen Ausführungsform (eines deterministischen endlichen Automaten) diesen ablaufen zu lassen. Der Bedarf an (Hardware-)Raum wächst hierbei mit der Größe des Algorithmus, wodurch die Bezeichnung erklärt wird.

Computing in Time hingegen ist der 'klassische' Ansatz, mit Hilfe von Prozessoren und dem zugehörigen Programm einen Algorithmus auszuführen. Dieses Programm wird hierbei in zeitlich sequentieller Weise zur Ausführung gebracht, sodass die angelsächsische Bezeichnung verständlich wird, da nun mit zunehmender Größe der Zeitbedarf (der Raumbedarf hingegen nur unwesentlich) wächst.

Anwendungen in der Praxis, die beide Methoden gleichzeitig nutzen, sind (noch) ausgesprochen selten. Mit anderen Worten: Beide Modelle sind offenbar verschiedenen Anwendungsklassen und –entwicklern zuzuordnen, wobei ein deutlicher (mengenmäßiger) Schwerpunkt bei den 'klassischen' Prozessoren liegt. Allerdings hat sich seit etwa dem Jahr 2000 eine neue Architekturklasse, mit "Reconfigurable Computing" bezeichnet, weiterentwickelt. Das wesentliche Merkmal dieser Klasse ist der Übergang zwischen den beiden anderen Rechenformen.

Für die Praxis (insbesondere in der Zukunft) wird daher eine neue Aufgabe entstehen, mit «*Design Space Exploration*» (oder auch «*Design Space Estimation*») treffend beschrieben. Man wird nicht nur die logisch korrekte und zeitlich deterministisch funktionierende, sondern auch die kostenmäßig günstigste Form realisieren müssen, und dies bei wachsender Anzahl von Möglichkeiten.

Es ist daher das Ziel dieser Vorlesung sowie des nachfolgenden Teils II, die Möglichkeiten zur Ausführung aufzuzeigen, wie sie sich aktuell darstellen. Der Schwerpunkt wird hierbei allerdings ebenfalls auf der Prozessor-basierten Architektur liegen, insbesondere Teil I widmet sich diesem Part in ausschließlicher Form.

Die Vorlesung Rechnertechnologie I beginnt im Kapitel 2 mit einer Einführung in die Grundlagen des Von-Neumann-Prozessormodells, insbesondere im Hinblick auf den Ablauf im Prozessor. Im weiteren Verlauf dieses Kapitels ein Beispiel in konstruktiver Art betrachtet. Konstruktiv bedeutet hierbei, dass neben der theore-

tischen Betrachtung ein Mikroprozessormodell entsprechend am Markt erhältlicher Mikroprozessoren eingeführt und im Detail dargestellt wird.

Kapitel 3 versucht, aus den gewonnenen Erkenntnissen Rückschlüsse darauf zu ziehen, was verbessert werden kann. Hierzu werden Klassifizierungssysteme für Prozessoren und Rechner betrachtet. Diese Klassifizierungssysteme versuchen, Eigenschaften, insbesondere relative Geschwindigkeiten, von Prozessor- und Rechnersystemen zu identifizieren und zu klassifizieren.

Kapitel 4 führt dann eine weitere Modell-CPU ein, diesmal auf dem RISC-Prinzip basierend. Dieser Prozessor besitzt eine 4stufige Pipeline und zeigt wesentliche Merkmale der kommerziell verfügbaren Varianten.

Kapitel 5 ist den superskalaren Mikroprozessoren gewidmet. Zu diesem Zweck werden die Grundprinzipien der superskalaren Ausführung und die Probleme, die damit gekoppelt sind, diskutiert. In Wechselwirkung mit den superskalaren Architekturen steht auch die Compilertechnologie, die im Kapitel 6 behandelt wird.

Die Vorlesung Rechnertechnologie II wird sich dann mit weiterführenden Themen sowie Forschungen befassen. Hierzu zählen die Very-Long-Instruction-Word-Architekturen (Kapitel 7) sowie alternative Ansätze (Kapitel 9).

Zum Schluss dieser einleitende Worte sei darauf verwiesen, dass dies ein Skript (und damit kein Buchersatz) ist. Für diese Vorlesung werden verschiedene Bücher empfohlen, die je nach persönlichem Schwerpunkt zusätzlich hinzugezogen werden sollten:

[1] *Hennessy, J. L., Patterson, D. A.*: Computer Architecture: A Quantitative Approach. – Second Edition – San Francisco: Morgan Kaufmann Publishers, 1996

[2] *Hennessy, J. L, Patterson, D. A.*: Computer Organization & Design: The Hardware/Software Interface. – Second Edition – San Francisco: Morgan Kaufmann Publishers, 1997

[3] *Christian Siemers*: Prozessorbau. – Carl Hanser Verlag München Wien, 1999

[4] *Oberschelp, W.; Vossen, G.*: Rechneraufbau und Rechnerstrukturen. – 8. Auflage. – München, Wien: R. Oldenbourg-Verlag, 2000

[5] *Flik, T.; Liebig, H.*: Mikroprozessortechnik. – 6. Auflage. – Berlin Heidelberg, New York: Springer-Verlag, 2001

[6] *Giloi, W.*: Rechnerarchitektur. Springer-Verlag, Berlin, Heidelberg, New York, 1981. ISBN 3-540-10352-X.

[7] *Beierlein, Th., Hagenbruch, O. (Hrsg.)*: Taschenbuch Mikroprozessortechnik. 2.Auflage – Fachbuchverlag Leipzig im Carl Hanser Verlag München Wien, 2001

[8] *Schiffmann, W.; Schmitz, R.; Weiland, J.*: Technische Informatik 1, Grundlagen der digitalen Elektronik. – 2. Auflage – Berlin, Heidelberg, New York: Springer Verlag, 2001

[9] *Schiffmann, W.; Schmitz, R. ; Weiland, J.*: Technische Informatik 2, Grundlagen der Computertechnik. – 2. Auflage – Berlin, Heidelberg, New York: Springer Verlag, 2001

[10] *Märtin, C. (Hrsg.)*: Rechnerarchitekturen – CPUs, Systeme, Software-Schnittstellen. – 2. Auflage – München, Wien: Carl Hanser Verlag, 2000.

[11] *Šilc, J.; Robic, B.; Ungerer, T.*: Processor Architecture. – Berlin, Heidelberg, New York: Spriner, 1999.

[12] *André Dehon, John Wawrzynek*, ”Reconfigurable Computing: What, Why and Implications for Design Automation”. Design Automation Conference DAC99, San Francisco, 1999.

[13] *Wen-Mei W. Hwu et. al.*: Compiler Technology for Future Microprocessors. Invited Paper in Proceedings of the IEEE, Special Issue on Microprocessors, Vol. 83 (12), p. 1625 .. 1640, 1995.

[14] *Sascha Wennekers, Christian Siemers*, ”Reconfigurable RISC – a New Approach for Space-efficient Superscalar Microprocessor Architecture” in: Proceedings of the International Conference on Architecture of Computing Systems ARCS 2002, Karlsruhe, Germany, April 2002. Springer Lecture Notes in Computer Science 2299, pp. 165–178.

[15] http://www.pactcorp.com/

[16] *John Crawford*, ”Introducing the Itanium Processors”. *IEEE Micro* Vol. 20(5), pp. 9–11, 2000.

[17] *Andreas Stiller*, ”Architektur für echte Programmierer: IA-64, EPIC und Itanium“. *ct* 2001, H. 13, S. 148–153.

[18] *Uwe Schneider, Dieter Werner*: Taschenbuch der Informatik. 4.Auflage – Fachbuchverlag Leipzig im Carl Hanser Verlag München Wien, 2001

[19] *Sigmund, U.; Ungerer, T.*: Ein mehrfädiger, superskalarer Mikroprozessor. – PARS-Mitteilungen 15, S. 75 .. 84 (1996)

[20] *Matthias Withopf*, ”Virtuelles Tandem – Hyper-Threading im neuen Pentium 4 mit 3,06 GHz“. ct 2002, H. 24, S. 120–127.

[21] *Christian Siemers, Axel Sikora* (Hrsg.), ”Taschenbuch Digitaltechnik”. Fachbuchverlag Leipzig im Carl Hanser Verlag, München Wien, Januar 2003. ISBN 3-446-21862-9

[22] *Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, Rumi Zahir*, ”Introducing the IA-64 Architecture”. *IEEE Micro* Vol. 20(5), pp. 12–23, 2000.

[23] *Harsh Sharangpani, Ken Arora*, ” Itanium Processor Microarchitecture”. *IEEE Micro* Vol. 20(5), pp. 24–43, 2000.

[24] *Cameron McNairy, Don Soltis*, ” Itanium 2 Processor Microarchitecture”. *IEEE Micro* Vol. 23(2), pp. 44–55, 2003.

[25] *Schmitt, F.-J.; von Wendorff, W.C.; Westerholz, K.*: Embedded-Control-Architekturen. Carl Hanser Verlag München Wien, 1999.

2 Von-Neumann-Prozessor

Das Grundprinzip eines einfachen, determinstischen, endlichen Automaten, wie es z.B. in [3] eingeführt wurde, wird sehr häufig im Rahmen von Hardwaredesigns genutzt. Hierdurch lässt sich bereits ein sequenziell arbeitender Rechner konzipieren. Der momentane Eingangsvektor des Automaten wird dafür sowohl als ein aktuelles Steuerwort, das den Programmfluss repräsentiert, als auch als der Input der Daten interpretiert, von dem ausgehend der Nachfolgezustand berechenbar ist. Dieser Nachfolgezustand wird seinerseits gespeichert und ggf. nochmals umgeformt als Ausgangsvektor – als das Ergebnis der Operation – zur Verfügung steht.

Dieses Konzept einschließlich Erweiterungen in Form des Kellerautomaten dient zusammen mit der Turingmaschine als theoretische Grundlage für Berechenbarkeiten, nicht zuletzt aufgrund der darin liegenden Universalität. In der Praxis der aktuellen Rechenanlagen jedoch wird ein anderes, gleichwohl äquivalentes Modell genutzt, das die Trennung zwischen Kontroll- und Datenfluss stärker zum Ausdruck bringt. Dieses Modell basiert auf den resümierenden Überlegungen von Burks, Goldstine und von Neumann und wird abkürzend *Von-Neumann-Rechnermodell* genannt.

Abschnitt 2.1 gibt eine Einführung in das Konzept nach Burks, Goldstine und von Neumann. Dieses Konzept hat in den vergangenen Jahrzehnten seit 1946 Ausprägungen gefunden, die bei vielen konkreten Produkten zu finden sind und in den nachfolgenden Abschnitten diskutiert werden.

Zum Verständnis insbesondere der aktuellen Mikroprozessoren (erhältlich seit 1971, Intel 4004) und Mikroprozessor-basierten Rechnersysteme ist es notwendig, sowohl die Hardware (zumindest in Teilen) als auch das Hardware/Software-Interface zu betrachten. Zum ersteren zählen die Bussysteme, insbesondere in ihren zeitlichen Eigenschaften (→ 2.2) sowie die verschiedenen Speicherarten (→ 2.3). Das Leit- und das Rechenwerk (→ 2.4) bilden den Übergang zum Hardware/Software-Interface. Hier steht im Vordergrund, die logische Funktionalität dieser Teile zu verstehen.

Das Hardware/Software-Interface selbst wird durch das Registermodell (→ 2.5, inklusive einer Klassifizierung der Prozessoren nach dem Operandenzugriff), den Befehlssatz (→ 2.6) sowie die Adressierungsarten (→ 2.7) gebildet. Eine genaue Betrachtung der Phasen der Befehlsbearbeitung (→ 2.8) und – als Erweiterung des Von-Neumann-Rechners – das Interrupt-Request-Konzept (→ 2.9) bilden den Abschluss.

Literatur zu diesem Kapitel:

[3] *Siemers, C.*: Prozessorbau. – Carl Hanser Verlag München Wien, 1999

[4] *Oberschelp, W.; Vossen, G.*: Rechneraufbau und Rechnerstrukturen. – 8. Auflage. – München, Wien: R. Oldenbourg-Verlag, 2000

[5] *Flik, T.; Liebig, H.*: Mikroprozessortechnik. – 6. Auflage. – Berlin Heidelberg, New York: Springer-Verlag, 2001

[6] Giloi, W.: Rechnerarchitektur. Springer-Verlag, Berlin, Heidelberg, New York, 1981. ISBN 3-540-10352-X.

[7] *Beierlein, Th., Hagenbruch, O. (Hrsg.)*: Taschenbuch Mikroprozessortechnik. 2.Auflage – Fachbuchverlag Leipzig im Carl Hanser Verlag München Wien, 2001

[8] *Schiffmann, W.; Schmitz, R.; Weiland, J.*: Technische Informatik 1, Grundlagen der digitalen Elektronik. – 2. Auflage – Berlin, Heidelberg, New York: Springer Verlag, 2001

[9] *Schiffmann, W.; Schmitz, R. ; Weiland, J.*: Technische Informatik 2, Grundlagen der Computertechnik. – 2. Auflage – Berlin, Heidelberg, New York: Springer Verlag, 2001

2.1 Grundkonzept

Zur Darstellung des Von-Neumann-Konzepts sind zwei wesentliche Teilkonzepte notwendig: Die Architektur des Rechnermodell mit den wesentlichen Komponenten sowie das Ablaufkonzept eines Programms. Dabei muss berücksichtigt werden, dass dieses Rechnermodell programmierbar ist. Die Programmierbarkeit bedeutet hierbei, dass die hergestellte Hardware ein Programm in Form eines (Maschinenlesbaren) Textes benötigt, um eine Aufgabe erfüllen zu können

Die Programmierbarkeit ist die praktische Ausprägung der Universalität. Es gelingt hierdurch, eine universelle Maschine bauen zu können und sie in einer gegenüber der Bauzeit wesentlich kleineren Zeit auf das jeweilige Problem (und damit auch wechselnd auf viele Probleme) anzupassen. Es sei hierbei erwähnt, dass die Universalität des Von-Neumann-Rechners ausschließlich durch den beschränkten Speicherplatz begrenzt wird, dass also berechenbare Probleme nur daran scheitern können.

2.1.1 Teilsysteme des Von-Neumann-Rechnermodells

Die kennzeichnenden Teilsysteme eines Rechners nach dem Von-Neumann-Prinzip sind (siehe hierzu auch [3], [4] und [9]):

1. Ein (zentralgesteuerter) Rechner ist aus den drei Grundbestandteilen
   - Zentraleinheit (*Central Processing Unit, CPU*)
   - Speicher (*Memory*)
   - Ein-/Ausgabeeinheit (*Input/Output Unit*)

   aufgebaut. Hinzu kommen noch Verbindungen zwischen diesen Teileinheiten, die als Busse bezeichnet werden. Die CPU übernimmt innerhalb dieser Drei-

teilung die Ausführung von Befehlen und enthält die dafür notwendige Ablaufsteuerung. Im Speicher werden sowohl die Daten als auch die Programme in Form von Bitfolgen abgelegt. Die Ein-/Ausgabeeinheit stellt die Verbindung zur Außenwelt in Form des Austausches von Programmen und Daten her.

2. Die Struktur des Rechners ist unabhängig von dem zu bearbeitenden speziellen Problem. Die Anpassung an die Aufgabenstellung erfolgt durch Speicherung eines eigenständigen Programms für jedes neue Problem im Speicher des Rechners. Dieses Programm enthält die notwendigen Informationen für die Steuerung des Rechners. Dieses Grundkonzept der Anpassung hat zu der Bezeichnung ‘programmgesteuerter Universalrechner’ (engl. ‘*stored-program machine*’) geführt.
3. Der Speicher besteht aus Plätzen fester Wortlänge, die einzeln mit Hilfe einer festen Adresse angesprochen werden können. Innerhalb des Speichers befinden sich sowohl Programmteile als auch Daten, zwischen denen – als Speicherinhalt – grundsätzlich nicht unterschieden wird.

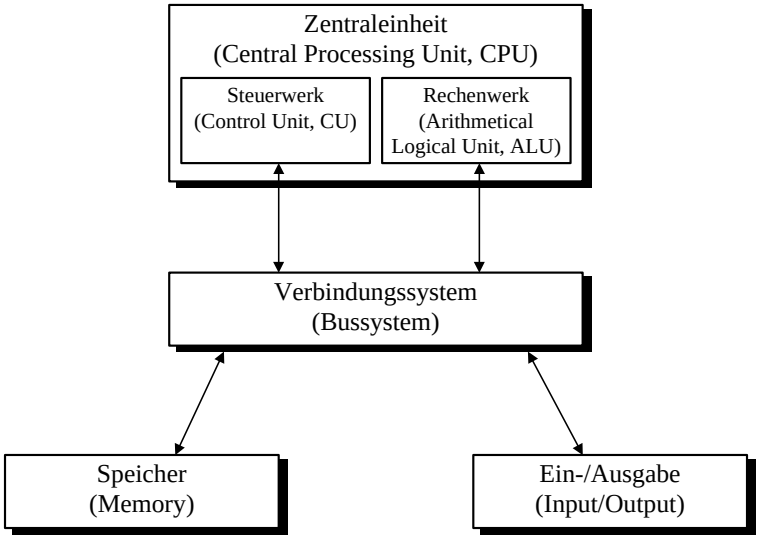


Bild 2.1: Prinzipieller Aufbau eines Von-Neumann-Rechners

Diese Beschreibungen des Von-Neumann-Rechners führen der grafischen Darstellung in Bild 2.1. Das ebenfalls nahezu gleichzeitig entwickelte Harvard-Modell zeigt einen ähnlichen Aufbau, im Wesentlichen wurden hier Code- und Datenspeicher strikt voneinander getrennt.

2.1.2 Ablaufprinzip des Von-Neumann-Rechners

Aus den genannten Grundprinzipien können die wesentlichen Charakteristika des Von-Neumann-Rechners (und im gleichen Maße des Harvard-Rechners) abgeleitet werden:

1. Zu jedem Zeitpunkt führt die CPU exakt einen Befehls aus. Die Steuerung der Bearbeitung liegt im Steuerwerk (Control Unit, CU), das in der Lage sein muss, alle notwendigen Schritte zur vollständigen Behandlung einleiten zu können. Innerhalb eines Befehls kann höchstens ein Datenwert bearbeitet, d.h. neu berechnet werden. Dieses Prinzip wird «*Single Instruction - Single Data (SISD)*» genannt.
2. Alle Inhalte von Speicherzellen, im folgenden *Speicherwörter* genannt, sind prinzipiell als Daten oder Befehle interpretierbar. Die Daten wiederum können als 'eigentliche' Daten oder als Referenzen auf andere Speicherzellen (Adressen) genutzt werden. Die jeweilige Verwendung eines Speicherinhalts richtet sich allein nach dem momentanen Kontext des laufenden Programms.
3. Als Konsequenz aus der vorgenannten Eigenschaft können Daten und Befehle nicht gegen ungerechtfertigten Zugriff geschützt werden, da sie gemeinsam ohne Unterscheidungsmöglichkeit im Speicher untergebracht sind.

Es stellt sich nun die Frage, wie eine Instruktion eigentlich aussehen muss bzw. wie der komplette Instruktionssatz eines Prozessors konzipiert werden muss. Tatsächlich ist diese Frage ad hoc nicht zu beantworten, zumindest nicht vollständig. Selbstverständlich ist die Antwort wichtig, da eine beliebige Rechenvorschrift, ein Algorithmus, auf eine Folge von Instruktionen abgebildet werden wird.

Nimmt man zunächst einmal an, dass der Befehlssatz eines Von-Neumann-Prozessors bekannt ist und den Befehl

```
ld Akkumulator, <Speicherstelle>
```

enthält, soll dieser in seinem typischen Verlauf diskutiert werden (→ Bild 2.2).

Im Rahmen dieser Befehlsbearbeitung (wie im Übrigen aller Ausführungen von Instruktionen) können zwei grobe Phasen identifiziert werden. In der Phase 1 lädt die CPU, gesteuert durch Abläufe in dem Steuerwerk CU, das Befehlswort aus dem Speicher einschließlich aller benötigten Zusatzinformationen wie Operanden, die in diesem Fall aus der Speicheradresse bestehend. Diese Phase zeigt sich am Bussystem, indem zwei lesende Zugriffe – je einer für das Befehlswort und den Operanden – sequenziell ablaufen. Das Steuerwerk speichert die Inhalte in dafür vorgesehene Befehlsregister und interpretiert sie für die weiteren Aktionen.

In der zweiten Phase wird nun der Inhalt der adressierten Speicherzelle und das Prozessor-interne Register, hier mit *Akkumulator* bezeichnet, kopiert. Hierzu trägt der Adressbus die Information über die gewünschte Speicherzelle, aus dem Befehlsregister stammend. Der Speicher, dem zugleich ein Lesevorgang signalisiert wird (Steuerbussignal), muss den Inhalt der Zelle auf den Datenbus

kopieren, so dass dieser abschließend im Akkumulator gespeichert werden kann. Adress-, Daten- und Steuerbus sind Teilsysteme der Verbindungseinrichtung (→ 2.2) im Von-Neumann-Rechner.

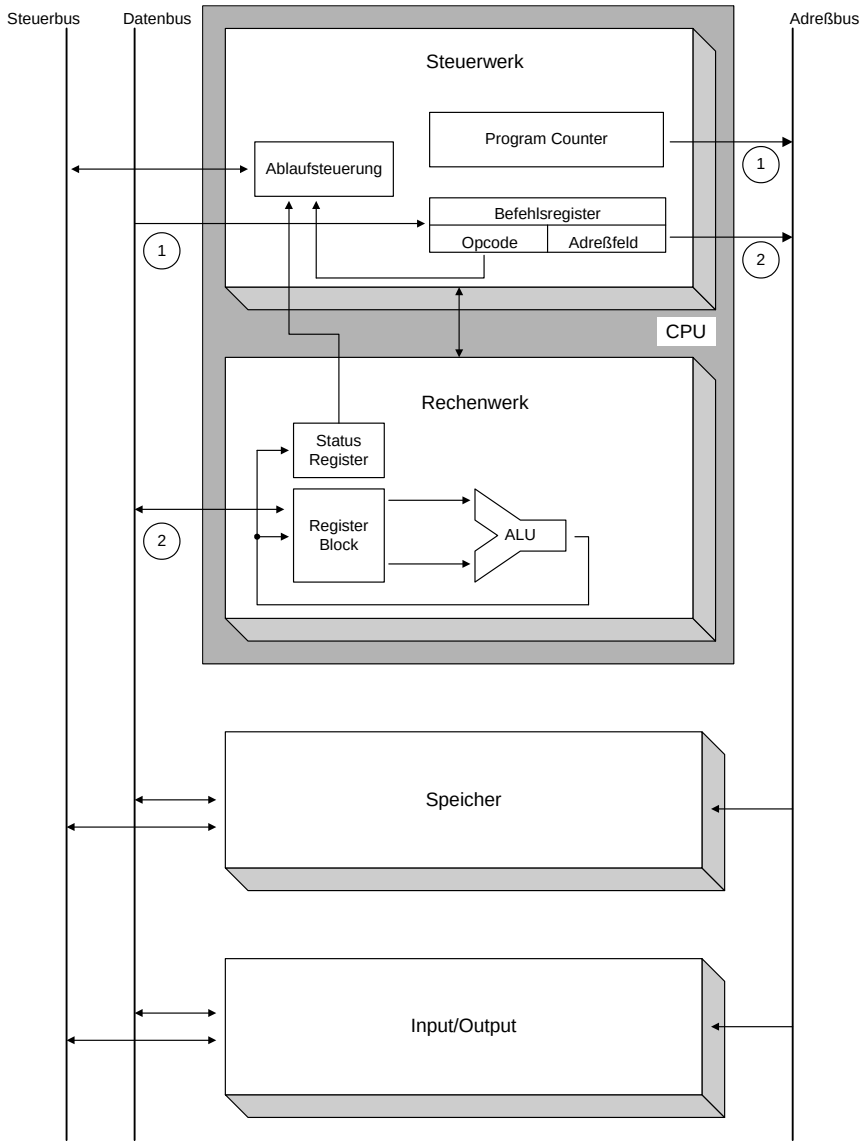


Bild 2.2: Befehlsablauf im Von-Neumann-Rechner

Das hier exemplarisch gezeigte Verfahren zweier Phasen in der Befehlsbearbeitung ist die technische Lösung für das grundsätzliche Problem des Von-Neumann-Rechners, nur eine Verbindungseinrichtung zu zwei verschiedenen Speicherinhalten – Programm und Daten – zu besitzen. Zur Lösung wird hier das Konzept des zeitlichen Multiplex des Bussystems eingesetzt:

1. In der sogenannten *Fetch- und Interpretationsphase* wird aufgrund der durch den Befehlszähler angezeigten Adresse der Inhalt einer Speicherzelle geladen und als *Befehl* interpretiert. Zu dieser Phase zählt im Allgemeinen auch das Laden von Operanden, zumeist von Adressen oder unmittelbaren Daten, sofern diese als Bestandteil des Befehls im Speicher stehen. Dieser weitere Vorgang wird durch die Interpretation des Befehls gesteuert.
2. In der darauffolgenden *Ausführungsphase* wird der Befehl nunmehr vollständig interpretiert und ausgeführt. Die Ausführung kann verschiedene Teile der CPU und des Bussystems sowie der angeschlossenen Einheiten in Anspruch nehmen, dies wird durch das Steuerwerk entsprechend gesteuert. Gemeinsam ist diesen Vorgängen, dass alle Speicherzelleninhalte, auch aus dem Ein-/Ausgabesystem, als *Daten* interpretiert werden.

Dieser zweistufige Ablauf muss für einen Befehl streng sequenziell ablaufen, da eine Abhängigkeit zwischen den verschiedenen Phasen existiert: Ergebnisse früherer Phasen werden für die weitere Bearbeitung benötigt. Spätere Variationen der Von-Neumann-CPU, z.B. die RISC-Architekturen (→ 4), beinhalten ein *Phasenpipelining*, das eine scheinbare Parallelität der Aktionen zueinander bewirkt. Dies stellt keinesfalls einen Widerspruch zu dem bisher Gesagten dar, denn die Parallelität im Phasenpipelining bezieht sich auf verschiedene Befehle, während die Bearbeitung eines Befehls weiterhin streng sequenziell bleibt.

Der Zeitmultiplex der Busnutzung ist notwendig geworden, da – wie aus dem Beispiel bereits ersichtlich wurde – das Bussystem für den Zugriff auf mehrere Arten von Speicherzelleninhalten genutzt wird. Dies wiederum hat seine Ursache darin, dass der Speicher fast immer im Mittelpunkt der Operationen steht, unabhängig davon, ob es Programm- oder Dateninhalte sind, auf die zugegriffen werden soll. Die CPU-Speicherkommunikation wird daher die Leistungsfähigkeit des Gesamtsystems entscheidend beeinflussen, was auch als *Von-Neumann-Flaschenhals (von Neuman bottle-neck)* bezeichnet wird.

Die Grundstruktur des Von-Neumann-Rechners wurde im Harvard-Modell dahingehend variiert, dass Code- und Datenspeicher voneinander getrennt sind und daher auch verschiedene, sich nicht gegenseitig störende Zugänge (Bussysteme) besitzen. Dies gilt auch für das modifizierte Harvard-Modell, das einen Gesamtadressraum für Code- und Daten kennt, hierbei aber unabhängige Zugänge anbietet. Bei Daten-intensiven Rechnungen, wie sie z.B. bei Signalverarbeitungsalgorithmen auftreten, kommen daher gerne Architekturen mit modifiziertem Harvard-Modell zum Einsatz (z.B. Digital Signal Processors, DSP).

Das Ablaufmodell nach Von-Neumann wird im Übrigen auch für das Harvard-Modell genutzt.

2.2 Bussysteme

Die in Bild 2.1 dargestellten Verbindungseinrichtung ist im Von-Neumann-Rechner als sogenanntes *Bussystem* ausgeführt. Hierunter wird ein System von Verbindungsleitungen verstanden, das in wesentlichen Teilen nicht einem einzigen Informationstransfer exklusiv, sondern vielen in zeitlichem Multiplex zur Verfügung steht.

Für ein solches Bussystem hat sich eine Dreiteilung in einen Adressbus, einen Datenbus und einen Steuerbus bewährt. Jedem dieser Teilsysteme ist dann eine genau spezifizierte Aufgabe zugeordnet, wie sie im Folgenden noch erläutert wird.

2.2.1 Grundsätzlicher Aufbau: in- und externe Busse

Eine exklusive Verbindung des Prozessors zu jeder Speicherzelle und zu jedem Teil im Ein- und Ausgabesystem wäre eine maximale Lösung des Verbindungssystems, die sich aus Gründen der immensen Ressourcen weitgehend verbietet. Anstelle eines derartigen Systems wurde im Rahmen des Von-Neumann-Rechnermodells, aber auch der anderen Entwicklungen für Rechnermodelle, eine Unterteilung des externen Verbindungssystems eingeführt, die sich als sehr sinnvoll und technisch gut realisierbar erwies.

Zunächst wird pro Prozessorarchitektur eine Bitbreite definiert, die als Anzahl der zu übertragenden Bits pro Speicherzugriff für alle Transfers gilt. Typischerweise sind dies 4, 8, 16, 24, 32 oder 64 Bit, je nach Ausführung des Prozessors. Ein Transfer von oder zum Speicher – äquivalentes gilt auch für das Ein-/Ausgabesystem – erfolgt immer in der angegebenen Bitparallelität, die zugehörigen Leitungen zum Transfer des Speicherzelleninhalts werden als *Datenbusleitungen* (*Data Bus*) bezeichnet (→ Bild 2.2 und 2.3).

Die externen Datenbusleitungen müssen bidirektional ausgeführt sein, um sowohl lesend als auch schreibend wirken zu können. Technisch gesehen werden Tristatefähige Treiberelemente (→ Bild 2.4) benötigt, die durch Steuersignale ein- bzw. ausschaltbar sind. Im ausgeschalteten Zustand (”dritter Zustand“) liefern diese Treiberelemente ein so schwaches elektrisches Signal, das von jedem anderen aktiven Treiberelement mit ’0’ oder ’1’ überstimmt werden kann. Auf diesem Prinzip beruht die Möglichkeit, in Bussystemen mehrere Einheiten zusammen schalten zu können, wobei zu einem Zeitpunkt maximal eine Einheit aktiv ist.

Die Speicherzellen selbst müssen zusätzlich adressiert werden, um eine Eindeutigkeit des Transfers zu gewährleisten. Diese Adressierung erfolgt über gesonderte Leitungen, die zusammenfassend als *Adressbus* (*Address Bus*) die eindeutige Zuordnung zwischen physikalischer Speicherzelle und gewünschter Adresse im Pro-

zessor herstellen. Der Adressbus muss aus Sicht der CPU nur unidirektional als Ausgang vorhanden sein, da es immer der Prozessor ist, der die konkrete Adresse bestimmt. Eine Ausnahme hierzu bilden nur Multimaster-fähige Bussysteme etwa in Mehrprozessorkonfigurationen. In diesen Systemen wird der Adressbus einer einzelnen CPU ggf. ausgeschaltet, um die Bustreiber anderer Busmaster zum Zugriff zuzulassen.

Das dritte Teilsystem der Verbindungseinrichtung wird als *Steuerbus* (*Control Bus*) bezeichnet. Hiermit werden alle Signale zusammengefasst, die der externen Steuerung eines Datentransfers am Bus dienen, beispielsweise zum Anzeigen der Transferrichtung (*READ* bzw. *WRITE*) oder auch zur Unterscheidung zwischen Speicher- und Ein-/Ausgabetransfer (*MEMORY* bzw. *IN/OUT*). Diese Signale besitzen im Allgemeinen eine weniger einheitliche Struktur, verglichen mit dem Adress- bzw. Datenbus. Eine genauere Diskussion der Aufgaben und eine Spezifikation der insbesondere zeitlichen Zusammenhänge lässt sich nur in Verbindung mit realen Speichertransfers erreichen (→ 2.2.2).

Die externen Bussysteme müssen innerhalb des Prozessors Quellen oder Senken besitzen, aus denen die Informationen erhalten bzw. in denen sie gespeichert werden. Innerhalb jeder CPU existiert daher ein im Allgemeinen wesentlich komplexeres Bussystem für Daten und Adressen, gesteuert durch das Steuer- oder Leitwerk (→ 2.4), während einige dieser Steuerleitungen auch extern vorhanden sind.

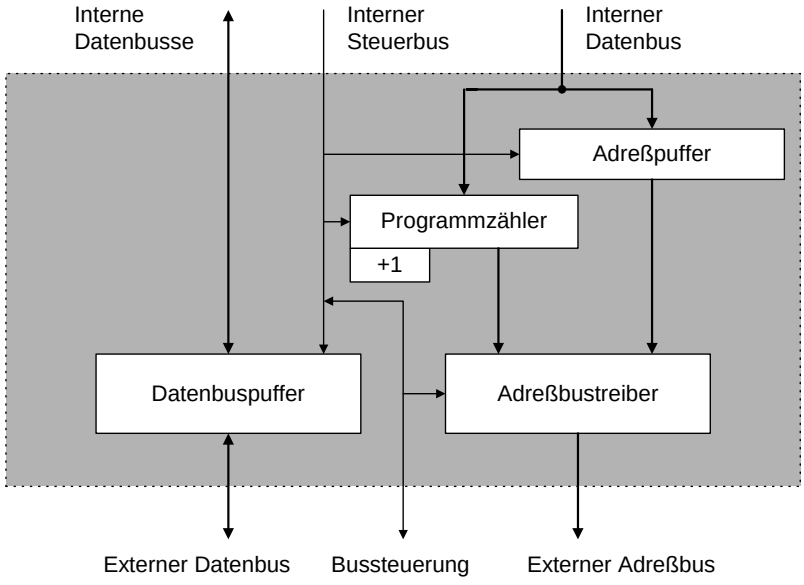


Bild 2.3: Die Systembus-Schnittstelle zwischen in- und externen Bussystemen

Bild 2.3 zeigt eine derartige Schnittstelle für eine Leitung. Der externe Datenbus ist mit den zugehörigen internen Systemen – hier sind durchaus mehrere denkbar – durch bidirektionale Treiberelemente verbunden (→ Bild 2.4).

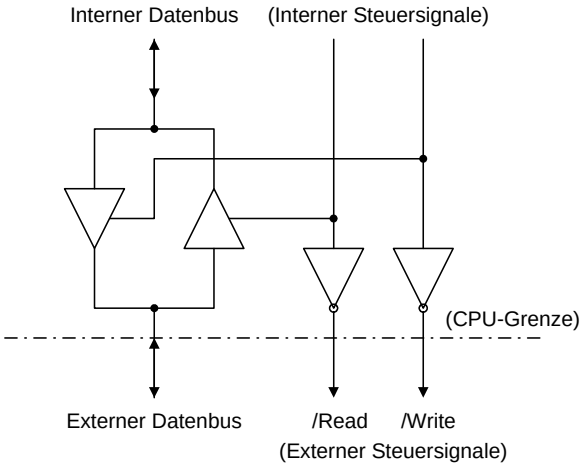


Bild 2.4: Bidirektionaler Datenbustreiber mit Steuersignalen

Als interne Quellen für die Adressbusleitungen können zum einen der Programmzähler (PC), zum anderen spezielle Adressregister dienen. Jeder Zugriff auf das nächste Befehlswort (Fetch) wird über den Adressbus adressiert, die Adresse stammt aus dem Programmzähler. Dieser Zugriff ist quasi obligatorisch, nach einem Zugriff wird der Wert um eine Zugriffsweite erhöht, in Bild 2.3 durch die Inkrementeinheit '+1' dargestellt. Bei Sprungbefehlen jedoch ändert sich der Wert des Zählers, indem ein neuer Wert, die Sprungzieladresse, in den PC geladen wird. Der nächste Fetch erfolgt dann auf die neue Adresse.

Der Programmzähler sowie das Adressregister erhalten den gültigen Wert über einen internen Datenbus. Das Adressregister entspricht dem Adressfeld des Befehlsregister aus Bild 2.2, hier wird die Zugriffsadresse für einen Speichertransfer (Daten) gespeichert, um als Quelle für den Adressbus in der Ausführungsphase – sofern dies benötigt wird – zu dienen.

2.2.2 Adress-, Daten- und Steuerbus im zeitlichen Verhalten

Die Ausführung des Bussystems als mehrfach genutzte Verbindungsleitungen bedeutet, dass für den zeitlichen Multiplex ein entsprechendes zeitliches Verhalten eingeprägt sein muss, da die Busleitungen nicht exklusiv für eine Verbindung genutzt werden können. Dieses zeitliche Verhalten ist durch ein *Handshake* be-

stimmt, mit dem die Einheiten, die einen Datentransfer durchführen, sich gegenseitig abstimmen. Dieses Handshakeverfahren ist in seinem zeitlichen Verhalten wiederum zumeist an den CPU-Takt gekoppelt.

Generell werden drei Arten von *Bushandshakes* unterschieden: synchrone, semisynchrone und asynchrone Busprotokolle. Die asynchronen Systembusse besitzen hierbei keinerlei Kopplung an einen Takt, sind daher sehr flexibel, aber auch aufwendig. Sie waren in der Motorola MC68000-Architektur realisiert worden, zurzeit findet man asynchrone Bussysteme jedoch nur bei asynchronen Prozessordesigns (Amulet 3E), so dass sie aus der Betrachtung hier herausfallen.

Bild 2.5 zeigt die logischen und (qualitativ) zeitlichen Zusammenhänge für einen *synchronen Systembus*. Kennzeichnend hierfür ist die starre Kopplung aller Signale an den CPU-Takt. In dieser Darstellung wurden zwei CPU-Takte pro Buszugriff angenommen, in der Praxis variiert dieses Verhältnis zwischen einem und zwölf Takten. Die aus diesem Verhältnis resultierende Periodizität mit der Länge *T* wird auch Bustakt genannt.

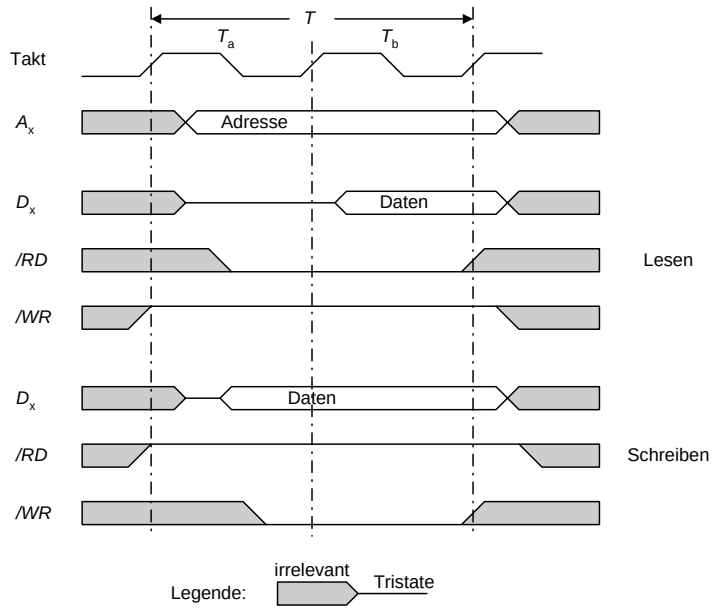


Bild 2.5: Zeitliche Abläufe auf einem synchronen Systembus

Während des gesamten Bustakts wird – mit einer gewissen Phasenverschiebung – die Adresse der Speicherzelle am Adressbus angelegt. Bei einem lesenden Zugriff erwartet die CPU ab einem definierten Zeitpunkt, in Bild 2.5 kurz nach Beginn des zweiten Takts T_b, ein stabiles Signal am Datenbus, das bis zur Übernahme in interne Register des Prozessors unveränderlich stabil sein muss. Die Kennzeichnung des lesenden Zugriffs erfolgt durch das Steuerbussignal */RD*, das wie allge-

mein üblich low-aktiv gekennzeichnet wurde. Die Übernahme der Daten erfolgt in Bild 2.5, gekoppelt an die positive Flanke am Ende des Bustaktes.

Der schreibende Zugriff am Bus verläuft sehr ähnlich, hier ist das Signal */WR* low-aktiv und */RD* bleibt inaktiv. In diesem Fall setzt der Prozessor die Signale am Datenbus durch Aktivschaltung der entsprechenden Treiber aus Bild 2.4.

Das synchrone Busprotokoll besitzt, wie bereits diskutiert, eine starre Kopplung zwischen Zugriffsdauer und CPU-Takt und damit ein nicht variables Zeitschema. Die wichtigste Konsequenz hieraus ist, dass alle Busteilnehmer die Zeiten, die ihnen zur Verfügung stehen, einhalten müssen. Ein gemischtes Design mit langsamen und schnellen Speicherbausteinen beispielsweise erzwingt einen Bustakt, der dem langsamsten Teilnehmer entsprechen muss. Da das Verhältnis zwischen Bus- und CPU-Takt nicht variierbar ist, wird zugleich dieser herabgesetzt und der Prozessor verlangsamt. Dieser Nachteil des ansonsten sehr einfachen Protokolls wird im semi-synchronen Busprotokoll vermieden.

Für das *semi-synchrone Busprotokoll* wird eine zusätzliche Leitung benutzt, die meist mit /Ready bezeichnet wird und seitens des Prozessors einen Eingang darstellt. Bild 2.6 zeigt die Verhältnisse für einen lesenden Zugriff. Liegt auf dieser Leitung rechtzeitig vor dem Speichern in das interne Prozessorregister ein aktives Signal vor, so wird dem Prozessor signalisiert, dass der Buszyklus in der minimalen Zeit beendet werden kann und die Daten gültig sind. /Ready steuert also die Übernahme und das Zurücksetzen von */RD*.

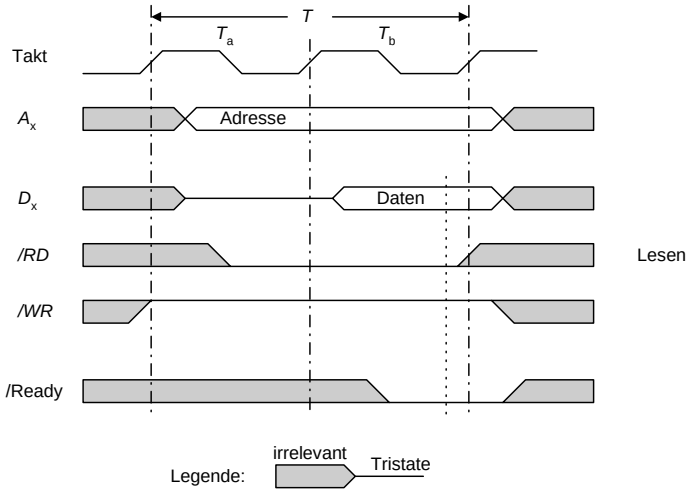


Bild 2.6: Zeitliche Abläufe auf einem semi-synchronen Systembus (lesender Zugriff)

Bild 2.7 zeigt diesen Vorgang für einen schreibenden Bustransfer, bei dem das /Ready-Signal nicht rechtzeitig vorliegt und somit für eine Verlängerung des Zugriffes sorgt.

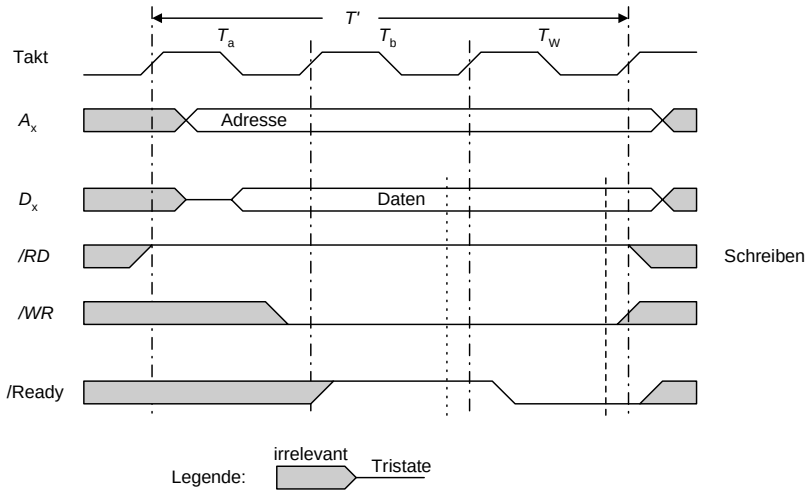


Bild 2.7: Zeitliche Abläufe auf einem semi-synchronen Systembus bei Zugriffsverlängerung

Das /Ready-Signal ist in dieser Darstellung zu dem Zeitpunkt, der zur Beendigung des Buszyklusses ohne Verlängerung einzuhalten wäre, definitiv inaktiv. Der Prozessor verlängert daraufhin den (Schreib-)Zyklus *T* um einen weiteren CPU-Takt, hier mit T_w bezeichnet, zum Zyklus *T'*. Innerhalb dieser Verlängerung, die prinzipiell noch um mehrere Wartezyklen erweiterbar ist, signalisiert der Speicher über die /Ready-Leitung, dass die Daten gespeichert worden sind, sodass der Prozessor diesen Bustransfer beenden und den nächsten starten kann.

Das /Ready-Signal ist also im Rahmen des Busprotokolls für die Einhaltung der Transferzeiten zuständig und synchronisiert Prozessor und Peripherie bzw. Speicher. Dieses Signal muss in einem Rechnerdesign ggf. separat erzeugt werden, falls die verwendeten Speicherbausteine keine Generierung vorsehen.

2.2.3 Vergleich des synchronen und semi-synchronen Busprotokolls

Das synchrone Busprotokoll war die erste Methode, um Prozessor und Speicher (sowie Peripherie) miteinander zu koppeln. Es wird aktuell bei relativ einfachen Mikroprozessoren (z.B. Ausführung in 4- und 8-Bit-Datenbusbreite) eingesetzt und ist besonders dann empfehlenswert, wenn es um die Einfachheit des Bussystems geht.

Voraussetzung hierfür ist die Auslegung aller Systemkomponenten auf etwa die gleiche Geschwindigkeit, um den angestrebten Betriebstakt nicht unnötig heruntersetzen zu müssen.

Für das semi-synchrone Busprotokoll gilt natürlich, dass das Prozessordesign etwas komplexer wird, bedingt durch die Synchronisierung mit dem /Ready-Signal. Aus diesem Grund werden nur Mikroprozessoren mit 'höheren' Datenbusbreiten (etwa ab 16 Bit) mit diesem Bussystem ausgeführt. Der Gewinn an Flexibilität im Verhältnis Prozessortakt zur Geschwindigkeit der anderen Busteilnehmer ist natürlich groß und kann zu einer entsprechenden Systementscheidung führen. Andererseits ist das semi-synchrone Busprotokoll sehr einfach in das synchrone zu überführen, indem die /Ready-Leitung des Prozessors permanent auf low gesetzt wird.

Das semi-synchrone Busprotokoll wird gegenüber dem asynchronen Busprotokoll bevorzugt, weil hier die Taktkopplung zu einem vergleichsweise einfacheren Design führt. Dies gilt insbesondere für sehr hoch getaktete Mikroprozessoren etwa im Bereich der PCs.

2.3 Speicher

Eines der Grundelemente des Von-Neumann-Rechners (wie auch des Harvard-Modells) wird mit *Speicher (Memory)* bezeichnet und im Abschnitt 2.2 bereits als Partner für Transfers am Bus behandelt. Dieser Speicher dient der Aufnahme sowohl von Programmcode als auch von Daten.

Jeder Speicherplatz ist über eine binäre Adresse ansprechbar und enthält eine spezifizierte Anzahl von binärwertigen Informationen, meist in Form von 1, 4, 8, 16 oder 32 bit (bit: Binary Digit). Im Rahmen eines Rechners muss die Adresse eines Speicherplatzes eindeutig definiert sein, um ein deterministisches Verhalten des Rechners zu erlangen.

Entsprechend den unterschiedlichen Einsatzkriterien sind verschiedene Formen des Speichers erhältlich. Im Allgemeinen unterscheidet man:

- *Nur-Lese-Speicher* (Read Only Memory, ROM): Der Speicherinhalt wird auf fabrikatorische Weise oder durch einen physikalischen Programmiervorgang (PROM, Programmable ROM) definiert und ist seitens des Rechners unveränderbar. Diese Speicherform hat den Vorteil, daß der Speicherinhalt ohne Zufuhr von Energie erhalten bleibt und somit bei jedem Einschalten des Rechners zur Verfügung steht.
- *Schreib-Lese-Speicher* (Random Access Read Write Memory, RAM): Der Speicherinhalt ist durch den Rechner bestimmbar (schreibender Vorgang) und kann jederzeit wiedererlangt werden (lesender Vorgang). Die Abkürzung RAM deutet hierbei auf eine weitere Eigenschaft des Speichers hin, die in den allermeisten Fällen auch für den Nur-Lese-Speicher gilt. Der Zugriff auf die Spei-

cherzellen ist wahlfrei (Random Access), d.h., es ist keinerlei Reihenfolge der Adressen im Zugriffsverfahren einzuhalten.

RAMs können ohne besondere Maßnahmen ihren Speicherinhalt bei Ausschalten des Rechners nicht speichern, so dass beim Einschalten ein undefinierter Inhalt vorliegt. Besondere Bauformen, wie Batteriepufferung, NVRAM (Non Volatile RAM) und FeRAM (Ferroelektrisches RAM), hingegen retten den Inhalt auch über den Verlust der Betriebsspannung und sind für besondere Anwendungen im Einsatz.

- Eine besondere Form könnte zukünftig das magnetoresistive RAM (MRAM) einnehmen, das die binärwertige Information in Form von zwei parallelen (1) oder anti-parallelen Magnetfeldern (0) speichert. Das Besondere hieran ist, dass MRAM Eigenschaften von ROM (Energie-freie Speicherung) und RAM (beliebig viele Schreibzyklen) vereinigt. MRAM werden ab 2004 am Markt erwartet.

Eine minimale Ausstattung eines Von-Neumann-Rechners beinhaltet einen Nur-Lese-Speicher für das nach Einschalten auszuführende Programm und einen Schreib-Lese-Speicher für die Variablen, die im Rahmen des Programms die Daten enthalten.

Eine genaue Darstellung von Speichertechnologien und –architekturen ist in Abschnitt 9.1 dieses Skripts zu finden.

2.4 Leit- und Rechenwerk

2.4.1 Register im Leit- und Rechenwerk

Die Konzeption des Von-Neumann-Rechners zeigt innerhalb der Central Processing Unit eine Zweiteilung der Aufgaben in Control Unit (Leit- oder Steuerwerk) und Arithmetical Logical Unit (Rechenwerk) (→ 2.1, Bild 2.1). Das Leitwerk hat dabei im Wesentlichen die Aufgabe, den gesamten Programmablauf sowie den Ablauf der einzelnen Instruktionen, jeweils in ihrer zeitlichen Sequenz, zu steuern. Die arithmetischen und logischen Rechenkapazitäten sind hingegen im Rechenwerk zusammengefasst.

Beide Teileinheiten benötigen zur Befehlsbearbeitung Speicherplatz, die ausschließlich vom Prozessor verwaltet werden und außerhalb des ’normalen’ Code- bzw. Datenspeichers liegen. Diese Speicherstellen werden im allgemeinen *Register* genannt. Sie können in drei Kategorien eingeteilt werden:

- Datenregister zur Aufnahme von Operanden sowie zur Speicherung von Ergebnissen
- Adressregister zur Adressierung von Operanden im Speicherbereich des Prozessors

- Steuerregister zur Steuerung des Ablaufs der normalen Befehlsbearbeitung und für besondere Situationen (Ausnahmebehandlung)

Die für eine maschinennahe Programmierung direkt anspechbaren Register werden im Registermodell (→ 2.5) zusammen gefasst. In diesem Fall müssen die Register auch adressiert werden bzw. dienen im Fall der Nutzung als Adressregister der Adressierung von Speicherzellen (→ 2.6).

Die Register in einem Prozessor können hierbei nicht immer eindeutig einem Bereich (Leitwerk oder Rechenwerk) zugeordnet werden. Am ehesten gelingt dieses für die *Datenregister*, die im Bereich des Rechenwerks (ALU) liegen. Diese Register besitzen eine zentrale Rolle im Rahmen der Berechnung neuer Werte. In modernen Konzepten wird die Anzahl der Datenregister möglichst auf hohem Niveau gehalten, da die Ausführungszeiten von Befehlen mit internen Daten deutlich unter denjenigen mit Daten aus dem Speicher liegen.

Die erwähnten *Adressregister* können dem Leitwerk (CU) zugerechnet werden. Diese Register sind für eine Adressierung von Operanden, die nicht in der sequenziellen Abfolge des Programmzählers liegen, zuständig. Diese Adressregister werden im Rahmen der Befehlsbearbeitung mit der berechneten Adresse geladen und bestimmen während des Bustransfers die am Adressbus liegenden Signale.

Zu den *Steuerregistern*, die wiederum weitestgehend dem Leitwerk zuzuordnen sind, zählen der Programmzähler, das Statusregister und der Stackpointer des Prozessors, wobei letzteres auch als Adressregister mit zusätzlicher Funktionalität bezeichnet werden kann.

2.4.2 Aufgaben des Leitwerks

Das Leitwerk des Von-Neumann-Prozessors ist in älteren und aktuell in einfacheren Architekturen als zentraler Automat ausgeführt. Hierin werden folgende Aufgaben erfüllt:

- Die Steuerung des *Instruction Fetch*, d.h. des Zugriffs auf den nächsten, auszuführenden Befehl
- Die Berechnung der nächsten Fetchadresse: Im Normalfall eines Befehls, der nicht in den Kontrollfluss eingreift, ist dies der im Adressraum an der nächsten Adresse liegende Befehl. Im Spezialfall der Kontrollflussbefehle (Sprünge, Verzweigungen) bestimmen diese die nächste Programmadresse, die im Allgemeinen außerhalb der Sequenz liegt.

  Beide Aufgaben ergeben die Ablaufsteuerung eines Programms.
- Die Steuerung der Befehlsbearbeitung der jeweiligen Instruktion im Prozessor: Hierzu zählen die Decodierung der Instruktion, die Anforderung der notwendigen Operanden (Daten), die Ansteuerung des Rechenwerks mit den entsprechenden Informationen zur Operation, die Egebnisspeicherung sowie ggf. die Behandlung von Ausnahmen.

Diese Aufgaben lassen sich sehr gut zentralisieren, sofern die Kommunikationslängen (!) im Prozessor eine untergeordnete Rolle spielen. Ein Vorteil der zentralen Lösung ist auch, dass jederzeit Rückmeldungen aus den einzelnen Teilen berücksichtigt werden können. Das in 2.1 und Bild 2.2 dargestellte Grundmodell des Von-Neumann-Rechners geht im Übrigen davon aus, dass alle Teilaktionen für einen Befehl ausgeführt werden, bevor der nächste Befehl beginnt.

Mit der Einführung des Phasenpipelinings (→ 3.2.2, 4.3) ergibt sich eine verteilte Ausführung der Befehle und damit auch eine verteilte Struktur des Leitwerks. Dies ist bei modernen Architekturen auch deshalb notwendig, weil die Signalübermittlung im Prozessor bei großen Entfernungen einen wesentlichen Teil der Verzögerung ausmacht (→ 4.7).

2.4.3 Rechenwerk

Das *Rechenwerk* (*Arithmetical Logical Unit, ALU*) in einem Von-Neumann-Prozessor ist im eigentlichen Sinn kein Rechen*werk*, sondern ein Rechen*netz*, wenn man es mit den Begriffen Schaltwerk und Schaltnetz [21] vergleicht. Es beinhaltet mehrere Komponenten zur Durchführung aller benötigten Berechnungen. Hierzu zählen mindestens das oder die Datenregister (→ 2.5), das eigentliche Berechnungsschaltnetz (die ALU) und die verbindenden (internen) Datenbusse.

Oftmals kommen weitere Komponenten zum Rechenwerk hinzu. So müssen beispielsweise Adressrechnungen für berechnete Sprungadressen durchgeführt werden, was innerhalb der zentralen ALU, jedoch auch im Rahmen eigenständiger Einheiten integriert werden kann. Die hier gewählte Darstellung beschränkt sich auf die minimalen Komponenten im Rechenwerk. Weiterführende Darstellungen sind z.B. in [4] zu finden.

Das Rechenwerk i.e.S, die *arithmetisch-logische Einheit* (ALU), integriert alle Berechnungskapazitäten für neue Datenwerte innerhalb des Prozessors. Bei speziellen Einheiten wie Multiplizierern, Multiply-Accumulate-Einheiten (MAC) oder Barrel-Shifter gibt es Ausnahmen von dieser Regel, diese sind häufig separat angeordnet.

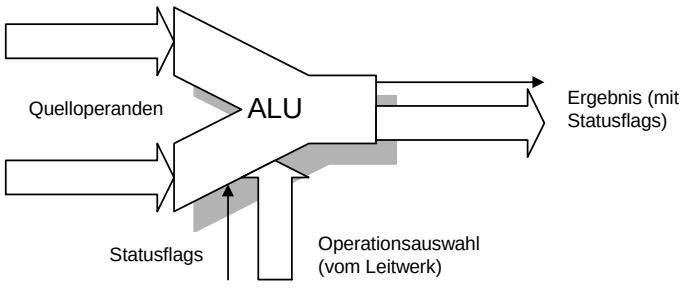


Bild 2.8: Arithmetisch-logische Einheit (ALU)

Die ALU besteht aus einem von außen steuerbarem Schaltnetz mit zwei Datenbussen am Eingang und zwei am Ausgang. Bild 2.8 gibt einen Überblick zur ALU. Die Steuereingänge werden von dem Leitwerk mit Signalen belegt, die aus dem Befehlswort extrahiert sind, und konfigurieren das Schaltnetz für den Zeitraum der Berechnung auf die gewünschte Verknüpfung. Die Operation zwischen den Daten ist dabei in Zusammenhang mit der Interpretation der jeweiligen Darstellung zu sehen. Folgende Klassen von Verknüpfungen sind in der ALU integriert:

- *Arithmetische Verknüpfungen*, wie Addition, Subtraktion, Vergleiche, Multiplikation und Division: Für die meisten Prozessoren sind diese Verknüpfungen auf Integerwerte der Daten beschränkt, während die Interpretation der Floating Point Darstellungen (→ [3]) entweder durch eine Softwareemulation oder innerhalb spezieller Coprozessoren erfolgt. Wie erwähnt können sehr aufwendige Verknüpfungen in speziellen Einheiten integriert sein.
- *Logische Verknüpfungen*, wie Disjunktion (ODER), Konjunktion (UND), Negation (NOT), Einer-Komplement (XOR): Die logischen Verknüpfungen sind auf eine binärwertige Interpretation der Daten fixiert.
- *Verschiebeoperationen*, wie Rotation, arithmetischer und logischer Shift: Auch in diesem Fall werden die Daten als Binärwerte interpretiert, obwohl einzelne Operationen auch einen Zusammenhang mit Arithmetik zeigen.

2.5 Registermodell

Wie bereits eingangs des Kapitels erwähnt wurde, gehört das Registermodell zu den ’sichtbaren’ Teilen des Prozessors: Die Softwareentwicklung auf maschinennaher Ebene kann und muss die durch Maschinenbefehle in ihren Werten veränderlichen Register berücksichtigen.

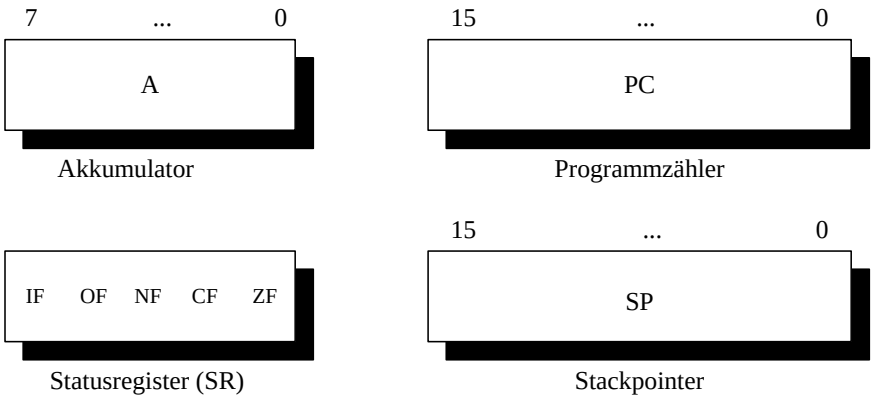


Bild 2.9: Minimales Registermodell

Die Variabilität bezieht sich dabei sowohl auf Register, die in Verbindung mit der ALU stehen (Datenregister) als auch auf Register, zum Leitwerk gehörend (Kontrollregister). Bild 2.9 zeigt ein minimales Registermodell, wie es z.B. in 'kleinen' 8-Bit-Mikrocontrollern zu finden ist. Hierbei wird deutlich, dass die Daten-bezogenen Register (Statusregister, Akkumulator) in der typischen Datenbreite ausgeführt sind, während die Adress-bezogenen Register (Stackpointer, Programmzähler) mit der hier angenommenen Breite von 16 Bit für die Adressen gewählt wurden.

2.5.1 Programmzähler

Das zentrale Register im Leitwerk ist der *Programmzähler* (*Program Counter*, *PC*, auch *Instruction Pointer*, *IP*). Dieses Register zeigt auf die aktuelle Stelle im Programmspeicher, die sich in Bearbeitung befindet.

Der Programmzähler wird regelmäßig dem Programmfluss entsprechend modifiziert:

- Nach einem Reset bzw. nach Einschalten der Versorgungsspannung (so genannter 'Kaltstart') und einer Wartezeit zum Einschwingen des Takts muss der Programmzähler mit einer definierten Adresse geladen werden, um ein deterministisches Verhalten des Prozessors nach einem Neustart zu ermöglichen. Diese Adresse kann selbst fixiert sein (direktes Setzen der Startadresse), beispielsweise bei der 8051-Familie auf die Adresse 0, sie kann jedoch auch an einer definierten Adresse im Speicherbereich stehen und von dort in den PC geladen werden (indirektes Setzen der Startadresse).
- Nach dem Laden eines Befehlsworts (Fetch) wird der Programmzähler um eine Zugriffseinheit (z.B. 2 Byte-Adressen für 16-Bit-Zugriffe) erhöht, er zeigt anschließend auf die nachfolgende Adresse. Diese Eigenschaft kennzeichnet den Programmzähler als Binärzähler.
- Im Rahmen eines decodierten *Sprungbefehls* wird der Programmzähler mit dem Operanden dieses Befehlsworts bzw. der daraus berechneten Sprungzieladresse neu geladen, er zeigt danach auf eine beliebige Adresse in dem Speicherbereich des Prozessors.

Der Programmzähler kann aufgrund der vorgenannten drei wesentlichen Eigenschaften als ladbarer Binärzähler, ggf. mit Reset- bzw. Set-Funktion seiner internen Flipflops, aufgefasst werden.

2.5.2 Statusregister

Das *Statusregister* eines Prozessors stellt ein sehr wichtiges Interface zwischen Leit- und Rechenwerk dar. Hier werden üblicherweise einige Statusbits, sogenannte Flags, gespeichert, deren Werte aus den Berechnungen des Rechenwerks, speziellen Befehlsworten des Kontrollflusses oder Betriebszuständen des Prozessors stammen.

Einige Flags eines typischen Statusregisters sind in Bild 2.10 dargestellt. In dieser oder ähnlicher Form sind sie häufig in Prozessoren integriert:

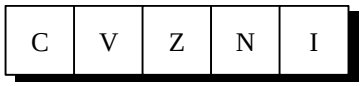


Bild 2.10: Statusregister mit Flags (Auswahl)

- Das *C-Flag* (Carry) entsteht bei arithmetischen Operationen wie der Addition als Übertrag, d.h. als berechnetes Bit, das nicht mehr in die Darstellung innerhalb des Zielregisters passt. In Kombination mit dem *V-Flag* (Overflow, auch als *OV* bezeichnet) lässt sich eine vorzeichenbehaftete Arithmetik programmieren, die auch Überläufe erkennt.

  Andere Einsatzbereiche des Carry-Flags sind Shift- und Rotationsbefehle für Registerinhalte. Beide Flags werden meist durch bedingte Sprung- bzw. Verzweigungsbefehle ausgewertet, deren Ausführung mit dem Zustand des jeweiligen Flags gekoppelt ist.
- Das *Z-Flag* (Zero) und das *N-Flag* (Negative) sind Flags, deren Wert oftmals durch Vergleiche oder bei Zähloperationen gesetzt und im weiteren Programmfluss wie C und V durch bedingte Sprungbefehle ausgewertet werden.
- Das *I-Flag* (Interrupt Disable oder Interrupt Enable, je nach Architektur) zeigt verschiedene Betriebszustände des Prozessors an und steht in engem Zusammenhang mit dem Interrupt-Konzept (→ 2.9), das eine Erweiterung des ursprünglichen Von-Neumann-Rechners darstellt. Im Rahmen des Statusregisters wird dieses Flag genutzt, um Unterbrechungen des normalen Programmflusses durch externe Anforderungen zu unterbinden bzw. zu ermöglichen.

  Das einfache I-Flag kann auch durch mehrfache Flags ersetzt werden. In diesem Fall wird der Zustand der Flags als Priorität gedeutet, die eine Unterbrechnung mindestens besitzen muss, um bearbeitet zu werden.

  Das I-Flag wird nicht durch bedingte Sprungbefehle ausgewertet, sondern im Rahmen eines globalen Konzepts für Prozessoren mit Unterbrechungsfähigkeit. Setzen bzw. Löschen dieses Bits erfolgt durch entsprechende Befehlsworte.

Neben den hier genannten Beispielen existieren in den verschiedenen Prozessorarchitekturen weitere Flags, insbesondere zur Erweiterung der arithmetischen Kapazitäten und zur Einführung bedingter Befehle (so genannte Predicate-Flags, → 6.2 und 7.2).

2.5.3 Stackpointer

Ein *Stack* (*Stapelspeicher*, *Kellerspeicher*) ist ein wichtiges Hilfsmittel zur Ermöglichung von Unterprogrammen und Unterbrechungen. Dieser Speicherbereich ist – im Gegensatz zum allgemeinen Speicher – nicht mit einem *wahlfreien* Zugriffsver-

fahren zu erreichen, das Zugriffe in beliebiger Adressfolge ermöglicht, sondern der nächstfolgende Transfer von oder zum Stack ist von der Adresse des vorhergehenden abhängig.

Es existieren aktuell zwei Varianten der Stackrealisierung. Die schnellere Variante benutzt einen ausgezeichneten Speicherbereich, der direkt im Prozessor implementiert ist und durch keinen anderen Zugriff erreicht werden kann. Diese Form des Stacks fällt meist recht klein aus, da Siliziumfläche auf dem Prozessor-Die benötigt wird. Andererseits ist der Zugriff sehr schnell, auch für Mehrfachzugriffe, da das (langsame) externe Bussystem umgangen wird.

Die zweite Variante benötigt einen Stackpointer im Prozessor und bildet jeden Stackzugriff auf den (externen) Speicher ab. Diese Variante ist die meist genutzte, da sie ressourcenschonend wirkt. Nachteilig sind die Geschwindigkeit sowie die immanenten Fehlermöglichkeiten, da der Speicherbereich des Stacks auch durch andere, 'normale' Datenzugriffe erreichbar ist.

Der Stack arbeitet in beiden Varianten nach dem LIFO-Prinzip (*Last In First Out*), das sich mit einem Tellerstapel vergleichen lässt: Lese- und Schreibzugriffe beziehen sich hierbei immer auf den obersten Teil des Stapels (*Top of Stack*). Schreibzugriffe zielen auf das nächste freie Element, Lesezugriffe auf das oberste besetzte Element, das damit gleichzeitig für neue Schreibzugriffe freigegeben wird.

Es sind drei Aktionen, die während der Bearbeitung eines Programms den Stack nutzen:

- *Unterprogrammsprünge (Funktionsaufrufe)*: Die Rücksprungadresse (= aktueller PC-Wert + "1") wird auf dem Stack gespeichert und beim Rücksprung wieder in den Programmzähler geladen.
- *Lokale Variablen (z.B. in Unterprogrammen)*: In imperativen oder fuinktionalen Sprachen wie C, Pascal mit intensiver Nutzung von Funktionen werden lokale Variablen, die einen beschränkten Gültigkeitsbereich besitzen, auf dem Stack angelegt und bei Verlassen der Routine wieder verworfen.
- *Unterbrechungsanforderungen* (Interrupt Requests, → 2.9): Sowohl die Rücksprungadresse nach Behandlung der IRQs als auch (falls erforderlich) gerettete Registerinhalte und das Prozessorstatusregister werden auf dem Stack zwischengespeichert.

Für einen explizit formulierten Datentransfer zwischen einem Datenregister (hier mit Akkumulator bezeichnet) und dem Stack werden meist besondere Transferbefehle angeboten, die mit PUSH und POP bezeichnet werden:

```
PUSH A    A → (--SP)
POP A     (SP++) → A
```

Fehler! Keine gültige Verknüpfung.

Bild 2.11: Stackoperation Pop A

PUSH A schreibt den Inhalt des Akkumulators in die Speicherzelle, auf die der Stackpointer SP zeigt, nachdem er zuvor dekrementiert wurde (Predekrement). POP A kopiert hingegen zunächst den Inhalt der Speicherzelle, auf die SP zeigt, in den Akkumulator und erhöht anschließend SP um 1 (Postinkrement, Bild 2.11). In diesem Beispiel 'wächst' der Stack nach unten (zu kleineren Adressen hin), wenn Daten abgelegt werden.

Die Adressierung der Speicherzelle erfolgt damit relativ zum Stackpointer (→ Bild 2.9). Dieses Register muss bei Start des Rechners nach Reset entweder durch die Hardware oder durch einen kurzen Abschnitt in der Software auf einen definierten Anfangswert gesetzt werden, um einen definierten Zustand zu erhalten und ggf. sporadisch auftretende Programmfehler zu verhindern. Zudem kann es durch einen zu kleinen Stack zu Abstürzen kommen, indem mehr Speicherplatz genutzt werden soll, als real vorhanden ist. Dies resultiert je nach Architektur in einem 'Wrap Around', der Stackpointer fängt wieder am Anfang (also z.B. bei hohen Adressen) an. Man spricht dann von *Stack Overflow*.

2.5.4 Datenregister

Grundsätzlich können zwei verschiedene Implementierungen von Datenregistern innerhalb des Rechenwerks unterschieden werden. In einer *stackorientierten (Rechenwerk-) Architektur* sind die Register ähnlich wie im allgemeinen Stack in einer LIFO-Struktur angeordnet, so dass auch für die internen Register ein Stackpointer existiert, der auf das aktuelle Register zeigt. Diese Architektur, die die umgekehrte polnische Notation (Reverse Polnish Notation, RPN) für die Formulierung von Algorithmen benötigt, wird derzeit kaum noch verwendet, da die Unterstützung durch Hochsprachen (z.B. Forth) sehr gering ist.

Die andere Implementierung organisiert die Datenregister in einem *Registerblock*, bei einer größeren Anzahl auch *Registerfile* genannt, mit einer wahlfreien Adressierung innerhalb des Rechenwerks. Die Wahlfreiheit des Zugriffes bedeutet, dass die Befehlsworte aus dem Programmspeicher, die eine Aktion im Rechenwerk beinhalten, in beliebiger Reihenfolge stehen dürfen (abgesehen von den Abhängigkeiten des Datenflusses).

Das Befehlsformat eines Prozessors (→ 2.6) kann 1-, 2- und 3-Adressbefehle beinhalten, in Ausnahmefällen auch mehr. Mit den Adressen sind die Quellen und das Ziel der Operation bezeichnet, so dass beispielsweise bei einer üblichen Addition zwei Quellen mit einem Ziel über die additive Verknüpfung verbunden sind. Die Variabilität in der Anzahl der Adressen begründet letztendlich die konkrete Ausführung der internen Daten- und Adressierungsbusse.

2.5.5 Klassifizierung von Prozessoren gemäß Operandenzugriff

Die Form des Operandenzugriffs für eine Operation ist sehr charakteristisch für Prozessorklassen, da sie das *interne Speichermodell* bestimmt. Aus diesem Grund wird diese Klassifizierung häufig genutzt.

Dieses interne Speichermodell trifft Aussagen darüber, wie die für Datenoperationen, etwa eine Addition, notwendigen Operanden vor und nach der Operation im Zugriffsbereich des Prozessors liegen. Hierbei können drei grundsätzliche Methoden unterschieden werden:

- Im Rahmen einer *Stack-Architektur* werden die Quell- und Zieloperanden auf dem Stack gespeichert und dort zugegriffen. Der Zugriff seitens des Prozessors kann hierbei sehr implizit erfolgen, etwa in einer exakt festgelegten Reihenfolge (umgekehrt polnische Notation).
- Die *Akkumulator-Architektur* (→ Bild 2.9) zeichnet sich hingegen durch einen, ggf. wenige Akkumulatoren als zentrale(s) Register aus. Dies bedeutet, dass in der Befehlscodierung bereits implizit festgelegt wird, welches Register als Quelle und ggf. Ziel zur Operation genutzt wird.
- Die *(General-Purpose-)Register-Architektur* wiederum bezieht ebenfalls mindestens einen Operanden aus einem internen Prozessorregister, dieses Register muss jedoch im Befehl explizit angegeben werden. In dieser Prozessorklasse werden folgende Unterklassifizierungen im Hinblick auf RISC-Architekturen (→ 4) unterschieden:
  - *Load/Store-Architekturen* besitzen ein ausschließliches Interface zwischen internen Prozessorregister und externem Speicher zum Datenaustausch zwischen diesen. Operationen zur Verknüpfung von Daten wirken ausschließlich zwischen Registerinhalten.
  - *Register-Speicher-Architekturen* stellen eine Erweiterung der Akkumulator-Architektur dar. Bei Befehlen dieser Prozessorsubklasse wird sowohl das Register als auch die Speicheradresse explizit angegeben.

Diese Prozessorklassen besitzen Vor- und Nachteile, die in Tabelle 2.1 zusammengefasst sind. Es bestehen daneben noch weitere Möglichkeiten zur Klassifizierung der Prozessoren, etwa durch Unterscheidung, ob der Zieloperand explizit angegeben werden kann oder implizit mit einem der Quelloperanden übereinstimmt.

Typ	Vorteile	Nachteile
Stack	Keine explizite Angabe von Operanden notwendig, daher minimale Opcodelänge für Arithmetikbefehle	Umfangreiche Austauschoperationen von und zum Stack notwendig, d.h. ggf. wachsende Codegröße
Akkumulator	Teilweise implizite Operandenangabe möglich (1. Quelloperand, Zieloperand) Einfachste Architektur, Verzicht auf Stack möglich	Das zentrale Register wird sehr stark genutzt, daher ggf. Engpass mit Notwendigkeit zur Zwischenspeicherung
Register-Speicher	Daten können ohne Ladezugriff für Operationen genutzt werden	Ladezugriffe im Speicherbereich benötigen wesentlich mehr Zeit, daher ggf. Verzögerung gegenüber Registerzugriff
Register-Register	Schnellstmögliche Operationen durch Registeroperanden	Explizite Ladezugriffe notwendig, Codegröße wächst

Tabelle 2.1: Vor- und Nachteile von Prozessorklassen

2.6 Befehlssatz

Um ein allgemeines, berechenbares Problem in einem Mikroprozessor zu lösen, muss dieser einen Befehls- bzw. Instruktionssatz anbieten, mit dem das Problem beschrieben werden kann. Dieser Befehlssatz muss folgende Dinge beinhalten:

- Eine ausreichende Anzahl an Operationen, um allgemeine Algorithmen formulieren zu können.
- Adressierungsarten, die die Adressierung der zum Befehl gehörenden Operanden ermöglichen.

Es wurde bewiesen, dass im Prinzip ein einziger (zusammengesetzter) Befehl genügt, um dem Kriterium der Universalität zu genügen. Dieser Befehl heißt

```
SUBBEQ X, Y, Z
```

(subtract and branch if equal) und besteht darin, X = X-Y auszuführen und an Z springen, wenn das Ergebnis 0 ist. Die Universalität wird hierbei durch die Operation Subtraktion in Verbindung mit der bedingten Verzweigung auf Z bzw. dem sequenziellen Weiterlauf im Programmfluss bei einem Ergebnis ungleich 0 erreicht, wobei die Adressierung insbesondere von Y ebenfalls eine Rolle spielt: Y kann die Adresse einer Speicherstelle oder eine direkte Konstante bedeuten.

In der Praxis existiert kein Mikroprozessor mit nur einem Befehl, minimal ausgelegte Befehlssätze liegen bei ca. 30 Instruktionen. Im Rahmen der CISC-Philosophie (Complex Instruction Set Computing) wurden sogar Befehlssätze mit möglichst vielen, dann auch komplexen Befehlen favorisiert, während die RISC-Philosophie (→ 4) von einfachen, reduzierten Befehlssätzen in Kombination mit schneller Ausführung der einzelnen Instruktion ausgeht.

Für die Befehlssätze werden im Allgemeinen folgende Gruppen von Instruktionen genutzt:

1. Die *Gruppe der Transferbefehle* beinhaltet alle Datenoperationen, die eine unveränderte Kopie in andere Speicherzellen oder Register bewirken. In diese Gruppe gehören Load/Store-Befehle zur Datenkopie zwischen externem Speicher und internem Register, Austauschbefehle zwischen den internen Registern sowie Stackbefehle.

   Beispiele hierfür sind LD (Load) und ST (Store), Mov (Move), Push und Pop.

2. Die *Gruppe der Flagbefehle* beinhaltet alle Instruktionen zur direkten Manipulation der Flags im Statusregister. Dies ist beim Carry- und beim Interrupt-Flag notwendig, um z.B. Vorbelegungen für Addition/Subtraktion vorzunehmen oder Programmteile vor Unterbrechungen zu schützen.

   Beispiele sind SEC (Set Carry), CLC (Clear Carry), SEI (Set Interrupt Flag) und CLI (Clear Interrupt Flag).

3. Die *Gruppe der arithmetisch/logischen Instruktionen* umfasst alle Datenmanipulationsbefehle. Hierzu zählen Addition, Subtraktion, Multiplikation und Division für die arithmetische Gruppe sowie logische Verknüpfungen wie AND, OR und XOR für die logische Gruppe. Weiterhin werden Rotations- und Shiftbefehle, die die Bitwerte eines Register um eine oder mehrere Stufen nach links/rechts verschieben oder rotieren lassen, zu dieser Gruppe gezählt.

4. Die *Gruppe der Kontrollflussbefehle* beinhaltet die unbedingten Sprungbefehle, die bedingten Branchbefehle (Verzweigungen), den Unterprogrammaufruf sowie die Rücksprunginstruktionen für Unterprogramme und Interrupt-Service-Routinen. Auch Befehle zum Aufruf von Software-Interrupts gehören in diese Kategorie.

   Diese Befehle verändern generell den Kontrollfluss eines Programms, indem sie den Program Counter neu setzen und somit von der sequenziellen Folge der Programmbearbeitung abweichen.

5. Weitere Befehle wie NOP (No Operation) werden meist zur Gruppe der sonstigen Befehle zusammengefasst. Hierunter fallen auch Instruktionen, die den Prozessor komplett anhalten (WAIT, STOP).

2.7 Adressierungsarten im Von-Neumann-Prozessor

Wie bereits im vorangegangenen Abschnitt diskutiert benötigt die Bearbeitung von Befehlen im Von-Neumann-Prozessor Operanden, in denen der Bezug zu den Daten hergestellt wird. Hierzu existiert eine Vielzahl von Möglichkeiten, auf diese Daten zuzugreifen. Grundsätzlich sollte dabei deutlich sein, dass – abgesehen von einigen Ausnahmen wie NOP (No Operation), WAIT und STOP – alle Befehle Operandenzugriff in irgendeiner Form benötigen, häufig sogar mehrere.

Eine grobe Einteilung der Operandenadressierung kann dadurch eingeführt werden, dass man implizite und explizite Angaben in der Befehlscodierung vorsieht. Eine *implizite Codierung* liegt dann vor, wenn der Operand bereits eindeutig in dem Befehl bestimmt wird. Dieser Fall liegt beispielsweise in Befehlen wie CLC (Clear Carry Flag) vor.

In allen anderen Fällen müssen die Operanden jedoch *explizit* angegeben werden. In dieser Kategorie kann wiederum grob dahingehend eingeteilt werden, dass die Daten in internen Registern des Prozessors oder im externen Speicher- sowie Ein-/Ausgabebereich liegen.

Für eine weitergehende Analyse ist dann die Interpretation der Daten notwendig. Gemäß dem Von-Neumann-Modell können die Daten nicht a priori in ihrer syntaktischen Bedeutung identifiziert werden; dies erfolgt ausschließlich im Kontext des Programmflusses, so auch im Sinne der Adressierung. Beispielsweise lauten die Assemblercodierungen für die beiden Befehle

```
LD  AX, #10
LD  AX, 10
```

nahezu identisch. Im ersten Fall soll die Zahl 10 in den Akkumulator AX geladen werden, was bedeutet, dass der Prozessor nach der Decodierung des Befehls einen Operandenzugriff auf die zu kopierende Zahl benötigt. Im zweiten Fall ist der eigentliche Operand 10 nicht als Zahl zu interpretieren, sondern als *Adresse* für die Speicherstelle, wo das Datum zur Kopie in den Akkumulator zu finden ist. Konsequenterweise sind nun auch zwei Operandenzugriffe notwendig, da der erste nur die Adresse und damit die Referenz für den zweiten Zugriff lädt und das benötigte Datum erst hiermit erhältlich ist.

Für eine formale und informelle Beschreibung der wichtigsten Adressierungsarten ist folgende Definition sehr hilfreich:

Definition 2.1

Für die formale Beschreibung des **Operandenzugriffs** seien folgende Abkürzungen und Schreibweisen definiert [3]:

reg: PC	Internes Register wird adressiert, PC als Program Counter (spezielles Register)

adr:	Referenzadresse auf eine externe Speicherzelle im Speicher- bzw. Ein-/Ausgabebereich
dat:	Datum (Inhalt einer Speicherzelle oder eines Registers)
()	Indirektion (Inhalt einer Speicherzelle oder eines Registers wird als Adresse interpretiert)
++	Inkrement (Pre- oder Postinkrement, je nach Stellung zum Operanden hin)
--	Dekrement (Pre- oder Postdekrement, je nach Stellung zum Operanden hin)
→	Kopieroperator (Operand links wird auf Operand rechts kopiert)

Die nachfolgende Auflistung der Adressierungsarten darf keineswegs als vollständig betrachtet werden, da häufig noch spezielle Formen der Adressbildung hinzukommen, auf die im Rahmen dieses Skripts nicht detailliert eingegangen werden kann. Hierzu zählen beispielsweise Adressbildungen für Algorithmen der digitalen Signalverarbeitung (Bit-Reverse Shuffling). Für weitere Studien sei hierzu [7] empfohlen.

2.7.1 Adressierungsarten im Einzelnen

Implizite Adressierung

Wie bereits erwähnt stellt die *implizite Adressierung* (*implicit*) eine einfache, dennoch weitverbreitete Adressierungsart dar. Der Operationscode selbst bestimmt eindeutig das Ziel des Zugriffes, sodass die Load-Phase für diese Adressierungsart entfallen kann.

Beispiel: `SEC` (Set Carry), `LDAA` (Load Accumulator A)

Formale Beschreibung: ‘1’ → (reg: Status.Carryflag)

(...) → (reg: Akkumulator)

Im letzteren Beispiel allerdings ist nur ein Teil der gesamten Adressierung implizit. Der LDAA-Befehl benötigt zwei Operanden, eine Quelle und ein Ziel. Das Ziel ist implizit im Operationscode enthalten, die Quelle hingegen muss anschließend noch explizit angegeben werden.

Registerdirekte Adressierung

Ist im Gegensatz zum Beispiel `LDAA` der Registeroperand nicht implizit festgelegt, so muss das Register durch Angabe explizit bestimmt werden. Dies wird *registerdirekte Adressierung* (*register direct, implied*) genannt. Die Codierung des jeweiligen Registers ist meist so kompakt möglich, dass dies als Teil des Befehlscodes erfolgen kann. Hierdurch kann wiederum die Load-Phase entfallen oder durch die interne Adressierung in ihrem Zeitbedarf stark minimiert werden.

Beispiel: `MOV AX, BX;` (MOVE, Datenkopie)

Formale Beschreibung: (reg: BX) → (reg: AX)

Dieses Beispiel zeigt eine zweifache registerdirekte Adressierung.

Registerindirekte Adressierung

Mit Hilfe eines Adressregisters lassen sich innerhalb des Prozessors Referenzen speichern, die zur Adressierung der eigentlichen Speicherstelle genutzt werden können. Dies ist eine Indirektion auf Basis von internen Registern, die als *registerindirekte Adressierungsart* (*register indirect*) bezeichnet wird.

Beispiel: `MOV AX, (BP);` (MOVE, Datenkopie)

Formale Beschreibung (adr:(reg:BP)) → (reg: AX)

Die Verwendung des zweiten Operanden, (BP) (Base Pointer), erfolgt in diesem Fall in der beschriebenen Art. Der Datenwert in BP wird in der Decode-Phase des Befehlsablaufs als Referenzadresse auf den Speicherbereich interpretiert und während der Load-Phase als Adresswert am Adressbus ausgegeben. Der Inhalt der entsprechenden Speicherzelle wird in das AX-Register kopiert.

Unmittelbare Adressierung

Programmkonstanten stehen bereits zur Übersetzungszeit fest und können daher auch im Programmteil gespeichert werden. Diese Programmkonstanten werden zur Befehlsbearbeitung dann als Datenwert direkt in ein Register geladen oder anderweitig verwendet. Die Interpretation eines Wertes im Programmspeicher als konstantes Datum wird als *unmittelbare Adressierung* (*immediate*, *literal*) bezeichnet.

Beispiel: `MOV AX, #10;` (Lade in AX die Zahl 10)

Formale Beschreibung: dat: (reg: PC++) → (AX)

Die Adressierung des zweiten Operanden ist unmittelbar. Der Zugriff auf diesen Operanden erfolgt während der Load-Phase, wobei die Adresse durch den Inhalt des Program Counters festgelegt wird. Im Anschluss hieran muss der PC inkrementiert werden.

Speicherdirekte Adressierung

Das auf den Befehlscode folgende Maschinenwort wird bei dieser Adressierungsart als Referenz, d.h. Adresse auf die externe Speicherstelle gewertet, in der das eigentliche Datum steht. Die Adresse muss daher zur Übersetzungszeit bekannt sein, der Inhalt nicht, so dass die *speicherdirekte Adressierung* (*direct*) dem Zugriff auf einfache Variablen z.B. aus Hochsprachen entspricht.

Beispiel: `MOV DX, 200h;` (Kopiere in DX den Inhalt der Speicherstelle 200h, dezimal 512)

Formale Beschreibung: (adr: (reg: PC++)) → (DX)

Diese Adressierung hat zur Folge, dass zur Load-Phase zwei Speicherzugriffe notwendig sind. Der erste Zugriff lädt die Adresse in ein internes Adressierungsregister, der zweite Zugriff erfolgt dann auf das eigentliche Ziel, z.B., um das Datum in den Prozessor zu kopieren.

Speicherindirekte Adressierung

Eine nochmalige Indirektion liegt bei der *speicherindirekten Adressierung* (*indirect*) vor. Das dem Befehlscode folgende Maschinenwort wird nunmehr als eine Adresse interpretiert, die auf eine Speicherstelle zeigt, die wiederum eine Adresse enthält. Die zweite Referenz weist dann schließlich auf das Datum.

Beispiel: `ADD BX, (100h);` (Addiere zu BX den Inhalt der Speicherstelle, auf die die Adresse verweist, die an der Speicherstelle 100h, dezimal 256 gespeichert ist)

Formale Beschreibung: (reg: BX) + (adr: (adr: (reg: PC++))) → (reg: BX)

Die erweiterte Indirektion ergibt einen weiteren Zugriffszyklus während der Load-Phase, da zweimalig auf Adressen und erst im dritten Zyklus auf das Datum zugegriffen werden kann. Als wesentliche Konsequenz ist eine Verlängerung der Load-Phase zu erwähnen, sodass von dieser Adressierungsart häufig abgesehen wird.

Im Vergleich zu Hochsprachen werden indirekte Adressierungen bei Referenzierungen über Zeiger (Pointer) benötigt. Um die Verlängerung durch zusätzliche Speicherzugriffszyklen zu vermeiden, wird jedoch häufig auf die registerindirekte Adressierung zurückgegriffen.

Indizierte Adressierung

Die *indizierte Adressierung* (*indexed*) wird meist als zusätzliche Modifizierung einer anderen Adressierung eingeführt, beispielsweise als speicherdirekte Adressierung mit Index. Die Ermittlung der Zugriffsadresse erfolgt dementsprechend zuerst wie in dem jeweiligen Basisverfahren, vor dem letztendlichen Datenzugriff wird ein in einem Register befindlicher Indexwert zur Adresse hinzugezählt.

Beispiel: `MOV AX, 100h (BX);` (Kopiere in AX das Datum, das sich an der Stelle 100h zuzüglich des momentanen Inhalts von BX befindet)

Formale Beschreibung: (adr: (reg: PC++)+(reg: BX)) → (reg: AX)

Die indizierte Adressierung ist für Tabellenbearbeitungen gut geeignet, da sich die Basisadresse hierbei nicht ändern muss, sondern nur der Zugriffsindex variiert wird. Die Anzahl der Speicherzyklen wird nicht erhöht, falls sich der Index wie in diesem Beispiel in einem internen Register befindet.

Absolute Adressierung

Die *absolute Adressierung* (*absolute*, auch *absolute long*) wird im Rahmen von Kontrollflussbefehlen wie Sprungbefehlen benutzt und besitzt große Ähnlichkeiten zur unmittelbaren Adressierung für Datenzugriffe. Das auf das Befehlswort folgende Maschinenwort wird in der entsprechend benötigten Wortbreite als absolute Adresse aufgefasst, mit der der Program Counter neu geladen wird.

Beispiel: `JUMP C000h;` (Springe an die Stelle C000h, dezimal 49152)

Formale Beschreibung: adr: (reg: PC++) → (PC)

Die absolute Adressierung benötigt exakt einen Speicherzugriff, um den Operanden zu laden. Der Ablauf am Bus entspricht damit der unmittelbaren Adressierung für Daten, wobei die Interpretation des Operanden im Rahmen von Kontrollflussbefehlen natürlich abweicht.

Eine Variation liegt in der *verkürzten absoluten Adressierung* (*absolute short*) vor. Hier wird nur ein Teil der PC-Adresse ersetzt, während die oberen Bits ab einer fest definierten Adresse erhalten bleiben. Der Vorteil dieser Adressierung liegt darin, dass die Speicherung der Sprungadresse weniger Platz benötigt. Nachteilig ist, dass mithilfe der verkürzten absoluten Adressierung nur ein Teil des Programmspeichers erreichbar ist.

Absolut indirekte Adressierung

Das auf das Befehlswort folgende Maschinenwort wird im Rahmen der *absolut indirekten Adressierung* (*absolute indirect*) als diejenige Adresse interpretiert, an der der Operand gespeichert ist. Dies bedeutet die gleiche Indirektion wie im Fall der speicherdirekten Adressierung.

Beispiel: `JUMP (400h);` (Springe an die Adresse, die an der Speicherstelle 400h, dezimal 1024 gespeichert ist)

Formale Beschreibung: adr: (adr: (reg: PC++)) → (PC)

Absolut indirekte Adressierung wird beispielsweise für Sprungtabellen benötigt. Die Tabelle enthält dabei die Zieladressen für die weiteren Programmteile, der Sprung dorthin kann über den Inhalt der Speicherzelle berechnet werden.

Relative Adressierung

Die *relative Adressierung* (*relative*) wird für Kontrollflussbefehle, insbesondere Verzweigungsbefehle genutzt. Das auf den Befehlscode folgende Maschinenwort wird so interpretiert, dass der Program Counter von seinem momentanen Wert um den dort angegebenen Wert variiert wird, wobei das Maschinenwort als vorzeichenbehafteter Integerwert zum PC addiert wird.

Beispiel: `BRA 40;` (Verzweige um 40 Adressen)

Formale Beschreibung: adr: {reg: PC + adr: (reg: PC++)} → (reg: PC)

Das tatsächliche Format im Operanden ist von dem jeweiligen Prozessor abhängig, meist wird ein 8-Bit-Format gewählt, um zwecks Verkürzung der Zugriffszeiten einen möglichst kurzen Operanden im Speicher zu halten. Die Addition ist vorzeichenbehaftet, um Vor- und Rückwärtssprünge zu gewährleisten. Die relative Adressierung ist aufgrund des Lokalitätsprinzips von Programmen außerordentlich effektiv.

2.7.2 Minimaler Satz von Adressierungen

Insbesondere für RISC-Konzepte, die neben der Reduktion der Instruktionsvielfalt auch die Adressierungsarten beschränken, ist es interessant, welche Sätze von Adressierungsarten als vollständig gelten. Die implizite Adressierung ist dabei immer integriert, sie ergibt sich sozusagen automatisch.

Für Kontrollflussbefehle wird ausschließlich die (lange) absolute Adressierung benötigt, wahlweise auch die absolut indirekte Adressierung. Die Nutzung von relativer Adressierung ist nicht zwingend, in der Praxis jedoch sehr nützlich, da diese Adressierung eine kompakte Speicherung ohne zusätzlichen Ladeaufwand in die Adressregister erlaubt.

Für Datenadressierungen ist in jedem Fall die unmittelbare Adressierung (Datenkonstanten) notwendig, ferner eine Adressierung mit Speicheradresse. Hierbei ist die direkte Adressierung allein ungünstig, weil so z.B. Datenfelder (Arrays) in Schleifen nur über selbstmodifizierenden Code angesprochen werden können. Günstiger ist die Wahl der indizierten Adressierung oder der Register-indirekten Adressierung, diese bieten gute Möglichkeiten zur Arraybearbeitung.

Im in Kapitel 4 vorgestellten RISC-Modell MPM3 wird die Kombination {unmittelbar, Register-indirekt, relativ, absolut-indirekt, implizit} gewählt. Diese Menge ist nicht ganz minimal, stellt aber einen guten Kompromiss zwischen Minimalität und praktischen Einsatzpunkten dar.

2.8 Phasen der Befehlsbearbeitung

Zum weiteren Verständnis der Befehlsbearbeitung werden alle Befehle in vier Klassen eingeteilt:

- Kopierbefehle zwischen internen Registern
- Arithmetische und logische Befehle
- Speichertransferbefehle zwischen internen Registern und externer Speicherzelle
- Kontrollflussbefehle im weitesten Sinn, wie Jump, Branch, Stop, Wait, No Operation etc.

Eine Analyse der Abläufe in diesen Befehlen bzw. Befehlsklassen kann anhand der folgenden Aufzählung erfolgen:

1. Die *Fetch*-Phase: Der erste Zugriff im Ablauf der Befehlsbearbeitung erfolgt, um den eigentlichen Befehlscode in interne Register zu laden. Maßgebend für die Adresse ist der aktuelle Stand des Programmzählers, der – wie bereits erwähnt – am Ende dieser Phase inkrementiert wird. Diese Phase ist für alle Befehle gleich.
2. Die *Decode*-Phase: Der im internen Register geladene Befehl wird nun decodiert, um weitere Aktionen entsprechend einleiten zu können. Diese Decodierung umfasst die Analyse des Befehls selbst, beispielsweise um Informationen zur Befehlsklasse sowie zu Art und Anzahl der benötigten Operanden und deren Adressierungsformat zu erhalten.
3. Die *Load*-Phase: Die Operanden, die zur Ausführung des Befehls notwendig sind, werden ggf. in interne Register geladen. Die Phase ist bereits sehr von der Interpretation des Befehls abhängig:
   - Bei Befehlen ohne Operanden ist diese Phase selbstverständlich leer
   - Operanden in internen Registern, etwa für ein `Move` zwischen zwei Registern, müssen nicht in interne Datenregister zur weiteren Befehlsbearbeitung geladen werden, sie können später direkt genutzt werden, so dass auch in diesem Fall diese Phase leer ist.
   - Externe Operanden können – dies wurd im vorangegangenen Abschnitt diskutiert – auf verschiedene Arten adressiert werden, wobei nunmehr ein mehrfacher Zugriff am Bus durchlaufen werden könnte. Im einfachsten Fall stehen die notwendigen Daten in den Speicherstellen, die unmittelbar auf den Befehl folgen, sodass die Adressierung durch den Programmzähler bei gleichzeitigem Inkrement erfolgt.
4. Die *Execute*-Phase: Nach dem Laden aller Operanden kann der Befehl ausgeführt werden. Maßgebend hierfür ist der geladene Operationscode im internen Befehlsregister.

   Die Execute-Phase unterscheidet sich erheblich für die vier Befehlsklassen und muss für jeden Befehl gesondert definiert werden. Arithmetisch-logische Befehle werden in der ALU ausgeführt, während Kontrollflussbefehle die Register im Leitwerk beeinflussen.
5. Die *Write-Back*-Phase: Die letzte Phase besteht in dem Zurückschreiben des Ergebnisses, beispielsweise in den Akkumulator, in andere interne Register oder in den externen Speicher. Diese Phase steht in engem Zusammenhang mit der Execute-Phase, sodass die Trennung gelegentlich willkürlich erscheinen kann.

Die Unterteilung in fünf auszuführende Phasen im Rahmen der Befehlsbearbeitung ist durchaus variierbar, und zwar begründbar in beide Richtungen. Weniger Phasen sind gleichbedeutend mit einer geringeren Anfälligkeit gegenüber Datenabhängigkeiten im Kontrollfluss (→ [3, Kap. 6], → 4.4), bei mehr Phasen ist die einzelne Phase einfacher und schneller durchführbar, so dass eine Beschleunigung der

Ausführung die Folge sein kann. Die Einteilung in die Ausführungsphasen ist in jedem Fall maßgeblich vom Design des Prozessors bestimmt.

2.9 Interrupt-Konzept im Von-Neumann-Prozessor

Die bisherige Darstellungen der Details umfasst den Von-Neumann-Prozessor in seiner ursprünglichen Definition. Im Laufe der Weiterentwicklung, insbesondere in Richtung Prozessrechner, hat sich jedoch die Notwendigkeit zu verschiedenen Erweiterungen ergeben, die die Kopplung zwischen Rechner und Außenwelt betreffen.

Definition 2.2:

Prozessoren, die in Prozessen mit Kontakt zur Außenwelt eingesetzt werden, müssen auf Ereignisse reagieren können, die in keinem synchronierbaren Zeitverhältnis zum sequenziellen Programmverlauf stehen. Diese Ereignisse werden **Interrupt Requests** (Unterbrechungsanforderungen)genannt, und die Erweiterung des Von-Neumann-Prozessors umfasst die Möglichkeit zur Reaktion auf derartiger externe Ereignisse.

Diese Definition enthält die Unterbrechungsanforderungen im engeren Sinn, die zur asynchronen Kopplung zwischen Außenprozessen und dem Rechner führen (können). Dieses Konzept ist durch die Einführung von **Software-Interrupts** und **Exception Handling** (*Ausnahmebehandlung*) erweitert worden.

Software-Interrupts sind durch spezielle Befehle (etwa `SWI`, Software Interrupt oder `TRAP`) möglich. Sie stellen eine gewollte, synchrone Unterbrechung des bisherigen Programmflusses dar, die wie die asynchronen Hardware-Interrupts behandelt werden. Software-Interrupts werden z.B. zur Kopplung von Programmen mit Betriebssystemaufrufen genutzt, da auf diese Weise nicht bekannt sein muss, an welcher Adresse im Programmspeicher diese Betriebssystemroutine gespeichert ist.

Ausnahmen entstehen, wenn im Programmfluss nicht-behebbare Rechenfehler (z.B. Division durch 0, Zugriff auf nicht vorhandene Speicherbereiche (→ 9.3) etc.) entstehen. Diese Unterbrechungen sind synchron, allerdings nicht gewollt. Die aufgerufene Unterbrechungsroutine soll in diesem Fall die entstandene Situation klären und den Rechner in einen sicheren Betriebszustand überführen.

Unabhängig von dem Typ der Unterbrechung werden diese alle im gleichen Konzept zur Unterbrechungsbehandlung integriert.

2.9.1 Konzept der Behandlung von Interrupts

Fast alle Prozessoren besitzen ein Möglichkeit zur Integration von Interrupts in ihre Programmierung und ihr Hardware-Interface, wobei sich zwischen den verschiedenen Prozessortypen große Gemeinsamkeiten im Konzept finden lassen. Unter

der Annahme einer einzigen Interruptquelle kann die Behandlung einer Unterbrechung sehr einfach in einen Hardware- und einen Softwareteil unterteilt werden.

Das eigentliche Interruptsignal ist zumeist als Bitleitung in den Prozessor ausgeführt, der im Allgemeinen in der Lage ist, das Signal zwischenzuspeichern. Man unterscheidet zudem flanken- und zustandssensitive Eingänge, wobei die Unterschiede im Folgenden irrelevant sind.

Das zwischengespeicherte Signal wird spätestens im Rahmen der nächstfolgenden Fetch-Phase ausgewertet, d.h., der gerade im Ablauf befindliche Befehl wird noch komplett ausgeführt, um einen sicheren Zustand der CPU zu gewährleisten. Bis zu diesem Zeitpunkt ist die Behandlung des Interrupts hängend (*pending*). Bild 2.12 zeigt die weitere Bearbeitung in dieser Phase unter Berücksichtigung der möglichen Unterbrechung.

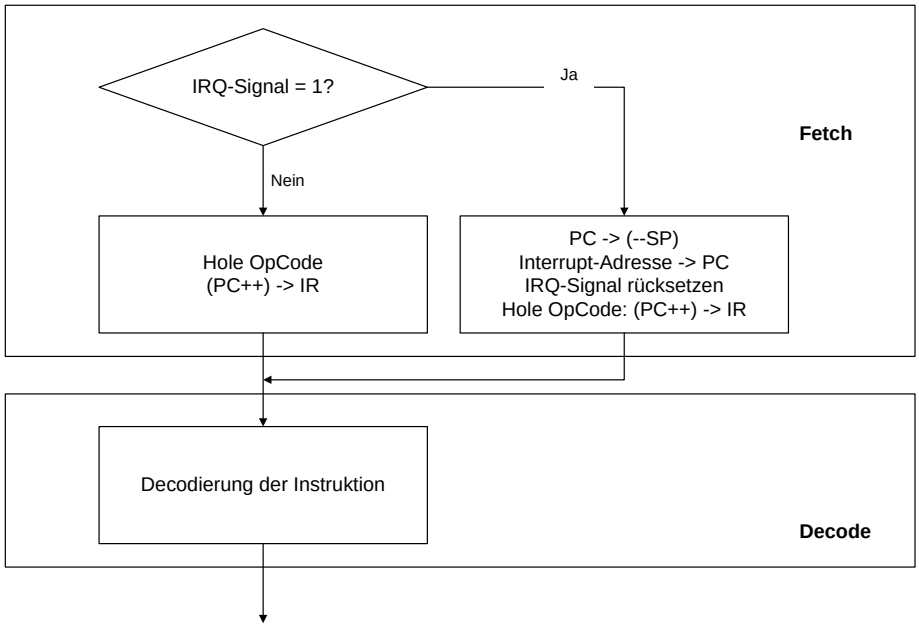


Bild 2.12: Erweiterung der Fetch-Phase für Interruptbehandlung

Ist das IRQ-Signal gesetzt, wird anstelle des Fetch auf den nächsten Operationscode der momentane Stand des Programmzählers PC gesichert, zumeist auf dem Stack. Dieses minimale Maß der Zustandssicherung ist notwendig, um am Ende der Interruptbehandlung den alten Bearbeitungsstand wiederherstellen zu können. Häufig werden an dieser Stelle weitere Register, beispielsweise das Statusregister, mit den Flags gespeichert.

Das IRQ-Signal wird ggf. an dieser Stelle zurückgesetzt und der Programmzähler mit einem für den Prozessor definierten Wert geladen. Dieser Wert zeigt auf die Stelle im Adressraum, an der sich die Softwareroutine zur eigentlichen Reaktion auf die Unterbrechung befindet.

Im Softwareteil wird die *Interrupt Service* Routine wie ein normales Programm durchlaufen. Der einzige Unterschied besteht ggf. in dem Rücksprungteil, wo die zwischengespeicherten Register restauriert werden müssen. Im einfachsten Fall entspricht dies dem Laden des Programmzählers mit dem zwischengespeicherten Wert. Bei automatischer Speicherung weiterer Register wird hier ein spezieller Befehl `RETI` (*Return from Interrupt*) benötigt, der diese bei Rücksprung wiederherstellt.

Dieses Basiskonzept kann auf zwei Arten erweitert werden:

1. Die Ausführung der Interrupt Service Routine sollte ggf. unterbindbar sein, um spezielle Softwareteile, deren zusammenhängende Ausführung kritisch ist, vor Unterbrechungen zu schützen. Diese Unterbindung wird durch ein Interrupt Enable Flag (bzw. Disable Flag, → 2.5.2 und 2.6) ermöglicht, das softwaregesteuert Phasen zusammenhängender Ausführung ermöglicht.
2. Bei mehreren Interruptquellen müssen zwei Probleme gelöst werden:

- Jedem Interrupt muss eine Startadresse für die Interrupt Service Routine zugeordnet werden.
- Wenn mehrere Interrupts gleichzeitig anstehen, muss eine Entscheidung getroffen werden, welcher Interrupt vorrangig bearbeitet wird.

  Hierfür sind mehrere Lösungsmethoden denkbar (für Details hierzu siehe [4]). Verschiedenen Interruptquellen werden als Erweiterung des einfachen Enable-Flags feste oder wechselnde Prioritäten zugeordnet, so dass eine Unterbrechung nur möglich ist, wenn die Priorität einen momentanen Wert überschreitet. Die zugehörige Serviceroutine wird in modernen Prozessorsystemen in einem mehrteiligen Interrupt-Acknowledge-Verfahren durch eine an die Quelle gebundene Vektornummer übermittelt.

2.9.2 Anwendungen

Für Interrupts existieren eine Vielzahl von Anwendungen, auch außerhalb der Kopplung zwischen externen Ereignissen und Programmablauf. Als wichtigste Gebiete können genannt werden:

Ein- und Ausgabe

Anwendungen im Bereich der Ein- und Ausgabe zählen zu den aus der Prozessrechnerwelt stammenden Aufgaben, die die Interruptverarbeitung generiert haben. Sie dienen zur Synchronisierung des Prozessors mit den angeschlossenen Peripheriegeräten. Alternativ wäre ein zyklisches Abfrageverfahren zu erwähnen (*Polling*), das jedoch zu einem großen zeitlichen Aufwand führen würde.

Betriebssysteme

Für Betriebssysteme stellen Interrupts zwei wichtige Mechanismen zur Verfügung. Zum einen kann in Multitaskingsystemen über Timer-gesteuerte Interrupts die Umschaltung zwischen Tasks realisiert werden. Ein Time-Slice-Verfahren gestattet die zeitliche Periodizität der Umschaltung mit einer gerechten Zuteilung von Rechenzeit.

Weiterhin können – wie erwähnt – Interrupts in einigen Prozessorarchitekturen auch durch Softwarebefehle ausgelöst werden. In diesem Fall ist zwar der ursprüngliche Anwendungsbereich der Service Routine – die Kopplung asynchroner Peripherie an den Prozessor – nicht mehr gegeben, dennoch bieten die Softwareinterrupts einen wesentlichen Vorteil gegenüber Unterprogrammaufrufen. Der Aufruf eines Interrupts erfolgt ausschließlich mit einer Nummer (dem Vektor), die Implementierungsdetails sind unwesentlich, insbesondere die Startadressen von Routinen werden durch den Prozessor bestimmt.

Fehlerbehandlung

Hard- und Softwarefehler, hierzu zählen etwa unerlaubte Zustände bei Operationen wie Division durch 0, unbekannte Operationscodes, können durch ein erweitertes Interruptsystem so behandelt werden, dass der Prozessor weiterhin in einem definierten Zustand bleibt bzw. diesen wieder erreicht. Diese besonderen Interrupts werden häufig als *Exceptions* (Ausnahmen) oder *Traps* (Fallen) bezeichnet und mit besonderen Vektornummern sowie hoher Priorität ausgestattet.

Die Trap Service Routine muss sich durch eine hohe Stabilität gegenüber den Fehlerquellen auszeichnen, d.h., es müssen alle fehlererzeugenden Zustände mit Sicherheit beseitigt werden.

2.9.3 Notwendige Erweiterungen im Prozessor

Wie aus Bild 2.12 bereits deutlich wurde, muss ein Prozessor zur Integration von Interrupts drei Erweiterungen in der Hardware bieten:

- Ein Speicherelement sollte das von außen auftreffende Signal den Spezifikationen entsprechend zwischenspeichern. Dieses Speicherelement kann aus einem rücksetzbaren D-Flipflop bestehen. Das Rücksetzen muss durch den Prozessor innerhalb der Service-Routine erfolgen.
- Die Fetch-Phase des Prozessors muss entsprechend erweitert werden, um die definierte Detektion einer Unterbrechung und den Übergang in die Service Routine zu gewährleisten. Diese Änderungen sind im Leitwerk zu finden, da hier die Ablaufsteuerung für die ersten Befehlsphasen integriert ist.
- Eine Erweiterung des Statusregisters (→ 2.5.2) durch das Interrupt Enable Flag ermöglicht das programmgesteuerte Verzögern der Bearbeitung bei zeitkriti-

schen Routinen. Dieses Flag muss durch besondere Befehle, etwa `CLI` und `SEI` (Clear bzw. Set Interrupt Enable Flag, → 2.6), setz- und löschbar sein.

Für die Integration einer einfachen Interruptquelle kann noch auf die Integration eines Stacks verzichtet werden. In diesem Fall muss der momentane Stand des Programmzählers anderweitig gespeichert werden, etwa in einem speziellen Schattenregister. Zusätzlich darf der Interrupt zum Zeitpunkt der Behandlung nicht ein zweites Mal auftreten, da ansonsten der gespeicherte Wert überschrieben werden würde.

Neben diesen Basiselementen sind für die weiterführenden Konzepte erhebliche Erweiterungen des Prozessors notwendig. Insbesondere sollte bei mehrfach möglichen Unterbrechungen nicht mehr auf einen Stack verzichtet werden. Die Einteilung in Prioritäten hingegen, ggf. weiterführende Maskierungen und die Bestimmung der Vektornummer, wird häufig in besondere Einheiten, den Interrupt-Request-Controllern integriert. Diese Controller werden als Peripherieelemente betrachtet und sind damit nicht mehr Bestandteil des Prozessors.

3 Klassifizierungssysteme für Prozessoren

Prozessoren zur Ausführung sind mittlerweile seit über 50 Jahren bekannt, sodass sich fast zwangsläufig eine Vielfalt von Typen bilden musste – nicht zuletzt durch den ewigen Bedarf nach mehr Rechenleistung. Dies führte gleichzeitig dazu, eine Klassifizierung von Prozessoren zu suchen, um die Leistungsfähigkeit zumindest relativ zueinander.

In diesem Kapitel der Vorlesung soll nun eine summarische, jedoch keineswegs vollständige Darstellung hiervon gegeben werden. In der Praxis hat sich herauskristallisiert, dass die Klassifizierung zwar in wenigen Bereichen angenommen wurde, vielmehr jedoch Benchmarkergebnisse für die Einordnung der jeweiligen Architektur relevant sind.

[6] *Giloi, W.*: Rechnerarchitektur. Springer-Verlag, Berlin, Heidelberg, New York, 1981. ISBN 3-540-10352-X.

[10] *Christian Märtin (Hrsg.)*: Rechnerarchitekturen – CPUs, Systeme, Software-Schnittstellen. – 2. Auflage – München, Wien: Carl Hanser Verlag, 2000.

[25] *Schmitt, F.-J.; von Wendorff, W.C.; Westerholz, K.*: Embedded-Control-Architekturen. Carl Hanser Verlag München Wien, 1999.

3.1 Bestimmende Begriffe der Rechnerarchitektur

Es sind wenige Prinzipien, die die Architektur eines Rechners bestimmen. Hierzu zählen die physikalische Struktur (und mit ihr die tatsächlichen elektrischen Verbindungen), das Operationsprinzip und diverse Strukturen zur Kontrolle, die im folgenden definiert werden [6].

Im ersten Teil werden hardwarebezogene Definitionen gegeben:

Definition 3.1:

Hauptsächliche *Hardware-Betriebsmittel* einer Rechnerarchitektur sind Prozessoren, Speicher, Verbindungseinrichtungen (Busse, Kanäle, Verbindungsnetzwerke) und Peripheriegeräte.

Definition 3.2:

Die *physikalische Struktur* eines Rechners umfasst den tatsächlichen Aufbau aus den Hardware-Betriebsmitteln mit dazugehörenden realen Signalverbindungen.

Definition 3.3:

Die *Hardware-Struktur* ist gegeben durch Art und Anzahl der Hardware-Betriebsmittel sowie den Regeln für die Kommunikation und Kooperation zwischen ihnen.

Definition 3.4:

Die *Kommunikationsregeln* werden gegeben durch die Protokolle, die den Informationsaustausch zwischen den Hardware-Betriebsmitteln regeln. Die *Kooperationsregeln* legen fest, wie die Hardware-Betriebsmittel zur Erfüllung einer gemeinsamen Aufgabe zusammenwirken.

Im zweiten Teil der Definitionen wird versucht, den Begriff der Rechnerarchitektur abzugrenzen. Hierzu sind Datentypen und Algorithmen notwendig:

Definition 3.5:

Die *Informationsstruktur* eines Rechners wird durch die (semantischen) Typen der Informationskomponenten in der Maschine bestimmt, der Repräsentation dieser Informationskomponenten und die Menge der auf sie anwendbaren Operationen. Die Informationsstruktur lässt sich als eine Menge von abstrakten Datentypen spezifizieren.

Diese Definition bedeutet u.a., dass (semantisch) verschiedene Datentypen eine durchaus identisch aussehende Repräsentation im Rechner besitzen können. Beispiel hierfür sind Maschinenbefehle, Integer- und Floating-Point-Werte in einer 32 Bit Repräsentation. All diese Informationskomponenten können zur Informationsstruktur des Rechners gehören, falls dieser darauf anwendbare Operationen besitzt.

Definition 3.6:

Die *Kontrollstruktur* eines Rechners wird durch Spezifikation der Algorithmen für die Interpretation und Transformation der Informationskomponenten der Maschine bestimmt.

Definition 3.7:

Das *Operationsprinzip* definiert das funktionelle Verhalten der Architektur durch Festlegung einer Informationsstruktur und einer Kontrollstruktur.

Definition 3.8:

Die *Rechnerarchitektur* ist bestimmt durch ein Operationsprinzip für die Hardware und die Struktur ihres Aufbaus aus den einzelnen Hardware-Betriebsmitteln.

Der dritte Teil umfasst die verbleibenden Begriffe für das Interface des Rechners zum Anwender hin:

Definition 3.9:

Die *Benutzerschnittstelle einer Rechnerarchitektur* besteht aus der Sprache, in der der Benutzer dem Betriebssystem Anweisungen erteilen kann (Job Control Language, JCL), den auf dem System verfügbaren Programmiersprachen und den Direktiven zur Benutzung der Anlage.

Definition 3.10:

Das *Hardware-Software-Interface* einer Central Processing Unit besteht aus den auf der CPU verfügbaren Assembler- bzw. Binärbefehlen, aus dem Registermodell, d.h. der modellhaften Vorstellung der verfügbaren (direkt beeinflussbaren) Register, sowie dem Speichermodell.

In der folgenden Darstellung der Klassifizierungen von *Prozessoren* und *Rechnern* wird besonders auf den Begriff der Rechnerarchitektur (und davon abgeleitet Prozessorarchitektur als Teil dieser) Bezug genommen. Es ist hierbei also das Operationsprinzip, weniger das Hardware-Software-Interface, das einen Maßstab benötigt.

3.2 Klassifizierungen von Rechnerarchitekturen

3.2.1 Die Klassifizierung nach Flynn

Die von Flynn angegebene Klassifizierung für Rechnerarchitekturen orientiert sich an der Effektivität verschiedener Organisationsformen. Rechner werden hierin als Operatoren auf zwei verschiedenen Informationsströmen, dem *Befehlstrom* und dem *Datenstrom*, angesehen. Dementsprechend ergibt sich eine zweidimensionale Klassifizierung nach den Kriterien

- Ein Rechner bearbeitet zu einem Zeitpunkt einen oder mehrere Instruktionen.
- Ein Rechner bearbeitet zu einem Zeitpunkt einen oder mehrere Datenwerte.

Die entspricht der Einteilung in die 4 Klassen *SISD*, *SIMD*, *MISD* und *MIMD*, wobei der Klasse MISD nur eine Bedeutung innerhalb von Teilbereichen der CPU, etwa im Pipeliningaufbau, zukommt. SISD bezeichnet die klassischen Von-Neumann-Rechner, SIMD die Vektorprozessoren, MIMD die Parallelrechner.

Die Flynn'sche Klassifizierung enthält zwei wesentliche Schwachpunkte:

1. Das sehr hohe Abstraktionsniveau der einzelnen Klassen führt dazu, daß sehr unterschiedliche Rechnerarchitekturen letztendlich in der gleichen Klasse geführt werden, obwohl sie unterschieden werden müssten.
2. Die Klasse MISD ist in der Systematik nur aus Vollständigkeitsgründen enthalten; gegenwärtige Rechnertypen in dieser Klasse existieren nicht.

Insbesondere der erste Schwachpunkt führt dazu, dass dieses Klassifizierungsschema in der Praxis zwar gerne für grobe Einteilungen (schlagwortartig) genutzt wird, real aber keine Relevanz besitzt und daher im Rahmen der weiterführenden Vorlesung nicht weiter angewendet wird.

3.2.2 Das Erlanger Klassifizierungssystem ECS

Das Erlanger Klassifizierungssystem ECS (Erlanger Classification System, [10]) existiert seit 1974 (mit Erweiterungen). Es unterscheidet drei verschiedene Ebenen des Parallelismus und notiert diese in einer einfachen Tripel-Notation. Die Ebenen selbst werden nochmals getrennt in Nebenläufigkeit (d.h. echter Parallelismus) und Pipelining. Für die Klassifizierung von zusammengesetzten, ggf. dynamisch rekonfigurierbaren Strukturen werden Operatoren für die Tripelstellen definiert.

Das ECS bietet somit zwei Aussagen zur Klassifizierung: Zum einen wird die *maximale Leistungsfähigkeit* (im Sinne der Parallelität) beschrieben (in relativem Maßstab), zum anderen sind auch Aussagen zur *Flexibilität* (Anzahl der Arbeitsmodi usw.) möglich. Da das Klassifikationsschema stark auf den Begriffen Serialität, Parallelität, Nebenläufigkeit und Pipelining aufsetzt, seien diese Begriffe (im Rahmen dieser Vorlesung) definiert. Dies ist nur möglich bezogen auf ein bestimmtes Abstraktionsniveau, in dem für die Rechnerarchitektur *passive Strukturen* wie Register, Daten und *Aktionen* bzw. *Operatoren* auf diesen Strukturen wie z.B. logische Verknüpfungen, ALUs usw. betrachtet werden.

Definition 3.11:

Serialität, bezogen auf ein bestimmtes Abstraktionsniveau, liegt dann vor, wenn auf diesem Niveau definierte Aktionen nicht gleichzeitig ausgeführt werden können. Dies bedeutet, dass zu einem Zeitpunkt **höchstens eine Aktion** ausgeführt werden kann.

Definition 3.12:

Parallelität, bezogen auf ein bestimmtes Abstraktionsniveau, liegt dann vor, wenn zu einem gewissen Zeitpunkt mehr als eine auf diesem Niveau definierte Aktion ausgeführt werden kann. Dieser Begriff wird weiter in Nebenläufigkeit und Pipelining unterteilt:

Definition 3.13:

Nebenläufigkeit, bezogen auf ein bestimmtes Abstraktionsniveau, liegt dann vor, wenn die Ressourcen des zu beschreibenden Rechners das gleichzeitige Ausführen **vollständiger** auf diesem Niveau definierter Aktionen erlauben.

Definition 3.14:

Pipelining, bezogen auf ein bestimmtes Abstraktionsniveau, liegt dann vor, wenn die auf diesem Niveau definierten Aktionen in *k* **Teilaktionen** unterteilt worden sind, die in einer – meist linearen – Anordnung spezialisierter, taktsynchron arbeitender Teilwerke T_1 .. T_k ausgeführt werden. Jede Aktion muss zu ihrer vollständigen Ausführung alle *k* Teilpfade (also die Pipeline) durchlaufen, wobei nach jedem Takt zum nächsten Teilwerk übergegangen wird. Die Gesamtausführungszeit einer Aktion beträgt also *k* Taktintervalle, jedoch können sich in einer Pipeline zu einem Zeitpunkt bis zu k Aktionen zeitlich überlappend in Bearbeitung befinden. In einer

Pipeline wird infolgedessen Parallelverarbeitung durchgeführt, wenn sich zu einem Zeitpunkt mehr als genau eine Aktion in Ausführung befindet.

Die Beschreibung der Grobstruktur eines Rechners durch ECS wird in drei logischen Ebenen vollzogen, die sich als reine Hardware-Elemente oder als Hardware-Elemente mit steuernden Programmelementen auffassen lassen:

- Das **Leitwerk** interpretiert ein gegebenes Maschinenprogramm Instruktion für Instruktion und steuert so die gesamten Abläufe im Rechner, indem das Programm interpretiert wird. Die Anzahl der Leitwerke wird mit *k* notiert.
- Das **Rechenwerk** führt gemäß den Steueranweisungen Sequenzen von Mikroinstruktionen aus, interpretiert also den einzelnen Maschinenbefehl bis zur vollständigen Ausführung. Die Anzahl der Rechenwerke wird mit *d* notiert.
- Die **elementare Stelle des Rechenwerks** führt gemäß der Steuerung eine Operation auf genau einer Bitstelle des Datenworts aus. Die Anzahl der elementaren Stelle wird mit *w* definiert.

Die Zahlen *k*, *d* und *w* definieren demnach den Umfang der Nebenläufigkeit des Rechners auf der jeweiligen Betrachtungsebene. Bei Pipeline-Organisationen können zusätzlich zu der Nebenläufigkeit *k'* spezialisierte Leitwerkteile ein größeres Programm bearbeiten, entsprechend *d'* spezialisierte Teile des Rechenwerks und *w'* elementare Teilwerke auf den jeweiligen Bitpositionen. Mit diesen eingeführten Größen lässt sich eine Beschreibung von Rechenanlagen angeben:

$$t_{Rechnertyp} = (k * k', d * d', w * w') \qquad (3.1)$$

Dieses Tripel charakterisiert also die ideale Eigenschaft eines Rechners, in der jede Form von Nebenläufigkeit und Pipelining vollständig in die Performance des Rechners eingeht. Einige Wertekombinationen lassen sich leicht identifizieren:

- *Serienrechner*: $k = 1$, $k' = 1$, $d = 1$, $d' = 1$, $w = 1$, $w' = 1$.
- *Bitparalleler Rechner, Parallelwortrechner*: $w > 1$
- *Multiprozessor*: $k > 1$
- *Array Prozessor*: $d > 1$
- *Macropipelining (Aufteilung eines Befehls auf mehrere Prozessoren)*: $k' > 1$
- *Befehlspipelining (>S<puter)*: $d' > 1$
- *Phasenpipelining (RISC)*: $w' > 1$

Im ECS werden noch weitere Operatoren eingeführt, die das Verhalten eines Rechners bei komplexen Gesamtstrukturen beschreiben. Der *Operator* '*' wird dabei sowohl für das Zusammensetzen innerhalb einer einzelnen, homogenen Rechnerstruktur (sprich CPU) wie auch für den Fall zusammengesetzter Strukturen, die wiederum wie ein Makropipelining funktionieren.

Für andere Fälle existieren die *Operatoren* '+' und 'v'. '+' beschreibt im Sinn eines ODER den Fall der Struktur, wo Teile alternativ oder parallel zueinander arbeiten können. 'v' wird für die Flexibilität einer Rechneranlage genutzt: Rekonfi-

gurierbare Hardware, Fehlertoleranz usw., die sich nicht in erhöhter Rechengeschwindigkeit, jedoch Sicherheit bzw. Flexibilität niederschlägt, wird mit diesem Operator dargestellt.

Das Erlanger Klassifikationssystem ergibt ein gewisses Maß der Leistungsfähigkeit von Rechnern, indem die Tripelwerte miteinander multipliziert werden. Dieses Maß ist weder absolut noch besonders anerkannt, kann aber zum Vergleich von Architekturen mit identischem Befehlssatz herangezogen werden. Die Einflüsse der Makrobefehle auf die Berechnungsgeschwindigkeit von Algorithmen wird im ECS nicht berücksichtigt.

3.3 Benchmarks

[25] *Schmitt, F.-J.; von Wendorff, W.C.; Westerholz, K.*: Embedded-Control-Architekturen. Carl Hanser Verlag München Wien, 1999.

Den im vorangegangenen Abschnitt dargestellten Klassifizierungen ist eines gemeinsam: Sie sind nicht zur effektiven Messung bzw. Darstellung der Leistungsfähigkeit geeignet. Für den Entwickler eines Mikroprozessor- oder –controllerbasierten Anwendung hingegen ist es unabdingbar, eine Möglichkeit zum echten Leistungsvergleich (was auch immer unter 'Leistung' verstanden wird) durchführen zu können. Hierzu werden im folgenden einige Anmerkungen gemacht.

3.3.1 Maßzahlen zur Bewertung von Mikrocontrollern

Um die Anforderungen eines *Anwendung* bewerten zu können, werden definierte Maßzahlen benötigt. Hierbei sollte unbedingt beachtet werden, dass diese Maßzahlen jeweils nur für eine spezifische Anwendung (Benchmark) bzw. für eine spezifische Hardwareumgebung (Hardwareplattform) gelten. Bei den nun folgenden Maßzahlen ist die Angabe eines Index $_x$ kennzeichnend für eine Anwendungsspezifische Maßzahl.

Performance

Die wohl wichtigste Maßzahl ist die *Performance* (Leistungsfähigkeit) eines Prozessors:

Definition 3.15:

Die Performance eines Prozessors ist als der Kehrwert der Zeitspanne definiert, welche dieser für eine anwendungsspezifische Aufgabe benötigt:

$$Performance_x = \frac{1}{Ausführungszeit_x}$$

Die Performance ist eine Maßzahl zur Bewertung des Gesamtsystems eines Mikroprozessors/–controllers. Für einige weitverbreitete Programme, häufig

synthetischen Ursprungs, wird sie in Einheiten-lose Maßzahlen umgewandelt. Zu diesen Programmen zählen Dhrystone, Whetstone, SPECint (8 C-Programme), SPECfp (10 Fortran-Programme), BAPCo (www.bapco.com).

Anwendungsspezifischer Leistungsverbrauch

Für Rechner mit Netzenergieversorgung ist dieses Maß zwar irrelevant, für Mikrocontrolleranwendungen auf Batteriebasis jedoch sehr wichtig:

Definition 3.16:

$$Anwendungsspezifischer\ Leistungsbedarf_x = \frac{\overline{elektrischer\ Leistungsbedarf}_x}{Performance_x}$$

Instruction Count

Der *Instruction Count* zählt die Anzahl der Instruktionen (Maschinenbefehle), um eine anwendungsspezifische Aufgabe zu codieren. Der Instruction Count stellt damit ein Maß für die Güte des Befehlssatzes dar.

Clock-Cycles-per-Instruction (CPI)

Bei Prozessoren ähnlichen Aufbaus – dies ist für RISC- und superskalare Prozessoren, die in den weiteren Kapiteln behandelt werden, der Fall – liefert die Angabe der CPI einen guten Anhalt zur Leistungsfähigkeit der jeweiligen Architektur:

Definition 3.17:

$$CPI_x = \frac{CPU - Taktrate \bullet Ausführungszeit_x}{Instruction - Count_x}$$

Demgegenüber sollten native *MIPS* (Mega Instructions Per Second), die als Quotienten der Anzahl der Instruktionen, dividiert durch die Ausführungszeit, sowie *MFLOPS* (Mega Floating-Point Operations Per Second) nicht verwendet werden, weil ihre Aussagekraft sehr gering ist. Hiermit wird keinerlei Bezug auf die Komplexität des Programms genommen, oftmals sind die Herstellerangaben auch nur Spitzenwerte, die theoretisch erreichbar sind.

Clock-Rate

Fast alle Mikrocontroller oder –prozessoren (Ausnahme: asynchrone, d.h. selbst-synchronisierende Prozessoren) verwenden einen Takt mit einer konstanten Rate, welche festlegt, wann die Hardware bestimmte Funktionen ausführt. Diese diskreten Zeitintervalle werden Zyklen (Cycles, Ticks, Perioden), der Kehrwert auch *Clock-Rate*. Diese Clock-Rate (Taktrate) ist nach oben durch Verlustleistung, Signallaufzeiten etc., nach unten ggf. durch dynamische Effekte begrenzt, wobei bei statischen Designs die minimale Taktrate 0 Hz beträgt.

Codedichte (Code-Density)

Definition 3.18:

$$Codedichte_x = \frac{1}{Größe\,des\,benötigten\,Instruktionsspeichers_x}$$

Die Codedichte spielt insbesondere in Systemen mit harten Speicherplatzbeschränkungen eine wichtige Rolle. Sie beurteilt gemeinsam den Instruktionssatz und die Güte des Compilers.

Kontextwechselzeit

In einem Multitasking- oder Multithreadingsystem (*Thread*: Faden, wird als in sich geschlossener Programmfaden innerhalb einer Task genutzt) muss der Prozessor zwischen verschiedenen Tasks bzw. Threads wechseln können und daher den Kontext wechseln. Hierbei sind verschiedene Register zu sichern, wie z.B.:

- Program Counter/Instruction Pointer (PC, IP)
- Processor Status Register
- Datenregister, Akkumulator
- Adressregister, Page Register
- Stack-Pointer-Register
- ggf. General-Purpose-Register

Definition 3.19:

$$Kontext-Wechsel-Zeit_x = Zeit\,zum\,Sichern\,des\,Kontext + Zeit\,zum\,Wiederherstellen\,des\,Kontext$$

Interrupt-Antwortzeit

Die *Interrupt-Antwortzeit* (*Interrupt Latency Time*) ist als Zeitspanne zwischen dem Auftreten eines Interrupts (asynchrone Programmunterbrechung durch externe Signalisierung) und der Ausführung des ersten Befehls definiert. Diese Zeit ist stark abhängig von dem momentanen Zustand aller äußeren Einflüsse (Anzahl der momentan aktiven Quellen) sowie von dem Kontext des Programms (nicht-unterbrechbare Codeabschnitte).

Interrupt-Overhead-Zeit

Die *Interrupt-Overhead-Zeit* ist als die maximale Zeit definiert, welche eine Interruptanforderung den Prozessorcore für andere Aufgaben blockiert, bis die erste Anweisung der entsprechenden Interrupt-Service-Routine (ISR) ausgeführt wird. Grund dieser Blockade können Überprüfung der Priorisierung, Verzweigung in die ISR, Kontextsicherung sowie Löschung der Interruptanforderung sein.

4 Einführung in die RISC-Architektur

Die RISC-Architektur, als *Reduced Instruction Set Computer Architektur* eingeführt, hat starke historische Wurzeln. Die Entwicklung der ersten Prozessortypen war eher auf die Implementierung immer komplexerer Befehlssätze ausgerichtet (CISC-Architektur)wofür es mehrere Gründe gab:

- Der Unterschied in der Zugriffsgeschwindigkeit auf Hauptspeicher und innerhalb der CPU führte dazu, dass Befehle, die Programmsequenzen in der CPU implementierten, wesentlich schneller liefern als externe Sequenzen. Der Unterschied in der Geschwindigkeit zwischen Mikroprogrammspeicher und Hauptspeicher betrug etwa Faktor 10, sodass die 10fache Anzahl an Befehle für ein Mikroprogramm benutzt werden durfte.

  Der Geschwindigkeitsunterschied existiert seit langem nicht mehr. Wie in Kapitel 8 gezeigt werden wird, weisen neueste Forschungsergebnisse jedoch einen durchaus ähnlichen Weg, wenn auch aus anderen Gründen.
- Durch die eingeführte Mikroprogrammierung war der Aufwand zu Erweiterung des Befehlssatzes relativ gering. Dies führte z.B. zu nachladbaren Befehlssätzen bei Großrechenanlagen.
- Die Einführung komplexer Befehle führte insbesondere bei Assemblerprogrammierung zu kompakten Programmen, vorteilhaft, da sowohl die Speicherkosten als auch die Ausführungszeit gesenkt werden konnten.
- Für die Hochsprachenprogrammierung und -übersetzung wurde zumindest erhofft, durch komplexe Befehle eine Brücke zwischen Maschinen- und Hochsprache zu schaffen.
- Der Zwang der Aufwärtskompatibilität (aus Marktgründen) brachte die Notwendigkeit mit sich, immer komplexere Befehlssätze zu implementieren.
- Ein komplexer Maschinenbefehlssatz galt als Beweis der Leistungsfähigkeit eines Rechners.

Die (virtuellen und realen) Vorteile gerieten in den 70er Jahren stark ins Schwanken. Folgende Effekte führten zu Verkehrung der Aussagen:

- Die Fortschritte in der Speichertechnologie führten zu einer drastischen Verringerung des Geschwindigkeitsunterschieds zwischen Haupt- und CPU-Speicher und vor allem zu wesentlich billigerem Speicher.
- Die sehr schlechte Ausnutzung des Befehlssatzes durch Compiler (im Gegensatz zur Annahme) bedeutete eine sehr schlechte Ausnutzung der CPU-Möglichkeiten: Nach Untersuchungen von IBM wurden in 80% des Codespeichers nur 5% des Befehlssatzes, in 95% 10% des Instruktionen und in 99% gerade 15% aller Befehle genutzt – ausgedrückt in der sogenannten 10 : 90 Regel.

- Die Ersetzung eines komplexen Maschinenbefehls durch mehrere einfache erwies sich (gelegentlich) als zeitsparend.
- Die Mikroprogramme wurden immer umfangreicher, sodass sie letztendlich nicht mehr in der CPU fest implementiert waren, sondern nachladbar gestaltet wurden. Die Folge waren Nachladestrategien, Schutzmechanismen und eine weitere Verkomplizierung der CPU, verlängerte Entwurfszeiten usw.
- Die Komplexität von Mikroprogrammen, insbesondere das erforderliche hohe Maß an Parallelität, bedingte eine erhöhte Fehleranfälligkeit dieser Mikroprogramme; Entwicklungssprachen für diese Form der Programmierung sind nicht vorhanden, sodass die Komplexität letztendlich durch die Entwickler gemeistert werden musste.

Aus all diesen Gesichtspunkten ergaben sich neue Entwicklungsansätze für Mikroprozessorgenerationen. Die heutige Definition eines RISC-Prozessors, dessen Struktur erst in den folgenden Abschnitten dargestellt wird, kann durch folgende Eigenschaften gut beschrieben werden:

1. Die Anzahl der Maschinenbefehle ist kleiner/ungefähr gleich 150
2. Die Anzahl der Adressierungsmodi ist ≤ 4
3. Anzahl der Befehlsformate ≤ 4
4. Anzahl der allgemeinen CPU-Register ≥ 32
5. Die Ausführung aller oder der meisten Maschinenbefehle erfolgt innerhalb eines (internen) Takts.
6. Der Speicherzugriff erfolgt nur über Load/Store-Befehle
7. Die Steuerung innerhalb der CPU ist festverdrahtet
8. Die CPU unterstützt in ihrem Befehlssatz höhere Programmiersprachen (beispielsweise durch CASE-Konstrukte usw.)

Werden von diesen acht Bedingungen mindestens 5 erfüllt, so kann die Architektur als RISC bezeichnet werden. Insbesondere die Eigenschaft der Befehlsausführung innerhalb eines Takts (zumindest bei Ausnutzung der Instruktionspipeline) erweist sich dabei als Programm-beschleunigend, denn dies führt tatsächlich zu schnelleren Programmen. Andererseits ist die pure Beschränkung auf wenige Befehle bei extremer Ausführung nicht sinnvoll, wie man an der 1-Befehl-Maschine erkennt: Diese Maschine kann nur den Befehl

SUBBEQ X, Y, Z

(subtract and branch if equal) ausführen, indem sie X = X-Y ausführt und an Z springt, wenn das Ergebnis 0 ist ($\rightarrow$ 2.6).

Es stellt sich damit die dringende Frage nach dem Optimum zwischen Einfachheit und endgültiger Programmausführungsgeschwindigkeit. Diese Frage ist in den Projekten Anfang der 80er Jahre durchaus anders beantwortet worden als dies heute der Fall ist (oder wäre). Der Streit um die Philosophie, CISC oder RISC, tritt Mitte

der 90er Jahre klar in den Hintergrund; allein die Ausführungszeit von Befehlen ist noch relevant.

Innerhalb dieser Vorlesung wird in diesem Kapitel der (historische) Weg von den Analysen der Befehlssätze und der Häufigkeit des Einsatzes einzelner Befehle über die Daten- und Kontrollflussanalyse in der CPU bis zur Performancebestimmung einer RISC-CPU nachvollzogen.

[1] *Hennessy, J. L., Patterson, D. A.*: Computer Architecture: A Quantitative Approach. – Second Edition – San Francisco: Morgan Kaufmann Publishers, 1996

[2] *Hennessy, J. L, Patterson, D. A.*: Computer Organization & Design: The Hardware/Software Interface. – Second Edition – San Francisco: Morgan Kaufmann Publishers, 1997

[3] *Siemers, C.*: Prozessorbau. – Carl Hanser Verlag München Wien, 1999

[7] *Beierlein, Th., Hagenbruch, O. (Hrsg.)*: Taschenbuch Mikroprozessortechnik. 2.Auflage – Fachbuchverlag Leipzig im Carl Hanser Verlag München Wien, 2001

[10] *Märtin, C. (Hrsg.)*: Rechnerarchitekturen – CPUs, Systeme, Software-Schnittstellen. – 2. Auflage – München, Wien: Carl Hanser Verlag, 2000.

[11] *Šilc, J.; Robic, B.; Ungerer, T.*: Processor Architecture. – Berlin, Heidelberg, New York: Spriner, 1999.

4.1 Analyse der Befehlssätze

Um die Reduzierung eines Befehlssatzes möglichst effektiv vornehmen zu können, wurde in dem ersten Universitätsprojekt RISC I/II *Entwurfsziele* (Patterson, University of California at Berkeley, 1980) formuliert. Im Anschluss daran konnten diverse Programme, in den Hochsprachen C und Pascal geschrieben, analysiert werden, wobei hierbei natürlich die Schwierigkeit bestand, dass diese Programme durch Compiler übersetzt wurden, die für die herkömmlichen Architekturen optimiert waren.

Die Entwurfsziele waren wie folgt definiert:

1. *Ein-Zyklus-Befehle*: Pro Takt sollte ein Maschinenbefehl ausgeführt werden. Der Komplexitätsgrad und die Ausführungsgeschwindigkeit eines RISC-Maschinenbefehls sollte in etwa zu denen bei der VAX-II/780 (Mikroprogramm!) äquivalent sein.
2. *Verzicht auf Mikroprogrammierung*: Die Mikroprogrammierung sollte durch ein festverdrahtetes Steuerwerk ersetzt werden. Hierdurch sollte neben der Schnelligkeit auch Chipfläche eingespart werden.
3. *Einheitliches Befehlsformat*: Alle Befehle sollten das gleiche Format, insbesondere die gleiche Länge besitzen. Hierdurch sollte eine einfachere Decodierung und eine einheitlichere Behandlung der Befehle allerdings auf Kosten der Programmlänge geschaffen werden.

4. *Load/Store-Architektur*: Das Interface zwischen Speicher und CPU sollte ausschließlich auf Load- und Store-Befehlen basieren, während alle anderen Befehle wie Arithmetik nur auf den Registern arbeiten durften. Dies implizierte eine größere Anzahl an Registern.
5. *Unterstützung höherer Programmiersprachen*: Um die Vorteile einer Assemblerprogrammierung, schnelleren Code zu erzeugen, zu relativieren, sollten Maschinenbefehle in der CPU so angeboten werden, dass Compiler sehr gut optimierten Code erzeugen könnten. Dies betrifft insbesondere Verzweigungsbefehle und ist weiterhin ein aktuelles Thema (siehe auch nachfolgende Kapitel).
6. *Keine Unterstützung für Gleitkommaarithmetik und Betriebssystem*: Die Gleitkommaarithmetik, in experimentellen Projekten meist eher hinderlich, da prinzipiell ähnlich gestaltbar wie Festkommaarithmetik, wurde bewusst fortgelassen. Interessanter ist hier der Verzicht auf Unterstützung von Betriebssystembefehlen wie etwa Supervisorcalls usw., der prinzipieller Natur war.
7. 3*2-Bit-Architektur*: Die Projekte waren auf die (1980 avandgardistischen) Datenbreite von 32 Bit ausgelegt; 8-, 16- und 32-Bit-Daten sollten bearbeitbar sein.

Zur Festlegung des Befehlssatzes der RISC I-Architektur mussten typische Programme auf zwei Charakteristika hin untersucht werden: Operanden und typische Befehle (in C oder PASCAL) sowie deren Übersetzung in Maschinensprache. Das RISC-I-Projekt zeigt demnach in typischer Weise die Wechselwirkung von Rechnerarchitektur und Compiler.

Die 8 Programme, die zur Analyse herangezogen wurden, bestanden aus:

- P1: Pascal-Compiler, geschrieben in Pascal
- P2: Unterprogramm in einem Entwurfssystem
- P3: Pascal Prettyprinter
- P4: Vergleichsprogramm zum Vergleich zweier Dateien
- C1: C-Compiler, geschrieben in C
- C2: Programm zum Zeichnen von VLSI-Maskenlayouts
- C3: Programm zum Formatieren von Text
- C4: Sortierprogramm unter Unix

Es muss aus heutiger Sicht angemerkt werden, dass die typischen Programme der 90er Jahre, Textverarbeitung und Graphikprogramme, sicher unterrepräsentiert sind; andererseits dürfte diese Mischung dem Standard von 1980 entsprochen haben. Bild 4.1 zeigt die dynamische Häufigkeit der Operandentypen für diese 8 Programme.

Operandentyp	Dynamische Häufigkeit (in Prozent) von Operandentypen in Programmen								Durchschnitt
	P1	P2	P3	P4	C1	C2	C3	C4	
Ganzzahlige Konstante	14	18	11	20	25	11	29	28	20
Skalare	63	68	46	54	37	45	66	62	55
Felder/ Strukturen	23	14	43	25	36	43	5	10	25

Bild 4.1: Dynamische Häufigkeit von Operandentypen für die Testprogramme zur RISC-I-Architektur

Die Unterstützung für die Operandentypen durch die Maschinenbefehle einer CPU kann in mehrfacher Form erfolgen: Floating Point Befehle (hier nicht weiter aufgeschlüsselt) sowie Adressierungsformen bieten hier eine Reihe von Möglichkeiten.

Die Hochsprachenkonstrukte in diesen Programmen lassen sich in verschiedene Kategorien einteilen: Zuweisungsbefehle (assign), Kontrollflussstrukturen (if, case, for, with, loop, while) und Prozeduraufrufe (call/return). Die in den Programmen auftretenden Häufigkeiten sind in Bild 4.2 in Tabellenform dargestellt.

Befehlstyp	Dynamische Häufigkeit (in Prozent) von Befehlstypen in Programmen								Durchschnitt	
	P1	P2	P3	P4	C1	C2	C3	C4	Pascal	C
assign	39	52	35	53	22	50	25	56	45	38
if	35	30	36	16	59	31	61	22	29	43
call	15	14	16	15	6	17	9	16	15	12
with	2	0	5	13	2	2	3	5	5	3
loop	5	5	5	4	9	0	1	1	5	3
case	4	0	1	0	2	0	0	0	1	< 1

Bild 4.2: Dynamische Häufigkeit von Hochsprachenanweisungen

Die dynamische Häufigkeit (hierin ist die Häufigkeit des Durchlaufs einbezogen) von Befehlen wird dann relevant, wenn die tatsächliche Ausführungszeit bestimmt wird. Dies erfolgte durch eine Übersetzung in die entsprechenden Maschinenbe-

fehle unter Berücksichtigung der notwendigen Speicherzugriffe (Code und Daten!). Folgendes Beispiel (Bild 4.3) möge dies kurz erläutern:

```
int a, b;                mov r0, a ;
                         mov r2, 0 ;
if( a > 0 )              cmp r0, 0 ; Vergleich
      b = a;             ble L1    ; Bedingter Sprung
else                     mov r2, r0; Nur für a > 0!
      b = 0;       L1: mov b, r2 ;


       (a)                          (b)
```

Bild 4.3: Beispielcode in C (bedingte Zuweisung) mit Übersetzung in Assemblercode

An diesem Beispiel wird deutlich, dass für den Vergleich 3 Befehle, 1 Variable und zwei Konstanten geladen werden müssen (6 Ladeoperationen); die eigentliche if-Bedingung einschließlich des else-Zweiges benötigt dagegen nur eine Verzweigung (eine Ladeoperation), die beiden Datenzuweisungen laufen mit ein oder zwei Befehlen und einem Store-Befehl (3–4 Ladeoperationen) ab. Es sollte vermerkt werden, dass diese Assemblerübersetzung bereits optimiert ist und nur als prinzipielles Beispiel dienen soll.

Die sich aus diesen Untersuchungen ergebenden, gewichteten Häufigkeiten (durch Multiplikation) für die einzelnen Befehle ergeben dann folgendes Bild:

Befehlstyp	Dynamische Häufigkeit (in Prozent)		Gewichtung mit Anzahl der Maschinenbefehle		Gewichtung mit Anzahl der Speicherzugriffe	
	Pascal	C	Pascal	C	Pascal	C
call/return	15	12	31	33	44	45
loop	5	3	42	32	33	26
assign	45	38	13	13	14	15
if	29	43	11	21	7	13
with	5	-	1	-	1	-
case	1	< 1	1	1	1	1
goto	-	3	-	-	-	-

Bild 4.4: Dynamische Häufigkeit von Befehlstypen ohne (a), mit Gewichtung der Anzahl der Maschinenbefehle (b) und der Anzahl der Speicherzugriffe (c).

Der weitere Verlauf der Konzeptionierung von RISC-Prozessoren lässt sich am Besten anhand einer Modell-CPU (MPM3) darstellen.

4.2 Konsequenzen für eine RISC-CPU

Nach den Analysen der Befehlssätze bzw. der zum Betrieb notwendigen Befehle kann man dazu übergehen, die CPU gemäß den RISC-Prinzipien zu entwickeln. Hierbei kann man von folgenden, gegenseitig abhängigen Schritten ausgehen:

- Wahl der Anzahl der Pipelinestufen
- Wahl der intrinsischen Datenbreiten
- Wahl des Registermodells
- Wahl des Befehlssatzes
- Korrektur des Pipeliningmodells zur möglichst schnellen (ungestörten) Ausführung der Programme

Die Abhängigkeiten der Schritte sind häufig impliziter als auf den ersten Blick erscheinend. Beispielsweise kann die intrinsische Datenbreite, eigentlich ein Parameter, der die elementare Datenoperation (16 bit, 32 bit usw.) bestimmt, den Befehlssatz und das Registermodell beeinflussen, weil die Befehle in diesem Datenformat auch codiert werden müssen.

Bei der Wahl des Phasenpipelinings und der Anzahl der Stufen müssen gemäß [3, Kapitel 6] folgende Vorgaben beachtet werden:

- Die einzelnen Phasen, die innerhalb der Architektur parallel zueinander ausgeführt werden sollen, müssen in ihrem Zeitbedarf zueinander balanciert definiert werden. Diese Forderung ist wichtig, um nicht einzelne Phasen, etwa das Instruction-Fetch in typischen CISC-Architekturen (z.B. MPM2, [3]), künstlich verlängern zu müssen, da die Load-Phase der vorhergehenden Befehls noch nicht abgeschlossen ist.

  Diese Forderung könnte auch zur Konsequenz haben, dass eine bestimme Phase gegebenenfalls nochmals unterteilt wird, um dann in diesen Teilen ebenfalls parallel zueinander ablaufen zu können.
- Im Design des Mikroprozessors müssen alle Möglichkeiten ausgeschöpft werden, um strukturelle Hazards (→ 4.4.2) zu verhindern bzw. in ihren Auswirkungen zu mildern. Hierbei ist es insbesondere notwendig, die Anzahl der externen Operandenzugriffe zu minimieren, da die gleichzeitige Nutzung von Adress- und Datenbus für Fetch, Load bzw. Write Back bei einem einfach ausgelegten Bussystem zum Konflikt führen muss.
- Die dritte Forderung besteht in der Entdeckung und Behebung, möglichst sogar Vermeidung von Kontrollfluss- (→ 4.4.3) und Datenhazards (→ 4.4.1), d.h. Abhängigkeiten zwischen den Instruktionen. Hierzu ist zu bemerken, dass bei Auftreten zumindest Wartezyklen einzufügen sind, da ansonsten mit falschen Ergebnissen zu rechnen ist.

4.3 MPM3: Beispiel für ein Prozessormodell mit Phasenpipelining

Die aufgestellten Anforderungen an ein Prozessormodell gemäß den RISC-Prinzipien einschließlich eines Phasenpipelinings werden in eine konkrete Modellarchitektur umgesetzt, wobei im Unterschied zu den vorhergehenden Modellen lediglich die Prinzipien im Vordergrund stehen.

Das Mikroprozessormodell #3 (MPM3) wird hierfür als ein Prozessor mit im Vergleich zum MPM2 [3] gleichwertigen arithmetischen und logischen Kapazitäten, aber einem abweichenden Programmiermodell definiert. Die Abweichungen ergeben sich hierbei aus den vorangegangenen Forderungen zum möglichst optimalen Phasenpipelining.

4.3.1 Prozessor-Architekturklasse und Programmiermodell

Die vorangegangenen Darstellungen lassen kaum Zweifel daran, dass für einen RISC-Prozessor mit Phasenpipelining die Anzahl der externen Speicherzugriffe pro Instruktion möglichst minimiert werden muss, um die strukturellen Hazards weitgehend zu vermeiden. Aus diesem Grund wird das MPM3 gemäß der Prozessorklassifizierung nach Operandenzugriff (→ 2.8) einer Register-Architektur mit ausschließlichem Datenaustausch zwischen internen Registern und externem Speicher durch Load/ Store-Befehle entsprechen. Dies wird im Allgemeinen als *Load/Store-Architektur* bezeichnet.

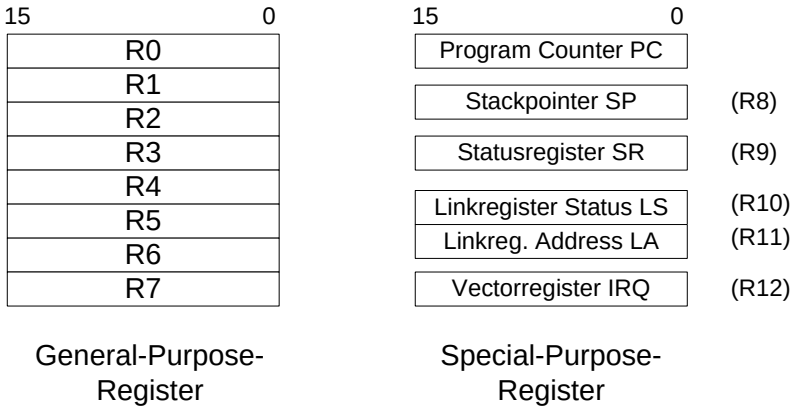


Bild 4.5: Programmiermodell MPM3

Das Programmiermodell in Bild 4.5 enthält 8 General-Purpose-Register R0 .. R7, die als Mindestausstattung für eine derartige Architektur angesehen werden können. Daneben sind weitere (Special-Purpose-)Register vorgesehen, deren Zweck in Zusammenhang mit den Befehlen des MPM3 (→ 4.3.2) zu sehen ist. Die Datenbreite ist einheitlich mit 16 Bit vorgesehen, wobei bei den speziellen Registern Abweichungen hiervon durch Auffüllung mit ’0‘ ausgeglichen werden. Die in Klammern angegebene Registerbezeichnung für einige Special-Purpose-Register betrifft die interne Codierung dieser bei Austauschbefehlen.

00h	IF	MF	0	0	NF	OF	ZF	CF

Bild 4.6: Statusregister MPM3

Das Statusregister (Bild 4.6) enthält die ”klassischen“ arithmetischen Flags Carry, Zero, Overflow und Negative, das Kontrollflussflag Interrupt sowie das Sonderflag Mode, mit dem der Zustand innerhalb einer Interrupt Service Routine bei Simulationen angezeigt werden kann.

4.3.2 Instruktionssatz MPM3

Der gravierendste Unterschied in den Instruktionssätzen der Modelle MPM2 [3] und MPM3 besteht auf den ersten Blick in der Einführung von Move-Instruktionen im MPM3, die eine Datenkopie zwischen zwei Registern beinhalten. Der Load- und Store-Befehl bleibt im MPM3 jeweils für den Transfer von Daten zwischen Register und Speicher erhalten, während die Transfer- und Exchange-Instruktionen des MPM2 komplett entfallen bzw. im Move enthalten sind.

Der zweite Unterschied liegt in den im MPM3 zugelassenen Adressierungen, die sich so weit wie möglich auf interne Register beziehen und bei externen Operationen teilweise verkürzte Operanden zulassen. Da das MPM3 keine Akkumulatorarchitektur mit dem zentralen Register darstellt, muss der Zieloperand ebenfalls angegeben werden, falls er nicht implizit mit einem Quelloperanden übereinstimmen soll. Die Adressierungsarten und die Instruktionen sind teilweise eng miteinander gekoppelt und werden daher im Folgenden gemeinsam diskutiert.

Der dritte Unterschied besteht aus einer neuen Form der Codierung einer Instruktion, die nunmehr kein einheitliches Format für alle Befehle zulässt, da sich die Codierung einschließlich der Operanden auf ein 16-Bit-Format beschränken muss. Dieser Unterschied hat die gravierendsten Auswirkungen. Für die Codierung steht nunmehr nicht mehr die besondere Decodierbarkeit, d.h., ein sorgfältig nach Befehlsinhalten getrenntes Format, im Vordergrund, sondern die Codierung der Operanden (0 bis 3) bestimmen das Befehlscodeformat. Oberstes Gebot ist es, die

Anzahl der Buszugriffe für jeden Befehl möglichst ein 1 zu beschränken, wobei die Fetch-Phase exakt einen Buszugriff benötigen wird.

Im MPM3 werden die Instruktionen in vier Befehlsgruppen eingeteilt, hier als Codierungsformate bezeichnet und im Folgenden diskutiert.

4.3.2.1 Das 3-Register-Format

Für eine Vielzahl von Befehlen ist die Angabe von drei Operanden notwendig: Ziel, Quelle1, Quelle0. Für diese Operanden sind im MPM3 lediglich die General-Purpose-Register R0 bis R7 notwendig, für die eine Codierung in 3 Bits ausreichend ist.

Bild 4.7 zeigt den Aufbau dieses Codeformats.

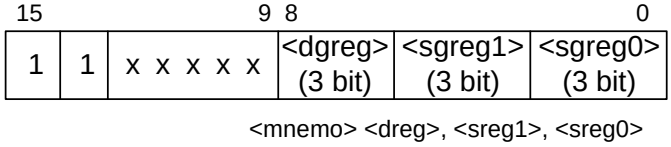


Bild 4.7: Codierungsformat für 3-Register-Befehle

In diesem Format lassen sich 32 verschiedene Instruktionen in 5 Bits codieren. Kennzeichnend für diese Codierungsgruppe ist die Codierung ’11‘ in den höchsten Bits.

4.3.2.2 Das 2-Register-Format

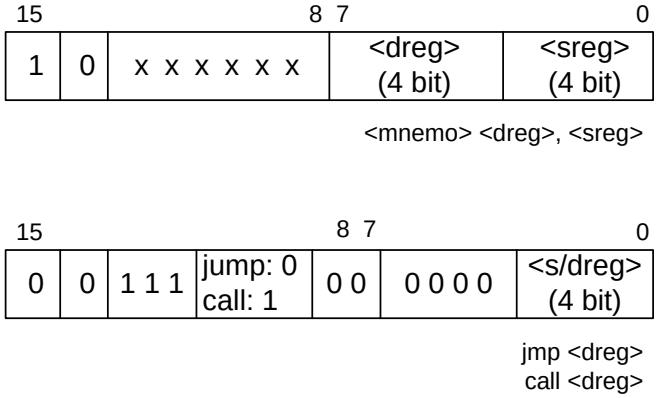


Bild 4.8: Codierungsformat für 2-Register-Befehle

Für alle Register R0 bis R12 gilt ein Format, bei dem lediglich Ziel- und Quellregister anzugeben sind und das aus diesem Grund 2-Register-Format heißt. Diese Register müssen in jeweils 4 Bit codiert werden, sodass hierfür das in Bild 4.8 dargestellte Codeformat gilt.

Innerhalb dieser Gruppe lassen sich 64 Befehle in 6 Bits codieren. Kennzeichnend ist hier ’10‘ in den höchstwertigen Bits.

4.3.2.3 Das «Register, Data»-Format

Um Programmkonstanten gleich welcher Art in ein Register zu laden, wird ein Format zum Transfer dieser Daten innerhalb der 16-Bit-Codierung der Instruktionen notwendig. Es kann natürlich nicht erwarten werden, dass Konstanten in voller 16-Bit-Breite codierbar sind, da sowohl der Befehl als auch das Zielregister einzutragen sind.

Die Lösung besteht darin, 8 Bit für die Programmkonstanten, 3 Bit für das Zielregister (R0 .. R7) und neben der Codierungsgruppe (’01‘) 3 Bit für 8 Befehle zu erhalten. Das Codierungsformat sind damit wie in Bild 4.9 dargestellt aus.

Als Sonderfälle sind – wie im vorigen Fall der 2-Register-Codierung – zwei Codierungsformate aus der Codegruppe für allgemeine Befehle vorgesehen. Dies erfolgt, um die geringe Anzahl an Befehlen in dieser Gruppe zu erhöhen.

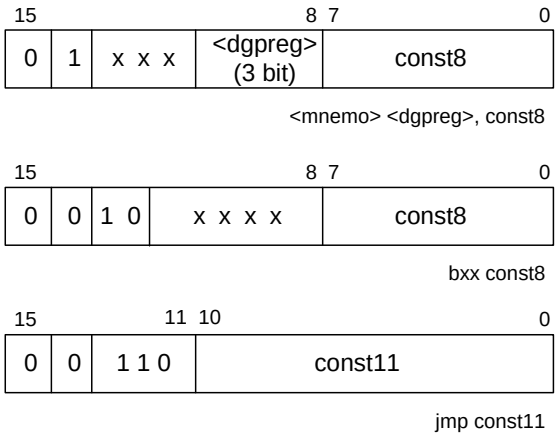


Bild 4.9: Codierungsformate für «Register, Data-Befehle»

Die zusätzlichen Befehle bestehen aus den bedingten Branch-Befehle, deren Bedingung in den 4 Codierungsbits untergebracht ist, sowie aus dem relativen (unkonditionierten) Sprungbefehl JMP, die aufgrund des Fehlens einer Bedingung oder eines Zielregisters 11 Bits für die Konstanten bereitstellen kann. Bei Berücksichtigung der Tatsache, dass jeder Befehle 16 Bits (= 2 Bytes) benötigt, können

somit die relativen Sprünge einen Adressraum von +258/-252 bzw. +4098/-4092 Adressen umfassen – für viele Applikationen ausreichend. Die Codierung der PC-relativen Information erfolgt dann im Wort-Format (16 Bit-Differenz).

4.3.2.4 Allgemeines Codierungsformat

Abgesehen von den oben bereits integrierten Ausnahmen werden alle allgemeinen Befehle, die insbesondere keine Operanden benötigen, in der Codegruppe mit ’00‘ als höchsten Bits codiert. Bit 13 dieser Codegruppe bleibt auf ’0‘, um von den Sonderformaten aus dieser Gruppe unterscheiden zu können. In den verbleibenden 13 Bits können 8192 verschiedene Instruktionen untergebracht werden.

4.3.3 Instruktionsgruppen und Adressierungen

Eine andere Betrachtungsweise für den Instruktionssatz geht davon aus, welche Befehle eigentlich mit welcher Adressierung sinnvoll bzw. notwendig sind. Hierzu werden alle Befehle in die Befehlsgruppen Transferbefehle, arithmetisch-logische Befehle, Flagbefehle und Kontrollflussbefehle eingeteilt.

4.3.3.1 Transferbefehle

Die Gruppe der Transferbefehle setzt sich aus den Instruktionen MOV, MOVH, LD, ST, PUSH und POP zusammen. Tabelle 4.1 gibt einen Überblick über die Bedeutung dieser Mnemonics und die zugelassenen Adressierungsarten.

PUSH und POP sind mit einer registerdirekten und einer impliziten Adressierung zugelassen. Die implizite Adressierung bezieht sich automatisch auf den Stack, wobei Predekrement beim Schreiben (PUSH) sowie Postinkrement beim Lesen (POP) auf den Stackpointer angewendet werden. Als Quelle bzw. Ziel sind mit Ausnahme des PC alle Register möglich, also auch die Special-Purpose-Register. Diese werden durch Angabe der Registernummer codiert.

Die Befehle LD und ST, die zur Kopplung zwischen Registern und Speicher benötigt werden, werden ausschließlich mit einer Register-indirekten Adressierung genutzt. Der Grund hierfür liegt in dem Anspruch, das Businterface möglichst gering zu belasten. Eine direkte Adressierung hätte hier neben dem Fetch der Instruktion zwei zusätzliche Buszyklen benötigt, während die Register-indirekte Adressierung nur einen Zyklus additiv zum Fetch in Anspruch nehmen muss. Als Registeroperanden sind in diesem Fall nur die General-Purpose-Register zulässig.

Die MOV- und MOVH-Befehle dienen zum Austausch der Daten zwischen Registern und der Belegung von Registern mit Konstanten. Die MOV-Instruktion ist dabei mit allen Registern als Operand zulässig. Da, wie aus der Codierung hervorgeht, noch Platz für 8 Bit innerhalb des Befehlsformats ist, können konstante Werte mit diesem Wertebereich in den Befehl selbst codiert werden und führen in der Load-Phase nicht zu einer zusätzlichen Busbelastung.

Mnemonic	Operanden	Bedeutung
MOV	<dreg>, <sreg> <dreg>, const8	Move Quelloperand in Zieloperand. Quelloperand kann Register oder 8-Bit-Konstante sein
MOVH	<dreg>, const8	Move Konstante (oberen 8 Bit) in obere 8 Bit des Zielregisters
LD	<dgpreg>, [<adrreg>]	Lade Zielregister mit dem Inhalt der Datenspeicherstelle, deren Adresse im Sourceregister steht
ST	[<adrreg>], <sgpreg>	Speicher Registerinhalt in Speicherstelle, Adresse im Zielregister
PUSH	<sreg>	Kopiere Registerinhalt auf den Stack
POP	<dreg>	Kopiere aktuelle Stackspeicher-stelle in Register

Legende:
<dreg>, <sreg>: Destination- bzw. Sourceregister R0 .. R12
<dgpreg>, <sgpreg>: Destination- bzw. Sourceregister R0 .. R7
<adrreg>: Adreßregister R0 .. R7
const8: Programmkonstante, 8 Bit

Tabelle 4.1: Transferbefehle im MPM3

Die Interpretation des konstanten Werts erfolgt mit Vorzeichen, es werden sowohl die unteren 8 Bits im Register als auch die oberen belegt, wobei die Erweiterung vorzeichenrichtig erfolgt (Bit 7 der Konstante wird auf die Bits 8 .. 15 des Registers kopiert).

Die MOVH-Instruktion hingegen überschreibt nur die oberen 8 Bits des gewählten Registers mit dem konstanten Wert. Transferbefehle beeinflussen keine Flags im Statusregister.

4.3.3.2 Arithmetisch-logische Befehle im MPM3

Die zweite Gruppe besteht aus allen Befehlen zur Arithmetik sowie zur logischen Verknüpfung. Die Anwendung dieser Befehle ist zunächst nur für General-Purpose-Register vorgesehen, wobei es von dieser Regel Ausnahmen gibt.

Die arithmetisch-logischen Befehle werden abhängig von der Anzahl der Operanden im 2- oder im 3-Register-Format codiert (Bild 4.7 und 4.8). Im 3-Register-Format stehen insgesamt 5 Bits zur Codierung der Operation zur Verfügung, im 2-Register-Format sogar 6 Bits. Hierdurch sind alle Befehle dieser Gruppe im MPM3 codierbar. Das allgemeine Format für Befehle mit zwei Sourceoperanden lautet:

<mnemo> <dgpreg>, <sgpreg1>, <sgpreg0>

Die Befehle mit nur einem Quellregister (und einem Zielregister) werden wie erwähnt im 2-Register-Format codiert. Hierzu zählen das Dekrement und Inkrement sowie die Schiebe- und Rotationsbefehle ASL, ASR, LSR, ROL und ROR. Als Nebeneffekt sind für diese Befehle alle Register R0 bis R12 erreichbar.

Die andere Ausnahme besteht in dem Vergleichsbefehl (CMP), für den aus Performancegründen sowohl eine Codierung im 2-Register-Format (Vergleich für alle Register R0 bis R12) als auch im Register, Data-Format vorgesehen ist. Für die CMP-Instruktion müssen zwei Source-, jedoch kein Zielregister (Destination) angegeben werden. Zusätzlich ist die Integration von Vergleichen mit Programmkonstanten für die Geschwindigkeit der Programme sehr wichtig.

Der Vergleichsbefehl setzt im Übrigen auch das Overflow-Flag, um bei Vergleichen für vorzeichenbehaftete Zahlen eine korrekte Entscheidung treffen zu können. Bis auf die Rückspeicherung entspricht dieses Verhalten dem des Subtraktionsbefehls.

4.3.3.3 Flagbefehle

Die wenigen Flagbefehle des MPM3 werden sehr einfach in einer auf 16 Bit erweiterten Form codiert (→ Tabelle 4.2).

4.3.3.4 Kontrollfluss-Befehle

Die letzte Gruppe, die Kontrollfluss-Befehle, besteht aus Verzweigungs-, Sprung- und Unterprogrammbefehlen. Abgesehen von den Befehlen NOP, RTI und RTS, die eine implizite Adressierung besitzen und im allgemeinen Format codiert werden, bieten sich für die anderen Befehle eine Register-indirekte und eine relative Adressierung an, um ohne zusätzliche Belastung des Busses die Sprünge durchzuführen.

Die relative Adressierung wird bei Branchbefehlen durch 8 Bit, bei dem Sprungbefehl (JMP) durch eine 11-Bit-Konstante beschrieben, wobei diese Konstante jeweils eine Sprungdifferenz in 16-Bit-Werten umfasst. Unbedingte Sprünge können aus diesem Grund weitere Entfernungen im Code überwinden, ohne auf die Register-indirekte Adressierung zurückgreifen zu müssen.

Bei den Branchbefehlen sind neu die Codierungen für Branch Above (BA, springt bei NF = 0 und ZF = 0) sowie Branch Below or Equal (BBE, verzweigt bei NF = 1 oder ZF = 1) hinzugekommen. Diese Befehle sind für vorzeichenrichtige Vergleiche und Kontrollstrukturen geeignet, während für vorzeichenlose Zahlen weiterhin BGT, BLE usw. genutzt werden sollten. Die Befehle BAE und BB entsprechen BPL bzw. BMI.

Für den CALL-Befehl muss die Rücksprungadresse zwischengespeichert werden. Während dies bei vielen CISC-Architekturen, z.B.dem MPM2 [3], automatisch auf dem Stack geschieht, wird für das MPM3 eine Speicherung im Link Address Register LA (R11) vorgesehen, aus dem heraus auch die Restaurierung beim

Rücksprung mittels RTS erfolgt. Soll zwecks Verhinderung eines Überschreibens dieses Register auf dem Stack gerettet werden, muss dies in der Unterprogrammroutine durch einen explizit aufgeführten PUSH R11 und die Rückspeicherung durch POP R11 geschehen.

Für einen Interrupt Request gilt das gleiche, ergänzt durch das Link Status Register (LS, R10). Hier werden beide Register mit den Werten vor Aufruf der Service Routine beschrieben, die Adresse der Service Routine muss zudem im Vector Register (VR, R12) stehen. Diese Form der Registerverwaltung ist zwar komplexer, entspricht jedoch der RISC-Philosophie, insbesondere, da dem Entwickler bzw. Compilerbauer alle Möglichkeiten in die Hand gegeben werden, nur die notwendigen Sicherungen durchzuführen und somit Zeit zu sparen.

4.3.3.5 Codetabelle für MPM3

Die in Tabelle 4.2 dargestellte Codierungstabelle für MPM3 zeigt das höherwertige Byte der jeweiligen Codierung für eine Instruktion. Durch die Einschränkung auf teilweise weniger als 8 Bit pro Code, begründet in dem erhöhten Platzbedarf der Operandendarstellung, treten einige Instruktionen mehrfach in der Tabelle auf, so z.B. MOV register, #data (40h .. 47h). Dies bedeutet in der Praxis, dass die Bitfolge ’01000xxx‘ den Befehle MOV mit der Adressierungsart immediate codiert.

Die in Tabelle 4.2 eingetragene Adressierungsarten bedeuten im einzelnen:

- Ohne Eintragung: Implicit, Instruktion benötigt keine Operanden.
- Imm.: Immediate, die Adressierung erfolgt im Format <register>, #data, wobei für das <register> eine Register-direkte (alle Register), für #data die unmittelbare Adressierung genutzt wird.

low high	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP	CLC	CLI	SEC	SEI				RTS	RTI						
1																
2	BCS relat.	BCC relat.	BEQ relat.	BNE relat.	BVS relat.	BVC relat.	BMI relat.	BPL relat.	BGT relat.	BLE relat.	BA relat.	BBE relat.				
3	JMP relat.	JMP relat.	JMP relat.	JMP relat.	JMP relat.	JMP relat.	JMP relat.	JMP relat.	JMP indirect				CALL indirect			
4	MOV #imm.	MOV #imm.	MOV #imm.	MOV #imm.	MOV #imm.	MOV #imm.	MOV #imm.	MOV #imm.								
5									MOVH #imm.	MOVH #imm.	MOVH #imm.	MOVH #imm.	MOVH #imm.	MOVH #imm.	MOVH #imm.	MOVH #imm.
6																
7	CMP #imm.	CMP #imm.	CMP #imm.	CMP #imm.	CMP #imm.	CMP #imm.	CMP #imm.	CMP #imm.								
8	MOV reg.-dir.				CMP reg.-dir.				LD indirect	ST indirect			POP reg.-dir.	PUSH reg.-dir.		
9	DEC reg.-dir.				INC reg.-dir.				ASL reg.-dir.				ASR reg.-dir.			
A	LSR reg.-dir.				ROL reg.-dir.				ROR reg.-dir.							
B																
C	ADD reg.-dir.	ADD reg.-dir.	ADC reg.-dir.	ADC reg.-dir.	SUB reg.-dir.	SUB reg.-dir.	SBC reg.-dir.	SBC reg.-dir.	AND reg.-dir.	AND reg.-dir.	EOR reg.-dir.	EOR reg.-dir.	OR reg.-dir.	OR reg.-dir.		
D																
E																
F																

Tabelle 4.2: Maschinencodes für MPM3

- Reg.-dir.: Register-direct, alle Operanden werden als interne Prozessorregister interpretiert. Die Anzahl variiert von einem Register (PUSH, POP) über zwei (MOV, CMP, ASL, ASR, LSR, ROL, ROR) auf drei Register (ADD, SUB, AND, EOR, OR), wobei grundsätzlich die Formate

  ```
  <operation> <ziel>, <quelle0>, <quelle1>
  <operation> <ziel>, <quelle>
  <operation> <quelle0>, <quelle1> (nur CMP)
  ```

  mit ziel = <quelle0> <operation> <quelle1> bzw. ziel = <operation> <quelle> gilt.
- Indirect: Die Indirektion bezieht sich immer auf einen Registerinhalt als Quelle (LD) oder Ziel (ST, JMP, CALL). Die Angabe des zweiten Operanden für LD und ST erfolgt Register-direkt.
- Relat.: Relative, die Adressierung erfolgt relativ zum aktuellen Program Counter im 8-Bit- (Branch-) bzw. 11-Bit-Format (JMP-Instruktion).
- Bei den Branchbefehlen gibt es mnemotechnische Doppelbedeutungen: BPL/ BAE, BMI/BB, BCS/BGE und BCC/BLT sind identisch.

4.3.4 Ablauf der Instruktionen im MPM3

Seitens der Befehle und Adressierungsarten wurde in der MPM3-Definition alles vorbereitet, um einen Ablauf jeder Instruktion in vier etwa gleichgewichtigen Phasen durchzuführen:

Fetch/Predecode: Die Fetch- und Predecode-Phase beinhaltet das Laden des Instruktionscodes einschließlich aller Operanden. Dies ist in einem Takt möglich, da das Datenbusinterface zum Speicher mit einer Breite von 16 Bit definiert und die Befehle entsprechend codiert wurden. Das Predecode beinhaltet die Umwandlung des ursprünglichen Formats in ein einheitliches internes Befehlsformat zur weiteren Verarbeitung sowie die Detektierung von Branch- und Sprungbefehlen.

Decode/Load: In der Decode/Load-Phase werden alle notwendigen Decodierungen zur weiteren Verarbeitung vollzogen sowie alle Operanden in interne Register zur Weiterverarbeitung geladen. Dies beinhaltet ein mögliches **Data Forwarding**.

Für Kontrollflussbefehle kann in dieser Phase frühestens entschieden werden, an welcher Stelle das Programm weitergeführt wird. Hierzu ist bei bedingten Verzweigungen ggf. ein *Data Forwarding* (→ 4.4.1) notwendig, um die Flags korrekt auswerten zu können.

Execute/Memory: Arithmetische und logische Operationen werden in dieser Phase aktiv. Für Load-Befehle erfolgt hier der Zugriff auf das Memory, sodass bei einer Von-Neumann-CPU mit einem Bussystem eine Verzögerung der nachfolgenden Befehle auftreten wird. Entsprechendes gilt für einen Store-Befehl und den daraus resultierenden Schreibzugriff auf das externe Memory.

Write Back: Alle Ergebnisse werden abschließend in dieser Phase in die Zielregister eingetragen.

4.4 Pipeline-Struktur

Die Darstellungen im vorangegangenen Abschnitt beruhen im Wesentlichen auf einer Abstimmung zwischen dem Befehlssatz, den notwendigen Adressierungsarten, dem Registersatz einschließlich der Special-Purpose-Register sowie der Pipeline-Grundstruktur. Letztere spielt insofern schon eine Rolle, weil die Beschränkung auf vier Pipelinestufen die Zusammenfassung von Execute und Memory Access bedeutet. Eine 5stufige Pipeline mit den Phasen Fetch, Decode/ Load, Execute, Memory Access und Write Back böte beispielsweise die Möglichkeit zu erweiterten Adressberechnungen (indizierte Adressierung) in der Execute-Phase. Dies muss aus Zeitgründen im Rahmen der 4stufigen Pipeline entfallen.

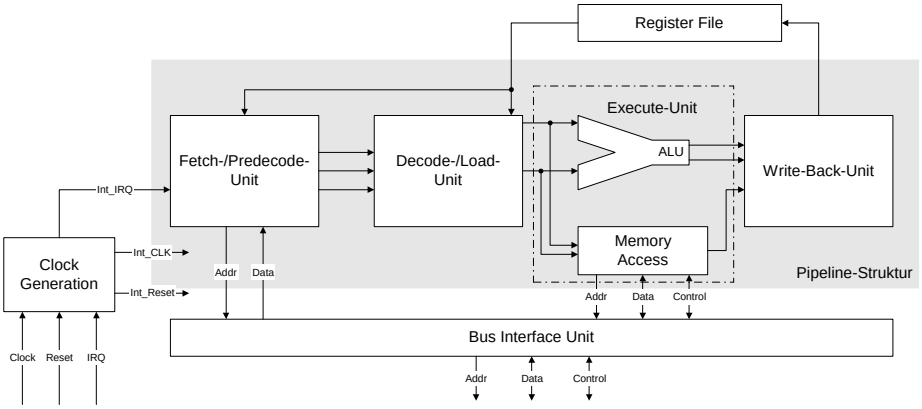


Bild 4.10: Basisversion Pipeline-Struktur (4stufig, MPM3)

Die Grundstruktur der Pipeline ist in Bild 4.10 dargestellt. Eine Instruktion, durch die Fetch/Predecode-Unit geladen, durchläuft diese Pipeline in 4 aufeinanderfolgenden Takten, während nachfolgende Befehle jeweils in den früher zu durchlaufen Pipelinestufen bearbeitet werden. Dieses Grundprinzip bedeutet eine mögliche Beschleunigung um den Faktor 4.

Bild 4.10 zeigt ebenfalls die übrigen Einheiten der Modell-CPU MPM3 in der Basisversion. Diese Einheiten haben teilweise reinen Supportcharakter (wie z.B. die Takterzeugung), wirken teilweise jedoch auch Pipeline-übergreifend (Bus Interface Unit, Register File) und sind dadurch besonders zu betrachten.

Bei der Ausführung von Programmen kommt es niemals zu einer Beschleunigung um den Faktor 4 gegenüber einer Version ohne Phasenpipelining, da sich Störungen (Hazards) einstellen werden. Diese Hazards lassen sich in 3 Kategorien einteilen: Kontrollfluss-, Daten- und strukturelle Hazards.

4.4.1 Datenhazards

Das folgende Programm, in der Assemblersprache von MPM3 verfasst, zeigt Datenabhängigkeiten zwischen unmittelbar aufeinanderfolgenden Befehlen.

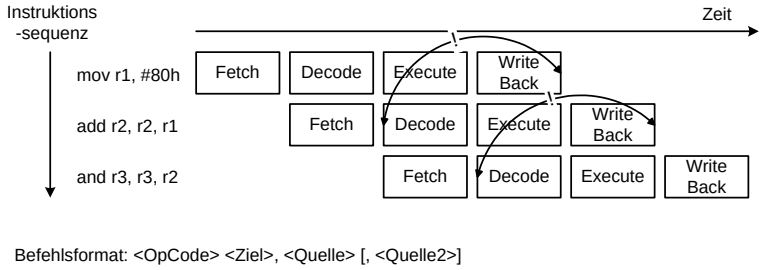


Bild 4.11: Befehlssequenz mit Datenabhängigkeiten

In dieser Sequenz tritt zweifach der Fall auf, dass in einer Decode/Load-Phase einer Instruktion Daten geladen werden, die am Ende der Write-Back-Phase der vorangegangenen Instruktion erst mit Sicherheit in den Registern gespeichert sein wird. Eine Pipeline, die auf diese Abhängigkeiten keine Rücksicht nimmt, wird daher falsche Rechenergebnisse liefern.

Das falsche Ergebnis muss dadurch verhindert werden, dass der Ablauf in der Pipeline entweder solange stoppt, bis die korrekten Inhalte vorliegen, oder durch geeignete Maßnahmen innerhalb der CPU mit den korrekten Werten verläuft. Vor allem letztere Lösung erschwert das Design einer CPU in erheblichem Maß!

Die Variation des Ablaufs in der Pipeline erfolgt durch Einfügen von Verzögerungen, zumeist als **Bubble** (Blase) oder *Injected Instruction* bezeichnet. Im Fall des vorliegenden Beispielprogramms ergibt dies folgendes, in Bild 4.12 dargestelltes Verhalten. Die Performance des Prozessors leidet unter dieser Maßnahme erheblich, im Beispiel wird sie von theoretisch 3 Zyklen für 3 Befehle auf 7 Zyklen (0.43 Instruktionen/Takt, IPC, Instructions per Cycle) gedrückt.

Eine Analyse der möglichen Ursachen für diese Form von Datenhazards ergibt für die RISC-Architekturen exakt einen Typ, den sogenannten **Read-After-Write-Hazard** (**RAW**), der zwar nicht in dem ursprünglichen Assemblerprogramm auftritt (hier wird von einer zeitlich sequenziellen Folge ausgegangen), wohl aber durch das Pipelining in seinen Konsequenzen stark bemerkbar ist. Dieser RAW-Hazard wird bei der superskalaren CPU als einziger nicht-behebbarer Hazard in Erscheinung treten, während für die Pipelining-CPU (RISC) außer der bisherigen Lösung des Einbaus von Wartezyklen in der Pipeline zwei weitere Möglichkeiten der Vermeidung von Konsequenzen existieren:

1. Im Assemblerprogramm wird, entweder durch den Assemblerprogrammierer oder durch Analysen im Compiler, weitgehend vermieden, dass ein Read-After-Write-Hazard zu zeitlichen Verzögerungen führt. Dies ist durch eine Sortierung der Befehle möglich.
2. In der CPU werden zusätzliche Teile integriert, die die Auswirkungen vermeiden. In diesem Fall kann ggf. auf ein Anhalten der Pipeline verzichtet werden.

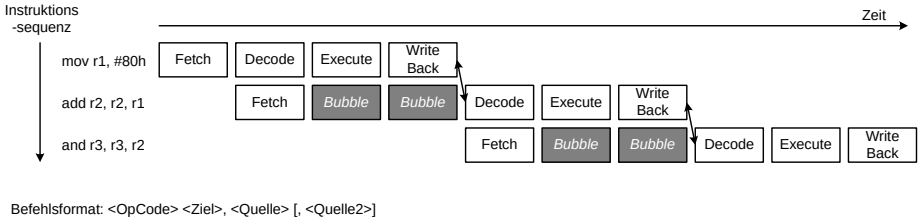


Bild 4.12: Behebung der fehlerhaften Berechnung bei Datenabhängigkeiten durch Einfügung von Wartezyklen (Bubbles)

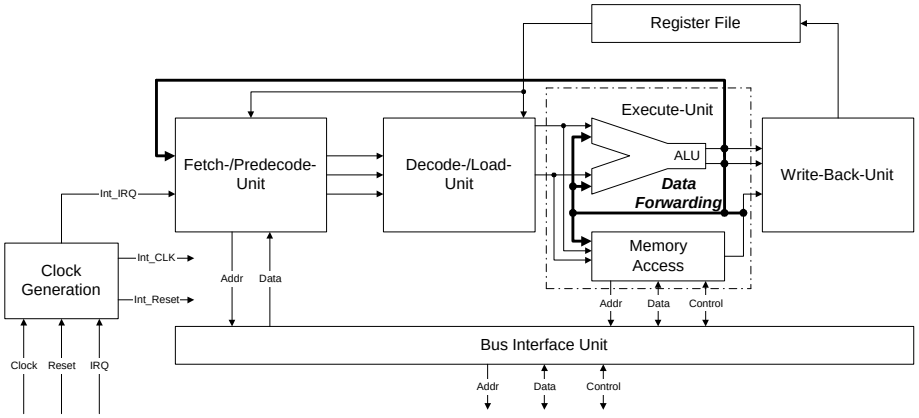


Bild 4.13: 4stufige Pipeline, erweitert durch Data Forwarding (fett gezeichnet) für Werte aus der ALU sowie der Memory Access Unit

Durch eine additive **Data Forwarding Unit**, die zusätzlich zu den bisher vorhandenen Einheiten innerhalb der Mikroarchitektur eingefügt wird, lassen sich weitere Auswirkungen mildern oder sogar ganz beseitigen, indem nicht auf das Schreiben eines Registers gewartet wird, sondern das am Ausgang der ALU vorhandene Ergebnis auf einen Eingang anstelle des späteren Registerinhalts zurückgeführt wird. Bild 4.13 zeigt dies für die 4stufige Pipeline der MPM3-Modell-CPU. In diesem Fall lässt sich der negative Einfluss von Datenabhängigkeiten im Programm nahezu gänzlich vermeiden, was für Prozessoren mit hoher Anzahl von Pipelinestufen nicht gilt. Die einzige, unvermeidbare Ausnahme gilt für Registerinhalte, die für Sprungbefehle (Jump Register-indirekt) benötigt werden. Da der Sprung in der Fetch-Unit zu einem neuen Program-Counter-Wert führen muss und somit dort ausgeführt wird, erreicht der Datenwert diese Einheit zwei Takte zu spät, falls der Wert im vorangegangenen Befehl noch berechnet wird.

Der Aufwand für das Data Forwarding erscheint relativ gering. In der Praxis nimmt dies jedoch einen erheblichen Anteil der Mikroprozessorkerns in Anspruch, da detektiert werden muss, ob die Registerinhalte oder ein Wert aus dem Forwarding genutzt werden muss, welcher Wert ausgewählt wird usw. Dennoch wird dies Verfahren bei allen gängigen Prozessoren eingesetzt, weil die Performanceverbesserung (→ 4.6) erheblich ist.

4.4.2 Strukturelle Hazards

Ein vollkommen anderer Konflikt entsteht, wenn zwei Einheiten in der Pipeline zugleich eine dritte nutzen müssen, um ihre Aufgabe zu erfüllen. Wie bereits Bild 4.11 zeigt, findet dies sowohl beim Register File als auch bei der Bus Interface Unit statt: Für beide Einheiten existieren zwei Einheiten in der Pipeline, die diese gegebenenfalls gleichzeitig nutzen wollen. Diese Form von Störung wird **struktureller Hazard** genannt.

Register File

Die Lösung für das Register File ist vergleichsweise einfach: Während die Write-Back-Einheit nur schreiben kann, kann die Decode/Load-Einheit nur lesen. Dieser Konflikt lässt sich dadurch auflösen, indem ein Multiportzugriff implementiert wird, der zugleich Datenkonflikte (Update eines gerade gelesenen Wertes) auflöst.

Speicherzugriffe

Der Speicherzugriff hingegen ist wirklich konkurrierend: In jedem Takt wird gemäß Pipeline-Ablaufstruktur ein Fetch (Laden einer codierten Instruktion) durchgeführt. Der Zugriff auf den Datenspeicher ist dabei sicher seltener, führt jedoch dann garantiert zu einem Konflikt.

Dieses Problem ist sehr ernst zu nehmen und wird bei Spezialprozessoren wie DSPs (Digitalen Signalprozessoren) noch wesentlich verschärft, weil häufig Algorithmen (z.B. Filterfunktionen, Echounterdrückung etc.) mit sehr hohem Datenaufkommen implementiert werden.

Die Einfügung der beiden potenziellen (bei DSPs sogar 3 oder 4 gleichzeitigen) Speicherzugriffe in eine zeitliche Sequenz innerhalb eines Taktes muss ausscheiden, weil Speicherzugriffe zu den langsamsten Aktionen bei Instruktionsabläufen gehören. Wie in weiterführenden Kapiteln noch ausgeführt werden wird, ist es nur über Speicherhierarchien überhaupt noch möglich, mit ungebremster Geschwindigkeit auf dem Speicher zu arbeiten. Eine zeitliche Sequenz mehrerer Zugriffe pro Takt würde den Takt selbst stark limitieren.

Um dennoch Auswirkungen dieses strukturellen Hazards zu verhindern oder wenigstens zu mindern, wurden folgende Verfahren bzw. Architekturen entwickelt:

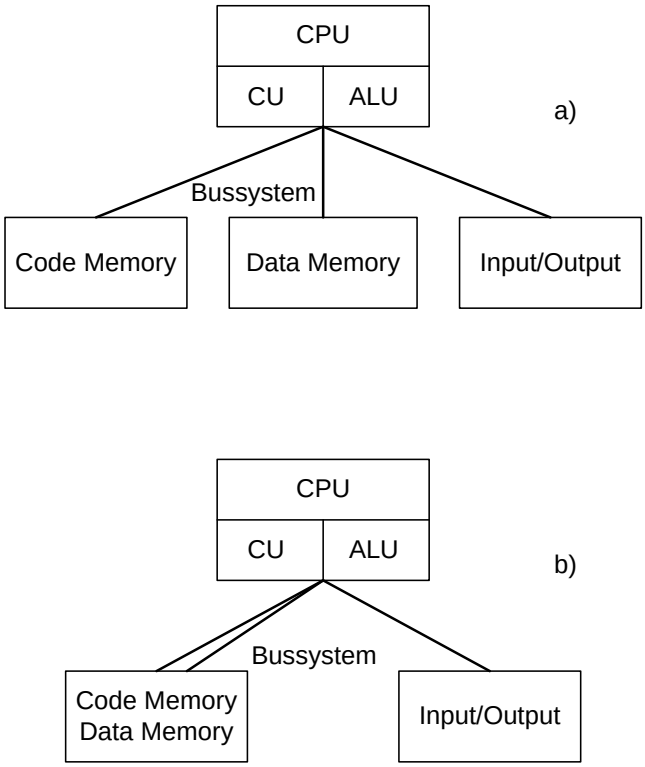


Bild 4.14: Harvard-Architektur a) ursprüngliche Variante mit getrennten Code- un Datenadressräumen b) modifizierte Variante mit einheitlichem Adressraum, aber getrennten Zugriffswegen

1. Die RISC-Architekturen sind mit einer möglichst großen Anzahl von Registern ausgestattet, um externe Speicherzugriffe zu minimieren. Das Prinzip beruht darauf, lokale Variablen möglichst lange im Register zu halten. Alle Rechenoperationen wie Addition oder logische Verknüpfungen sind auf Registern als Operanden definiert, der Speicherzugriff erfolgt ausschließlich via Load- und Store-Befehlen (LD, ST, PUSH, POP).

   Dieses Verfahren minimiert zumindest die Anzahl der Zugriffe. Weiterführende Architekturen (SPARC, Intel IA-64) nutzen sehr große Register-Files auch dadurch, dass hier für Unterprogramme Übergabewerte (Aufrufparameter) gespeichert werden können (globale und lokale Registerfenster). Dies spart das umständliche Kopieren der Registerinhalte auf den Stack.
2. Die Harvard-Architektur (Bild 4.14) bietet einen aufwendigen, jedoch perekten Ausweg. Bild 4.14 zeigt die ursprüngliche Definition mit getrennten Adressräumen für Instruktionen und Daten Diese Definition wurde so gewählt, um

Instruktions- und Datenzugriffe gegenseitig auszuschließen. Die modifizierte Variante hingegen kennt nur einen Adressraum, aber 2 (oder mehrere) Zugriffswege. Man spricht auch von **Multiporting**, auf den Speicher bezogen.

Die modifizierte Harvard-Architektur wird heute meist auf Cache-Speicher-Ebene (→ 7) im Level-1-Cache verwendet, nicht jedoch auf Hauptspeicherebene. Die Gründe sind einfach: Die Kosten für externes Multiporting sind sehr hoch, außerdem ist der Hauptspeicher meist so langsam, dass Wartezyklen eingefügt werden müssen (und damit ein Multiporting vergleichsweise sinnlos ist).

Auf L1-Cache-Ebene hingegen werden alle Zugriffe in einem Takt durchgeführt, der Cache ist mittlerweile fast schon integraler Bestandteil des Prozessorkerns. Hier wirkt sich Multiporting sehr beschleunigend aus.

Für DSPs wird ausschließlich Multiporting genutzt, da die Datenmengen jeden Registerfile (Variante 1) überfordern würden.

Weitere strukturelle Hazards

Ein weiterer struktureller Hazard tritt auf, wenn die Folgen eines Kontrollflusshazards (→ 4.4.3) gemindert werden sollen. Dies wird im nachfolgenden Abschnitt besprochen.

4.4.3 Kontrollflusshazards

Die dritte Störungsart bei der Ausführung von Programmen in Prozessoren mit Pipelining wird als Kontrollflusshazards bezeichnet. Diese Art ist die aktuell wichtigste, da Datenabhängigkeiten sowie strukturelle Hazards im Wesentlichen aufgelöst oder doch entscheidend gemindert werden können.

4.4.3.1 Ursache und Auswirkungen der Kontrollflusshazards

Kontrollflusshazards entstehen an den Stellen im Programm, an denen der Kontrollfluss (der Programmverlauf) von der üblichen Sequenz abweicht. Hierfür muss ein Sprungbefehl (unkonditionierte oder bedingte Sprünge wie JMP (jump), BNE (branch if not equal) oder Unterprogrammaufrufe wie CALL sowie RET (return)) im Programmcode vorliegen, und das ist aktuell recht häufig (→ 4.1). Man geht aktuell davon aus, dass alle 4–5 Befehle ein Kontrollflussbefehl, in der überwiegenden Anzahl der Fälle ein bedingter Verzweigungsbefehl (conditional branch) auftritt.

Das Problem bei der Ausführung der Kontrollflussbefehle ist ein zweifaches: Zum einen muss die neue PC-Adresse ermittelt werden, ggf. durch Berechnung, zum anderen ist die Sprungrichtung bei den bedingten Befehlen unbekannt und ggf. von der Ausführung des direkten Vorgängerbefehls abhängig.

4.4.3.2 Berechnung der Sprungadresse

Die Komplexität der Ermittlung der neuen PC-Adresse hängt von der verwendeten Adressierungsart ab. Die Angabe einer absoluten Sprungadresse (codiert als vollständige Adresse) entfällt meist bei RISC-Architekturen, da nicht genügend Codierungsplatz im Befehlsformat vorhanden ist (→ 4.3.3.4, 4.3.3.5, Tabelle 4.2). Jedes andere Format benötigt hingegen Zusatzinformationen und/oder Rechenkapazität:

- Bei verkürzter absoluter Adressierung (absolut short, im MPM3 nicht implementiert) wird nur ein Teil der Adresse (die niederwertigen *k* Bits) gespeichert, die höherwertigen Bits des PC bleiben erhalten. Dieses Adressierungsformat benötigt die geringste Rechenkapazität und ist im Rahmen der Fetch/Predecode-Phase direkt ausführbar. Nachteilig ist, dass auf diese Weise nur ein Teil des Adressraums erreichbar ist.
- Bei indirekter Adressierung (Register-indirect) wird die neue PC-Adresse zuvor in einem Register gespeichert und beim Sprung in den PC übertragen. Hier ist keine Rechenkapazität erforderlich, aber ggf. ein Data Forwarding (→ Bild 4.13 Weg von Execute- in Fetch-Einheit). Dieses Forwarding kann im Fall des MPM3 die Anzahl der Wartezyklen auf 1 begrenzen.
- Bei relativer Adressierung (PC-relative), die für bedingte Verzweigungen sehr beliebt ist, muss die neue PC-Adresse durch vorzeichenbehaftete Addition des Program Counter und des Offsets berechnet werden. Dies könnte im Programmablauf bereits in der Phase 2 (eigentlich Decode/Load) geschehen, benötigt aber die ALU, die zu diesem Zeitpunkt mit dem Vorläuferbefehl befasst ist (struktureller Hazard). Aus diesem Grund muss bei relativer Adressierung ohne weitere Maßnahmen mit 2 Wartezyklen gerechnet werden, bevor das Ergebnis per Data Forwarding an die Fetch-Einheit weiterleitet.

Das relative Adressierungsformat ist sehr beliebt, da es sehr kompakt ist und durch den Orts-unabhängigen Offset (im Gegensatz zum verkürzten absoluten Format) jederzeit verwendet werden kann. Die Programmlokalität erlaubt eine häufige Nutzung auch bei eingeschränktem Versatz (z.B. +127/-128 Programmadressen bei 8-Bit-Offset).

Konsequenterweise wird in Prozessorarchitekturen versucht, gerade die Ausführung dieser Adressierung mit möglichst geringer Verzögerung auszuführen. Um die Adressrechnung vorzuziehen, wird zur Auflösung des strukturellen Hazards eine Addierereinheit (Look-Ahead Resolution) eingefügt (→ Bild 4.15). Dieses Verfahren verkürzt die Anzahl der Wartezyklen (im Rahmen der 4stufigen Pipeline) auf 1, sodass jeder Branch-Befehl in der MPM3-Architektur in 2 Zyklen ausführbar ist.

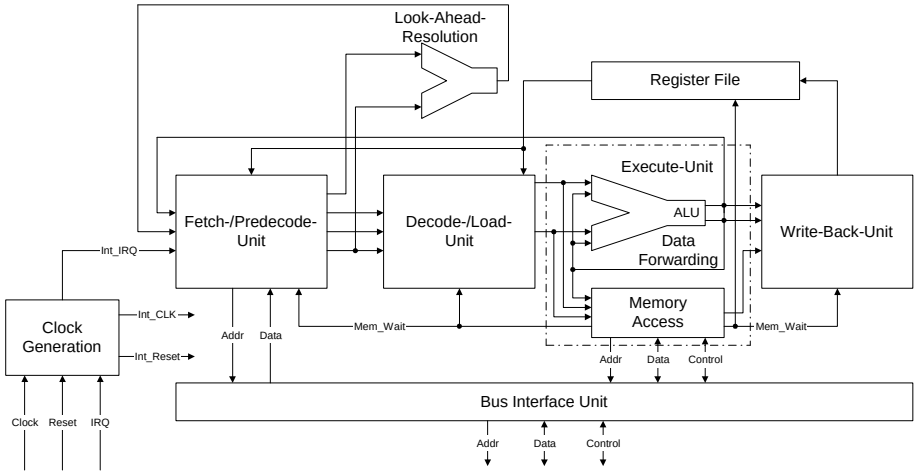


Bild 4.15: 4stufiges Pipelining mit Data Forwarding und Look-Ahead Resolution

4.4.3.3 Weiterentwicklung der Verzweigungsbefehle

Die Häufigkeit von Verzweigungsbefehlen lässt es als notwendig erscheinen, weitere Maßnahmen zur Reduktion auf einen Ausführungstakt (0 Wartezyklen) zu ergreifen. Die Motivation ist sehr einfach: Allein durch die 20prozentige Häufigkeit der Verzweigungsbefehle mit 1 Wartezyklus reduziert sich die theoretische Befehlsrate von 1 IPC (Instruction per Clock) auf 0,83 IPC (6 Takte für 5 Befehle).

Die Maßnahmen gliedern sich in 2 Kategorien: Definition neuer Branchbefehle ohne Wartezyklen und die Einführung einer Verzweigungsvorhersage.

Delayed-Branch-Befehle

Die sogenannten **Delayed-Branch-Befehle** führen die Verzweigung erst nach dem unmittelbar nachfolgenden Befehl aus. Dieser nachfolgende Befehl steht im **Delay-Slot** und muss also in jedem Fall ausführbar sein (weil er sowohl im verzweigenden als auch im nicht-verzweigenden Fall zur Ausführung kommt).

Dieser Delay-Slot kann in jedem Fall durch einen NOP-Befehl (No Operation) gefüllt werden (wodurch allerdings nichts gewonnen ist). Meist ist die Verwendung der Befehle im Delay-Slot eingeschränkt, beispielsweise dürfen keine Branchbefehle dort eingesetzt werden.

Die Delayed-Branch-Befehle führen natürlich zu unleserlichem Assemblercode, weil bei der Interpretation des Codes (durch den Menschen) auch der nachfolgende Befehl hinzugerechnet werden muss. Die automatische Generierung eines nichttrivialen Befehls im Delay-Slot durch einen Compiler erweist sich meist als

schwierig. Da zudem die Delayed-Branch-Befehle für superskalare Prozessoren (→ 5) keinen Vorteil bringen, wird diese Technik wenig eingesetzt.

Eine Variation der *Branchbefehle mit Delay-Slot* sind die **Branch-Likely-Befehle** mit Delay-Slot. Diese arbeiten prinzipiell ähnlich, auch hier wird die nachfolgende Instruktion ausgeführt. Diese Ausführung wird allerdings nur beendet (mit Schreiben der Ergebnisse, wenn die Verzweigung wirklich durchgeführt wurde. Im Fall der Nicht-Verzweigung wird der Befehl im Delay-Slot nachträglich zum NOP-Befehl mutiert.

Die Branch-Likely-Befehle zielen auf Schleifen, und zwar auf die Verzweigung am Ende. Diese wird gewöhnlich häufig durchlaufen, und nur einmalig nicht ausgeführt (am Ende der Ausführung der Schleife). Wie das Beispiel in Bild 4.16 zeigt, ist in diesem Fall die automatische Besetzung des Delay-Slots mit einem nicht-trivialen Befehl immer möglich, sodass die Branch-Likely-Befehle besser nutzbar sind. Die Bemerkung zum Einsatz in superskalaren Prozesoren gilt allerdings auch hier.

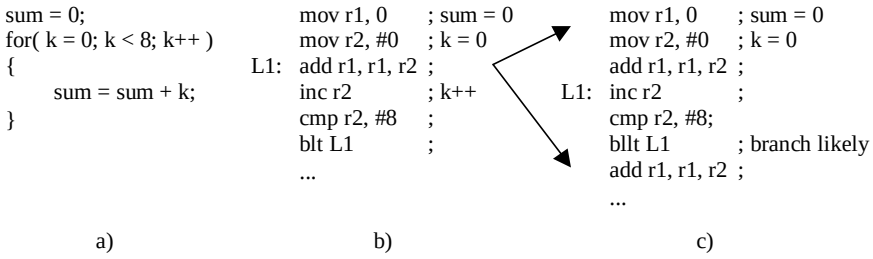


Bild 4.16: Verwendung von Branch-Likely-Befehlen
a) Sourcecode b) Übersetzung ohne Branch-Likely-Instruktionen c) Generierung von Assemblercode mit Branch-Likely-Befehlen

Verzweigungsvorhersage

Die Einführung und Verwendung von Branchbefehlen mit Delay-Slot stellt im Allgemeinen keine befriedigende Lösung dar, insbesondere im Hinblick auf superskalare Mikroprozessoren. Aus diesem Grund wurde schon frühzeitig eine **Verzweigungsvorhersage** (**Branch Prediction**) eingeführt, deren Ziel es ist, mit hoher (statistischer) Wahrscheinlichkeit den korrekten Verlauf des Programms vorherzubestimmen.

Die Verzweigungsvorhersage stützt sich entweder auf Annahmen über den Programmverlauf (statische Sprungvorhersage) oder auf Rückschlüsse aus dem bisherigen Verlauf (dynamische Sprungvorhersage).

Über alle Sprung- und Verzweigungsbefehle (auch die unkonditionierten) gemittelt verzweigen etwa zwei Drittel aller Fälle, während ein Drittel den Kontrollfluss

nicht ändert. Grob lässt sich dadurch abschätzen, dass Rückwärtssprünge häufig aus Schleifen (Loop) stammen und nahezu immer verzweigen (Rückwärtssprung), Vorwärtssprünge jedoch aus bedingten Kontrollstrukturen mit einer Gleichverteilung der Sprungwahrscheinlichkeiten («Branch Taken»/«Branch Not Taken»).

Im Rahmen einer **statischen Sprungvorhersage** (*Static Branch Prediction*) wird im einfachsten Fall daher angenommen, dass sich der Kontrollfluss nicht ändert. Die nachfolgenden Befehle werden ausgeführt, die Ergebnisse jedoch erst gespeichert, wenn sicher ist, dass die Verzweigung nicht eingetreten ist. Im anderen Fall werden alle Operationen nachträglich als «No Operation» deklariert und die Ergebnisse verworfen. Dies führt im Fall der 4stufigen Pipeline (MPM3) zu einem Mittel von 1,67 CPI (Clocks per Instruction) oder einem IPC von 0,6 pro Verzweigunsbefehl.

Als Erweiterung der *statischen Sprungvorhersage* wird für Vorwärtssprünge ein «Branch Not Taken» (keine Kontrollflussänderung), für Rückwärtssprünge hingegen ein «Branch Taken» angenommen. Dies bedeutet jedoch, dass das Sprungziel in einem **Branch Target Buffer** oder **Branch Target Cache** gespeichert werden muss, da die Look-Ahead Resolution ebenfalls einen Takt zur Adressberechnung benötigt. Daher wird meist die erste Rückwärtsverzweigung verzögert ausgeführt und das Sprungziel für weitere Durchläufe gespeichert.

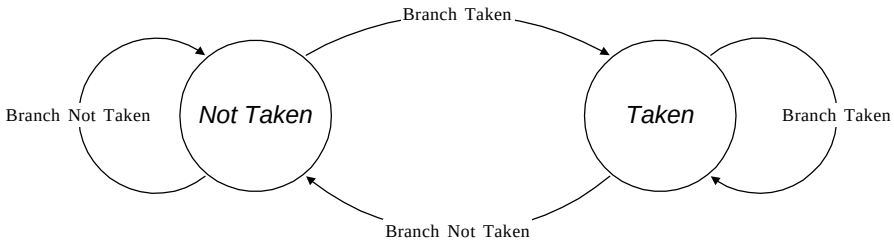


Bild 4.17a: 1-Bit-saturierender Zähler als Verzweigungsvorhersage

In Form einer **dynamischen Sprungvorhersage** (*Dynamic Branch Prediction*) und eines internen Speichers für die letzten Sprungziele (*Branch Target Cache*) ist eine weitere Verringerung der mittleren Wartezeit durch Kontrollflusshazards möglich. Die Sprungvorhersage wird häufig in Form eines Zustandsautomaten (*1- oder k-bit Vorhersageautomat, k-bit Predictor*) realisiert. Bild 4.17a/b zeigt diesen als 1- und 2-bit saturierenden Zähler mit den Zuständen «Taken» und «Not Taken» bzw. «Strongly Taken», «Weakly Taken», «Weakly Not Taken» und «Strongly Not Taken».

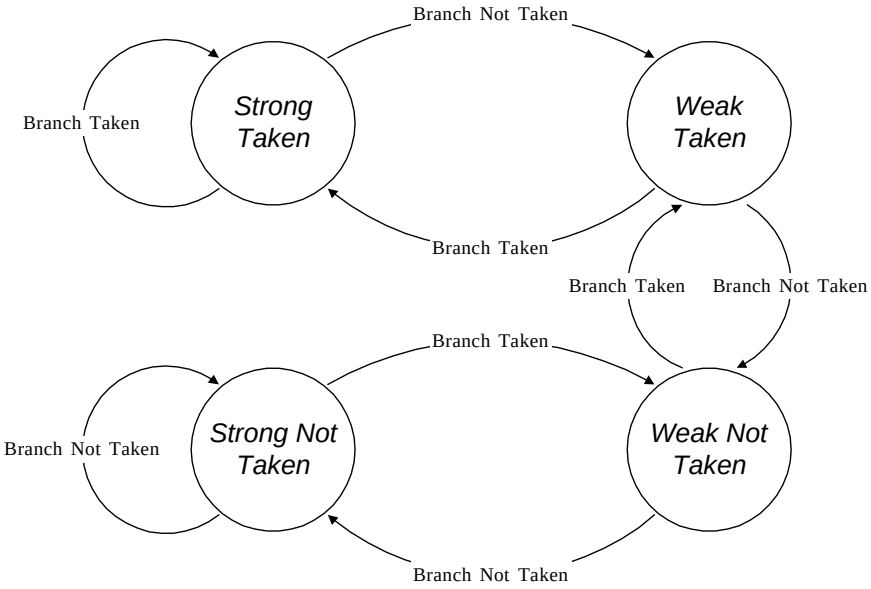


Bild 4.17b: 2-Bit-saturierender Zähler als Verzweigungsvorhersage

Die Anzahl der Vorhersageautomaten und der Einträge im Branch Target Cache bestimmt die Anzahl der Verzweigungen in einem Bereich, der aktuell ausgeführt wird. Um also mehrere Sprünge in einem Bereich, etwa eine Schleife, berücksichtigen und vorhersagen zu können, werden mehrere dieser Automaten benötigt. Die Unterscheidung zwischen den einzelnen Verzweigungen im Programmkontext erfolgt dann meist über die niederwertigen Bits der Programmadresse (Beispiel: unteren 10 Bits bei 1024 Sprungvorhersageautomaten mit Branch Target Cache).

Wechselwirkungen zwischen den einzelnen Sprüngen bzw. Muster in dem Sprungverhalten (= Wechselwirkung mit sich selbst) werden so nicht berücksichtigt, sind in der Praxis jedoch häufig vorhanden. Mit diesen Erkenntnissen wurden als Weiterentwicklung die **Korrelations-basierten Vorhersagen** (*Correlation-Based Predictors*) eingeführt [11].

Ein **(*m*,*n*)-Korrelations-basierter Vorhersageautomat** besteht aus 2^m Vorhersageautomaten für jeweils eine Verzweigung, und jeder dieser Vorhersageautomaten ist wiederum als *n*-bit-Vorhersageautomat aufgebaut. In einem **Branch History Register** (BHR) wird das globale Verhalten der *m* letzten Verzweigungen in Form eines *m*-bit-Shiftregister gespeichert. Die entstehenden 2^m Muster dienen als Index in die **Pattern History Table** (PHT), dem Array der 2^m Vorhersageautomaten mit typischerweise 2-bit-Auflösung. Um die getroffene Vorhersage für verschiedene Verzweigungen im Programm nutzen zu können, existieren meist eine Reihe derartiger (*m*,*n*)-Korrelations-basierter Vorhersageautomaten pro Prozessor (typi-

scherweise 1024, jeder Verzweigung wird dann anhand der unteren 10 Adressbits ein Automat zugewiesen, und zu jedem Eintrag gehört ein *Branch-Target-Cache*-Eintrag). Zu dem Korrelations-basierten Vorhersageautomat existieren eine Reihe von ähnlichen Varianten [11].

4.5 Unterprogrammsprünge und Ausnahmebehandlung bei Pipelining

Die **Ausnahmebehandlung** (*Exception Handling*) sowie die Unterstützung bei externen **Unterbrechungen** (*Interrupt Requests*) müssen ebenso wie Unterprogrammaufrufe in das Pipelinesystem eingebunden werden. Beide Aktionen sind aber – der Natur der Sache entsprechend – in gewissem Sinn 'Pipeline-feindlich', daher bedarf es besonderer Aufmerksamkeit.

Das MPM2-Modell [3] als Beispiel für CISC-Architekturen zeigt eine erhebliche Busaktivität bei Aufrufen von Unterprogrammen (CALL-Befehl) sowie bei Eintritt in die **Interrupt Service Routine**. Der Grund hierfür liegt in der Sicherung der Rücksprungadresse und – bei Interrupt Request – des Statusregisters auf dem Stack. Entsprechend umgekehrte Funktionalität bei den Rücksprungbefehlen RTS bzw. RTI erzeugt auch hier zusätzliche lesende Aktivitäten am Bus.

Für MPM3, als Prototyp für einen Prozessor mit Phasenpipelining gedacht, müssen diese Bustransfers verhindert werden, um im System der Instruktionen zu verweilen. Hierzu dienen die Linkregister für Status und Adresse (R10/R11), in denen bei einer CALL-Instruktion sowie bei dem Aufruf einer Interrupt Service Routine die entsprechenden Register gespeichert werden. Beim Rücksprung werden die Prozessorregister (Status und Programmzähler) mit dem Inhalt dieser Register zurückgeschrieben. Diese Operationen benötigen keinen externen Bus.

Dieses Verfahren erscheint zunächst unvollständig, da ein zweiter Aufruf den Inhalt der Linkregister wieder überschreiben würde. Die Sicherung der Registerinhalte wird jedoch nur auf den Programmierer verschoben, denn er kann – falls eine weitere Funktion aufgerufen werden soll oder der Interrupt während der Interrupt-Service-Routine wieder freigegeben wird – mit Hilfe der PUSH-Instruktionen beliebige Register auf dem Stack zwecks Rekonstruktion retten. Hierbei muss genau darauf geachtet werden, dass keine Datenzerstörungen etwa durch auftretende IRQs auftreten können, dies ist durch eine sorgfältige Programmierung jedoch möglich.

Bei der Behandlung von Ausnahmesituationen – der Interrupt Request gehört in diese Kategorie – muss ferner ein jederzeit deterministischer Zustand des Prozessors gewährleistet sein. Dies ist für Architekturen ohne Pipelining kein Problem, da die Ausnahme erst nach vollständiger Beendigung eines Befehls durch eine besondere Instruktion in die Bearbeitung eintreten kann. Hierdurch ist ein deterministischer Zustand jederzeit gesichert.

Beim MPM3 muss in einer Pipelining-Version der Start der Ausnahmebehandlung in die Pipeline eingefügt werden, während sich noch andere Befehle in der Ausführung befinden. Der Interrupt-Request-Vektor, d.h. die Startadresse der zugehörigen Service Routine, wird aus Performancegründen ebenfalls in einem Register gehalten (R12). Er darf beispielsweise nicht unmittelbar vor Aufruf verändert werden, oder dieser Schreibvorgang muss bei der Bearbeitung des IRQ berücksichtigt werden (RAW-Hazard). Gleiches gilt für einen SEI-Befehl: Liegt dieser vor dem IRQ-Aufruf in der Pipeline, ist jedoch noch nicht in Bearbeitung, so muss der Aufruf der Service Routine ggf. nachträglich annulliert werden.

Um dies zu gewährleisten, wird beim MPM3 bei Auftreten eines Interrupt Requests – die Detektion einer solchen Anforderung ist auf die fallende Flanke am IRQ-Eingang des MPM3 sensitiv – und dem momentanen Vorliegen eines gelöschten Interrupt Disable Flags die Pipeline geleert. Hierzu werden 3 Pipeline Stalls eingefügt. Ist am Ende dieser 3 Wartezyklen das Interrupt Flag immer noch gelöscht, wird in die Service Routine gesprungen, ansonsten muss die Behandlung des Interrupts wieder von vorne beginnen (also einschließlich dreier Wartezyklen, wenn IF wieder gelöscht ist).

4.6 Beispielprogramme zur Bestimmung des CPI bei MPM3

Die hier aufgeführten Beispielprogramme für MPM3 dienen ausschließlich als Testprogramme zur Bestimmung der Performance.

INIT_ARR.ASM: Dieses Programm initialisiert ein eindimensionales Integer-Array mit 0. Der zulässige Wertebereich des Index ist 0 .. 255.

MOV_AVRG.ASM: Für die insgesamt 64 Elemente eines eindimensionales Integer-Array werden für jeweils 4 Elemente ein Mittelwert berechnet. Anschließend wird der Beginn der Mittelwertroutine um 1 Element verschoben, sodass insgesamt 61 Mittelwerte in einem zweiten Array zu speichern sind.

PARITY.ASM: Die gerade Parität aller Bitmuster (für 8 Bits) wird in diesem Programm bestimmt und zusammen mit dem originalen Byte als Bit 8 am Outputport ausgegeben.

WORDCOUN.ASM: Dieses Programm entspricht der inneren Routine von Wordcount, wie es als Tool für Unix-Betriebssysteme bekannt ist. Es zählt Character, Wörter und Zeilen in dem Text, der als Programmkonstante im Assemblercode angefügt ist.

Ziel dieses Programms ist es, die bedingt durch die stark zergliederte Programmstruktur häufige Ausführung von

Branchbefehlen einschließlich der Rückwirkungen auf das CPI zu bestimmen.

CRC-8.ASM: Die Checksummenberechnung, wie sie z.B. im Header der ATM-Zellen zu finden ist, wird in diesem Programm beschrieben. Hierin befinden sich drei Abschnitte:

Im ersten Abschnitt wird die Syndrom-Tabelle ausgefüllt. Diese Tabelle dient der schnellen Berechnung der CRC-8 Checksumme für 4-Byte Header (diese CRC wird als fünftes Byte angefügt).

Der zweite Teil besteht in der Generierung der Fehlertabelle. Die 256 Einträge hier lassen eine Bestimmung des zu korrigierenden Fehlers im Header (5 Byte, also einschließlich CRC-8) zu.

Der dritte Teil testet dann einen korrekten Zellheader, 40 davon abgeleitete mit korrigierbarem 1-bit-Fehler sowie 40 unkorrigierbare (gleichwohl detektierbare) Header.

SEL_SORT.ASM: Das bekannte Sortierverfahren Selection Sort wird in diesem Programm codiert und an zwei Arrays mit je 100 Elementen getestet.

QUICKSRT.ASM: Das bekannte Sortierverfahren Quicksort wird in diesem Programm codiert und an zwei Arrays mit je 100 Elementen getestet.

Die Tabellen 4.3/4.4 gibt einige Aussagen zur Performance. Hierbei ist zu beachten, dass die Werte für NoPipe (ohne Pipelining), Safe (Pipelining, Waitstates werden bereits bei der Möglichkeit einer Datenabhängigkeit entsprechend eingefügt) und Data Forwarding (Datenabhängigkeiten werden entsprechend aufgelöst) durch einen Instruktionssatzsimulator erhalten wurden, während die Version mit Branch Target Cache (BTC) zusätzlich auch noch eine Look-Ahead-Resolution beinhaltet und als VHDL-Modell simuliert wurde.

Deutlich sichtbar sind die Geschwindigkeitsgewinne in Richtung der theoretischen Performance von 1 Instruktion/Takt (IPC), insbesondere für das Harvard-Modell mit Multiporting und Branch Target Cache.

Programm	# Instr.	MPM3 NoPipe	MPM3 Safe	MPM3 Data F.	MPM3 BTC	
INIT_ARR	1030	4120 4.0 0.25	2310 2.24 0.45	1541 1.50 0.67	1287 1.25 0.80	# Clocks CPI IPC
MOV_AVRG	1114	4456 4.0 0.25	2470 2.22 0.45	1422 1.28 0.78	1363 1.22 0.82	# Clocks CPI IPC
PARITY	13317	53268 4.0 0.25	25861 1.94 0.51	17925 1.34 0.74	15365 1.15 0.87	# Clocks CPI IPC
WORDCOUN	23974	95896 4.0 0.25	49610 2.07 0.48	33953 1.42 0.71	26880 1.12 0.89	# Clocks CPI IPC
CRC-8 (1^{st} Tab)	17172	68688 4.0 0.25	33821 1.97 0.51	22809 1.32 0.75	20249 1.18 0.85	# Clocks CPI IPC
CRC-8 (2^{nd} Tab)	4141	16564 4.0 0.25	8542 2.06 0.48	5550 1.34 0.75	4999 1.21 0.83	# Clocks CPI IPC
CRC-8 (run)	16981	67924 4.0 0.25	32880 1.94 0.52	21881 1.29 0.78	20962 1.23 0.81	# Clocks CPI IPC
SEL_SORT	114379	457516 4.0 0.25	250139 2.19 0.46	155999 1.36 0.73	145673 1.27 0.79	# Clocks CPI IPC
QUICKSRT	24808	99232 4.0 0.25	56804 2.29 0.44	35819 1.44 0.69	33812 1.36 0.73	# Clocks CPI IPC
Mean (Sum)	216916	867664 4.0 0.25	462437 2.13 0.47	296899 1.37 0.73	270590 1.25 0.80	# Clocks CPI IPC

Tabelle 4.3: Simulierte Performancewerte für MPM3 (Singleport)

Programm	# Instr.	MPM3 NoPipe	MPM3 Safe	MPM3 Data F.	MPM3 BTC	
INIT_ARR	1030	4120 4.0 0.25	2054 2.00 0.50	1285 1.25 0.80	1032 1.00 1.00	# Clocks CPI IPC
MOV_AVRG	1114	4456 4.0 0.25	2223 2.00 0.50	1175 1.05 0.95	1116 1.00 1.00	# Clocks CPI IPC
PARITY	13317	53268 4.0 0.25	25605 1.92 0.52	17669 1.33 0.75	15108 1.13 0.88	# Clocks CPI IPC
WORDCOUN	23974	95896 4.0 0.25	47709 1.99 0.50	32052 1.33 0.75	24978 1.04 0.96	# Clocks CPI IPC
CRC-8 (1st Tab)	17172	68688 4.0 0.25	33562 1.95 0.51	22550 1.31 0.76	19991 1.16 0.86	# Clocks CPI IPC
CRC-8 (2nd Tab)	4141	16564 4.0 0.25	8115 1.96 0.51	5123 1.24 0.81	4605 1.11 0.90	# Clocks CPI IPC
CRC-8 (run)	16981	67924 4.0 0.25	30519 1.80 0.56	19520 1.15 0.87	18683 1.10 0.91	# Clocks CPI IPC
SEL_SORT	114379	457516 4.0 0.25	228727 2.00 0.50	134587 1.18 0.85	124262 1.09 0.92	# Clocks CPI IPC
QUICKSRT	24808	99232 4.0 0.25	49177 1.98 0.50	28591 1.15 0.87	27048 1.09 0.92	# Clocks CPI IPC
Mean (Sum)	216916	867664 4.0 0.25	427691 1.97 0.51	262552 1.21 0.83	236823 1.09 0.92	# Clocks CPI IPC

Tabelle 4.4: Simulierte Performancewerte für MPM3 (Multiport)

4.7 Wechselwirkungen zwischen Technologie und Architektur

Die ersten Jahrzehnte Einführung der Siliziumtechnologie und der Höchstintegration galten Technologie und Architektur der Bausteine als weitgehend unabhängig voneinander. Mittlerweile haben jedoch Effekte in den Bausteinen zu einer gegen-

seitigen **Wechselwirkungen zwischen Technologie und Architektur** geführt, die sich allerdings selten quantifizieren lassen. Für das Pipelining eines RISC-Prozessors (und darüber hinaus für superskalare Prozessoren) hingegen lässt sich ein Modell angeben, das für diesen Fall zu quantitativen Aussagen führt. Ideal wäre dabei eine Angabe, die Strukturbreite mit optimaler Anzahl der Pipelinestufen verbinden würde. Dies ist zurzeit unerreichbar, dennoch sind die Aussagen sehr interessant.

4.7.1 ”Analoges“ Modell für den Durchsatz

Nach [10] wird für eine CPU eine (konstante) Bearbeitungszeit *T* pro Instruktion angenommen, auch als Workload bezeichnet. Die Unterteilung in eine Anzahl von *S* Schritten (Phasenpipelining, → 3.2.1, 4.2) liefert in erster Näherung eine verringerte Netto-Bearbeitungszeit *T*/*S* pro Instruktion.

Richtet man also den Takt so ein, dass eine Taktzeit T_{clock} genau der Größe *T/S* entspricht, und setzt man ferner voraus, dass alle Pipelinestufen wirklich exakt gleichgewichtig sind, dann kann dieses Optimum theoretisch erreicht werden. Dies ergäbe für die Bearbeitungszeit ein Minimum bei möglichst hoher Anzahl von Pipelinestufen, allerdings wirken zwei Effekte dieser linearen Erhöhung entgegen.

Zunächst treten im Programmfluss bedingte Sprungbefehle mit unbekannter Sprungrichtung und Sprungziel auf, so dass der aktuelle Instruktionsfluss an dieser Stelle häufig nur angenommen werden kann. Dies führt zu Fehlvorhersagen mit resultierendem Zeitverlust.

Zur Quantifizierung dieses Effekts wird in diesem Modell angenommen, dass über alle Instruktionen gemittelt eine fehlerhafte Vorhersage des Programmflusses mit einer Wahrscheinlichkeit *b* vorliegt und dies bei jedem Fall zu einem Verlust von $S - k$ ($k = 1, 2, ...S$) Takten führt. Hierdurch vergrößert sich die durchschnittliche Anzahl der Takte pro Instruktion (CPI, Clocks per Instruction) von theoretisch 1 auf

$$CPI = 1 + (S - k) \cdot b \tag{4.1}$$

Der zweite Effekt blieb bislang unbeachtet und tritt durch die innere Struktur eines Prozessors mit Pipelining auf. Die einzelnen Stufen werden durch Registersätze (Flipflops) voneinander getrennt (→ 4.4). Diese Register speichern die Daten zwischen den Pipeliningstufen, und hieraus resultiert ein zusätzlicher Zeitbedarf *C* (Setup- und Hold-Time der Register) pro Stufe. Quantitativ wird damit die effektive Zeit pro Pipeliningstufe durch

$$T_{Pipestage} = \frac{T}{S} + C \tag{4.2}$$

beschrieben; theoretisch werden zwar nur (*S*-1)mal die Zwischenwerte zwischen zwei Pipelinestufen gespeichert, praktisch kann aber die letzte Stufe auch nicht schneller arbeiten, da der Takt starr gekoppelt ist. C/T ist damit der Anteil der

Speicherzeit (für eine Stufe) am ursprünglichen (Gesamt-) Workload. Hieraus folgt mit Gleichung (4.1) die gesamte Zeit pro Instruktion (TPI) zu

$$TPI = \left(1 + (S - k) \cdot b\right) \cdot \left(\frac{T}{S} + C\right) \tag{4.3}$$

bestimmt ist. Die Invertierung hiervon, also die Instruktionsrate pro Zeiteinheit, wird als Pipelinedurchsatz *G* bezeichnet:

$$G = \frac{1}{TPI} = \frac{1}{T} \cdot \frac{1}{1 + (S - k) \cdot b} \cdot \frac{1}{1/S + C/T} \tag{4.4}$$

In diesem Modell sind nunmehr die Kontrollflusshazards durch *b* wie die (Silizium-) Technologie durch *C*/*T* sowie *T* selbst berücksichtigt. Leitet man *G* nach *S* ab, so erhält man

$$\frac{dG}{dS} = \frac{(1 - b \cdot k) - b \cdot S^2 \cdot C/T}{T \cdot (1 + (S - k) \cdot b)^2 \cdot (1 + S \cdot C/T)^2} \tag{4.5}$$

Für die optimale Anzahl der Pipeliningstufen S_{opt} folgt heraus

$$S_{opt} = \sqrt{\frac{(1 - b \cdot k)}{b \cdot C/T}} \tag{4.6}$$

Gleichung 4.6 zeigt, dass die Speicherzeit zwischen zwei Pipelinestufen in jedem Fall berücksichtigt werden muss: *C* = 0 bedeutet, dass kein Speicher vorhanden ist oder berücksichtigt werden muss. In diesem Fall wäre die optimale Anzahl der Pipelinestufen nach ∞ strebend, was der Beobachtung widerspricht.

Um die Abhängigkeiten des Pipelinedurchsatzes von den wesentlichen Parametern b und C/T darstellen zu können, wird einer dieser Parameter konstant gehalten und der andere als Parameter der entstehenden Kurvenschar genutzt.

Zunächst werden nur die Wartezyklen, die aus den Kontrollflusshazards resultieren, berücksichtigt (Bild 4.18). Hieraus ergeben sich Kurven, die für große Pipelinestufenzahlen ihrem Grenzwert entgegenstreben und somit kein Maximum zeigen.

Weiterhin fällt auf, dass die Variation von b große Durchsatzsprünge ergibt. Bei einem angenommenen Verhältnis von 1:5 zwischen Verzweigungsbefehlen und der gesamten Anzahl der Instruktionen bedeutet b = 0,2, dass die Ausführung der Verzweigungsbefehle nicht vorhergesagt wird, demnach also immer entsprechend gewartet wird (Vorhersagequalität 0 %). Dies verändert sich für b = 0,1 auf 50 % Vorhersagequalität, und für b = 0,01 sind es bereits sehr gute 95 %. Zwischen b = 0,2 und b = 0,01 variiert der Durchsatz bei ansonsten gleichen Werten immerhin fast um den Faktor 4 (für S = 20).

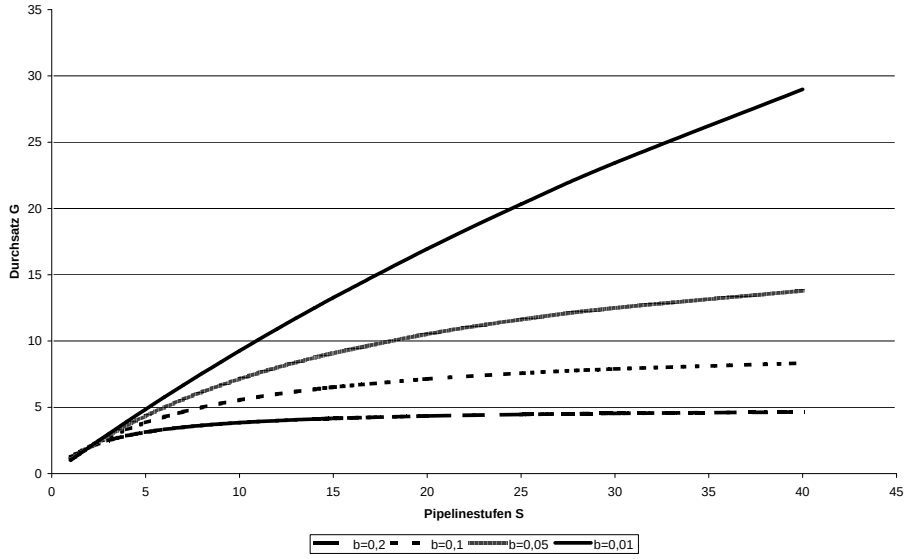


Bild 4.18 Einfache Variante des Pipeline-Modells (C/T = 0): Nur die Kontrollflusshazards führen zu einer Verschlechterung des Durchsatzes

Bild 4.19 zeigt die Abhängigkeit des Pipelinedurchsatzes für einen RISC-Prozessor gemäß dem Modell (4.4) bzw. (4.6). Hier wurden $k = 1$ und $b = 0{,}1$ als feste Parameter gewählt, während C/T über die verschiedenen Kurven variiert. Der Darstellung dieses Modells liegt die Annahme zugrunde, dass die Gesamtberechnungszeiten T in erster Näherung konstant bleiben (abgesehen von Änderungen in der Siliziumtechnologie, die hier unberücksichtigt bleiben), während sich C und damit C/T verringert.

Die Werte für C/T sind dabei durchaus realistisch. Für den Cell-Prozessor von IBM wird bei 23 Pipelinestufen ein so genanntes 11FO4-Design (→ 4.7.2) gewählt, bei dem 8 Gatterlaufzeiten ”Rechnen“ mit 3 Gatterlaufzeiten „Speichern“ pro Pipelinestufe durchzuführen sind. Dies ergibt einen rechnerischen Wert von $C/T = 3/(23*11) = 0{,}012$.

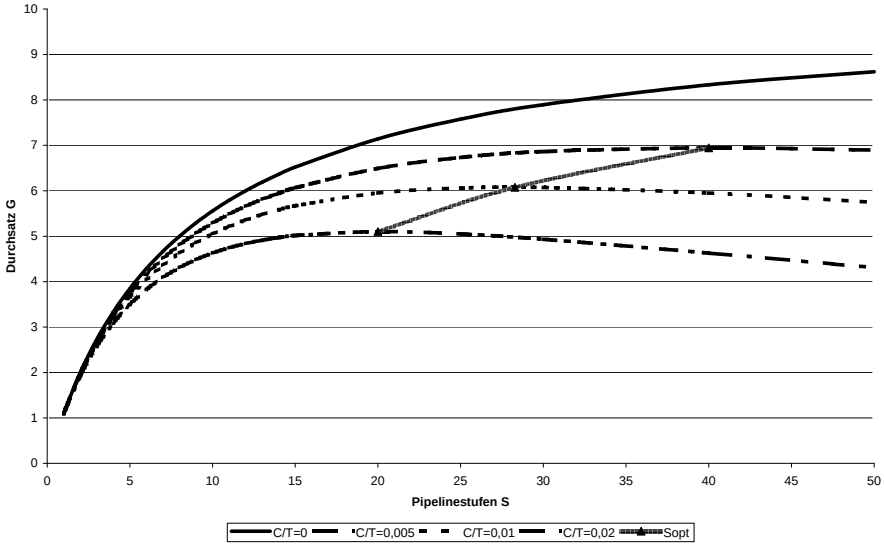


Bild 4.19: Pipelinedurchsatz *G*(*S*) eines RISC-Prozessors mit *C*/*T* als Parameter

In Bild 4.19 fällt auf, dass die optimale Stufenanzahl für die gewählten C/T-Werte erst bei sehr hohen Pipelinestufenanzahlen erreicht werden, Werte, die nicht bzw. nicht mehr gewählt werden. Der genaue Grund hierfür muss noch erläutert werden (→ 4.7.2).

In Bild 4.20 werden nun zwei Versionen eines Prozessors miteinander verglichen. Für den Cell-Prozessor ist bekannt, dass bei einer 23stufigen Pipeline die einzelne Pipelinestufe im 11FO4-Design implementiert ist, andere Pipelines jedoch im 22FO4-Design (z.B. Floating Point) implementiert sind. In diesem Bild werden jetzt die 11FO4- mit der 22FO4-Variante gegenüber gestellt, allerdings mit unterschiedlichen Güten in der Sprungvorhersage (b = 0,1 bzw. 0,01).

Die Grafik zeigt, dass die beiden Designs auf etwa den gleichen Durchsatz kommen – eine sehr interessante Aussage. Berücksichtigt man nun, dass im 11FO4-Design die Sprungvorhersage sehr einfach (statisch) ist, so ist hier eine Menge ”Luft“ noch enthalten, d.h., die Performance ließe sich noch sehr gut erhöhen.

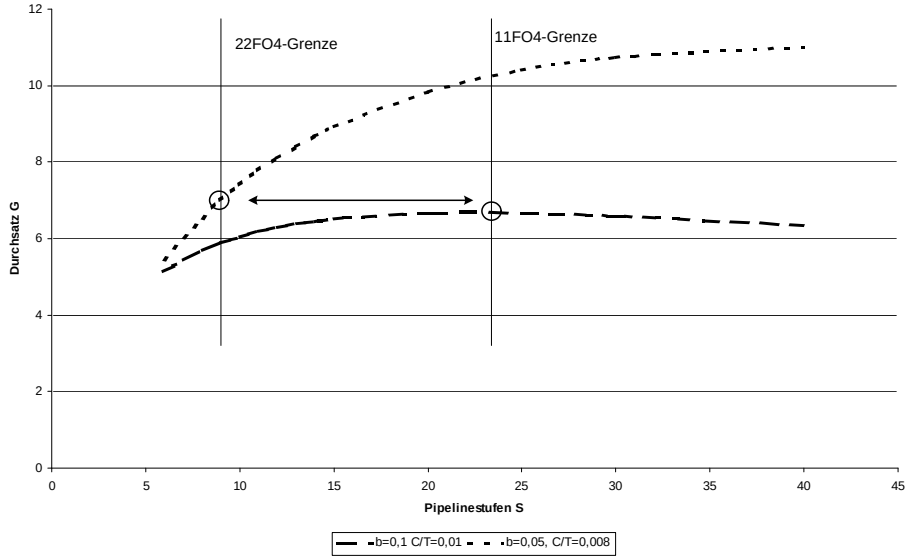


Bild 4.20: Vergleich zweier Implementierungen für den Cell-Prozessor: Hohe Pipelinestufenanzahl mit schlechter Verzweigungsvorhersage versus Geringere Pipelinestufenanzahl mit guter Verzweigungsvorhersage

4.7.2 Die Körnigkeit des Rechnens

Es klang im letzten Abschnitt schon an, dass es noch eine weitere Grenze des Pipelinings geben muss. Diese besteht in dem Aufbau des Prozessors auf Transistorebene, hier werden Gatter in CMOS-Technologie verwendet (siehe Bild 4.21).

Jede Operation in der Pipeline muss in eine Verschaltung von CMOS-Transistorpaaren umgesetzt werden, und zwar in eine für jede Stufe möglichst gleichlange. Dabei hat sich herausgestellt, dass die Machbarkeitsgrenze bei ”11FO4“ liegt [27], dies bedeutet dass in Serie maximal 11 NAND-Gatter mit einem Fanout von 4 verschaltet sind. Hiervon werden 2–3 Gatter für die Speicherung zwischen zwei Pipelinestufen benötigt [26].

Was bedeuten diese Zahlen? Der Fanout von 4 heißt, dass die Transistoren im Aufbau des Gatters aufgrund ihrer Größe in der Lage sind, 4 nachfolgende Eingänge mit genügend Strom zu versorgen, um für ein (zügiges) Umschalten zu sorgen (Anmerkung: Der Aufbau in Bild 4.21 besitzt 4 Eingänge, und diese Zahl steht nicht in direktem Zusammenhang mit dem FO4-Design). Damit ist die Anzahl der nachfolgenden Eingänge natürlich begrenzt.

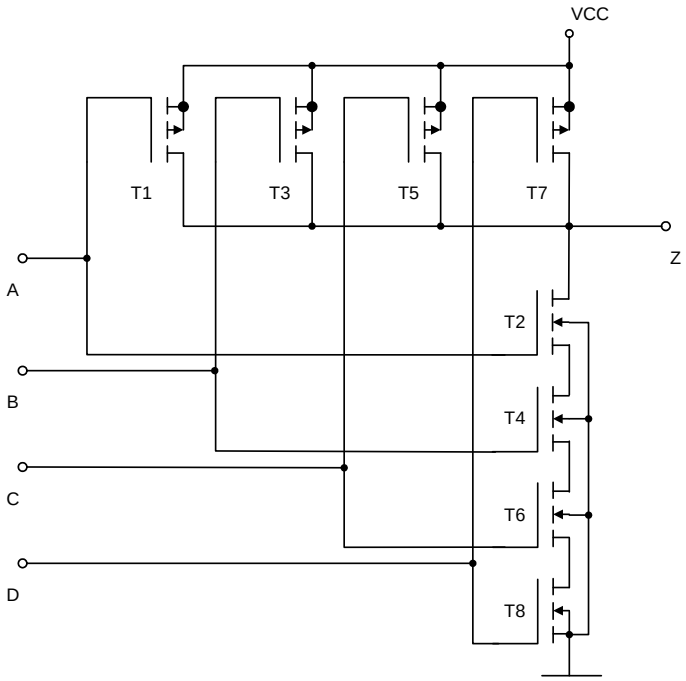


Bild 4.21 NAND-Gatter mit 4 Eingängen in CMOS-Ausführung

Indirekt begrenzt die Treiberfähigkeit damit die Ausdehnung der Pipelinestufe, denn allzu viele Gatter können einem Ausgang ja nicht folgen. Eine Ausdehnung einer Pipelinestufe in vertikaler Richtung (bei gedachter horizontaler ”Rechenrichtung“) wird außerdem durch die dann verlängerten Signalübertragungszeiten eingeschränkt.

Somit hat sich in der Praxis der Wert ”11FO4“ als Grenze erwiesen, und auch das ist schon eine Meisterleistung. Diese Grenze ist in Form einer maximalen Pipelinestufenzahl in Bild 4.20 eingezeichnet (Cell-Prozessor: 23 [26]), darüber wären weitere Pipelinestufen kontraproduktiv.

Was bleibt dann zu tun? Bild 4.20 zeigt einen Weg: Im Cell-Design wurde bei schlechter Branchvorhersage das Pipelinedesign so weitgehend wie möglich durchgeführt. Verbessert man nun die Verzweigungsvorhersage, dann kann der Durchsatz durchaus noch um den Faktor 2 erhöht werden. Eine Grenze wird aber dennoch irgendwann erreicht, dann bleibt nur noch Superskalarität ($\rightarrow$ 5), Simultaneous Multithreading (SMT, $\rightarrow$ 8) oder Chip-Integrated Multithreading (CMP, $\rightarrow$ 8).

5 Der Aufbau superskalarer Architekturen

Die Befehlssequenzialität, die für von-Neumann-ähnliche Pipeline-Architekturen als letzte Forderung übrigbleibt, scheint die maximale Performance auf 1 Instruktion/Takt festzulegen. Innerhalb der RISC-CPU war eine Beschleunigung auf dieses Maß ein großer Erfolg und nur über zusätzliche Maßnahmen wie eine Forwarding Unit erreichbar.

Nunmehr wird auf der Ebene der Single-CPU noch mehr verlangt: Die 'Schallgrenze' von 1 Instruktion/Takt soll durchbrochen werden. Dies erweist sich als mehr als eine pure Erweiterung der RISC-Philosophie, da die verschiedensten Maßnahmen erst in der Zusammenfassung dieses Ziel ergeben. Hierzu gehören die folgenden Teile:

- Der Prozessor muss in der Lage sein, mehrere Instruktionen pro Takt zu laden und zu decodieren.
- Branch-Instruktionen, also bedingte Sprünge dürfen möglichst nicht zu Behinderungen des Instruktionsflusses führen.
- Datenabhängigkeiten, die wie im Fall der RISC-CPU behandelt werden müssen, treten in der superskalaren CPU in erhöhtem Maß auf und müssen entweder vermieden werden oder in ihren Auswirkungen beherrschbar sein. Dies bedeutet u.a., dass die Ergebnisse nach dem Takt durchaus noch sortiert werden müssen, um die Berechnung korrekt durchzuführen!
- Alle Maßnahmen zusammen führen dann zu einem echten Parallelismus auf Instruktionsebene, aus diesem Grund auch *Instruction Level Parallelism* (ILP) genannt.

Diese Maßnahmen werden im Folgenden beschrieben.

5.1 Die Beschreibung des Ziels

Die Entwicklung von superskalaren Mikroprozessoren erfolgte nicht losgelöst von den bisherigen Prozessoren. Es sind zumeist (mit Ausnahmen) Prozessoren, die das obere Ende der Rechenleistung innerhalb einer Prozessorfamilie darstellen, wobei die Prozessorfamilie durch eine für alle Mitglieder kompatible *binäre Maschinensprache* oder *Maschinencode* definiert ist. Diese Maschinensprache soll auch für einen superskalaren Prozessor unverändert gelten, sodass ausgesagt werden muss, was sie eigentlich definiert:

Während in sequenziellen Ausführungsmodellen (bis hin zum RISC-Prozessor) die Maschinensprache durchaus definierte, wie ein Befehl im Prozessor ausgeführt wurde, muss dies zu der Aussage für superskalare Rechner abgeschwächt werden, wo die binäre Sprache definiert, was ausgeführt wird. Der geringe Unterschied ist

durchaus wichtig, da bei paralleler Ausführung kein exakter Determinismus der Ausführungsreihenfolge mehr möglich ist.

Es bleibt jedoch die Aussage, dass das *Konzept des präzisen Zustands* innerhalb der CPU erhalten bleiben muss. Dieses Konzept besagt, dass bei Unterbrechungen etwa durch Interrupts oder Ausnahmen (Exceptions) der Zustand der CPU sicherbar und wiederherstellbar sein muss, falls diese die Ausnahmen durch Sonderprogramme behandelt. Dieses Konzept ist eng mit dem Modell der sequenziellen Ausführung verknüpft und bedeutet sowohl für RISC-CPUs wie für superskalare Prozessoren entsprechende Hardware, um dies zu gewährleisten.

Das Ziel der superskalaren Rechnerarchitektur ist es nun, durch parallele Ausführung von Instruktionen die durchschnittliche Bearbeitungszeit pro Instruktion zu senken, wobei die Instruktionslatenzzeit, d.h. die Zeit, die maximal pro Befehl verstreicht, gegenüber der RISC-CPU nicht erhöht wird.

Dies beinhaltet folgende Einzelheiten für das Design einer superskalaren CPU:

- Strategien zum Fetch mehrerer Instruktionen pro Takt, wobei insbesondere der Einfluss von bedingten Sprungbefehlen berücksichtigt werden muss.
- Methoden zur Detektierung von virtuellen und wirklichen Abhängigkeiten zwischen Registerinhalten einschließlich der Mechanismen zur Vermeidung der Folgen.
- Methoden zur Initiierung der Ausführung mehrerer Instruktionen parallel zueinander.
- Ressourcen zur parallelen Ausführung mehrerer Instruktionen; hierzu gehören insbesondere mehrere Ausführungseinheiten sowie Speicherhierarchien, die die Datenflüsse aufnehmen können.
- Ausführung mehrerer Load/Store Pipelines in enger Abstimmung mit den Ausführungsstrategien für Instruktionen. Die Load/Store Pipelines müssen insbesondere eine Entkopplung zwischen dem Speicher (mit i.a. unvorhersagbarer Zugriffszeit) und den Registern schaffen.
- Methoden zur Bestimmung der korrekten Ausführungsreihenfolge und insbesondere Speicherreihenfolge in den Registern. Diese Funktionseinheit ist besonders wichtig, da die sequentielle Reihenfolge der Befehlsbeschreibung in eine Ergebnisreihenfolge mündet, und genau diese Sequenz muss beibehalten werden.

Zusammenfassend kann man sagen, dass ein superskalarer Prozessor zwar die *Instruktionssequenzialität* nicht mehr einhalten kann, jedoch immer noch die *Ergebnissequenzialität* bietet (und bieten muss).

5.2 Programmdarstellung, Abhängigkeiten und parallele Ausführung am Beispiel eines Programms

Der Ablauf der Programmierung eines Rechners kann typischerweise durch die Erstellung des Sourcecodes in einer Hochsprache, die Übersetzung in Assembler bzw. Binärcode und die weiteren Vorgänge unterteilt werden. Der Linkerlauf ist in diesem Zusammenhang von keinem Interesse, da im Wesentlichen nur Adressen gesetzt werden. Bild 5.1 zeigt eine kurze C-Routine, aus einem Sortierprogramm stammend, sowie die Übersetzung in einen nicht optimierten Assemblercode.

```
for( i = 0; i < last; i++ )
{
  if( a[i] > a[i+1] )
  {
    temp = a[i];
    a[i] = a[i+1];
    a[i+1] = temp;
    change++;
  }
}
```

(a)

```
L2: move    r3, r7      ; &a[i] -> r3
    load    r8, (r3)    ; a[i] -> r8
    add     r3, r3, 4   ; &a[i+1]
    load    r9, (r3)    ; a[i+1] -> r9
    ble     r8, r9, L3  ; branch a[i] ≤ a[i+1]

    move    r3, r7      ; &a[i] -> r3
    store   r9, (r3)    ; a[i] neu
    add     r3, r3, 4   ; &a[i+1]
    store   r8, (r3)    ; a[i+1] neu
    add     r5, r5, 1   ; change++

L3: add     r6, r6, 1   ; i++
    add     r7, r7, 4   ; &a[i] -> r7
    blt     r6, r4, L2  ; branch i < last
```

(b)

Bild 5.1: Ein C-Programm (zur Sortierung) und die nicht-optimierte Assemblerübersetzung

Diese Übersetzung zeigt sehr deutlich die Kontrollflussabhängigkeit dieser Architekturen: Jeder Befehl wird erreicht, indem die vorhergehenden Befehle durchlaufen werden (*Program Counter Increment*) oder indem ein Sprungbefehl auf die Codespeicherstelle zeigt (*PC Update*). Beide Kontrollflussabhängigkeiten müssen bei der parallelen Ausführung von Instruktionen berücksichtigt werden.

Zunächst ist das Assemblerprogramm in 5.1 (b) in 3 *Basisblöcke* aufgeteilt:

Definition 5.1:

Ein Basisblock ist die maximale Anzahl von Instruktionen (um eine gewählte Stelle), die exakt einen Eintritts- und einen Austrittspunkt besitzt.

Ein Basisblock entspricht damit der um den gewählten Punkt maximale Folge von Instruktionen ohne Verzweigungsbefehl (Ausnahme: letzter Befehl) und ohne Sprungmarke (Ausnahme: am ersten Befehl). Dieser Block wird immer vollständig

durchlaufen, falls das Programm den Anfang erreicht. Somit kann ein Basisblock prinzipiell in paralleler Weise ausgeführt werden und ist damit die Größe im Befehlsfluss, auf die sich Compiler und superskalare Architekturen konzentrieren.

Weitere Parallelität kann durch Zusammenfassung mehrerer Basisblöcke etwa durch spekulative Ausführung von Branchbefehlen erhalten werden. Trifft die Ausführungsannahme nicht zu, müssen alle Ergebnisse sowie die Instruktionspipeline entsprechend korrigiert werden. Hierdurch wird deutlich, dass zwischen superskalarer Rechnerarchitektur und Compilerbau (insbesondere Codegenerator und Optimierer) eine erhebliche Wechselwirkung besteht. Diese Abhängigkeiten werden in Kapitel 6 näher erläutert.

Innerhalb eines Basisblocks existieren jedoch auch Datenabhängigkeiten, die wie im Fall der RISC-CPU weiter zu untersuchen sind. Diese Datenabhängigkeiten bestehen zwischen verschiedenen Instruktionen, falls diese auf die gleichen Speicherstellen bzw. Register lesend oder schreibend zugreifen.

Im Gegensatz zu den RISC-CPUs, die lediglich den Read-After-Write- (RAW) Hazard kennen(→ 4.4.1), können aufgrund der parallelen Ausführung von Instruktionen auch Write-After-Read- (WAR) sowie Write-After-Write- (WAW) Hazards auftreten. Diese Abhängigkeiten sind in folgender Abbildung, die sich auf den ersten Basisblock aus Bild 5.1 bezieht, dargestellt:

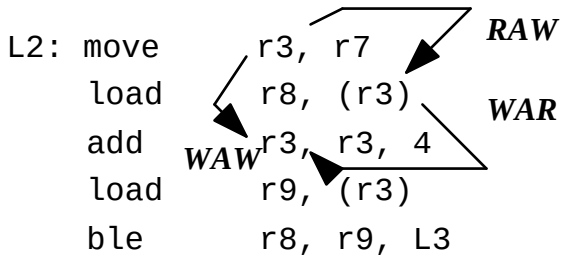


Bild 5.2: Beispiele für Datenabhängigkeiten

WAW-Hazards müssen berücksichtigt werden, da die Reihenfolge des schreibenden Zugangs zu einem Register nicht mehr durch die Sequenz der Befehle, die ja nun parallel zueinander ausgeführt werden, gegeben ist. Der korrekte, letzte Schreibzugriff muss detektierbar sein, sodass entweder eine Zerstückelung der parallelen Ausführung erfolgt oder durch spezielle Zusätze eine Entscheidung zum korrekten Wert nachträglich erfolgen kann. Letzteres impliziert zusätzliche Register zur Aufnahme aller Zwischenwerte sowie eine Entscheidungseinheit, während die erste Lösung letztlich unerwünscht ist.

WAW-Hazards können mehrfach erzeugt werden; Gründe hierfür sind:

- Nicht-optimierter Code
- Begrenzte Registeranzahlen

- Schleifen, die mehrfach zueinander parallel ausgeführt werden

Die andere Sorte der neuen Abhängigkeiten, die WAR-Hazards, treten auf, da unbedingt gewährleistet werden muss, dass zur Berechnung die gültigen, in diesem Fall älteren Werte in einem Register (oder Speicherstelle) genommen werden. Die parallele Ausführung gewährleistet dies keineswegs mehr, da eine Zugriffsreihenfolge nicht mehr bestimmbar ist. WAR und WAW Abhängigkeiten werden als virtuell oder künstlich bezeichnet, da sie durch geeignete Maßnahmen wirkungslos gemacht werden können. RAW hingegen bleibt eine reale, unauflösbare Datenabhängigkeit.

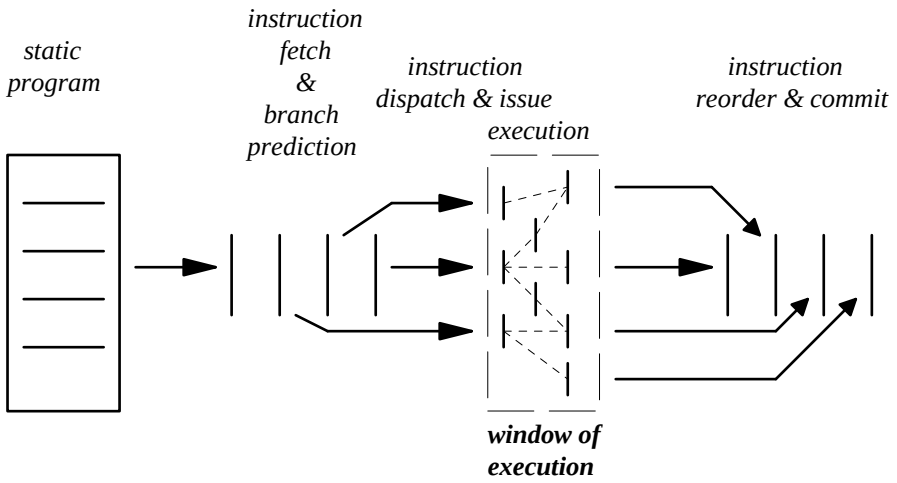


Bild 5.3: Konzept des Programmablaufs innerhalb einer superskalaren CPU

Der Ablauf eines Programms ist in Bild 5.3 dargestellt. Das statische Programm wird durch die Instruction Fetches einschließlich einer Branchvorhersage (*Branch Prediction*) geladen, dann im Ausführungsfenster mit weitestgehender Parallelität ausgeführt, und am Schluss werden die Ergebnisse in der gültigen Reihenfolge sortiert. Insbesondere die spekulative Ausführung kann dabei Probleme erzeugen, da ggf. Befehle ausgeführt wurden, die dem Programmfluss entsprechend nicht zur Ausführung gekommen wären. Es ist eine Aufgabe der Commit-Einheit, diese Ergebnisse dann fortzulassen.

5.3 Die Mikroarchitektur einer typischen superskalaren CPU

Die Mikroarchitektur einer superskalaren CPU ist in ihrem typischen Aufbau in Bild 5.4 dargestellt. Bei dieser Darstellung wurde bewusst auf den üblicherweise integrierten Teil für Floating Point Zahlen verzichtet.

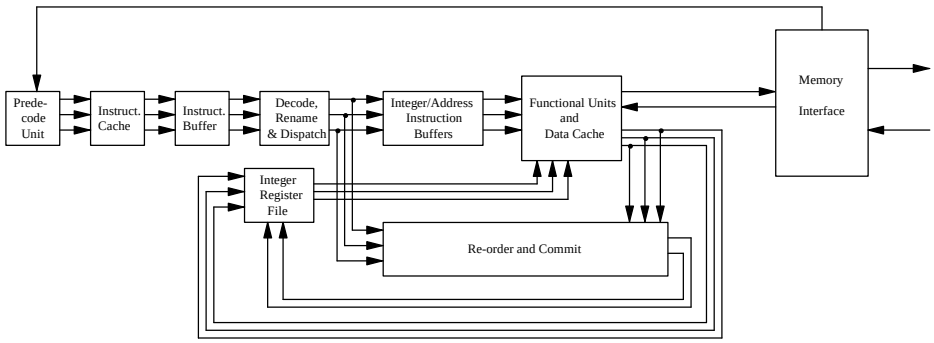


Bild 5.4: Typische Organisation eines superskalaren Prozessors

Der Organisation des superskalaren Prozessors unterliegt weiterhin eine Pipelinestruktur entsprechend den Darstellungen zur RISC-CPU. Die Pipeline in den superskalaren Rechnern ist hierbei i.a. mit einer größeren Anzahl von Stufen versehen, wie bereits aus obiger Zeichnung hervorgeht. Die einzelnen Funktionsblöcke besitzen folgende Aufgaben:

5.3.1 Instruction Fetch und Predecode

Die Fetch Phase des Prozessors umfasst mehrere Teilaktionen, an denen die Blöcke *Predecode*, *Instruction Cache* und *Instruction Buffer* beteiligt sind. Der Instruction Cache entspricht im wesentlichen der gewohnten Funktionalität. Die maximal erreichbare Performance eines superskalaren Rechners macht es unabdingbar, einen Instruktionscache zu nutzen. Die minimale Fetchrate muss mindestens gleich der maximalen Performance (in Instruktionen pro Takt) sein, Reserven sind beispielsweise für Cache Misses notwendig. Sollte sich innerhalb einer Cache Line eine Branch-Instruktion befinden, so können je nach Strategie Zugriffe an anderen Stellen notwendig werden.

Die vorgeschaltete Predecode Unit wird zur Erzeugung von Steuerinformationen genutzt. Diese ermöglichen unter Umständen schnellere Auswertung in nachgeschalteten Stufen und werden innerhalb des Cache additiv gespeichert. Der Instruktionsbuffer kann ggf. entfallen, dient jedoch dazu, einen einheitlichen, von Cache Misses unabhängigen Pufferspeicher zum inneren Teil der CPU zu haben.

Dieser Instruktionsbuffer enthält die Instruktionen, deren Fetchphase nunmehr abgeschlossen ist.

Exakt betrachtet stellt die Phase des Instruction Fetch einen Instruction Block Fetch dar, da mehrere Instruktionen anschließend zur parallelen Ausführung kommen sollen. Die Strategie zum Laden eines kompletten Blocks erstreckt sich mindestens auf einen Basisblock, der von keinen Verzweigungsbefehlen unterbrochen wird, sodass der Program Counter bei jeder Instruktion inkrementiert wird. Unbedingte Sprünge können ebenso behandelt werden, lediglich kann dies eher zu Cache Misses führen.

Konditionierte Branches müssen gesondert behandelt werden, da sie zu Verzögerungen in der Bearbeitung führen können. Die Gesamtperformance eines superskalaren Rechners hängt sehr entscheidend von dem Aufwand zur bestmöglichen Auswertung dieser Befehle ab. Folgende Teile gehören zur Behandlung der Branches (→ 4.4.3):

1. Erkennung eines konditionierten Sprungbefehls
2. Bestimmung der Sprungrichtung (Vorhersage!)
3. Berechnung des Sprungziels
4. Weitere Ausführung des Instruction Fetch, ggf. mit neuem Program Counter.

Die *Erkennung eines konditionierten Sprungbefehls* wird in erheblichem Maß durch die Codierung innerhalb der Maschinensprache oder durch eine Predecode Unit und zwischengespeicherte Informationsbits erleichtert. Letztere Einheit ist besonders dann nützlich, falls der Maschinencode aus Kompatibilitätsgründen nicht ‘selbstdecodierend’ ausgeführt ist.

Die *Bestimmung der Sprungrichtung* ist meist nicht unmittelbar möglich, da dies erst durch die Berechnung im zumeist vorhergehenden Instruktionsblock erfolgt, die Ergebnisse zum Zeitpunkt des Fetch aber nicht bekannt sind. Aus diesem Grund wird eine Sprungrichtungsvorhersage (Sprung oder kein Sprung) durch diverse Verfahren eingeführt. Das einfachste Verfahren besteht in der *statischen Vorhersage* durch Compilerinformationen (codiert) oder Erfahrungswerte (z.B. Rücksprünge werden immer genommen, da sie in Loops verwendet werden).

Die andere grundsätzliche Methode verwendet dynamische Informationen, die während des Programmlaufs entstehen und in *Branch Prediction Tables* gehalten werden. Eine typische Implementierung besteht in Zählern (saturierend, ohne Überlauf!), die die genommene Richtung aufwärts/abwärts zählen und eine Sprungrichtung je nach Zählerstand vorhersagen. Simulationen bestätigen diese Ansätze selbst bei geringen Zählertiefen (z.B. 2 Bit). Allerdings müssen mehr als ein Zähler genutzt werden, um die einzelnen Verzweigungen voneinander zu unterscheiden (typisch: 1024 Zähler á 2 Bit).

Solche Zählerarrays werden in High-Performance-Mikroprozessoren zu Korrelations-basierten Vorhersageautomaten (→ 4.4.3.3) zusammengefasst, um auch die

Vorgeschichte von Verzweigungen und Abhängigkeiten zu anderen Programmteilen erfassen zu können.

Bei einer falschen Vorhersage müssen – wie bereits erwähnt – alle nach der Verzweigung ausgeführten Instruktionen rückgängig gemacht werden, da der Instruktionsteilblock neu geladen und ausgeführt wird.

Die *Berechnung des Sprungziels* erfolgt in dem gewohnten Umfang. Branchbefehle enthalten die Sprungzielangabe zumeist als PC-relative Information, sodass eine einfache Integeraddition wie im Fall der RISC-CPU das neue Sprungziel liefert.

Die *weitere Ausführung der Instruktionen* ist im Fall eines Sprungs durchaus mit weiteren Überlegungen verbunden. Im Fall der RISC-CPU führte dies zu einer Verzögerung, da die Phase des Instruction Fetch für die nächste Instruktion bereits gelaufen war und das Ergebnis ungültig wurde. Für superskalare Prozessoren bietet sich hier ein anderes Verfahren an, bei dem die Entkopplung zwischen Ausführung und Fetch von Instruktionen stärker in Betracht kommt. Gewöhnlicherweise wird der Update des Program Counters durch die Zwischenspeicherung im Instruction Buffer aufgefangen; alternative Techniken verfolgen sogar beide Kontrollflusspfade, um zumindest bei einfachen Verzweigungen einen garantiert richtigen Pfad auszuführen.

5.3.2 Instruction Decoding, Renaming und Dispatch

Während der nächsten Phase lädt der Prozessor die Instruktionen aus dem Instruction Buffer, decodiert sie, prüft auf Datenabhängigkeiten und ordnet sie den zur Verfügung stehenden Hardwareeinheiten zu. Dies beinhaltet demnach mehrere Teilphasen, die im einzelnen erklärt werden.

Während der *Decode Phase* werden ein oder mehrere Tupel von Erkennungsbits pro Instruktion erzeugt. Diese Erkennungstupel beinhalten eine geordnete Liste von

1. der auszuführenden Operation,
2. den Kennungen der Speicherelemente für den Input zu diesen Operationen und
3. der Kennung des Speicherelements für die Ausgabe der Operation.

In dem statischen, vom Compiler erzeugten Programm bestehen die Speicherelemente aus den im Programmiermodell sichtbaren Teilen, also den *logischen Registern* oder dem *Hauptspeicher*. Zur Beseitigung der virtuellen Hazards WAR and WAW, die zu erhöhter Parallelität in der Ausführung führt, entsprechen insbesondere die logischen Register oftmals davon abweichenden physikalischen Speicherstellen. Während in einer streng sequentiellen Ausführung diese Speicherstellen zu verschiedenen Zeitpunkten beschrieben bzw. gelesen werden, wird diese Zeitsequenz nunmehr auf verschiedene Orte abgebildet.

Bild 5.5 zeigt das Verhalten in dieser Phase, die neben dem Decode auch ein *Rename* enthält. Durch die Zuordnung von Tupeln zu den Instruktionen wird das

Umbenennen stark erleichtert, wobei natürlich die nachfolgenden Anweisungen, die dieses Register als Input nutzen, ebenfalls umgenannt werden müssen.

Das in Bild 5.5 gezeigte Renaming nutzt einen Pool von physikalischen Registern, deren Anzahl die der logischen übersteigt. Anhand einer Mapping Tabelle kann dann jedem logischen Register ein physikalisches aus einer Freiliste zugeordnet werden, sodass die Freiliste verkleinert wird. Das Verfahren ordnet in dem Beispiel r3 (bisher R1) auf R2 zu, wobei R1 weiterhin belegt ist (nämlich mit dem Lesewert für die Addition). R1 wird nach der Operation frei und kann dann genutzt werden. Hierdurch ist die vergrößerte Anzahl von physikalischen Registern erklärlich. Das Dispatch von Instruktionen, also die Zuordnung zu physikalischen Einheiten, muss zeitweise gestoppt werden, falls keine physikalischen Register mehr frei sind, aber benötigt werden.

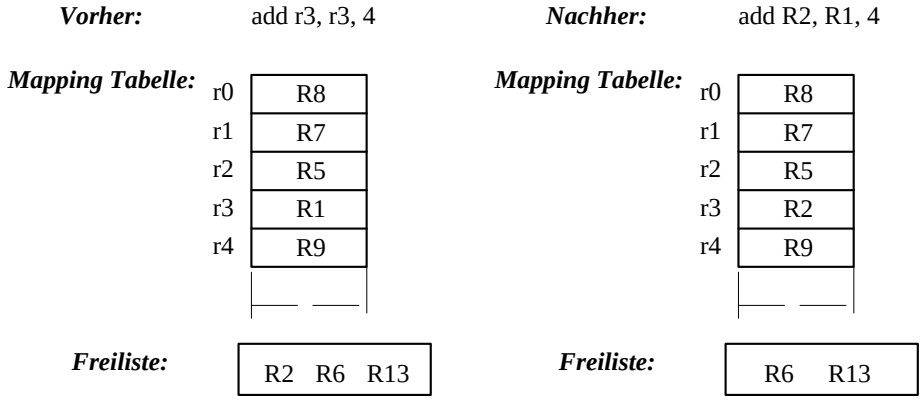


Bild 5.5: Renaming Phase: Die logischen Register (rx) werden physikalischen (Ry) zugeordnet

Die Freigabe von physikalischen Registern erfolgt durch den letzten Lesevorgang: Nach diesem Zugriff wird es nicht mehr benötigt und kann der Freiliste wieder zugeführt werden, was allerdings durchaus einen komplizierten Verwaltungsvorgang bedeuten kann: Insbesondere die Feststellung des letzten Lesevorgangs ist durchaus komplex. Eine Methode besteht in der Installation eines Zählers mit jedem physikalischen Register, der mit jedem Source-Mapping dieses Registers inkrementiert und jedem Lesevorgang dekrementiert wird, sodass bei Erreichen der 0 das Register wieder frei ist, falls das zugehörige logische Register in der Zwischenzeit umbenannt wurde. Einfacher (jedoch ggf. mit längerer Zuordnung) ist jedoch die Lösung, auf die Umbenennung durch eine spätere Instruktion, die Wertzuweisung und vor allem das Commit (Wertvalidierung, siehe nachfolgenden Teil) zu warten; in diesem Fall ist das Register garantiert frei für neue Zuordnungen, wobei dieses späte Release der Register eine garantierte Wiederherstellung im Fall eines Interrupt Requests zulässt.

Die andere Methode nutzt ein 1:1 Verhältnis zwischen logischen und physikalischen Registern sowie zusätzlich einen Pufferspeicher mit einem Eintrag für jede aktive Instruktion. Dieser Puffer wird *Reorder Buffer* genannt, da er auch als Mittel zur Herstellung der korrekten Berechnungsreihenfolge dient. Aktive Instruktion sind zur Ausführung freigegeben (dispatched) oder bereits ausgeführt, jedoch noch nicht in ihrem Wert validiert (committed). Dieser Pufferspeicher ist zumeist als FIFO-Speicher, in Hardware als Ringspeicher mit Kopf- und Endzeiger implementiert. Jeder Instruktion wird ein Eintrag am Ende des Speichers zugeordnet, sofern sie zur Ausführung freigegeben wird.

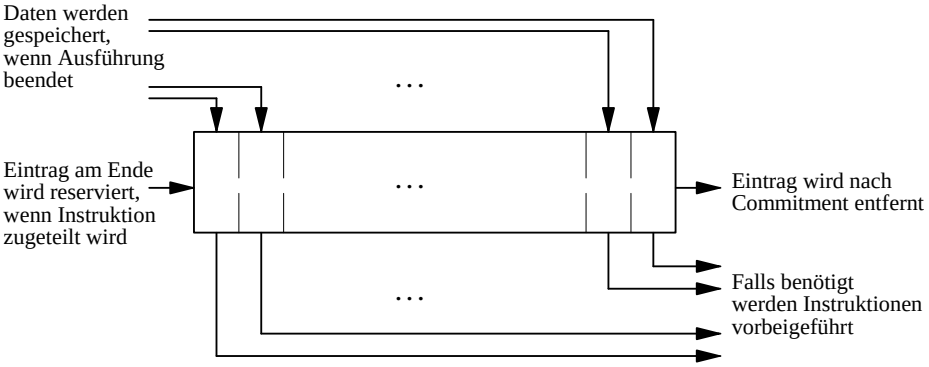


Bild 5.6: Funktionsweise des Reorder Buffers

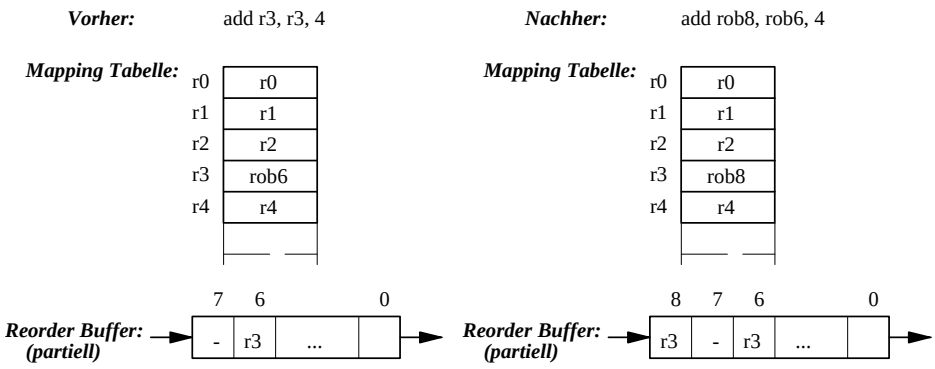


Bild 5.7: Renaming Funktion mit Hilfe des Reorder Buffers

Die Wertzuweisung des Ergebnisses einer Instruktion erfolgt nun in den Reorder Buffer bei gleichzeitiger Zuweisung der entsprechenden Referenzen auf diesen

Puffer. Die Referenzen erfolgen wiederum durch Mapping Tabellen, und zwar zum Zeitpunkt der Dekodierung. In dem Beispiel in Bild 5.7 wird dem Sourceregister r3 der Eintrag rob6 zugewiesen, während das Ergebnis in rob8 eingetragen wird. Erreicht die Instruktion nun den Kopfpointer des Puffers, wird der Eintrag herausgenommen und in das Register übertragen, falls dies bereits möglich ist. Der Zugriff auf den Reorder Buffer erfolgt nach den Regeln des FIFO-Zugriffs, sodass zwei Situationen den Fortlauf blockieren können: Nicht beendete Instruktionen bzw. Referenzen auf den rob-Eintrag sowie ein voller Reorder Buffer. Letztere Situation stoppt die Dekodierungen vorübergehend, die erste muss für die superskalare CPU grundsätzlich lösbar sein!

Gemeinsam ist beiden Verfahren, dass sie die virtuellen WAW und WAR-Hazards durch das dynamische Register Renaming auflösen können, während die realen RAW-Hazards erhalten bleiben. Letztere führen tatsächlich zu Verzögerungen in der Pipeline.

5.3.3 Instruction Issuing und parallele Ausführung

Die Erstellung von Instruktionstupeln, die alle notwendigen Informationen zur Ausführung beinhalten, führt zur nächsten Pipelinestufe, in der die Instruktionen den parallel zueinander vorhandenen Hardwareressourcen zugeteilt werden. Dies ist das eigentliche Herz der superskalaren CPU, denn an dieser Stelle beginnt das Ausführungsfenster.

Mit *Instruction Issue* wird der Test zur Laufzeit bezeichnet, ob die Daten und Ressourcen, die zur Ausführung benötigt werden, tatsächlich vorhanden sind. Ressourcen bezeichnen dabei beispielsweise die Verknüpfungen, Load/Store Pipelines usw., sodass beide Teile tatsächlich vorhanden sein müssen. Idealerweise sollte natürlich eine Instruktion zur Ausführung kommen, wenn alle Eingangsdaten vorhanden sind.

Integer Unit 1		Integer Unit 2		Memory Unit		Branch Unit	
move	R1,r7						
add	R2,R1,4	move	R3,r7	load	r8,(R1)		
add	R4,R3,4	add	r5,r5,1	load	r9,(R2)		
add	r6,r6,1	add	r7,r7,4	store	r9,(R3)	ble	r8,r9,L3
				store	r8,(R4)	blt	r6,r4,L2

Bemerkung: r3 wird auf die physikalischen Register R1, R2, R3 und R4 gemappt

Bild 5.8: Beispiel einer parallelen Ausführung

Das Beispiel aus Bild 5.1 wird in einer Modellarchitektur mit zwei Integer-, einer Branch- und einer Speicherzugriffseinheit zur Ausführung gebracht. Da dieses Beispiel einen nicht optimierten Assemblercode als Grundlage hat, ist auch die

Ausführung in der superskalaren Modell-CPU nicht optimal: Jeder vertikale Schritt bedeutet einen Zeittakt, die Parallelität wird nicht besonders ausgenutzt. Weiterhin wird angenommen, dass der Befehl ble zu keinem Sprung führt, lediglich blt, da dieser Befehl die Schleife begründet. Das Register r3 wurde im Renaming auf die physikalischen Register dargestellt, die anderen Register nicht (obwohl auch sie entsprechend abgebildet werden müssen.

Ferner existiert ein wesentlicher Unterschied in der Wirkungsweise der Branch Unit im Vergleich zu den übrigen Teilen. Branches bedeuten, falls der Sprung durchgeführt wird, einen Update des Program Counters. Dieser wird entweder spekulativ, dann bereits in der Instruction Unit ausgeführt, indem der weitere Programmfluss angenommen wird, oder es werden beide Zweige parallel zueinander ausgeführt. In beiden Fällen entscheidet erst die nachfolgende Commit-Unit, welcher Weg korrekt war und ob es zu einer erneuten Ausführung eines Teils des Instruktionsblocks kommen muss. Im Gegensatz dazu werden in den anderen Einheiten wirkliche Aktionen durchgeführt.

Die Organisation des Instruction Issue Buffers ist auf verschiedene Weisen möglich, wobei drei Basiskonzepte kurz dargestellt werden:

1. Die *Single Queue Methode*: Für eine einzige Warteschlange wird kein Register Renaming benötigt, sofern keine Out-of-Order Zuweisung möglich ist. Die Verfügbarkeit von Operanden kann mit Hilfe einfacher Reservierungsbits verwaltet werden, indem jedes Register als reserviert angesehen wird, falls eine Registermodifizierung vorgenommen wird, sodass bei jeder Vollendung des Schreibzugriffs die Reservierung wieder gelöscht wird. Die Instruktion wird zur Ausführung freigegeben, falls keine Reservierungen bei deren Operanden mehr vorhanden sind.
2. Die *Multiple Queue Methode*: Innerhalb der einzelnen Queues werden die Instruktionen in der entsprechenden Reihenfolge freigegeben; da jedoch mehrere Warteschlangen parallel zueinander existieren, die zugehörigen Befehlsgruppen zugeordnet sein werden, können Instruktionen durchaus in Out-of-Order Reihenfolge freigegeben werden, sodass ein Register Renaming erforderlich sein kann, eventuell in eingeschränkter Form. Im obigen Beispiel, in dem eine Warteschlange für Floating Point, Load/Store und Integer Operationen vorgesehen sind, könnten beispielsweise nur die Register, die aus dem Speicher heraus mit neuen Werten geladen werden, umbenannt werden. Dies würde ein Voreilen dieser Load-Operationen vor die anderen Warteschlangen ermöglichen.
3. Die Methode der *Reservation Stations* erlaubt die Freigabe von beliebigen Operationen in Out-of-Order-Reihenfolge. Jede Operation, die freigegeben werden soll, erhält ein Feld mit einigen Einträgen zum Operationstyp und zur Operandenverwaltung; die Operanden werden ständig auf Verfügbarkeit getestet und in dem Operandenfeld eingetragen, als Wert oder als Pointer. Die Ausführung der Operation wird exakt bei Verfügbarkeit aller Operanden

freigegeben. Die reale Ausführung dieser Reservierungen kann auf einzelne Befehlsgruppen aufgeteilt werden, um Datenpfade zu reduzieren.

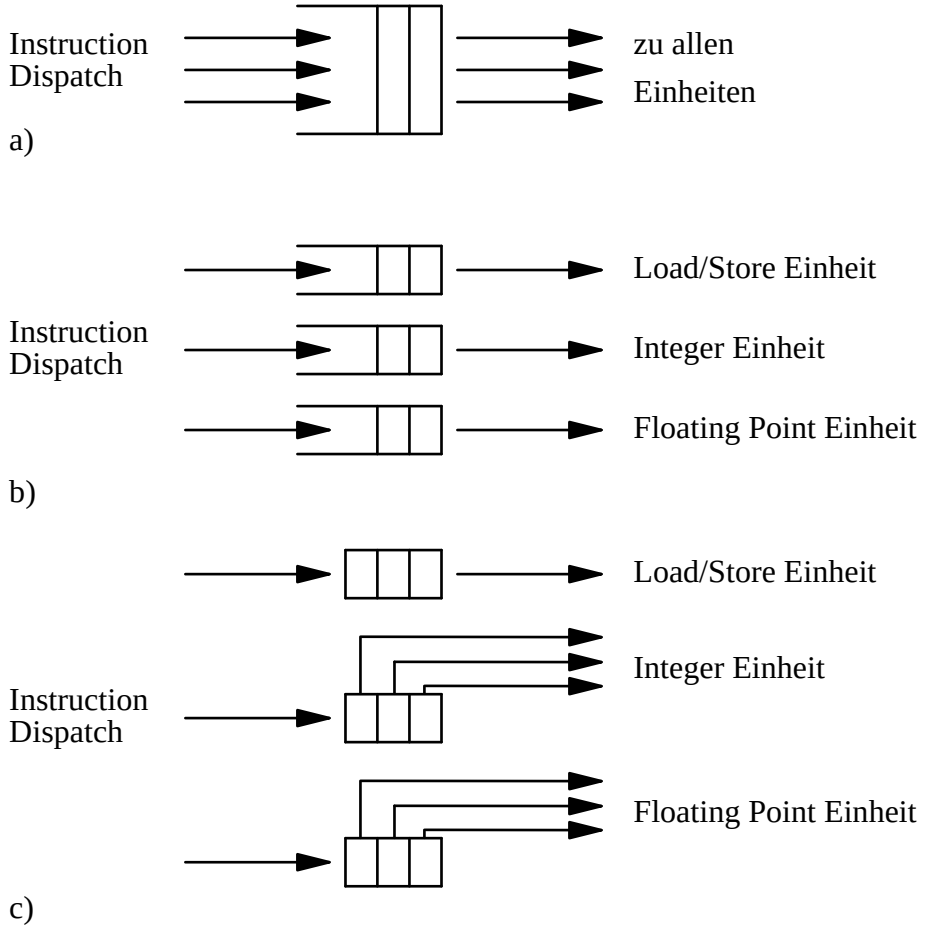


Bild 5.9: Methoden für Instruction Issue Queues: a) Single Queue, b) Multiple Queue, c) Multiple Queue mit partiellem Out-Of-Order-Issueing

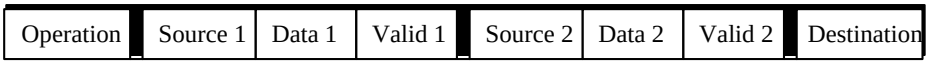


Bild 5.10: Reservation Stations

5.3.4 Die Behandlung von Speicherzugriffen

Das RISC-Konzept, Speicherzugriffe nur für explizite Load/Store-Befehle zu erlauben, wird auch in superskalaren Rechnern angewendet, da dieses Verfahren den Speicherzugriff deutlich kanalisiert. Die Begründung in RISC-Architekturen lag insbesondere in der Möglichkeit, Speicherzugriffe ohne weitere Arithmetik in einer 4stufigen Pipeline zu behandeln; zusätzliche Arithmetik hätte die Anzahl der Stufen vergrößert.

Für superskalare CPUs gilt die Einschränkung für Speicherzugriffe per se nicht! Die Befehle werden nicht mehr innerhalb eines starren Pipelineschemas ausgeführt, das für jede Stufe eine bestimmte Aktion vorsieht. Die Beschränkung auf die expliziten Load/Store-Befehle bedeutet eine Aufwandsbegrenzung, prinzipiell sind jedoch alle Verfahren auf die arithmetischen, logischen Befehle erweiterbar.

Abgesehen von den im folgenden zu besprechenden Vorgängen in der CPU sind für superskalare Rechner dedizierte, hierarchische Speicherkonzepte (→ 9) notwendig, um die entsprechenden Zugriffsgeschwindigkeiten nutzen zu können. Die Hierarchie stellt sich zumeist als dreifach heraus: L1-Cache (primär, klein, sehr schnell), L2-Cache (sekundär, größer, etwas langsamer) und der eigentliche Hauptspeicher (groß, langsam, eventuell mit Virtual Memory Management) bieten genügend Flexibilität, wobei der Primär-Cache meistens (mit Ausnahmen) auf dem Prozessor-Chip integriert ist.

Die Behandlung von Load/Store-Befehlen unterscheidet sich deshalb von den übrigen Befehlen ohne externem Speicherzugriff, weil die Ausführung freigegeben werden muss, ohne dass die Operanden bereits bekannt sind. Die Bestimmung der Speicherstelle benötigt eine *Adresskalkulation*, die ihrerseits Bestandteil der Ausführungsphase ist. Das Ergebnis dieser Kalkulation ist eine logische Adresse, die durch eine *Adressübersetzung* (address translation) wiederum in eine physikalische Adresse umgesetzt wird. Letzterer Teil ist nicht zwingend Bestandteil der CPU, häufig aber integriert, z.B. als Translation Lookaside Buffer (TLB), in dem kürzlich geladene Pages gespeichert sind.

Die Adressübersetzung und der Speicherzugriff können – und werden – als überlappende Operation ausgeführt, um Zeit zu sparen. Die CPU versucht den Zugriff sofort nach Beendigung der Adresskalkulation, und ein integrierter Primär-Cache kann zu einem durchaus beschleunigten Treffer führen. Weitere Verfahren zur Beschleunigung externer Speicherzugriffe bestehen in der überlappenden Ausführung dieser Zugriffe durch mehrere Load/Store-Pipelines und Überlappung mit Instruktionen ohne Speicherzugriff (bei Ausklammerung der Hazards). Das Erlauben von Speicherzugriffen in Out-of-Order-Reihenfolge hingegen ist problematisch, da dieses Verfahren erhebliche Komplikationen mit sich bringt.

Die Speicherzugriffe werden häufig über Registerinhalte definiert: Die Adresse für einen Speicherzugriff wird in einem Register gehalten, die Adresskalkulation erfolgt wie oben dargestellt innerhalb der Ausführungsphase. Eine Out-of-Order-

Ausführung dieser Befehle bedeutet jedoch nicht mehr, dass nur die Registerinhalte gegenüber WAR- und WAW-Hazards geschützt, also umbenannt werden müssen, sondern auch die Speicherinhalte, was angesichts der Anzahl der Speicherzellen unmöglich ist. Die Folge hiervon ist, dass für diesen Fall nur eng begrenzte Subsets von Speicherstellen, die also zum aktiven Satz zählen, derartig behandelt werden können.

Andererseits ist die Möglichkeit, nur einen Speicherzugriff pro Zeiteinheit (Takt) auszuführen, sehr begrenzend und wird den Flaschenhals dieser Architektur darstellen. Beispiele hierzu werden im Kapitel 6 gezeigt, wo konkrete Programme dargestellt sind. Die parallele Ausführung von Speicherzugriffen hingegen benötigt Speicherhierarchien, die *Multiported* ausgeführt sind, weil ansonsten der Speicher wiederum serialisiert. Meist wird der primäre Cache als Multiport-Speicher ausgeführt, da statistisch die Wahrscheinlichkeit zum Zugriff auf höhere Ebenen gering ist (falls die Größe des primären Caches ausreichend ist). Die Multiportfähigkeit wird durch Speicherzellen, die mehrere Zugänge haben, durch mehrere Speicherbänke mit entsprechender Verteilung der Speicherinhalte oder durch mehrfache serielle Zugriffe pro Takt erreicht. Der Speicher muss im Übrigen *nonblocking* sein, falls die überlappende Ausführung eines Speicherzugriffs erlaubt ist. In diesem Fall darf ein Cache-Miss nicht zur Blockade der anderen Operationen führen.

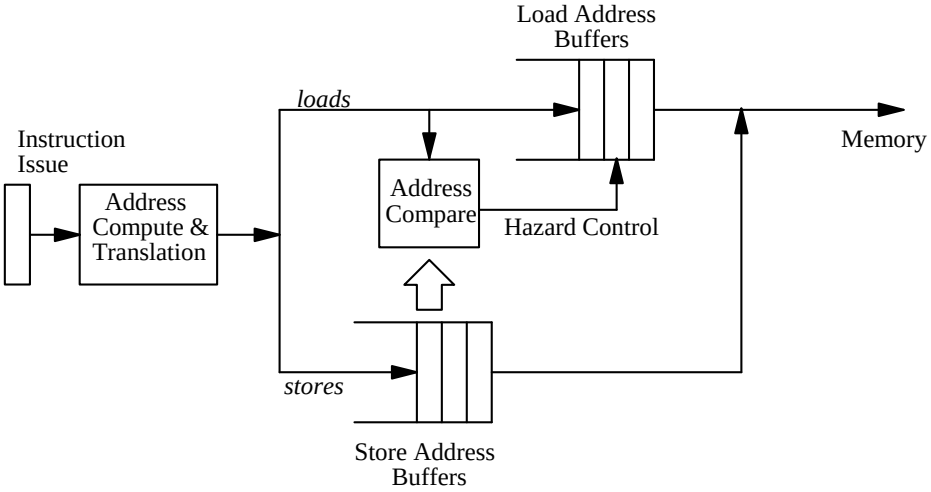


Bild 5.11: Hazard Detection Logik für Speicherzugriffe

Die Überwachung von Hazards kann über die Einführung von *Store Address Buffers* (→ Bild 5.11) erfolgen. Durch Adressvergleich kann überprüft werden, ob Ladevorgänge mit noch ausstehenden Schreibvorgängen zu Hazards führen können, wobei in nachstehender Abbildung insbesondere die ‘echten’ RAW-

Hazards detektiert werden; eine Erweiterung für die virtuellen Hazards ist natürlich möglich.

5.3.5 Die Commit Phase der Befehlsausführung

Die wesentliche Eigenschaft der superskalaren CPU, die Ausführung von Instruktionen in nebenläufiger Weise, führt zu einer Serie von Ergebnissen, die nunmehr in der korrekten sequentiellen Reihenfolge (*Ergebnissequenzialität*!) gespeichert werden müssen. Dieses Erfordernis wird noch durch die zusätzlichen Verfahren wie spekulative Ausführung oder Out-of-Order-Ausführung verschärft: *Die CPU muss in einen präzisen Zustand versetzt werden.*

Der präzise Zustand ist zu zwei verschiedenen Zeitpunkten notwendig: Normalerweise muss nach der Ausführung einer Reihe von Instruktionen die besagte Ergebnissequenzialität hergestellt werden, andererseits besitzt die superskalare CPU die Möglichkeit zur Handhabung von Interrupt Requests, nach deren Behandlung die CPU ebenfalls wieder in einen präzisen Zustand zurückkehren muss (*Recovery*). Hierfür kommen zwei Techniken zur Anwendung:

Das erste Verfahren nutzt Checkpoints, an denen der tatsächliche Zustand der CPU einen Update erfährt und der bisherige in einem *History Buffer* (mit entsprechender Tiefe) gespeichert wird. Hierdurch ist jederzeit ein präziser Zustand herstellbar (auch für die Vergangenheit), wobei innerhalb der Commit Phase nur der korrekte Zustand hergestellt werden muss und die nicht weiter benötigten aus dem History Buffer entfernt werden.

Das zweite Verfahren steht in engem Zusammenhang mit dem Reorder Buffer und wird daher bevorzugt. Der Zustand der Maschine wird dabei in zwei aufgeteilt, wobei der *physikalische Zustand* bei jeder ausgeführten Instruktion gesetzt wird, während der *architekturelle Zustand* erst in der sequenziellen Reihenfolge der Operationen beschrieben wird, wenn ein eventueller spekulativer Status geklärt ist. Hierbei ist es notwendig, dass der spekulative Zustand ebenfalls in einem Reorder Buffer geführt wird; während der Commit Phase wird dann dieser Zustand in den architekturellen Zustand (und den physikalischen) übernommen, die Register aus dem Reorder Buffer in den Register File geschrieben bzw. die Store-Operationen durchgeführt. Die Einträge im Reorder Buffer (Bild 5.6) erhalten hierzu weitere Informationen, in denen der Program Counter Wert, Interrupt Bedingungen, usw. Datenwerte müssen nur dann im Reorder Buffer gehalten werden, wenn kein physikalisches Registerfile (mit mehr als den logischen Registern) vorhanden ist.

Die Einführung eines Reorder Buffers mit allen notwendigen Informationen erweist sich als gute Lösung zur Herstellung eines präzisen Zustands bei gleichzeitiger Durchführung des Register Renamings.

5.4 Einige Beispiele für superskalare Architekturen

Im Folgenden sollen zwei Architekturen zusammenfassend dargestellt werden, um die konkrete Ausführung von superskalaren CPUs darzustellen. Als Beispiele dienen hierbei Die MIPS R10000- und die DEC Alpha 21164-Architektur, die beide mit relativ gut strukturierten Befehlssätzen arbeiten. Diese Architekturen sind nicht mehr die aktuellsten, stellen aber aufgrund ihrer Struktur sehr gute Beispiele für superskalare Architekturen dar.

Im Anschluss daran wird der Pentium-4 diskutiert. Dieser Prozessor stellt mit seinem CISC-Befehlssatz sozusagen das Gegenteil dessen dar, was hier diskutiert wurde: superskalare Mikroprozessoren als Weiterentwicklung von RISC-Prozessoren.

5.4.1 MIPS R10000

Bild 5.12 zeigt den Aufbau der MIPS R10000-CPU. Sie kann als Beispiel für eine typische CPU superskalaren Aufbaus gelten, wobei im Folgenden die exakten Eigenschaften dargestellt sind.

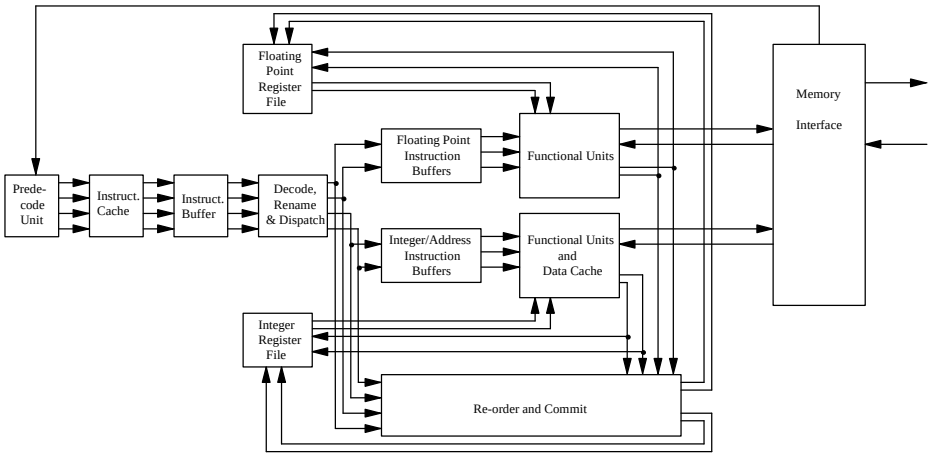


Bild 5.12: Aufbau MIPS R10000

In der R10000 werden zu einem Zeitpunkt 4 Instruktionen vom Instruktionscache geladen. Die Informationen in diesem Cache enthalten 4 zusätzliche Predecode Bits zur Unterscheidung des Instruktionstyps. Die Ausführung von Branch Befehlen wird vorhergesagt, wobei jedem der insgesamt 512 Einträge im Instruktionscache ein 2-Bit-Zähler (ohne Überlauf) zugeordnet ist, in dem die bisherige Verzweigungsgeschichte gezählt wird.

Vorhergesagte Sprünge führen zu einem zusätzlichen Takt zum Laden der neuen Instruktionen aus dem Cache, während die ohne Sprung nachfolgenden in einem Extra Cache, dem Resume Cache, gespeichert werden, um bei falscher Vorhersage schnellstmöglich nachladen zu können. Der Umfang dieses Resumespeichers beträgt 4 Instruktionsblöcke, sodass auf diese Weise 4 Branchbefehle pro Zeit hantierbar sind.

Das Register Renaming benutzt einen physikalischen Registerfile, der mit 64 Registern die zweifache Größe des logischen File (32) besitzt. Hierzu wird jedem logischen Register aus den Befehlsoperanden ein Eintrag in einer Remapping Tabelle auf ein physikalisches Register gegeben.

Die drei Instruction Queues für Speicherzugriffe, Integer und Floating Point Operationen haben eine Tiefe von 16 Einträgen, wobei zu einem Zeitpunkt 4 Instruktionen gleichzeitig zugeordnet werden können. Die Einträge entsprechen in etwa den Reservation Stations, wobei pro Eintrag Pointer auf die Operanden, jedoch keine Validbits vorhanden sind. Jede Instruktion kann zur Ausführung kommen, falls alle Operanden ein ‘not busy’ zeigen, wofür alle Datenquellen überwacht werden.

Die 5 Funktionseinheiten bestehen aus einem Adressaddierer, zwei Integer-ALUs, einem Floating Point Addierer und einer Floating Point Einheit für Multiplikation, Division und Quadratwurzel. Die beiden Integer-ALUs sind nicht vollkommen identisch, beide beinhalten die Basisoperationen Addition/Subtraktion/Logik, eine jedoch nur die Shiftoperationen, die andere Multiplikation und Division zusätzlich.

Generell werden ältere Operationen eher zur Ausführung gebracht, prinzipiell wird dies jedoch nur für Speicheroperationen zur einfacheren Detektion von Hazards durchgeführt. Die R10000-CPU besitzt einen primären On-Chip-Cache mit 32 kByte Tiefe, 2-Wege assoziativ, 32-Byte-Lines und unterstützt einen Sekundär-cache.

Die Herstellung eines präzisen Zustands wird durch einen Reorder Buffer gewährleistet. 4 Instruktionen pro Takt können für gültig erklärt werden, in der ursprünglichen Reihenfolge des Programms. Bei diesem Verfahren wird der Inhalt des physikalischen Registers zum neuen gültigen Wert erklärt, das Register wiederum freigegeben. Exception Bedingungen werden zwischengespeichert, um bei Ausnahmen den vorhergehenden Zustand wiederherzustellen. Bei Branchvor-hersage wird eine Kopie aller Register angefertigt, um bei falscher Vorhersage die korrekten Werte wiederherzustellen.

5.4.2 Alpha 21164

Die DEC Alpha 21164-CPU stellt eine insgesamt vereinfachtere Architektur für superskalare CPU dar. Diese Vereinfachung scheint in Zusammenhang mit den sehr großen Taktraten zu stehen.

Die Alpha-CPU besitzt einen Instruction Cache von 8 kByte Größe, aus dem zu einem Zeitpunkt 4 Instruktionen in einen von zwei Instruction Buffer geladen werden können. Diese Instruction Buffer sind jeweils 4 Einträge groß, die Befehle kommen ausschließlich in sequentieller Reihenfolge zur Ausführung, wodurch jeder Buffer erst geleert sein muss, bevor der andere benutzt wird. Dies bedeutet eine Restriktion in der Ausführung, das Design wird jedoch sehr vereinfacht.

Die Branchvorhersage erfolgt wiederum durch 2-Bit-Zähler (pro Instruktion im Puffer einen); im Unterschied zur R10000 ist aber nur eine Vorhersage möglich, ein weiterer Branchbefehl wird solange verzögert, bis der vorangegangene ausgeführt ist.

Die Zuführung ausführbereiter Instruktionen erfolgt durch die Single Queue Methode, d.h. eine parallele Zuführung ist nicht möglich. Hierdurch werden Maßnahmen zum Register Renaming unnötig, das Design bleibt einfach. Zur Ausführung von Instruktionen stehen 4 Funktionseinheiten zur Verfügung: 2 Integer ALUs (nicht identisch, eine für Shift und Multiplikation, die andere für Branchevaluation), ein Floating Point Addierer und ein Floating Point Multiplizierer.

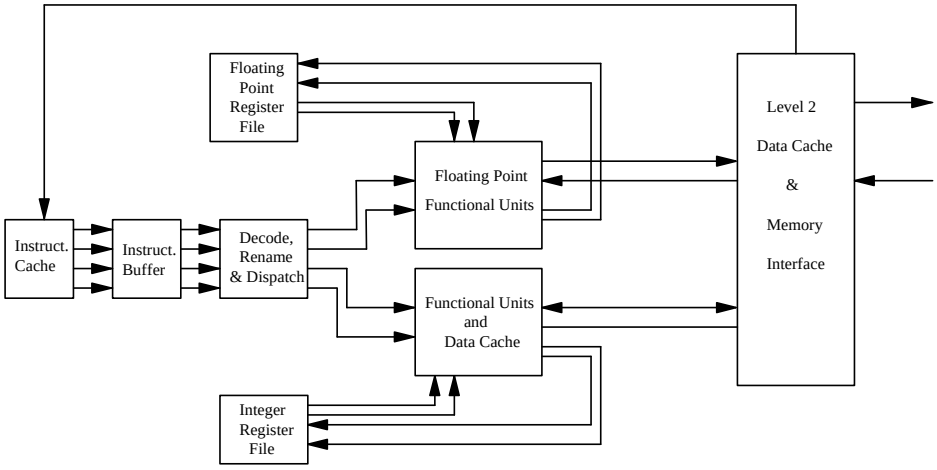


Bild 5.13: Organisation der DEC Alpha 21164

Das Cache Konzept umfasst einen Instruktionscache (8 kByte), einen primären Datencache (8 kByte, direct mapped) und einen gemeinsamen, externen Sekundärcache (96 kByte, 3-Wege-assoziativ). Die primären Caches können Datenzugriffe in einem Takt behandeln und sind in der Lage, eine Reihe von ausstehenden Fehlzugriffen zu behandeln (6 Miss Address File Einträge).

Die Ergebnissequenz wird in einem einfachen Reorder Buffer erfasst und entsprechend für gültig erklärt. Dies beinhaltet das Update der Integer Register in

der originalen Programmsequenz, während Floating Point Operationen dies auch Out-OF-Order dürfen. Hieraus resultiert, dass die Floating Point Register ggf. in eine nicht-präzisen Zustand sein können, der gelegentlich korrigiert wird.

5.4.3 Intel Pentium-4

Die Intel Pentium-4-Architektur ist eine über viele Stufen weiterentwickelte Architektur, deren Ursprung 1978 mit dem Intel 8086 gelegt wurde. Zu diesem Zeitpunkt war RISC noch eine Nischenarchitektur, so verwundert es nicht, dass dieser Mikroprozessor eine CISC-Architektur darstellt.

Die Befehlsbearbeitung im Pentium 4 ist zweigeteilt: Die Fetch- und Decodierphase sind für das Interface zum Instruktionsspeicher verantwortlich, die interne Ausführungspipeline andererseits sorgt für eine davon entkoppelte Ausführung der Befehle.

Zunächst fällt das Fehlen eines L1-Instruktionscache auf, nur der L2-Cache ist verfügbar. Dies ist möglich, weil der Fetch/Decodier- und der Execute-Teil voneinander getrennt sind, sodass auf einen L1-Cache mit einem Zugriff von 1 Takt verzichtet werden kann.

Der L1-Cache wird in gewissem Sinn durch den Trace-Cache, der bereits zum Execute-Teil gehört, ersetzt: Hier werden die bereits durchlaufenen Befehle mit geschrieben (’getraced’), einschließlich einer Branch Prediction mit Branch Target Buffer (die beiden BTBs sind verschieden!). Dieses Verfahren bewährt sich sehr bei Schleifen, denn hier werden in der Regel die Instruktionspfade mehrfach durchlaufen.

Zentrales Moment ist jedoch die Umsetzung der Assemblerbefehle in µOps (Mikrooperationen). Hierdurch wird das notwendige Interface geschaffen, um die CISC-Befehlsstruktur mit z.T. erheblicher Abweichung vom RISC-Ideal (immer gleiche Anzahl von Bearbeitungsschritten pro Befehl) ausgeglichen, indem bei komplexen Befehlen mehrere µOps generiert werden.

Die µOps werden dann in einer 20stufigen Pipeline bearbeitet: Insgesamt 6 Pipelines stehen zur Verfügung (Bild 5.14), 3 Fetches von µOps aus dem Trace-Cache werden pro Takt zur Ausführung gestartet.

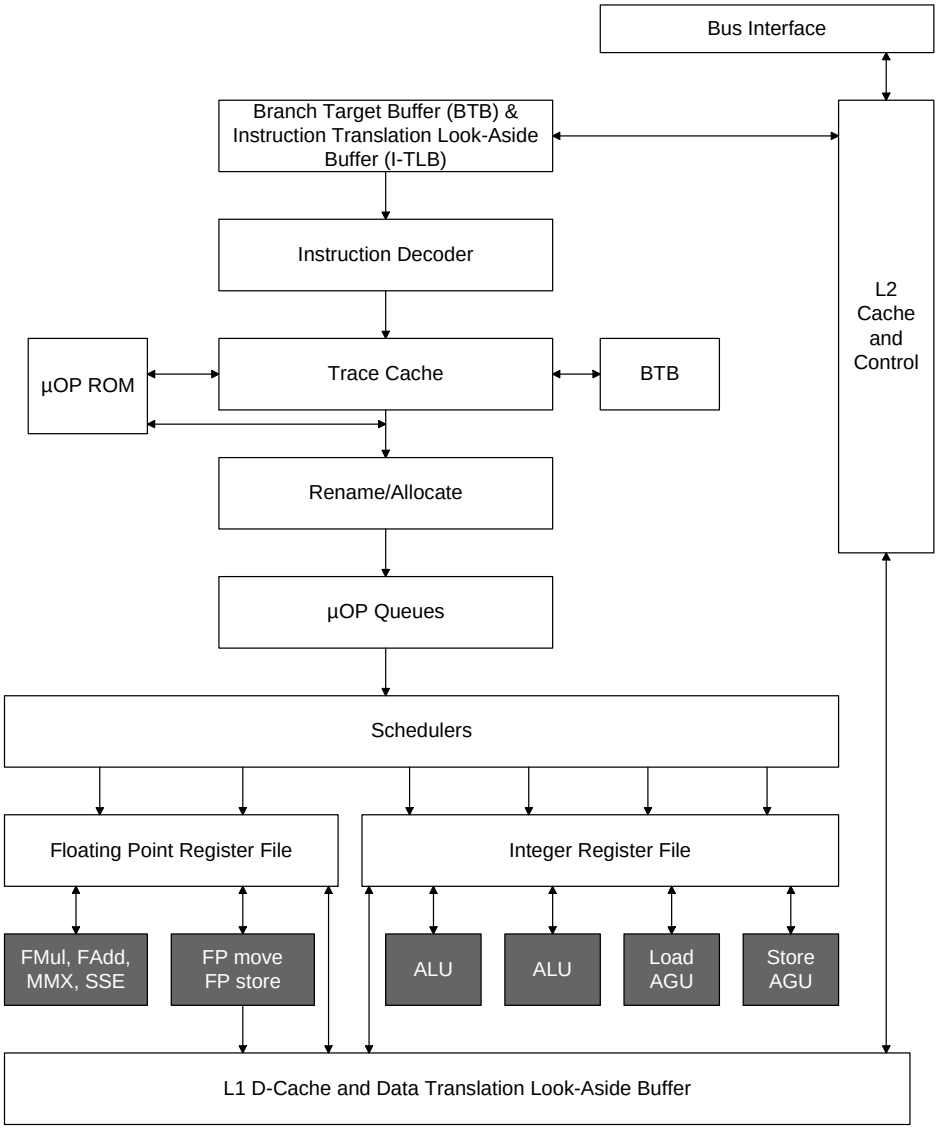


Bild 5.14: Mikroarchitektur Intel Pentium 4

6 Wechselwirkungen zwischen superskalarer Rechnerarchitektur und Compilertechnologie

Die Entwicklung von Prozessoren in den letzten Jahren sowie die vermutete Entwicklung für die nächsten Jahre lässt sich durch folgende Zahlen gut darstellen:

- 1990 konnten Mainstream-Mikroprozessoren 1 Instruktion pro Takt ausführen (typische RISC-Technologie)
- 1995 betrug diese Zahl bei Mainstream-Mikroprozessoren etwa 4 Instruktionen pro Takt
- In naher Zukunft werden 16 Instruktionen pro Takt in paralleler Ausführung als Richtwert vermutet.

Die letzte Zahl eines theoretischen IPC (Instruction per Clock) von 16 wurde im Jahre 1995 bereits für das Jahr 2000 vermutet, trat bislang jedoch noch nicht ein. Die Gründe hierfür sind einfach: Man hat es bislang nicht geschafft, eine vernünftige praktische Performance aus den theoretischen Wert herauszuholen. Allein der Fetch von 16 Instruktionen, der für eine 16fache Parallelität notwendig ist, erweist sich als schwierig: Bei klassischen Befehlssätzen ist jeder 5. bis 6. Befehl eine bedingte Verzweigung, sodass über viele Spekulationen hinweg Instruktionen geladen werden müssten.

Dennoch: Die theoretische Performance der Prozessoren, bezogen auf den Takt, steigt weiter. Prozessoren und Compiler jedoch haben sich schon seit längerem gegenseitig beeinflusst: Angepasste Registersätze sowie einige Assemblerbefehle (etwa für den Stackframe bei Funktionsaufrufen) zeigen solche Beeinflussungen auch in der Gegenrichtung. In den folgenden Abschnitten wird nun diskutiert, warum es jetzt neue Compilertechnologie und -strategien geben muss und worin diese tatsächlich bestehen. Einige Beispielprogramme in den Abschnitten runden den Überblick ab.

6.1 Einführung in die Problematik: Warum werden neue Compilertechnologien benötigt?

Traditionelle Compilertechnologien beruhen auf Strategien, die unter folgenden Aspekten zusammengefasst werden können:

- Nicht durchlaufene Codebereiche werden eliminiert (Dead Code Elimination, für Programmgröße)

- Unnötige Codebereiche, die zwar durchlaufen, aber nicht benötigt werden, da sie keinen Nettoeffekt erzeugen, werden ebenfalls gelöscht (für Programmschnelligkeit)
- Variablen, die häufig benötigt werden oder nur für Zwischenergebnisse notwendig sind, werden in (High-Speed-) Registern gehalten, um auf längere Speicherzugriffe zu verzichten (Erhöhung der Ausführungszeit von Programmroutinen).

Diese Optimierungen werden weiterhin durchgeführt werden, da diese Form auch bei Instruktionsparallelismus Erfolge zeigt. Zusätzliche Optimierungen sind jedoch dringend notwendig, wobei nun die Parallelisierbarkeit im Vordergrund steht. Immerhin ist bei einem Prozessor, der die 4fache parallele Ausführung von Instruktionen zueinander ermöglicht, eine Performancesteigerung im gleichen Faktor theoretisch möglich, falls eben das Programm parallelisierbar übersetzt wird. Dieser Faktor alleine ist genügend Motivation für neue Compilertechnologien, die im Folgenden vorgestellt werden.

6.2 Strategien für eine Optimierung im Hinblick auf superskalare Architekturen

Das Vorgehen innerhalb dieses Abschnitts ist dreiteilig: Zunächst werden die Verfahren zur Optimierung von compiliertem Code im Einzelnen vorgestellt. Hierunter fallen einige Basiskonzepte, das Formen von Superblöcken zur Vergrößerung der Instruktionsblöcke ohne Unterbrechung, das Einfügen von **bedingten Instruktionen** mit Steuerungsflags, **Register Renaming** zur Compilezeit (!), **Loop Unrolling**.

Im zweiten Teil wird auf einen Hauptbestandteil der Technologien eingegangen: Die bedingten Befehle mit Steuerungsflags. Im dritten Teil folgt dann die Analyse von Speicherabhängigkeiten: Beide Teile gehören zu den neueren Compilertechnologien und dürften Bestandteil der zukünftigen Generationen sein.

6.2.1 Compilierung für Instruction Level Parallelism I: Basiskonzepte

Anhand einer kurzen Routine, die in Schleifenform einem Array b[] neue Werte in Abhängigkeit eines Arrays a[] zuweist, können einige der Konzepte gezeigt werden.

Bei der Übersetzung in der Assemblercode wurde ein Pseudocode zugrundegelegt, der nach dem Schema <OpCode> <DestReg>, <SourceReg1>, <SourceReg2> aufgebaut ist. Ferner sind für die eigentliche Schleife den Assemblerbefehlen Nummern zugeteilt, um die späteren Standort identifizieren zu können. Für die Bestimmung der Laufzeit einer Schleife gelten folgende Regeln:

(a)

```
int a[N+1], b[N+1];
int i, n = N;
for( i = 0; i < n; i++ )
{ if( a[i] != 0 )
     b[i] = b[i] + b[i+1];
  else
     b[i] = 0;
}
```

(b)

	Nr.	Assemblercode	T
L0:		mov r3, 0	-
		mul r7, n, 4	-
		add r1, b, 4	-
L1:	1	ld r2, mem(a+r3)	0
	2	beq r2, 0, L3	2
L2:	3	ld r4, mem(b+r3)	2
	4	ld r5, mem(r1+r3)	2
	5	add r6, r5, r4	4
	6	st mem(b+r3), r6	5
	7	jmp L4	5
L3:	8	st mem(b+r3), 0	-
L4:	9	add r3, r3, 4	5
	10	blt r3, r7, L1	6

(c)

```
Registerinhalte:
r1   =    &b[1]
r2   =    a[i]
r3   =    i * 4
r4   =    b[i]
r5   =    b[i+1]
r6   =    b[i] + b[i+1]
r7   =    n * 4
```

Bild 6.1: (a) Beispiel für Programmschleife mit bedingter Zuweisung, (b) Traditionelle Übersetzung, (c) Registernutzung

1. Es können prinzipiell beliebig viele Assemblerbefehle parallel zueinander zur Ausführung gelangen, wobei allerdings eine *Out-Of-Order-Execution* nicht betrachtet wird.
2. Alle virtuellen Hazards wie WAW und WAR werden durch die Hardware des Prozessors gelöst.
3. Jeder Speicherzugriff hat einen Taktbedarf von 2 Takten, alle anderen Befehle einen von einem Takt.
4. Die Problematik der Vorhersage von Branchbefehlen mit ggf. Korrektur wird nicht weiter berücksichtigt, die Schleifen werden sozusagen immer korrekt durchlaufen!

Das Assemblerlisting in Bild 6.1 b) zeigt unter diesen Umständen folgende Struktur: Innerhalb der Schleife entstehen 4 Basisblöcke, mit L1 bis L4 bezeichnet. Bild 6.2 stellt den Kontrollflussgraphen dar, wie er für den `IF`- und den `ELSE`-Zweig durchlaufen wird. In dem Kontrollflussgraph werden die Basisblöcke durch Pfeile verbunden, wenn ein Übergang möglich ist.

Die Abhängigkeit zwischen der Instruktion 1 und 2 (1 schreibt r2, 2 liest r2 zum Vergleich) wurde als **RAW**-Hazard (→ 4.4.1) bezeichnet. Die Konsequenz ist ein Warten auf die Ausführung, sodass nicht parallel ausgeführt wird. Um die Abhängigkeiten darzustellen, wird ein Abhängigkeitsgraph zwischen den einzelnen Instruktionen dargestellt, und zwar bezüglich Register- wie Speicherabhängigkeiten (Bild 6.3 a und b).

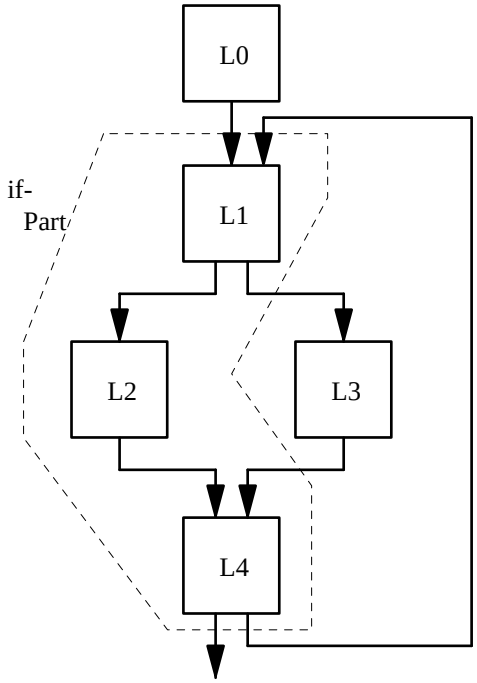


Bild 6.2: Kontrollflussgraph für Beispielprogramm aus Bild 6.1

Während die dargestellte Abhängigkeit zwischen den Instruktionen 1 und 2 als Flussabhängigkeit bezeichnet wird, wird auch die Abhängigkeit zwischen den Instruktionen 6 und 9, als WAR-Hazard bezeichnet (→ 5.2), aufgeführt. In der Sprache der Compilerbauer wird dies als Anti-Abhängigkeit bezeichnet.

Die Output Dependency ist in dem Beispiel nicht direkt zu entdecken; sie existiert jedoch in Form von zwei Hintereinanderausführungen der Schleife, wo beispielsweise in Instruktion 4 der gleiche Inhalt in ein Register geschrieben wird wie in Instruktion 3 der nächsten Schleife. Dies wird als Output-Abhängigkeit, exakter Schleifen-bedingte Output-Abhängigkeit (*Loop-Carried Output Dependency*) bezeichnet.

Bild 6.3 zeigt die jeweiligen Abhängigkeiten in diesem Beispiel.

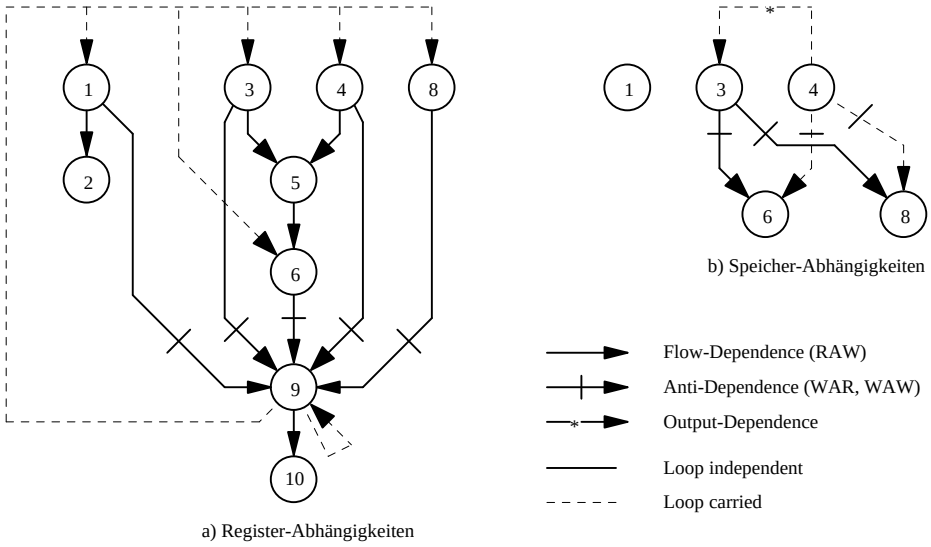


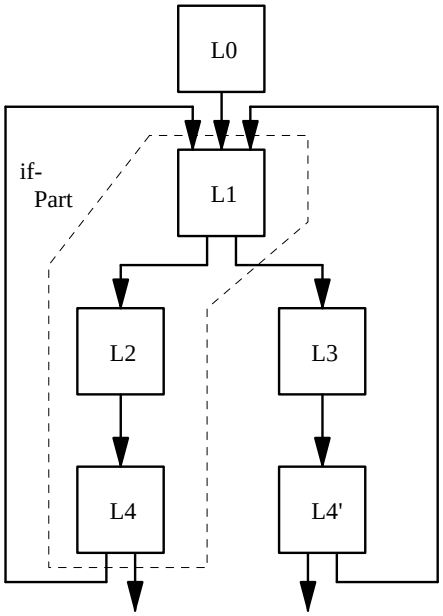
Bild 6.3: Abhängigkeitsgraph für a) Register-, b) Speicherabhängigkeiten

Die bisherige Analyse für das Beispiel zeigt einen nur sehr geringen Anteil an paralleler Ausführung auf Instruktionslevel: Im `IF`-Teil werden pro Schleife 9 Befehle in 6 Takten ausgeführt, die Performance beträgt also 1,5 Instr./Takt. Der Grund hierfür ist sehr einfach, da die Compilation viele Basisblöcke mit jeweils nur wenigen Befehlen geliefert hat.
Zwei Ansätze zur Vergrößerung dieser Basisblöcke werden im Folgenden diskutiert. Der erste Ansatz versucht, die Größe des wesentlichen Basisblocks zu vergrößern, also durch Zusammenfassung zu einem Superblock, der innerhalb der Ausführung möglichst häufig durchlaufen werden sollte (der andere Zweig möge also eine seltene Ausnahme sein) die Performance zu verbessern.

In dem Beispiel aus Bild 6.1 sei nun angenommen, dass der `ELSE`-Zweig selten durchlaufen wird. Der Assemblercode und der Kontrollflussgraph werden nun so variiert, dass der Block L4 kopiert und die Abfolge der Blöcke L1, L2 und L4 als ein Superblock (punktiert eingerahmt) aufgefasst wird:

Die eigentliche Optimierung für den Assemblercode besteht nun darin, dass die Ladeoperation parallel zueinander ausgeführt werden, wodurch zwei Takte gespart werden können. Diese Ladeoperation werden für den `ELSE`-Teil umsonst durchgeführt. In diesem Beispiel führt dies zu keiner Verschlechterung der Geschwindigkeit für den `ELSE`-Zweig, es würden weiterhin 3 Takte benötigt werden (diese Taktanzahl war in Bild 6.1 nicht dargestellt). Man möge bitte beachten, dass Verzögerungen durch falsche Vorhersagen bei den Verzweigungsbefehlen nicht

berücksichtigt werden! Dies kann für die Allgemeinheit jedoch nicht vorhergesagt werden, ggf. erfolgt die Optimierung der Hauptschleife auf Kosten des Sonderfalls.



	Nr.	Assembler Code		Takt
L1:	1	ld	r2, mem(a + r3)	0
	3	ld	r4, mem(b + r3)	0
	4	ld	r5, mem(r1 + r3)	0
	2	beq	r2, 0, L3	2
	5	add	r6, r5, r4	2
	6	st	mem(b + r3), r6	3
	9	add	r3, r3, 4	3
	10	blt	r3, r7, L1	4

4 Zyklen Bearbeitung pro Schleife

2 Instruktionen/Takt

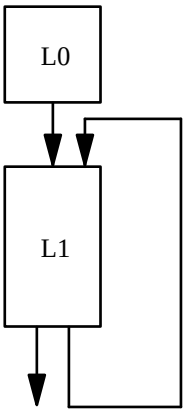
Bild 6.4: Assemblercode für `IF`-Teil und Kontrollflussgraph für Superblock-Version

Das Resultat dieser Bemühungen ist eine Verbesserung der Performance auf 2.0 Instruktionen pro Takt bei Verminderung der benötigten Takte auf 4 pro Schleife. Diese Beschleunigung beruht auf der Erhöhung der Möglichkeit zur parallelen Ausführung (Load-Befehle), wobei das Verfahren als *Global Acyclic Scheduling* bezeichnet wird.

Das Attribut global entstammt dabei dem Vorgehen, über die bisherigen Basisblockgrenzen hinweg Befehle zu verschieben, während die Bezeichnung azyklisch für die Begrenzung innerhalb des azyklischen Teils stammt. Die Verschiebung der Load-Befehle aus dem Bereich nach dem Branch-Befehl in den vorher, die ja eine garantierte Ausführung bewirken, wird als *Speculative Code Motion* bezeichnet.

Die Verschiebungen sind zulässig, wenn die Ausführung des `ELSE`-Teils keine falschen Resultate hervorruft. Dies muss vom Compiler zusätzlich geprüft werden, was in diesem Beispiel natürlich gewährleistet ist. Zusammenfassend kann gesagt werden, dass die Einfügung von Superblöcken den größten Gewinn erzielt, wenn ein Pfad des Codes besonders häufig durchlaufen wird. Bei mehreren Pfaden kann diese Verfahren jedoch eher das Gegenteil zur Laufzeit bewirken, da dann die Optimierungen ggf. an falscher Stelle vorgenommen wurden.

Die zweite Methode ist in Bild 6.5 dargestellt. Hierbei werden neue Assemblerbefehle von der CPU gefordert, sodass die Wechselwirkung zwischen Compilerbau und CPU-Befehlssatz einmal mehr deutlich wird. Das Kernstück des Verfahrens besteht in dem Ersatz von Branchbefehlen durch bedingte (andere) Befehle mit Vorhersageflags, die sogenannte *Predicated Execution*.



	Nr.	Assembler Code		Takt
L1:	1	ld r2, mem(a + r3)		0
	3	ld r4, mem(b + r3)		0
	4	ld r5, mem(r1 + r3)		0
	5	add r6, r5, r4		2
	2	peq p1, r2, 0		2
	6	st mem(b + r3), r6	(/p1)	3
	8	st mem(b + r3), 0	(p1)	3
	9	add r3, r3, 4		3
	10	blt r3, r7, L1		4

4 Zyklen Bearbeitung pro Schleife

2 Instruktionen/Takt

Bild 6.5: Bildung größerer Blöcke durch bedingte Befehle

Dem Compiler wird die Aufgabe gestellt, einen Branch-Befehl (mit oder ohne integrierten Vergleichsbefehl) in eine Abfolge von Instruktionen zur Belegung der Aussageflags und zur Ausführung der bedingten Befehle umzusetzen. Dieser Vorgang wird *If-Conversion* genannt. Die Einfügung zusätzlicher Befehle wird durch die parallele Ausführung ausgeglichen, zusätzlich werden nunmehr nicht mehrere Basisblocks, sondern nur noch (in diesem Beispiel) ein Basisblock generiert. Ein derartiger, aus der *If-Conversion* entstandener Block von Befehlen wird als **Hyperblock** bezeichnet. Die bedingten Befehle, die sich im obigen Beispiel gegenseitig ausschließen, werden ebenfalls parallel zueinander ausgeführt, wobei nur einer der beiden wirklich zu einem Ergebnis führt. Bild 6.5 stellt wiederum beide Pfade im ursprünglichen Programm dar. Das Ergebnis für die Einführung bedingter Befehle anstelle von Branch-Befehlen beträgt im Übrigen wiederum 4 Takte bei 2 Instruktionen/Takt; es unterscheidet sich somit nicht von der vorhergehenden Optimierung.

6.2.2 Compilierung für Instruction Level Parallelism II: Loop Unrolling

Die bisherigen Maßnahmen dienten der Vergrößerung der Basisblöcke durch Überwindung der bedingten Verzweigungen. Die entsprechenden Sourcecodekonstrukte bestanden in den Einfach- oder Mehrfachverzweigungen.

Als weitere Maßnahme zur Verbesserung der Performance in superskalaren Prozessoren muss die statistische Auswirkung der Schleifen verringert werden. Die Branchbefehle am Ende von Schleifen, die Rückwärtssprünge mit zumeist häufiger Ausführung darstellen, können nicht verhindert werden, es sei denn, die Schleifen werden auf Kosten der Codelänge in linearen Code umgesetzt, der dann – unter Berücksichtigung von Datenabhängigkeiten – parallel zueinander ausführbar ist.

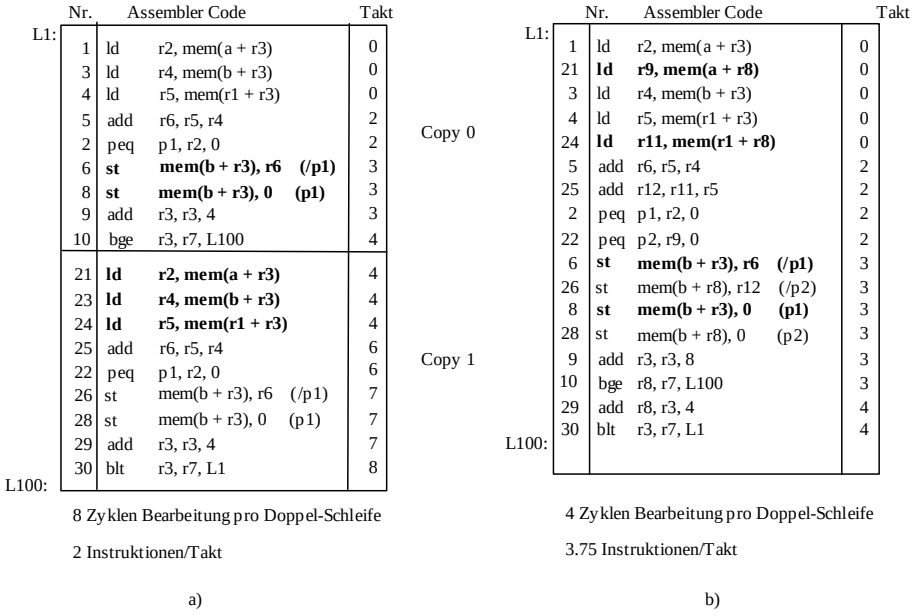


Bild 6.6: Loop Unrolling ohne (a) und mit (b) Analyse und Beseitigung der Datenabhängigkeiten

Das Verfahren des **Loop Unrolling**, wie es in der Praxis angewendet wird, betrachtet tatsächlich eine kleine Anzahl von Schleifen, z.B. 2, und versucht diese zusammenzulegen und ggf. Datenabhängigkeiten zu beseitigen. Die geringe Anzahl hat durchaus Gründe: So ist häufig zur Compilezeit nicht entscheidbar, wie viele Schleifen durchlaufen werden müssen; dies wäre aber unabdingbar für ein komplettes Unrolling ohne weitere Branchbefehle, die ja zwecks Erzeugung größerer Basisblöcke verhindert werden sollen. Zudem würde eine Zusammenfassung

von vielen Schleifen nur noch geringe Performancesteigerung bei stark zunehmender Codegröße bedeuten.

In Bild 6.6a wurde versucht, zwei aufeinanderfolgende Schleifen, jeweils einen Hyperblock (→ Bild 6.5) darstellend, direkt miteinander zu verbinden, indem sie nacheinander ohne Branchbefehl ausgeführt werden. Das Beispiel zeigt, dass dieses einfache Verfahren nicht zu einer Performancesteigerung führt. Ursache hierfür sind die Datenabhängigkeiten in den Registern und dem Speicher, da die beiden (ehemaligen) Schleifen nur als einfache Kopien hintereinander aufgereiht sind.

Eine Beseitigung der Abhängigkeiten zur Erreichung höherer Performance (durch dann mögliche parallele Ausführung) ist das Ziel dieses Abschnitts. Hierzu müssen die durch Loop Unrolling entstandenen Abhängigkeiten kurz aufgeführt werden:

- WAR-Hazards bzw. Anti-Abhängigkeiten entstehen durch die Aneinanderfügung zweier Schleifen. Dies fiel bisher nicht ins Gewicht, da die Schleifen durch einen Vergleich und anschließenden Sprung (1 Takt Bedarf!) voneinander getrennt waren.
- Speicherabhängigkeiten können denselben Effekt erzeugen. Während bei Registern eine Analyse durch den Compiler möglich ist (die Register werden immer direkt adressiert), kann dies je nach zulässigen Adressierungsmodi für Speicherzugriffe nicht möglich sein, und zwar dann, wenn eine Indirektion vorliegt. Die Konsequenz: Bei unmöglicher Analyse muss der Compiler in konservativer Weise annehmen, dass Abhängigkeiten vorliegen, um garantiert korrekte Ergebnisse zur Laufzeit zu produzieren.

In Bild 6.6a sind Speicherabhängigkeiten sichtbar: Die Instruktionen 6 bzw. 8 (eine dieser wird in jedem Fall durchlaufen) bezeichnen ggf. die gleichen Speicherstellen wie die (neuen) Instruktionen 21, 23 und 24. Die Problematik besteht jedoch in der Bestimmung, ob diese Speicherstellen nun tatsächlich übereinstimmen.

Die Abhängigkeit zwischen 6/8 und 21 ist ggf. relativ einfach zu bestimmen, falls a[] und b[] voneinander unabhängige, nicht überlappende Arrays darstellen. Schwieriger liegen die Dinge bei 23 und 24 in Verbindung mit 6/8, da nun eine echte Abhängigkeitsanalyse die Adressfindung in r1 und r3 mit allen Fällen durchlaufen muss.

In diesem Fall ist es möglich, eine Unabhängigkeit dieser Load- und Store-Instruktionen zu finden. Weiterhin können Ladezugriffe aus zwei (Teil-)Schleifen analysiert werden (b[i+1] der ersten Teilschleife und b[i] der zweiten sind identisch), die beim *Unrolling* zu einem Zugriff zusammenfassbar sind. Mit diesen Methoden und Annahme kann der Compiler dann eine Umstellung der Befehle vornehmen, verbunden mit einem *Renaming*, das Ergebnis ist in Bild 6.6b dargestellt und zeigt eine nahezu Verdopplung der Performance.

6.2.3 Compilierung für bedingte Befehle mit Steuerungsbits

Zwei wichtige Bereiche, die beide mit relativ neuen Techniken (im Sinn des Compilerbaus) verbunden sind, sollen in den folgenden Abschnitten näher betrachtet werden: Der Ersatz bedingter Sprungbefehle durch bedingte (andere) Befehle einerseits und die Durchführung von Speicherabhängigkeitsanalysen bei Loop Unrolling andererseits.

Für die Einführung bedingter Befehle, Gegenstand dieses Abschnitts, soll als Beispiel die Übersetzung des Sourcecodes der inneren Schleife von wc (word count, UNIX) dienen. Dieser Code ist in Bild 6.7 dargestellt.

```
      linect = wordct = charct = token = 0;
      for( ;; )
      {
A:      if( --(fp)->cnt < 0 )
C:        c = filbuf( fp );
        else
B:        c = *(fp)->ptr++;
D:      if( c == EOF ) break;
E:      charct++;
F:      if( (c > ' ' ) && c < 0177 )
        {
H:        if( !token )
          {
K:           wordct++;
             token++;
          }
          continue;
        }
G:      if( c == '\n' )
I:        linect++;
J:      else if( c != ' ' && c != '\t' )
L:        continue;
M:      token = 0;
      }
```

Bild 6.7: Sourcecode für innere Schleife von wc

Die erste Übersetzung in einen ‘klassischen’ Assemblercode zeigt die Abhängigkeit dieses Codes von Branchbefehlen: 8 der 28 Befehle sind bedingte Branches, weitere 4 Sprungbefehle sind vorhanden.

Bild 6.8 zeigt eine ’klassische’ Assemblerübersetzung (a) sowie einen Kontrollflussgraph mit den Messergebnisse mit einem Lauf über einen Text mit ca. 105 k Character. Anhand dieser Laufzeitanalyse lässt sich (leider) feststellen, dass kein Weg innerhalb des Codes wirklich dominant ist, obwohl der Weg A → B → D → E → F → H → A mit 58% der häufigste Weg ist (dieser Weg entspricht dem Fall, dass ein Buchstabe nicht der erste oder der letzte Buchstabe in einem Wort ist, wodurch die einzig geforderte Aktion im Erhöhen der Anzahl der gelesenen Character besteht). Immerhin müssen für diese Aktion in Bild 6.8a 9 Instruktionen, darunter 5 Branchbefehle durchlaufen werden.

```
 1  A:  add    r6,r6,-1
 2      blt    r6,0,C
 3  B:  load   r4,mem(r7+0)
 4      add    r7,r7,1
 5  D:  beq    r4,-1,EXIT
 6  E:  add    r1,r1,1
 7      bge    32,r4,G
 8  F:  bge    r4,127,G
 9  H:  bne    0,r8,A
10  K:  add    r2,r2,1
11      add    r8,r8,1
12      jmp    A
13  G:  beq    r4,10,I
14  J:  bne    r4,32,L
15  M:  mov    r8,0
16      jmp    A
17  I:  add    r3,r3,1
18      jmp    M
19  L:  bne    r4,9,A
20      jmp    M
21  C:  store  mem(r5+0),r6
22      store  mem(r5+4),r7
23      mov    Parm0,r5
24      jsr    filbuf
25      mov    r4,Ret0
26      load   r6,mem(r5+0)
27      load   r7,mem(r5+4)
28      jmp    D
```

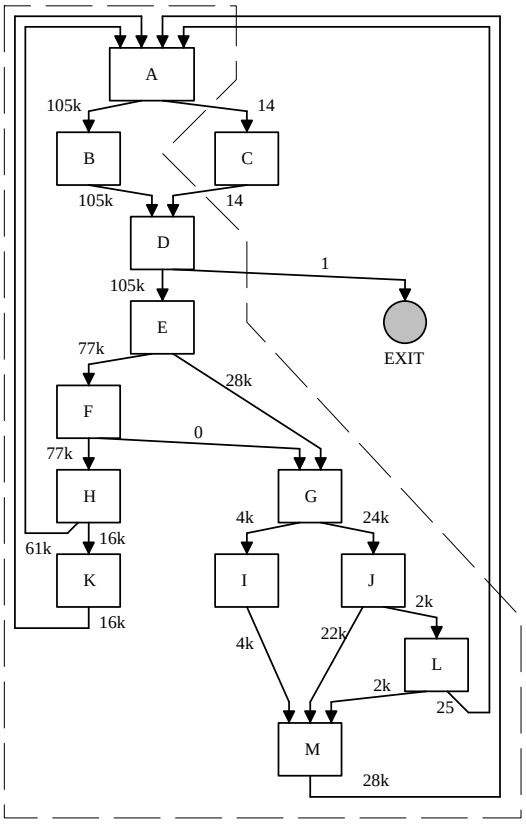


Bild 6.8: Assemblercode (a) und Kontrollflussgraph (b) für wc-Routine

Hierbei wird für Instruction Level Parallelism die Problematik sichtbar: Branches müssen richtig vorhergesagt werden, um den Instruktionsfluss parallel abarbeiten zu können. Bei dem Programm kommen nun zwei Effekte zum Tragen, die eine Bearbeitung mit hoher Performance verhindern:

- Die Vorhersage eines bedingten Sprunges muss zwar möglichst akkurat durchgeführt werden, die Problemstellung selbst verhindert diese Möglichkeit jedoch, wie aus den Wahrscheinlichkeiten in Bild 6.8b hervorgeht.
- Die Möglichkeit von superskalaren Prozessoren zur Vorhersage und Ausführung von Branches sind zumeist sehr begrenzt. Die Vorhersage bezieht sich auf eine gewisse Tiefe, die Ausführung dabei auf das Commitment der ausgeführten Befehle. Meist ist die parallele Ausführung von 4 Instruktionen, darunter nur eine Verzweigung, möglich.

Die reine Hardwarelösung, vom Prozessor die Durchführung mehrerer Branchbefehle zu fordern, bedeutet nichts anderes, als dass eine Mehrfachvorhersage existieren muss sowie ein komplexes Entscheidungsnetzwerk, bis zu welchem Teil die Vorhersage gültig war. Diese Hardware könnte die Taktfrequenz des Prozessors begrenzen – eine sicher negative Nebenwirkung.

Optimal ist jedoch die Verhinderung von bedingten Branchbefehlen, da bedingte Moves usw. wesentlich besser parallel zu anderen Befehlen ausführbar sind. Zwei Verfahren zur Einführung von *Predicated Instructions* bieten sich an: Compilierung für Predicate-Befehle sowie einzelne Optimierungen.

Zum Zweck der Compilierung wird ein Hyperblocknetzwerk eingeführt: Ein *Hyperblock* besteht aus mehreren Basisblöcken, zwischen denen Übergänge mittels Kontrollfluss existieren, die nur einen Startblock haben (der gesamte Hyperblock kann also nur über einen Zugang erreicht werden), jedoch an mehreren Stellen verlassen werden können.

Dieser Hyperblock wird nunmehr so übersetzt, dass mit Hilfe der *if-Conversion* alle Übergänge zwischen den Basisblöcken durch bedingte Befehle ersetzt werden. Ausnahmen hiervon bilden die Sprünge nach außerhalb, also zum `EXIT` des Hyperblocks.

Theoretisch sicher ein gutes Verfahren, in der Praxis werden sich jedoch andere Problemfelder bieten: Die Formierung von langen Blöcken kann neue Hazards entstehen lassen, die Anzahl der benötigten Register kann bei paralleler Ausführung zu groß werden, oder die Länge des kritischen Codes, nach dem ein sicherer Berechnungszustand wiederhergestellt werden kann, kann groß werden. Dies sind zwar eigentlich gewollte Effekte (zumindest zum Teil), andererseits ist alleine durch die Compilierung in Richtung dieser neuen Befehle nur ein Teil gewonnen, ein anderer eventuell verloren. Der Compiler muss daher Hyperblock-Synthese, Registerbenutzung, Hazardvermeidung und Kritische Pfadlänge miteinander ausbalancieren.

In Bild 6.9 ist der gestrichelt gezeichnete Teil in einen Code mit Nutzung der *Predicated Instructions* umgesetzt worden, und zwar in a) mit einfachen pxx-Befehlen (handoptimiert, → 6.2.1), b) mit doppelten pxx-Befehlen, die auf zwar

Vorhersageflags wirken und diese ggf. mit den vorhergehenden Inhalten verknüpfen. Der allgemeine Befehlstyp für das Listing in Abb. 4.9b sieht wie folgt aus:

```
p<cmp> Pout1(<type>), Pout2(<type>), src1, src2 (Pin);
```

```
                                          Takt
 1  A:  add    r6,r6,-1                    0
 2      blt    r6,0,C                      1
 3      load   r4,mem(r7+0)                1
 4      add    r7,r7,1                     1
 5      beq    r4,-1,EXIT                  3
 6      add    r1,r1,1                     3
 7      pge    p4,32,r4                    3
 8      pgep   p4,r4,127      (/p4)  4
 9      pnep   p3,0,r8        (/p4)  5
10      addp   r2,r2,1        (p3)   6
11      addp   r8,r8,1        (p3)   6
13      peqp   p6,r4,10       (p4)   6
17      addp   r3,r3,1        (p6)   7
14a     pnep   p8,r4,32       (/p6)  7
14b     palp   p8             (p6)   8
14c     pnep   p6,r4,32       (/p6)  8
19      peqp   p6,r4,9        (p8)   9
15      movp   r8,0           (p6)  10
16      jmp    A                     10
```

a)

```
                                                    Takt
 0      pclr   p4,p6                                0
 1  A:  add    r6,r6,-1                             0
 2      blt    r6,0,C                               1
 3      load   r4,mem(r7+0)                         1
 4      add    r7,r7,1                              1
 5      beq    r4,-1,EXIT                           3
 6      add    r1,r1,1                              3
 7      pge    p4(OR),p1(/U),32,r4                  3
 8      pge    p4(OR),p2(/U),r4,127        (p1)     4
 9      pne    p3(/U),-,0,r8               (p2)     5
10      add    r2,r2,1                     (p3)     6
11      add    r8,r8,1                     (p3)     6
13      peq    p7(U),-,r4,10               (p4)     6
17      add    r3,r3,1                     (p7)     7
13'     peq    p6(U),p5(/U),r4,10          (p4)     7
14      pne    p6(/OR),p8(U),r4,32         (p5)     8
19      peq    p6(OR),-,r4,9               (p8)     9
15      mov    r8,0                        (p6)    10
16      jmp    A                                   10
```

Registerinhalte:

r1 = charct	r5 = fp
r2 = wordct	r6 = fp->cnt
r3 = linect	r7 = fp->ptr
r4 = c	r8 = token

Zugehörigkeit der Flags:

p1 ~ F	p5 ~ J
p2 ~ H	p6 ~ M
p3 ~ G	p7 ~ I
p4 ~ K	p8 ~ L

b)

Bild 6.9: Code mit bedingten Befehlen, a) einfache Ausführung, b) komplexe pxx-Befehle

Dieser Befehl besitzt also zwei Zielflags, die via Verknüpfung in <type> (z.B.: U entspricht Unconditional, OR dem ODER usw.) aus dem Vergleich von src1 und src2 entstehen. Dieser Befehl selbst ist wieder konditionierbar über Pin.

Das Ergebnis besteht in dem Verbleiben von zwei Branch-Befehlen, einmal auf den Abschnitt C, auf den die Abschnitte D bis M dann als Codekopie folgen, zum anderen als `EXIT`-Sprung. Bei Verwendung eines 2-Bit-Zähler für die Branchvorhersage wird in dem Beispiel bei 14 Einsprüngen in den C-Teil eine maximale Fehlvorhersage von 56 möglich sein (vorher: 52 k)!

Einer der positiven Nebeneffekte der Formung solcher Hyperblocks liegt in der Generierung von mehr Möglichkeiten zur klassischen Optimierung durch *Compiler-Instruction-Level-Parallelising-Techniken*, die den Code nach parallelisierbaren Instruktionen umsortieren, die Nutzung von gemeinsamen Unterausdrücken (*Common Subexpression Elimination*), können im Rahmen der möglichen Ressourcen sehr gut ausgeführt werden. Das vorhergehende Beispiel zeigt sich jedoch

deshalb so freundlich, weil ein Block gefunden werden kann, der sehr dominant ausgeführt wird und zu einem Hyperblock zusammenfassbar ist.

Bei anderen Verzweigungen und Schleifen lässt sich dies nicht so einfach bestimmen, falls nicht die Möglichkeiten der superskalaren CPU überstiegen werden sollen. Die Formung eines Hyperblocks ist zwar immer möglich, dieser muss jedoch im Rahmen der CPU-Ressourcen ausführbar sein, da ansonsten die gewünschten Effekte eher ausbleiben. Z.B. kann eine Vielzahl von Austrittspunkten bereits die Branchvorhersage wieder sehr in Anspruch nehmen, so dass wiederum kein optimaler Code durch fehlerhafte Vorhersagen entsteht.

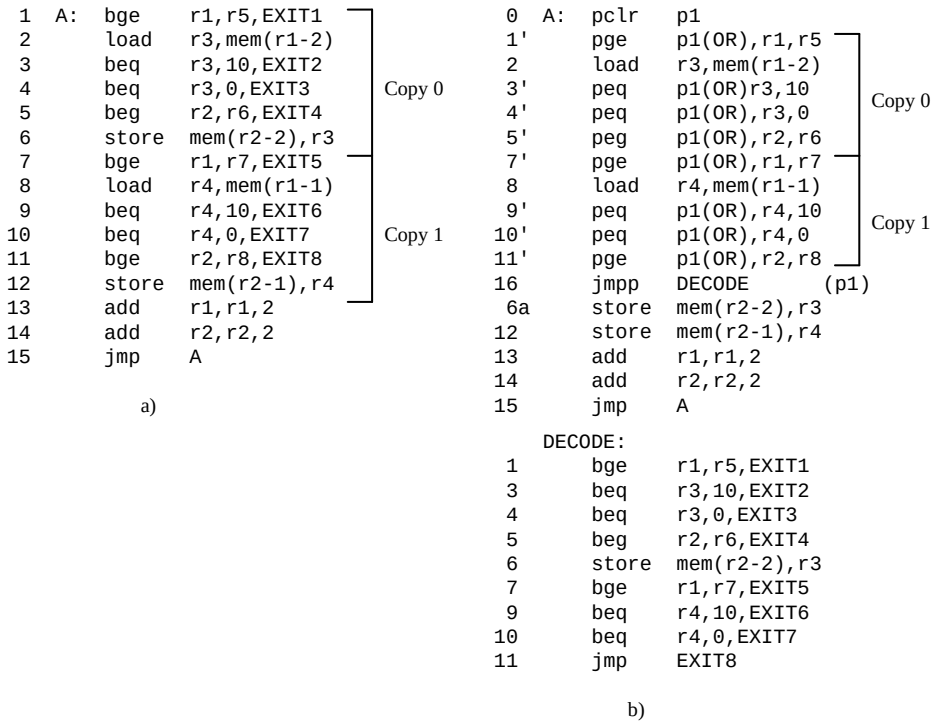


Bild 6.10: Assemblercode für grep a) klassisch, b) mit Branch Combining

Das nächste Beispiel zeigt die übersetzte grep-Routine, in der eine Menge an Verzweigungen zum `EXIT` auffallen. Bei 15 Befehlen sind immerhin 8 Branches und ein Sprungbefehl vorhanden, sodass bei Ausführung von 1 Branchbefehl pro Takt nur noch eine theoretische Performance von 1,67 Instr./Takt möglich wäre.

Die diversen `EXIT`-Branches können nun zusammengefasst werden, indem pxx-Befehle das Flag `p1` bestimmen und der bedingte Sprungbefehl `jmp (p1)` durchgeführt wird. Diese Technik wird *Branch Combining* genannt und bietet sich an,

wenn sehr viele Austrittspunkte aus einem Block existieren. Das Beispiel in Bild 6.10 ist natürlich deshalb sehr gut geeignet, da sich die gesamten Branches durch eine Kette von Vorhersagevergleiche mit OR-Kombination überführen lassen. Die Branches werden im Fall des Austritts aus der Schleife zwar nochmals durchlaufen, der Schleifendurchlauf ist aber stark beschleunigt.

Es existieren noch weitere Methoden zur Optimierung durch *Predicated Instructions*, die sich jedoch außerhalb des Bereichs dieser Vorlesung bewegen. Eine experimentelle Evaluation der bisherigen Ergebnisse zur Optimierung ergibt folgendes Bild:

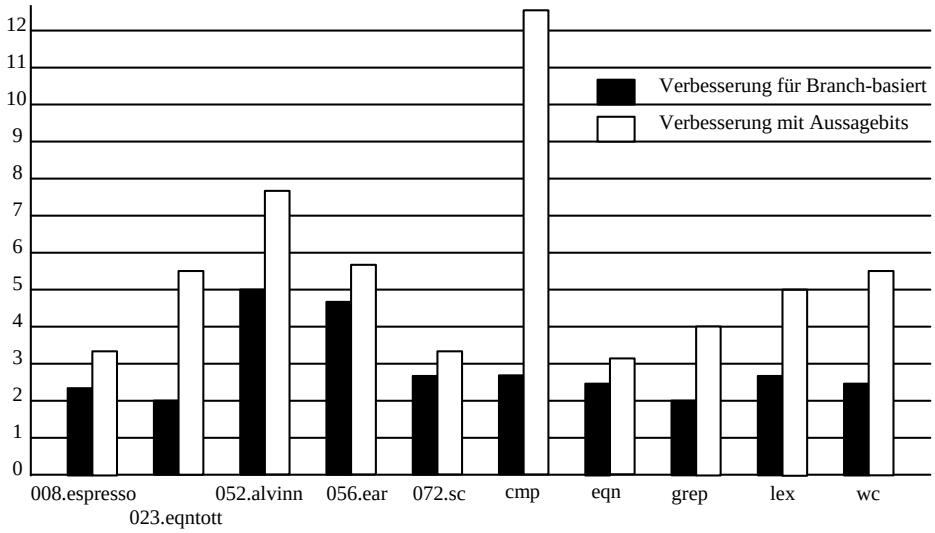


Bild 6.11: Performanceverbesserung für 8-fach superskalare CPU durch bedingte Befehle mit Aussageflags

Die Herkunft der getesteten Programme: 5 entstammen Benchmarks (SPEC CFP92: 052.alvinn, 056.ear, SPEC CINT92: 008.espresso, 023.eqntott, 072.sc), die anderen stellen gebräuchliche UNIX-Utilities dar.

6.2.4 Generierung von Aussagen zu Speicherabhängigkeiten

Die zweite Optimierungsmethode, die Umsortierung von Instruktionen zur besseren Nutzung der parallelen Ressourcen, benötigt Aussagen zu Abhängigkeiten der Instruktionen untereinander. Während Registerabhängigkeiten leicht zu detektieren sind (wenn auf eine indirekte Adressierung der Register verzichtet wird, was

praktisch immer der Fall ist), gilt dies für Speicherzugriffe keinesfalls: Letztere sind teilweise schwer zu detektieren.

Gewöhnlich konzentriert sich die Analyse von Datenabhängigkeiten auf die Ebene des Sourcecodes. Folgende Maßnahmen sind dabei relativ einfach und in vielen Compilern vorhanden:

- Detektierung der Unabhängigkeit bei Zugriff auf einzelne, globale Variablen
- Detektierung der Unabhängigkeit bei Zugriff auf einzelne Stackvariablen (an unterschiedlichen Stackpositionen)

Die interessante Frage ist jedoch diejenige, wie Zugriffe innerhalb eines Arrays voneinander abhängig sind. Hierbei entstehen zwei Fragen: Wie können Zugriffe auf Mitglieder des Arrays, indiziert durch andere Variablen, auf Abhängigkeit untereinander untersucht werden, und wie können (insbesondere in C genutzte) Aliasnamen auf Pointer bzw. Arraymitglieder hierbei berücksichtigt werden.

Letztere Frage bedeutet, dass die Analyse insbesondere über Funktionsgrenzen hinweg durchgeführt werden muss, sie wird global. Ferner wird auch deutlich, dass die Codeanalyse sehr zeitaufwendig werden kann. Aus diesem Grund wird in einem Compiler versucht, die Analyse so früh wie möglich (entsprechend so unaufwendig und so akkurat wie möglich) zu machen und die gewonnenen Informationen weiterzutragen. Bild 6.12 zeigt das prinzipielle Vorgehen dieser Methode der frühen Sourcecodeanalyse.

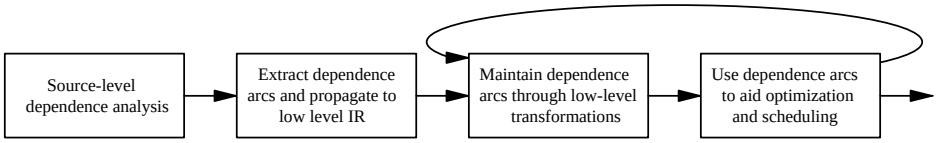


Bild 6.12: Übersicht zur Generierung und Pflege der Abhängigkeitsinformationen

Im Unterschied zum klassischen Ansatz, mit Hilfe einer **Sourcecodeanalyse** eine **Sourcecodetransformation** durchzuführen, zeigt diese Abbildung, dass die gewonnenen Sourcecodeinformationen zu einer **Low Level Transformation** genutzt werden!

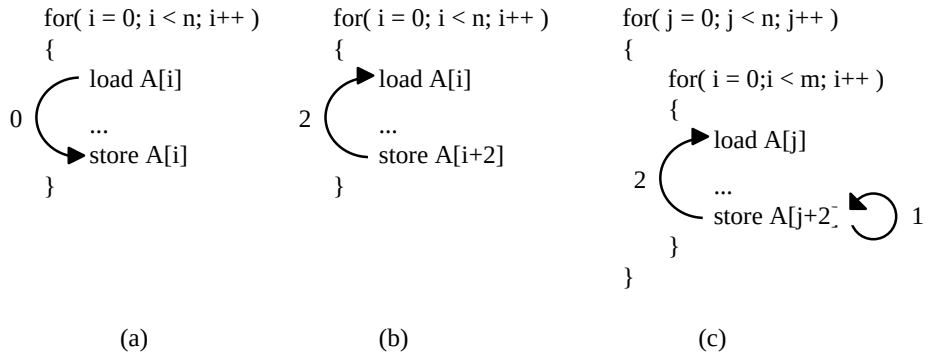


Bild 6.13: Beispiele zum Codescheduling: a) Schleifen-unabhängige Abhängigkeit
b) schleifenabhängige Abhängigkeit
c) Schleifen-getragene Abhängigkeit (Outer-loop-carried)

Bild 6.13 zeigt drei verschiedene Abhängigkeiten im Code. Hieran kann bereits gezeigt werden, dass das **azyklische Codescheduling** nur dann parallelisiert bzw. in der Reihenfolge vertauscht werden kann, wenn keine Schleifen-unabhängigen (loop-independent) Abhängigkeiten existieren. Dies ist in Bild 6.13a nicht der Fall, da beide Zugriffe definitiv auf die gleiche Speicherstelle zugreifen; dies erfolgt innerhalb jeder Schleife! Im Übrigen darf in keinem Fall eine Datenabhängigkeit möglich sein, um ein *Reordering* durchzuführen; diese Aussage kann ggf. zur Compilezeit unmöglich sein.

Bei schleifenabhängiger Datenabhängigkeit wie in Bild 6.13b ist die Anzahl der Schleifen, die Zugriffe auf die gleiche Speicherstelle erzeugen, auch als *Abhängigkeitsdistanz* (dependence distance) bezeichnet, relevant. Im obigen Beispiel sind dies 2 Schleifen, so daß zwei Schleifen bequem zusammengefasst und im Code umgestellt werden können (bei Abwesenheit weiterer Abhängigkeiten, die ggf. Schleifen-unabhängig sein können). Wie in Bild 6.13c dargestellt, ist dabei die tragende Schleife relevant: Für die innere Schleife (Index i) besteht eine Abhängigkeitsdistanz von 1, da die Schreib- und Lesevorgänge immer auf dieselbe Stelle zeigen, während in der äußeren Schleife die Distanz 2 beträgt. Beim Codescheduling muss jeweils nur die momentane Schleife berücksichtigt werden, also die zyklischen und nichtzyklische Abhängigkeiten innerhalb dieser.

Die für eine Optimierung notwendigen Abhängigkeitsnotationen sind in Bild 6.14 dargestellt. Es müssen natürlich sichere Informationen zur Codeoptimierung existieren, oder der Compiler muss die Annahme der unlösbaren Abhängigkeit treffen. Bild 6.15 zeigt ein Beispiel für eine einfache, aber wirksame Optimierung: Redundante Ladeoperationen werden vermieden.

Kategorie	*Mögliche Werte*
type	flow, anti, output, input
distance	(integer), unknown
carrying loop	none, (loop identifier)
certainty	definite, maybe

Bild 6.14: Tabelle zu benötigten Abhängigkeitsinformationen

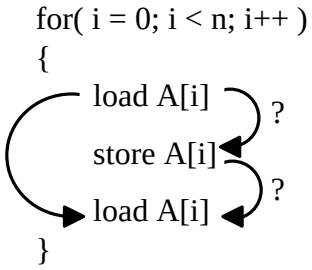


Bild 6.15: Vermeidung redundanter Ladeoperationen

Zwei Bedingungen müssen für die Eliminierung der Load-Instruktion erfüllt sein: Es muss eine definitiv Schleifen-unabhängige Speicherabhängigkeit sein (*i* darf sich nicht ändern) und es darf keine Veränderung des Speicherinhalts geben, d.h., alle Speicheroperationen zwischen den Load-Befehlen müssen definitiv auf andere Stellen zeigen.

Die Optimierungsmöglichkeiten durch Reordering des Codes werden – soviel sollte aus dem bisher dargestellten Überlegungen deutlich geworden sein – von der Akkuratheit der Abhängigkeitsanalyse begrenzt. Da im Sourcecode eine Analyse noch am Besten durchführbar ist, werden meistens die Ergebnisse dort gewonnen und müssen nun auf das Assemblerniveau mit allen Codetransformationen mitgeführt werden.

Die *Pflege der Abhängigkeitsdarstellungen* ist ein wichtiges Thema im Optimierungsteil von Compilern. Die Abhängigkeitspfeile innerhalb eines Schleifenkörpers sind natürlich trivial bei Codeumstellungen nachzuführen. Bei ausschließlich schleifenabhängiger Datenabhängigkeit bewirkt ein *Reordering* der Befehle beispielsweise keine Änderung der Abhängigkeitsdistanz und somit keine Änderung der Verhältnisse.

Wird allerdings die Schleifenstruktur geändert, ist die Nachführung der Abhängigkeiten weniger trivial. Eine der wichtigsten Änderungen ist hierbei das Loop Unrolling (→ 6.2.2). Bei Nutzung der Abhängigkeitsdistanz kann der Compiler durchaus größere Schleifen gewinnen und den Code umsortieren!

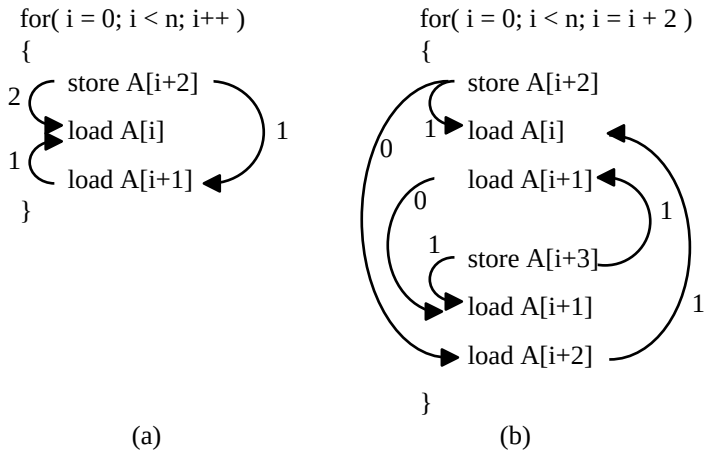


Bild 6.16: Beispiel zum Loop Unrolling: a) Originalcode, b) neuer Code

Bild 6.16 zeigt ein Beispiel zum Loop Unrolling und zu den resultierenden Abhängigkeiten im Code. Durch die Nutzung der Abhängigkeitsdistanz (hier: 2) in der Originalschleife können die neuen Abhängigkeitsdistanzen bestimmt werden.

Hierzu geht der Compiler von dem ersten Befehl mit Abhängigkeitsinformation aus und bestimmt zwei Größen für jeden anderen Befehl: Die Schleifennummer der Kopie, in der sich der Befehl befindet, und die neue Abhängigkeitsdistanz. Dies erfolgt gemäß der Formeln

$$\mathrm{Copy}_{\mathrm{dest}} = (\mathrm{Copy}_{\mathrm{src}} + \mathrm{Dist}_{\mathrm{old}}) \bmod n$$

und

$$\mathrm{Dist}_{\mathrm{new}} = (\mathrm{Copy}_{\mathrm{src}} + \mathrm{Dist}_{\mathrm{old}}) \operatorname{div} n$$

wobei $\mathrm{Copy}_{\mathrm{Src}}$ die Kopienummer des Ausgangspunkts für die Abhängigkeit, $\mathrm{Copy}_{\mathrm{Dest}}$ die Kopienummer für das Ziel, $\mathrm{Dist}_{\mathrm{Old}}$ die bisherige Distanz und $\mathrm{Dist}_{\mathrm{New}}$ die neue Distanz darstellen. *n* bezeichnet die Anzahl der Kopien, im obigen Beispiel 2.

Falls der Compiler die Distanzinformationen nicht nutzen kann, muß er hier immer 1 annehmen (im Fall 0 handelt es sich um Schleifen-unabhängige Informationen). Damit sind ggf. größere Reordering-Maßnahmen des Compilers verbaut, daher sind die Distanzinformationen sehr wichtig für durchgreifende Optimierungen.

Bild 6.17 zeigt abschließend die Wirkung der Datenabhängigkeitsanalyse mit anschließendem Code-Reordering. Die Programme benutzen alle Integer-Arithmetik und wurden auf einer superskalaren CPU mit 8facher Parallelität zum Ablauf gebracht. Eine Beschleunigung von 20% über alle Programme mit signifikanten Ausreißern nach oben kann hierbei gezeigt werden.

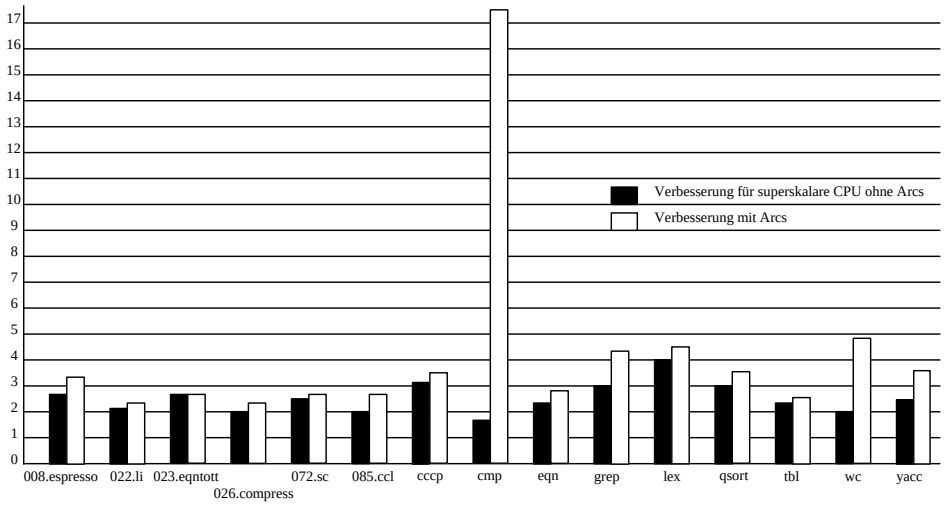


Bild 6.17: Integer Benchmark: Beschleunigung durch superskalare CPU mit 8facher Parallelität, a) ohne Analyse, b) mit Analyse.

Die Testprogramme entstammen wiederum den SPEC CINT92 Benchmarks bzw. stellen UNIX-Utilities dar.

7 Very-Long-Instruction-Word- (VLIW-) Architekturen

7.1 Allgemeines zu VLIW-Architekturen

VLIW-Architekturen sind durch ein breites Befehlswort, das in Felder eingeteilt ist, aus denen mehrere Funktionseinheiten unabhängig voneinander gesteuert werden können, gekennzeichnet. Ein zentrales Leitwerk holt in jedem Taktzyklus einen breiten Befehl aus dem Befehlsspeicher und stößt die im Befehlswort codierten Operationen für die einzelnen Funktionseinheiten zur Ausführung an. Die einzelnen Funktionseinheiten sind über eine zentrale Registerdatei miteinander verbunden. Jede Funktionseinheit hat zwei Leseeingänge und einen Schreibausgang. Die Registerdatei wird mit Daten aus dem Datenspeicher versorgt, was bei einer großen Anzahl von Registern eine hohe Speicherbandbreite erfordert.

Diese Architekturbeschreibung zeigt die wesentlichen Mechanismen der VLIW-Architekturen: Die Teilbefehle sind ähnlich zu denen der RISC-Architektur aufgebaut, die 3-Adress-Registerstruktur der Befehle ist direkt nachzuvollziehen. Direktdaten (Programmkonstanten) können im Befehlsfeld eingetragen werden. Die einzigen Operationen, die auf den Speicher zugreifen können, sind Load/Store-Operationen.

Die Kontrollflussstrategie entspricht ebenfalls der der RISC-Prozessoren: Änderungen des sequenziellen Programmflusses erfolgen durch Sprünge, Verzweigungen und Unterprogrammaufrufe. Hierdurch entsteht die gleiche Systematik wie bei superskalaren CPUs, da die Programmflusskontrolle in ihrer expliziten Ausführung durch Flags (1-Bit-Register) beeinflusst wird, die ihrerseits ihren Wahrheitswert erst durch Vergleichs-, ggf. auch andere Operationen erhalten.

Im Unterschied zu superskalaren Prozessoren erfolgt die Steuerung des (sequenziellen) Programmflusses (d.h. das *Instruction Scheduling*) in wesentlichen Teilen zur Compilezeit! In jedem (Lang-)Befehl sind die Operationen codiert, die dann in einem Taktzyklus auf den voneinander unabhängigen Funktionseinheiten auszuführen sind.

Der Compiler für die VLIW-Architektur hat also die Aufgabe, aus einem zunächst sequenziellen Strom von Operationen mit Hilfe einer genauen Analyse des Steuer- und des Datenflusses die voneinander unabhängigen Operationen herauszufinden. Die als gleichzeitig innerhalb eines Takts ausführbar analysierten Einzelbefehle werden dann in die breiten Befehlswörter gepackt. Hierfür ist wesentlich, dass die Ausführungszeit eines jeden Befehls zur Übersetzungszeit bekannt ist.

Die Einschränkung, dass die Steuerung ’nur’ im Wesentlichen zur Compilezeit erfolgt, also zu einem geringeren Anteil auch zur Laufzeit möglich ist, ist natürlich

keine echte Einschränkung. Das Instruction Scheduling zur Laufzeit kann ggf. mehr Laufzeitoptimierung beinhalten, also Informationen verarbeiten, die der Compiler nicht haben kann (z.B. spekulative Schlussfolgerungen aus dem bisherigen Programmverlauf. Aus diesem Grund wird allgemein davon ausgegangen, dass auch zur Laufzeit noch ein zusätzliches Scheduling erfolgen kann.

Eine Erhöhung der Instruktionsparallelität wird durch die *bedingte Ausführung von Instruktionen* erreicht: Wie im Fall der superskalaren CPU werden möglichst viele Befehle mit einem zuordnungsfähigen Bedingungsflag ausgestattet, so dass bei Erfüllung der Bedingung diese Instruktion ausgeführt wird. Die Zeit zur Auswertung des Flags wird zur Gesamtausführungszeit des Befehls hinzugerechnet, es ist dabei angestrebt, alles innerhalb eines Takts ablaufen zu lassen. Eine Operation, die bedingt codiert ist und deren Bedingung nicht zutrifft, verhält sich wie ein NOP-Befehl (No Operation).

Die Herkunft der Bedingungsflags kann dem normalen CPU-Modell entsprechen (also Zero, Carry usw.), oder es werden neue, zusätzliche Flags zur Verfügung gestellt.

7.1.1 Compilerstrategien

Die praktisch erreichbare Performance einer VLIW-Architektur hängt sehr wesentlichen von der Parallelisierungsstrategie des Compilers ab. Während in der superskalaren CPU zum Zeitpunkt der Compilierung viele Aktionen ablaufen können, die die Parallelisierungsfähigkeit des Programms erhöhen, muss der VLIW-Compiler die Befehle zusätzlich noch sortieren und zusammenfassen.

Nach einer sorgfältigen Analyse und einem Umbau des Programms (Ersatz der Branch-Befehle durch bedingte Ausführung von Befehlen, Loop Unrolling usw.) wird das Programm anhand des *Percolation Scheduling Algorithmus* nach A. Nicolau (1985) parallelisiert und die langen Befehlswörter zusammengesetzt. Hierbei wird ein *paralleler Flussgraph* gewählt, dessen Knoten die gleichzeitig in einem Taktzyklus ausführbaren Operationen enthalten können und dessen (gerichtete) Kanten den Steuerfluss im Programm anzeigen.

Hierfür existieren im Wesentlichen drei Kerntransformationen, die *move-op* (zum Verschieben von Operationen zum Vorgänger hin), *move-cj* (zum Verschieben von Verzweigungen zum Vorgänger hin) und *delete* (zum Löschen von inzwischen leeren Knoten) angewendet werden können. Die *Strategie* dieser Anwendung hingegen ist in verschiedener Weise zu wählen, um ein Optimum an Performance zu erreichen, hierin unterscheiden sich sogar einzelne Applikationen voneinander.

```
int a, b;

if( a > 0 )
   b = a;
else
   b = 0;
```

(a)

```
mov  r0, a;
mov  r1, b;
pgt  r0, 0, p1;  Setzen von p1
movp r2, r0, p1; Nur für a > 0
movp r2, 0, /p1; Nur für a <= 0
mov  b, r2;
```

(b)

Bild 7.1 C-Sourcecode und Assemblerübersetzung mit Branch-Ersatz

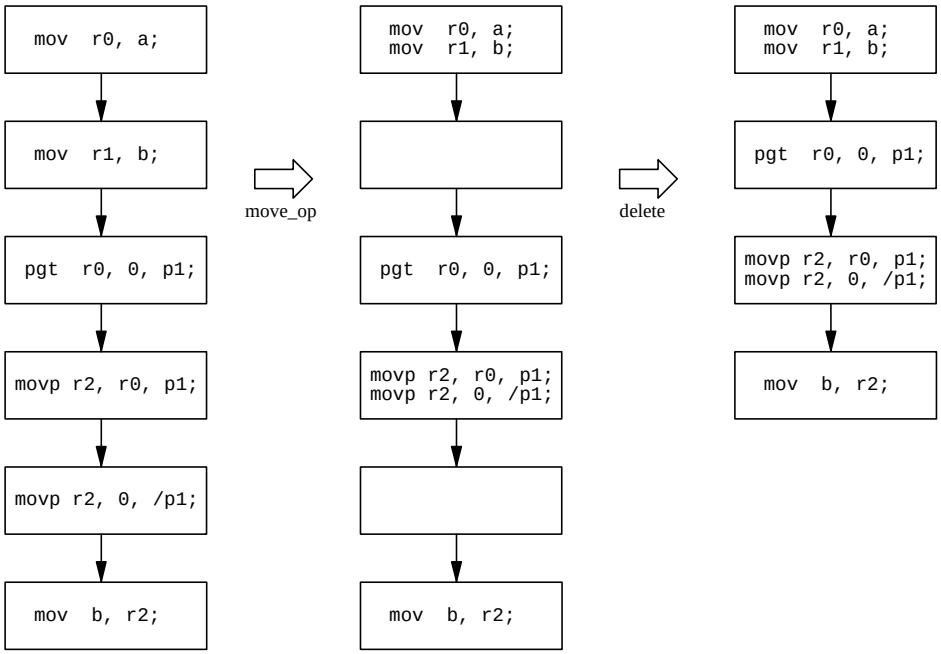


Bild 7.2 Percolation Scheduling Algorithmus

7.1.2 Zusammenfassung

Die VLIW-Architekturen zeichnen sich im Wesentlichen dadurch aus, dass die Parallelität der Befehlsausführung bereits zur Compilezeit festgeschrieben werden kann, dies im Unterschied zur superskalaren CPU.

Performancegewinne um Größenordnungen sind hierdurch kaum vorstellbar. Die Analyse des Objektcodes kann zwar wesentlich detaillierter und vor allem weitreichender durchgeführt werden, prinzipiell gibt es jedoch keine Unterschiede zwischen superskalarer CPU mit dynamischem Scheduling und VLIW-Architektur mit der Möglichkeit zum statischen Scheduling durch den Compiler.

Der wesentliche Vorteil einer VLIW- gegenüber einer superskalaren Architektur besteht darin, dass ein wesentlicher Teil der Hardware fortfallen kann. Das Data Forwarding innerhalb der VLIW-CPU (→ 4.4.1) muss in jedem Fall erhalten bleiben: Ein Fortfall ergäbe zwar drastische Einsparungen in der Hardware – es wird davon gesprochen, dass dieses Data Forwarding bis zum vierfachen Platz einer ALU einnehmen kann –, allerdings wären die Einschränkungen für den Compiler sehr drastisch, da alle Datenabhängigkeiten über viele Takte beachtet werden müssten.

Die einzusparenden Teile betreffen das Register Renaming, das Instruction Dispatching und Instruction Issueing (→ 5.3). Diese Teile können ganz oder in weiten Teilen entfallen, da der VLIW-Compiler die Arbeit übernimmt.

Der wesentliche Nachteil liegt in der Codegröße: Die zentrale Ansatz der VLIW-Architekturen besteht ja darin, einen großen Block mit mehreren darin enthaltenen Instruktionen als die atomare Einheit im Code anzusehen. Werden die möglichen Instruktionen nicht genutzt (diese müssen dann durch NOP-Befehle ersetzt werden), wird der real genutzte Code unverhältnismäßig groß codiert. Eine Abhilfe kann hier durch Codierungen geschaffen werden, die keine festgelegten Plätze für bestimmte Instruktionsgruppen besitzen und das Ende des genutzten Codes durch ein bestimmtes ansonsten nicht genutztes Bitmuster markieren (sog. CLIW-, Configurable-Long-Instruction-Word-Architektur).

7.2 Intel IA-64 Architektur

Als die herausragende Architektur, die das VLIW-Prinzip implementiert, kann die IA-64 Architektur von Intel und HP angesehen werden. Die grundlegenden Arbeiten starteten in 1994 [16], und für diese Architektur wurde eigens ein neues Prinzip kreiert, mit EPIC (Explicitly Parallel Instruction Computing) bezeichnet [16, 17, 22, 23]. Mittlerweile existiert bereits die zweite Architekturstufe Intanium 2 [24].

Zunächst muss diese EPIC-Philosophie erläutert werden. Im Anschluss daran werden die Mikroarchitektur, der Aufbau des Prozessorkerns in einer ersten Implementierung (Itanium) sowie Fragen zur Software und zum Programmablauf behandelt.

7.2.1 Explicitly Parallel Instruction Computing

Intel bezeichnet EPIC als neue Philosophie, vergleichbar mit RISC (→ 4) und zugleich Antwort auf den langjährigen Zwist zwischen RISC (Reduced Instruction Set Computing) und CISC (Complex Instruction Set Computing). In der Tat besteht die Hauptaufgabe moderner xx-skalarer Prozessoren (superskalar → 5, multiskalar → 8) aus dem Verteilen der anstehenden Aufgaben auf viele parallel arbeitende Einheiten. Bei den dynamischen Varianten versucht eine sehr komplexe Hardware jedes Mal aufs Neue, die aufeinander folgenden Befehle zu analysieren und mit allen Tricks eine möglichst parallele Ausführung herbeizuführen.

Diese Aufgabe wird in EPIC auf den Compiler abgebildet und ist damit programmstatisch. Dies ist die 'klassische' VLIW-Lösung, worin besteht nun der Unterschied zwischen EPIC und einfachem VLIW?

Intel gibt 3 wesentliche Eigenschaften zu EPIC an. Die EPIC-Architektur bietet

- Mechanismen, die dem Compiler ein Instruction Scheduling auf effiziente Weise ermöglichen,
- genügend Ressourcen wie Register und Funktionseinheiten, um die Operationen parallel zu bearbeiten und die Ergebnisse auch parallel zu speichern sowie
- Möglichkeiten, die Hardware mit weiteren Informationen des Compilers zu versorgen.

Mit anderen Worten: In EPIC sind Compiler und Hardware sehr stark aufeinander abgestimmt, anonsten zeigen sich keine bahnbrechenden Neuerungen. Im Detail hingegen gibt es Neuerungen und nützliche Zusätze, die im Folgenden auch erläutert werden sollen.

7.2.2 Befehlsformat

Die (einfachen) Instruktionen der IA-64 Architektur werden in 41 Bits gespeichert. Zur Codierung werden alle Befehle in verschiedene Klassen eingeteilt, für deren Ausführung wiederum verschiedene Hardwareeinheiten intern zur Verfügung stehen. Bild 7.3 zeigt die Zusammenhänge.

Die Codierung wird für die einzelnen Instruktionstypen unterschiedlich sein, da z.T. sich widersprechende Eigenschaften realisiert werden müssen. Allerdings wurde versucht, die Codierung möglichst einheitlich zu gestalten, was dem RISC-Prinzip entspricht: Jede Instruktion passt komplett mit allen Daten in einen Instruktionsslot.

Bild 7.4 zeigt bereits eine Ausnahme: Die Codierung von 64 bit Immediate Daten. Hier gilt die gleiche Problematik wie bei RISC-Architekturen, da die Codierung der Daten nicht in das Befehlsgrundformat passt. Die Lösung bei IA-64 heißt Extended Format (X) und bedeutet, dass die Zahl auf zwei Instruktionen verteilt

wird. Die Verteilung wird durch das Codierungsformat ”0110“ sowie das Bit 20 (extension bit, auf ’1’) angezeigt.

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I- or M-Unit
I	Non-ALU Integer	I-Unit
M	Memory	M-unit
F	Floating Point	F-Unit
B	Branch	B-Unit
L+X	Extended	I-Unit

Bild 7.3 IA-64 Instruktions- und Ausführungstypen

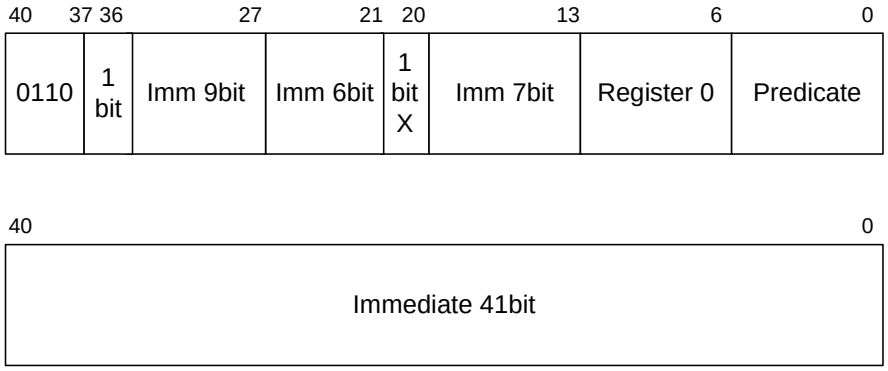


Bild 7.4 IA-64 Instruktionsformat X (Immediate 64 bit) für movl-Instruktion

Bild 7.5a zeigt ein Beispiel für ein allgemeines Instruktionsformat: Je 7 Bits für die beiden Sourceregister und das Zielregister (ergibt je 128 adressierbare Register), 14 Bits für den Operationscode und 6 Bits für die Bedingungen (Predicate-Flags) stehen zur Verfügung

Die Verknüpfung mit Predicate-Flags (Bedingungsflags, → 6.2.3) wurde in der IA-64 Architektur nahezu durchgängig hergestellt, nur sehr wenige Instruktionen sind ausschließlich unbedingt ausführbar. 64 Flags (P0 bis P63) sind vorhanden, die binäre Nummer des Flags, mit dem verknüpft wird, wird in dem Predicate-Feld angegeben. Die Verknüpfung mit P0 (= konstant ’1’) ergibt eine unbedingte Ausführung.

Drei solcher Instruktionen werden in der IA-64 Architektur zu einem Instruktionsbundle zusammengefasst (Bild 7.5b). Diese Zusammenfassung unterliegt strikten Regeln, es sind keineswegs alle Kombinationen erlaubt:

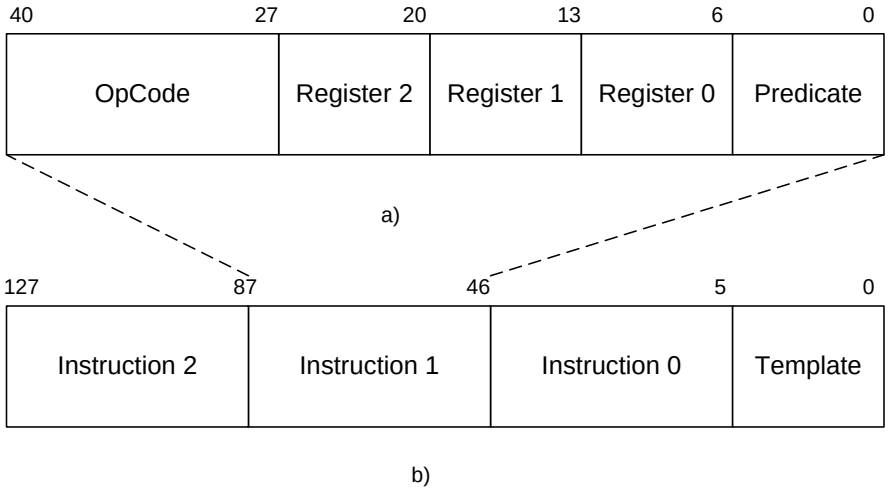


Bild 7.5 IA-64 Bundle- und Instruktionsformat
a) Format für I-Type-Instruktionen b) Bundle-Format

- Die drei Instruktionen müssen voneinander unabhängig ausführbar sein, Datenabhängigkeiten dürfen nicht vorkommen. Dies bedeutet, dass das Übersetzungsprogramm (Compiler oder der Assemblerübersetzer) dies prüfen muss. Im Falle des Compilers wird natürlich gefordert, dass die Reihenfolge der Basisinstruktionen im Instruction Scheduling optimal ausgeführt sein muss, also nicht nur prüft, sondern die optimale Reihenfolge herstellt.
- Nicht jeder Instruktionsslot innerhalb des Instruktionsbündel darf beliebig besetzt werden, derzeit sind insgesamt 24 Kombinationen erlaubt. Welche Kombination in dem aktuellen Instruktionsbündel vorgefunden wird, wird in dem Template mittels 5 Bits (= 32 mögliche Kombinationen) codiert.

Im Assembler ist es möglich, dass der/die SoftwareentwicklerIn die Instruktionsbündel selbst definiert, also zusammenfasst. Dies ist mit der Gefahr einer Datenabhängigkeit verbunden [17]: In diesem Fall wird der Prozessor eine Ausnahme auslösen, da für ihn das Programm nicht mehr korrekt ausführbar ist (Anmerkung: Die Reihenfolge der Instruktionen in den Slots sagt nicht über die Reihenfolge in dem ursprünglichen Programm aus).

7.2.3 Registersatz

Im vorangegangenen Text wurde es schon deutlich, dass die IA-64 Architektur einen erheblichen Satz von Registern zur Verfügung stellt. Bild 7.6 zeigt den Aufbau und die Anzahl für die Daten-Register.

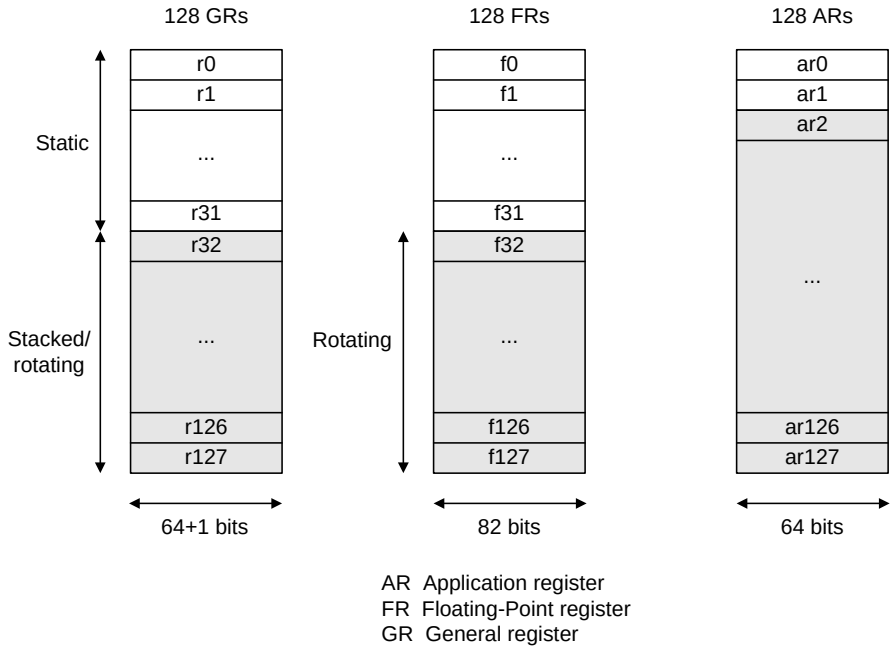


Bild 7.6 Registersatz IA-64, Datenregister [23]

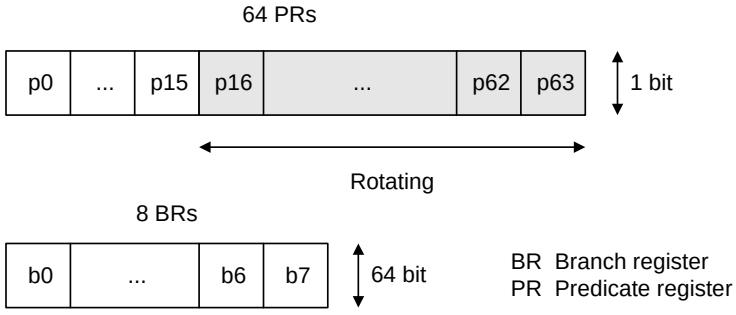


Bild 7.7 Registersatz IA-64, Kontrollflussregister [23]

IA-64 bietet nun mit 128 Allzweck-Registern (r0...r127) 16-mal mehr Register als IA-32 an, die zudem alle (64+1)-bittig sind. Das neben den 64 Datenbits vorhandene 65. Bit trägt den Namen *Not a Thing* (NaT). Diese 128 Register (von denen das erste, r0, dauerhaft null ist) teilen sich in zwei Bereiche: die ersten 32 sind statisch und die restlichen 'stacked', diese werden dynamisch verwaltet.

Man kann sich das so wie in einem Hochsprachenprogramm mit globalen und lokalen Variablen vorstellen. Der Compiler gibt bei einer Unterroutine an, wie viele Register er für Ein- und Ausgabe und lokale Berechnungen benötigt. Das vermeidet ein Großteil der langwierigen Push- und Pop-Operationen, wie sie bei Unterprogrammaufrufen fast aller Prozessorarchitekturen (insbesondere bei IA-32, der Standard-Intel-Architektur) an der Tagesordnung sind.

Der IA-64-Prozessor kann immerhin aus seinem Pool von 96 Registern schöpfen. Und wenn die Software damit nicht auskommt, sorgt eine so genannte Register Stack Engine (RSE) dafür, dass 'ältere' Register aus höher liegenden Prozeduren in den Speicher (beziehungsweise Cache) ausgelagert werden. Den eigentlichen Körper einer Subroutine kann man dann schön übersichtlich mit den Registernamen in0, in1 ... loc0, loc1 ... out0, out1 gestalten, die auf die Register r32...r127 gemappt sind. Register-Mapping hat allerdings den Nachteil einer zusätzlichen Pipeline-Stufe für das Renaming (→ 5.3.2).

Lokale Register bieten zusätzlich die Möglichkeit einer Parallelverarbeitung durch Registerrotation. Die statische Variante, das sogenannte Loop Unrolling, war in 6.2.2 vorgestellt worden: Hier werden mehrere Schleifendurchgänge zu einem einzigen zusammengefasst, die Register umbenannt (Compiletime Register Renaming) und die Instruktionen neu sortiert. Im günstigen Fall können so auch Instruktionen entfallen (z.B. bei Load-Befehlen), durch die Zusammenfassung und das Neusortieren werden aber vor allem Laufzeitgewinne (Vermeidung von Verzweigungen etc.) erzielt.

Im dynamischen Fall könnten mehrere Schleifendurchläufe gleichzeitig gestartet werden, wenn entsprechende Ressourcen frei wären und natürlich eine Unabhängigkeit der Daten gewährleistet ist. Diese Variante wird als einer der Ansatzpunkte für ein sog. Microthreading (→ 8) gesehen, wenn versucht wird, die Ausführung eines einzelnen Programms (oder besser: Threads) dynamisch zu parallelisieren, und zwar nicht nur auf Instruktionsebene.

Bei der IA-64 Architektur liegt sozusagen ein mittlerer Falll vor: Wenn in kleinen Schleifen die zu berechnenden Daten voneinander unabhängig sind, kann IA-64 schon mit der Berechnung der nächsten Iterationen beginnen, während die erste noch im Gange ist. Er mappt dann einfach bei jedem Durchlauf die beteiligten Register auf andere physische Register - sofern noch welche frei sind. Dies ist allerdings noch Aufgabe des Compilers, dies so einzurichten.

Statisches und dynamisches Loop Unrolling haben Vor-und Nachteile: Im statischen Loop Unrolling kann der Code vergrößert werden, schädlich insbesondere

für L1-Caches (→ 9), hingegen verspricht die Optimierung des Instruction Scheduling mehr Optimierungsmöglichkeiten als im dynamischen Fall.

Der dynamische Fall wird insbesondere im Multithreading, ausgerichtet auf Microthreads, erhebliche Vorteile haben.

Seitens der Gleitkommaregister bietet die IA-64 Architektur nun mit insgesamt 128 Gleitkommaregistern genügend Kapazität an. Alle sind nach dem 3-Adress-Schema ansprechbar – sieht man mal von den ersten beiden ab, die fest auf 0 beziehungsweise 1 stehen. Genauso wie das Integer-Pendant teilen sich diese FP-Register in 32 statische und 96 dynamische auf, mit den gleichen Vorteilen und Rotationsmöglichkeiten wie oben beschrieben.

Die FP-Register sind mit 82 Bit sogar etwas breiter als bei IA-32 (80 Bit). Und da sich 82 Bit relativ schlecht abspeichern lassen, hat Intel ein neues Gleitkommaspeicherformat (mit viel Reserve für die Zukunft) definiert, das gleich 128 Bit (16 Bytes) belegt - neben dem Double-Precision-Format (DP) eine recht natürliche Größe für 64-Bit-Systeme. Und das Schöne daran ist, dass man damit in einem Takt gleich zwei DP-Werte laden oder speichern kann.

Die höhere 82-Bit-Genauigkeit hat allerdings einen praktischen Hintergrund: Division, Wurzel, Sinus, Tangens, Exponentialfunktion oder Logarithmus sind in der IA-64 Architektur nicht vorhanden. All diese Funktionen muss der Compiler via Softwarebibliothek bereitstellen. Die Algorithmen erfordern zum Abspeichern der Zwischenwerte die beiden zusätzlichen Bits, damit das Ergebnis im Rahmen der geforderten IEEE-Genauigkeit bleibt.

Das Fehlen vor allem der elementaren Funktionen Division und Wurzelziehen sieht zunächst nach einem Nachteil aus. Das ist aber mitnichten der Fall. De facto arbeiten nämlich übliche Gleitkommaeinheiten bei diesen Funktionen auch nur mit einem Softwarealgorithmus, der im Unterschied zu IA-64 fest in Mikrocode eingebrannt ist. Solche Berechnungen sind dann aber in der Regel kaum oder gar nicht ’pipelined’: Die Recheneinheit ist während der gesamten Berechnung weitgehend belegt und kann keine neuen Befehle entgegennehmen.

Die IA-64-Divisionsroutine ist hingegen weitgehend pipelined, und der Compiler hat vor allem die Möglichkeit, zwischen verschiedenen Optimierungen zu wählen. Soll eine einzelne Division möglichst schnell fertig werden, so wählt er die ’Latency-optimierte Fassung’. Sollen jedoch viele voneinander unabhängige Divisionen durchgeschoben werden, so kommt die Throughput-optimierte Version zum Einsatz. Bei dieser dauert es zwar länger, bis die erste Division abgeschlossen ist, im Schnitt liegt aber die mittlere Rechenzeit pro Division drastisch darunter. So dauert eine Division mit Datentyp Double im Latency-optimierten Fall 35 Takte (PowerPC: 31, Pentium III: 32). Throughput-optimiert sinkt die mittlere Divisionszeit auf nur 5 Takte (PowerPC weiterhin 31, Pentium III ca. 29).

IA-64 unterstützt auch SIMD (Single Instruction Multiple Data) für Single-Precision-Gleitkomma, allerdings nicht wie bei SSE vierfach parallel mit 128-bittigen Registern, sondern 64-bittig, also zweifach parallel. Das gilt allerdings pro Befehl,

wobei der Prozessor mehrere Gleitkommabefehle gleichzeitig ausführen kann – der Itanium beispielsweise zwei.

Wichtig für wissenschaftliche Anwendungen ist der mächtige MAC-Befehl (Multiply-Add). Pro Takt und pro Gleitkommaeinheit kann der Prozessor damit zwei Double Precision Operationen: Multiplikation und Addition ausführen (im Durchsatz). Reicht die einfache Genauigkeit, so steigt der Durchsatz dank SIMD auf vier Operationen.

Neben den beiden GP- und FP-Registersätzen unterstützt IA-64 noch eine Vielzahl von Spezialregistern (→ Bild 7.6, 7.7): Die 128 Applikationsregister enthalten eine Vielzahl von Spezialregistern, die für Applikationen sichtbar sind, so z.B. Kernelregister, Statusregister und Loop-Counter (64 Bit). Weitere Register enthalten die CPU-ID und diverse Performance-Monitore (mindestens 4 Register).

Die 64 Predicateflags dienen den bedingten Befehlen und wurden bereits erwähnt. Das Konzept der bedingten Befehle, die mithilfe der sog. *if-conversion* Verzweigungsbefehle, aus Verzweigungsstrukturen (und nicht Schleifen) stammend, verhindern können, wurd bereits in den 90er Jahren entwickelt und ist z.B. in den ARM-Architekturen ab V4 enthalten. Anders als dort wurden bei IA-64 64 eigenständige Bits definiert (und nicht nur das Carry-Bit genutzt), so dass dem Compiler bei der IA-64 Architektur wesentlich mehr Möglichkeiten gegeben sind.

Last not least enthalten die 8 Branchregister die Informationen für die letzten 8 Sprünge (function call linkage and return [22]). Hier werden die Rückkehradressen der Unterprogrammaufrufe gespeichert, ohne dafür den Stack benutzen zu müssen.

7.2.4 Datenspekulationen

Alle bisher behandelten spekulativen Ausführungen in einem (superskalaren) Prozessor betrafen den Kontrollfluss: Dies erschien und erscheint auch besonders lohnenswert, weil hier eine Menge an Ausführungszeit gewonnen werden kann. Wie jedoch schon im Abschnitt 5.3.4 gezeigt wurde, besitzen die Speicherzugriffe einen Sonderstatus innerhalb der Instruktionen. Sie werden in High-Performance-Prozessoren gesondert behandelt, um Ausführungszeit zu sparen.

Das Speichersystem zeigt lange Verzögerungszeiten bei Hauptspeicherzugriffen, und die Speicherhierarchie zur (statistischen) Verkürzung dieser Zeiten (→ 9) ist auch nicht unproblematisch, weil die Daten (und Befehle) erst im Cache sein müssen. In Zusammenhang mit den meist vorhandenen Datenabhängigkeiten für nachfolgende Befehle wird die Dringlichkeit deutlich, insbesondere Load-Befehle in ihrer Ausführung zu beschleunigen.

Hier setzt die spekulative Ausführung der Datenbefehle ein: Nicht die Datenwerte werden ’erraten’, sondern die Ladezugriffe werden vorzeitig – spekulativ – ausgeführt. Dies kann üblicherweise auf zwei Arten erfolgen:

- Cache-Ladebefehle können im Programmfluss anzeigen, dass in Kürze ein bestimmter Datenbereich benötigt wird. In diesem Fall kann das Speichersystem damit beginnen, den Cache mit den Informationen zu füllen.
- Ladebefehle können in der Ausführung vorgezogen werden. In diesem Fall können die Daten auch im Hauptspeicher sein, um rechtzeitig geladen zu werden, falls natürlich die Wartezeit entsprechend kurz ist.

Die IA-64 Architektur verfolgt den Weg der spekulativen Ladebefehle (Advanced Load, die durch den Assemblerprogrammierer oder den Compiler genutzt werden können. Der Weg dahin ist nicht trivial:

- Die Advanced-Load-Befehle werden durch den Compiler eingefügt, wenn die Load- und Store-Befehle nicht mit Sicherheit so umsortiert werden können, wie es aus Programmablaufgründen optimal wäre: Load-Befehle sollen so früh wie möglich beginnen.
- Der Zugriff erfolgt so, dass das Ergebnis in einer Advanced-Load-Address-Table (ALAT) gespeichert wird. An der eigentlichen Stelle im Programmcode, wo der Zugriff sicher erfolgen könnte, steht eine zweite Instruktion (Check- oder Load-Instruktion), die bei vorhandenem Ziel den Wert aus der ALAT in einem Takt in das Register kopiert.
- Jeder Schreibzugriff auf die gleiche Adresse, der vor dem zweiten Lesezugriff aktiv wird, macht den ALAT-Eintrag ungültig.
- Spekulative Datenzugriffe können mit spekulativer Kontrollflussausführung (Branch Prediction) kombiniert werden. Zeigt sich dabei, dass der Kontrollfluss nicht ausgeführt werden durfte, werden ggf. Registerinhalte per NaT-Bit (Not-a-Thing) für ungültig erklärt.

7.2.5 Itanium Mikroarchitektur

Die erste Inkarnation der IA-64 Architektur heißt Itanium und besitzt 25 Millionen Transistorfunktionen. Vier Integer-, vier Multimedia- und zwei Gleitkommaeinheiten (Single/Double Precision) bilden zusammen mit zwei weiteren Gleitkommaeinheiten mit Single-Precision für SIMD, drei Sprung- und zwei Lade-/Speichereinheiten die rechnerische Kapazität dieses Bausteins. Zwei Befehlsbündel, mithin bis zu sechs Befehle, vermag der Itanium pro Takt an diese Einheiten zu vergeben – er arbeitet also neben der statischen Parallelität der VLIW-Befehle auch noch superskalar.

Die Pipeline (Bild 7.8) ist mit ihrer Länge von zehn Stufen für den trotz seiner vielen Register und Einheiten recht einfach strukturierten Prozessor relativ lang, aber ein guter Kompromiss zwischen ’sequenzieller Parallelität’ (Pipeline möglichst lang, ergibt kurze Bearbeitungszeiten pro Stufe) und Sprunganfälligkeit (Pipeline möglichst kurz, um den Verlust bei Fehlvorhersage klein zu halten). Dank der Prädikationen hat der Prozessor seltener mit bedingten Sprüngen zu kämpfen.

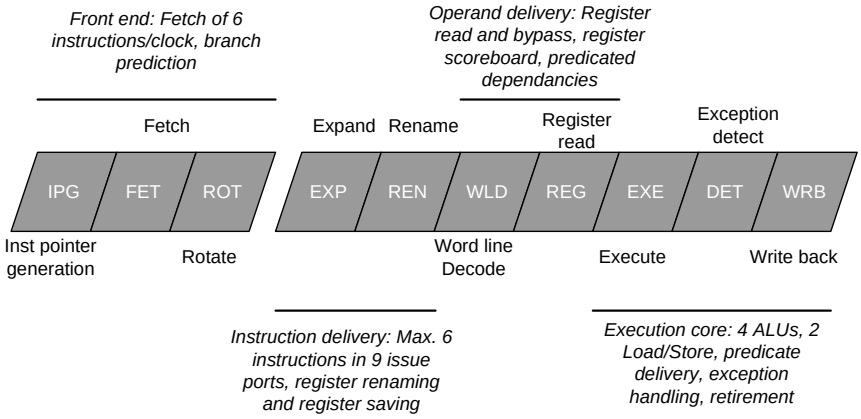


Bild 7.8 10stufige Pipeline des Itanium [23]

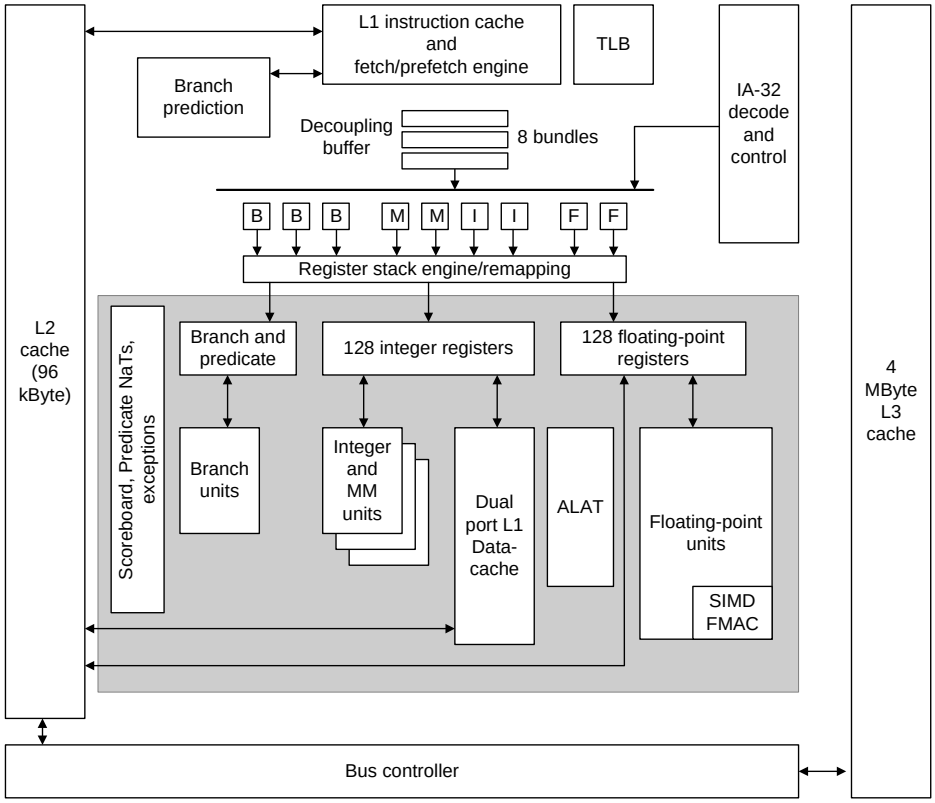
Bild 7.9 zeigt den Blockaufbau des Itanium. 9 Issue-Ports (zwei FPU, zwei Memory, zwei Integer und drei Branches) werden die insgesamt 17 Funktionseinheiten versorgt. Einen aufwendigen Dispatcher wie bei superskalaren Prozessoren gibt es nicht: die Bündel warten in einem kleinen Puffer (mit einer Kapazität von 8 Bündeln), bis sie als Ganzes über die Ports an freie Einheiten verteilt werden können.

Die L1-Caches sind mit jeweils 16 KByte für Instruktionen und Daten (4-Wege assoziativ, 32 Byte je Cache-Line) nicht gerade üppig ausgelegt, insbesondere unter Berücksichtigung der Tatsache, dass 64-Bit-Systeme vergleichsweise großzügig mit Speicherplatz umgehen. Hinzu kommt, dass der L1-Daten-Cache nur als Write-Through ausgelegt ist.

Die beiden FPUs greifen nur auf den L2-Cache zu, sodass der L1-Cache etwas entlastet wird. Die etwas größere Latenzzeit des L2-Caches fällt bei den möglichen FPU-'Doppelzugriffen' von 16 Bytes weniger ins Gewicht. Allerdings ist auch der L2-Cache mit 96 KByte sehr klein ausgefallen. Es ist als Writeback-Cache mit 64-Bit-ECC (Error Correction Coding) ausgelegt. So wichtig dieses ECC für einen Serverprozessor ist, für die 32-Bit-Emulation stellt das einen weiteren Stolperstein (im Sinne einer Performanceminderung) dar, denn bei 8-, 16- oder 32-bittigen Schreibvorgängen muss der Prozessor erst einmal die zugehörige Cacheline lesen, verändern, die neue ECC-Prüfsumme bilden und hernach alles abspeichern.

Der L3-Cache sitzt extern auf einem Modul, einigermaßen zügig über einen 128-bittigen Backside-Bus mit vollem Prozessortakt (750 MHz) angekoppelt. So erreicht er eine ordentliche Bandbreite von 12 GByte/s. Mit 2 oder 4 MByte hat er auch eine vernünftige Größe. Zum System hin arbeitet Itanium mit einem 64-bittigen Bus, mit 133 MHz Takt und Double Data Rate (2 GByte/s). Das liegt deutlich unter dem des Pentium 4 (3,2 GByte/s). Und auch andere Serverprozes-

soren sind hier (dank breiterer Busse) oft schneller. Dafür kann man vier Itanium-Prozessoren ohne weitere Zusatzhardware über den Systembus zusammenkoppeln, was vergleichsweise preiswerte Vierfachserver ermöglicht.



ALAT Advanced Load Address Table
TLB Translation Look-Aside Buffer

Bild 7.9 Blockarchitektur des Itanium [22]

7.2.6 Fazit IA-64 Architektur

Die IA-64 Architektur ist mit der EPIC-Philosophie angetreten, um die Architekturlandschaft durch eine neue, von VLIW abgeleitete Variante zu revolutionieren. Die genaue Analyse zeigt jedoch, dass IA-64 im Kern die VLIW-Architektur annimmt und diese durch einige Erweiterungen für Compiler und für den Betrieb besser auslegt.

Als evolutionäre Weiterentwicklungen sind die Kombination von VLIW und superskalarer Architektur zu nennen: Die Analyse auf Datenabhängigkeiten erweist sich im Fall der IA-64-Instruktionsbündel offenbar als so einfach, dass ohne viel Aufwand mehrere Bündel zur Ausführung gebracht werden können.

Die Grundlage dieser Struktur, die Instruktionsbündel, werden von dem Prozessor nur noch auf ’Fehler’ (vorhandene Datenabhängigkeiten) getestet, ein Verfahren, das vergleichsweise einfach ist.

Eine wirkliche Neuerung ist die Datenspekulation, besser: die Datenzugriffsspekulation. Diese ermöglichen vorzeitige, aber eben korrigierbare Zugriffe und damit dem Compiler ein Instruction Scheduling, das bei den Speicherzugriffen erheblich verbessert ist.

Die ersten Prozessoren, Itanium von Intel, kamen sehr schnell in die Kritik: Zu langsam (bei einem Betriebstakt von 750 MHz) und vor allem zu schlechte Performance beim Ausführen von IA-32-Code. Wesentliche Schwächen hierzu konnten in der Auslegung des Caches und des Speicherinterfaces identifiziert werden – Eigenschaften, die nicht in der Architektur, sondern in der Implementierung liegen.

8 Multithreading

RISC und Superskalarität wurden erfunden, um die Hardware- und die Instruktionslevelparallelität zu realisieren. Allgemein teilt man die Parallelitätsebenen der Programmausführung in 4 verschiedene Klassen ein (Bild 8.1).

Hardwareparallelität
Instruktionslevelparallelität
Threadlevelparallelität
Prozessparallelität

Bild 8.1 Ebenen der Parallelität bei Programmausführung

Die 'Enden' dieser Parallelitätsskala sind wohlbekannt, Hardware- und Prozessparallelität werden seit Jahren genutzt und unterstützt (gleichwohl ist der Ansatz der Prozessparallelität, also eines MIMD-Rechners gemäß Flynn'scher Klassifikation, in seiner Güte sehr abhängig von der Software). Die Instruktionslevelparallelität wird ebenfalls schon sehr ausgenutzt, hier sind ebenfalls schon viele Möglichkeiten ausgenutzt, so dass für die nächste Zukunft die Threadlevelparallelität als Forschungs- und Entwicklungsthema bleibt.

Es stellt sich die Frage, was ein *Thread* eigentlich ist [18]:

Definition 8.1

Ein **Thread** ist ein *Aktivitätsträger* (sequenzieller *Ausführungsfaden*) mit minimalem Kontext (Stack und Register) innerhalb einer *Ausführungsumgebung* (Prozess). Jeder Prozess besitzt in diesem Fall mindestens einen (initialen) Thread. Alle Threads, die zu ein und demselben Prozess gehören, benutzen denselben Adressraum sowie weitere Betriebsmittel dieses Prozesses gemeinsam.

Diese Definition ist in gewissem Sinn "weich", sie lässt eine Menge Interpretationsraum. Threads können durch den/die SoftwareentwicklerIn definiert sein (Entwicklung von Multithreaded Programmen), ebenso könnte man kleinere Einheiten, bis hin zu den Basisblöcken (→ 5.2), als Thread bezeichnen, allerdings mit anderem Kontext und Eigenschaften.

Dementsprechend werden in diesem Kapitel zwei Ansätze verfolgt: Feinkörnige und grobkörnige Architekturen zur Ausnutzung des Multithreading. Die Grenze ist natürlich schwer zu ziehen, was die Körnigkeit betrifft, dennoch gibt es ein klares Unterscheidungskriterium: Die Datenabhängigkeiten.

- Bei grobkörnigem Multithreading geht man davon aus, dass die Daten, auf denen zwei Befehle zweier verschiedener Threads gleichzeitig operieren, grund-

sätzlich nicht voneinander abhängen, so dass auch die Zeitpunkte der Operationen nicht voneinander abhängen. Dies hat seinen Ursprung in der Tatsache, dass diese Threads im Wesentlichen durch die Softwareentwicklung explizit definiert sind.

Tatsächliche Datenabhängigkeiten, die durch den gemeinsamen Adressraum entstehen, können durch entsprechende Programmentwicklung vermieden oder gemindert werden (lokale Daten). Bei Auftreten muss die Hardware natürlich entsprechend reagieren.

- Bei feinkörnigem Multithreading gilt die Annahme der Unabhängigkeit nicht mehr, hier muss auch auf Registerebene mit Abhängigkeiten gerechnet werden. Gerade diese Einschränkung – die Register zählen nicht mehr zum lokalen Threadkontext – ergibt die Unterscheidung der beiden Ansätze.

In den folgenden Abschnitten werden zwei Ansätze für feinkörnige Parallelität im Multithreading vorgestellt (Multiskalarer Prozessor, Trace-Prozessor), gefolgt von dem weit verbreiteten Ansatz des Simultaneous Multithreading (SMT). Im dritten Abschnitt wird dann die Implementierung bei Intel Pentium 4 als konkretes Beispiel vorgestellt.

8.1 Feinkörnige Parallelität

Der Ansatz zu einer Ausführung von parallelen Threads mit einer Feinkörnigkeit besteht in der Fokussierung auf Instruktionsblöcke. Hier liegt es nahe, die Basisblöcke (→ 5.2) bzw. die später beschriebenen Hyperblöcke (→ 6.2.3) zugrunde zu legen. Diese Blöcke beinhalten einen ungestörten sequenziellen Fluss von Instruktionen und waren somit schon ideal für die superskalare Ausführung.

Bild 8.2a zeigt den Kontrollflussgraphen (CFG, Control Flow Graph) eines Programms: Die Instruktionsblöcke – auch als *Microthreads* bezeichnet – werden als Knoten, die Übergänge zu anderen Blöcken als Kanten aufgefasst. Die Darstellung als CFG beinhaltet zunächst nur die statische Repräsentation des Programms, Aussagen über den tatsächlichen Programmverlauf sind nicht enthalten.

Der dynamische Kontrollfluss ist ebenfalls in Bild 8.2a dargestellt. Hier werden – je nach Programmkontext – verschiedene Knoten durchlaufen. Die Auswertung dieses aktuellen Programmverlaufs wird nun in verschiedenen Ansätzen unterschiedlich behandelt.

8.1.1 Multiskalarer Prozessor

Im *multiskalaren* Prozessor werden verschiedene Knoten des CFG, also Instruktionsblöcke, auf verschiedene Prozessorelemente (PE) des Prozessors zur Ausführung abgebildet. Diese Abbildung bewirkt, dass der Prozessor mehr auf Instruktions*block*basis als auf Instruktionsbasis arbeitet.

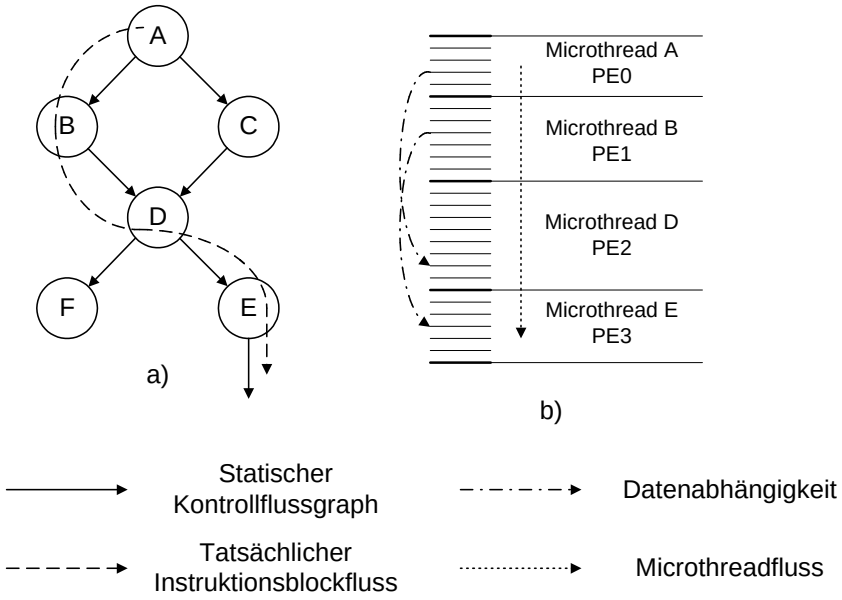


Bild 8.2 Kontrollflussgraph und Microthreading
a) Statischer und dynamischer Kontrollflussgraph (CFG) b) Abbildung auf Prozessorelemente im multiskalaren Prozessor

Bild 8.2b zeigt eine derartige Abbildung der Microthreads auf die Prozessorelemente. Die multiskalare Architektur [11] besticht dabei durch einen einfachen Ansatz: Anstelle eines superskalaren Prozessors werden die Ressourcen auf mehrere, eher als RISC-Architektur ausgeführte Prozessoren (PE genannt) verteilt, und jeder dieser PEs bekommt einen Microthread zur Ausführung. Der Name *multiskalar* rührt aus diesem Ansatz her, dass viele skalare Ausführungseinheiten nebeneinander arbeiten.

Der bestehend einfache Ansatz wird durch die dynamische Verwaltung und vor allem die Datenabhängigkeiten zwischen verschiedenen Microthreads komplex. Im Unterschied zu einem Multiprozessoransatz (meist als CMP, Chip-integrated Multiprocessing bezeichnet) wird hier nicht durch den Compiler oder den Assemblerprogrammierer bestimmt, welcher Prozessor welchen Teil berechnet, sondern dynamisch zur Laufzeit. Dies bedeutet, dass eine dynamische Zuordnung von PEs zu Microthreads durchgeführt werden muss, ggf. ohne zusätzliche Informationen durch den Compiler.

Die zweite Schwierigkeit besteht in den möglichen Datenabhängigkeiten, angedeutet in Bild 8.2b. Wie sollen solche Datenabhängigkeiten aufgelöst oder auch nur erkannt werden?

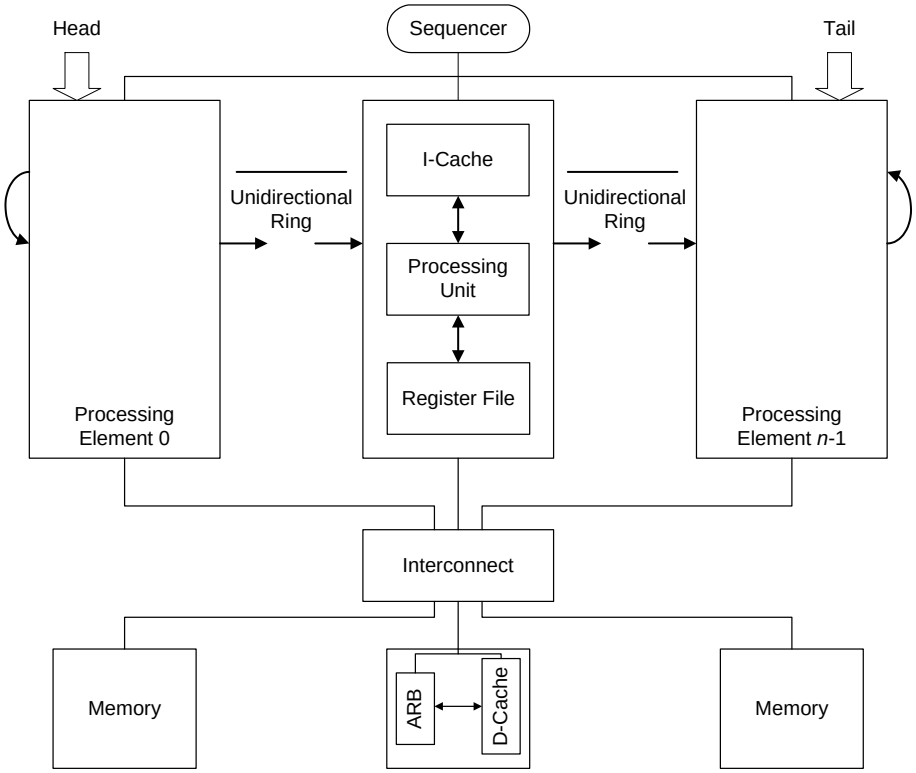


Bild 8.3 Blockarchitektur eines multiskalaren Prozessors [11]

Bild 8.3 gibt Aufschluss über den Aufbau und damit auch die Arbeitsweise des multiskalaren Ansatzes. Die Processing Elements PE (Anzahl *n*) sind in einer Art einfachen Verkettung zu einem Ring miteinander kommunizierend verbunden. Zunächst ist es die Aufgabe des Sequencers, die Microthreads an die PEs zu verteilen. Hierzu ist auch die Branch Prediction in dieser Einheit enthalten. Die einzelnen PEs sind für den Fetch der entsprechenden Instruktionen verantwortlich und besitzen dazu einen Instruktionscache (→ 9.2).

Jedes PE enthält auch ein vollständiges Register File (mit je einem physikalischen Register pro logischem Register). Beim Instruktionsblockfetch wird detektiert, welche logischen Register potenziell beschrieben werden. Alle im Ring nachfolgenden PEs werden über diesen potenziellen (und später auch tatsächlichen) Schreibzugriff informiert, so dass RAW-Hazards detektiert und durch Warten aufgelöst werden. Die Kommunikation erfolgt z.B. über Bitmasken.

Die Auflösung der Speicherzugriffe ist bereits schwieriger. Hierzu ist über einen gemeinsamen Interconnect ein Address Resolution Buffer (ARB) vorgesehen, dessen Aufgabe in der Auflösung von Schreib-/Lesezugriffen auf gemeinsame

Adressen, Vermeidung von Zugriffskonflikten etc. besteht. Letztendlich muss jedoch deutlich gesagt werden, dass der Speicherzugriff einer der Engpässe in dem multiskalaren Ansatz sein wird.

Zusammenfassend findet man in der multiskalaren Architektur zwei Formen der Spekulation:

- Kontrollflussspekulation in der Sequencer-Einheit: Hier wird aufgrund der Vorinformationen und weiterer Annahmen über den Verlauf der Instruktionsblöcke spekuliert.
- Datenspekulationen: Ähnlich wie bei der IA-64 Architektur (→ 7.2.4) können Datenzugriffe im Speicher spekulativ ausgeführt werden

8.1.2 Trace-Prozessor

Der Trace-Prozessor ist ein anderer Ansatz, zu einer feinkörnigen, parallelen Threadausführung zu kommen. Hier wird bei einem ersten Durchlauf der Instruktionsfluss in einem speziellen Instruktionscache, dem *Tracecache*, aufgezeichnet. Hierbei werden Instruktionsblockgrenzen ebenso durchlaufen wie alle anderen Instruktionen, so dass in Tracecache mehrere Kontrollflussspekulationen implizit gespeichert sind.

Definition 8.2:

Ein **Trace** ist eine Sequenz von Instruktionen, die potenziell mehrere Instruktionsblöcke überstreichen kann. Sie startet an einem beliebigen Startpunkt im dynamischen Instruktionsfluss. Der *Trace* ist durch den Startpunkt, die Wegewahl an den Verzweigungspunkten und den Endpunkt komplett definiert.

Definition 8.3:

Ein **Tracecache** ist ein spezialisierter Instruktionscache, der die dynamische Folge von Instruktionen anstelle der statischen Instruktionen speichert. Diese dynamische Folge entsteht durch die Ausführung des Programms.

Die Anzahl der Basisblöcke, die ein Trace überstreicht, wird als **Branch-Predictor-Throughput** (Durchsatz) bezeichnet.

Die Implementierung eines Tracecache kann in Form der Startadresse und der Branchvorhersagen erfolgen (minimaler Speicheraufwand). Diese Form hat den Nachteil, dass der Instruktionsfetch trotzdem im Instruktionscache oder im Hauptspeicher erfolgen muss, weil diese Informationen nicht vorliegen. Die bessere Methode besteht dann darin, die komplette Folge der Instruktionen im Cache zu speichern. In diesem Fall kann das Laden des kompletten Codes in einem Takt erfolgen, allerdings mit erhöhtem Aufwand an Speicher.

Bild 8.4 zeigt eine mögliche Implementierung eines Trace-Prozessors. Im Prinzip so aufgebaut wie der multiskalare Prozessor (mit dem Unterschied in der Fetch-Einheit), wird in dieser Darstellung eine andere Art der Kommunikation genutzt. Das Processing Element 0 führt den aktuellen Trace aus, während die anderen die

zukünftigen Traces unter spekulativen Bedingungen bearbeiten. Die globalen Register dienen dabei dem Datenaustausch – mit anderen Worten: Das Programm muss für den Trace-Prozessor compiliert sein, da die Datenabhängigkeiten auf Registerebene bekannt sein müssen.

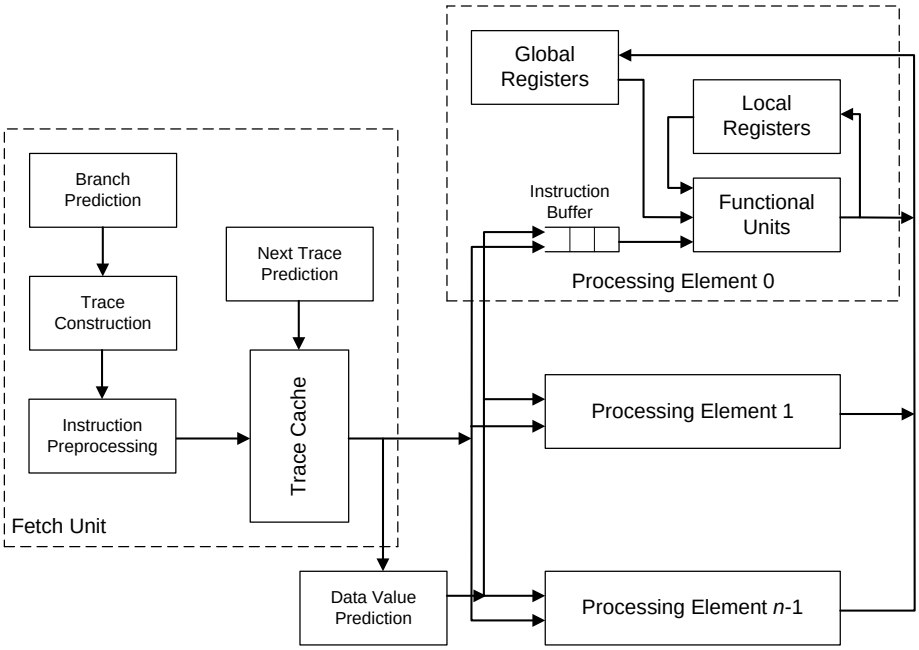


Bild 8.4 Trace Processor [11]

8.2 Grobkörnige Parallelität

8.2.1 Allgemeines zur Multithreaded Architekturen

Die grobkörnige Parallelität wird im Allgemeinen lieber gewählt, weil sie für weniger Datenabhängigkeiten steht. Im Extremfall – auf Prozessebene – ist eine Unabhängigkeit sogar garantiert, bedingt durch die Definition von Prozessen und deren exklusiven Speicherbereichen.

Unterhalb der Prozessebene, also auf Threadebene, gilt die Annahme aber ebenfalls weitgehend. Einer der wesentlichen Anlässe, sich mit der quasi gleichzeitigen Ausführung von Threads in einem Prozessor überhaupt zu befassen, entstammt den Speicherlatenzzeiten. Insbesondere bei Mehrprozessorsystemen und physikalisch verteiltem, logisch aber globalem Speicher, sind diese Zeiten ggf. sehr groß.

Folgende Messwerte können für ein System mit 4 Alpha 21164, 300 MHz (1998) angegeben werden [11]:

- 7 Wartezyklen für einen L1-Cache-Miss, aber einen Hit im L2-Cache (On-Chip) desselben Prozessors
- 21 Wartezyklen für einen L2-Cache-Miss, aber einen L3-Cache-Hit (On-Board)
- 80 Wartezyklen für einen L3-Cache-Miss, so dass im Hauptspeicher zugegriffen werden muss
- 125 Wartezyklen für einen 'Dirty Miss', d.h. ein Zugriff, dessen Wert aus einem Cache eines anderen Prozessors geholt werden muss.

In derartigen Systemen (die Zeiten gehen für Systeme mit noch mehr Prozessor noch wesentlich höher) muss man erwarten, dass ein wesentlicher Anteil der Rechenzeit für das Warten auf Datenwerte verwendet werden muss. Letztendlich liegt die Ursache hierfür darin, dass das Konzept der Von-Neumann-Maschine zwei Zustände berücksichtigen muss: Den *Prozessorzustand*, bestehend aus dem Aktivitätsregister (Program Counter) und dem Registerkontext, sowie den *Speicherzustand*. Diese Zustände müssen für jede Instruktion so abgestimmt werden, dass die nächste Instruktion beginnen kann – dies bedeutet häufig Warten.

Diese Variante der Mikroprozessoren, aktuell fast ausschließlich verwendet, wird als *singlethreaded* bezeichnet. Im Gegensatz dazu ist eine *multithreaded* Architektur dadurch gekennzeichnet, dass sie zwar nur einen Prozessorzustand kennt, dieser aber aus einem Satz von Aktivitätsregistern und einem Satz von Registerkontexten gekennzeichnet ist. Hieraus ergeben sich dann aber andere Möglichkeiten im Fortschreiten der Programmausführung.

In Anbetracht der Leistungssteigerungen, die im Rahmen von superskalaren Rechnerarchitekturen noch möglich sind und i.Allg. für sehr begrenzt gehalten werden, besteht durch die Nutzung und vor allem Unterstützung grobkörniger Parallelität in Prozessoren die Chance zu erheblichen Leistungssteigerungen. Die praktischen Leistungsfähigkeiten von superskalaren CPUs sind bei 4 bis 8facher Superskalarität auf eine Performance von ca. 1.5 Instruktionen pro Takt begrenzt (z.B. bei Verwendung der SPEC92-Benchmarks: 0.96 .. 1.77 beim MPS620 PowerPC).

Um dies zu erreichen, wird bei Multithreaded-Architekturen bei Blockierung eines Threads etwa durch Speicherzugriffe auf einen anderen Thread umgeschaltet. Dieser Kontextswitch ist deshalb möglich, weil mehrere Prozessorzustände vorhanden sind. Bei einer singlethreaded Architektur wäre der Kontextswitch wahrscheinlich zu aufwendig.

8.2.2 Prinzipielle Ansätze zu Multithreaded-Architekturen

Durch die unterschiedlichen Methoden zur Implementierung des Kontextswitches lassen sich drei verschiedene Techniken für Multithreaded Prozessoren unterscheiden:

- Die *Cycle-by-Cycle-Interleaving Technik*: Mit jedem Prozessortakt wird ein Befehl eines anderen Kontrollfadens in die Prozessorpipeline eingespeist. Dieses Verfahren besitzt den Nachteil einer geringen Leistung, falls nur wenige Kontrollfäden (Threads) als Last zur Verfügung stehen, da im Regelfall erst der Befehl eines Kontrollfadens die Pipeline verlassen muss, bevor die nächste Instruktion (dieses Kontrollfadens) geladen werden. Diese einfache Form des Threadschedulings im Prozessor hat also pro Faden nicht die Eigenschaften eines Befehlspipelinings der RISC-CPUs.

  Andererseits kann gerade in diesem Fall die Befehlspipeline sehr einfach ausgeführt werden, da keine Daten- oder Kontrollflussabhängigkeiten zu detektieren sind: Jeder Befehl eines Threads verlässt die Pipeline komplett, bevor der nächste Befehl geladen wird. Auch der Kontextswitch besitzt keinen Overhead, da er jedes Mal ausgeführt wird, also die besonderen Bedingungen (wie im Fall der Block-Interleaving-Technik) eines Wechsels nicht detektiert werden müssen.

- Die *Block-Interleaving-Technik*: Die Befehle eines einzelnen Kontrollfadens werden solange aufeinanderfolgend ausgeführt, bis ein Ereignis eintritt, das zu Wartezeiten führt. In diesem Fall wird der Kontextswitch durchgeführt, wobei als Wartezeiten beispielsweise eine fehlende Synchronisation, ein fehlgeschlagener Cachezugriff oder auch direkt Load- sowie Store-Zugriffe definiert sind.

  Der Nachteil dieser Technik ist ggf. die Tatsache, dass solche Wartezeiten erst spät in der Befehlspipeline erkannt werden können und damit zu einem hohen Wechselaufwand (mehrere Takte) führen (für fehlende Synchronisation oder fehlgeschlagene Cachezugriffe) bzw. recht häufig ausgeführt werden; letzteres ist für Load/Store-Befehle der Fall, die bereits im Befehlsstrom decodierbar sind, aber auch sehr häufig auftreten (können).

- Die *Simultaneous-Multithreading-Technik*: Die Ausführungseinheiten eines superskalaren Prozessors werden simultan aus mehreren Befehlspuffern bestückt. Jedem Kontrollfaden ist ein eigener Registersatz und ein eigener Befehlspuffer zugeordnet, so dass als Fazit die Superskalartechnik mit breiter Zuordnungsbandbreite und die Multithreadingtechnik miteinander kombiniert werden. Die Ausführungsparallelität kann in wesentlich vergrößerten Maß ausgenutzt werden, da mehrere Threads mit voneinander unabhängigen Befehlsströmen einen Input für die Funktionseinheiten liefern, die Abhängigkeiten also sehr gering sind.

8.2.3 Vergleich der Interleaving-basierten Ansätze

Bild 8.5 zeigt die Cycle-by-Cycle-Interleaving-Technik im Vergleich zur RISC-CPU. Die Basis-CPU wird hierbei mit einer Pipelinelänge von 4 Stufen angenommen. Die Ausführung eines Befehls dauert in der Regel 1 Takt, die Ausnahmen entstehen durch z.B. Daten- oder Kontrollflusshazards sowie Speicherlatenzzeiten (die im Übrigen auch zu den Datenhazards gezählt werden können.

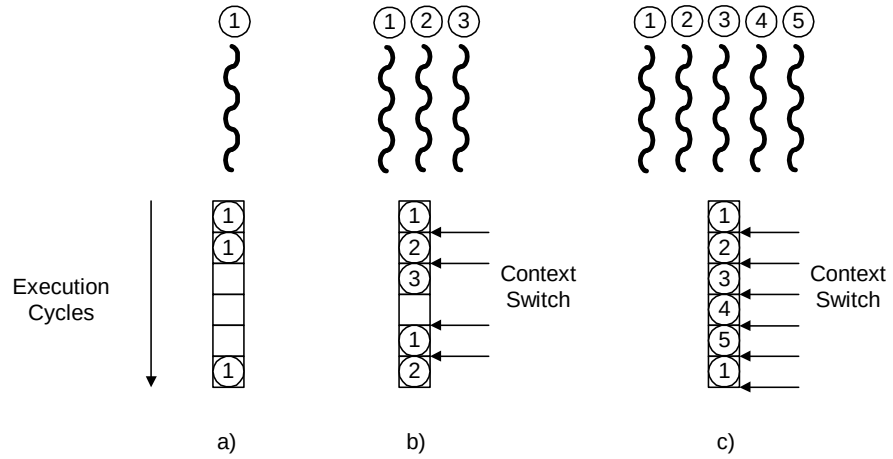


Bild 8.5 Multithreadingansätze Teil 1
a) RISC-Prozessor b) Cycle-by-Cycle-Interleaving mit geringer Anzahl von Threads
c) Cycle-by-Cycle-Interleaving mit ausreichender Anzahl von Threads

Die CPU in Cycle-by-Cycle-Interleaving-Technik führt in jedem Takt einen Kontextswitch durch, soweit dies möglich ist. Die Anzahl der ausführbaren Threads ist mit 3 in dem Beispiel b) zu klein, um die vierstufige Pipeline komplett zu füllen. Aus diesem Grund kommt es zu Pipelinestalls.

Die wesentliche Quelle für Pipelinestalls können jedoch Speicherlatenzzeiten sein. Wie bereits dargestellt sind Wartezyklen von mehr als 4 sehr schnell möglich, auch bei Verwendung von L2- und L3-Caches. Dies bedeutet, dass bei längeren Wartezeiten sehr schnell ein Pipelinestall (mit erheblicher Anzahl von Wartetakten eintreten wird.

Bild 8.6 stellt den Fall der skalaren (RISC-) CPU mit Block-Interleaving-Technik zum Multithreading dar. Diese Variante schaltet im Kontext um, wenn bestimmte Bedingungen (z.B. Load/Store-Befehle) auftauchen. Das Umschalten bedeutet, dass diese Bedingung detektiert werden muss, was in der Regel in der Decode-Phase, ggf. auch in der Fetch-Phase möglich ist. Hieraus können Wartezyklen (Bild 8.6: 1 Takt) entstehen.

Der wesentliche Vorteil diese Variante besteht darin, dass zum Füllen der Pipeline wesentlich weniger Threads benötigt werden, auch bei Load/Store-Zugriffen auf den Speicher. Hierdurch können auch längere Wartephasen ausgeglichen werden, mit zusätzlicher Hardware sogar mehrere Load/Store-Zugriffe (z.B. von mehreren Threads). Diese Zugriffe werden dann in Reihenfolge bearbeitet.

All diesen skalaren Varianten ist gemeinsam, dass Datenabhängigkeiten im Speicher (die Register sind als Thread-lokaler Kontext exklusiv vorhanden) nicht auftreten können.

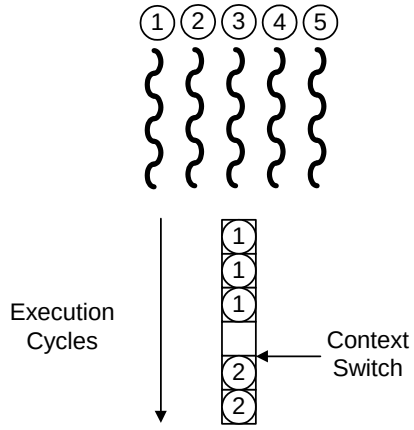


Bild 8.6 Skalare multithreaded CPU mit Block-Interleaving-Technik

Die Cycle-by-Cyle- und die Block-Interleaving-Technik lassen sich natürlich auch auf superskalare und VLIW-Architekturen anwenden. Die Unterschiede zwischen diesen beiden Varianten sind in diesem Zusammenhang nicht relevant, so dass Bild 8.7 nur die VLIW-Variante zeigt (N: No Operation, diese Operation füllt leere Slots in der VLIW-Architektur und entfällt bei der superskalaren Variante).

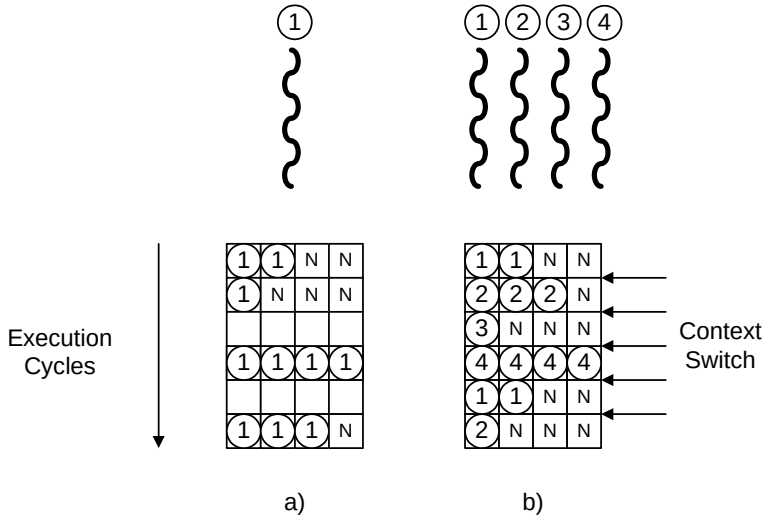


Bild 8.7 VLIW-Architektur
a) Normale Ausführung b) Cycle-by-Cycle-Interleaving-Technik

Wichtig für diese Form des Multithreading ist die Erkenntnis, dass die Ausführungsslots in der Vertikalen ausgefüllt werden, nicht jedoch in der Horizontalen. Dies wird im *Simultaneous Multithreadung* erreicht.

8.2.4 Simultaneous Multithreading

Der Ansatz zum *Simultaneous Multithreading* (SMT) entstammt nicht (nur) dem Bestreben, Latenzzeiten durch die Ausführung anderer Threads zu überbrücken, vielmehr sollen die Einheiten einer superskalaren oder VLIW-Architektur besser ausgenutzt werden. Hierzu werden mehrere Threads nicht nur scheinbar, sondern wirklich simultan zur Ausführung gebracht. Bild 8.8 demonstriert das Vorgehen.

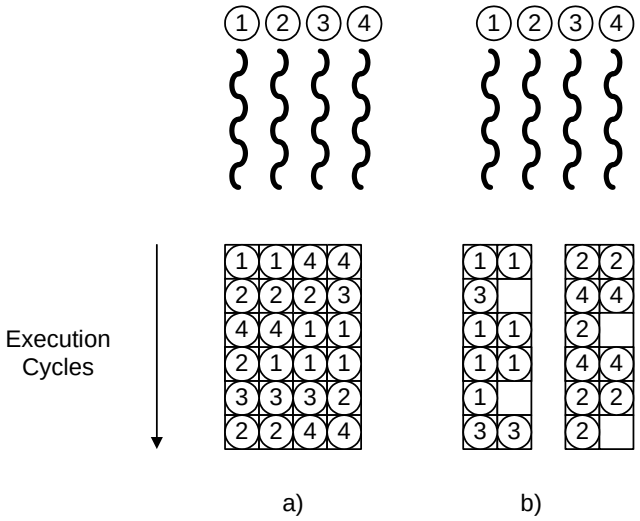


Bild 8.8 Ansätze zur parallelen Ausführung von Threads
a) Simultaneous Multithreading (SMT) b) Chip-integrated Multiprocessing (CMP)

Zur simultanen Ausführung mehrerer Threads sind grundsätzlich zwei Ansätze denkbar: Das (wirkliche) Simultaneous Multithreading (SMT, Bild 8.8a) und das Chip-integrated Multiprocessing (CMP, Bild 8.8b). Im CMP-Ansatz werden mehrere Prozessoren komplett auf einem Chip integriert und mit einem gemeinsamen Speicherinterface sowie gemeinsamen Speicher versehen. Häufig wird einer der Prozessoren (in Bild 8.8b sind nur 2 mit jeweils zweifacher Superskalarität angenommen, es können aber sehr viel mehr verwendet werden) dazu genutzt, die Zuteilung der Threads zu den Prozessoreinheiten zu übernehmen.

Der CMP-Ansatz ist vergleichsweise einfach zu realisieren, da die Prozessoren voneinander unabhängig agieren und nur über das Speicherinterface gekoppelt sind. Nachteilig wirkt sich natürlich aus, dass die Slots horizontal wiederum nicht aufgefüllt werden können.

Der SMT-Ansatz zeichnet sich dadurch aus, dass gegenüber dem Threadstrom scheinbar nur eine CPU ausgeführt ist, die dann durch eine geeignete Steuerung mit ausführbaren Instruktionen mehrerer Threads möglichst komplett ausgelastet wird.

Dies bedeutet konkret für den Aufbau eines SMT-CPU:

- Für jeden Thread, der parallel ausgeführt werden soll, muss ein Registersatz (ggf. Ausnahme bei Special Purpose Registern) inklusive eines Program Counters vorhanden sein.

- Die Einheiten des superskalaren Prozessors sind einzeln an verschiedene Threads zuordnungsfähig.
- Eine Zuordnungsstrategie, die den einzelnen Threads mit gleicher Priorität Ausführungszeit zuordnet.

8.2.5 Konkrete Ausführung einer SMT-fähigen CPU

Am Beispiel der SMT-fähigen Implementierung einer vereinfachten Version des PowerPC MPS604 [11, 19] soll der Aufbau eines derartigen Prozessors näher gezeigt werden. Bild 8.9 stellt den Blockaufbau dar.

In dem Aufbau können 4 Teile identifiziert werden:

- Die Steuerpipeline, bestehend aus Fetch, Decode, Dispatch/Issue und Completion/Retire Unit mit den zugehörigen Steuerpipeline-Puffern
- Mehrere, voneinander unabhängige Ausführungseinheiten (Integer, Floating Point, Load-/Store-Pipeline, Thread-Control Unit)
- Befehls- und Daten-Cache-Speicher mit zugehöriger Cachesteuerung sowie Branch-Target-Address-Cache (BTAC)
- Register Files mit Register Renaming und Activation Frame Cache.

Die Einheiten sind untereinander verbunden, wobei die Anzahl der Verbindung durchaus skalierbar sein kann. Die Anzahlen der Steuerpipeline-Puffer und der Registerfiles stimmen miteinander überein, um entsprechende Threads vollständig unterstützen zu können. Die Anzahl der Function Units kann hiervon jedoch abweichen, sie bewirkt den maximal möglichen Instruktionsparallelismus.

Für die einzelnen Einheiten müssen ggf. besondere Ausführungsformen gewählt werden; so sollte der Befehlspuffer nicht blockierend sein, d.h. im Fall eines Cache-Miss kann die Befehlsladeeinheit trotzdem weiter Befehle für einen anderen Kontrollfaden laden. Durch diese Konstruktion werden Befehlsladeeinheit und aktueller Speicherzugriff voneinander entkoppelt.

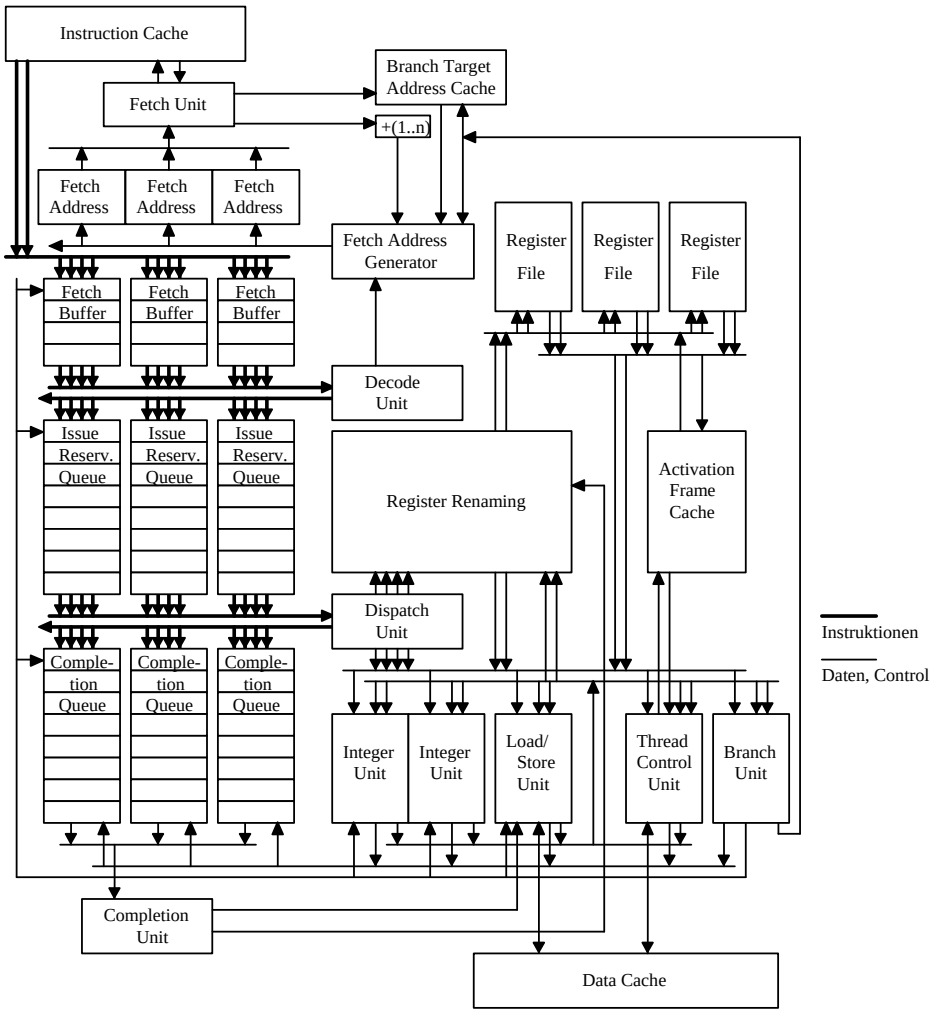


Bild 8.9 Aufbau einer SMT-fähigen CPU [19]

Die singuläre Ausführung der Dispatch Unit (Zuordnung) und der Completion Unit (Reordering and Commit Unit, Vervollständigung) ist durch die gemeinsame Aufgabe für alle Befehlsfäden gegeben: Der Dispatcher hat eine Querschnittsfunktion über alle aktiven Threads, die Completion Unit kann mehrere Befehle gleichzeitig vollenden und ist damit vom Befehlsfaden unabhängig.

Die SMT-fähige CPU, auch als Karlsruhe-Prozessor bekannt, wurde in erheblichem Maße skalierbar ausgeführt. Hierzu ist ein Activation Frame Cache integriert, der Daten aus gerade nicht in Ausführung befindlichen Threads speichern kann.

Diese Speicherung ermöglicht ein Umschalten ohne signifikanten Overhead und somit ein Multithreading über die Anzahl der Registersätze hinaus.

8.2.6 Zusammenfassung und Bewertung

Die Multithreaded CPU ist ein vielversprechender, weiterführender Ansatz, insbesondere in der Kombination mit einer superskalaren Architektur. Das Ziel, die vorhandenen (und in Zukunft erweiterten) Ressourcen auszunutzen, kann durch die Bemühung mehrerer Threads zum Laden der CPU erreicht werden. Hierbei steigt zwar nicht die Performance des einzelnen Threads, wohl aber die des Gesamtsystems.

Zurzeit sind zwei Trends zu beobachten: CMP und SMT. CMP gilt dabei als wesentlich einfacher in der Implementierung, da die Kontrollmechanismen zur Zuordnung von Instruktionen mehrerer Threads in eine Ausführungspipeline fehlen: Mehrere Prozessoren besitzen eben mehrere Ausführungspipelines.

Der SMT-Ansatz hingegen verspricht mehr Ausnutzung der vorhandenen Ressourcen. Derzeit ist nicht abschätzbar, welcher Weg aus Sicht der Hersteller endgültig beschritten wird, es zeigt sich jedoch eine Tendenz in Richtung SMT.

Der Weg des SMT-Ansatzes in industrielle Produkte – mit dem Intel Pentium 4 3.06 GHz (→ 8.3) im Jahre 2002 erstmalig beschritten – ist allein deshalb so schwierig, weil die Architektur auch in schnell ausführbare Hardware umgesetzt werden muss. Angesichts der Tatsache, dass die wesentlichen Verzögerungen in einer VLSI-Schaltung im Strukturbereich < 130 nm nicht mehr von den Transistoren, sondern von den Leitungen stammen, dürfte deutlich werden, dass hier ein sorgfältiges Layout (sog. Floorplan für die Lage und Verbindungsstruktur im Baustein) angefertigt werden muss.

8.3 Pentium 4 mit Hyperthreading (Intel)

Mit der für 3.06 GHz ausgelegten Version des Pentium 4 hat Intel eine Variante der bis dahin superskalaren Architektur eingeführt, die SMT-fähig ist. Intel bezeichnet diese Variante als Hyperthreading. Eine ausführliche Darstellung ist in [20] zu finden.

Die Hyperthreading-Architektur sieht eine 2-Thread-Struktur vor, d.h., µOps (→ 5.4.3) aus zwei verschiedenen Threads können zum Ablauf kommen. Im Rahmen der 20stufigen Pipeline (zur Ausführung der µOps) sowie der vorgelagerten Fetch-, Decode- und Übersetzungsphasen ergibt sich dann eine Ausführungsstruktur wie in Bild 8.10 gezeigt (Anmerkung: Die exakte Aufteilung in die Pipelinestufen ist nicht gezeigt!).

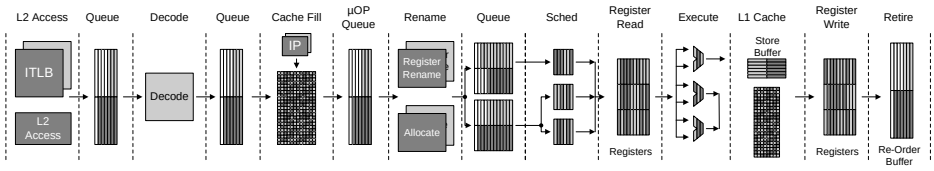


Bild 8.10 Ablaufstruktur Pentium 4 mit Hyperthreading [20]

Die wesentlichen Einheiten, die hierbei doppelt ausgelegt sind, bestehen aus den µOp-Queues, den Registersätzen (einschließlich Renaming), den Queues vor dem Scheduling, dem Store-Buffer und dem Reorder-Buffer, der Rest wird gemeinsam genutzt. Dies führt zu einem geringen Overhead im Silizium, wie Bild 8.11 deutlich zeigt.

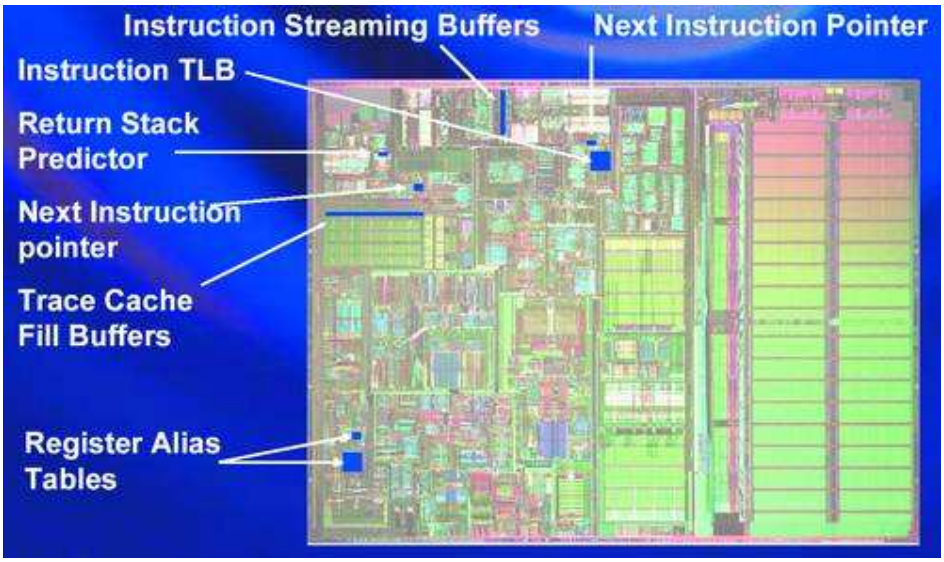
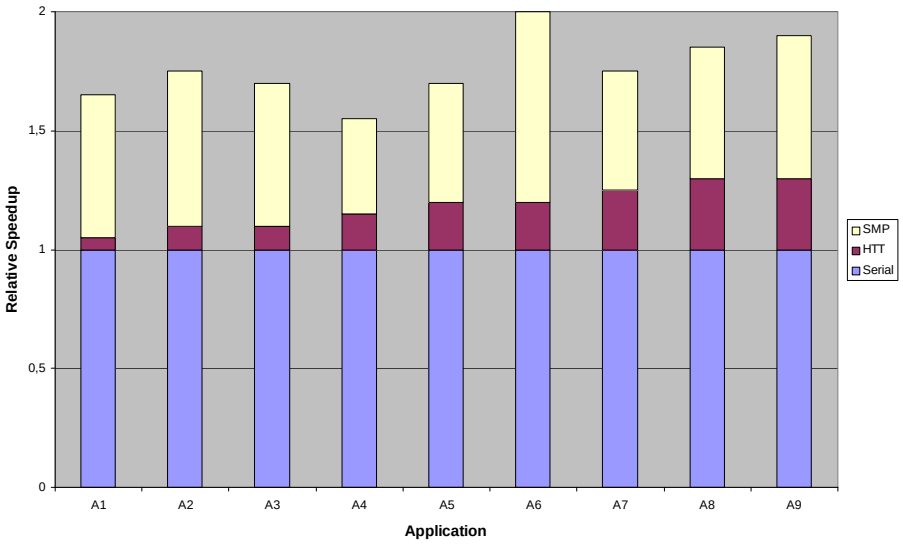


Bild 8.11 Layout Pentium 4 mit Hyperthreading (Quelle: [20])
Die neuen Teile sind blau markiert.

Probleme bei der Ausführung zweier Threads und einer damit erhofften Performancesteigerung treten in der Praxis dann auf, wenn die Zugriffe auf den Speicher sich gegenseitig behindern. Dies tritt in Erscheinung, da der Prozessor die virtuellen Adressen für Einträge im L1-Datencache unabhängig von dessen Größe nur mit 20 Bits gespeichert, also jedes MByte eine Wiederholung vorkommt. Tritt jetzt also der Fall ein, dass die Speicherbereiche der beiden Threads um exakt ein Vielfaches von einem MByte auseinander liegen (was beim Stack sehr schnell der

Fall sein kann), dann verdrängen sich die Threadzugriffe gegenseitig aus dem Cache.

Ansonsten ergibt sich aus der Zweiteilung mit geringem Overhead eine signifikante Performancesteigerung (Bild 8.12). Hier wird die Beschleunigung gegenüber einem einzigen Prozessor gemessen, indem das Hyperthrading (SMT) sowie ein Symmetric Multiprocessing (SMP) mit zwei gleichartigen Prozessoren eingeführt wird.



A1	Engineering	A6	Genetics
A2	Genetics	A7	CFD
A3	Chemistry	A8	FEA
A4	Engineering	A9	FEA
A5	Weather		

Bild 8.12 Performancesteigerungen Hyperthreading [20]

9 Speichertechnologie und Speicherhierarchien

Der Speicher als zentrales Element im Von-Neumann-Modell hat im Laufe der Zeit drei wesentliche Weiterentwicklungen erfahren: Diese betreffen die Speichertechnologie (→ 9.1), die Speicherhierarchie (→ 9.2) und das Speichermanagement (→ 9.3).

In diesem Kapitel werden also wesentliche Eigenschaften von Speichersystemen behandelt, wobei die Einschränkung auf den sogenannten Hauptspeicher, also einem Ausschluss der Massenspeicher liegt. Bild 9.1 gibt einen Überblick zu den Speicherfamilien (in vereinfachter Version).

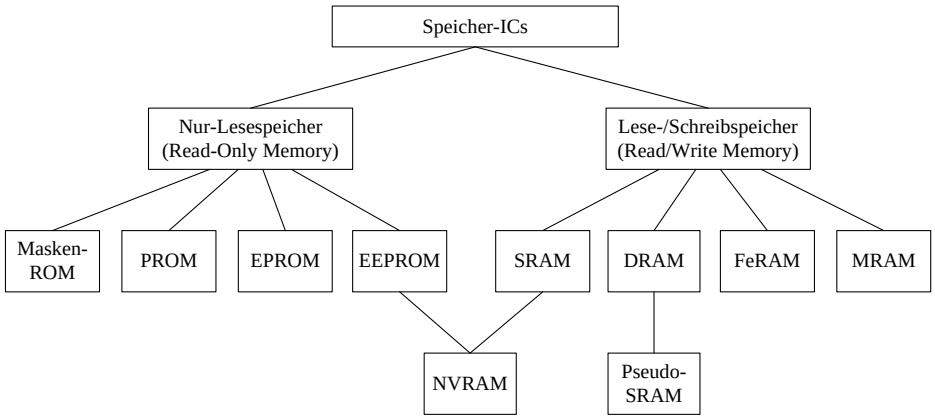


Bild 9.1 Speicherbausteinfamilie

9.1 Speichertechnologien und Speicherbausteinstrukturen

Literaturhinweise: [21, Kapitel 9], [7, Kapitel 5]

9.1.1 Dynamische RAMs (DRAM)

RAMs, die Abkürzung steht - etwas missverständlich - für "Random Access Read/-Write Memory", sollen die Fähigkeit besitzen, einen Speicherinhalt durch einen Schreibvorgang zu definieren und diesen bei jedem Lesevorgang entsprechend auszugeben, bis die Speicherung durch einen erneuten Schreibvorgang verändert wurde. Im Gegensatz zu den Programmiervorgängen bei EPROMs und EEPROMs

verlaufen Lese- und Schreibvorgängen in vergleichbaren – meist sogar identischen – Zeiten ab.

Diese Eigenschaft besagt noch nichts über das Einschaltverhalten bzw. eventuelle Datenverluste bei Spannungsausfall, gewöhnliche RAMs verlieren jedoch ihren Speicherinhalt und zeigen beim Einschalten der Versorgungsspannung einen undefinierten Zustand.

Die einfachste Methode zur Speicherung eines Binary-Digit-(BIT-)Werts ist die der Ladungsspeicherung in einem Kondensator. Dieses Verfahren wird in größtem Maß dadurch angewandt, daß der Kondensator durch einen MOS-Transistor eben mit ladungsspeichernder Wirkung ersetzt wird. Bild 9.2 zeigt das Prinzip:

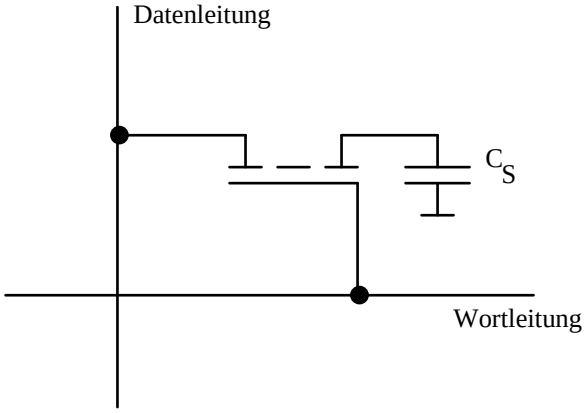


Bild 9.2 Prinzip der DRAMs

Der Speicherkondensator C_S wird durch die Kapazität der Drainzone mit der Substratschicht gebildet. Die dabei erzielten Kapazitäten liegen im Bereich 10^{-15} Farad (fF, femtoFarad), wobei ein geladener Kondensator den Wert 1, ein ungeladener den Wert 0 repräsentieren soll.

Die Speicherzelle wird mittels der Wortleitung angesprochen, die eine Lese- wie Schreibkopplung zwischen der Datenleitung und dem Kondensator herstellt. Im Lesezyklus "treibt" der Kondensator die Datenleitung bei entsprechender Ladung, im Schreibzyklus wird die vorhandene Ladung überschrieben. Dieses Treiben bedeutet die Erzeugung eines Stromimpulses in einem Leseverstärker, der diesen in eine zuverlässige 0/1-Information umsetzen muss; der Stromimpuls (bei vorhandener Ladung) ist schwach ausgeprägt und wird mit zunehmender Integrationsdichte (256 MBit- und 1 GBit-Chips!) durch die parasitären Leitungskapazitäten bei geringerwerdenden Kapazitäten der MOS-Strukturen immer schwächer, so dass der Aufwand für den Leseverstärker nicht unerheblich ist.

Konstruktionsbedingt verliert die Speicherzelle ihre Ladung und damit Information auf zwei Arten:

- Der Lesevorgang bewirkt zwar ein Treiben der Datenleitung, dies bedeutet jedoch zugleich den Abfluss der Ladung.
- Leckströme lassen eine Ladung auch ohne Lesevorgang abfließen

Als Konsequenz daraus muss die Information in den Speicherzellen ständig erneuert werden, also durch ein Rückschreiben der Ladung unmittelbar nach dem Auslesen sowie ständig (ca < 2 ms) durch einen zyklischen Vorgang, Refresh genannt. Aus diesem dauernden Lese-/Schreibvorgang resultiert auch der Name der ICs, dynamische RAMs.

Der externe Aufbau von DRAMs ist beispielhaft in Abb. 2-3 dargestellt.

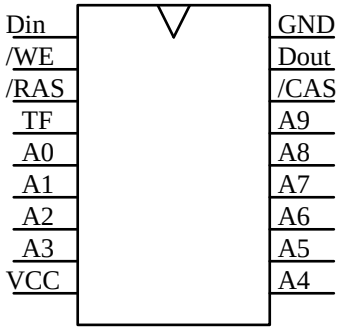


Bild 9.3 Pinout eines 1M * 1-Bit DRAM 21010

Die Bedeutung der Pins im Einzelnen:

A0 bis A9

gestatten das Anlegen einer 10.Bit Adresse, die, da zur Adressierung von 1 MBit 20 Adreßleitungen notwendig sind, zweifach angelegt werden muss. Die dafür notwendigen Steuersignale lauten

RAS und CAS,

die Row Address Strobe (Zeilenadressteil, zumeist zuerst übertragen) und Column Address Strobe (Spaltenadressteil) bedeuten und teilweise die Übernahme in Zwischenregister innerhalb der Bausteins bewirken..

/WE

signalisiert im aktiv low Zustand einen Schreibvorgang,

D_{in} und D_{out}

sind die Datenleitungen für Schreiben und Lesen einer Speicherzelle im IC.

TF

bedeutet Test Function, hat also keine Relevanz im Betrieb und kann offengelassen werden.

VCC und GND

stellen die entsprechenden Versorgungspins für den IC dar.

Intern sind DRAMs in einer Matrix-artigen Form für das sogenannte Speicherzellenfeld aufgebaut, was aus dem Blockschaltbild in Bild 9.4 hervorgeht:

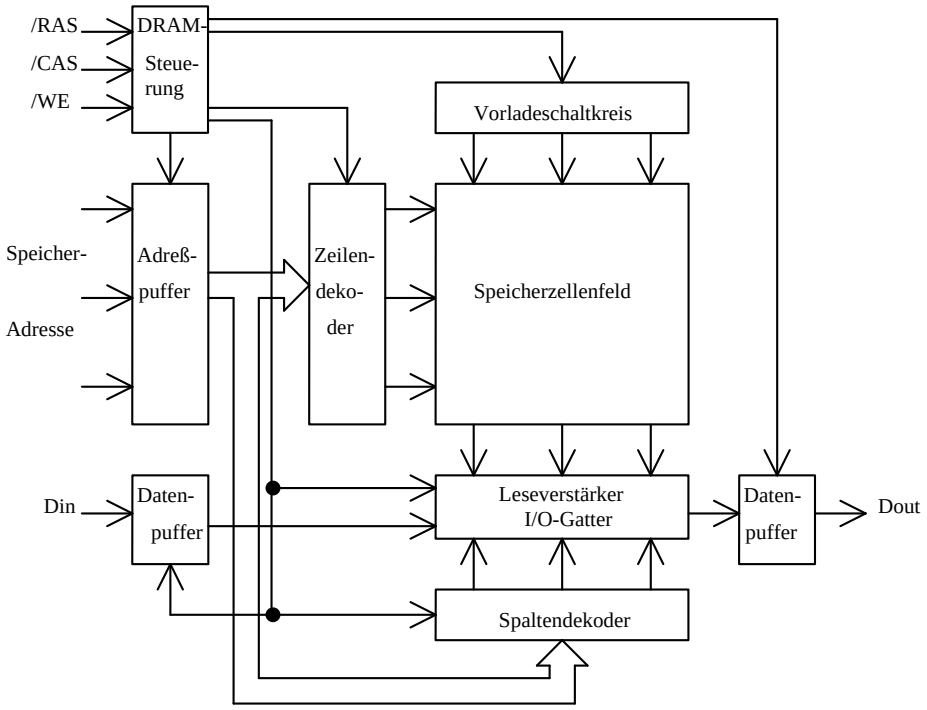


Bild 9.4 Blockschaltbild DRAM

Das Speicherzellenfeld besteht aus individuell adressierbaren ("Random Access") Speicherzellen meist á 1- oder 4- Bit. Wie bereits erwähnt, wird für einen Baustein mit 1M Speicherzellen (1 048 576) hierfür 20 Adressenleitungen benötigt, die beim DRAM in 2 * 10 mit sukzessiver Übertragung aufgeteilt sind.

Der zeitliche Verlauf ist in qualitativer Form in Bild 9.5 dargestellt. Zunächst wird die Zeile (Row) adressiert und mittels des /RAS-Signals in den internen Puffer übernommen. Mit dem anschließenden zweiten Adressteil und dem /CAS-Signal beginnt der Lese- oder Schreibvorgang, je nach Signal /WE.

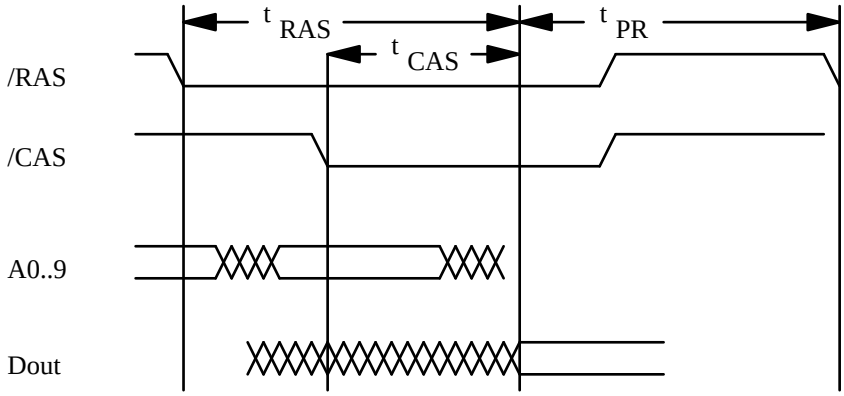


Bild 9.5 Qualitative zeitliche Verläufe beim DRAM-Zugriff (lesend)

Die in Bild 9.5 angegebenen Zeiten, t_{RAS} für die gesamte Zugriffszeit auf eine Speicherzelle, t_{CAS} für den Zugriffsabschnitt für die Spalte und t_{PR} für die Vorladezeit des Row Address Strobes (Preload), haben eine große Bedeutung für den gesamten Zugriff eines Prozessors auf den Speicherinhalt. Die gesamte Zykluszeit beträgt dabei

$$t_{Cyc} = t_{RAS} + t_{PR}$$

während die Herstellerangaben für den Zugriff zumeist nur t_{RAS} betreffen. Da als Faustregel t_{PR} etwa 80% von t_{RAS} beträgt, ist die gesamte Zykluszeit eines Lese- oder Schreibvorgangs etwa das 1,8fache der angegebenen Zugriffszeit. Für die angegebene Zugriffszeit von 70 ns bedeutet dies somit eine Zykluszeit von 126 ns; rechnet man hierzu eine Signallaufzeit, bedingt durch Gatterlaufzeiten etc. von ca. 20 ns, so ergibt sich eine maximale Frequenz von ca 14 MHz, mit der ein 80x86-Prozessor mit zwei Takten Zugriffszeit ohne Wartezyklen betrieben werden dürfte, bzw. von ca. 20 MHz bei einem Wartezyklus usw.

Diese drastische Reduzierung der Prozessorgeschwindigkeit bei Speicherzugriffen ist der entscheidende Grund für beschleunigende Maßnahmen, von denen im Folgenden drei behandelt werden sollen:

- Page Mode,
- Static Column und
- Interleaving

Unter **Page Mode** wird eine (lesende oder schreibende) Zugriffsart verstanden, die nicht mehr bei jedem Zugriff die Zeilen- und Spaltenadresse in das DRAM lädt, um dann eine Speicherzelle auszulesen, sondern nur noch die Spaltenadresse, während die Zeilenadresse konstant und damit auch das /RAS-Signal aktiv bleiben. Dieses Prinzip funktioniert natürlich nur unter mehreren Voraussetzungen:

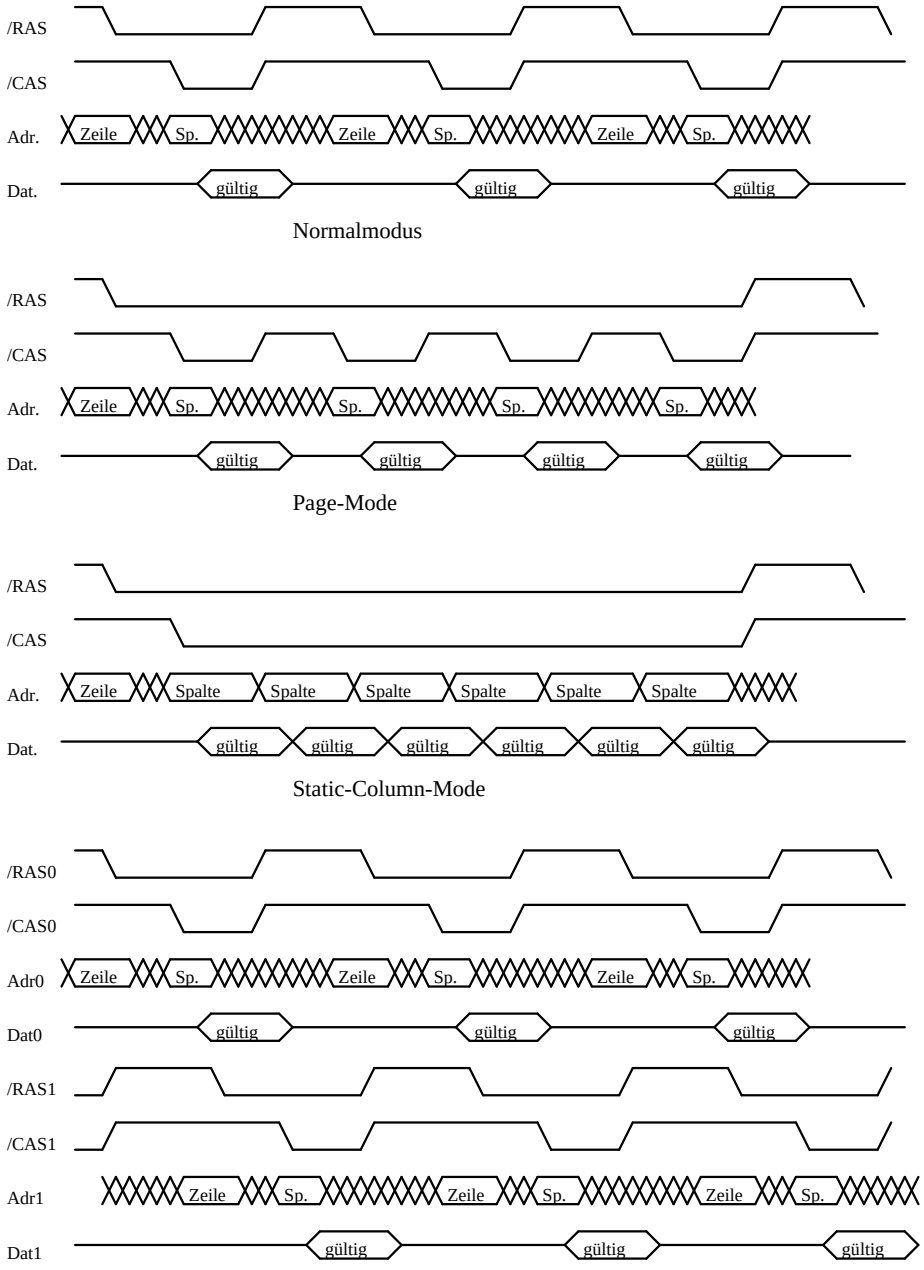


Bild 9.6 Zugriffsverfahren bei DRAM

- Die DRAMs müssen diesen Mode unterstützen, d.h., sie müssen den verkürzten Zugriff selbständig erkennen und nicht jedesmal einen vollständigen Zyklus erwarten.
- Die Steuerung der DRAMs – der Prozessor kann unter diesen Umständen und auch aus anderen Gründen, s.u., nicht direkt auf die DRAMs zugreifen – muss einen Zugriff mit identischer Seitennummer bzw. Zeilenadresse erkennen und den Zugriff entsprechend steuern.
- Das Programm bzw. Programmteile müssen innerhalb der Page ablaufen (Lokalitätsprinzip, auch Daten und Code gemischt!), was aber durch empirische Messungen mit großer Wahrscheinlichkeit bestätigt wurde. Die Seitengröße eines Page-Mode-DRAM mit einer Kapazität von 1 MBit beträgt 1024 Adressen, bei einem 16 MBit-DRAM bereits 16384 Adressen, worin hinreichend Platz für die Lokalität (mit eventueller Ausnahme von Datenzugriffen) liegt.

Die Reduzierung der Zugriffszeit für jeden sukzessiven Zugriff beträgt ca 50%, für die Zykluszeit ca 70%. Diese bedeutet, dass alle nachfolgenden Zugriffe auf die erwähnten 70 ns DRAMs mit einem Prozessortakt von 33 MHz ohne Waitstates ablaufen können. Die Anzahl der Zugriffe mit konstanten Zeilenadresse ist durch interne Vorgänge im DRAM begrenzt, die Grenze liegt bei ca 20 (Bausteinabhängig), wobei dann ein kompletter Zugriff wiederum notwendig wird.

Im **Static-Column-Mode** geht man noch einen Schritt weiter, indem auf die /CAS-Steuerung noch verzichtet und lediglich die Spaltenadresse jeweils für den nächsten Zugriff geändert wird. Es obliegt damit der internen DRAM-Steuerung, die Änderungen zu erkennen, Zwischenspeicherung etc. mittels des /CAS-Signals und deren Laufzeiten werden eingespart, wobei der neuerliche Gewinn nicht mehr so drastisch zu bewerten ist.

Interleaving wird eine aus dem Großrechnerbereich übertragene Aufspaltung des Speicherbereichs in mehrere Speicherbänke genannt. Der Vorteil dieses Verfahrens ist darin zu sehen, dass der einzelne Zugriff auf eine Bank mit der normalen Geschwindigkeit ablaufen kann, während der nachfolgende, sofern er eine andere Bank betrifft, bereits beginnen kann. Im PC-Bereich wird häufig das 2-Wege-Interleaving genutzt, was bedeutet, dass sich zwei Speicherbänke im PC befinden und dass man hofft, zwei aufeinanderfolgende Speicherzugriffe (für Daten oder Code) betreffen nacheinander die beiden Speicherbänke.

Bild 9.6 fasst die drei beschriebenen Zugriffsverfahren im Vergleich zum normalen Zugriff zusammen.

Neben dem Lese- bzw. Schreibzugriff ist noch der bereits erwähnte Refresh, also das ständige Auffrischen der Ladungsinhalte und somit der Informationen, zu behandeln. Dieser Refresh wird zusätzlich – auch mit der Gefahr der Zugriffsbehinderung der CPU – zu den normalen Zugriffe durchgeführt. Die dabei gebräuchlichste Art ist der RAS-only-Refresh, bei dem eine Zeilenadresse angelegt und dem DRAM mittels /RAS mitgeteilt wird, der angedeutete Zugriff aber durch Ausbleiben der Spaltenadresse und vor allem des /CAS-Signals aber nicht beendet

wird. Die interne Logik des DRAM leitet dann ein Auslesen und Wiedereinschreiben aller Werte der Refreshzeile (also im Beispiel des 1 MBit DRAM von 1024 Zellinhalten) ein, eben den Refresh. Diese Art des Refresh muss für einen kompletten Durchgang 1024mal ausgeführt werden, z.B. innerhalb des Zeitraums von 2 Millisekunden, was in einem 80x86-System bei 10 MHz Prozessortakt und Zugriffen ohne Wait States, also mit 2 Taktzyklen, zu einer Busbelastung von 1% führt.

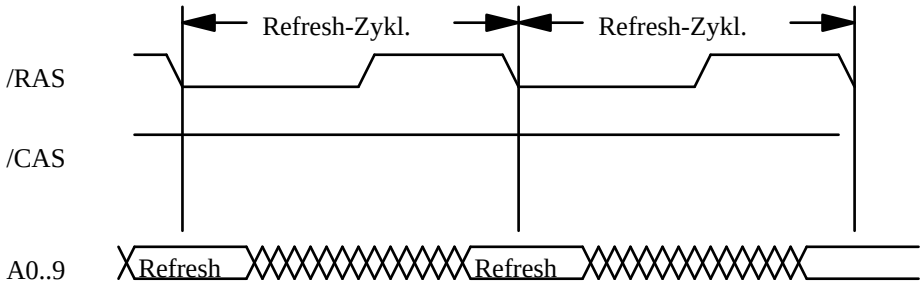


Bild 9.7 RAS-only-Refresh bei DRAM

Trotz der erwähnten, konstruktionsbedingten Nachteile gerade im Bereich der Zugriffszeiten werden dynamische RAMs heute als der Standardspeicher in den meisten Computern eingesetzt. Die Gründe dafür sind:

- Baulich kleine Form durch gemultiplexten Adressbus
- Intern geringer Aufwand duch 1 Transistor pro Speicherzelle, daher größte Kapazität pro Technologieschritt

Die Ausführung in Computern erfolgt zumeist in der Breite des Datenbusses der CPU, ergänzt durch Paritätsbits, die – gemäß der Hamming-Distanz von 2 – bei einfacher Parität die Erkennung von Fehlern, nicht aber deren Korrektur erlauben. SIMM und SIP-Module in PCs sind ein gutes Beispiel für diese Technologie, sie werden in der Organisationsform 1M * 9 bzw. 4M * 9 usw. angeboten, also in 9 Bit Breite.

9.1.2 Statische RAMs (SRAM)

Während die DRAMs quasi zu den Massenprodukten unter den Schreib-/Lesespeichern gehören, sind SRAMs - technologiebedingt - teurer, in vielen Einsatzfällen aber unabdingbar. Der interne Aufbau von SRAMs besteht aus Daten-Flipflops, einer speziellen Version von RS-Flipflops mit einer entsprechenden Zugriffslogik.

Bild 9.8 zeigt den Aufbau eines Daten-FF mittels AND-, NOT- und NOR-Gattern; die tatsächliche Implementierung besteht natürlich aus Transistoren direkt (Bild 9.9), weniger aus der Zusammenschaltung von Gattern.

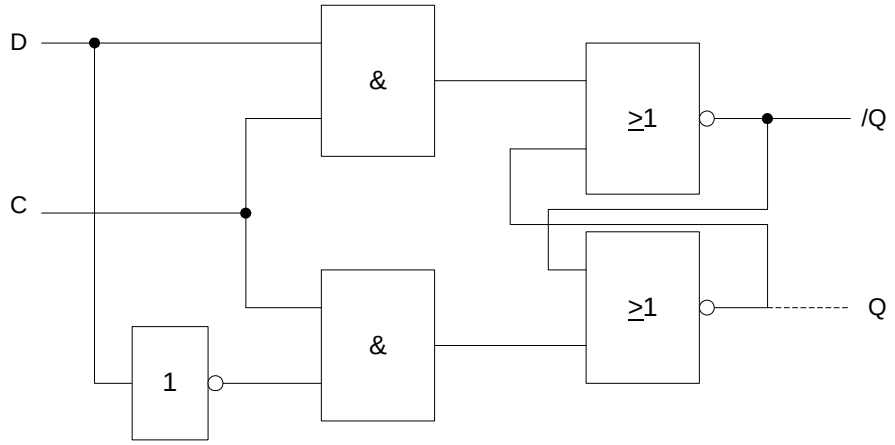


Bild 9.8 D-Flipflop mit Zustandssteuerung

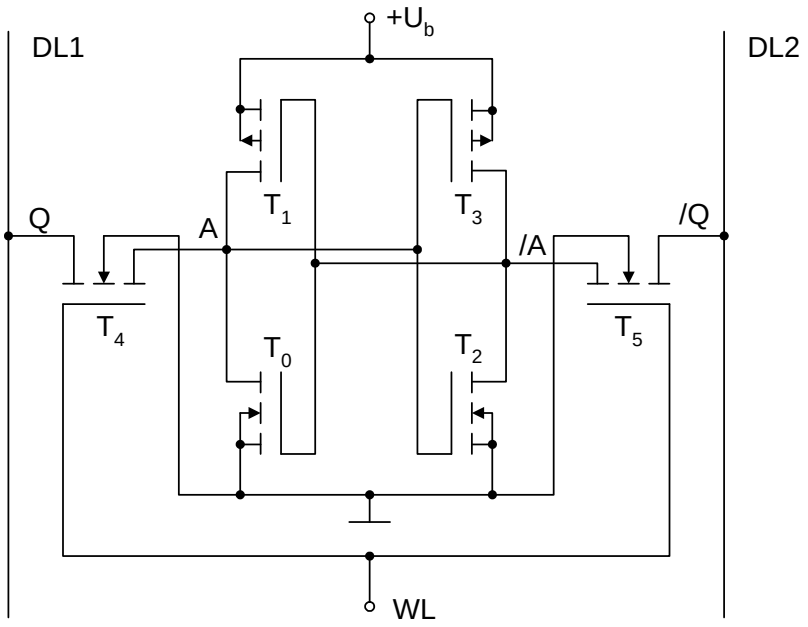


Bild 9.9 RAM-Speicherzelle, CMOS-Technologie

Das zentrale Flipflop, auch bistabile Kippstufe genannt, befindet sich an der rechten Seite und wird von den beiden NOR-Gattern gebildet. Bezeichnet man die

Eingänge mit S und R, so sind folgende Wertepaare zulässig und führen zu folgenden Ausgängen:

S	R	Q	/Q
0	0	(Zustand bleibt erhalten)	
1	0	1	0
0	1	0	1
1	1	(Instabil, daher nicht erlaubt)	

Tabelle 9.1 Zustände und Übergänge beim RS-Basisflipflop

Im zustandsgesteuerten Daten-FF wird das nicht-erlaubte Eingangswertepaar '11' durch die Beschaltung S = /R unterdrückt. Dies würde für sich genommen aber niemals einen zeitinvarianten Zustand am Ausgang ergeben, Q folgte ohne weitere Zusätze ständig dem Dateneingang D.

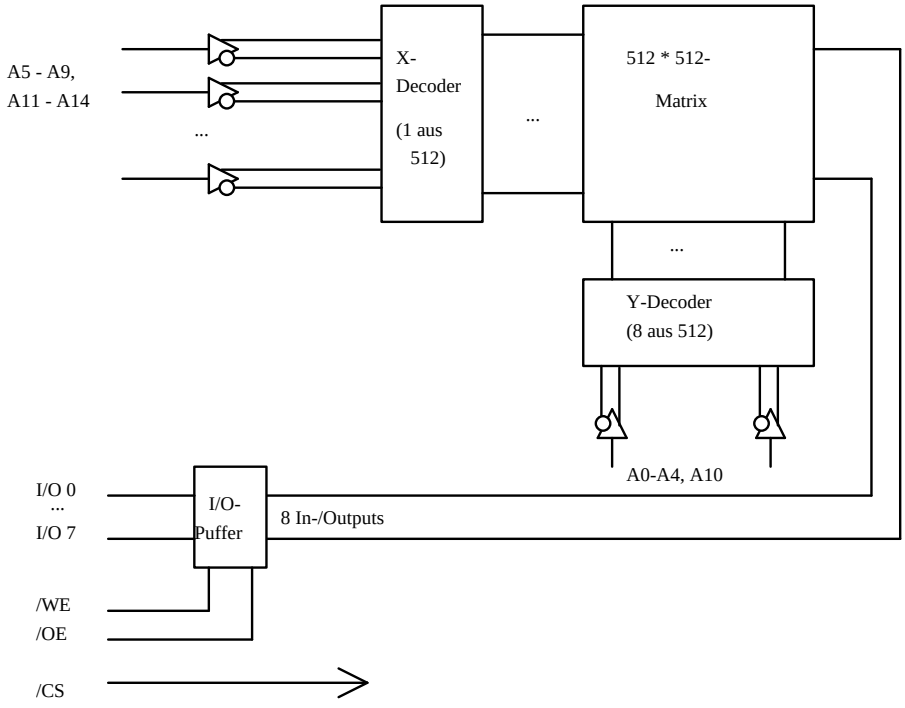


Bild 9.10 Blockschaltbild SRAM 62256

Die zeitliche Invarianz ergibt sich durch den Freigabeeingang C, der mittels seines Zustands das Verhalten des Basisflipflops steuert. Solange C den Wert 0 annimmt, kann das RS-FF seine Ausgänge nicht verändern; mit C = 1 wird der Ausgang Q transparent auf den Wert von D geschaltet, /Q (zumeist nicht herausgeführt) entsprechend auf /D. Diese Speicherzelle benötigt – solange kein Versorgungsspannungsabfall eintritt – keinerlei Refresh, worin einer der wesentlichen Vorteile der SRAMs liegt.

Die Blockstruktur von SRAMs, die zumeist als 4- (Nibble) oder 8-Bit (Byte) SRAMs organisiert sind, ist in Bild 9.10 dargestellt. Die eingezeichneten Adressen A0 bis A14 werden innerhalb des SRAMs nicht mehr zwischengespeichert, sondern müssen gleichzeitig – und über die Zugriffsdauer stabil – am IC anliegen. Je nach Zugriffsrichtung werden dann Ausgangs- oder Eingangspuffer aktiv geschaltet, so dass die entsprechende Operation ablaufen kann.

Die nicht-gemultiplexten Adressen bewirken, daß diese auch am Pinout sichtbar sein müssen. Bild 9.11 gibt für die 32 KByte (32K * 8 Bit) Bausteine einen Überblick zum Pinout, wobei die scheinbar unsortierte Anordnung insbesondere des Adressenteils historische Gründe hat. Ein 32 KByte SRAM wird z.B. im 28poligen DIP-Gehäuse geliefert und hat dann wesentlich größere Ausmaße im Vergleich zu DRAMs.

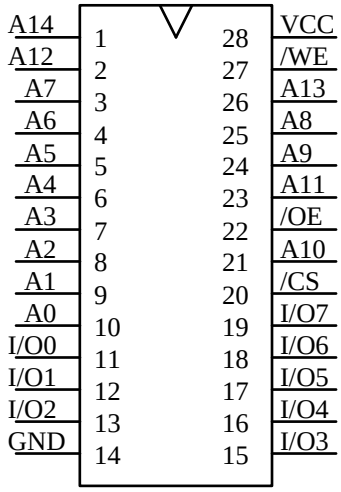


Bild 9.11: Pinout des SRAM 62256

Die Bedeutungen in Bild 9.11 der Abkürzungen lauten:

A0 bis A14

stellen der Adressbus dar, wobei zur Adressierung von 32 kByte 15 Adressleitungen auch tatsächlich notwendig sind.

I/O0 bis I/O7

häufig auch mit D0 bis D7 bezeichnet, werden mit dem Datenbus verbunden und zeigen ein Input- der Output-Verhalten, je nach Belegung von /WE.

/CS

selektiert den Baustein, unabhängig von der Zugriffsrichtung (Chip Select)

/OE

signalisiert in Zusammenhang mit /CS einen Lesevorgang (Output Enable)

/WE

entspricht in Zusammenhang mit /CS einem Schreibvorgang (Write Enable)

SRAMs benötigen in der tatsächlichen Realisierung pro Speicherzelle 6 Transistoren, DRAMs – je nach Adressierung – 1 bis 3. Dies erklärt, warum SRAMs in der jeweiligen Technologiestufe immer um einen Faktor von ca 4 weniger Speicherkapazität bieten und somit entsprechend teurer sind, zumindest pro Speicherzelle.

Andererseits bieten SRAMs deutliche Vorteile:

- mit Ausnahme der Dekodierung werden keine externen Zusatzschaltkreise wie Refreshgeneratoren benötigt
- Batteriepufferung zur Langzeitspeicherung von Daten ist möglich
- Deutlich schnellere Zugriffs- und Zykluszeiten bis zu 10 ns sind möglich, da kein Adressmultiplexing auftritt und die SRAMs aufgrund ihres internen Aufbaus wesentlich stärkere Speichersignale mit schnelleren Ansprechzeiten der Puffer bieten

Die Vorteile zeigen bereits die Einsatzgebiete von SRAMs:

- Kleine, Einplatinencomputer mit möglichst geringem Hardwareaufwand
- Zusatzspeicher zur batteriegepufferten Speicherung von Konfigurationsdaten
- Schnelle Speicher, z.B. für Cache-Speicher zur Beschleunigung von Programmabläufen.

9.1.3 Nur-Lesespeicher (PROM, EPROM, EEPROM)

Ein ROM, ein Read-Only-Memory, hat seinen Namen durch die Eigenschaft in Computersystemen, nur auslesbar, aber nicht beschreibbar zu sein, bekommen.

Diese Eigenschaft impliziert natürlich sofort, dass der Speicherinhalt zu jedem Zeitpunkt, also auch bei Verlust der Versorgungsspannung, eindeutig definiert sein muss, ansonsten wären diese Bausteine unbrauchbar.

Reine ROMs, die also bei der Herstellung bereits den späteren Inhalt mitgeteilt bekommen, sind aus einem leicht ersichtlichen Grund sehr selten geworden:

Der Herstellungsprozess, es handelt sich schließlich um einen Anwender-spezifischen IC, ist für Einzelexemplare zu teuer, eine gewisse Serie muss sofort aufgelegt werden. Das Programm, das in solche ICs produziert wird, muss aber außerordentlich gut ausgetestet sein und es muss zudem absehbar sein, dass keine Zusätze in absehbarer Zukunft hinzukommen sollen (Fehlerfreiheit und Zukunftssicherheit). Beide Probleme sind aber in Zusammenhang mit Computern schwer lösbare Probleme.

Nicht-löschbare ROMs werden daher heutzutage als Masken-ROMs (mit einer Metallmaske, die erst im letzten Herstellungsgang einer ansonsten für alle ROMs einheitlichen Fertigung) bei Massenproduktion oder als PROMs, die also nach der Herstellung einmal programmierbar sind, gefertigt. Diese PROMs, meist in bipolarer Technologie aufgebaut, haben intern Schmelzsicherungen eingebaut, die einmalig per Überspannung durchgebrannt werden können:

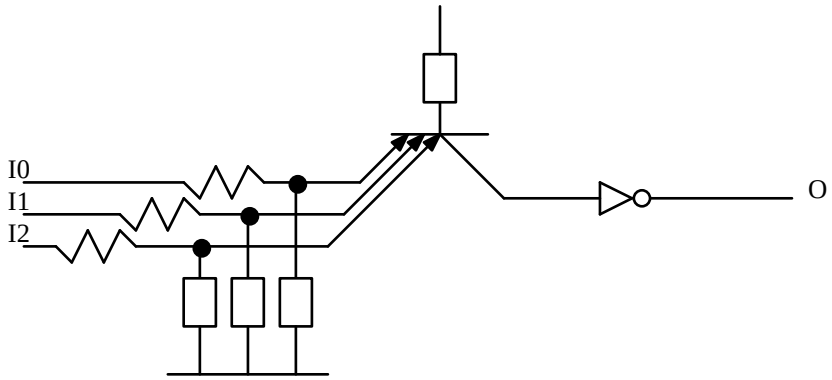


Bild 9.12: Aufbau ODER-Zelle mit pnp-Multi-Emitter-Transistor

Die in Bild 9.12 dargestellte ODER-Verknüpfung von maximal 3 Eingängen(I_x) bewirkt einen High Pegel am Ausgang O, falls einer der Eingänge auf high liegt und dessen Sicherung intakt ist, ansonsten low. Im ersten Fall wird die Basis-Emitter-Diode leitend und der Transistor schaltet durch, so dass das Potential vor dem Inverter auf high liegt. Dies ist exakt das Prinzip der PROMs, wobei die Sicherung während des Programmiervorgangs durchgebrannt werden.

Sehr häufig im Einsatz sind seit vielen Jahren **EPROMs**, **Flash-EEPROMs** und **EEPROMs** (Erasable Programmable Read Only Memory bzw. Electrically Erasable Programmable Read Only Memory). Diese Speicher-ICs beruhen auf einer

gemeinsamen Technologie, dem sog. Floating Gate; innerhalb eines MOS-Transistors stellt dies ein zusätzliches Gate dar, das im Prinzip isoliert angebracht ist, auf dem aber während des Programmiervorgangs Ladungen aufgebracht werden können (insbesondere durch den sogenannten Tunneleffekt). Bild 9.13 zeigt eine solche EPROM-Zelle im unprogrammierten Zustand, Bild 9-14 während des Programmiervorgangs:

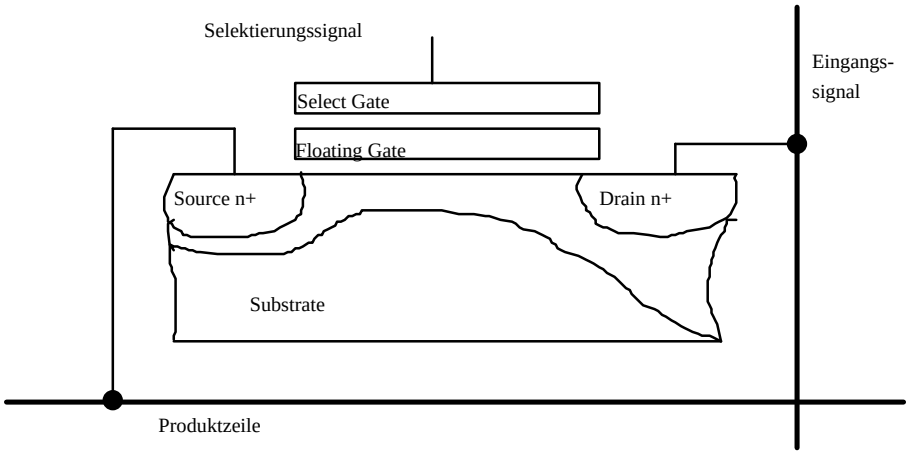


Bild 9.13: EPROM-Zelle im unprogrammierten Zustand

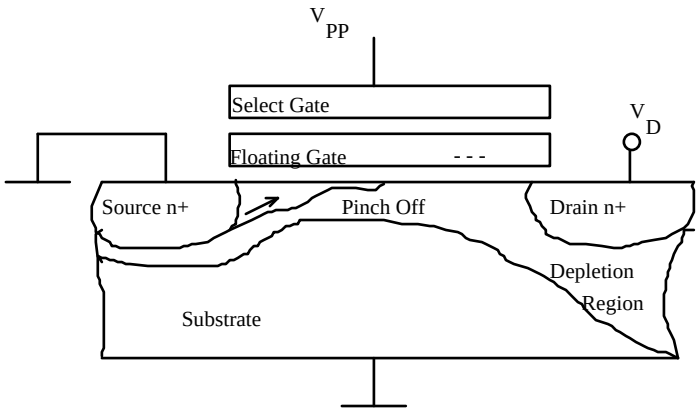


Bild 9.14: EPROM-Zelle während des Programmiervorgangs

Während des Programmiervorgangs wird durch eine (bei EPROMs extern angelegte, bei EEPROMs intern erzeugte) Überspannung V_{PP} (z.B. 12,5 V) ein elektrisches Feld erzeugt, das in der Lage ist, Elektronen mit einer gewissen Wahrschein-

lichkeit durch die das Floating Gate isolierende Umgebung diffundieren zu lassen und so dort Ladung anzusammeln.

Nach der Programmierung bewirkt ein geladenes (also programmiertes) Floating Gate, dass keine Kopplung zwischen Source und Drain via Select Gate stattfindet; dies ist genau die Wirkung einer gelöschten Fuse, die ein Eingangssignal ebenfalls nicht durchlässt und somit am Ausgang - für sich genommen - nur ein '0'-Signal mitliefert.

Das Löschen der Programmierung geschieht auf dreierlei Weisen, hierin unterscheiden sich die drei Bausteintypen:

- Bei EPROMs wird mittels UV-Licht bei 253,7 nm (Hg-Linie) gelöscht, sie besitzen daher ein Quarzfenster. Das UV-Licht beschleunigt die Elektronen so stark, dass sie das Floating Gate wieder verlassen können (lichtelektrischer Effekt)
- Bei EEPROMs, die intern sehr viel komplizierter und auch größer aufgebaut sind, werden die Ladungen durch 'umgepolte' elektrische Felder aus dem Floating Gate abgezogen, und zwar auf Zellenbasis.
- Flash-EPROMs sind ähnlich zu EPROMs aufgebaut, gleichwohl werden sie elektrisch gelöscht, allerdings nur blockweise oder sogar im gesamten IC.

Trotz der Wiederbeschreibbarkeit gelten – im Gegensatz zu SRAMs – die EPROMs und EEPROMs nicht als Schreib-/Lesespeicher, da die Programmierung einen erheblich anderen Betriebsmodus mit wesentlich längeren Zugriffszeiten im Vergleich zum normalen Lesen darstellt.

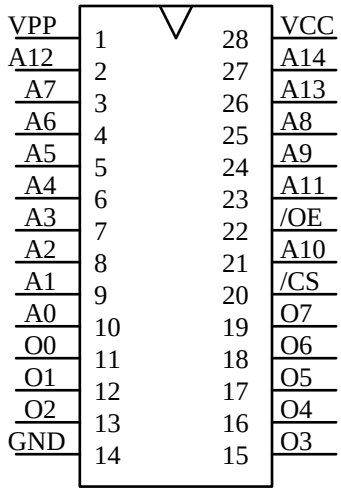


Bild 9.15: Pinout des EPROM 27256

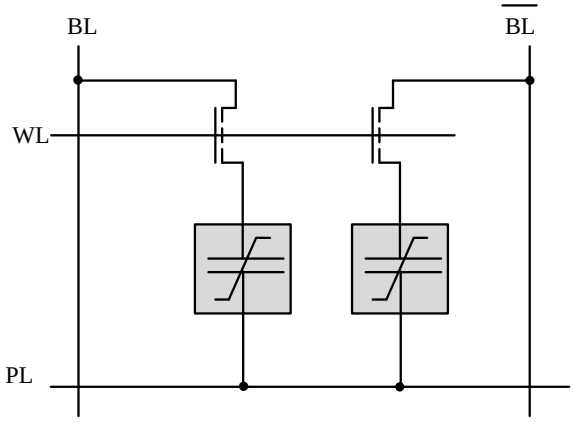
Die drei erwähnten Festwertspeicher haben gemeinsam, dass sie sowohl programmierbar als auch löschbar sind. Daneben gibt es sie in identischen Bauformen und Anschlussbelegungen; ein typisches EPROM, 27256, organisiert als 32K * 8 Bit, ist in Bild 9.15 dargestellt. Man beachte hierbei die geringen, aber merklichen Unterschiede zum SRAM 62256!

9.1.4 Ferroelektrische RAMs

Eine bereits seit 1987 bekannte Bauform von Speicher ist in der Zwischenzeit auch lieferbar geworden: Ferroelektrische RAMs. Bei dieser Bausteinfamilie beruht der Speichereffekt nicht auf Ladungsträger, sondern auf dem ferroelektrischen Effekt, der ähnlich wie ein ferromagnetischer Effekt (”Dauermagnet“) wirkt – eben auf elektrische und nicht magnetische Weise.

Dieser Effekt, der nur in wenigen Materialien nachweisbar ist, liegt in der Eigenschaft von Kristallstrukturen begründet, die eine durch ein äußeres elektrisches Feld hervorgerufene Polarisierung (Ausrichtung) auch nach dem Abschalten beibehält. Diese Ausrichtung wird als (nahezu idealer) Informationsspeicher genutzt, da nunmehr die Speicherung auch bei vollständigem Verlust der Betriebsspannung erhalten bleibt und trotzdem jederzeit änderbar ist. Dies ist durch eine entsprechende Molekülstruktur möglich.

Bild 9.16 zeigt den Aufbau der 2T/2C-FRAM-Zellstruktur. Diese Doppelstruktur ist zwar prinzipiell nicht notwendig, zurzeit jedoch in Gebrauch (neuerdings sind auch bereits 1T/1C-Architekturen erhältlich). Im Unterschied zur DRAM-Struktur liegt der Kondensator hier nicht auf Masse, sondern auf eine Plateline, um die Umprogrammierung zu ermöglichen.



BL: Bitleitung
WL: Wordleitung
PL: Plateleitung

Bild 9.16 Aufbau FRAM-Zelle

9.1.5 Magnetoresistive RAMs

Ursprünglich war es die magnetische Technologie, die eine Speicherung von zweiwertigen Zuständen ermöglichte. Diese Form wurde als Kernspeicher bezeichnet und führte zu sehr aufwendigen, weil aus kleinen Spulen zusammen gesetzten, Speicher.

Die neue Form der magnetischen Speicherung unterscheidet sich hiervon grundlegend, sie nutzt den magentoresistiven Effekt aus. Ein Strom, der senkrecht durch zwei magnetische Schichten geführt wird, ist von den Magnetisierungen dieser Schichten – insbesondere von den gegenseitigen Ausrichtungen – abhängig. Parallele Magnetfelder zeigen einen geringeren Widerstand als anti-parallele.

Bild 9.17 zeigt den Aufbau einer binärwertigen Zelle (ohne Auswertelogik). Die beiden magnetisierbaren Schichten sind durch eine dünne Isolierschicht voneinander getrennt. Die Referenzschicht bleibt in ihrer Orientierung immer erhalten, während die Programmierschicht durch den Programmierstrom I_P parallel oder anti-parallel magnetisiert werden kann. Zum Lesen wird ein kruzzeitiger Lesestrom (I_R) senkrecht zu den Schichten gemessen und so der aktuelle Widerstand bestimmt.

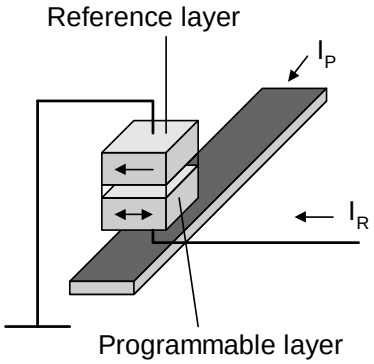


Bild 9.17 Aufbau einer magnetoresistiven Zelle

Die MRAM-Technologie soll ab ca. 2004 am Markt erhältlich sein. Da sie auf einer Speicherung der Informationen in einem Transistor beruht und zudem offenbar besser skalierbar ist als die DRAM-Technologie, wird ihr das Potenzial zur Mainstream-Technologie zugewiesen.

9.1.6 Zusammenfassung Speichertechnologien

Die folgende Tabelle fasst die Eigenschaften von FRAM, SRAM, DRAM, Flash-EEPROM und EEPROM zusammen.

	FRAM	MRAM	DRAM	SRAM	Flash	EEPROM
Relative Dichte	2-3	4	4	1	2	1.5
Nicht-Flüchtigkeit	Ja	Ja	-	(Pufferbat.)	Ja	Ja
Schreibgeschw. (pro Byte)	150-200 ns	10 ns (?)	50-100 ns	5-100 ns	10-1000 µs	5-10 ms
Anzahl Zyklen	10^{10}-10^{13} (W/R)	∞	∞	∞	$<10^6$ (W)	$<10^6$ (W)
Wesentl. Eigenschaft	geringe Leistungs-aufnahme	hohe Dichte, Geschwin-digkeit	hohe Dichte	Geschwin-digkeit	Große Speicher	Kleine nicht-flüchtige Speicher

Tabelle 9.2 Vergleich Speichertypen

9.2 Speicherhierarchien

Literaturhinweise: [7, Kapitel 5], [10, Kapitel 8]

Das besondere Problem der Speicher wird deutlich, wenn man sich die Wachstumszahlen der letzten Jahrzehnte einmal vor Augen führt. Die wesentlichen Komponenten und Eigenschaften zu den Mikroprozessoren – Anzahl der Transistoren pro Chip, dynamische Verlustleistung pro Transistor und Takt sowie der Takt selbst –, dann haben sich diese alle 18 Monate um den Faktor 2 verbessert (heißt: Taktfrequenz und Zahl der Transistoren erhöht, dynamische Verlustleistung erniedrigt).

Für den Speicher gilt dies ebenfalls, bis auf die Geschwindigkeit. Diese hat sich in den letzten Jahrzehnten – bezogen auf das im Wesentlichen eingesetzte DRAM – nur um 3% pro Jahr erhöht, viel zu wenig, um mit den Mikroprozessoren mithalten zu können. Dies bewirkt, dass ein DRAM-Zugriff im Hauptspeicher schon einige 100 Takte an Wartezeit mit sich bringen kann.

Die Lösung der Architekten heißt **Speicherhierarchie**. Hier wird ein kleiner Speicher, der stellvertretend für alle Speicherbereiche im Hauptspeicher arbeiten kann, mit so schnellem Silizium (auf SRAM-Basis) ausgestattet, dass dieser wieder im Prozessortakt ohne Wartezyklen arbeiten kann. Dieser Speicher wird als Cache bezeichnet:

Definition 9.1:

Ein **Cache** ist ein schneller Zwischenspeicher innerhalb der CPU oder in CPU-Nähe, der Befehle und/oder Operanden von Programmen bereithalten soll, auf die in einem bestimmten Zeitraum häufig zugegriffen werden muss. Jeder Cache besteht aus einem *Datenbereich* und einem *Tag-Bereich* (Identifikationsbereich)

Die Kapazität des Cache-Datenbereichs ist in der Regel deutlich kleiner als die Hauptspeicherkapazität. Der Tag-Bereich enthält den Teil der CPU-Adresse, der die Herkunft eines Cache-Eintrags aus dem Hauptspeicher eindeutig identifiziert. Für einen Cache sind *strukturelle Parameter*, eine *Organisationsform* und eine *Ersetzungsstrategie* erforderlich.

9.2.1 Cache-Hierarchieebenen

Bedingt durch die hohen Taktraten werden die sogenannten L1-Caches nur noch auf dem Mikroprozessorchip selbst integriert. Zudem bleibt ihre Größe beschränkt, da physikalisch große Strukturen (→ Bild 8.11) schnell zu größeren Verzögerungszeiten führen. Aus diesem Grund liegt die L1-Cachegröße meist bei 16 bis 64 KByte, allerdings getrennt für Instruktionen und Daten.

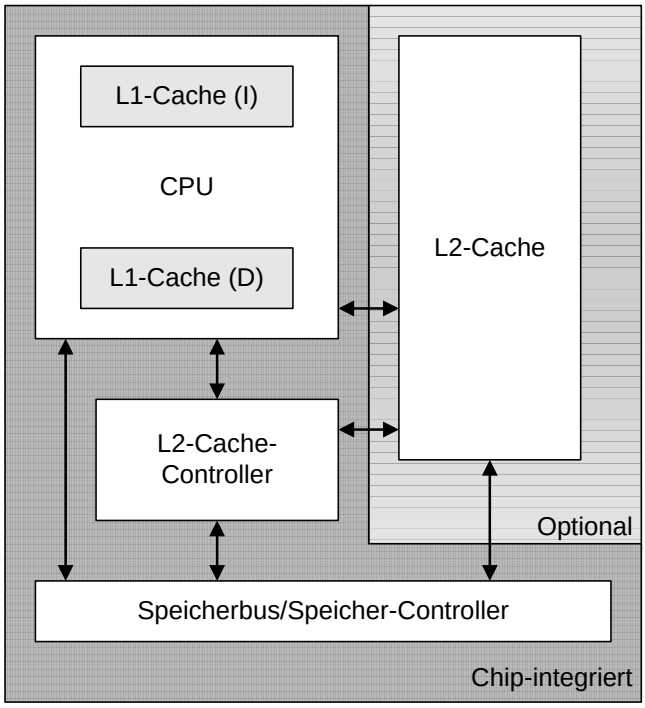


Bild 9.18 Mikroprozessorsystem mit integrierten L1-Caches (I, D) und gemeinsamen L2-Cache

Bild 9.18 zeigt den Aufbau eines modernen Mikroprozessors mit L1- und L2-Cache. Dieser L2-Cache gilt als zweite Ebene und ist wesentlich größer ausgelegt (ca. Faktor 8 .. 20), allerdings auch langsamer (d.h. mit mindestens 1 Wartezyklus, häufig jedoch mehr). Der L2-Cache hat damit typischerweise 256 KByte bis 4

MByte Speicherkapazität, fast immer gemeinsam für Daten und Instruktionen. Schließlich existieren insbesondere für Serversysteme mit hohen Speicheranforderungen auch L3-Caches. Während der L2-Cache auch On-Chip sein kann, gilt dies für den L3-Cache nicht mehr, hier wird nur das Interface integriert.

9.2.2 Cache-Organisation

Die Wahrscheinlichkeit, benötigte Daten oder Instruktionen im Cache zu finden, hängt von der Datenkapazität, der Hardwarestruktur und der Organisationsform des Caches ab. Bei vorgegebener Datenkapazität bestimmt die Organisationsform, in welche Cache-Einträge ein bestimmter Hauptspeicherblock der Größe *m* Bytes jeweils übertragen werden kann und wie er beim Zugriff identifiziert wird.

Definition 9.2:

Der Datenbereich eines Cache setzt sich aus Cache-Einträgen zusammen. Ein **Cache-Eintrag** (*Cache Line*) ist eine feste Zahl *m* von (im Hauptspeicher) aufeinanderfolgenden Daten- oder Befehlsbytes. Die Zahl *m* wird als *Eintragsgröße* (*Line Size*) bezeichnet.

Definition 9.3:

Jedem Cache-Eintrag ist ein **Index** (*Eintragsnummer*) zugeordnet, über den er angesteuert wird. Verweist ein Index auf mehrere Einträge, dann werden diese Einträge als **Set** (*Eintragsgruppe*) bezeichnet.

Definition 9.4:

Jedem Cache-Eintrag ist ein **Tag** (*Identifikator*) zugeordnet, das eindeutig festlegt, aus welchem Teil des Hauptspeichers der Cache-Eintrag stammt. Tags werden im Tag-Bereich des Cache abgespeichert.

Die Organisation des Cache-Speicher legt nun fest, an welcher Stelle im Cache die Werte aus dem Hauptspeicher gespeichert werden können, d.h., in welcher Art die realen Adressen des Hauptspeichers in den Cache abgebildet werden. Hierzu sei beispielhaft angenommen, im System existieren 1.048.576 Adressen (1 MByte, 20-bit-Adresse), und der Cache-Speicher hat exakt 1024 Speicherstellen á 1 Byte (1 KByte, 10-bit-Adresse). Soll also 1 Byte aus dem Hauptspeicher in einer Speicherstelle im Cache zwischengespeichert werden (die Line Size beträgt allerdings 4), muss nicht nur der Datenwert, sondern auch die Adressinformation angelegt werden, da 10 bit an Information fehlen.

Bild 9.19 zeigt die beiden Extremfälle auf, die die Cache-Organisation annehmen kann. Im ersten Fall wird jeder Hauptspeicheradresse exakt eine Cache-Adresse durch Beschränkung auf die Adressbits A2 .. A9 zugewiesen. Dies führt zu 256 Cachezeilen á 4 Byte, zusätzlich müssen die fehlenden Adressbits A10 .. A19 im Tag gespeichert werden. Das V-Bit (Valid) zeigt dabei an, ob die Zeileninformation gültig ist.

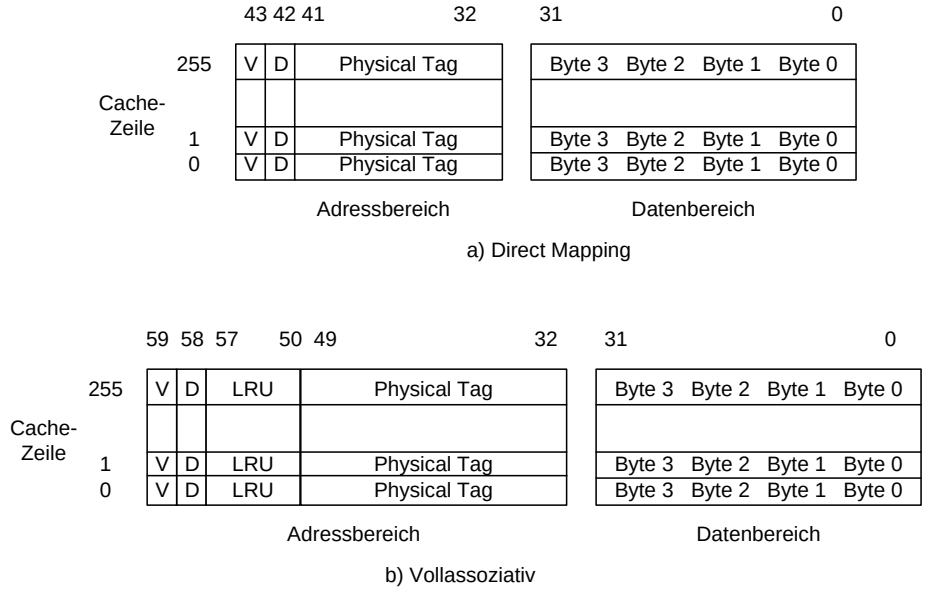


Bild 9.19 Cache-Organisation a) Direct Mapping b) Vollassoziativ

Bild 9.19b zeigt den vollassoziativen Fall, bei dem eine Speicherung des Datums oder Befehls an *jeder* Cache-Adresse möglich ist. Dies führt zu der Notwendigkeit, die komplette Adresse A2 .. A19 zu speichern, und weiterhin muss eine Strategie gewählt werden, welche Zeile als nächste ersetzt wird (→ 9.2.3).

Der vollassoziative Cache-Speicher ist natürlich viel flexibler, wird aber selten genutzt. Dies liegt daran, dass bei Suchen nach einem Cache-Hit nicht mehr eine, sondern alle Tag-Adressen parallel zueinander mit der Zugriffsadresse verglichen werden müssen, und der Schaltungsaufwand steigt enorm an (linear mit der Anzahl der Cachezeilen). Der in der Praxis oft gewählte Mittelweg findet man in der *k*-Wege-assoziativen Architektur, in der eine Hauptspeicheradresse an *k* Cache-Adressen zu finden ist. Der Wert von *k* ist meist 2 oder 4. Auch hier muss natürlich ein Ersetzungs-Algorithmus eingebaut werden, und es sind mehrere, zueinander parallele Vergleiche notwendig, aber eben nicht so viele wie bei der vollassoziativen Architektur.

Ein erfolgreicher Lesezugriff wird als **Read Hit** (oder auch *Cache Hit*) bezeichnet, ein Fehltreffer als **Read Miss**. Nach einem Read Miss muss das Datum oder die Instruktion aus dem Hauptspeicher bzw. einem tieferen Cache geladen werden, was zu Wartezyklen führt.

Werden hingegen Daten zurückgeschrieben, bezeichnet man mit **Write Hit** einen Schreibzugriff, der die (bisherigen) Daten im Cache vorfindet und nun überschreibt. Liegt hingegen ein **Write Miss** vor (dieser kann entstehen, weil die Daten

zwischenzeitlich verdrängt wurden), so muss eine Strategie gewählt werden, den Schreibvorgang trotzdem erfolgreich zu beenden. Dies kann bedeuten, dass der Wert zunächst wieder in den Cache geladen und dann modifiziert wird (**Fetch On Write** oder **Write-Allocate**), oder der Wert wird nur in den niedrigeren Hierarchien geschrieben (**Write Around** oder **No-Write Allocate**).

Eine weitere Strategie betrifft das Zurückschreiben der Daten, die im Cache nach dem ursprünglichen Laden verändert wurden. Diese Daten werden durch ein *Dirty-Bit* (→ Bild 9.19, D-Bit) gekennzeichnet. Zwei Strategien zum Zurückschreiben der Daten etwa bei Überschreiben der Cache-Zeile werden unterschieden:

- **Write-Back Strategie**: Die veränderten Cache-Inhalte werden nur dann im L2-Cache oder Hauptspeicher aktualisiert, wenn die Zeile aus dem Cache verdrängt wird. Diese Strategie minimiert die schreibenden Zugriffe auf den L2-Cache bzw. Hauptspeicher, führt allerdings zu Problemen bei Multiprozessorsystemen, Systemen mit DMA (Direct Memory Access) und mit Speichermanagement und virtuellen Adressbereichen, da der Hauptspeicher ggf. nicht konsistent ist.
- **Write-Through Strategie**: Bei jedem Cache-Eintrag wird das Datum nach einem Write Hit auch in die niedrigeren Ebenen geschrieben. Dies führt zu einer Datenkonsistenz, ist einfacher zu implementieren, bedingt allerdings auch höheren Datenverkehr (mit ggf. Wartezeiten).

9.2.3 Cache-Ersetzungsstrategie

Caches verschiedener Ebenen sind fast ausschließlich so organisiert, dass der Inhalt der höheren Ebenen auch in den niedrigeren Ebenen vorhanden ist. Dies ist einfach zu implementieren und bedeutet, dass bei einem Verdrängen aus einer höheren Ebene nur die mit dem Dirty-Bit gekennzeichneten Daten zurückgeschrieben werden müssen.

Die Ersetzungsstrategie entfällt beim Direct Mapping, weil hier nur eine Cachezeile als Speicherort für einen Hauptspeichereintrag gewählt werden kann. Alle teil- und vollassoziativen Cacheorganisationen benötigen hingegen eine Entscheidung über die zu verdrängende Cache-Zeile. Der schlimmste Fall einer zyklischen, gegenseitigen Verdrängung wird Thrashing (”Flattern“) bezeichnet. Dieser Fall tritt in der Praxis nur bei Caches mit Direct Mapping auf.

Als Ersetzungsstrategie wird meist der Least-Recently-Used- (LRU-) Algorithmus gewählt, der sich leicht durch Zähler (→ Bild 9.19: 8 bit) implementieren lässt. Zufallsstrategien sind ebenfalls möglich und führen zu ähnlichen Ergebnissen.

9.2.4 Probleme beim Einsatz von Cachespeicher

Es existieren tatsächlich einige Probleme in der Nutzung von Cachespeicher, die sich durch folgende Punkte klassifizieren lassen:

- Thrashing: Selten kommt es durch die spezifische Nutzung innerhalb eines Algorithmus zu gegenseitigen Verdrängen aus dem Cache. Dies führt dann zu

einer Verlängerung der Laufzeit, weil der Cache nicht optimal für den Algorithmus organisiert ist.

- Dateninkonsistenzen: In Multiprozessorsystemen müssen globale Daten konsistent gehalten werden, um ungültige Rechnungen zu vermeiden. Dies wird meist durch die Wahl einer Write-Through Strategie erreicht, die aber im Einzelfall erhebliche Laufzeitverzögerungen mit sich bringen kann (Beispiel: Intel Pentium III bei Ausführung des 'Sieb des Eratosthenes').
- Echtzeitfähigkeit: Bei Einsatz von Caches ist keine Vorhersage über die Geschwindigkeit der Algorithmen möglich, außer, man nimmt den Worst-Case an (WCET: Worst-Case Execution Times). Dies bedeutet zwar keine Verschlechterung, aber auch keine Verbesserung in der Schätzung der Echtzeitfähigkeit eines Systems.

9.2.5 Scratch-Pad Memory

Die fehlende Verbesserung im Echtzeitbereich führte im Bereich der eingebetteten Systeme dazu, den Speicherbereich, der üblicherweise als Cache genutzt wurde, ganz oder teilweise als sogenanntes Scratch-Pad Memory zu nutzen. Hierunter versteht man einen Speicher, der auf dem Chip integriert ist und als *Ersatz* für den entsprechenden Hauptspeicher fungiert.

Dem Scratch-Pad Memory wird also ein Adressbereich zugewiesen, der im Hauptspeicher dann ausgeblendet wird. Da dieser Speicher mit Prozessorgeschwindigkeit arbeitet, kann nun in einem beschränkten Daten und/oder Codebereich mit voller Geschwindigkeit gearbeitet werden, und dies garantiert. Auf diese Weise erreicht man eine Verbesserung der Echtzeitfähigkeit.

Diese Form des Speichers wird häufig für hochpriorisierte Interrupt-Service-Routinen oder spezielle Datenbereiche, auf die sehr häufig zugegriffen werden, genutzt.

9.3 Speichermanagement

Literaturhinweise: [7, Kapitel 5.6], [10, Kapitel 8.7]

Der Hauptspeicher eines Mikroprozessor-basierten Systems wird gemäß dem Von-Neumann-Modell als zugänglich für Instruktionen und Daten sowie als 'flach' organisiert betrachtet. Die flache Organisation bedeutet, dass auf den Speicher von der niedrigsten bis zur höchsten Adresse ohne Segmentierungen, Sprünge etc. zugegriffen werden kann. Es existieren gibt drei Gründe, von diesem Speicherbild abzurücken:

- Für Speicherbereiche müssen Schutzmechanismen eingeführt werden. Es werden – trotz des einheitlichen Adressraums – Bereiche vorhanden sein, in denen

ausschließlich Code vorhanden ist, während andere Bereiche Daten oder auch dem Input/Output (Memory-Mapped) vorbehalten sind.

Im Allgemeinen wird es verboten sein, in einen Codebereich hineinzuschreiben, oder einen Datenbereich als Programm ausführen zu wollen. Derartige Schutzfunktionen können via Hardware eingeführt werden und machen die Programmausführung sicherer. Die Hardware wird im Allgemeinen als **Memory Protection Unit** (MPU, → 9.3.1) bezeichnet.

- Der virtuelle Speicherbereich, der prinzipiell von einem Programm genutzt werden kann, ist größer als der physikalisch implementierte. Bei einer Adressierung von 32 Bit (= 4 GByte Adressraum) bzw. 64 Bit (= 2^{64} Adressraum ohne zusätzliche Maßnahmen) ist es nicht besonders schwierig, diesen Bereich zu überschreiten.

  Im Rahmen einer besonderen Einheit – der **Memory Management Unit** und der dort enthaltenen virtuellen Adressierung – können die virtuellen Adressen auf physikalische umgerechnet werden, so dass man scheinbar einen sehr großen Speicher vorfindet (→ 9.3.2).

  Eine der wesentlichen Motivationen, den vorhandenen physikalischen Speicher durch eine virtuelle Adressierung zu verwalten, entstammt auch dem Multitasking-Betrieb. Hier wird ein Verweilen mehrerer Tasks im Speicher – zumindest teilweise – gefordert, zudem müssen die einzelnen Tasks 'ihren' Speicherbereich wiederfinden (oder relokatierbar sein).

- Die 32-Bit- und 64-Bit-Prozessoren besitzen oft verschiedene Betriebsmodi, z.B. Usermode und Systemmode. Diese Modi unterscheiden sich z.B. durch Befehle, die meist im Usermode unzulässig sind, und eben durch Speicherbereiche, um gegenseitig geschützt zu sein.

9.3.1 Memory Protection Unit (MPU)

Die Memory Protection Unit (MPU) ist meist Bestandteil der übergeordneten Memory Management Unit (MMU). In eingebetteten Systemen hingegen ist sie auch isoliert (also ohne MMU) zu finden, so dass ihre Funktionalität gesondert dargestellt werden soll.

Der Speicherbereich wird in sogenannte **Pages** (eine deutsche Übersetzung lautete Kachel, hat sich aber nicht durchgesetzt) eingeteilt, wobei oft die Pagegröße konfigurierbar ist (etwa im Rahmen 1 KByte bis 256 MByte, bei 64-Bit-Systemen auch 4 GByte). Diese Pages haben in einen Bezug auf die virtuellen Adressen eines Programms, also diejenigen, aus denen ein Prozess (= dynamischer Ablauf eines Programms) die Instruktionen und Daten scheinbar lädt bzw. in sie hineinschreibt. Diese virtuellen Adressen entsprechen bei einem reinen MPU-System natürlich den physikalischen Adressen.

Bei einem beliebigen (Schreib- oder Lese-) Zugriff wird in der Hardware der MPU geprüft, ob dieser Zugriff aufgrund der Tabelleneinträge erlaubt ist oder nicht. Im

negativen Fall wird eine Address Exception, also eine Ausnahmebehandlung initiiert. Diese Ausnahmebehandlung, meist zum Betriebssystem gehörend, kann dann die Situation bereinigen, muss aber von einem Programmfehler ausgehen.

Aus Sicht der CPU muss die Zugriffsüberprüfung vor dem Zugriff, insbesondere dem schreibenden Zugriff bei Datenoperationen erfolgen. Fehlzugriffe im Instruktionsfetch müssen in der Regel zu einem Prozessabbruch führen, hier kann der Zugriff erst nachträglich kontrolliert werden. Eine rechtzeitige (d.h. Fehlwerte vermeidende) Unterdrückung bei Datenoperationen ist möglich, wenn die MPU in einer Pipelinephase des Prozessors, also kurz von Memory Access, arbeitet.

Eine einfache Erweiterung der MPU-Funktion kann dadurch erreicht werden, dass den Speicherbereichen eine Prozessidentifikation zugeordnet und somit die Zugriffsrechte nicht nur global, sondern Prozess-orientiert verwaltet. Der Aufwand an Verwaltung beschränkt sich dabei auf eine Tabelle, in der pro Eintrag Speicherbeginn, ggf. Speicherende, Zugriffsart (r, w, x) und Prozessidentifikator eingetragen werden müssen.

9.3.2 Virtuelle Adressierung

Die virtuelle Adressierung beruht ebenfalls auf einer Einteilung des Speichers in Pages, d.h. in zusammenhängende Bereiche fester (meist konfigurierbarer) Größe. Diese Pages werden dann über gegenüber der MPU erheblich erweiterten Tabellen verwaltet.

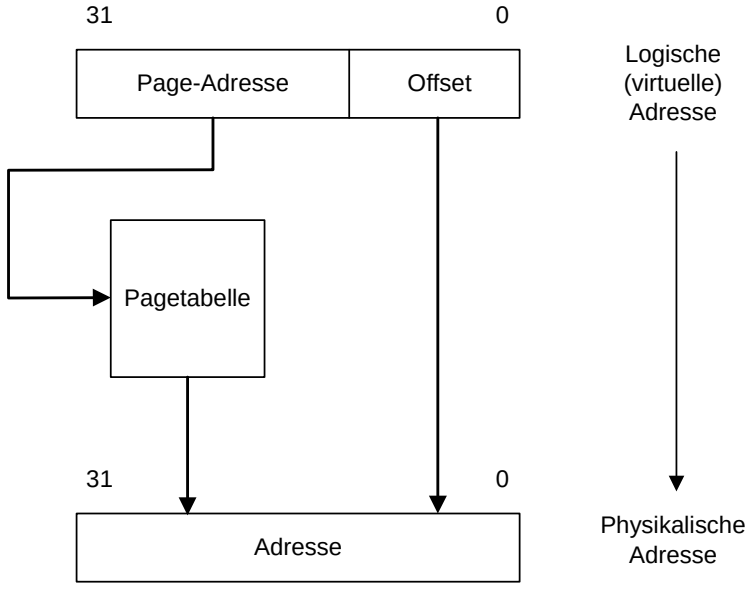


Bild 9.20 Aufbau Adressumsetzung Memory Management Unit

Bild 9.20 zeigt das Prinzip der Adressumsetzung innerhalb der MMU. Die logische Adresse eines Prozesses, auch als virtuelle Adresse bezeichnet, wird in zwei Segmenten interpretiert: Die Page-Adresse sowie der Offset.

Die physikalische Adresse, die dann am Adressbus ausgegeben wird, entsteht durch Tabellenzugriff und anschließende 'Mischung' mit dem Offsetteil. Dieses vergleichsweise einfache Verfahren kann in der Praxis (Beispiel: Intel Pentium) sehr weitgehend variiert werden (Bild 9.21).

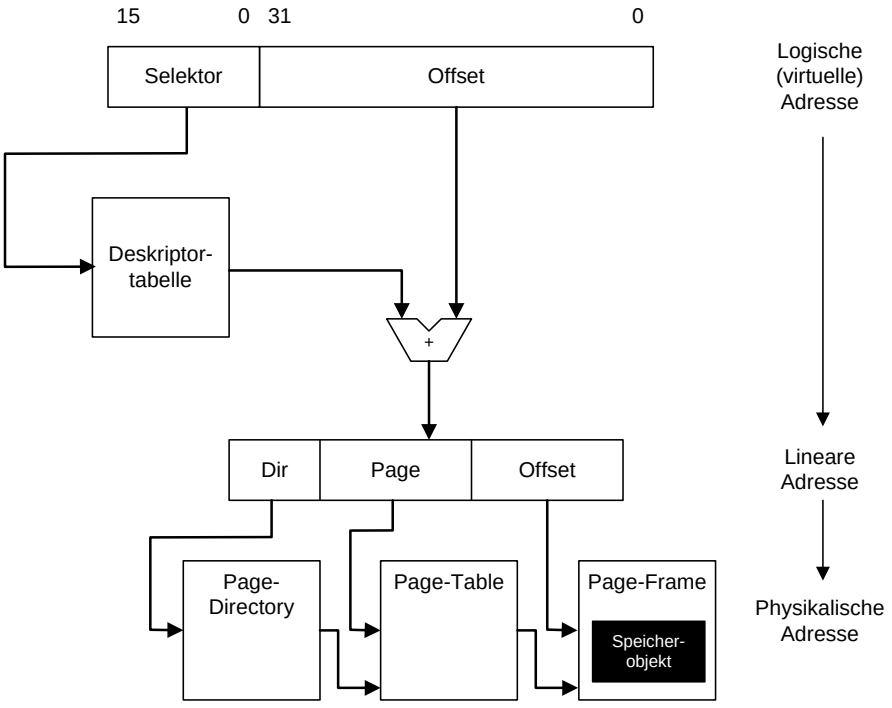


Bild 9.21 Aufbau Adressumsetzung im Pentium

Die Deskriptortabelle beinhaltet Größe und Zugriffsrechte auf den angewählten Bereich; das Selektorregister entspricht den ehemaligen Segmentregister der 8086-Architektur. Durch dieses komplexe Verfahren ist eine große Vielfalt an Konfigurationen möglich, allerdings ändert sich am Prinzip nichts.

In jedem Fall bedeutet die Adressbildung einen Takt Rechenzeit (bei Speicherung der Pagetabelle im Prozessor), und dieser Takt sollte in die Pipeline des Prozessors integriert sein. Die Pagetabelle kann sehr groß werden und ihrerseits nur im Hauptspeicher verfügbar sein. Um den damit verbundenen Geschwindigkeitsnachteil auszugleichen, wird meist ein Translation Look-Aside Buffer (TLB) im Prozessor selbst als eine Art Cache für Pagetabelleneinträge genutzt.

Eine virtuelle Adressierung benötigt natürlich einen Massenspeicher, um sinnvoll eingesetzt zu werden. Für PC- und ähnliche Systeme ist dies durch Harddisk gewährleistet, in eingebetteten Systemen geht man z.B. von einem Massenspeicher im Netzwerk aus.

Insgesamt müssen auch Memory Management und Cache aufeinander abgestimmt sein, denn ein Pagewechsel im Hauptspeicher (**Swapping**) bedeutet auch, dass die Cacheeinträge für ungültig erklärt werden müssen, da ein Zurückschreiben dann unmöglich ist. Als Strategie für ein Verdrängen einer Page aus dem Hauptspeicher bietet sich wiederum der LRU-Algorithmus an. Das Thrashing, beim Cache schon erwähnt, stellt auch bei dem Memory Management ein Problem dar.

In der Praxis kann die Speicherverwaltung – Aufgabe des Betriebssystems – noch dadurch kompliziert werden, dass einzelne Bereiche als non-cacheable, andere als unveränderlich zugeordnet usw. definiert werden.

10 Weiterentwicklungen und alternative Modelle

Aktuell wird eine Vielzahl von Modellen diskutiert, die als Alternativen zu dem Von-Neumann-Rechnermodell gelten. Ziel dieser gesamten Bemühungen ist es, Nachteile der Von-Neumann-Architektur (insbesondere Verarbeitungsgeschwindigkeit oder Reaktionsfähigkeit) zu vermeiden.

Um dies systematisch zu behandeln, wird zunächst eine kurze Einführung in das Gebiet des Reconfigurable Computing gegeben. Hierbei steht das Ausführungsprinzip, sprich die Art der Steuerung des Programms im Vordergrund.

Im Anschluss daran werden diverse Architekturen besprochen: Das UCB/UCM-Konzept (TU Clausthal) mit dem Vorläufer >S<puter und der einfachsten Implementierung in Form der rRISC-Maschine (reconfigurable RISC), der Xputer (Univ. Kaiserslautern) und die XPP-Architektur der Firma PACT (München).

10.1 Einführung Reconfigurable Computing

Bekannte Formen von programmierbaren Rechnern sind der Von-Neumann-Rechner und die Programmable Logic Devices (PLD, für eine Einführung hierzu siehe Vorlesung Technische Informatik II sowie Configurable Computing). Um einmal zu verdeutlichen, worin der wesentliche Unterschied dieser Architekturen in Bezug auf die Programmausführung besteht, sei ein Beispiel angeführt.

```
#define INPORT_ADR 0x1000
#define OUTPORT_ADR 0x2000

unsigned char *a = INPORT_ADR;
unsigned char *b = OUTPORT_ADR;

main()
{
  while(1) {
    if( *a == 0 )
      *b = 1;
    else
      *b = 0;
  }
}
```

Bild 10.1: C-Programm zur NOR-Funktion für 8 binärwertigen Eingängen

Das Problem besteht in einer NOR-Funktion (Verneinung des logischen ODER) von binärwertigen Signalen an einem Eingang mit 8-Bit-Breite. Bild 10.1 zeigt dies als Endlosprogramm, formuliert in der Sprache C. Die NOR-Funktion der 8 Eingänge (es wird vorausgesetzt, dass `unsigned char` einer Datenbreite von 8 Bits entspricht) wird am Ausgangsport (Adresse 0x2000), Bit 0 dargestellt.

```
L0:  MOV   R1, #a      ; Adresse von a
     MOV   R2, #b      ; in R1, von b in R2

L1:  LD    R3, (R1)    ; Inport-Bits in R3
     CMP   R3, #0      ; IF-Bedingung
     BNE   L3          ;

L2:  MOV   R4, #1      ; IF-Zweig
     JMP   L4          ;

L3:  MOV   R4, #0      ; ELSE-Zweig

L4:  ST    (R2), R4    ;
     JMP   L1          ;
```

Bild 10.2: Assemblerübersetzung für den Sourcecode aus Bild 10.1

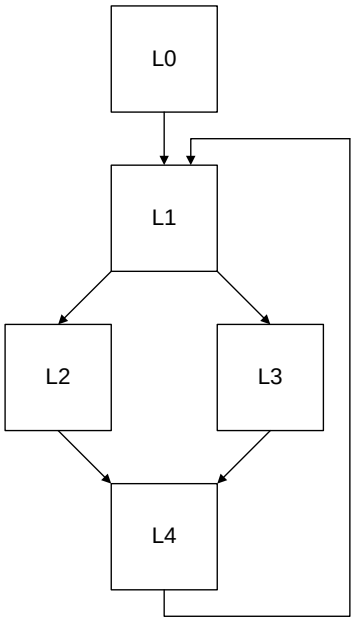


Bild 10.3: Programmfluss mit Basisblöcken

Die Übersetzung in eine fiktive Assemblersprache (für eine typische RISC-Architektur, z.B. MPM3, → 4.3) in Bild 10.2 zeigt natürlich eine Endlosschleife. Sie zeigt aber insbesondere, dass dieses Programm, so klein es ist, sehr zergliedert wird. Dies würde für eine superskalare Ausführung große Probleme bereiten, auf entsprechenden Instruction Level Parallelism zu kommen (→ 5.2).

Abgesehen von der starken Zergliederung der Assemblerübersetzung in diesem Beispiel (Bild 10.3) nimmt das Durchlaufen einer Programmschleife einige Takte in Anspruch: Für eine RISC-Architektur mit theoretisch 1 Instruktion/Takt benötigt die Programmschleife (also ohne Block L0) 6 (`ELSE`-Zweig) bzw. 7 (`IF`-Zweig) Takte im Minimum. Eine Eingangsänderung wird erst nach 5 Takten minimal, 12 Takten maximal am Aus sichtbar, bei 100 MHz Taktfrequenz also im Mittel nach 85 ± 35 ns. Die Kontrollflussverzweigungen und die Datenabhängigkeiten verhindern übrigens in diesem Fall eine Beschleunigung durch superskalare Ausführung.

Die gleiche Problemstellung kann wie bereits erwähnt auch in Hardware gelöst werden und führt zur Implementierung eines NOR-Gatters. Die Lösung in Bild 10.4 entspricht dabei der üblichen Vorgehensweise: Die PAL-Struktur, die in vielen Bausteinen mit programmierbarer Logik vorhanden ist, weist programmierbare UND-Eingänge mit dahinterliegenden, festverdrahteten ODER-Gattern auf. Dies entspricht der Disjunktiven Normalform DNF, und so lautet die optimale Lösung für PLDs (in Booleschem Assembler):

```
OUTPORT0 =      /INPORT7 * /INPORT6 * /INPORT5 * /INPORT4 *
                /INPORT3 * /INPORT2 * /INPORT1 * /INPORT0;
```

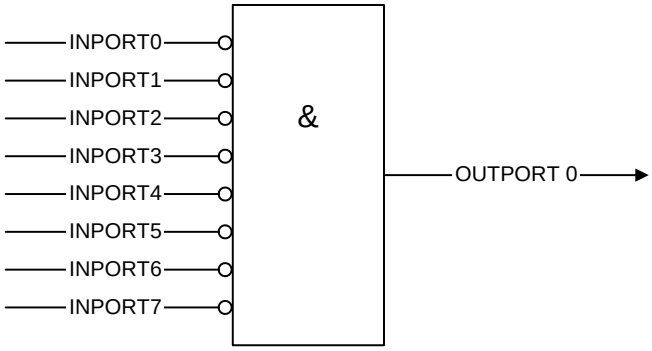


Bild 10.4: Optimale Hardwarelösung für PLDs

Eine Eingangsänderung ist in diesem Fall nach einer Gatterlaufzeit am Ausgang sichtbar, also z.B. nach 5 ns. Diese Zeit ist konstant für alle Änderungen, wesentlichen Abweichungen von diesem Wert existieren nicht (abgesehen von Laufzeitschwankungen durch Temperaturänderungen etc.).

Damit sollte deutlich sein, dass beide Lösungen die gleiche logische Funktion beschreiben, dies allerdings auf vollkommen unterschiedlichem Weg. Die Mikroprozessor-basierte Lösung bedient sich der vordefinierten Mikroprozessorbefehle, im Wesentlichen aus Datenoperationen und Kontrollflussinstruktionen bestehend. Diese Befehle werden hintereinander ausgeführt, das Programm wird in einer zeitlichen Sequenz durchlaufen.

Die PLD-basierte Lösung besteht daraus, ebenfalls vordefinierte Basisoperationen (meist NOT (Invertierung), AND sowie OR, oft noch mit abschließendem XOR) zu nutzen. Diese werden jedoch mit Hilfe der Programmierung verdrahtet, also strukturiert. Man spricht im Allgemeinen von der *Ausführungsdimension*. Die Sequenzen in Mikroprozessoren bedeuten, dass jedes dort ausgeführte Programm in der Zeit abläuft. Erhöht man die Komplexität eines Algorithmus, benötigt der Prozessor mehr Zeit zur Ausführung. Dies wird als *Computing in Time* bezeichnet, im Gegensatz zu *Computing in Space*, bei dem die Komplexität bzw. Berechnung in der (Silizium-)Fläche steckt.

Man kann diese beiden Ausführungsdimensionen und ihre derzeitigen Realisierungsformen (Mikroprozessor und PLD) als Extrempunkte auffassen: Auf der einen Seite eine Sequenz festgefügter Instruktionen, auf der anderen Seite eine einzige Instruktion, die aber durch die strukturale Programmierung aus kleinen Einheiten zusammengefügt ist und den kompletten Algorithmus enthält. Man findet auf dieser Skala (Bild 10.5) *Sequential Computing* und *Configurable Computing* vor.

Bild 10.5: Zusammenhang Configurable und Sequential Computing

Zugleich mit dieser Darstellung stellt sich die Frage nach Übergangsformen, vielleicht sogar skalierbaren Varianten, die je nach Bedarf auf die eine oder andere Form abgebildet werden können. Genau dies bietet das Reconfigurable Computing, bei dem ein Algorithmus – ganz allgemein gesprochen – auf die Sequenz von Konfigurationen abgebildet wird.

Diese Darstellung erklärt schon vieles, aber sie grenzt noch nicht scharf ab. Hierzu zeigt Tabelle 10.1 die charakteristischen Zeiten für PLD und Mikroprozessoren (µP).

Während man bei PLDs auch heute noch fast ausschließlich davon ausgeht, eine einzige Konfiguration (= Instruktion) in den Speicher zu laden und damit einen guten Ersatz für ASICs zu haben, geht der Ansatz des Reconfigurable Computing einen deutlichen Schritt weiter. Hier ist es grundsätzlich möglich, die aktuell laufende Konfiguration zumindest partiell durch eine andere zu ersetzen. Eine Hardwareplattform, die das ermöglicht, kann natürlich auf eine einzige Konfiguration skaliert werden. Worin besteht aber der Unterschied zum Sequential Computing eines Mikroprozessors?

Tabelle 10.1 gibt auch hierzu Auskunft [12]. Man könnte es so auffassen, dass ein Mikroprozessor eine Konfiguration (genannt Instruktion) aufnimmt, interpretiert und ausführt. Gleiches macht eine rekonfigurierbare Hardware, von der angenommen wird, dass das Laden der Konfiguration auch in einem (schnellen) Takt erfolgen kann. Der wesentliche Unterschied zwischen diesen beiden Formen ist derjenige, dass der Mikroprozessor die Instruktion nach Ausführung *maschinendefiniert* verwirft, während bei dies bei rekonfigurierbarer Hardware *applikations-* oder *userdefiniert* erfolgen kann.

	PLD, configurable	Reconf. Hardware	µP
Binding time	load config.	load config.	cycle
Binding duration	End of Application	user-defined	cycle
Programming time	ms - s	cycle	cycle
Number of Instructions	1	>1	>>1

Tabelle 10.1: Charakteristische Zeiten für programmierbare Einheiten [12]

Zusammenfassend ergeben die verschiedenen Ausführungsformen eines Rechenprogramms folgende Möglichkeiten:

Alle drei Formen der programmierbaren Hardware ermöglichen es dem/der Systementwickler/in, ein berechenbares Problem zu lösen, sie besitzen nach entsprechender Programmierung alle die gleiche (logische oder arithmetische) makroskopische Funktionalität. Hardware für Sequential Computing (Mikroprozessoren) optimieren dabei auf die Fläche zuungunsten der Ausführungszeit, Hardware für Configurable Computing (PLD) auf die Zeit zuungunsten der Fläche, und bei Hardware für Reconfigurable Computing kann man zwischen beiden Optimierungsformen variieren: Größere Konfigurationen bedeuten mehr Fläche, mehr Konfigurationen hingegen mehr Ausführungszeit (Space-Time-Mapping) .

Nimmt man zu der Darstellung der Anzahl und Variabilität der Instruktionen in Bild 10.5 als dritte Darstellungsdimension noch die Komplexität der Operation hinzu, so erhält man die komplette Klassifizierung der verschiedenen Rechnerplattformen in Bild 10.6. Hier wird bei den Operationen zwischen logisch (Bitweise) und arithmetisch (Wort-weise) unterschieden. Gemäß dieser Darstellung existieren 2 Klassen von Hardwareplattformen für Reconfigurable Computing: Multicontext-PLDs (→ Vorlesung Configurable Computing) und die UCB-Klasse.

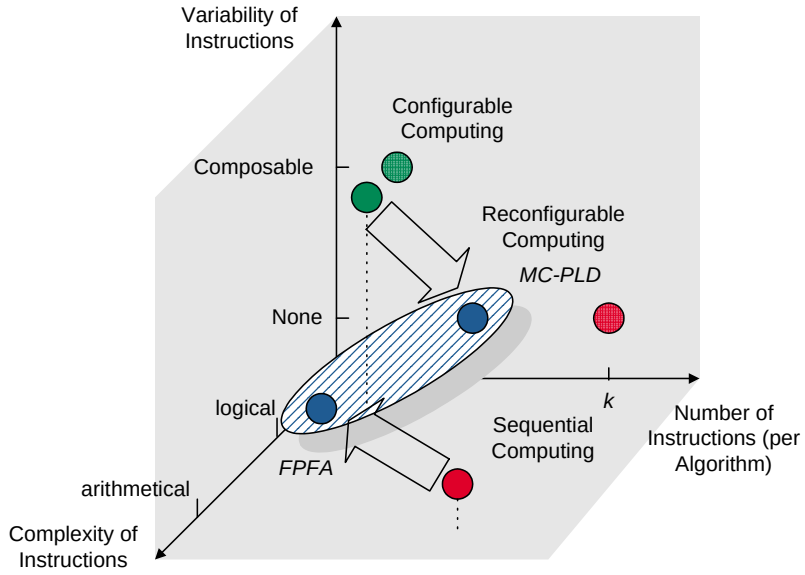


Bild 10.6: Bereich Reconfigurable Computing

Die beiden Komplexitätsstufen lassen sich auch definieren und werden auf die Abhängigkeit eines Ausgangsbits von den Eingangsbits zurückgeführt: Bei logischen

Funktionen besteht nur die Abhängigkeit von einem Eingangsbit pro Eingangsvektor (Beispiel: logisches AND oder Shift-Befehl), bei arithmetischen ist das Ausgangsbit von mehreren Eingangsbits pro Eingangsvektor (Beispiel: Addition).

Diese Komplexität beinhaltet die Anpassbarkeit an verschiedene Algorithmenklassen: Die Multicontext-PLDs mit ihren logischen Operationen sind für Bit-Level-Algorithmen sehr gut geeignet, also z.B. für irreguläre Steueralgorithmen, kompakte Speicherung von Zustandscodierungen etc., die UCB-Strukturen besser für arithmetisch basierte Algorithmen etwa aus der digitalen Signalverarbeitung.

10.2 UCB/UCM-Konzept

Die UCB-Klasse der rekonfigurierbaren Architekturen eignet sich – zumindest aus Operationssicht – gut als weiterführende Architektur für Mikroprozessoren. Dies soll in diesem Abschnitt anhand des (auch an der TU Clausthal entwickelten) UCB/UCM-Konzepts. Dieses Konzept beinhaltet Block-orientierte Strukturen (UCB: Universal Configurable Block) mit übergeordneten Maschinen (UCM: Universal Configurable Machine) zur Verwaltung der Blöcke, Scheduling etc.

Die Darstellung wird mit dem >S<puter begonnen, der aus der Variation des Instruction Scheduling der superskalaren Prozessoren entstammt und aus dem die UCBs weiterentwickelt wurden.

10.2.1 Einführung in das >S<puter-Prinzip

Bild 10.7 zeigt den Aufbau eines superskalaren Prozessors bei bewusster Fortlassung der Floating-Point-Einheit. Die im Folgenden dargestellten Eigenschaften der ausführenden Einheit für den Integerteil des Rechners lassen sich auf die Floating-Point-Unit 1:1 übertragen.

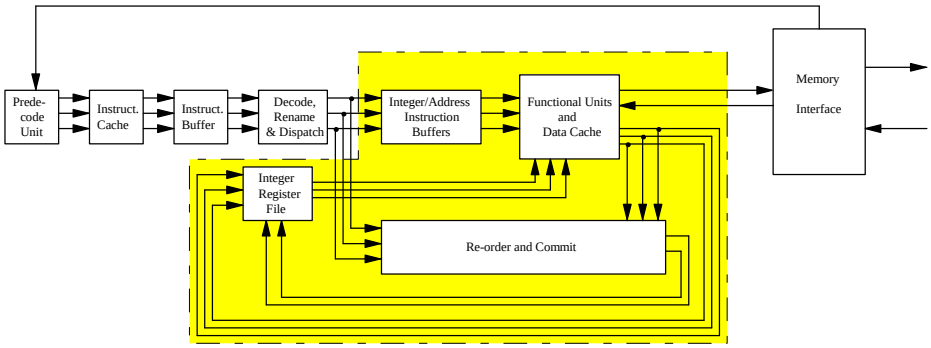


Bild 10.7: Typische Organisation eines superskalaren Prozessors

Innerhalb des Prozessors werden das Integer Register File, die Functional Units und die Re-order-and-Commit-Unit zu einer neuen Einheit (in Bild10.7 unterlegt), der s-Paradigmen-Unit oder abkürzend s-Unit, zusammengefaßt. Diese Architektur wird im s-Paradigmen-Modell wie in Bild 10.8 variiert.

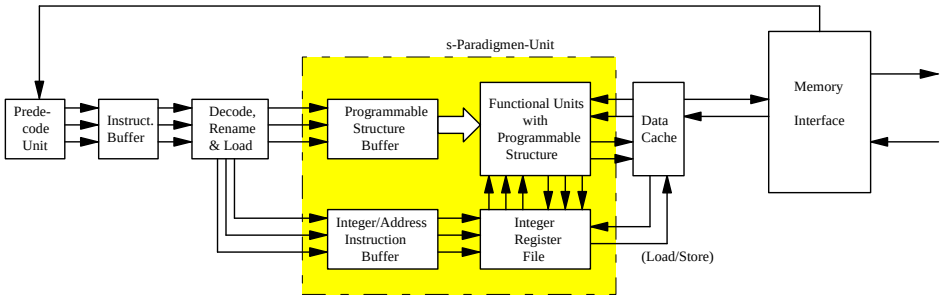


Bild 10.8: Mikroarchitektur des s-Paradigmen-Modells

Das s-Paradigmen-Modell kennt vier Klassen von Maschinenbefehlen:

- Branch- und Jump-Befehle zur Kontrollflusssteuerung
- Load-/Store-Befehle zum Datentransfer zwischen Registern und Speicherbereich
- Arithmetische und logische Befehle zur Berechnung
- sonstige Befehle wie No Operation, Wait, Stop etc., die im weitesten Sinne auch der Kontrollflusssteuerung dienen

Während die Befehlsklassen zur Kontrollflusssteuerung wie bisher belassen und damit entsprechend dem Standard in superskalaren Rechnern ausgeführt werden, nehmen die Load/Store-Befehle und die arithmetisch/logischen Befehle eine neue Stellung ein.

Load/Store-Befehle werden entweder mit Hilfe einer oder mehrerer Load/Store-Pipeline(s) zum Datentransfer zwischen dem Integer Register File und dem Datenspeicher (Cache, Hauptspeicher) eingesetzt und dann auch wie bisher bearbeitet, oder sie werden zu den arithmetisch/logischen Befehlen hinzugefügt und in das nun beschriebene Kernstück des s-Paradigmen-Modells integriert. Die Entscheidung hierüber obliegt dem Systemdesigner der CPU. Move-Befehle hingegen, die einen Datentransfer zwischen den Registern des Prozessors bewirken, gehören grundsätzlich zu diesem Modell.

Die arithmetisch/logischen Befehle (und die hinzugefügten Load/Store-Befehle) werden ihrer Programmiersequenz zufolge in eine Struktur von aufeinanderfolgenden Hardwareverknüpfungen übersetzt. Zu diesem Zweck bietet die Functional Unit im >S<puter eine programmierbare Struktur und feste verdrahtete arithmetisch/logische Verknüpfungen (sowie ggf. Zugriffsfunktionen wie Load/Store-

Pipelines) an, die durch die Struktur miteinander verknüpft werden können und in der Reihenfolge der Strukturierung bearbeitet werden. Bild 10.9 erläutert diese Funktionalität:

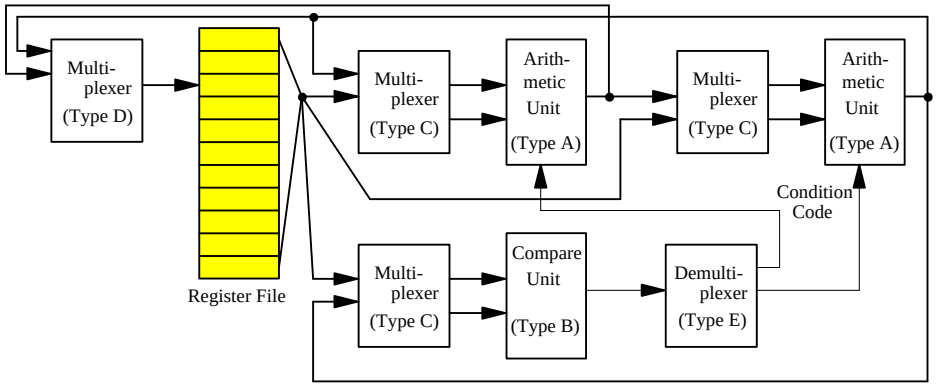


Bild 10.9: Prinzipstruktur der Functional Unit des s-Paradigmen-Modells

Die Teileinheiten der Functional Unit und das Register File (in Bild 10.9 hinterlegt) werden durch eine Vielzahl von Datenverbindungen und Multiplexern verbunden. Die Datenverbindungen sind dabei als Busse mit der jeweiligen internen Datenbusbreite ausgeführt (z.B. 32 Bit), eine Ausnahme bilden die Condition Codes, die als Bitleitungen (dünn gezeichnet) implementiert sind. Innerhalb der Functional Unit existieren 5 Typen von Teileinheiten:

- *Arithmetic Unit* (AU, Type A): Diese Einheit sollte nicht mit der herkömmlichen Arithmetical-Logical-Unit (ALU) verwechselt werden. Die AU enthält eine oder wenige, dann konfigurierbare Verknüpfungen, beispielsweise die Möglichkeit, zwei Integerzahlen zu addieren. Sie liefert ein Ergebnis an ihrem Ausgang, wenn die Eingänge entsprechend beschaltet werden, und dieses Ergebnis kann innerhalb des Schaltnetzes weiterverwendet werden. Die AU ist gekennzeichnet durch zwei Eingangsbusse und einen Ausgangsbus, gegebenenfalls durch eine Konfigurationsmöglichkeit (Auswahl der Verknüpfung, im Extremfall entsprechend einer ALU) und durch Konditionalbits. In einigen Fällen, beispielsweise bei der Multiplikation, kann die Breite des Ausgangsbusses abweichen, um die Berechnungen zu ermöglichen.
- *Compare Unit* (CoU, Type B): Für die bedingte Ausführung einer Verknüpfung werden Condition Code Bits durch einen konfigurierbaren Vergleich in der CoU erzeugt. Der Vergleich läßt sich bei Verwendung von 3 Konfigurationsbits auf >, >=, <, <=, !=, ==, TRUE oder FALSE einstellen. Kennzeichen der CoU sind zwei Eingangsbusse und ein Ausgangsbit sowie die Konfigurierbarkeit.
- Multiplexer (Mul_C, Type C): Multiplexer vom Typ C belegen die Eingänge der AUs und der CoUs mit jeweils zwei Eingangswerten (in der vollen Verar-

beitungsbreite). Hierfür benötigen sie eine Anzahl von Konfigurierungsbits, die sich aus der Anzahl von Eingängen und den zwei Ausgängen ergeben. Kennzeichen für Mul_C-Einheiten sind zwei Ausgangsbusse.

- Multiplexer (Mul_D, Type D): Für die Belegung der Register mit den Ergebniswerten wird lediglich ein Ausgangsbus und damit auch nur die Hälfte der Konfigurationsbits benötigt. Ein Mul_D unterscheidet sich daher vom Mul_C durch die Anzahl seiner Ausgangsleitungen.
- Demultiplexer (Demul, Type E): Die Ergebnisse der Vergleiche (CoU) müssen an entsprechende AUs weitergeleitet werden. Im Gegensatz zu den Multiplexern, die eine Quellenauswahl vornehmen, liegt hier eine Zielauswahl vor.

Die Verbindungen zwischen den Teileinheiten können komplett oder teilweise ausgeführt sein, abhängig von der Anzahl der zur Verfügung stehenden Konfigurationsbits in der Gesamtheit. In einer Beispielarchitektur im nächsten Abschnitt wird eine vollständige Verbindbarkeit gezeigt sowie die Zahl der notwendigen Bits daraus berechnet.

Der Sinn der strukturierbaren Functional Unit, der s-Paradigmen-Unit, liegt nun darin, die Struktur entsprechend den Maschinenbefehlen in einem (Basis-, Super- oder Hyper-)Block eines Programms anzupassen (→ 5, 6).

Die Programmierung erfolgt im s-Paradigmenmodell durch das Laden von Konfigurationsbits für die Teileinheiten. Diese werden im Programmable Structure Buffer zwischengespeichert und bei Bearbeitung des Blocks in die s-Unit geladen, die dadurch entsprechend strukturiert ist und den Block bearbeiten kann. Die Bearbeitung bezieht sich dabei lediglich auf die arithmetischen und logischen Verknüpfungen zwischen Registerinhalten, ggf. auch mit Speicherinhalten (falls eine entsprechende Load/Store-Pipeline zur Verfügung steht), während alle anderen Befehle, insbesondere Load/Store und Kontrollflussbefehle wie bisher üblich ablaufen.

Die Generierung der Konfigurationsbits kann entweder im Assembler erfolgen (Compiletime-basierte Generierung), es ist auch prinzipiell möglich, sie in der CPU, etwa durch einen funktionell erweiterten Programmable Structure Buffer zur Laufzeit erzeugen zu lassen (Runtime-basierte Generierung).

10.2.2 Beispiel für eine Realisierung eines >S<puters

Die Struktur der s-Unit mit festverdrahteten AUs sowie CoUs und konfigurierbaren Wegen zwischen diesen Teileinheiten legt zunächst fest, dass die Multiplexer das programmierbare Element in dieser Konfiguration darstellen. Die feste Verdrahtung insbesondere der AUs wurde gewählt, um die Anzahl der zu ladenden Bits möglichst gering zu halten. In einer weiteren Stufe der Flexibilisierung könnten auch die AUs programmierbar sein, d.h. auf einfachen Strukturen wie NAND-Gattern oder Disjunktiven Normalformen (DNF) aufbauend wäre eine nahezu beliebige Funktionalität bereits in einer AU integrierbar.

Arithmetic Units beinhalten beispielsweise folgende Funktionalitäten:

- Arithmetische Verknüpfungen wie Addition, Subtraktion, Multiplikation, Division
- Logische Verknüpfungen wie AND, OR, XOR, NOT, Komplement (Zweier-)
- Shift-Funktionen wie arithmetische oder logische Shifts nach rechts/links
- Bedingte Datentransfers, abhängig von Eingangsbits (2 Wege-Multiplexer, im Unterschied zu den Mul_C und Mul_D-Teileinheiten zur Laufzeit umschaltbar)

Die Basis für die Programmierung der Struktur, also der beiden Multiplexertypen, besteht in RAM-Zellen. Hiermit ist eine sehr schnelle Konfigurierung gewährleistet, während andere Technologien wie EEPROM längere Zeit benötigen und eher für den Einsatz von programmierbaren AUs denkbar wären. Mit Hilfe von n Bits können dann 2^n Wege geschaltet werden, so daß für einen Mul_C bei 32 Eingängen 2 * 5 Bits, für einen Mul_D 5 Bits zur Konfigurierung notwendig wären. Bild 10.10 zeigt die prinzipielle Struktur der Multiplexer.

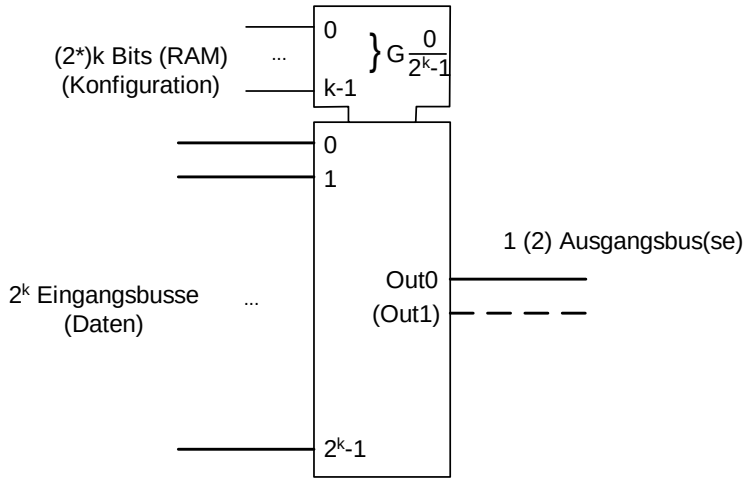


Bild 10.10: Multiplexerstruktur

In einer Modellarchitektur, die hier zu Erläuterungszwecken vorgestellt wird, seien folgende Teileinheiten und Register implementiert:

- 4 Addierer (AU), bei denen durch ein Condition Code Bit die Addition durchgeführt (TRUE) oder das erste Wort unverändert durchgelassen wird.
- 1 logische Einheit mit AND, OR, XOR und Zweier-Komplementbildung, konfigurierbar durch 2 Bits (AU)
- 1 Multiplizierer (AU)
- 1 Dividierer (AU)

- 1 Shift-Funktionseinheit (AU), die mit Hilfe von 2 Bits auf links/rechts und arithmetisch/logisch konfiguriert wird
- 2 dynamische Pfad-Multiplexer, die durch ein Bit im Steuereingang einen der beiden Eingangsbusse am Ausgang anliegen lassen (AU). Diese Multiplexer sollten nicht mit den Multiplexer Typ C oder D verwechselt werden, da die hier vorgeschlagene Teileinheit die Auswahl dynamisch schaltet.
- 6 Vergleichseinheiten mit je einem Bitausgang, konfigurierbar über 3 Bits auf 8 Vergleichsarten (CoU)
- 12 logische Register R0 bis R11, die in einem Pool von beispielsweise 24 physikalischen Registern per Register Renaming identifiziert werden.
- 4 Konstantenregister C0 bis C3, in denen die im Instruktionscode codierten Konstanten während der Bearbeitung eines Blocks gespeichert werden.

Dies ergibt eine Gesamtheit von 32 zu verbindenden Teilen innerhalb der s-Unit. Wird eine vollständige Verbindbarkeit gefordert, so müssen die Multiplexer im Wesentlichen mit 5 bzw. 2 * 5 Bits konfiguriert werden. Zur Verbindung aller Einheiten werden 10 Multiplexer Typ C, 12 vom Typ D und 6 Demultiplexer vom Typ E benötigt. Da angenommen wird, daß die Konditionierung durch Vergleichsoperationen sich nur auf AUs bezieht, benötigen die Demultiplexer nur 3 Bits zur Konfiguration. Damit ergibt sich die Gesamtzahl der Konfigurationsbits (für Multiplexer und konfigurierbare AUs) zu 200 Bits.

Diese Gesamtzahl soll lediglich einen Überblick zur Komplexität der Konfiguration geben. Um praxisnahe Zahlen zu erreichen, müssen umfangreiche Simulationen die real benötigten AUs und CoUs bestimmen. Weiterhin fehlt dem Modell bisher eine Behandlung von Flags, die durch gesonderte AUs mit Auswerteeigenschaften möglich wäre. Um einen Überlauf bei arithmetischen Operationen zu verhindern bzw. zu detektieren, sind genügend dimensionierte Datenbusse und entsprechende Auswerteeinheiten notwendig, die aus Übersichtsgründen fortgelassen wurden.

10.2.3 Der Befehlssatz eines >S<puters

Wie bereits bei den superskalaren CPUs gezeigt wurde, kann durch den Einsatz von bedingten Befehlen mit Vorhersagebits eine gute Hyperblockstruktur erhalten werden. Dies lässt sich auch beim >S<puter zeigen (in Beispielen, siehe nächsten Abschnitt), so dass die entsprechende Befehlssatzerweiterung an dieser Stelle erfolgt.

Für den Maschinenbefehlssatz des >S<puters werden zusätzlich folgende Befehle angenommen:

- PEQ <Dest>, <Source>, <Zielbit> (Gleichheit)
- PNE <Dest>, <Source>, <Zielbit> (Ungleichheit)
- PGE <Dest>, <Source>, <Zielbit> (größer oder gleich: <Dest> >= <Source>)

- PGT <Dest>, <Source>, <Zielbit> (größer als)
- PLE <Dest>, <Source>, <Zielbit> (kleiner oder gleich)
- PLT <Dest>, <Source>, <Zielbit> (kleiner als)

Eine Erweiterung dieses Befehlssatz für Setzen von Konditionsbits ist natürlich denkbar, insbesondere die bereits angedeutete Verknüpfung dieser Bits mit früheren Werten. Daneben müssen diese Konditionsbits auswertbar sein, was durch die Einführung von bedingten Verschiebe- und arithmetisch/logischen Befehlen erfolgen kann. Im Folgenden wird für die Modellarchitektur des >S<puters daher vorausgesetzt, dass alle Verschiebefehle mit einer Kondition belegbar sind (movp) und dass die arithmetisch/logischen Befehle so ausgeführt werden, dass der 1. Verknüpfungsoperand durchgelassen wird, falls die Bedingung nicht erfüllt ist. Im Fall der

addp <Ziel>, <Operand_1>, <Operand_2>, <Prediction_bit>

Verknüpfung wird also <Operand_1> in das Zielregister geladen, falls das <Prediction_bit> gelöscht ist, ansonsten die Summe von <Operand_1> und <Operand_2>.

Die weiteren Methoden zur Steigerung des Durchsatzes in >S<putern entsprechen denen für superskalare Mikroprozessoren. Hierzu zählen bezüglich der Programmierung in Assembler bzw. C (als Beispiel für eine Hochsprache):

- Nutzung der bedingten Ausführung von Befehlen zur Schaffung von größeren Blöcken ohne Kontrollflußstrukturen
- Unrolling von Schleifen bis zum Maximum der Ressourcen
- Abhängigkeitsanalyse und Beseitigung von Abhängigkeiten durch (Compile-Time-) Register Renaming

Nachdem dies zu einem Optimum für eine superskalare Architektur geführt wurde, werden die vorhandenen Blöcke erneut analysiert und in die strukturale Programmierung übersetzt. Diese Übersetzung kann zur Compilezeit erfolgen, wobei der Vorteil in der intensiven Analyse ohne Benutzung von Silizium im Zielsystem zu sehen ist. Die strukturale Programmierung wird dabei durch die vorhandene Abhängigkeitsanalyse und vor allem die *Beseitigung* erheblich unterstützt, so dass die Befehle in Datenflüsse und damit in die Struktur umsetzbar sind. Diese Struktur besitzt dann keine Zyklen oder Rückkopplungen, die für die Hardwarestrukturierung bei asynchronem Design unbrauchbar wäre.

10.2.4 Die Bestimmung von Ausführungszeiten

Der Performancegewinn ergibt sich aus der Ausnutzung zweier Vorteile gegenüber einer ‘klassischen’ Architektur eines superskalaren Mikroprozessors. Die ALU, die innerhalb der bisherigen Architektur vervielfacht wird, wird nunmehr aufgespalten, so dass die Einzelteile unabhängig voneinander nutzbar sind. Unter der Annahme, dass die Laufzeit innerhalb der programmierbaren Struktur so klein bleibt, dass die

Ergebnisse unabhängig von dem Datenflussweg innerhalb eines Takts vorliegen, ergibt dies eine im Schnitt bessere Ausnutzung und kleinere Ausführungszeiten.

Der zweite Vorteil liegt darin, dass Read-After-Write-Hazards in geeigneter Weise aufgelöst werden können. Read-After-Write bedeutet ja, dass auf ein Ergebnis nach Berechnung zugegriffen werden muss, um den weiteren Rechenfluss aufrechtzuerhalten. Im Fall der s-Paradigmen-Unit liegt dieses Ergebnis jedoch vor, bevor es gespeichert wird, und es kann innerhalb der Struktur bereits mit richtigem Wert weiterbenutzt werden. Dies ist gleichbedeutend mit dem Gewinn eines Ausführungstakts, der in der bisherigen Variante zur Speicherung durchlaufen werden müsste.

Die Steuerung bzw. Bestimmung der Ausführungszeit innerhalb der s-Unit nimmt jedoch einen zentralen Punkt innerhalb der Hardwareimplementierung ein. Folgende Verfahren bieten sich hierzu an:

- Die struktural programmierbare Hardware wird so konzipiert und mit dem maximal zulässigen Takt abgestimmt, dass die Laufzeit innerhalb der s-Unit für jeden Fall so dimensioniert ist, dass das Ergebnis nach einem Takt in den Registern speicherbar ist.
- Die Hardware, die in jedem Fall asynchron miteinander verknüpft ist (eine Synchronisierung erfolgt in jedem Fall erst bei den Registern, weshalb auch eine Abhängigkeitsbeseitigung notwendig ist), liefert beim Durchlauf ein Ready-Signal mit, das die Übernahme in die Register steuert. Diese Form lässt ggf. höhere Taktraten zu, wobei z.B. im Regelfall ein Takt zum Durchlauf benötigt wird und in Ausnahmefällen zwei Takte.

Im folgenden Abschnitt werden zwei Beispielprogramme analysiert, übersetzt und für eine superskalare Architektur bisheriger Implementierung optimiert. Die Geschwindigkeiten im Ablauf dieses Maschinenprogramms werden vergleichend zum >S<puter dargestellt. Diese Beispiele stammt aus [13], um die Ergebnisse entsprechend vergleichbar gestalten zu können.

10.2.5 Beispielprogramme für die >S<puterarchitektur

Das erste Beispiel wurde bereits an anderer Stelle zitiert: Eine bedingte Zuweisung an eine einfache Variable. Bild 10.11 zeigt den Sourcecode und die Assemblerübersetzung (mit p-Befehlen), Bild 10.12 die Umsetzung in die programmierbare Struktur:

```
int a, b;

if( a > 0 )
   b = a;
else
   b = 0;
```

(a)

```
load  r0, a;
pgt   r0, 0, p1;  Setzen von p1
movp  r2, r0, p1; Nur für a > 0
movp  r2, 0, /p1; Nur für a <= 0
store b, r2;
```

(b)

Bild 10.11: C-Sourcecode und Assemblerübersetzung mit Branch-Ersatz

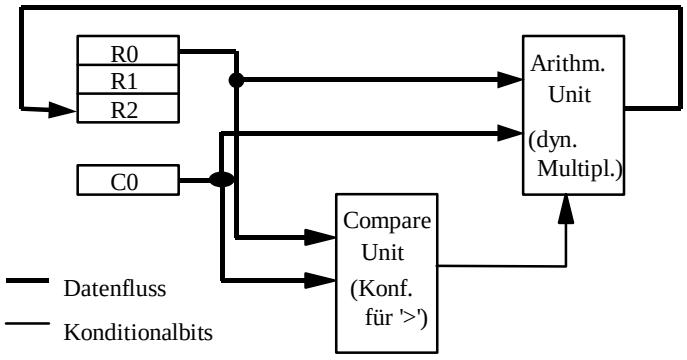


Bild 10.12: Datenflussübersetzung für Assemblercode

Die Übersetzung erfolgt durch Zuordnung eines Vergleichers, konfiguriert für ‘größer-als’ und eines dynamischen Multiplexers, der die Auswahl aus den beiden einfließenden Datenströmen anhand des Vergleichbits vornimmt, zu diesem Ausführungsblock. Die Load/Store-Befehle verbleiben in der bisherigen Form und sind nicht dargestellt. Zusätzlich muss das Register C0 (für Konstanten) mit dem Vergleichswert, hier ‘0’, geladen werden.

Während für die Bearbeitung obiger Zuweisung in einer superskalaren CPU sechs Takte benötigt werden (2 für die Load-Zugriffe, 1 für Bestimmung von p1, 1 für die bedingte Zuweisung, 2 für Store), reduziert sich dies im >S<puter bei Annahme der Taktunabhängigkeit der Ausführung auf fünf Takte: Zuweisung an p1 und bedingte Zuweisung laufen asynchron hintereinander ab, der Takt synchronisiert dies bei Schreibzugriff auf r2.

Bild 10.13 zeigt den C-Sourcecode des zweiten Beispiels. Das Programm besteht in einer Zuweisungsschleife an ein Array b[] in Abhängigkeit eines Arrays a[], wobei hier besonders die Read-After-Write-Abhängigkeiten zu analysieren sind.

```
int a[LIMIT], b[LIMIT];
int i, n;

...

for( i = 0; i < n; i++ )
{
    if( a[i] != 0 )
        b[i] = b[i] + b[i+1];
    else
        b[i] = 0;
}
```

Bild 10.13: C-Sourcecode

Label	Nr.	Instruktion		Zyklus
L0:		mov	r2, 0	
		mul	r6, n, 4	
		add	r0, b, 4	
L1:	1	ld	r1, mem(a+r2)	0
	2	beq	r1, 0, L3	2
L2:	3	ld	r3, mem(b+r2)	2
	4	ld	r4, mem(r0+r2)	2
	5	add	r5, r4, r3	4
	6	st	mem(b+r2), r5	5
	7	jmp	L4	5
L3:	8	st	mem(b+r2), 0	
L4:	9	add	r2, r2, 4	5
	10	blt	r2, r6, L1	6

Registerinhalte:

r0 = &b[1]
r1 = a[i]
r2 = i * 4
r3 = b[i]
r4 = b[i+1]
r5 = b[i] + b[i+1]
r6 = n * 4

Bild 10.14: Assemblercode (1. Optimierung)

Das interessante an diesem Code ist die Reihenfolge der Adressen, auf die jeweils zugegriffen wird, da die Zuweisung in einem Teil der Schleife b[i] + b[i+1] lautet und somit das zweite Element des 1. Zugriffs gleich dem ersten Element der zweiten Schleife ist. Die Übersetzung des C-Sourcecodes mit einem optimierenden Compiler, ausgelegt für tradionelle Architekturen, ergibt dann folgendes Assemblerlisting (Bild 10.14).

Der Kontrollflussgraph in Bild 10.15 zeigt die Wege, auf denen dieser Code durchlaufen wird. Der Compiler selbst hat den Assemblercode so optimiert, dass einige Basisblocks entstehen (in Bild 10.14 durch Linien getrennt). Die aufeinander-

folgenden Instruktionen 1 und 2 bedeuten beispielsweise, dass hier ein Read-After-Write-Hazard vorliegt: r1 wird zuerst beschrieben und kann erst danach zum Vergleich mit 0 gelesen werden. Die Abhängigkeit bewirkt, dass keine parallele Ausführung möglich ist. Die Spalte für den aktuellen Zyklus bezieht sich auf eine superskalare Architektur, die prinzipiell parallele Aktionen durchführen kann. Die Berechnung einer Schleife dauert im Maximalfall 6 Zyklen (bei angenommenen 2 Zyklen für einen Datentransfer zum Hauptspeicher), so dass für den *then*-Part bei 9 durchlaufenen Instruktionen 1,5 Instr./Zyklus ausgeführt werden.

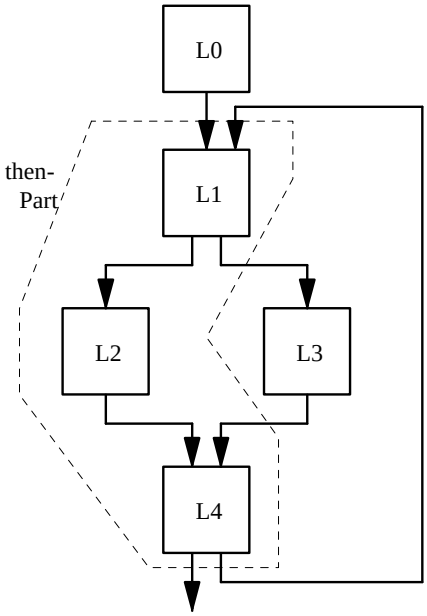


Bild 10.15: Kontrollflußgraph für Assemblierung (1.Optimierung)

Durch eine 1:1-Kopie des Blocks L4 für den *else*-Part, eine Veränderung der Reihenfolge der Anweisungen und den Ersatz der bedingten Verzweigung durch die bedingte Ausführung der Speicherbefehle kann eine Vergrößerung des Basisblocks erreicht werden. Das zugehörige Assemblerlisting zeigt eine Beschleunigung einer Schleife auf 4 Zyklen (bei Annahme, dass die letzte Verzweigung richtig vorhergesagt wird):

Label	Nr.	Instruktion	Zyklus
L0:		mov r2, 0	
		mul r6, n, 4	
		add r0, b, 4	
L1:	1	ld r1, mem(a+r2)	0
	3	ld r3, mem(b+r2)	0
	4	ld r4, mem(r0+r2)	0
	5	add r5, r4, r3	2
	2	**pne p1, r1, 0**	2
	6	stp mem(b+r2), r5 (p1)	3
	8	stp mem(b+r2), 0 (/p1)	3
	9	add r2, r2, 4	3
	10	blt r2, r6, L1	4

Bild 10.16: Assemblercode (2. Optimierung)

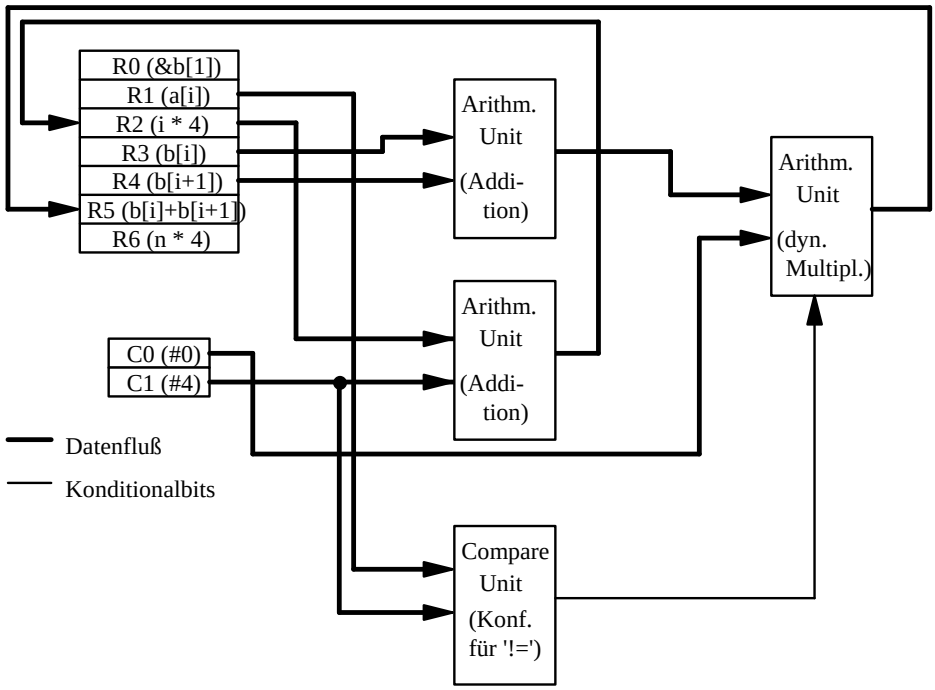


Bild 10.17: Datenflussübersetzung für Assemblercode (Bild 10.16)

Der Durchsatz pro Schleife vergrößert sich für eine superskalare Architektur durch diese Maßnahmen auf 2 Instr./Takt. Nimmt man nun an, dass die s-Paradigmen-

Unit ein beliebiges Schaltnetz in einem Takt bearbeiten kann, dann benötigt die Implementierung im >S<puter hingegen für alle Berechnungen einschließlich der bedingten Wertzuweisung einen Takt. Der Durchlauf pro Schleife verkürzt sich hierdurch auf 3 Takte, der Durchsatz beträgt 2,66 Instr./Takt. Bild 10.17 zeigt die Struktur in der s-Unit. Die Load/Store-Befehle, die der Kommunikation mit dem externen Speicher dienen, müssen ebenfalls arithmetische Berechnung für die effektive Adresse vornehmen. Diese sind hier nicht gezeichnet, obwohl prinzipiell auch Addierer aus der s-Unit für die Adressenaddition nutzbar wären.

Die letzte Stufe der hier gezeigten Optimierungen behandelt die Verbesserung der Performance durch ein Zusammenfassen zweier Schleifendurchläufe zu einem (*Loop Unrolling*) mit nachfolgender Abhängigkeitsanalyse und -behebung. Diese Optimierung steigert den Durchsatz, falls eine unabhängige, parallele Bearbeitung beider Teilschleifen möglich ist. Daher verwendet diese Methode zur Beseitigung von Abhängigkeiten ein Compile-Time Register Renaming. Bild 10.18 zeigt das Ergebnis nach Optimierung [13]:

```
Label      Nr.  Instruktion                      Zyklus
L0:             mov   r2, 0
                mov   r7, 4
                mul   r6, n, 4
                add   r0, b, 4
L1:        1    ld    r1, mem(a+r2)              0
          21    ld    r8, mem(a+r7)              0
           3    ld    r3, mem(b+r2)              0
           4    ld    r4, mem(r0+r2)             0
          24    ld    r9, mem(r0+r7)             0
           5    add   r5, r4, r3                 2
          25    add   r10, r9, r4                2
           2    pne   p1, r1, 0                  2
           2    pne   p2, r8, 0                  2
           6    stp   mem(b+r2), r5 (p1)         3
          26    stp   mem(b+r7), r10 (p2)        3
           8    stp   mem(b+r2), 0 (/p1)         3
          28    stp   mem(b+r7), 0 (/p2)         3
           9    add   r2, r7, 8                  3
          10    bge   r7, r6, L100               3
          29    add   r7, r2, 4                  3
          10    blt   r2, r6, L1                 4
L100:
```

Registerinhalte:

r0 = &b[1]
r1 = a[i]
r2 = i * 4
r3 = b[i]
r4 = b[i+1]
r5 = b[i] + b[i+1]
r6 = n * 4
r7 = (i+1) * 4
r8 = a[i+1]
r9 = b[i+2]
r10 = b[i+1] + b[i+2]

Bild 10.18: Assemblercode (3. Optimierung)

Durch die parallele Bearbeitung zweier Schleifen sinkt die Bearbeitungszeit auf durchschnittlich 2 Takte pro (ehemaliger) einfacher Schleife, wobei als Paralleli-

sierungsmaß, nunmehr 3,75 Instr./Takt ausgeführt werden. Dies gilt für die superskalare Architektur, während der >S<puter, im konkreten Modell dieses Abschnitts, eine weitere Steigerung hervorbringt.

Abgesehen von den Adressberechnungen werden vier Addition in einer Doppelschleife sowie zwei bedingte Zuweisungen gefordert. Diese Ressourcen sind im Modell vorhanden, das seinerseits mit Additionskapazitäten speziell für Schleifendurchführungen ausgestattet wurde. Damit lässt sich der gesamte Block der Additionen und Wertzuweisungen in einem Takt ausführen, wiederum unter Annahme, dass das Schaltnetz dies stabil während des Takts durchlaufen lässt. Die durchschnittliche Bearbeitungszeit pro einfacher Schleife liegt dann bei 3 Takten, dies ergibt eine Rate von 5 Instr./Takt.

10.2.6 Reconfigurable RISC

Das >S<puter-Prinzip – Aufsplittung der ALU in atomare Operationseinheiten und Konfiguration von Datenpfaden – lässt sich natürlich auch auf eine RISC-CPU anwenden. Dies wurde in [14] als rRISC-Architektur publiziert.

10.2.6.1 Basisarchitektur rRISC

Das Grundmodell, in Bild 10.19 basierend auf der Modell-CPU MPM3 (→ 4.4), wird durch einen Fetch Look-Aside Buffer ergänzt. Dieser Speicher arbeitet im Phasenpipelining parallel zur Decode/Load-Unit, falls es sich um das Schreiben von übersetzten Informationen handelt. Lesend wird dieser Speicher parallel zum Fetch im Hauptspeicher (bzw. Cache) genutzt.

Die Wahl des RISC-Grundmodells als 4stufige Pipeline stellt grundsätzlich keine Beschränkung dar. Wie noch gezeigt werden wird, geht nicht die Anzahl der Pipelinestufen in die Gewinn/Verlustrechnung zur CPI-Bestimmung ein, sondern die Anzahl der verlorenen Takte bei Fehlvorhersage (und spekulativer Ausführung). Bedingt durch diesen Umstand kann die vergleichsweise einfache Architektur zur Darstellung aller Effekte genutzt werden.

Die angestrebte Instruktionsparallelität selbst kann nur durch parallel ausführende Teileinheiten erreicht werden. Hierfür werden alle Instruktionen in folgender Weise klassifiziert:

1. Load/Store-Befehle: Hierzu zählen neben den Befehlen Load und Store selbst auch Push- und Pop-Instruktionen, die einen Zugriff auf den Stack ermöglichen.
2. Move-Befehle: Alle Kopier- (Move) und Austausch-Instruktionen (Xchg) zwischen Registern des Prozessors sind hier zusammengefasst.
3. Befehle zur Integer-Arithmetik.

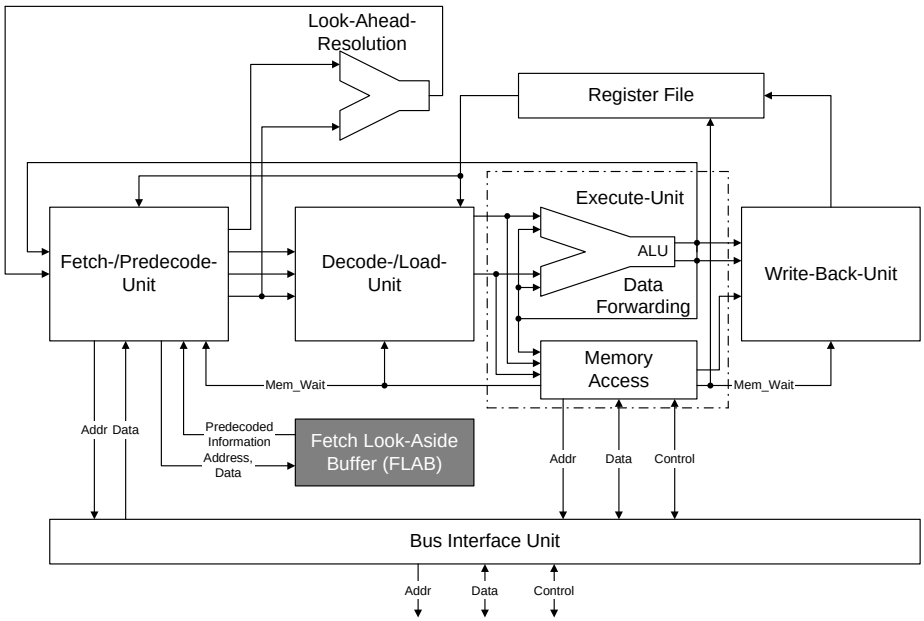


Bild 10.19: Blockarchitektur einer 4stufigen RISC-Architektur mit ergänzendem FLAB

4. Logische Befehle: In dieser Klasse sind alle logischen Operationen einschließlich der Shift- und Rotationsoperationen zusammengefasst.
5. Floating-Point Operationen (entfallen in diesem Fall).
6. Kontrollflussbefehle: Bedingte und unbedingte Sprungbefehle bilden – unabhängig von der gewählten Sprungzieladressierung – eine weitere Klasse von Befehlen.
7. NOP (No Operation): Die NOP-Instruktion wird ggf. zur programmtechnischen Verzögerung genutzt; sie dient allerdings gelegentlich auch als Fülloperation, die im Programmspeicher selbst als Platzhalter genutzt wird, ohne irgendeine Funktion zu haben.
8. Weitere Kontrollflussbefehle: Je nach Mikroprozessortyp können weitere Befehle wie STOP und WAIT existieren, die direkten Einfluss auf Ausführungsmodi haben. Alle weiteren Befehle sind in der letzten Klasse zusammengefasst. Weitere Befehle, die die Flags beeinflussen (z.B. SEC, Set Carry) und darüber den Kontrollfluss steuern können, zählen in diesen Bereich.

Bei einer angestrebten Parallelisierung der Befehlsausführung bietet es sich im einfachsten Fall an, mehreren Instruktionen aus disjunkten Klassen einen Ausführungsslot anzubieten. Bei der hier gewählten Architektur ist die Klasse 5 leer. Als

erster Ansatz zur parallelen Ausführung werden Alle Befehle der Klasse 1 zur Klasse M (Memory Access), alle Befehle der Klassen 2, 3 und 4 zur Klasse I (interne Befehlsausführung) sowie alle Kontrollflussbefehle der Klasse 6 zur Klasse C (Kontrollfluss, control flow) gezählt. Je ein Befehl der Klassen M, I und C können parallel zur Ausführung kommen und werden in einer Struktur im FLAB gespeichert. Diese Version wird als rRISC Level 1 bezeichnet.

Die Befehlsklasse 7 stellt eine Besonderheit dar. Wird ein NOP-Befehl als Platzhalter genutzt, kann seine Ausführung gänzlich entfallen. In diesem Fall bedeutet die Integration des NOP in einer Befehlszusammenfassung nur, dass ein Befehlszähler um eine Einheit inkrementiert werden muss, andere Informationen sind nicht notwendig. Befehle der Klasse 8 bleiben bei Auswertung paralleler Instruktionen aktuell unberücksichtigt.

10.2.6.2 rRISC Level 2 und 3

Neben dem Basisansatz, Instruktionen aus verschiedenen Klassen zu kombinieren, bietet sich die Möglichkeit, auch innerhalb der Befehlsklassen nach Parallelisierungsmöglichkeiten zu suchen. In den Forschungsarbeiten wurden zwei Strategien verfolgt: Zum einen können insbesondere bei den arithmetisch/logischen Befehlen Operationen identifiziert werden, die in disjunkten ALU-Bereichen ablaufen, zum anderen können sehr preiswerte Funktionen wie move (Kopie eines Registers in ein anderes) vervielfacht werden.

	Instructions	Parallelisation degree
Level 1	1 M-class 1 C-class 1 I-class	3 (4)
Level 2	1 M-class 1 C-class 1 AL-subclass (A, DI, L) 4 MC-subclass	7 (11)
Level 3	1 M-class 1 C-class 1 A-subclass 1 L-subclass 1 DI-subclass 4 MC-subclass	9 (13)

Tabelle 10.2: Definition der rRISC-Level für das modifizierte MPM3-Modell

Zu diesem Zweck werden die Instruktionen weiter unterteilt: Die Subklasse MC (Move/Copy) enthält alle Befehle, die Daten zwischen Registern verschieben oder

kopieren. Die arithmetischen und logischen Operationen werden weiterhin in die Subklassen A (arithmetisch), L (logisch und DI (Dekrement/Inkrement) unterteilt, so dass die Klasse I nunmehr durch die Subklassen A, L, DI und MC dargestellt werden kann. Tabelle 10.2 stellt dar, wie sich die verschiedenen rRISC-Level durch ihre Parallelitäten unterscheiden.

Der maximale Parallelitätsgrad muss noch kurz erläutert werden. Die erstgenannte Zahl (Level 3: 9) entsteht durch die Summe über die Anzahl der Instruktionsklassen mit der Anzahl der Instruktionen pro Klasse, jeweils ohne Berücksichtigung der NOPs. Die zweite Zahl (Level 3: 13) entstegt dadurch, dass die MC-Subklasse pro Instruktion jeweils ein MOV/MOVH-Paar mit unmittelbarer Adressierung (jeder Befehl enthält nur 8 Bit Daten, zusammen 16 Bit) enthalten kann.

10.2.6.3 Aufbau und Funktionsweise des Fetch Look-Aside Buffers

Der Fetch Look-Aside Buffer (FLAB) ist auf zweifache Weise mit der Fetch-Einheit gekoppelt (Bild 10.20). Jede aus dem Speicher geladene Instruktion wird an den FLAB übermittelt und im algorithmischen Teil in eine temporäre Zeile integriert. Bei Beginn eines Fetches wird parallel zum Speicher im FLAB gelesen, ob eine Zeile mit der entsprechenden Startadresse gespeichert ist. Liegt ein FLAB-Hit vor, so werden alle Zeileninformationen an die Fetch-Einheit übertragen, und der Speicherzugriff kann abgebrochen werden.

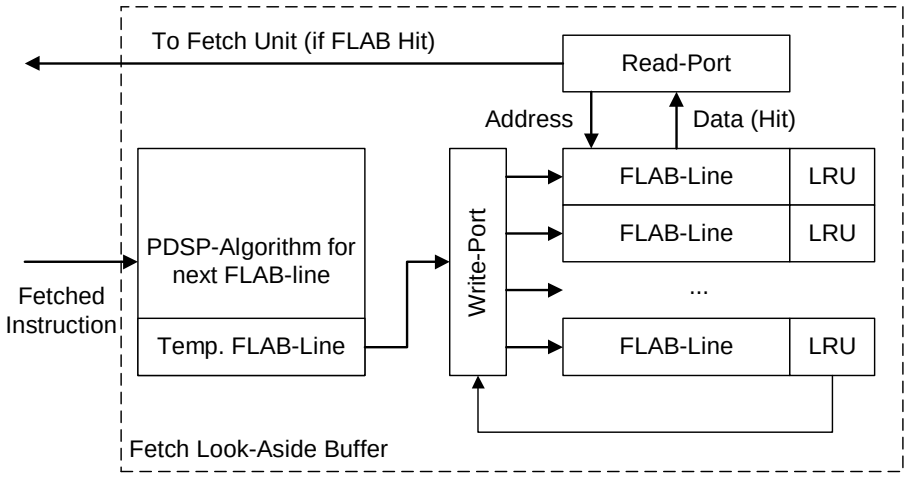


Bild 10.20: Mikroarchitektur des Fetch Look-Aside Buffers

Im Fetch Look-Aside Buffer selbst werden die Informationen gespeichert, die durch eine Form des Code Morphings aus den sequenziellen Befehlen gewonnen

werden. Hierzu wird eine Gruppe von Befehlen zwecks späterer paralleler Ausführung in folgender Form gespeichert:

Start Address	No. of Instr.	Class I Code (arithm./logical or move)	Class M Code (memory access)	Class C Code (control flow)

Bild 10.21: Speicherformat FLAB-Zeile

Bild 10.21 zeigt das Speicherformat für rRISC Level 1. Mit Hilfe des nachfolgend beschriebenen Algorithmus wird ein in den Prozessor kommender Instruktionsstrom in die entsprechenden Zeilenabschnitte gespeichert. Hierbei erfolgen im Sinn eines Code Morphing einige Umwandlungen.

Zur Speicherung der I-Instruktionen werden Operationscode, Quellregister bzw. unmittelbare Daten als Quellen und das Zielregister benötigt. Die unmittelbaren Daten – bei RISC-Prozessoren mit einem Instruktionsformat, dessen Breite identisch mit der Datenbreite ist, können diese nicht mit der vollen Datenbreite gespeichert werden – werden entsprechend den Konventionen auf volle Breite erweitert. Präfix-Instruktionen zur Erweiterung der Konstanten können hier direkt integriert werden, so dass dies einer Verschmelzung der Befehle mit den Daten entspricht.

M-Instruktionen werden unverändert gespeichert, da hier keine Zusammenfassungen möglich sind. Bei C-Instruktionen wird als Sprungziel immer die vollständige Adresse geschrieben, außerdem lassen sich alle Hinweise zur Sprungvorhersage in der FLAB-Zeile sichern. Bei der hier dargestellten Lösung, basierend auf dem MPM3-Modell, werden pro Zeile im FLAB-Speicher 157 Bits, zusätzlich für die Datenhazard-Informationen für nachfolgende Instruktionen (zur Erkennung von Read-after-Write-Hazards) 20 Bits benötigt.

10.2.6.4 Übersetzungsalgorithmus für Code Morphing

Innerhalb des algorithmischen Teils des FLAB wird der ursprüngliche Code in einen Code mit Strukturinformationen umgewandelt. Dies erfolgt, um Datenabhängigkeiten zu erkennen und eine parallele Ausführbarkeit der integrierten Operationen zu ermöglichen. Die Umwandlung stellt eine Abwandlung des Code Morphing dar.

Der hier verwendete Algorithmus wird mit Procedural Driven Structural Programming (PDSP) genannt, weil aus den sequenziellen Instruktionen eine Strukturinformation gewonnen wird. Im Wesentlichen besteht PDSP aus den folgenden Schritten:

1. Die aktuelle Instruktion wird gemäß ihrer Klassenzugehörigkeit auf ihre Übersetzbarkeit geprüft. Dies führt ggf. zur Beendigung der aktuellen FLAB-Zeile. Als integrierbar sind bei rRISC alle Befehle der I-, M- und C-Klasse sowie der NOP-Befehl definiert.

2. Ist die Instruktion übersetzbar, wird geprüft, ob eine freie Ressource vorhanden ist. Eine negative Antwort führt zur Beendigung, eine positive zur vorläufigen Belegung dieser Ressource und Eintragung in die Strukturinformation. Ein NOP-Befehl wird als Spezialfall durch einfaches Erhöhen des Befehlszählers integriert.
3. Nach der vorläufigen Übersetzung werden die Datenabhängigkeiten überprüft. Im Allgemeinen können Datenabhängigkeiten (RAW-Hazard) in die Struktur übersetzt werden; im speziellen Fall der rRISC-Architektur wurde jedoch auf ein Data Forwarding innerhalb der Struktur verzichtet, so dass Datenabhängigkeiten zur Beendigung des Algorithmus führen.
4. Als Spezialfall werden MOV/MOVH-Instruktionspaare (mit unmittelbarer Adressierung) der Modellarchitektur als ein Befehl behandelt. Ein MOV #-Befehl kann die unteren 8 Bits in ein Register laden, ein MOVH #-Befehl die oberen. Ein direktes Aufeinanderfolgen mit identischem Zielregister wird als Instruktionspaar gewertet und entsprechend übersetzt.

Als Endkriterien für eine einzelne FLAB-Zeile sind also nicht-integrierbare Instruktionen, Ressourcenbelegung und Datenabhängigkeiten definiert. Ist eine Zeile beendet und enthält sie mehr als eine Instruktion, so wird sie im Pufferbereich gespeichert. Der Ersatz von älteren FLAB-Zeilen erfolgt über einen Least-Recently-Used- (LRU-) Algorithmus, der Speicher ist im Übrigen voll-assoziativ aufgebaut.

Im Phasenschema des RISC-Prozessors wird am Ende des 2. Takts einer Befehlsbearbeitung, also parallel zum Ende der Decode-/Load-Phase, das Ergebnis der Instruktionsübersetzung im temporären Teil gespeichert. Parallel hierzu sind die Bestimmung eines Endkriteriums sowie die Auswertung des LRU-Algorithmus erfolgt, so dass auch die Speicherung in einer regulären FLAB-Zeile erfolgt. Damit steht das Übersetzungsergebnis 2 Takte nach Beginn der Befehlsbearbeitung zur Nutzung im Mikroprozessor bereit. Dies bedeutet, dass auch im extremen Fall einer Programmschleife mit 2 Instruktionen (I-Instruktion und anschließendes Branch, z.B. für Wartezähler) diese ab der zweiten Ausführung als Struktur verfügbar ist, da jede verzweigende Branch-Instruktion bei der ersten Ausführung 2 Takte benötigt (nach Look-Ahead-Resolution).

Die Einträge in der FLAB-Zeile bleiben unverändert, mit einer Ausnahme. Diese Ausnahme besteht in den Verzweigungsinformationen, die ständig aktualisiert werden. In dem hier gewählten Modell wurde eine dynamische Branchvorhersage mit 1 Bit gewählt, sodass in jedem FLAB-Eintrag mit Branchbefehl der letzte Verzweigungsweg aktuell gespeichert ist.

10.2.6.5 Beispiel für die Funktion des FLAB

Das folgende Beispiel dient der Verdeutlichung des Effekts, den man mit Hilfe des FLAB und der Integration von Instruktionen in eine parallel ausführbare Struktur erreichen kann. Es ist einer Patenschrift der Firma Transmeta entnommen, wo es ebenfalls zur Darstellung der Leistungsfähigkeit des Code Morphing dient.

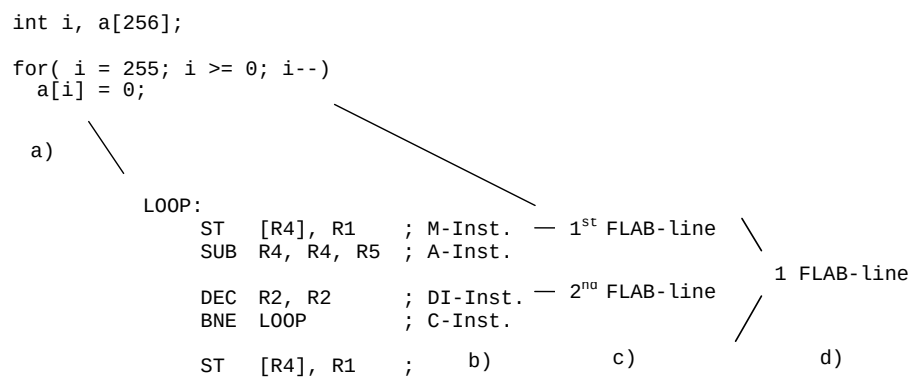


Bild 10.22: Beispiel zur Übersetzung in FLAB-Zeile
a) C-Sourcecode b) Assemblerübersetzung für Modell-CPU
c) FLAB-Zuordnung (Level 1) d) FALB-Zuordnung (Level 3)

Im Rahmen einer kurzen Programmschleife wird ein Array initialisiert. Bild 10.22 zeigt die entsprechende Assemblerübersetzung aus Übersichtsgründen ohne Initialisierung der Register. Die ersten beiden Instruktionen lassen sich in einer FLAB-Zeile zusammenfassen, da hier keine RAW-Hazards auftreten (R4 wird erst nach dem Speichern ST, wo es als Ziel dient, um 2 (in R5 stehend) dekrementiert). Gleiches gilt für die Kombination der beiden unteren Befehle, obwohl der Branchbefehl von dem Ergebnis des Dekrementbefehls abhängt. Hier tritt eine Branchvorhersage auf.

Bei korrekter Vorhersage (ab 2. Durchlauf) wird die Assemblerschleife bei einer RISC-Architektur in 4 Takten, bei rRISC Level 1 in 2 Takten und bei rRISC Level 3 in 1 Takt durchlaufen.

10.2.6.6 Programmausführung und Simulationsergebnisse

Ein kritischer Aspekt bei der Einführung der neuen Architektur ist derjenige, dass ggf. Fehlvorhersagen bei bedingten Verzweigungen zu einer erhöhter Anzahl an Takten zur Wiederherstellung des korrekten Programmzustands führen. Die Ausführung eines Branch-Befehls benötigt bei der Modell-CPU in der RISC-Variante 1 Takt bei korrekter Vorhersage, 2 Takte bei Fehlvorhersage. In der rRISC-Variante wird die Branch-Instruktion, falls sie in einer FLAB-Zeile integriert ist, parallel zu einer anderen Instruktion ausgeführt, benötigt selbst also keinen Takt. Die Abhängigkeit von vorangegangenen Instruktionen, die in der gleichen FLAB-Zeile sein können, bewirkt jedoch, dass 2 Takte Verlust bei Fehlvorhersage auftreten.

Die Verlusttakte für FLAB-Zeile und normale Branch-Instruktion stimmen in diesem Fall überein, sodass keine erhöhte Anzahl bei Fehlvorhersage auftritt. Im Allgemeinen lässt sich eine Formel für den Fall angeben, dass keine zusätzlichen

Maßnahmen wie Wartezyklen eingefügt werden müssen, um irreversible Ergebnisse zu verhindern.

Für diese Formel sei ein *k*-stufiges Pipelining angenommen. Am Ende der Stufe *kf* sei der Fetch beendet, am Ende der Stufe *kst* seien die Statusflags, die einen Branch beeinflussen, per Data Forwarding verfügbar, und *kexe* sei die Stufe, die irreversibel Ergebnisse speichern würde. Unter diesen Annahmen muss die Ungleichung

$$kexe \geq kst - kf \qquad (10.1)$$

gelten, um ohne Taktverlust (bei Branchvorhersage) rRISC einführen zu können.

Die nachfolgenden Testprogramme wurden mit Hilfe eines VHDL-Modells für die rRISC-Variante einer Modell-CPU zyklusgenau simuliert. Hierbei wurden keinerlei Speichereffekte wie Cache-Konfiguration, langsamer Hauptspeicher etc. in Betracht gezogen, lediglich eine von-Neumann-Konfiguration (ein Port zum Speicher) und Harvard-Konfiguration (zwei voneinander unabhängige Ports) wurden simuliert.

Die gewählte Konfiguration umfasste die Integration je einer I-, M- und C-Klasse-Instruktion in einer FLAB-Zeile. Dies wird als rRISC-Level 1 bezeichnet und wurde mit einer gleichartigen RISC-Architektur mit dynamischer Branchvorhersage (1-Bit) verglichen (Level 0). Die Anzahl der FLAB-Zeilen, die gleichzeitig gespeichert war, wurde dabei variiert.

Zum Test dienten folgende Programme (wie schon bei MPM3, → 4.6):

INIT_ARR: Dieses Programm initialisiert ein eindimensionales Integer-Array mit 0. Der zulässige Wertebereich des Index ist 0 .. 255.

MOV_AVRG: Für die insgesamt 64 Elemente eines eindimensionales Integer-Array werden für jeweils 4 Elemente ein Mittelwert berechnet. Anschließend wird der Beginn der Mittelwertroutine um 1 Element verschoben, so dass insgesamt 61 Mittelwerte in einem zweiten Array zu speichern sind.

PARITY: Die gerade Parität aller Bitmuster (für 8 Bits) wird in diesem Programm bestimmt und zusammen mit dem originalen Byte als Bit 8 am Outputport ausgegeben.

WORDCOUN: Dieses Programm entspricht der inneren Routine von Wordcount, wie es als Tool für Unix-Betriebssysteme bekannt ist. Es zählt Character, Wörter und Zeilen in dem Text, der als Programmkonstante im Assemblercode angefügt ist.

CRC-8: Die Checksummenberechnung, wie sie z.B. im Header der ATM-Zellen zu finden ist, wird in diesem Programm beschrieben. Hierin befinden sich drei Abschnitte:

Im ersten Abschnitt wird die Syndrom-Tabelle ausgefüllt. Diese Tabelle dient der schnellen Berechnung der CRC-8 Checksumme für 4-Byte Header (diese CRC wird als fünftes Byte angefügt).

Der zweite Teil besteht in der Generierung der Fehlertabelle. Die 256 Einträge hier lassen eine Bestimmung des zu korrigierenden Fehlers im Header (5 Byte, also einschließlich CRC-8) zu.

Der dritte Teil testet dann einen korrekten Zellheader, 40 davon abgeleitete mit korrigierbarem 1-bit-Fehler sowie 40 unkorrigierbare (gleichwohl detektierbare) Header.

SEL_SORT: Das bekannte Sortierverfahren Selection Sort wird in diesem Programm codiert und an zwei Arrays mit je 100 Elementen getestet.

QUICKSRT: Das bekannte Sortierverfahren Quicksort wird in diesem Programm codiert und an zwei Arrays mit je 100 Elementen getestet.

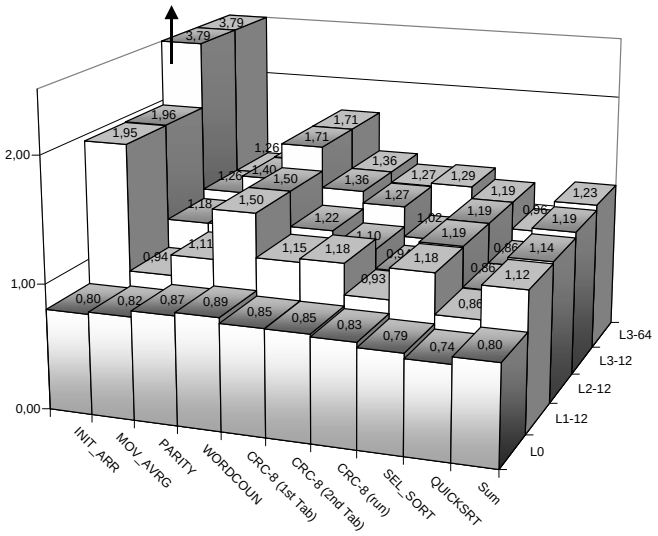


Bild 10.23: Simulationsergebnisse für rRISC

Alle Testprogramme zeigen IPC-Werte von etwa 0,8 für die RISC-Architektur. Die Steigerung mit wachsendem rRISC-Level und Anzahl der FLAB-Zeilen bleibt immer < 2 IPC, mit Ausnahme des bereits erwähnten Testprogramms zur Initialisierung einer Arrays. Das Programm Quicksort (rekursive Programmierung!) bleibt bei einem IPC-Wert < 1. Dieses Programm zeigt einen so unregelmäßigen Verlauf, dass es selten zu einem zweiten Durchgang durch gespeicherte Programmbereiche kommt, sodass eine wesentliche Beschleunigung entfällt.

Die Werte für eine Harvard-Architektur mit getrennten Ports für Code und Daten liegen alle um ca. 0,12 IPC höher.

10.2.6.7 Schätzung der Flächeneffizienz

Aus den gegebenen Performancewerten, gemessen in IPC und Schätzungen des Flächenbedarfs aus den VHDL-Beschreibungen lässt sich eine relative Flächeneffizienz für die einzelnen Varianten angeben. Hierbei wird ein quantitativer Zusammenhang zwischen Ausführungszeit T und Flächenbedarf A zugrunde gelegt:

$$A \cdot T^n = Const. \quad mit \; n = 1 \; .. \; 2 \tag{10.3}$$

Gleichung (10.3) zeigt einen allgemeinen Zusammenhang, der zunächst für einzelne Operationen gefunden wurde, jedoch auch für komplette Mikroprozessorarchitekturen genutzt wird. Mit n = 2, dieser Wert wird bei arithmetischen Schaltungen meist angenommen, kann die Flächeneffizienz

$$E = \frac{1}{T \cdot \sqrt{A}} \tag{10.4}$$

definiert werden. Bild 10.24 zeigt die resultierenden Werte, bezogen auf die Grundversion des Mikroprozessors (L0). Diese wurden durch Bestimmung der erzeugten Signale im Design als relative Zahl für *A* und 1/IPC als relativer Wert für *T* erhalten.

Die Darstellung in Bild 10.24 zeigt für den simulierten 16-Bit-Mikroprozessor, dass die Werte für die Effizienz bei steigender Performance sinken. Die maximale Effizienz liegt hier beim Basismodell, und hierfür folgende Gründe gegeben werden:

- Das 16-Bit-Mikroprozessormodell ist sehr kompakt ausgelegt, sodass die Anzahl der internen Signale recht klein ist. Insbesondere die Anzahlen der internen Register (13) und Operationen sind recht klein. Bezogen auf den kompakten Grundwert steigt der Aufwand für den FLAB-Speicher stark an.
- Neben der Eigenschaft, die Zusammenfassung der Befehle zu Strukturen zu speichern, kann der Fetch Look-Aside Buffer auch als Cache wirken. Dies ist allerdings in den Schätzungen nicht berücksichtigt.

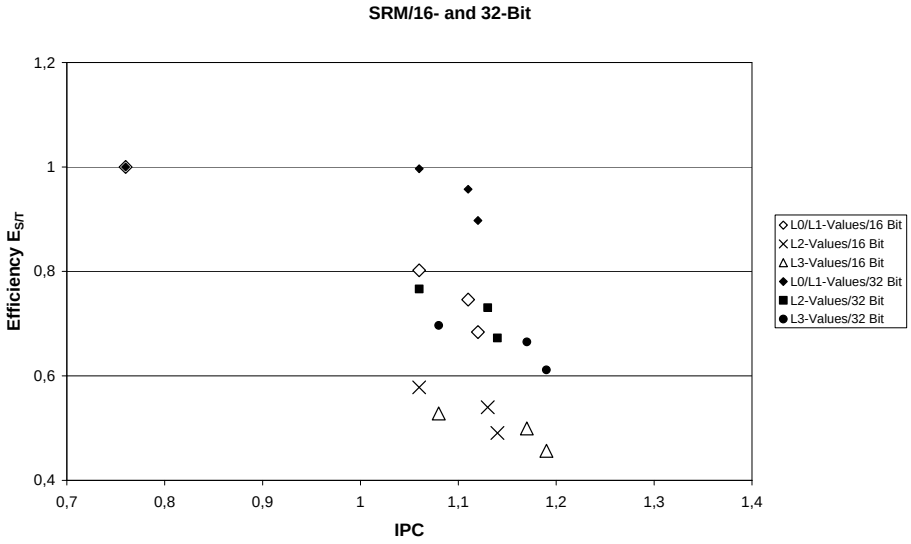


Bild 10.24: Zur Flächeneffizienz des rRISC-Ansatzes in Abhängigkeit von der Performance

Aus den Werten für das 16-Bit-Prozessormodell wurden dann Werte für eine komplexere 32-Bit-Mikroprozessorarchitektur extrapoliert. Die Werte (Bild 10.24) zeigen dabei ein deutlich anderes Bild: Die Effizienz sinkt erst bei höheren Anzahlen von FLAB-Zeilen. Während eine sinkende Flächeneffizienz bei gesteigerter Performance, ausgedrückt in IPC-Werten, beim Übergang von skalaren zu superskalaren Architekturen den Normalfall darstellt, lässt sich durch Einführung des FLAB-Konzepts eine gleichzeitige Effizienz- und Performancesteigerung für komplexere Mikroprozessoren bzw. Mikrocontrollerkerne erzielen oder zumindest der Effizienzverlust mindern.

10.2.7 UCB/UCM-Konzept

Die s-Unit des >S<puter-Konzepts sowie der Übersetzungsalgorithmus PDSP lassen sich aus dem Konzept des superskalaren Mikroprozessors herauslösen und einzeln bzw. in neuer Zusammensetzung benutzen. Hieraus ergibt sich eine interessante Mischung aus Programmier- und Ausführungsmodell.

Das optimale Modell besteht – nach aktuellem Kenntnisstand – darin, dass die Programmcodierung einem sequenziellen Modell folgen kann, die Ausführung jedoch an die aktuellen Randbedingungen (auch zur Laufzeit) angepasst werden kann.

Genau dies wird von dem UCB/UCM-Modell (Universal Configurable Block/ Machine) unterstützt. Es wird daher in diesem Abschnitt näher vorgestellt.

10.2.7.1 Ausführungsphasen dieses Modells

Die ausführenden Blöcke (UCB) benötigen im Prinzip eine Konfiguration, die den Datenweg innerhalb des Blocks für eine Zeitspanne festlegt. Mehr noch als das, eigentlich ist ein Konfigurationsfluss notwendig, da nicht davon ausgegangen werden kann, dass im allgemeinen Fall eine einzige Konfiguration etwa eine komplette Schleife beinhaltet.

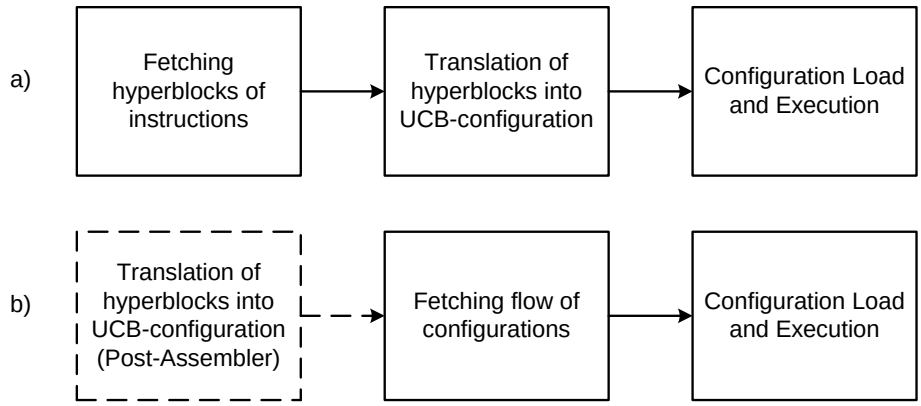


Bild 10.25 Ausführungsphasen im UCM-Modell
a) mit Runtime-Übersetzung (PDSP) b) mit Post-Assemblerlauf

Dieser Konfigurationsfluss kann aus dem Instruktionsfluss gewonnen werden, dies ist aus der >S<puter-Architektur bekannt. Damit ergeben sich 3 (grobe) Ausführungsphasen, die nicht mit dem Phasenpipelining einer RISC-Architektur verwechselt werden dürfen (Bild 10.25a): *Instruction Fetch*, *Translation* und *Configuration Load and Execution*.

Im Fall einer Übersetzung zur Compilezeit – dies wird als Post-Assemblerlauf bezeichnet, unterliegt aber den gleichen Regeln wie die Übersetzung zur Laufzeit – verringert sich die Anzahl der Grobphasen auf 2 (Bild 10.25b). Unabhängig davon, wann diese Übersetzung durchgeführt wird, gilt jedoch, dass das Programmiermodell mit dem des zugrundeliegenden Mikroprozessors übereinstimmt, so dass die bekannten Compiler einschließlich aller Optimierungen weiterhin benutzt werden können.

Die nicht starr ausgeführte Kopplung zwischen Fetch/Übersetzung (wie sie definitiv bei Mikroprozessoren der Fall ist) ermöglicht den wichtigen Schritt, die Konfiguration solange auszuführen, wie sie benötigt wird: Dieses Kriterium wurde in Abschnitt 10.1 als Kriterium für Reconfigurable Computing genannt.

10.2.7.2 Universal Configurable Blocks

Ein Universal Configurable Block (UCB) (Bild 10.26) besteht wie die s-Unit des >S<puter aus Operatoreinheiten und konfigurierbaren Datenpfaden. Gegenüber dem >S<puter wurden nur wenige Teile ergänzt:

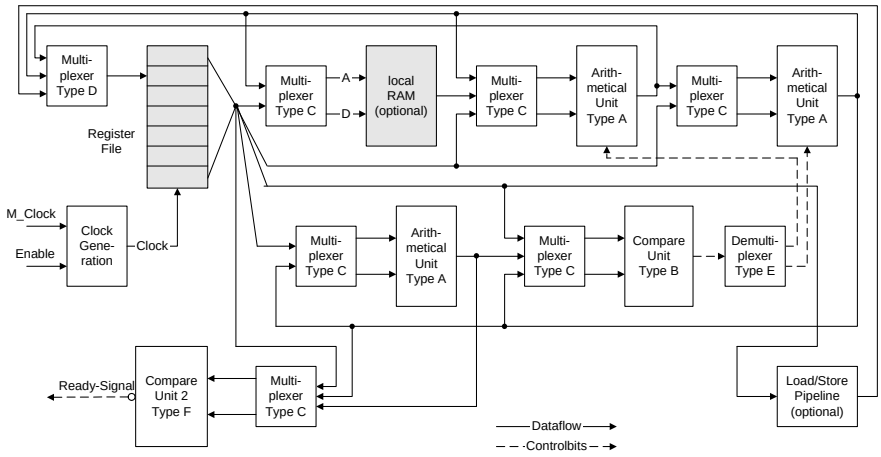


Bild 10.26: Mikroarchitektur Universal Configurable Block (UCB)

- *Arithmetic Unit* (AU, Type A): Die AU enthält eine oder wenige, dann konfigurierbare Verknüpfungen, beispielsweise die Möglichkeit, zwei Integerzahlen zu addieren. Sie liefert ein Ergebnis an ihrem Ausgang, wenn die Eingänge entsprechend beschaltet werden, und dieses Ergebnis kann innerhalb des Schaltnetzes weiterverwendet werden. Die AU ist gekennzeichnet durch zwei Eingangsbusse und einen Ausgangsbus, gegebenenfalls durch eine Konfigurationsmöglichkeit (Auswahl der Verknüpfung, im Extremfall entsprechend einer ALU) und durch Konditionalbits. In einigen Fällen, beispielsweise bei der Multiplikation, kann die Breite des Ausgangsbusses abweichen, um die Berechnungen zu ermöglichen.
- *Compare Unit* (CoU, Type B): Für die bedingte Ausführung einer Verknüpfung werden Condition Code Bits durch einen konfigurierbaren Vergleich in der CoU erzeugt. Der Vergleich läßt sich bei Verwendung von 3 Konfigurationsbits auf >, >=, <, <=, !=, ==, TRUE oder FALSE einstellen. Kennzeichen der CoU sind zwei Eingangsbusse und ein Ausgangsbit sowie die Konfigurierbarkeit.
- Multiplexer (Mul_C, Type C): Multiplexer vom Typ C belegen die Eingänge der AUs und der CoUs mit jeweils zwei Eingangswerten (in der vollen Verarbeitungsbreite). Hierfür benötigen sie eine Anzahl von Konfigurierungsbits, die sich aus der Anzahl von Eingängen und den zwei Ausgängen ergeben. Kennzeichen für Mul_C-Einheiten sind zwei Ausgangsbusse.

- Multiplexer (Mul_D, Type D): Für die Belegung der Register mit den Ergebniswerten wird lediglich ein Ausgangsbus und damit auch nur die Hälfte der Konfigurationsbits benötigt. Ein Mul_D unterscheidet sich daher vom Mul_C durch die Anzahl seiner Ausgangsleitungen.
- Demultiplexer (Demul, Type E): Die Ergebnisse der Vergleiche (CoU) müssen an entsprechende AUs weitergeleitet werden. Im Gegensatz zu den Multiplexern, die eine Quellenauswahl vornehmen, liegt hier eine Zielauswahl vor.
- *Compare Unit 2* (CoU2, type F): Diese (optionale) Einheit erzeugt ein (konfigurierbares) Ready-Signal, das von Steuereinheiten genutzt werden kann, z.B. von dem Hyperblock-Sequencer (siehe nächsten Abschnitt).
- Zusätzliche Hardware zur Synchronisierung des Takts mit in- und externen Steuersignalen.
- Privates RAM (Scratchpad-RAM) zur Unterstützung von Daten-intensiven Algorithmen. Insbesondere Algorithmen der Signalverarbeitung oder Interpolationen basieren auf Datenfeldern, die wichtige Parameter enthalten, meist aber die Größe von Registern überschreiten.. Diese Algorithmen können durch ein privates RAM unterstützt werden.

10.2.7.3 Procedural-driven Structural Programming

Der Übersetzungsalgorithmus, der zur Erzeugung einer Konfiguration aus einem Instruktionsfluss benötigt wird, entspricht demjenigen für rRISC und >S<puter (→ 10.2.6.4) und wird Procedural-Driven Structural Programming (PDSP) genannt. In speziellen Fällen werden zusätzlich Datenabhängigkeiten erkannt und aufgelöst, wie in Bild 10.27 dargestellt.

In diesem Beispiel wird der Inhalt von r1 (Variable *a*) mit der Konstanten 0 verglichen, abhängig davon wird der Inhalt in r2 (Variable *b*) kopiert oder nicht. Dies ist eine klare Datenabhängigkeit (RAW), die aber bei der Übersetzung in die Konfiguration aufgelöst wird und 'lediglich' eine Verlängerung der Bearbeitungszeit nach sich zieht.

Die Übersetzung von Verzweigungsbefehlen muss speziell berücksichtigt werden. Während alle Branches, die aus Verzweigungen stammen, prinzipiell in bedingte Befehle (siehe Bild 10.27) übersetzt werden können, gilt dies nicht für Rückwärtssprünge, deren Herkunft in den Schleifen liegt. Schleifen können zwar entrollt werden, dies liefert im allgemeinen Fall nur eine Verringerung der Anzahl der Sprünge.

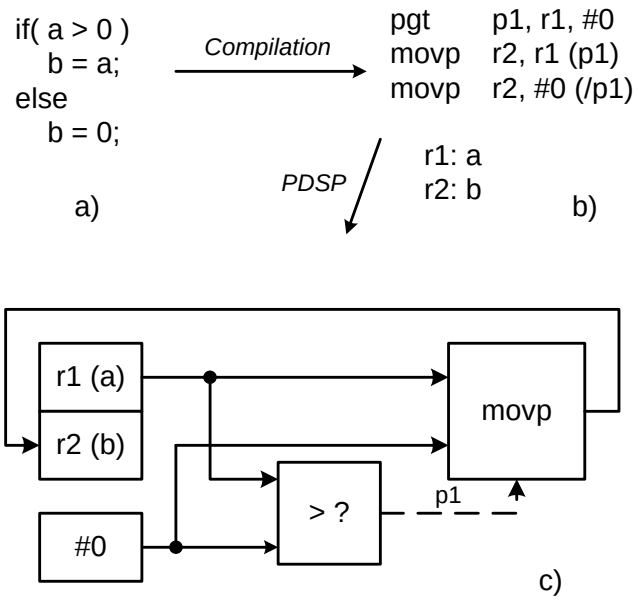


Bild 10.27 PDSP-Übersetzung am Beispiel einer IF-Verzweigung
a) C-Sourcecode b) Assemblercode mit bedingten Befehlen c) Konfiguration für UCB

Konsequenterweise müssen die UCBs (oder die übergeordnete Maschine) Verzweigungen behandeln können. Eine solche Verzweigung bedeutet ggf. den Wechsel einer Konfiguration.

10.2.7.4 Multicontext-UCB

Die allgemeine Lösung zur Verwaltung von mehreren, sequenziell auszuführenden Konfigurationen sowie von Verzweigungen lautet *Multicontext-UCB*. Dieser besteht aus einem ausführenden UCB sowie mehreren Speicherebenen, die ausführbare Konfigurationen enthalten können. Der Kontextswitch von einer Konfiguration zu einer anderen kann durch paralleles Laden im Rahmen eines Takts erfolgen, sofern die Konfigurationen vorgeladen sind.

Die Konfigurationssequenz sowie die Verzweigungen werden durch den Konfigurationsmanager verwaltet. Diese neue Einheit gehört insoweit zum UCB, wie dieser über mehrere Speicherebenen verfügt, ansonsten zur nachfolgend beschriebenen höheren Maschine.

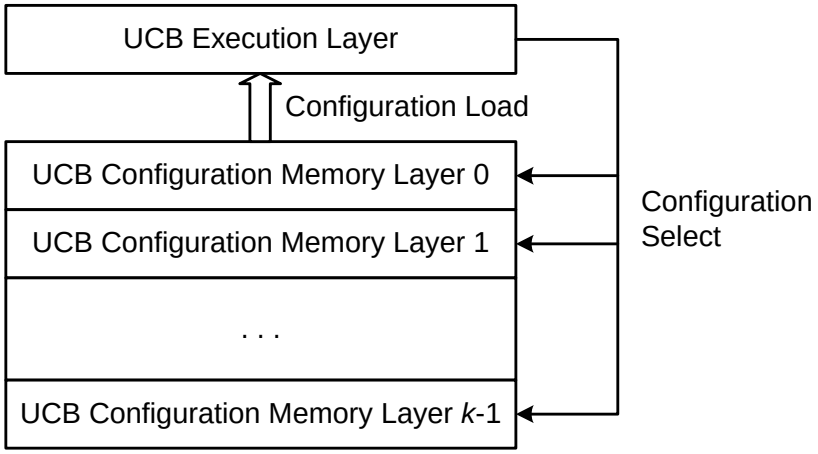


Bild 10.28 Multicontext-UCB

10.2.7.5 Universal Configurable Machine

Ein UCB muss mit fertigen Konfigurationen versehen werden, diese Übersetzung obliegt dem PDSP-Algorithmus. Allerdings treten eine Reihe von weiteren Aufgaben auf, die allesamt in der höheren Maschine, mit UCM (Universal Configurable Machine) bezeichnet, vereinigt sind:

- Während der PDSP-Algorithmus einen Instruktionsblock übersetzt, muss die Fetch-Einheit den Kontrollfluss innerhalb des Programms (bzw. Threads) weiter verfolgen. Diese Aufgabe zählt zur UCM.
- Bei mehreren UCB-Blöcken müssen diese mit verschiedenen Aufgaben versorgt und ggf. synchronisiert werden.

Bild 10.29 gibt einen Überblick zur Mikroarchitektur der UCMs. Hier sind mehrere Blöcke mit jeweils Signalisierungen und Kommunikationspfaden vorgesehen. Diese UCM-Architektur kann durch 4 Parameter, (b,h,s,c), mit b als die Anzahl der UCBs, h als der Hardware Scheduler, s als der Instruktionsblock Sequencer und c als Beschreibung der internen Kommunikation, beschrieben werden.

Die Fähigkeit einer UCM zur sequenziellen Ausführung von UCB-Konfigurationen (Hyperblocks) ist durch den Parameter s beschrieben. $s = 1$ bedeutet hierbei, dass der sequencer vorhanden ist, $s = 0$ entsprechend nicht. Die Fähigkeit zur sequenziellen Bearbeitung bedeutet, dass die UCM wie ein Mikroprozessor arbeiten kann.

Trotz der offensichtlichen Einschränkung für $s = 0$ sind UCMs mit dieser Eigenschaft durchaus sinnvoll, insbesondere in kleineren Applikationen. In diesen Applikationen, bei denen davon ausgegangen wird, dass nur eine kleine Anzahl von

Rekonfigurationen überhaupt notwendig sind, wird das Programm in dem Multicontext-UCB selbst gespeichert. Konsequenterweise besteht die kleinste UCM, eine (1,0,0,-)-UCM aus einem UCB mit Multicontext-Konfigurationsspeicher, einer PDSP-Übersetzungseinheit und einem Bootloader für den Start.

Der Parameter *h* ist wesentlich interessanter. *h* = 1 beschreibt die Fähigkeit der UCM, UCBs zu schedulen, ihnen also exklusive oder sequenziell fortschreitende Aufgaben zuzuordnen. Dies ist eine sehr wichtige Funktion, die in den Bereich der Betriebssysteme fällt: Das Scheduling von Ressourcen bedeutet, dass die UCM auf besondere Aufgaben – Interrupt Service beispielsweise – vorbereitet sein kann.

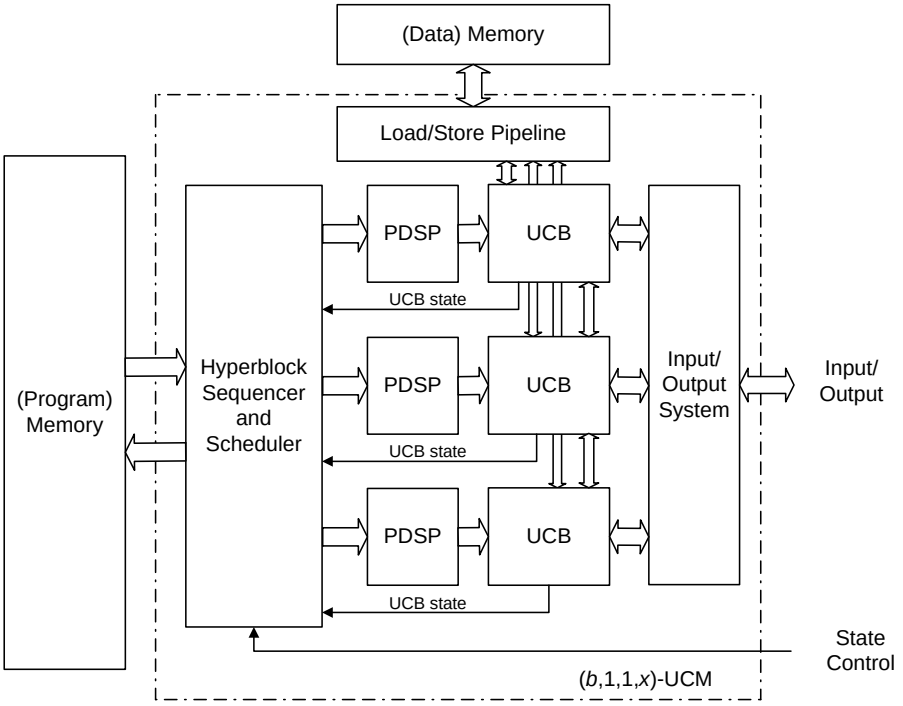


Bild 10.29 Universal Configurable Machine

Der Parameter *c* wurde zur Klassifizierung hinzugefügt, weil die Intra-Chip-Kommunikation gerade für Multiblock-Anwendungen mit zur Laufzeit veränderlichen Zuordnungen sehr wichtig ist. Dieser Parameter beschreibt Funktionalitäten: Die vordefinierte Funktion ‘lc’ (loosely coupled) beschreibt dabei jene Kopplung, in der Statusleitungen zwischen verschiedenen UCBs verfügbar sind, ein Datentransfer jedoch via Hauptspeicher ablaufen muss. Im Gegensatz dazu wird mit ‘tc’ (tightly coupled) die Kopplung benachbarter UCBs via Shared-Register beschrieben. Eng gekoppelte Systeme können gerade bei hohem Datenkommunikationsauf-

kommen sehr effektiv arbeiten. Weiterhin wird mit '-' keine Kommunikation verfügbar, mit 'x' eine nicht explizit definierte Kopplung beschrieben.

10.2.7.6 Konsequenzen für verschiedene UCM-Typen

Universal Configurable Machines mit $b > 1$ (multi-UCB) können sohol zur Erhöhung der Performance, zur Verbesserung (bzw. Herstellung) von Echtzeitfähigkeit sowie für zuverlässige Systeme genutzt werden. Die Performance kann dabei auf verschiedene Weisen unterstützt werden: Ein möglicher Weg besteht darin, dass verschiedene UCBs verschiedene Threads bzw. Prozesse ablaufen lassen können. Dies entspricht mehr dem CMP-Ansatz (Chip-Integrated Multiprocessing) für Mikroprozessorarchitekturen [11], während für SMT-Ansätze (Simultaneous Multithreading) [11] mehr Unterstützung innerhalb der UCBs notwendig sein wird.

Eine andere Möglichkeit zur Performanceverbesserung besteht darin, dass gänzlich andere, für bestimmte Einsatzgebiete sehr gut geeignete Architekturen auf die UCM abgebildet werden. Dies kann für den Xputer (→ 10.4) sehr einfach geschehen, dieser entspricht einer (b,0,0,tc)-UCM. Für Daten-intensive Applikationen etwa aus dem Bereich der Bildverarbeitung erhält man hier große Beschleunigungsfaktoren bis zu 100 und mehr, verglichen mit Mikroprozessor-basierten Lösungen.

Echtzeitapplikationen werden durch das Space/Time-Mapping unterstützt, die fundamentale Eigenschaft dieses Ansatzes. Hierdurch wird die sogenannte Eventdichte E, die die Reaktionsfähigkeit des Systems beschreibt, um Faktoren von ca. 100 bei Nutzung von (b,1,1,tc)-UCMs erhöht.

10.3 XPP-Architektur (Fa. PACT)

Die von PACT [15] vorgestellte Architektur wird mit XPP bezeichnet, was *eXtreme Processing Platform* steht. Diese Plattform ist ebenfalls als Coprozessor konzipiert, mit Hilfe eines RISC-Prozessorkerns, der auch auf dem Chip integriert sein kann (und wird), wird hieraus ein vollständiger Prozessor mit enormen Rechenkapazitäten.

Diese Rechnerplattform verlangt ein hohes Maß an Skalierbarkeit. Dies bezieht sich einerseits auf das Hardwaredesign, denn die Parallelität in der Ausführung ist nur durch Hardwareparallelitäten zu erreichen, also die Rückführung auf kleinere, reproduzierbare Einheiten. Skalierbarkeit bezieht sich andererseits auch auf die Softwareseite, denn die parallele Nutzung einiger Einheiten sollte keinesfalls die Nutzung weiterer verhindern. Die Hardware muss demnach auf die in der Softwarebeschreibung vorhandene Parallelität skaliert werden können, sowohl nach oben hin zu sehr großen Anwendungen als auch nach unten. Wie aus der Welt der Programmable Logic Devices bekannt ist, müssen die Hardwarearchitekten zudem einen Kompromiss zwischen Granularität und effektiver Rechengeschwin-

digkeit geben: Große Blöcke neigen zur Verschwendung von ungenutzten Ressourcen, sind aber schnell, kleine Blöcke lassen eine effektive Nutzung zu, auf Kosten der Rechengeschwindigkeit.

Die XPP-Architektur weist 4 Ebenen der Hierarchie auf, die der Hardware-Objekte, der Processing Array Elements (PAE), der Processing Array Cluster (PAC) und die der Devices selbst. Die Hardware-Objekte beinhalten elementare Operationen wie Verknüpfungen, Speicher oder Routing. Aus Sicht der Konfiguration sind dies die Ziele, mit denen die Laufzeit-Struktur des Bausteins festgelegt werden.

Der/die Softwareentwickler/in sieht hingegen die Processing Array Elements, die aus Objekten zusammengesetzt sind. Er/sie kann ein PAE für den Algorithmus nutzen, teilweise sogar die Objekte darin. Der Algorithmus wird aber nicht nur auf ein PAE, sondern auf den Cluster (PAC) oder sogar auf das gesamte Device abgebildet. Diese nächsthöheren Ebenen dienen damit u.a. der Chipverwaltung und der Skalierbarkeit, weniger der Programmierung.

10.3.1 Hardware-Objekte

Auf unterster Ebene existieren 9 verschiedene Hardware-Objekte, das ALU-Object, RAM-Object, Forward-Register-Object (FREG), Backward-Register-Object (BREG), das Programmable Gate Array Object (PGA), Data Channels, Event Channels, Network Switch Object und die I/O-Objects. Das ALU-Object (siehe Bild 10.30) besteht im Innern aus einem rekonfigurierbaren Zustandsautomaten, bei dem in den Eingangsregistern Daten und Events gespeichert werden. Die Art der Verarbeitung ist dann abhängig von der Konfiguration, denn die ALU wird auf die üblichen Verknüpfungen wie Addition, Multiplikation etc. eingestellt. Nicht nur das Ergebnis, sondern auch Events für nachfolgende Einheiten sind im Ausgangsregister speicherbar.

Die Eingangsdaten und –events stammen aus dem horizontalen Routing, die Ausgänge werden wieder in dieses Routing geführt. Je Richtung sind hierfür 6 Leitungssysteme (32 Bit für Daten) vorgesehen. Die ALU kann dabei maximal 3 Eingangsbusse zu 2 Ausgängen verarbeiten, beispielsweise in Form der MAC-Operation, die 32 * 32 Bit multipliziert und zu einem 32 Bit Wert addiert, die Ausgangsbreite beträgt dann 64 Bit.

Forward- und Backward-Register Object dienen dem vertikalen Routing im Rahmen einer höheren Einheit, dem Processing Array Cluster (PAC, siehe Bild 17). Während zwischen den einzelnen PAEs, die direkt aus Objekten zusammengesetzt sind, ein (segmentierbares) horizontales Routing besteht, wird das vertikale Routing konfiguriert. FREG und BREG können Daten speichern, BREG kann auch direkt verbinden und Berechnungen durchführen. So existiert hier eine vereinfachte ALU (Addition, Subtraktion) und eine Look-Up Table (4 Eingänge) für logische Verknüpfungen der Eventleitungen [17] [22].

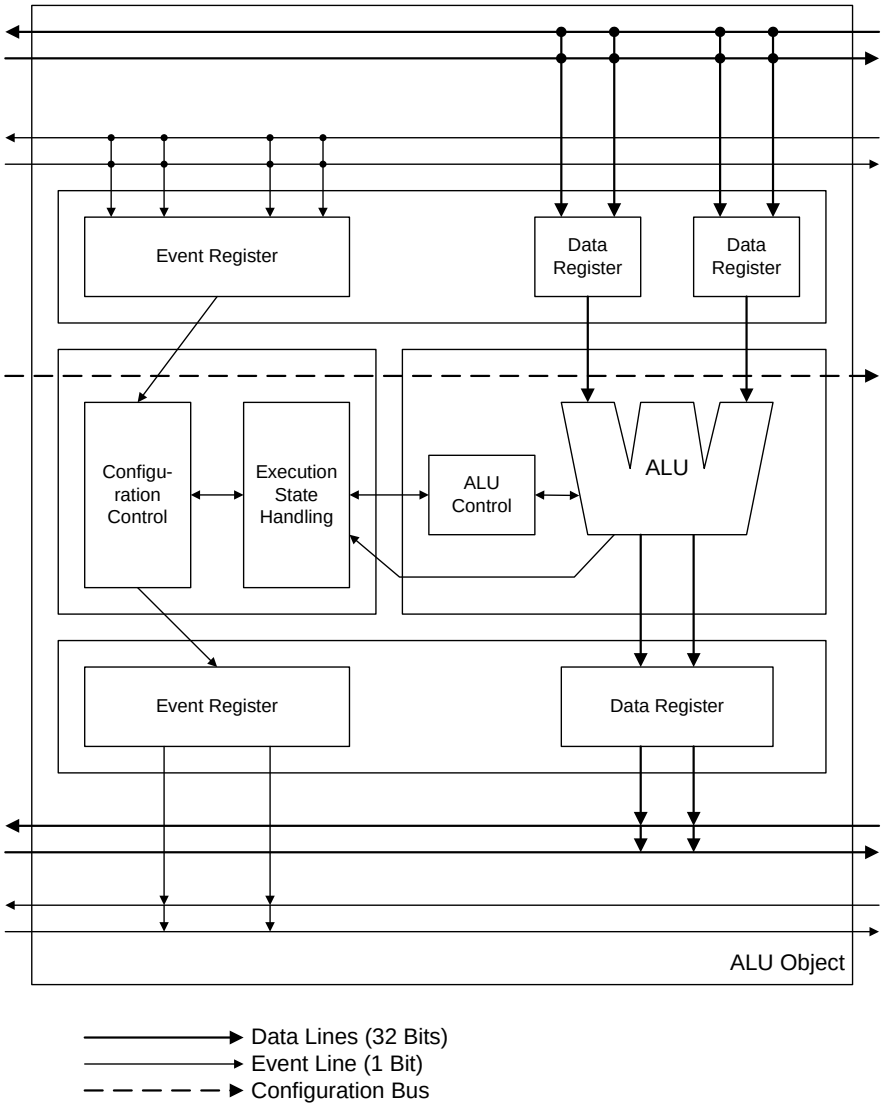


Bild 10.30: ALU-Object in XPP-Architektur

Das RAM-Object enthält lokales SRAM für Look-Up Table innerhalb von Algorithmen und zur Speicherung von Ergebnissen. Die Speichertiefe einer derartigen RAM-Zelle ist naturgemäß begrenzt, aktuell werden pro Zelle 256 Worte (á 32 Bit) gespeichert. Dies bietet für viele Algorithmen gerade aus der Signalverarbeitung (denken Sie an Filter- und Transformationsalgorithmen) genügend Raum, um Koeffizienten und andere Konstanten nahe an den Rechenwerken zu lagern.

Reicht der Speicherplatz nicht aus, bietet sich die Möglichkeiten, über die I/O-Objekte externes SRAM mit nahezu beliebiger Größe anzuschließen. Diese I/O-Objekte sind den PACs zugeordnet, in der aktuellen Konfiguration der XPU128-ES hat jedes 2 Ports á 32 Bit. Ein derartiges I/O-Object kann in einem speziellen RAM-Mode betrieben werden, sodass Adress- Steuer- und Datenleitungen spezifisch zugewiesen sind.

Interessant an den aufgezählten Hardware-Objekten ist, dass auch FPGA-ähnliche Strukturen in der XPP-Architektur existieren. Diese PGAs waren in den ursprünglichen Darstellungen noch nicht vorhanden, sind nunmehr jedoch vorgesehen, da sich Kontrollstrukturen, insbesondere auf Bitebene, besser so implementieren lassen.

Das letzte der aufgezählten Hardwareobjekte ist das Network Switch Object (Switch-Object). Zwei Arten von Kommunikation sind in der XPP-Architektur zu finden: Daten und Ereignisse. Beide werden über horizontale Signalleitungen, die stückweise in Teileinheiten der XPP-Architektur vorhanden sind, verteilt. Zur Konfiguration eines Netzwerks zwecks Kommunikation sind dann die Switch-Objects notwendig, die ihrerseits die Leitungsstücke miteinander verbinden bzw. voneinander trennen. Man kann es also so formulieren: Das horizontale Routing ist segmentierbar, das vertikale Routing über FREG und BREG einrichtbar.

10.3.2 PAEs und PACs

Bild 10.31 zeigt die nächste Hierarchieebene in der XPP-Architektur, das Processing Array Element (PAE). Dieses Element ist aus den Hardwareobjekten sowie Signalleitungen zusammengesetzt, und es existieren zwei wichtige Typen: ALU-PAE und RAM-PAE. Beide PAE-Typen sind nahezu identisch aufgebaut, der Unterschied liegt in dem RAM- bzw. ALU-Object.

Die PAEs besitzen drei wesentliche Kommunikationssysteme: Datenleitungen in der generischen Breite (32 Bit), Eventleitungen mit 1 Bit Breite und den Konfigurationsbus. Die PAEs stellen damit das operative Herzstück der XPP-Architektur dar, denn hier wird die Operation, das lokale Speichern und das Datenrouting konfiguriert, und die Events steuern den Datenfluss innerhalb des gesamten Rechners.

Diese Eventsteuerung ist ausgesprochen wichtig, denn die Gültigkeit der Operationen ist von der Verfügbarkeit der Daten abhängig. Die Events steuern den Datenfluss, indem z.B. ALU-Operationen gestattet oder unterdrückt werden. Hierzu sind die Eventleitungen RESET, STEP, STOP, GO und RELOAD an jeder ALU definiert, die entsprechende Operationen auslösen, während die ALU selbst die Events U, V und W (abhängig von der definierten Operation) erzeugt. Die generierten Events können den klassischen Flags entsprechen, sie können auch (im Rahmen) selbst definiert werden.

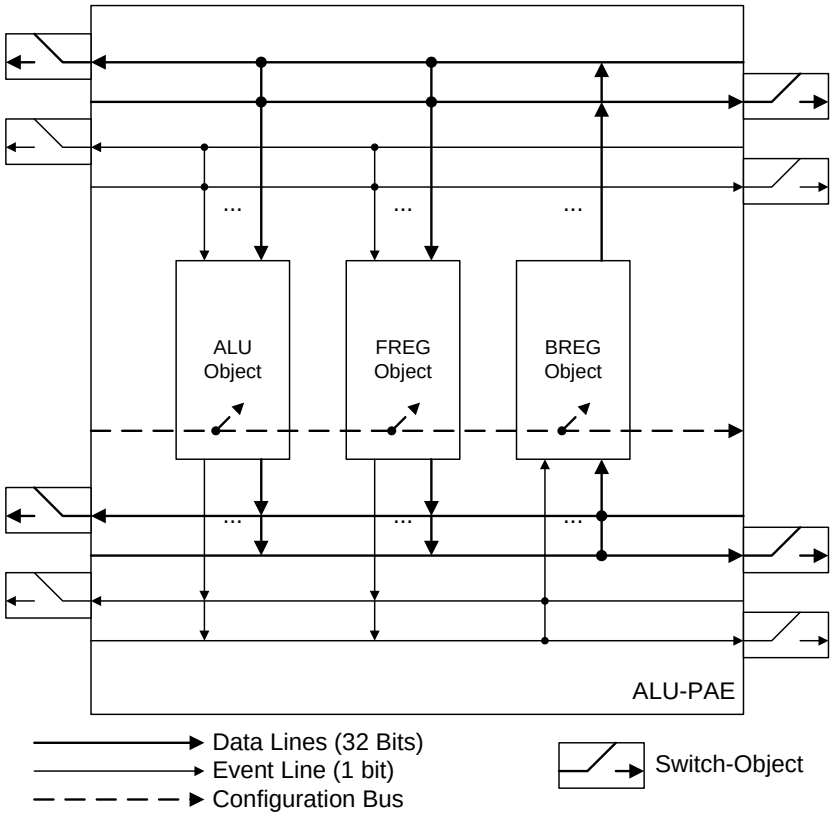


Bild 10.31: Processing Array Element (ALU-Typ) mit Switch Objects

Einige von diesen PAEs sind zusammen mit I/O-Objekten, Connect-PAEs (CON-PAE) zur Verbindung mit anderen Clustern und diversen Switch-Objekten zu einem Processing Array Cluster (PAC, siehe Bild 10.32) zusammengefasst. Beim verfügbaren Referenzchip XPU128-ES sind dies 64 ALU-PAEs und 16 RAM-PAEs (aber kein PGA-PAE), und zu jedem PAC gehören 4 I/O-Objekte mit je 2 Ports á 32 Bit. Bei anderen Devices kann die Zusammensetzung eines PACs durchaus variieren Jeder PAC ist mit einem Configuration Manager (CM) gekoppelt, dessen Aufgabe in der dynamische (Re-)Konfiguration besteht.

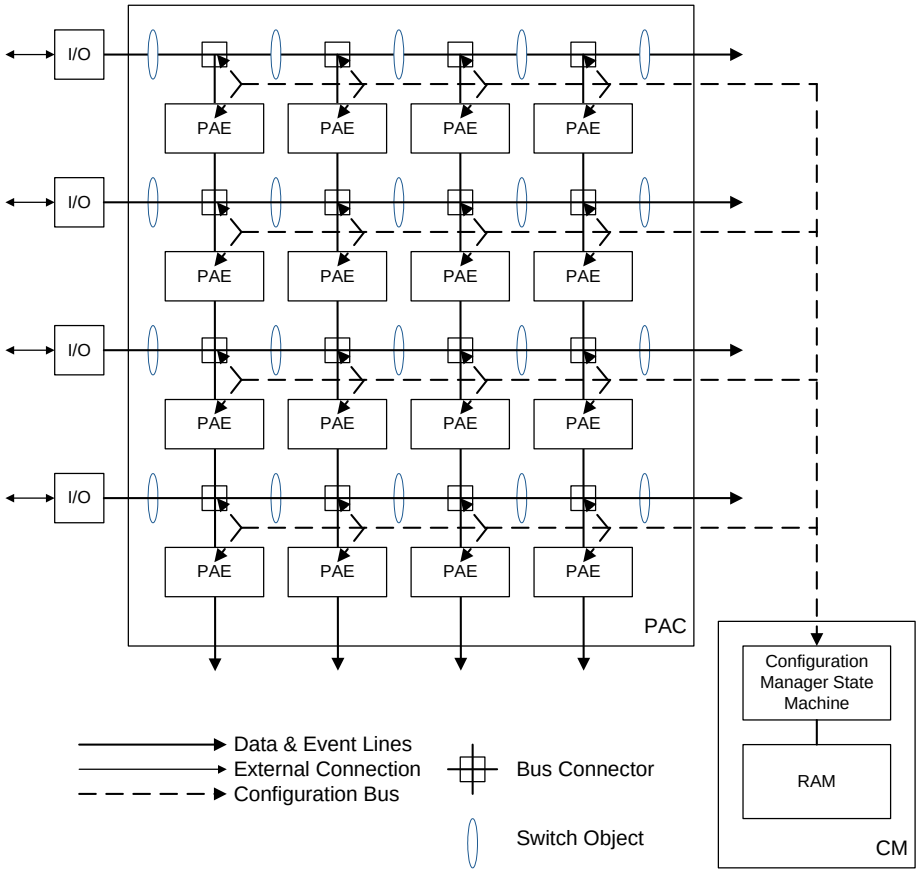


Bild 10.32: Aufbau Processing Array Cluster (PAC) der XPP-Architektur (ohne CON-PAE)

In einem XPP-Device sind dann mehrere PACs integriert (Bild 10.33), in der aktuellen XPU128-ES sind es 2 PACs, bei der XPU192 4. Die CMs selbst werden hierbei mit einer supervising CM (SCM) verbunden, um den Programm- und Konfigurationslauf im gesamten XPP-Device steuern zu können. Es besteht auch die Möglichkeit, mehrere XPP-Chips über die I/O-Objekte miteinander zu koppeln. In diesem Fall muss eine externe SCM definiert werden, damit auch die Kopplung wie ein ausführendes Device erscheint.

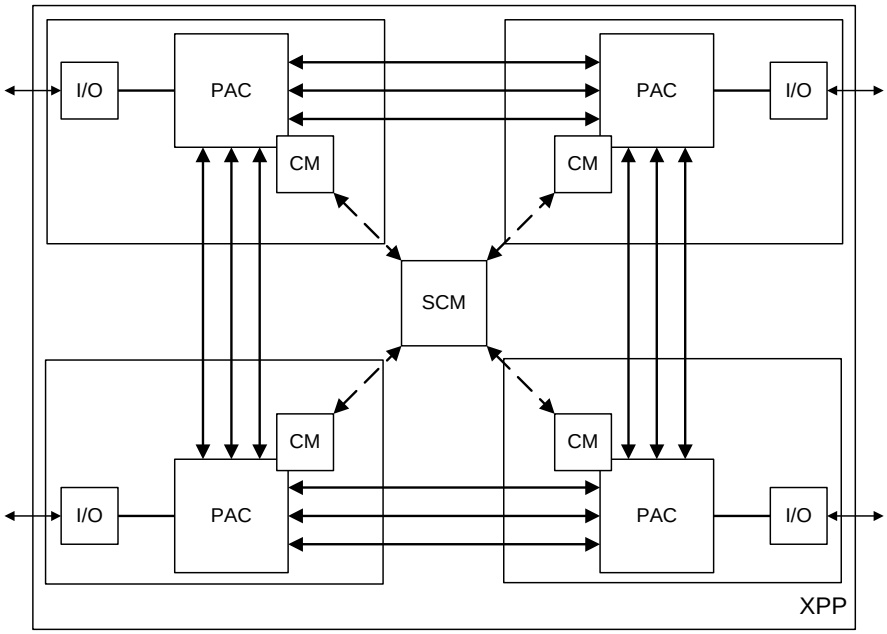


Bild 10.33: XPP-Architektur, bestehend aus mehreren PACs und CMs mit Supervising Configuration Manager

Mit anderen Worten: Die XPP-Architektur ist auf Skalierbarkeit ausgerichtet. Die Kombination Processing Array Cluster/Configuration Manager (PAC/CM) ist die Einheit, die ein Programm ausführen kann, auf die also ein Programm auch compiliert werden muss. Sind mehrere dieser Einheiten auf einem Chip oder Chip-übergreifend vorhanden, dann wird – entsprechende Parallelität im Programm vorausgesetzt – das Gesamtprogramm auf diese Einheiten verteilt und entsprechend schneller ausgeführt. Diese Form der Threadparallelität skaliert vergleichsweise linear mit der Anzahl der ausführenden Einheiten, was man von Instruktionsparallelität nicht behaupten kann.

10.3.3 Programmausführung

Ein für ein XPP-Device geschriebenes Programm muss in einen Datenstrom mit zugehörigen Ereignissen, in ein Array von Operationen und in einen Konfigurationsstrom übersetzt werden. Der Konfigurationsstrom wird notwendig, wenn man nicht nur eine einzige Konfiguration in ein PAC laden möchte, sondern dynamisch rekonfigurieren will. In diesem Fall muss die Rekonfiguration natürlich mit dem Datenstrom synchronisiert werden, und dieser Synchronisationsmechanismus ist in der Hardware vorhanden, nur der Compiler hat wieder zusätzlich die Aufgabe, die Events dafür zu erzeugen.

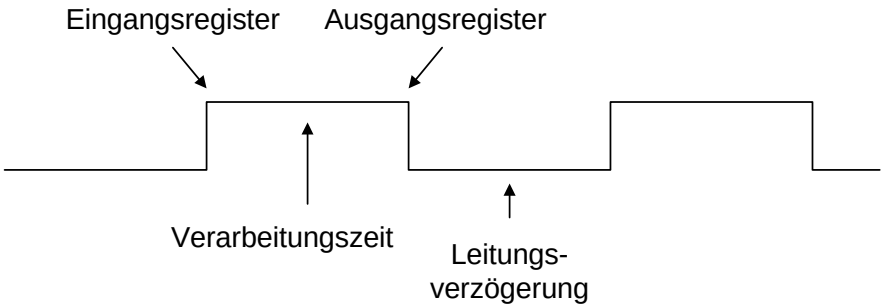


Bild 10.34: Ablaufphasen in XPP-Architektur

Der eigentliche Operationsablauf in einem PAC bzw. einer PAE ist natürlich getaktet (Bild 10.34). Während eines Takts werden die Eingangsdaten gespeichert (positive Taktflanke), in der PAE verarbeitet (1. Taktphase), und am Ausgangsregister gespeichert (negative Taktflanke). Die verbleibende 2. Taktphase dient der Leitungsverzögerung zum Verteilen der entstandenen Daten via interner Verbindungen. Bei 100 MHz Systemtakt bleiben pro Taktphase gerade einmal 5 ns.

Instruktions- bzw. Operationsparallelitäten sind in der Ausführung in verschiedenen PAEs möglich, nicht innerhalb einer PAE selbst, sieht man einmal von dem Routing in FREG/BREG mit der Speicherung bzw. den vorhandenen Berechnungsmöglichkeiten einmal ab..

10.3.4 Programmentwicklung

Die wohl wichtigste Frage, die noch offen ist, ist die nach der Programmentwicklung. Die ausführende Hardware wird optimal durch kooperierende Threads ausgenutzt, also Programmfäden, die eine Weile unabhängig voneinander rechnen, um sich dann zu synchronisieren bzw. Daten auszutauschen. Hier wird das Datenflussprinzip deutlich, wobei es auf Threadlevel sozusagen grobkörniger und innerhalb der Threads feinkörniger angewendet wird.

Entwickler mit Erfahrung in der Hardware erkennen auch Parallelitäten hierzu. Bis auf einfache Beispiele läuft ein Hardwaredesign nämlich als System kooperierender Automaten ab, wobei jeder Automat (Finite State Machine, FSM) für sich parallel zu allen anderen arbeitet, nach jedem Zyklus jedoch durch Signale den Zustand der Kooperationspartner erfährt.

Die Assemblersprache der XPP-Architektur ist dann auch Hardware-ähnlicher, als man dies von einem Prozessor erwarten sollte. Sie wird Native Mapping Language (NML) genannt, und man beschreibt darin für jede PAE, einzeln für jedes ALU-, FREG und BREG-Objekt die Funktion und die Belegung der Eingänge. Das Codebeispiel in Bild 10.35 entspricht der Bestimmung der Determinanten einer 2*2-Matrix:

```
// Beispiel für Determinantenberechnung
//
//      det = a*d - b*c
//
XPU (V1, 1,1, 8,8, 6,6)          // XPU Größendefinition

In_stream_a IN0 @ $0,$1          // Eingangsdaten
In_stream_b IN0 @ $0,$1
In_stream_c IN0 @ $11,$1
In_stream_d IN0 @ $11,$1


F2_1a FREG @ $2, $1              // Zwischenspeicherung
  D = In_stream_a.DOUT

F2_1b FREG @ $2, $1
  D = In_stream_b.DOUT

F3_1c FREG @ $3, $1
  D = In_stream_c.DOUT

F3_1d FREG @ $3, $1
  D = In_stream_d.DOUT


// Jetzt die eigentliche Arithmetik:

P3_1 MULADD @ $2,$2
  A = F2_1a.Q                     // a * d
  B = F3_1d.Q
  C =! 0

P3_2 MULADD @ $3, $2
  A = f2_1b.Q                     // b * c
  B = f3_1c.Q
  C =! 0

B3_2 DBREG_SUB @ $3, $2           // a * d - b * c
  NOREG(D)
  D1 = P3_1.Q
  D0 = P3_2.Q                     // Am Ausgang von BREG: det
```

Bild 10.35: Sourcecodebeispiel in NML

Dieses Beispiel zeigt die Berechnung det = a*d – b*c für vier Datenströme, die von außen kommen. Die eingehenden Werte für a, b, c und d werden zwischengespeichert und in zwei ALUs, die als MULADD konfiguriert sind, miteinander verknüpft. Die anschließende Subtraktion erfolgt im Backward-Registerteil (B3_2) und benötigt keine zusätzliche ALU, sodass die Berechnung mit 2 ALUs auskommt. Die Beschreibung besteht darin, die einzelnen PAEs und I/O-Objects gezielt zu adressieren (es wird der Ort der PAE angegeben, in dem die Operation stattfinden soll), die Eingänge zuzuweisen und die Art der Operation auszuwählen. Die Ortszuweisung und die gesamte Kommunikation machen es zwingend notwendig, dass die Applikation von Softwaredesigner partitioniert, abgebildet und (innerhalb des PACs) gescheduled wird!

Die Partitionierung bedeutet hierbei, dass eine Aufteilung des Algorithmus in parallele und sequenzielle Teile erfolgen muss. Diese Partitionierung hat entscheidenden Einfluss auf die spezifische Rechengeschwindigkeit. Die Sequenzialität, die in den Berechnungsvorschriften steckt, wird in die Reihenfolge der durchzuführenden Operationen verlagert, und dieses Scheduling ist ebenfalls Aufgabe des Entwicklers, wenn er NML nutzt.

10.3.5 Dynamische Rekonfiguration

Das Beispiel in Bild 10.35 zeigt nur den statischen Teil der Sprache, der die Konfiguration von Modulen ermöglicht. Der besondere Charme des Reconfigurable Computing besteht jedoch darin, weder viele fixierte Instruktionen (sequenzielles Rechnen bei Prozessoren) oder eine Konfiguration zu verwenden, sondern eine freie Folge von Konfigurationen.

Für ein dynamisches Rekonfigurieren müssen Processing Array Cluster (PAC) und Configuration Manager (CM) über ein programmierbares Interface verfügen. Hier kommen nun die Events (→ Bild 10.31) ins Spiel, denn genau sie können das Ende von Operationen anzeigen. Die Events werden gemäß Programmierinformationen erzeugt und können den Verlauf der Rechnungen steuern, beispielsweise einen konfigurierten Zähler starten oder die Erzeugung von Datenpaketen anstoßen.

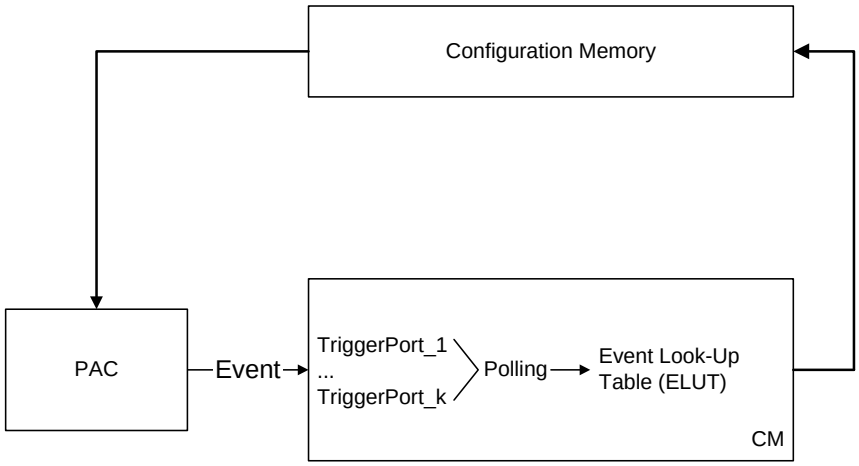


Bild 10.36: Interface PAC und CM für dynamische Rekonfiguration

Diese Steuerbarkeit innerhalb einer Konfiguration wird durch die Steuerbarkeit von Konfigurationswechseln ergänzt. In dem zu einem PAC zugeordneten Configuration Manager werden alle definierten Triggerports per Polling (der CM ist auf Prozessorbasis programmiert) abgefragt. Tritt ein Event auf, dann wird anhand der Event Look-Up Table (ELUT) die zugehörige ID der Konfiguration

gewonnen und aus dem Configuration Memory in den PAC geladen. Wichtig ist dabei, dass das Eintreffen des Events den Zeitpunkt bestimmt, und dass über den Wert des Events (0 oder 1) die aktuelle Konfiguration wählbar ist. Konstrukte wie

```
IF(this_event == 0) config1; ELSE config2;
```

sind damit möglich, sodass eine komplette Steuerung des Programmflusses auch auf Konfigurationsebene möglich ist. Kombinationen der Events mit AND bzw. OR (nicht gemischt) betreffen das zeitliche Eintreffen dieser: Bei der AND-Verknüpfung müssen alle Events eingegangen sein, bei der OR-Verknüpfung genügt eines. Die Rekonfigurationsbefehle in der NML enthalten ferner Möglichkeiten, diese direkt durchzuführen, eine Auswahl wie im obigen Pseudocode einzufügen oder die Identifikationsnummer der Konfiguration im PAC selbst auszurechnen (indirekte Adressierung).

Nutzt ein Softwareentwickler dies, dann muss er/sie die Lage der einzelnen PAEs innerhalb des PAC immer noch explizit angeben. Jedes Teilprogramm, nennen wir es wieder Thread, wird wie oben angegeben. Ist in diesen Teilprogrammen ein Endkriterium enthalten, beispielsweise ein Zähler von 0 bis 255 oder eine andere Bedingung, kann dies als Triggerport für das nächste Teilprogramm genutzt werden. Das in Bild 10.35 gezeigte Modul Polynom stellt einen solchen Thread dar, allerdings ohne explizites Endkriterium.

In einem zweiten Programm beschreibt man dann die Gesamtapplikation, d.h. die Reihenfolge der Module, die Bedingungen zur Rekonfigurierung usw. Die Datenkommunikation obliegt dabei dem/der EntwicklerIn, denn die Daten, beispielsweise in den lokalen RAMs, bleiben bei Rekonfiguration erhalten (mit Ausnahme der Programmkonstanten).

Diese NML dürfte für Spezialprogramme, etwa Bibliotheksfunktionen, sehr geeignet sein, da sie die Fähigkeiten der PACT-Architektur bestmöglich ausnutzt. Genau genommen beschreibt man ein rekonfigurierbares Programm mit Hilfe von Threads und einem kooperativen Multithreading, also auch ein kleines Stückchen Betriebssystem.

Für einen Einsatz in der Breite ist NML sicher nicht geeignet (und wohl auch nicht vorgesehen). Eine derartige programmierbare Struktur, wie es die XPP-Architektur darstellt, ist bestens geeignet für Filteralgorithmen, Bildverarbeitung, als Grafikprozessor etc., und die Sprache der Wahl dürfte wohl C sein.

Ein C-Compiler ist erhältlich, liefert allerdings noch nicht die Codequalität eines NML-Programmierers. Im Designflow wird die Rolle der XPU als Coprozessor deutlich. Ähnlich zu dem Ablauf beim Xputer-C (→ 10.4) werden Host- und XPU-Teil auf Sourcecodeebene getrennt und mit Interfaceroutinen ausgestattet. Der C-Compiler wird es dann in die NML übersetzen, und dann laufen die zeitaufwendigen Teile in der XPU.

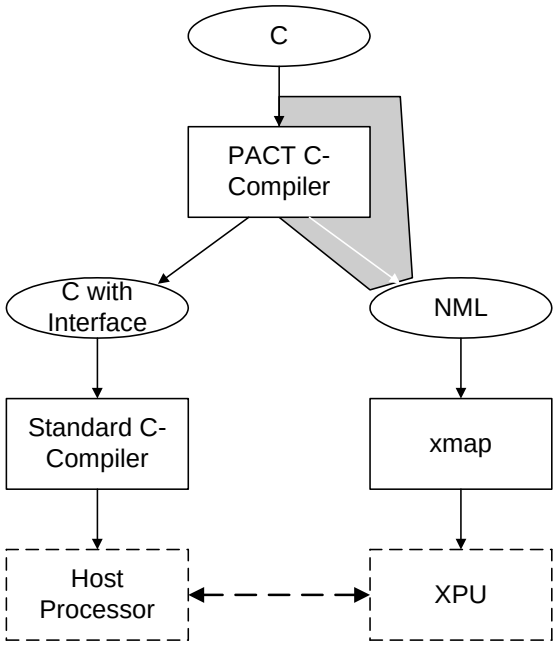


Bild 10.37: XPU C-Compiler Designfluss

Der umrandete Teil ist dabei der kritische Abschnitt in diesem Designfluss. Hier werden Partitionierung und Scheduling von dem Compiler verlangt, sogar in mehrfacher Weise. Es gilt, aus dem sequenziellen C-Code, der bereits in Host- und XPU-Teil partitioniert und mit Kommunikationsroutinen versehen ist, die parallelen Teile herauszufinden und das Scheduling zu bestimmen. Noch schwieriger wird der Weg sein, aus C-Code eine Folge von Rekonfigurationen generieren zu wollen, denn dies bedeutet zwei Freiheitsgrade: Größe/Inhalt der Konfigurationen versus Rekonfigurationssequenz. Hier ein Optimum zu finden, heißt, sich noch weiter auf Neuland begeben zu müssen.

10.4 Der Xputer (Univ. Kaiserslautern)

Der Xputer versucht einen neuen Ansatz: Ausgehend von dem in einer (Software-) Hochsprache formulierten Ansatz (bisher in C) wird dies nicht in ein Assemblerprogramm übersetzt, sondern dieses Programm wird auf ständig wiederkehrende Teile analysiert, die dann in reprogrammierbarer Hardware weitestgehend programmiert sehr schnell ausgeführt werden können.

Die bisherigen Implementierungen des Xputer sind dementsprechend als Coprozessor ausgeführt. Dieser Coprozessoransatz, als Accelerator proklamiert, bietet

sich natürlich an, da nur Teile des Gesamtprogramms derartig analysiert werden können. Die Geschwindigkitsgwinne erreichen mit diesem Verfahren durchaus Faktoren bis 100.

Der Xputer kann allerdings nur dann akzeptiert werden, wenn die Generierung des Coprozessors automatisch erfolgt. Jede 'hand-made' Definition eines Prozessors wird garantiert zu aufwendig und eingeschränkt sein, um außer in Spezialfällen eine echte Alternative darstellen zu können.

Im ersten Teil dieses Unterkapitels wird der Xputer im Detail eingeführt; der zweite Teil geht dann auf die Verfahren und Tools ein, die eine automatische Generierung des Coprozessors durchzuführen.

10.4.1 Detailbeschreibung des Xputer

Die grundlegende Architektur des Xputer umfasst drei Teile:

- Das Data Memory
- Die rekonfigurierbare Arithmetisch-Logische Unit (rALU)
- Der Data Sequencer zum Datenzugriff, der mit Hilfe mehrerer Generic Address Generators (GAGs) den Datenzugriff gestattet.

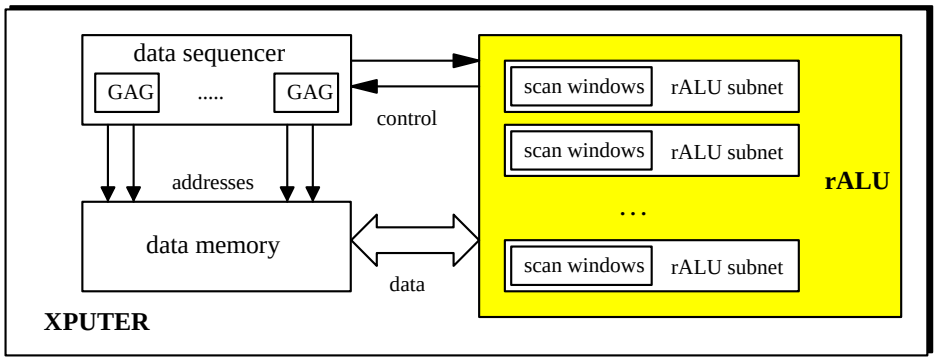


Bild 10.38: Architektur des Xputer

Aus diesen Bestandteile wird bereits ersichtlich, dass der Xputer keine einfache Prozessorarchitektur ist, sondern vielmehr eine auf Datenfluss optimierte Gesamtrechnerarchitektur darstellt. Zu den einzelnen Bestandteilen können folgende Kurzbeschreibungen gegeben werden:

Das Data Memory ist zunächst zweidimensional (!) angeordnet; höhere Dimensionalität kann natürlich (wie im Fall des normalerweise eindimensionalen Speichers) hineininterpretiert werden. Innerhalb dieses Speichers werden alle Daten gelagert, die während einer Applikation benötigt oder verändert werden.

Die Mehrdimensionalität hat einen besonderen Sinn: Die Applikationsdaten werden in diesem Speicher in besonderer (applikationsspezifischer) Weise (*Data Map* genannt) untergebracht, um den Datenzugriff zu optimieren.

Zu diesem Optimierungszweck sind die Scan Windows, die das Dateninterface zwischen rALU und Data Memory darstellen, in ihrer Größe (Runtime-)skalierbar: Ein Datenzugriff wird nicht nur auf das zentrale Datum, sondern auch die benachbarten Datenzellen erfolgen.

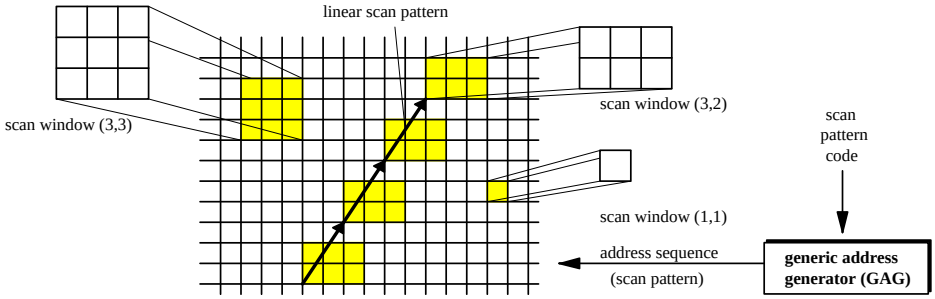
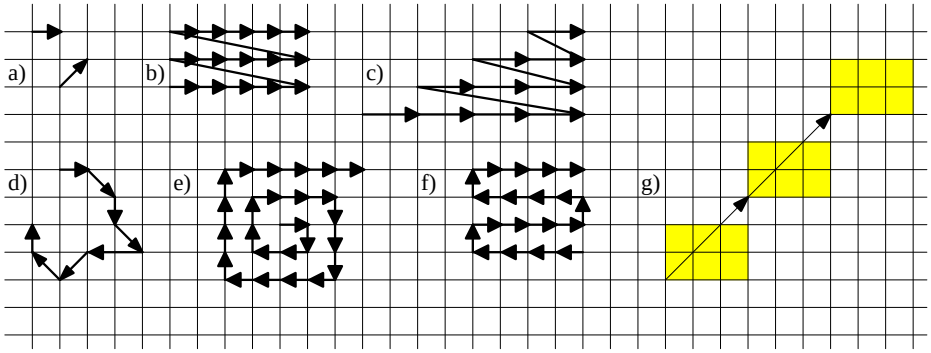


Bild 10.39: Lineare Adressierungssequenz sowie verschiedene Scan Window Größen

Die rALU, das rechnerische Herz des Xputer, kann komplexe Operationen mit vielen In- und Outputs enthalten. Diese Operationen sind Applikations-spezifisch (*Compound Operator*) und können daher sehr optimiert angewendet werden, wenngleich das Problem der Generierung dieser Operationen sichtbar ist. Die rALU wird in diverse Subnetze eingeteilt, um verschiedene Applikation parallel zueinander zu unterstützen. Die rALU wird in Feldprogrammierbaren Gate Arrays (FPGAs) untergebracht, wodurch sich technisch die leichte Umprogrammierbarkeit erklären lässt.

Die Adressierung der Datenmengen erfolgt durch die Generic Address Generators (GAGs), deren Anwendung auf folgenden Überlegungen basiert: Viele Algorithmen zeigen ein immer wiederkehrendes Verhalten beim Datenzugriff. Genau dieses Verhalten wird den GAGs als Parameter übergeben, die dann eine (festverdrahtete) Hardwaresequenz gemäß diesem Parameter zum Datenzugriff ablaufen lassen. Bild 10.40 zeigt einige Beispiele für diese Zugriffssequenzen.



a) single step
b) + c) video scan
d) user-defined
e) spiral scan
f) zig-zag-scan
g) linear scan with scan window

Bild 10.40: Basiszugriffsformen zum Datenzugriff

Das Zusammenführen von festverdrahteten Zugriffsschemen und programmierbaren Zugriffsfenster gewährleistet dann einen Datenzugriff ohne größeren Berechnungsoverhead. In der Praxis besteht ein GAG aus zwei Teilen, wobei in dem ersten (Handle Address Generator) die logische Adresse und im zweiten (Memory Address Generator) die physikalische Adresse berechnet wird.

10.4.2 Der X-C-Compiler

Während das Hardwarekonzept des Xputer sicher revolutionär ist, um drastische Steigerungen der Performance zu erreichen, muss beim Softwarekonzept eines solchen Rechnertyps natürlich auf eine gewisse Kompatibilität zum bisherigen Standard geachtet werden. Aus diesem Grund wird ein geringfügig eingeschränktes C (Xputer-C) eingeführt, bei dem Methoden zur dynamischen Speicherverwaltung mit Pointern, Systemcalls und rekursiver Programmierung ausgeschlossen sind.

Input/Output-Bedienung und die dynamische Speicherverwaltung werden im Xputerkonzept in den Hostprozessor verlagert (Xputer ist ein Coprozessorkonzept!). Unter diesen Randbedingungen werden X-C-Programme zunächst einer Kontrollfluss- und einer Datenstrukturanalyse unterzogen. Die Datenstruktur dient dabei einer weitergehenden Parallelisierung (entsprechend Verfahren für Vektor- und Parallelprozessoren) sowie der Aufteilung auf die zweidimensionale Speichermatrix.

Aus diesen Dateninformationen werden weiterhin die Größe der Scan Windows und die (Haupt-) Bewegungsrichtung abgeleitet, während die Kontrollflussinformationen in eine Hardwareprogrammierung für die rALUs umgewandelt werden. Diese Informationen enthalten dann in codierter Form die gewonnenen Operanden mit allen Zugriffen.

Weiterhin werden Parametersets für die Scanpattern generiert; der Kontrollfluss wird also quasi in eine Reihe von Zugriffe mit automatisch verschalteter Verknüpfung umgesetzt, soweit dies möglich ist.

10.4.3 Kurze Bewertung des Xputer-Konzepts

Mit der Einschränkung der nicht universellen Verwendbarkeit des Xputers – das Coprozessorkonzept kann nur auf Tasks bzw. Threads angewendet werden, die nicht einem Umschaltprozess unterliegen – stellt dieses Konzept eine hochinteressante, revolutionäre Neuerung dar. Datenfluss-, ASP- (Application Specific Processors) und superskalare Konzepte fließen hier derart ineinander, dass der Performancegewinn mit einem Faktor von 83, gemessen im Vergleich zu Workstations mit superskalaren Prozessoren, nicht verwundern muss.

Es sollte hierbei auch nicht übersehen werden, dass der Xputer per (eingeschränktem) C programmiert wird und somit dem ‘normalen’ Programmierer zugänglich ist. Die Generierung der Adressierungsinformationen, der Superbefehle und der Datenlagerung erfolgt ohne Nutzereingaben oder besondere C-Pseudobefehle.

Das rALU-Feld kann prinzipiell auch aus dem Xputer herausgelöst und singulär verwendet werden. Hierdurch erhält man eine zur XPP-Architektur ähnliche Struktur (genannt ’Kress-Array’). Ein solches Array könnte applikationsspezifisch entwickelt und integriert werden – ein Vorgehen, das auch in der XPP-Architektur durchgeführt wird.

Literaturverzeichnis

[1] *Hennessy, J. L., Patterson, D. A.*: Computer Architecture: A Quantitative Approach. – Second Edition – San Francisco: Morgan Kaufmann Publishers, 1996

[2] *Hennessy, J. L, Patterson, D. A.*: Computer Organization & Design: The Hardware/Software Interface. – Second Edition – San Francisco: Morgan Kaufmann Publishers, 1997

[3] *Christian Siemers*: Prozessorbau. – Carl Hanser Verlag München Wien, 1999

[4] *Oberschelp, W.; Vossen, G.*: Rechneraufbau und Rechnerstrukturen. – 8. Auflage. – München, Wien: R. Oldenbourg-Verlag, 2000

[5] *Flik, T.; Liebig, H.*: Mikroprozessortechnik. – 6. Auflage. – Berlin Heidelberg, New York: Springer-Verlag, 2001

[6] *Giloi, W.*: Rechnerarchitektur. Springer-Verlag, Berlin, Heidelberg, New York, 1981. ISBN 3-540-10352-X.

[7] *Beierlein, Th., Hagenbruch, O. (Hrsg.)*: Taschenbuch Mikroprozessortechnik. 2.Auflage – Fachbuchverlag Leipzig im Carl Hanser Verlag München Wien, 2001

[8] *Schiffmann, W.; Schmitz, R.; Weiland, J.*: Technische Informatik 1, Grundlagen der digitalen Elektronik. – 2. Auflage – Berlin, Heidelberg, New York: Springer Verlag, 2001

[9] *Schiffmann, W.; Schmitz, R. ; Weiland, J.*: Technische Informatik 2, Grundlagen der Computertechnik. – 2. Auflage – Berlin, Heidelberg, New York: Springer Verlag, 2001

[10] *Märtin, C. (Hrsg.)*: Rechnerarchitekturen – CPUs, Systeme, Software-Schnittstellen. – 2. Auflage – München, Wien: Carl Hanser Verlag, 2000.

[11] *Šilc, J.; Robic, B.; Ungerer, T.*: Processor Architecture. – Berlin, Heidelberg, New York: Springer, 1999.

[12] *André Dehon, John Wawrzynek,* ”Reconfigurable Computing: What, Why and Implications for Design Automation”. Design Automation Conference DAC99, San Francisco, 1999.

[13] *Wen-Mei W. Hwu et. al.*: Compiler Technology for Future Microprocessors. Invited Paper in Proceedings of the IEEE, Special Issue on Microprocessors, Vol. 83 (12), p. 1625 .. 1640, 1995.

[14] *Sascha Wennekers, Christian Siemers,* ”Reconfigurable RISC – a New Approach for Space-efficient Superscalar Microprocessor Architecture” in: Proceedings of the International Conference on Architecture of Computing Systems ARCS 2002, Karlsruhe, Germany, April 2002. Springer Lecture Notes in Computer Science 2299, pp. 165–178.

[15] http://www.pactcorp.com/

[16] *John Crawford*, "Introducing the Itanium Processors". *IEEE Micro* Vol. 20(5), pp. 9–11, 2000.

[17] *Andreas Stiller*, "Architektur für echte Programmierer: IA-64, EPIC und Itanium". *ct* 2001, H. 13, S. 148–153.

[18] *Uwe Schneider, Dieter Werner*: Taschenbuch der Informatik. 4.Auflage – Fachbuchverlag Leipzig im Carl Hanser Verlag München Wien, 2001

[19] *Sigmund, U.; Ungerer, T.*: Ein mehrfädiger, superskalarer Mikroprozessor. – PARS-Mitteilungen 15, S. 75 .. 84 (1996)

[20] *Matthias Withopf*, "Virtuelles Tandem – Hyper-Threading im neuen Pentium 4 mit 3,06 GHz". ct 2002, H. 24, S. 120–127.

[21] *Christian Siemers, Axel Sikora* (Hrsg.), "Taschenbuch Digitaltechnik". Fachbuchverlag Leipzig im Carl Hanser Verlag, München Wien, Januar 2003. ISBN 3-446-21862-9

[22] *Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, Rumi Zahir*, "Introducing the IA-64 Architecture". *IEEE Micro* Vol. 20(5), pp. 12–23, 2000.

[23] *Harsh Sharangpani, Ken Arora, "* Itanium Processor Microarchitecture". *IEEE Micro* Vol. 20(5), pp. 24–43, 2000.

[24] *Cameron McNairy, Don Soltis, "* Itanium 2 Processor Microarchitecture". *IEEE Micro* Vol. 23(2), pp. 44–55, 2003.

[25] *Schmitt, F.-J.; von Wendorff, W.C.; Westerholz, K.*: Embedded-Control-Architekturen. Carl Hanser Verlag München Wien, 1999.

[26] *Osamu Takahashi et.al.*: Power-Conscious Design of the Cell Processor's Synergistic Processor Element. *IEEE Micro* 25(5), pp. 10–18, 2005.

[27] Christian Siemers, "Gatter und Pipelines". c't Magazin für Computertechnik 23(8) S.190–192 (2007).

Sachwortverzeichnis

C

R

S

T

U

V

W

X

Z