9. Pipelining und Parallelverarbeitung

9.1 Einführung

Wir haben an anderer Stelle bereits festgestellt, daß der Von-Neumann-Computer eine systematische Schwachstelle hat, also ein "Bottleneck".
Es wird jeweils nur ein einziger Befehl bearbeitet.
Wie bereits an Beispielen gezeigt, wird jeder "Makro"-Befehl des Befehlssatzes in der Regel durch eine Reihe nacheinander ablaufender Mikro-Befehle abgearbeitet.

Eine Erhöhung der Verarbeitungsleistung eines Rechners ist nun prinzipiell dadurch möglich, daß man die Bearbeitung von Befehlen in geeigneter Weise parallelisiert.

Dazu gibt es verschiedene Ansätze.

Die von-Neumann-Architektur kann man auch als eine "Single Instruction-Single Data-Maschine" (SISD) ansehen. Es werden jeweils nur die für die Bearbeitung eines Datenwortes notwendigen Daten bearbeitet, wobei diese Daten im mathematischen Sinne "Skalare" sind, also eindimensional.

Eine Möglichkeit der Parallelisierung ist die gleichzeitige Verarbeitung mehrerer Befehle, von denen jeder wiederum aus skalare Daten arbeitet. Solche Rechner folgen dann dem Prinzip "Multiple-Instruction-Single Data" (MISD). Dazu gehören z. B. auch die heute verbreiteten High-End-Mikroprozessoren, die als "Superskalare" Maschinen bezeichnet werden. Eine andere Alternative sind die "Very Long Instruction Word (VLIW-) Prozessoren.

Eine andere Möglichkeit ist die Beschleunigung auf der Datenseite. Für viele Anwendungen numerischer Art, z. B. die Wettervorhersage oder die Simulation in der Halbleiterphysik, der Hydrodynamik usw. werden Daten in Form großer Felder angelegt, im mathematischen Sinn also als Vektoren und Matrizen.
Kann man mit einem Befehl eine bezüglich der Daten mehrdimensionale Operation ausführen, also z. B. die vollständige Multiplikation von 2 Vektoren mit allen Komponenten, so lassen sich numerische Berechnungen erheblich beschleunigen.

Maschinen, die das unterstützen, folgen dem "Single Instruction-Multiple Data"-Prinzip (SIMD). Zur Familie dieser Rechner gehören die Superrechner der Fa. Cray.
Rechner dieser Art weisen typisch sehr große Registerlängen von mindestens 64 Bit auf.

Im weitestgehenden Fall werden Befehle und Daten jeweils parallel verarbeitet, man hat also eine "Multi-Instruction-Multiple Data"-Maschine.
Erst das sind Parallelrechner im eigentlichen Sinne. Dazu gehören sowohl numerische Superrechner mit mehreren Prozessoren (meistens 3 bis 5, selten mehr als 10), aber auch massiv parallele Rechner mit Dutzenden bis Tausenden von Mikroprozessoren. Ein Beispiel dafür ist die an der CMU (Carnegie Mellon University, Pittburgh, Pa) entwickelte Connection Machine.
Auch die Parallelrechner einiger europäischer Hersteller (z. B. Parsytech in Aachen) gehören dazu. Bei solchen massiv parallelen Architekturen ergeben sich neue Probleme der Rechnerkommunikation. Man unterscheidet beispielsweise dann Architekturen, bei denen alle Prozessoren einen gemeinsamen Speicher haben, von solchen, bei denen jeder Prozessor seinen eigenen Speicher besitzt.

9.2 Pipelining

9.2.1 Grundlagen

Das grundlegende Bestreben bei der Konstruktion moderner Rechner ist die Erhöhung der Leistung. Dies ist in hohem Maße durch Fortschritte auf drei Gebieten erfolgt:

a. Miniaturisierung und Steigerung der Verarbeitungsgeschwindigkeit der elektronischen Komponenten: dies war bisher der wesentliche Grund für die Leistungssteigerung

b. Rechnerarchitektur bei den einzelnen Prozessoren: Hier haben Pipelining und VLIW-Architekturen wesentliche Fortschritte gebracht.

c. Parallelverarbeitung: Wesentliche Fortschritte hat die Aufteilung von arithmetisch intensiven Aufgaben auf mehrere Prozessoren gebracht.

Nachfolgend sollen zunächst Grundlagen und Probleme des Pipelining vorgestellt werden, da dieses Prinzip in fast allen modernen "superskalaren" Prozessoren verwendet wird.

Die grundlegende Idee ist die parallele Ausführung mehrerer Befehle.
Beim Pipelining liegen dabei mehrere Befehle wie auf einem Fließband in der Bearbeitung und werden teilweise parallel ausgeführt. Entsprechend der Technik beim Fließband werden dabei die bei einem Befehl auszuführenden Operationen in kleine Abschnitte zerlegt.
Entsprechend den Stationen eines Fließbandes spricht man von Pipeline-Stufen (pipe-stage) oder von Pipeline-Segmenten (pipe-segments).
Die Stufen sind in einer festen Reihung miteinander verbunden.
Ein Befehl tritt in die erste Stufe ein und verläßt nach vollständiger Bearbeitung die Pipeline wieder bei der n-ten Stufe.

Der Durchsatz einer Pipeline wird davon bestimmt, wie oft Befehle nach Bearbeitung die Pipeline wider verlassen. Die Pipeline-Stufen sind miteinander verbunden und synchronisiert. Die erforderliche Zeit für den Transport eines Befehls in der Pipeline ist ein Maschinenzyklus. Die Länge eines Maschinenzyklus wird von der langsamsten Pipeline-Stufe bestimmt. Der Maschinenzyklus kann einem Taktzyklus der Maschine entsprechen, manchmal sind aber auch 2 oder (selten) mehr. Dann hat das Taktschema verschiedene Phasen.

Eine Pipeline wird dann am besten funktionieren, wenn nicht viele schnelle Stufen auf eine langsame Stufe warten müssen. Die Aufgabe des Rechnerarchitekten ist es hier also, die "Arbeit" möglichst gleichmäßig auf mehrere Pipeline-Stufen zu verteilen.

Ist diese Befehlsbearbeitung optimal verteilt, so läßt sich ein "Gewinn-Faktor " definieren, der sich im Idealfall ergibt aus:

$$G = \frac{\text{Befehlsausführungszeit der ohne Pipeline implementierten Maschine}}{\text{Zahl der Pipeline-Stufen}}$$

Im Idealfall ist also gegenüber der normalen Maschine ein Vorteil zu erwarten, der proportional der Anzahl der Pipeline-Stufen wird.
Real ist dies nicht ganz zutreffend, weil:

- die Ausführungszeiten pro Pipeline-Stufe nicht gleich sind,
- das Pipelining einen Overhead an Hardware (und damit Laufzeit) erfordert
- eine beliebig hohe Zahl an Pipelining-Stufen nicht sinnvoll eingesetzt werden kann.

Pipelining dient vorrangig der Reduzierung der mittleren Ausführungszahl pro Befehl.
Dazu kann entweder die Taktzyklus-Zeit verringert werden oder die Anzahl von Taktzyklen pro Befehl verringert werden. Meistens wird beides in Kombination durchgeführt, wobei die letzere Maßnahme den höheren Gewinn bringt.

Pipelining ist deshalb möglich, weil es in einem sequentiellen Befehlsstrom durchaus nutzbare Parallelität zwischen Befehlen gibt. Es wird also nicht immer das Ergebnis der n-ten Operation eines Programms direkt notwendig für die (n+1)-te Operation sein.

Gegenüber anderen Methoden der Leistungssteigerung hat Pipelining den entscheidenden Vorteil, daß es für den Benutzer direkt nicht sichtbar wird. Der Nutzer einer nach dem Pipeline-Verfahren arbeitenden Maschine hat also keine Notwendigkeit, sich auf neue Paradigmen der Software-Implementierung umzustellen.

Als Beispiel für Methoden und Probleme der Pipeline-Verarbeitung soll die im Kapitel 7 eingeführte DLX-Maschine verwendet werden.

9.2.2 Basis-Pipeline für die DLX - Maschine

Die Ausführung von DLX-Befehlen kann man auf 5 Grundschritte "verteilen":

1. IF - "Instruction-Fetch" (Befehl aus dem Speicher holen)

2. ID - "Instruction-Decode and register fetch" (Befehlsdecodierung und Holen von Register-Inhalten)

3. EX - "EXecution and effective address calculation" (Ausführung und Adreßberechnung)

4. MEM - "MEMory access" (Speicherzugriff)

5. WB - "WRITE-Back" (Rückschreiben zum Zielregister)

Nummer des Befehls	Taktnummer 1	2	3	4	5	6	7	8	9
Befehl i	IF	ID	EX	MEM	WB				
Befehl i +1		IF	ID	EX	MEM	WB			
Befehl i + 2			IF	ID	EX	MEM	WB		
Befehl i + 3				IF	ID	EX	MEM	WB	
Befehl i + 4					IF	ID	EX	MEM	WB

Abb. 9.1: Einfaches Pipelining-Schema für die DLX-Maschine

Für die DLX könnte man eine einfache Pipeline dadurch realisieren, daß man zu jedem Taktzyklus einen neuen Befehl holt.
Jede der Stufen vorstehend angegebenen Stufen bei der Befehlsausführung wird auf eine Pipeline-Stufe abgebildet. Zwar benötigt immer noch jeder Befehl 5 Zyklen, aber bei jedem Zyklus werden 5 Befehle parallel bearbeitet.

Eigentlich reduziert das Pipelining die Durchführungszeit eines einzelnen Befehls nicht, durch den Overhead wird die sogar noch ansteigen.
Trotzdem wird wegen des insgesamt höheren Befehlsdurchsatzes ein Anwenderprogramm schneller ablaufen. Das Schema der Ausführung ist in Abb. 9.2 für die DLX nochmals in anderer Form dargestellt.

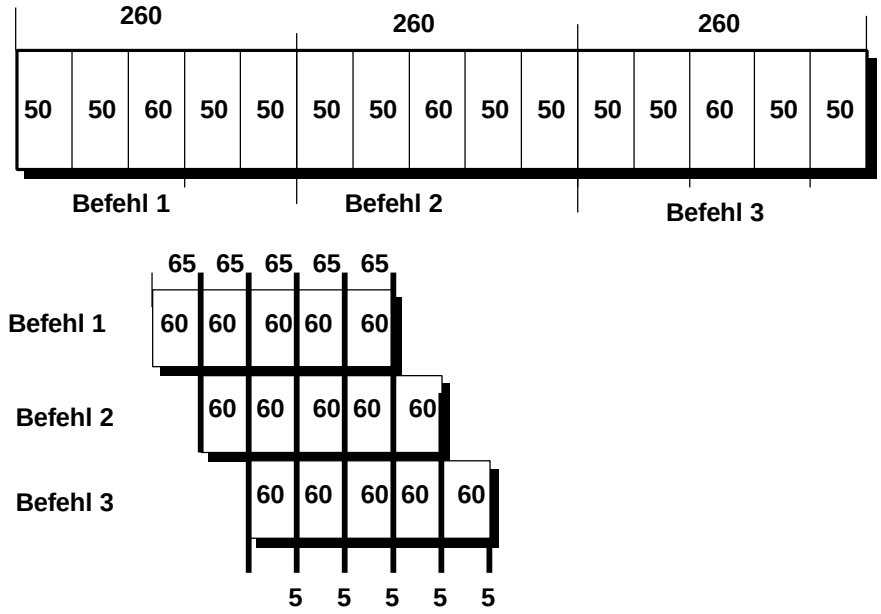

Abb. 9.2: Schema der Ausführung von drei Befehlen mit und ohne Pipeline

In der Version ohne Pipeline wird jeder Befehl in seine Mikrobefehle zerlegt, und diese werden nacheinander ausgeführt. Dabei ist es durchaus möglich, daß die Zeitdauer für die einzelnen Mikrobefehle unterschiedlich ist. In der mit einer Pipeline versehenen Maschine werden dagegen die zu bis zu drei Befehlen gehörenden Mikro-Instruktionen parallel abgearbeitet. Dazu müssen nun die Zeitintervalle für die einzelnen Ausführungen von Mikrobefehlen gleich lang sein, außerdem sind zusätzliche Zeitaufwendungen für den Overhead notwendig.

In der Pipeline-Maschine muß also der Takt der Geschwindigkeit der langsamsten Stufe angepaßt werden.
Für die erreichte Beschleunigung gilt:

$$Bs = \frac{\text{durchschnittliche Befehlsausführung ohne Pipelining}}{\text{durchschnittliche Befehlsausführung mit Pipelining}} = \frac{260}{65} = 4$$

Der Overhead ist maßgeblich bedingt durch die Zwischenspeicherung von Ergebnissen in Latches. Deshalb sind schnelle Latches ein Muß für Pipeline-Maschinen. Der zur Installierung der Pipeline notwendige Overhead setzt dem Maß der Parallelität recht enge Grenzen nach oben. Reale Pipelines besitzen etwa fünf bis zehn Stufen.

9.2.3 Reale Auslegung der Pipeline

So einfach das oben beschriebene Schema im Grunde aussieht, so schwierig ist es in der Praxis, die Pipeline tatsächlich "zum Laufen zu bringen". Zunächst ist dazu genau zu bestimmen, was sich bei jedem Maschinenzyklus ereignen soll. Dazu ist eine optimale Aufteilung der Ressourcen auf die einzelnen Schritte vorzusehen. Zum Beispiel wird eine ALU nicht gleichzeitig eine Addition durchführen und eine effektive Adresse errechnen können. Ggf. sind zusätzliche Ressourcen (z. B. Addierer) notwendig.
Hier ist auch schon einsichtig, daß man bei der Konstruktion des Pipelining für eine Maschine mit komplexem "kryptischen" Befehlssatz mehr Schwierigkeiten haben wird als bei einem einfachen Befehlssatz, wie ihn z. B. die DLX hat.
Dazu muß dann ggf. auch die Ausführung von Teilfunktionen im Taktschema modifiziert werden.

Jede Pipeline-Stufe ist in jedem Taktzyklus aktiv. Dies bedeutet, daß alle Operationen der einzelnen Stufen jeweils innerhalb eines einzigen Taktes angeschlossen werden müssen und daß jede Kombination von Operationen gleichzeitig ausführbar ist.

Die Anforderungen an den Datenpfad der Maschine kann man wie folgt zusammenfassen:

1. Der Programmzähler (PC) muß in jedem Taktzyklus inkrementiert werden. Diese Inkrementierung muß schon in der IF-Phase statt in der ID-Phase erfolgen.
   Da die ALU parallel für arithmetische Operationen benutzt werden muß, ist ein separater Inkrementierer notwendig.
2. In jedem Takt muß ein neuer Befehl geholt werden, also jeweils ein IF ablaufen
3. In jedem Takt kann ein neues Datenwort notwendig sein, wozu jeweils MEM benötigt wird.
4. Es werden getrennte Memory-Daten-Register für das Laden (LMDR) und das Speichern (SMDR) notwendig, denn das Laden von Daten in einem Befehl kann sich mit dem Speichern vom Daten in einem anderen Befehl zeitlich überschneiden.
5. Es werden in der DLX-Architektur zusätzlich drei Latches benötigt, um Daten zwischenzuspeichern, die später im Befehlsablauf (also z. B. zwei Takte später) desselben Befehls benötigt werden, die aber in der Pipeline vorher durch einen anderen Befehl modifiziert werden würden. Diese zu rettenden Informationen sind der jeweilige Befehl selbst, der Ausgangswert der ALU nach deren erster Benutzung und der Programmzähler.

Der im Abschnitt Kapitel 7 diskutierte Befehlsablauf für die DLX muß für das Pipelining modifiziert werden.
Die nachfolgende Tabelle gibt an, was sich in jeder Pipeline-Stufe und -Einheit ereignen kann.

Tabelle 9.1: Funktionale Haupteinheiten und Ereignisse in jeder Pipeline-Stufe

Stufe	**PC-Einheit**	**Speicher**	**Datenpfad**
IF	PC <= PC + 4;	IR <= Mem[PC]	
ID	PC1 <= PC;	IR1 <= IR	A <= Rs1; B <= Rs2;
EX			DMAR <= A + $(IR1_{16})^{16}$## $IR1_{16..31}$; SMDR <= B; **oder** ALUoutput <= A op (B or $(IR_{16})^{16}$##$IR1_{16..31}$); **oder** ALUoutput <= PC1 + $(IR1_{16})^{16}$ ## $IR1_{16...31}$); cond < = (A op 0);
MEM	if (cond)	LMDR<= Mem[DMAR];	ALUoutput1<= ALUoutput; PC <= ALUoutput;
		oder Mem[DMAR]<=SMDR;	
WB			Rd <= ALUoutput1 oder LMDR;

Tabelle 9.1 zeigt die "Belegung" der einzelnen Pipeline-Stufen.
In einigen Stufen können nicht alle Aktionen auftreten, weil sie vom jeweiligen Befehl abhängen. In der EX-Stufe gibt es drei Möglichkeiten. Die erste tritt nur bei Ladebefehlen auf. Die zweite kann bei ALU-Operationen mit dem Eingangswert B oder den niederen, vorzeichenerweiterten Bits des IR auftreten und zwar in Abhängigkeit davon, ob es ein Register-Register- oder ein Register-Immediate-Befehl ist. Die dritte Operation tritt nur bei Verzweigungen auf.
Die Variablen ALUoutput1, PC1 und IR1 retten Variablen zur Bearbeitung in späteren Pipeline-Stufen in spezielle zusätzliche Register. Um die Pipeline zum Laufen zu bringen, muß ein Speichersystem vorhanden sein, das in jedem Takt das Lesen und Speichern von Daten erlaubt. Der Entwurfsaufwand speziell für ein solches Speicher-Management ist relativ hoch.
Die nachfolgende Tabelle gibt die Ereignisse jeder Stufe des DLX-Pipeline an.

Tabelle 9.2: Ereignisse jeder Stufe der DLX-Pipeline

Stufe	**ALU-Befehle**	**Load/Store-Bef.**	**Verzw.-Befehle**
IF	IR <= Mem[PC]; PC <= PC + 4;	IR <= Mem[PC] PC <= PC + 4;	IR <= Mem[PC]; PC <= PC + 4;
ID	A <= Rs1; B <= Rs2; PC1<= PC; IR1<= IR;	A <= Rs1; B <= Rs2; PC1<= PC; IR1<= IR;	A <= Rs1; B <= Rs2; PC1 <= PC; IR1 <= IR;
EX	ALUout <= A op B; **oder** ALUout <= A op ((IR1$_{16}$)^{16}##IR1$_{16...31}$);	DMAR<=A+ ((IR1$_{16}$)^{16}##IR1$_{16...31}$); SMDR <= B;	ALUout<=PC1+ ((IR1$_{16}$)^{16}##IR1$_{16...31}$); cond<= (A op 0);
MEM	ALUout1 <= ALUout;	LMDR <= Mem[DMAR]; **oder** Mem[DMAR] <= SMDR;	if (cond) PC<=ALUout;
WB	Rd<= ALUout1;	Rd <= LMDR;	

Während der ersten beiden Pipeline-Stufen sind die Befehle noch nicht dekodiert. Deshalb sind diese Stufen für alle Operationen identisch. Man kann bereits jetzt bestimmte Register "auf Verdacht" laden. Es ist aber kritisch, irgendwelche Register vor dem Dekodieren des Befehls zu laden. Andererseits kann für das Laden der Register nach dem Dekodieren eine weitere Pipeline-Stufe notwendig sein.
Wegen des festen Befehlsformats der DLX vereinfacht sich die Dekodierung. Es werden immer beide Registerfelder des Befehlsformats dekodiert und es erfolgt ein Registerzugriff. Auch der PC und das Immediate-Feld können der ALU übergeben werden.
Zu Beginn einer ALU-Operation werden durch Schaltung der Multiplexer entsprechend dem Opcode die richtigen Eingangswerte bereitgestellt.
Bei der DLX-Organisation treten alle befehlsabhängigen Operationen in der EX-Stufe oder später auf. Sprünge werden im wesentlichen wie Verzweigungen behandelt, wobei Verzweigungen einen 16-Bit-Offset haben, Sprünge einen 26-Bit-Offset.

Eine kritische Ressource bei Pipeline-Maschinen ist das Speichersystem. Bei der Pipeline-Maschine erfolgen bis zu 2 Speicherzugriffe pro Takt anstatt der zwei Zugriffe in fünf Takten bei der entsprechenden Maschine ohne Pipelining. Um diese schnellen Speicherzugriffe zu ermöglichen, werden oft getrennte, schnelle Befehls- und Daten-Caches verwendet.

In der EX-Stufe kann die Pipeline für drei verschiedene Funktionen genutzt werden: Berechnung einer effektiven Datenadresse, Berechnung einer Verzweigeadresse oder ein ALU-Befehl.
Da die Befehle der DLX einfach sind, können diese Operationen nicht gemeinsam in einem Befehl vorkommen, was die Konstruktion der Pipeline wesentlich erleichtert.
Die Pipeline würde nun einwandfrei funktionieren, wenn jeder Befehl von jedem anderen Befehl in der Pipeline unabhängig wäre. Dies ist aber nicht der Fall und führt zu erheblichen Problemen.

9.2.4 Pipeline-Hazards

Es gibt Situationen, welche die Ausführung des nächsten Befehls aus dem Befehlsstrom im zugeordneten Taktzyklus verhindern. Solche Situationen werden als Pipeline-Hazards oder auch kurz Hazards bezeichnet. Man unterscheidet drei verschieden Typen von Hazards:

1. **Struktur-Hazards** (structural hazards) resultieren aus Konflikten, weil die begrenzte vorhandene Hardware (ALU etc.) nicht alle möglichen Befehlskombinationen simultan ausführen kann.
   Z. B. kann bei Verfügbarkeit nur einer ALU nicht gleichzeitig eine Adreßberechnung und eine arithmetische Addition erfolgen.

2. **Daten-Hazards** (data hazards) verursacht ein Befehl, der so vom Ergebnis eines vorherigen Befehls abhängig ist, daß er eigentlich erst nach dessen Bearbeitung ausgeführt werden kann.

3. **Steuer-Hazards** (control hazards) ergeben sich aus dem Pipelining von Verzweigungen und anderen Befehlen, welche den Programmzähler (PC) ändern.

Wie bei einer normalen Maschine, bei der z. B. ein langsamer Zugriff zu einem externen Speicher zu behandeln ist, müssen gegebenenfalls Wartezyklen (stalls) eingeführt werden. Diese dienen dann dazu, die Verfügbarkeit von Ressourcen (z. B. ALU) abzuwarten oder das Ergebnis eines vorherigen Befehls abzuwarten. Im Unterschied zur normalen Maschine sind aber mehrere Befehle in der Pipeline. Ein Wartezyklus in einer Pipeline bedeutet, daß ein oder mehrere Befehle zu verzögern sind, während andere weiter abgearbeitet werden könnten. Muß aber ein Befehl angehalten werden, so erfordert es das Prinzip der Pipeline, daß die folgenden Befehle auch angehalten werden. Alle Befehle vor dem wartenden Befehl werden abgearbeitet, es wird aber kein neuer Befehl geholt.

Natürlich wirken sich Wartezyklen (stalls) in der Pipeline negativ auf den Durchsatz und damit die Maschinenleistung insgesamt aus. Nimmt man an, daß alle Pipeline-Zyklen gleich lang sind, so kann man die Formel angeben:

$$\text{Pipeline-Beschleunigung} = \frac{\text{Ideal-CPI} * \text{Pipeline-Tiefe}}{\text{Ideal-CPI} + \text{Pipeline-Wartezyklen}}$$

(CPI: Zyklen pro Instruktion)

Hier nicht beachtet ist die potentiell notwendige Erhöhung von Takt-Zykluszeiten infolge des Pipeline-Overheads. Insbesondere bei Maschinen mit komplexen Befehlssätzen kann dieser Overhead erheblich sein.

9.2.5 Struktur-Hazards

Idealerweise sollten bei einer Pipeline-Maschine alle möglichen Kombinationen von Befehlen in der Pipeline behandelt werden können.

Dies kann durchaus nicht nur die Einführung von Hilfsregistern zur Zwischenspeicherung von Daten, sondern auch die Duplizierung von Hardware-Ressourcen erfordern, also z. B. mehrere ALUs oder Addierer.

Wenn einige Befehlsfolgen infolge von Ressourcen-Konflikten nicht untergebracht werden können, dann legen strukturelle oder Struktur-Hazards vor. Der praktisch wichtigste Engpaß ist durch nicht vollständig Pipeline-gemäße Implementierung von Funtionseinheiten gegeben.
Nutzt eine Folge von Befehlen eine solche Funktionseinheit, so können diese nicht sequentiell in die Pipeline eingeleitet werden.

Viele Pipeline-Maschinen haben nur eine Speicher-Pipeline für Daten und Befehle. Dann muß die Pipeline einen Taktzyklus warten, wenn der Befehl einen Speicher-Zugriff enthält. Die Maschine kann den nächsten Befehl nicht holen, weil die Speicher-Schnittstelle zunächst für das Holen des Datums benutzt werden muß. Als Beispiel sei wiederum die Pipeline der DLX-Maschine betrachtet.

Tabelle 9.3: Pipeline mit Stalls wegen Struktur-Hazards beim Speicherzugriff

Befehl	Taktzyklus-Nummer								
	1	2	3	4	5	6	7	8	9
Ladebefehl	IF	ID	EX	MEM	WB				
Befehl i + 1		IF	ID	EX	MEM	WB			
Befehl i + 2			IF	ID	EX	MEM	WB		
Befehl i + 3				stall	IF	ID	EX	MEM	WB
Befehl i + 4						IF	ID	EX	MEM

Die Pipeline hat einen Wartezyklus (stall) infolge eines Struktur-Hazards: Wegen eines Ladebefehls mit einem Speicher-Port kann die Maschine nicht zugleich ein Einlesen eines Datums und des nächsten Befehls durchführen. Die Pipeline wird zu einem Wartezyklus veranlaßt, im Taktzyklus 4 wird der Befehl i + 3 noch nicht geladen. Während der zu holende Befehl wartet, können die anderen Befehle "normal" abgearbeitet werden. Nach dem Wartezyklus wird die Pipeline normal fortgesetzt.

Natürlich ist es möglich, eine Maschine so zu bauen, daß Struktur-Hazards vermieden werden. Dies kann aber nicht nur zu erheblichen Mehraufwendungen an Hardware führen, die sich in der Praxis kaum auswirken, sondern kann auch eine Verlängerung des Taktzyklus bedingen.
Baut man z. B. mehrere Multiplizierer ein, um bei mehreren aufeinanderfolgenden Float-Multiplikationen die Pipeline zu unterstützen, und nimmt dafür längere Taktzyklen in kauf, so wird die Maschine bei Programmen, die Float-Multiplikationen kaum benötigen, sogar langsamer werden.
Viele auch neuere Maschinen (z. B auch der Pentium) haben keine volle Pipeline-Gleitkomma-Einheit.

9.2.6 Daten-Hazards

Mit dem Pipelining wird die zeitliche Ausführung von Befehlen beeinflußt. Daten-Hazards treten auf, wenn die Reihenfolge des Operanden-Zugriffs durch die Pipeline gegenüber der normalen sequentiellen Ausführung geändert ist. Betrachtet seien die Befehle:

```
ADD   R1, R2, R3
SUB   R4, R1, R5
```

Das Register R1 ist gleichzeitig das Ziel des Add-Befehls und die Quelle des SUB-Befehls. Der Konflikt wird im nachfolgenden Ablaufplan der Pipeline deutlich:

Tabelle 9.4: Daten-Hazard bei arithmetischen Befehlen

Befehl	Taktzyklus					
	1	2	3	4	5	6
ADD-Befehl	IF	ID	EX	MEM	WB dat.-wrt.	
SUB-Befehl		IF	IDdat. read	EX	MEM	WB

Der ADD-Befehl schreibt ein Register, das der SUB-Befehl als Quelle benutzt. Aber der ADD-Befehl ist erst 3 Taktzyklen nachdem der SUB-Befehl mit dem Lesen beginnt, mit dem Schreiben fertig.
Typischerweise wird der Subtraktionsbefehl also ein falsches Ergebnis einlesen. Dies ist aber nicht notwendigerweise der Fall: Wenn z. B. zwischen den beiden Befehlen eine Interrupt-Behandlung stattfindet, kann das Ergebnis sogar "zufällig" richtig werden.

Das im Beispiel gezeigte Problem wird mit einer einfachen Hardware-Modifikation gelöst, dem Forwarding (auch Bypassing oder Short-Cutting genannt).
Das Ergebnis einer ALU-Operation wird automatisch und "default" zum Eingangs-Latch derselben ALU durchgeschaltet und dort zwischengespeichert. Wenn die Forwarding-Hardware erkennt, daß die vorangegangene ALU-Operation ein Register zu schreiben hat, das die Quelle der gegenwärtigen ALU-Operation ist, dann wird durch die Steuerung das im Eingangs-Latch gespeicherte Ergebnis anstelle des Inhalts des Registers verwendet. Hat der SUB-Befehl einen Wartezyklus oder tritt zwischendurch ein Interrupt auf, so wird der Bypaß nicht aktiviert, also das Register als Quelle genutzt.
Bei der DLX-Pipeline muß nicht nur das Ergebnis für den unmittelbar folgenden Befehl bereitgestellt werden, sondern auch für den darauf folgenden. Beim dritten Befehl in der Folge wirken die Stufen ID und WB überlappend. Entsprechend muß ein weiteres Ereignis weitergeleitet werden. Die Situation für eine Befehlsfolge ist in Abb. 9. 3 dargestellt.

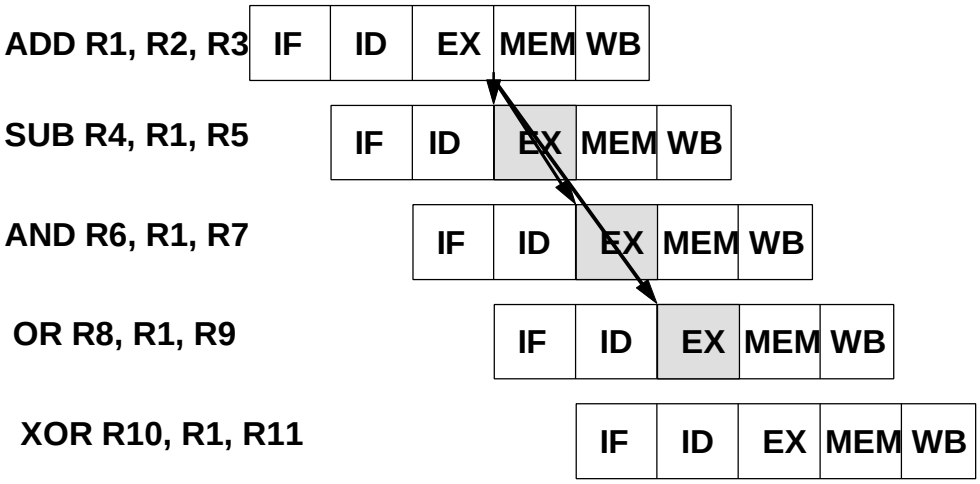

Abb. 9. 3: Befehlspipeline mit Befehlen, die ein Ergebnis-Forwarding erfordern

Der ADD-Befehl setzt R1, und die folgenden vier Befehle benutzen das Ergebnis. Der Wert von R1 muß zum SUB-, AND-, und OR-Befehl weitergeleitet werden. Erst wenn der XOR-Befehl in der ID-Phase R1 liest, ist das richtige Ergebnis dort verfügbar.
Das Forwarding ist eine kostenintensive Maßnahme, wenn es für jede mögliche Folge den Bypaß nutzender Befehle implementiert werden soll.

Es gibt eine andere Möglichkeit, diesen Overhead in Grenzen zu halten. Wenn zu einem Register-File zweimal pro Takt zugegriffen wird, so kann man in der ersten Hälfte von WB schreiben und in der zweiten Hälfte von ID lesen. Dadurch kann sich die Anzahl der Bypaß-Stufen verringern.

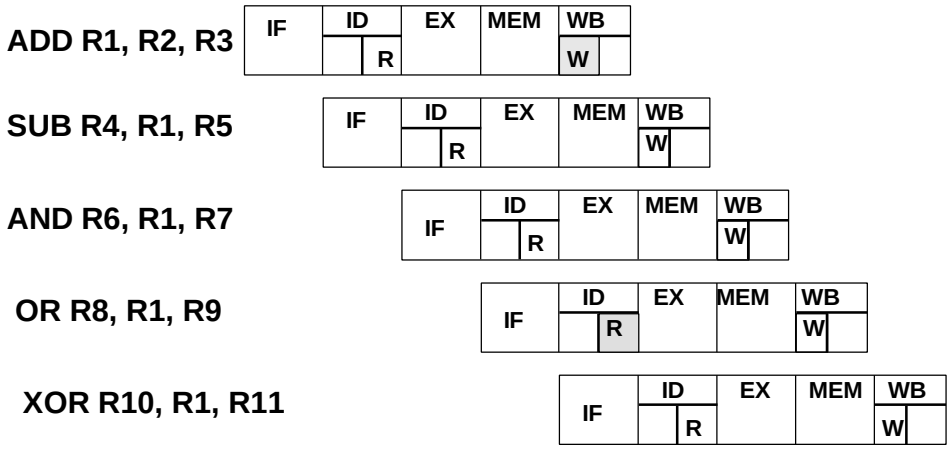

Abb. 9. 4: Pipeline mit Lese- und Schreibzugriff pro Takt

Der SUB- und der AND-Befehl erfordern immer noch den Wert von R1 über den Bypass, der Wert wird in der EX-Stufe benötigt. Wenn aber beim OR-Befehl, der auch R1 nutzt, das Schreiben beendet ist, das Ergebnis bereits eingeschrieben ist, so erübrigt sich das Forwarding. Beim XOR-Befehl ist der Inhalt von R1 verfügbar.

Für das Forwarding benötigt jede Bypaß-Ebene ein Latch und ein Komparator-Paar, um zu testen, ob die angrenzenden Befehle die gleichen Register als Ziel und Quelle haben. Abb. 9. 5 zeigt eine ALU mit Bypaß-Einheit.

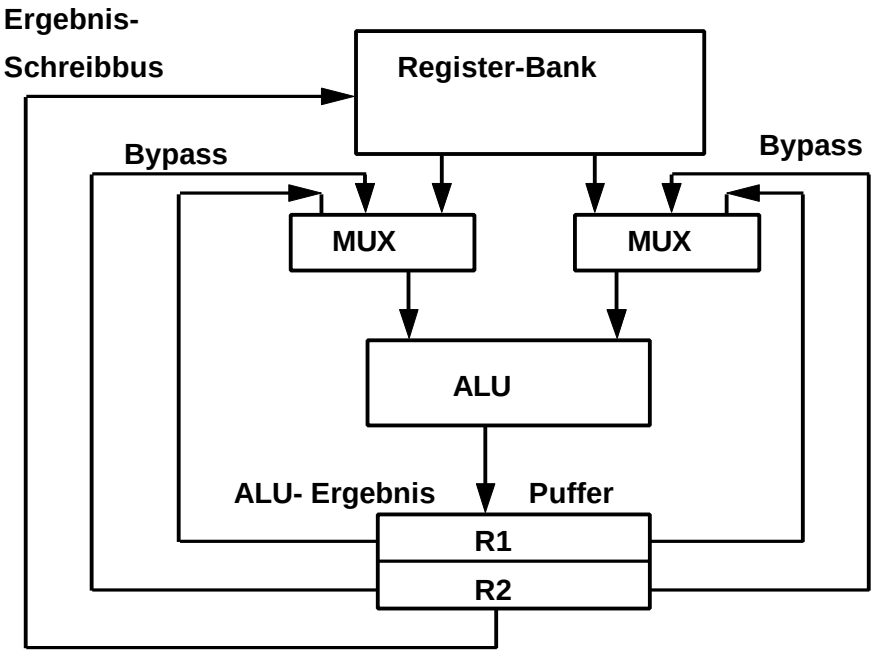

Abb. 9. 5: ALU mit Bypaß für Pipelining

Zwei ALU-Ergebnispuffer werden benötigt, um die ALU-Ergebnisse in den nächsten beiden WB - Stufen in die Zielregister zu speichern.
Bei ALU-Operationen wird das Forwarding immer ausgeführt, wenn Befehle, die ein Resultat als Quelle benötigen, in die EX-Phase eintreten. Die Befehle, welche den Wert für das Forwarding

berechneten, können in der MEM- oder WB-Stufe sein. Die Ergebnisse in den Puffern können entweder Eingangswerte für die Register oder für die ALU sein, die Steuerung erfolgt über ein Paar von Multiplexern. Diese Multiplexer werden ihrerseits von der Steuereinheit kontrolliert, die dann die Quellen und Ziele aller Operationen in der Pipeline kennen und verfolgen muß, oder über eine spezielle, mit der Bypaß-Einheit verbundene Steuerlogik.
Diese Steuerlogik muß dann für jeden Befehlsschritt prüfen, ob einer der zwei vorangegangenen Befehle ein Register schrieb, das den Eingangswert des aktuellen Befehls beinhalten soll. Wenn das erfüllt ist, werden die Multiplexer auf "Forwarding" gesetzt. Damit entfällt jetzt die Notwendigkeit für einen Wartezyklus der Pipeline.

Ein Hazard wird also potentiell immer dann auftreten, wenn es Abhängigkeiten zwischen Befehlen gibt und diese von der Lokalität im Code her dicht genug sind, um den Ablauf der Pipeline zu beeinflussen. Hier waren zunächst nur Konflikte dargestellt, die sich auf Variablen in den CPU-Registern beziehen. Natürlich kann sich die Abhängigkeit auch auf Inhalte des Arbeitsspeichers beziehen.

Die DLX-Pipeline ist so organisiert, daß die Abarbeitung von Befehlen stets in der "richtigen" Reihenfolge geschieht. Wenn z. B. ein Datum im Hauptspeicher (oder Cache) nicht vorhanden ist und erst von der nächsten Stufe der Speicherhierarchie geholt werden muß, stoppt die DLX die Pipeline und wartet. Bei anderen Maschinen ist es nicht nur vorstellbar, sondern sogar üblich, daß man Lade- und Speicherbefehle in einer vom Programm abweichenden Ordnung ausführt.

Insgesamt können Daten-Hazards in drei Kategorien eingeteilt werden:

- RAW (read after write): Schritt j versucht einen Wert zu lesen, bevor er in Stufe i geschrieben wurde. Das ist (wie oben besprochen) der allgemeinste Hazard-Typ

- WAR (write after read): Schritt j versucht einen Wert zu schreiben, bevor er in Stufe i gelesen werden konnte. Damit wird ein "richtiger" Wert vorzeitig durch einen "falschen" überschrieben. Da in der Beispiel-Pipeline alle Leseoperationen in ID und alle Schreiboperationen in WB liegen, kann sich dieser Fall hier nicht ereignen.

- WAW (write after write): Schritt j versucht einen Operanden zu schreiben, bevor er in Schritt i geschrieben wurde. Es wird die Reihenfolge der Schreiboperationen vertauscht., damit wird ein falscher Wert im Register hinterlassen. Dies ist in Pipelines möglich, die in mehr als einer Stufe schreiben. Da die DLX-Pipeline nur in WB schreibt, ist dieser Fehler hier ausgeschlossen.

Ein nicht durch Forwarding voll vermeidbarer Konfliktzustand kann dann auftreten, wenn einem Load-Befehl ein arithmetischer (z. B. ADD) folgt.
Als Beispiel sei die folgende Befehlsfolge betrachtet:

Tab. 9. 6: Konflikt zwischen Load- und arithmetischem Befehl

Befehlsfolge:

```
LW   R1, 32(R6)
ADD  R4, R1, R7
SUB  R5, R1, R8
AND  R6, R1, R7
```

Befehlspipeline:	1	2	3	4	5
LW R1, 32 (R6)	IF	ID	EX	**MEM**	WB

ADD R4, R1, R7	IF	ID	**EX**	MEM	
SUB R5, R1, R8		IF	ID	**EX**	
AND R6, R1, R7			IF	ID	

Das Laden ist nicht bis zum Schluß des MEM-Zyklus erfolgt, und der ADD-Befehl braucht die Daten im EX. Mittels eines Forwarding könnte man aber das Ergebnis von LW direkt vom MDR zur ALU durchgeben, es wäre für den SUB-Befehl rechtzeitig verfügbar. Beim ADD-Befehl wird aber der Inhalt von R1 zu Beginn eines Taktes benötigt, an dessen Ende er erst verfügbar ist. Hier muß zusätzliche Hardware eine Pipeline-Blockierung (Pipeline-Interlock) durchführen.
Die Pipeline wird im "Stall"-Zustand gehalten, bis die Hazard-Bedingung beendet ist. Im vorliegenden Fall ermöglicht der Stall das Laden der Daten vom Speicher. Sie können dann durch Forwarding schnellstmöglich weitergeleitet werden.

Tabelle 9.7: Aufheben des Interlocks beim Speicherzugriff durch Stall

beliebiger Befehl	IF	ID	EX	MEM	WB				
LW R1, 32 (R6)		IF	ID	EX	**MEM**	WB			
ADD R4, R1, R7			IF	ID	stall	**EX**	MEM	WB	
SUB R5, R1, R8				IF	stall	**ID**	EX	MEM	WB
AND R6, R1, R7					stall	IF	ID	EX	MEM

Der Prozeß der Befehlsübergabe von der Befehlsdekodierstufe (ID) in die Ausführungsstufe (EX) der Pipeline wird als Befehlsübergabe (instruction issue) bezeichnet. Für die DLX-FK (Festkomma)-Pipeline kann während der ID-Stufe eine Prüfung auf das mögliche Auftreten von Daten-Hazards erfolgen. Wenn ein Daten-Hazard vorliegt, so wird der nächste Befehl zunächst zurückgehalten und durch einen Leerzyklus ersetzt
Es ist in jedem Fall günstig, Hazards so früh wie möglich zu erkennen, um nicht später durch einen nicht ausführbaren Befehl gestartete Operationen, die den Zustand der ganzen Maschine insgesamt betreffen, rücksetzen zu müssen.

Wartezyklen sind bei Pipeline-Prozessoren ziemlich häufig. Sie treten fast bei jeder Anweisung wie A = B + C auf. Dort ergibt sich jeweils ein Wartezyklus für das Laden des zweiten Datenwertes. Der ADD-Befehl muß angehalten werde, bis das Laden von C erfolgt ist. Dagegen führt das Speichern (store) weniger häufig zu Wartezyklen, weil z. B. das Ergebnis der Addition durch Forwarding zum MDR weitergeleitet werden kann.

Tabelle 9.8: Befehlsfolge für A = B + C

LW R1, B	IF	ID	EX	MEM	WB				
LW R2, C		IF	ID	EX	MEM	WB			
ADD R3, R1, R2			IF	ID	stall	EX	MEM	WB	
SW A, R3				IF	stall	ID	EX	MEM	WB

Das Einfügen von Stall-Befehlen geht natürlich stark zu Lasten der Leistung der CPU insgesamt. Man könnte versuchen, die Häufigkeit der Daten-Konflikte dadurch zu verringern, daß man bei der Compilierung eines Programms die Reihenfolge der Befehle optimiert, also z. B. die Abhängigkeit direkt aufeinanderfolgender Befehle systematisch verringert. Eine solche Methode bei der Compilierung wird "Pipeline Scheduling" oder "Instruction Scheduling" genannt.
Methoden dieser Art arbeiten insgesamt zufriedenstellend, so daß einige Typen von Pipeline-Maschinen eine geeignete Befehlsfolge geradezu voraussetzen. Ein Ladebefehl, der verbietet, daß der nächste Befehl sein Ergebnis nutzt, wird als verzögerter Ladebefehl (delayed load) bezeichnet.
Kann der Compiler ein entsprechendes Scheduling nicht erreichen, so wird ein Leerbefehl (noop) eingeschoben.

9.2.7 Erkennung von Daten-Hazards in einfachen Pipelines

Bei jeder Pipelining-Maschine wird eine Überwachungseinheit notwendig, die Konflikte frühzeitig erkennt und entweder durch Forwarding (bei Struktur-Hazards), durch Verzögerung der Pipeline (bei Daten-Hazards) oder durch eine Kombination beider Methoden verhindert. Wie aufwendig diese Kontrolleinheit wird, hängt einmal von der Komplexität des Befehlssatzes, zusätzlich aber auch von der Anzahl der Pipeline-Stufen ab. Für eine Maschine mit komplexen, lang laufenden Befehlen und gleichzeitig einer hohen Zahl von Pipeline-Stufen kann eine Art zentraler Tabelle notwendig werden. Für die DLX-Maschine mit einem einfachen Befehlssatz und einer nur fünfstufigen Pipeline hält sich der Aufwand in vertretbaren Grenzen. Im wesentlichen sind hier nur Load-Befehle zu überwachen und bei Daten-Hazards sowohl ein "stall"-Befehl einzufügen als auch die Bypass-Hardware zwischen den Registern zu aktivieren.

Benötigt wird als zusätzliche Hardware:

- zusätzliche Multiplexer am Eingang der ALU (dieselben wie für das Forwarding)
- ein zusätzlicher Verbindungspfad vom Memory-Datenregister (MDR) zu den beiden Multiplexer-Eingängen der ALU
- vier Vergleicher (Komparatoren), um jeweils die Quellen-Registerfelder (also die Teile im Befehlswort, welche Adress-Information enthalten) und die Ziel-Registerfelder vorheriger Operationen vergleichen zu können.

Diese Vergleicher prüfen auf Ladebefehls-Blockierungen zu Beginn eines EX-Zyklus.
Die DLX-Architektur ist also, was die Komplexität der Hardware-Aufwandes betrifft, für ihre Festkomma-Pipeline recht günstig.

Als Beispiel ist in Tab. 9.10 eine typische Befehlsfolge in mehreren Versionen dargestellt.

Tabelle 9.10: Befehlsfolgen und Pipeline-Aktionen

Fall	**Beispielfolge**	**Aktion der Kontrolleinheit**

Keine Abhäng.	LW **R1**, 45 (R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		normaler Ablauf, da kein Hazard möglich

Abh. erfordert Wartezyklus	LW **R1**, 45 (R2)
	ADD R5, **R1**, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		Vergleicher erkennen die Nutzung von R1 in ADD und halten den ADD-Befehl, SUB und OR) an

Abhängigkeit mit Forwarding vermieden	LW **R1**, 45 (R2)
	ADD R5, R6, R7
	SUB R8, **R1**, R7
	OR R9, R6, R7
		Vergleicher erkennen die Nutzung von R1 in SUB und stellen das Ergebnis des Ladebefehls durch Forwarding rechtz. für die EX-Phase v. SUB bereit

Abhängigkeit von Zugriffen in zeitgerechter Reihenfolge	LW **R1**, 45 (R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, **R1**, R7
		Keine Aktion erforderlich, da das Lesen von R1 durch OR in der 2. Hälfte der ID-Phase und das Schreiben der zu ladenden Daten in der 1. Hälfte erfolgt

9.2.8 Steuer-Hazards

Steuer-Hazards sind potentiell noch gefährlicher als Daten-Hazards und können einen größeren Leistungsverlust bewirken.

Verzweigungs- und Sprungbefehle sind die Ursache für Steuer-Hazards.
Wenn ein Verzweigungsbefehl den PC mit seiner Zieladresse ändert, so liegt eine ausgeführte (taken) Verzweigung, anderenfalls eine nicht ausgeführte Verzweigung vor (dann wird einfach 4 addiert, um zum nächsten 32 Bit-Befehl zu gelangen). Wenn in der Pipeline der Befehl i eine ausgeführte Verzweigung ist, dann wird normalerweise der PC bis zum Ende von MEM nicht geändert. Erst dann ist nämlich das Ergebnis der Adreßberechnung und ggf. eines Vergleichs (der den Sprung bedingt auslöst) bekannt. Das bedeutet 3 Wartezyklen, bis der neue Wert des PC bekannt ist und der richtige Befehl zur Fortsetzung des Programms geholt werden kann. Die folgende Tabelle zeigt den Drei-Zyklen-Wartezyklus bei einem Steuer-Hazard für die DLX-Architektur im Idealfall.

Tabelle 9.11 a: Ideale DLX-Pipeline mit Wartezyklen nach einem Steuer-Hazard

Verzw.-Befehl	IF	ID	EX	MEM	WB					
Bef. i + 1			stall	stall	stall	IF	ID	EX	MEM	WB
Bef. i + 2				stall	stall	stall	IF	ID	EX	MEM
Bef. i + 3					stall	stall	stall	IF	ID	EX
Bef. i + 4						stall	stall	stall	IF	ID
Bef. i + 5							stall	stall	stall	IF
Bef. i + 6								stall	stall	stall

Die Schwierigkeit besteht darin, daß der Verzweigebefehl nicht dekodiert ist, bis der Befehl i + 1 geholt worden ist. Dieser Befehl wird also geholt, aber wahrscheinlich nicht benötigt. Deshalb wird der wirkliche Ablauf in folgender Tabelle dargestellt:

Tabelle 9.11 b: Reale DLX-Pipeline bei mit Wartezyklen bei einem Steuer-Hazard

Verzw.-Bef.	IF	ID	EX	MEM	WB				
Bef. i + 1		IF	stall	stall	IF	ID	EX	MEM	WB
Bef. i + 2			stall	stall	stall	IF	ID	EX	MEM
Bef. i + 3				stall	stall	stall	IF	ID	EX
Bef. i + 4					stall	stall	stall	IF	ID
Bef. i + 5						stall	stall	stall	IF
Bef. i + 6							stall	stall	stall

Das "ideale" Verhalten nach 9.11 a ist nicht möglich, da erst am Ende des Hol-Vorgangs für den Befehl Nr. (i + 1) durch die Dekodierung des Befehls i klar ist, daß eine Verzweigungsbedingung vorliegt. Man hat also möglicherweise den falschen Befehl geholt. Aus diesem Grund muß nach der Verzweigung der Hol-Vorgang wiederholt werden, wenn das Ziel der Verzweigung bekannt geworden ist.
Trotzdem ist der Verlust von drei Taktzyklen für die Leistungsfähigkeit des Rechners ein signifikanter Verlust.
Bei einer anzunehmenden mittleren Häufigkeit von 30% für Verzweigungsbefehlen in Programmen und einem CPI-Wert (Cycles per Instruction) von 1 erreicht die Maschine bei 3 Wartezyklen (und einer Verfünffachung der Leistung durch 5 Pipeline-Stufen im Idealfall) nur etwa die Hälfte der idealen Leistungssteigerung durch die Pipeline. Das ist natürlich ein ziemlich katastrophales Ergebnis.

Die Anzahl der Warteschritte bei einem Verzweigungsbefehl kann durch zwei Maßnahmen reduziert werden:

1. Früheres Herausfinden der ausgeführten oder nicht ausgeführten Verzweigung in der Pipeline.

2. Früheres Berechnen der Zieladresse für den PC bei auszuführender Verzweigung (PC + 4 oder Verzweigungsadresse)

Tatsächlich ist eine Kombination beider Tricks notwendig.
Dazu sind Änderungen in der Abarbeitung der Befehle in Kombination mit zusätzlicher Hardware notwendig.

Bei der DLX gibt es nur Verzweigungsbefehle (BEQZ und BNEZ), welche die Prüfung einer Variablen auf gleich / ungleich 0 erfordern. Man kann diese Prüfung unter Verwendung einer Speziallogik bis zum Ende des ID-Zyklus durchführen. Soll der Vorteil einer frühen Entscheidung genutzt werden, so ist es günstig, den Wert des PC für beide Fälle (Verzweigung / keine Verzweigung) möglichst früh zu berechnen.
Dann benötigt man für die Berechnung der Verzweigungs-Zieladresse eine separaten Addierer, der während der ID-Phase addieren kann. Mit diesem Addierer und Fällung der Entscheidung während der ID-Phase bleibt dann lediglich ein Wartezyklus bei Verzweigungen übrig.

Tab. 9.12: Überarbeitete Pipeline-Struktur zur Minimierung der Steuer-Hazards bei Verzweigungen

Pipeline-Stufe	**Verzweigebefehl**
IF	IR <= Mem[PC]; PC <= PC + 4;
ID	A <= Rs1; B <= Rs2; PC1 <= PC; IR1 <= IR; **BTA <= PC + ($(IR_{16})^{16}$##$IR_{16....31}$)** **if (Rs1 op 0) PC <= BTA**
EX	(bleibt)
MEM	(bleibt)
WB	(bleibt)

Die Berechnung der Verzweigungsadresse (branch target address) wird jetzt während der ID-Phase durchgeführt, und zwar bei allen Befehlen vor deren endgültiger Dekodierung. Nur bei erfüllter Verzweigungsbedingung (R1 op 0) wird diese Adresse auch wirklich benötigt und als letzter Aktion der ID-Phase in den Befehlszähler geladen. Ob eine Verzweigung wirklich durchzuführen ist, muß zu diesem Zeitpunkt bekannt sein, muß also schon in IF oder spätestens am Anfang von ID erkannt sein. Wenn die Verzweigung schon in ID durchgeführt wird, bleiben die Phasen EX, MEM und WB davon unberührt.
Zusätzliche Komplikationen ergeben sich bei Sprüngen (jumps), die einen noch größeren Offset haben können. Zur Bildung der effektiven Adresse ist ein zusätzlicher Addierer hilfreich, der den Inhalt von PC und den der niederen 26 Bit des Befehlsregisters addiert.
Damit hat die Pipeline-Struktur der DLX eine wesentliche Überarbeitung erfahren. Die Modifikationen sind fett dargestellt. Während der ID-Phase ist eine Branch-Target-Addition (BTA-Addition) eingefügt, die bei allen Befehlen ausgeführt wird.
Die DLX ist ein einfacher Fall. Bei vielen realen Rechnern sind die Probleme noch größer. Bei tiefen Pipelines kann die Verzögerung auch mehr als 3-4 Taktzyklen betragen. Man benötigt also über den

speziellen (einfachen) Fall der DLX hinausgehende Strategien für die Reduzierung von Verzweigungsverlusten.

9.2.9 Generelle Maßnahmen Reduzierung der Pipeline-Verzweigungsverluste

Wir benötigen also eine Strategie, mit Verzweigungen effizient umzugehen.
Dazu wäre es hilfreich, zu wissen, wie häufig welche Verzweigungen zu erwarten sind. Nach Messungen an Standard-Anwendungen für die VAX betreffen ca. 11-17% aller Befehle bedingte Verzweigungen, nur ca. 2-8% sind unbedingte Sprünge bzw. Verzweigungen. Etwa 50% der bedingten Sprünge werden auch ausgeführt, 75% aller Sprünge und Verzweigungen gehen bezüglich der Befehlszählers in Vorwärtsrichtung.
Im einfachsten Fall wird die Pipeline eingefroren. Es werden alle Befehle nach der Verzweigung angehalten, bis das Verzweigungssignal bekannt ist. Dies ist vor allem eine einfache Lösung (entspricht den beiden letzen Tabellen).
Ein besseres und nur leicht komplexeres Schema ist es, die Verzweigung als nicht ausgeführt anzunehmen und so der Hardware einfach die Fortsetzung der Operation zu gestatten. Stellt man doch eine Verzweigung fest, muß dieser Zustand "repariert“ werden, im anderen Fall kann man "ungestört" und ohne Verluste weiterrechnen.
Das nachfolgende Schema zeigt den Zustand der Pipeline mit Nichtausführungsvoraussage, wenn die Verzweigung nicht ausgeführt (oben) und ausgeführt (unten) ist.

Nicht ausg. Verzw.-Bef. IF	**ID**	EX	MEM	WB					
Befehl i+1		**IF**	ID	EX	MEM	WB			
Befehl i+2			IF	ID	EX	MEM	WB		
Befehl i+3				IF	ID	EX	MEM	WB	
Befehl i+4					IF	ID	EX	MEM	WB

Ausführter Verzw. Bef. IF	**ID**	EX	MEM	WB					
Befehl i+1		**IF**	**IF**	ID	EX	MEM	WB		
Befehl i+2			stall	IF	ID	EX	MEM	WB	
Befehl i+3				stall	IF	ID	EX	MEM	WB
Befehl i+4					stall	IF	ID	EX	MEM

Wenn die Verzweigung bei ID nicht ausgeführt wird, erfolgt eine ganz normale Fortsetzung der Operation. Wenn die Verzweigung während der ID-Phase als "auszuführen" erkannt wird, erfolgt als nächster Schritt das Holen des Befehls beim Verzweigungsziel. Das verursacht bei allen der Verzweigung nachfolgenden Befehlen eine Verzögerung von einem Takt.
Zunächst darf der Maschinenzustand nicht geändert werden, bis das Verzweigungsergebnis definitiv bekannt ist. Bei nicht erfolgter Verzweigung läuft im hier bei der DLX verwendeten Schema der "Nichtausführungs-Voraussage" die Maschine einfach weiter. Tritt dagegen die Verzweigung auf, muß die Pipeline gestoppt werden und das Befehleholen wird neu gestartet.

Ein alternatives Schema nimmt die Verzweigung zunächst als erfolgt an und führt entsprechend bei nicht erfolgter Verzweigung den Rücksetzvorgang und einen „stall" ein. Bei der DLX macht diese Variante wegen der sehr einfachen Verzweigungsbefehle keinen Sinn, wohl aber bei Maschinen mit komplexeren Verzweigungsbedingungen im Befehlssatz.

9.2.10 Scheduling und Loop-Unrolling

Bisher sind wir implizit davon ausgegangen, daß die vom Nutzerprogramm vorgesehene Befehlsfolge tatsächlich eingehalten wird.
Wie aus der Compiler-Technologie bekannt ist (oder noch später behandelt wird), führen moderne Compiler durchaus Optimierungen durch, die auch das Umordnen von Befehlsfolgen beinhalten. Die

Erzeugung des zeitlichen Ablaufs von Befehlen wird als "scheduling" bezeichnet. Scheduling ist eine Technologie, die sowohl bei Compilern für Software, aber z. B. auch bei der automatischen Hardware-Synthese Verwendung findet.
"Statisches Scheduling" erzeugt die Befehlsfolge im Voraus und damit ohne die Kenntnis von Bedingungen, die sich erst durch Daten ergeben.
Aufwendiger aber auch wirksamer ist "dynamisches Scheduling", bei dem die Reihenfolge der Befehle erst während des Programm-Ablaufs festgelegt wird.

In einigen Typen von Maschinen wird eine sogenannte Methode der "verzögerten Verzweigungen " verwendet. Dazu betrachten wir die folgende Befehlsfolge:

Verzweigebefehl
sequentieller Nachfolger 1
sequentieller Nachfolger 2
.....
sequentieller Nachfolger n

Verzweigeziel (wenn ausgeführt)

Die sequentiellen Nachfolger befinden sich auf den sogenannten "Verzweigeverzögerungsplätzen". Das sind Befehle, die unter der Annahme, daß die Verzweigung nicht erfolgt, noch abgearbeitet werden können, bevor endgültig bekannt ist, ob nun tatsächlich verzweigt wird oder nicht. Im Fall der Verzweigung müßten also diese Befehle in der Pipeline zurückgesetzt werden. Die Strategie ist nun, diesen "Unsicherheitsraum" sinnvoll zu nutzen und mit Befehlen zu füllen, die auf jeden Fall und unabhängig vom Ergebnis der Verzweigungsbedingung abgearbeitet werden müssen. Dazu muß durch entsprechendes "Scheduling" eine Umordnung der Befehle erfolgen.

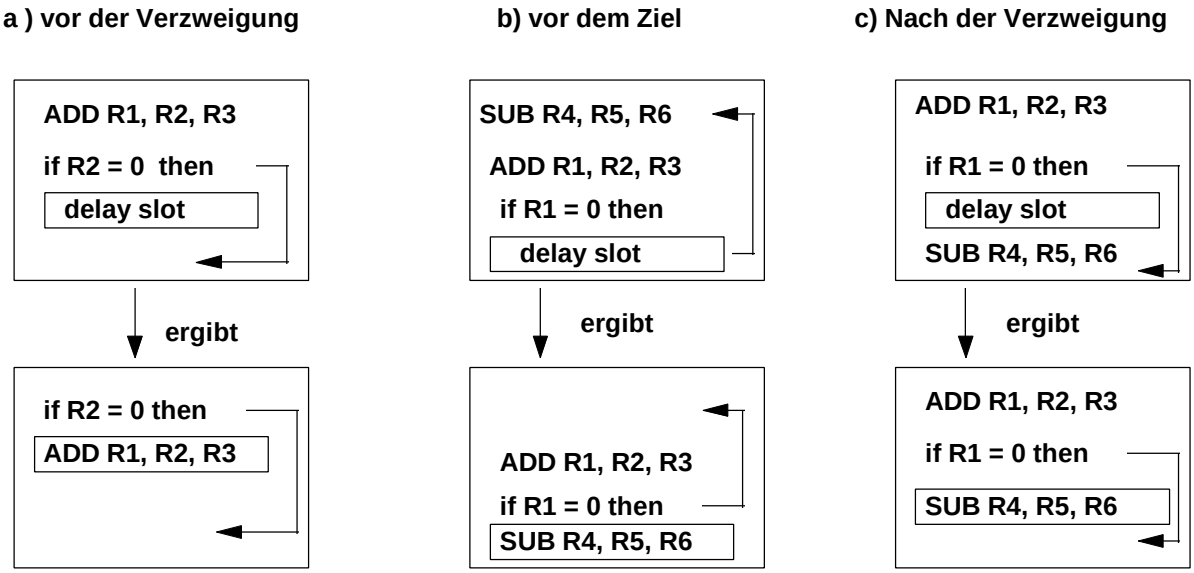

Abb. 9.6: Umordnung von Befehlen zur Vermeidung von Verzweigungsverlusten

Im einfachsten Fall werden die "Delay Slots" mit Befehlen gefüllt, die unabhängig von der Verzweigung durchzuführen sind (a). Im Fall (b) ist diese Unabhängigkeit nicht gegeben, da die Verzweigungsbedingung vom vorherigen Befehl abhängig ist. Jetzt belegt der Scheduler den Delay-Slot mit dem Zielbefehl der Verzweigung. Dazu muß gewöhnlich der Zielbefehl kopiert werden, da er am ursprünglichen Platz auch von einem anderen Pfad erreicht werden kann und deshalb erhalten bleiben muß.

Fall (b) würde dann bevorzugt, wenn die Verzweigung mit hoher Wahrscheinlichkeit eintritt, also beispielsweise in Schleifen-Anweisungen. Im Fall (c) wird der Befehl in den Delay Slot eingefügt, der auftritt, wenn die Verzweigung nicht auftritt.
In jedem Fall muß gesichert sein, daß dann, wenn die Verzweigung nicht in die "erwartete" Richtung läuft, kein falsches Ergebnis entsteht. Der nächste Befehl nach der Verzweigung dürfte also nicht direkt lesend auf das Register R4 zugreifen.

Für den einfachen Fall, daß es nur einen solchen Verzögerungsplatz gibt und etwa in 50% der Fälle die Verzweigungsbedingung erfüllt ist, wird man mit etwa 80% Wahrscheinlichkeit durch eine solche einfache Umordnung einen Verlust an Rechenzeit vermeiden können. Die in Tabelle 9.13 angegebenen Ergebnisse gelten für die 5-stufige Pipeline der DLX, die im Idealfall einen 5-fache Beschleunigung der Befehlsverarbeitung erreichen könnte.

Tab. 9.13: Relative Leistung der DLX- Pipeline bei unterschiedlicher Implementierung der Verzweigung

Scheduling-Schema	Verzw.-verluste (Zyklen)	Effekt. CPI	Beschl. genüb. Masch. ohne Pipe	Beschl. genü. Masch. mit Pipe-Wartezykl.
Pipel. mit Wartezyklen	3	1,42	3,52	1,0
Voraussage: ausgeführt	1	1,14	4,39	1,25
Voraussage: nicht ausgef.	1	1,09	4,59	1,30
Verzögerte Verzweigung	0,5	1,07	4,67	1,33

Eine wesentliche Methode beim Scheduling ist das "Loop unrolling".
Es ist in der Regel einfacher und schneller, eine kleinere, nur einige Male zu durchlaufende Programmschleife "auszurollen" und damit direkt rein sequentiellen Code zu erzeugen. Auch dies ist "erprobte" Compiler-Technologie.
Gerade mit gutem Scheduling und Compiler-Methoden läßt sich die effiziente Ausführung von Verzweigungen in Pipeline-Maschinen wesentlich unterstützen.

9.2.11 Interrupt-Probleme bei Maschinen mit Pipelines

Die Interrupt-Bearbeitung ist bei Pipeline-Maschinen nochmals komplexer und problematischer ist als bei "normalen" Rechnern. Es ist sogar theoretische schwer, bei Pipeline-Maschinen mathematisch vorauszuberechnen, wann eine bestimmte Interrupt-Bearbeitung erfolgt. Mit Methoden wie "dynamischem Scheduling" wird eine solche Maschine weiter "unberechenbar". Viele Praktiker ziehen deshalb für Realzeit-kritische Anwendungen "einfache" Prozessoren vor.
In der Pipeline-Maschine zieht sich einmal die Abarbeitung jedes einzelnen Befehls über mehrere Zyklen hin, andererseits werden auch mehrere Befehle überlappend bearbeitet. Deshalb ist es besonders schwierig, die Bearbeitung eines Interrupts effizient zu organisieren, den jeweiligen Zustand der Maschine vor den Interrupt zu retten und das "Wiederaufsetzen" zu organisieren, also die Maschine restartbar zu machen. Im Fall des Interrupts muß zunächst die Pipeline sicher abgeschlossen werden und alle wichtigen Status-Informationen müssen gerettet werden.

Nach Abarbeitung des Interrupts wird zunächst der Status wiederhergestellt und der letzte noch nicht beendete Befehl nochmals bearbeitet.
Dazu wird man zunächst im IF-Modus schon den PC-Wert eines in die Ausführung gehenden Befehls sichern. Bei einer Organisation des Rechners in der Form, daß beim Interrupt der laufende Befehl noch abgearbeitet wird, würde man beim Wiederaufsetzen den nächsten Befehl, d. h. den mit der nächstfolgenden Speicheradresse starten. Ist der wiederaufgegriffene Befehl ein Verzweigungs- oder Sprungbefehl, so ist zunächst die Zieladresse zu bestimmen.

Wenn ein Interrupt auftritt, sind folgende Schritte auszuführen, um den Zustand der Pipeline zu retten. Wir nehmen an, daß bei der Ausführung eines Befehls eine Interrupt-Meldung auftritt, die einen Fehler anzeigt (z. B. arithmetischer Überlauf, Seitenfehler beim Speicherzugriff etc.)

1. Erzwingen eines TRAP-Befehls zur Interrupt-Barbeitung in der Pipeline beim nächsten IF
2. Verhinderung des Schreibens im fehlerhaften Befehl und für alle anderen in der Pipeline befindlichen Befehle, bis die Interrupt-Bearbeitung ausgeführt ist.
3. Nachdem die Interrupt-Routine vom Betriebssystem eingeschaltet wird, rettet diese zunächst den PC des unterbrochenen Befehls. Dies ist die Voraussetzung für ein späteres Wiederaufsetzen.

Wir haben aber gerade gelernt, daß für ein reibungsloses "Füllen" der Pipeline gerade auch bei Verzweigungen eine Umordnung der Befehle sogar zur Laufzeit möglich ist. Wenn dies geschieht, so reicht es jetzt nicht mehr, nur den aktuellen Inhalt des PCs zu retten, sondern man muß mehrere PC-Inhalte retten und zurückspeichern. Genau muß ein PC mehr gerettet werden als die maximale Länge der Verzweigungs-Verzögerungen betragen kann.

Ist der Interrupt beendet, so wird die Pipeline zurückgeladen. Im Idealfall wird man die Maschine so konstruieren, daß alle Befehle vor dem zum Interrupt führenden fehlerhaften Befehl voll ausgeführt werden und beendet sind und alle nachfolgenden Befehle neu gestartet werden. In diesem Fall hat die Maschinen einen „präzisen Interrupt" Die korrekte Behandlung des Interrupts erfordert, daß der abgebrochene Befehl am Ende des Interrupts keinen Einfluß mehr hat.
Bei einigen Maschinen kann es aber auch vorkommen, daß der fehlerhafte Befehl vor der Abarbeitung des Interrupts noch ein Ergebnis schreiben kann.

Weil in einer Pipeline mehrere Befehle gleichzeitig abgearbeitet werden, können sogar in einem Takt mehrere Interrupts gleichzeitig auftreten. Beispiel:

LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

Hier ist das gleichzeitige Auftreten eines Seitenfehlers (wenn ein Datenwort nicht aus dem Hauptspeicher geladen werden kann, weil es erst von der Platte geholt werden muß) und eines arithmetischen Fehlers möglich.
Bei der Abarbeitung wird im Idealfall zunächst der Seitenfehler beseitigt, die Pipeline neu gestartet und erst nachfolgend und unabhängig der arithmetische Fehler behandelt. Leider funktioniert die Abarbeitung nicht immer so einfach. So kann der Interrupt für einen späteren Befehl auch vor dem Interrupt für einen früheren Befehl auftreten. So kann im obigen Beispiel der LW (load word) einen Interrupt in der MEM-Phase bekommen, ADD kann einen Interrupt durch einen Seitenfehler auch schon in der IF-Phase bekommen.

Um solche Probleme zu beherrschen, wird man als eine Lösung zu jedem Interrupt einen Interrupt-Statusvektor definieren und diesen in jeder Pipeline-Stufe mitführen und prüfen lassen. Interrupts werden dann in der Reihenfolge bearbeitet, die dem "natürlichen" Befehlsablauf entsprechen würde.
Eine andere Möglichkeit besteht darin, jeden Interrupt direkt nach dem Erscheinen abzuarbeiten, also das Auftreten mehrerer Interrupts mit zu koordinierender Behandlung möglichst auszu-schließen. Tabelle 9.14 gibt eine Übersicht über die problematischen Interupts der DLX-Maschine.

Tab. 9.14: Kritische Interrupts der DLX-Pipeline-Maschine

Pipeline-Stufe	Auftretende Problem-Interrupts
IF	Seitenfehler beim IF, nichtausgerichteter Speicherzugriff, Speicherschutzverletzung
ID	Undefinierter oder illegaler Befehlscode
EX	Arithmetischer Interrupt
MEM	Seitenfehler beim Datenholen, nichtausgerichteter Speicherzugriff, Speicherschutzverletzung
WB	keine

Generell kann man wohl vermerken, daß Pipeline-Maschinen in Anwendungen, die stark von externen Interrupts getrieben sind, keine sonderlich guten Leistungen zeigen, da sie nach jedem Interrupt die Pipeline neu organisieren müssen.
Darauf kann ein Programmierer schon beim Entwurf etwa eines Steuerungs- und Regelungssystems Rücksicht nehmen. Abgesehen davon, daß High-End-Prozessoren für "embedded controller" Anwendungen wegen Kosten und Leistungsverbrauch nicht häufig eingesetzt werden, würden sie gerade dort im Widerspruch zu ihrer Architektur verwendet werden.

9.2.12 Pipeline-Maschine mit Mehrzyklus-Operationen

In der bisher behandelten DLX-Architektur sind wir davon ausgegangen, dass die arithmetischen Operationen in der Regel in einem Taktzyklus abzuschließen sind. Das gilt aber in der Regel nur für Festkomma (FK-)Addition und -Subtraktion, während Multiplikation und Division allgemein und auch Gleitkomma (GK)-Addition und -Subtraktion eine größere Anzahl von Maschinenzyklen erfordern.
Damit sind solche Operationen mit dem "normalen" Aufbau der DLX-Pipeline nicht sonderlich kompatibel. Man könnte zwar theoretisch auch die GK-Operationen in einem Takt ausführen, aber das würde zu sehr großen Logik-Bausteinen verbunden mit sehr langen Taktzeiten führen. In der Regel wird man deshalb eine spezielle Gleitkomma-Pipeline vorsehen, welche für die einzelnen Schritte, so sie mehr Zeit benötigen (z. B. Gleikomma-Divison), auch mehr Zeit bereitstellt.

Die Gleitkomma-Pipeline kann im wesentlichen denselben Aufbau wie die Festkomma-Pipeline haben mit zwei signifikanten Unterschieden:

1. Der EX-Zyklus kann so oft wie notwendig wiederholt werden, um eine komplexe Operation zu beenden. Dabei kann die Anzahl der Wiederholungen für verschiedene Operationen unterschiedlich sein.

2. Es kann mehrere GK-Funktionseinheiten geben. Wartezyklen der Pipeline sind möglich und notwendig, wenn Struktur- oder Daten-Hazards auftreten.

Üblich sind in der Praxis die folgenden arithmetischen Einheiten:

a. die "normale" Festkomma-Einheit, also meistens die ALU
b. ein GK- und FK-Multiplizierer
c. ein GK-Addierer (und -Subtrahierer)
d. ein GK- und FK-Subtrahierer

Wir wollen zunächst annehmen, daß unsere DLX-Maschine über diese Funktionseinheiten verfügt und damit eine Gleikomma-fähige Pipeline-Struktur benötigt. Sie soll aber keine "getrennte" Gleitkomma-Pipeline haben (moderne superskalare Prozessoren haben tatsächlich zwei oder mehr Pipelines).

Dann soll die normale FK-Einheit alle Lade- und Speicherbefehle sowohl für die Festkomma- als auch für die Gleitkomma-Register ausführen.

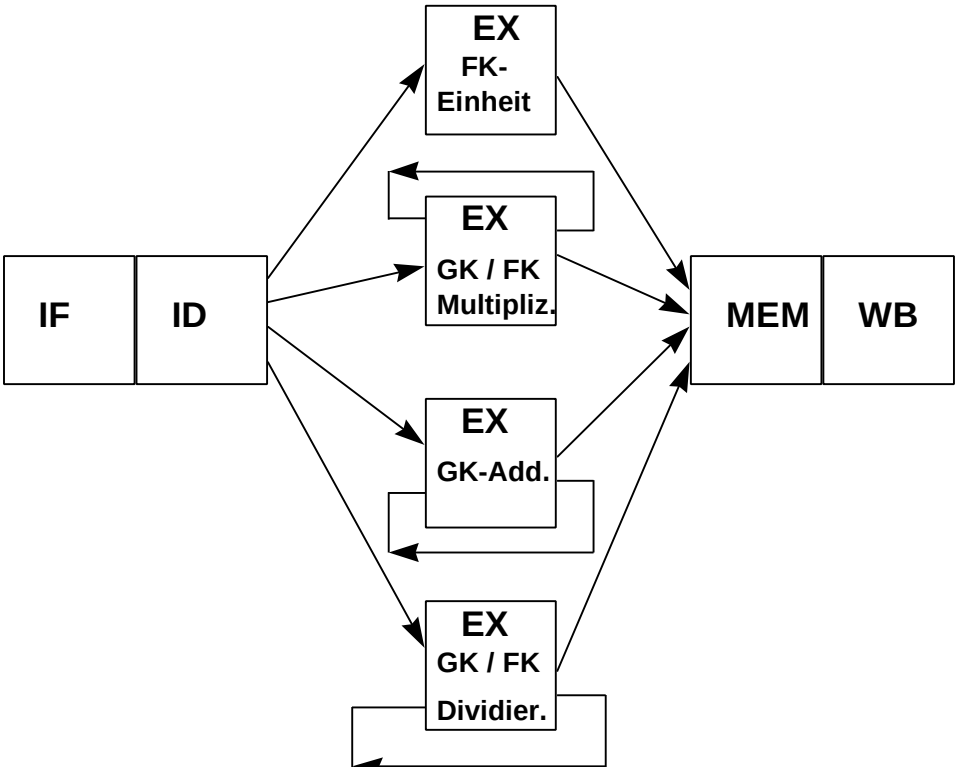

Abb. 9.7: DLX-Pipeline-Maschine mit zusätzlichen GK-Funktionseinheiten ohne Pipeline

Es sei weiterhin angenommen, daß diese Funktionseinheiten selbst nicht mit Pipelines versehen sind. Dies bedeutet, daß kein auf einen arithmetischen Befehl zur Ausführung kommt, bevor nicht der vorhergehende die EX-Phase verlassen hat.

Wenn also ein Befehl in der EX-Phase nicht in die benötigte Funktionseinheit gelangen kann, so werden alle nachfolgenden Befehle verzögert.

Natürlich können diese zusätzlichen Randbedingungen die Pipeline ganz erheblich ins Stocken bringen. Für eine komplexe arithmetische Operation wie etwa eine Gleitkomma-Division kann die EX-Phase durchaus 30 bis 50 mal wiederholt werden. Damit sind Pipeline-Konflikte vorprogrammiert:
Will man jetzt z. B. Struktur- und Datenhazards vermeiden, so müssen dazu entsprechend lange Sequenzen von Befehlen analysiert und ggf. durch Umstellen der Reihenfolge optimiert werden. Eine zeitliche Überlappung von FK- und GK-Befehlen wird man sich leisten müssen, um nicht FK-Befehle durch GK-Operationen unmäßig zu verzögern. Es ist auch üblich, getrennte FK- und GK-Register zu implementieren, was die Ausführung der Steuerung wesentlich vereinfacht.

Nachfolgend sei vereinfachend angenommen, daß alle GK-Operationen dieselbe Anzahl von Takten benötigen, z. B. 20, zur Ausführung der Operationen in der EX-Phase benötigen.
Auftreten können dann nur Daten-Hazards vom Typ "Read after Write"-RAW.

Soll ein neuer GK-Befehl zur Ausführung an eine der vier Funktionseinheiten übergeben werden, so sind folgende Schritte notwendig:

1. Prüfung auf Struktur-Hazards, d. h. warten, bis die benötigte Funktionseinheit nicht mehr besetzt ist

2. Prüfung auf RAW Daten-Hazards: Dies bedeutet zu warten, bis das benötigte Quellregister, aus dem gelesen werden soll, nicht mehr als Ziel-Register eines anderen, parallel abgearbeiteten Befehls in einer EX-Phase einer anderen Funktionseinheit ist.

3. Prüfung auf Forwarding: Hier ist herauszufinden, ob das Zielregister eines Befehls in MEM oder WB einem der Quellregister für den auszuführenden GK-Befehl entspricht.

Eine weitere Komplikation kann sich ergeben, wenn mehrere parallel ausgeführte GK-Operationen gleichzeitig die WB-Stufe erreichen und gleichzeitig einen Zugriff auf die Register-Bank benötigen.

Nehmen wir (was real ist) an, daß die GK-Operationen auch unterschiedlich lang sein können, so ergeben sich weitere Konfliktmöglichkeiten. GK-Additionen werden nämlich typisch in etwa 5 Zyklen abgearbeitet sein, während Multiplikationen etwa 10 Operationen und Divisionen 20 und mehr benötigen. Man muß also die Steuerung so auslegen, dass eine variable Anzahl von Schleifen-Operationen möglich ist.
Nun können auch alle Arten von Daten-Hazards auftreten (write-after-read-WAR, write-after-write-WAW). Zusätzliche Probleme bei der Interrupt-Behandlung sind auch anzunehmen.

Um Konflikte beim Rückspeichern auszuschließen, kann man den einzelnen Funktionseinheiten eine feste "Priorität" zuordnen, z. B. hätte der GK-Multiplizierer stets Vortritt vor der FK-ALU.
Wenn die benötigten Taktzahlen der GK-Einheiten unterschiedlich sind, so können jetzt auch WAR- und WAW-Konflikte auftreten, da der Zeitpunkt für das Rückschreiben der Ergebnisse nicht mehr fest ist. WAR-Konflikte können aber verhindert werden, wenn alle GK-Befehle ihre Register zur gleichen Zeit lesen.
WAW-Hazards sind noch möglich, weil Befehle ihre Ergebnisse in anderer Reihenfolge schreiben können als dies im Programm vorgesehen ist.

Beispiel:

```
DIVF  F0, F2, F4
SUBF  F0, F8, F10
```

Hier ist ein WAW-Hazard zwischen dem Divisions- und dem Subtraktionsbefehl zu erwarten. Die Subtraktion wird zuerst beendet und schreibt ihr Ergebnis vor der Division. Dieser Fall tritt aber tatsächlich nur auf, wenn das Ergebnis der Division überschrieben wurde, ohne daß es tatsächlich jemals von einem Befehl genutzt wurde. Wenn es eine Nutzung von F0 zwischen DIVF und SUBF gäbe, dann würde wegen der Datenabhängigkeit in der Pipeline ein Wartezyklus auftreten, und SUBF würde nicht in die EX-Phase übergehen.
Trotzdem sollte das Problem hier nicht vergessen werden, weil bei bestimmten Befehlsfolgen doch Konflikte auftreten können.

Zur Lösung dieser Konflikte gibt es zwei sinnvolle Wege:

1. Verzögerung der Übergabe des Subtraktionsbefehls, bis DIVF in die MEM-Phase eintritt.
2. Bei Hazard-Erkennung Aussetzung der Rückschreib-Phase des Divisionsbefehls

Beide Lösungen sind gleichwertig bezüglich des (jeweils wegen der relativen Seltenheit des Konflikts) eher unkritischen Einflusses auf den Durchsatz der Pipeline. Man wird also die Lösung wählen, welche bezüglich der Implementierung im Steuerwerk weniger Aufwand verursacht.
Ein weiteres Problem bei langlaufenden Befehlen sei durch die nachstehende Befehlsfolge beschrieben:

```
DIVF  F0, F2, F4
ADDF F10, F10, F8
SUBF  F12, F12, F14
```

Diese Befehlsfolge sieht zunächst harmlos aus, weil keine Abhängigkeiten zwischen den Daten ersichtlich sind. Allerdings werden ADDF und SUBF abgeschlossen sein, bevor DIVF ein Ergebnis bringt.
Man bezeichnet diesen Effekt als "out-of-order completion", d. h. "Fertigstellung außerhalb der Reihenfolge". Er ist typisch für langlaufende Pipelines.

Im Normalbetrieb ist dieser Effekt unkritisch, böse werden die Verhältnisse erst, wenn in der Phase, wo die späteren Befehle fertig sind, der langlaufende frühere aber nicht, ein Interrupt auftritt. Hier sei angenommen, daß bei SUBF ein Arithmetik-Interrupt auftritt. Dann kann das zu unpräzisen Interrupt-Bedingungen führen. Bei der FK-Pipeline war es dann notwendig, zur Interrupt-Bearbeitung die Pipeline leerlaufen zu lassen. Wenn z. B. DIVF nach dem Abschluß der Addition einen GK-Interrupt auslöst, so ist auch eine präzise Interrupt-Verarbeitung mit Wiederherstellung des vorherigen Zustandes nicht immer möglich. Hier hat z. B. ADDF selbst einen seiner Operatoren zerstört. Er ist nicht mal mittels Software-Prozeduren rekonstruierbar.

Als Abhilfe existieren vier Varianten:

1. Das Problem wird nicht beachtet, man nimmt einen unpräzisen Interrupt in kauf.
   Dies bedeutet auf der anderen Seite, daß bestimmte Typen von Interrupts nicht erlaubt sind oder "extern" behandelt werden, ohne die Pipeline zu stoppen.

2. Speichern der Zwischenergebnisse in einer Warteschlange, bis alle früher übergebenen Operationen beendet sind. Dies kann eine recht lange Queue und entsprechenden HW-Aufwand bedeuten. Zusätzlich ist eine Bypaß-Funktion bereitgestellt werden, damit die Übergabe von Befehlen während der Wartezeit fortgesetzt werden kann. Dazu sind wiederum Vergleicher und große Multiplexer notwendig.

3. Zulassung bedingt "unpräziser" Interrupts. Es wird aber so viel Information zurückbehalten, daß eine Software-Routine (Trap-Routine) eine präzise Befehlsfolge zur Behandlung des Interrupts erzeugen kann.

4. Ein hybrides Schema, das die Fortsetzung der Befehlsübergabe nur erlaubt, wenn alle Befehle vor dem übergebenen Befehl ohne Verursachung eines Interrupts angeschlossen sind. Dies garantiert im Falle eines Interrupts, daß alle Befehle vor dem unterbrochenen Befehl und kein nachfolgender abgeschlossen werden. Dies bedeutet manchmal das Anhalten der Maschine, um eine präzise Interrupt-Verarbeitung zu sichern.

Die letztere Methode wird z. B. in den MIPS 2000 / 3000-Prozessoren verwendet, die in UNIX-Servern im Einsatz sind.

9.2.13 Statisches und dynamisches Scheduling, Verzweigungsvoraussage

An dieser Stelle sind wir nun fast bis zu den Methoden und Verfahren vorgedrungen, welche in modernen Prozessoren Verwendung finden.
Ein wesentliches Problem ist das Scheduling von Befehlen, das heißt die Festlegung der zeitlichen Abfolge der Verarbeitung.

Bereits teilweise kennengelernt haben wir statische Scheduling-Verfahren.
Bei Datenabhängigkeiten zwischen mehreren Befehlen wird das Holen neuer Befehle eingestellt, bis die Abhängigkeit beseitigt ist. Für die Minimierung der dafür notwendigen Wartezyklen durch Umordnung von Befehlen ist Software verantwortlich, die z. B. ist das Scheduling der einzelnen Assembler-Befehle bereits eine Aufgabe des Compilers, der prozessor-spezifischen Code aus einer Hochsprache heraus erzeugt.

Dynamisches Scheduling wird dagegen erst zur Laufzeit des Programms durch entsprechende zusätzliche Hardware ausgeführt. Dies benötigt zwar einen nicht unerheblich höheren Aufwand an Hardware, aber erleichtert dem Compiler die Aufgabe und macht dessen Code von einer speziellen Organisation der Pipeline unabhängig.
Dynamisches Scheduling hat bei modernen Maschinen eine ganz wesentliche Bedeutung gewonnen, und zwar für zwei Aufgaben:

- die Erkennung und Auflösung von Daten-Hazards in tiefen Pipelines
- die Erkennung von Verzweigungsbedingungen, die Vorhersage ob die Sprung- oder Verzweiungsbedingung auftritt oder nicht und die frühzeitige Errechnung des Ziels des Sprunges oder der Verzweigung.

Zunächst sollen Methoden zur Auflösung von Daten-Abhängigkeiten bzw. zum Umordnen von Befehlen zu diesem Zweck betrachtet werden. Gegeben ist die Befehlsfolge:

```
DIVF  F0, F2, F4
ADDF F10, F0, F8
SUBF  F6, F6, F14
```

ADDF ist von DIVF anhängig, so daß dessen Ergebnis abgewartet werden muß. SUBF wartet dann ebenfalls, obwohl wegen Datenunabhängigkeit von den vorherigen Befehlen eine Verarbeitung möglich wäre. Man benötigt also ein Verfahren, um durch Hardware die Befehle entsprechend umzuordnen.

Die Dekodierung eines Befehls erfolgt in der DLX in der ID-Stufe der Befehlsverarbeitung. Man wird nun die ID-Stufe eine Teilung der Befehlsübergabe in zwei Phasen durchführen müssen:

- Testen auf Struktur-Hazards
- Warten auf die Abwesenheit von Daten-Hazards.

Dazu wird man nun die bisher verwendete Aufteilung in zwei Schritte:

1. ID: Befehlsdekodierung, Test auf alle Hazards, Operanden-Holen
2. EX: Befehlsausführung

in drei Schritte umorganisieren müssen:

1. Übergabe: Befehlsdekodierung, Test auf Hazards
2. Operanden lesen: warten, bis das Auftreten von Hazards ausgeschlossen ist, dann Operanden lesen
3. Befehlsausführung

Die Übergabestufe wird von allen Befehlen durchlaufen. In der zweiten Stufe können sich dann Befehle über einen speziellen Mechanismus, zum Beispiel Scoreboarding, gegenseitig anhalten oder über Bypass-Mechanismen Ergebnisse vorzeitig bereitstellen.

Neben dem Scoreboarding hat der sogenannte Tomasul-Algorithmus für das dynamische Scheduling eine gewisse Bedeutung erlangt.

Diese Mechanismen sind auf die Vermeidung von Daten-Hazards ausgerichtet. Wir haben aber schon an anderer Stelle gelernt, daß Steuer-Hazards mindestens ebenso kritisch sind. Eine spezielle Erweiterung bei modernen Prozessoren dient deshalb der dynamischen Voraussage von Verzweigungen. Das Ergebnis wird in einem speziellen Verzweige-Voraussagungspuffer abgelegt.

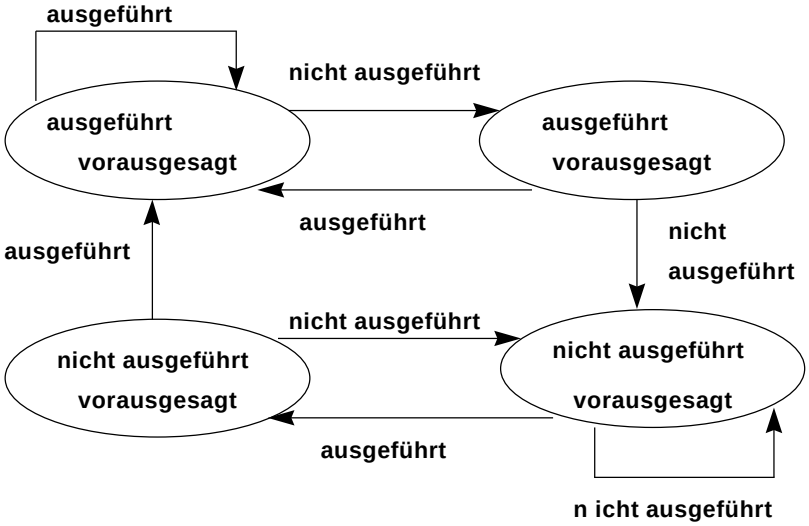

Abb. 9.8: Zustandsdiagramm für die Vorhersage von Verzweigungen

Da Verzweigungen innerhalb von Programmschleifen oft mehrmals in Folge ausgeführt bzw. nicht ausgeführt werden, hat sich ein Schema, bei dem ein "Verzweige-Vorhersagebit" jeweils nur die letzte Verzeigungsoperation berücksichtigt, als nicht sehr sinnvoll erwiesen. Praktisch günstiger ist eine Methode, bei der die Vorhersage erst dann umgekehrt wird, wenn die Voraussage zweimal nacheinander falsch war (Abb. 9.8).

Der Verzweige-Voraussagepuffer wird oft als kleiner, spezieller Cache implementiert, zu dem mit der Befehlsadresse während der IF-Pipeline-Stufe zugegriffen wird. Eine andere Möglichkeit ist die Realisierung als ein Bit-Paar, das jedem Wort im Befehls-Cache angehängt wird und das man mit dem Befehl holt.

Bei jedem Befehl, der sowohl eine Verzweigung beinhaltet als auch die Ausführung der Verzweigung voraussagt, beginnt möglichst früh die Bestimmung der Verzweigungsadresse, sobald der PC bekannt ist. Im anderen Fall wird mit Holen und sequentieller Ausführung weitergearbeitet. Bei falscher Voraussage erfolgt eine Modifikation der Voraussage-Bits.

Für die DLX-Pipeline ist dieses Schema direkt nicht geeignet, da dort gleichzeitig festgestellt wird, ob die Verzweigung ausgeführt wird und wo sich das Ziel befindet.

```
ADD R1, R2, R3

IF (R4 LE 1000) DO                  1. Verzweigung
INCR R4

ADD R6, R1, R6

ELSE

SUB R1, R4, R3


IF (R1 GE 0) DO                     2. Verzweigung

DECR R1

ADD R6, R7, R8
ELSE
CONTINUE
```

Abb. 9.9: Programmverzweigungen

An sich könnte man annehmen, daß Verzweigungen mit etwa 50% Häufigkeit auftreten. Praktische Programme zeigen allerdings, daß solche Verzweigungen typischerweise, bedingt durch Programmschleifen, mit einer höheren Wahrscheinlichkeit "gehäuft" auftreten. Daher ergibt sich bei einer Vorhersage der hier beschriebenen Art im Mittel eine Trefferquote von nahezu 90 %.

Damit sind also solche Vorhersagen, insbesondere für Prozessoren mit "tiefen" Pipelines, potentiell sehr nützlich.

Bei der DLX ist die Pipeline so aufgebaut, daß man am Ende der IF-Phase wissen muß, von welcher Adresse der nächste Befehl zu holen ist. Man muß also wissen, ob der noch nicht dekodierte Befehl eine Verzweigung beinhaltet, und welchen Wert ggf. der PC haben wird. Ist diese Information zu diesem Zeitpunkt bekannt, so kann man sogar eine Verzweige-Verzögerung von null haben.
Man kann also "vorsorglich" eine Tabelle anlegen, in der für eine Reihe von Verzweigungsbefehlen eines Programms jeweils mit diesem Befehl die nächste PC-Adresse angegeben wird und außerdem vermerkt wird, ob die Verzweigung als vorhergesagt zu gelten hat oder nicht. Eine solche Tabelle werden wir als "Verzweigezielpuffer" bezeichnen. Dieser Puffer wird im Cache-Speicher des Rechners angelegt (siehe Kapitel 10).

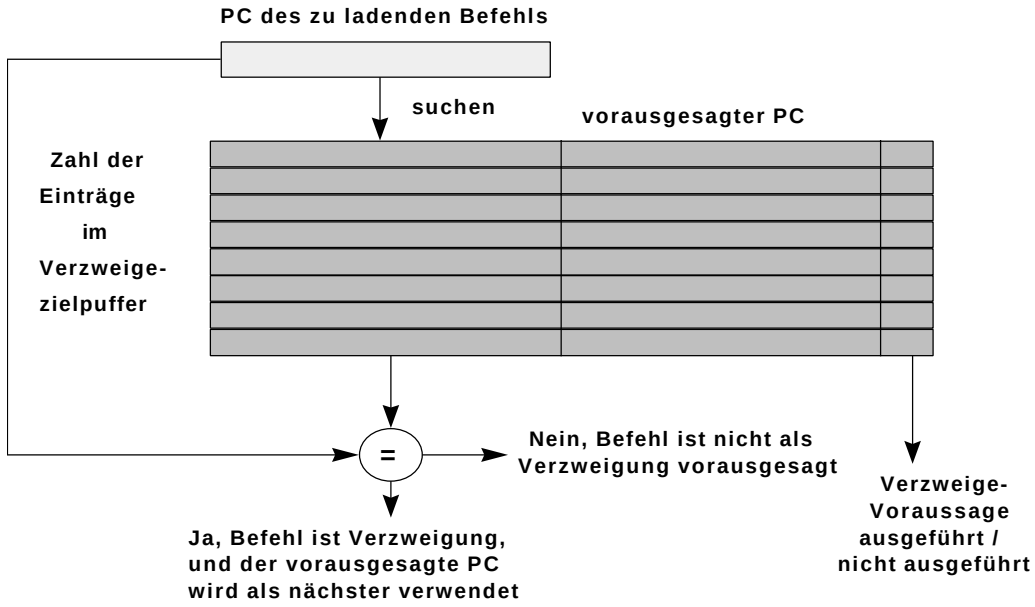

Abb. 9.10a: Verzweigungs-Zielpuffer

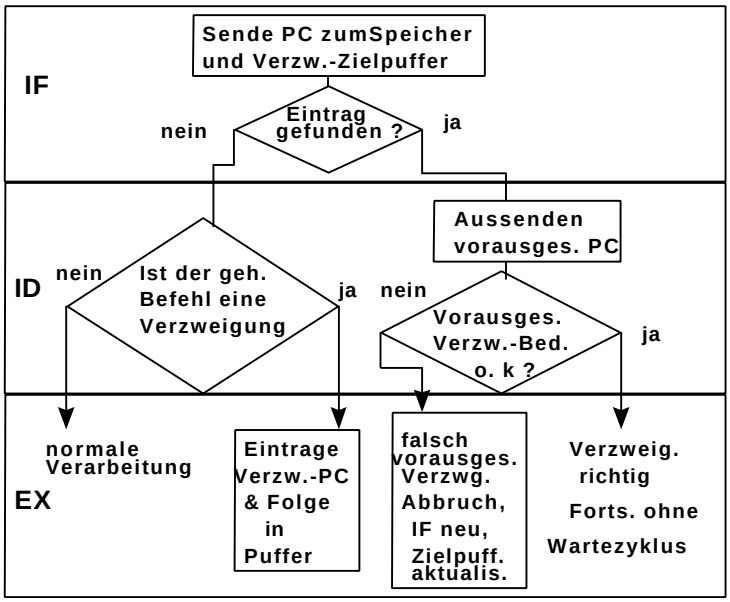

Abb. 9.10b: Befehlsabarbeitung bei DLX mit Verzweigungs-Zielpuffer

Damit kann man im Mittel in 80 bis 90% der Fälle Wartezyklen bei Verzweigungen vermeiden.

Tabelle 9.15: Verzögerungen bei Verzweigebefehlen

Befehle im Puffer	Voraussage	Aktuelle Verzweigung	Verlustzyklen
ja	ausgeführt	ausgeführt	0
ja	ausgeführt	n. ausgef.	2
ja	n. ausgef.	n. ausgef.	0
ja	n. ausgef.	ausgef.	2
nein		ausgef.	2
nein		n. ausgef.	1

Es ist nun nicht mehr sehr vernünftig, über dieses Maß hinaus die Verzweigungsbedingungen richtig vorhersagen zu wollen. Es ist jetzt effektiver, dafür zu sorgen, daß im Fall falscher Vorhersagen ein schneller neuer Lesevorgang möglich wird. Das kann z. B. bedeuten, daß man Register-Blöcke mit mehreren Load/Store-Einheiten verwendet, um Lese- und Schreiboperationen teilweise überlappend ausführen zu können. Auch die Verkürzung eines Lesevorganges aus dem Memory durch Einsatz schneller Zwischenspeicher bringt erhebliche Vorteile.

9.3 Superskalare Maschinen

9.3.1 Erweiterte Befehlsebenen-Parallelität

Auch in ihrer gepipelineten Form bleibt unsere DLX eine Maschine, die pro Taktzyklus nur einen Befehl übergibt. Für eine weitere Leistungssteigerung ist es deshalb der konsequente Schritt, den CPI (cycles per instruction, also Taktzyklen pro Befehl) auf einen Wert kleiner als eins zu drücken. Man wird dann pro Takt mehr als einen Befehl übergeben müssen.
Wie wir schon gesehen haben, erfordert bereits die Implementierung der Pipeline
ein ganzes Maßnahmen-Bündel, um die Abhängigkeit von Befehlen untereinander zu minimieren und mögliche vorhandene Parallelität optimal zu nutzen. Zwei abhängige Befehle müssen durch einen Abstand getrennt werden, welcher der Pipeline-Latenzzeit zum ersten der Befehle entspricht.
Tabelle 9.16 zeigt die Latenzzeiten für unterschiedliche Arten von Befehlen. Verzweigungen haben immer noch Latenzzeiten von einem Takt.

Zunächst kann man nun auf der Seite des Compilers einiges tun.
Eine beliebte Technik ist das Aufrollen von Programmschleifen (loop unrolling).
Wenn z. B. eine Programmschleife, die aus wenigen Statements besteht und z. B. dreimal oder fünfmal ausgeführt wird, vom Compiler in eine durchgängige Befehlsfolge umgesetzt wird, so ergeben sich gleich mehrere Vorteile:

Tabelle 9.16: Latenzzeiten typischer Operationen

Erzeugender Befehl für Ergebnis	auf Ergebnis zugreif. Befehl	Latenzzeit Zyklen
GK-ALU-Operation	GK-ALU-Operation	3
GK-ALU-Operation	Speichern doppelt	2
Laden doppelt	GK-ALU-Operation	1
Laden doppelt	Speichern doppelt	0

Zunächst muß die für Pipeline-Prozessoren relativ aufwendige Behandlung von Verzweigungen nicht mehr durchgeführt werden. Mit dem aufgerollten Code kann man dann auch noch in wesentlich verbessertem Maße ein statisches oder dynamisches Scheduling mit, wo notwendig, Umordnung von Befehlen durchführen.

Wenn alle dieser Möglichkeiten erschöpft sind, ist der nächste Schritt eine echte Parallelverarbeitung: Die Maschine bekommt zwei oder noch mehr Pipelines, welche jeweils parallel zueinander und in abgestimmter Arbeitsweise Befehle einlesen und abarbeiten können.

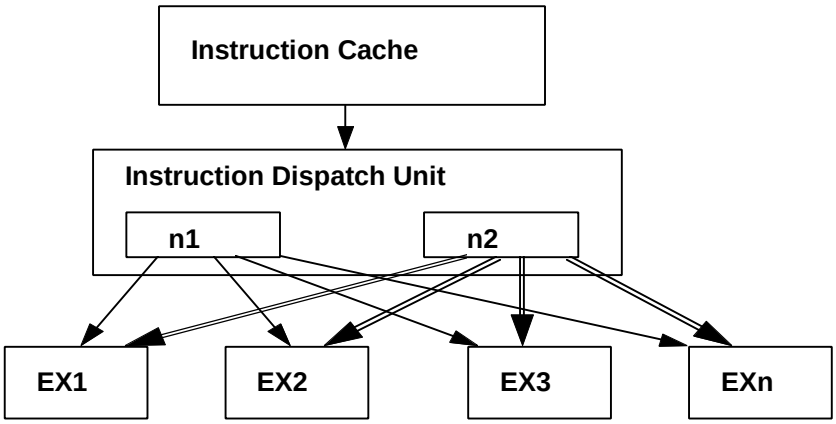

Abb. 9.11: Befehlsablauf bei einem superskalaren Prozessor mit zwei Pipelines

In den heute verwendeten Maschinen wird der Befehlsstrom von einem schnellen Zwischenspeicher (Instruction Cache) aus eingelesen. Dann muß eine "instruction dispatch unit (Befehls-Zuordnungseinheit) die Befehle bezüglich ihrer Art und ihrer Kompatibilität miteinander analysieren. Dann werden die Befehle den einzelnen Pipelines zur parallelen Bearbeitung zugeleitet. In der Regel sind zwei bis vier parallele Pipelines vorhanden (2 beim Pentium, 3 beim Pentium Pro, 4 beim IBM Power PC 604 und 620). Genauere Analysen normaler Befehlsströme haben ergeben, daß eine Parallelität von mehr als ca. 5 Prozessen sich kaum daraus ableiten läßt.
Der große Vorteil der Superskaleren Architektur ist die völlige Kompatibilität mit normalen Programmiersprachen und Compilern. Der Anwender merkt, bis auf die höhere Bearbeitungs-geschwindigkeit, bei gutartigen Programmen nichts von der superskalaren Natur des Prozessors.

9.3.2 Die Superskalare DLX-Maschine

Auch in ihrer gepipelineten Form bleibt unsere DLX eine Maschine, die pro Taktzyklus nur einen Befehl übergibt.
Rechner, die pro Taktzyklus mehr als einen Befehl übergeben, heißen "Superskalare" Maschinen. Dazu müssen die Befehle natürlich durch den Compiler geeignet "sortiert" werden.
In einer superskalaren Maschine kann also die Hardware eine kleine Zahl unabhängiger Befehle (meist 2 bis 4) in einem einzigen Takt übergeben.

Nehmen wir an, die DLX würde zur superskalaren Maschine mit 2 parallel abzuarbeitenden Befehlen erweitert. Es wäre zum Beispiel sinnvoll, wenn einer der Befehle eine Lade-, Speicher-, Verzweige-, oder Festkomma-ALU-Operation wäre, dagegen der andere Befehl eine Fließkomma (GK-) Operation.
Das ist viel einfacher als beliebige Doppel-Übergaben.

Die folgende Tabelle zeigt die so definierte Superskalare Pipeline in Funktion. Die Übergabe von zwei Befehlen je Zyklus wird für das Holen und Dekodieren jeweils 64 Bit erfordern. Um die Dekodierung zu vereinfachen, kann man zusätzlich festlegen, daß die Befehle paarweise im Speicher auf eine 64-Bit-Grenze ausgerichtet sind und der Festkomma (FK-) Teil zu Beginn steht.

Befehlstyp	**Pipeline-Stufe**							
FK-Befehl	IF	ID	EX	MEM	WB			
GK-Befehl	IF	ID	EX	MEM	WB			
FK-Befehl		IF	ID	EX	MEM	WB		
GK-Befehl		IF	ID	EX	MEM	WB		
FK-Befehl			IF	ID	EX	MEM	WB	
GK-Befehl			IF	ID	EX	MEM	WB	
FK-Befehl				IF	ID	EX	MEM	WB
GK-Befehl				IF	ID	EX	MEM	WB

Abb. 9.11: Superskalare Pipeline

Nicht dargestellt ist hier, wie die Gleitkomma-Operationen den EX-Zyklus erweitern. Es gibt jedoch hier keinen wesentlichen Unterschied zur gewöhnlichen DLX-Pipeline. Mit dieser Methode wird im wesentlichen die Abarbeitung von Gleitkomma-Operationen wesentlich beschleunigt. Natürlich müssen dazu eine Gleitkomma-fähige Pipeline-Einheiten oder mehrfache unabhängige Einheiten (Addiere, ALUs) realisiert werden. Ansonsten würden durch "besetzte" GK-Einheiten laufend Wartezyklen auftreten. Durch die parallele eines FK- und eines GK-Befehls wird der notwendige zusätzliche Aufwand minimiert, da die entsprechenden Befehle zumeist unterschiedliche Register und Funktionseinheiten benutzen.
Es ist allerdings möglich, daß auf der FK-Seite ein Befehl auftritt, der Lade-, Speicher-, oder Transportbefehle für Gleitkomma-Zahlen beinhaltet. Dies erzeugt dann Konflikte um die Ports der GK-Register-Ports. Die Lösung besteht in der Erkennung eines strukturellen Hazards und einer entsprechenden Verzögerung der Pipeline (FK-Seite) um einen Takt.

Eine andere Lösung ist die Bereitstellung je eines zusätzlichen Ports zum Lesen und Schreiben bei den Gleitkomma-Registern. Zur Auflösung von Struktur-Hazards müßten auch die vorstehend eingeführten Bypass-Einrichtungen doppelt ausgeführt werden.
Bei superskalaren Architekturen existieren weitere Optimierungsmöglichkeiten durch "Scheduling", also Umordnen der Befehle im Programmablauf schon durch den Compiler. Das Ausrollen von Schleifen (loop unrolling) gehört dazu, aber außerdem auch die Bereitstellung nebenläufig abarbeitbarer GK- und FK- Programmteile.
Für viele Aufgaben, in den meisten Fällen sogar die absolute Mehrzahl der Aufgaben, wird die FK-Verarbeitung deutlich wichtiger als die GK-Verarbeitung sein. Deshalb ist die Parallelisierung GK / FK, wie hier im Beispiel der DLX-Maschine vorgestellt, nicht unbedingt ein brauchbares Modell für die Praxis.

9.3.3 Pentium-Prozessor

Pipelines sind seit Jahren "Stand der Technik" auch bei High-End-Mikroprozessoren.
In der Serie der Intel-Prozessoren besitzt schon der 80286 eine Pipeline, in verbesserter Form auch alle Nachfolgemodelle. Allerdings ist der Pentium der erste Prozessor, der superskalar organisiert ist. Er besitzt 2 Pipelines.

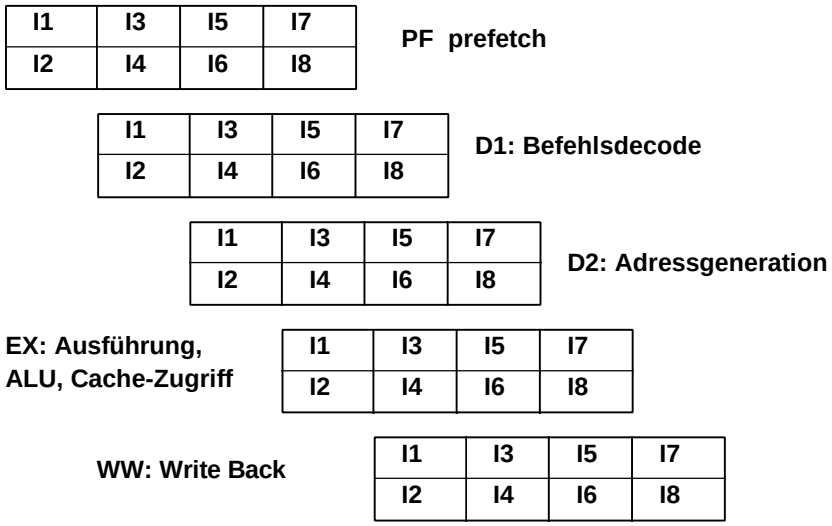

Abb. 9.12: Pipeline-Struktur beim Pentium Prozessor

Die Pipelines werden als U- und V-Pipeline bezeichnet. Die U-Pipeline unterstützt den vollen 80x86-Befehlssatz, während die V-Pipeline nur einen einfachen Befehlssatz kann. Damit ist der Pentium nur dann eine voll superskalere Maschine, wenn er quasi als RISC-Prozessor eingesetzt wird.

Die Bedingungen für das Paaren von zwei Befehlen sind:

- beide Befehle sind vom "einfachen" Typ
- es existiert keine Abhängigkeit zwischen den Befehlen bezüglich des Zugriffs auf dieselben Register
- kein Befehl darf ein Displacement oder ein Immediate enthalten
- komplexe 80x86-Befehle (mit Präfix) können nur in der U-Pipeline behandelt werden

Die meisten Befehle werden innerhalb eines Taktzyklus ausgeführt, eine Mikroprogrammsteuerung entfällt. Auch Verzweigungsbefehle können innerhalb eines Befehlspaars auftreten, müssen aber dann als 2. Befehl stehen. In der Ausführung folgt der Befehl der V-Pipe der U-Pipe. Ein Stall der einen Pipeline wirkt sich auch auf die andere entsprechend aus.

Die Befehlsschritte sind:

- Prefetch: Zugriff zum Befehlscache (oder zum Hauptspeicher)

Dazu stehen 2 Prefetch-Buffer zur Verfügung. Während der eine Befehls lädt, kann der andere auf das Ergebnis der Analyse des Verzweigungs-Zielpuffers warten.

- D1: 2 Dekodiereinheiten bearbeiten 2 neue Befehle. Es erfolgt hier die Analyse auf Kompatibilität. Ist ein komplexer Befehl mit Präfix vorhanden, so erfordert dieser einen Extra-Takt zur Dekodierung.

- D2: Adressberechnung für Operanden, in einem Takt

- EX: Ausführungsschritt für arithmetische, logische Operationen und Cache-Zugriffe. Es sind auch Ausführungen mit mehreren Takten möglich.

- WB: Aktivierung aller Befehle, die den Prozessor-Status ändern.

Die Pipes U und V beginnen und verlassen D1, D2 gleichzeitig. Bei Verzögerungen auf einer Pipeline ist die andere gleichermaßen betroffen. U und V beginnen auch EX gleichzeitig. Auch hier wirken sich Verzögerungen gegenseitig aus. Wird U verzögert, so "stallt" auch V. Verzögert sich V in EX, so wird der mit ihm gepaarte U-Befehl vorzeitig ausgeführt. Nachfolgende Befehle müssen warten, bis in beiden Pipelines die Befehle bis zum WB vorgerückt sind.

Von besonderer Bedeutung ist beim Pentium eine sogenannte "Prefetch Unit".
Im normalen Betrieb wird mit nachrangiger Priorität ein paar von 32 Bit-Prefetch-Buffern geladen. Sie arbeiten mit dem Branch Target Puffer zusammen. Befehle werden zunächst so lange kontinuierlich geladen und den Pipelines zugeführt, bis ein Sprung- oder Verzweigungsbefehl auftritt. Dann bestimmt der Vergleich mit dem Verzweigungspuffer (BTB) den weiteren Befehlsablauf. Ohne Verzweigung wird der Ablauf normal fortgesetzt, wenn ja, wird der erste Puffer dafür benötigt und ist ausgelastet. Der Prefetch-Vorgang wird dann durch den zweiten Puffer übernommen,
Bei Feststellung einer falschen Verzweigungs-Voraussage wird die Pipeline zurückgesetzt und neu geladen.

Die Fließkomma-Einheit des Pentium ist mit der Integer-Einheit integriert. Diese Einheit ist selbst wieder in 8 Stufen gepipelint. Von diesen 8 Stufen werden für Fixkomma-Operationen nur 5 benötigt, es wird dann ein "bypass" eingefügt.

Die 8 Phasen sind:

- PF prefetch
- D1 Befehlsdekodierung
- D2 Adresserzeugung
- EX Lesen von Speichern und Registern, Datenwandlung vom Fließkomma-Format in externe Formate
- X1 Ausführungsschritt 1: Umwandlung Speicher-Formate in Fließkomma-Format, Ablegen in Fließkomma-Registern, Bypass 1
- X2 Ausführungsschritt 2
- WF Rundungen, Ablegen in Register-Dateien, Bypass 2
- ER Fehlerbeseitigung, Aktualisierung des Status-Wortes

Für die Befehle der Fließkomma-Einheit ist eine paarweise Bearbeitung möglich.
Fließkomma-Befehle können nicht mit Festkomma-Befehlen gepaart werden, es gibt aber verschieden miteinander paarweise verarbeitbare Festkomma-Befehle.

9.4 VLIW-Prozessoren

Die superskalare Maschine ließe sich praktisch bis auf die gleichzeitige Übergabe von drei oder vier Befehlen erweitern. Hier wird es aber schon sehr schwierig, die Abstimmung zwischen diesen Befehlen vorzunehmen, also z. B. Abhängigkeiten der Befehle voneinander zu erkennen und den Ablauf entsprechend zu organisieren.
Eine Alternative ist eine LIW (long instruction word) oder VLIW (very long instruction word)-Architektur.
Solche VLIWs benutzen mehrfache unabhängige Funktionseinheiten.
Anstatt nun zu versuchen, diesen Einheiten mehrere unabhängige Befehle parallel zu geben, packen VLIW-Maschinen mehrere Operationen in einen sehr langen Befehl.
Solch ein VLIW-Befehl kann dann z. B. zwei FK-Operationen, zwei GK-Operationen, zwei Speicherzugriffe und eine Verzweigung einschließen.
Ein einziger Befehl würde dann einen Satz von Feldern für jede Operation haben mit etwa 16 bis 24 Bit pro Feld. Damit ergeben sich dann Befehlslängen von 128 Bit bis zu 168 Bit.

Auch hier ist die Leistungsfähigkeit der Maschine wieder stark von der Qualität des Kompilers abhängig. Er muß, z. B. durch das Aufrollen von Programmschleifen, ausreichend lange Programmabschnitte erzeugen.
Es wird sogar das Scheduling von Befehlen über die Grenzen von Programm-Blöcken hinweg vorgenommen. Die dort verwendete Methode wird als Trace-Scheduling bezeichnet.

Als Beispiel sein eine VLIW-Architektur angenommen, die zwei Speicherzugriffe, zwei Gleitkomma-Operationen und eine FK-Operation oder Verzweigung in jedem Taktzyklus abarbeiten kann.

Speicherzugr.1	Speicherzugr.2	GK-Operat. 1	GK-Operat. 2	FK-Op.
LD F0, O(R1)	LD F6,-8(R1)			
LD F10, -16(R1)	LD F14,-24(R1)			
LD F18, -32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2	
LD F26, -48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2	
		ADDD F20,F18,F2	ADDD F24,F22,F2	
SD O(R1), F4	SD -8(R1), F8	ADDD F28,F26,F2		
SD -16(R1),F12	SD -24(R1),F16			
SD -32(R1),F20	SD -40(R1),F24			SUB R1,R1,#48
SD -0(R1), F28				BNEZ R1, LOOP

Abb. 9.13: Befehlsverarbeitung bei VLIW-Maschinen

Eine Programmschleife ist 6-fach aufgerollt. Damit sind Wartezyklen bei der Verzweigung eliminiert. Der Code läuft in 9 Zyklen ab. Die Übergaberate beträgt 23 Operationen in 9 Zyklen oder 2,5 Operationen pro Zyklus. Die Effektivität beträgt etwa 60%. Dabei ist eine viel größere Registerzahl erforderlich, als die DLX sie normalerweise benötigen würde.

Es ist möglich, VLIW-Maschinen zu bauen, die mehr als die hier gezeigten 5 Operationen gleichzeitig ausführen können.
Grenzen sind durch folgende Grenzen gegeben:

- begrenzte Parallelität im Code
- begrenzte Hardware-Ressourcen
- begrenzte Code-Längendehnung

Selbst wenn Schleifen weitgehend aufgerollt werden, ergibt sich immer nur eine begrenzte Anzahl sinnvoll parallel abarbeitbarer Befehle.
Auf den ersten Blick erscheint im obigen Beispiel die VLIW-Maschine recht gut ausgelastet. Wenn aber die wichtigen Funktionseinheiten selbst wieder als Pipeline-Einheiten ausgeführt sind, braucht man auch noch sinnvoll Ketten von Operationen zur Auslastung der Maschine.
Als Faustregel kann gelten, daß man eine Anzahl unabhängiger Operationen braucht, um eine solche Maschine sinnvoll zu beschäftigen, die sich aus der Anzahl der parallel verfügbaren Funktionseinheiten, multipliziert mir der durchschnittlichen Pipeline-Tiefe ergibt.
Um eine VLIW-Maschine mit z. B. 5 Funktionseinheiten sinnvoll zu beschäftigen, benötigt man etwa 15 bis 20 Operationen.

Das zweite Problem ergab sich aus den steigenden Hardware-Kosten. Bei Prozessor-Implementierung in Großintegration ist das Duplizieren arithmetischer und logischer Einheiten kein zu großes Problem.

Es eine große Zunahme in der Breite von Registern und Speichern, die aber wie der Aufwand für die Funktionseinheiten nur linear zunimmt. Man benötigt aber zunehmend mehr Schnittstellen (ports) an GK- und FK-Registerbänken (für unser DLX-Beispiel 5 Leseports und 2 Schreibports am FK-Registerfile, 4 Lese- und 2 Schreibports am GK-Registerfile). Gerade die zunehmende Anzahl von Ports und die damit verbundene Notwendigkeit zu mehr Schaltern und Verbindungen bringt aber mehr als linear steigenden Aufwand.
Das Problem ist also hier das Speichersystem, das zunehmend komplexer und damit auch potentiell langsamer wird.

Schließlich erfordert die VLIW-Architektur in hohem Maße das Aufrollen von Schleifen, um sinnvoll arbeiten zu können. Das wiederum benötigt Speicherplatz. Um den Speicherplatz im Hauptspeicher nicht zu sehr zu strapazieren, gibt es die Methode, Befehle im Hauptspeicher kompaktiert abzulegen und die Expandierung erst bei der Übertragung in den Cache vorzunehmen.
Hier stößt man aber an Grenzen, welche die Frage rechtfertigen, ob nicht eine echt parallele Maschine und hier insbesondere ein Vektorprozessor die bessere Alternative darstellt.

9.5 Vektorprozessoren

9.5.1 Weshalb Vektormaschinen?

In diesem Kapitel haben wir die Methode des Pipelining im Detail betrachtet. Mittels der Kombination von Compiler-Techniken und Hardware-Pipelines kann man die Leistung einer Maschine etwa verdoppeln. Diese Technik hat in 2 Hinsichten Grenzen:

1. Taktzykluszeiten werden tendentiell bei Maschinen mit tiefen Pipelines langsamer.

2. Befehlshole- und Befehlsdekodierungsrate sind begrenzt. Man schafft es in der Regel nicht, mehr als nur ein paar Befehle pro Takt zu holen und zu übergeben. Bei den meisten Pipeline-Maschinen ist die durchschnittliche Zahl von Befehlen pro Takt sogar kleiner als eins.

Tiefes Pipelining und Übergabe mehrerer Befehle gleichzeitig bringen erhebliche "Scheduling"-Probleme wegen der Abhängigkeit der Befehle voneinander. Hier ergibt sich also insgesamt ein Flaschenhals.

Vektormaschinen sind eine echte Erweiterung. Sie verfügen über Operationen auf einem höheren Niveau, die Vektoren, also lineare Zahlenfelder, verarbeiten.

Eine typische Vektor-Operation wäre die gleichzeitige Addition von 2 Gleitkomma-Vektoren mit jeweils 64 Elementen, das Ergebnis ist ein Vektor mit ebenfalls 64 Elementen.

Vektor-Operationen haben mehrere bedeutende Eigenschaften, welche den obigen "Bottleneck" umgehen helfen:

- Die Berechnung von Ergebnissen ist unabhängig von vorangegangenen Ergebnissen. Damit ist ein sehr tiefes Pipelining ohne Daten-Hazards möglich. Vorausgegangen ist die Entscheidung des Programmierers oder des Compilers, daß ein Vektor-Befehl benutzt werden soll.

- Ein einziger Vekorbefehl spezifiziert einen ganzen komplexen Vorgang, vergleichbar mit der Ausführung einer ganzen Schleife. Damit ist der Befehlsstrom insgesamt reduziert. Der auch als "Flynnscher Flaschenhals" bezeichnete Engpaß beim Einlesen und Dekodieren von Befehlen wirkt sich weniger stark aus.

- Zum Speicher zugreifende Vektorbefehle haben ein bekanntes Zugriffsmuster. Wenn die Vektor-Elemente alle angrenzend abgespeichert sind, dann funktioniert das Holen und Laden sehr effizient. Beim Zugriff auf den Hauptspeicher wird der Zeitaufwand für den Zugriff pro Vektor nur einmal notwendig, statt mehrfach nacheinander wie bei skalaren Maschinen.

– Eine ganze Programmschleife, die ja für die Prüfung der Verzweigungsbedingung mehrfache Problem für das Pipelining brachte, wird durch einen Vektor-Befehl ersetzt.

Vektor-Befehle können also schneller gemacht werden als äquivalente Folgen von skalaren Befehlen es sein könnten.
Die Abarbeitung der einzelnen Elemente eines Vektors erfolgt wieder unter Anwendung von Pipeline-Operationen. Die Pipeline schließt arithmetische Operationen (Addition, Multiplikation) ein, aber auch Berechnungen von Effektiv-Adressen und Speicherzugriffe.
Während ein "einfacher" Vektor-Rechner dem Prinzip "single instruction- multiple data" entspricht, gibt es natürlich auch Erweiterungen in Richtung auf "multiple instruction - multiple data".
Die meisten Hochleistungsmaschinen erlauben sogar die parallele Ausführung mehrerer Vektor-Operationen.

9.5.2 Eine Basis-Vektormaschine

Die typische Vektormaschine besteht aus einer gewöhnlichen Pipeline-Skalareinheit, ergänzt um die Vektoreinheit.
Die Funktionseinheiten innerhalb der Vektoreinheit werden in der Regel mehr als einen Taktzyklus für die jweilige Operation benötigen. Dies ermöglicht auf der andren Seite kurze Taktzykluszeiten. Langlaufende Vektor-Operationen sind möglich, welche nicht direkt zu Hazards führen.

Man unterscheidet zwei grundlegende Typen von Vektor-Architekturen:
Vektor-Registermaschinen und Speicher-Speicher-Vektormaschinen.
Ganz analog zu den bisher betrachteten skalaren Maschinen werden bei Vektor-Registermaschinen alle Vektoroperationen bis auf Laden und Speichern nur zwischen Registern durchgeführt. Zu dieser Gruppe gehören z. B. die Cray-Maschinen CRAY-1, CRAY-2, X-MP und Y-MP sowie die Supercomputer japanischer Hersteller (NEC SX/2, Fujitsu VP200, Hitachi S820).
Speicher-Speicher-Vektormaschinen werden heute nicht mehr gebaut. Von diesem Typ waren die CDC-Superrechner aus den 70er und 80er Jahren.

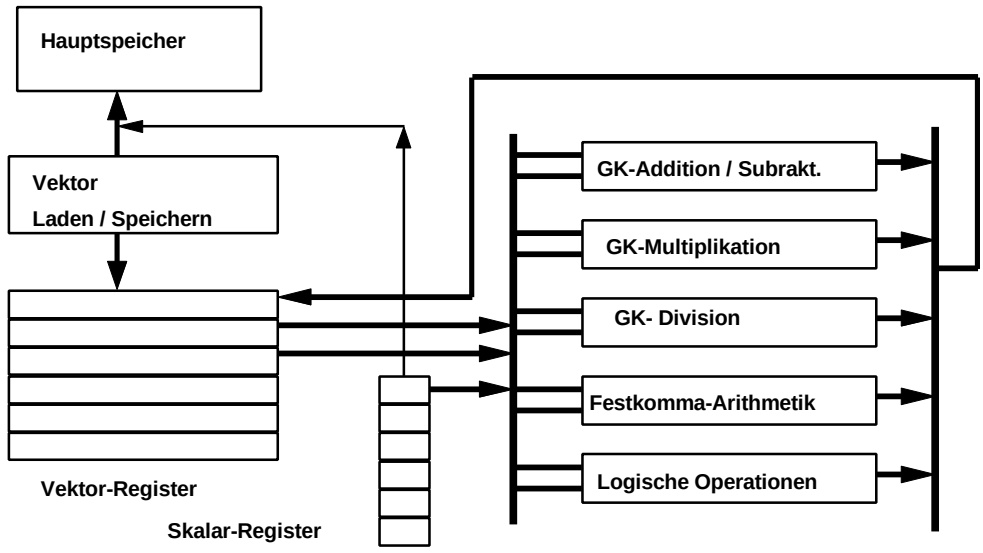

Abb. 9. 14: Aufbau einer Vektor-Registerarchitektur DLXV

Der prinzipielle Aufbau einer Vektor-Registerarchitektur für die DLX ist in Abb. 9.14 angegeben. Sie ist in ihrem Aufbau an die CRAY-1 angelehnt und soll nachfolgend als DLXV bezeichnet werden.
Ihr FK-Teil entspricht der (gepipelined) DLX, der Vektor-Teil ist eine logische Vektor-Erweiterung für die DLX.

Die Hauptkomponenten lassen sich wie folgt charakterisieren:

Vektor-Register:
Jedes Vektor-Register hat eine Bank fester Länge, die einen einzelnen Vektor beinhaltet. Die DLXV soll 8 Vektor-Register haben, jedes Vektor-Register besteht aus 64 Doppelworten. Dabei hat jedes Vektorregister mindesten 2 Lese- und 2 Schreibports. Dies ist notwendig, um in hohem Maße Datenaustausch zwischen den Vektor-Registern parallel ausführen zu können.

Vektor-Funktionseinheiten:
Jede Vektor-Funktionseinheit ist eine vollständige Pipeline und kann eine neue Operation in jedem Taktzyklus starten. Für die Erkennung der unterschiedlichen möglichen Pipeline-Hazards ist eine spezielle Steuereinheit erforderlich, die sowohl Struktur-Hazards (bei Konflikten bezüglich des Zugriffs auf funktionelle Einheiten) als auch Daten-Hazards (bei Register-Zugriffen) erkennt. Die DLXV hat (Abb. 8. 19) fünf Funktionseinheiten für Vektor-Operationen.

Vektor Lade / Speicher-Einheit:
Eine Vektor-Speichereinheit dient als Zwischenspeicher für den Transport von Vektoren von und zum Hauptspeicher. Die zusätzlichen Load/Store-Befehle für die DLXV sind wirken auf vollständige Pipeline. Es können nach einer anfänglichen Latenzzeit ein Datenverkehr zwischen Vektorregistern und Speicher erfolgen, der ein Wort pro Taktzyklus überträgt.

Ein Satz von Skalarregistern:
Natürlich benötigt eine Vektor-Maschine immer auch einen Registersatz für skalare Operationen. Er wird zum Beispiel für die Berechnung von Adressen benötigt, welche die Vektor Load/Store-Einheiten benötigen. Er kann aber auch die Eingangsdaten für Vektor-Einheiten bereitstellen.
Dieser Satz umfaßt die "normalen" 32 Universalregister und die 32 Gleitkomma-Register der DLX.

Die DLXV verfügt gegenüber der DLX natürlich auch über einen erweiterten Befehlssatz. Sinnvollerweise nimmt man den Befehlssatz der DLX und hängt für die bezüglich Vektor-Operationen ein V an. Betroffen sind davon die doppelt genauen Gleitkomma-Operationen.

Für einen solchen Befehl werden die Eingangsdaten dann entweder von 2 Vektor-Registern oder von einem Vektor- und einem Skalarregister geliefert. Bei letzteren Befehlen wird ein "SV" als Kennzeichnung angehängt. Beispielsweise macht es Sinn, alle Elemente eines Vektor mit demselben skalaren Faktor zu multiplizieren. Vektor-Operationen haben aber immer ein Vektor-Register als Zielregister. Im DLXV-Befehlssatz stehen LV und SV stehen für "Load Vector "und "Store Vector".

Es wird jeweils ein ganzer Vektor doppelt genauer Gleitkomma-Daten gleichzeitig vom Speicher in ein Vektor-Register geladen bzw. vom Vektor-Register in den Speicher geladen. Ein Operand ist jeweils das zu benutzende Vektor-Register, der andere Operand ist ein DLX-Universalregister, dessen Inhalt die Startadresse des Vektors im Speicher angibt.

Vektor-Befehl	**Operanden**	**Funktion**
ADDV	V1, V2, V3	Add. die Elem. von V2 und V3, speichere in V1
ADDSV	V1, F0, V2	Add. F0 zu jed. Elem. von V2, speichere in V1
SUBV	V1, V2, V3	Subtr. elementweise V3 von V2, speich. in V1
SUBVS	V1, V2, F0	Subtr. F0 von jed. Elem. von V2, speichere in V1
SUBSV	V1, F0, V2	Subtr. V2-Elem. von F0, speichere in V1
MULTV	V1, V2, V3	Multipl. V2- und V3-Elemente, Ergebn. in V1

MULTSV	V1, F0, V2	Multipl. F0 m. d. Elem. von V2, Erg. in V1
DIVV	V1, V2, V3	Division d. E. von V2 d. E. von V3, Ergebn. in V1
DIVVS	V1, V2, F0	Division d. E. von V2 d. F0, Ergebn. in V1
DIVSV	V1, F0, V2	Division von F0 d. Elem. v. V2, Ergebn. in V1
LV	V1, R1	Laden Vekt.-Reg. V1 vom Speich., beg. Adr. R1
SV	R1, V1	Speich. Vekt. Reg. V1, beg. b. Adresse in R1
LVWS	V1, (R1, R2)	Lad. V1 v. Speich., beg. Adr. R1, Schrittw. R2 (z. B. Adressen mit R1 + i + R2), i=0,1,2 etc.
SVWS	(R1, R2), V1	Speich. V1 ab Adr. aus R1, Schrittw. R2
LVI	V1, (R1 + V2)	Lad. V1 mit einem Vektor, dess. E. R1 + V2(i) sd.
SVI	(R1 + V2), V1	Speich. V1 mit e. Vekt., dess. Elem. R1+V2(i) sind
CVI	V1, R1	Erzeug. eines Ind. Vekt. durch Speich. der Werte 0, 1*R1, 2*R1, ...63*R1 in V1
S_V	V1, V2	Vergl. (EQ, NE, GT, LT, GE, LE) d. El. in V1 mit V2. Wenn die Bedg. wahr ist, Speicherg. einer 1 in einem zugeordneten Bit-Vektor, sonst 0. Speich.
S_SV	F0, V1	d. result. Bit-Vekt. im Vekt.-Maskenreg. (VM) Bei S_SV wird ein Skalarwert als Operd. ben.
POP	R1, VM	Zählen der Einsen im Vekt.-Maskenregister und und Speichern von deren Anzahl in R1
CVM		Setzen des Vekt.-Maskenreg. auf 1 in allen Posit.
MOVI2S	VLR, R1	Übetr. des Inh. von R1 z. Vekt.-Längenregister
MOVS2I	R1, VLR	Übertr. d. Inh. v. Vekt.-Längenreg. zu R1
MOVF2S	VM, F0	Übetr. d. Inh. von F0 z. Vekt.-Maskenregister
MOVS2F	F0, VM	Übetr. d. Inh. d. Vekt.-Maskenreg. zu F0

Die hier implizit in die Architektur eingeführten zusätzlichen Register "Vektor-Längenregister" (VLR) und Vektor-Maskenregister (VM) werden notwendig, wenn Vektoren eine andere als die "genormte" Länge haben und wenn Vektoren nicht kompakt und durchgehend, sondern verteilt im Speicher abgelegt sind.

9.5.3 Beispiel für die Funktion einer Vektor-Maschine

Eine Vektormaschine ist am besten zu verstehen, wenn man sich einen typischen Anwendungsfall anschaut. Als typisches Vektor-Problem kann gelten:

Y = a * X + Y

Dabei sind Y und X Vektoren, a ist ein skalarer Faktor. Soll ein solcher Befehl skalar ausgeführt werden, so sind sogenannte SAXPY (single-precision a*X + Y) oder DAXPY (double-precision a*X + Y) -Operationen. Sie bilden den Kern von Standard-Programmen (benchmarks), mit deren Hilfe die Leistung von Rechnern gemessen wird.

Wir betrachten die DLX- Befehlsfolge im Vergleich zur DLXV-Befehlsfolge für den Anwendungsfall der DAXPY-Schleife.

Die DLX-Befehlsfolge lautet:

```
LD      F0, a

ADDI    R4, Rx,#512;    Laden ab letzter Adresse
loop:
LD      F2, 0(Rx);      Laden X(i)
MULTD   F2,F0, F2;      a*X(i)
LD      F4,0(Ry);       Laden Y(i)
ADDD    F4,F2,F4;       a*X(i) + Y (i)
SD      F4, 0(Ry);      Speichern zu Y(i)
ADDI    Rx,Rx, #8;      Inkrementierung Index für X
ADDI    Ry,Ry, #8;      Inkrementierung Index für Y
SUB     R20,R4,Rx;      Dekrementierg. d. Schleifen.-Index
BNZ     R20,loop;       Test auf letzten Schleifendurchlauf
```

Die Befehlsfolge für die Vektor-Maschine DLXV würde entsprechend lauten:

```
LD      F0, a;          Laden des Skalars a
LV      V1, Rx;         Laden des Vektors X
MULTSV  V2,F0,V1;       Vektor-Skalar-Multiplikation
LV      V3,Ry;          Laden Vektor Y
ADDV V4,V2,V3;          Vektoraddition
SV      Ry, V4;         Speichern Ergebnisvektor
```

Die Vektor-Maschine kommt hier mit 6 Befehlen im Unterschied von fast 600 bei der DLX aus. Demnach wird der Overhead für das Laden und Dekodieren von Befehlen ganz wesentlich reduziert. Die Overhead-Befehle, beispielsweise die für das Setzen der Indices in Datenfeldern, werden bei der Vektor-Maschine nicht benötigt. Ein weiterer Unterschied betrifft die Pipeline: In der normalen DLX muß jeder ADDD-Befehl auf einen MULTD warten, und jeder SD muß auf einen ADDD warten. Bei der Vektor-Maschine treten zwar auch Wartevorgänge auf, aber nur genau einmal pro Vektor-Operation, die skalare Maschine wartet einmal pro Vektor-Element. Geht man von Vektoren mit 64 Feldern aus (die alle benutzt werden), so ist die Häufigkeit des Wartevorgangs bei der DLX 64 mal höher als bei der DLXV.

9.5.4 Vektor-Anlaufzeit, Initiierungsrate

Als spezielle Pipeline-Maschine kann ein Vektorrechner, hier z. B. die DLXV, nicht jeweils "aus dem Stand" anlaufen.
Eine Anlaufzeit (start-up-time) wird benötigt, um die Pipeline für die Vektor-Operation zu laden. Hier ist die Tiefe der Pipeline bestimmend für den Zeitaufwand. Beispielsweise bedeutet eine Latenzzeit von 10 Taktzyklen, daß die Operation 10 Taktzyklen erfordert und die Pipeline die Tiefe 10 aufweist.
Die Initiierungsrate ist die Zeit pro Ergebnis, wenn der Vektorbefehl läuft.

Diese Initiierungsrate ist gewöhnlich 1 pro Taktzyklus. Es gibt aber durchaus Superrechner, die 2 oder mehr Ergebnisse pro Takt erzeugen können. Andere Rechner haben Einheiten, die nicht vollständig in Pipelines ausgeführt sind.
Ein Vektorrechner und allgemeiner eine Pipeline-Maschine hat eine Abschlußrate (completion rate), die mindestens gleich der Initialisierungsrate sein muß. Sonst "verstopft" die Pipeline, es können keine Zwischenergebnisse mehr abgelegt werden.
Die Zeit für den Anschluß einer einzigen Vektor-Operation derLänge n kann angegeben werden zu:

Abschlußzeit = Anlaufzeit + n * Initiierungsrate

Die Einflüße auf die Anlaufzeit und die Initiierungszeit sind komplex.

Für Register-Register-Operationen ist die Anlaufzeit (in Taktzyklen) gleich der Pipeline-Tiefe der Funktionseinheit. Eine Pipeline-Tiefe von 10 bedeutet also, daß erste Ergebnisse frühestens nach 10 Taktzyklen verfügbar sind.

Die Initiierungsrate ist davon bestimmt, wie oft die erforderliche Vektor-Funktionseinheit einen Operanden aufnehmen kann. Wenn die Initiierungsrate einen Takt pro Ergebnis beträgt, dann gilt:

$$\text{Pipeline - Tiefe} = \frac{\text{Gesamtzeit der Funktionseinheiten}}{\text{Taktzykluszeit}}$$

Wenn also z. B. eine Operation 10 Taktzyklen erfordert, dann muß die Pipeline-Tiefe 10 betragen, um eine Initiierungsrate von eins zu erreichen.

Die tatsächliche Pipeline-Tiefe von Funktionseinheiten variierte in einem weiten Bereich, etwa 2 bis 20 Stufen sind realistisch. Die am häufigsten genutzten Einheiten haben ca. 4 bis 8 Stufen.

Für die DLXV wurde eine Pipeline-Tiefe von 6 bis 7 gewählt, wie sie z. B. auch die Cray-1 hat. Für die Gleitkomma-Addition werden 6 Taktzyklen, für die Multiplikation 7 benötigt. Alle Funktionseinheiten haben ein vollständige Pipeline.
Wenn eine Vektorberechnung von einer vorherigen, noch nicht abgeschlossenen Berechnung abhängt, so muß sie angehalten und ein Wartezyklus eingeführt werden. Dann ergeben sich 4 zusätzliche Taktzyklen als Startverzögerung.
Verzögerungen dieser Art sind typische für Vektor-Maschinen, da in vielen Fällen keine Bypaß-Hardware zur Verfügung steht.

Probleme bereiten also abhängige Vektor-Operationen, da sie Zeit für das Schreiben und Lesen von Operatoren benötigen. Abhängige aufeinanderfolgende Vektor-Operationen haben die volle Latenzzeit einzelner Vektor-Operationen.

Dagegen können unabhängige Vektor-Operationen, die verschiedene Funktionseinheiten nutzen, ohne Verlust oder Verzögerung übergeben werden.
Von besonderer Komplexität ist das Lade- / Speicherverhalten von Vektoreinheiten.
Die Anlaufzeit für einen Lade-Speicher-Befehl ist die Zeit, die benötigt wird, um das erste Wort vom Speicher in ein Register zu laden. Wenn dann der Rest des Vektors ohne Wartezyklen bereitgestellt werden kann, dann ist die Vektor-Initiierungsrate gleich der Rate, mit der neue Worte geholt und gespeichert werden können.
Es ist für Vektorrechner ganz typisch, daß die Verzögerungen der Startphase von Lade-/Speichereinheiten höher sind als die von Funktionseinheiten. Bei einigen Maschinen sind bis zu 50 Taktzyklen erforderlich.

Für die DLXV kann man mit einer Anlaufzeit von ca. 12 Taktzyklen auskommen, die Vorbildmaschinen Cray-1 und Cray-XMP haben Verzögerungen zwischen 9 und 17 Taktzyklen.
Bei Spiecherbefehlen wird man keine Anlaufzeit definieren können, da sie ja in dem Sinn keine Ergebnisse erzeugen. Wenn aber ein arithmetischer oder logischer Befehl auf den Abschluß eines Speicherbefehls warten muß, ergeben sich entsprechend lange Latenzzeiten.

Die nachfolgende Tabelle zeigt die Startverzögerungen (in Taktzyklen) bei der DLXV für Vektor-Operationen.

Operation	**Startverzögerung**
Vektor-Addition	6
Vektor-Multiplikation	7
Vektor-Division	20
Vektorladen	12

Wenn man für den "laufenden" Vektor-Betrieb das Holen und Speichern eines Wortes pro Takt erreichen will, so benötigt auch das Speichersystem eine besondere Organisation. Dies wird erreicht durch sogenannte "Speicherbanken". Jede Speicherbank wirkt wie ein eigener kleiner Speicher, der zu verschiedenen Adressen parallel zu anderen Banken zugreifen kann.

Die Organisation des Speichersystems muß auf die Organisation der Befehlsverarbeitung abgestimmt sein. Die Anzahl der Banken im Speichersystem muß auf die Pipeline-Tiefe abgestimmt sein, da die Speicherbänke die Initiierungsrate für Operationen bestimmen, welche die Funktionseinheiten nutzen. Die Zykluszeit des Speichers kann ein wesentlicher Engpaß sein. Bei DRAMs z. B. ist die Zykluszeit etwa das Doppelte der Zugriffszeit. Dann benötigt der Prozessor eine entsprechend doppelt so hohe Anzahl von Banken wie im Fall schneller Speicherbausteine (z. B. SRAMs).

9.5.5 Weitere Probleme der Vektor-Technologie

Die "Feinheiten" der Technologie der Vektorrechner sollen hier nicht im Detail diskutiert werden, da diese Maschinen in der heutigen Praxis eher Exoten darstellen.

Probleme gibt es dann, wenn Vektoren nicht die Standard-Länge besitzen oder im Speicher nicht zusammenhängend abgelegt sind. Deshalb sind spezielle Register notwendig, um Vektorlängen zu steuern.

Vektorrechner benötigen eine spezielle Compiler-Technologie, welche arithmetische Probleme erst "aufbereitet". Trotzdem ist oft noch eine spezielle Programmierung erforderlich.

Vektor-Maschinen waren und wurden als "Number-Cruncher" für sehr rechenaufwendige Programme wie Simulation (Aerodynamik, Hydodynamik, Wettervorhersage) eingesetzt.
Bei Programmen, welche im wesentlichen mit logischen Werten oder mit Festkomma-Zahlen arbeiten, bleibt ihr Nutzen eher beschränkt.
Sie haben heute starke Konkurrenz durch massiv parallele Maschinen, die aus einer Vielzahl von Mikroprozessoren aufgebaut sind.

Während Vektorrechner "Single-Instruction / Multiple Data " (SIMD)-Maschinen sind, die allenfalls eine geringe Zahl parallel arbeitender Prozessoren haben (2-5), kann man aus Mikroprozessoren "Multiple-Instruction / Multiple Data" (MIMD) - Maschinen aufbauen.

Diese können aus hunderten bis tausenden parallel arbeitender Mikroprozessoren bestehen (z. B. Connection Machine).

9.6 Beispiele realer Maschinen: Die Intel-Serie

Die neueren Typen der Intel-80x86-Serie besitzen ab dem 80286 eine Pipeline. Beim Pentium und Pentium Pro, dies sind superskalare Prozessoren, sind sogar jeweils mehrere Pipelines implementiert.
Ab dem 386-er Prozessor und erst recht der 80486 und der Pentium sind in ihrer Komplexität den Architekturen von Großrechnern aus den 70er und 80er-Jahren durchaus ebenbürtig.

Der 80386 kann als erste 32-Bit-Maschine gelten. Er hat eine Pipeline und eine interne 32-Bit-Organisation. Optional ist er aber auch kompatibel zu früheren Prozessoren als 16-Bit-Maschine betreibbar. Adreß-und Datenbus sind intern jeweils 32 Bit breit.
Extern hat der 80386 SX einen 16 Bit-Datenbus und einen 24 Bit-Adressbus, der 80386 DX eine volle 32-Bit-Breite der Busse .
Für die effiziente Verarbeitung von Floating-Point-Operationen muß der 80386 um einen 80387-Koprozessor erweitert werden, der mit 32 Bit Breite asynchron angekoppelt wird.
Bereits der 80386-Prozessor besitzt eine Code-Prefetch-Einheit, die eine Tiefe von 12 Byte umfaßt. In Phasen, in denen kein Speicherzugriff zum Zweck des Datantransfers notwendig ist, erfolgt vorausschauend ein Transfer von Befehlen in diese Prefetch-Unit, um durch Analyse von Abhängigkeiten und ggf. Umordnen die Pipeline optimal füllen zu können. Der 80386 hat folgende Funktionseinheiten:

- Bus-Schnittstelleneinheit (Bus Interface Unit)
- Code-Prefetch-Einheit
- Befehls-Dekodier-Einheit
- Ausführungseinheit
- Segmentierungs-Einheit
- Seiteneinheit

Diese Funktionseinheiten arbeiten zu einem großen Teil parallel und nebenläufig.
Der 386 kann sowohl 32-Bit- als auch 16-Bit-Befehlscode verarbeiten. Es ist sogar zulässig, 16-Bit- und 32-Bit-Codemodule gemischt zu verwenden, sogar ein Mischung innerhalb eines Moduls ist möglich.

Architektonisch lehnt sich der 80486 eng an den 80386 an. Er ist wie der 80386 eine 32-Bit-Maschine, jedoch mit direkt integrierter Floating-Point-Unit. Bei den 80486 SX-Typen ist diese Einheit zwar physikalisch vorhanden, jedoch logisch "tot". Um sie zu ersetzen, wird ein 80487 Coprozessor eingesetzt. Der 80486 DX-Prozessor ist dagegen mit einer funktionierenden Floating-Point-Einheit ausgestattet.
(Nach Tips aus PC-Büchern ist der 80487-Coprozessor nicht als ein kompletter 80486 DX, der bei Einbau auf dem Board den 80486 SX komplett außer Betrieb setzt und alles allein macht.)
Als weitere Neuerungen hat der 80486 einen 8 kByte Cache-Speicher für Befehle integriert, ist also nicht so stark wie der 80386 auf einen externen Cache angewiesen.

Wesentliche Geschwindigkeitssteigerungen gegenüber dem 80386 erreicht der 80486 durch die Quasi-Einführung der RISC-Technologie.
Viele Befehle, für die der 80386 mehrere Befehlszyklen benötigt, werden im 80486 einem Zyklus abgearbeitet, z. B. die meisten Festkomma-Befehle.
Bezüglich des Speicherzugriffs kann der 80486 erstmals "Burst"-Zyklen organisieren, das ist das Lesen mehrerer aufeinanderfolgender Speicherzellen ohne jeweils explizite Adressierung der Speicherzellen. Die interne Taktrate hat dieselbe Frequenz wie der externe Takt (bis auf den 80486 DX-2, der mit interner Taktverdopplung arbeitet).

Der 80486 hat eine 5-stufige Pipeline mit den Stufen:

- PF (Prefetch)
- D1 Befehlsdekodierung
- D2 Adressgenerierung
- EX Ausführung (ALU und Cache-Zugriff)
- WB Write-Back

Anders als "richtige" RISC-Prozessoren kann der 80486 schon mit einem einzigen Befehl auf Cache-Inhalte zugreifen.

Die nach außen benötigte Bus-Breite kann durch Steuerbits von außen auf 8-, 16-, oder 32 Bit eingestellt und von Zyklus zu Zyklus modifiziert werden.

Der Pentium beinhaltet wiederum die wesentlichen Architekturen des 80486 mit einigen wesentlichen Erweiterungen:

- ein zweiter integrierter Cache-Speicher (Daten-Cache)
- eine Vorhersage-Einheit für Programmverzweigungen
- eine zweite Pipeline.

Die Pipelines werden als "U-Pipeline" und "V-Pipeline" bezeichnet. Die U-Pipeline kann jeden Befehl der 80X86-Architektur ausführen, die V-Pipeline kann nur "einfache" Befehle. Man kann deshalb sagen, daß der Pentium bezüglich der U-Pipeline ein CISC-Prozessor ist, dagegen auf der V-Pipeline ein RISC-Prozessor.
Zwei parallel verarbeitete Befehle müssen "gepaart" werden und dabei bestimmte Abhängigkeits-Relationen einhalten. So müssen es z. B. "einfache" Befehle sein, also keine Komplexbefehle aus dem 80X86-Befehlssatz. Es dürfen auch keine Abhängigkeiten zwischen ihnen bezüglich des Lesens und Schreibens von Registern bestehen.

(Wie flink ein Pentium mit beiden Pipelines ist, das beweist er bei Betrieb als "RISC-Maschine" unter LINUX).

Die Floating-Point-Einheiten wurden bezüglich der verwendeten Algorithmen aber auch durch eine verbesserte Implementierung wesentlich schneller.

Für den "Prefetch" von Befehlen wirken der Cache-Speicher und zwei Prefetch-Buffer von jeweils 32 Byte Größe zusammen. Darüber hinaus existiert ein spezieller Zwischenspeicher für die Verzweigungsvorhersage (Branch-Target Buffer).