10 Pipelining

Durch Pipelining kann man viele Abläufe im Rechner beschleunigen. Insbesondere ist es eine Implementierungstechnik, die CPUs schneller macht.

Die Idee ist die der Fließbandverarbeitung in Produktionsprozessen. Die Ausführung eines Befehls wird in einzelne Stufen eingeteilt, so dass später diese Stufen von aufeinanderfolgenden Befehlen zeitlich überlappend ausgeführt werden können.

10.1 Implementierungstechnik

Man spricht von **Stufen der Pipeline, pipe stages,** pipe segments.

Der Vergleich mit der Autoproduktion liegt nahe: Die von Henry Ford eingeführte Fließbandverarbeitung maximierte den Durchsatz, d.h. so viele Autos wie möglich werden pro Stunde produziert. Und das, obwohl die Produktion eines Autos viele Stunden dauern kann.

Hier: **So viele Instruktionen wie möglich sollen in einer Zeiteinheit ausgeführt werden.**

10.1.1 Durchsatz

Da die Stufen der Pipeline hintereinander hängen, muss die Weitergabe der Autos (der Instruktionen) synchron, also **getaktet** stattfinden. Sonst würde es irgendwo einen Stau geben.

Der Takt für diese Weitergabe wird bei uns der Maschinentakt sein.

Die Länge des Maschinentaktzyklus ist daher bestimmt von der maximalen Verarbeitungszeit der Stufen in der Pipeline für eine Berechnung.

Der Entwerfer der Pipeline sollte daher anstreben, alle Stufen so zu gestalten, dass die Verarbeitung innerhalb der Stufen etwa gleichlang dauert.
Im Idealfall ist die durchschnittliche Ausführungszeit für eine Instruktion auf einer „gepipelineten“ Maschine

$$\frac{\textit{Zeit für die nicht gepipelinete Verarbeitung}}{\textit{Anzahl der Stufen}}$$

Dadurch ist der speedup durch pipelining höchstens gleich der Anzahl der Stufen.

Wie bei der Autoproduktion, wo mit einer n-stufigen Pipeline n mal so viele Autos gefertigt werden können.

Durch Pipelining benötigt man

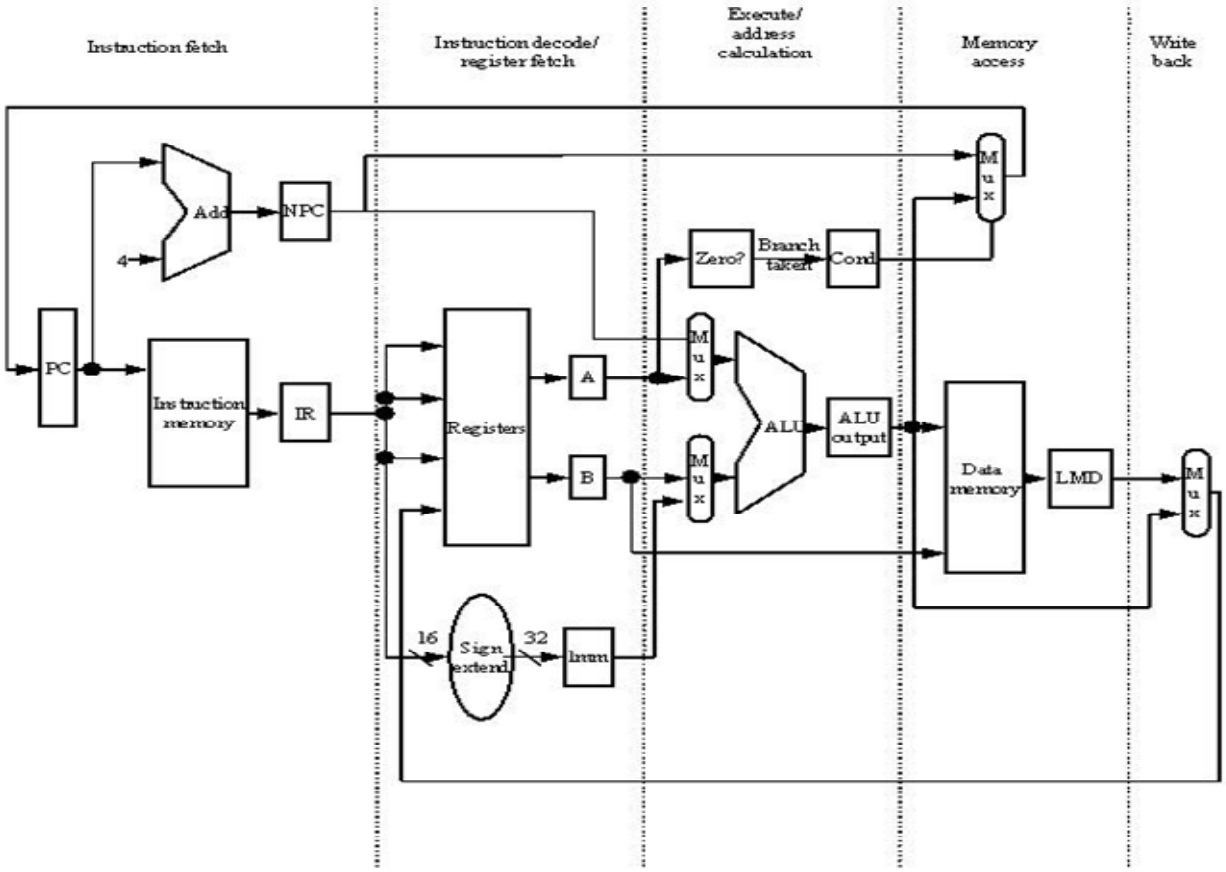
- **weniger Maschinentakte für einen Befehl oder**
- **weniger Zeit für einen Taktzyklus oder**
- **beides**

Wenn die Ursprungsmaschine mehrere Takte für die Ausführung einer Instruktion brauchte, gilt ersteres.

Wenn die Ursprungsmaschine einen langen Takt für die Ausführung einer Instruktion brauchte, wird durch Pipelining die Taktzykluszeit verringert.
Pipelining ist **für den Benutzer** des Rechners **unsichtbar.** Die dafür erforderlichen Mechanismen in der Hardware wollen wir hier diskutieren.

10.1.2 Die DLX-Pipeline

Wir werden die Probleme des Pipelining an der DLX studieren, weil sie einfach ist und alle wesentlichen Merkmale eines modernen Prozessors aufweist. Wir kennen bereits eine nicht gepipelinete (sequentielle) Version der DLX:



Sie lässt sich leicht in eine gepipelinete Version umbauen.

Die Ausführung jeder Instruktion dauert 5 Taktzyklen:

1. Instruction Fetch Zyklus: (IF):

IR <-- Mem[PC]

NPC <-- PC + 4

Hole den Wert aus dem Speicher, wobei die Adresse im gegenwärtigen PC ist und speichere ihn im IR (Instruction-Register). Erhöhe den PC um 4, denn dort steht die nächste Instruktion. Speichere diesen Wert in NPC (neuer PC).

2. Instruction decode / Register fetch Zyklus (ID):

A <-- Regs[$IR_{6..10}$]

B <-- Regs[$IR_{11..15}$]

IMM <-- $((IR_{16})^{16}$##$IR_{16..31}$

Dekodiere die Instruktion in IR und hole die Werte aus den adressierten Registern. Die Werte der Register werden zunächst temporär in A und B gelatcht (zwischengespeichert) für die Benutzung in den folgenden Taktzyklen. Die unteren 16 Bit von IR werden sign-extended und ebenso im temporären Register IMM gespeichert.

Dekodierung und Register lesen werden in einem Zyklus durchgeführt. Das ist möglich, weil die Adressbits von Registern und offset im IR (Befehl) ja an festen Positionen stehen 6..10, 11..15, 16..31. Es kann sein, dass wir ein Register lesen, das wir gar nicht brauchen. Das schadet aber nichts, es wird sonst einfach nicht benutzt.

Der Immediate Operand offset wird um 16 Kopien des Vorzeichenbits ergänzt und ebenfalls gelesen, falls er im nachfolgenden Takt gebraucht wird.

3. Execution / effective adress Zyklus (EX):

Hier können vier verschiedene Operationen ausgeführt werden, abhängig vom DLX-Befehlstyp:

Befehl mit Speicherzugriff (load/store):

ALUoutput <-- A + IMM

Die effektive Adresse wird ausgerechnet und im temporären Register ALUoutput gespeichert.

Register-Register ALU-Befehl:

ALUoutput <-- A func B

Die ALU wendet die Operation func (die in Bits 0..5 und 22..31 des Opcodes spezifiziert ist) auf A und B an und speichert das Ergebnis im temporären Register ALUoutput.

Register-Immediate ALU-Befehl:

ALUoutput <-- A op IMM

Die ALU wendet die Operation op (die in Bits 0..5 des Opcodes spezifiziert ist) auf A und IMM an und speichert das Ergebnis im temporären Register ALUoutput.

Verzweigungs-Befehl:

ALUoutput <-- NPC + IMM

Cond <-- (A op 0)

Die ALU berechnet die Adresse des Sprungziels relativ zum PC. Cond ist ein temporäres Register, in dem das Ergebnis der Bedingung gespeichert wird. Op ist im Opcode spezifiziert und ist z.B. gleich bei BEQZ oder ungleich bei BNEZ.

Wegen der load/store Architektur kommt es nie vor, dass gleichzeitig eine Speicheradresse für einen Datenzugriff berechnet und eine arithmetische Operation ausgeführt werden muss. Daher sind die Phasen Execution und Effective Adress im selben Zyklus möglich.

4. Memory access / branch completion Zyklus (MEM):

Die in diesem Zyklus aktiven Befehle sind load, store und branches:

Lade Befehl (load):

LMD <-- Mem[ALUoutput]

Das Wort, adressiert mit ALUoutput, wird ins temporäre Register LMD geschrieben.

Speicher Befehl (store):

Mem[ALUoutput] <-- B

Der Wert aus B wird im Speicher unter der Adresse in ALUoutput gespeichert.

Verzweigungs Befehl (branch):

if cond then

PC <-- ALUoutput

else PC <-- NPC

Wenn die Verzweigung ausgeführt wird, wird der PC auf das Sprungziel, das in ALUoutput gespeichert ist gesetzt, sonst auf den NPC. Ob verzweigt wird, ist im Execute Zyklus nach cond geschrieben worden.

5. Write back Zyklus (WB):

Register-Register ALU Befehl:

Regs[$IR_{16..20}$] <-- ALUoutput

Register-Immediate ALU Befehl:

Regs[$IR_{11..15}$] <-- ALUoutput

Lade Befehl:

Regs[$IR_{11..15}$] <-- LMD

Das Ergebnis wird ins Register geschrieben. Es kommt entweder aus dem ALUoutput oder aus dem Speicher (LMD). Das Zielregister kann an zwei Stellen im Befehl codiert sein, abhängig vom Opcode.

Am Anfang oder am Ende jedes Zyklus steht jedes Ergebnis in einem Speicher, entweder einem GPR oder in einer Hauptspeicherstelle oder in einem temporären Register (LMD, IMM, A, B, NPC, ALUoutput, Cond). Die temporären Register halten ihre Werte nur während der aktuellen Instruktion, während alle anderen Speicher sichtbare Teile des Prozessorzustands sind.
Wir können die Maschine jetzt so umbauen, dass wir fast ohne Veränderung in jedem Takt die Ausführung einer Instruktion beginnen können. Das wird mit Pipelining bezeichnet. Es

werden synchron getaktete Registersätze zwischen je zwei der eben beschriebenen 5 Zyklen eingebaut. In diese Register, den so genannten **Pipeline-Latches**, nehmen wir alle Werte auf, die zwischen aufeinanderfolgenden Zyklen weitergegeben werden müssen, z.B. das IR, die Register A, B und IMM, den ALUoutput, das LMD, den NPC usw. Nun kann die ID-Phase des ersten Befehls gleichzeitig mit der IF-Phase des zweiten Befehls ausgeführt werden. Einen Takt später ist der erste Befehl in der EX-Phase, der zweite in der ID-Phase und der dritte in der IF-Phase. So wird immer weiter gearbeitet.
Das resultiert in einem Ausführungsmuster wie auf der folgenden Folie:

Befehl	**Takt**								
	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**

Die Takte beim Pipelining und die versetzt parallele Bearbeitung mehrerer Befehle. Zu jedem Takt darf nur eine der fünf Phasen (IF, ID, EX, MEM, WB) gleichzeitig aktiv sein.

10.2 Ausführung der Befehle in der Pipeline

Pipelining ist nicht so einfach wie es hier zunächst erscheint. Im Folgenden wollen wir die Probleme behandeln, die mit Pipelining verbunden sind.

Unterschiedliche Operationen müssen zum Zeitpunkt i unterschiedliche Teile der Hardware nutzen. Um das zu überprüfen, benutzen wir eine vereinfachte Darstellung des DLX-Datenpfades, den wir entsprechend der Pipeline zu sich selbst verschieben.

An vier Stellen tauchen Schwierigkeiten auf:

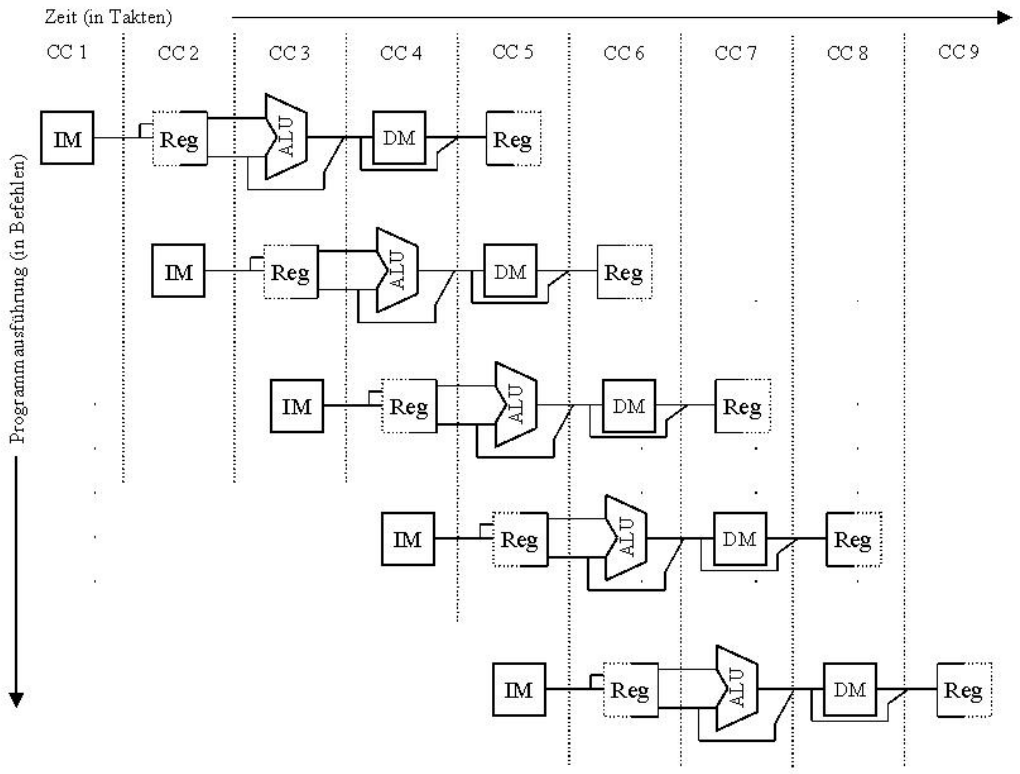
1. **Im IF Zyklus und im MEM Zyklus wird auf den Speicher zugegriffen**. Wäre dies physikalisch derselbe Speicher, so könnten nicht in einem Taktzyklus beide Zugriffe stattfinden. Daher verwenden wir zwei verschieden Caches, einen **Daten-Cache**, auf den im **MEM Zyklus** zugegriffen wird und einen **Befehls-Cache**, den wir im **IF-Zyklus** benutzen. Ein **Cache-Speicher** enthält dabei eine Kopie des relevanten Bereiches des Hauptspeichers, z.B. einen kleinen Bereich des Code-Segmentes um den PC herum, oder Auszüge aus dem Daten- und Stack-Bereich. Der Cache ist klein gegenüber dem Hauptspeicher, aber als SRAM aufgebaut und daher schnell und in einem Taktzyklus addressierbar. Nebenbei: der Speicher muss in der gepipelineten Version fünfmal so viele Daten liefern wie in der einfachen Version. Der dadurch entstehende Engpass zum Speicher ist der Preis für die höhere Performance.

2. **Die Register werden im ID und im WB-Zyklus benutzt**.
3. **PC.** In der vereinfachten Darstellung des Datenpfades haben wir den PC nicht drin.

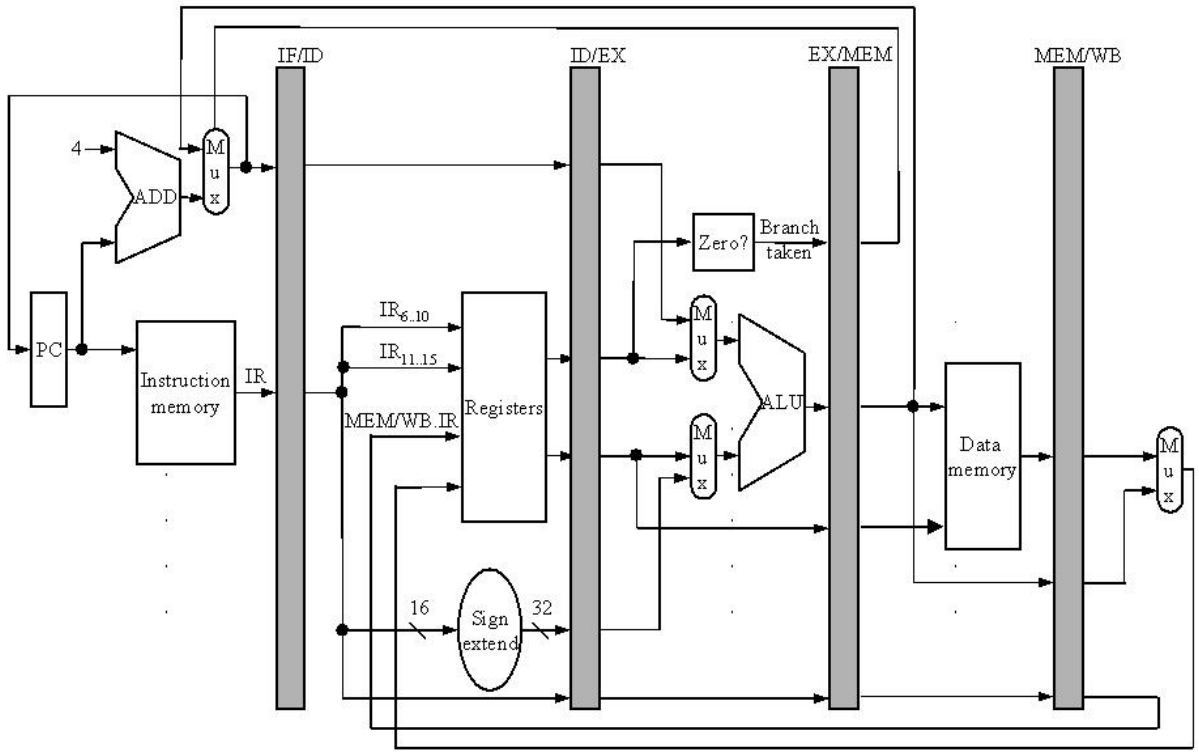
   PC muss in jedem Takt inkrementiert werden, und zwar in der IF-Phase.
4. **Jede Verzweigung verändert PC erst in MEM-Phase.** Aber das Inkrementieren passiert bereits in IF-Phase. Dies erfordert eine spezielle Behandlung von Verzweigungsbefehlen, die wir hier nicht behandeln wollen. Wir sagen stattdessen: Wenn dieser Fall eintritt, muss die Pipeline für einige Takte unterbrochen (gestaut) werden.

Folgende Voraussetzungen müssen für einen reibungslosen Ablauf gegeben sein:

- **Jede Stufe ist in jedem Takt aktiv.**
- **Jede Kombination von gleichzeitig aktiven Stufen muss möglich sein.**
- **Werte, die zwischen zwei Stufen der Pipeline weitergereicht werden, müssen in Registern gespeichert werden (gelatcht).**



Die Register sind in der nächsten Abbildung zu sehen. Sie werden mit den Stufen bezeichnet, zwischen denen sie liegen.



Alle temporären Register aus unserem ersten Entwurf können jetzt in diese Register mit aufgenommen werden.

Die Pipeline-Register halten aber Daten **und** Kontrollinformation.

Beispiel: Die IR-Information muss mit den Stufen der Pipeline weiterwandern, damit zum Beispiel in der WB-Phase das Ergebnis in das richtige Register geschrieben wird, das zu dem alten Befehl gehört.
Jeder Schaltvorgang passiert in einem Takt, wobei die Eingaben aus dem Register vor der entsprechenden Phase genommen werden und die Ausgaben in die Register nach der Phase geschrieben werden.

Die folgende Abbildung zeigt die Aktivitäten in den einzelnen Phasen unter dieser Sicht:

Stufe	Was wird alles getan		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if EX/MEM.cond {EX/MEM.ALU Output} else {PC+4});		
ID	ID/EX.A ← Regs[IF/ID.$IR_{6..10}$]; ID/EX.B ← Regs[IF/ID.$IR_{11..15}$]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← $(IR_{16})^{16}$ ## $IR_{16..31}$;		
	ALU Befehl	**Load oder store Befehl**	**Verzweigungsbefehl**
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM. ALUOutput ← ID/EX. A func ID/EX. B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm; EX/MEM.cond ← 0;	EX/MEM.IR ← ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.cond ← 0; EX/MEM.B ← ID/EX.B;	 EX/MEM.ALUOutput ← ID/EX.NPC+ID.EX.Imm; EX/MEM.cond ← (ID/EX.A *op* 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs [MEM/WB. $IR_{16..20}$] ← MEM/WB.ALUOutput; or Regs[MEM/WB.$IR_{11..15}$] ← MEM/WB.ALUOutput;	Regs[MEM/WB.$IR_{11..15}$] ← MEM/WB.LMD;	

Die Aktivitäten in den ersten zwei Stufen sind nicht befehlsabhängig. Das muss auch so sein, weil wir den Befehl ja erst am Ende der zweiten Stufe interpretieren können.

Durch die festen Bit-Positionen der Operandenregister im IR-Feld ist die Dekodierung und das Register-Lesen in einer Phase möglich.

Um den Ablauf in dieser einfachen Pipeline zu steuern, ist die Steuerung der vier Multiplexer in dem Diagramm erforderlich:

oberer ALU-input-Mux: **Verzweigung oder nicht**

unterer ALU-input-Mux: **Register-Register-Befehl oder nicht**

IF-Mux: **EX/MEM.cond**

WB-Mux: **load oder ALU-Operation**

Es gibt einen fünften (nicht eingezeichneten Mux), der beim WB auswählt, wo im MEM/WB.IR die Adresse des Zielregisters steht, nämlich an Bits 16..20 bei einem Register-Register-ALU-Befehl und an Bits 11..15 bei einem Immediate- oder Load-Befehl.

10.2.1 Performance Verbesserung durch Pipelining

Der Durchsatz wird verbessert, nicht die Ausführungszeit der einzelnen Instruktion.

Im Gegenteil, durch den **Pipeline-overhead**, die Tatsache, dass die Stufen **nie perfekt ausbalanciert** (Takt wird bestimmt durch die langsamste Stufe) sind und die zusätzlichen **Latches mit ihren Setup-Zeiten und Schaltzeiten** wird die **einzelne Instruktion meist langsamer**.

Aber insgesamt: **Programme laufen schneller, obwohl keine einzige Instruktion individuell schneller läuft.**

Beispiel: Ungepipelinete DLX: vier Zyklen branch- oder store-Operationen, fünf für ALU- oder load-Operationen. ALU 30%, branch 20%, store 10%, load 40%. 200 MHz Takt. Angenommen, durch den Pipeline-Overhead brauchen wir eine ns mehr pro Stufe. Welchen speedup erhalten wir durch die Pipeline?

Durchschnittliche Ausführungszeit für einen Befehl =

Zykluszeit * durchschnittliche Anzahl Zyklen pro Befehl =

5ns * ((20%+10%)*4 + (30%+40%)*5) =

23,5 ns.

In der Pipeline-Version muss der Takt mit der Zykluszeit der langsamsten Stufe laufen plus Overhead. Also 5ns+1ns.

$$\boldsymbol{speedup = \frac{ungepipelinete\ Ausführungszeit}{gepipelinete\ Ausführungszeit} = \frac{23,5}{6} = 3,9}$$

Soweit würde die Pipeline gut für paarweise unabhängige Integer-Befehle funktionieren. In der Realität hängen Befehle aber voneinander ab. Dieses Problem werden wir im Folgenden behandeln.

10.3 Pipeline Hazards

Es gibt Fälle, in denen die Ausführung einer Instruktion in der Pipeline nicht in dem für sie ursprünglichen Takt möglich ist. Diese werden **Hazards** genannt. Es gibt drei Typen:

1. **Strukturhazards** treten auf, wenn die Hardware die Kombination zweier Operationen, die gleichzeitig laufen sollen, nicht ausführen kann. Beispiel: Schreiben in den Speicher gleichzeitig mit IF bei nur einem Speicher.
2. **Datenhazards** treten auf, wenn das Ergebnis einer Operation der Operand einer nachfolgenden Operation ist, dies Ergebnis aber nicht rechtzeitig vorliegt.
3. **Steuerungshazards, Control hazards** treten auf bei Verzweigungen in der Pipeline oder anderen Operationen, die den PC verändern.

Hazards führen zu einem **Stau (stall)** der Pipeline. Pipeline-Staus sind aufwendiger als z.B. Cache-Miss-Staus, denn einige Operationen in der Pipe müssen weiterlaufen, andere müssen angehalten werden.

In der Pipeline müssen alle Instruktionen, die schon länger in der Pipeline sind als die gestaute, weiterlaufen, während alle jüngeren ebenfalls gestaut werden müssen. Würden die älteren nicht weiterverarbeitet, so würde der Stau sich nicht abbauen lassen.

Als Folge werden keine neuen Instruktionen gefetcht (gelesen in IF), solange der Stau dauert.

10.3.1 Struktur Hazards

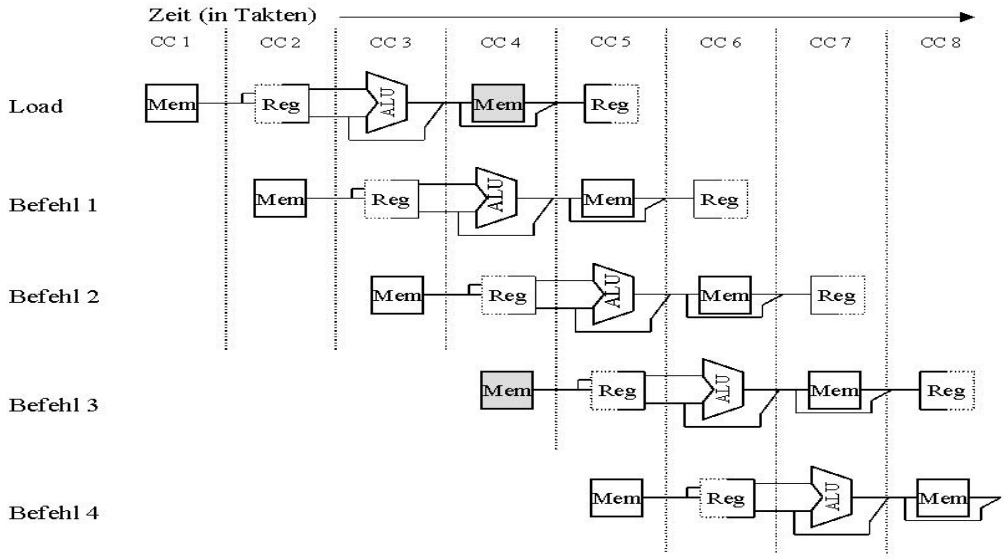
Die überlappende Arbeit an allen Phasen der Pipeline gleichzeitig erfordert, dass alle Ressourcen oft genug vorhanden sind, so dass alle Kombinationen von Aufgaben in unterschiedlichen Stufen der Pipeline gleichzeitig vorkommen können. Sonst bekommen wir einen Struktur Hazard.

Typischer Fall: Eine Einheit ist nicht voll gepipelined: Dann können die folgenden Instruktionen, die diese Einheit nutzen, nicht mit der Geschwindigkeit 1 pro Takt abgearbeitet werden.

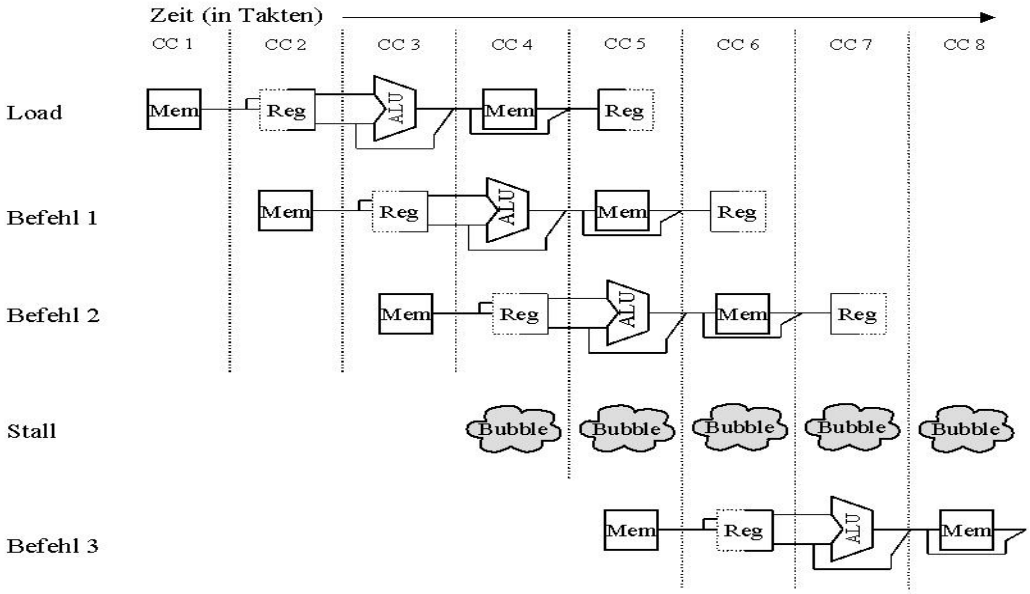
Ein anderer häufiger Fall ist der, dass z.B. der Registerblock nur einen Schreibvorgang pro Takt erlaubt, aber zwei aufeinanderfolgende Befehle in unterschiedlichen Phasen beide ein writeback in ein Register ausführen wollen.

Wenn ein Struktur Hazard erkannt wird, staut die Pipeline, was die CPI auf einen Wert > 1 erhöht.

Anderes Beispiel: Eine Maschine hat keinen getrennten Daten- und Instruktions-Cache. Wenn gleichzeitig auf ein Datum und eine Instruktion zugegriffen wird, entsteht ein Struktur Hazard. Die folgende Folie zeigt solch einen Fall. Die leer arbeitende Phase (durch den Stau verursacht) wird allgemein eine pipeline bubble (Blase) genannt, da sie durch die Pipeline wandert, ohne sinnvolle Arbeit zu tragen.



Problem: Befehl 1 (MEM) und Befehl 3 (IF) wollen in CC4 gleichzeitig auf MEM zugreifen. Das erzeugt einen Struktur Hazard.



Lösung: Statt Befehl 3 direkt auszuführen, wird ein Bubble (stall) eingeführt und der Befehl 3 greift erst in CC5 auf MEM zu.

Zur Darstellung der Situation in einer Pipeline ziehen einige Entwerfer eine andere Form des Diagramms vor, das zwar nicht so anschaulich, aber dafür kompakter und genauso aussagefähig ist.

Ein Beispiel dafür: Für jeden Befehl wird direkt die Pipeline Stage in den Takt eingetragen. So ist sofort ersichtlich, wenn zwei Befehle gleichzeitig auf eine Stage zugreifen wollen.

Befehl	Takt									
	1	2	3	4	5	6	7	8	9	10
Load Befehl	**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**	**WB**
Befehl i+5							**IF**	**ID**	**EX**	**MEM**
Befehl i+6								**IF**	**ID**	**EX**

10.3.2 Daten Hazards

Betrachten wir folgendes Assembler-Programmstück

ADD R1, R2, R3

SUB R4, R5, R1

AND R6, R1, R7

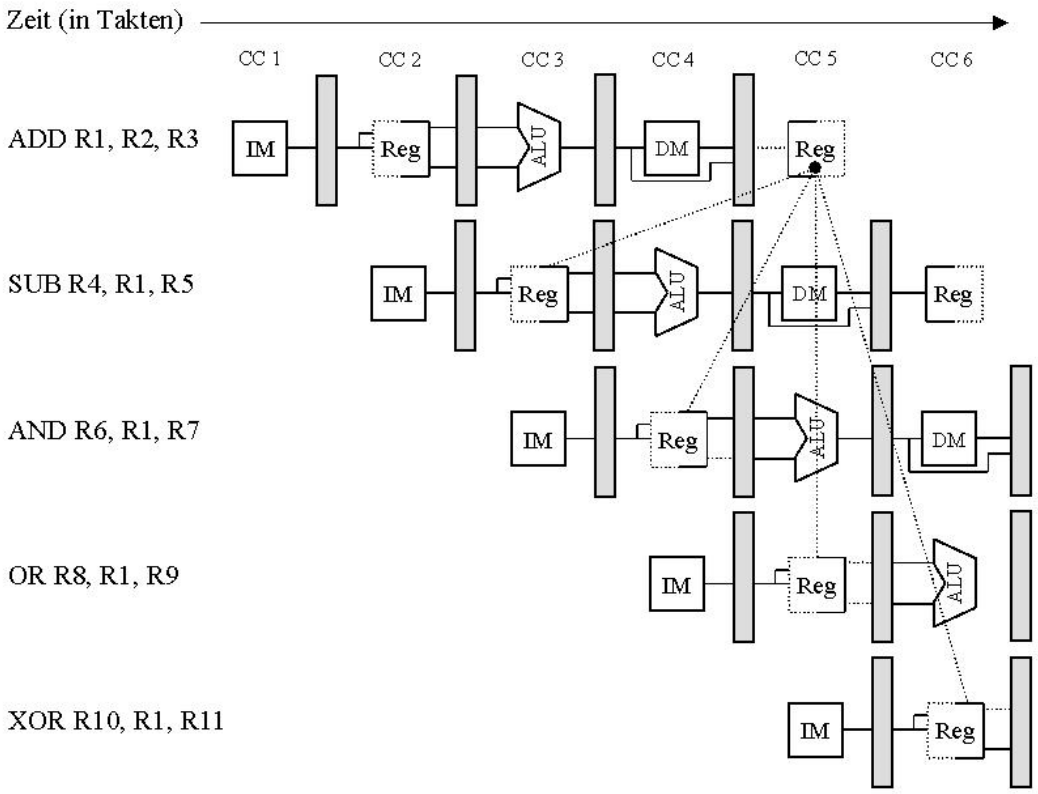
OR R8, R1, R9

XOR R10, R1, R11

Alle Befehle nach ADD brauchen das Ergebnis von ADD. Wie man auf der folgenden Folie sieht, schreibt ADD das Ergebnis aber erst in seiner WB-Phase nach R1.

Aber SUB liest R1 bereits in deren ID-Phase. Dies nennen wir einen **Daten Hazard.**

Wenn wir nichts dagegen unternehmen, wird SUB den alten Wert von R1 lesen. Oder noch schlimmer, wir wissen nicht, welchen Wert SUB bekommt, denn wenn ein Interrupt dazu führen würde, daß die Pipeline zwischen ADD und SUB unterbrochen würde, so würde ADD noch bis zur WB-Phase ausgeführt werden, und wenn der Prozess neu mit SUB gestartet würde, bekäme SUB sogar den richtigen Operanden aus R1.



Der AND-Befehl leidet genauso unter dem Hazard. Auch er bekommt den falschen Wert aus R1.

Das XOR arbeitet richtig, denn der Lesevorgang des XOR ist zeitlich nach dem WB des ADD.

Das OR können wir dadurch sicher machen, dass in der Implementation dafür gesorgt wird, dass jeweils in der ersten Hälfte des Taktes ins Register geschrieben wird und in der zweiten Hälfte gelesen. Dies wird in der Zeichnung durch die halben Kästen angedeutet.

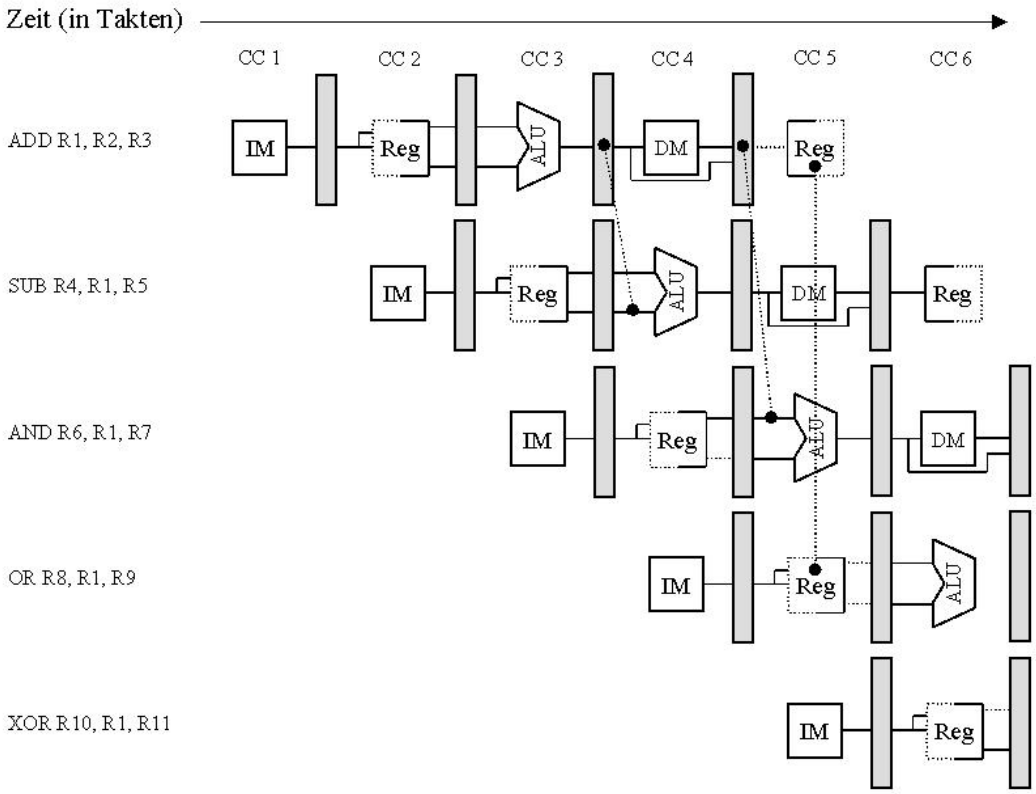
Wie vermeiden wir nun Staus bei SUB und AND?

Das Zauberwort heißt **forwarding**.

Das geht folgendermaßen:

1. **Das Ergebnis der ALU (aus EX/MEM) wird (auch) in den ALU-Eingang zurückgeführt.**

2. **Eine spezielle Forwarding Logik, die nur nach solchen Situationen sucht, wählt die zurückgeschriebene Version anstelle des Wertes aus dem Register als tatsächlichen Operanden aus und leitet ihn an die ALU.**



Beim Forwarding ist bemerkenswert: Wenn SUB selbst gestallt wird, benötigt der Befehl nicht die rückgeführte Version des Ergebnisses, sondern kann ganz normal aus R1 zugreifen.

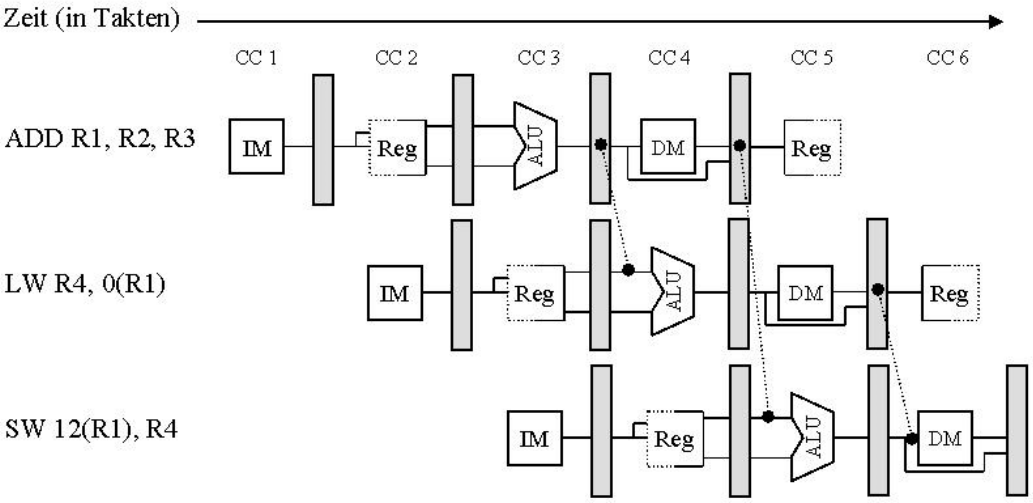
Das Beispiel zeigt auch, dass das neue Ergebnis auch für den übernächsten Befehl (AND) in rückgeführter Form zur Verfügung gehalten werden muss.

Forwarding ist eine Technik der beschleunigten Weiterleitung von Ergebnissen an die richtige Verarbeitungseinheit, wenn der natürliche Fluss der Pipeline für einen solchen Transport nicht ausreicht.

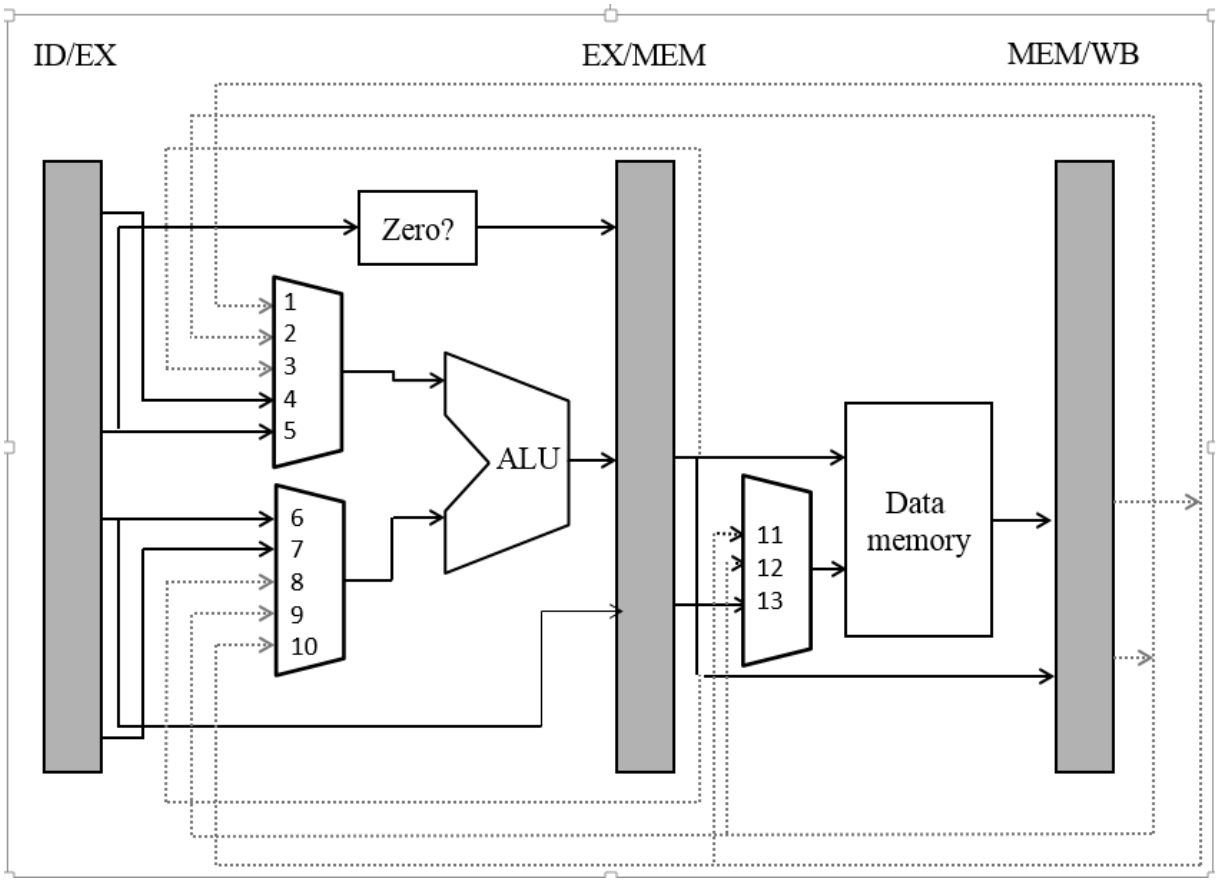
Ein anderes Beispiel:

```
ADD   R1, R2, R3
LW    R4, 0(R1)
SW    12(R1), R4
```

Um hier den Daten Hazard zu vermeiden, müssten wir R1 rechtzeitig zum ALU-Eingang bringen und R4 zum Speichereingang. Die folgende Abbildung zeigt die Forwarding-Pfade, die für dieses Problem erforderlich sind.



Für die Forwarding-Pfade muss man in der DLX Rückleitungen aus den Pipeline-Latches an die Multiplexer vorsehen, die jeweils dann genutzt werden können, wenn die Folge von Befehlen das Forwarding erforderlich macht. Die folgende Abbildung zeigt einen Ausschnitt aus dem DLX-Datenpfad, der diese zusätzlichen Leitungen enthält.



10.3.3 Daten Hazards, die Staus verursachen müssen

Es gibt Daten Hazards, die nicht durch forwarding zu entschärfen sind:

LW R1, 0(R2)

SUB R4, R1, R3

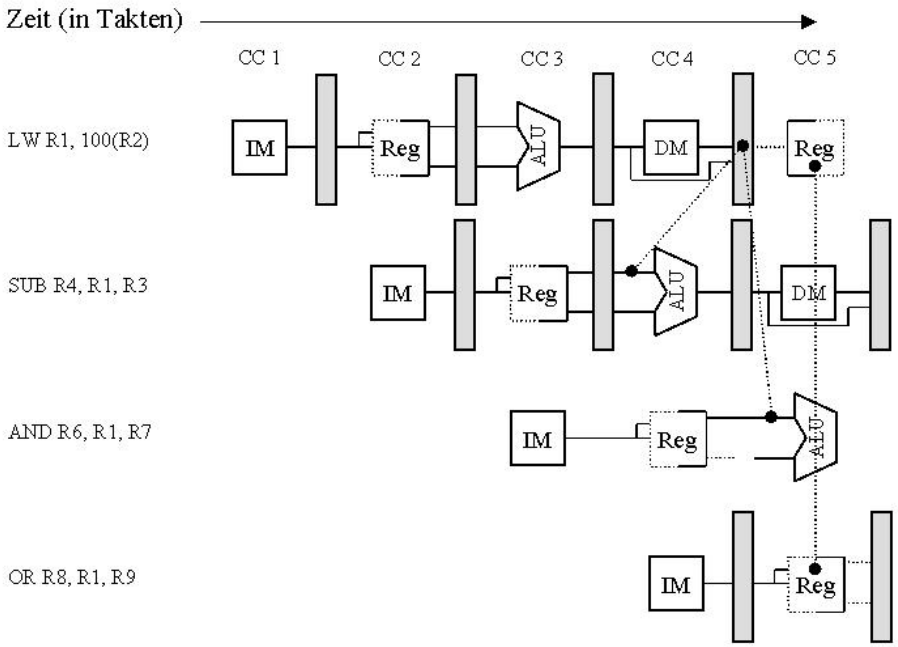
AND R6, R1, R7

OR R8, R1, R9

Die LW-Instruktion hat den Wert für R1 erst nach deren MEM-Phase, aber bereits während ihrer MEM-Phase läuft die EX-Phase der SUB-Instruktion, die den Wert von R1 benötigt.

Ein forwarding-Pfad, der dies verhindern kann, müsste einen Zeitsprung rückwärts machen können, eine Fähigkeit, die selbst modernen Rechnerarchitekten noch versagt ist.

Wir können zwar für die AND und OR-Operation forwarden, aber nicht für SUB.
Für diesen Fall benötigen wir spezielle Hardwaremechanismen, genannt **Pipeline interlock**.



Pipeline interlock staut die Pipeline.

Nur die Instruktionen ab der Verbraucherinstruktion für den Datenhazard werden verzögert.

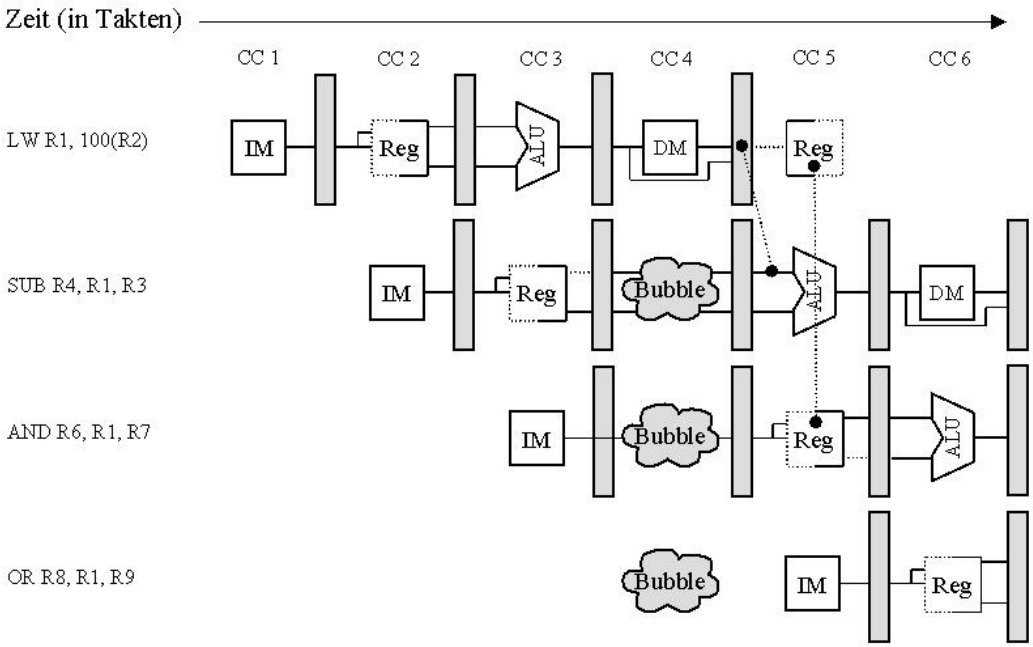
Die Quellinstruktion und alle davor müssen weiterbearbeitet werden.

Dazwischen entsteht eine Pause, eine sogenannte **Pipeline-blase (bubble)**, in der nichts Sinnvolles berechnet wird.

Im Beispiel sieht das so aus:

Alles verzögert sich um einen Takt ab der Bubble.

In Takt vier wird keine Instruktion begonnen und also auch keine gefetcht.



LW R1,0(R1)	IF	ID	EX	MEM	WB				
SUB R4,R1,R3		IF	ID	EX	MEM	WB			
AND R6,R1,R7			IF	ID	EX	MEM	WB		
OR R8,R1,R9				IF	ID	EX	MEM	WB	

LW R1,0(R1)	IF	ID	EX	MEM	WB				
SUB R4,R1,R3		IF	ID	**stall**	EX	MEM	WB		
AND R6,R1,R7			IF	**stall**	ID	EX	MEM	WB	
OR R8,R1,R9				**stall**	IF	ID	EX	MEM	WB

10.3.4 Kontrollhazards

Kosten mehr pro Stau bei der DLX.

Ausgeführte Verzweigung: Sprung zur Zieladresse (taken branch)

Nicht ausgeführte Verzweigung: PC+4 (not taken branch, untaken)

Erst in der Execute-Phase wird die Zieladresse berechnet, die Condition ausgewertet. D.h. erst in der MEM-Phase stehen der neue PC und das Cond-Register zur Verfügung.

Vor der ID-Phase wissen wir noch nicht, dass es ein Verzweigungsbefehl ist, daher können wir erst in der ID-Phase stauen. Das bedeutet, dass der Nachfolgebefehl noch bis in die IF-Phase kommt.

Das Stauen der Pipeline und Warten, ob die Verzweigung ausgeführt wird oder nicht resultiert, in folgendem Ablauf:

Branch-Befehl	IF	ID	EX	MEM	WB				
Branch + 1		IF	stau	stau	IF	ID	EX	MEM	WB
Branch + 2						IF	ID	EX	MEM
Branch + 3							IF	ID	EX

Der Stau in diesem Falle ist anders als beim Datenhazard, denn die erste IF-Phase muss im Falle eines Verzweigungsbefehls ja durch eine zweite IF-Phase überschrieben werden. Daher muss die Hardware im Staufalle das IF/ID-Register auf 0 setzen (no-op).

Wenn die Verzweigung nicht ausgeführt wird, ist der Stau eigentlich nicht erforderlich, denn wir haben mit dem Branch +1 Befehl bereits den richtigen Befehl gefetcht. Wir werden gleich sehen, wie man das ausnutzen kann, um Performance zu gewinnen.

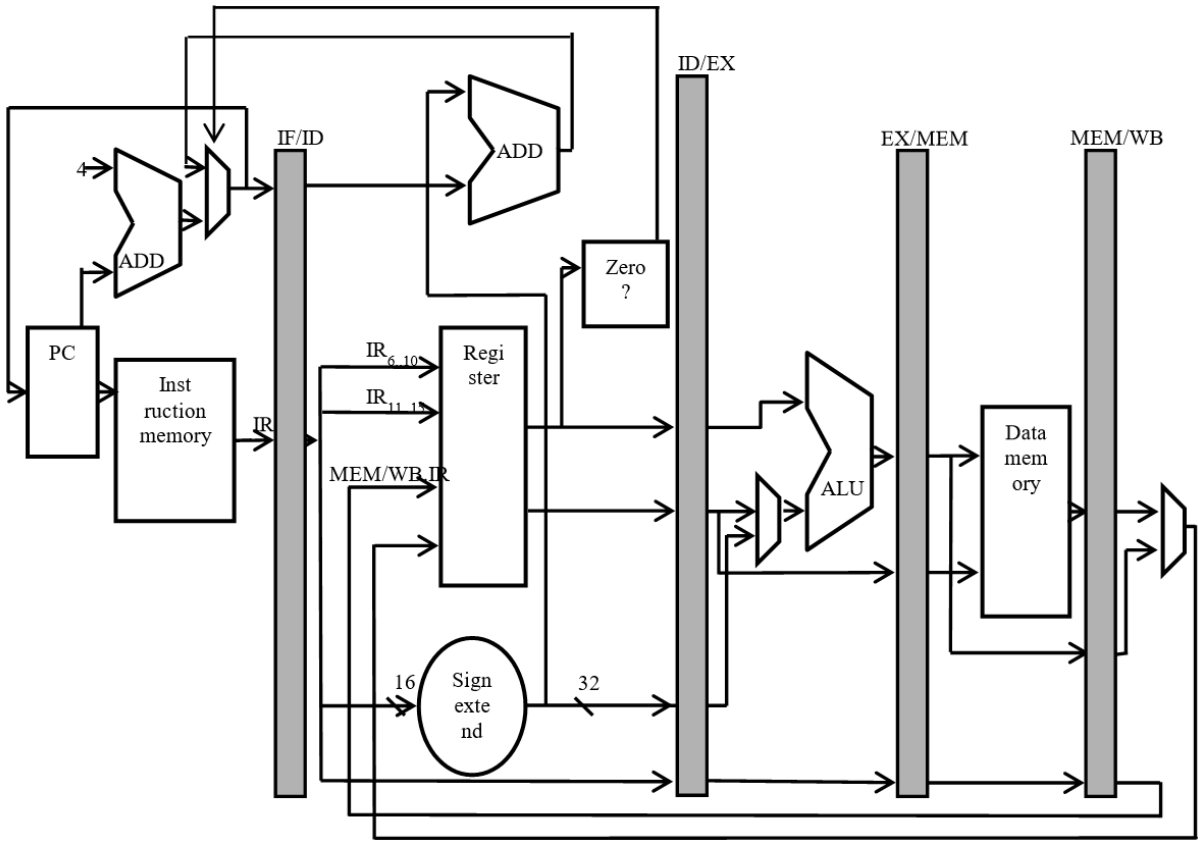
Aber zuerst: **Wie können wir die Stau-Strafe vermindern?**

1. Früher herausfinden, ob verzweigt wird oder nicht.

2. Sprungziel eher berechnen.

Zusätzliche ALU (nur Adder) für die Berechnung des Verzweigungsziels bereits in der ID-Phase.

Cond wird auch bereits in der ID-Phase berechnet.



Die Abfolge der Schaltvorgänge in der Pipeline wird sich dadurch folgendermaßen ändern:

Stufe	**Verzweigungsbefehl**		
IF	IF/ID.NPC, PC	⟵	If(IF/ID.$IR_{0..5}$ = BRANCH und Regs[IF/ID.$IR_{6..10}$] op 0) {NPC + (IF/ID IR_{16})^{16} ## IF/ID.$IR_{16..31}$} else {PC + 4}
	IF/ID.IR	⟵	MEM [PC]
ID	ID/EX.A	⟵	Regs[IF/ID.$IR_{6..10}$];
	ID/EX.B	⟵	Regs[IF/ID.$IR_{11..15}$];
	ID/EX.IR	⟵	IF/ID.IR;
EX	ID/EX.IMM	⟵	(IF/ID.IR_{16})^{16} ## IF/ID.$IR_{16..31}$
MEM			
WB			

Wie kann man die Verzweigungsstrafe minimieren?

Vier Strategien, die zur Compilezeit gewählt werden können:

1) **freeze**
2) **predict not taken, treat as not taken**
3) **predict taken, treat as taken**
4) **delayed branch**

1. Freeze

Stauen bis das Ergebnis der Verzweigung bekannt ist. **Vorteil: Einfachheit für Soft- und Hardware. Nachteil: hoher CPI-Wert.**

2. Treat as not taken

Jede Verzweigung wird zunächst so behandelt, als werde sie nicht ausgeführt. Die Pipeline fährt fort, Instruktionen zu holen und zu dekodieren, bis das Ergebnis des Tests vorliegt. Im Falle, dass der Sprung nicht genommen werden soll, wird normal (ohne Penalty) weitergemacht. Im Falle, dass gesprungen werden soll, werden die fälschlich gefetchten und dekodierten Befehle von der Hardware in no-ops umgewandelt, wodurch alle temporären Register und alle GPRs erhalten bleiben, bis das Sprungziel neu gefetcht worden ist.

Das führt zu folgendem Pipeline-Diagramm bei der ausgeführten Verzweigung

Ausgeführter Branch-Befehl	IF	ID	EX	MEM	WB		
Branch + 1		IF	stau	stau	stau	stau	
Sprungziel			IF	ID	EX	MEM	WB
Sprungziel + 1				IF	ID	EX	MEM

Beziehungsweise bei der nicht ausgeführten Verzweigung

Nicht ausgeführter Branch-Befehl	IF	ID	EX	MEM	WB		
Branch + 1		IF	ID	EX	MEM	WB	
Branch + 2			IF	ID	EX	MEM	WB
Branch + 3				IF	ID	EX	MEM

Vorteil: Keine Strafe, wenn nicht genommene Verzweigung. Nachteil: mehr Verzweigungen werden ausgeführt als nicht ausgeführt. Daher:

3. Treat as taken

Jede Verzweigung wird zunächst so behandelt, als werde sie ausgeführt. Die Pipeline fährt fort, Instruktionen ab dem Sprungziel zu holen und zu dekodieren, bis das Ergebnis des Tests vorliegt. Im Falle, dass der Sprung genommen werden soll, wird normal (ohne Penalty) weitergemacht. Im Falle, dass nicht gesprungen werden soll, werden die fälschlich gefetchten

und dekodierten Befehle von der Hardware in no-ops umgewandelt, wodurch alle temporären Register und alle GPRs erhalten bleiben, bis neu gefetcht worden ist.

Bei unserer DLX bringt diese Strategie nichts, denn wir kennen das Sprungziel ja erst, wenn wir auch die Bedingung ausgewertet haben. Dies ist aber bei anderen Maschinen, z. B. welchen mit einer tieferen Pipeline, nicht der Fall. Daher gilt das folgende Diagramm so nicht für die DLX.

Vorteil: Keine oder nur geringe Strafe bei Verzweigung. Nachteil: Kostet extra Hardware, früh herauszufinden, ob verzweigt wird.

4. delayed branch

Hinter dem Verzweigungsbefehl gibt es sogenannte delay slots. Das sind die möglichen Nachfolgebefehle, die begonnen werden, bevor die Sprungentscheidung getroffen ist. Diese delay slots werden durch Scheduling (also sinnvolles Vertauschen von Maschinenbefehlen) mit anderen Instruktionen gefüllt, die auf jeden Fall (also unabhängig vom Ausgang des Vergleichs) ausgeführt werden.

Bei der DLX haben wir einen Delay-slot (dies ist üblich für Maschinen mit delayed branch).

Die Situation sieht also so aus wie in dem folgenden Diagramm für ausgeführte bzw. nicht ausgeführte Verzweigung.

Ausgeführter Branch-Befehl	IF	ID	EX	MEM	WB		
Branch delay Befehl		IF	ID	EX	MEM	WB	
Sprungziel			IF	ID	EX	MEM	WB
Sprungziel + 1			IF	ID	EX	MEM	

Beziehungsweise bei der nicht ausgeführten Verzweigung

Nicht ausgeführter Branch-Befehl	IF	ID	EX	MEM	WB		
Branch delay Befehl		IF	ID	EX	MEM	WB	
Branch + 1			IF	ID	EX	MEM	WB
Branch + 2				IF	ID	EX	MEM

Der Compiler ist nun verantwortlich dafür. dass die Instruktion im Delay Slot valide und nützlich ist. Es gibt **drei Möglichkeiten**, die er hat, um die Validität mit Sicherheit und die Nützlichkeit mit größter zur Compilezeit ermittelbarer Wahrscheinlichkeit zu gewährleisten:

1. Schedule from before

2. Schedule from target

3. Schedule from fall through

1. From Before:

```
                ADD         R1, R2, R3
                BEQZ        R2, Sprungziel
                Delay slot
.
.
.
Sprungziel
```

Wird zu:

```
                BEQZ        R2, Sprungziel
                ADD         R1, R2, R3
.
.
.
Sprungziel
```

From before ist immer von Vorteil, wenn es möglich ist. Aber man muss einen Befehl, finden dessen Ergebnis nicht die Bedingung beeinflusst. Wenn from before nicht möglich ist, wählt der Compiler eine der anderen Möglichkeiten:

2. From target

```
                ADD         R1, R2, R3
                BEQZ        R2, Sprungziel
                Delay slot
.
.
.
Sprungziel      SUB         R6, R7, R8
```

Wird zu

```
                ADD         R1, R2, R3
                BEQZ        R2, Sprungziel+4
                SUB         R6, R7, R8
.
.
.
Sprungziel      SUB         R6, R7, R8
```

Man wählt für den delay slot eine Instruktion vom Ziel des Sprunges. Diese Variante soll gewählt werden, wenn die Ausführung des Sprungs wahrscheinlich ist, z. B. bei Rücksprüngen für Schleifen. Dabei ist zu bedenken:

- Die Instruktion muss kopiert werden, weil das Sprungziel auch von einer anderen Stelle aus angesprungen werden könnte.
- Die Instruktion darf im Falle, dass der Sprung nicht ausgeführt wird, nicht falsch sein, d. h. das Zielregister muss in diesem Falle redundant sein.
- Die Instruktion darf nicht noch einmal ausgeführt werden, falls sie durch den Schedule nach dem Branch ausgeführt wird.

3. From fall through

```
            ADD         R1, R2, R3
            BEQZ        R2, Sprungziel
            Delay slot
.
            SUB         R6, R7, R8
.
Sprungziel
```

Wird zu:

```
            ADD         R1, R2, R3
            BEQZ        R2, Sprungziel
            SUB         R6, R7, R8
.
.
.
Sprungziel
```

Man wählt für den delay slot einen Befehl nach dem Sprungbefehl. Diese Variante ist vorzuziehen, wenn der Sprung mit größter Wahrscheinlichkeit nicht ausgeführt wird.

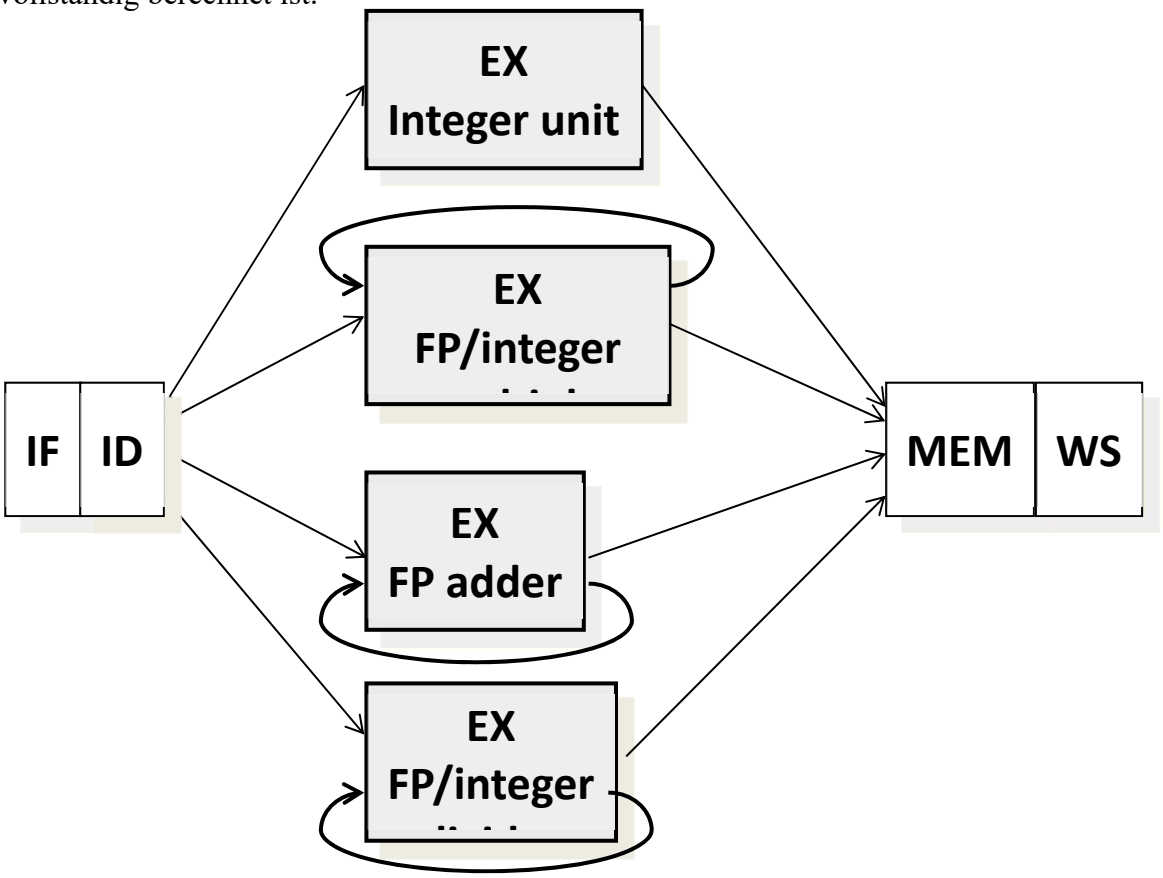
Auch hier muss die Ausführung des delay-Befehls legal sein, falls der Sprung doch in unerwartete Richtung geht. Dies wäre z. B. der Fall, wenn R6 ein Register ist, das zu dieser Zeit sonst von niemandem benutzt wird.

10.4 Erweiterung der DLX-Pipeline für Floating-Point Arithmetik

Es ist bei heutigen Taktfrequenzen unmöglich, FP-Operationen in einem Takt auszuführen. Daher macht man folgende Veränderungen für FP-Operationen.

Anstelle einer **EX-Einheit** stellt man vier verschiedene Einheiten zur Verfügung. Die erste ist die alte **Integer-Einheit**. Die zweite ist für **FP- und Integer-Multiplikation**, die dritte für **FP-Addition** und die vierte für **FP- und Integer-Division**.

Da die Operationen außerhalb der EX-Einheit nicht in einem Zyklus ausgeführt werden können, müssen die Teilergebnisse wieder an deren Eingang zurückgeführt werden, bis das Ergebnis vollständig berechnet ist.



Nun liegt es nahe, auch die anderen Einheiten ihrerseits als Pipelines zu implementieren. Dazu muss man zunächst wissen, wie lange die jeweiligen Berechnungen dauern, d. h. wie viele Stufen die jeweilige Pipeline haben wird.

Latency ist die Anzahl der Takte, die der Befehl zusätzlich zur alten EX-Phase (ungepipelined für die Ausführung) braucht. Soviele+1 Stufen der Pipeline sieht man vor.

Die Tiefe der Pipeline ist also die Latency + 1.

Mit **Initiation interval** wird die Anzahl der Takte bezeichnet, die erforderlich sind, bevor nach Beginn einer Berechnung die nächste Instanz der Berechnung in der Pipeline beginnen kann.

Wir sehen an der Tabelle, dass alle Einheiten voll in Pipelinestufen der Länge 1 unterteilt sind, außer Division und SQRT, die nicht gepipelined sind.

Einheit	Latency	Initiation Intervall
Integer ALU	0	1
FP Add	3	1
FP Mult	6	1
FP Div	14	15

Es werden jetzt natürlich **zusätzliche Pipeline-Register** notwendig: M1/M2, M2/M3, ..., A1/A2, ID/EX, ID/M1, ID/A1, ID/DIV, M7/MEM, A4/MEM, DIV/MEM

Die unterschiedlichen Pipelinetiefen verursachen nun neue Probleme:

Befehl	1	2	3	4	5	6	7	8	9	10	11
MULTD	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
ADDD		IF	ID	A1	A2	A3	A4	MEM	WB		
LD			IF	ID	EX	MEM	WB				
LD				IF	ID	EX	MEM	WB			

1. **RAW-Staus werden wesentlich häufiger**. Die Kontrolle für Pipeline-Staus ist aufwendiger, obwohl sie im Prinzip mit den Methoden, die wir kennengelernt haben, zu bewältigen ist.

Beispiel oben:

```
MULTD   F0, F2, F4
ADD     F8, F6, F0
```

oder

```
MULTD   F0, F2, F4
ADD     F8, F6, F4
LD      F6, 400(R2)
SD      320(R1), F0
```

2. **Struktur-Hazards werden möglich**, da die Division nicht gepipelined ist.

Beispiel

```
DIVD    F0, F2, F4
DIVD    F6, F8, F10
```

3. Weil die Operationen unterschiedlich lang sind, müssen mehr als eine Schreiboperation in den Registerfile pro Takt möglich sein. Wenn dasselbe Register beschrieben werden soll, entsteht ein **Struktur Hazard**.

4. **WAW Hazards werden möglich**.

Beispiel:

```
MULTD   F0, F2, F4
ADDD    F0, F6, F8
```

5. **Instruktionen werden in anderer Reihenfolge fertig, als sie begonnen werden.**

Das nennt man out-of-order completion oder OOO-completion. Alle heutigen Prozessoren haben leistungsfähige Mechanismen, um all diese Probleme zu lösen.

Im folgenden Beispiel sehen wir eine typische Folge von FP-Befehlen (daxpy-loop, **d**ouble precision **a*X p**lus **Y**)). Jede Instruktion muss hier auf den Vorgänger warten, obwohl schon alle Möglichkeiten für forwarding ausgenutzt sind.

Interessant ist ferner, dass das SD am Ende noch einen weiteren Takt gestaut werden muss, weil es nicht gleichzeitig in der MEM Phase sein kann wie das ADDD (denn die Leitungen sind nur einfach vorhanden, d.h. Daten können nicht von zwei Befehlen gleichzeitig in der MEM Phase verarbeitet werden).

Befehl	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F4, 0(R2)	IF	ID	EX	MEM	WB												
MULTD F8, F4, F6	IF	ID	stau	M1	M2	M3	M4	M5	M6	M7	MEM	WB					
ADD F2, F8, F10				IF	ID	stau	stau	stau	stau	stau	A1	A2	A3	A4	MEM	WB	
SD 0(R3) F2						IF	ID	EX	stau	stau	stau	stau	stau	stau	stau	MEM	

Zu 3. Weiteres Beispiel:

Drei Befehle kommen gleichzeitig in ihre MEM und WB-Phase. Die MEM-Phase ist unkritisch, denn die Konflikte können ohne großen Hardwareaufwand behoben werden (nur mehrfache Register und Leitungen).

Befehl	1	2	3	4	5	6	7	8	9	10	11
MULTD F2,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
.											
.											
ADD F8,F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
.											
.											
LD F8, 0(R2)							IF	ID	EX	MEM	WB

In der WB-Phase kommt es zu Problemen.

1. Lösung: Mehrere Write-Ports. Teuer, denn der Fall kommt zu selten vor, als das dieser Aufwand gerechtfertigt wäre.

2. Lösung: Die späteren Befehle werden gestaut, so dass nacheinander geschrieben werden kann.

Was ist dazu notwendig?

Zwei Möglichkeiten:

1. Alle machen erst mal bis zur **MEM Phase** weiter und stauen dann. Vorteil: Man kann andere, u.U. zwischendurch aufgetretene Staus ausnutzen, so dass für WB kein weiterer Stau erforderlich ist. Nachteil: Wenn gestaut werden muss, müssen rückwärts in der Pipeline alle

Stufen angehalten werden. Das ist sehr aufwendig. Viel einfacher ist es, wenn weiterhin alle Hazards in der ID-Phase des zweiten Befehls erkannt und behandelt werden. Daher

2. Ein Schieberegister mit der Länge n des längsten Befehls (Pipelinetiefe) in der ID-Phase führt Buch über die Benutzung des Write-ports. Jeder Befehl, der den Write-port braucht, schreibt eine 1 ins Schieberegister an der Stelle k, wenn er in n-k Takten schreiben will. Wenn dort schon eine Eins steht, wird er einen Takt gestaut, wartet einen Takt und versucht es wieder.