Kapitel 10: Pipelining

- Pipelining (Fließbandtechnik): Implementierungstechnik
  Vielfältig angewendet in der Rechnerarchitektur.
- Pipelining macht CPUs schnell.
- Pipelining ist wie Fließbandverarbeitung.

Hintereinanderausführung und Verschachtelung von Stufen der Pipeline, pipe stages.

Vergleich Autoproduktion: Maximiert 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: Soviele Instruktionen wie möglich sollen in einer Zeiteinheit ausgeführt werden, der Durchsatz soll erhöht werden.

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

Pipelining

Der Takt für diese Weitergabe wird bei uns der Maschinentakt sein.
Die Länge des Maschinentaktzyklus ist daher bestimmt von der maximalen Verarbeitungszeit der Einheiten 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.

Bei Verarbeitung ohne Pipelining benötigt eine Instruktion eine bestimmte Anzahl von Takten (CPI: Clock per Instruction). Das Ziel von Pipelining ist es, durch Schachtelung der Instruktionsabarbeitung in Stufen den mittleren CPI über alle Instruktionen des Programms zu reduzieren und so einen speedup >1 zu erreichen:

$$speedup = \frac{CPI_{keinPipelining}}{CPI_{mitPipelining}} \leq \frac{CPI_{keinPipelining}}{(CPI_{keinPipelining} \,/\, Anzahl\ \ Stufen)} = AnzahlStufen$$

Dadurch ist der speedup durch Pipelining höchstens gleich der Anzahl der Stufen. Dies entspricht der Autoproduktion, bei der einer n-stufigen Pipeline n mal so viele Autos gefertigt werden können wie ohne Fließband.

Berechnung des speedups: Amdahl‘s Gesetz

Amdahl‘s Gesetz gibt generell darüber Aufschluss, wie groß der gesamte speedup s_g bei einer Operation ausfällt, wenn nur ein Teil der Operationen beschleunigt werden kann, ein anderer Teil aber nicht. Dazu ist erforderlich, daß zwei Parameter bekannt sind:
1.) Der Anteil der beschleunigbaren Operationen an der gesamten Aufgabe, genauer, der Zeitanteil, den diese Operationen einnehmen. Dieser Anteil wird als A_b bezeichnet.
2.) Die Zeitverbesserung (speedup s_b), die bei dem beschleunigbarem Anteil der Operation erreicht werden kann, d.h. um wieviel kann ich diese Teiloperation beschleunigen. Dieses wird als s_b bezeichnet.
Der gesamt-Speedup s_g berechnet sich aus dem Verhältnis der Zeiten Z_u/Z_b vor und nach der Beschleunigung:

$$s_g = \frac{Z_{unbeschleunigt}}{Z_{beschleunigt}} = \frac{Z_u}{Z_b}$$

Die Gesamtzeit nach Beschleunigung kann nun wie folgt aufgeteilt werden: Z_b = (Zeitanteil der nicht beschleunigbaren Operationen) + (Zeitanteil der beschleunigbaren Operationen):

$$Z_b = (1 - A_b) \cdot Z_u + A_b \cdot \frac{Z_u}{s_b} = Z_u \cdot \left(1 - A_b + \frac{A_b}{s_b}\right)$$

Einsetzen in Gesamt-Speedup ergibt Amdahl‘s Gesetz

$$s_g = \frac{Z_u}{Z_b} = \frac{1}{1 - A_b + \frac{A_b}{s_b}}$$

Beispiel:

Angenommen wir können eine Operation um einen Faktor 10 beschleunigen, die zu 40% der Ausführungszeit ausgeführt wird. Welchen Speedup erreichen wir dadurch für die gesamte Ausführung?

$Anteil_{beschleunigt} = 0{,}4$ => $Anteil_{unbeschleunigt} = (1 - 0{,}4) = 0{,}6$

$Speedup_{beschleunigt} = 10$ aber nur für 40% der Operationen

$Speedup_{gesamt} = 1/ (0{,}6 + 0{,}4/10) = 1/0{,}64 = 1{,}56$

Man sieht hier, dass es sich evtl. nicht lohnt, eine Operation mit großen Mühen um den Faktor 10 zu beschleunigen, wenn sie nur in 40% der Zeit angewendet wird. Daher sollten möglichst immer die Operationen beschleunigt werden, die sehr häufig ausgeführt werden.

$$Speedup = \frac{1}{1 - Anteil_{beschleungt} + \frac{Anteil_{beschleungt}}{Speedup_{beschleungt}}}$$

Vorteile des Pipelining

Pipelining hat einen guten Speedup, da sie auf jede Operation wirkt und daher der Anteil der beschleunigbaren Operationen (idealerweise) A_b=1,0 ist. Die Ausnahmen werden wir noch diskutieren.

Pipelining kann

- CPI verringern oder
- Zykluszeit verringern oder
- beides

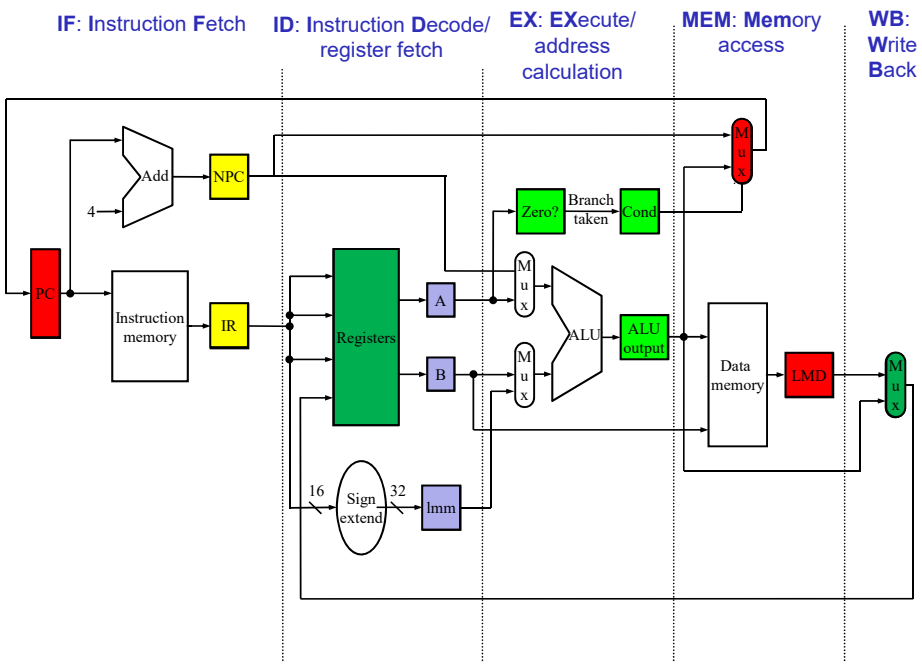
- Wenn die Ursprungsmaschine mehrere Takte für die Ausführung einer Instruktion brauchte, wird durch Pipelining die CPI verringert (ist bein DLX der Fall).
- Wenn die Ursprungsmaschine einen langen Takt für die Ausführung einer Instruktion brauchte, wird durch Pipelining die Taktzykluszeit verringert.
- Pipelining benutzt Parallelverarbeitung innerhalb der Prozessoren. Vorteil: Es ist für den Benutzer unsichtbar.

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. Zunächst gehen wir von einer Version der DLX ohne Pipelining aus:

- nicht optimal, aber so, daß sie sich leicht in eine Version mit Pipelining umbauen läßt.

- die Ausführung jeder Instruktion dauert 5 Taktzyklen.



DLX-Datenpfad mit Taktzyklen (ohne Pipelining)

Die 5 DLX-Taktphasen

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 für die Benutzung in den folgenden Taktzyklen. Die unteren 16 Bit von IR werden sign-extended und ebenso im temporären Register IMM gespeichert.

Die 5 DLX-Taktphasen

Dekodierung und Registerlesen werden in einem Zyklus durchgeführt. Das ist möglich, weil die Adressbits 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 wird um 16 Kopien des Vorzeichenbits ergänzt und ebenfalls gelesen, falls er im nachfolgenden Takt gebraucht wird.

3. Execution / Effective Address Zyklus (**EX**):

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

a) Befehl mit Speicherzugriff (load/store):

ALUoutput <-- A + IMM

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

Die 5 DLX-Taktphasen

b) Register-Register ALU-Befehl:

ALUoutput <-- A func B

Die ALU wendet die Operation func (die in Bits 0..5 und 22..32 des Opcodes spezifiziert ist) auf A und B an und speichert das Ergebnis im temporären Register ALUoutput.

c) 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.

Die 5 DLX-Taktphasen

d) 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 Address Bestimmung im selben Zyklus möglich.

Die 5 DLX-Taktphasen

4. Memory Access / Branch Completion Zyklus (**MEM**):

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

a) Lade Befehl (load):

LMD	←	Mem[ALUoutput]
PC	←	NPC

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

b) Speicher Befehl (store):

Mem[ALUoutput]	←	B
PC	←	NPC

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

Die 5 DLX-Taktphasen

c) 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.

Die 5 DLX-Taktphasen

5. Write back Zyklus (**WB**):

Hier werden die Ergebnisse aus ALUoutput oder Speicher in die Register transferiert.

a) Register-Register ALU Befehl:

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

b) Register-Immediate ALU Befehl:

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

c) Lade-Befehl:

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

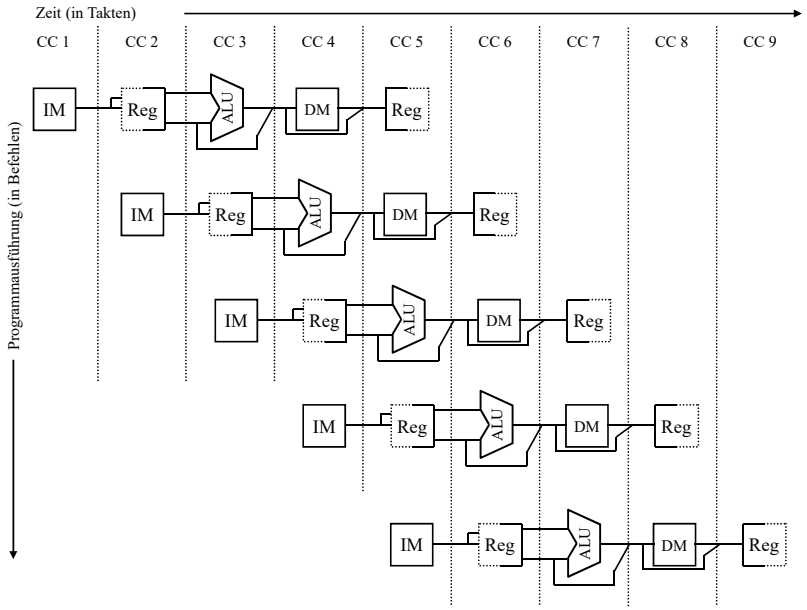
Das Ergebnis wird ins Register geschrieben. Es kommt entweder aus dem ALUoutput oder aus dem Speicher (LMD). Das Zielregister kann an zwei Stellen im Befehl codiert sein, abhängig vom Opcode.

Erlärung zum Datenpfad-Bild

- 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.
- In dieser Implementierung brauchen Verzweigungsbefehle und store-Befehle vier Takte und alle anderen Befehle 5 Takte. Setzt man die Verzweigungshäufigkeit mit 12% und store mit 8% (aus statistischen Untersuchungen bekannt), so kommt man auf eine CPI von 4,80.

Wir können die Maschine jetzt so umbauen, daß wir fast ohne Veränderung in jedem Takt die Ausführung einer Instruktion beginnen können. Das wird als Pipelining bezeichnet. Das resultiert in einem Ausführungsmuster wie auf der folgenden Folie dargestellt.

Vereinfachte Darstellung des DLX-Datenpfads



Das Pipeline Diagramm

	Takt								
Befehl	**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

Probleme beim Pipelining

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 drei Stellen tauchen Schwierigkeiten auf:

1. Gleichzeitiger Speicherzugriff: 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 (schnelle Zwischenspeicher), einen Daten-Cache, auf den im MEM Zyklus zugegriffen wird und einen Befehls-Cache, den wir im IF-Zyklus benutzen. Nebenbei: der Speicher muß in der gepipelineten Version fünf mal so viele Daten liefern wie in der einfachen Version. Der dadurch entstehende Engpass zum Speicher ist der Preis für die höhere Performance.

2. Gleichzeitiger Registerzugriff: Die Register werden im ID und im WB-Zyklus benutzt. Dieses Problem kann durch Hardware und geschickten Zugriff (erst schreiben pos. Flanke, dann lesen neg. Flanke) gemeistert werden.

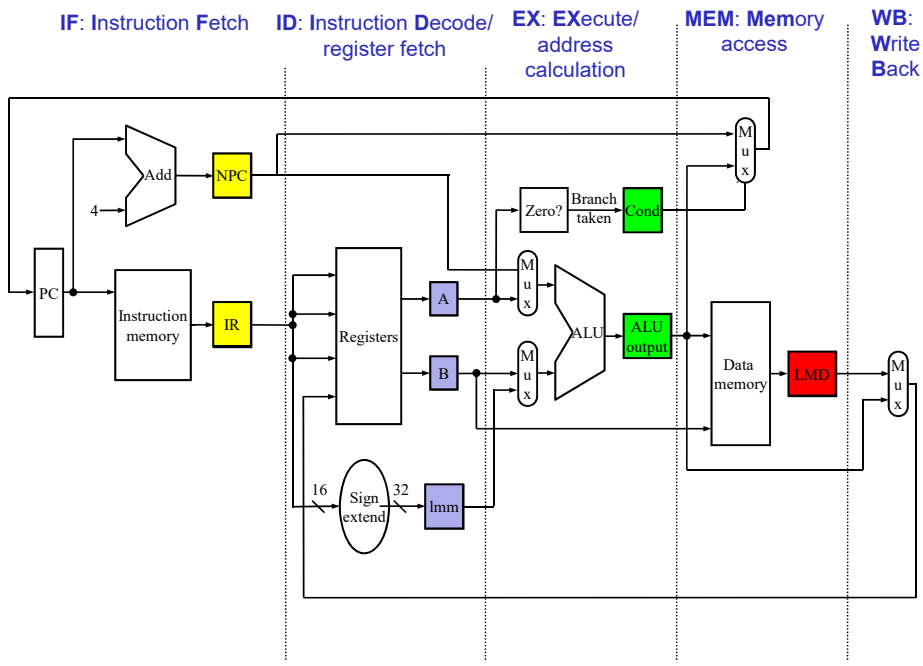
Probleme beim Pipelining

3. Anpassung des PC (Program Counter). In der vereinfachten Darstellung des Datenpfades haben wir den PC nicht beachtet. Der PC muss in jedem Takt inkrementiert werden, und zwar in der IF-Phase. Verzweigungen verändern den PC, aber erst in MEM-Phase. Aber: das Inkrementieren auf NPC passiert bereits in IF-Phase. Wie damit umzugehen ist, wird später dargestellt.

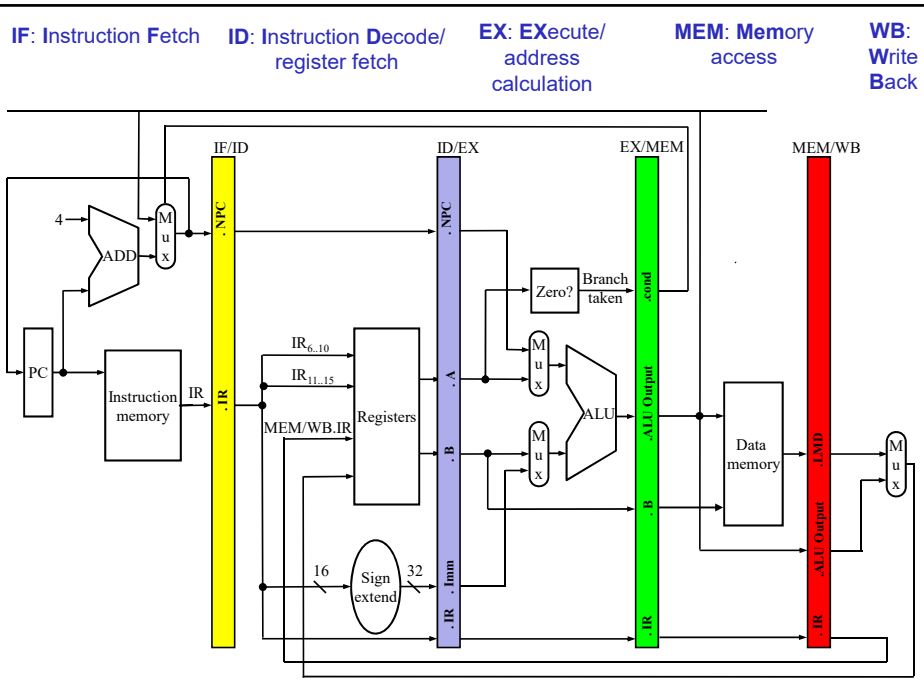
Daher folgt bei Pipelining: Jede Stufe ist in jedem Takt aktiv.

- 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 auf der nächsten Folie zu sehen. Sie werden mit den Namen der Stufen bezeichnet, zwischen denen sie liegen.



DLX-Datenpfad mit Taktzyklen (ohne Pipelining)



DLX-Datenpfad mit Pipelining-Zwischenregistern

Aufgaben der Pipelining-Register

Alle temporären Register aus unserem ersten Entwurf können jetzt in diesen Registern mit aufgenommen werden. Die Pipeline-Register halten aber sowohl Daten als auch 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 Pipeline-Register können dabei synchron gelesen und danach beschrieben werden.

Die folgende Folie zeigt die Aktivitäten in den einzelnen Phasen aus dieser Sicht.

Pipelining-Register in den Pipeline-Phasen

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;	

Steuerung der Pipeline-Phasen durch Multiplexer

Die Aktivitäten in den ersten zwei Stufen IF und ID 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

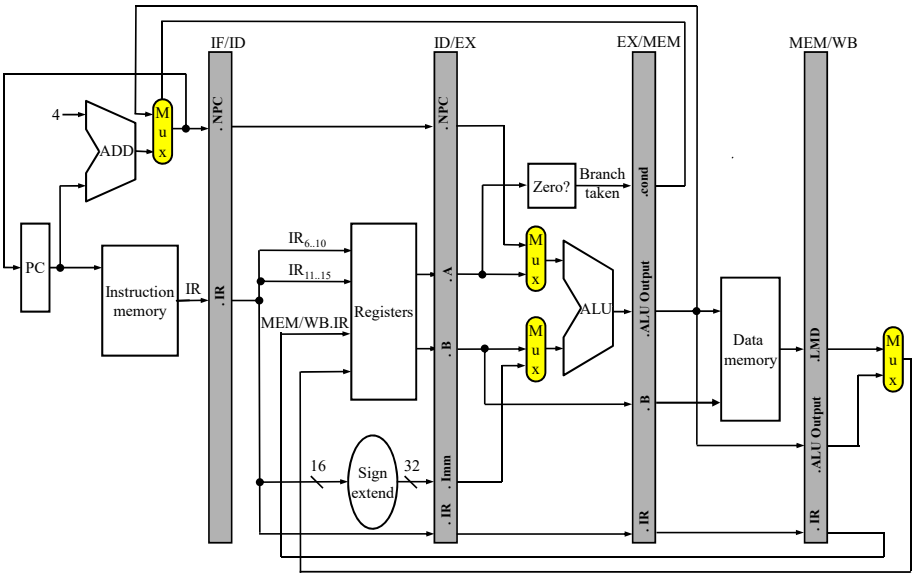
unterer ALU-input-Mux: Register-Register-Befehl oder nicht

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.

Steuerung der Pipeline-Phasen durch Multiplexer



DLX-Datenpfad mit Pipelining-Zwischenregistern, Kontrollfluss durch Multiplexer

Performance Verbesserung durch Pipelining

Durchsatz insgesamt wird verbessert - nicht Ausführungszeit einer Instruktion

im Gegenteil, durch den Pipeline-overhead, durch 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 Verweilzeit der einzelnen Instruktion im Prozessor meist länger.

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

Beispiel: DLX ohne Pipeline: 1 GHz Takt, vier Zyklen für ALU- und branch-Operationen, fünf für Speicher-Operationen. Typische Auslastung: ALU 40%, branch 20%, load/store 40%. Angenommen, durch den Pipeline-Overhead brauchen wir 0,2 ns mehr pro Stufe. Welchen speedup erhalten wir durch die Pipeline?

Durchschnittliche Ausführungszeit für einen Befehl ohne Pipelining =
Zykluszeit * durchschnittliche CPI =1ns * ((40%+20%)*4 + 40%*5) = 4,4 ns.

In der Pipeline-Version läuft der Takt mit der Zykluszeit der langsamsten Stufe plus Overhead, d. h. 1ns + 0,2ns = 1,2ns. Der speedup ist daher

speedup = (Zeit ohne Pipelining) / (Zeit mit Pipelining) = 4.4 ns / 1.2 ns = 3.67

Probleme bei der Umsetzung des Pipelining

Grundsätzlich würde die Pipeline gut für paarweise unabhängige Integer-Befehle funktionieren. In der Realität hängen Befehle aber voneinander ab. Dieses Problem und das der Verarbeitung von Gleitkommabefehlen werden wir im Folgenden behandeln. Die hier auftretenden Probleme werden Pipeline Hazards genannt. Im Fall eines Hazards (Gefahr) ist die Ausführung einer Instruktion in der Pipeline nicht in dem für sie ursprünglichen angedachten Takt möglich ist. Es gibt drei Typen von Hazards:

Struktur Hazards treten auf, wenn die Hardware die Kombination zweier Operationen, die gleichzeitig laufen sollen, nicht ausführen kann. Beispiel: Schreiben in den Speicher (MEM) gleichzeitig mit IF bei nur einem Speicher.

Daten Hazards treten auf, wenn das Ergebnis einer Operation der Operand einer nachfolgenden Operation ist, dies Ergebnis aber nicht rechtzeitig (d. h. in der ID-Phase) vorliegt.

Steuerungs- oder Control Hazards treten auf bei Verzweigungen in der Pipeline oder anderen Operationen, die den PC verändern.

Pipelining Hazards

Hazards führen zu einem Stau (stall) der Pipeline. Pipeline-Staus sind aufwendig, 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 Instruktionen ebenfalls gestaut werden müssen, bis der Stau aufgelöst ist. Würden die älteren nicht weiterverarbeitet, so würde der Stau sich nicht abbauen lassen.

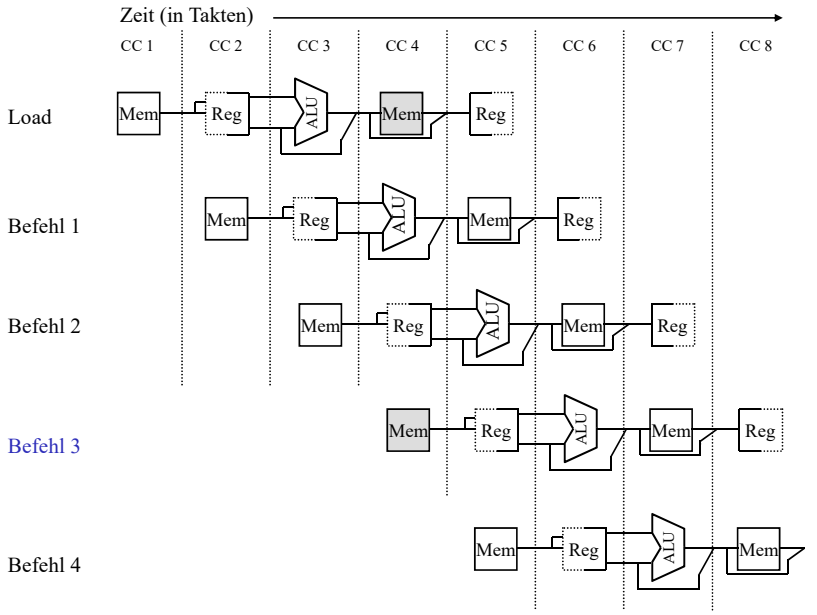
Als Folge werden keine neuen Instruktionen geholt, solange der Stau andauert. Dadurch verlängert sich die mittlere Bearbeitungszeit pro Instruktion und der Pipeline-speedup wird verringert.

Im Folgenden werden die verschiedenen Hazards und ihre Behebung diskutiert.

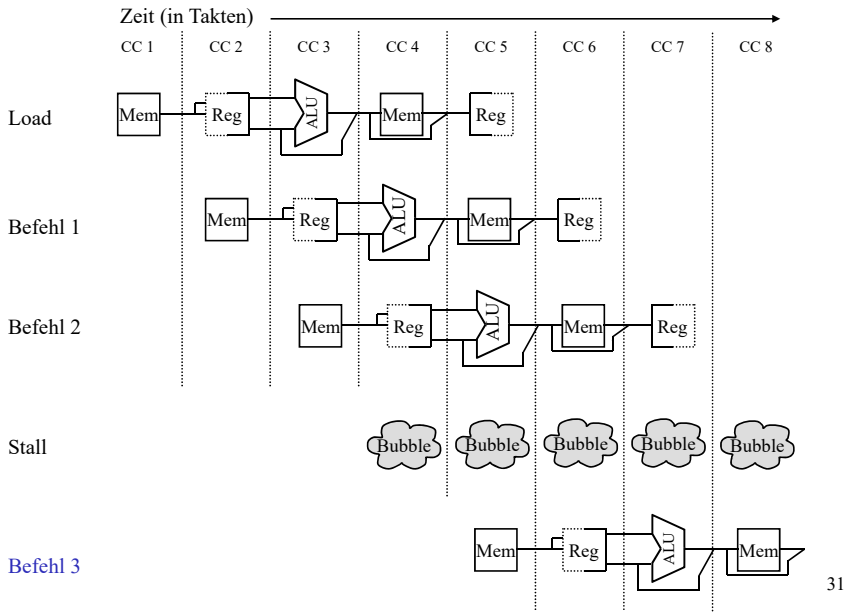
1. Strukturhazards

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 Strukturhazard.
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 write back in ein Register ausführen wollen.
Wenn ein Strukturhazard erkannt wird, staut die Pipeline, was die CPI auf einen Wert > 1 erhöht.
Weiteres Beispiel: Eine Maschine hat keinen getrennten Daten- und Instruktions-Cache. Wenn gleichzeitig auf ein Datum und eine Instruktion zugegriffen wird, entsteht ein Strukturhazard. Die folgende Folie zeigt solch einen Fall. Die leer arbeitende Phase (Leerbefehl No Op, durch den Stau verursacht) wird allgemein eine pipeline bubble (Blase) genannt, da sie durch die Pipeline wandert, ohne sinnvolle Arbeit zu tragen.

Strukturhazard in Befehl 3: IF (3) und MEM (1) beide in Clock Cycle CC 4



Auflösung des Strukturhazards durch Einführen eines Bubbles: Befehl 3 IF erst in CC 5



Hazard-Identifikation im Pipeline Diagramm

Das Pipeline-Diagramm zeigt die Verschiebung des Befehls 3 durch ein stall (Bubble) direkt als Funktion der Taktzyklen auf. Die Verarbeitung aller nachfolgenden Befehle wird um 1 Takt verzögert. Eine spezielle Logik sucht nach solchen Strukturhazards und löst sie durch Einfügen eines stall auf. Alternativ könnte die Hardware angepasst werden, so dass gleichzeitiges schreiben und lesen ermöglicht wird.

	Takt									
Befehl	**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

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, speziell einen RAW (read after write) 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.

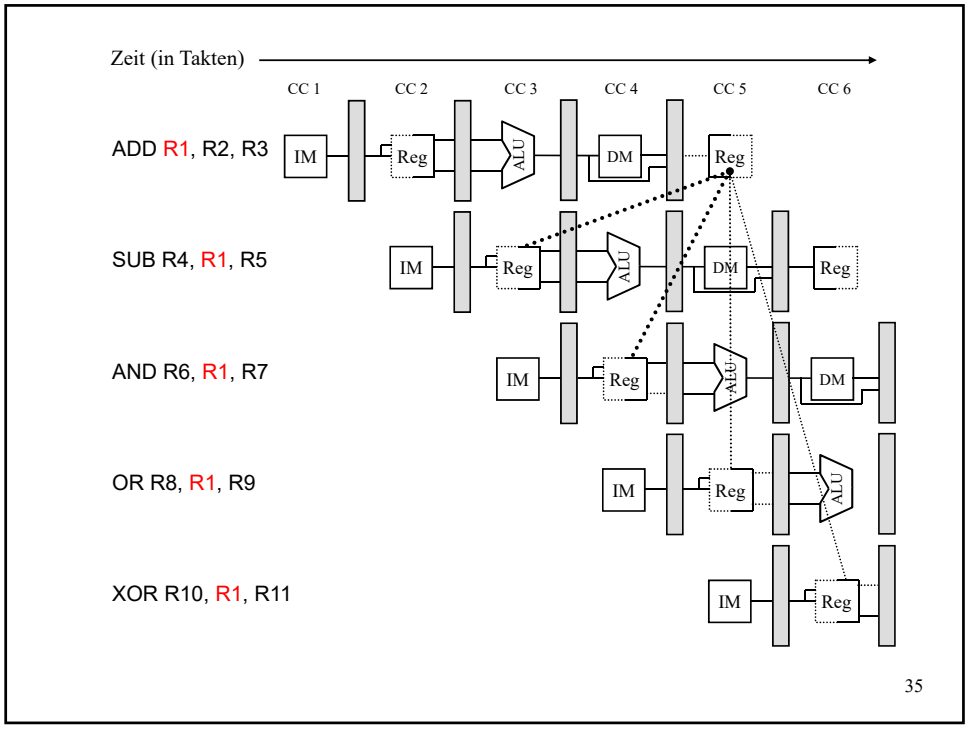
Forwarding

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. Eine Lösung wäre ein stalling, aber das kostet Performanz.
Wie vermeiden wir nun stalls bei SUB und AND?

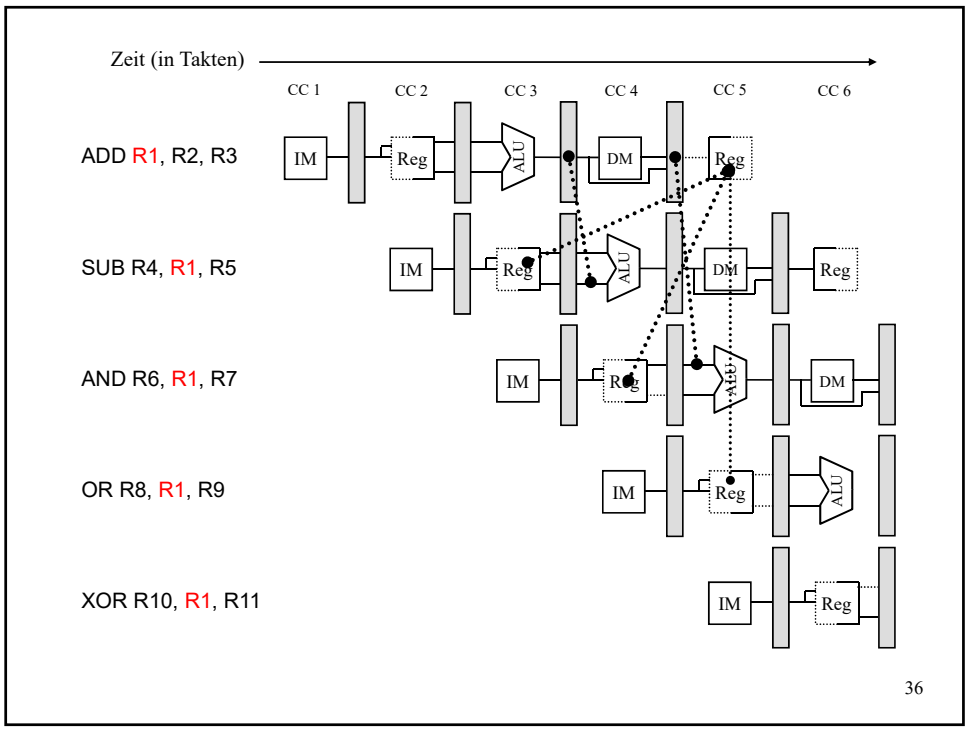
Das Zauberwort heißt forwarding.
Das geht folgendermaßen:

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 zurückgeschrieben Version anstelle des Wertes aus dem Register als tatsächlichen Operanden aus und leitet ihn an die ALU.


Zeit (in Takten)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM
Reg

35


Zeit (in Takten)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM
Reg

36

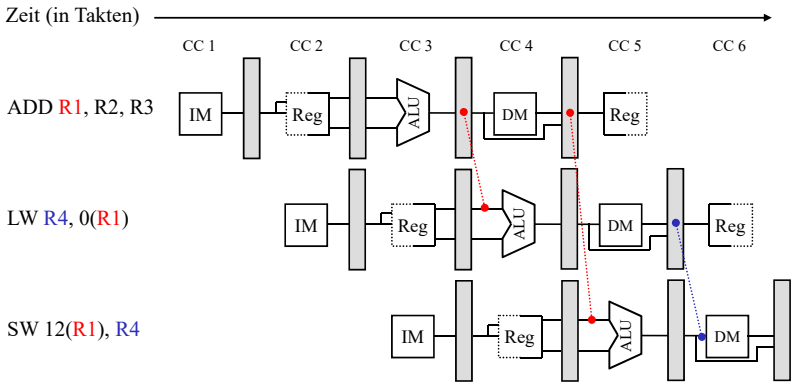
Das Beispiel zeigt auch, dass das neue Ergebnis auch für den übernächsten Befehl (AND) in rückgeführter Form zur Verfügung gehalten werden muss.

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 Folie zeigt die Forwarding-Pfade, die für dieses Problem erforderlich sind.

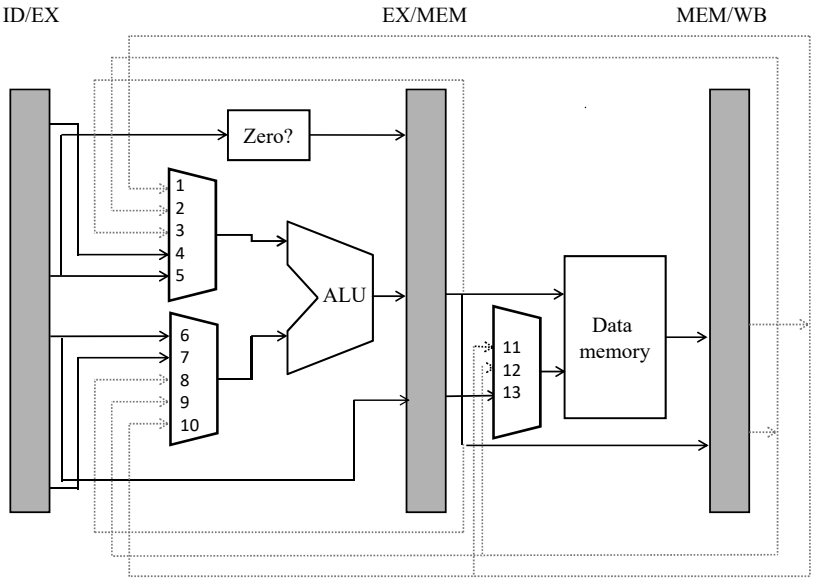


RAW-Hazard bei Befehlskette ADD, SUB

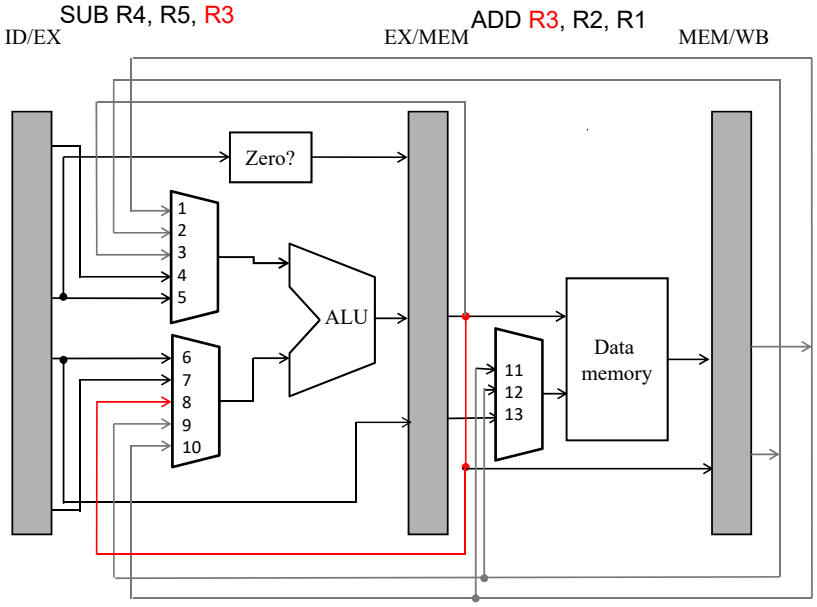
Nun können sehr unterschiedliche Datenabhängigkeiten zwischen aufeinanderfolgenden Befehlen auftauchen. Um all diesen gerecht zu werden, müssen die beiden Datenweg-Multiplexer vor der ALU mehr als zwei Eingänge bekommen und vor dem Eingang ins Daten-Memory muss auch ein Datenweg-Multiplexer eingefügt werden. Diese zusätzlichen Multiplexereingänge werden mit geeigneten Werten aus den Pipeline-Latches beschaltet, die ohne diese zusätzlichen Verbindungen nicht rechtzeitig an der ALU bzw. am Daten-Memory anliegen würden und so zu einem Stau der Pipeline führen würden. Diese Verbindungen nennt man Forwarding-Pfade.

Auf der folgenden Folie sind alle Forwarding-Pfade eingezeichnet, die mögliche RAW-Hazards beheben können. Wenn zum Beispiel die Befehlsfolge
ADD R3, R2, R1
SUB R4, R5, R3
zu bearbeiten ist, besteht eine Datenabhängigkeit durch die Verwendung von R3. R3 wird normalerweise erst in der WB-Phase des ersten Befehls in den Registersatz geschrieben, wird aber bereits in der ID-Phase des zweiten Befehls im Registersatz erwartet. Daher wird die Hardware den Multiplexereingang 8 verwenden, um diesen Wert bereits direkt nach der EX-Phase des ersten Befehls zu kopieren und im nächsten Takt in die EX-Phase des zweiten Befehls in den unteren Datenweg-Multiplexer der ALU einzuspeisen.

Forwarding-Pfade für Read after Write Hazards



Pipeline: ADD-Befehl in MEM-Phase, SUB-Befehl in EX-Phase



Einteilung von Daten Hazards in Kategorien

Ein Daten Hazard entsteht, wenn von zwei Operationen, die zu dicht aufeinander folgen, auf ein Datum zugegriffen wird. In unseren Beispielen waren das immer Register-Operationen. Das muss aber nicht so sein.

Viele Daten Hazards treten auch bei Speicher-Zugriffen auf. In der DLX-Pipeline allerdings nicht, da Speicherzugriffe immer in der vorgesehenen Reihenfolge ausgeführt werden.

Es gibt drei Typen von Daten Hazards:

a) RAW: read after write:

Das ist der Typ, den wir in den Beispielen hatten. Eine Operation schreibt, eine kurz darauf folgende versucht das Ergebnis zu lesen und würde noch das alte Ergebnis bekommen. Oft können diese Hazards durch forwarding vermieden werden.

b) WAW: write after write:

Zwei aufeinanderfolgende Operationen wollen an dieselbe Stelle schreiben. Wenn die erste länger dauert als die zweite, kann es sein, dass zuerst die zweite schreibt und dann die erste den neuen Wert wieder überschreibt. Das kann in unserer einfachen Integer-DLX nicht passieren, aber eine leichte Modifikation der DLX-Pipeline würde solche WAW Hazards bewirken:
Annahme: ALU-Befehle schreiben schon in der vierten Phase zurück ins Zielregister, Speicherzugriffe brauchen zwei Takte, also zwei MEM-Phasen.

LW	R1, 0(R2)	IF	ID	EX	MEM1	MEM2	WB
ADD	R1, R2, R3		IF	ID	EX	WB	

Hier wird zunächst R1 aus MEM geladen, danach soll die Addition R1 berechnen. Durch die Verzögerung wird das Additionsergebnis vom Ladebefehl überschrieben, die Addition wird gelöscht. (Anmerkung: Beispiel ist etwas konstruiert, da LW hier nicht sinnvoll ist, aber dennoch möglich).

Wenn in unterschiedlichen Phasen geschrieben wird, können auch Struktur-Hazards entstehen, da gleichzeitig zwei Befehle (die sich in unterschiedlichen Stufen ihrer Pipeline befinden) auf dieselbe Stelle schreibend zugreifen wollen.

c) WAR: write after read

Dieser Typ ist selten: Ein Befehl liest einen Wert, aber bevor er dazu kommt, hat sein Nachfolger bereits einen neuen Wert auf diese Stelle geschrieben.
Selten, weil das nur passiert, wenn sehr spät in der Pipeline gelesen wird und sehr früh geschrieben.
Das kann in unserer einfachen Integer-DLX nicht passieren, aber eine Modifikation der DLX-Pipeline würde solche WAR Hazards möglich machen: Speicherzugriffe brauchen drei Takte, also drei MEM-Phasen. Beim store wird der Lesezugriff erst gegen Ende des MEM3 gemacht, gleichzeitig wird das Register R2 erst dann ausgelesen.

SW	0(R1), R2	IF	ID	EX	MEM1	MEM2	*MEM3*
ADD	R2, R4, R3		IF	ID	EX	*WB*	

Während SW noch den MEM3 vorbereitet und R2 erst jetzt in den Zwischenspeicher gelesen wird, überschreibt ADD bereits R2.

d) RAR: read after read

Reine Lesezugriffe sind immer erlaubt, da sie den Zustand der Register nicht verändern. Sie erzeugen keine Hazards.

Daten Hazards, die zwangsläufig Staus verursachen

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 folgende Folie zeigt das Problem:
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. Das passiert immer, wenn auf einen Ladebefehl unmittelbar ein Registerbefehl folgt. 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. Das Problem liegt darin, dass der Ladebefehl das Datum zu spät erhält. Es kann keine frühere Phase durch forwarding angezapft werden.

Für diesen Fall benötigen wir spezielle Hardwaremechanismen, genannt Pipeline interlock.

Pipeline interlock:

staut die Pipeline mit einem Bubble:

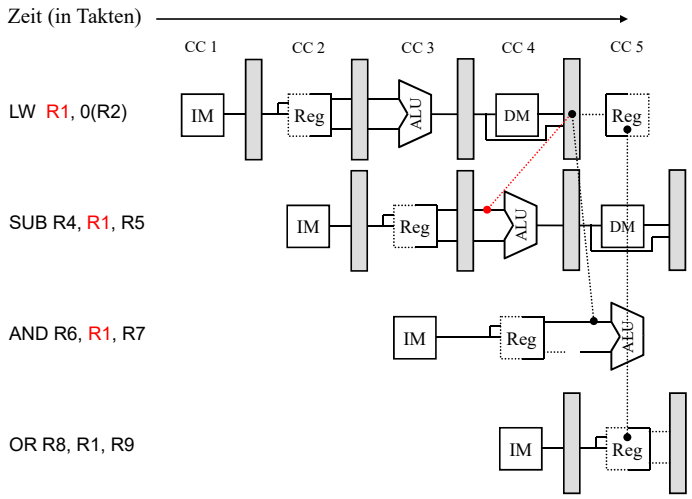
- nur die Instruktionen ab der Verbraucherinstruktion für den Datenhazard
- die Quellinstruktion und alle davor müssen weiterbearbeitet werden.

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

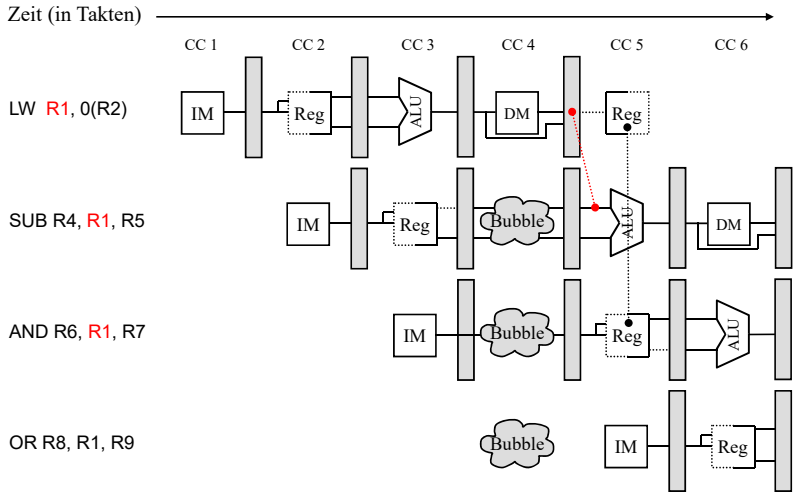
Im Beispiel sieht das so aus:

- alles verzögert sich um einen Takt ab der Bubble.
- In Takt vier wird keine Instruktion begonnen und also auch keine gefetcht.

Prozessfluss für load data hazard



Prozessfluss nach pipeline interlock



Pipeline diagramme für load data hazard und pipeline interlock:

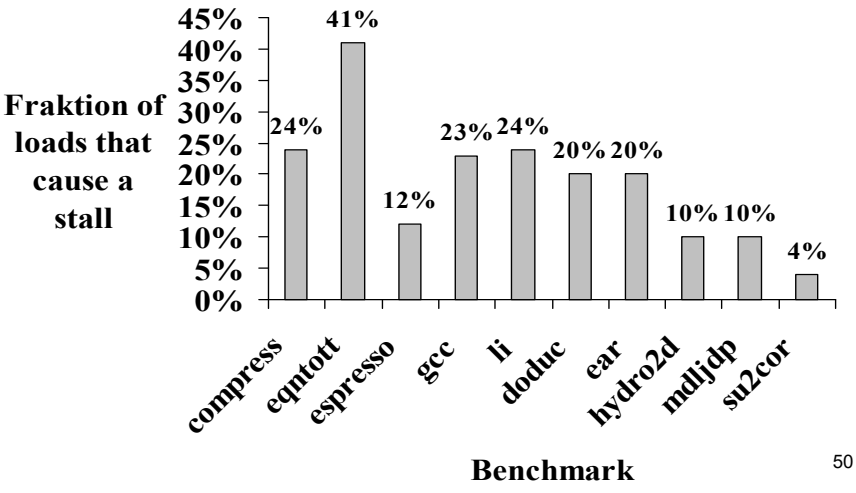
LW R1,0(R2)	IF	ID	EX	MEM	WB			
SUB R4,R1,R5		IF	ID	EX	MEM	WB		
AND R6,R1,R7			IF	ID	EX	MEM	VB	
OR R8,R1,R9				IF	ID	EX	MEM	WB

Nach Pipeline Interlock:

LW R1,0(R2)	IF	ID	EX	MEM	WB			
SUB R4,R1,R5		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

Statistische Auswertung von Lade-Staus aus Messungen

Die Häufigkeit von Load-Staus ist aus folgender Folie für die Spec-Benchmarks zu entnehmen:



Performanzverlust durch pipeline interlock

Der Verlust hängt davon ab, wie oft ein load data hazard im Mittel auftritt.

Beispiel, aus den spec-Benchmarks:
25% der Befehls sind loads.
In der Hälfte der Fälle benötigt die auf load folgende Instruktion den geladenen Operanden. Dieser Hazard produziert 1 Zyklus Verzögerung. Der Takt für volles Pipelining ist 1.2 ns.

Wieviel langsamer ist diese Maschine gegenüber der CPI_{pipe}=1,2ns Maschine?

CPI_{load} = 0,5*CPI_{pipe} + 0,5*2*CPI_{pipe} = 1,5*CPI_{pipe}=1,8ns (Verlangsamung durch pipeline interlock)

CPI_{gesamt} = 0,75*CPI_{pipe} + 0,25*CPI_{load} = 1,35 ns

Antwort: Interlock verlangsamt die Maschine um einen Faktor CPI_{pipe}/ CPI_{gesamt} = 1,2/1,35 = 0,88 auf 88% ohne Interlock.

Der speedup der pipeline sinkt hierbei auf (4,4ns/1,35ns) = 3,26 gegenüber 3,67 ohne pipeline interlock.

Hazard-Erkennung in der DLX Pipeline

Während der ID-Phase werden die Steuersignale erzeugt, die notwendig sind, um auf Hazards zu reagieren (entweder forwarding oder interlock).
Je eher in der Pipeline man eine solche Situation erkennt, desto eher kann man reagieren. Wichtig ist, dass nie eine Situation auftritt, bei der man eine Veränderung des Prozessorstatus schon vorgenommen hat, die man dann aufgrund von einem Stau wieder zurücknehmen muss. Dies wäre nämlich sehr aufwendig.

Beispiel: Wann ist forwarding möglich, und wann entsteht ein pipeline interlock?
Wir erkennen das Problem in der ID-Phase des Befehls, der u.U. gestallt werden muss. Die folgende Tabelle zeigt die Situationen, die auftreten können und die entweder durch forwarding oder durch ein Pipeline-Interlock behoben werden:

Das erste und zweite Szenario ist unkritisch: es gibt (Fall 1) entweder keine Datenabhängigkeit oder (Fall 2) die Register-Hardware löst diesen Hazard.
Das dritte Szenario (Fall 3) kann durch forwarding gelöst werden.

Das vierte Szenario (Fall 4) ist ein Load-Hazard mit Interlock und erfordert einen Stau des momentanen und aller darauffolgenden Befehle. Der Stau wird erkannt, wenn ADD in der ID-Phase ist und gleichzeitig LW in der EX-Phase (d.h. das Datum ist noch nicht ins LMD geladen).

Load-Hazard und Vermeidung

Szenarien LW R1,45(R2)	Beispiel Code	Vermeidungsstrategie

Fall 1: Keine Abhängigkeit zu R1 in den nächsten Befehlen	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6,R7
	OR R9,R6,R7
		Es kann kein Hazard entstehen, da R1 in keinem der drei auf LW folgenden Befehle als Eingabe vorkommt

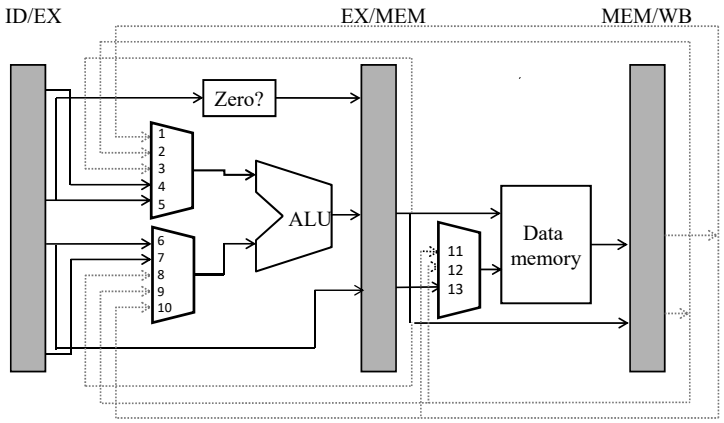
Fall 2: Datahazard: OR benötigt R1 in ID-Phase als Eingabe (durch Hardware im Register-zugriff beseitigt)	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6,R7
	OR R9,R6,R1
		R1 wird in WB von LW in der positiven Registerflanke in das Register geschrieben, in ID von OR in der negativen Flanke aus dem Register gelesen, daher kein Stau

Fall 3: Datahazard: SUB benötigt R1 in EX-Phase (durch Forwarding ermöglicht)	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6,R1
	OR R9,R6,R7
		Forwarding-Vergleicher erkennen Abhängigkeit von LW und SUB in der EX-Phase und leiten den Inhalt von LMD direkt nach ALU-Input über Forwarding-MUX, daher kein Stau

Fall 4: Datahazard: ADD benötigt R1 in EX-Phase, es wird ein Pipeline-Interlock notwendig	LW R1, 45(R2)
	ADD R5, R6, R1
	SUB R8, R6,R7
	OR R9,R6,R7
		Interlock-Vergleicher erkennen Abhängigkeit von LW und ADD in der ID-Phase und blockieren (stall) ADD,SUB,OR um 1 Takt, bevor ADD EX ausführen kann.

Fall 3: Data Hazard mit Forwarding-Erkennung

Die Forwarding-Hardware ist schon bekannt. Zusätzlich zu Vergleichern und Schaltnetzen, die wir brauchen, um die Bedingungen zu überprüfen, ob eine Situation für forwarding vorliegt, müssen wir die Multiplexer an den Eingängen der ALU vergrößern und einen weiteren Multiplexer am Eingang des Speichers einfügen. Diese Multiplexer werden mit den Registern verbunden, aus denen die Zwischenergebnisse im forwarding-Fälle genommen werden müssen.



Beispiele für die 13 möglichen Forwarding-Szenarien

Zeile 1: ADD R1, R2, R3
ADD R4, R1, R5

Zeile 2: ADD R1, R2, R3
ADD R4, R5, R1

Zeile 3: ADD R1, R2, R3
OR R7, R8, R9
ADD R4, R1, R5

Zeile 4: ADD R1, R2, R3
OR R7, R8, R9
ADD R4, R5, R1

Zeile 5: ADDI R1, R2, #1000
ADD R4, R1, R5

Zeile 6: ADDI R1, R2, #1000
ADD R4, R5, R1

Zeile7: ADDI R1, R2, #1000
OR R7, R8, R9
ADD R4, R1, R5

Zeile 8: ADDI R1, R2, #1000
OR R7, R8, R9
ADD R4, R5, R1

Zeile 9: LW R1, 1000(R2)
OR R7, R8, R9
ADD R4, R1, R5

Zeile 10: LW R1, 1000(R2)
OR R7, R8, R9
ADD R4, R5, R1

Zeile 11: ADD R1, R2, R3
SW 100(R4), R1

Zeile 12: ADDI R1, R2, #1000
SW 100(R4), R1

Zeile 13: LW R1, 1000(R2)
SW 100(R4), R1

Vergleich EX/MEM.IR <-> ID/EX.IR

Vergleich MEM/WB.IR <-> ID/EX.IR

Test der Bedingungen für die 13 mögliche Forwarding-Szenarien

Pipeline-Register in dem die erste Operation steht	Instruktionstyp der ersten Operation	Pipeline-Register, in dem die zweite Operation ist	Instruktionstyp der zweiten Operation	Ziel des Ergebnisses des Forwarding	Bedingung für den Forwarding-Fall: teste auf Gleicheit der Register-adressen im IR
1. EX/MEM	Register-Register-ALU	ID/EX	R-R-ALU, ALU-immediate, load, store, branch	3: Oberer ALU-Mux	EX/MEM.$IR_{16..20}$ = ID/EX.$IR_{6..10}$
2. EX/MEM	Register-Register-ALU	ID/EX	Register-Register-ALU	8: Unterer ALU-Mux	EX/MEM.$IR_{16..20}$ = ID/EX.$IR_{11..15}$
3. MEM/WB	Register-Register-ALU	ID/EX	R-R-ALU, ALU-immediate, load, store, branch	2: Oberer ALU-Mux	MEM/WB.$IR_{16..20}$ = ID/EX.$IR_{6..10}$
4. MEM/WB	Register-Register-ALU	ID/EX	Register-Register-ALU	9: Unterer ALU-Mux	MEM/WB.$IR_{16..20}$ = ID/EX.$IR_{11..15}$
5. EX/MEM	ALU-immediate	ID/EX	R-R-ALU, ALU-immediate, load, store, branch	3: Oberer ALU-Mux	EX/MEM.$IR_{11..15}$ = ID/EX.$IR_{6..10}$
6. EX/MEM	ALU-immediate	ID/EX	Register-Register-ALU	8: Unterer ALU-Mux	MEM/WB.$IR_{11..15}$ = ID/EX. $IR_{11..15}$
7. MEM/WB	ALU-immediate	ID/EX	R-R-ALU, ALU-immediate, load, store, branch	2: Oberer ALU-Mux	MEM/WB.$IR_{11..15}$ = ID/EX.$IR_{6..10}$
8. MEM/WB	ALU-immediate	ID/EX	Register-Register-ALU	9: Unterer ALU-Mux	MEM/WB.$IR_{11..15}$ = ID/EX.$IR_{11..15}$
9. MEM/WB	Load	ID/EX	R-R-ALU, ALU-immediate, load, store, branch	1: Oberer ALU-Mux	MEM/WB.$IR_{11..15}$ = ID/EX.$IR_{6..10}$
10. MEM/WB	Load	ID/EX	Register-Register-ALU	10: Unterer ALU-Mux	MEM/WB.$IR_{11..15}$ = ID/EX.$IR_{11..15}$
11. EX/MEM	Register-Register-ALU	ID/EX	Store	12: DM-Mux	EX/MEM.$IR_{16..20}$ = ID/EX.$IR_{11..15}$
12. EX/MEM	ALU-immediate	ID/EX	Store	12: DM-Mux	EX/MEM.$IR_{11..15}$= ID/EX.$IR_{11..15}$
13. EX/MEM	Load	ID/EX	Store	11: DM-Mux	EX/MEM.$IR_{11..15}$= ID/EX.$IR_{11..15}$

Fall 4: Load-Hazard mit Interlock-Erkennung

Die folgende Tabelle zeigt die Problemsituationen, in denen ein Interlock eingefügt werden muss, anhand der in den Registern stehenden Bits, die anhand der in den Pipeline-Registern mitgeführten Befehls-Opcodes und der entsprechenden Ziel- und Quellenregister erkannt werden können.
Ein Komparatornetzwerk vergleicht die Inhalte der beiden aufeinanderfolgenden Befehle und Registeradressen in der Pipeline, die in den Pipelineregistern ID/EX.IR und IF/ID.IR stehen. [$IR_{0...5}$] hält den OPCODE, [$IR_{6...10}$], [$IR_{11...15}$] die entsprechenden Ziel- und Quellregisteradressen.

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	**Opcode field of IF/ID ($IF/ID.IRp_{0..5}$)**	**Matching operand fields**
Load	Register-register ALU	$ID/EX.IR_{11..15} = IF/ID.IR_{6..10}$
Load	Register-register ALU	$ID/EX.IR_{11..15} = IF/ID.IR_{11..15}$

1. LW R1, 45(R2)
   ADD R5, R6, R1

2. LW R1, 45(R2)
   ADD R5, R1, R6

Compiler-Unterstützung bei Hazards

Viele Stau-Typen sind recht häufig

Beispiel: Berechne A = B + C (erzeugt pipeline interlock)

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

Pipeline Scheduling zur Vermeidung von Staus

Beispiel: Erzeuge gutes Maschinenprogramm für
A = B + C;
D = E - F;
durch geeignete Anordnung/Umstellung der Bearbeitungsreihenfolge:

```
LW    R2, B          / lade B
LW    R3, C          / lade C
LW    R5, E          / lade E (vorgezogen, das entschärft Konflikt mit R3)
ADD   R1, R2, R3     / berechne A, aber speichere noch nicht ab
LW    R6, F          / lade F
SW    A, R1          / jetzt A speichern, das entschärft den Konflikt mit R6
SUB   R4, R5, R6     / berechne D
SW    D, R4          / speichere D
```

Beide interlocks LW R3,C zu ADD R1,R2,R3 und LW R6,F zu SUB R4,R5,R6 sind nun eliminiert. Es gibt einen Hazard zwischen SUB und SW, aber der kann mit forwarding behoben werden.

Definition: Pipeline Scheduling ist die Umstellung der Reihenfolge von Befehlen zur Vermeidung von Pipeline-Staus. Das Verhalten des Programms muss dabei erhalten bleiben. Der Compiler ist dafür zuständig.

Pipeline Scheduling zur Vermeidung von Staus

Man kann an diesem Beispiel beobachten, dass die Benutzung unterschiedlicher Register für R2, R3 und R5, R6 notwendig ist. Insbesondere wegen der Befehlsumstellung würde eine doppelte Benutzung von R2 oder R3 für R5 zu falschem Verhalten führen.

Wir lernen daraus:
- Pipeline Scheduling kostet im allgemeinen zusätzliche Register.

Moderne Compiler benutzen Pipeline Scheduling in der Phase der *Lokalen Optimierung* in *basic blocks*. Das sind Code-Segmente linear nacheinander auszuführender Instruktionen ohne Ein- und Aussprünge, außer am Anfang und am Ende.

Für so einfache Pipelines wie unsere DLX (mit nur kurzen Stall-Verzögerungen, nur 1 bei load) ist dies ein adäquates Vorgehen. Wir müssen lediglich die Abhängigkeiten der Befehle als einen Graphen mit gerichteten Kanten zeichnen und müssen die Reihenfolge so festlegen, dass alle Abhängigkeiten erfüllt sind, ohne dass ein Stau-Konflikt auftaucht.

4. Kontrollhazards

Kontroll Hazards entstehen bei Verzweigungen (Branches).
Sie können aufwändig werden. Es wird aufgrund des condition flag entschieden was passiert:

- Ausgeführte Verzweigung (branch taken): Sprung zur Zieladresse
- Nicht ausgeführte Verzweigung (branch not taken, untaken) : PC+4

Problem: Erst in der Execute-Phase wird die Zieladresse berechnet und 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	stall	stall	IF	ID	EX	MEM	WB
Branch +2						IF	ID	EX	MEM
Branch +3							IF	ID	EX

Behandlung von Kontrollhazards

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 Staufall das Zielregister im 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 mit PC+4 geholt. Wir werden gleich sehen, wie man das ausnutzen kann, um Performance zu gewinnen.

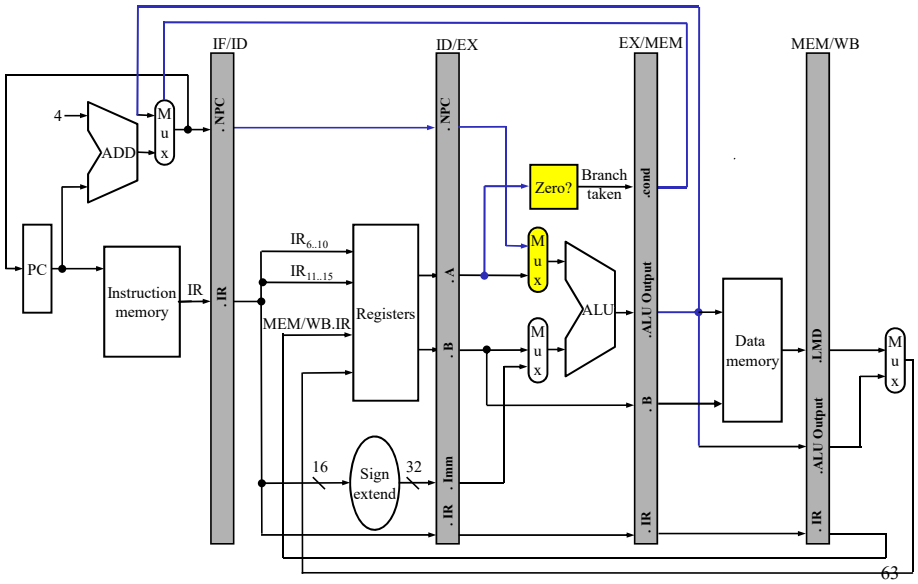
Aber zuerst: Wie können wir die Stau-Strafe vermindern?

1. Früher herausfinden, ob verzweigt wird oder nicht.
2. Sprungziel eher berechnen.

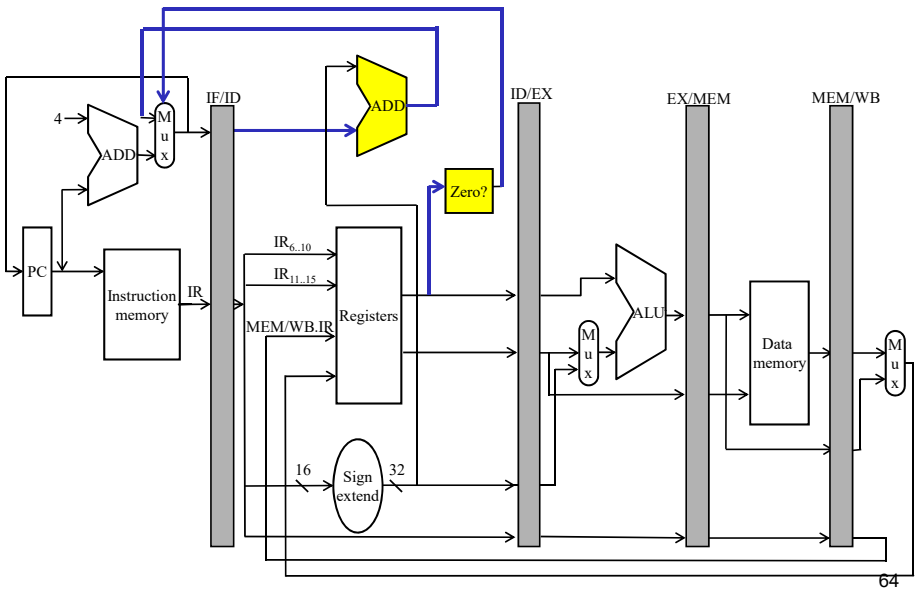
=> Zusätzliche ALU (nur Adder) für die Berechnung des Verzweigungsziels bereits in der ID-Phase. Cond wird auch bereits in der ID-Phase berechnet.

Die nächsten Folien zeigen den entsprechend veränderten Datenpfad.

DLX-Datenpfad mit Pipelining-Zwischenregistern



DLX-Datenpfad mit ID-Adder für Verzweigungs-Adressberechnung



Frühe Berechnung der Sprungadresse

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;
	ID/EX.IMM ⟵	($IF/ID.IR_{16}$)^{16}## $IF/ID.IR_{16..31}$
EX		
MEM		
WB		

Statistiken zum Sprungverhalten von Programmen

Weil Verzweigungsbefehle die Pipeline-Performance sehr stark beeinflussen können, stellen wir eine Analyse des Verhaltens von Programmen mit Sprüngen an:

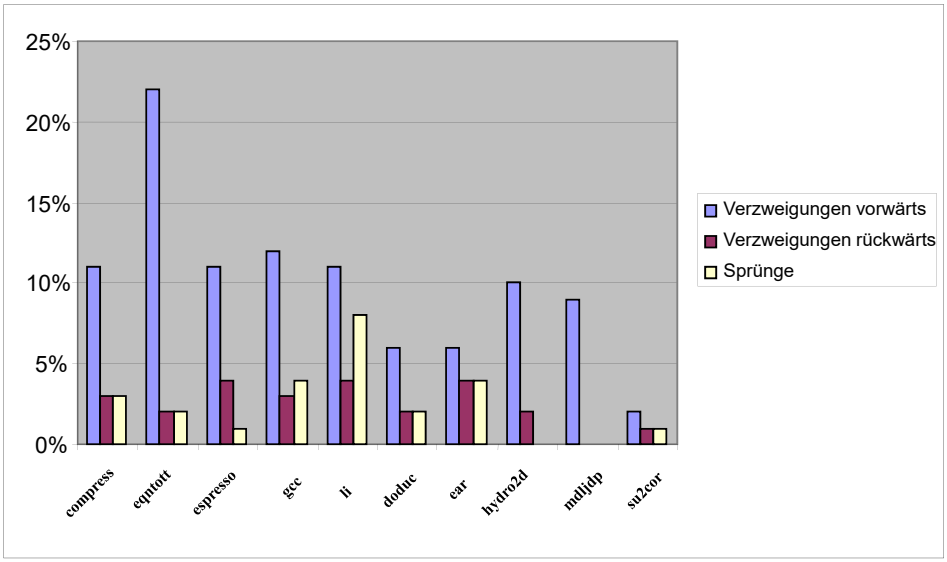
Auf der folgenden Folie sehen Sie das Verhalten der SPEC-Programme bezüglich Sprüngen (Jump) und Verzweigungen (branch), die in vorwärts und rückwärts unterteilt sind.

Integer Programme: 14-16% Branches, weniger Sprünge

Floating-Point Programme: 3-12% Branches, weniger Sprünge

Verhältnis Vorwärts zu rückwärts ca. 3:1
das scheint verwunderlich, da man ja bei Schleifen zurückspringt. Aber: bei If wird vorwärts gesprungen (2 Branches pro if-Statement). Daher dieses Verhältnis.

Statistiken zum Sprungverhalten im spec-Benchmark



Wie kann man die Verzweigungsstrafe minimieren?

Vier Strategien, die zur Compilezeit gewählt werden können:

- freeze
- predict not taken, treat as not taken
- predict taken, treat as taken
- 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 geholten und dekodierten Befehle von der Hardware in no-ops umgewandelt, wodurch alle temporären Register und alle GPRs erhalten bleiben, bis das Sprungziel neu geholt worden ist.

Annahme: Treat as not taken (nehme an, es gebe keinen Sprung)

Nicht ausgeführter Branch	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	WB	

Ausgeführter Branch	IF	ID	EX	MEM	WB				
Branch +1 (falscher PC)		IF	No-op	No-op	No-op	No-op			
Sprungziel (jetzt richtig)			IF	ID	EX	MEM	WB		
Sprungziel +1				IF	ID	EX	MEM	WB	

Vorteil: Keine Strafe, wenn tatsächlich kein Sprung benötigt wird.
Nachteil: Verzögerung, wenn ein Sprung notwendig ist

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 stalling) weitergemacht. Im Falle, dass nicht gesprungen werden soll, werden die fälschlich geholten und dekodierten Befehle von der Hardware in no-ops umgewandelt, wodurch alle temporären Register und alle GPRs erhalten bleiben, bis neu gelesen 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 Diagramm auf der folgenden Folie so nicht für die DLX.

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

Annahme: Treat as taken (immer Sprungannahme)

Ausgeführter Branch	IF	ID	EX	MEM	WB			
Sprungziel		IF	ID	EX	MEM	WB		
Sprungziel +1			IF	ID	EX	MEM	WB	
Sprungziel +2				IF	ID	EX	MEM	WB

Nicht ausgeführter Branch	IF	ID	EX	MEM	WB			
Sprungziel (falsches Sprungziel)		IF	ID	No-op	No-op	No-op		
Branch +1 (jetzt korrigiert)			IF	ID	EX	MEM	WB	
Branch +2				IF	ID	EX	MEM	WB

4. Delayed branch (verzögerter Sprung)

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 Compíler Scheduling 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.

Delayed branch Technik (verzögerter Sprung)

Nicht ausgeführteVerzweigung	IF	ID	EX	MEM	WB					
Branch delay instruction (unabhängig)	IF	ID	EX	MEM	WB					
Instruction *i* +1		IF	ID	EX	MEM	WB				
Instruction *i* +2			IF	ID	EX	MEM	WB			
Instruction *i* +3				IF	ID	EX	MEM	WB		

AusgeführteVerzweigung	IF	ID	EX	MEM	WB					
Branch delay instruction		IF	ID	EX	MEM	WB				
Branch target			IF	ID	EX	MEM	WB			
Branch target +1				IF	ID	EX	MEM	WB		
Branch target +2					IF	ID	EX	MEM	WB	

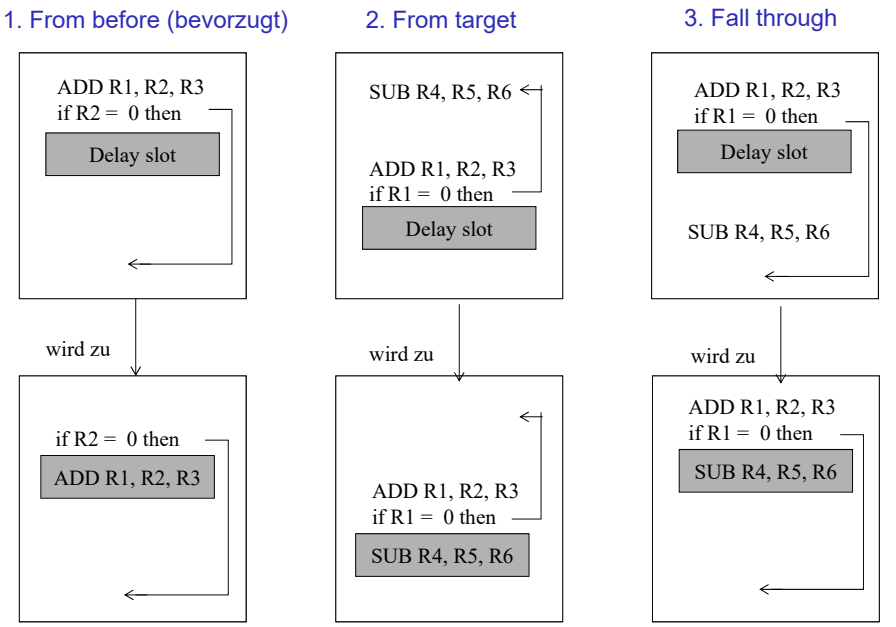
Bei delayed branching wird nach dem branch-Befehl eine unabhängige Instruktion im delay slot eingefüllt und ausgeführt. In der Pipeline wird bei ID der Sprungbefehl (mit/ohne Verzweigung) berechnet. Dadurch entsteht kein stall.

Delayed branch Technik (verzögerter Sprung)

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

Diese Möglichkeiten werden auf der folgenden Folie illustriert:



Delayed branch Technik (verzögerter Sprung)

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:

From target: 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:

1. Die Instruktion muss kopiert werden, weil das Sprungziel auch von einer anderen Stelle aus angesprungen werden könnte.

2. Die Instruktion darf im Falle, dass der Sprung nicht ausgeführt wird, nicht falsch sein, d. h. das Zielregister muss in diesem Falle redundant sein.

3. Die Instruktion darf nicht noch einmal ausgeführt werden, falls sie durch den Schedule nach dem Branch ausgeführt wird.

From fall through: 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 R4 ein Register ist, dass zu dieser Zeit sonst von niemandem benutzt wird.

Delayed branch Technik (verzögerter Sprung)

Worin liegt der Vorteil gegenüber dem normalen *treat as not taken?*

1. *From before* bringt immer Vorteil
2. Man kann Nutzen daraus ziehen, wenn man etwas über die Wahrscheinlichkeit sagen kann, mit der der Sprung ausgeführt wird.

Scheduling-strategie:	*Bedingung für legale Ausführung:*	Wann wird die Performanz verbessert?
From before	Darf nicht auf das Register schreiben, wenn die Bedingung abgefragt wird	immer
From target	Umgestellter Befehl muss OK sein, auch wenn nicht gesprungen wird	Wenn gesprungen wird (kann das Programm verlängern, da Kopie des eingefügten Befehls)
From fall through	Befehl muss OK sein, auch wenn gesprungen wird	Wenn nicht gesprungen wird (evtl. Kopie des eingefügten Befehls)

Delayed branch Technik (verzögerter Sprung)

Je tiefer die Pipeline wird, desto geringer wird der Performancegewinn durch delayed branches. Für unsere DLX mit einem delay-slot gilt das noch nicht, denn für Ein-slot-Systeme ist der Delayed Branch attraktiv wegen guter Performance bei geringem Hardwareaufwand.

Das Einzige, was man braucht, ist eine zusätzliche Kopie des PC. Warum?

Wenn bei der Ausführung des Befehls im delay-slot ein Interrupt auftritt, muss im Anschluss an die Interrupt-Behandlung der Befehl im delay-slot neu ausgeführt werden und der Befehl am Verzweigungsziel danach. Diese haben aber keine aufeinanderfolgenden Adressen. Daher müssen beide Adressen in zwei PCs gehalten werden, und in der Pipeline mitwandern, bis der Befehl im delay-slot fertig ist.

Bei heutigen Maschinen wird mehr Wert auf Hardware-Unterstützung bei der Verzweigungsvorhersage (branch prediction) gelegt. Dafür wird aber der delayed branch nicht mehr immer unterstützt.

Cancelling branch (Sprungabbruch)

Zusätzlich zur Instruktion wird dem Maschinenprogramm vom Compiler Informationen über das prognostizierte Sprungverhalten mitgegeben. Solange diese Prognose richtig liegt, wird der Befehl nach dem Sprung ausgeführt. Wenn sie falsch liegt, wird er gecancelt, d. h. durch einen no-op ersetzt.

Diese Technik nutzt ebenfalls die Möglichkeit der Prognose von Sprungwahrscheinlichkeiten zur Compilezeit aus.

Sie hat aber auf der anderen Seite nicht den Nachteil, das das Zielregister des Befehls im delay-slot redundant sein muss wie in der vorherigen Compiler Scheduling Technik.

Die meisten Maschinen bieten beide Möglichkeiten: nicht-cancelling und cancelling (meist cancel if not taken) delayed branches. Dies Kombination hat sich in der Praxis bewährt.

Predict taken cancelling branch (Sprungabbruch)

AusgeführteVerzweigung	IF	ID	EX	MEM	WB	(Vorhersage stimmt)	
Branch delay instruction (i+1)	IF	ID	EX	MEM	WB		
Branch target		IF	ID	EX	MEM	WB	
Branch target +1			IF	ID	EX	MEM	WB
Branch target +2				IF	ID	EX	MEM

Nicht ausgeführteVerzweigung	IF	ID	EX	MEM	WB	(Vorhersage stimmt nicht)	
Branch delay instruction (i + 1)		IF	ID	idle	idle	idle	(Abbruch)
Instruction *i* +2			IF	ID	EX	MEM	WB
Instruction *i* +3				IF	ID	EX	MEM
Instruction *i* +4					IF	ID	EX