Pipelining beim DLX 560 Prozessor

Pipelining : Implementierungstechnik
Vielfältig angewendet in der Rechnerarchitektur.

Pipelining macht CPUs schnell.

Pipelining ist wie Fließbandverarbeitung.

Hintereinanderausführung von **Stufen der Pipeline, pipe stages.**

Vergleich Autoproduktion: Maximiert den Durchsatz, d.h. soviele 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. Durchsatz.**

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.

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.

Im Idealfall ist die Taktzykluszeit einer Maschine mit Pipelining

$$\frac{\textit{Zeit für die Verarbeitung ohne Pipeling}}{\textit{Anzahl der Stufen}}$$

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.

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.

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

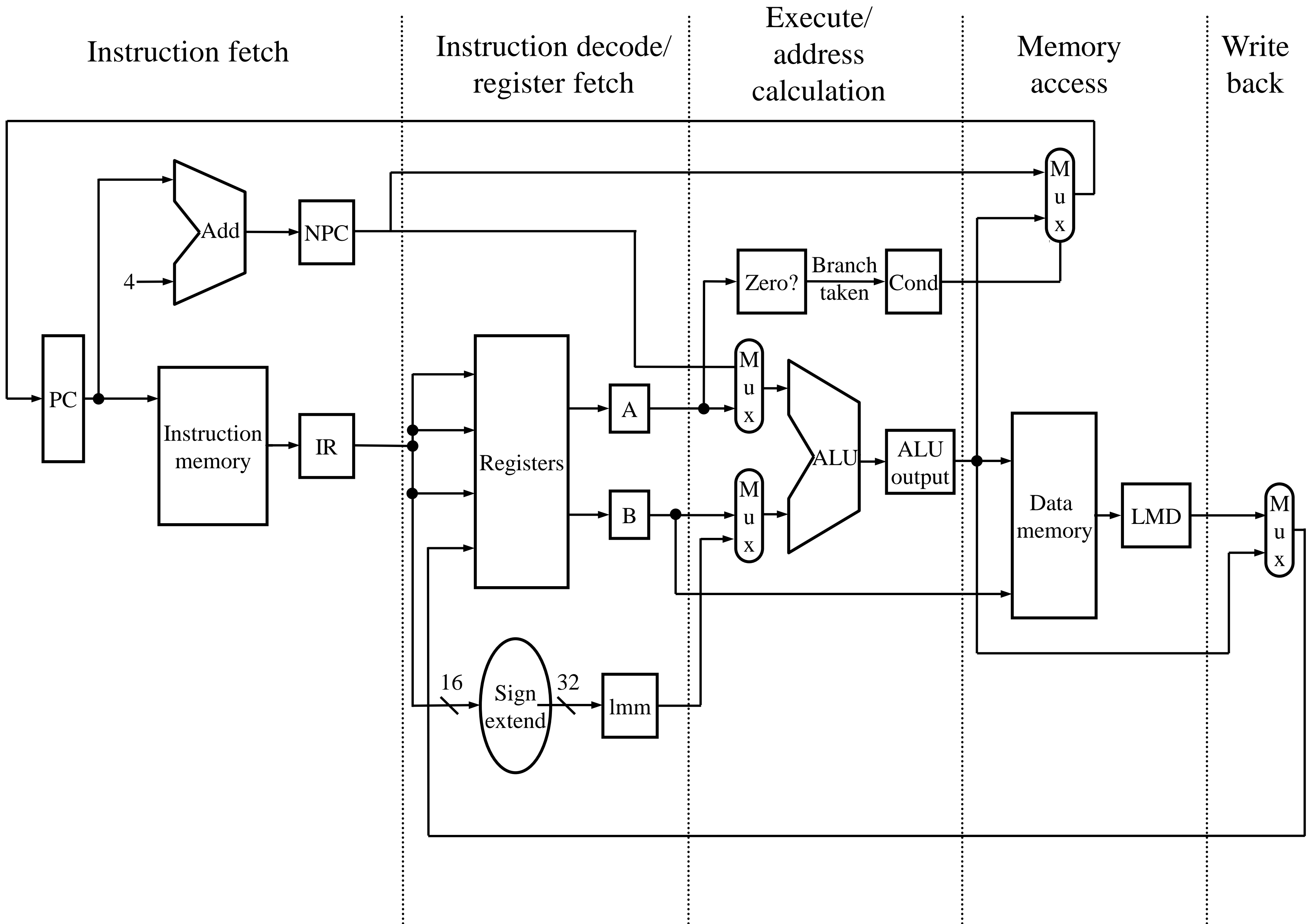
- **für den Benutzer unsichtbar.**

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 Pipelinig 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)

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	**<--**	$\mathbf{Regs[IR_{6..10}]}$
B	**<--**	$\mathbf{Regs[IR_{11..15}]}$
IMM	**<--**	$\mathbf{((IR_{16})^{16}\#\#IR_{16..31}}$

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.

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:

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..32 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 B 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 Address Bestimmung 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]**
PC	**←**	**NPC**

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

Speicher Befehl (store):

Mem[ALUoutput]	**←**	**B**
PC	**←**	**NPC**

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:

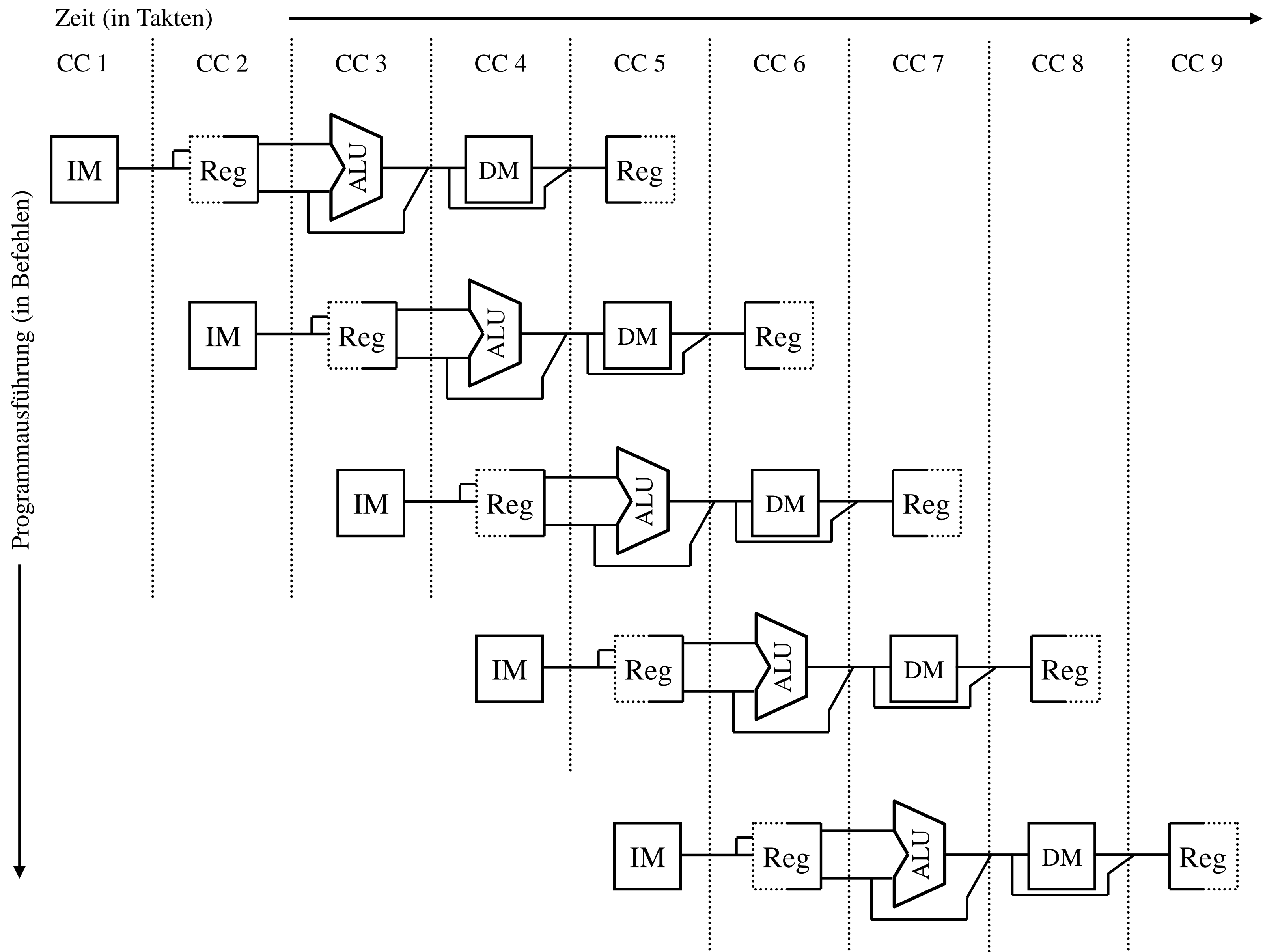
$$\mathbf{Regs[IR_{11..15}] \quad <\!-\!- \quad LMD}$$

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% (so wie in den Untersuchungen im letzten Kapitel geschehen) 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 mit **Pipelining** bezeichnet. Das resultiert in einem Ausführungsmuster wie auf der folgenden Folie dargestellt.



Vereinfachte Darstellung des DLX-Datenpfads

Pipeline Diagramm

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

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. 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. 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. Die Register werden im ID und im WB-Zyklus benutzt. Sie werden später im Studium sehen, wie man zwei Reads und ein Write in einem Zyklus ausführen kann.

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.

Problem Verzweigung

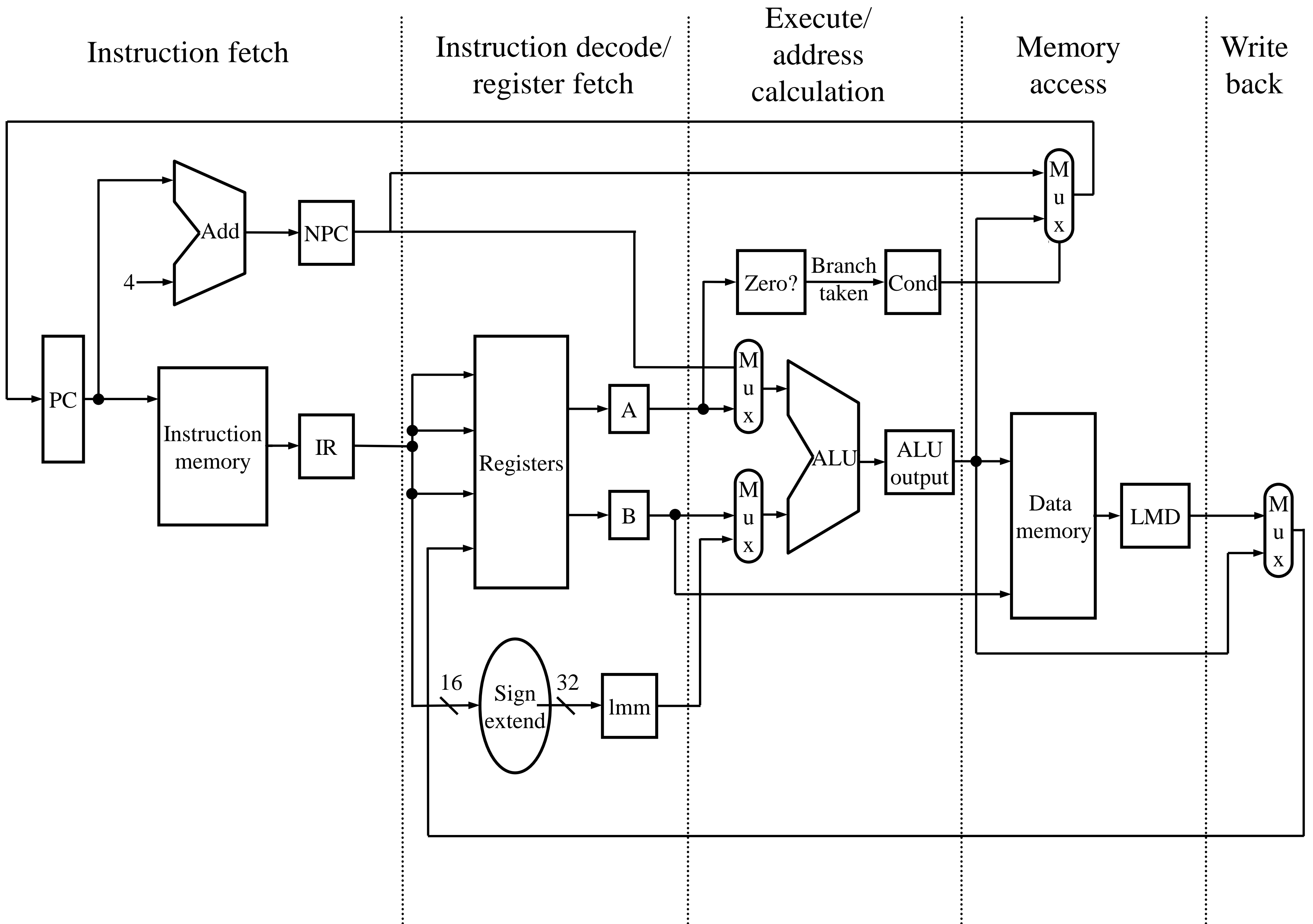
verändert PC aber erst in MEM-Phase. Aber das Inkrementieren passiert bereits in IF-Phase. Wie damit umzugehen ist, wird später dargestellt.

Jede Stufe ist in jedem Takt aktiv.

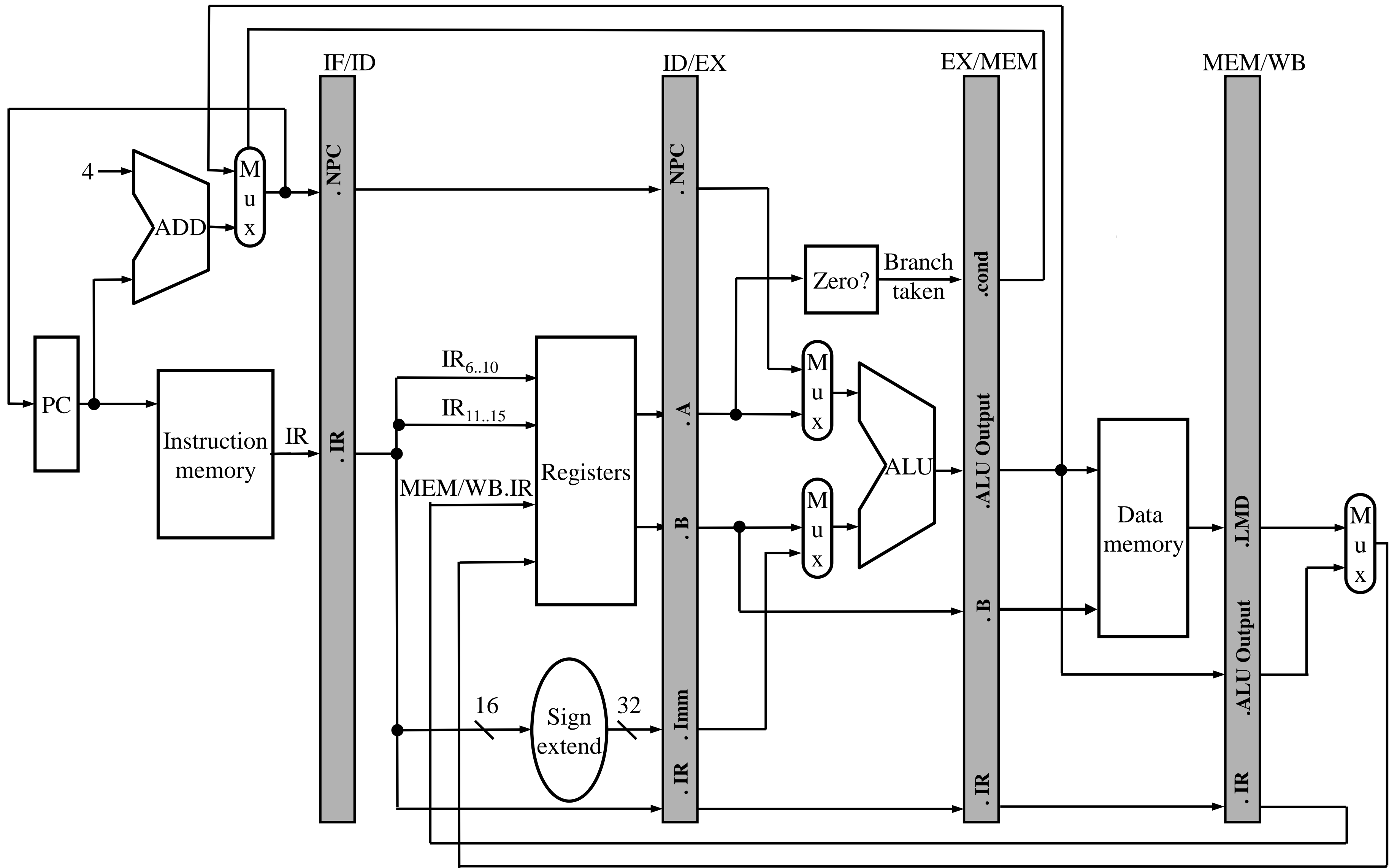
Jede Kombination von gleichzeitig aktiven Stufen muß möglich sein.

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 Stufen bezeichnet, zwischen denen sie liegen.



DLX-Datenpfad mit Taktzyklen (ohne Pipelining)


IF/ID
ID/EX
EX/MEM
MEM/WB
4
ADD
M
u
x
PC
Instruction
memory
IR
. NPC
. IR
. NPC
IR6..10
IR11..15
MEM/WB.IR
Registers
. A
. B
Zero?
Branch
taken
.cond
M
u
x
ALU
M
u
x
.ALU Output
. B
Data
memory
.LMD
.ALU Output
M
u
x
16
Sign
extend
32
. Imm
. IR
.IR
. IR

Alle temporären Register aus unserem ersten Entwurf können jetzt in diesen Registern 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 Folie 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.

Performance Verbesserung durch Pipelining

Durchsatz 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: vier Zyklen für ALU- und branch-Operationen, fünf für Speicher-Operationen. ALU 40%, branch 20%, load/store 40%. 1 GHz Takt. 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 =
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.

$$speedup = \frac{Zeit\ für\ ungepipelinete\ Ausführung}{Zeit\ für\ gepipelinete\ Ausführung} =$$

$$= \frac{4{,}4ns}{1{,}2ns} = 3{,}67$$

Soweit 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:

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:

Strukturhazards 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.

Datenhazards 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.

Steuerungshazards, Control hazards treten auf bei Verzweigungen in der Pipeline oder anderen Operationen, die den PC verändern.

Hazards führen zu eine **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, währen 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 geholt, solange der Stau dauert.

Performance von Pipelines mit Staus (stall)

$CPI_{pipelined}$ = Ideal CPI + Pipeline stall clock cycles per instruction

= 1 + Pipeline stall clock cycles per instruction

Wenn man mal vom Pipeline-overhead absieht, ist der

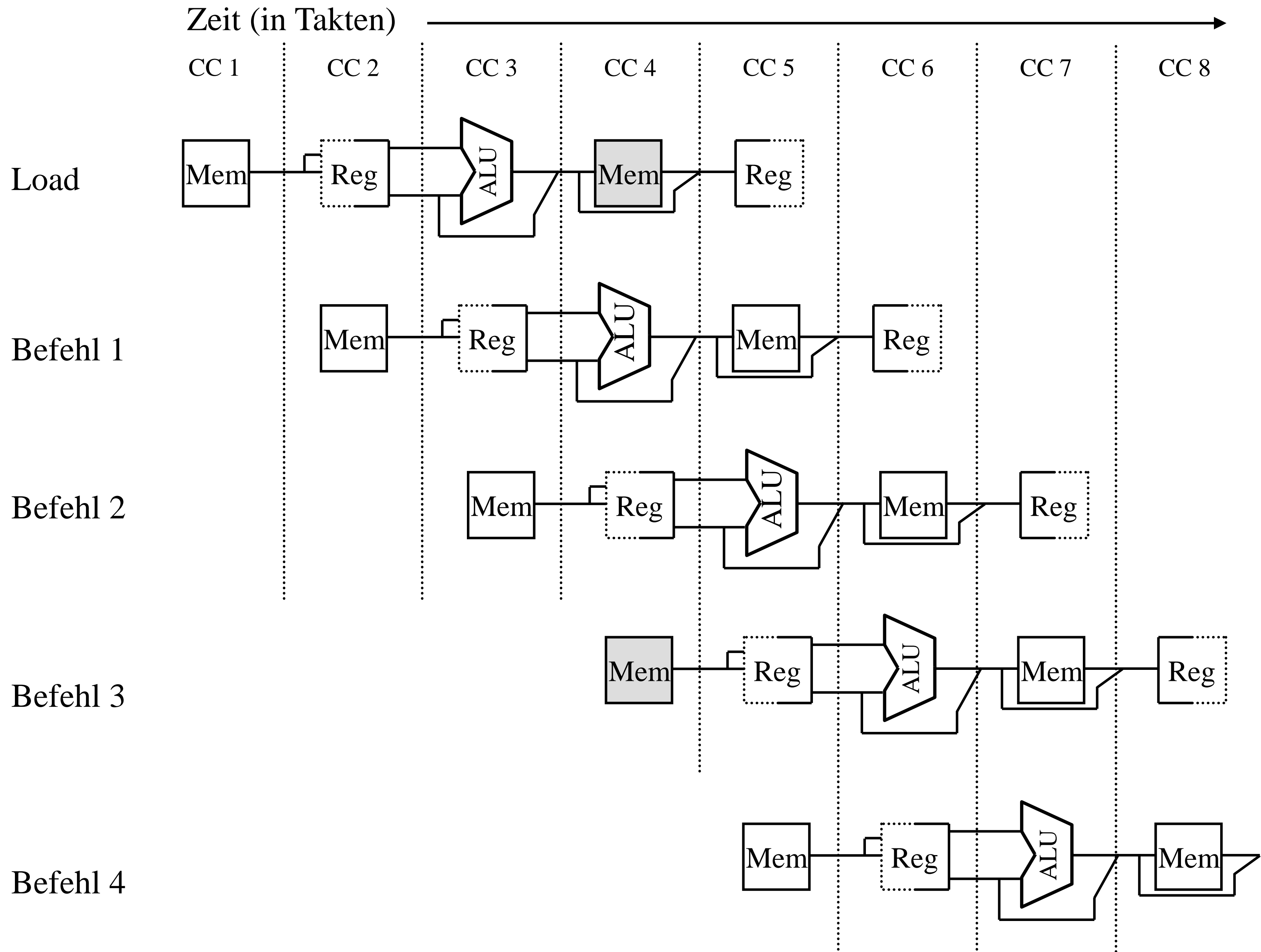
$$speedup = \frac{CPI_{unpipelined}}{1 + pipeline\ stall\ cycles\ per\ instruction}$$

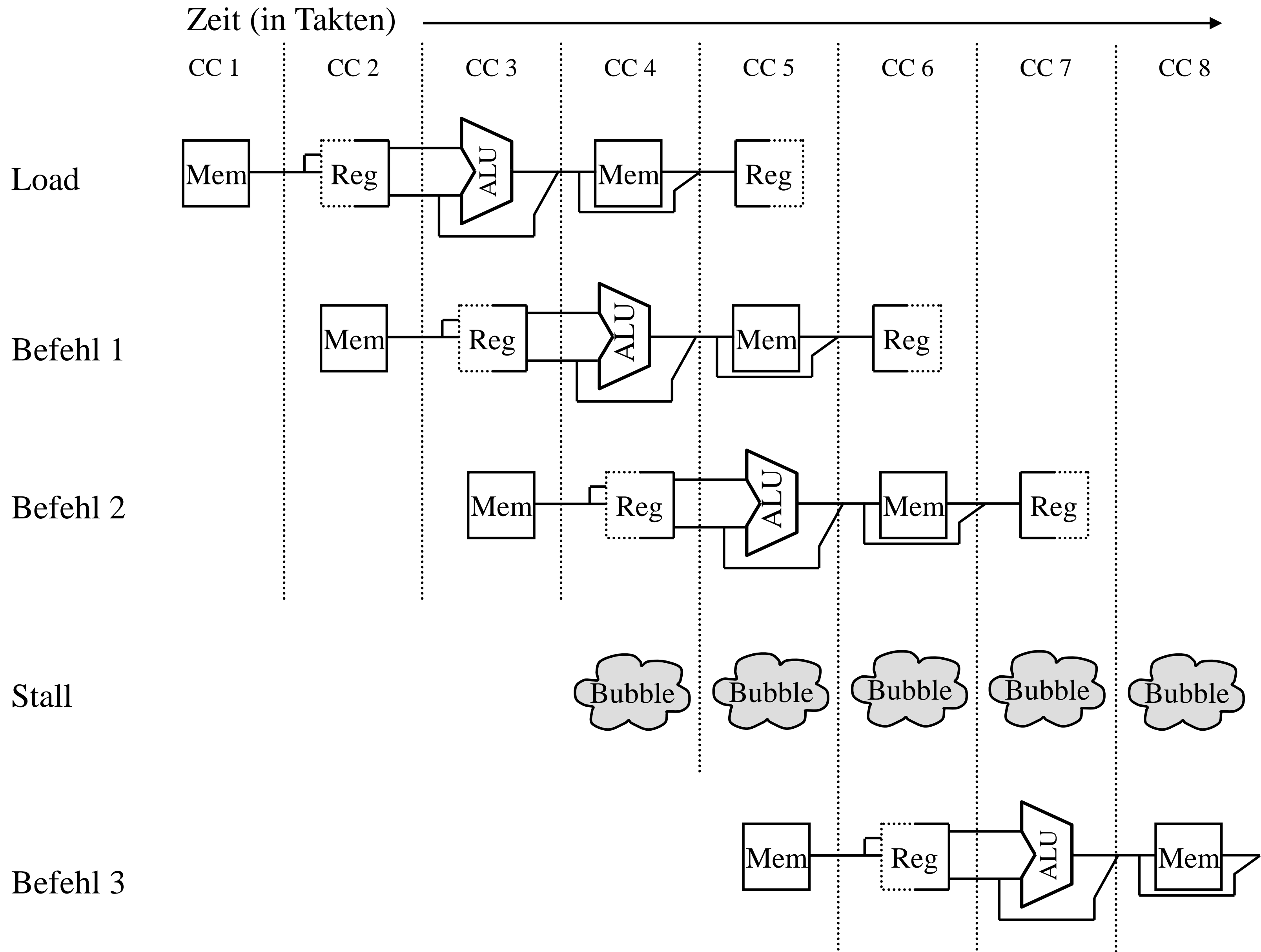
Der wichtigste Spezialfall ist der, bei dem alle Befehle gleichviel Takte brauchen.

$$speedup = \frac{Pipeline\ Tiefe}{1 + pipeline\ stall\ cycles\ per\ instruction}$$

Strukturhazards

Die überlappende Arbeit an allen Phasen der Pipeline gleichzeitig erfordert, dass alle Resourcen 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 Registerfile nur einen Schreibvorgang pro Takt erlaubt, aber zwei aufeinanderfolgende Befehle in unterschiedlichen Phasen beide ein writeback in ein Register ausführen wollen.
Wenn ein Strukturhazard 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 Strukturhazard. 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.


Zeit (in Takten)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
Load
Befehl 1
Befehl 2
Befehl 3
Befehl 4
Mem
Reg
ALU


Zeit (in Takten)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
Load
Mem
Reg
ALU
Mem
Reg
Befehl 1
Mem
Reg
ALU
Mem
Reg
Befehl 2
Mem
Reg
ALU
Mem
Reg
Stall
Bubble
Bubble
Bubble
Bubble
Bubble
Befehl 3
Mem
Reg
ALU
Mem

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 zeigt die folgende Folie.

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

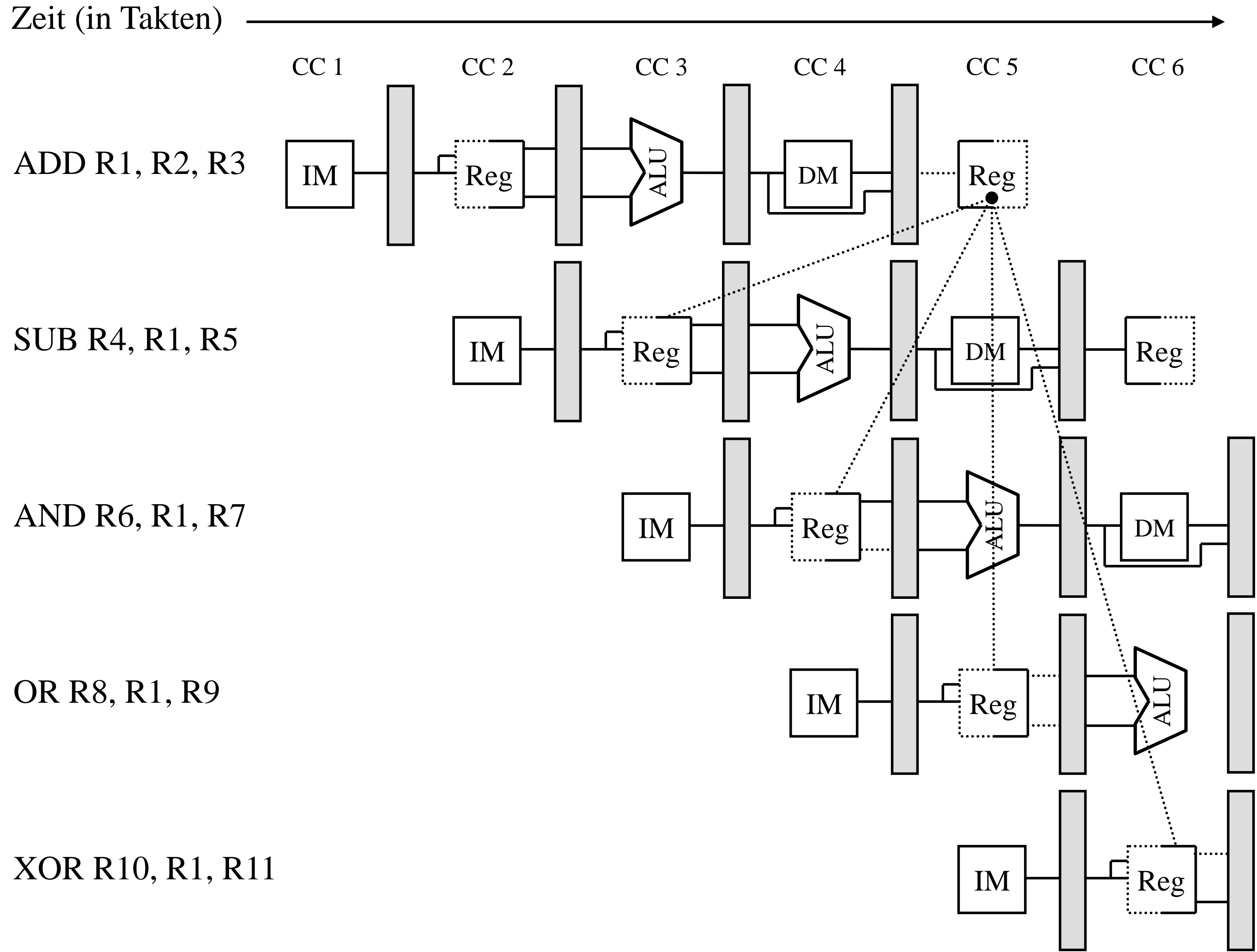
Daten Hazards

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.


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

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.

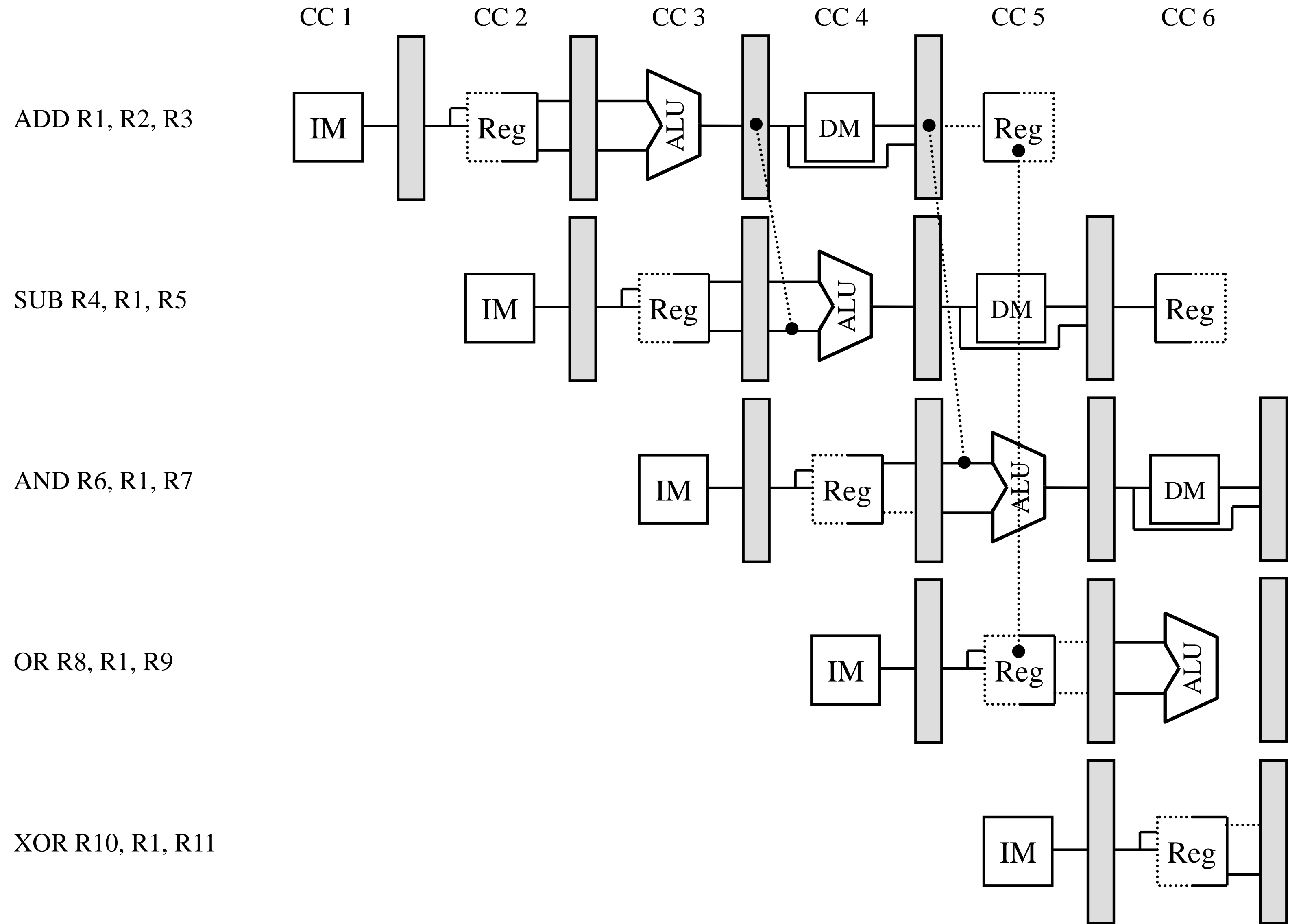
Wie vermeiden wir nun stalls bei SUB und AND?

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
IM
Reg
ALU
DM
Reg
IM
Reg
ALU
DM
IM
Reg
ALU
IM
Reg

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.

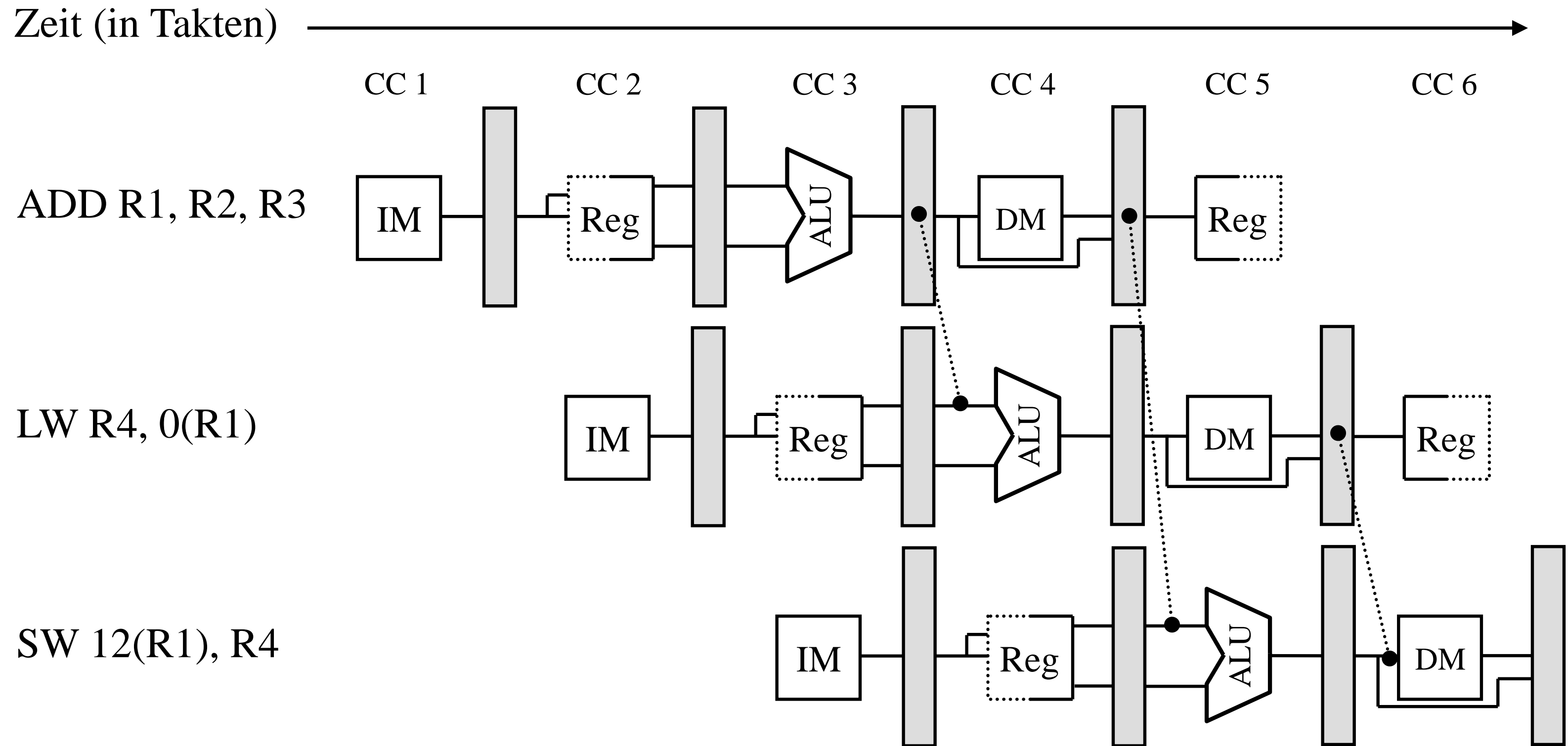
Das Beispiel zeigt auch, daß das neue Ergebnis auch für den übernächsten Befehl (AND) in rückgeführter Form zur Verfügung gehalten werden muß.

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


Zeit (in Takten)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
ADD R1, R2, R3
LW R4, 0(R1)
SW 12(R1), R4
IM
Reg
ALU
DM
Reg
IM
Reg
ALU
DM
Reg
IM
Reg
ALU
DM

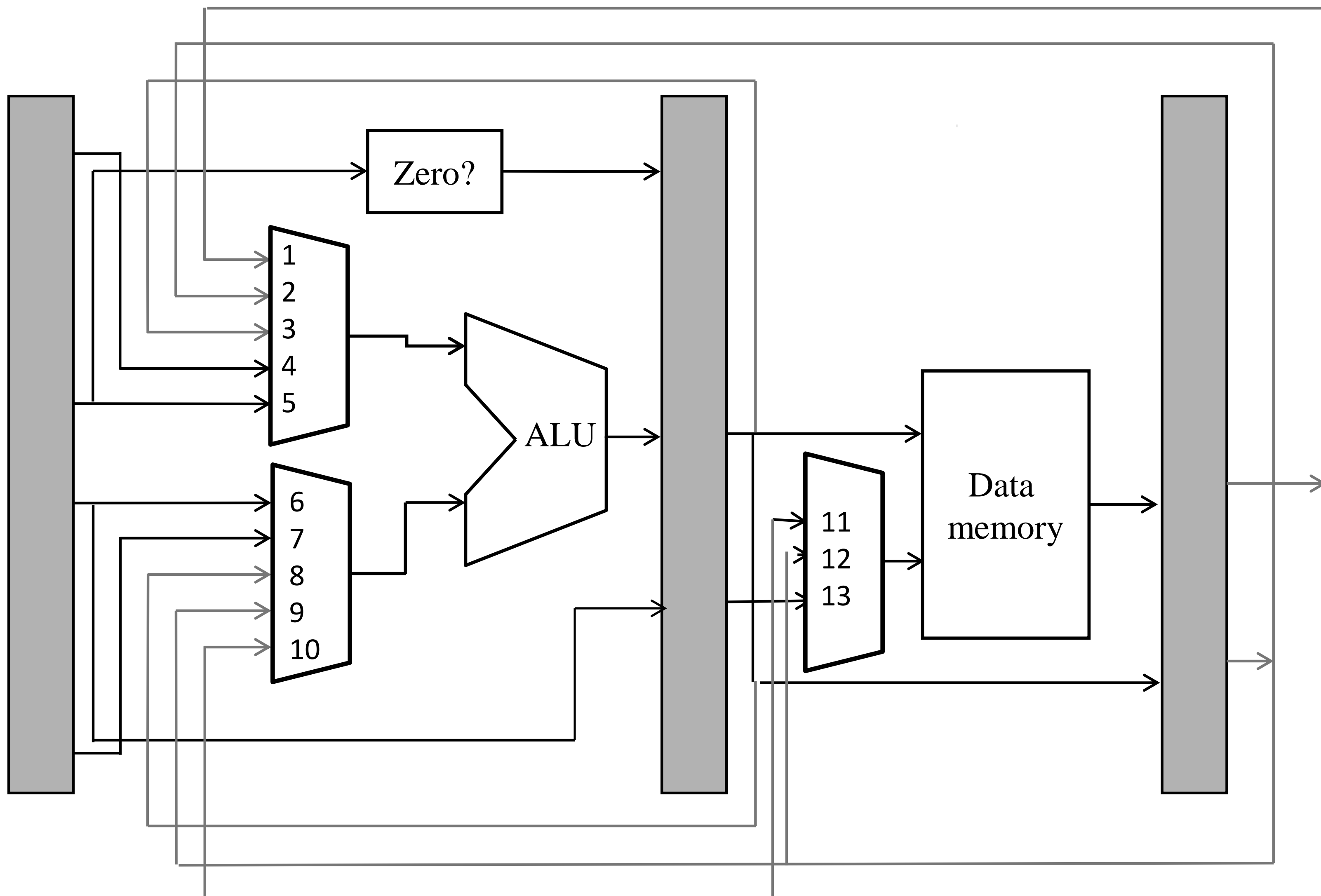
Erforderliche Forwarding-Pfade für RAW-Hazards

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.


ID/EX
EX/MEM
MEM/WB
Zero?
1
2
3
4
5
ALU
6
7
8
9
10
11
12
13
Data memory

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:

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.

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 möglich machen: 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**	

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

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 die obige Modifikation der DLX-Pipeline würde solche WAW Hazards möglich machen:
Speicherzugriffe brauchen zwei Takte, also zwei MEM-Phasen.
Beim store wird der Lesezugriff erst gegen Ende des MEM2 gemacht.

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

RAR ist kein Hazard

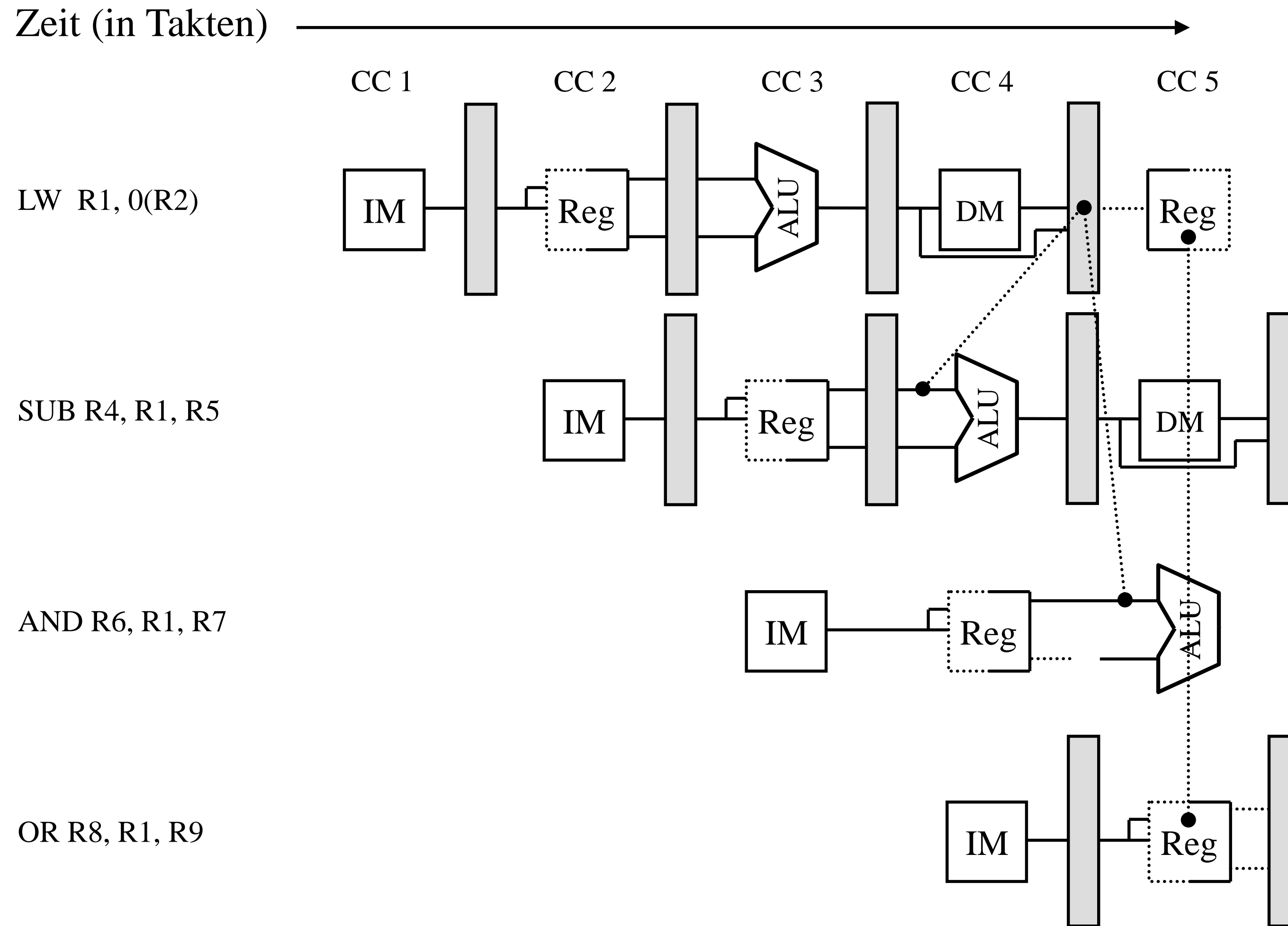
Daten Hazards, die Staus verursachen müssen

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 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.
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**.


Zeit (in Takten)
CC 1
CC 2
CC 3
CC 4
CC 5
LW R1, 0(R2)
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
IM
Reg
ALU
DM
Reg
IM
Reg
ALU
DM
IM
Reg
ALU
IM
Reg

Pipeline interlock staut die Pipeline

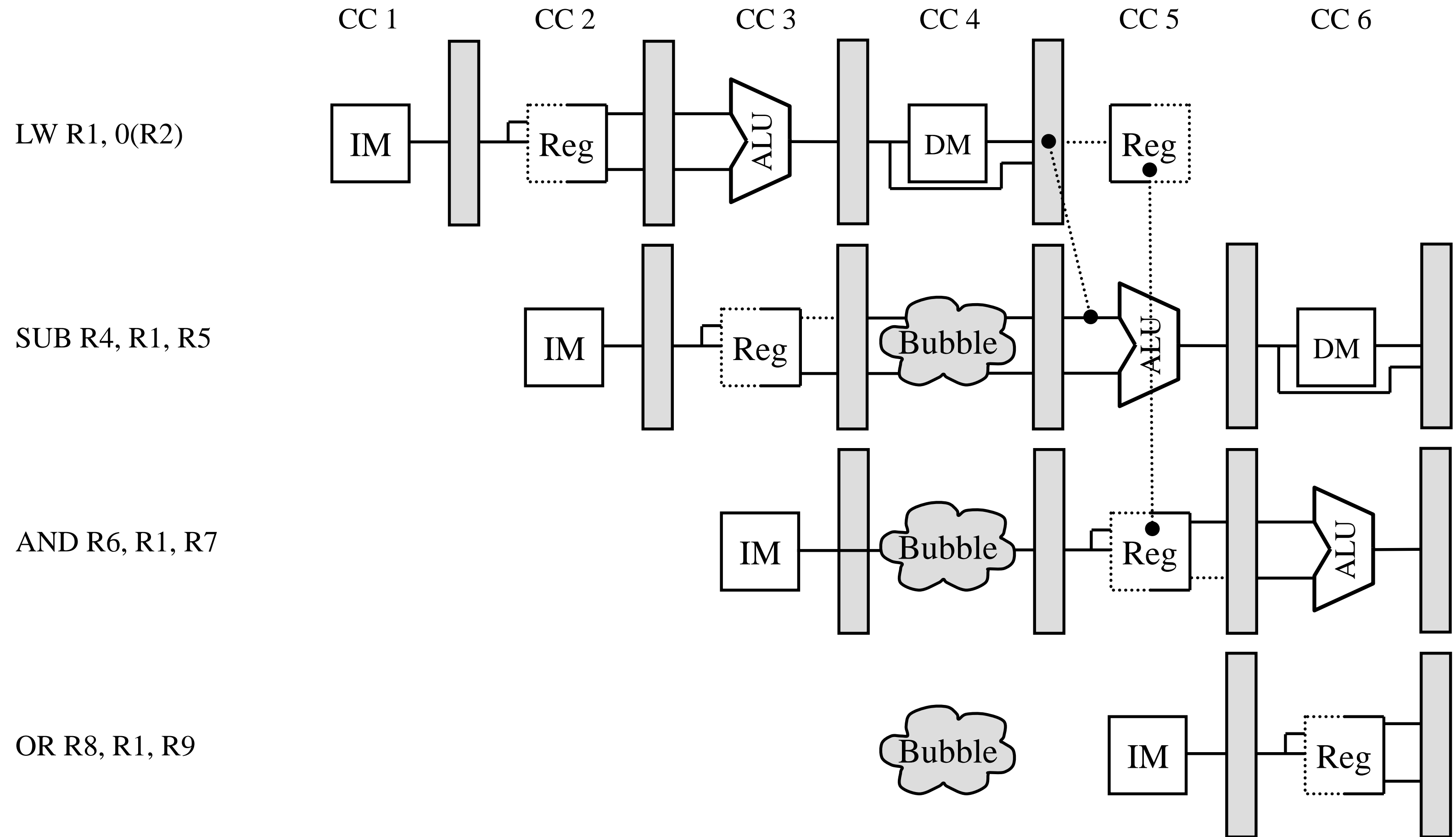
nur die Instruktionen ab der Verbraucherinstruktion für den Datenhazard

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.


Zeit (in Takten)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
LW R1, 0(R2)
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
IM
Reg
ALU
DM
Bubble

LW R1,0(R1)	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	

LW R1,0(R1)	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

Beispiel:
30% der Befehls sind loads.
In der Hälfte der Fälle benötigt die auf load folgende Instruktion den geladenen Operanden.
Dieser Hazard produziert 1 Zyklus Verzögerung.
Wieviel langsamer ist diese Maschine gegenüber der idealen CPI=1 Maschine?

$CPI_{nach\ load} = 0{,}5*1 + 0{,}5*2 = 1{,}5$

$CPI = 0{,}70*1 + 0{,}30*\ 1{,}5 = 1{,}15$

Antwort: Die ideale Maschine ist um einen Faktor 1,15 schneller.

In der Regel gibt es bei Integer-Programmen häufiger einen load-Stau als bei FP-Programmen. Das liegt daran, daß FP-Programme oft regulärer aufgebaut sind.

Man nennt den Zeitschlitz nach einem load den load-slot.

Nach globaler Optimierung sind häufig viele der load slots bereits belegt. Man braucht aber leere load slots für das Scheduling.
Typische load-Frequenz in diesen Programmen ist zwischen 19 und 34%. Im Durchschnitt 24%

Die CPI wächst hier um zwischen 0,01 und 0,15 durch load Staus.