Kapitel 10 (5. Teil MIPS)
Pipeline-Implementierung

Technische Grundlagen der Informatik 2
(Rechnertechnologie 2)
SS 2006

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen

Auf Basis von Material von

Rolf Hoffmann

FG Rechnerarchitektur

Technische Universität Darmstadt

In Anlehnung an das Patterson/Hennessy: Computer Organization & Design, 2nd Edition, Chapter 6

Es sind auch die Folien von Dr. M. G. Wahl (Univ. Siegen, Inst. Mikrosystemtechnik) und ähnliche aus den Grundzügen der Informatik II, SS03, von Prof. Dr. Oskar von Stryk verwendet worden.

Inhalt

- Thema: **Pipelining**
- Prinzip
- Bearbeitungszeit
- Anwendungen
  - Arithmetisches Pipelining
  - Prozessor-Pipelining
  - Befehlspipelining
- Befehlspipelining
  1. Ressourcen-Konflikt
  2. Datenkonflikt
  3. Steuerflußkonflikt
- Befehlspipelining bei MIPS
  - Pipeline-Entwurf
  - vom Eintakt-Rechenwerk zum Pipelining
  - Einfügen von Pipeline-Registern
  - Pipeline-Steuerung
  - Datenabhängigkeit
  - Datenhürde
    - Forwarding
    - Stall
  - Steuerflußkonflikt
- Weitere Leistungssteigerung

Grundlegende Überlegungen

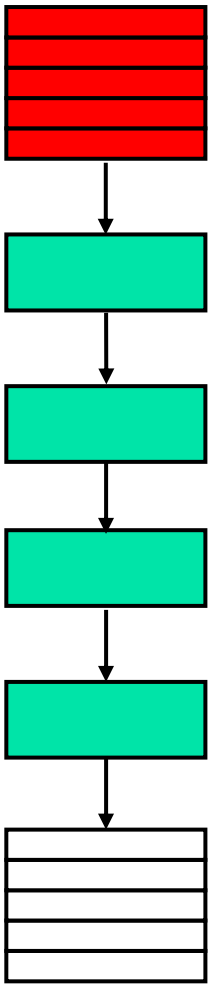
Bisher erreicht:

- Mehrtakt-Implementierung für eine Teilmenge der MIPS-Befehle,
- dadurch zwar mehrere Taktzyklen zur Abarbeitung eines Befehls, aber insgesamt schneller als bei Eintakt-Implementierung.
- Auslastung der Hardware: Mehrtakt-Implementierung benutzt die Hardware-Einheiten (Ressourcen) nur in bestimmten Steuerzuständen.
- Ziel: Effizienzsteigerung durch Nutzung möglichst vieler Ressourcen zu jedem Zeitpunkt
- Lösungsansatz: Parallele Bearbeitung mehrerer Befehle in Form einer Fließbandverarbeitung (Pipelining)

Pipelining Prinzip (1)

- Ziel
  - Es ist eine Menge von ähnlichen Objekten/Ergebnissen in möglichst kurzer Zeit zu produzieren. (Maximierung des Durchsatzes)
- Voraussetzungen
  - Die Produktion kann in aufeinanderfolgenden Teilschritten erfolgen, die in etwa gleich lange dauern.
  - Es gibt ausreichend viele Ressourcen.
- Lösung
  - Für jeden Bearbeitungsschritt wird eine Pipelinestufe vorgesehen.
  - Alle Pipelinestufen arbeiten parallel und liefern ihre Teilergebnisse synchronisiert durch einen Takt an die nächste Stufe weiter.

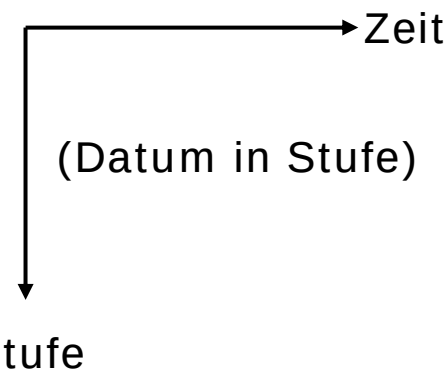
Pipelining Prinzip (2)



Eingabeobjekte (Daten, Befehle, ...), Input-Stream: D1 .. Dn

Takt:	1	2	3	4	5	6	7	8
Stufe1:	D1	D2	D3	D4	D5			
Stufe2:		D1	D2	D3	D4	D5		
Stufe3:			D1	D2	D3	D4	D5	
Stufe4:				D1	D2	D3	D4	D5

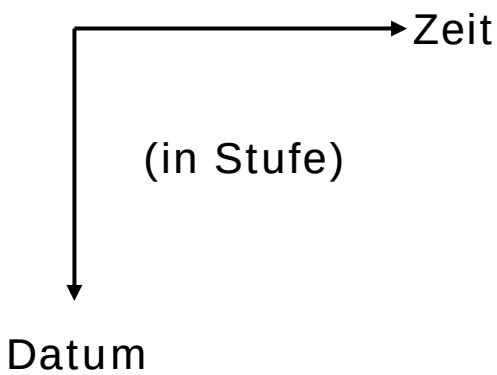
Ausgabeobjekte (Daten, Befehle, ...),
Output-Stream



Darstellung:
Inhalt der Stufen über der Zeit

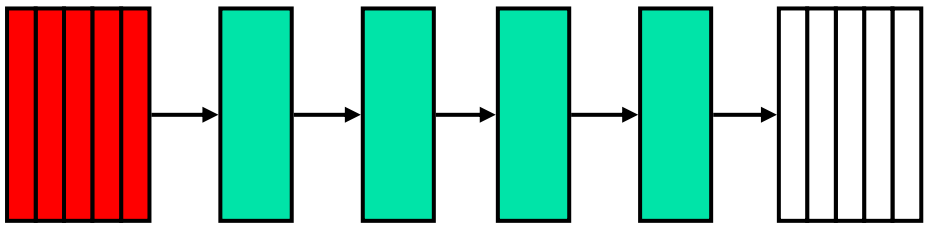
Pipelining Prinzip (3)

Takt	1.	2.	3.	4.	5.	6.	7.	8.
Datum1	Stufe1	Stufe2	Stufe3	Stufe4				
Datum2		Stufe1	Stufe2	Stufe3	Stufe4			
Datum3			Stufe1	Stufe2	Stufe3	Stufe4		
Datum4				Stufe1	Stufe2	Stufe3	Stufe4	
Datum5					Stufe1	Stufe2	Stufe3	Stufe4



Schrittweise Verarbeitung der Daten

Bearbeitungszeit (1)



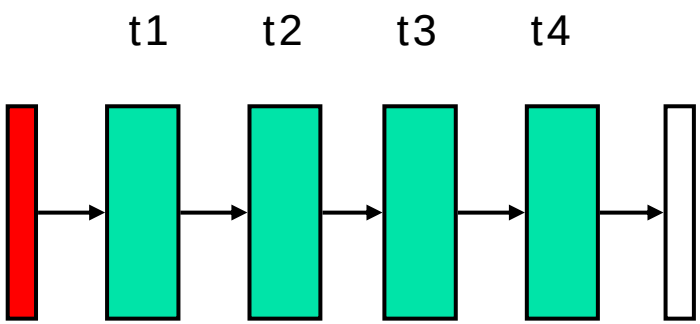
n=4
m=5

- *n* = Anzahl der Stufen
- *m* = Anzahl der zu bearbeitenden Objekte
- *T* = Gesamtbearbeitungszeit in einem Schritt

- **Fall *n=1*, ohne Pipelinestufen**
  - für *m=1* Objekt: $t_{ges}(m=1,n=1) = T$
  - für *m* Objekte: $t_{ges}(m,1) = mT$

Bearbeitungszeit (2)

- **Fall *n>1, n* Pipelinestufen**
  - nur ***m=1* Objekt** wird bearbeitet, die Verweildauer (Durchlaufzeit, Latenz) beträgt dann
    - bei **asynchroner Weitergabe**: $t_{ges}\ (1,n) = Summe(t_i) = T_{asyn}$
    - bei **synchroner Weitergabe**: $t_{ges}\ (1,n) = n\ t_{max} = \boldsymbol{T_{syn}}$
      mit $t_{max} = max\ (t_i) = Taktzeit$



Bearbeitungszeit (3)

- **Fall *n>1, n* Pipelinestufen**
  - ***m* Objekte** werden bearbeitet, bei synchroner Weitergabe

$t_{ges}(m,n) = n\ t_{max} + (m\text{-}1)\ t_{max}$

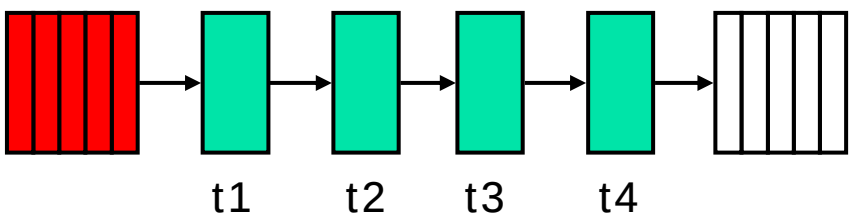
(für das erste Ergebnis, Latenzzeit) + (für die weiteren Ergebnisse)

$= (n\text{-}1)\ t_{max} + m\ t_{max}$

(Füllen start-up) + (für m Ergebnisse)

$$t_{ges}(m,n) = T_{syn} + (m\text{-}1)\ T_{syn}/n$$

  - für sehr große *m:* Start-Up-Zeit ist vernachlässigbar
    - $t_{ges} \approx m\ t_{max}$, d. h. pro Takt ein neues Ergebnis
    - $Z \approx mT_{asyn}/(mT_{syn}/n) = n\ \underbrace{T_{asyn}/T_{syn}}_{\text{kleiner als 1}}$



Speed-Up = Beschleunigungsfaktor = Zeit für sequentielle Bearbeitung / Zeit für Pipeline-Verarbeitung

Formel

$$t_{ges}(m,n) = T_{syn} + (m-1)\,T_{syn}/n$$
$$= (n-1)*T_{syn}/n + m*T_{syn}/n$$

- t_{ges} — Zeit zur Bearbeitung von m Objekten
- n — Anzahl der Stufen
- m — Anzahl der Objekte
- $T=T_{syn}$ — Verweildauer, Durchlaufzeit
- $t_{max}=T_{syn}/n$ — Taktzeit
- $(n-1)*T_{syn}/n$ — Latenzzeit, Zeit um die Pipeline zu füllen

Pipelining: Eigenschaften

- Die Gesamtbearbeitungszeit für eine einzelne Aufgabe ist beim Pipelining höher im Vergleich zur asynchronen Verarbeitung in einem Schritt
- Die Pipelinegeschwindigkeit wird durch die langsamste Stufe bestimmt.
- Durchsatz steigt mit der Anzahl *n* der Pipelinestufen, wenn
  - die Pipeline immer weiter in etwa gleich lange Stufen geteilt werden kann.
  - die Zeit zum Zwischenspeichern der Teilergebnisse "vernachlässigt" werden kann (Pufferregister verursachen Zeitverzögerung).
- Durchsatz wird reduziert durch
  - ungleiche Länge der Stufen
  - Zeit zum Füllen (am Anfang) und Leeren der Pipeline (am Ende)
  - Eventuelle Abhängigkeiten zwischen den Stufen

Durchsatz und Latenz

- Beispiel: Datentransfer zwischen DA und BS
- Kombi, 565 l Kofferraumvolumen
  - Bepackt mit 2260 LTO-3 Magnetbändern (je 400 GB)
  - Angenommene Fahrzeit: 4h
  - **Durchsatz**: 904 TB / 4h → 62,8 GB/s
  - ... ziemlich fix
- Mit gleichem Ansatz
  - Web-Surfen?
  - WoW spielen?
  - Via VoIP telefonieren?
  - Nein, da **Latenz** zwischen zwei Datentransfers 4h ist
    - Fällt jeweils für Senden und Empfangen an!

Verschiedene Anwendungen des Pipelining

- Bearbeitung von Datenströmen
  - Digitale Signalverarbeitung
- Arithmetisches Pipelining: Arithmetische Operationen werden in Teilschritten ausgeführt (z. B. Gleitkommaoperationen)
- Prozessor-Pipelining: in jeder Bearbeitungsstufe sitzt ein Prozessor.
- Befehlspipelining, im folgenden beim MIPS
  - Befehl holen, decodieren und ausführen wird ständig wiederholt

Befehlspipelining

- die auszuführenden Schritte bei RISC-Befehlen, beispielhaft
  - Register-Register-Befehl
    - Holen, Decodieren, Register-Lesen, ALU-Operation, Register-Schreiben.
  - Load-Befehl
    - Holen, Decodieren, Adresse berechnen, aus dem Cache oder Hauptspeicher lesen, Register-Schreiben.
  - Sprungbefehl
    - Holen, Decodieren, Sprungadresse berechnen, Befehlszähler neu setzen, *schon vorverarbeitete Nachfolge-Befehle beenden oder ihre Wirkung verhindern*.

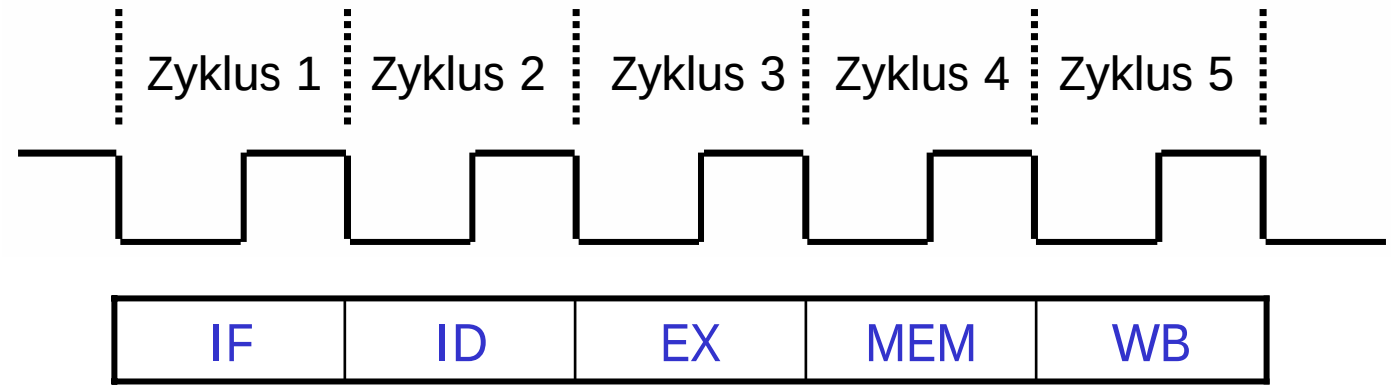
Befehlspipelining bei MIPS

- Die Ausführung eines komplexen Programms bedeutet in der Regel das Ausführen von Milliarden von Mikrooperationen.

→ Durchsatzmaximierung ist eines der wesentlichen Entwurfsprobleme.

- Was erleichtert die Implementierung (z. B. bei MIPS)?
  - Alle Befehle sollten in etwa gleich lang sein.
  - Registerreferenzen sollten in allen Befehle an derselben Stelle platziert sein.
  - Speicherzugriffe sollten nur in Lade- und Speicherbefehlen vorkommen.
  - Je Befehl wird höchstens ein Rechenergebnis erzeugt, das in ein Register geschrieben wird.

MIPS-Befehlszyklen am Beispiel `lw`



IF: Lade Befehl aus dem Programmspeicher

ID: Lade Register und decodiere Befehl

EX: Berechne die Speicheradresse

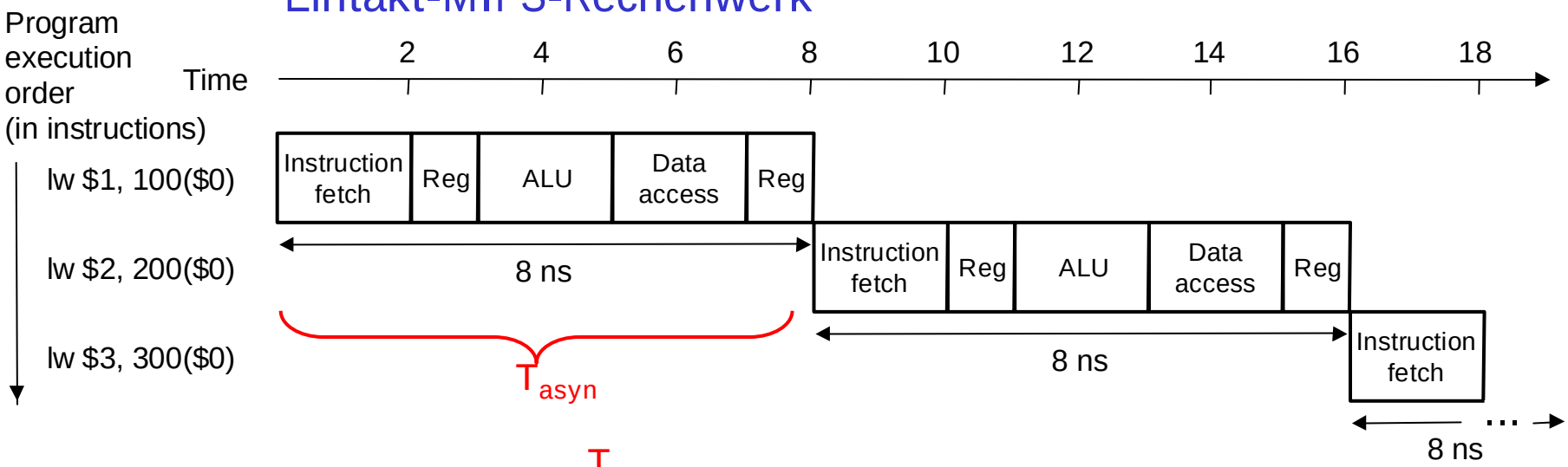
MEM: Lies/schreibe die Daten aus/in Datenspeicher

WB: Schreibe die Daten ins Zielregister

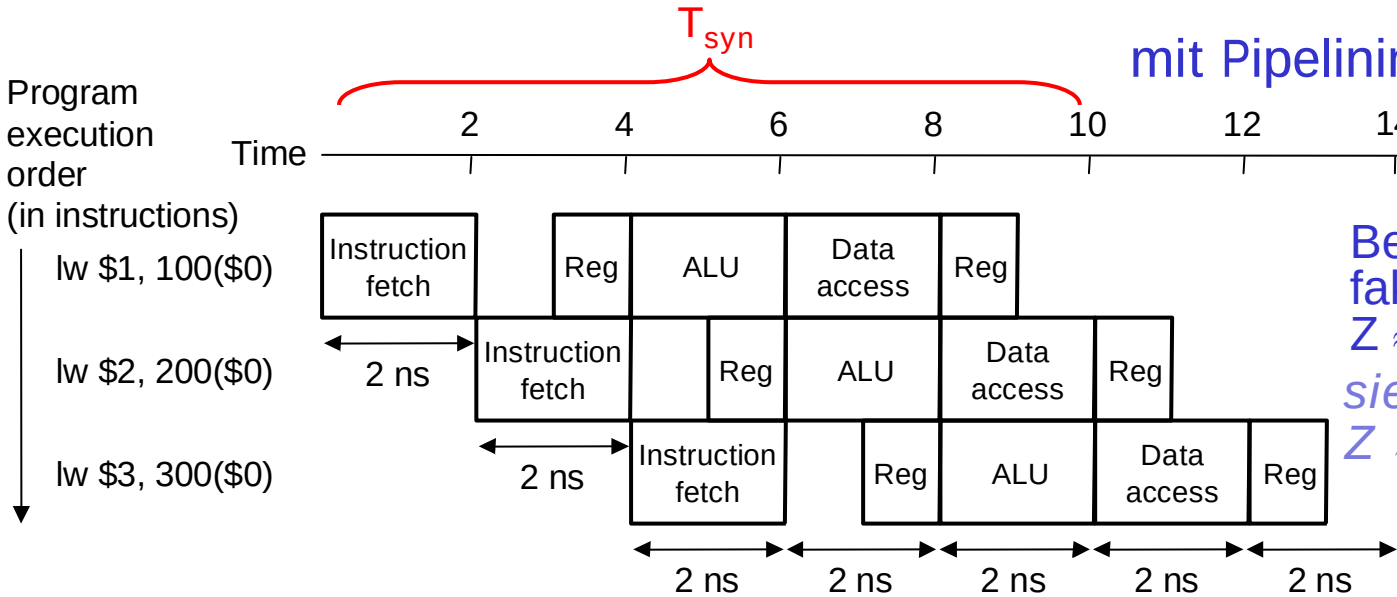
Die Überlegungen in Kapitel 5 sind auf die acht MIPS-Befehle **`lw, sw, add, sub, and, or, slt, beq`** beschränkt, die jeweils in maximal 5 Taktzyklen ausgeführt werden. Die übrigen MIPS-Befehle werden hier nicht untersucht.

Erhöhung des Befehlsdurchsatzes

Eintakt-MIPS-Rechenwerk



mit Pipelining



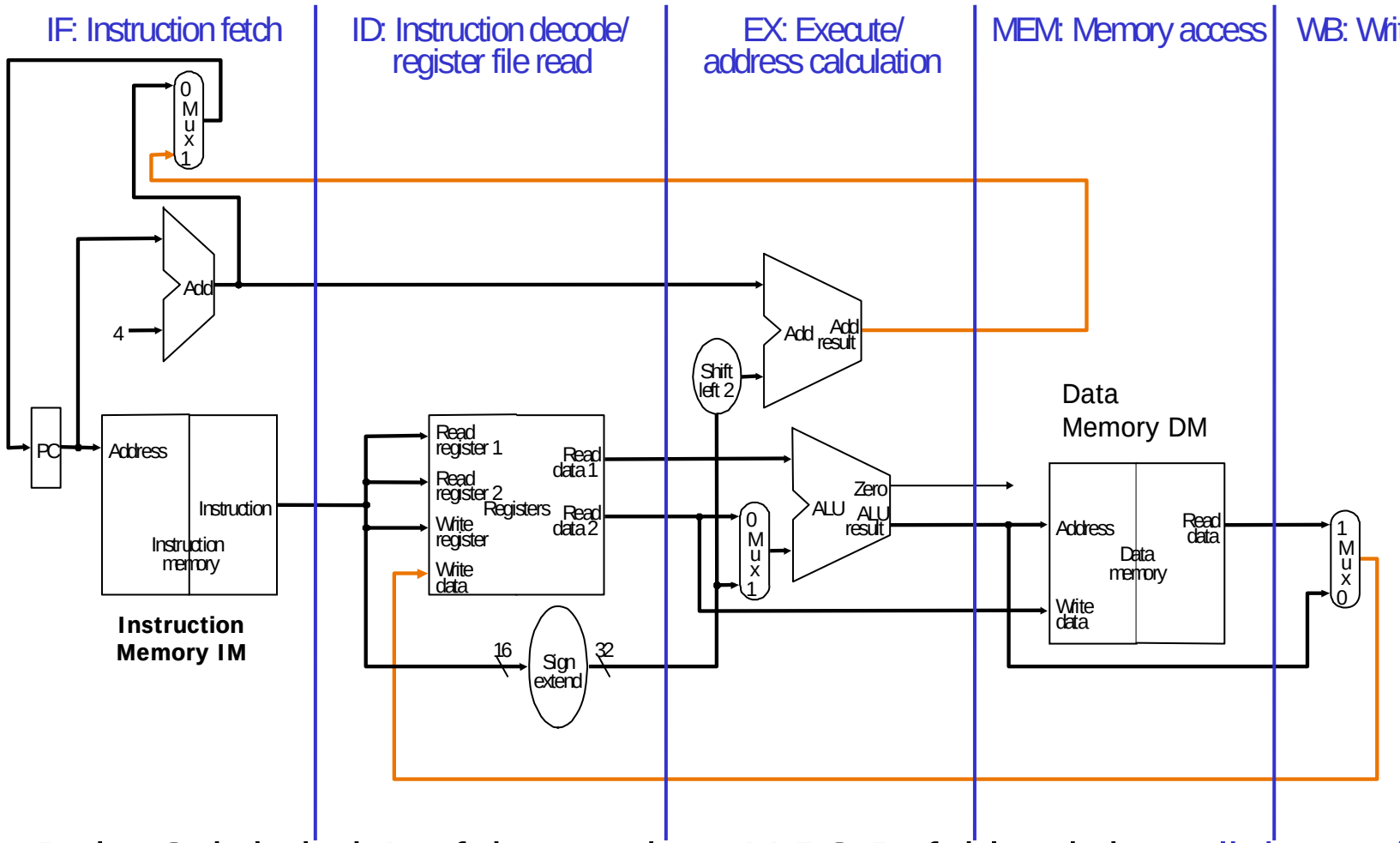
Beschleunigungs-faktor im Beispiel:
$Z \approx 5 * 8/10 = 4$

siehe Folie 10-9:
$Z \approx n\ T_{asyn}/T_{syn}$

Pipeline-Entwurf

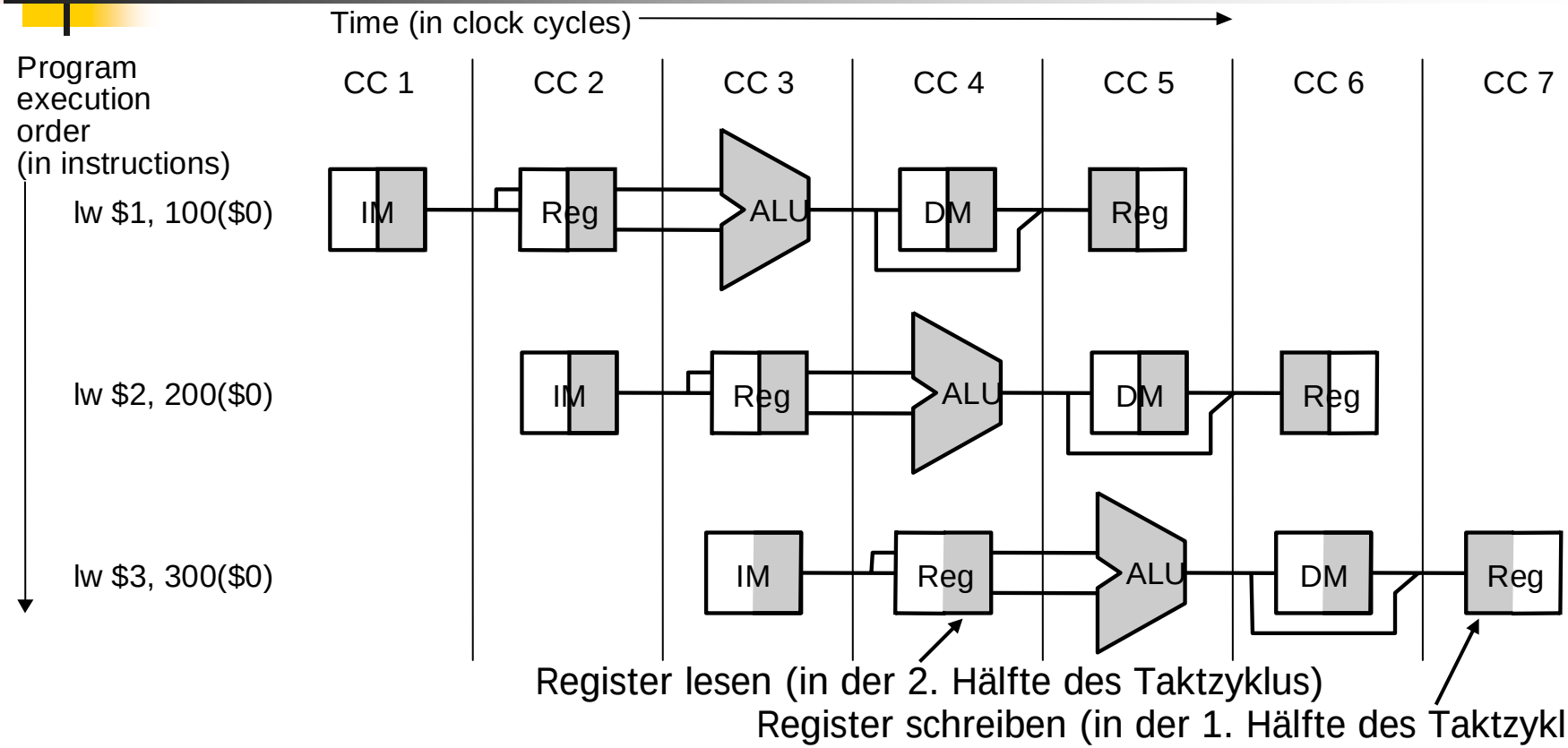
- Vorgehen:
  - Erst wird angenommen, daß jeder Befehl ein eigenes spezielles Operationswerk zur Verfügung hätte. Die Operationswerke werden in Stufen geteilt.
  - Dann wird die zeitliche Abarbeitung aller einzelnen Operationswerke in Zusammenhang gesetzt und daraus durch Überlagerung ein gemeinsames Operationswerk entworfen.

Eintakt-Rechenwerk und Befehlszyklen



- Jeder Schritt bei Ausführung eines MIPS-Befehls wird von links nach rechts fortgeführt.
- Zwei Ausnahmen: WB-Schritt und Wahl des nächsten PC

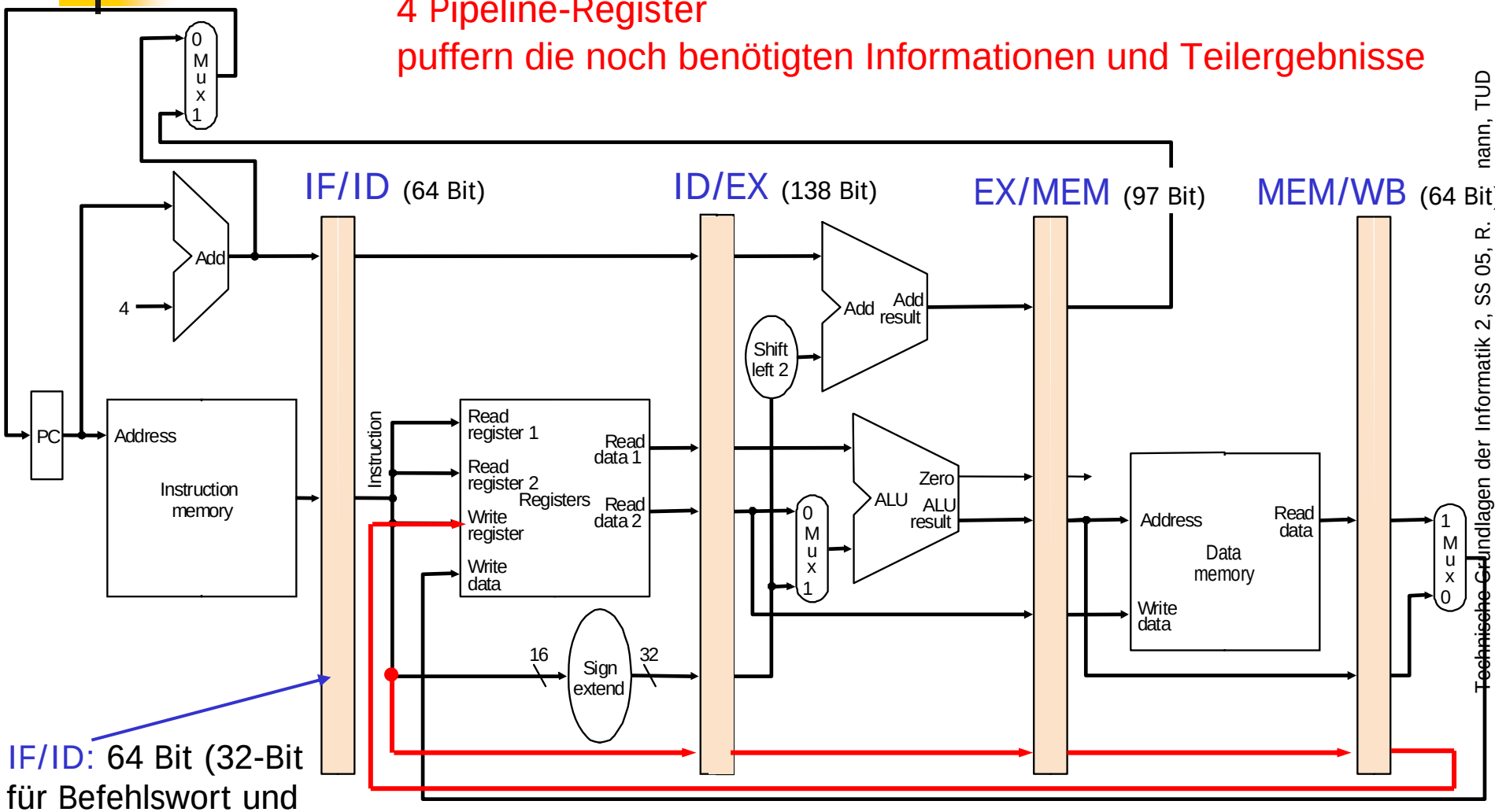
vom Eintakt-Rechenwerk zum Pipelining



- Befehlsspeicher IM wird nur in jeweils einer der 5 Stufen eines Befehls verwendet, d. h. kann während der übrigen 4 Stufen von anderen Befehlen verwendet werden.
- Um den Datenwert eines einzelnen Befehls für die übrigen 4 Stufen zu erhalten, muß der aus dem IM gelesen Wert in einem Register zwischengespeichert werden. → Pipeline-Register zwischen Stufen.

Einfügen von Pipeline-Registern

4 Pipeline-Register
puffern die noch benötigten Informationen und Teilergebnisse

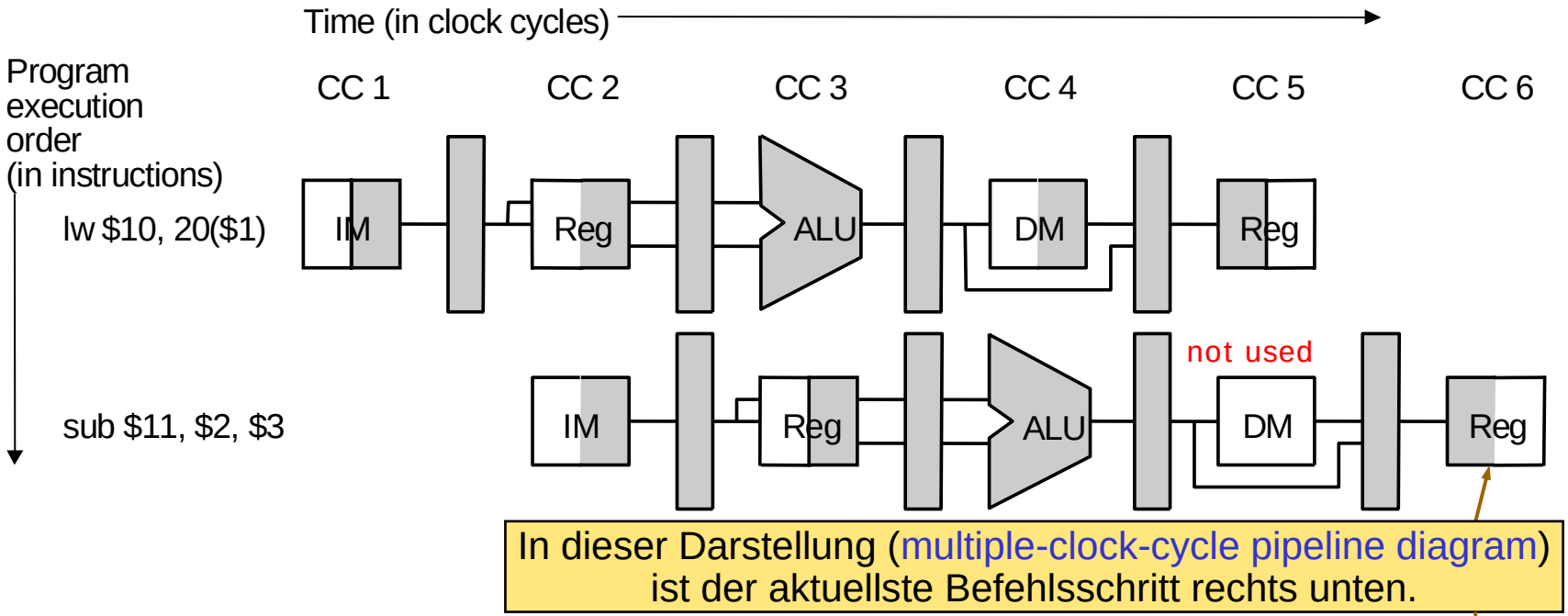


IF/ID: 64 Bit (32-Bit für Befehlswort und 32-Bit für erhöhte PC-Adresse)

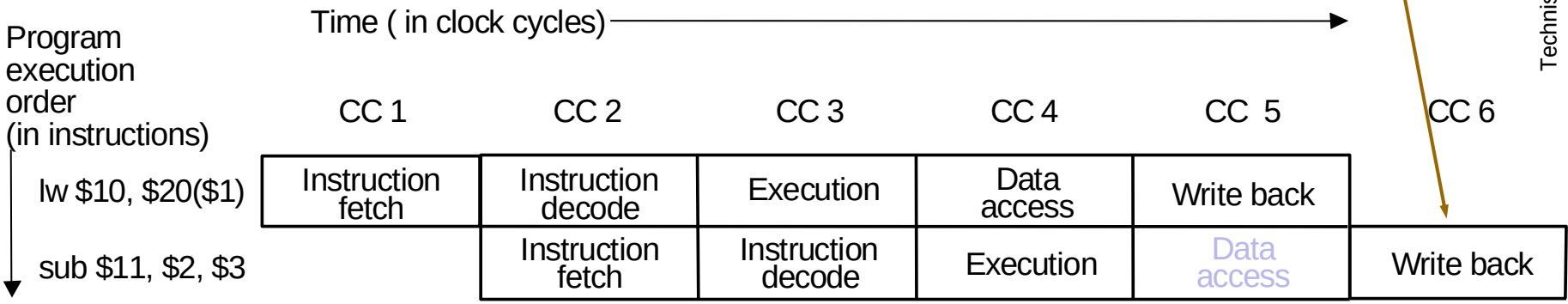
Modifiziertes Operationswerk: Nummer des Write-Registers und zu schreibender Datenwert kommen aus Pipeline-Register MEM/WB.

Darstellung der Aktivitäten

- Darstellung der benutzten physikalischen Ressourcen



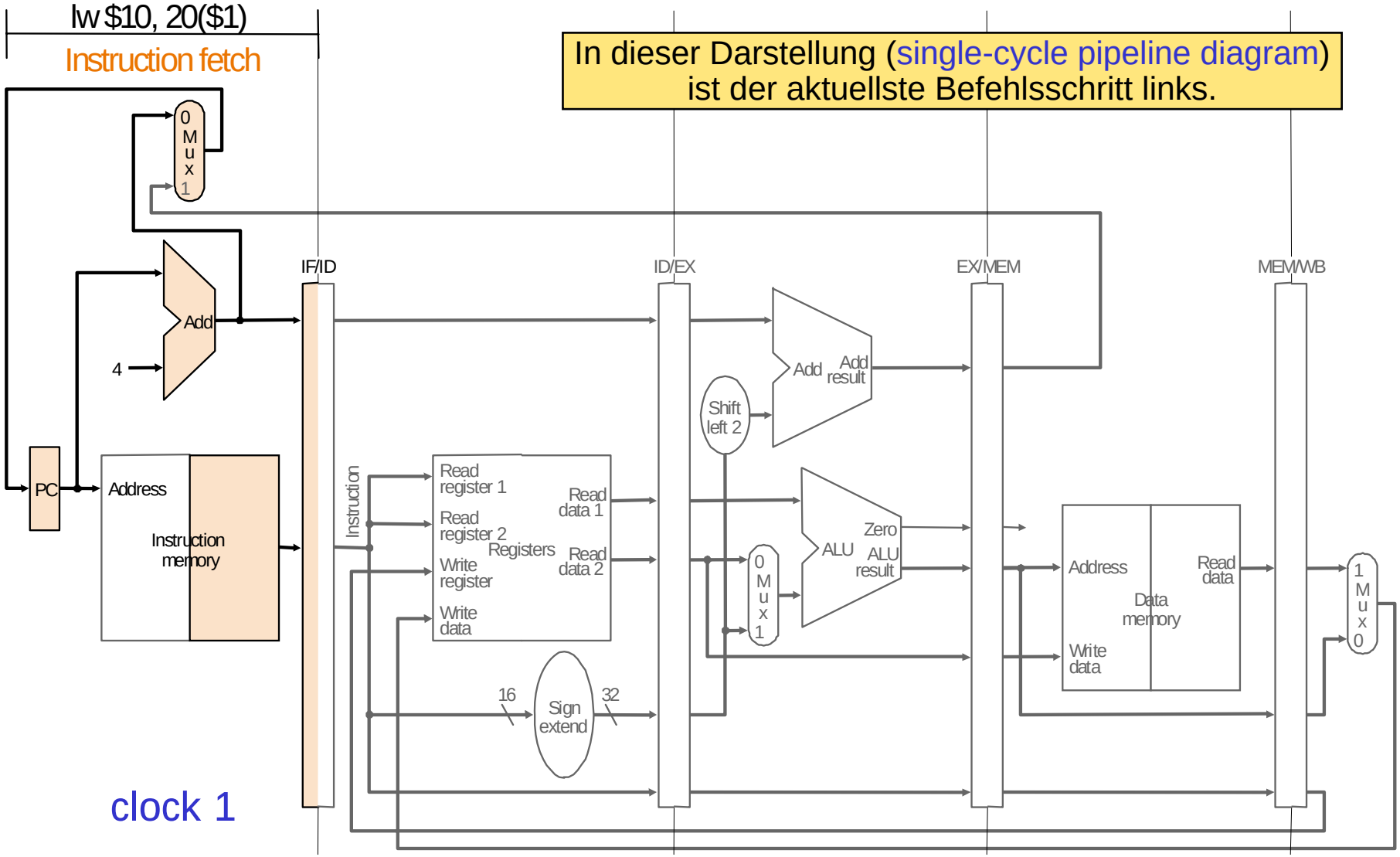
- konventionelle Darstellung mit Namen der einzelnen Schritte:



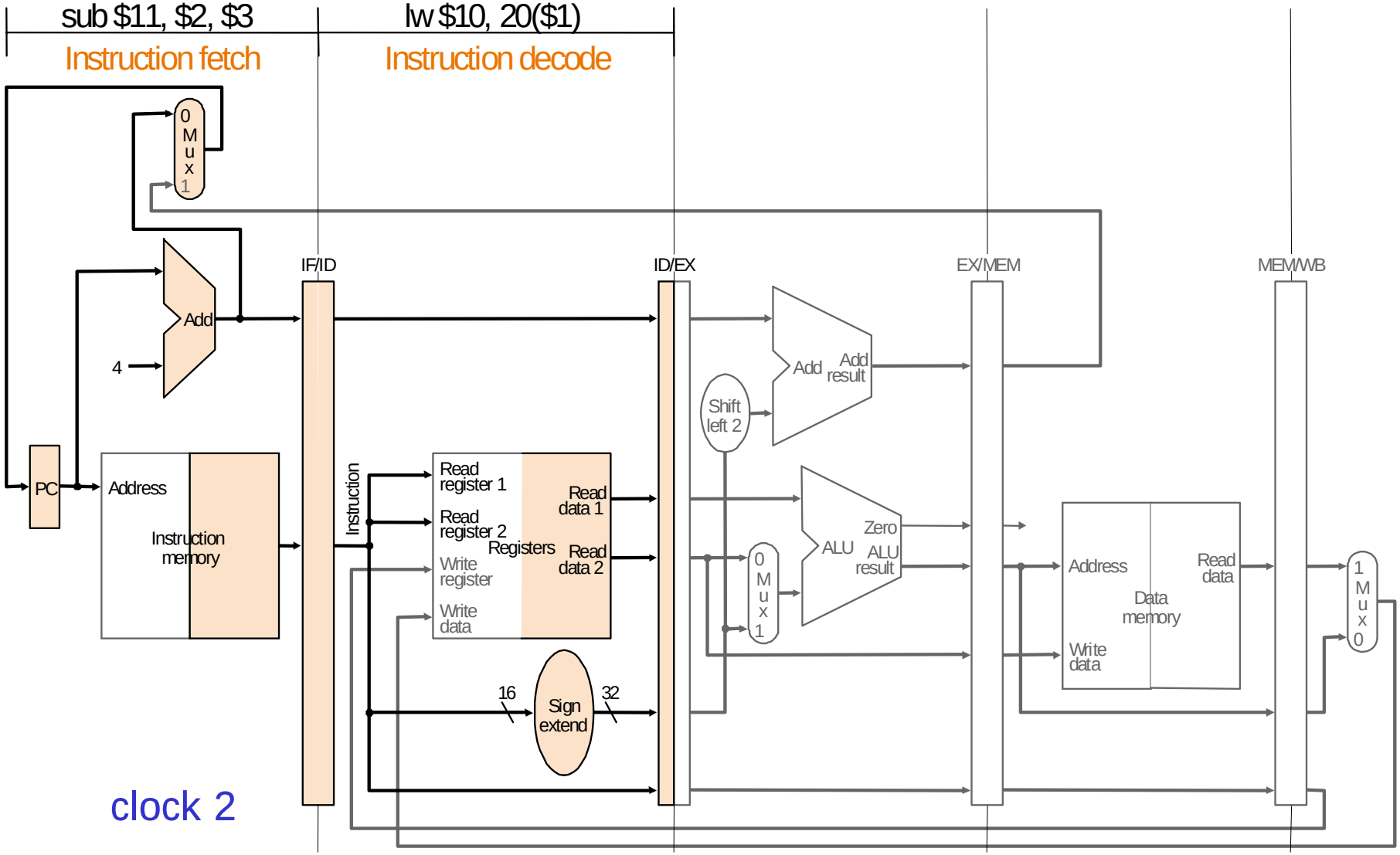
Betrieb der Pipeline

- Optimale Nutzung, wenn 5 Befehle gleichzeitig aktiv in den 5 Stufen
- Anfangsphase: Füllen der Pipeline
- Betriebsphase: Gleichzeitige Bearbeitung in der Pipeline
- Endphase: Leeren der Pipeline
- Beispiel:
  - `lw`
  - `sub`

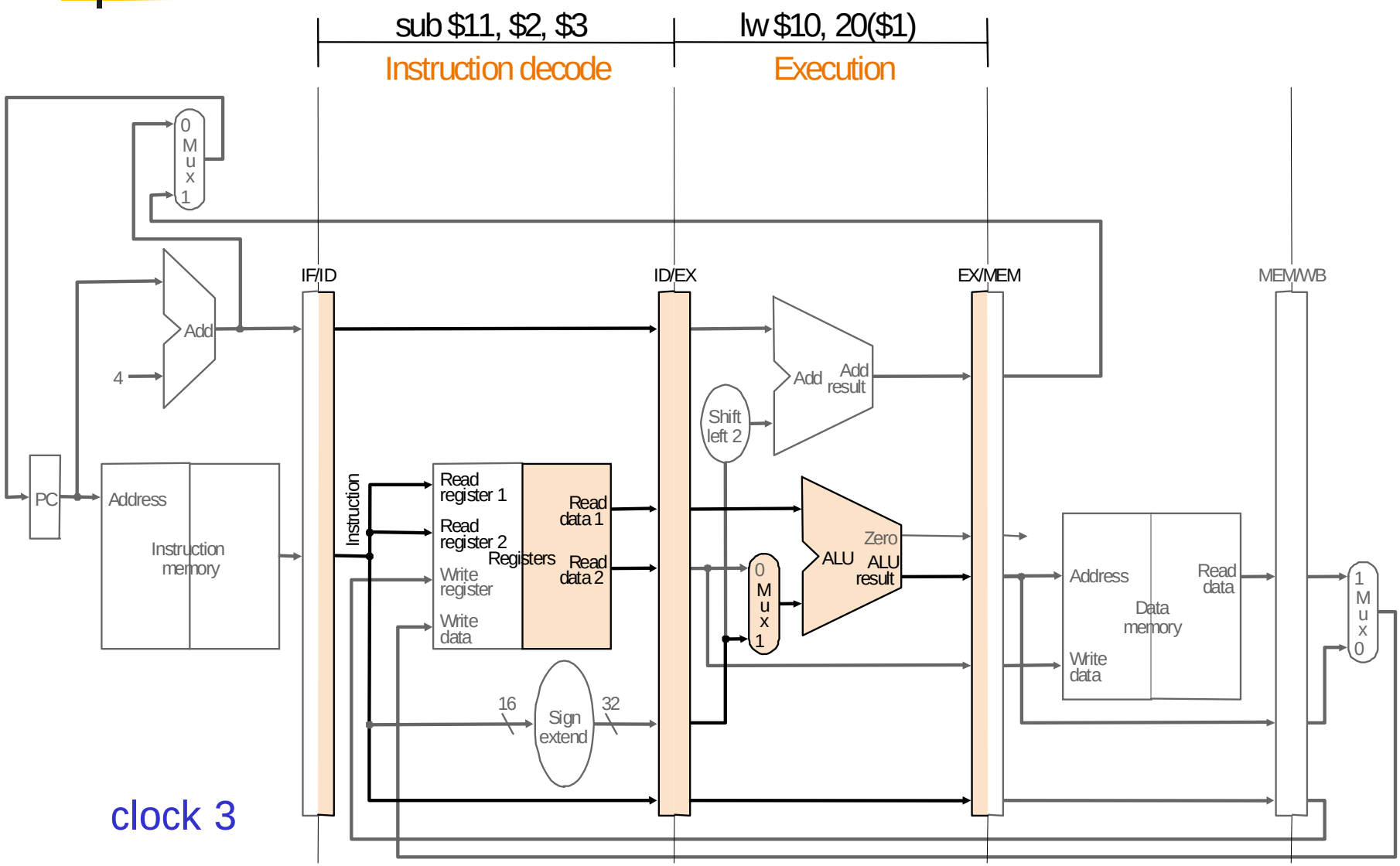
Beispiel zur Pipeline (1)



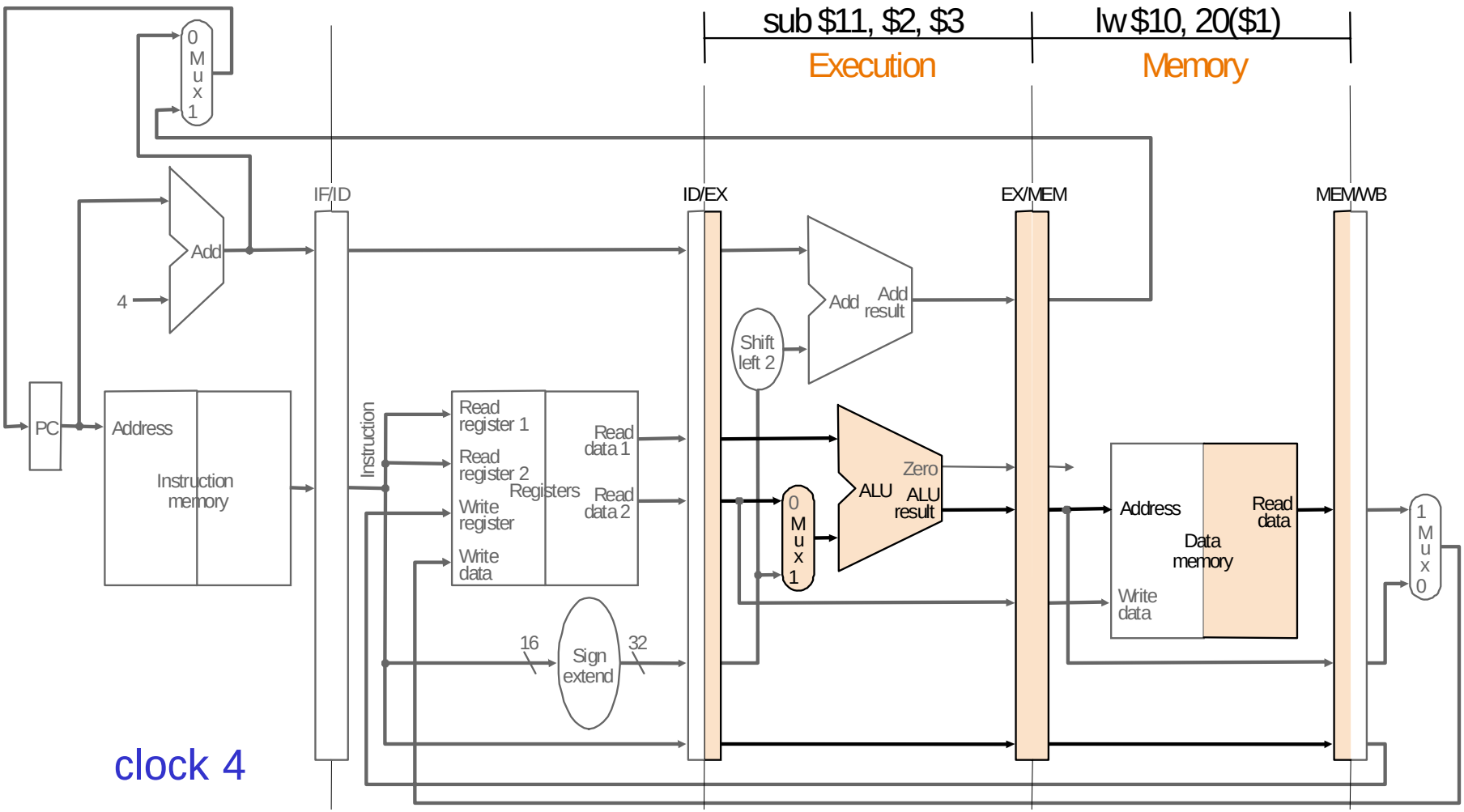
Beispiel zur Pipeline (2)



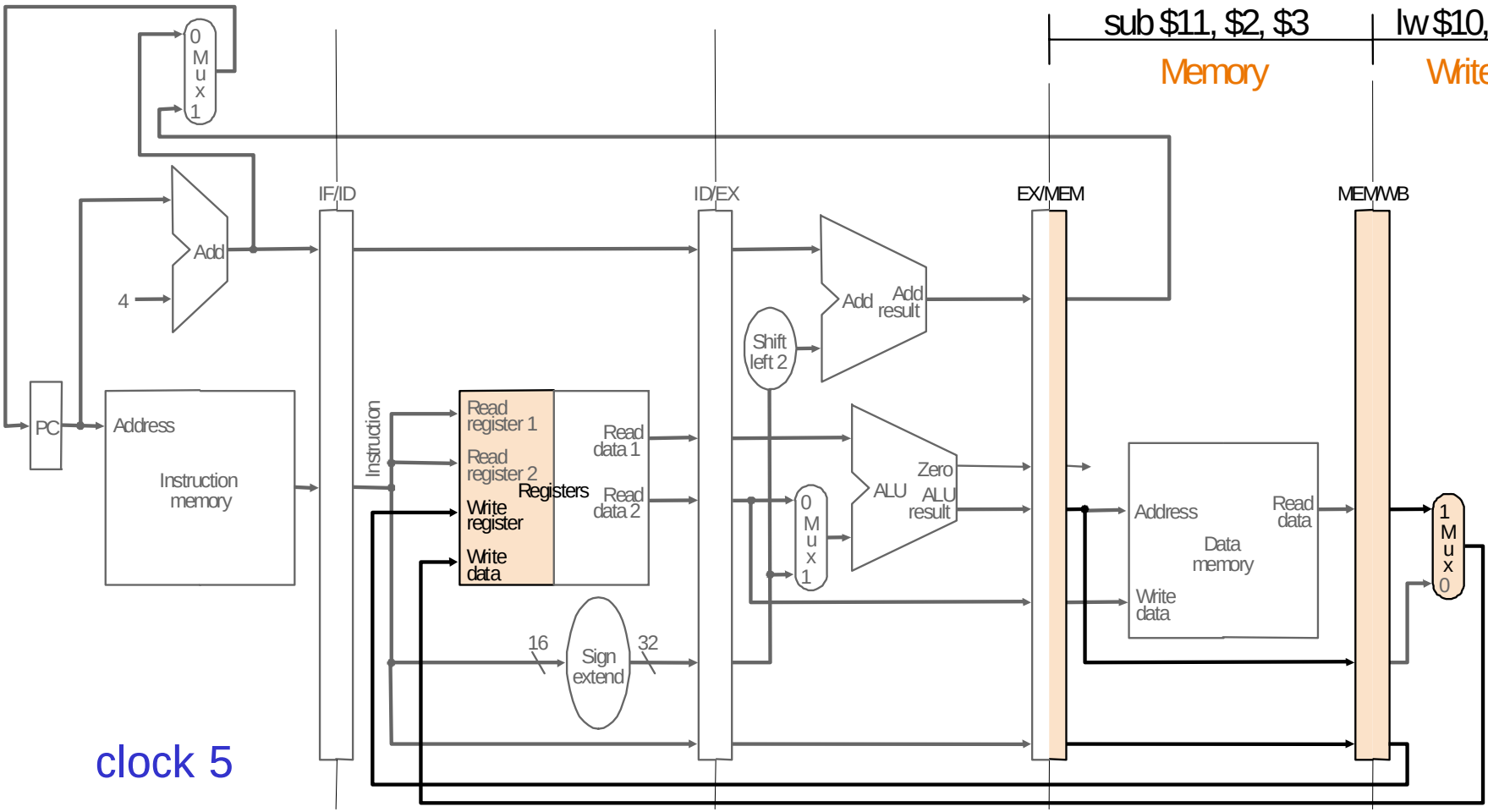
Beispiel zur Pipeline (3)



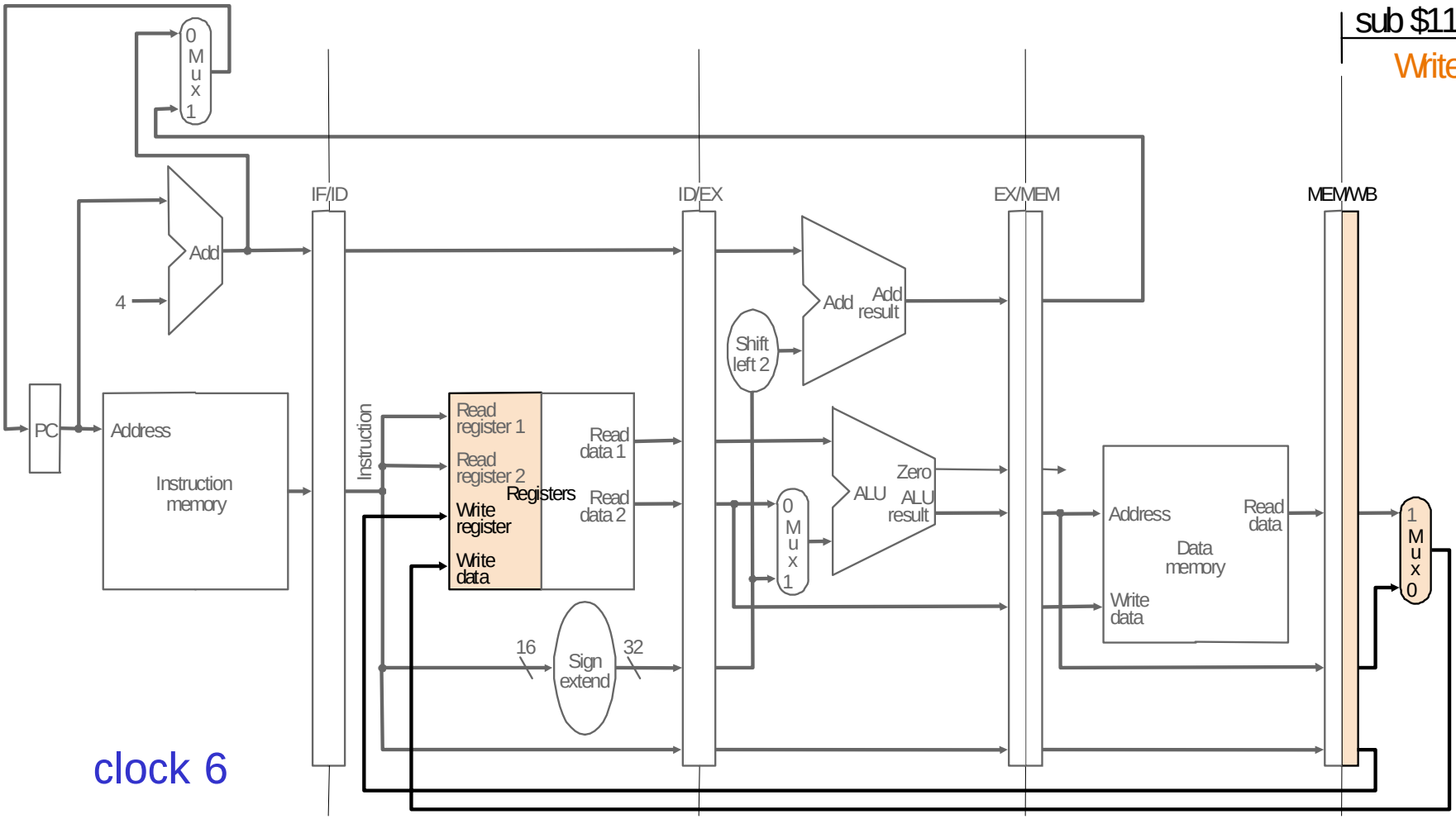
Beispiel zur Pipeline (4)



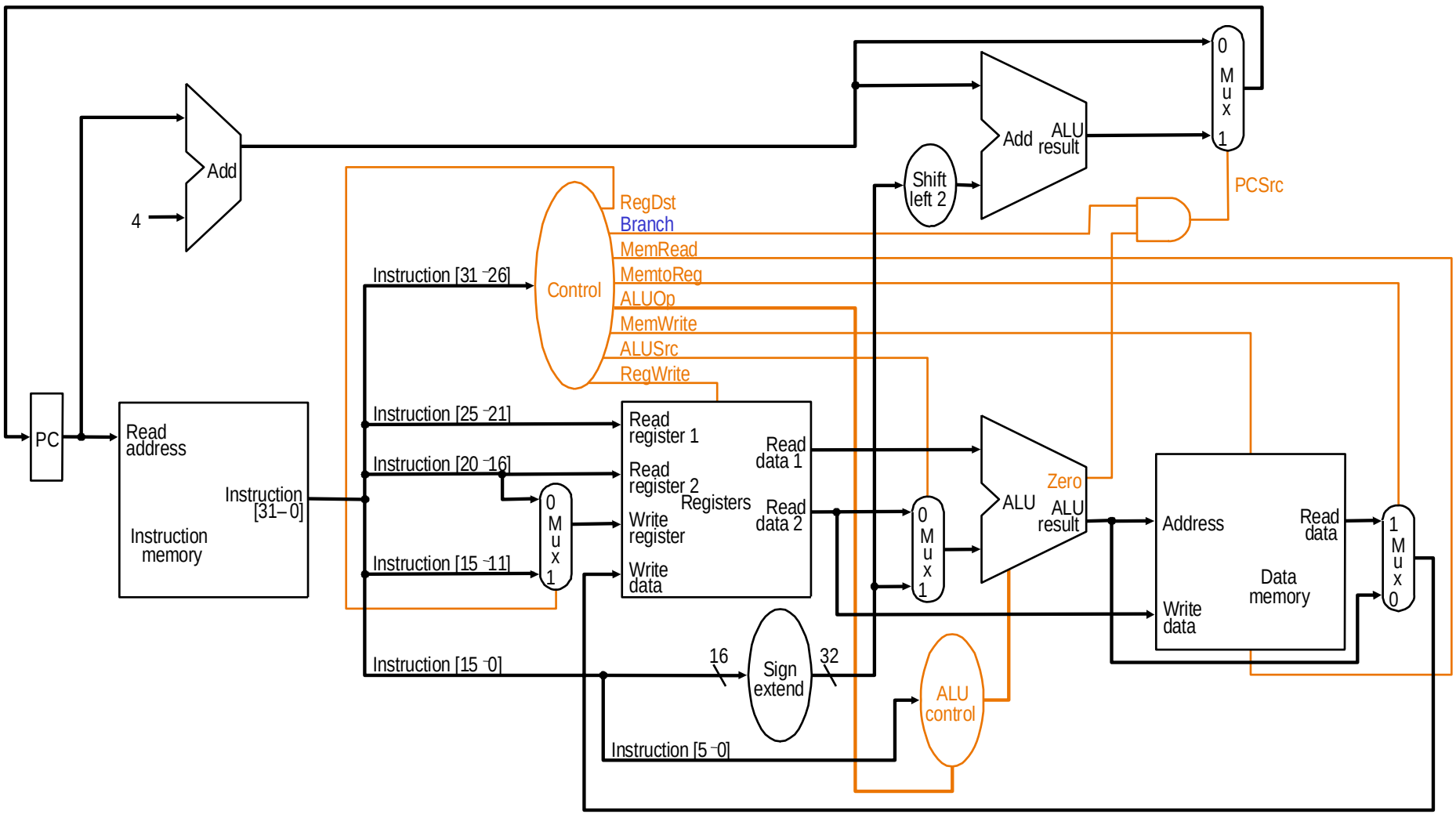
Beispiel zur Pipeline (5)



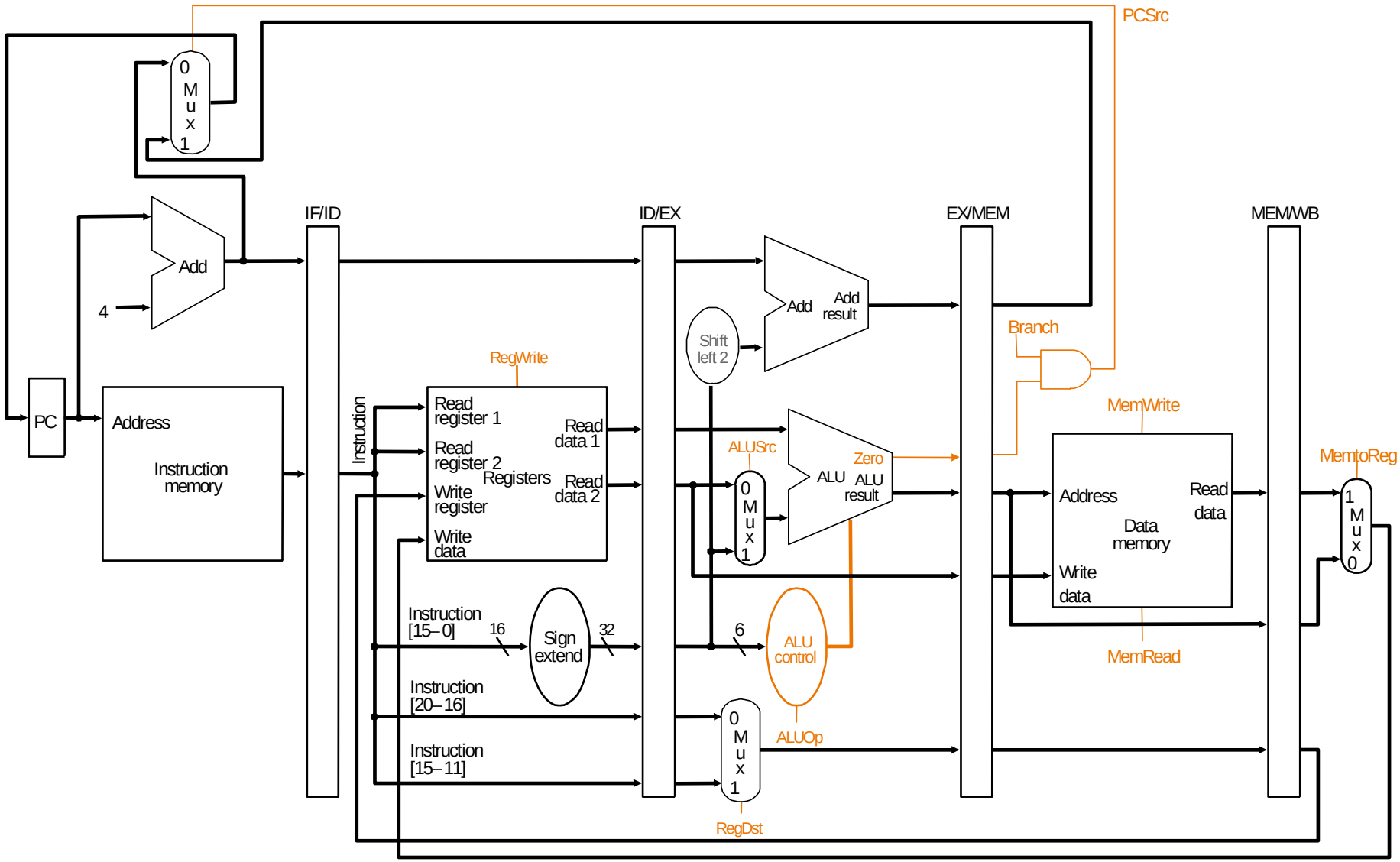
Beispiel zur Pipeline (6)



Eintakt-Implementierung (Wdh.)



Pipeline-Steuersignale



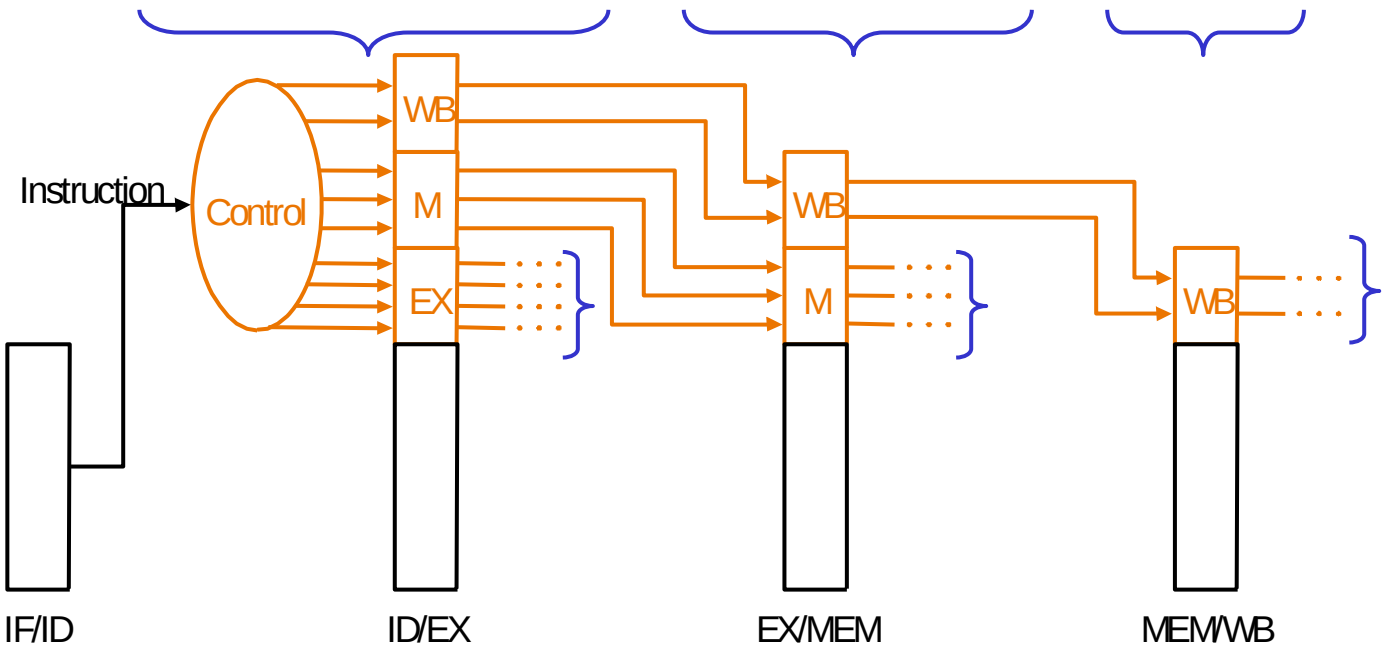
Pipeline-Steuerung

- Gleiche Bedeutung der Steuersignale wie beim Eintakt-Rechenwerk
- Gruppierung der Steuersignale nach den 5 Stufen. Was muß in jeder Stufe gesteuert werden?
  - IF: Befehl laden und PC erhöhen
  - ID: Decodiere Befehl / Lade Register
  - EX: Ausführung
  - MEM: Speicherstufe
  - WB: Zurückschreiben von Daten ins Zielregister
- Durchreichen der Steuersignale bzw. des Opcodes

Pipeline-Steuerung

- Gruppierung der Steuersignale nach den letzten 3 Pipelinestufen:

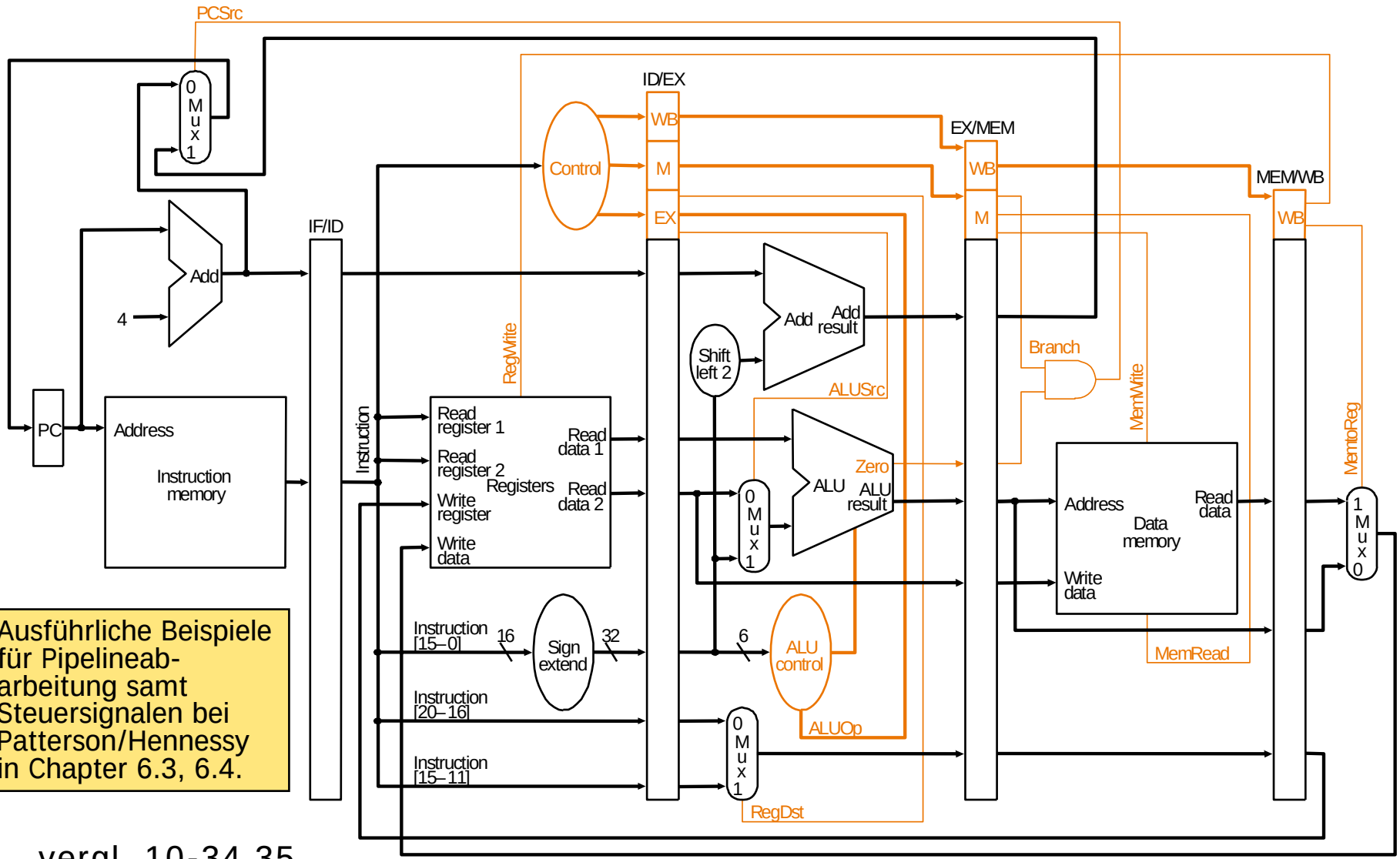
Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X



- Durchreichen von Steuersignalen über erweiterte Pipeline-Register wie Daten:

Rechenwerk mit Steuerwerk

- Steuersignale für die letzten 3 Pipelinestufen werden in ID-Stufe erzeugt und dann durchgereicht.



Ausführliche Beispiele für Pipelineab-arbeitung samt Steuersignalen bei Patterson/Hennessy in Chapter 6.3, 6.4.

vergl. 10-34,35

Probleme (1)

- In der Pipeline befinden sich immer bis zu n Befehle gleichzeitig in Bearbeitung.
- Die Befehle konkurrieren um die Ressourcen
- Zwischen den Befehlen gibt es unterschiedliche Abhängigkeiten, die bei der Pipelineverarbeitung zu Situationen (Hazards, Hemmnisse, Hürden, Konflikte) führen, die in der Hardware oder Software speziell behandelt werden müssen.
- Unterschiedliche Komplexität der Befehle erschwert den Pipeline-Entwurf
  - Anzahl der nacheinander auszuführenden Mikrooperationen ist nicht konstant (variiert je nach Befehl und kann von Daten in der Ausführung abhängig sein)

Probleme (2)

- Die Befehlspipelinestufen können nicht immer genutzt werden, wegen der möglichen Konflikte

1. Ressourcen-Konflikt (Strukturkonflikt, structural Hazard) →
2. Datenkonflikt (Data Hazard, Datenhürde) →
3. Steuerflußkonflikt (Control Hazard, Sprungabhängigkeit, Procedural Dependency) →

1. Ressourcen-Konflikt

- entsteht durch die begrenzten Ressourcen
- Die Anzahl, Art und Reihenfolge der Zugriffe auf die Hardware-Einheiten (wie Speicher, Register, ALUs, Adreßrechenwerk, MMU, Cache) variiert je nach Befehl. Bei der parallelen Ausführung in der Pipeline kommt es zu Zugriffskonflikten, wenn nicht genügend viele Hardware-Ausführungseinheiten bzw. Zugriffsmöglichkeiten zur Verfügung stehen.
- Auflösung: **Wartezyklen** (Bubbles)

2. Datenhürde, Datenkonflikt

- entsteht durch Datenabhängigkeiten
- (echte Datenabhängigkeit) Der Befehl i+k, k>1 benötigt das Ergebnis des Befehls i, der sich noch in der Berechnung befindet (true data dependency)
  - Auflösung: Wartezyklen, Forwarding
- (falsche Datenabhängigkeit, name dependency)
  - entsteht nur durch Umordnung der Ausführungsreihenfolge (out-of-order execution) bei modernen Prozessoren, dabei können sich Befehle überholen. **Ist bei dem MIPS nicht möglich**. Die Konflikte sind
  - Zielregisterkonflikt (zwei Befehle wollen gleichzeitig ein Zielregister schreiben)
  - Gegenabhängigkeit (ein überholender Befehl will schon ein Datum ändern, daß noch zur Berechnung benötigt wird)
  - Auflösung: Benutzung von verdeckten **Renaming-Registern**

3. Steuerflußkonflikt

- entsteht durch eine verzögerte Ausführung des Sprungs
  - weil das Sprungziel erst noch berechnet werden muß (z. B. relativer Sprung, Branch).
  - weil die Berechnung der Sprungbedingung noch nicht beendet ist.
- Die Entscheidung zu Auswahl des nächsten Befehls wird meist getroffen, bevor das Sprungziel feststeht, um Wartezyklen zu vermeiden. Es ist deshalb nicht sicher, daß der in die Pipeline gefütterte Befehl wirklich als nächster ausgeführt werden sollte.
- Auflösung
  - Die Befehle, die hinter dem Sprung stehen (**Delayed-Slot-Befehle**) werden schon in die Pipeline geholt
    - entweder werden sie bis zum Ende ausgeführt
    - oder sie werden annulliert (Wirkung: Warten)
  - Es werden die Befehle in die Pipe geholt, die entweder **hinter dem Sprung** stehen oder **am Sprungziel**. Auf Grund der Vergangenheit wird der wahrscheinlichere Weg eingeschlagen (**spekulative Ausführung**).

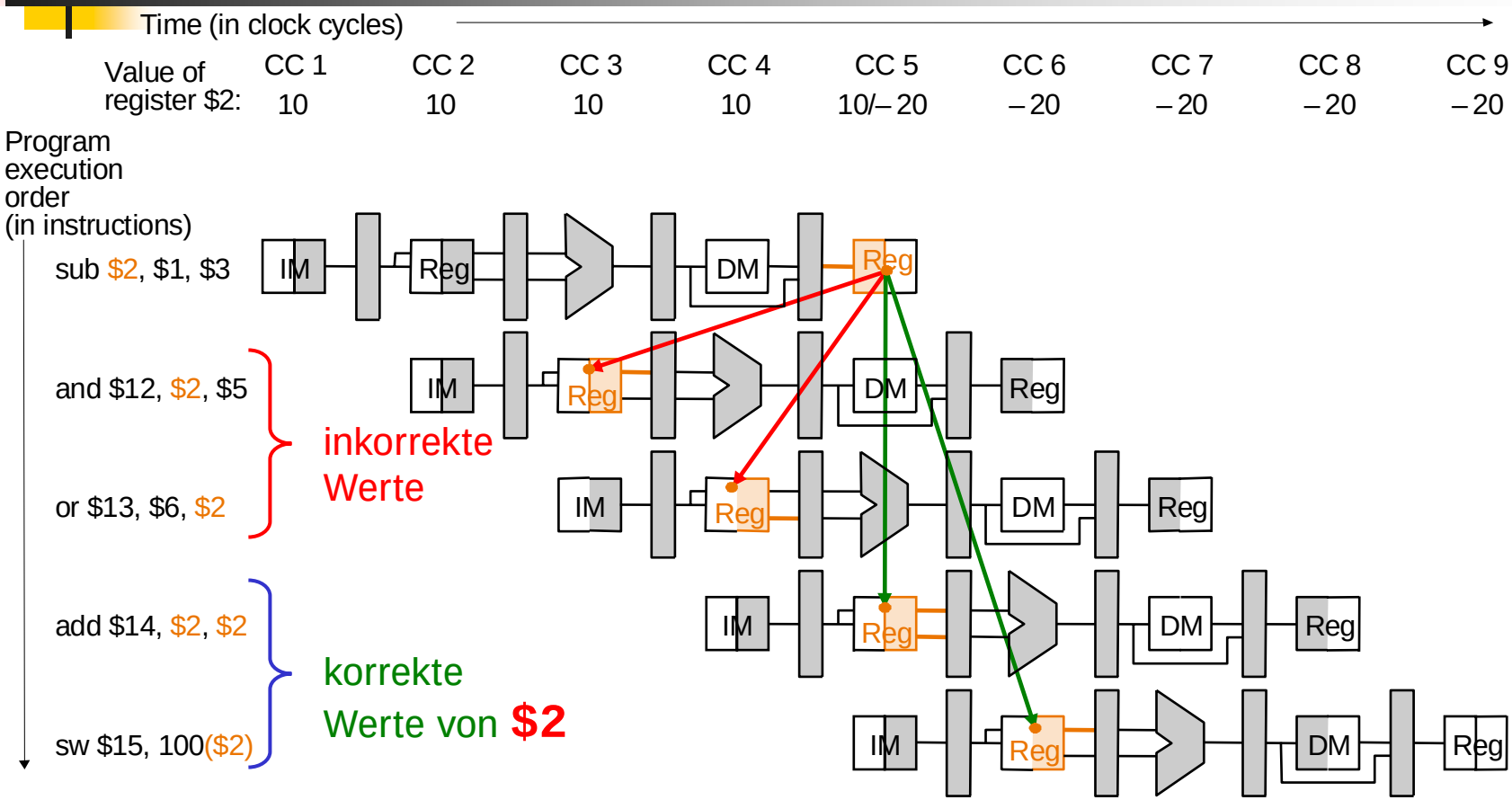
Datenabhängigkeit

- Beispiel:

```
sub  $2,   $1, $3
and  $12,  $2, $5  //soll den neuen Wert benutzen
or   $13,  $6, $2
add  $14,  $2, $2
sw   $15, 100($2)
```

- **sub** berechnet $2, wird in den Folgebefehlen verwendet.
- Falls **$2** den Wert **10** vor **sub** und **-20** danach hat, soll ab **and** der Wert **-20** verwendet werden.

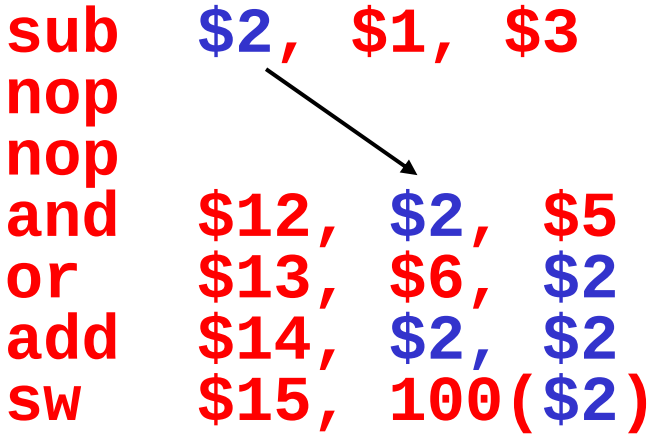
Datenabhängigkeit



- Abhängige Schritte sind **durch Pfeile** gekennzeichnet.
- Der **korrekte Wert von $2** liegt erst nach der 1. Phase (Register Schreiben) von Zyklus 5 vor.
- Abhängigkeiten, die in der Zeit „rückwärts“ zeigen, sind **Datenhürden (data hazards).**

Datenhürde: Software-Lösung

- Compiler vermeidet bei Transformation in Assembler solche Hürden.
- z. B. durch Einfügen zweier unabhängiger Operationen:



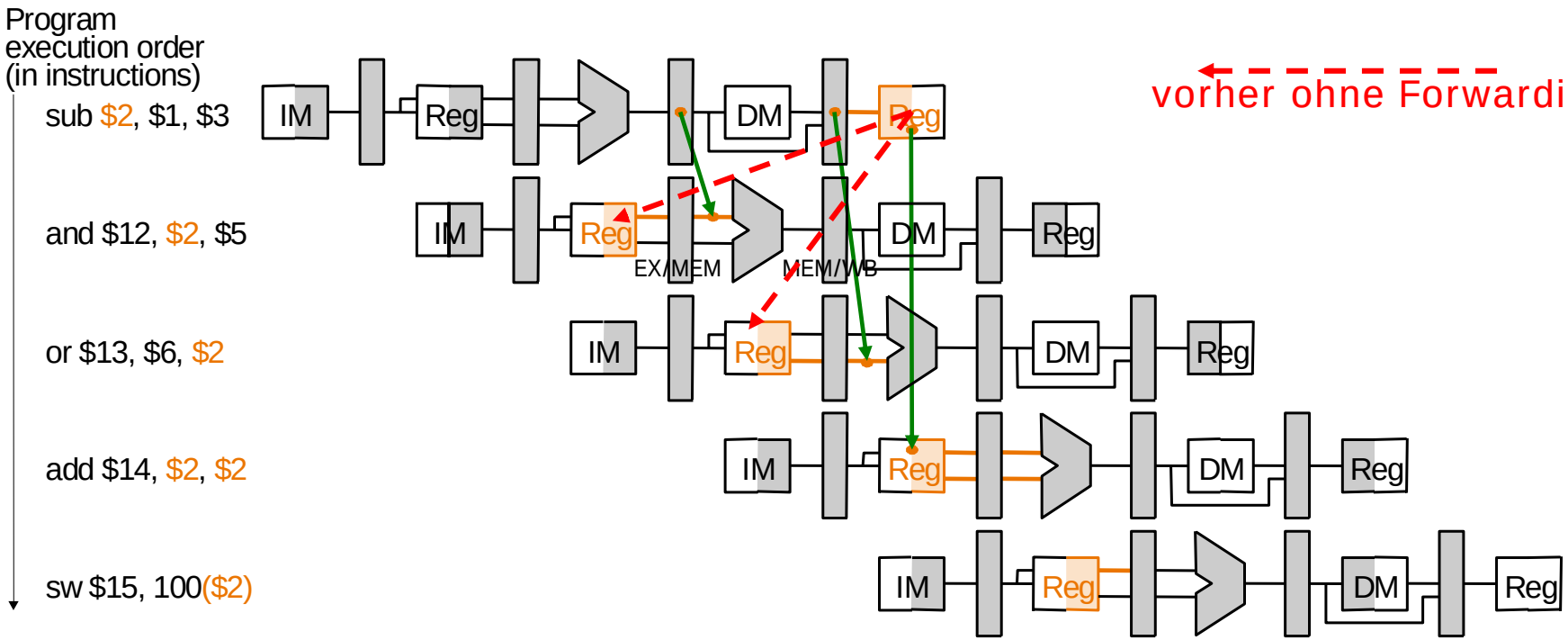
```
sub  $2, $1, $3
nop
nop
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

- **nop** (no operation) modifiziert weder Daten noch schreibt es ein Ergebnis.
- Problem: Bremst das Ganze beträchtlich! (Und tritt zu häufig auf.)
- Besser: Hardware-Lösung

Datenhürde: Vorreichen (Forwarding)

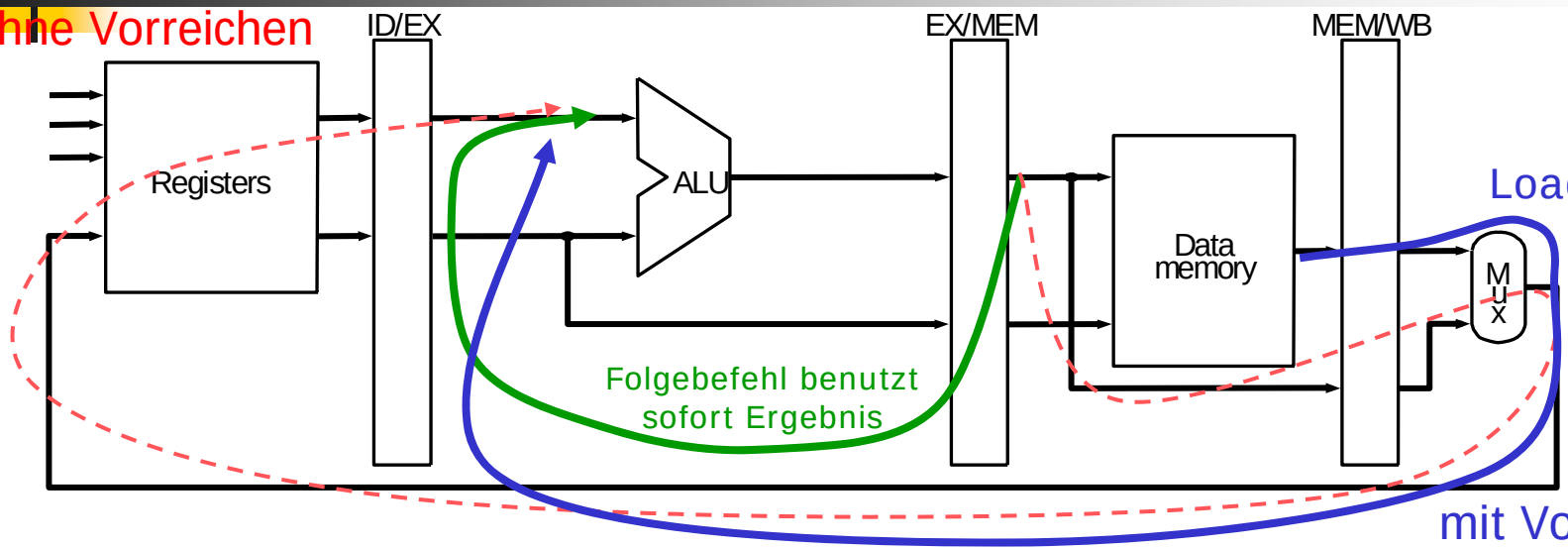
Time (in clock cycles)

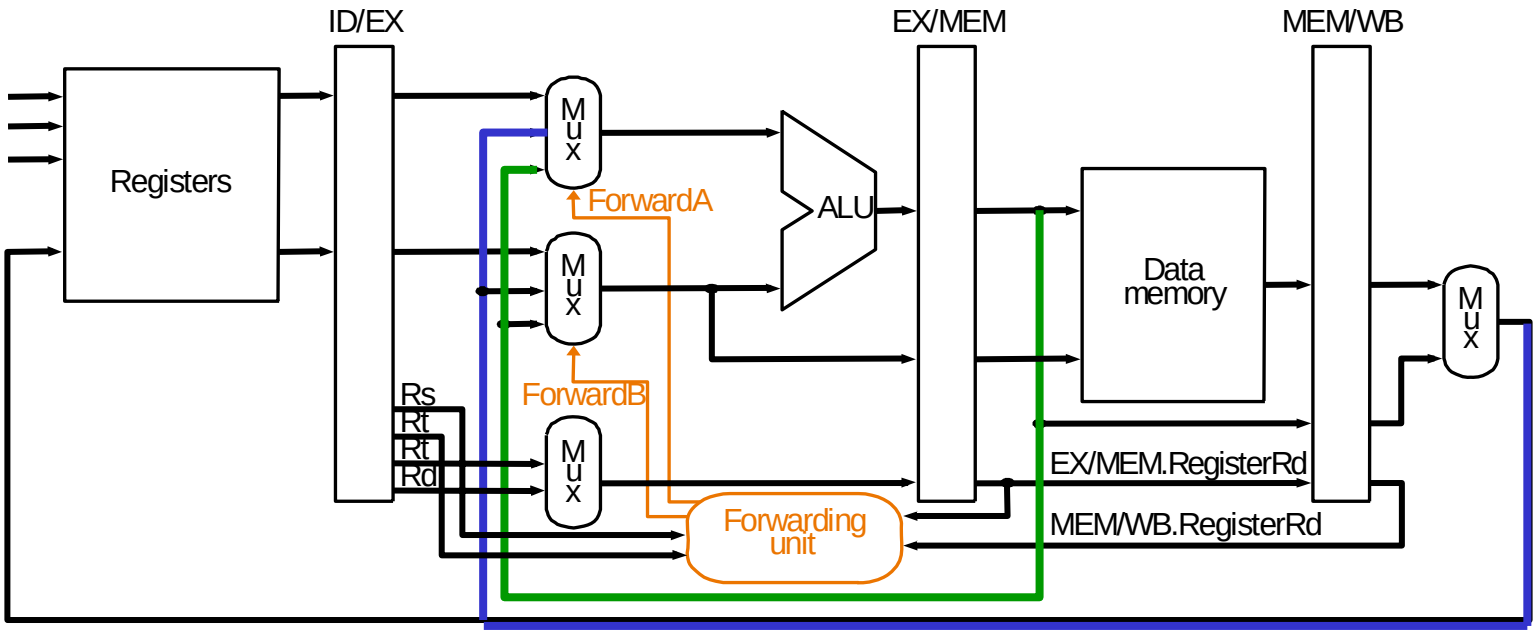
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM :	X	X	X	–20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	–20	X	X	X	X



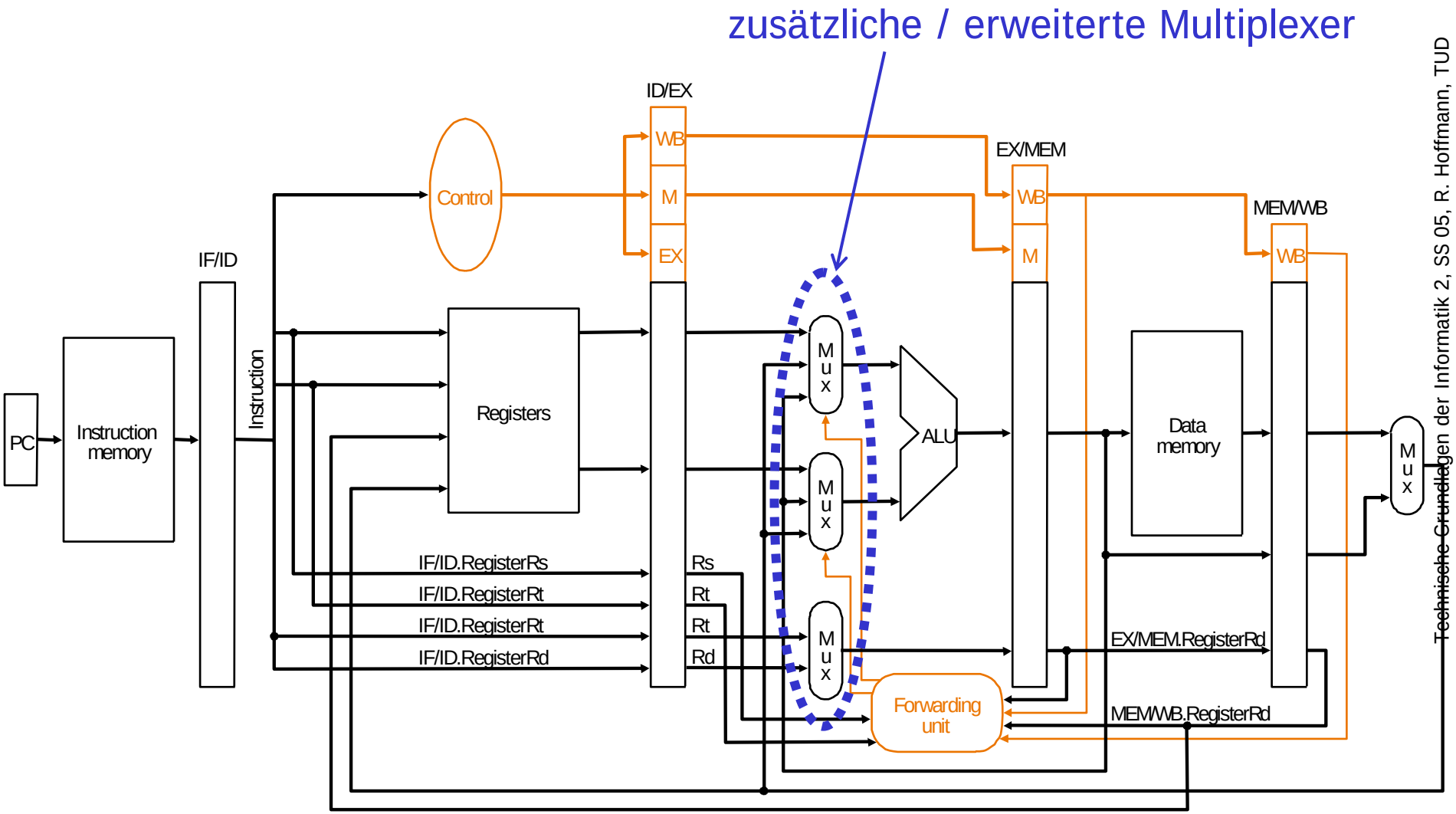
- Verwendung von bereits verfügbaren Zwischenergebnissen in Pipeline-Registern ohne Speicherung im Registersatz abzuwarten
- Vorreichen durch Registersatz: Schreiben/Lesen auf gleichem Register im gleichen Zyklus (z. B. Zyklus 5 im Bild, 1. Phase: Schreiben, 2. Phase: Lesen)
- Vorreichen zur ALU (von EX/MEM bzw. MEM/WB)

Hardware-Erweiterung fürs Vorreichen

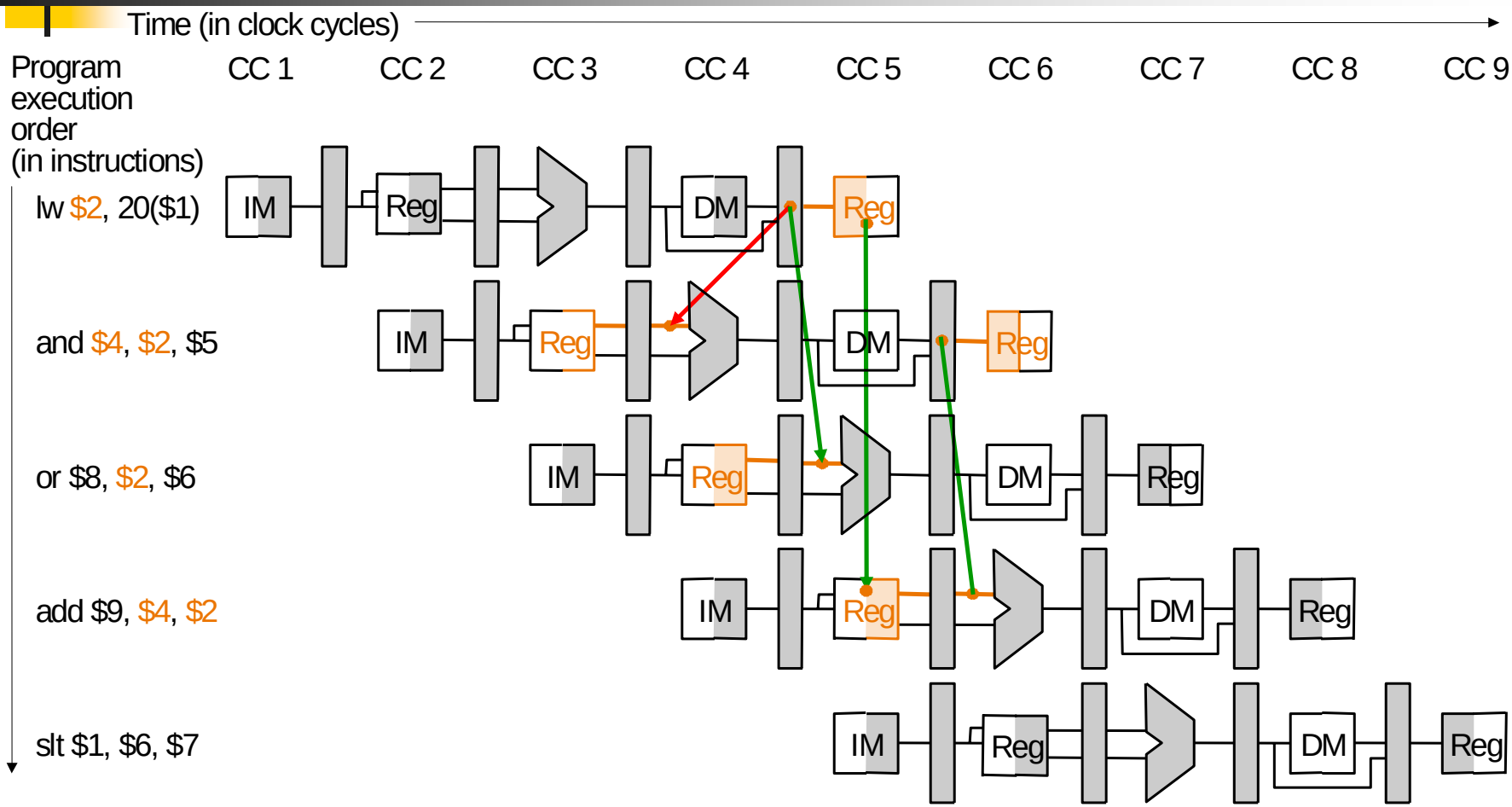




Hardware-Erweiterung fürs Vorreichen

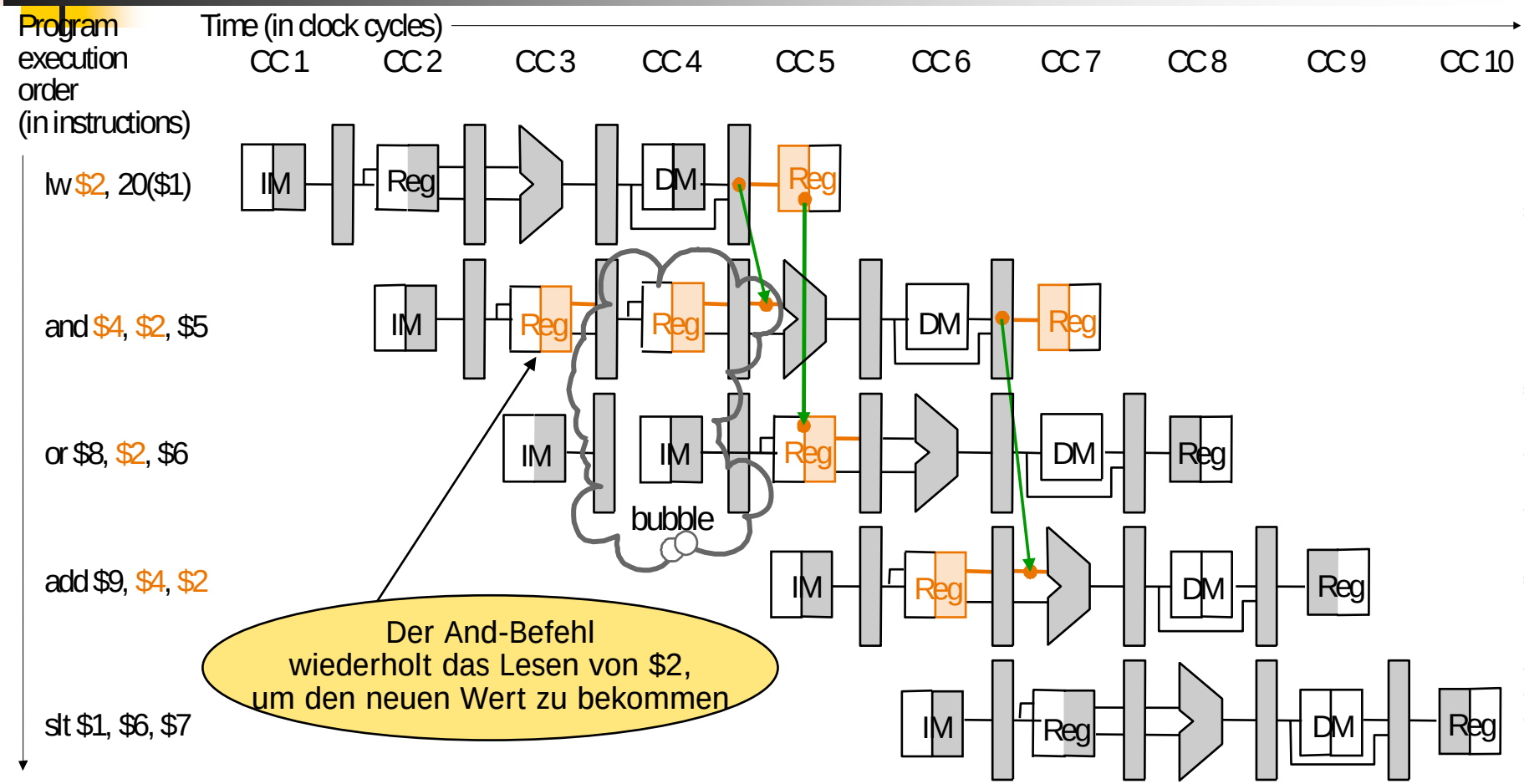


Nicht immer kann man vorreichen



- `lw` kann immer noch eine Datenhürde verursachen.
- Beispiel: Nachfolgender Befehl versucht, das Register zu lesen, dessen Wert gerade aus dem Speicher geladen wird. Das Lesen muß abgewartet werden (1 Takt Verzögerung).

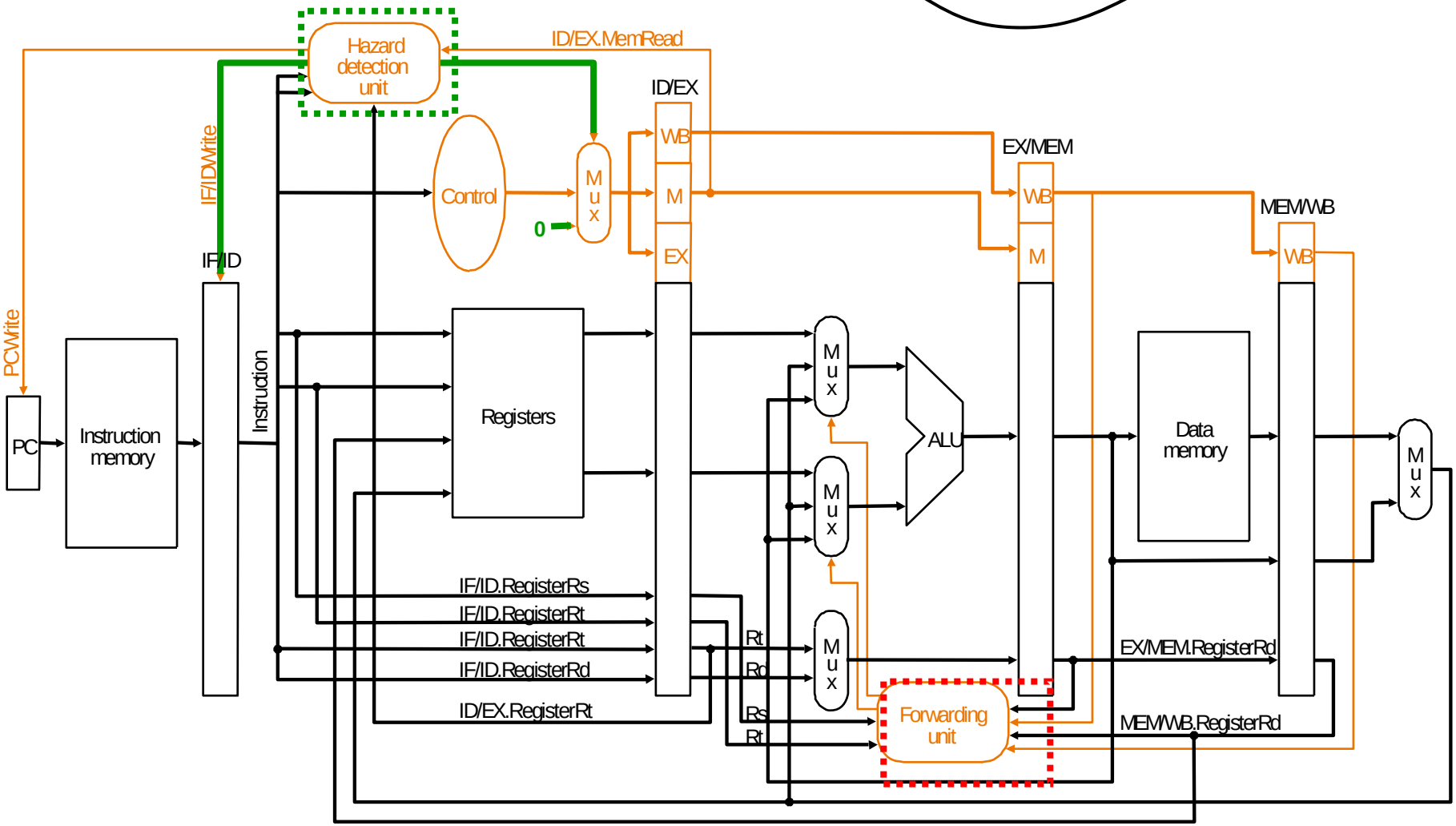
Lösung: Anhalten (Stall)



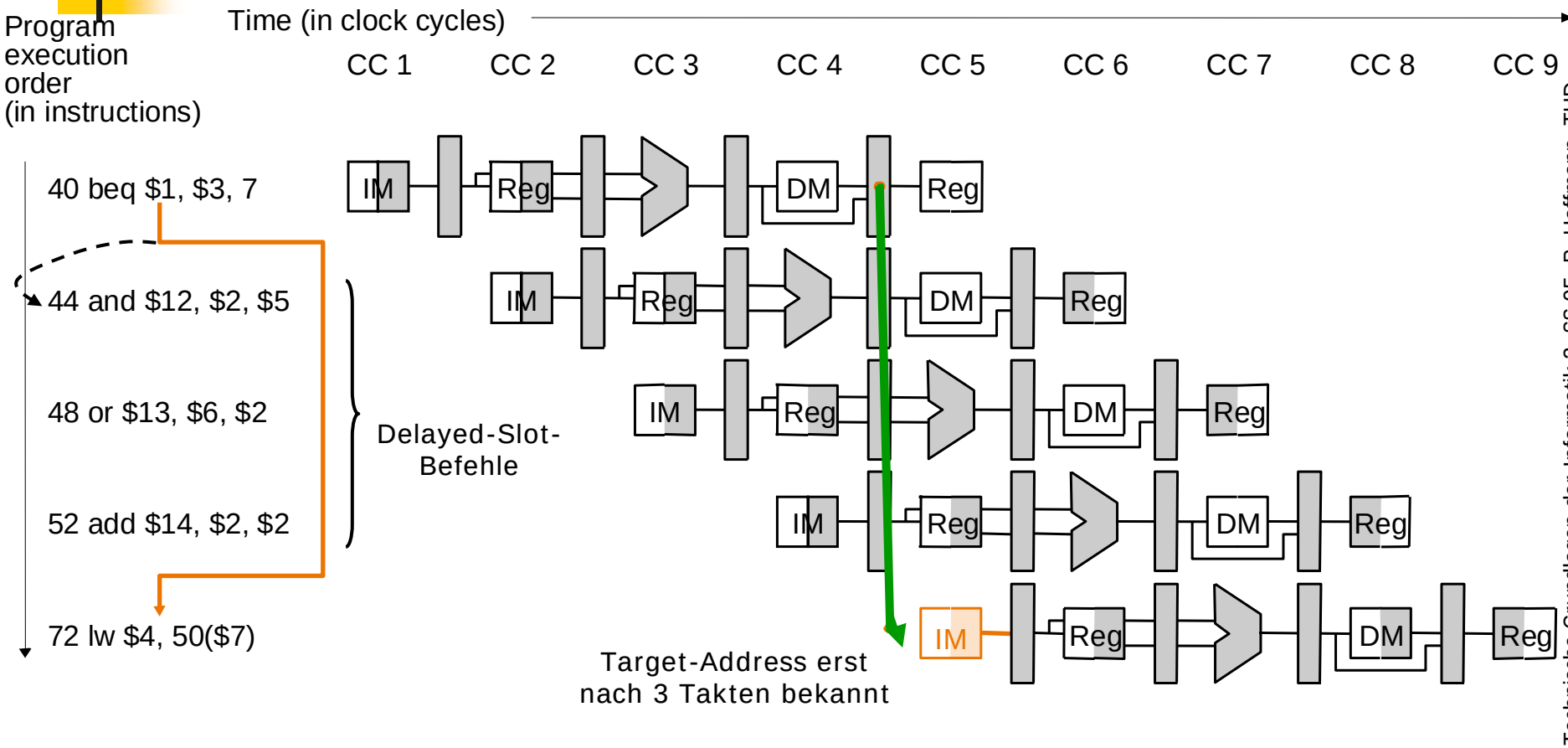
- Anhalten (stall) der Pipeline durch Belassen des Befehls im gleichen Abschnitt, d. h. Einfügen eines **nop** Befehls ab dem EX-Abschnitt (EX, WB, MEM-Steuersignale auf 0).
- Verhindern, daß PC fortgeschrieben und ID-Register geändert wird.

Anhalten der Pipeline

- Es wird eine **Hazard Detection Unit** benötigt, um den Folgebefehl rechtzeitig „aufzuhalten“. Sie erkennt die Befehlsfolge (lw **$2**.. , and .. **$2** ..)

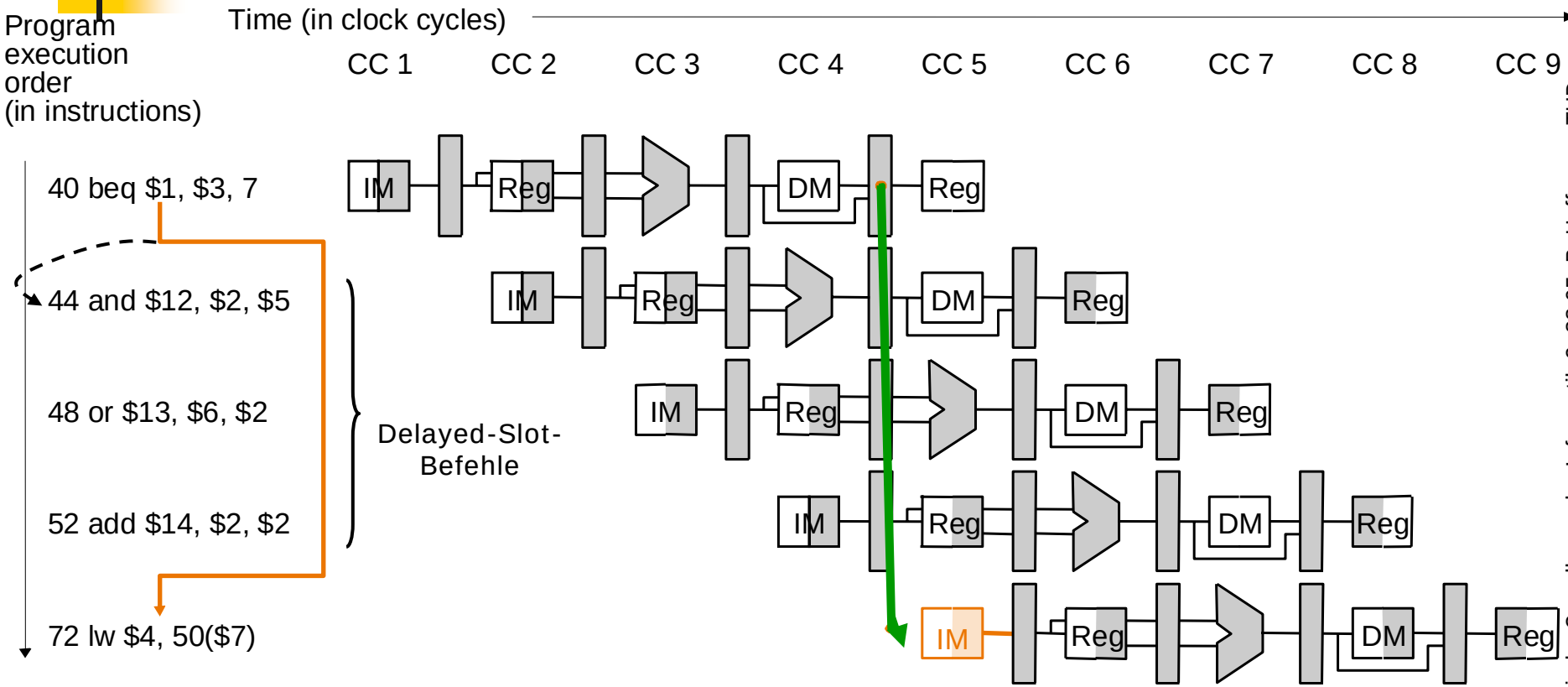


Steuerflußkonflikt bei bedingter Verzweigung (1)



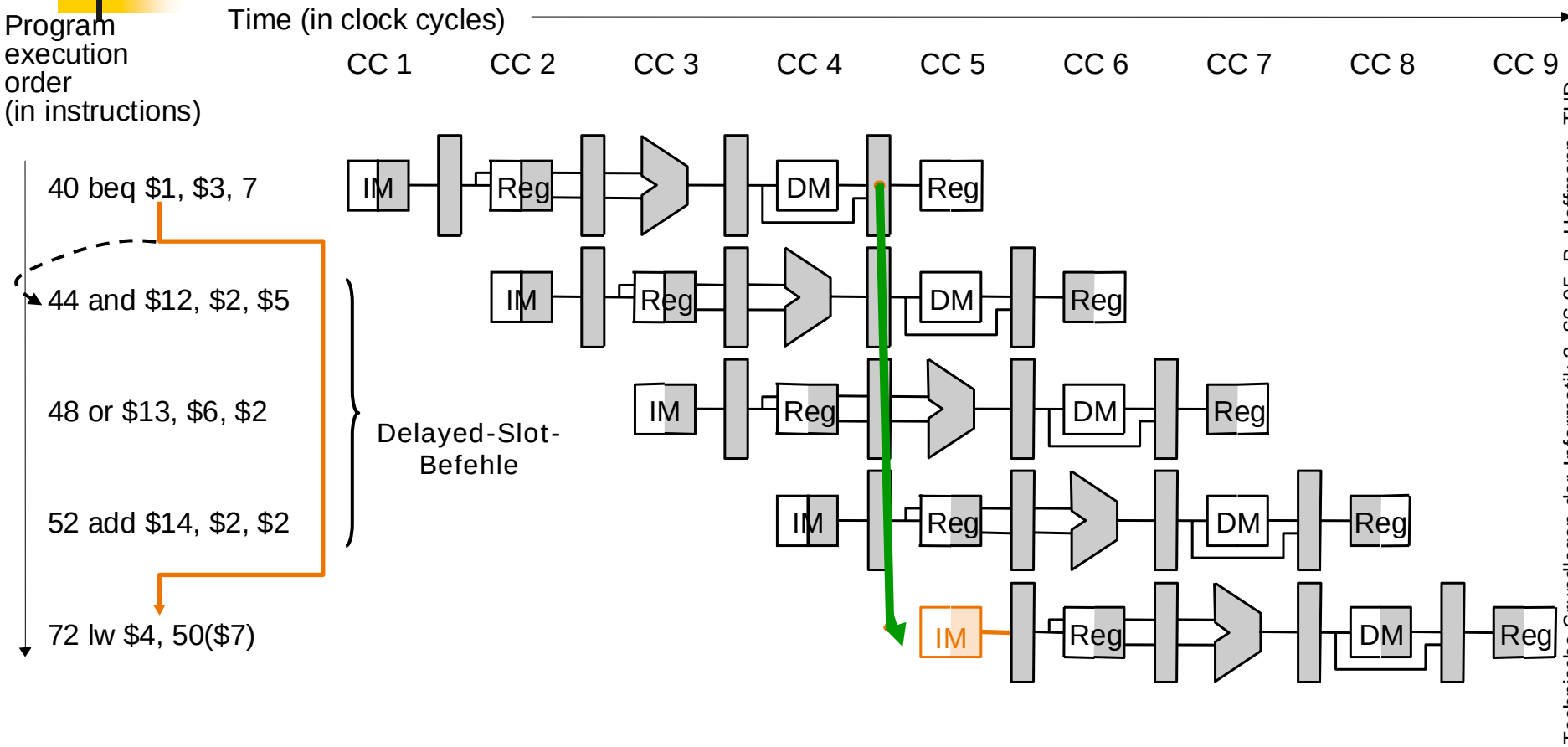
- Problem: Sprungzielberechnung erst nach der WB-Phase abgeschlossen. 3 Delayed-Slot-Befehle befinden sich schon in der Pipeline.
- **1. Lösung:** Anhalten (Beispiel: 3 Anhalte-(stall)-Zyklen) bis Sprungziel ermittelt, um die Ausführung der Delayed-Slot-Befehle zu verhindern!

Steuerflußkonflikt bei bedingter Verzweigung (2)



- **2. Lösung:** Annahme kein Sprung (Branch not taken) und Folgebefehle (Delayed-Slot-Befehle) starten; bei nachträglicher Erkennung eines Sprunges →
  - (a) DS-Befehle regulär beenden. Die Delayed-Slot-Befehle werden immer ausgeführt und können ggf. eine sinnvolle Berechnung (unabh. vom Sprung durchführen). Auffüllung mit nops, falls keine sinnvollen Aktionen möglich.
  - (b) Der alte Zustand wird restauriert. Wirkung der DS-Befehle rückgängig machen.

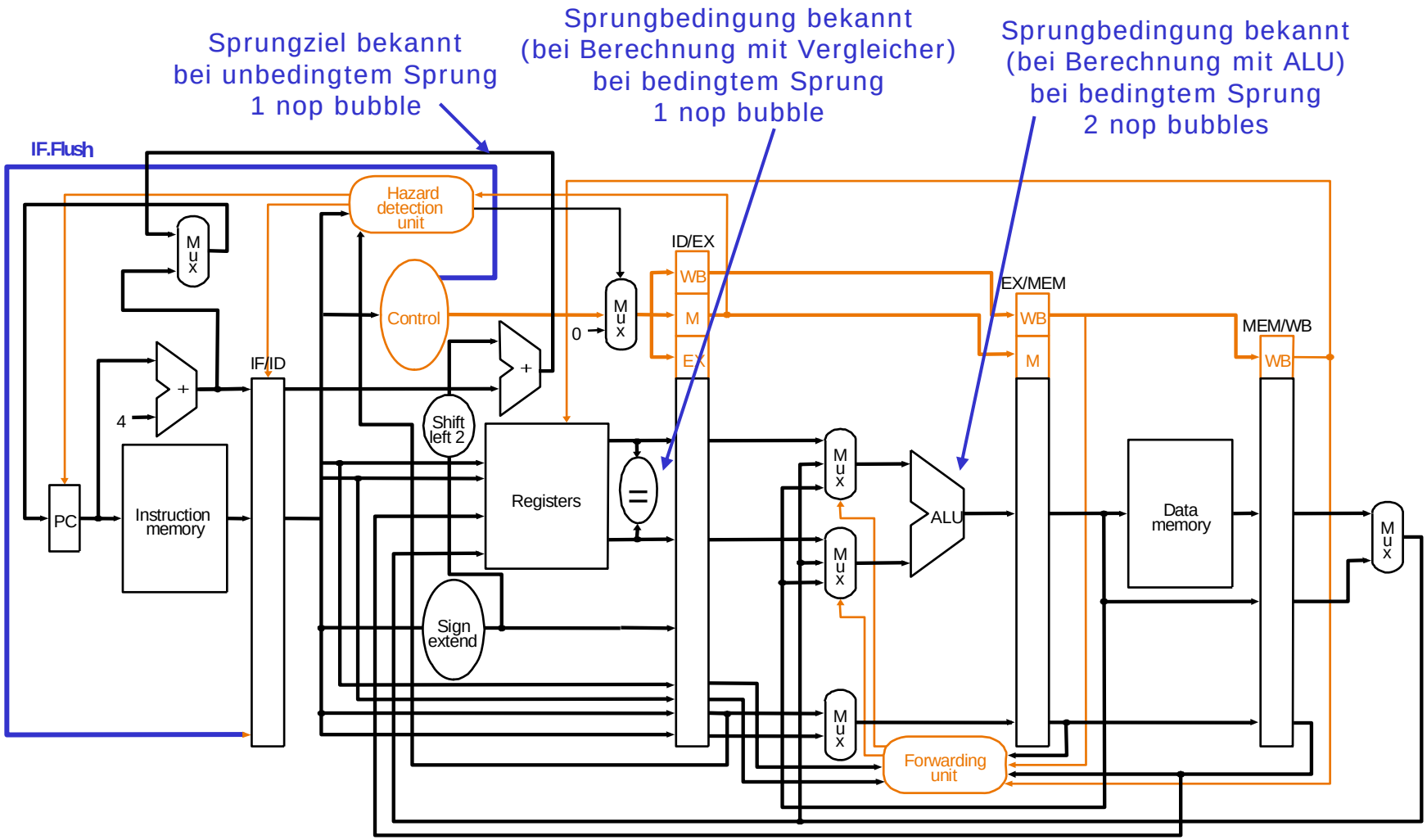
Steuerflußkonflikt bei bedingter Verzweigung (3)



- **3. Lösung:** Der dynamischen Programmablaufs wird verfolgt und die Sprünge in der Vergangenheit werden aufgezeichnet. Aus dieser Information wird eine Sprungvorhersage getroffen. Dadurch wird mit hoher Wahrscheinlichkeit der richtige Weg eingeschlagen.

Steuerung mit Verkürzung der Verzweigung

- **4. Lösung:** Sprungzielberechnung von 4. Stufe auf 2. Stufe vorziehen und Hardware zum Löschen (flush) des bereits geholten Befehls (steht vor dem IF/ID-Register) durch Überschreiben mit NOPs.



Weitere Leistungssteigerung (1)

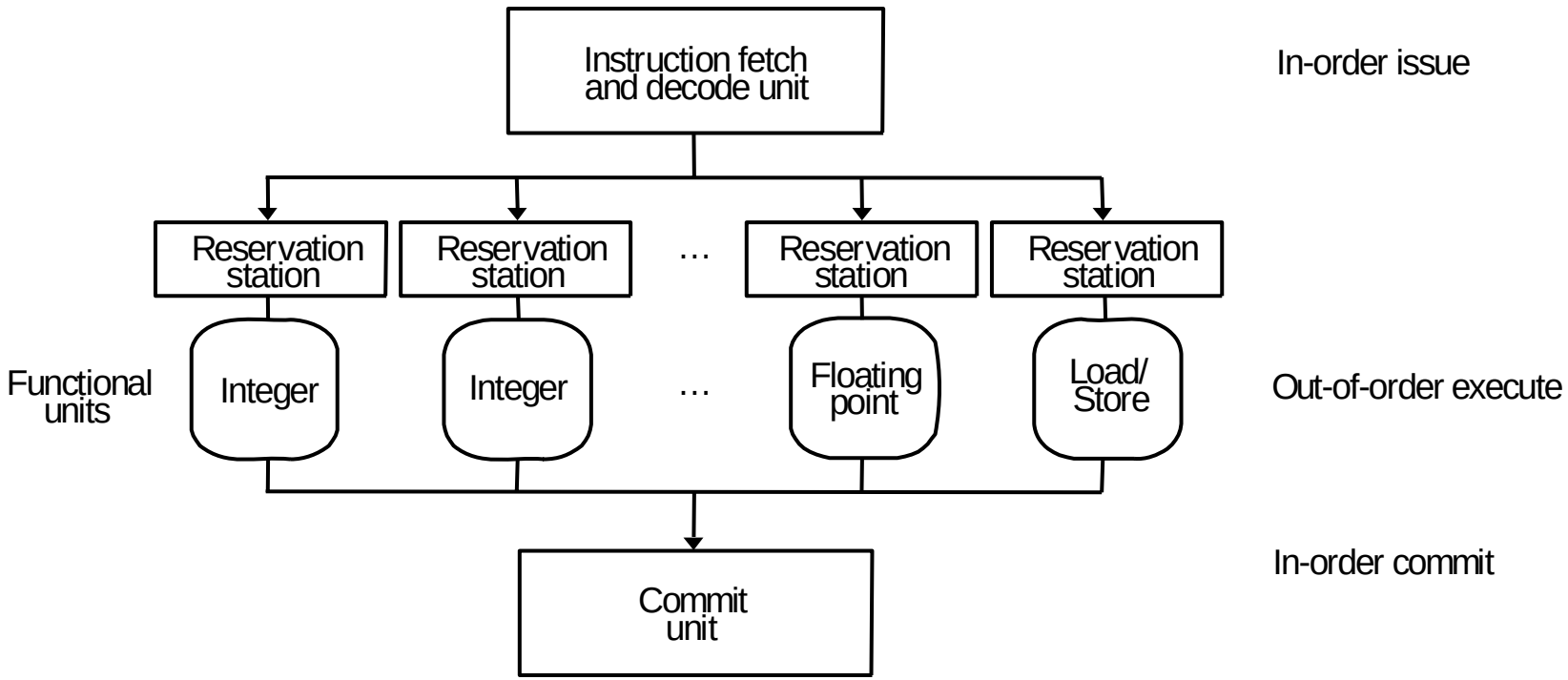
- Compiler: Vermeiden des Anhaltens der Pipeline von Compilerseite durch Umordnung von Befehlen (Lösung 2 a)
- Superpipelining: Längere Fließbänder mit 8 und mehr Stufen
- Superskalare Prozessoren:
  - In jedem Taktzyklus Start mehrerer Befehle (z. B. 2 bis 8).
  - Voraussetzung: Zusätzliche Hardware-Einheiten notwendig zur Vermeidung von Ressourcenkonflikten. Z. B. Start zweier ALU-Operationen erfordert zwei ALUs.
- z. B. superskalare Version von MIPS
  - Gleichzeitig eine ALU- oder Verzweigung sowie eine Lade- oder Speicheroperation aktivierbar.
  - In einem 64 Bit Befehlsregister können zwei aufeinander folgende Befehle abgelegt und decodiert werden.

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Weitere Leistungssteigerung (2)

Dynamisches Pipelining (dynamic pipeline scheduling):

- Hardware zum Aufspüren und Abarbeiten **unabhängiger Befehle**
- in-order issue: Befehle werden in Programmordnung aktiviert
- out-of-order execution: Ausführung in unabhängigen funktionalen Einheiten mit Befehlspuffern (reservation stations) und Ergebnispuffern
- in-order commit: fertig bearbeitete Befehle werden in Programmreihenfolge in die Architektur-Register oder Speicher gebracht.



Aus der Technik

- Alle modernen Prozessoren sind äußerst komplex.
- Compaq/DEC Alpha 21264:
  - 9-stufige Pipeline, 6 Befehle gleichzeitig
- Intel/HP Itanium:
  - bis zu 6 IA-64-Befehle gleichzeitig
- DEC Alpha 21164, PowerPC G3, MIPS R12000, UltraSparc-II, HP PA-8500:
  - bis zu 4 Befehle pro Takt
- PowerPC, Pentium:
  - Sprungvorhersage-Tabelle