

# 6. Pipelines und Instruction-Level Parallelism

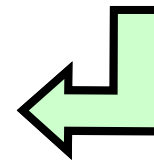
---

- **Wie können wir die Performance weiter steigern?**
  - Technologieverbesserungen
    - geringere Verzögerungszeiten
    - höhere Taktfrequenz
    - führt häufig aber auch zu mehr Verlustleistung (Kühlung!)
- **Wir suchen nach weiteren Architekturverbesserungen.**
  - bisher
    - sequentielle Verarbeitung von Befehlen
    - ist ausgereizt
      - Befehle benötigen so viele Takte (und damit so viel Zeit), wie sie eben benötigen.
  - einziger Ausweg
    - Irgendwie müssen mehrere Instruktionen parallel ausgeführt werden.
    - *Instruction-Level Parallelism* (ILP)

# Instruction-Level Parallelism

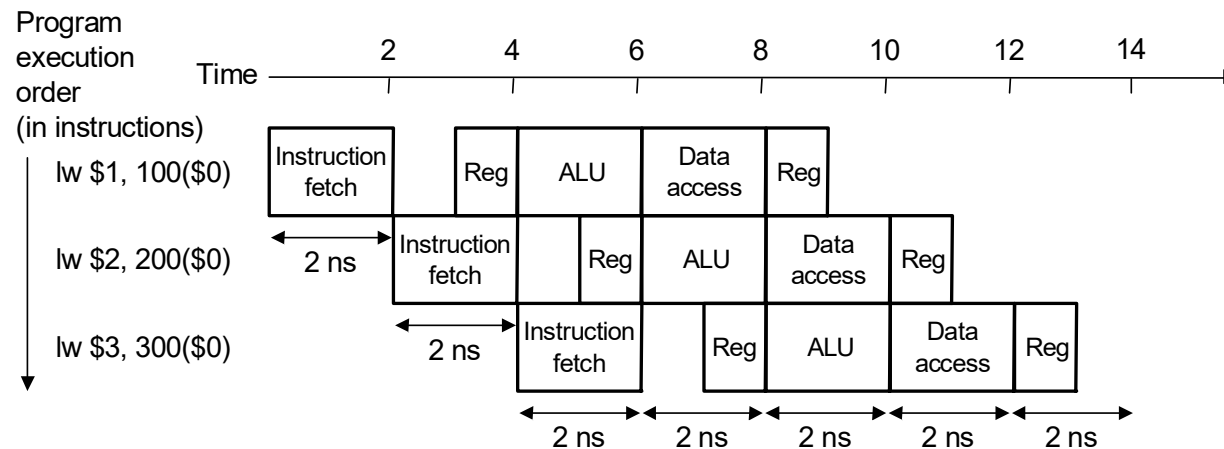
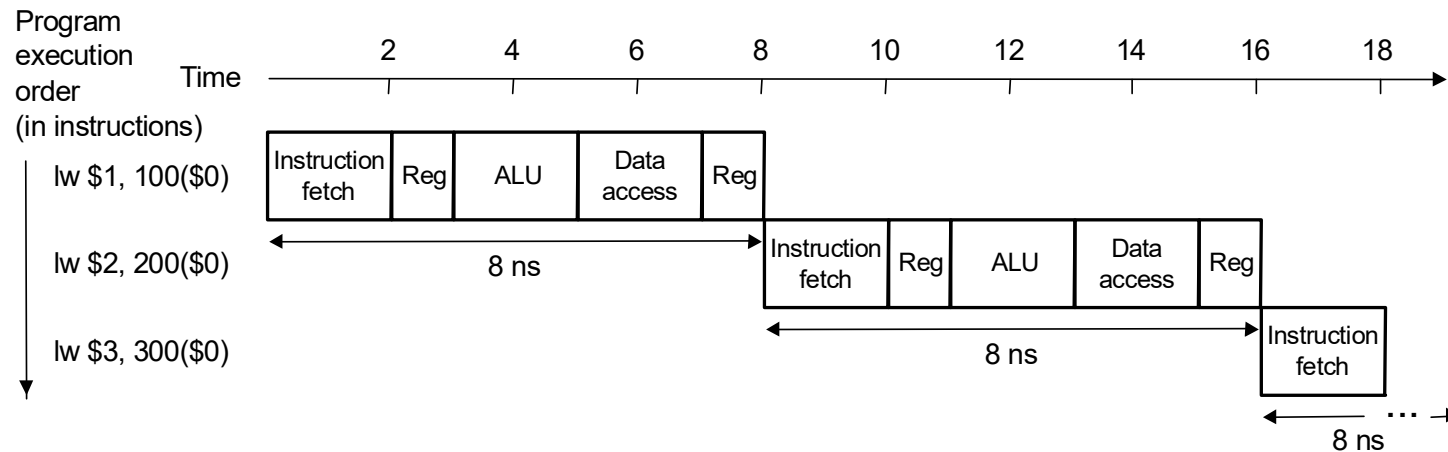
---

- **Verschiedene Möglichkeiten ILP auszunutzen**
  - Pipeline
  - Superskalare Architekturen
  - Dynamic Pipeline Scheduling
- **Wann können sequentielle Instruktionen parallel verarbeitet werden?**
  - wenn es keine Datenabhängigkeiten zwischen den Operationen gibt
    - wenn eine Instruktion das Ergebnis einer anderen benötigt, können die Instruktionen nicht parallel abgearbeitet werden
      - die Reihenfolge muss dann beibehalten werden
      - allerdings dürfen die Instruktionen *zumindest teilweise* überlappen
- **Idee der Pipeline!**



# Pipelines

- starte die Bearbeitung des nächsten Befehls, sobald der erste Schritt des aktuellen Befehls beendet wurde



# Pipelines (2)

---

- **Speedup**

- Beschleunigungsfaktor für den Durchsatz gegenüber einer Architektur ohne Pipeline
- maximaler Speedup ist die Anzahl der Stufen  $k$  in der Pipeline
  - alle  $k$  Stufen der Pipeline arbeiten parallel an verschiedenen Befehlen (ILP)
  - Mehrfachausnutzung der Einheiten wie im Multizyklus-Datenpfad ist daher nicht möglich
  - Pipeline-Datenpfad basiert deshalb auf dem Ein-Zyklus-Datenpfad
- der Durchsatz wird erhöht
- die Latenzzeit wird nicht reduziert
  - wird sogar gegenüber Mehrzyklus-Datenpfad im Mittel vergrößert, da alle Befehle alle  $k$  Stufen durchlaufen müssen

- **Pipelines sind *das* Schlüsselement, um moderne Prozessoren (seit ca. 1985) schnell zu machen**

# Pipelines (3)

---

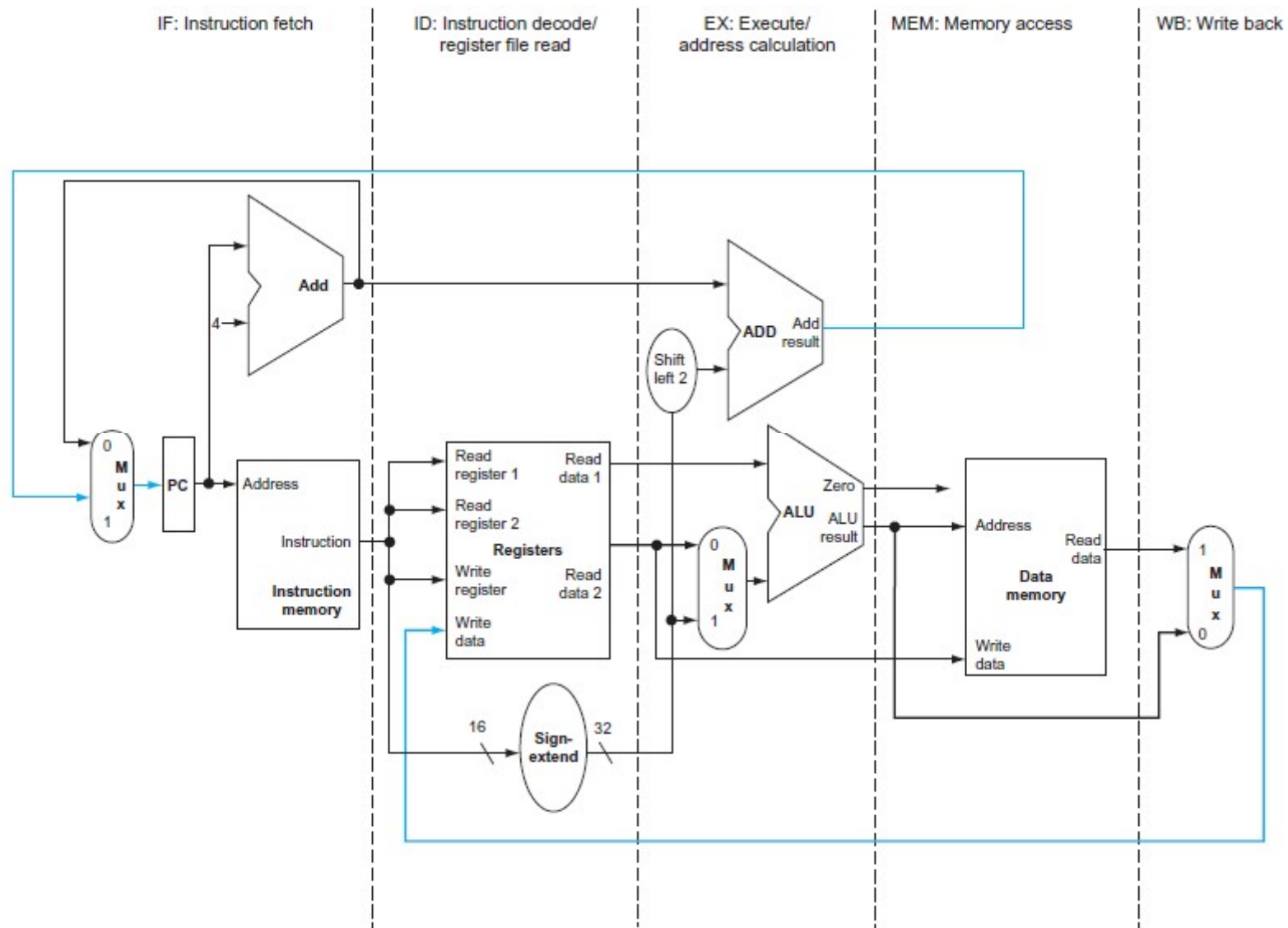
- **Was macht es in unserem Fall einfach?**
  - alle Instruktionen bestehen aus einem Wort
  - nur wenige Instruktions-Formate
  - Speicher-Operanden nur in `load` und `store` Instruktionen
- **Was macht es auch in unserem Fall schwierig?**
  - Konflikte (Hazards)
    - Struktur-Konflikte (*structural Hazards*)
      - verschiedene Pipelinestufen greifen gleichzeitig auf nur einmal vorhandene Ressourcen zu (z.B. Speicher)
    - Datenabhängigkeits-Konflikte (*Data Hazards*)
      - Instruktion benötigt Ergebnis einer vorherigen Instruktion
    - Kontrollfluss-Konflikte (*Control Hazards*)
      - z.B. bei bedingten Sprüngen: wo steht der nächste auszuführende Befehl?

# Pipelines (4)

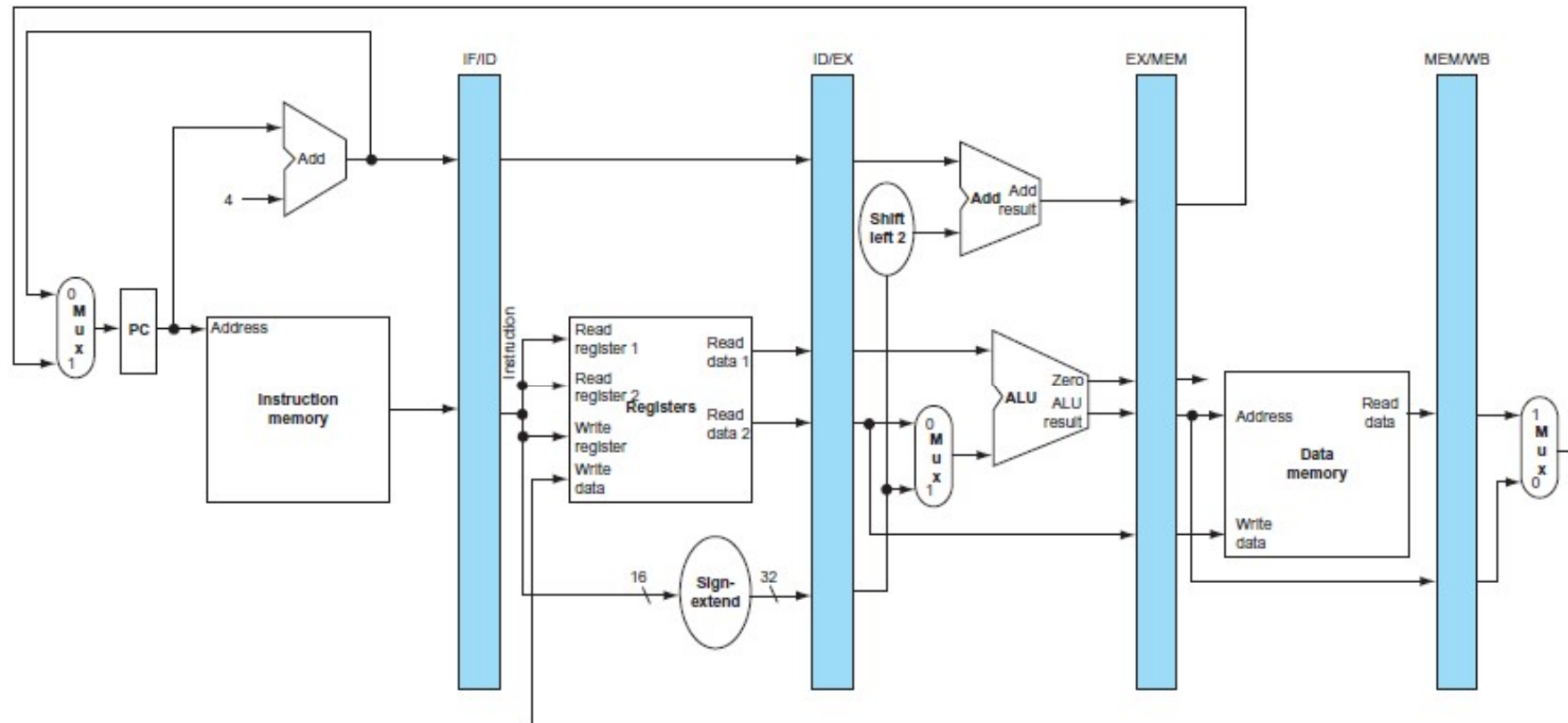
---

- Wir werden eine einfache Pipeline bauen und uns diese Themen ansehen.
- Wir werden über moderne Prozessoren reden und zeigen, was es wirklich schwierig macht.
  - Behandlung von Exceptions
  - Erhöhung der Performance durch out-of-order Execution, etc.

# Idee: Ausgangspunkt Ein-Zyklus-Datenpfad



# Datenpfad mit Pipelineregister





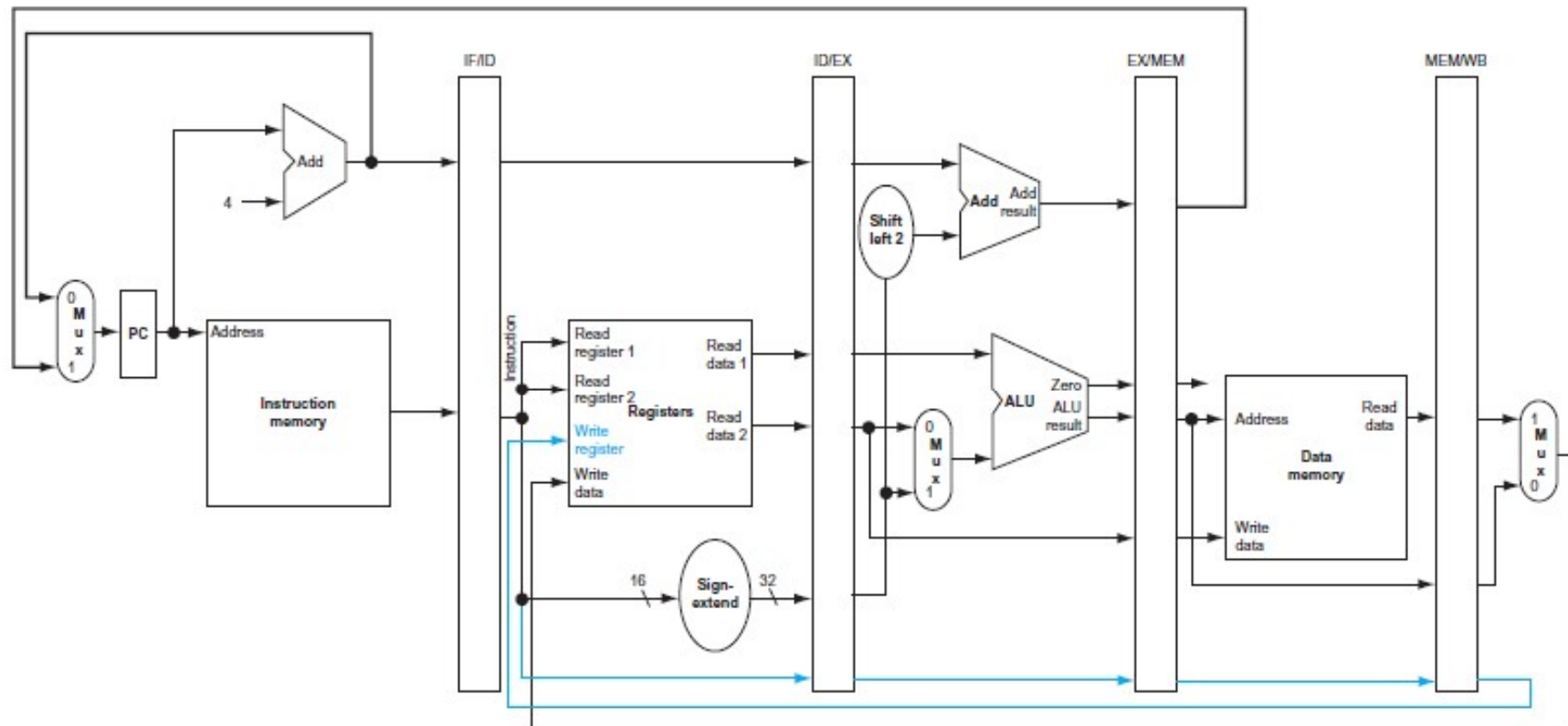
# Problem mit diesem Datenpfad

---

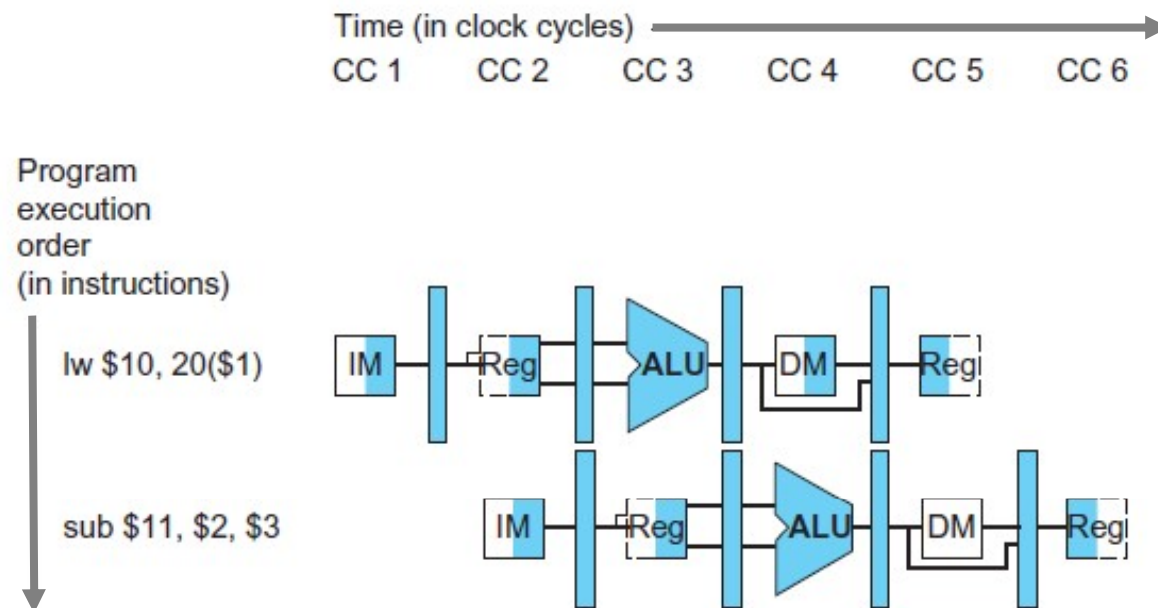
- Datenfluss fast überall von links nach rechts
- Nur beim Zurückschreiben der Ergebnisse fließen Daten von rechts nach links.
  - Schreiben der neuen Adresse in den PC
  - Schreiben der Ergebnisse in die Register
- *Wo ist das Problem mit diesem Datenpfad?*
- *Gib ein Beispiel für eine Instruktion, die das Problem zeigt!*

Die Registernummer für das Schreiben muss mit den Daten mitgeführt werden.

# Korrigierter Datenpfad



# Grafische Darstellung für Pipelines



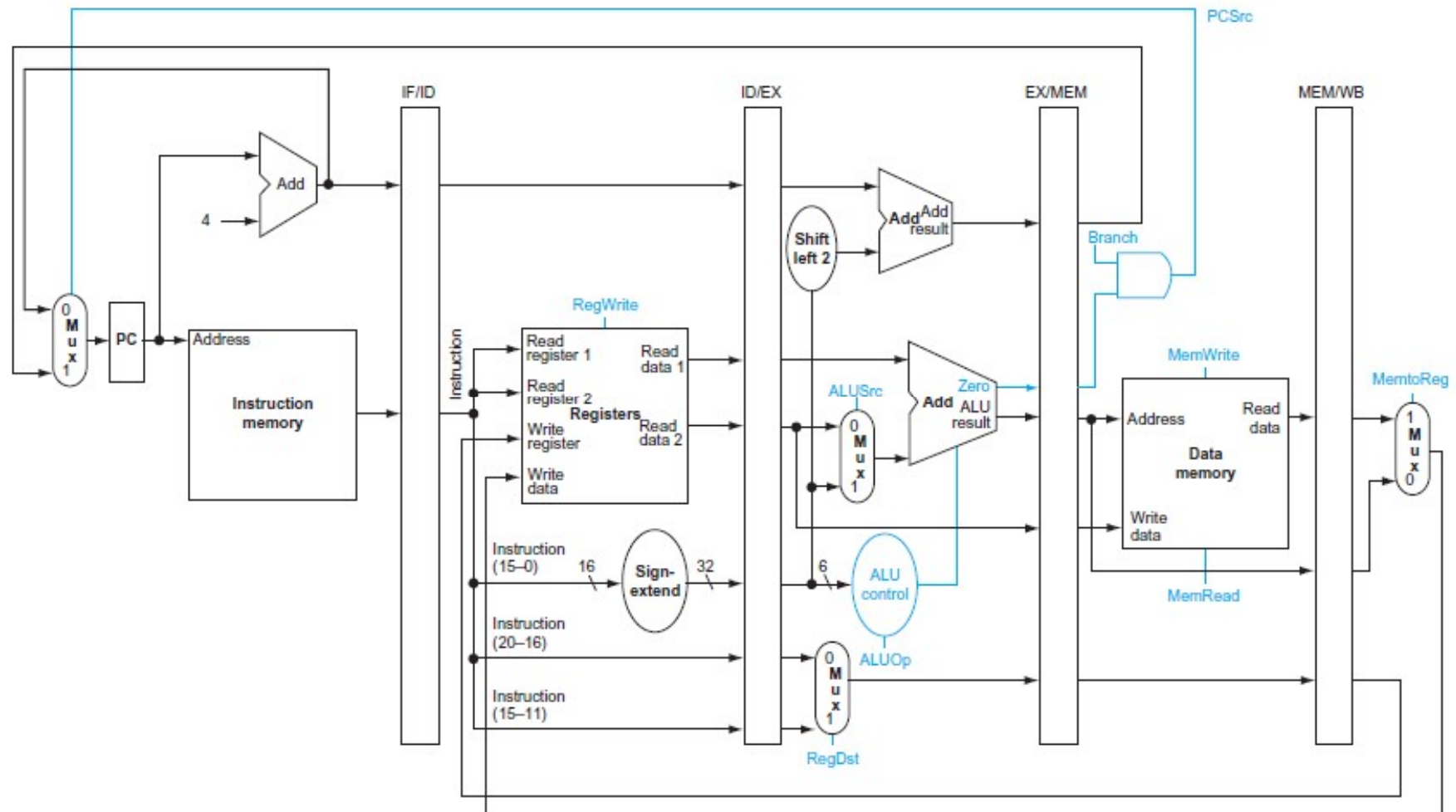
- Schattierung zeigt, ob ein Element in einem Taktzyklus benutzt wird
  - links blau: wird zum Schreiben benutzt
    - Daten müssen bereits vor der steigenden Taktflanke vorhanden sein (*setup time*) und werden mit der Taktflanke übernommen
  - rechts blau: wird kombinatorisch ausgelesen
    - Daten sind am Ausgang eine kurze Zeit nach Änderung der Eingangssignale (die mit der steigenden Flanke erfolgt) sichtbar

# Grafische Darstellung für Pipelines (2)

---

- dient zur Beantwortung von Fragen wie
  - In welchem Taktzyklus ist eine bestimmte Instruktion fertig?
  - Was macht die ALU in Taktzyklus 4?

# Pipeline Steuersignale



# Pipeline Steuerung (2)

---

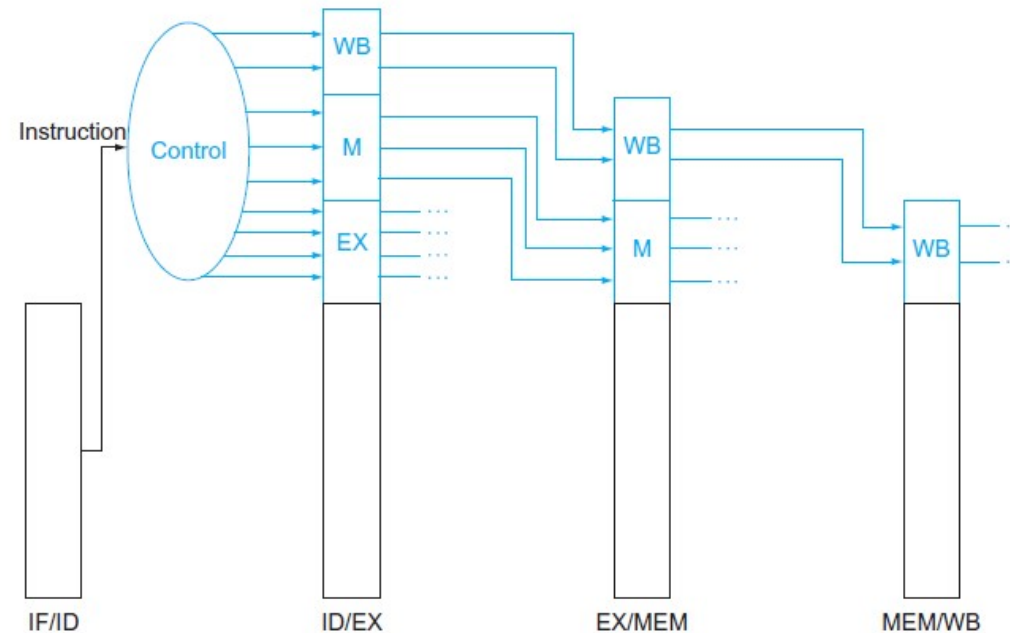
- Was muss in jeder Stufe gesteuert werden?
  - Instruction Fetch und PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back
- Wie würde man die Steuerung an einem Fließband in der Autoproduktion handhaben?
  - Ein raffiniertes Steuerungszentrum, das jedem jederzeit sagt, was er zu tun hat?
  - Sollten wir also eine Finite-State-Machine benutzen?
    - die bis zu 5 verschiedene Instruktionen simultan in den verschiedenen Stufen steuert?

Besser: man klebt an jedes Auto einen Zettel,  
auf dem steht, was in jeder Stufe zu tun ist

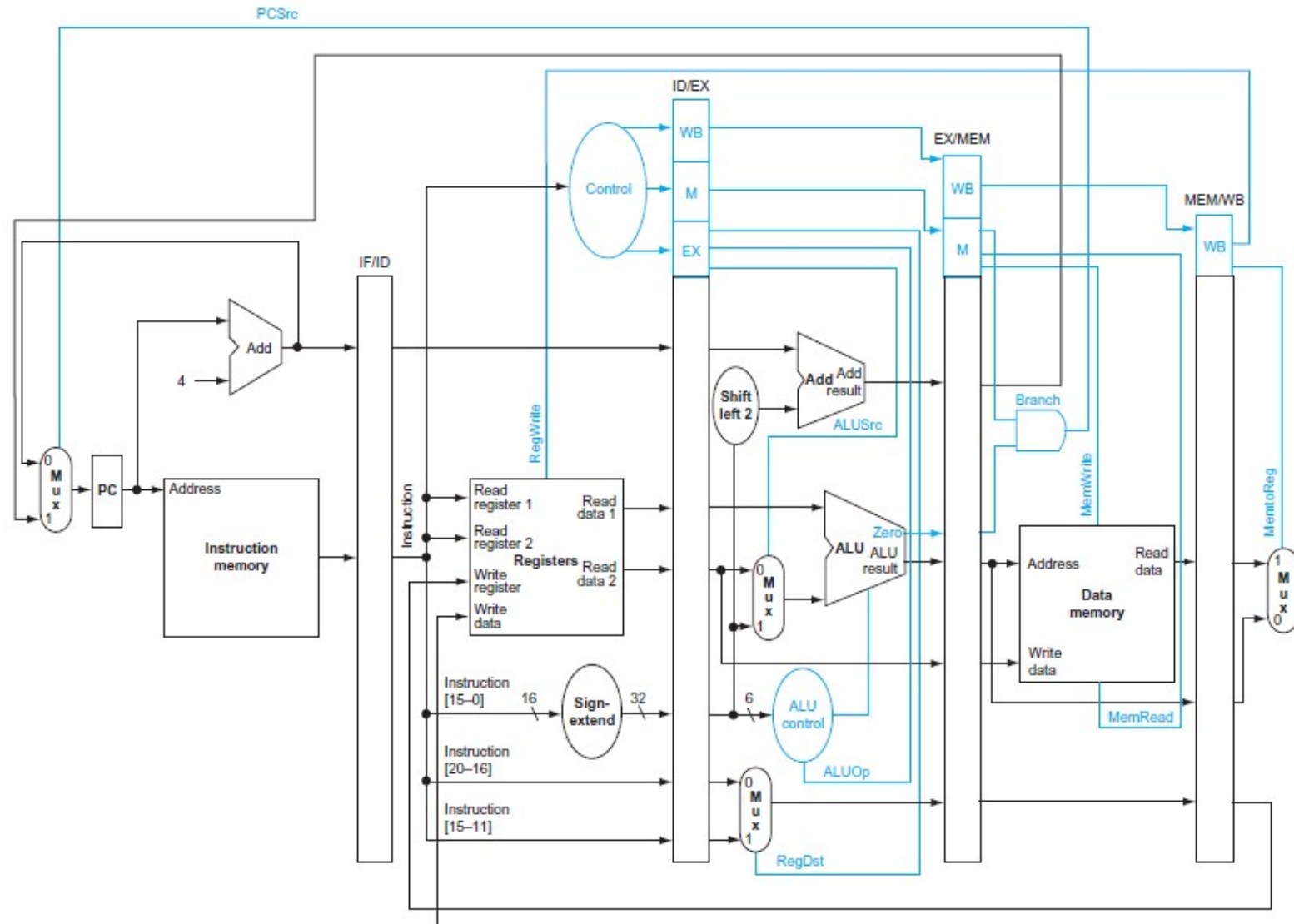
# Pipeline Steuerung (3)

- Reiche Steuersignale mit den Daten weiter

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-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



# Datenpfad mit Steuerung





# Zusammenfassung Pipeline-Architektur

---

- **Datenpfad**
  - Funktionale Einheiten wie beim Ein-Zyklus-Datenpfad
  - Register nach funktionalen Einheiten wie beim Mehrzyklus-Datenpfad
- **Steuerung**
  - Erzeugung der Steuersignale wie beim Ein-Zyklus-Datenpfad (also Schaltfunktion)
  - Steuersignale für weiter hinten liegende Pipelinestufen werden mit den Daten weitergereicht (Pipeline-Register für Steuersignale)
- **Warum ist das Ganze schneller als beim Mehrzyklus-Datenpfad?**
  - ein einzelner Befehl benötigt immer 5 Takte (Latenzzeit)
    - wie der worst case beim Mehrzyklus-Datenpfad
  - nach dem Füllen der Pipeline erhält man in jedem Takt ein Ergebnis (Durchsatz)
    - wie beim Ein-Zyklus-Datenpfad
    - aber ähnlich hoher Takt wie beim Mehrzyklus-Datenpfad

# Pipeline-Konflikte (Hazards)

---

- **Leider ist die Realität nicht ganz so einfach**
  - Pipelinekonflikte
  - drei Arten: Struktur-, Datenabhängigkeits- und Kontrollfluss-Konflikte
- **Struktur-Konflikte**
  - zwei Stufen der Pipeline benötigen dieselbe Ressource
  - Beispiele
    - Beispiel: gemeinsamer Daten- und Instruktionsspeicher
      - Konflikt: IF holt Befehl und MEM holt Daten
      - Abhilfe (in unserem Datenpfad schon realisiert)
        - » getrennter Speicher für Instruktionen und Daten (Harvard Architektur)
    - Beispiel: arithmetische Instruktionen würden in MEM Stufe Ergebnisse ins Register schreiben
      - MEM und WB würden auf Register File zugreifen
  - die MIPS-Architektur vermeidet Strukturkonflikte bereits vollständig

# Datenabhängigkeiten

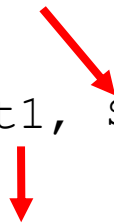
- Instruktion  $j$  ist datenabhängig von Instruktion  $i$ , wenn
  - Instruktion  $i$  ein Resultat erzeugt, das von Instruktion  $j$  benötigt wird
  - Instruktion  $j$  datenabhängig von Instruktion  $k$ , und Instruktion  $k$  datenabhängig von Instruktion  $i$  für irgendein  $k$  ist

- Beispiel

```
add  $t1, $t2, $t3
```

```
addi $t1, $t1, 1
```

```
sw   $t1, 4($s2)
```



- Datenabhängigkeiten sind Eigenschaften von Programmen
- ob Datenabhängigkeiten zu einem Konflikt führen, hängt von der Prozessorarchitektur ab

# Datenabhängigkeiten (2)

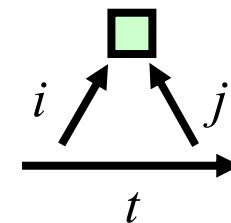
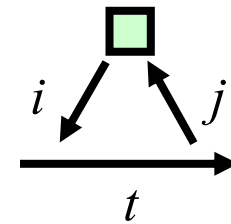
---

- **Daten fließen über Register oder Speicher**
  - Register
    - Datenabhängigkeiten können leicht entdeckt werden
    - Registernamen (bzw. –adressen) verändern sich nicht von Instruktion zu Instruktion
  - Speicher
    - Datenabhängigkeiten sind hier schon schwieriger zu entdecken
      - `sw $t1, 100($s1)` und `lw $t2, 0($s2)` verweisen vielleicht auf dieselbe Adresse (Inhalte von `$s1` und `$s2` müssten dazu analysiert werden)
      - `sw $t1, 0($s1)` und `lw $t2, 0($s1)` benutzen evtl. verschiedene Adressen (der Inhalt von `$s1` hat sich zwischendurch evtl. verändert)

# Namensabhängigkeiten

---

- treten auf, wenn zwei Instruktionen zwar dasselbe Register oder dieselbe Speicherstelle (hier Namen genannt) benutzen, aber *kein* Datenfluss zwischen diesen Instruktionen auftritt
- **im Folgenden gilt immer**
  - Instruktion  $i$  wird im Programm zeitlich vor Instruktion  $j$  ausgeführt
  - beide benutzen denselben Namen (Speicherort)
- **es gibt zwei Arten von Namensabhängigkeiten**
  - Anti-Abhängigkeit
    - Instruktion  $i$  liest und Instruktion  $j$  schreibt
    - Reihenfolge darf nicht vertauscht werden
      - sonst wird der falsche Wert gelesen
  - Ausgabe-Abhängigkeit
    - beide Instruktionen schreiben
    - Reihenfolge darf nicht vertauscht werden
      - sonst bleibt der falsche Wert am Ende gespeichert



# Namensabhängigkeiten (2)

---

- Namensabhängigkeiten sind keine echten Datenabhängigkeiten
- Instruktionen können parallel ausgeführt werden oder ihre Reihenfolge kann verändert werden, wenn man die Namen der Daten (also den Speicherort) wechselt
  - einfacher für Register
  - Austausch der Namen (Benutzen anderer Register!)
    - statisch durch Compiler
    - dynamisch durch Prozessorhardware

# Datenabhängigkeitskonflikte

---

- *Data Hazards*

- treten in Pipelines dann auf, wenn Datenabhängigkeiten zwischen Instruktionen existieren, die *zu dicht* aufeinander folgen
  - Umordnen der Reihenfolge oder Überlappen der Ausführung führen dann zu einer falschen Reihenfolge der Datenzugriffe
- die "Programmreihenfolge" muss erhalten bleiben
  - Reihenfolge, die bei sequentieller Abarbeitung gilt

- **Ziel**

- Parallelität ausnutzen wo immer es geht
- Programmreihenfolge dort erhalten, wo sonst Data Hazards auftreten würden

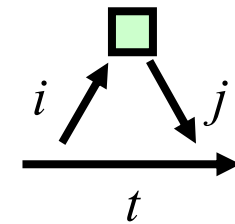
# Datenabhängigkeitskonflikte (2)

---

- **Klassifikation der Data Hazards (*i* liegt vor *j*!)**

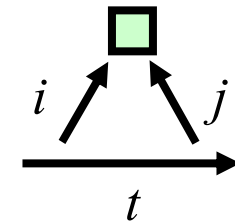
- RAW (*Read After Write*)

- Instr. *j* versucht einen Wert zu lesen, bevor Instr. *i* ihn schreibt
- echte Datenabhängigkeit
- häufigster Hazard
- tritt auch in Pipelines auf (s.u.)



- WAW (*Write After Write*)

- Instr. *j* schreibt einen Wert, bevor Instr. *i* ihn schreibt
- Ausgabe-Abhängigkeit
- am Ende bleibt der falsche Wert im Register stehen
- in Pipelines nur vorhanden, wenn in verschiedenen Stufen geschrieben wird
- könnte auftreten, wenn Instruktionen durch verschieden lange Pipelines laufen
  - z.B. Multiplikation in Floating Point Pipeline, Ergebnis nach  $\$f1$
  - anschließend *load* Instruktion in Register  $\$f1$ , die schneller endet

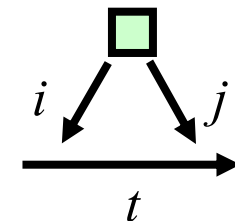
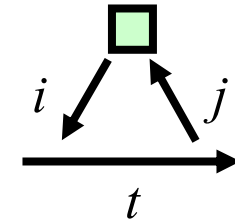




# Datenabhängigkeitskonflikte (3)

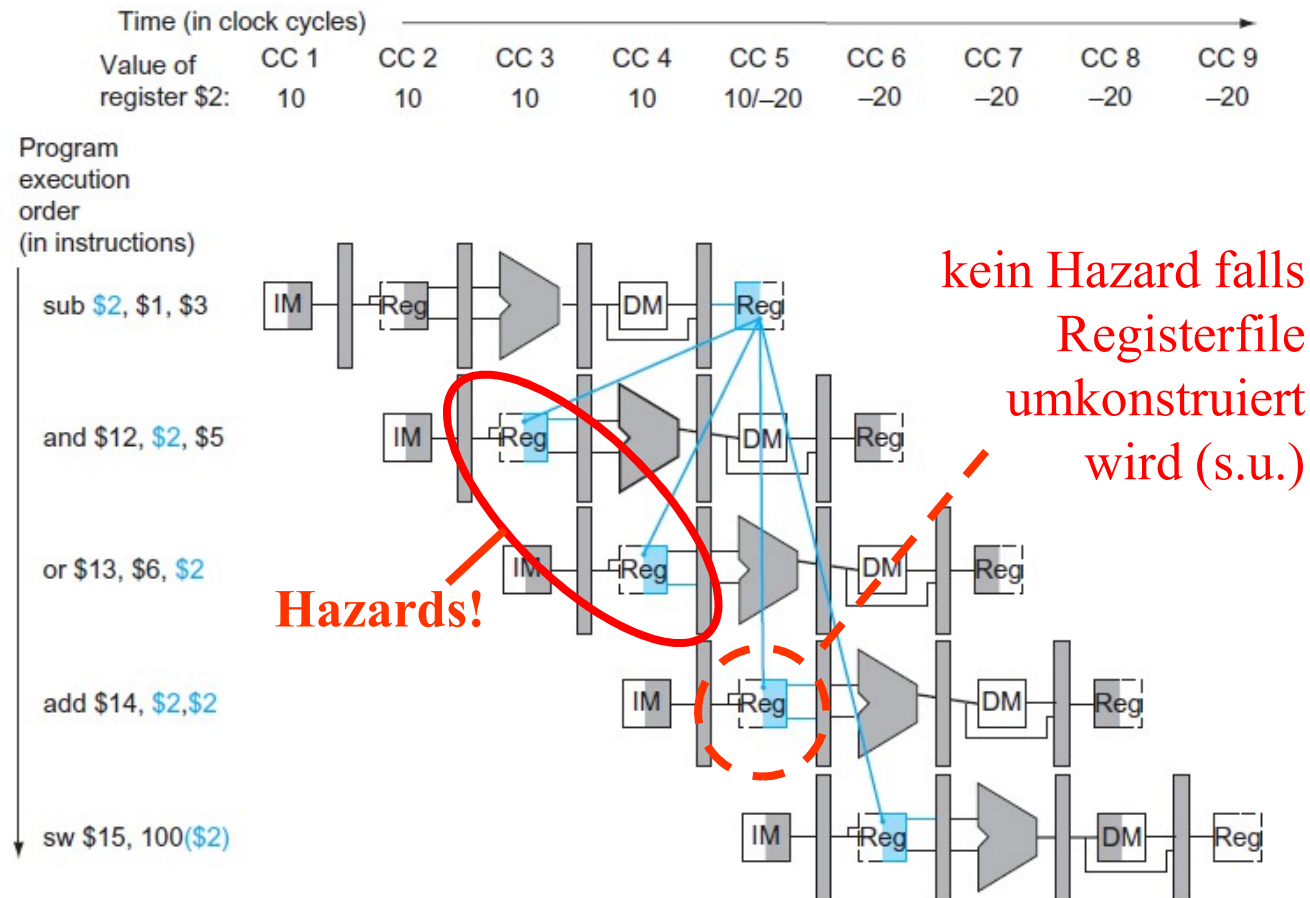
---

- WAR (*Write After Read*)
  - Instr.  $j$  schreibt einen Wert, bevor Instr.  $i$  ihn liest
  - Anti-Abhängigkeit
  - Instr.  $i$  liest irrtümlich schon den neuen Wert
  - tritt normalerweise in Pipelines nicht auf
    - Lesen findet in frühen Stufen der Pipeline statt
    - Schreiben findet in späten Stufen statt
  - kann nur auftreten wenn es Befehle gibt, die in der Pipeline
    - früh schreiben
    - spät lesen
  - oder wenn Befehle umsortiert werden (s.u.)
- Achtung: RAR (*Read After Read*) ist kein Hazard



# Datenabhängigkeiten in MIPS Pipeline

- Problem, wenn nächste Instruktion gestartet wird, bevor die vorherige beendet wurde
  - Abhängigkeiten, die “rückwärts in der Zeit” laufen (*Data-Hazards*)



# Software Lösung

---

- Compiler muss garantieren, dass keine Hazards auftreten
  - Einfügen von "nop's" (no operations, leere Instruktionen) in den Instruktionsfluss
- Wo werden die nop's eingefügt?

sub	<b>\$2</b> , \$1, \$3		sub	\$2, \$1, \$3
and	\$12, <b>\$2</b> , \$5		<b>nop</b>	
or	\$13, \$6, <b>\$2</b>	⇒	<b>nop</b>	
add	\$14, <b>\$2</b> , <b>\$2</b>		<b>nop</b>	
sw	\$15, 100 ( <b>\$2</b> )		and	\$12, \$2, \$5
			or	\$13, \$6, \$2
			add	\$14, \$2, \$2
			sw	\$15, 100 (\$2)

- **Problem: deutliche Reduktion der Performance**
  - Situation kommt einfach zu häufig vor!

# Operand Forwarding

---

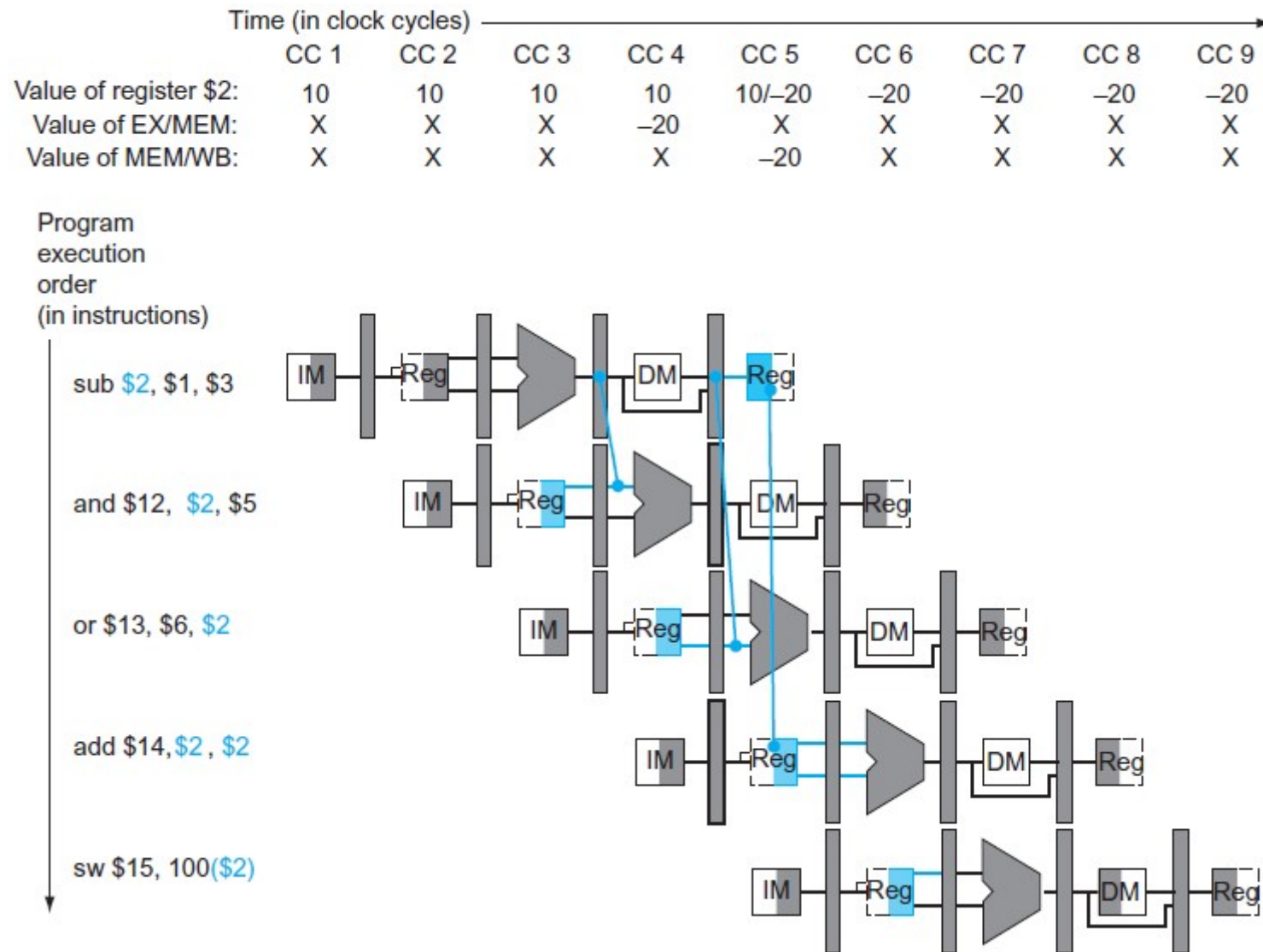
- **Operand Forwarding**
  - engl. *forwarding*: deutsch Weiterleitung
  - benutze Zwischenergebnisse schon bevor sie ins Register geschrieben werden
- **drei neue Datenquellen**
  - Lesen vom Ausgang der ALU
    - neuer Datenpfad
  - Lesen vom Ausgang des Datenspeichers
    - neuer Datenpfad
  - Lesen vom Register, das gerade erst beschrieben wird
    - statt der gespeicherten Daten werden die Daten gelesen, die gerade erst geschrieben werden
    - Änderung der Architektur des Registerfiles

# Änderung des Register Files

---

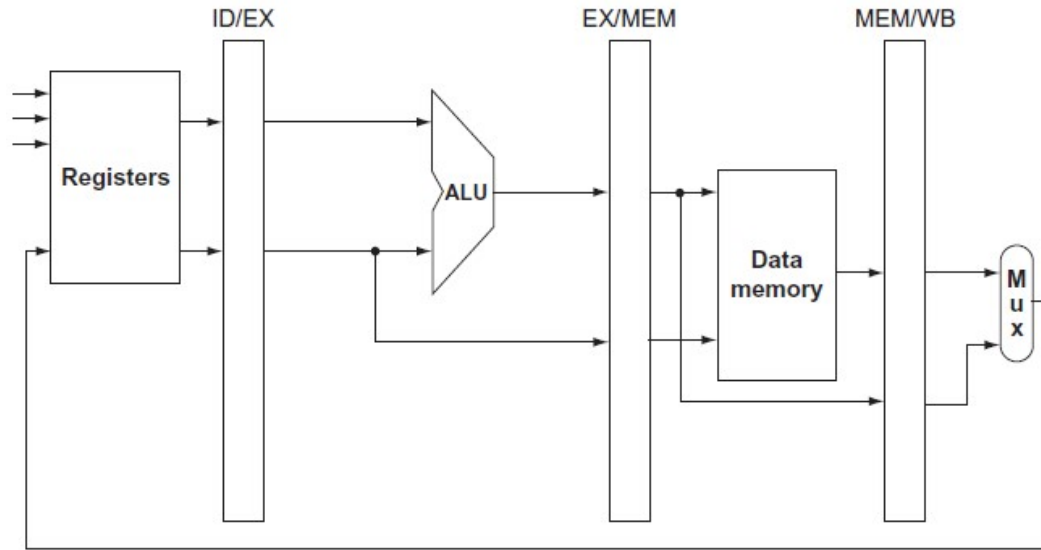
- **siehe Anhang B.8 in Patterson/Hennessy**
  - im Folgenden wird angenommen, dass ein Wert, der in einem Takt in ein Register geschrieben wird, im selben Takt bereits wieder ausgelesen werden kann
    - die zu schreibenden Werte, die **noch nicht in den D-Flipflops gespeichert** sind, erscheinen also schon am Ausgangsport des Registerfiles, falls die Schreib- und Leseadressen identisch sind
    - man kann nicht, wie sonst üblich, einen alten Wert lesen und im selben Takt einen neuen Wert in dasselbe Register schreiben
  - gegenüber der bisherigen Architektur wird zusätzliche Logik benötigt!
    - siehe Übungen

# Operand Forwarding (2)

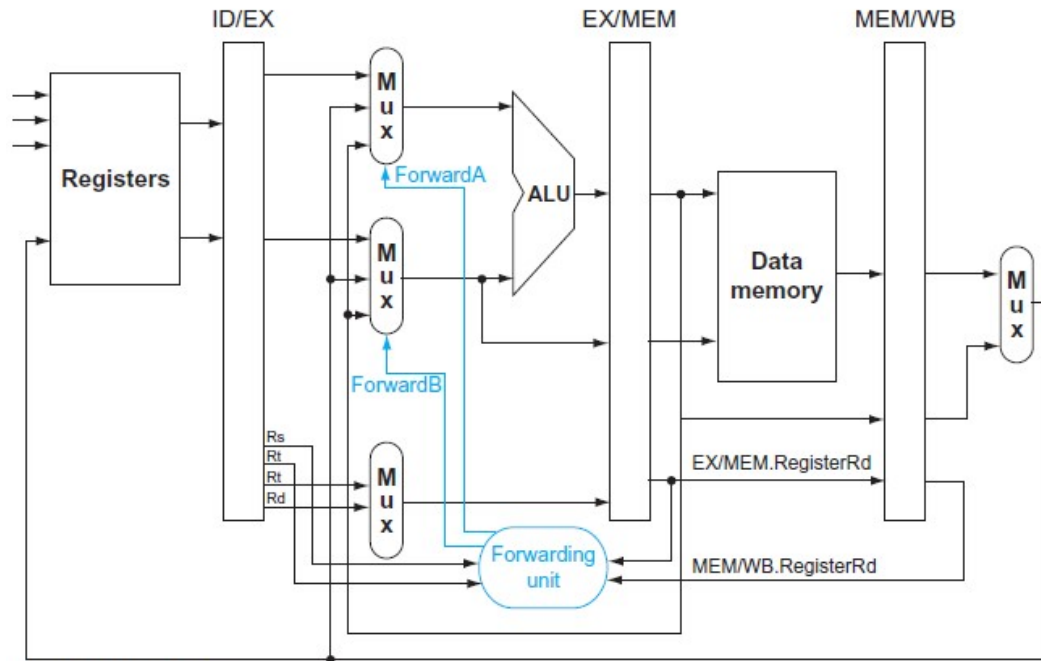


## Operand Forwarding (3)

## ohne Forwarding



a. No forwarding



b. With forwarding

mit Forwarding

# Operand Forwarding (4)

---

- Steuerung der neuen Datenpfade

<b>MUX Steuerleitungen</b>	<b>Daten- quelle</b>	<b>Erklärung</b>
<b>ForwardA = 00</b>	<b>ID/EX</b>	<b>1. Operand ALU kommt vom Register-File</b>
<b>ForwardA = 10</b>	<b>EX/MEM</b>	<b>1. Operand ALU kommt vom letzten ALU-Ergebnis</b>
<b>ForwardA = 01</b>	<b>MEM/WB</b>	<b>1. Operand ALU kommt vom Datenspeicher oder einem früheren ALU-Ergebnis</b>
<b>ForwardB = 00</b>	<b>ID/EX</b>	<b>2. Operand ALU kommt vom Register-File</b>
<b>ForwardB = 10</b>	<b>EX/MEM</b>	<b>2. Operand ALU kommt vom letzten ALU-Ergebnis</b>
<b>ForwardB = 01</b>	<b>MEM/WB</b>	<b>2. Operand ALU kommt vom Datenspeicher oder einem früheren ALU-Ergebnis</b>



# Forwarding Unit

---

- **dient der Hazard-Erkennung**
  - (vor dem Punkt: Name des Pipelineregisters)
  - grundsätzliche Bedingungen für einen Hazard

<pre>EX/MEM.RegisterRd = ID/EX.RegisterRs EX/MEM.RegisterRd = ID/EX.RegisterRt  MEM/WB.RegisterRd = ID/EX.RegisterRs MEM/WB.RegisterRd = ID/EX.RegisterRt</pre>
---

- zu beachten ist aber ferner
  - einige Instruktionen schreiben keine Register
    - Forwarding würde dann die falschen Daten liefern
    - Abhilfe: Steuersignal `RegWrite` abfragen
  - `$0` enthält immer die Null, auch wenn Instruktionen auf `$0` schreiben
    - Forwarding würde dann nicht die Null liefern
    - Abhilfe: Registeradresse mit Adresse 0 vergleichen

# Forwarding Unit (2)

---

- **EX Hazard**
  - Operand Forwarding vom letzten Ergebnis der ALU
- **Bedingungen und Steuersignale**

```
if    (EX/MEM.RegWrite
and   (EX/MEM.RegisterRd ≠ 0)
and   (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if    (EX/MEM.RegWrite
and   (EX/MEM.RegisterRd ≠ 0)
and   (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

# Forwarding Unit (3)

---

- **MEM Hazard**
  - Operand Forwarding vom Datenspeicher oder einem früheren ALU-Ergebnis
- **Bedingungen und Steuersignale**

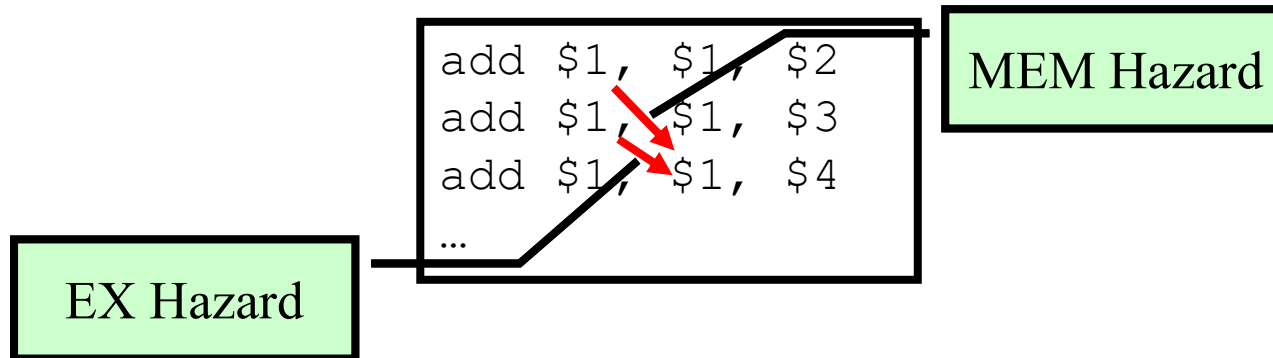
```
if    (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 0)
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if    (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 0)
and  (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

- **Was passiert wenn EX und MEM Hazard gleichzeitig auftreten?**
  - EX Hazard hat höhere Priorität, da Ergebnis neuer ist

# Forwarding Unit (4)

- **Beispiel für gleichzeitigen EX und MEM Hazard**

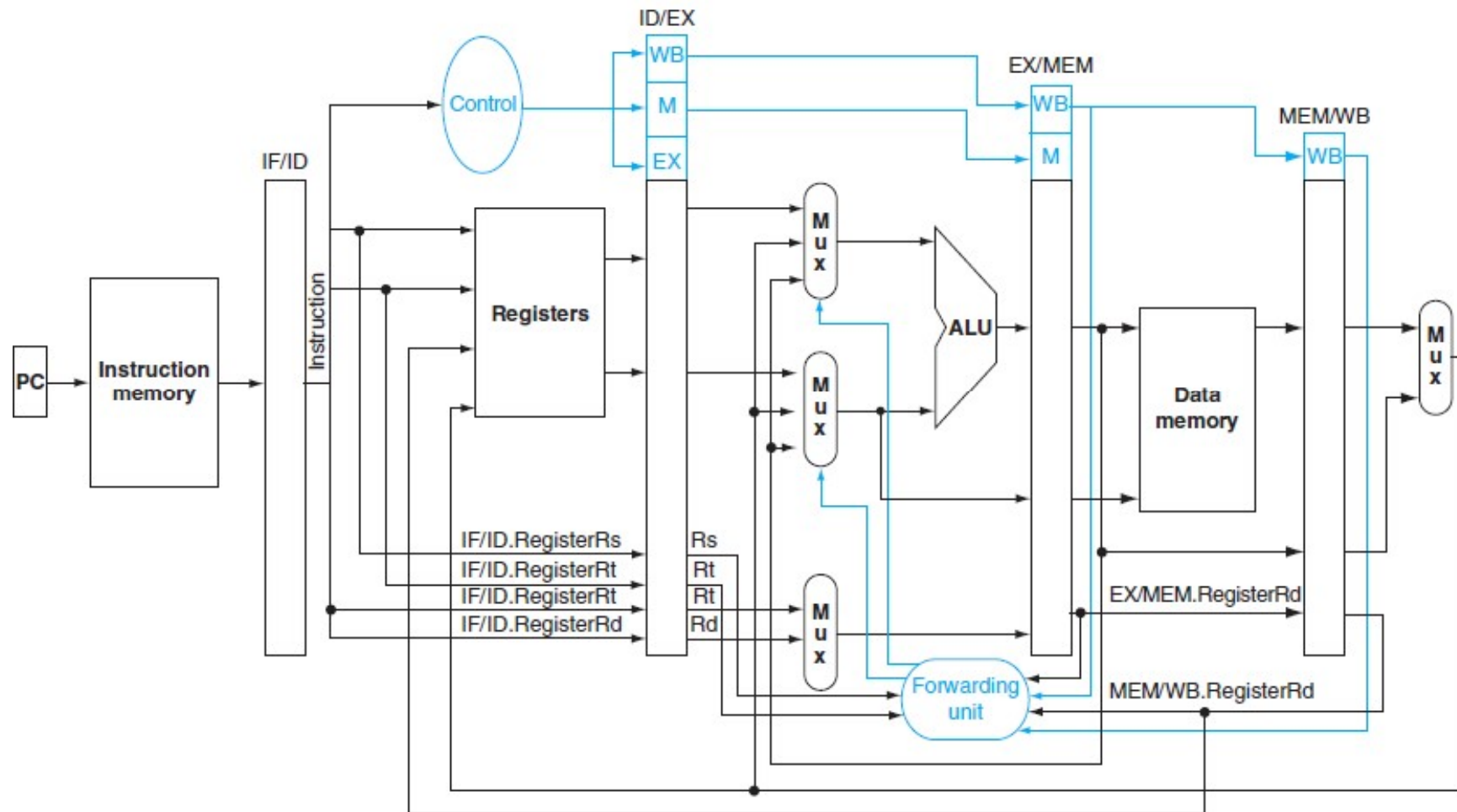


- hier werden Werte zu einem Register hinzuaddiert

```
if    (EX/MEM.RegWrite
and   (EX/MEM.RegisterRd ≠ 0)
and   (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
else
    if    (MEM/WB.RegWrite
and   (MEM/WB.RegisterRd ≠ 0)
and   (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
else ForwardA = 00
```

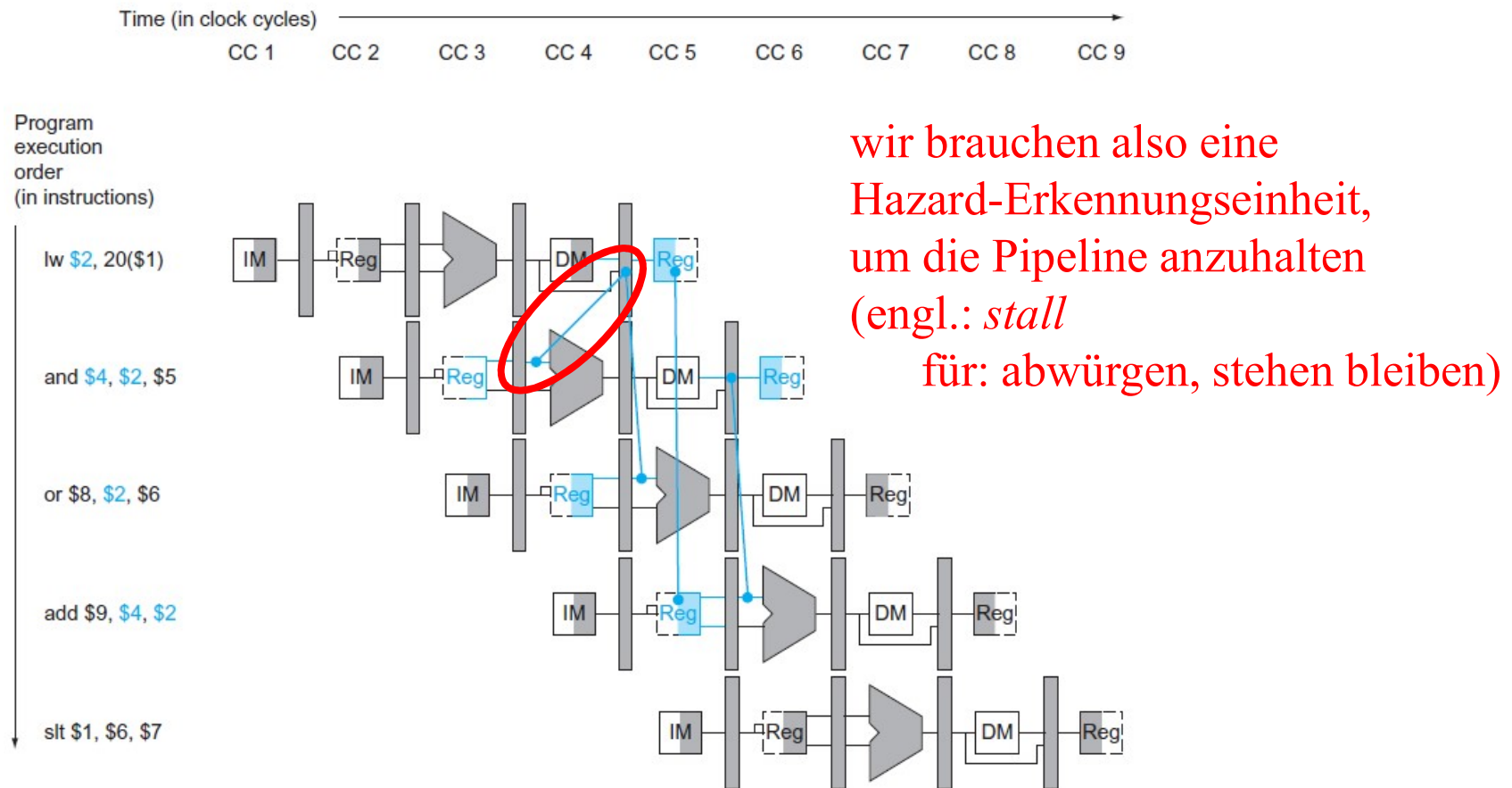
- analog für ForwardB

# Operand Forwarding mit Forwarding Unit



# Forwarding geht nicht immer

- **lw kann immer noch einen Hazard erzeugen**
  - eine Instruktion versucht ein Register zu lesen, unmittelbar nachdem die load Instruktion dasselbe Register beschreiben sollte



# Softwarelösung: Umordnen von Code

---

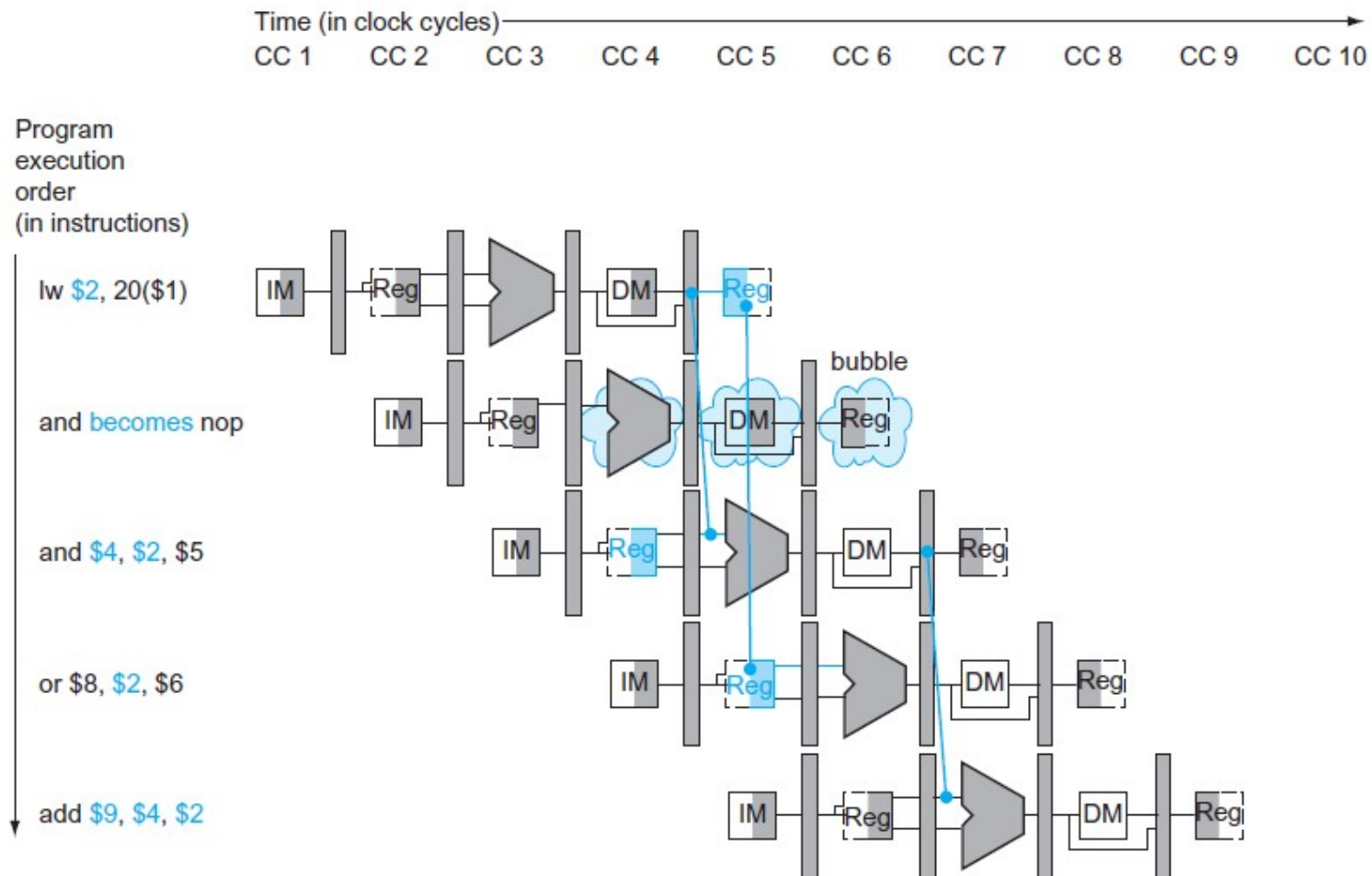
- **Beispiel swap Prozedur:**

```
                                # $t1 enthält die Adresse von v[k]
lw  $t0, 0($t1)                # $t0 = v[k] (temporärer Wert)
lw  $t2, 4($t1)                # $t2 = v[k+1]
sw  $t2, 0($t1)                # v[k] = $t2
sw  $t0, 4($t1)                # v[k+1] = $t0 (temporärer Wert)
```

- **Wo ist der Hazard?**
- **Abhilfe: Umordnen des Codes**
  - nicht immer möglich!
    - es könnten weitere Abhängigkeiten bestehen

```
                                # $t1 enthält die Adresse von v[k]
lw  $t0, 0($t1)                # $t0 = v[k] (temporärer Wert)
lw  $t2, 4($t1)                # $t2 = v[k+1]
sw  $t0, 4($t1)                # v[k+1] = $t0 (temporärer Wert)
sw  $t2, 0($t1)                # v[k] = $t2
```

\_\_\_\_\_





# Stalling (2)

---

- **Luftblase**

- der betreffende Befehl und alle folgenden werden für einen Takt angehalten
- die vorherigen Befehle laufen in der Pipeline aber weiter
  - müssen sie auch, denn wir warten ja auf deren Ergebnis
- dadurch reißt die Pipeline auf, es entsteht so etwas wie eine Luftblase, eine eingefügte **nop** Instruktion

- **Realisierung**

- verhindern, dass sich PC und IF/ID Pipeline-Register verändern
- alle Steuersignale im ID/EX Register so setzen, dass eine **nop** Instruktion simuliert wird (in unserem Fall: alles auf 0)
- Daten und Steuersignale werden damit effektiv für einen Takt angehalten, als ob sich eine Luftblase (*bubble*) in einer Wasserleitung befindet, in der sich keine relevanten Daten (Wasser) befinden

# Stalling (3)

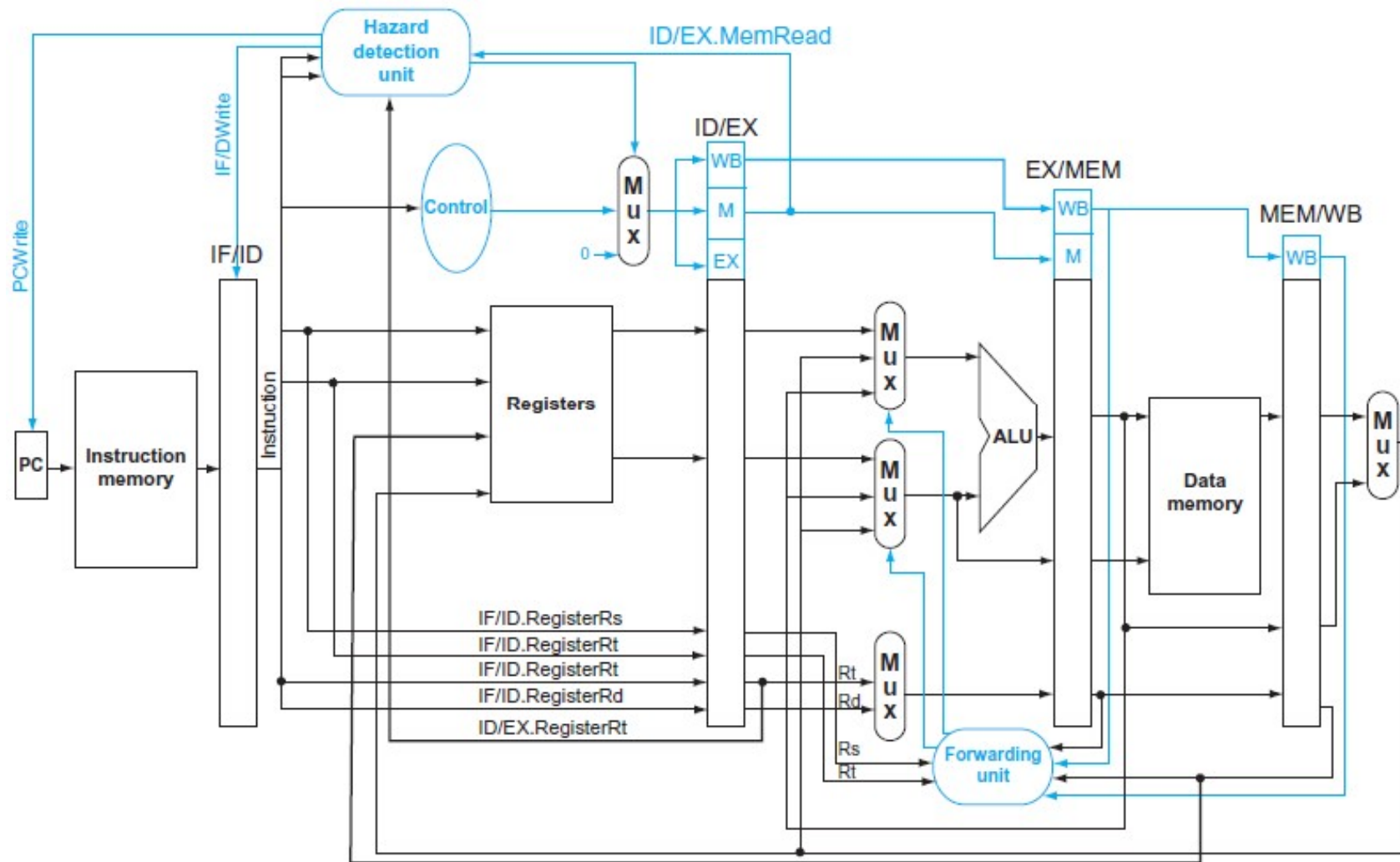
---

- **Hazard Detection Unit**

- Bedingung für Hazard (in der ID-Stufe zu testen)
  - Datenspeicher wird gelesen
  - Daten sollen in dasselbe Register geschrieben werden, aus dem einer der Operanden des aktuellen Befehls stammt

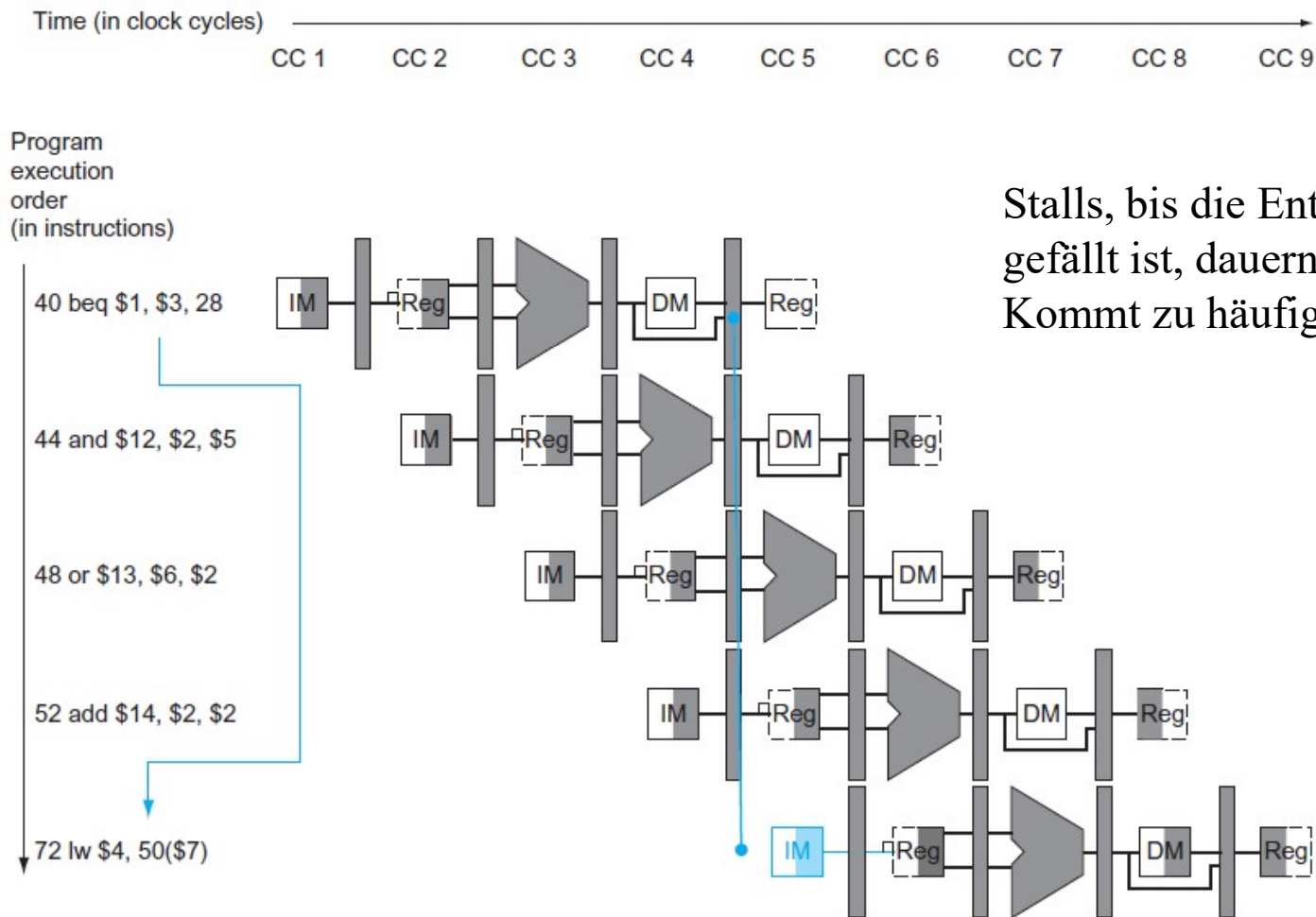
```
if    (ID/EX.MemRead  
    and ((ID/EX.RegisterRt = IF/ID.RegisterRs)  
        or (ID/EX.RegisterRt = IF/ID.RegisterRt))) stall
```

# Hazard Detection Unit mit Stalling



# Branch Hazards

- Wenn die Entscheidung gefällt wird, dass gesprungen wird, befinden sich bereits andere Instruktionen in der Pipeline!

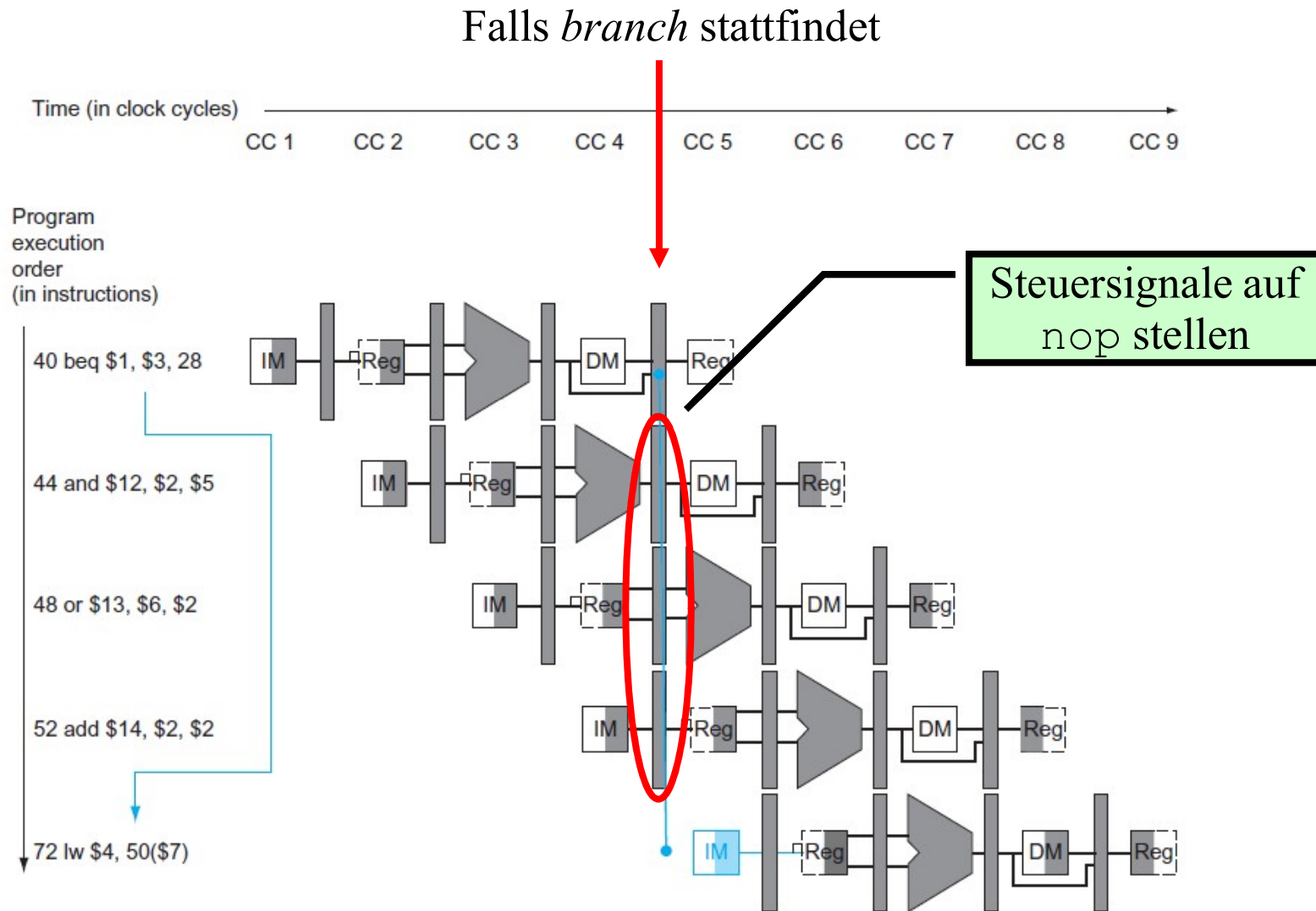


# Branch Hazards (2)

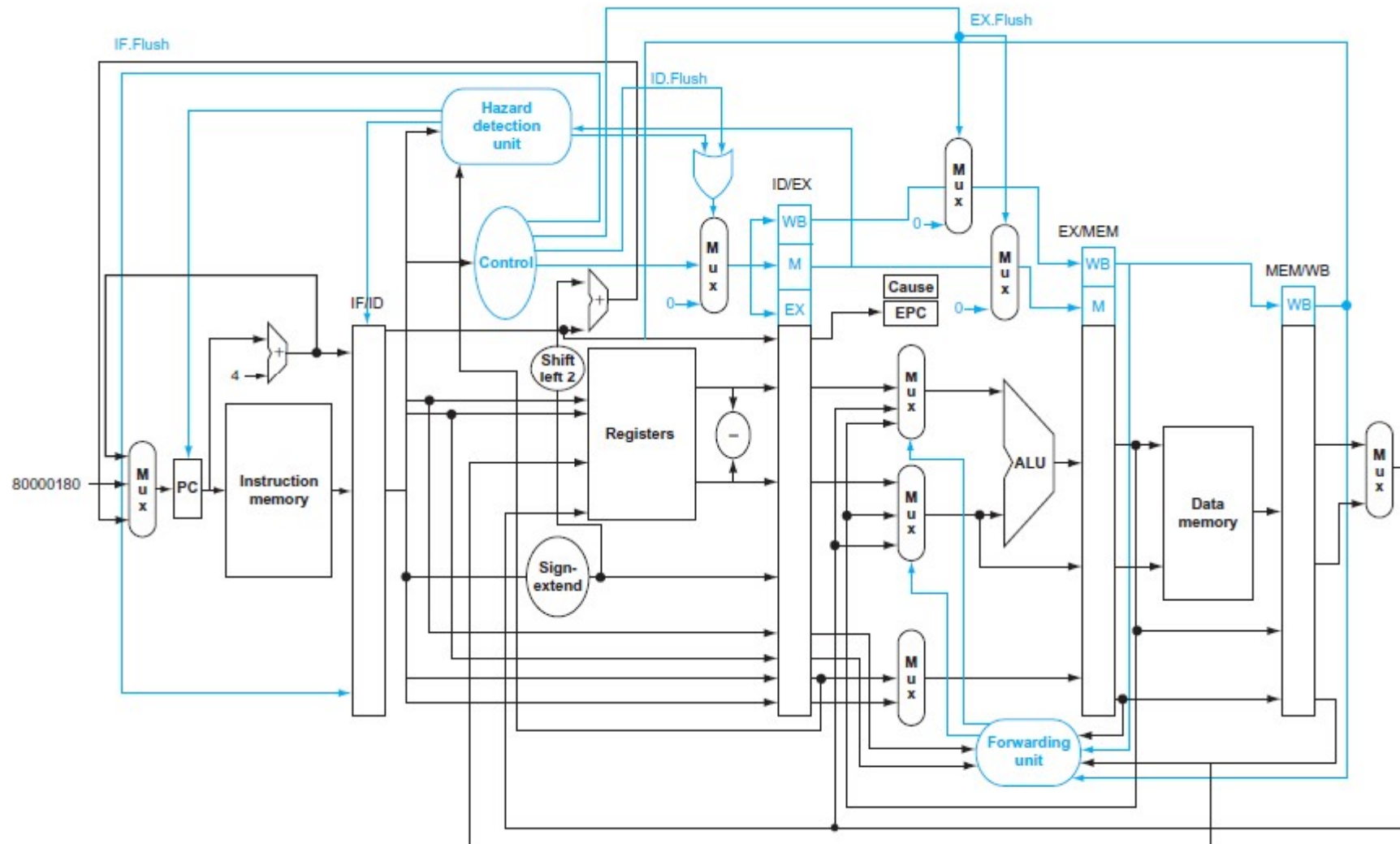
---

- Wir tun also zunächst so, als würde kein Sprung stattfinden.
  - entspricht einer Vorhersage, dass die Verzweigung "nicht ausgeführt wird"
- Wir benötigen Hardware, um die Instruktionen aus der Pipeline zu entfernen (engl. *flush the pipeline*, Pipeline ausspülen), falls die Vorhersage nicht eintrifft.
- **Realisierung von Flushing**
  - ähnlich zu *Stalling*
  - aber die Steuersignale von drei Befehlen müssen auf `nop` gesetzt werden
    - Steuersignale in den drei Pipeline Registern IF/ID, ID/EX, EX/MEM
    - Weitere Multiplexer notwendig (nicht in den Schaltplänen gezeigt)

# Realisierung von Flushing



# Realisierung von Flushing (2)



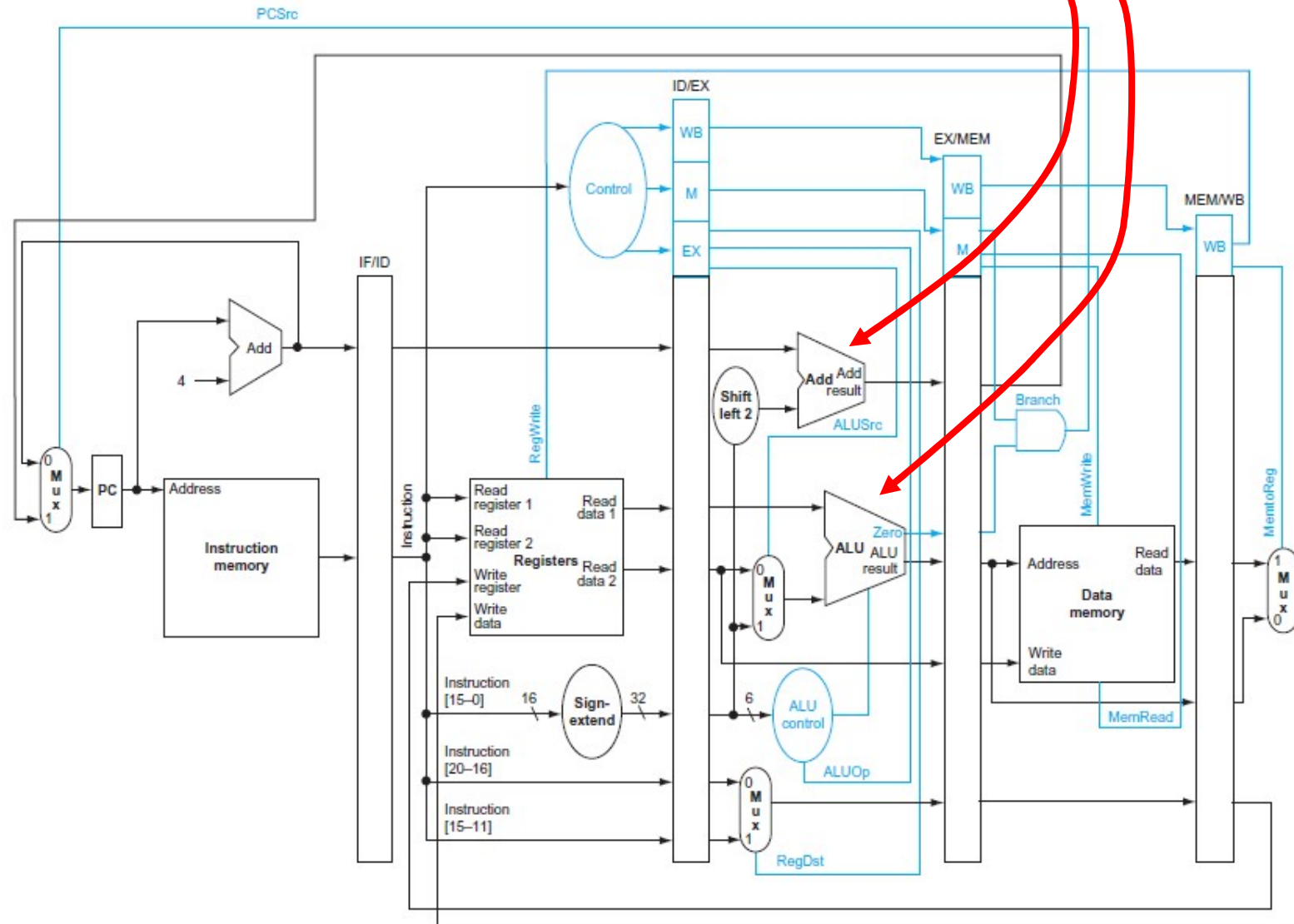
# Erhöhung der Performance

---

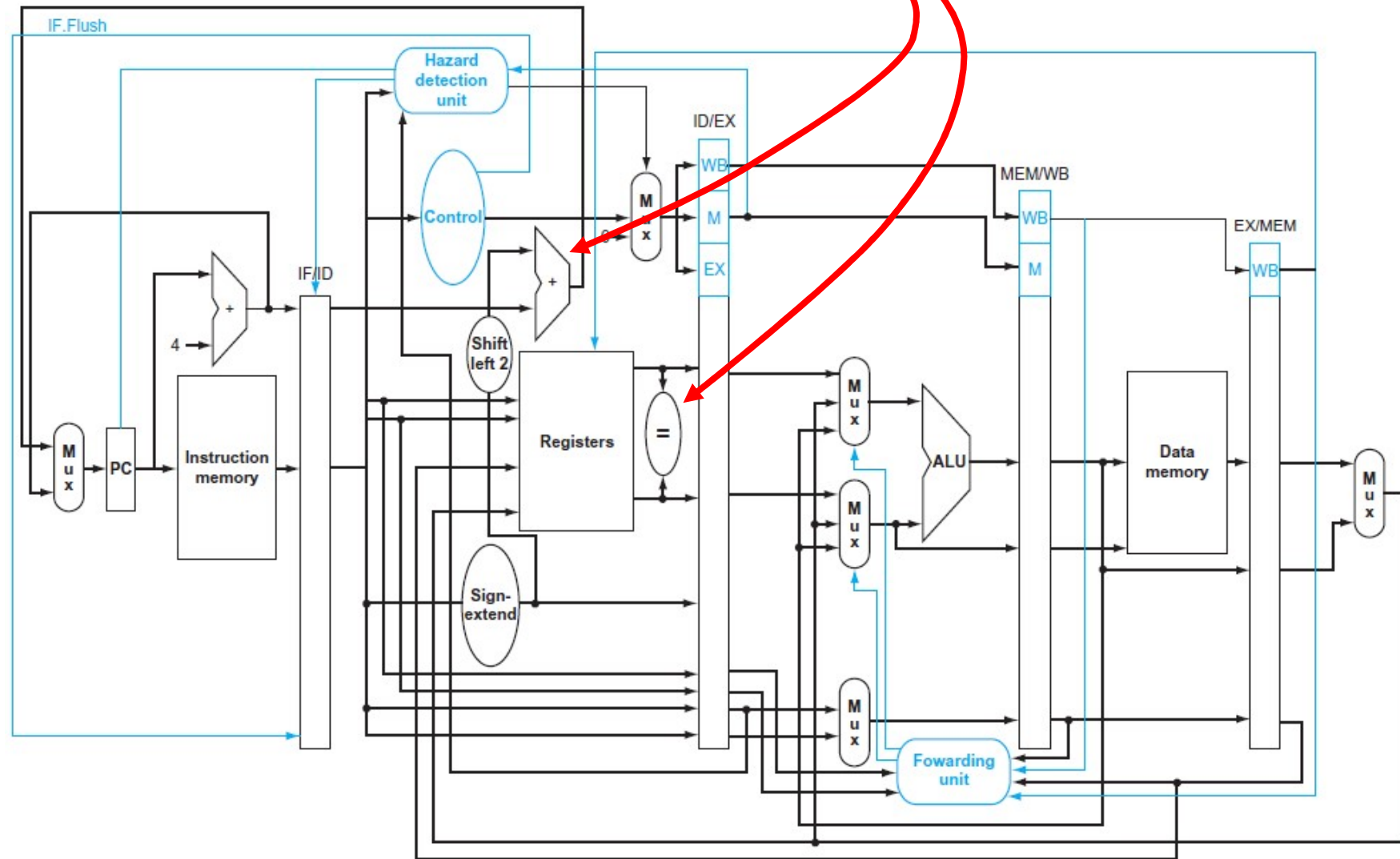
- **Sprungentscheidung möglichst früh fällen**
  - dadurch werden nicht unnötig viele Instruktionen in die Pipeline gesteckt
  - geht nur, wenn auch die Adressberechnung entsprechend früh durchgeführt wird



# Vorher: späte Sprungentscheidung



# Frühe Sprungentscheidung



# Frühe Sprungentscheidung (2)

---

- Vergleichsoperation nicht erst in der ALU sondern mit XOR-Gattern (sehr schnell) bereits in der ID Stufe
- 2 Takte gespart
  - Sprungentscheidung und Adressberechnung einen Takt früher
  - Daten gehen vom Addierer direkt in den PC, nicht über ein Pipelineregister
- Nur ein Befehl muss entfernt werden.
- *Forwarding* und *Hazard detection unit* müssen entsprechend erweitert werden, damit Sprünge, die von Resultaten früherer Operationen abhängen (Datenabhängigkeiten), immer noch korrekt funktionieren
  - Details werden hier nicht weiter diskutiert

# Frühe Sprungentscheidung (3)

---

- **Problem**

- Die Sprungentscheidung kann nicht immer in der ID Phase gefällt werden.
  - Test auf Gleichheit mit XOR-Gattern ist zwar relativ schnell
  - verlängert aber auch die Bearbeitungszeit in ID und damit evtl. die Taktperiode
  - Andere bedingte Sprünge erfordern komplexere Berechnungen in der EX Phase (z.B. BLT, daher wird darauf verzichtet, siehe SLT).

- **Ausweg**

- Möglichst geschickt (s.u.) raten, ob gesprungen wird.
  - Sprungvorhersage
- Falsch raten ist nicht fatal.
  - Dann muss die Pipeline eben geleert werden.
  - Kostet dann ein paar Takte mehr Rechenzeit.

# Dynamische Sprungvorhersage

---

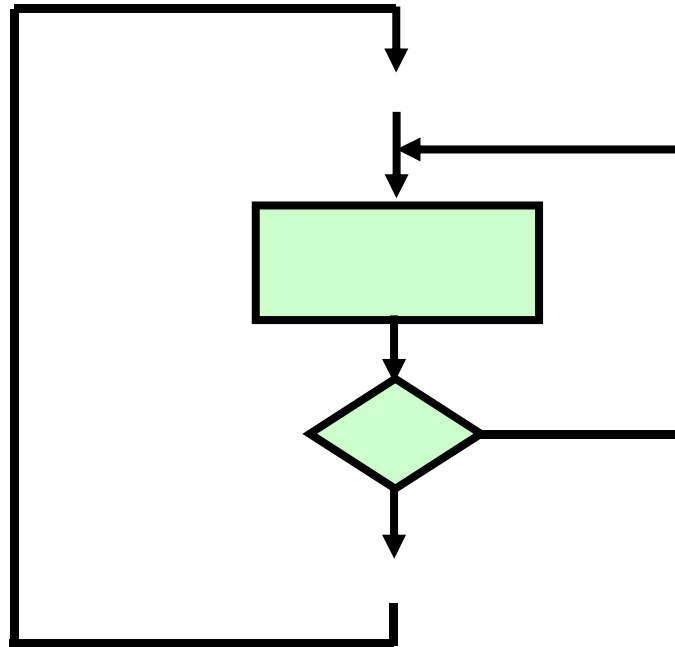
- **Vorhersage davon abhängig machen, ob beim letzten Mal gesprungen wurde oder nicht.**
  - Kleiner Speicher, der mit den untersten Bits der Adresse des Sprungbefehls adressiert wird (wie Cache, s.u.).
    - Ist nicht eindeutig, aber mit hoher Wahrscheinlichkeit wird die Speicherstelle nicht von einem anderen Sprung überschrieben.
  - Ein Bit wird gespeichert, das sagt, ob beim letzten Mal bei dieser Instruktion gesprungen wurde oder nicht.
  - Bit gibt nur Hinweis, muss nicht stimmen
    - Stellt sich später heraus, dass Vorhersage falsch war, wird Pipeline *geflusht*.
    - Es macht also nichts, wenn das Bit in Wahrheit von einem anderen Sprungbefehl stammt, dessen Adresse zufällig dieselben unteren Bits hat.
    - Kostet dann bei falscher Vorhersage ein paar Takte mehr Rechenzeit.
    - Die Wahrscheinlichkeit, dass die folgenden Befehle von der richtigen Stelle stammen wird also erhöht.

# Dynamische Sprungvorhersage (2)

---

- **Beispiel Schleife**

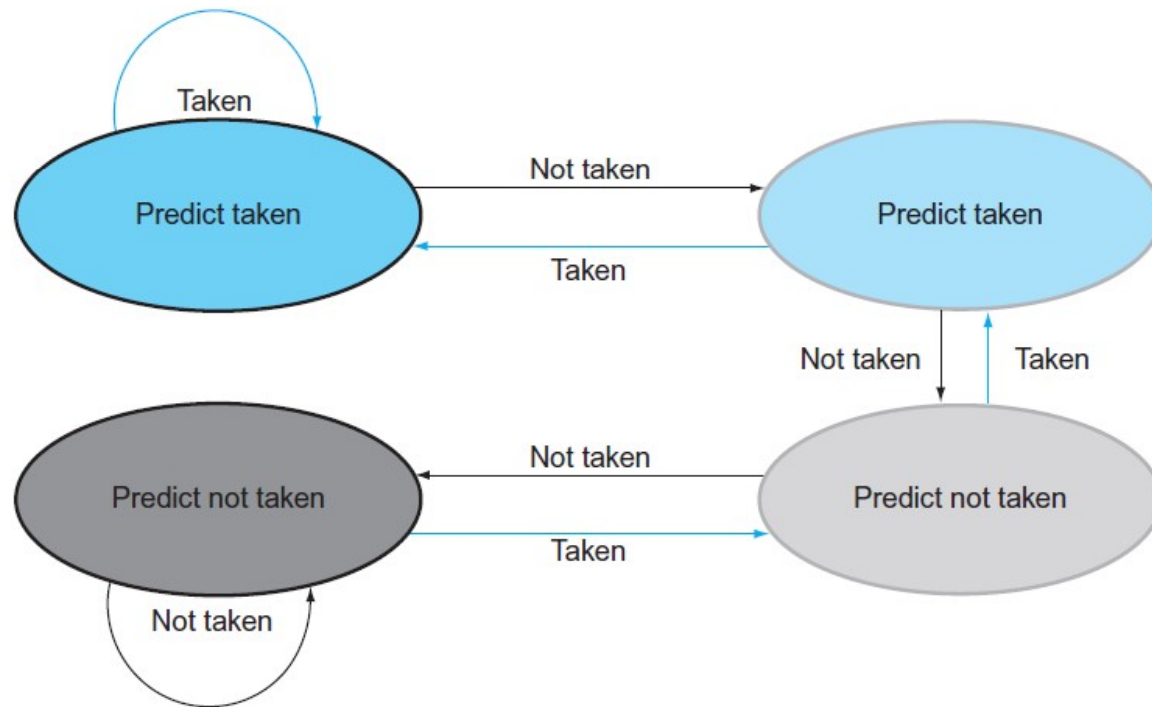
- Wird normalerweise mehrfach durchlaufen.
- Schema mit nur einem Bit sagt in der Regel *zweimal* falsch voraus!
  - Falschvorhersage beim Verlassen der Schleife ist nicht zu verhindern.
  - Wird die Schleife das nächste Mal ausgeführt, wird beim ersten Sprung auch falsch vorhergesagt, da beim letzten Mal nicht gesprungen wurde.



# Dynamische Sprungvorhersage (3)

- **Abhilfe**

- Vorhersage muss zweimal falsch sein, bevor die Vorhersage gewechselt wird
- dann wird nur beim Verlassen der Schleife falsch vorhergesagt



2 Bits speichern, um  
die vier Zustände zu  
kodieren

# Realisierung mit Branch-Prediction Buffer

---

- **In der ID (*Instruction Decode*) Phase**
  - wird die Sprungadresse berechnet
  - steht die normale sequentielle Adresse fest (wird in IF Phase berechnet)
  - wird die Sprungvorhersage gemacht
- **Damit kann der Sprung am Ende der ID Phase ausgeführt werden.**
  - Findet der Sprung tatsächlich statt und wird korrekt vorausgesagt, geht trotzdem ein Takt verloren.
  - Bei falscher Vorhersage entsprechend mehr Takte.
- **Der Speicher, mit dem die Sprungvorhersage gemacht wird, wird auch *branch-prediction buffer* genannt.**



# Branch-Target Buffer

---

- **Optimal wäre**

- wenn am Ende der IF (Instruction Fetch) Phase die Adresse für die nächste Instruktion feststehen würde
- Dazu müsste bekannt sein
  - ob es sich um eine Verzweigung handelt (die Instruktion ist noch gar nicht dekodiert)
  - ob gesprungen wird (Sprungentscheidung ist noch nicht gefällt worden)
  - wohin gesprungen wird (Adressberechnung hat noch nicht stattgefunden)

- **Idee**

- Benutze einen Speicher, der neben der Sprungvorhersage noch die Adresse für den nächsten Befehl nach einer Programmverzweigung speichert (Sprungadresse).
  - *branch-target buffer* oder *branch-target cache*

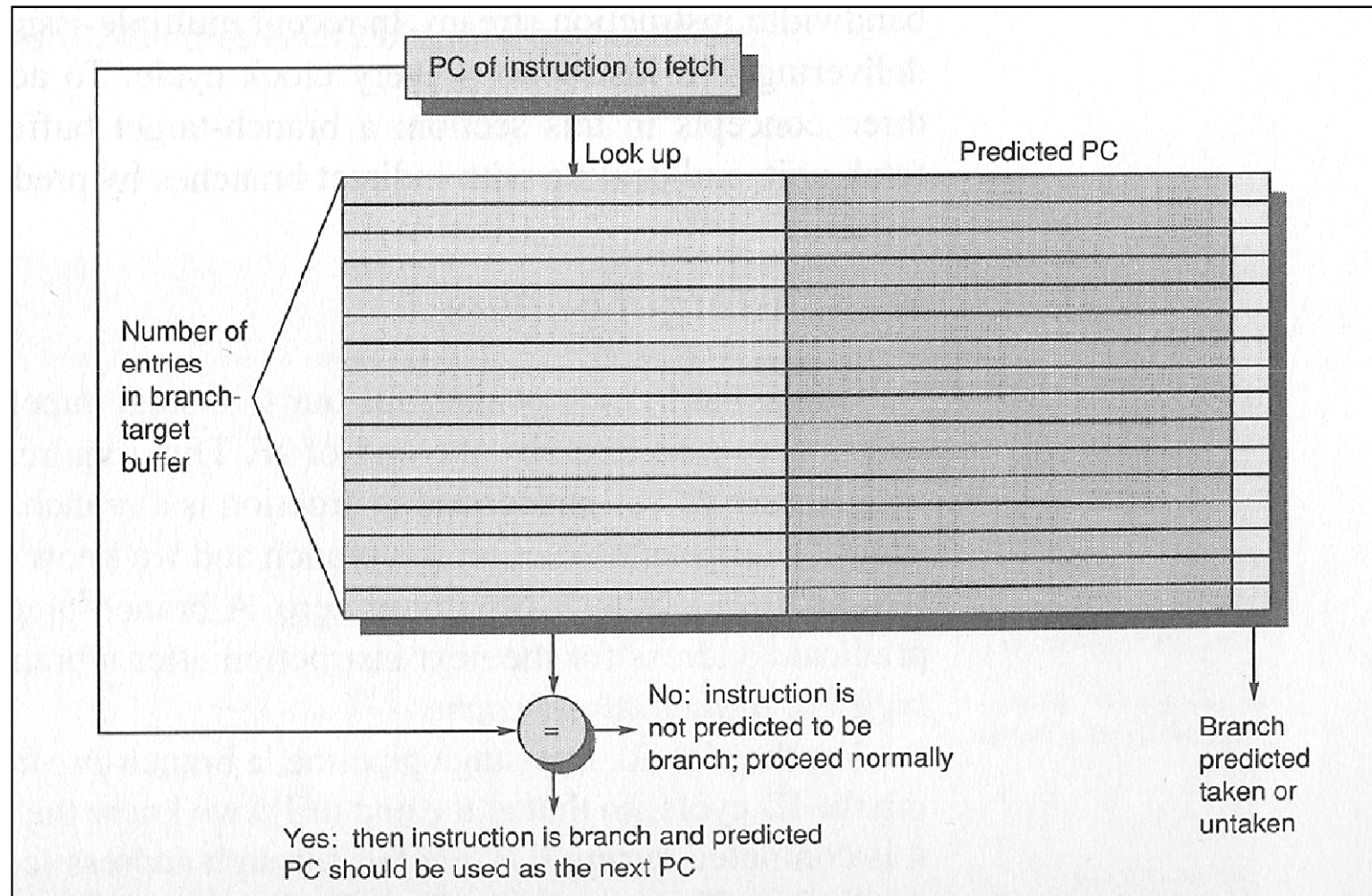
# Branch-Target Buffer (2)

---

- Falls im branch-target buffer eine Sprungadresse gespeichert ist, handelt es sich bei dem Befehl um einen Sprung und es wird vorhergesagt, dass dorthin verzweigt wird.
- Falls die Vorhersage richtig ist, wird **kein einziger Takt** verschenkt.
- **Wichtig**
  - Damit nicht auch für andere (Nicht-Sprung-)Befehle ständig Sprünge vorhergesagt werden, muss beim Abfragen sichergestellt werden, dass die gespeicherte Information auch wirklich von der aktuellen Speicherstelle kommt.
  - Daher verlässt man sich nicht nur auf die untersten Adressbits, sondern speichert die zugehörigen restlichen Adressbits im Branch-Target-Buffer (*tag bits* genau wie bei einem Cache).
  - Stimmen die nicht überein, wird kein Sprung vorausgesagt.
    - Funktionsweise genau wie bei einem Cache (s.u.)

# Branch-Target Buffer (3)

- Hardware ist identisch mit der von einem Cache (s.u.)



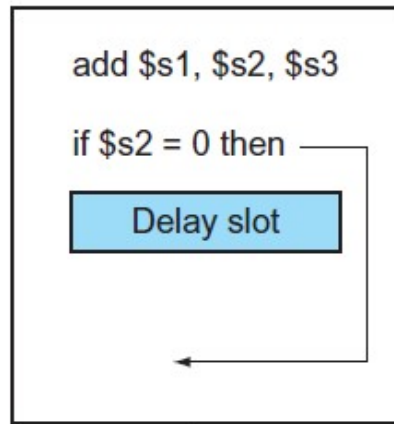
# Compiler Unterstützung

---

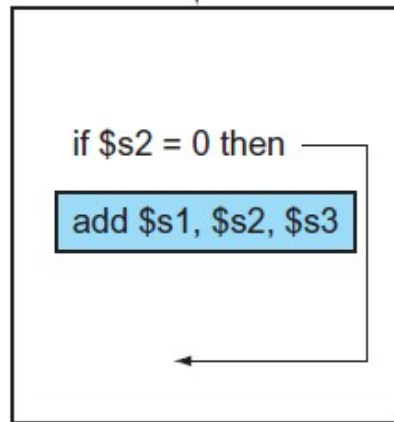
- **Umdefinition der Verzweigung**
  - füge einen “*branch delay slot*” hinzu
  - die nächste Instruktion nach der Verzweigung wird dann *immer* ausgeführt
    - Die Verzweigung findet also erst verzögert nach der nächsten Instruktion tatsächlich statt.
    - Das ist eine Änderung der Semantik der *branch* Instruktion!
  - Compiler muss den *Slot* (Schlitz) mit etwas Nützlichem füllen
    - gibt es keine nützliche Instruktion **muss** eine `nop` Instruktion eingefügt werden

# Delayed Branch

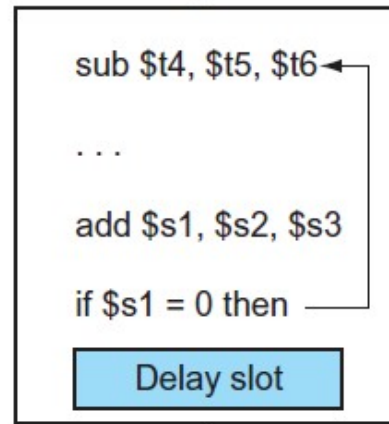
a. From before



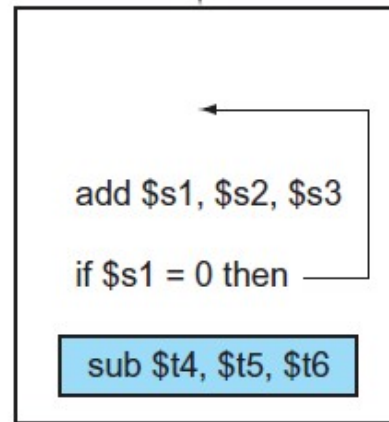
Becomes



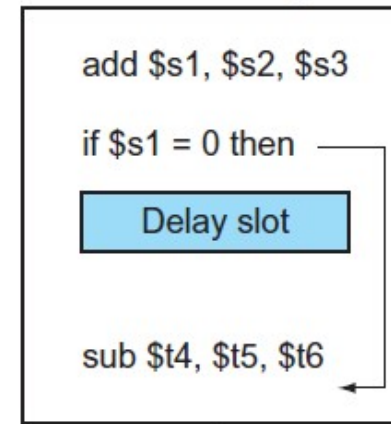
b. From target



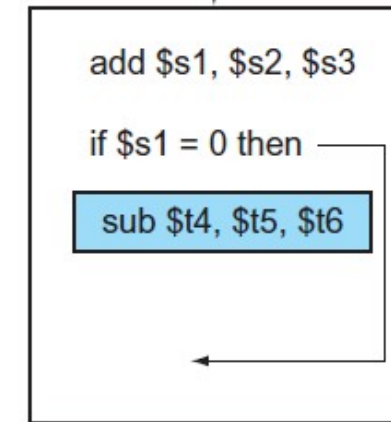
Becomes



c. From fall-through



Becomes



# Exceptions

---

- **Mögliche Ursachen**

- Overflow, Underflow, Division durch 0
- I/O-Anforderung (Interrupt)
- Aufrufen eines OS-Dienstes (SW-Interrupt)
- undefinierte Instruktion
- HW-Fehlfunktion

- **Mehrstufige Pipelines**

- bei uns: 5 Instruktionen befinden sich gleichzeitig in der Pipeline
- Probleme
  - Zuordnung der Exception zu einem Befehl
  - mehrere Exceptions können gleichzeitig auftreten
    - Exceptions müssen priorisiert werden
    - HW sortiert Exceptions, so dass die früheste Instruktion unterbrochen wird, auch wenn deren Exception erst nach einer anderen auftritt

# Exceptions (2)

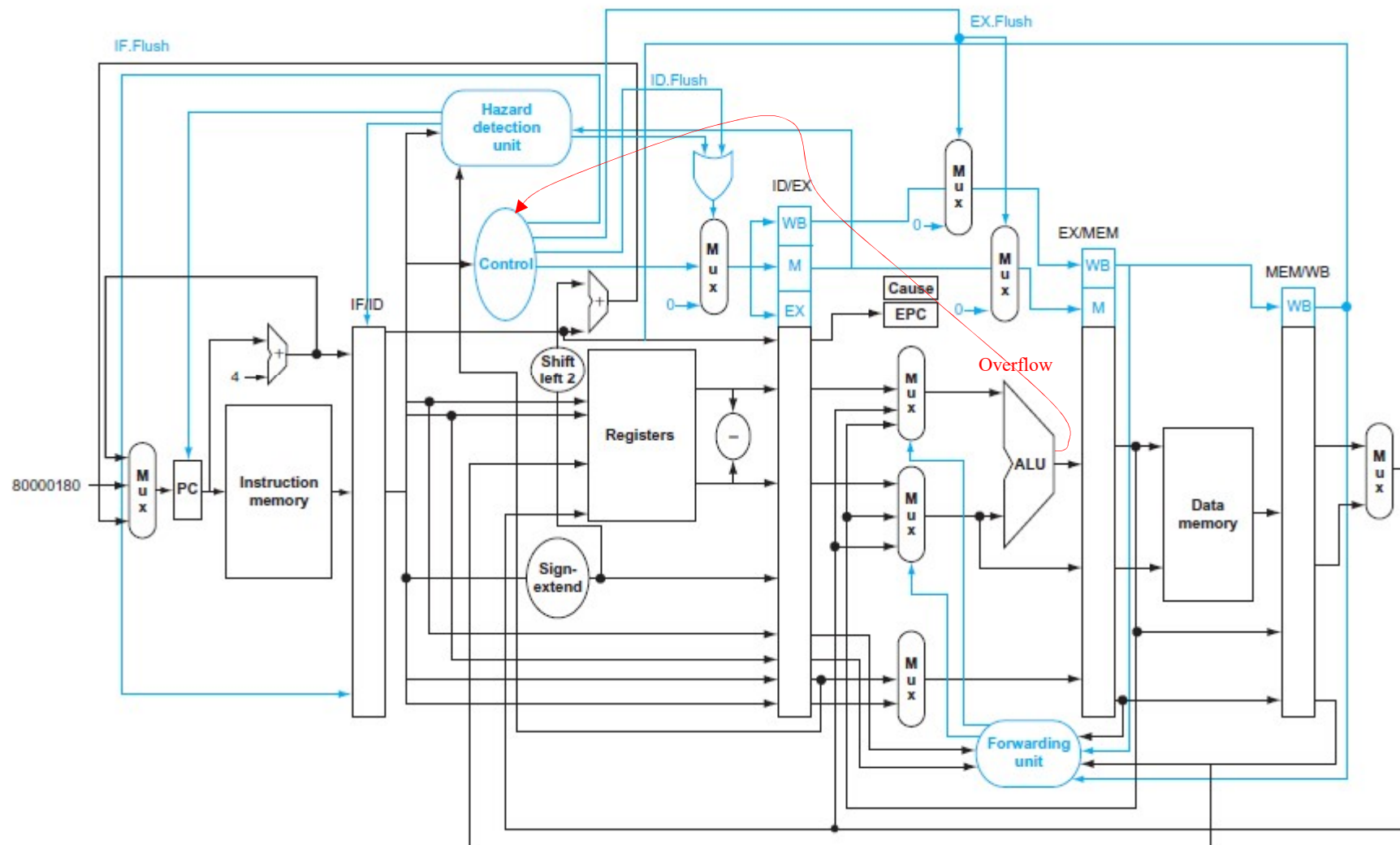
---

- **Beispiel**

- Overflow in `add $1, $2, $1`
  - Speichern der Befehlsadresse + 4 im EPC
    - Subtrahierer soll hier eingespart werden
    - Betriebssystem muss dann die 4 per Software subtrahieren
  - Speichern der Ursache in `Cause`
  - Abbruch der verursachenden Instruktion
    - sonst wird \$1 verändert und die Ursache des Overflows ist nicht mehr rekonstruierbar (im Allgemeinen!)
  - Instruktionen vor der verursachenden Instruktion werden noch zu Ende geführt
  - Flushen der Pipeline
    - nachfolgende Befehle dürfen nicht ausgeführt werden
    - Betriebssystem soll zunächst die Kontrolle bekommen
  - Verzweigen ins Betriebssystem
    - Sprung zu vordefinierter Adresse



# Exceptions in der Pipeline

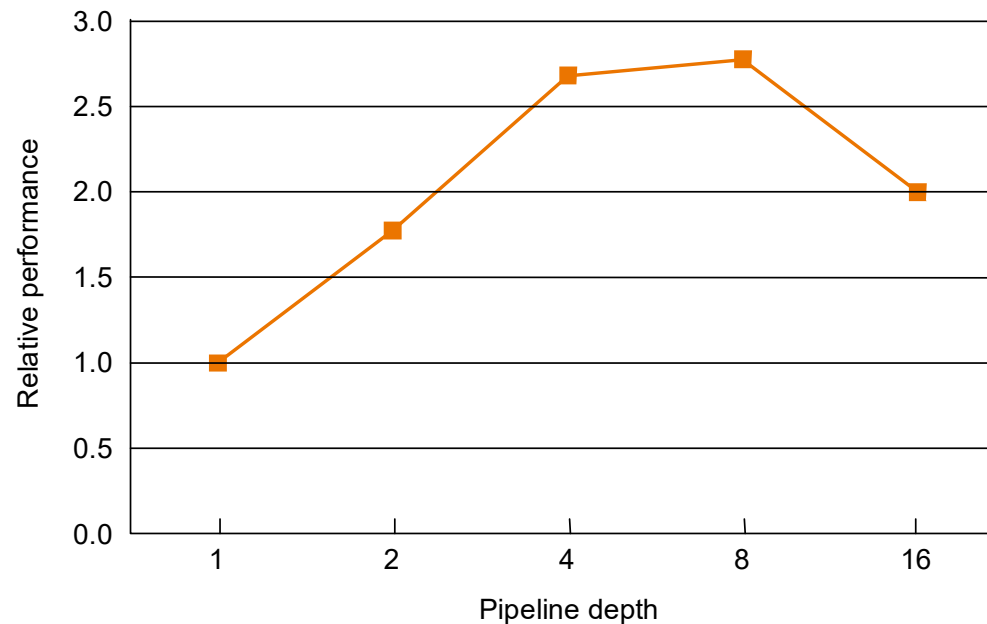




# Superpipelining

---

- Noch längere Pipelines
  - Idee: max. Speedup gleich Anzahl Stufen
    - Manchmal hat man 8 und mehr Stufen allein im Execute-Teil.
  - Nachteile
    - Overhead durch neue Register
    - Ausbalancieren wird immer schwieriger.
    - Wahrscheinlichkeit und Auswirkungen von Hazards nehmen zu.



# Weitere Erhöhung der ILP

---

- Pipeline bearbeitet pro Takt maximal einen Befehl
- Wie kann man den Parallelisierungsgrad für Instruktionen weiter erhöhen?
  - wir müssen mehr als einen Befehl pro Takt bearbeiten
- **zwei wesentliche Spielarten**
  - Superskalare Prozessoren
    - mehrere Pipelines
    - variable Anzahl von Instruktionen pro Takt
    - statisches (Compiler) oder dynamisches (Hardware) Scheduling
  - VLIW
    - *Very Long Instruction Word*
    - feste Anzahl von Instruktionen formatiert als ein sehr langes Instruktionsword
    - Parallelisierung wird durch den Compiler explizit gemacht
      - auch EPIC (*Explicit Parallel Instruction Computer*) genannt

# Superskalare CPU's

---

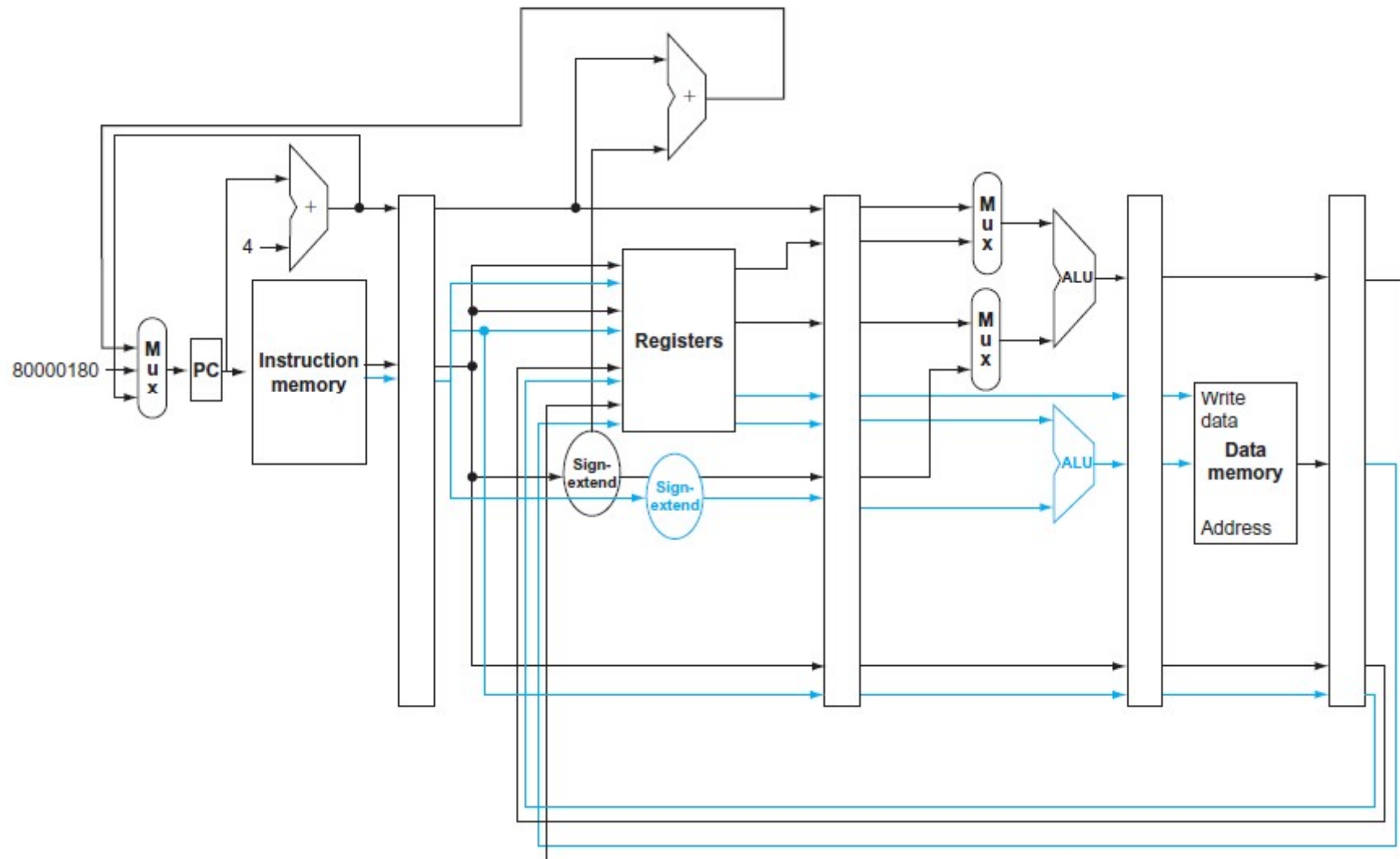
- Vervielfachung der Komponenten, so dass mehr als ein Befehl pro Takt abgearbeitet werden kann
- CPI: *Cycles Per Instruction*
  - kann dann kleiner als 1 werden
  - dann nimmt man besser den Kehrwert IPC: *Instructions Per Cycle*
- Mehraufwand, unabhängige Befehle zu finden, die parallel ausgeführt werden können
  - statisch durch Compiler
  - dynamisch durch die Hardware

# Superskalärer MIPS als Beispiel

---

- **Zwei Befehle pro Takt**
  - Die komplette Pipeline muss nicht verdoppelt werden, wenn man in jeder Pipeline nur bestimmte Befehlstypen unterstützt, z.B.
    1. Pipeline: ALU-Instruktionen und Sprünge
    2. Pipeline: load und store
- **64 bit müssen pro Takt gelesen und dekodiert werden**
- **Vereinfachende Annahme**
  - Befehle seien richtig gepaart (ALU bzw. Sprung kommt als erstes) und liegen auf 64 bit Grenzen (*alignment*).
  - Sonst müsste man die Befehle untersuchen und bei Bedarf austauschen, bevor man sie in die Pipeline schickt.
  - Würde Konflikterkennung (*hazard detection*) erschweren.
  - Den Aufwand würde man in einem echten MIPS aber treiben.

# Superskalärer MIPS (2)



# Superskalärer MIPS (3)

---

- **Hardware arbeitet dynamisch**
  - d.h. sie führt *nur einen* Befehl aus, falls Bedingungen nicht erfüllt sind
  - die andere Hälfte bekommt ein *stall*
- **Hardwareerweiterungen**
  - zusätzliche Ports für Registersatz
  - zweite ALU für Berechnung der effektiven Adresse bei *load* und *store*
  - sonst würde man strukturelle Konflikte in die Pipeline einbauen

# Superskalarer MIPS (4)

---

- **Problem**
  - Nach einem *lw* darf der nächste Befehl nicht auf den gelesenen Wert zugreifen, um einen *stall* der Pipeline für einen Takt zu verhindern.
  - Bei unserer superskalaren Architektur können sogar die zwei nächsten Befehle nicht auf den Wert zugreifen.
- **Compiler müssen Code für die Pipelines noch besser optimieren.**

# Superskalärer MIPS (5)

- **Beispiel**

```
Loop: lw    $t0, 0($s1)      # $t0=array element
      add   $t0,$t0,$s2      # add scalar in $s2
      sw    $t0, 0($s1)      # store result
      addi  $s1,$s1,-4       # decrement pointer
      bne   $s1,$zero,Loop   # branch $s1!=0
```

- umordnen, um möglichst viele *stalls* zu vermeiden

	ALU oder branch	Datentransfer
Loop:		lw \$t0, <b>0</b> (\$s1)
	addi \$s1,\$s1, <b>-4</b>	
	add \$t0,\$t0,\$s2	
	bne \$s1,\$zero,Loop	sw \$t0, <b>4</b> (\$s1)

- 4 Taktzyklen für 5 Befehle: 0,8 CPI bzw. 1.25 IPC
  - Hardware erlaubt theoretisch 0,5 CPI bzw. 2 IPC



# Superskalärer MIPS (6)

---

- **Schleifen entrollen (*loop unrolling*)**
  - Technik, Schleifen zu beschleunigen
  - Vereinfachende Annahme
    - Schleifenindex ist Vielfaches von 4
      - fasse 4 Schleifendurchläufe zu einem zusammen

	ALU oder branch	Datentransfer
Loop:	<code>addi \$s1, \$s1, -16</code>	<code>lw \$t0, 0(\$s1)</code>
		<code>lw \$t1, 12(\$s1)</code>
	<code>add \$t0, \$t0, \$s2</code>	<code>lw \$t2, 8(\$s1)</code>
	<code>add \$t1, \$t1, \$s2</code>	<code>lw \$t3, 4(\$s1)</code>
	<code>add \$t2, \$t2, \$s2</code>	<code>sw \$t0, 16(\$s1)</code>
	<code>add \$t3, \$t3, \$s2</code>	<code>sw \$t1, 12(\$s1)</code>
		<code>sw \$t2, 8(\$s1)</code>
	<code>bne \$s1, \$zero, Loop</code>	<code>sw \$t3, 4(\$s1)</code>

# Superskalarer MIPS (7)

---

- 8 Taktzyklen für 14 Befehle: 0,57 CPI bzw. 1.75 IPC
- 8 Taktzyklen für effektiv 4 Schleifendurchläufe
- entspricht 2 Taktzyklen für 1 Schleifendurchlauf
  - Faktor 2 schneller
    - durch Einsparung von Befehlen (Entrollen der Schleife)
    - durch bessere Ausnutzung der superskalaren Architektur

# Dynamic Pipeline Scheduling

---

- bei einem Pipelinekonflikt müssen nachfolgende Befehle warten, obwohl sie ausgeführt werden könnten
- Beispiel:

```
lw    $t0, 20($s2)
add   $t1, $t0, $t2
sub   $s4, $s4, $s3
addi  $t5, $s4, 100
```

- add nach lw bewirkt *pipeline stall*
- sub und addi könnten aber bereits ausgeführt werden, da alle Operanden zur Verfügung stehen

# Dynamic Pipeline Scheduling (2)

---

- **Idee**

- Instruktionen werden weiterhin in Programmierreihenfolge der EX Stufe zugeführt (*in-order issue*) (Engl. *issue*: Ausgabe, Heft, Emission)
- Instruktionen, deren Operanden noch nicht zur Verfügung stehen, müssen warten
- die Ausführung kann sofort beginnen, wenn alle Operanden zur Verfügung stehen

- **d.h.**

- Beginn der Ausführung der Instruktionen ist *out-of-order*
- daraus folgt aber auch: *out-of-order* Fertigstellung der Instruktionen

# Dynamic Pipeline Scheduling (3)

---

- **Probleme**

- Hazards

- WAR und WAW Hazards (s.o.) werden nun möglich und müssen verhindert werden

- Exceptions

- Exceptions dürfen nur von Instruktionen ausgelöst werden, von denen der Prozessor weiß, dass sie auch ausgeführt werden müssen
    - Möglichkeit von "ungenauen" Exceptions
      - Prozessor wird in einem Zustand unterbrochen, der nicht exakt dem Zustand entspricht, der bei Bearbeitung in Programmreihenfolge vorliegen würde
        - » Instruktionen, die vor der auslösenden Instruktion liegen, sind evtl. noch in Bearbeitung
        - » Instruktionen, die nach der auslösenden Instruktion liegen, sind evtl. schon fertig

# Dynamic Pipeline Scheduling (4)

---

- **Funktionale Einheiten**

- Ziel: pro Takt soll eine Instruktion bearbeitet werden (Durchsatz)
- Instruktionen werden so früh wie möglich gestartet (sobald alle Operanden verfügbar sind)
- wenn eine Instruktion wegen eines Konflikts angehalten werden muss, kann die nächste Instruktionen in einer freien funktionalen Einheit gestartet werden
- dazu benötigt man
  - mehrere funktionale Einheiten (ALU's, evtl. spezialisiert auf besondere Operationen)
  - oder Pipelines
  - oder beides gemischt
- der Einfachheit halber wird im Folgenden angenommen, dass mehrere getrennte funktionale Einheiten zur Verfügung stehen
  - ist aber ohne Probleme auf *pipelined functional units* verallgemeinerbar

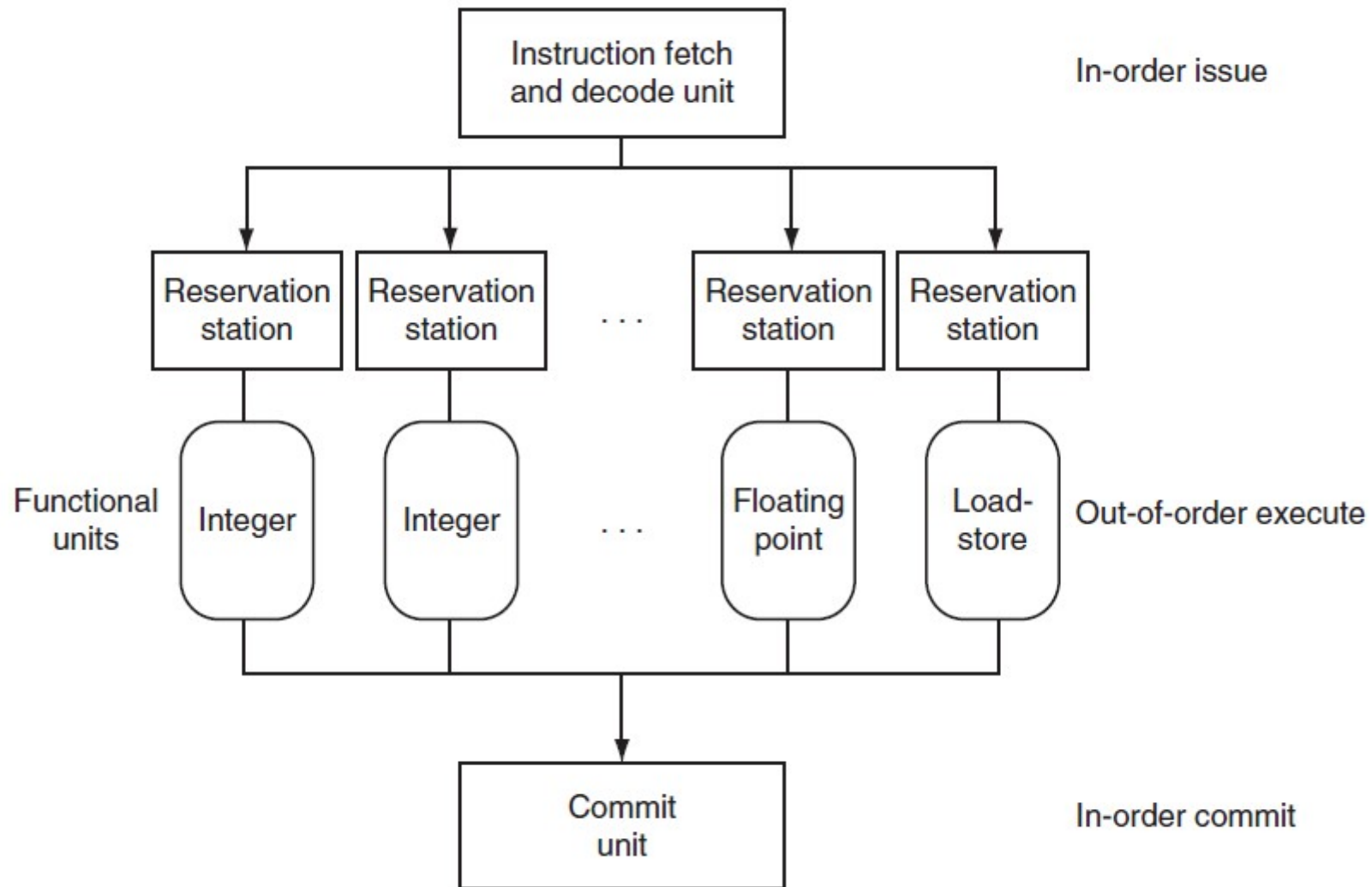
# Dynamic Pipeline Scheduling (5)

---

- **Aufbau, typischerweise in 3 Stufen**
  - *Instruction fetch and decode*
    - holt und dekodiert Befehl
    - sendet Befehl an eine von vielen funktionalen Einheiten in der *Execute*-Stufe
    - arbeitet noch in Programmierreihenfolge (*in-order issue*, Ausgabe in richtiger Reihenfolge)
  - Funktionale Einheiten
    - besitzen Buffer (*reservation station*) für die Operanden und Operationen
    - sobald alle Operanden da sind und die funktionale Einheit frei ist, wird das Resultat berechnet
    - Ausführung in beliebiger Reihenfolge (*out-of-order execute*)
  - *Commit Unit*
    - (engl. *commit*: übergeben, sich verpflichten)
    - entscheidet, wann es sicher ist, ein Resultat in ein Register oder in den Speicher abzulegen (*in-order commit*)

# Dynamic Pipeline Scheduling (6)

---





# Dynamic Pipeline Scheduling (7)

---

- **Steuerung sehr viel schwieriger zu realisieren als für statisches Pipelining**
  - Meist kombiniert mit *branch prediction* (nennt man dann auch *speculative execution*)
  - *Commit unit* muss in der Lage sein, alle Resultate zu löschen, die nach einem falsch vorhergesagten Sprungbefehl bereits initiiert wurden
  - Oft wird *dynamic scheduling* sogar mit superskalarer Arbeitsweise kombiniert
    - Jede funktionale Einheit kann dann z.B. parallel 4 Resultate berechnen.

# Algorithmen für Dynamic Pipeline Scheduling

---

- Scoreboard, 1964
  - Wurde ursprünglich für Control Data CDC 6600 entwickelt.
  - Garantiert, dass keine WAW Konflikte auftreten, indem die Instruktionen angehalten (*stall*) werden, bis Konflikt aufgelöst ist (kostet Zeit).
  - Verhindert RAW Konflikte, indem Operationen erst gestartet werden, wenn alle Operanden verfügbar sind.
  - Bei WAR Konflikten wird das Zurückschreiben des Ergebnisses verzögert, bis alle lesenden Operationen gelesen haben (*stall*, kostet Zeit).
- Tomasulo's Algorithmus, 1967
  - Wurde ursprünglich für IBM 360/91 von Robert Tomasulo entworfen.
  - Verhindert RAW Konflikte, indem die Verfügbarkeit von Operanden überwacht wird (wie oben).
  - Verhindert WAW und WAR Konflikte durch Register-Umbenennung.
    - Keine Datenabhängigkeit

# Abschließende Bemerkungen

---

- **Pipelining ist nicht einfach**
  - 1. Auflage von Patterson, Hennessy enthielt einen Pipeline Bug
  - Nicht entdeckt von 100 Reviewern und Kursen in 18 Universitäten
  - Erst als jemand versuchte, den Computer tatsächlich nach Buch zu bauen, fiel der Fehler auf.
- **Pipelining Ideen sind abhängig von der Technologie**
  - Optimale Anzahl der Stufen hängt von der Technologie ab.
    - Früher waren 5 Stufen mit *delayed branch* optimal.
    - Mit zunehmender Transistorzahl konnten Pipelines immer länger werden.
    - Logik ist viel schneller als Speicher, etc.
  - Viele funktionale Einheiten und *dynamic pipeline scheduling* sind heute sinnvoller.

# Abschließende Bemerkungen (2)

---

- **Design des Befehlssatzes kann Pipelining ungünstig beeinflussen**
  - Komplizierte Befehlssätze machen es schwer, Pipelinestufen auszubalancieren und Konflikte zu erkennen.
  - Raffinierte Adressierungsarten erschweren das Erkennen von Abhängigkeiten, Mehrfachzugriffe auf Speicher erschweren die Pipelinesteuerung.

# Abschließende Bemerkungen (3)

---

