Technische Grundlagen der Informatik 2

Rechnerorganisation

Kapitel 6:
Pipelining

Prof. Dr. Ben Juurlink

Fachgebiet: Architektur eingebetteter Systeme
Institut für Technische Informatik und Mikroelektronik
Fak. IV – Elektrotechnik und Informatik

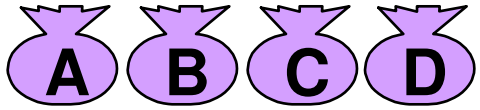
SS 2013/2014

Ziele

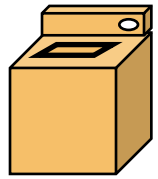
- Nach dieser Vorlesung sollten sie in der Lage sein…
  - Die Grundlagen von Pipelining zu erklären.
  - Pipeliningkonflikte und mögliche Lösungen zu erläutern (Datenkonflikte, Steuerkonflikte , Forwarding).
  - MIPS-Code zu modifizieren, um die Leistung für Prozessoren mit Pipelining zu erhöhen.
  - Die Veränderungen zu benennen, die notwendig sind, um einen Befehl zum Instruktionssatz des pipelined Prozessors hinzuzufügen.
  - Die Steuerung des Pipeline-Prozessors zu beschreiben.
  - Zu erläutern, was Sprung-Vorhersage (*branch prediction*) ist und was Superskalar-Prozessoren sind.

Beispiel (Wäscheproblem - Ann, Brian, Cathy, Dave)

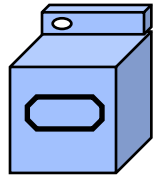
- Jeder von ihnen hatte eine Ladung Kleidung zu waschen, zu trocknen und zu falten.



- Waschmaschine benötigt 30 Minuten.

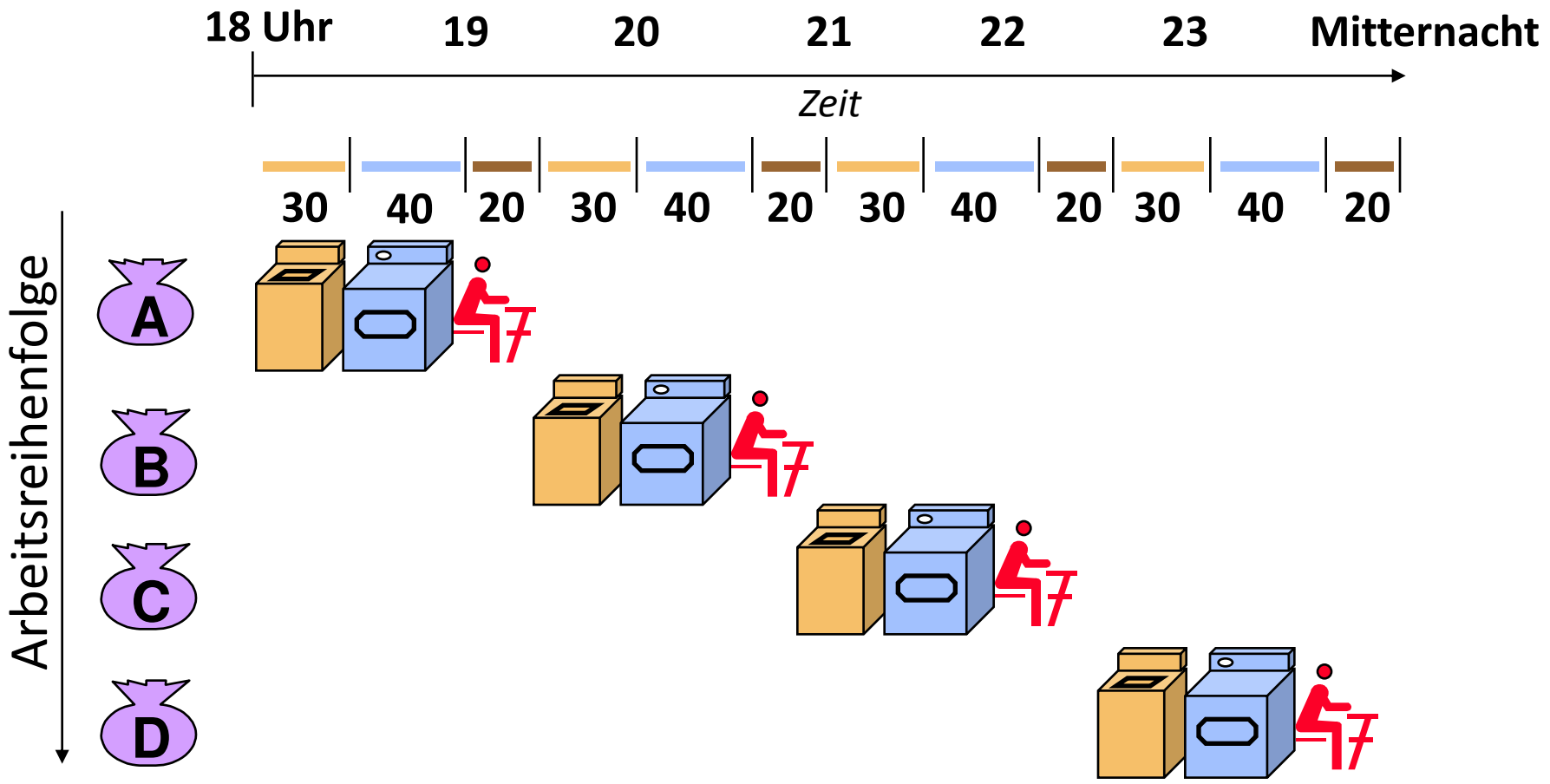


- Der Trockner ist nach 40 Minuten fertig.



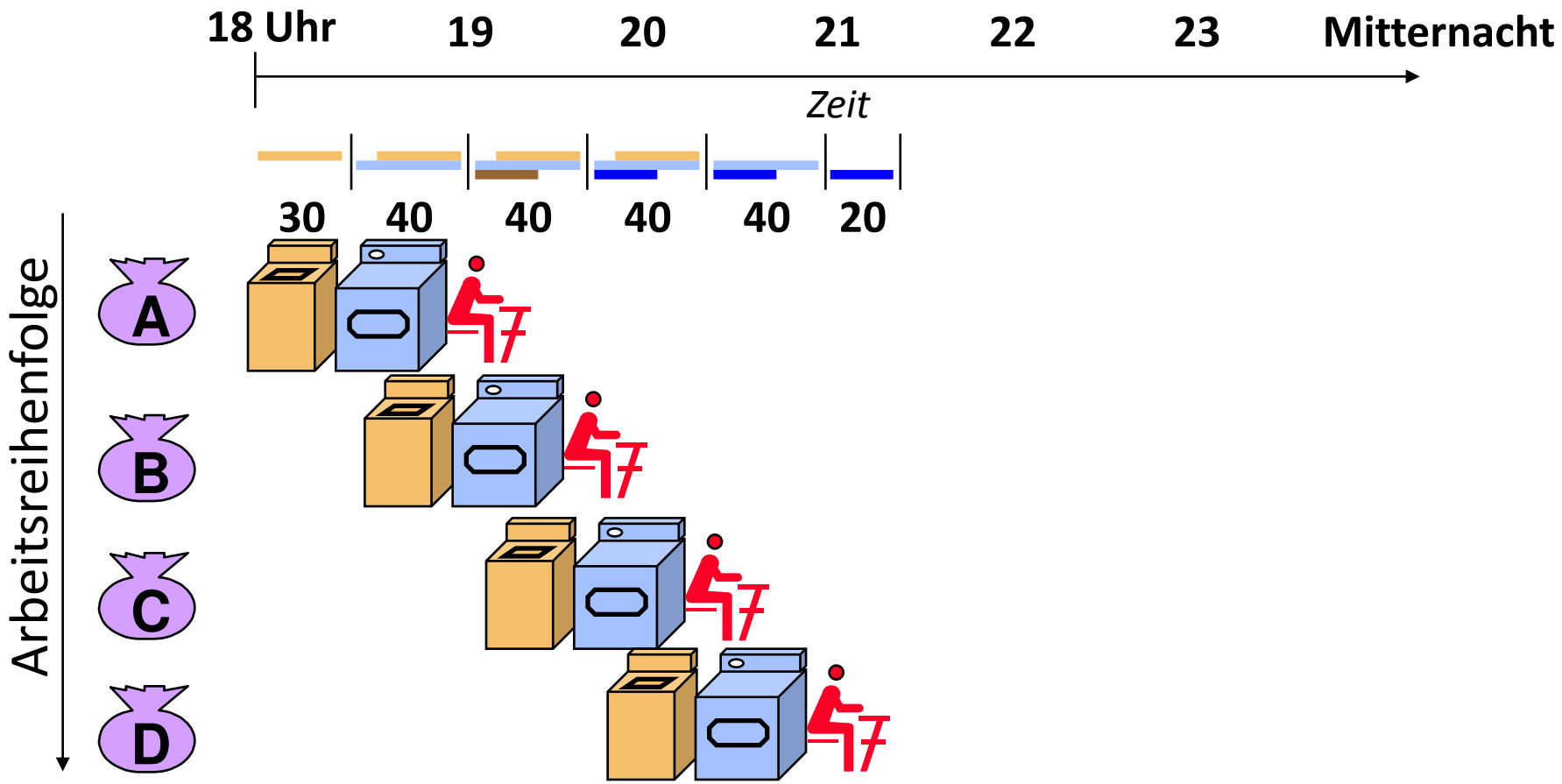
- Fürs Falten werden 20 Minuten benötigt.

Wäscheproblem (sequentiell)



- Gesamter Vorgang für 4 Wäschen dauert 6 Stunden.

Wäscheproblem (Pipeline)- Starte die Arbeit sobald wie möglich



- Mit Pipelining ist das Wäscheproblem in 3,5 Stunden gelöst.

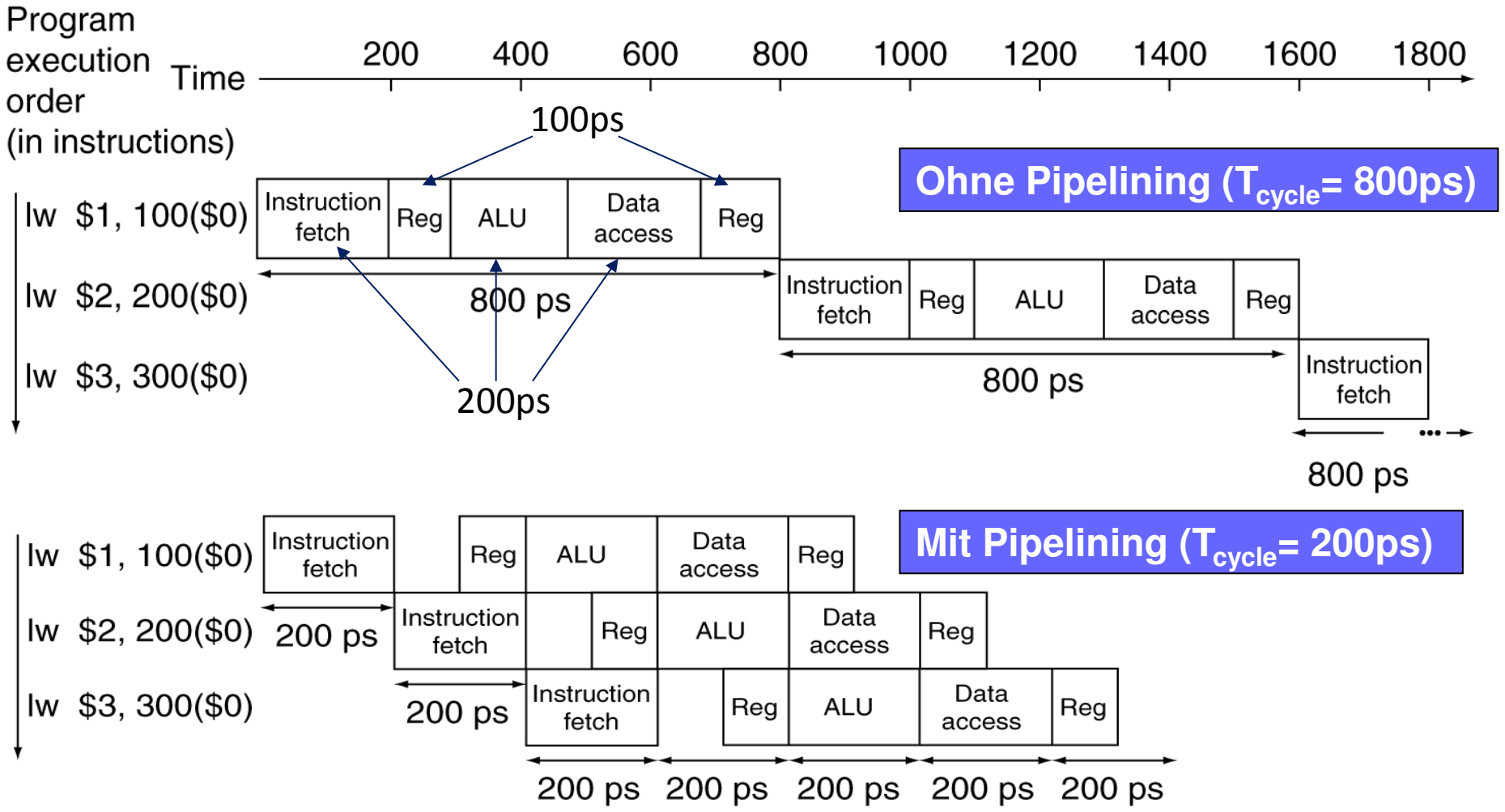
- MIPS-Befehle haben typischerweise 5 Ausführungsschritte:
  1. Befehlsholschritt - Instruction Fetch (IF)
  2. Befehlsentschlüsselung- und Registerholschritt - Instruction Decode (ID)
  3. Ausführung oder Adressberechnung - Execution (EX)
  4. Speicherzugriff – Memory Access (MEM)
  5. Rückschreiben - Write Back (WB)

- Nicht alle Befehle brauchen alle Stufen, aber sie müssen diese Stufen passieren, da die Pipeline synchron bleiben muss.

Befehl	Zyklus								
	1	2	3	4	5	6	7	8	9
Befehl 1	IF	ID	EX	MEM	WB				
Befehl 2		IF	ID	EX	MEM	WB			
Befehl 3			IF	ID	EX	MEM	WB		
Befehl 4				IF	ID	EX	MEM	WB	
Befehl 5					IF	ID	EX	MEM	WB

- Pipelining verbessert die Leistung durch das Erhöhen des Durchsatzes, **nicht** durch Reduzierung der Ausführungszeit der einzelnen Befehle.

Leistung (mit und ohne Pipelining)



Best möglicher Speedup

- Best möglicher Speedup gegenüber Eintakt-Implementierung = Anzahl der Stufen

- Erreichen wir das?

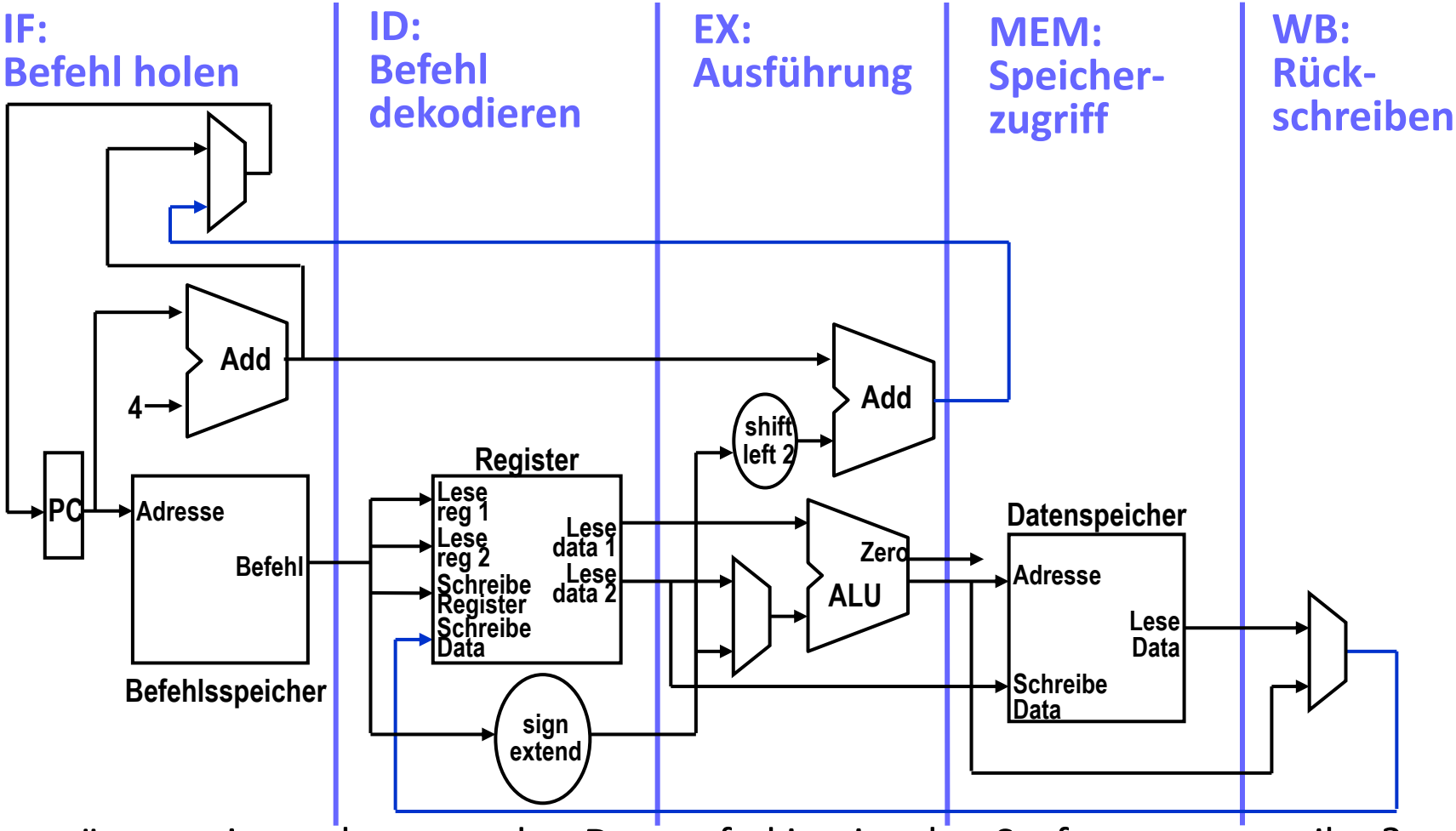
- MIPS-ISA wurde für Pipelining entworfen.
  - Alle Befehle 32 Bit lang.
    - einfach in einem Takt Befehl zu holen und PC zu inkrementieren
    - Vgl. x86: 1- bis 17-Byte Instruktionen
- Wenige und regelmäßige Befehlsformate
  - Dadurch können in der selben Stufe Befehle dekodiert und Register gelesen werden.
- Load-/Store-Adressierung
  - Kann in der 3. Stufe Adresse berechnen und in der 4. Stufe auf den Speicher zugreifen.

Pipeline Konflikte / Hazards

- Konflikte, Hemmnisse (*Hazards*): Situationen, die das Ausführen des nächsten Befehls im nächsten Takt verhindert.

- Strukturkonflikte (*structural hazards*)
  - benötigte Ressource ist belegt
- Datenkonflikte (*data hazards*)
  - auf das Ergebnis eines vorherigen Befehls muss gewartet werden
- Steuerkonflikte (*control hazards*)
  - Falls verzweigt (*branch*) wird, befinden sich bereits andere Befehle in der Pipeline

Basisidee: unterteile Eintaktdatenpfad

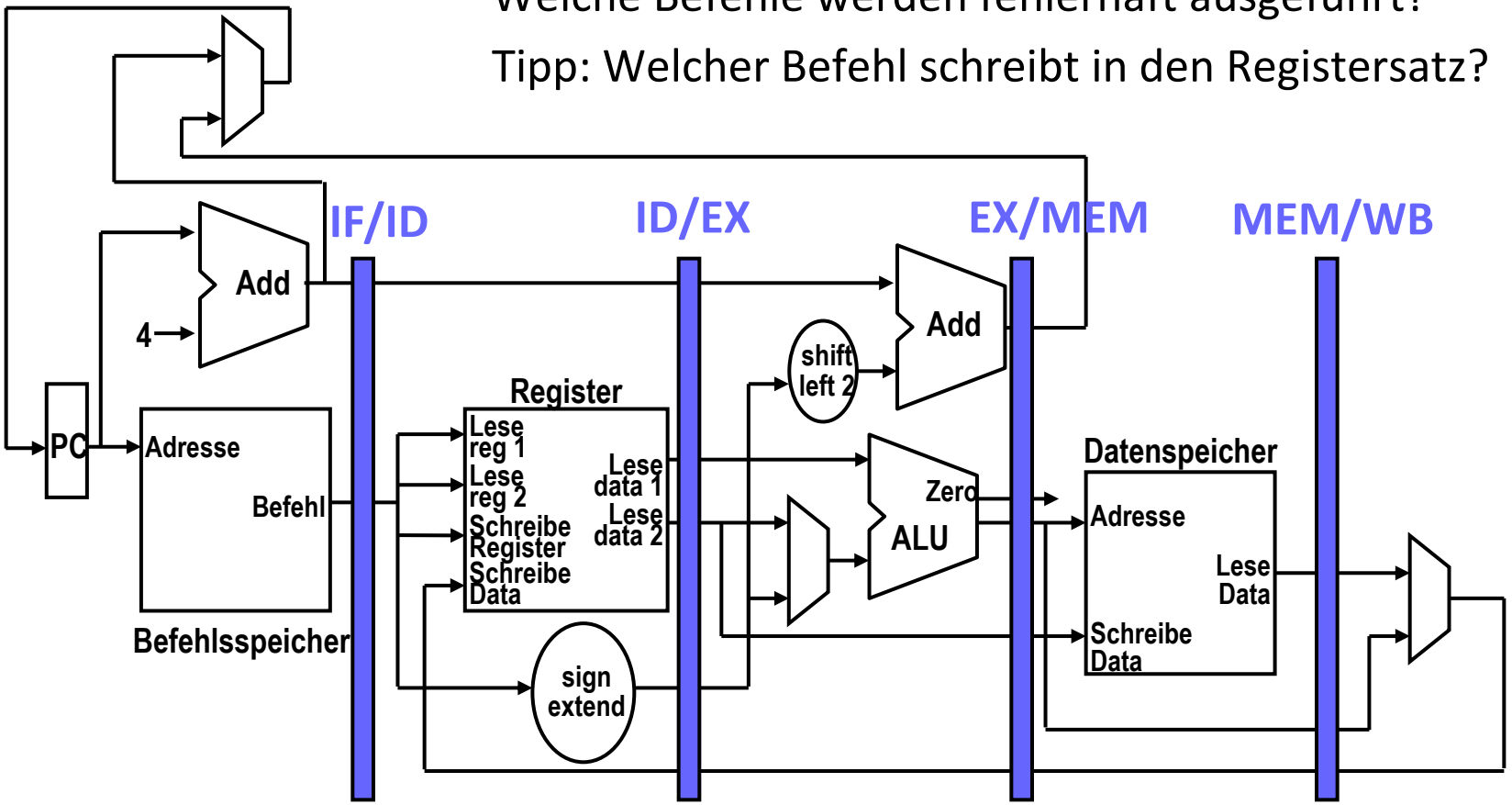


Was müssen wir machen, um den Datenpfad in einzelne Stufen zu unterteilen?

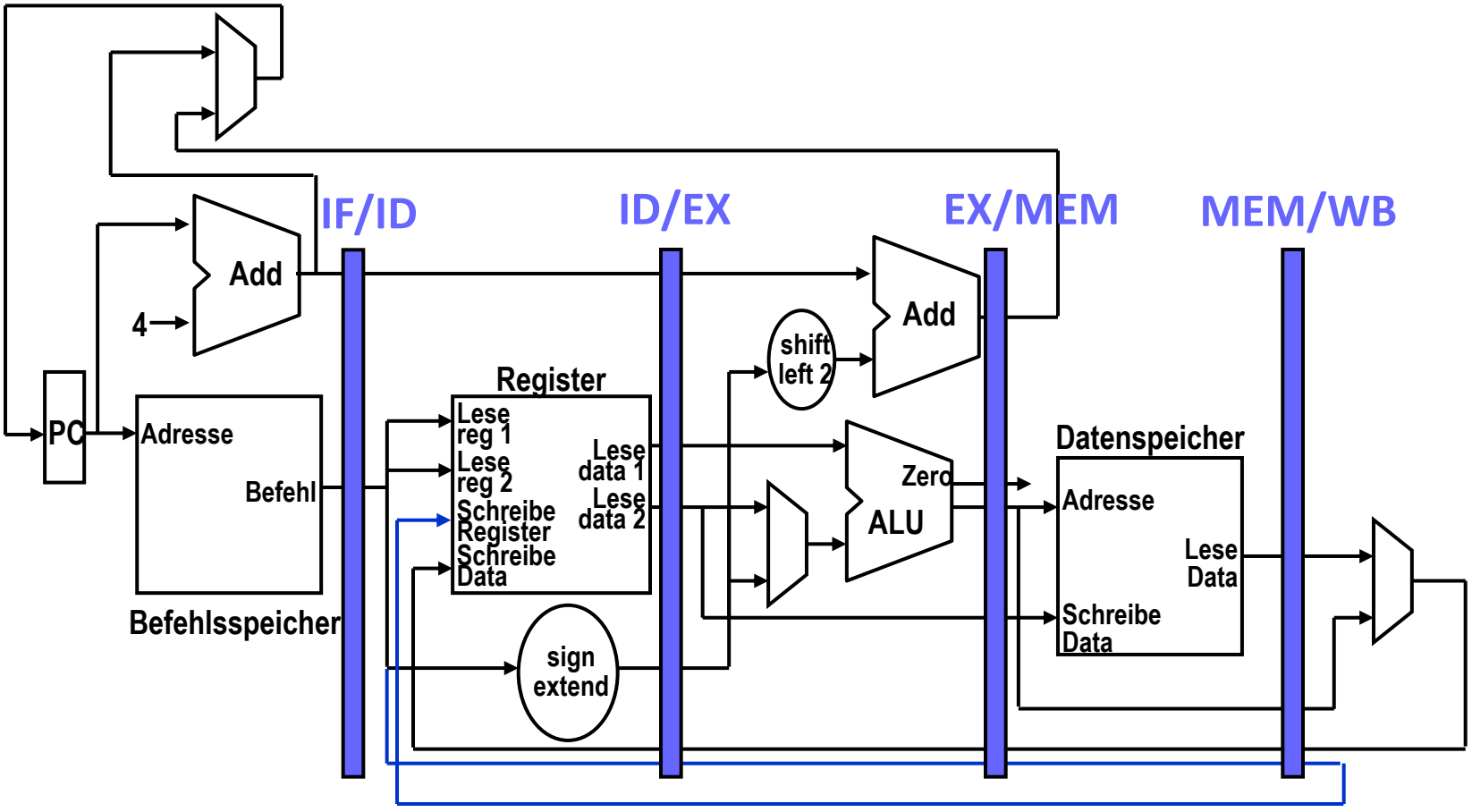
Datenpfad mit Pipelining

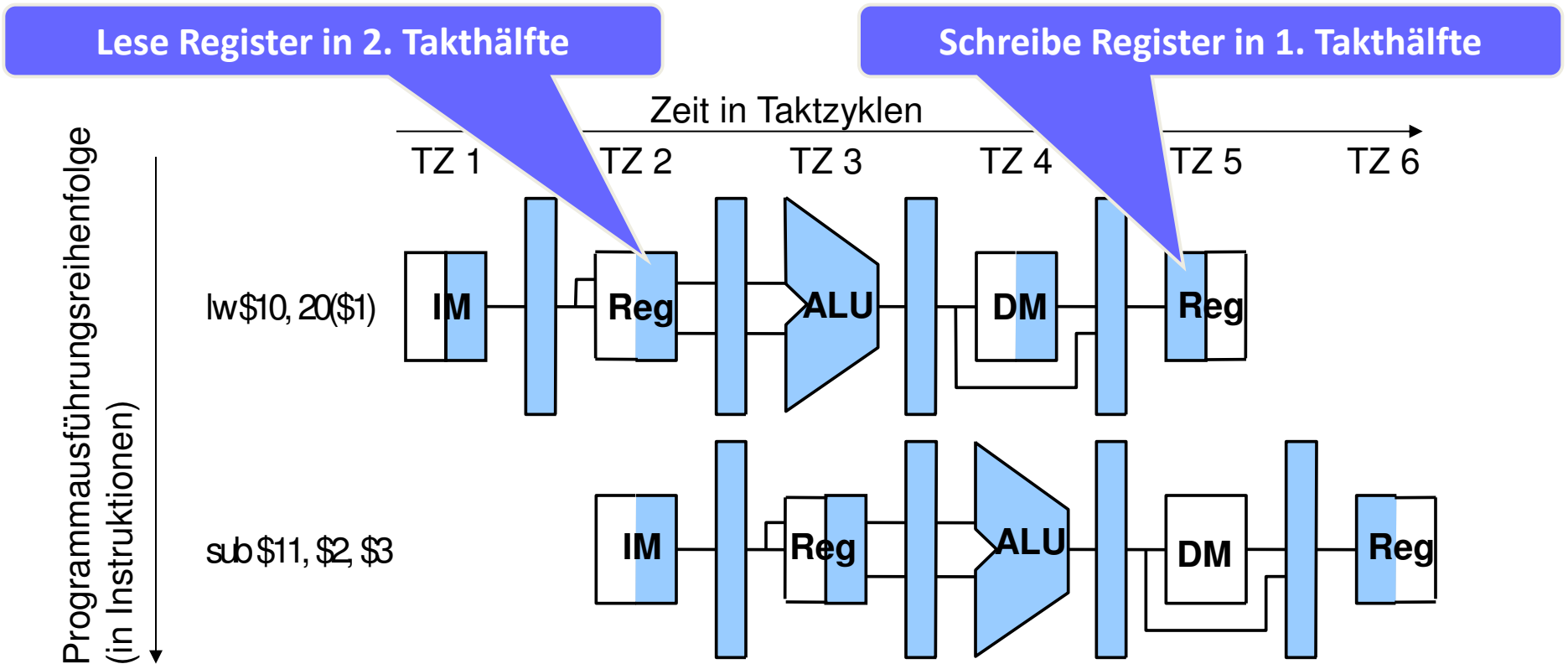
- Es gibt ein Fehler im Design. Können sie ihn finden? Welche Befehle werden fehlerhaft ausgeführt?

  Tipp: Welcher Befehl schreibt in den Registersatz?



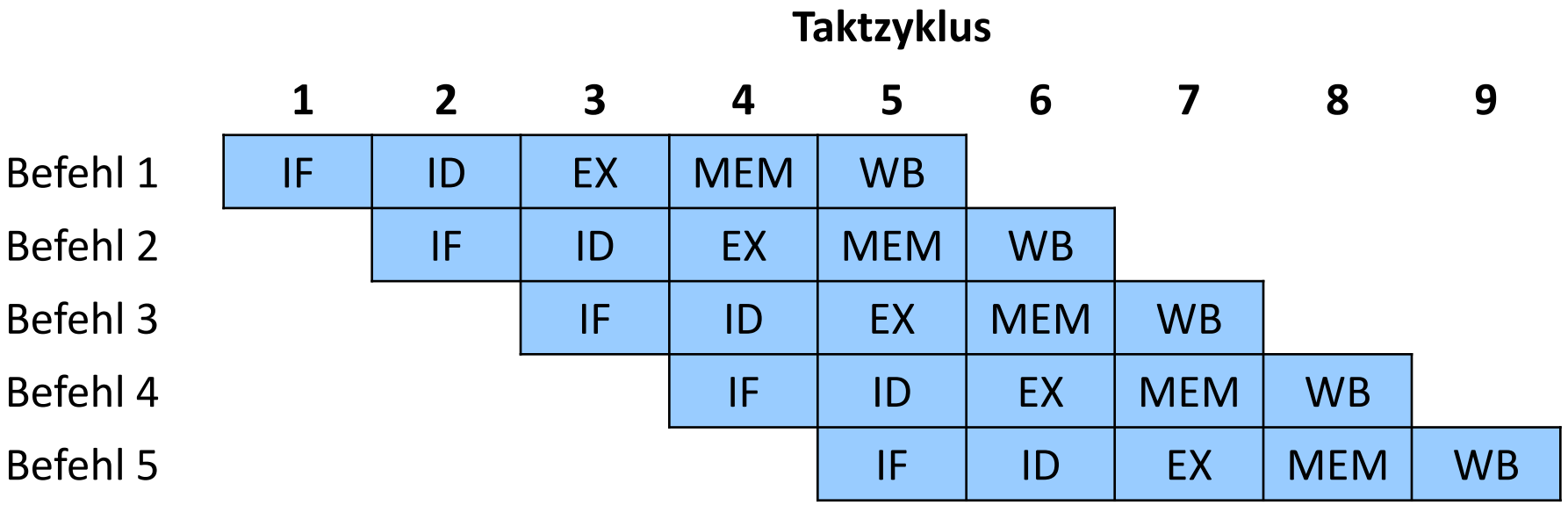
Korrekter Datenpfad mit Pipelining



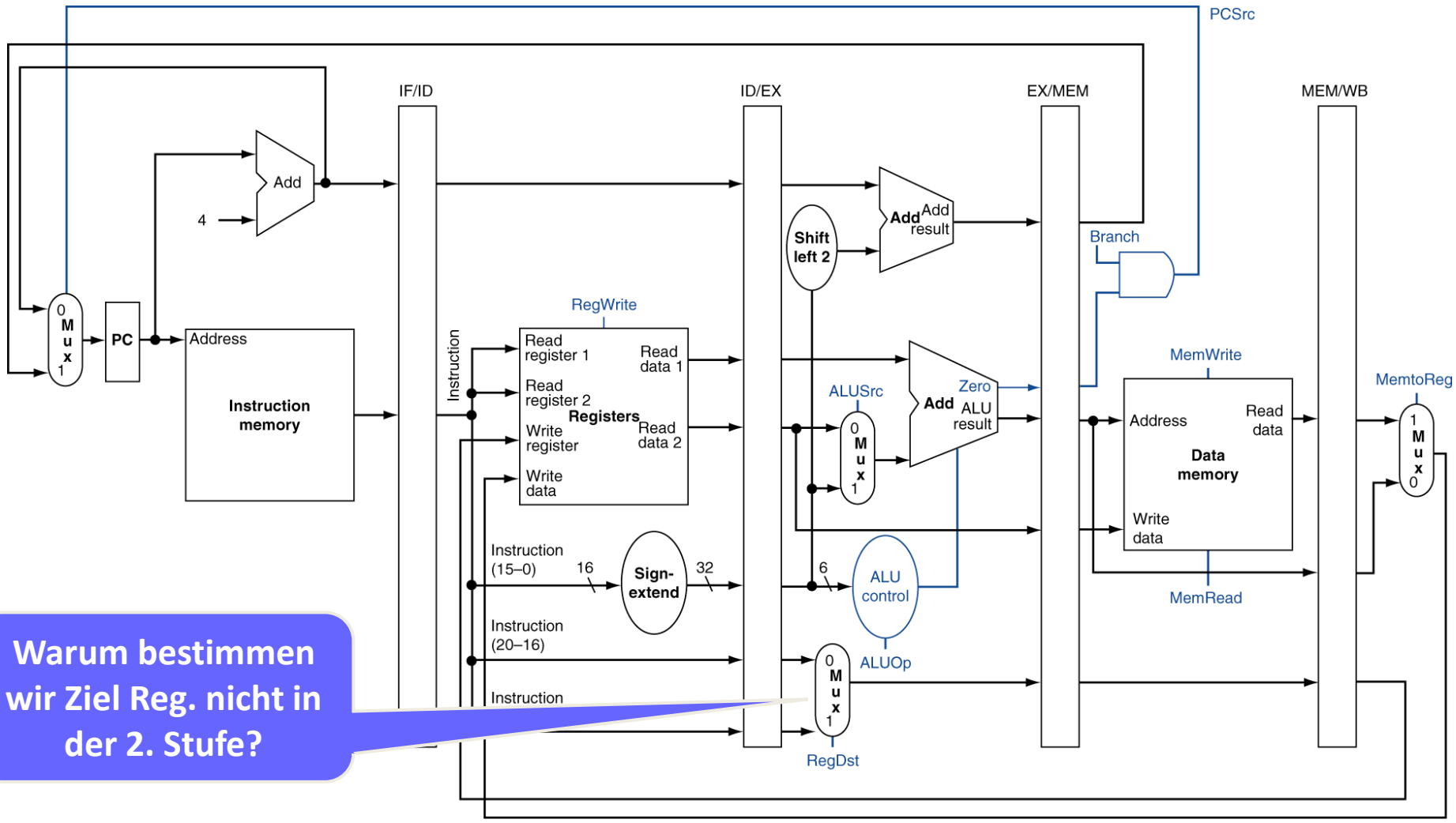


- Wie viele Takte werden zum Ausführen dieses Codes benötigt?
- Was macht die ALU im 4. Taktzyklus?

- Traditionelle Darstellung:



Steuerung der Pipeline 1



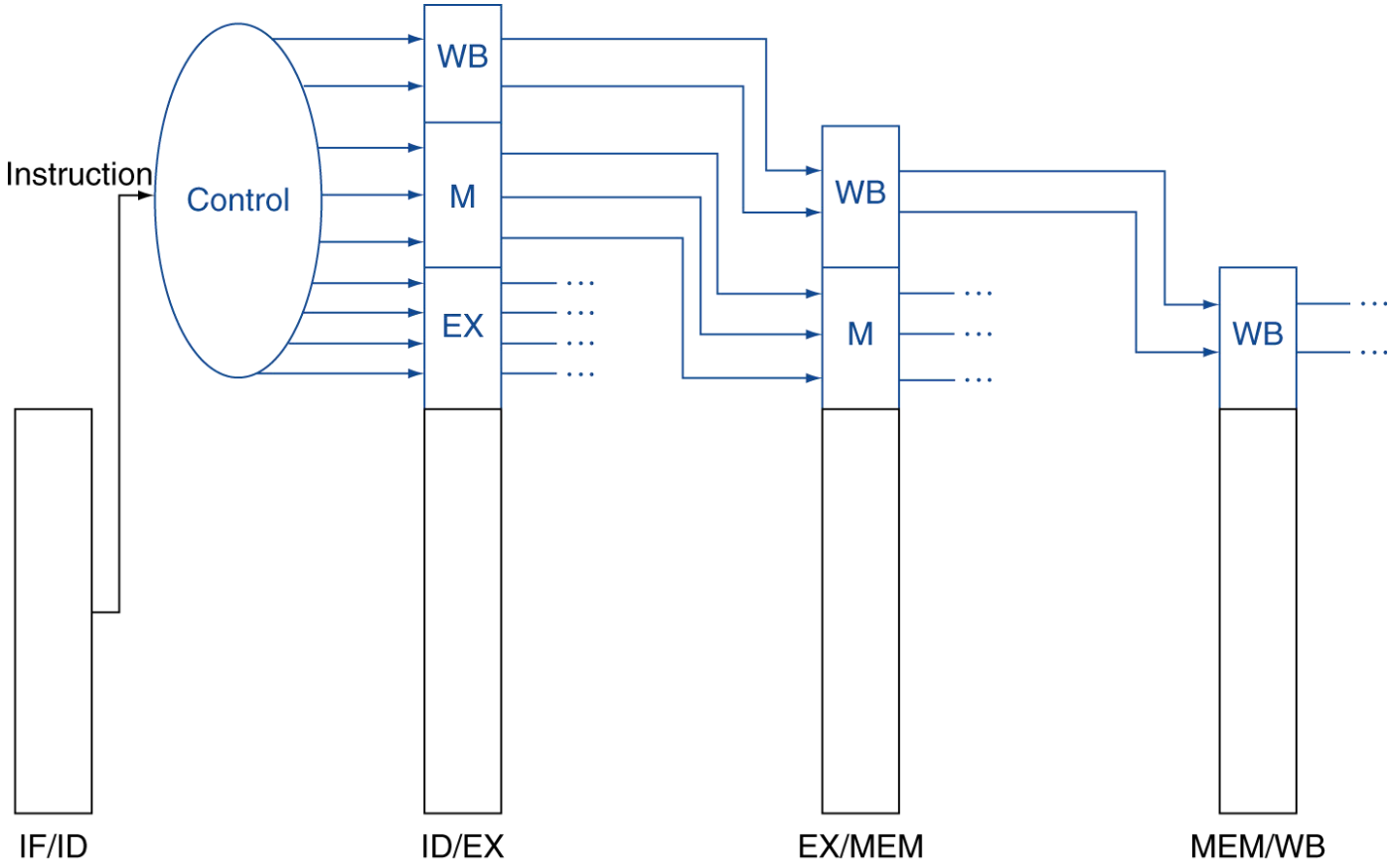
Warum bestimmen wir Ziel Reg. nicht in der 2. Stufe?

Steuerung der Pipeline 2

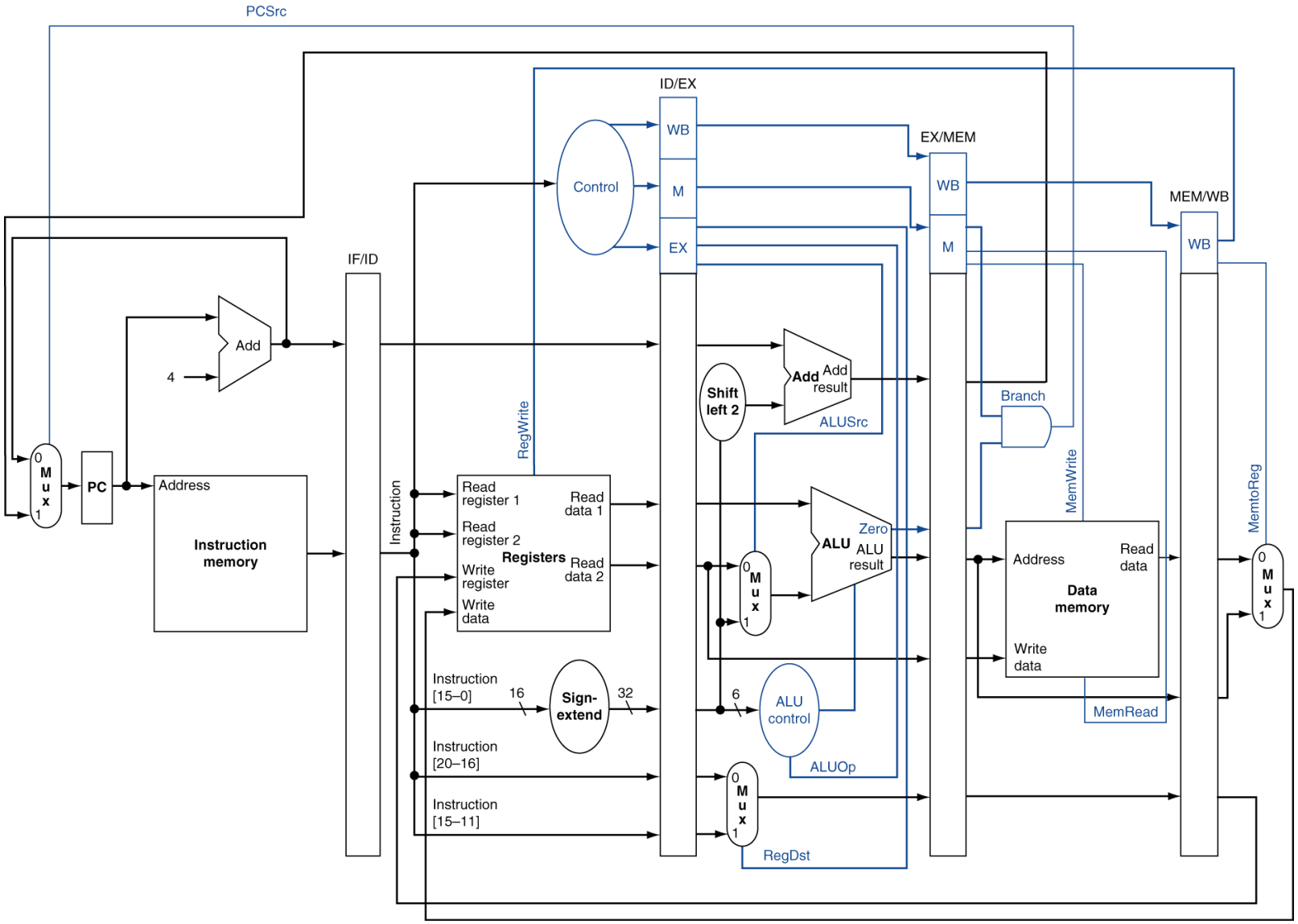
- Steuersignale sind identisch zu denen in der Eintakt-Implementierung.
- Was muss in jeder Stufe gesteuert werden?
  - Befehl holen und PC Inkrementieren
  - Befehl entschlüsseln / Register lesen
  - Befehl ausführen oder Adresse berechnen
  - Speicherzugriff
  - Ergebnis Rückschreiben

	Ausführung/Adresse berechnen				Speicherzugriff			Rückschreiben	
Befehl	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
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

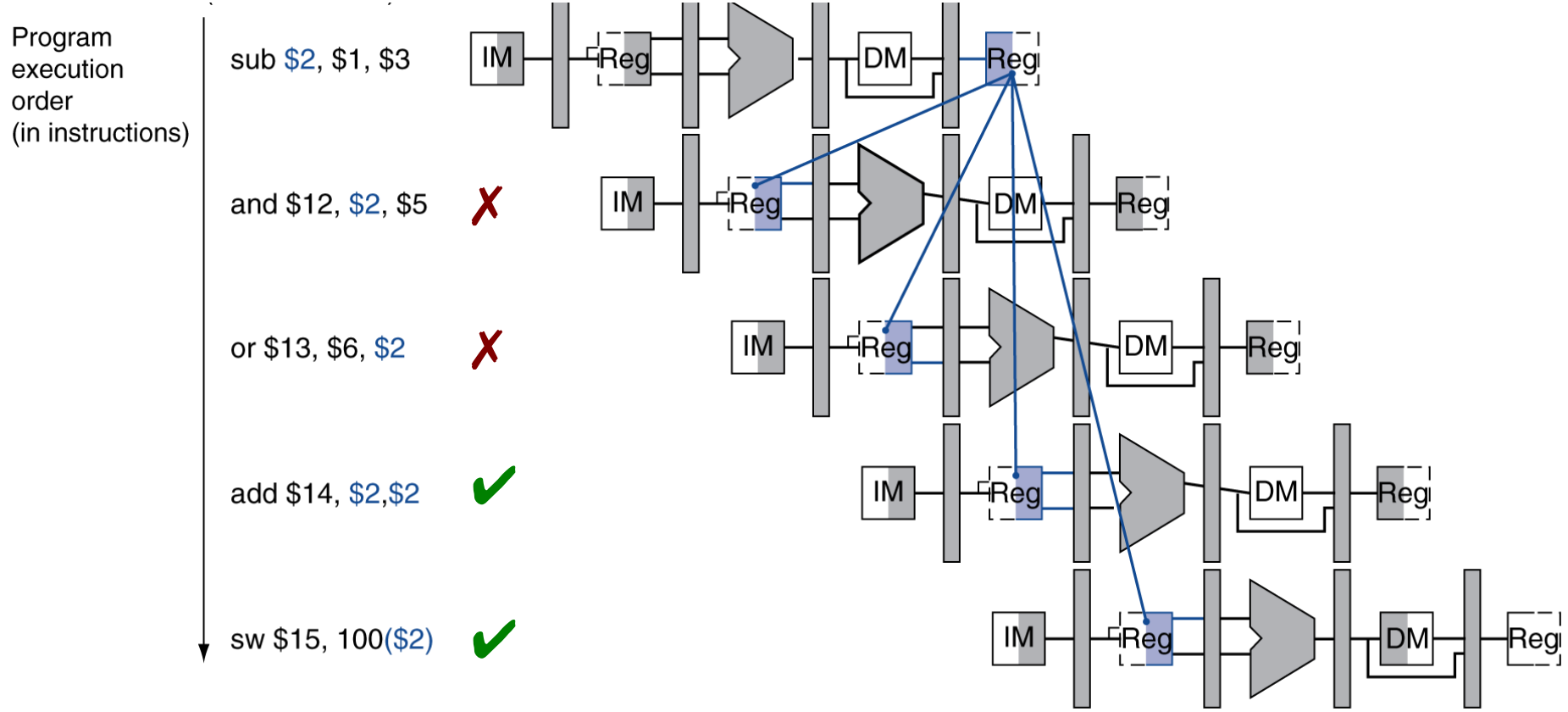
- Durchschalten der Steuersignale durch die Stufen wie Daten.



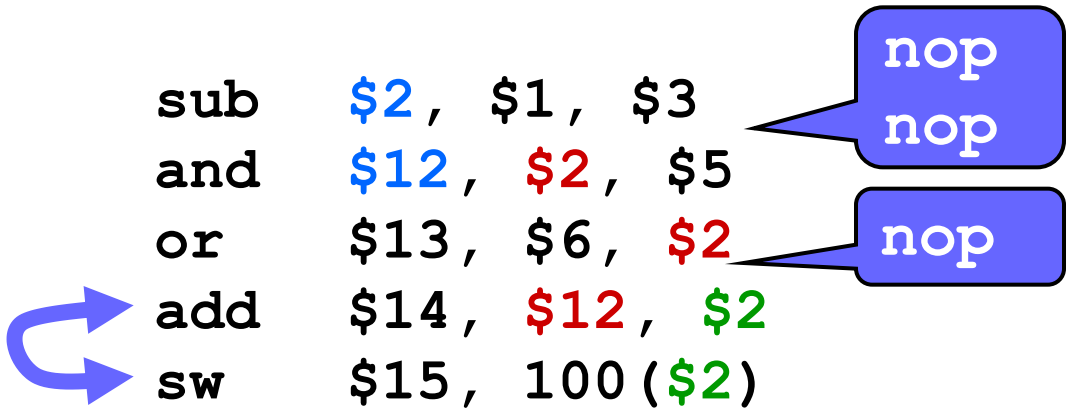
Steuerung der Pipeline 4



- Ergebnis wird im Takt **i** geschrieben, soll aber bereits im Takt **i-2** und **i-1** gelesen werden.

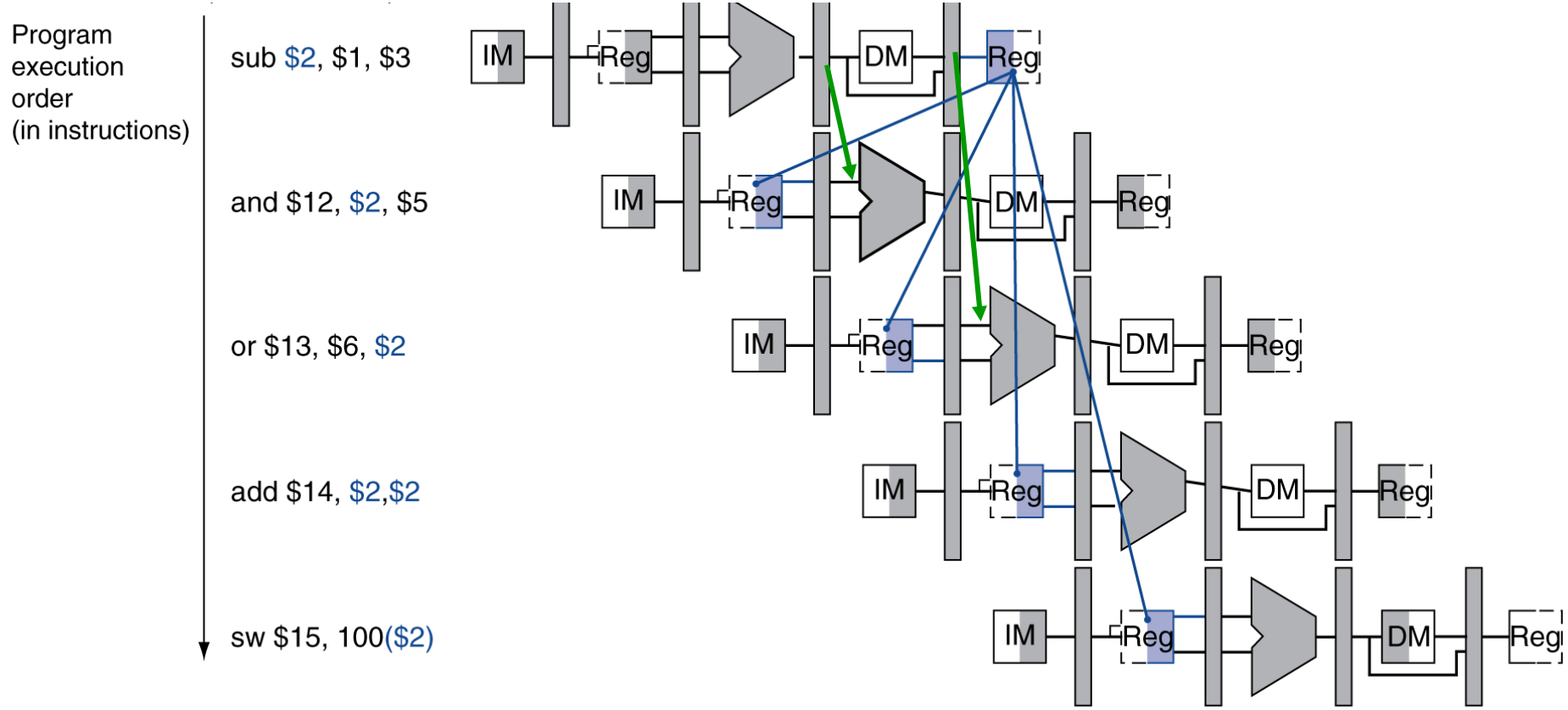


- Compiler muss garantieren, dass keine Konflikte auftreten.
- Wo müssen wir „No Operations“ (NOPs) einfügen?

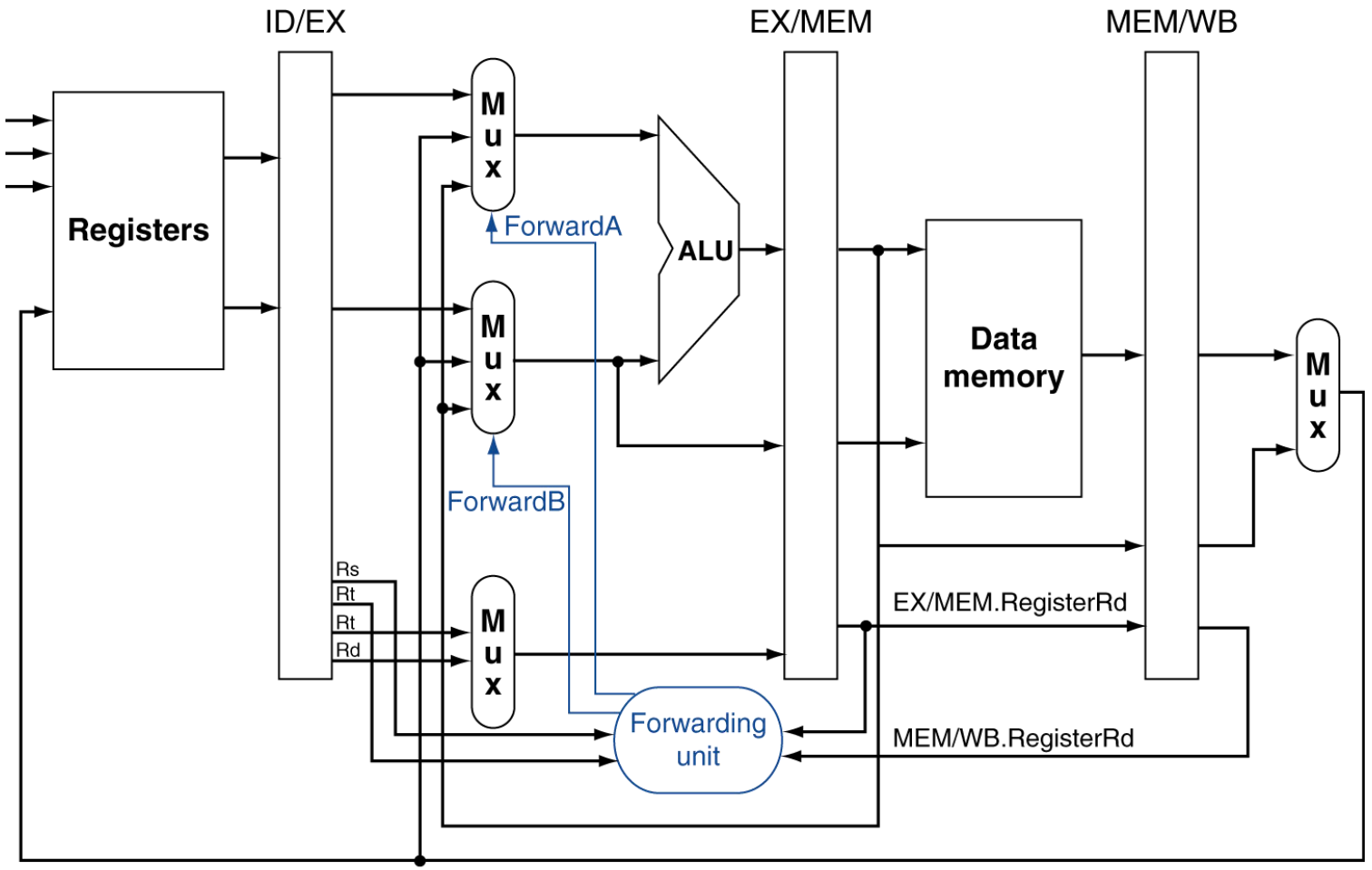


- Aber das verlangsamt die Ausführung extrem!
  - Kann manchmal durch Veränderung der Befehlsfolge vermieden werden.

- Ergebnisse werden sofort weitergeleitet, ohne darauf zu warten bis sie in den Registersatz zurückgeschrieben wurden.



Forwarding Einheit



Bedingungen zum Erkennen von Konflikten

- EX/MEM-Konflikt:

```
if (EX/MEM.RegWrite && EX/MEM.Rd≠0){
    if (EX/MEM.Rd==ID/EX.Rs) ForwardA = 10;
    if (EX/MEM.Rd==ID/EX.Rt) ForwardB = 10;
}
```

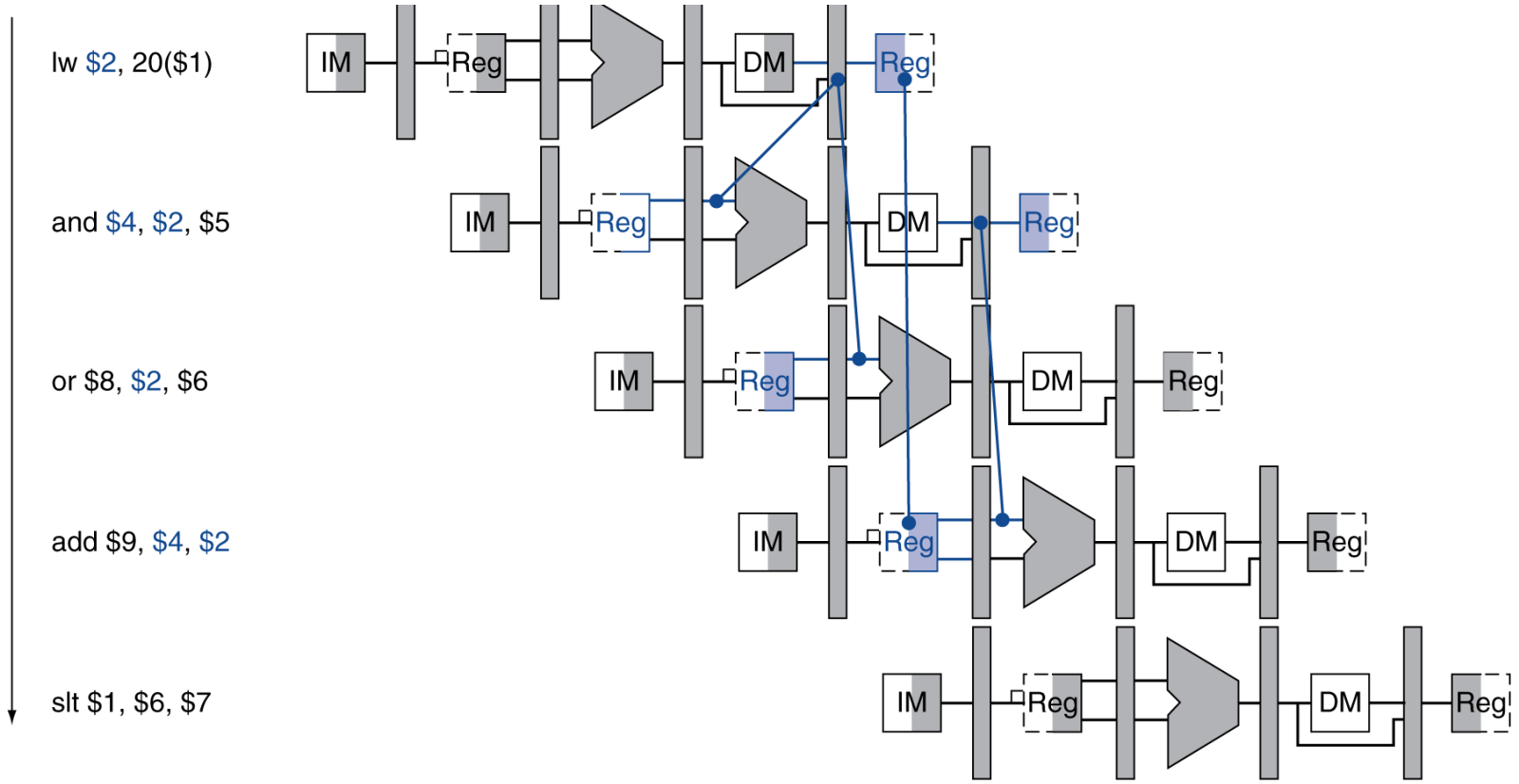
- MEM/WB-Konflikt:

```
if (MEM/WB.RegWrite && MEM/WB.Rd≠0){
    if (EX/MEM.Rd≠ID/EX.Rs && MEM/WB.Rd==ID/EX.Rs)
        ForwardA = 01;
    if (EX/MEM.Rd≠ID/EX.Rt && MEM/WB.Rd==ID/EX.Rt)
        ForwardB = 01;
}
```

- IMHO Spezifikation unvollständig, da ForwardA und ForwardB nie auf 00 gesetzt werden

Load-Use Konflikt

- Kann nicht immer “forwarden”: **`lw`** kann Konflikt auslösen.
  - Befehl nutzt Register, dass durch vorheriges Ladebefehl geschrieben wird.



Abhandeln von Load-Use Konflikten

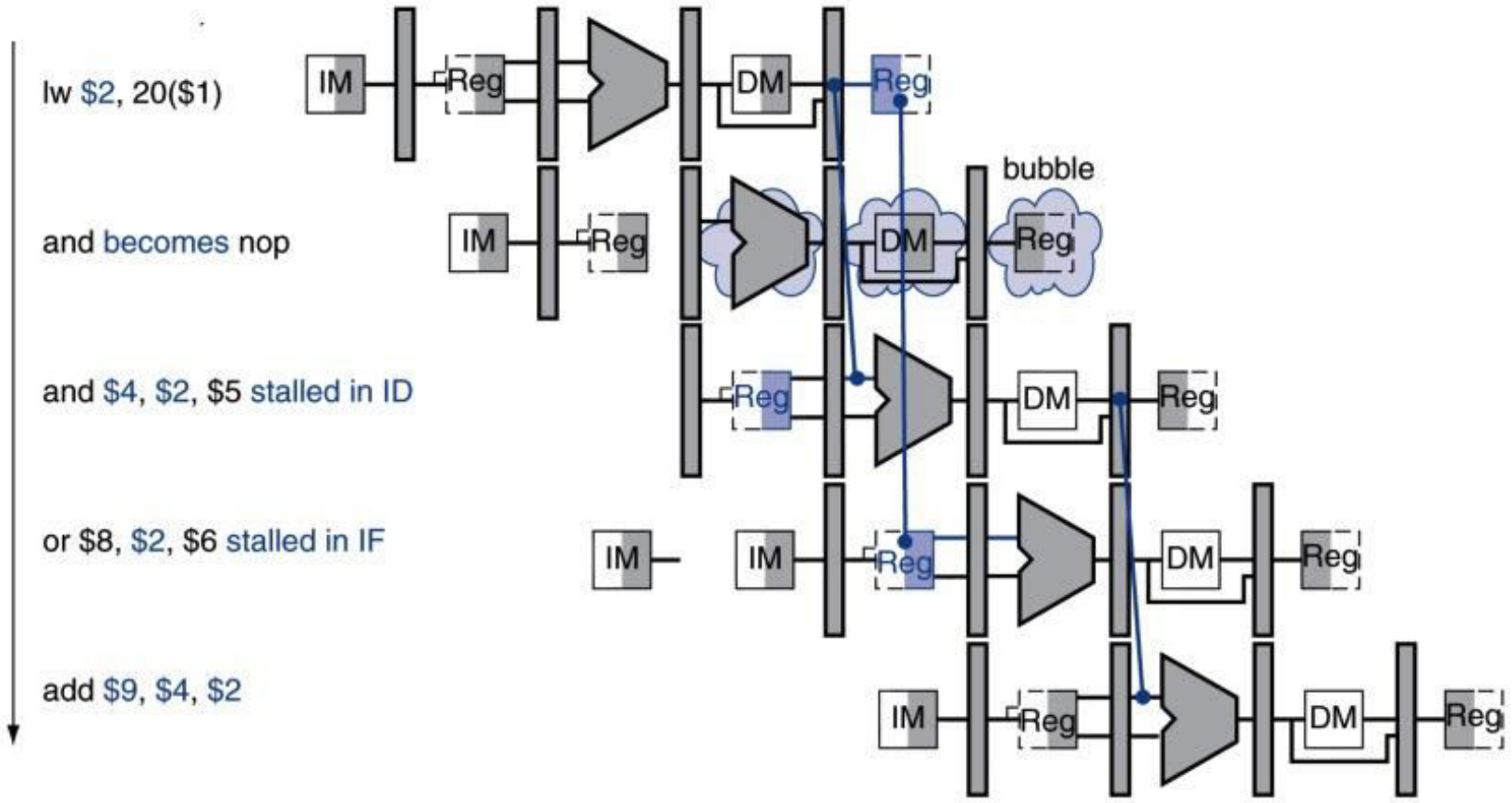
- Man benötigt
  - Einheit zum erkennen von solchen Konflikten (*Hazard Detection Unit*)
  - Möglichkeit die Pipeline anzuhalten

- Erkennen solcher Konflikte:

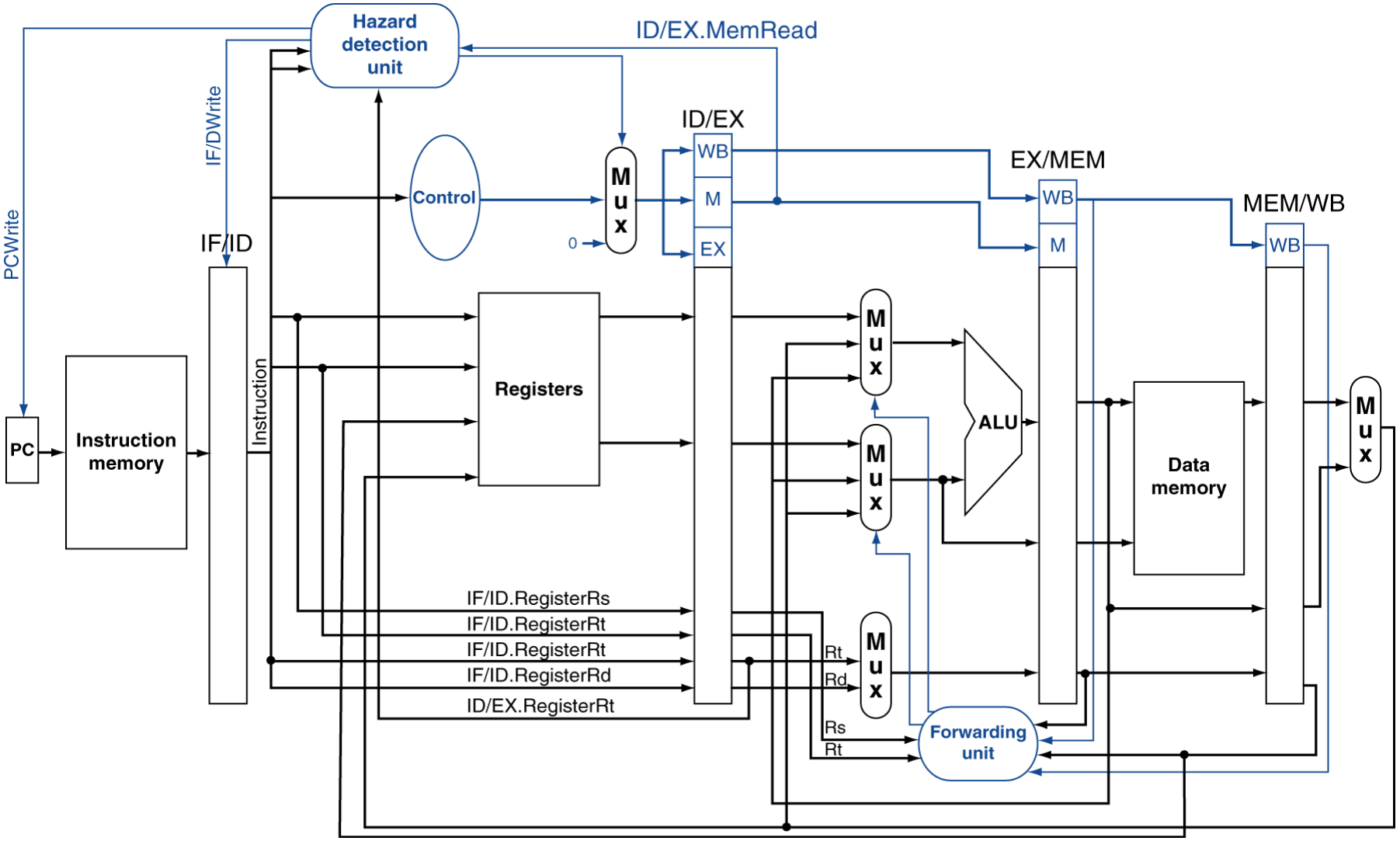
```
ID/EX.MemRead & (ID/EX.Rt=IF/ID.Rs | ID/EX.Rt=IF/ID.Rt)
```

- Anhalten der Pipeline:
  - Befehle nach dem Ladebefehl in der selben Stufe halten
    - Write-Signalen zu PC- und IF/ID-Pipeline-Register hinzufügen
  - NOP-Befehl einfügen
    - Einfügen einer „Blase“ (*bubble*)

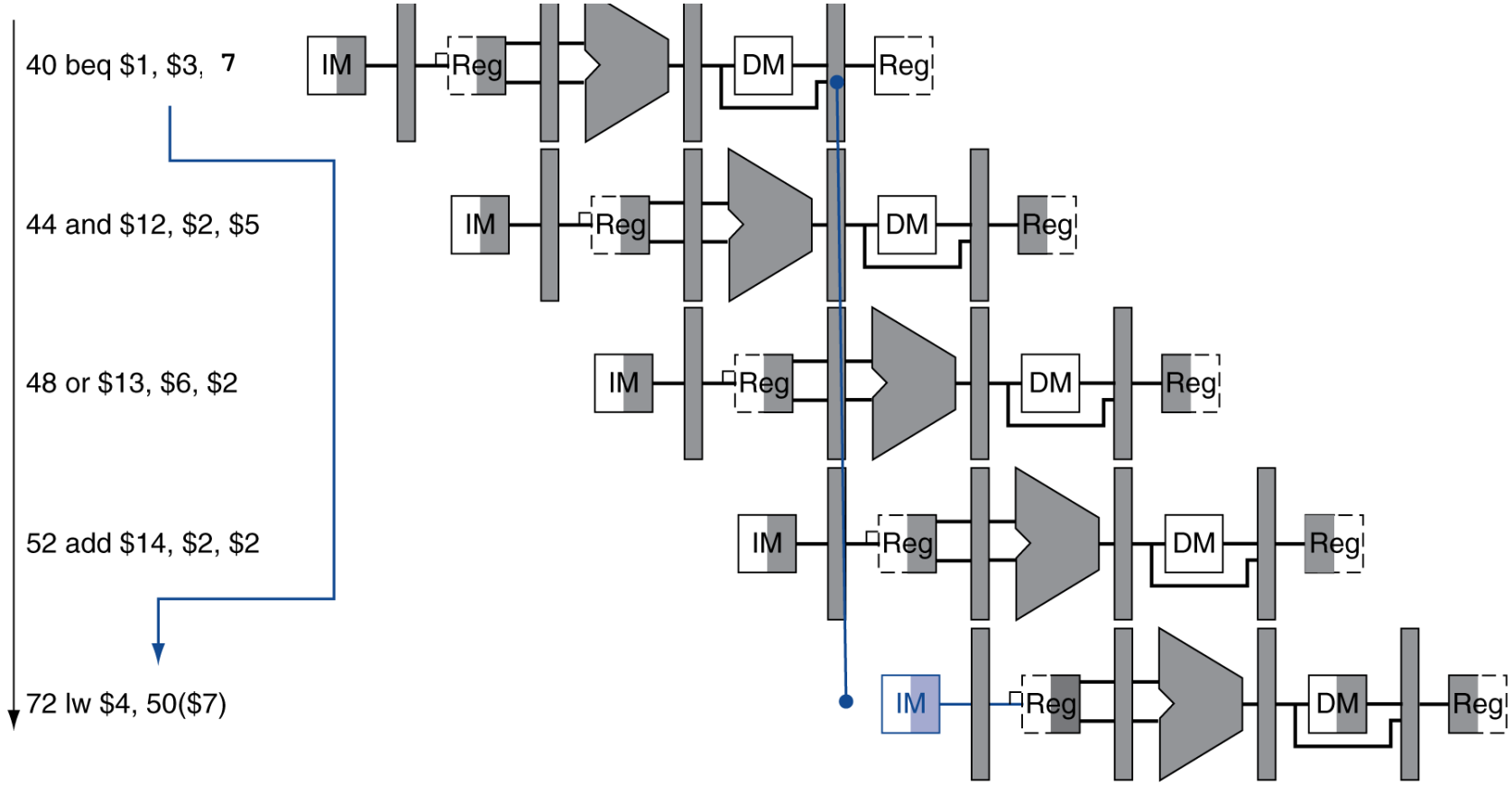
Verzögerung der Pipeline



Datenpfad mit Hazard-Detection-Einheit

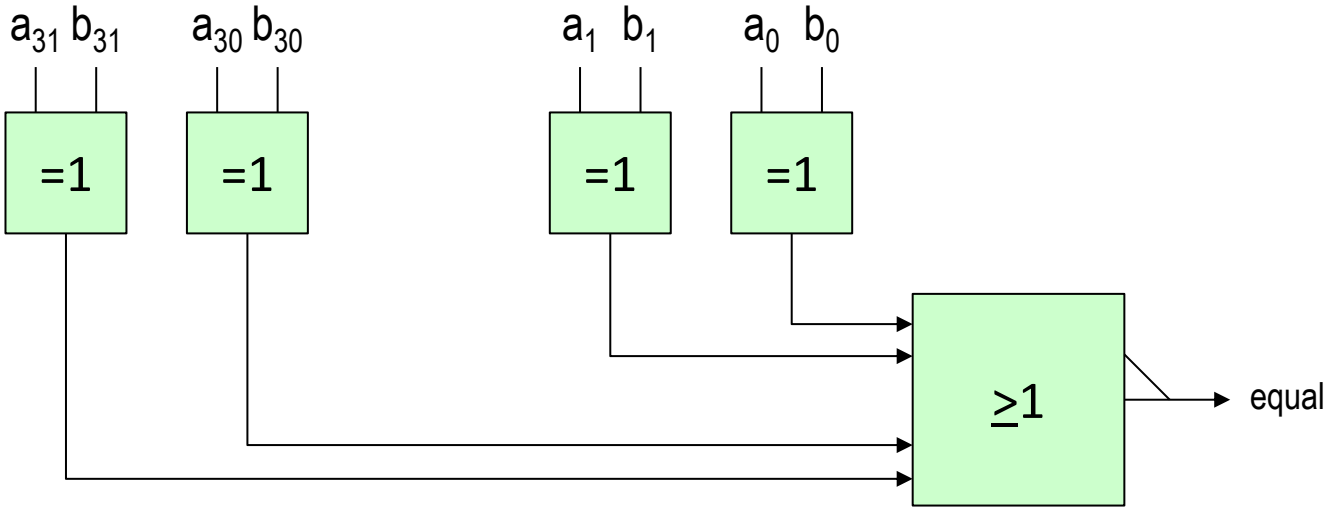


Steuerkonflikte / Verzweigungskonflikte



Lösungen für Steuerkonflikte 1

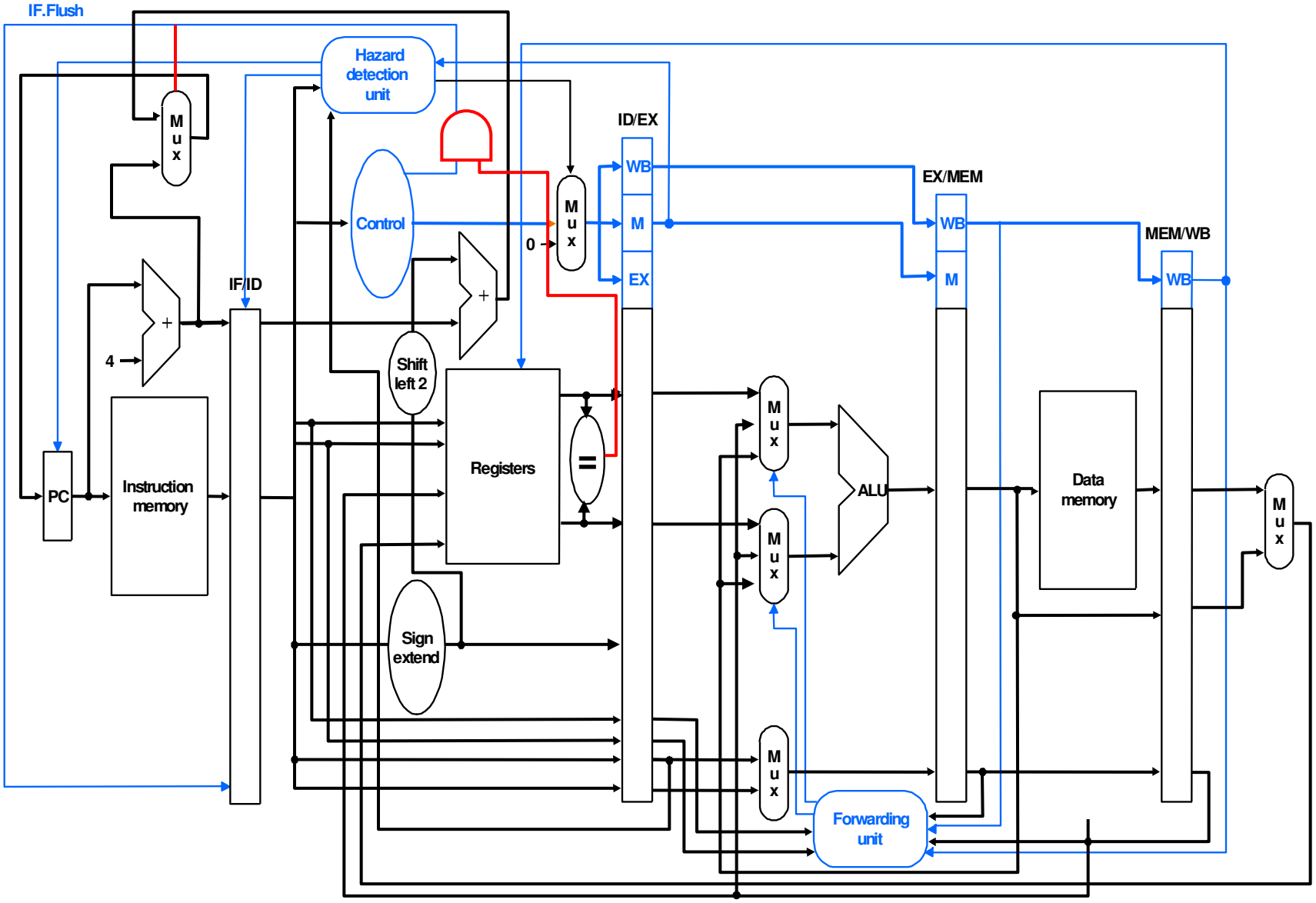
- Pipeline anhalten, wenn eine Verzweigung auftritt.
  - 3 Warte-Zyklen bei jeder Verzweigung
- Sprungausführung in frühere Stufe verlegen
  - zur EX-Stufe ist einfach (s. Folie 21)
  - beim MIPS auch zur ID-Stufe möglich
    - MIPS unterstützt nur einfache Verzweigungen (**`beq, bne, bltz,`** …)



Lösungen für Steuerkonflikte 2

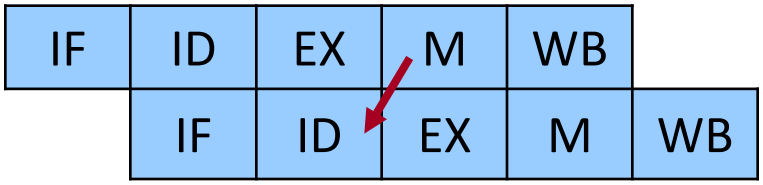
- Verzweigung in ID-Stufe verlagern und
  - annehmen, dass nicht verzweigt wird
    - Lösche (*flush*) nächsten Befehl, falls Annahme falsch
    - ist Sprungbedingung falsch, so entsteht kein Verlust
    - wird jedoch gesprungen, so geht 1 Zyklus verloren
  - *Branch Delay Slot* (Verzögerter Sprung) hinzufügen
    - Befehl nach der Verzweigung wird immer ausgeführt
    - Compiler muss *Delay Slot* mit sinnvollem Befehl oder **`nop`** füllen
    - Lösungsansatz wurde im ersten MIPS-Prozessor verwendet.
    - Delay Slot existiert immer noch beim MIPS. Warum?
  - Dynamische Sprungvorhersage (*dynamic branch prediction*)

Verzweigen in ID-Stufe und Annahme, dass Sprung nicht ausgeführt wird



- Forwarding bei Befehlen mit nur einem (`lw,addi,...`) oder keine (`j,...`) Quellregistern
- Um Verzweigungen in ID-Stufe zu verlagern
  - zusätzliche Forwardlogik in ID-Stufe nötig
  - Verzweigungen können keine Register verwenden, welches durch einen vorherigen ALU- oder Ladebefehl geschrieben wurde.
    - solche Datenkonflikte erkennen und Pipeline anhalten

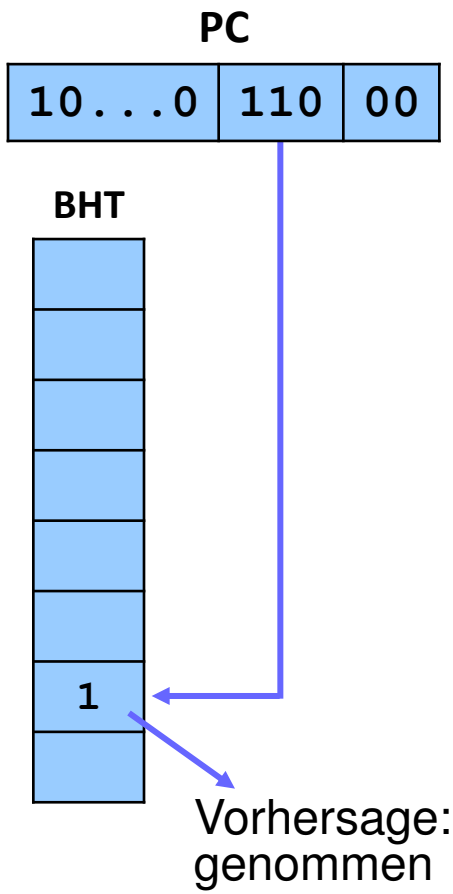
```
addi $t0,$t0,1
beq  $t0,$t1,-4
```



Dynamische Sprungvorhersage

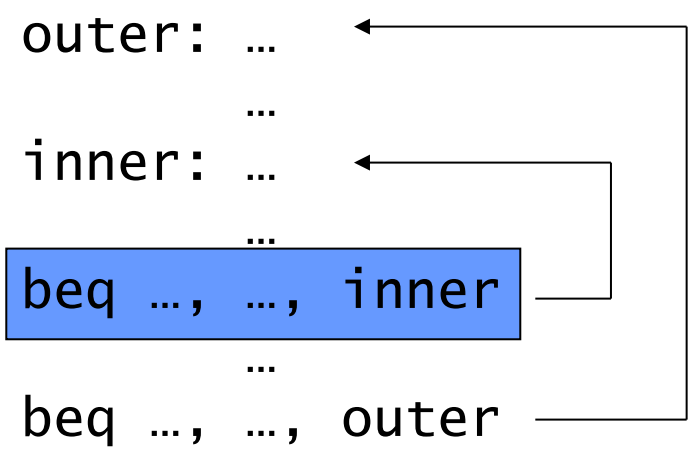
- Moderne Prozessoren haben tiefere Pipelines
  - Z. B. Intel Core i7: 16 Stufen
  - Sprungverzögerung (*branch penalty*) nimmt zu
  - Unmöglich alle *delay slots* mit nützlichen Befehlen zu füllen
- Dynamische Sprungvorhersage (*dynamic branch prediction*)
  - Vorhersage von Sprung an Hand seines Verlaufs
    - Falls voriges Mal genommen (*taken*), nehme an wieder genommen
    - Falls nicht genommen (*not taken*), nehme an wieder nicht

- Sprungvorhersagepuffer (*branch prediction buffer*) / Sprungverlaufstabelle (*branch history table* [BHT])
  - Indiziert durch unteren Teil der Befehlsadresse (PC)
  - Speichert die vorherigen Sprungergebnisse (genommen/nicht genommen [taken/not taken])
- Vorhersage eines Sprungs:
  - Tabelle nachschlagen
    - Annahme: Sprungergebnis bleibt gleich
  - Nächster Befehl holen aus vorhergesagter Richtung
  - Wenn Vorhersage falsch, lösche (flush) Pipeline und invertiere Vorhersagebit



Nachteil 1-Bit-Sprungvorhersage

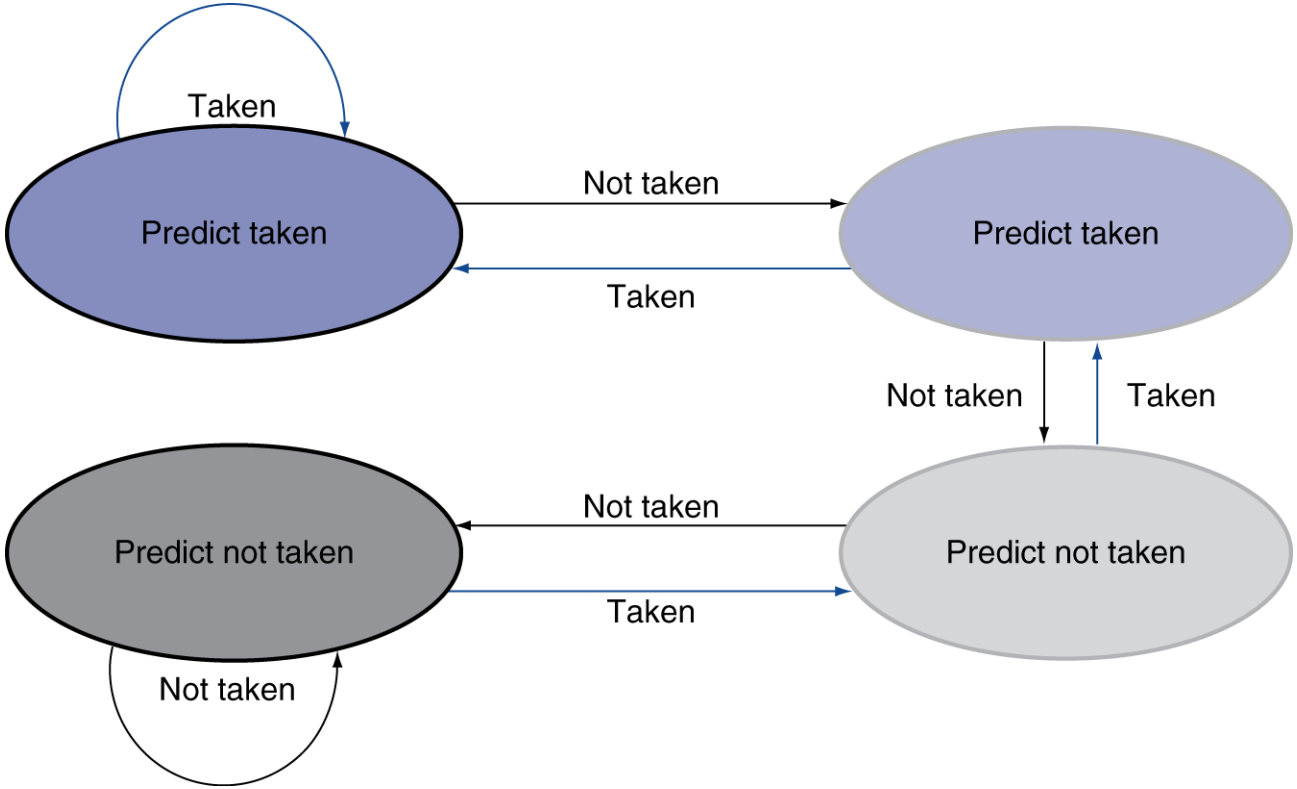
- Zwei falsche Vorhersagen pro Durchgang einer inneren Schliefe



- Fehlvorhersage (*taken*) bei letzte Iteration der innere Schleife
- Beim nächsten Eintreten in innere Schleife wiederum Fehlvorhersage (*not taken*)

2-Bit (bimodal) Prädiktor

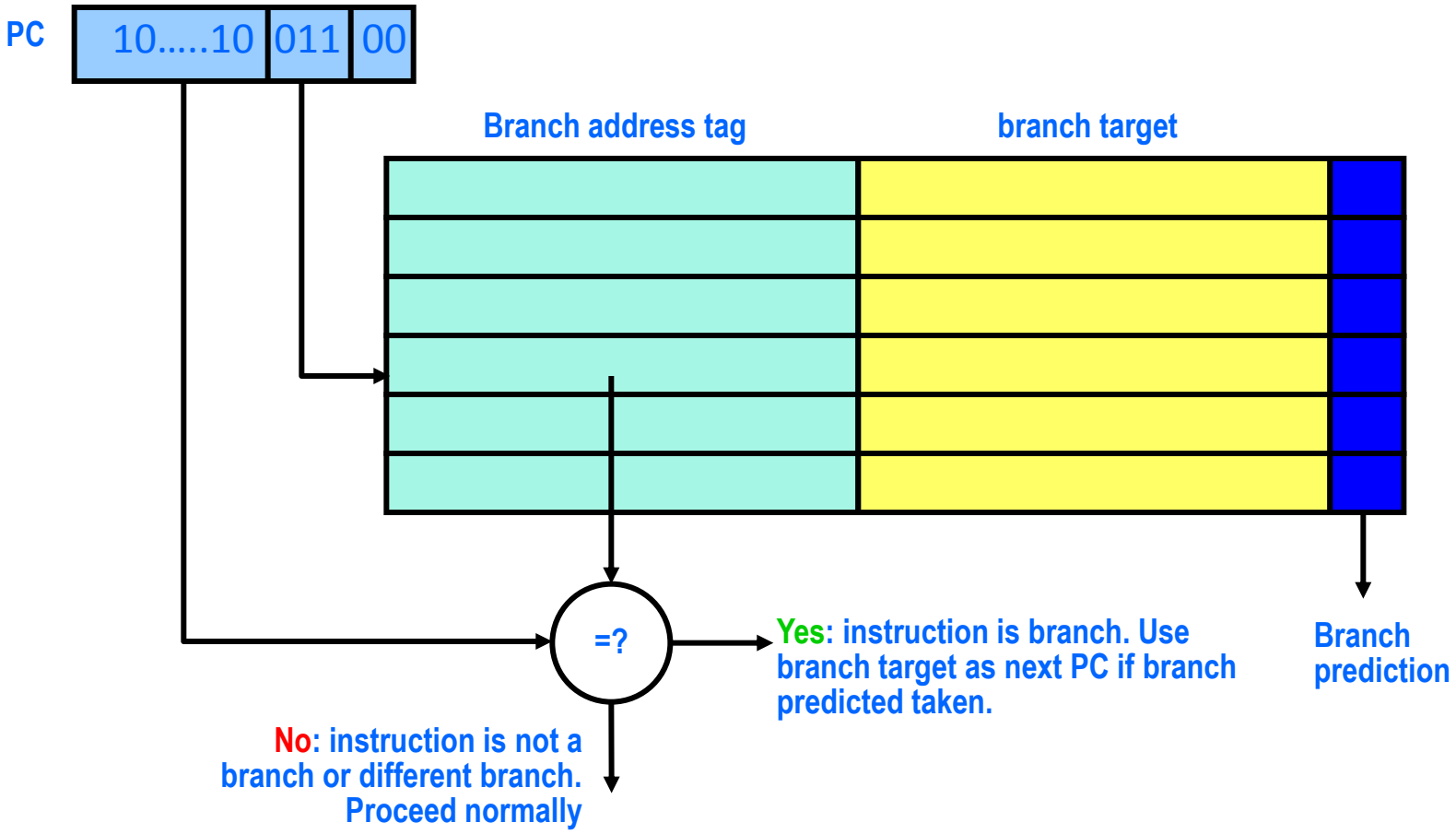
- Ändere Vorhersage erst, wenn 2 Fehlvorhersagen auftreten.



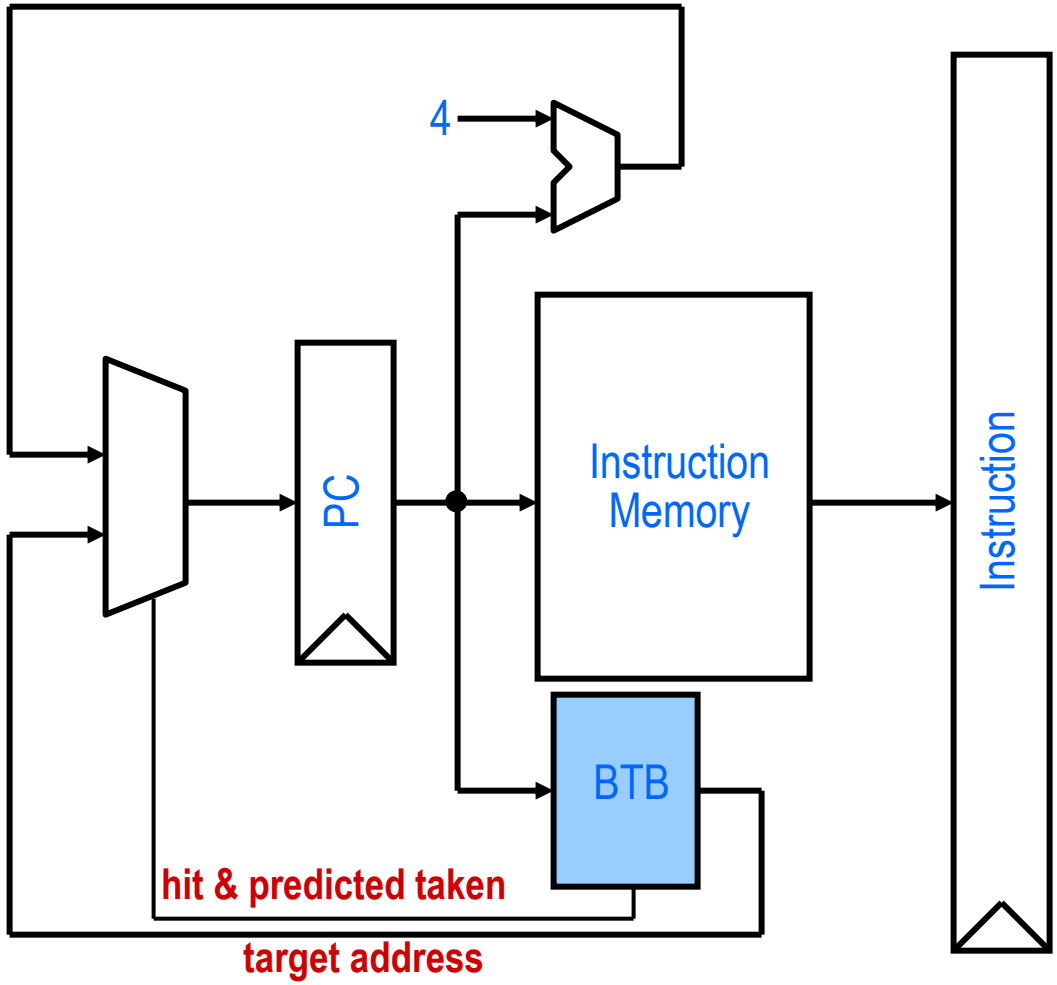
Sprungzielpuffer (*Branch Target Buffer*)

- Bisherige Sprung-Prädiktoren sagen nur Sprungrichtung voraus
  - Leistungssteigerung nur wenn Sprungzieladresse schon bekannt
- In 5-stufigen MIPS Pipeline Sprungrichtung und Zieladresse in ID-Stufe berechnet (+ branch delay slot)
- Um Sprungvorhersage zur IF-Stufe zu verlagern, brauchen Zieladresse und Vorhersage zur gleichen Zeit (vor Befehl geholt wurde)
- Sprungzielpuffer (*Branch Target Buffer* (BTB)): Sprungvorhersage **UND** Zieladresse
  - Sprung = Sprung im BTB?
    - könnte jeder Befehl sein (Befehl noch nicht entschlüsselt)
  - obere Teil des Befehlszählers = obere Teil der Befehlsadresse des Sprungs im BTB (*Tag*)?
    - BTB eine Art Cache (nächste Vorlesung)

Sprungzielpuffer (*Branch Target Buffer*)



BTB in MIPS-Pipeline



- Moderne Prozessoren sind sehr kompliziert
  - Intel Core i7:
    - 16 Pipeline Stufen
    - verarbeitet bis zu 4/6 Befehle gleichzeitig
      - gleichzeitig holen, entschlüsseln, ausführen, u.s.w. (*Superskalar*)
    - CPU bestimmt in welche Reihenfolge Befehle ausgeführt werden (*out-of-order execution*, dynamische Scheduling)
    - Register werden umbenennt (*register renaming*) um Parallelität auf Befehlsebene (*instruction-level parallelism*) zu erhöhen
    - 1 Kern kann 2 Programme/Fäden gleichzeitig bearbeiten (Hardwareseitiges Multithreading)
    - Sehr fortgeschrittene Sprung-Prädiktoren
- Diese und andere Merkmale moderner Prozessoren werden im Modul *Advanced Computer Architecture* tiefgehender diskutiert.

Fazit

- Pipelining erhöht den Durchsatz in dem die Ausführung von verschiedenen Befehlen überlappt wird.
- MIPS für Pipelining entworfen
  - Alle Befehle 32 Bit lang
  - Quellregister immer an gleicher Stelle
  - Sehr einfache Sprungbefehle
- Strukturkonflikte
  - repliziere Funktionelle Einheiten
- Datenabhängigkeiten/-konflikte
  - Forwarding zur frühzeitigen Bereitstellung der Operanden
  - Erkennen von Konflikten (Hazard Detection Unit)
  - Umordnen der Reihenfolge der Befehle (Compiler, out-of-order execution)
- Steuerkonflikte
  - Vorziehen der Sprungentscheidung +
  - nächster Befehl verwerfen (Pipeline Flush) oder Branch Delay Slot
- Nächste Vorlesung: Speicherhierarchie (Kapitel 7)

Fazit Eintakt- vs. Mehrzyklen- vs. Pipelined Prozessor

- Eintaktprozessor

`lw`					`add`				`beq`		
IF	ID	EX	MA	WB	IF	ID	EX	WB	IF	ID	EX

- Mehrzyklenprozessor

`lw`					`add`				`beq`		
IF	ID	EX	MA	WB	IF	ID	EX	WB	IF	ID	EX

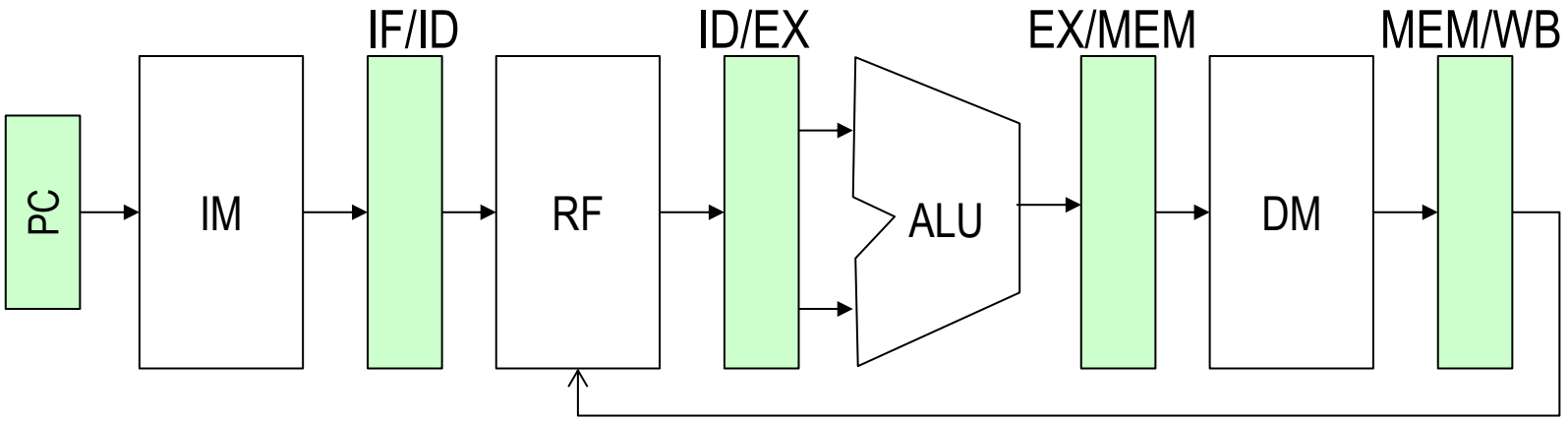
- Pipelined Prozessor

	cc 1	cc 2	cc3	cc 4	cc 5	cc 6	cc 7
`lw`	IF	ID	EX	MA	WB		
`add`		IF	ID	EX	MA	WB	
`beq`			IF	ID	EX	MA	WB

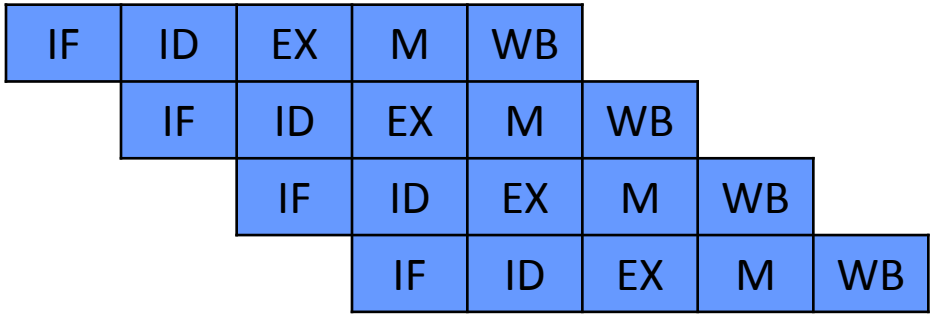
Back-up folien

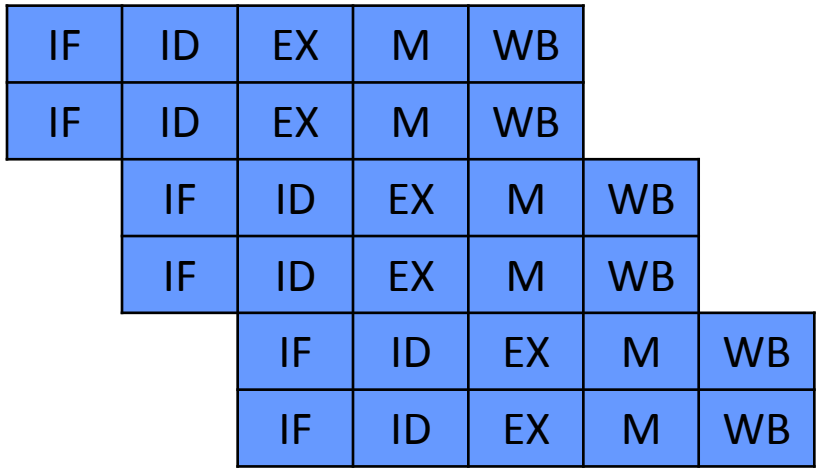
Wann Forwarding?

- Wenn (einer) der Quellregister des Befehls in der EX-Stufe mit dem Zielregister des Befehls in der MEM- oder WB-Stufe übereinstimmt
  - `EX/MEM.Rd==ID/EX.Rs`
  - `EX/MEM.Rd==ID/EX.Rt`
  - `MEM/WB.Rd==ID/EX.Rs`
  - `MEM/WB.Rd==ID/EX.Rt`
- + Befehl soll zum Registersatz schreiben (`RegWrite`) und Zielreg. nicht $0



- Moderne Hochleistungsprozessoren nutzen tiefere Pipelines.
  - 10 bis 20 Pipeline-Stufen sind nicht unüblich.
- Zusätzlich sind moderne Prozessoren meist superskalar.
  - In jedem Taktzyklus wird versucht mehrere Befehle zu laden und auszuführen.





→ Falsche Sprungvorhersagen sind noch problematischer

→ Verwenden der dynamischen Sprungvorhersage (dynamic branch prediction)