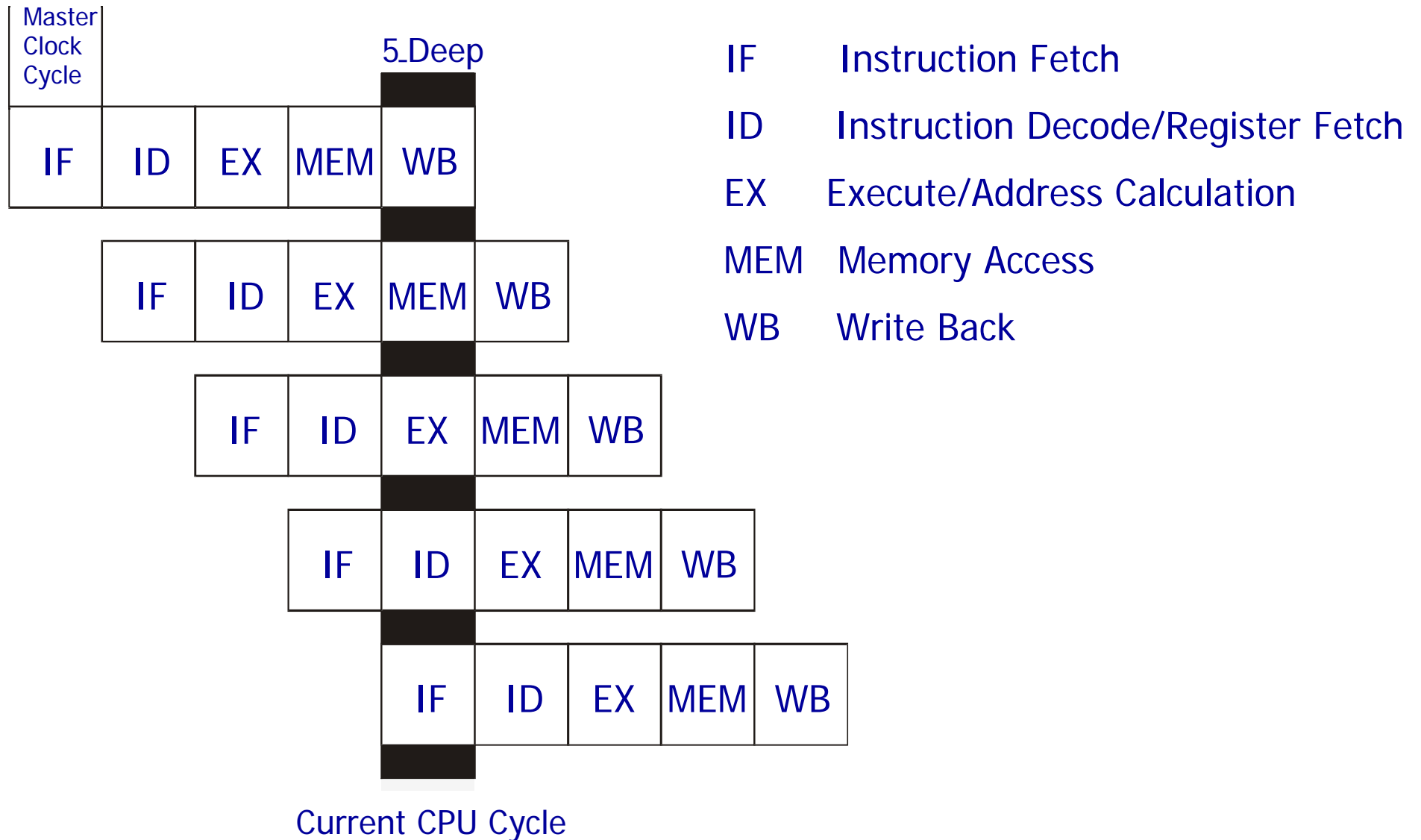

5. Übung

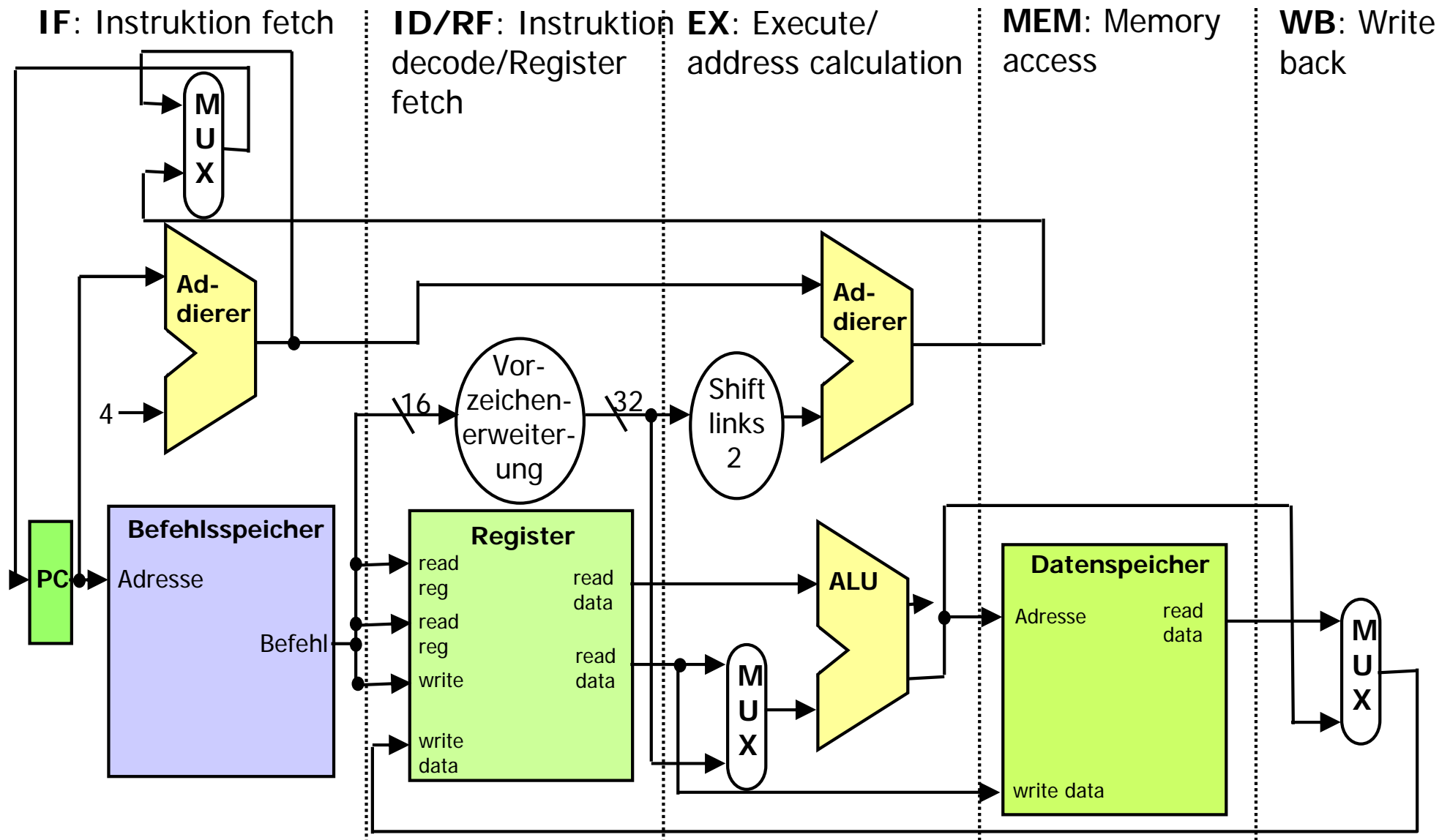
- **DLX-Pipeline**
- **Abhängigkeiten**
- **Konflikte und deren Lösungen**



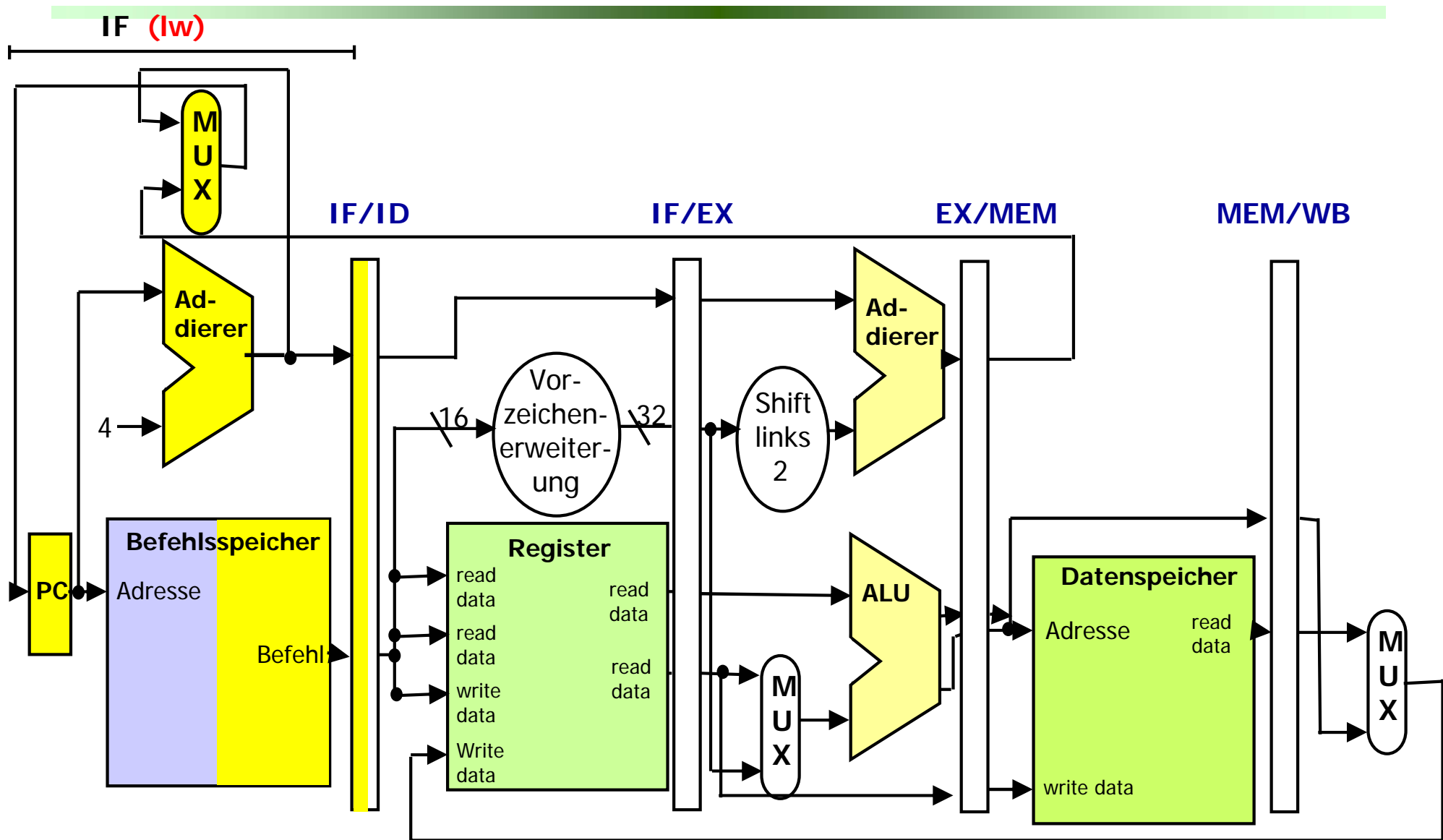
DLX Pipeline



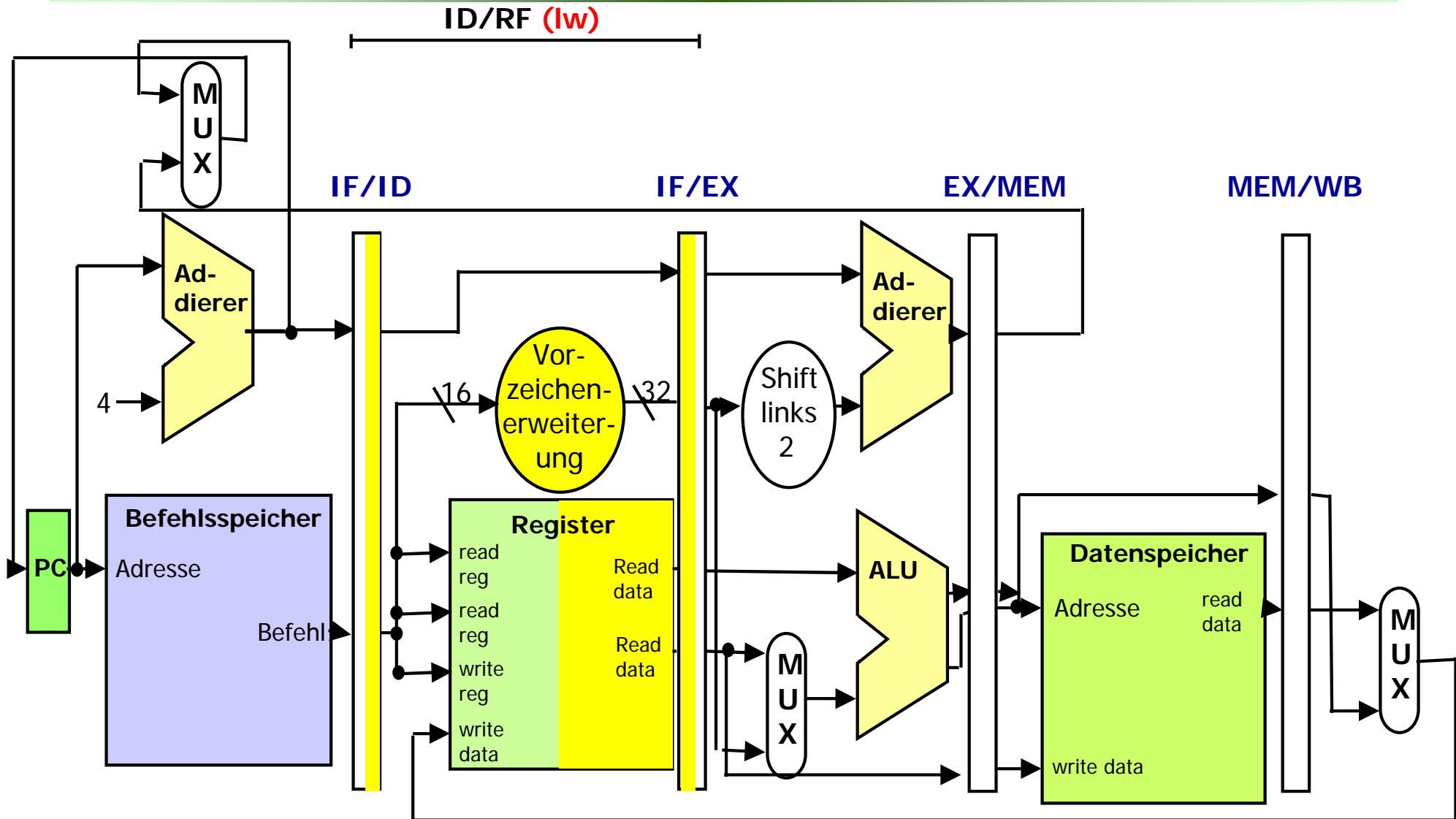
Pipelining in MIPS-Architektur



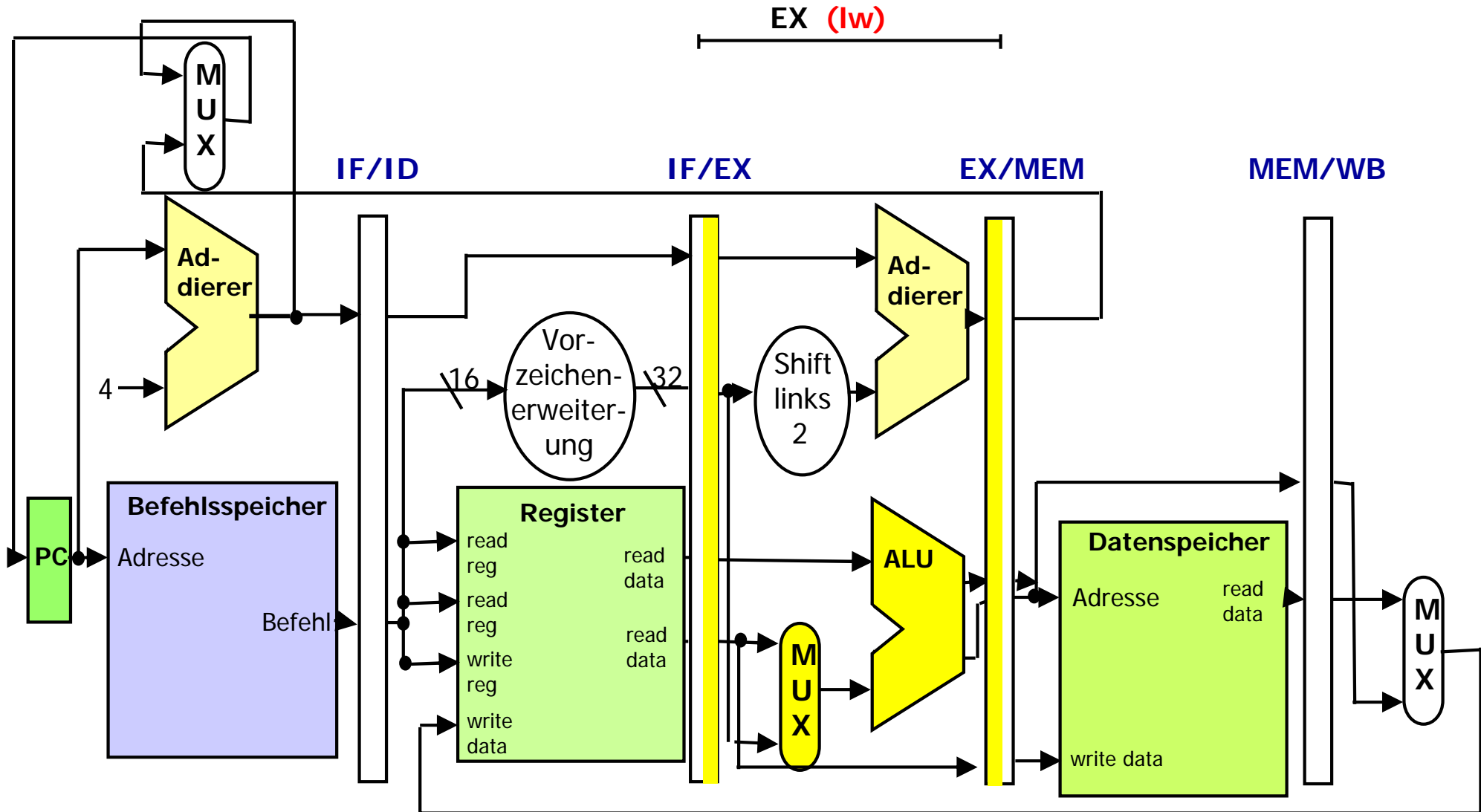
DLX Pipelinestufen



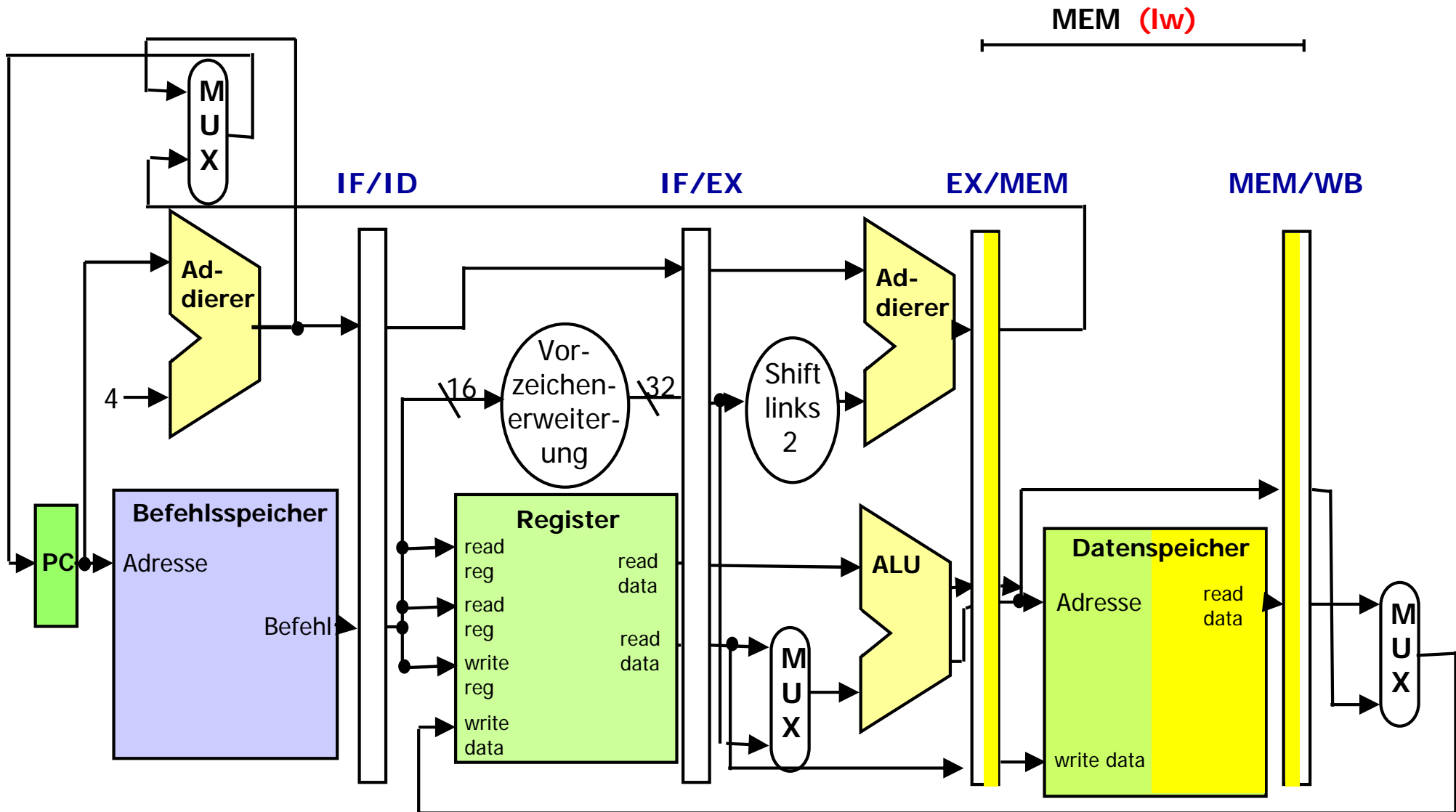
DLX Pipelinestufen



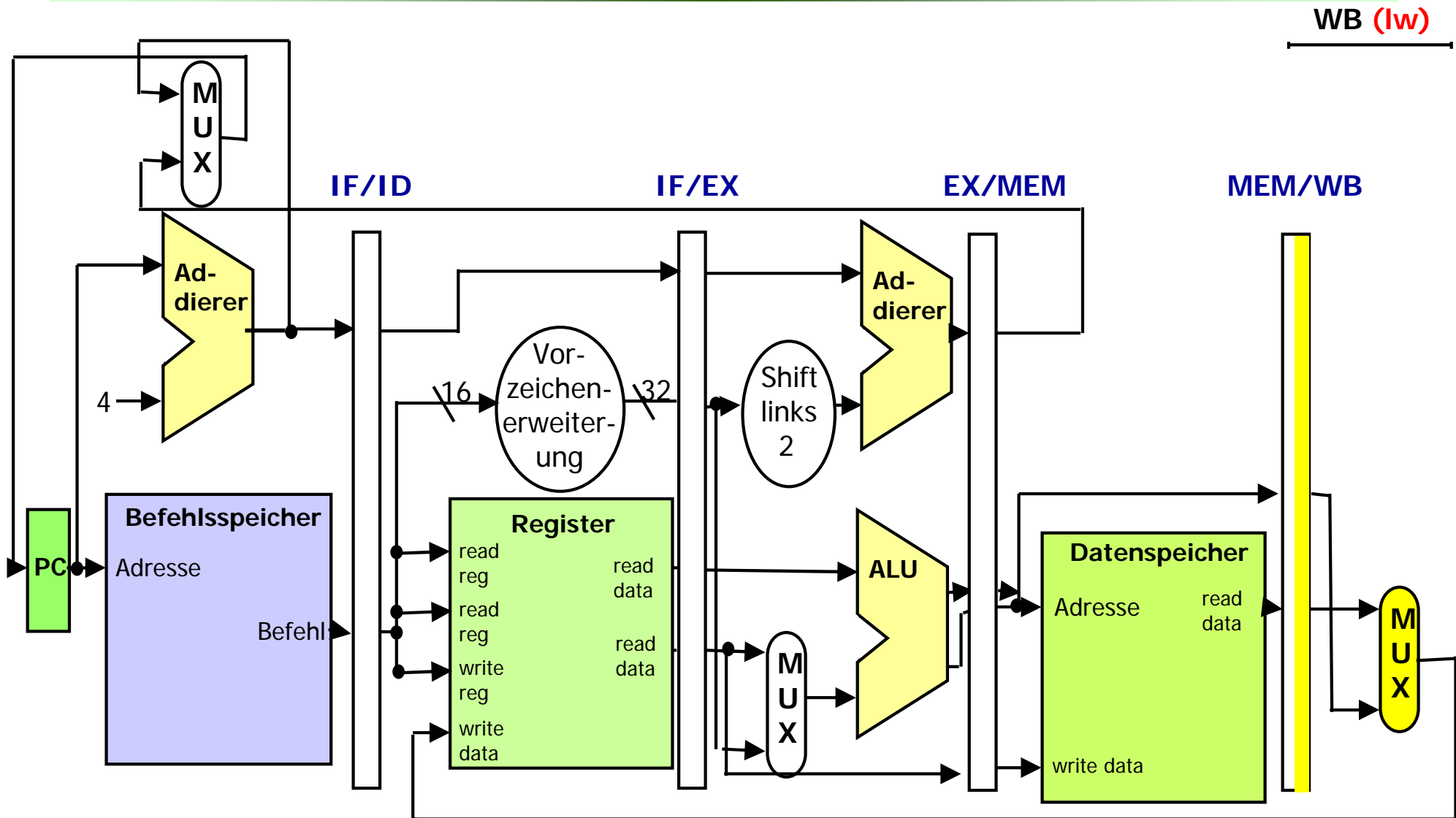
DLX Pipelinestufen



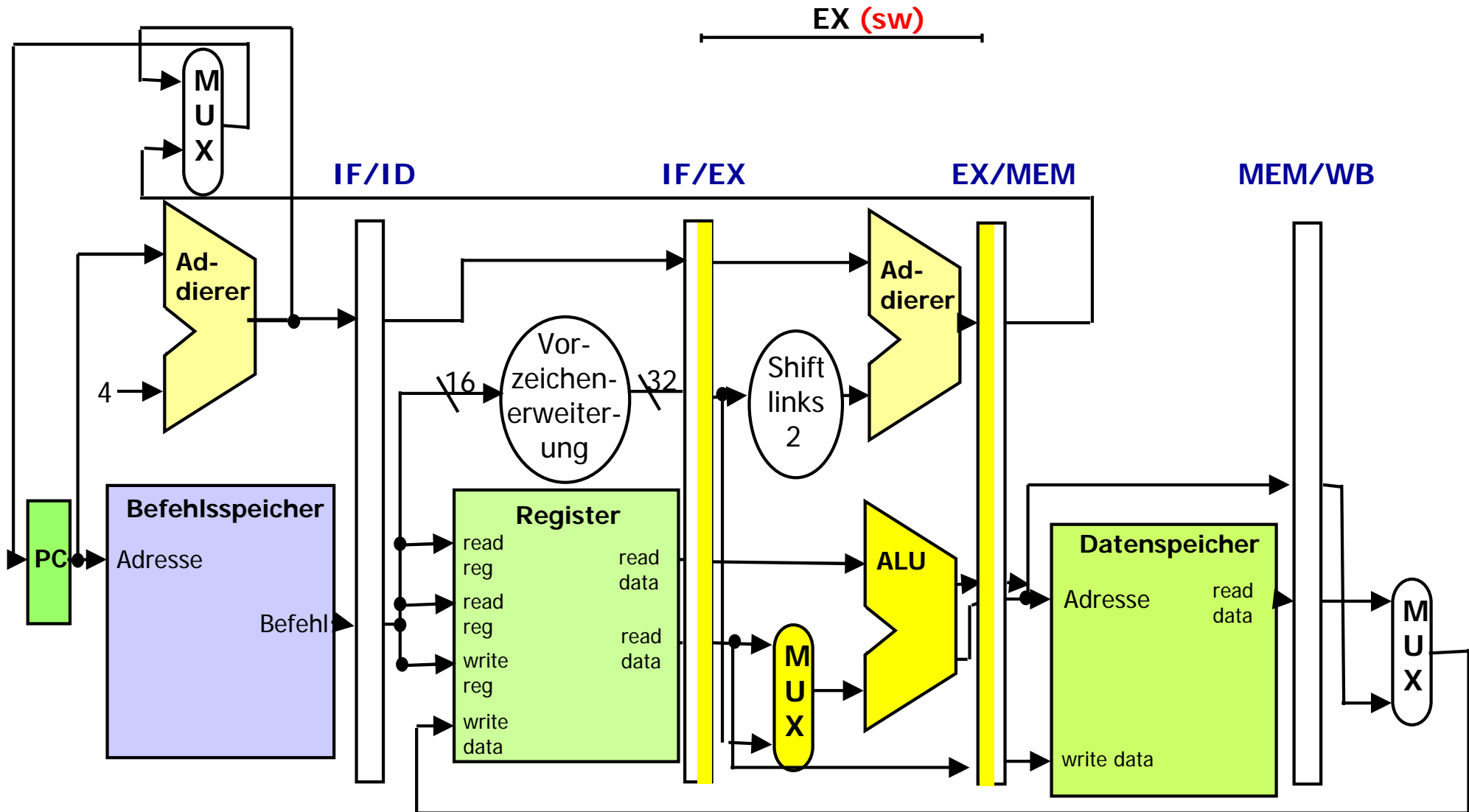
DLX Pipelinestufen



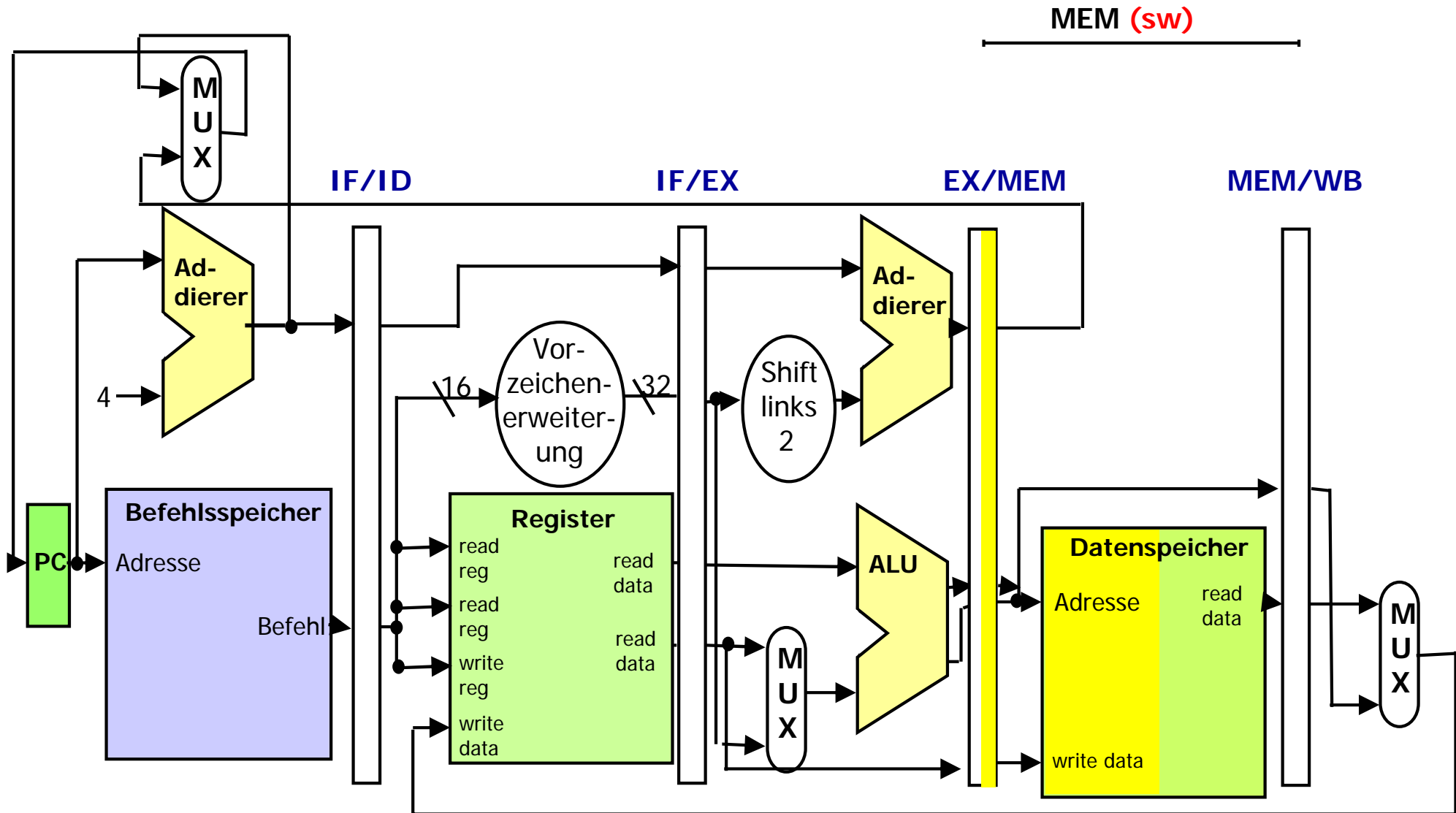
DLX Pipelinestufen



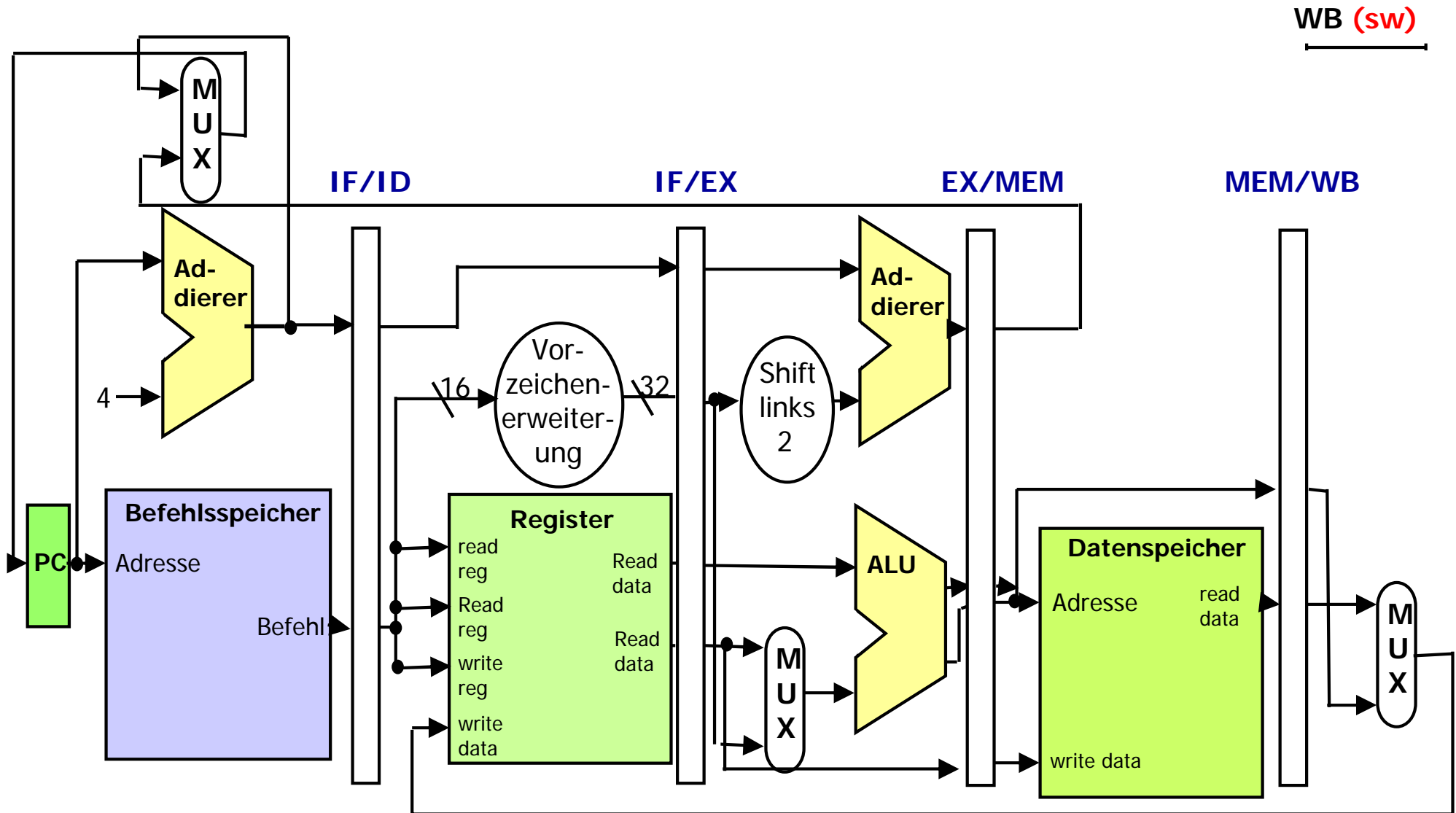
5.4. DLX Pipelinestufen



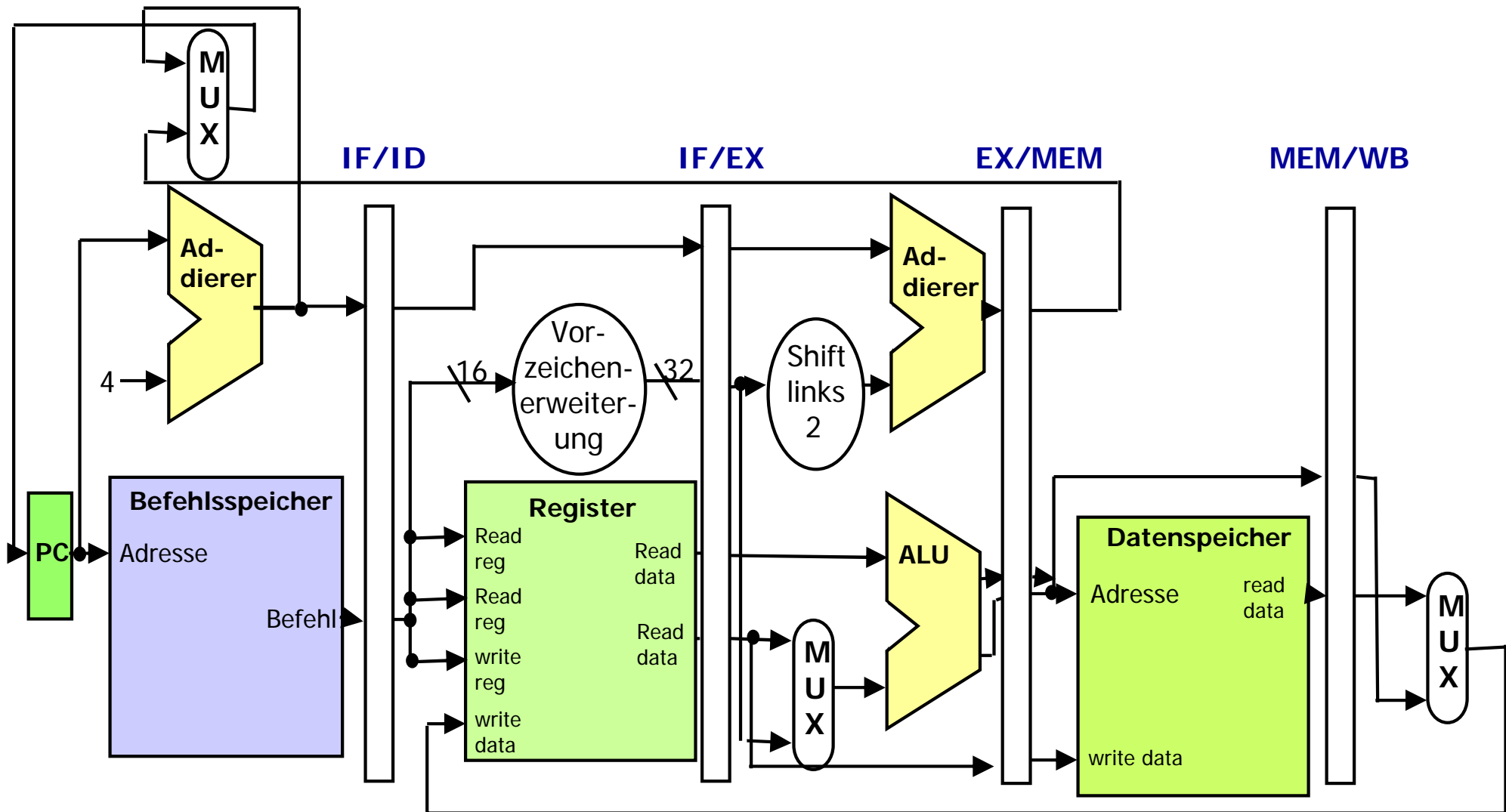
DLX Pipelinestufen



DLX Pipelinestufen

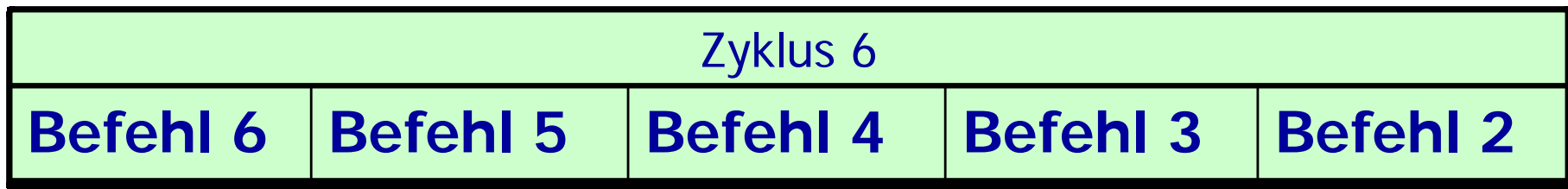
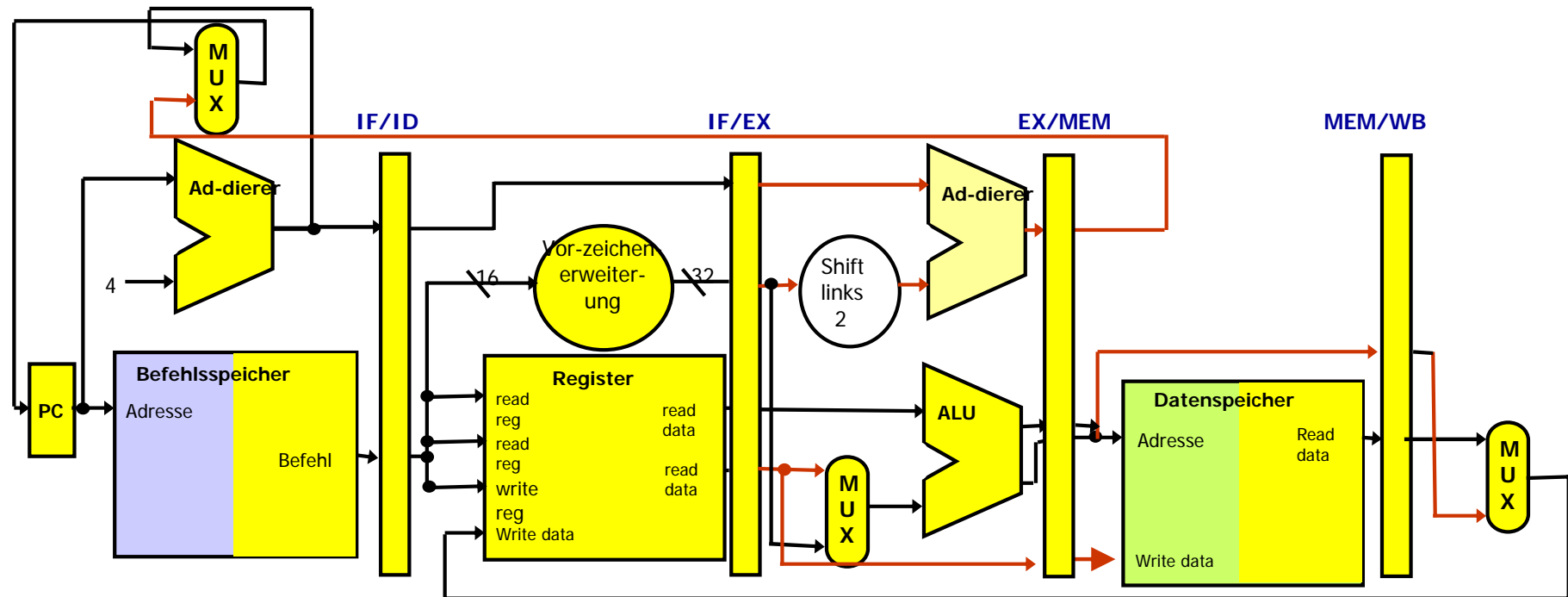


DLX Pipelinestufen



Pipeline Konflikte

- Bei einer gefüllten Pipeline wird pro Taktzyklus ein Befehl beendet.



Drei Arten von Pipeline-Konflikten

- **Datenkonflikte:** Treten auf, wenn ein Operand ist in der Pipeline (noch) nicht verfügbar.
 - Datenkonflikte werden durch Datenabhängigkeiten im Befehlsstrom erzeugt
- **Struktur- oder Ressourcenkonflikte:** Treten auf, wenn zwei Pipeline-Stufen dieselbe Ressource benötigen, auf diese aber nur einmal zugegriffen werden kann.
- **Steuerflusskonflikte** treten bei Programmsteuerbefehlen auf:
 - wenn in der IF-Stufe die Zieladresse des als nächstes auszuführenden Befehls noch nicht berechnet ist bzw.
 - wenn im Falle eines bedingten Sprunges noch nicht klar ist, ob überhaupt gesprungen wird.



Pipelinekonflikte

- Ressourcenkonflikte: DLX-Pipeline ist entsprechend angepasst, so dass keine Ressourcenkonflikte auftreten
- Datenkonflikte:
 - RAW: Befehl $i+1$ liest einen Wert bevor Befehl i geschrieben hat
 - WAW: Kann nicht auftreten, da die DLX-Pipeline nur in WB schreibt
 - WAR: Alle Lesezugriffe erfolgen in der ID/RF und alle Schreibzugriffe in WB
- Steuerkonflikte:
 - Pipeline muss wegen branch geleert und neu gefüllt werden



Aufgabe 1

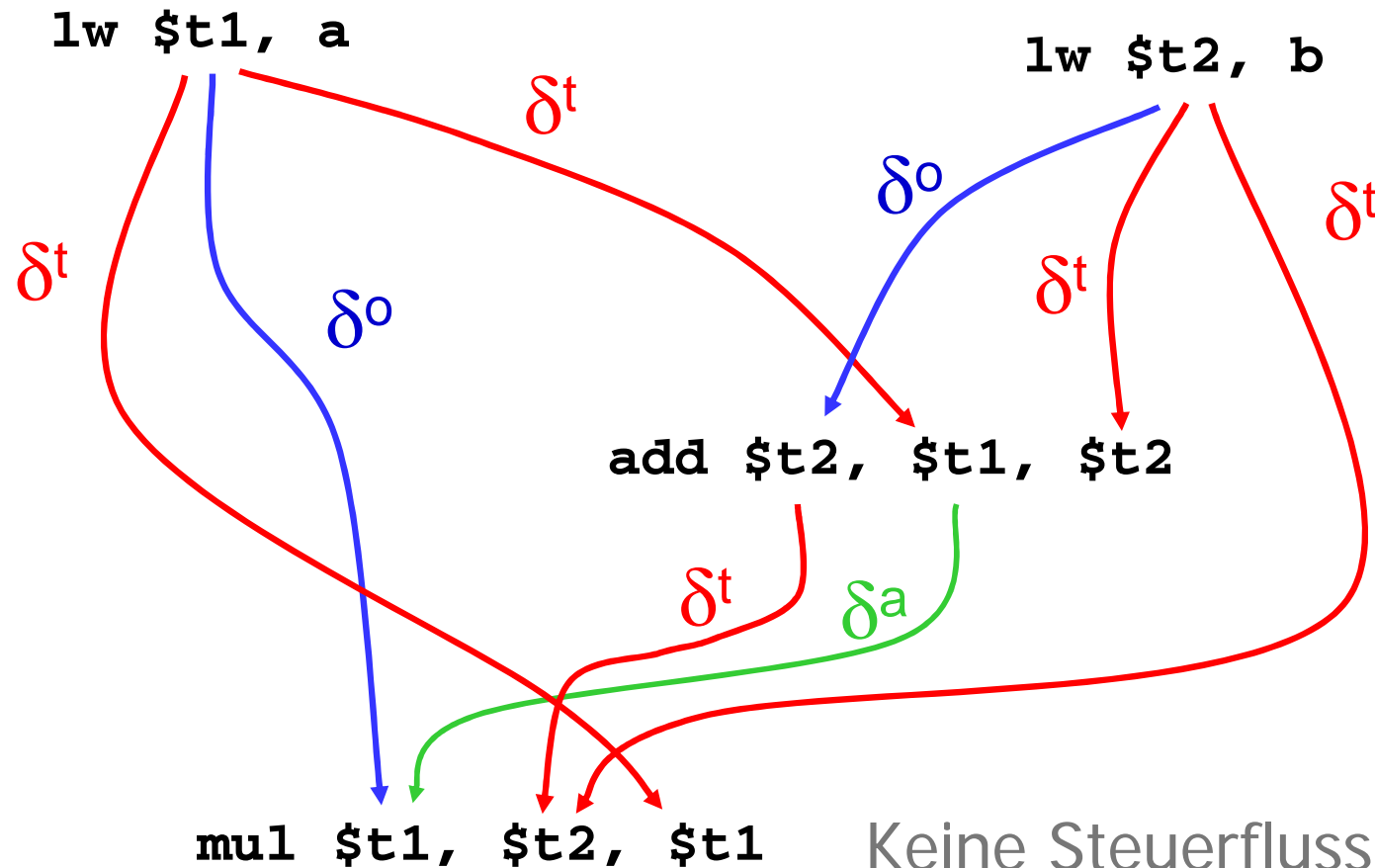
Betrachten Sie das folgende sequentielle Programmstück, in dem die Konstanten **a** und **b** Speicheradressen darstellen:

```
S1:  lw    $t1, a           ; $t1 := [a]
S2:  lw    $t2, b           ; $t2 := [b]
S3:  add   $t2, $t1, $t2
S4:  mul   $t1, $t2, $t1
```



Aufgabe 1.1

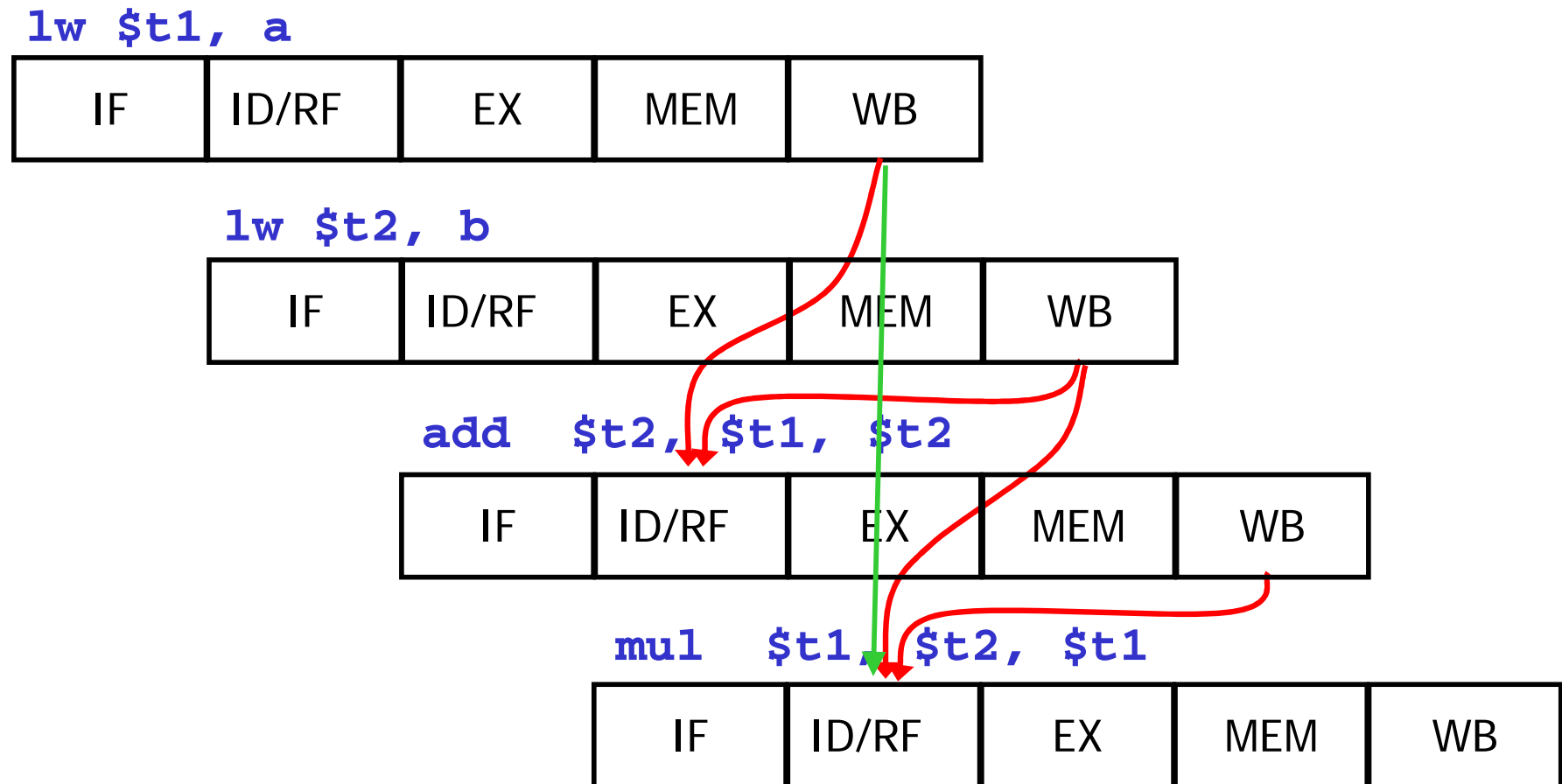
- Bestimmen Sie alle Daten- und Steuerflussabhängigkeiten in diesem Programmstück



Keine Steuerflussabhängigkeit

Aufgabe 1.2

2. Wieviele Pipelinekonflikte treten auf?



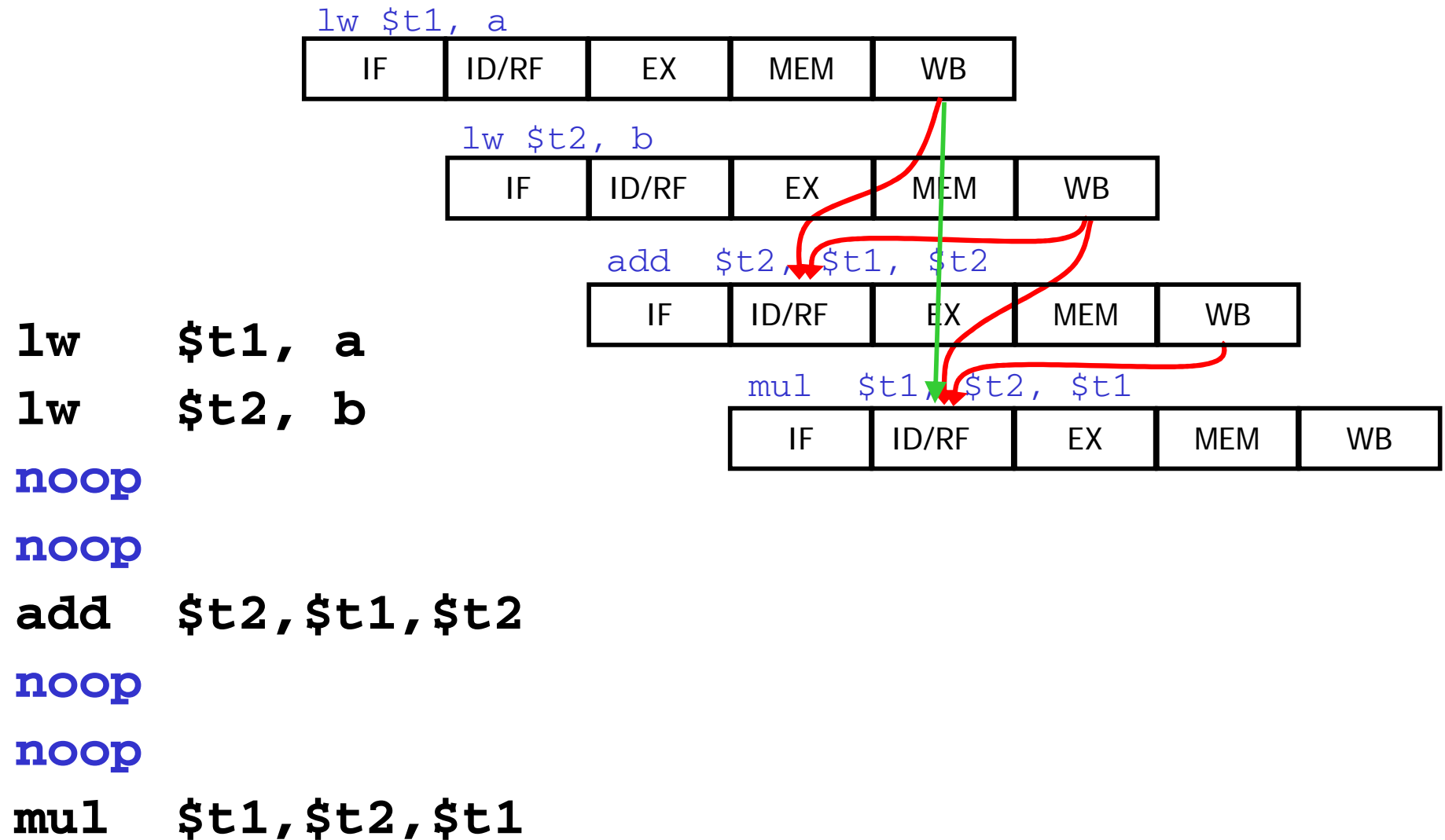
Aufgabe 1

3. Die auftretenden Pipelinekonflikte werden von der Hardware nicht erkannt und müssen vom Compiler durch Einfügen von NOOP-Befehlen behandelt werden.

Ergänzen Sie das Programmstück so, dass auch die Pipelinekonflikte berücksichtigt werden



Aufgabe 1.3

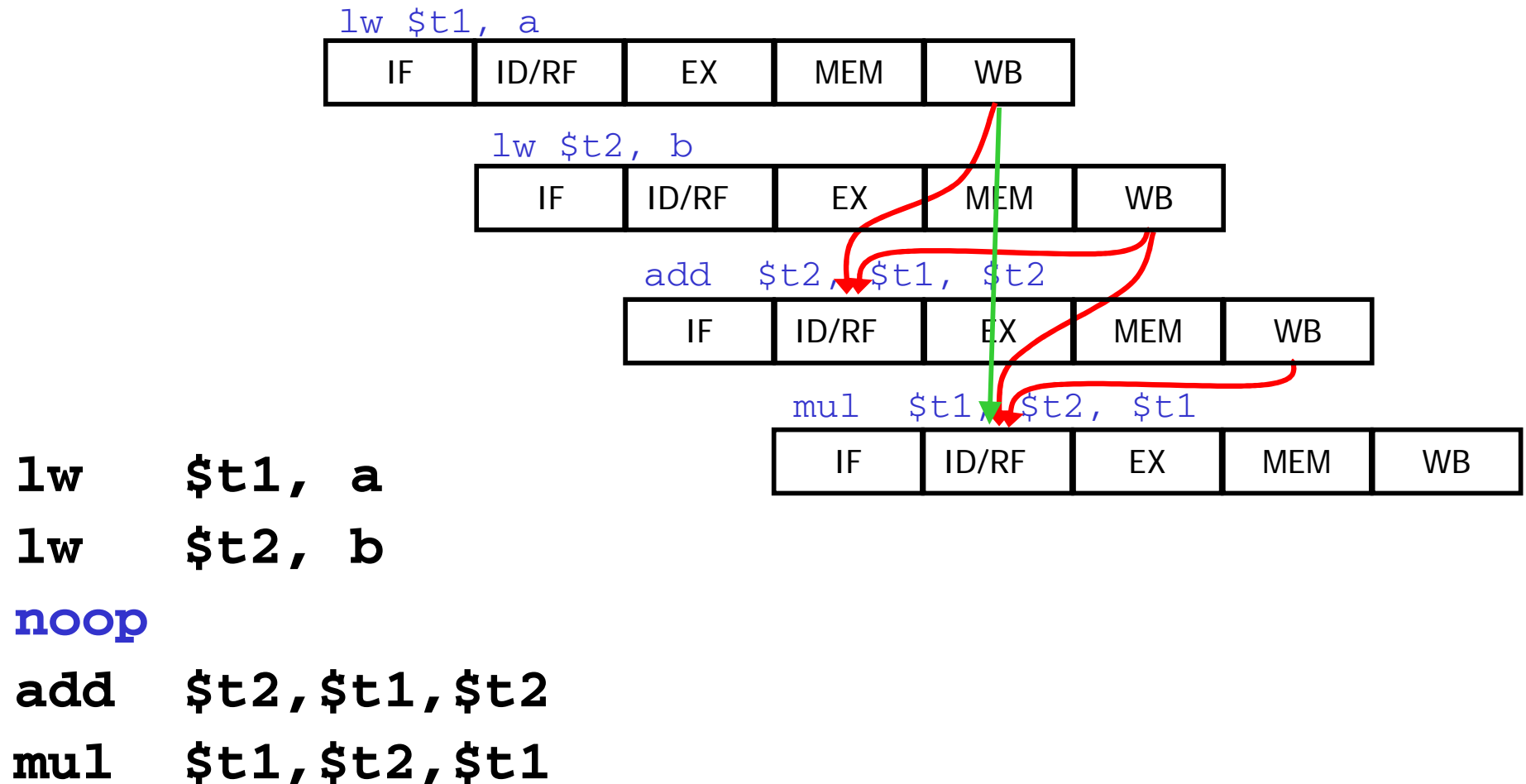


Aufgabe 1

4. Welche der NOOP-Befehle sind noch notwendig, falls die auftretenden Pipelinekonflikte von der Hardware erkannt werden und durch *Load Forwarding* und *Result Forwarding* behandelt werden?



Aufgabe 1.4



Aufgabe 2

Das folgende MIPS-Programmstück soll auf einem Prozessor mit DLX-Pipeline ausgeführt werden.

```
S1:      lw $t1, 1000($zero)
S2:      lw $t2, 1004($zero)
S3:      add $t3, $t2, $t1
S4:      addi $t1, $t2, 8
S5:      subi $t4, $zero, 2
S6:      and $t5, $t3, $t2
S7:      sw $t4, 1000($zero)
S8:      sw $t5, 1004($zero)
S9:      sw $t1, 1008($zero)
```

- Bestimmen Sie alle Datenabhängigkeiten im Programmstück.



Aufgabe 2.1

S1: lw \$t1, 1000(\$zero)

S2: lw \$t2, 1004(\$zero)

S3: add \$t3, \$t2, \$t1

S4: addi \$t1, \$t2, 8

S5: subi \$t4, \$zero, 2

S6: and \$t5, \$t3, \$t2

S7: sw \$t4, 1000(\$zero)

S8: sw \$t5, 1004(\$zero)

S9: sw \$t1, 1008(\$zero)



Lösung 2.1

- Echte Abhängigkeiten (True Dependence)

$S1 \rightarrow S3$ (R1)

$S1 \rightarrow S9$ (R1)

$S2 \rightarrow S3$ (R2)

$S2 \rightarrow S4$ (R2)

$S2 \rightarrow S6$ (R2)

$S3 \rightarrow S6$ (R3)

$S4 \rightarrow S9$ (R1)

$S5 \rightarrow S7$ (R4)

$S6 \rightarrow S8$ (R5)

- Gegenabhängigkeiten (Anti-Dependence): $S3 \rightarrow S4$ (R1)
- Ausgabe-Abhängigkeiten (Output Dependence): $S1 \rightarrow S4$ (R1)



Aufgabe 2.2

- Nehmen Sie an, dass keine Forwarding-Techniken implementiert sind und die auftretenden Pipelinekonflikte durch Einfügen von NOP (No Operation) Befehlen behoben werden müssen.

Ergänzen Sie das obige Programm, so dass es korrekte Ergebnisse liefert. Sie dürfen dabei die Reihenfolge der Befehle nicht ändern und so wenig NOP-Befehle wie möglich einfügen.



Lösung 2.2

S1: lw \$t1, 1000(\$zero)

S2: lw \$t2, 1004(\$zero)

S3: add \$t3, \$t2, \$t1

S4: addi \$t1, \$t2, 8

S5: subi \$t4, \$zero, 2

S6: and \$t5, \$t3, \$t2

S7: sw \$t4, 1000(\$zero)

S8: sw \$t5, 1004(\$zero)

S9: sw \$t1, 1008(\$zero)

{ noop
noop

{ noop

•Echte Abhängigkeiten (True Dependence)

S1 → S3 (R1)

S1 → S9 (R1)

S2 → S3 (R2)

S2 → S4 (R2)

S2 → S6 (R2)

S3 → S6 (R3)

S4 → S9 (R1)

S5 → S7 (R4)

S6 → S8 (R5)



Aufgabe 3

Gegeben sei das folgende DLX-Programm

S1: add \$t1, \$zero, \$zero

S2: lw \$t3, 1500(\$zero)

S3: loop: lw \$t4, 5000(\$t1)

S4: add \$t5, \$t4, \$t3

S5: sw \$t5, 400(\$t1)

S6: addi \$t1, \$t1, 4

S7: subi \$t2, \$t1, 400

S8: bnez \$t2, loop

S9: end: srli \$t1, \$t1, 2

S10: sw \$t1, 2000(\$zero)

RAW nach load

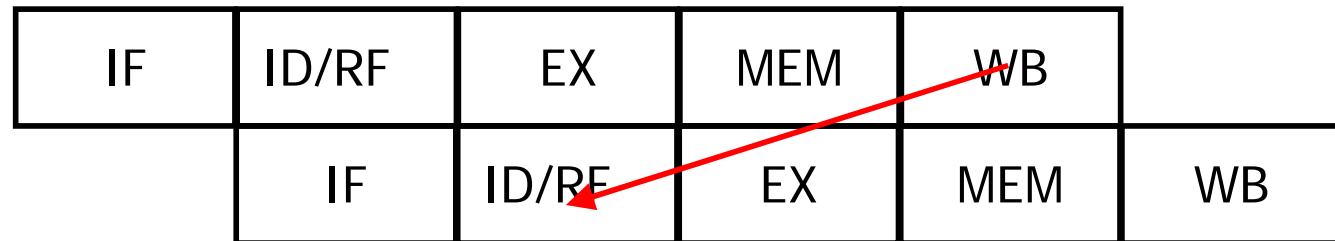
Steuerflussabhängigkeit
wegen branch



RAW nach load

S3: loop: lw \$t4, 5000(\$t1)

S4: add \$t5, \$t4, \$t3



- lw schreibt \$t4 nachdem add \$t4 bereits ausgelesen hat.
- Forwarding: EX-Phase von add muss um einen Takt verschoben werden



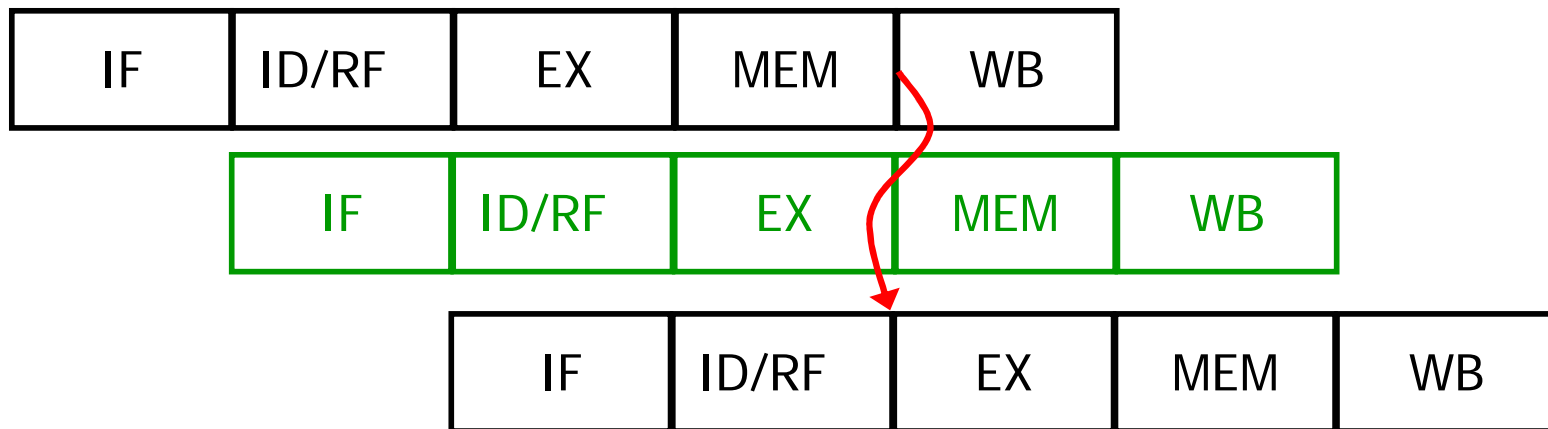
RAW nach load

S3: loop: lw \$t4, 5000(\$t1)

 noop

S4: add \$t5, \$t4, \$t3

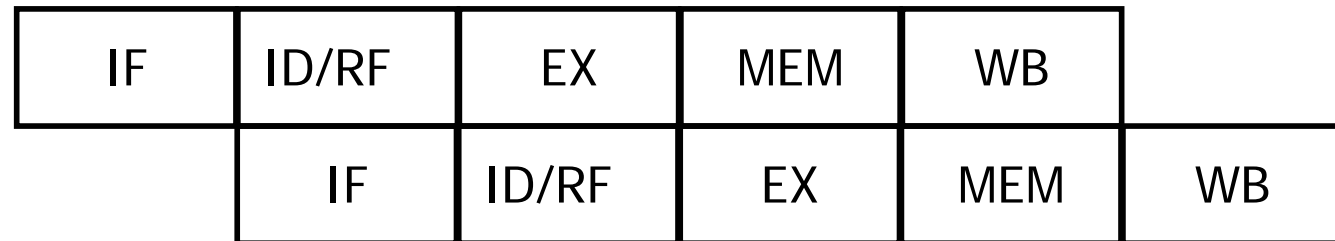
Kann durch
unabhängigen Befehl
ersetzt werden



- lw schreibt \$t4 nachdem add \$t4 bereits ausgelesen hat.
- Forwarding: EX-Phase von add muss um einen Takt verschoben werden

Steuerflussabhängigkeit nach branch

```
S8:          bnez $t2, loop
S9:    end:   srli $t1, $t1, 2
```



Annahme:

Dekodierung, Berechnung der Sprungzieladresse und
Rückschreibung des PCs in der ID-Stufe

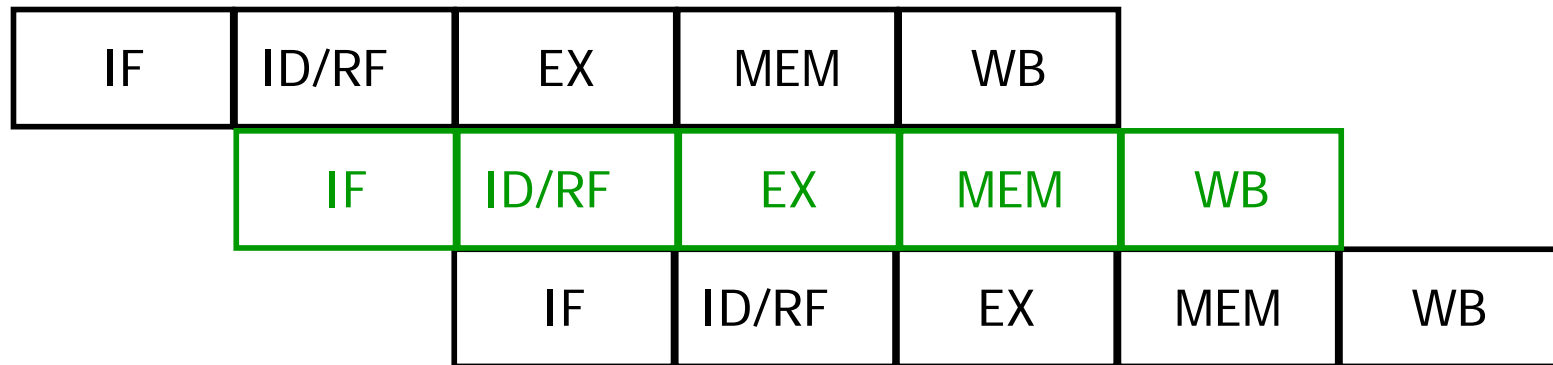
- Bei branch wird in ID der Kontrollfluss geändert
- Nächster Befehl ist bereits in der Pipeline
- Immer ein noop unabhängig vom Folgebefehl



Steuerflussabhängigkeit nach branch

```
S8:          bnez $t2, loop
             noop
S9:  end:    srli $t1, $t1, 2
```

Kann durch
unabhängigen Befehl
ersetzt werden



- Bei branch wird in ID der Kontrollfluss geändert
- Nächster Befehl ist bereits in der Pipeline
- Immer ein noop unabhängig vom Folgebefehl

Code mit noops

```
S1:          add $t1, $zero, $zero
S2:          lw  $t3, 1500($zero)
S3:  loop:   lw  $t4, 5000($t1)
           nop
S4:          add $t5, $t4, $t3
S5:          sw  $t5, 400($t1)
S6:          addi $t1, $t1, 4
S7:          subi $t2, $t1, 400
S8:          bnez $t2, loop
           nop
S9:  end:    srli $t1, $t1, 2
S10:         sw  $t1, 2000($zero)
```

8 Taktzyklen pro
Schleifendurchlauf



Vermeidung von noops durch Umorganisieren der Schleife

S1: add \$t1, \$zero, \$zero

S2: lw \$t3, 1500(\$zero)

S3: loop: lw \$t4, 5000(\$t1)

noop

S4: add \$t5, \$t4, \$t3

S5: sw \$t5, 400(\$t1)

S6: addi \$t1, \$t1, 4

S7: subi \$t2, \$t1, 400

S8: bnez \$t2, loop

noop

S9: end: srli \$t1, \$t1, 2

S10: sw \$t1, 2000(\$zero)



Vermeidung von noops durch Umorganisieren der Schleife

S1: add \$t1, \$zero, \$zero

S2: lw \$t3, 1500(\$zero)

S3: loop: lw \$t4, 5000(\$t1)

S6: addi \$t1, \$t1, 4

S4: add \$t5, \$t4, \$t3

S5: sw \$t5, 396(\$t1)

S7: subi \$t2, \$t1, 400

S8: bnez \$t2, loop

noop

S9: end: srli \$t1, \$t1, 2

S10: sw \$t1, 2000(\$zero)



Vermeidung von noops durch Umorganisieren der Schleife

S1: add \$t1, \$zero, \$zero

S2: lw \$t3, 1500(\$zero)

S3: loop: lw \$t4, 5000(\$t1)

S6: addi \$t1, \$t1, 4

S4: add \$t5, \$t4, \$t3

S5: sw \$t5, 396(\$t1)

S7: subi \$t2, \$t1, 400

S8: bnez \$t2, loop

noop

S9: end: srli \$t1, \$t1, 2

S10: sw \$t1, 2000(\$zero)



Vermeidung von noops durch Umorganisieren der Schleife

S1: add \$t1, \$zero, \$zero

S2: lw \$t3, 1500(\$zero)

S3: loop: lw \$t4, 5000(\$t1)

S6: addi \$t1, \$t1, 4

S4: add \$t5, \$t4, \$t3

S7: subi \$t2, \$t1, 400

S8: bnez \$t2, loop

S5: sw \$t5, 396(\$t1)

S9: end: srli \$t1, \$t1, 2

S10: sw \$t1, 2000(\$zero)



Vermeidung von noops durch Umorganisieren der Schleife

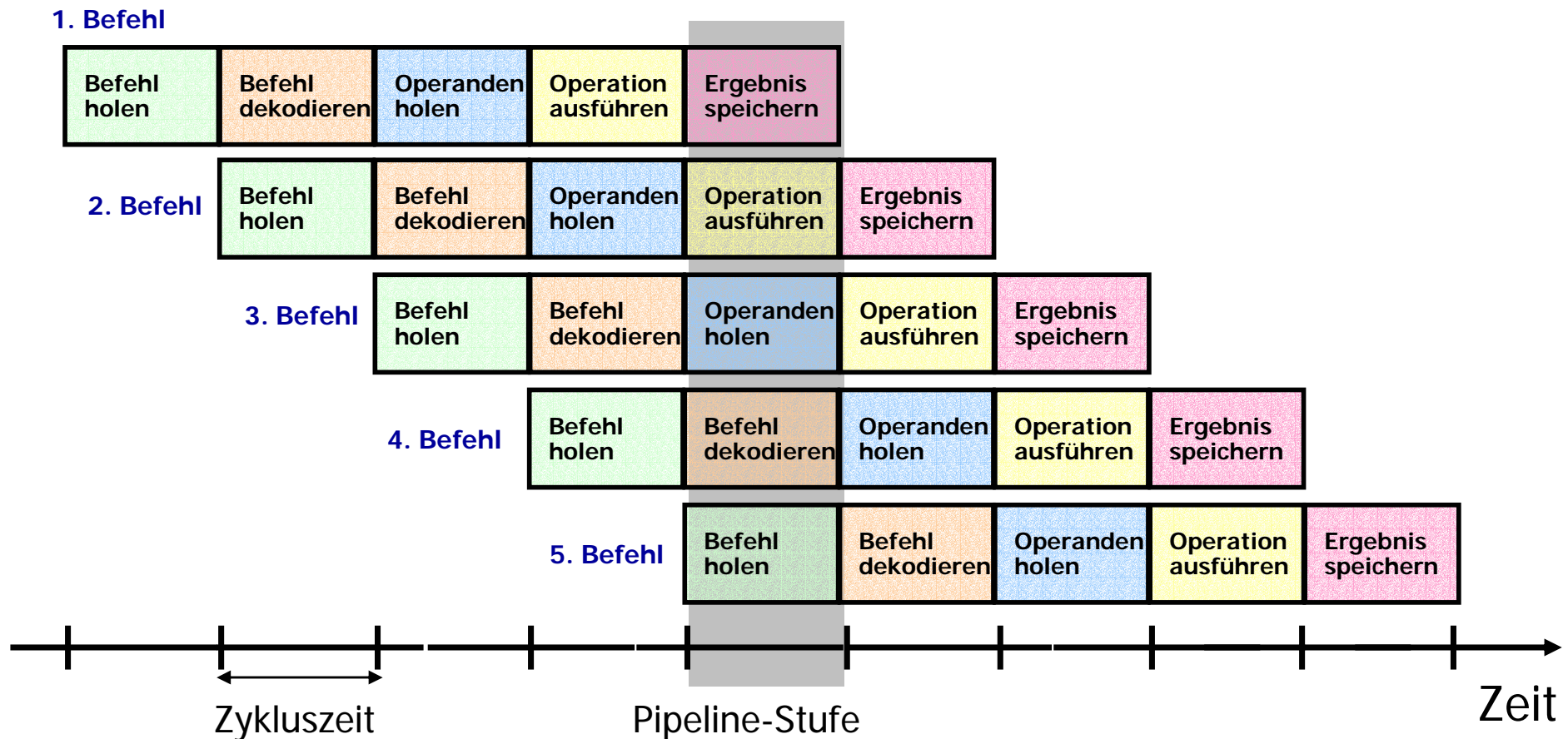
```
S1:          add $t1, $zero, $zero
S2:          lw  $t3, 1500($zero)
S3:  loop:    lw  $t4, 5000($t1)
S6:          addi $t1, $t1, 4
S4:          add $t5, $t4, $t3
S7:          subi $t2, $t1, 400
S8:          bnez $t2, loop
S5:          sw  $t5, 396($t1)
S9:  end:     srli $t1, $t1, 2
S10:         sw  $t1, 2000($zero)
```

Alle noops wurden
beseitigt

6 Taktzyklen pro
Schleifendurchlauf

Aufgabe 4

Gegeben sei eine 5-stufige Pipeline:



Aufgabe 4

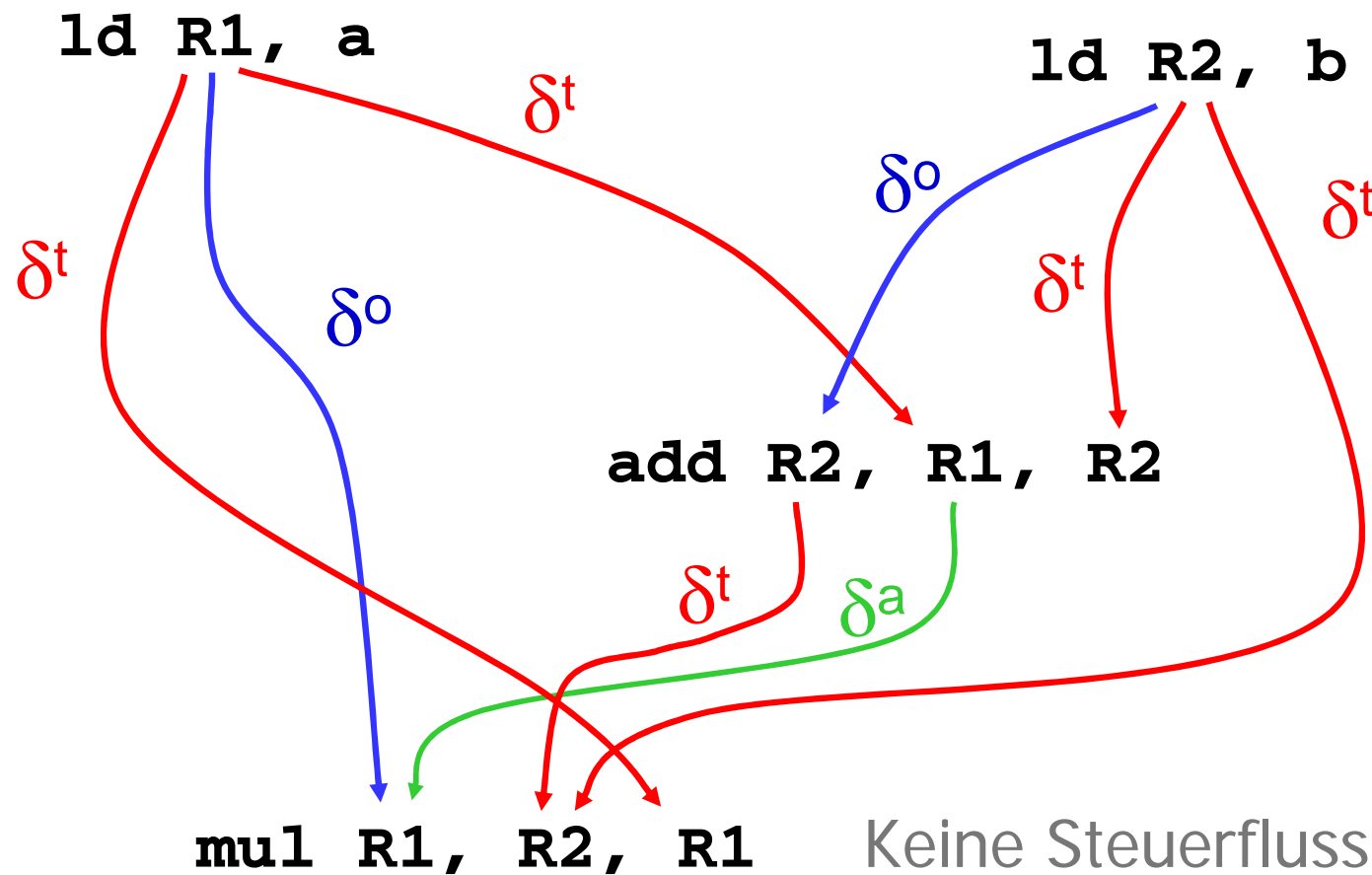
Betrachten Sie das folgende sequentielle Programmstück, in dem die Konstanten **a** und **b** Speicheradressen darstellen:

```
m1:  ld    R1, a      ; R1 := [a]
m2:  ld    R2, b      ; R2 := [b]
m3:  add   R2, R1, R2  ; R2 := R1+R2
m4:  mul   R1, R2, R1  ; R1 := R1*R2
```

In der Pipeline-Struktur ist ein Schreibvorgang in das entsprechende Zielregister erst am Ende der Ergebnis-Speichern-Phase abgeschlossen.

Aufgabe 4.1

1. Bestimmen Sie alle Daten- und Steuerflussabhängigkeiten in diesem Programmstück



Aufgabe 4.2

2. Wieviele Pipelinekonflikte treten auf?

ld R1, a



ld R2, b



add R2, R1, R2



mul R1, R2, R1



Aufgabe 4.3

3. Die auftretenden Pipelinekonflikte werden von der Hardware nicht erkannt und müssen vom Compiler durch Einfügen von NOOP-Befehlen behandelt werden.

Ergänzen Sie das Programmstück so, dass auch die Pipelinekonflikte berücksichtigt werden



Aufgabe 4.3

ld R1, a

ld R2, b

noop

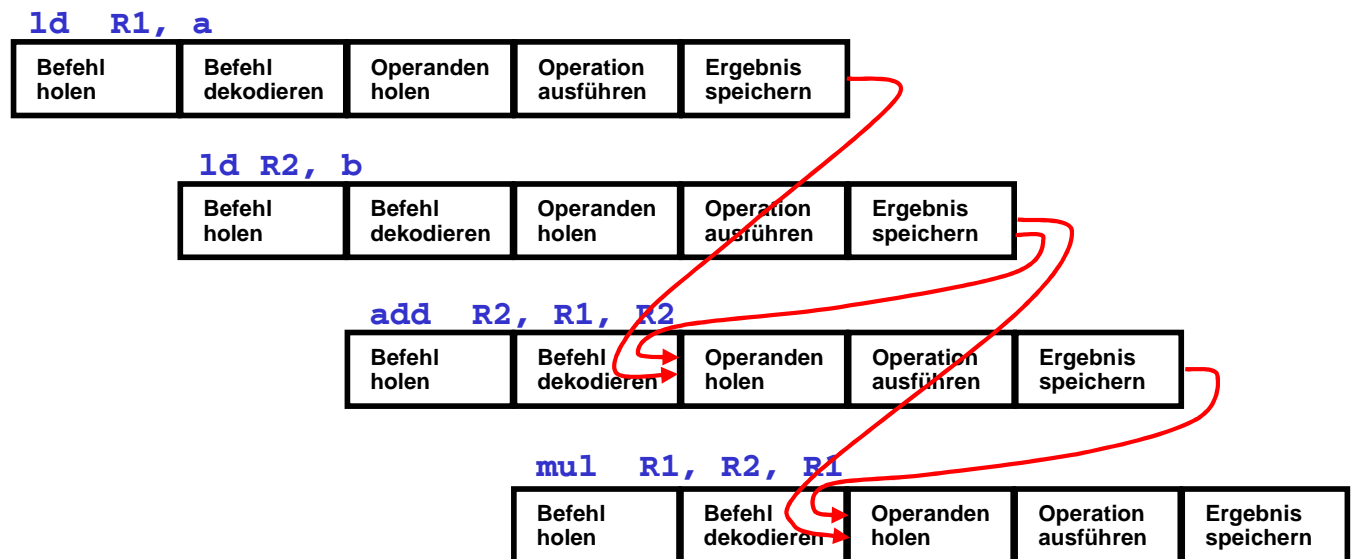
noop

add R2, R1, R2

noop

noop

mul R1, R2, R1



Aufgabe 4.4

4. Welche der NOOP-Befehle sind noch notwendig, falls die auftretenden Pipelinekonflikte von der Hardware erkannt werden und durch *Load Forwarding* und *Result Forwarding* behandelt werden?



Aufgabe 4.4

ld R1, a



ld R2, b



add R2, R1, R2



mul R1, R2, R1



Aufgabe 4.4

Keine !!!

ld R1, a



ld R2, b



add R2, R1, R2



mul R1, R2, R1



Aufgabe 5

Beim Pipeline-Prozessor ist die Kontrolle der Befehlspipeline vollständig dem Compiler übertragen. Die einzelnen Befehle werden in einer fünfstufigen Befehlspipeline (Befehl holen, Befehl dekodieren, Operanden holen, Operation ausführen und Ergebnis speichern) verarbeitet. **Erst am Ende der Ergebnis-speichern-Phase ist ein Schreibvorgang in das entsprechende Zielregister abgeschlossen.**

Betrachten Sie das folgende sequentielle Programmstück:

```
m1:  add  R1,R1,R1  ; R1 := R1+R1
m2:  add  R2,R1,R1  ; R2 := R1+R1
m3:  add  R2,R1,R2  ; R2 := R1+R2
```



Aufgabe 5.1

```
add    R1, R1, R1
add    R2, R1, R1
add    R2, R1, R2
```

1. Welchen Wert enthält das Register R2 nach Abarbeitung dieser Befehlsfolge, wenn R1 mit 4 und R2 mit 7 initialisiert ist?

add R1, R1, R1

F	D: add	R: 4, 4	E: 4 + 4	W: 8 → R1
---	--------	---------	----------	-----------

add R2, R1, R1

F	D: add	R: 4, 4	E: 4 + 4	W: 8 → R2
---	--------	---------	----------	-----------

add R2, R1, R2

F	D: add	R: 4, 7	E: 4 + 7	W: 11 → R2
---	--------	---------	----------	------------

R2 = 11



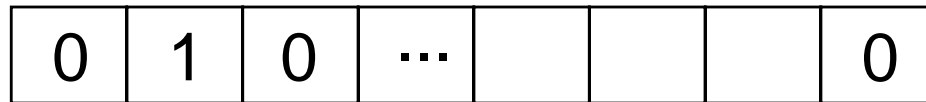
Aufgabe 5.2

2. Wieviele Takte benötigt das Programm bis zur vollständigen Leerung *der Befehlspipeline*, wenn *zusätzlich* Scoreboarding und *Result Forwarding* eingesetzt werden?

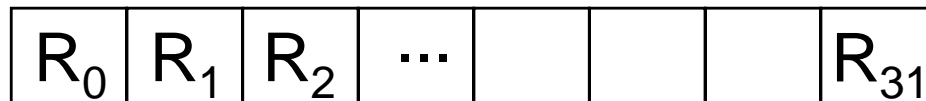


Scoreboarding-Technik

- Dient zum Erkennen von Datenabhängigkeiten.
- Jedem allgemeinen Register (und Fließkommaregister) wird ein Bit in einem **Scoreboard Register** zugeordnet.



Scoreboard Register
für 32 Register



- Ein Bit im Scoreboard Register wird nach der Decodierung gesetzt, wenn das entsprechende Register als Ziel einer Operation dient.
- Das Bit bleibt gesetzt, bis das Ergebnis in das Register geschrieben wird; danach wird es zurückgesetzt.



Scoreboarding-Technik

- Durch Prüfung der Scoreboard-Bits kann für jeden Befehl ein durch Datenabhängigkeit drohender Pipelinekonflikt erkannt werden.
- Behandlung des Konflikts durch die Verzögerung der Ausführung des Befehls, dessen Operandenregister ein gesetztes Scoreboard-Bit besitzt, bis zum Rücksetzen des Bits.

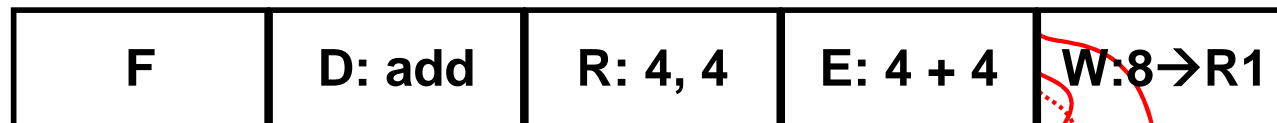
Scoreboarding ist eine Technik, mit der Datenabhängigkeiten durch die Hardware erkannt und im einfachsten Fall durch Verzögerungen behandelt werden können.



Aufgabe 5.2

2. Wieviele Takte benötigt das Programm bis zur vollständigen Leerung der Befehlspipeline, wenn zusätzlich Scoreboarding und Result Forwarding eingesetzt werden?

add R1, R1, R1



add R2, R1, R1



add R2, R1, R2



7 Takte



Aufgabe 6

Gegeben sei eine Pipeline-Struktur, bei der die absoluten Sprungbefehle (Assemblerbefehl: **ba**) mit einem Verzögerungszeitschlitz (*Delay-Slot*) ausgeführt werden.

Das folgende kleine Programmstück besteht aus fünf Assemblerbefehlen, die mit den Labels **m1**, **m2**, ..., **m5** markiert sind.

m1 :	add	R2 , R2 , R2
m2 :	ba	m1
m3 :	ba	m4
m4 :	add	R1 , R2 , R2
m5 :	add	R1 , R1 , R1

Aufgabe 6

Geben Sie die Ausführungsreihenfolge der Befehle bei Ausführung des Programmstücks an. Die Befehle sollen durch die zugehörigen Labels abgekürzt werden, d.h. es ist eine Folge der Form **m_i**, **m_j**, **m_k**, ... anzugeben

```
m1 :    add    R2 , R2 , R2  
m2 :    ba     m1  
m3 :    ba     m4  
m4 :    add    R1 , R2 , R2  
m5 :    add    R1 , R1 , R1
```

Ausführungsreihenfolge: **m1** **m2** **m3** **m1** **m4** **m5**



Aufgabe 6

Die ersten beiden Befehle werden sequentiell ausgeführt.

→ m1 – m2. Da absolute Sprünge einen Delay-Slot besitzen, werden sie um einen Takt verzögert, d. h. der Befehl hinter dem Sprungbefehl wird auch noch ausgeführt.

Deshalb ist der Sprungbefehl m3 bereits in der Pipeline, bevor der Sprungbefehl m2 ausgeführt wird.

→ m2 – m3.

Sobald der Sprungbefehl m2 ausgeführt ist, wird der erste Befehl an der Zieladresse, also m1, in die Pipeline geladen.

→ m3 – m1.

Dann wird der Sprungbefehl m3 ausgeführt, so dass als nächster Befehl m4 in die Pipeline geladen wird.

→ m1 – m4. Da jetzt kein Sprungbefehl mehr ausgeführt wird, folgt der nächste Befehl nach m4, also m5.

→ m4 – m5.

