Übung 4

Karlsruher Institut für Technologie

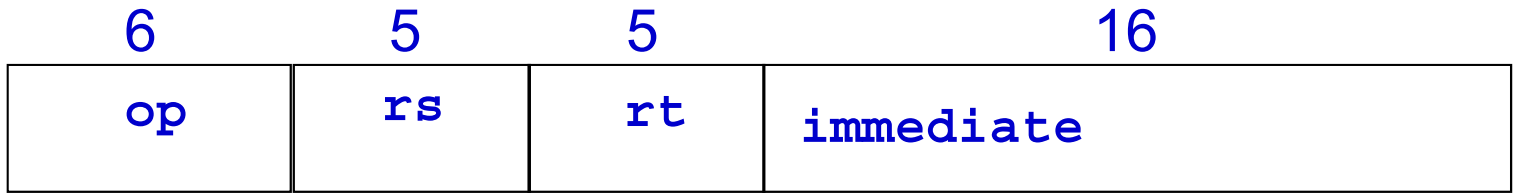
Pipelining

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

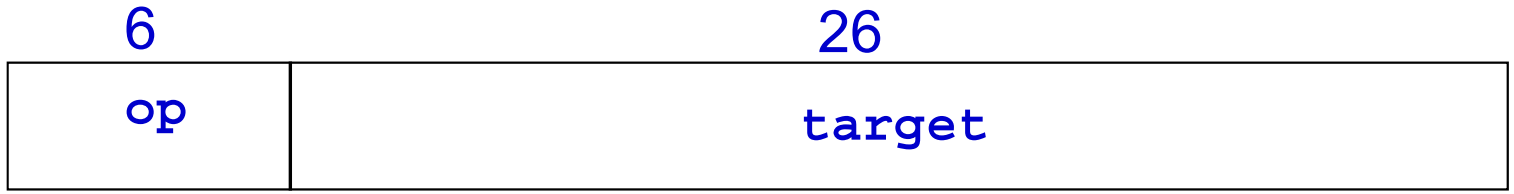
MIPS-Befehlsformate

Der MIPS-Prozessor hat ausschließlich Befehle fester Länge (32-Bit). Die Befehle werden in Typ I, J und R unterteilt:

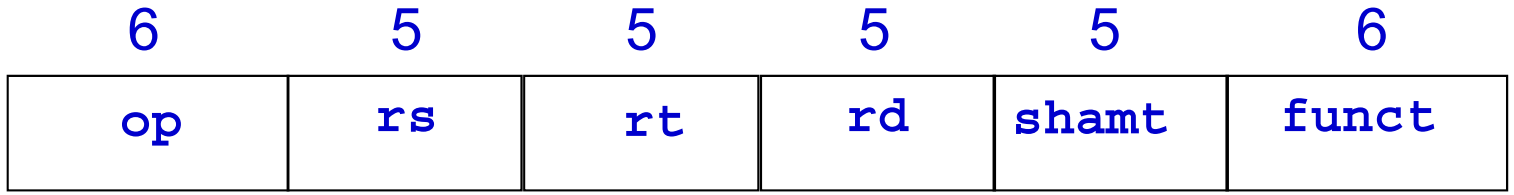
Typ I



Typ J



Typ R

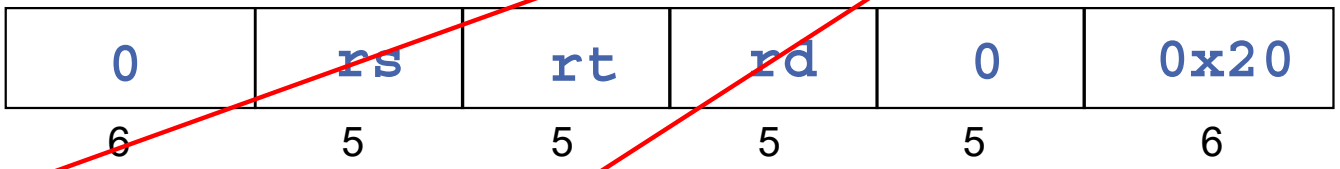


Welches Register wird gelesen bzw. beschrieben?

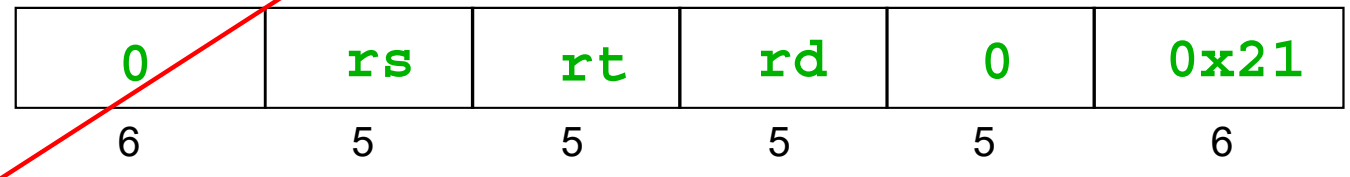
Befehlsformate (z. B. bei der Addition):

Zielregister

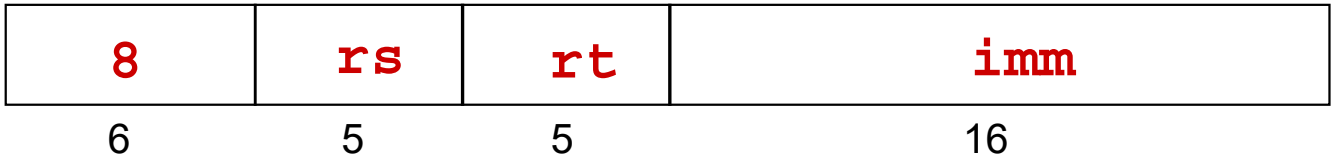
```
add rd,rs,rt
```




```
addu rd,rs,rt
```



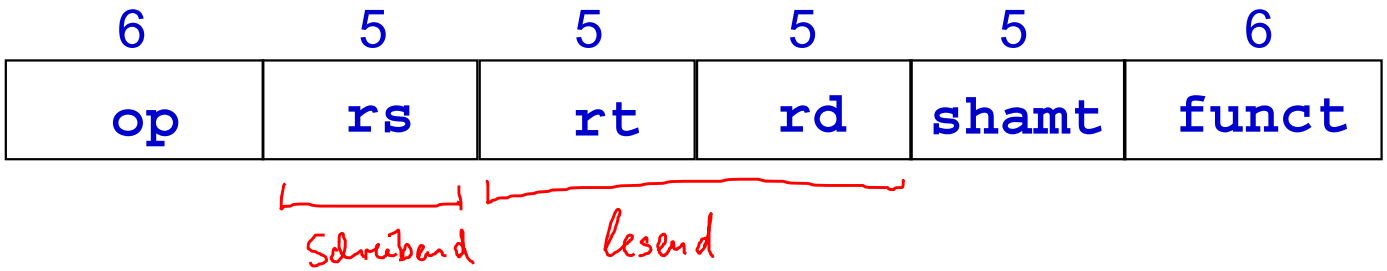

```
addi rt,rs,imm
```



Befehlsabarbeitung und Datenpfade in MIPS

- **Befehle vom R-Typ:** `opcode` r_s, r_t, r_d
  - Arithmetisch logische Befehle: `add, sub, and, or`
  - Vergleichsbefehle: `slt`

Typ R



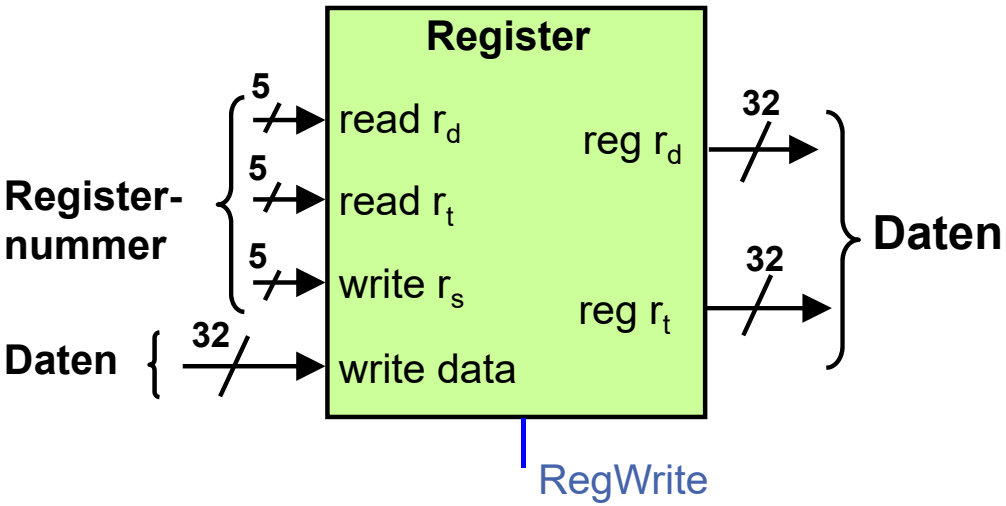
- Befehle haben 3 Operanden
- Operanden stehen in Registern
- Lesezugriff auf beiden Operanden-Register und
- ein Schreibzugriff auf das Zielregister

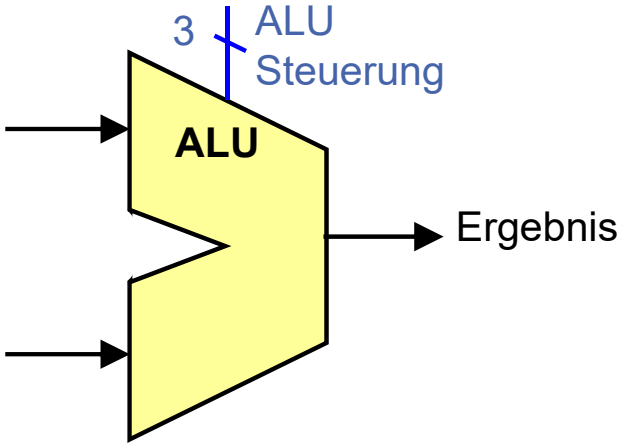
Befehlsabarbeitung und Datenpfade in MIPS

- **Befehle vom R-Typ:**

6	5	5	5	5	6
op	rs	rt	rd	shamt	funct

`opcode` r_s, r_t, r_d



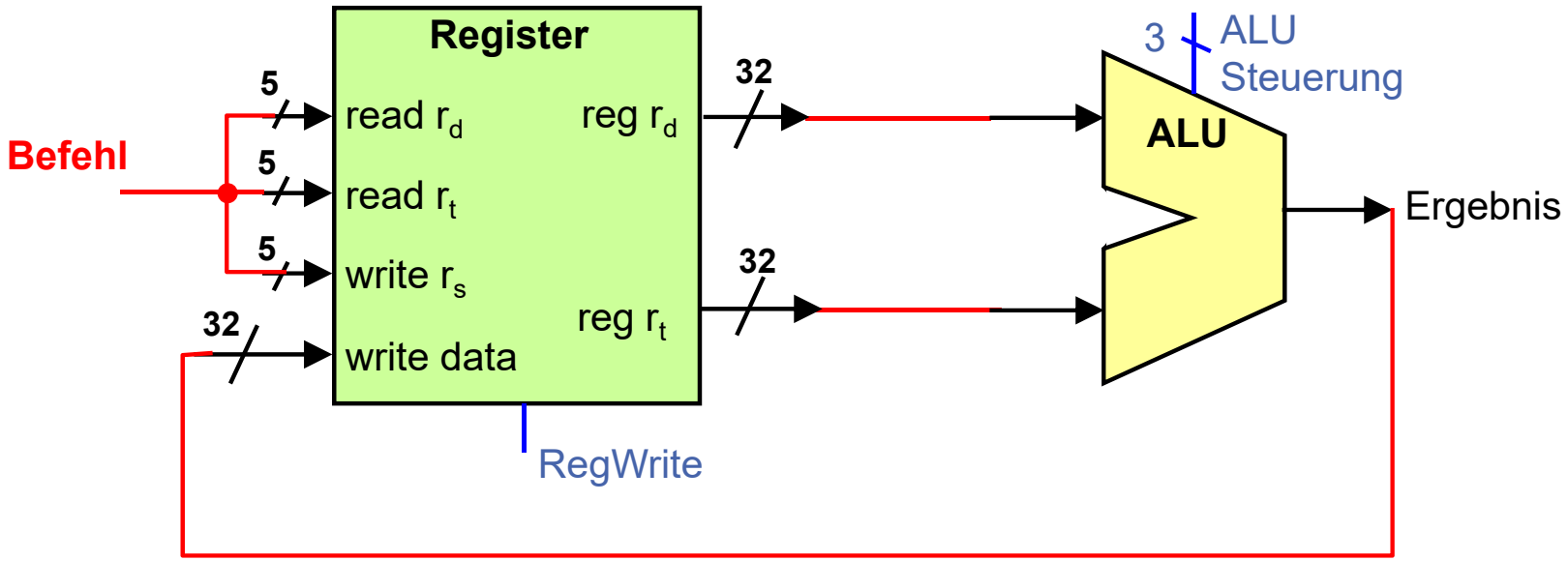


- Lesezugriff auf beiden Operanden-Register und
- ein Schreibzugriff auf das Zielregister

Befehlsabarbeitung und Datenpfade in MIPS

- **Befehle vom R-Typ:**

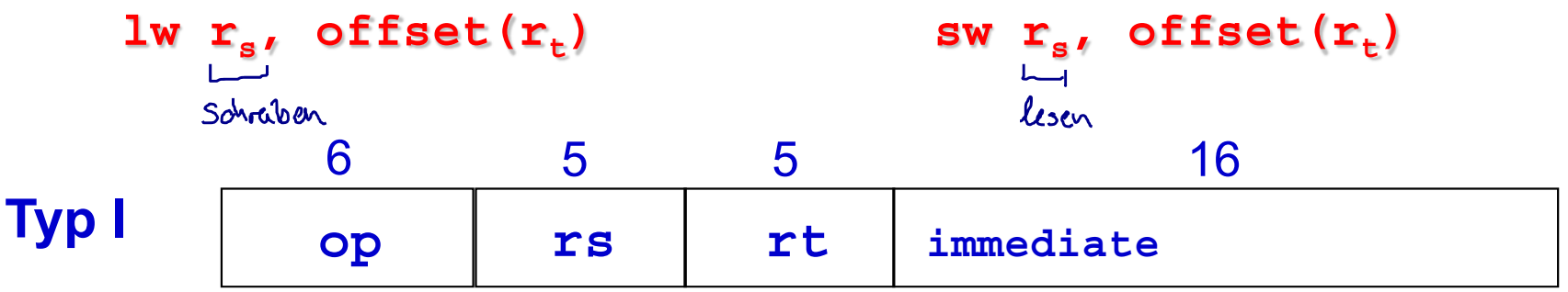
6	5	5	5	5	6
op	rs	rt	rd	shamt	funct



- Lesezugriff auf beiden Operanden-Register und
- ein Schreibzugriff auf das Zielregister

Befehlsabarbeitung und Datenpfade in MIPS

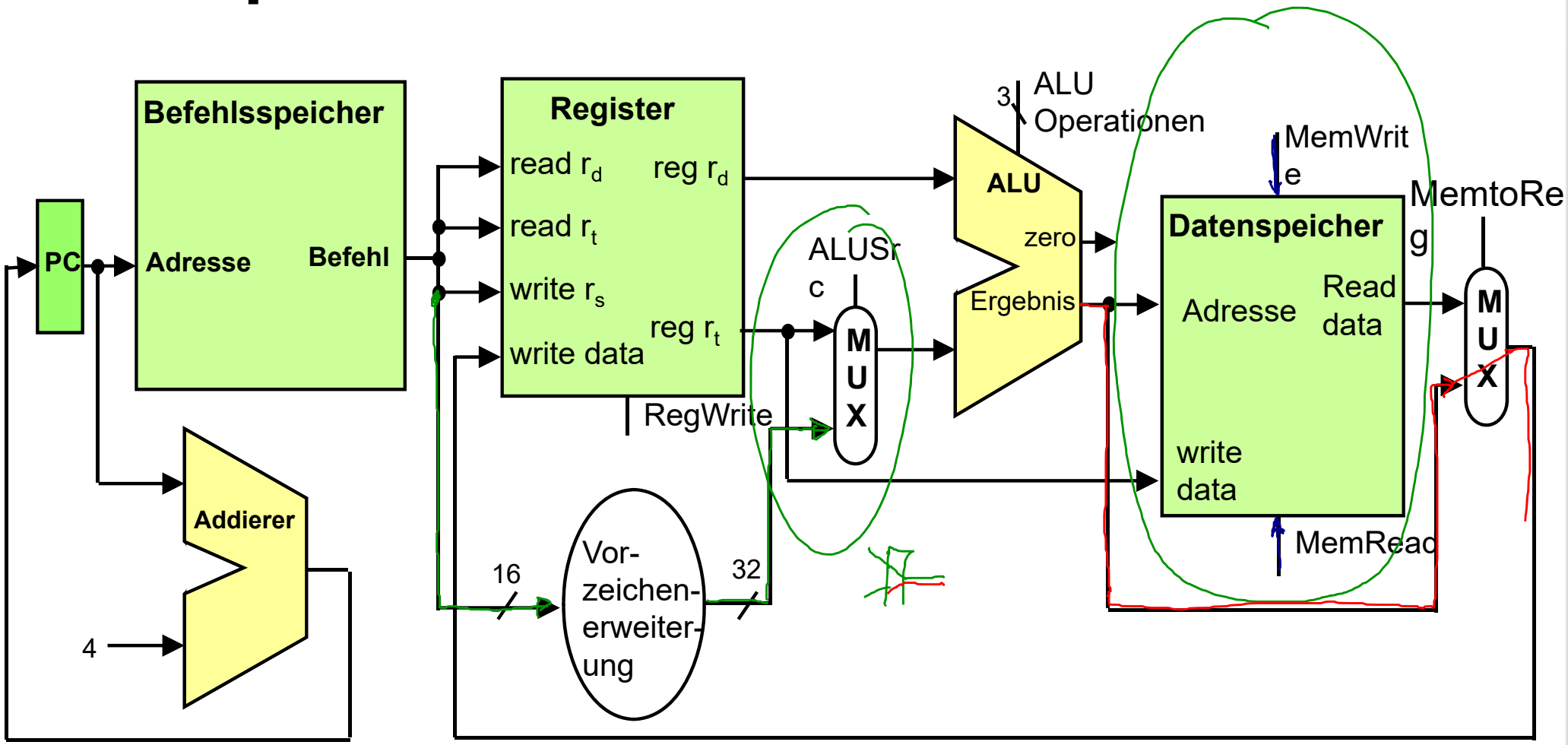
- **Lade- und Speicherbefehle *(load and store)***



- Speicheradresse wird durch die Addition einer Basisadresse im Register r_t zu einem vorzeichenbehafteten 16-bit Offset berechnet

- Bei Speicherbefehlen wird das zu speichernde Wort aus dem Register r_s gelesen. Bei Ladebefehlen wird das Wort ins Register r_s geladen

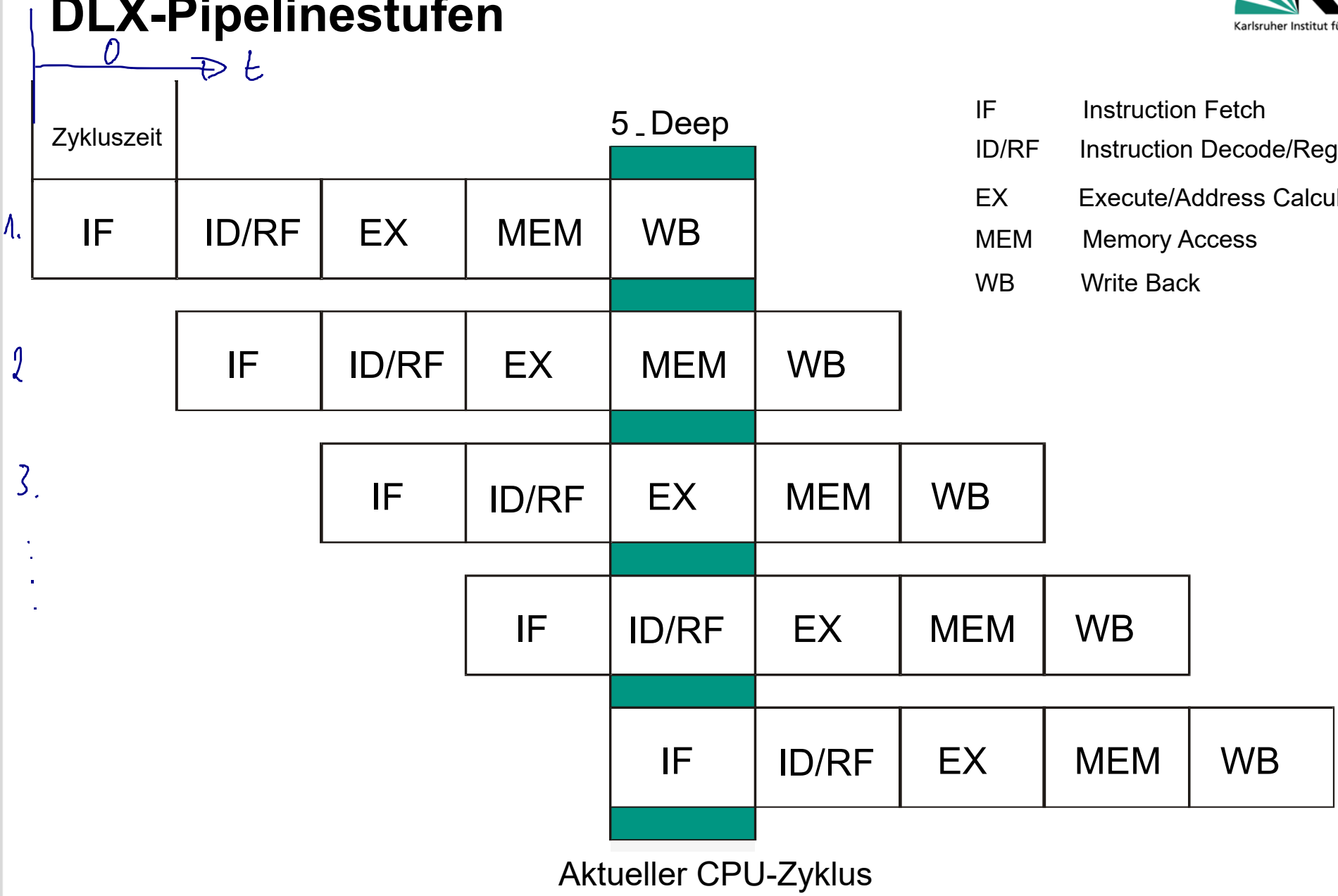
Befehlsabarbeitung und Datenpfade in MIPS



Einheit für das Holen von Befehlen

Datenpfade für Lade-Speicherbefehle und Befehle vom R-Typ

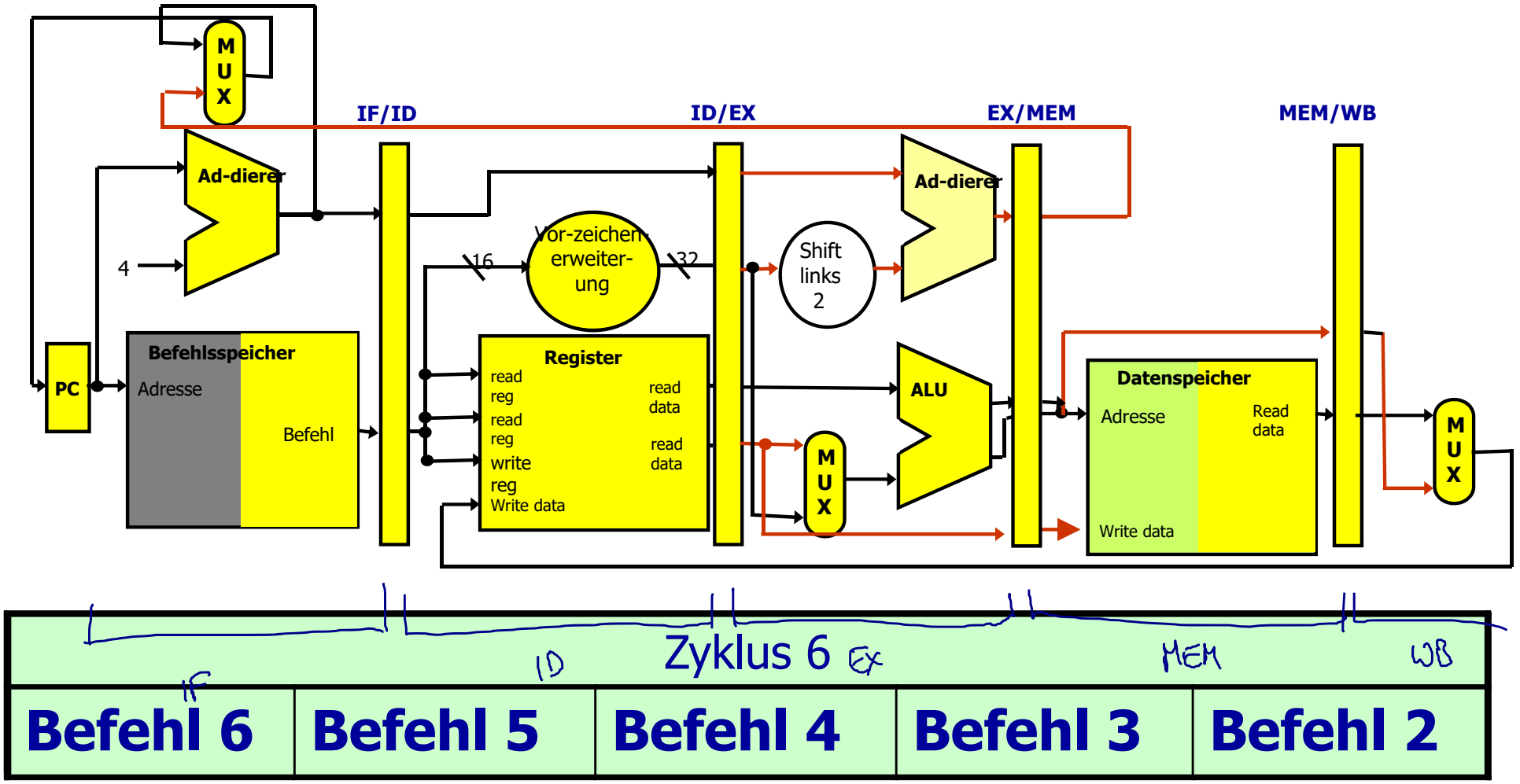
DLX-Pipelinestufen



IF	Instruction Fetch
ID/RF	Instruction Decode/Register Fetch
EX	Execute/Address Calculation
MEM	Memory Access
WB	Write Back

Pipeline-Konflikte

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



Drei Arten von Pipeline-Konflikten

- **Datenkonflikte:** Treten auf, wenn ein Operand in der Pipeline (noch) nicht verfügbar ist.
  - 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 Holphase 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 in der DLX Pipeline

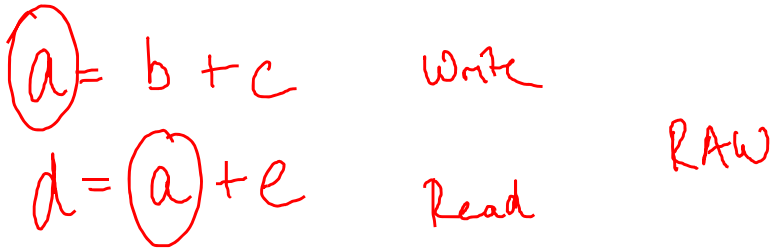
- Datenkonflikte:
  - RAW: Befehl $i + 1$ liest einen Wert bevor Befehl i geschrieben hat
  - WAW: Kann nicht auftreten, da die DLX-Pipeline nur in WB schreibt
  - WAR: Alle Lesezugriffe erfolgen in der ID/RF und alle Schreibzugriffe in WB
- Steuerflusskonflikte:
  - Pipeline muss wegen einer Verzweigung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

Datenabhängigkeiten

- Zwischen zwei aufeinander folgenden Befehlen $Inst_1$ und $Inst_2$ besteht eine

  echte Datenabhängigkeit (*true dependence*) δ^t RAW

  von $Inst_1$ zu $Inst_2$, wenn $Inst_1$ seine Ausgabe in ein Register *Reg* (oder in den Speicher) schreibt, das von $Inst_2$ als Eingabe gelesen wird.



Datenabhängigkeiten

- Zwischen zwei aufeinander folgenden Befehlen $Inst_1$ und $Inst_2$ besteht eine

  Gegenabhängigkeit (*antidependence*) δ^a WAR

  von $Inst_1$ zu $Inst_2$, falls $Inst_1$ Daten von einem Register *Reg* (oder einer Speicherstelle) liest, das anschließend von $Inst_2$ überschrieben wird.

Datenabhängigkeiten

- Zwischen zwei aufeinander folgenden Befehlen $Inst_1$ und $Inst_2$ besteht eine

  Ausgabeabhängigkeit (*output dependence*) δ^o

  von $Inst_2$ zu $Inst_1$, wenn beide in das gleiche Register *Reg* (oder eine Speicherstelle) schreiben und $Inst_2$ sein Ergebnis nach $Inst_1$ schreibt.

@ = b + c Write

@ = d + e Write

WAW

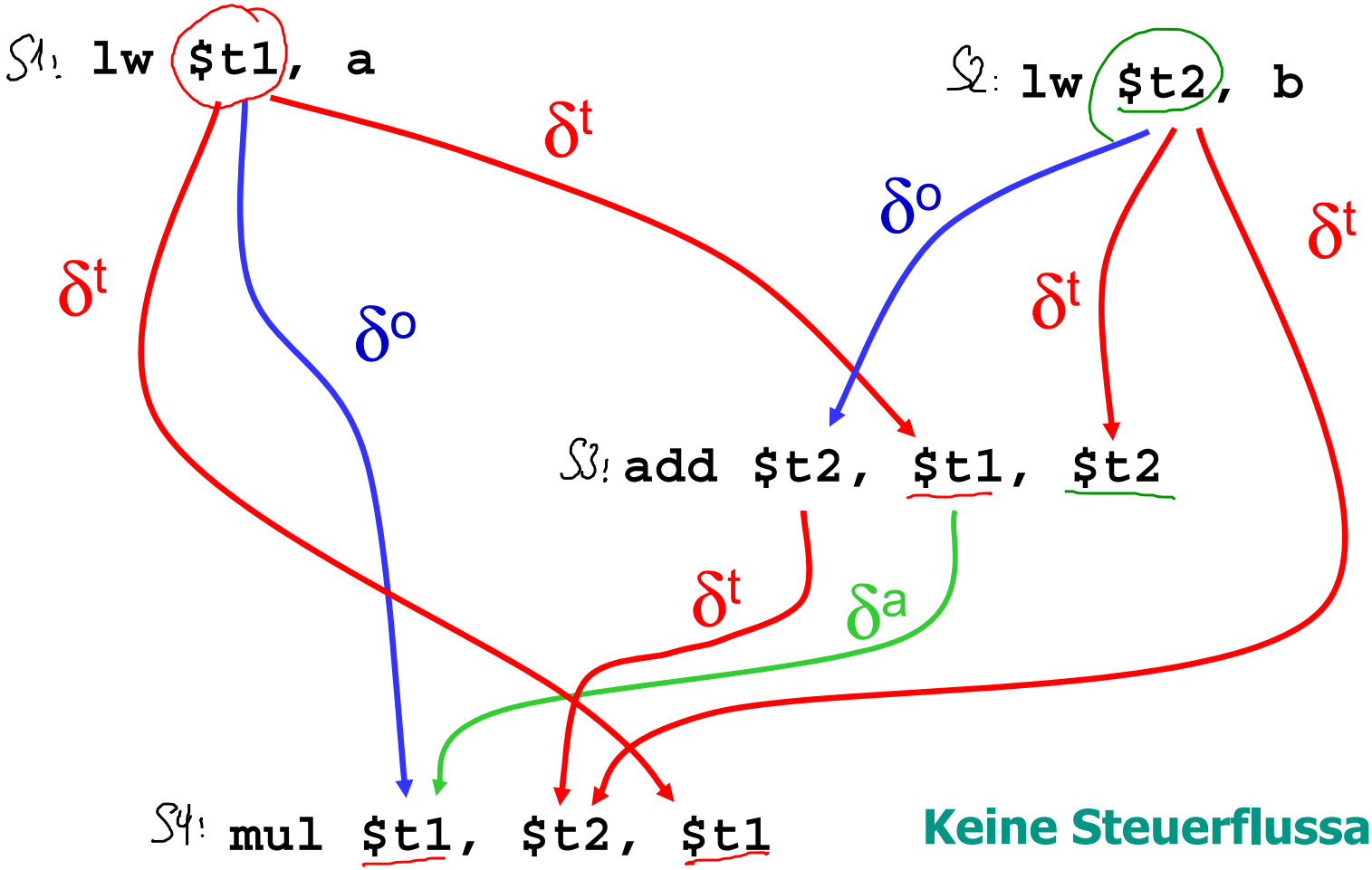
Aufgabe 1

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

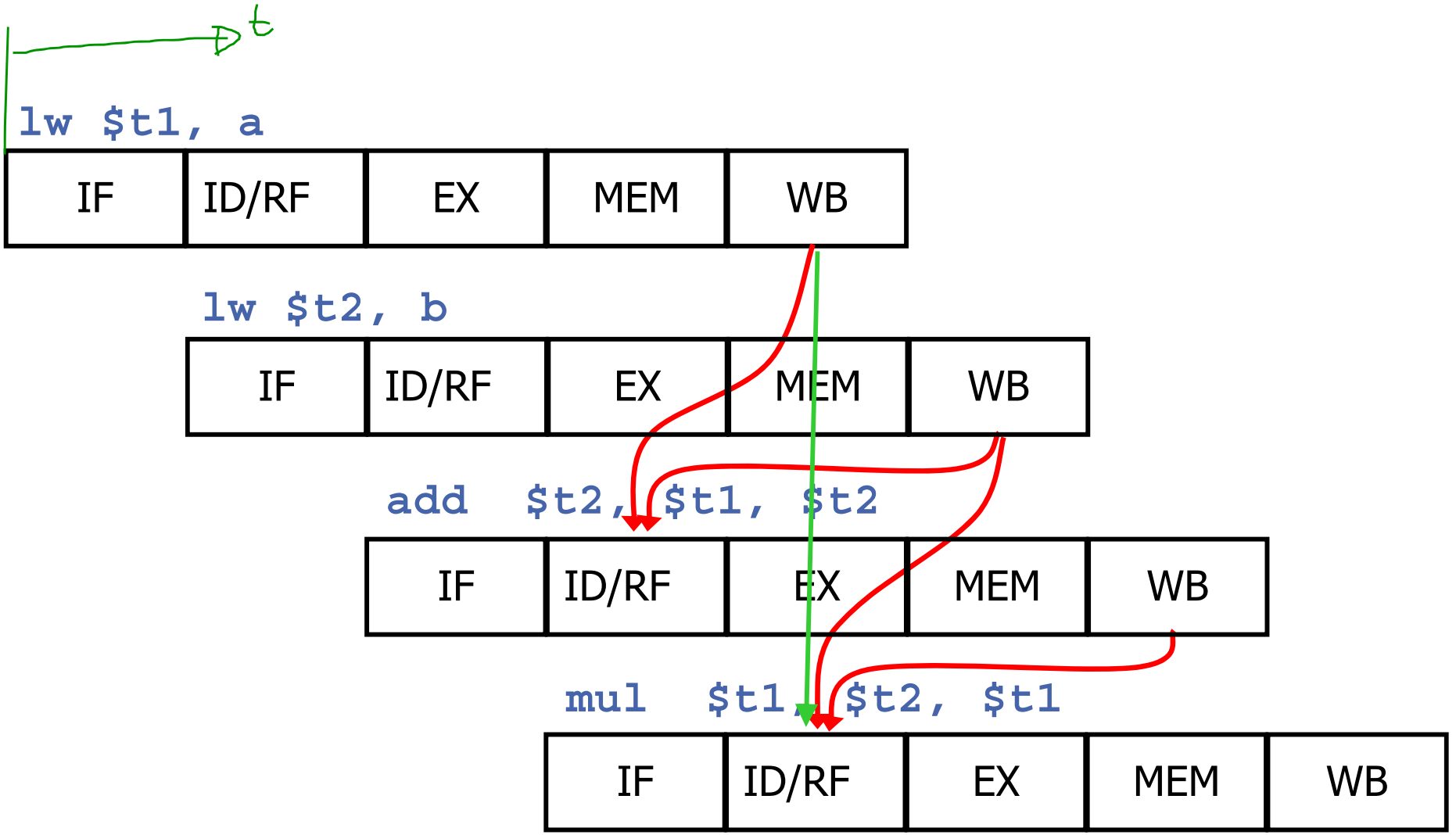
1. Bestimmen Sie alle Daten- und Steuerfluss-abhängigkeiten in diesem Programmstück



Keine Steuerflussabhängigkeit

Aufgabe 1.2

2. Wieviele Pipelinekonflikte treten auf?

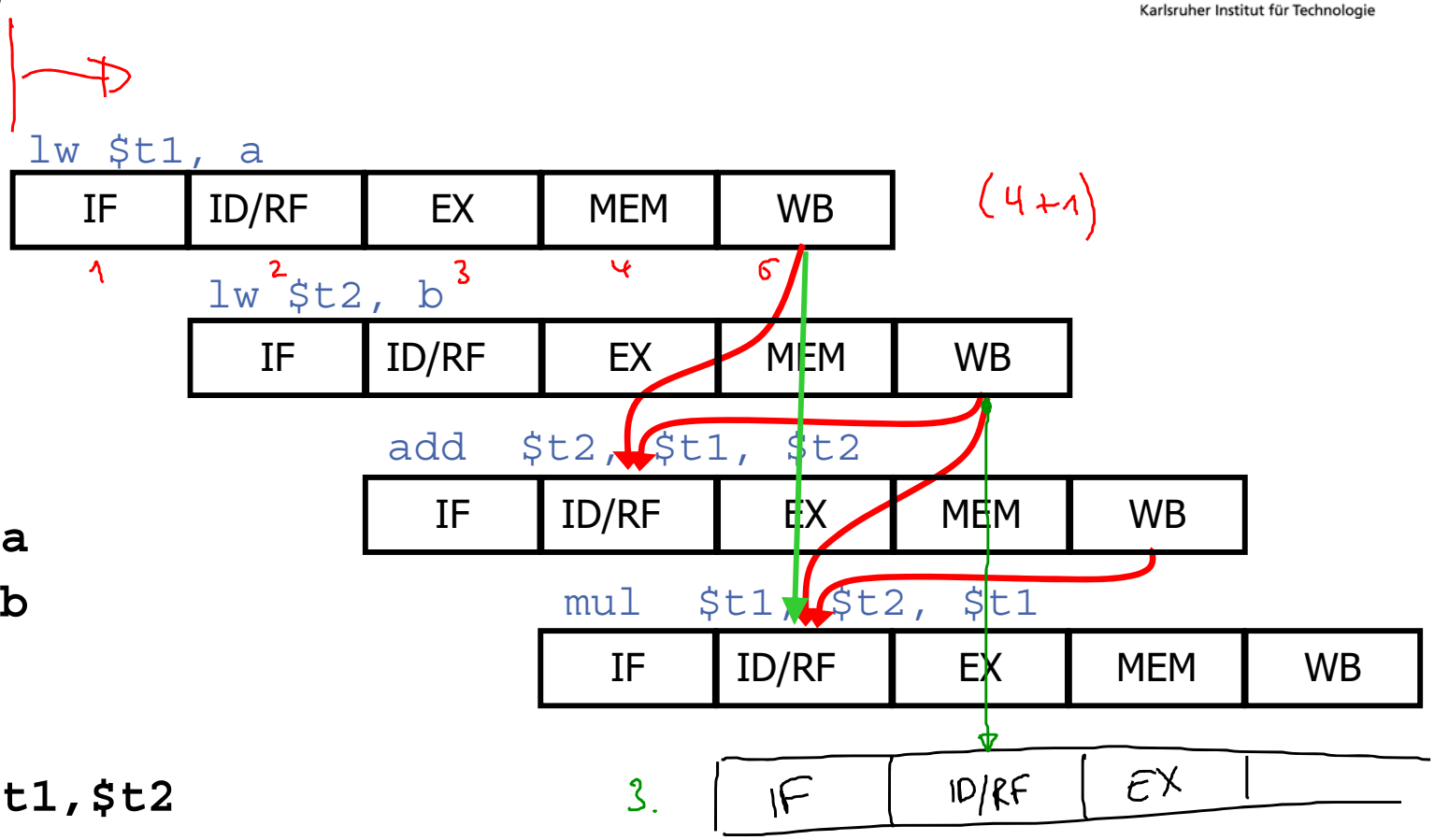


Aufgabe 1

3. Die auftretenden Pipelinekonflikte werden von der Hardware nicht erkannt und müssen vom Compiler durch Einfügen von NOP-Befehlen (No Operation) behandelt werden.

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

Aufgabe 1.3




```
lw      $t1, a
lw      $t2, b
nop
nop
add     $t2,$t1,$t2
nop
nop
mul     $t1,$t2,$t1
```

8 Befehle ⇒ 8 Takte + 4 Takte = 12 Takte

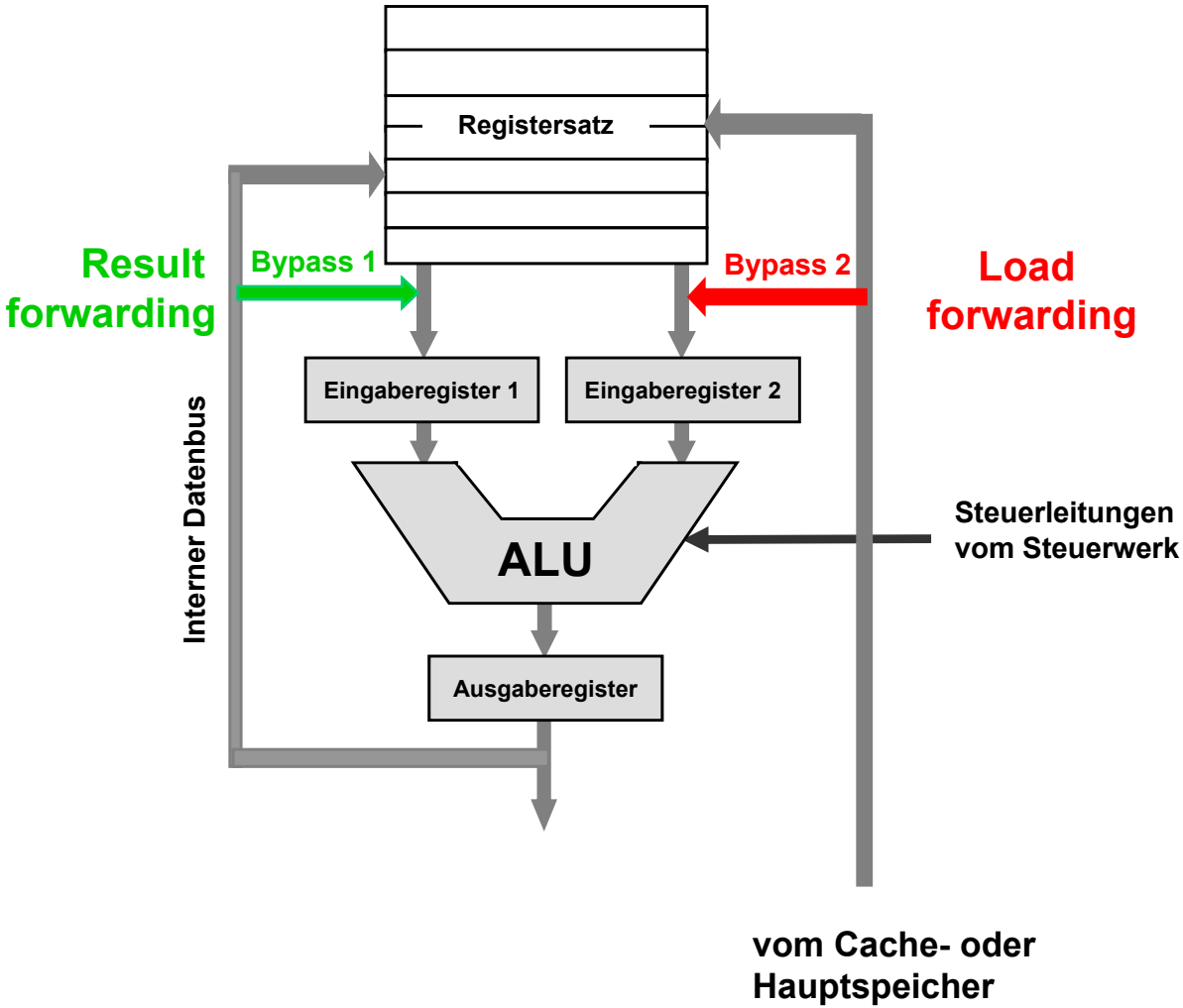
Aufgabe 1.4

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

Forwarding-Techniken

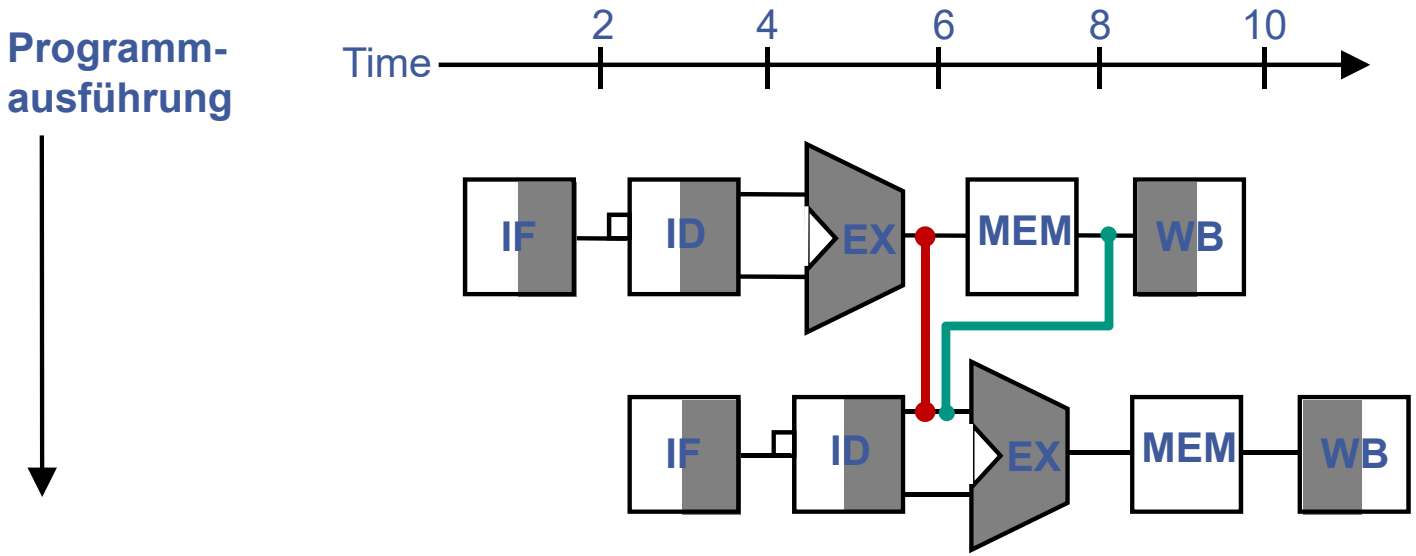
Forwarding:

- Rückführung des ALU-Ausgaberegister (*result forwarding*) bzw. des Ladewertregisters (*load forwarding*) auf die ALU-Eingaberegister
- Erhöhter Hardware- und Steuerungsaufwand



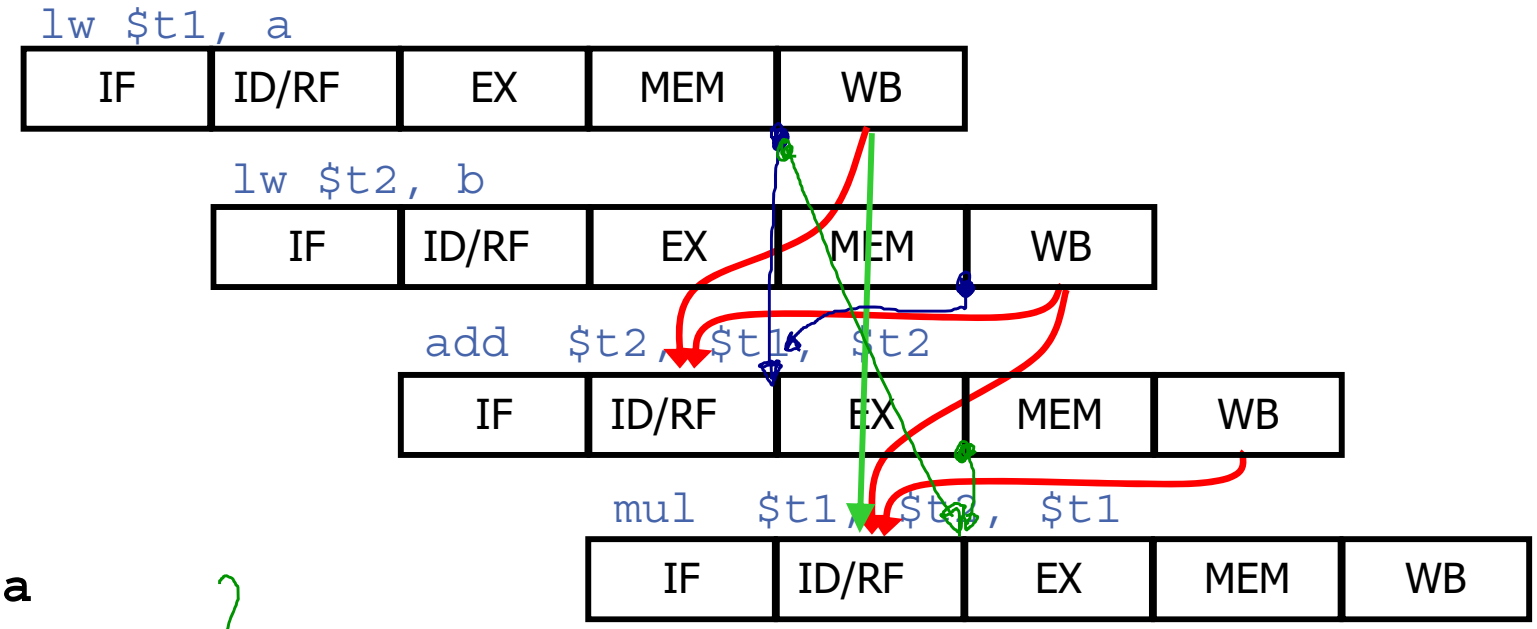
Forwarding-Techniken

Result Forwarding **Load Forwarding**



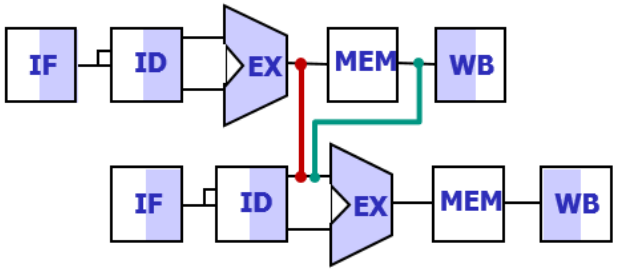
Erhöhter Hardware- und Steuerungsaufwand für Forwarding-Logik und zusätzliche Datenpfade

Aufgabe 1.4




```
lw      $t1, a
lw      $t2, b
nop
add     $t2,$t1,$t2
mul     $t1,$t2,$t1
```

5 Takte + 4 Takte = 9 Takte



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)
```

WRITE $t1, WRITE $t2, READ $t1, READ $t2, WRITE $t1, READ $t2, READ $t1

δ^t:
- S1 → S3 ($t1)
- S1 → S9 ($t1)
- S2 → S3 ($t2)
- S2 → S4 ($t2)
- S2 → S6 ($t2)
- ⋮

δ^a:
- S3 → S4 ($t1)
- ⋮

Lösung 2.1

- Echte Abhängigkeiten (True Dependence)

  S1 → S3 ($t1) S1 → S9 ($t1)

  S2 → S3 ($t2) S2 → S4 ($t2) S2 → S6 ($t2)

  S3 → S6 ($t3)

  S4 → S9 ($t1)

  S5 → S7 ($t4)

  S6 → S8 ($t5)

- Gegenabhängigkeiten (Anti-Dependence): S3 → S4 ($t1)
- Ausgabe-Abhängigkeiten (Output Dependence): S1 → S4 ($t1)

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.

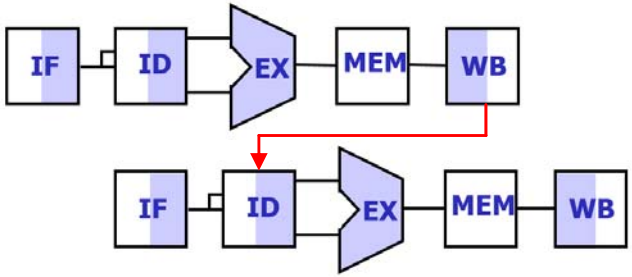
  Ergänzen Sie das obige Programm, so dass es korrekte Ergebnisse liefert. Sie dürfen dabei die Reihenfolge der Befehle nicht ändern und so wenig NOP-Befehle wie möglich einfügen.

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)
```

nop
nop

nop



Echte Abhängigkeiten (True Dependence)

S1 → S3 ($t1)	S1 → S9 ($t1)	
S2 → S3 ($t2)	S2 → S4 ($t2)	S2 → S6 ($t2)
	S3 → S6 ($t3)	
	S4 → S9 ($t1)	
	S5 → S7 ($t4)	
	S6 → S8 ($t5)	

Aufgabe 2.3

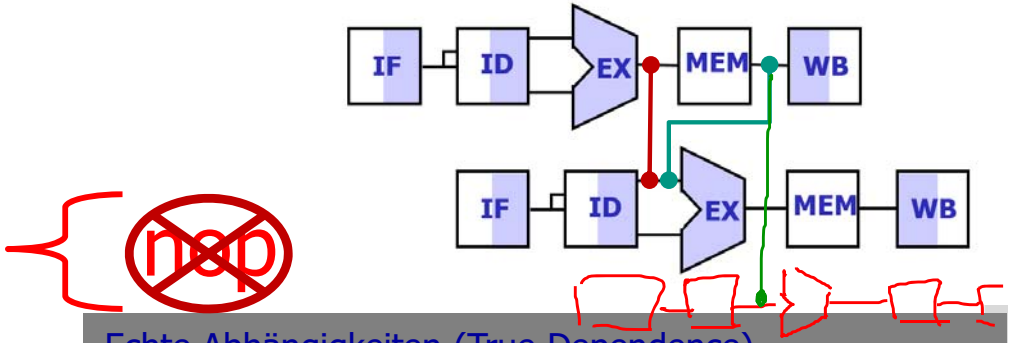
- Welche der NOP-Befehle sind noch notwendig, falls die auftretenden Pipelinekonflikte von der Hardware erkannt werden und durch Load Forwarding und Result Forwarding behandelt werden?

Lösung 2.3

```
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)
```

nop

nop



nop

Echte Abhängigkeiten (True Dependence)

S1 → S3 ($t1)
S2 → S3 ($t2)
S1 → S9 ($t1)
S2 → S4 ($t2)
S2 → S6 ($t2)
S3 → S6 ($t3)
S4 → S9 ($t1)
S5 → S7 ($t4)
S6 → S8 ($t5)

Aufgabe 3

Gegeben sei das folgende DLX-Programm

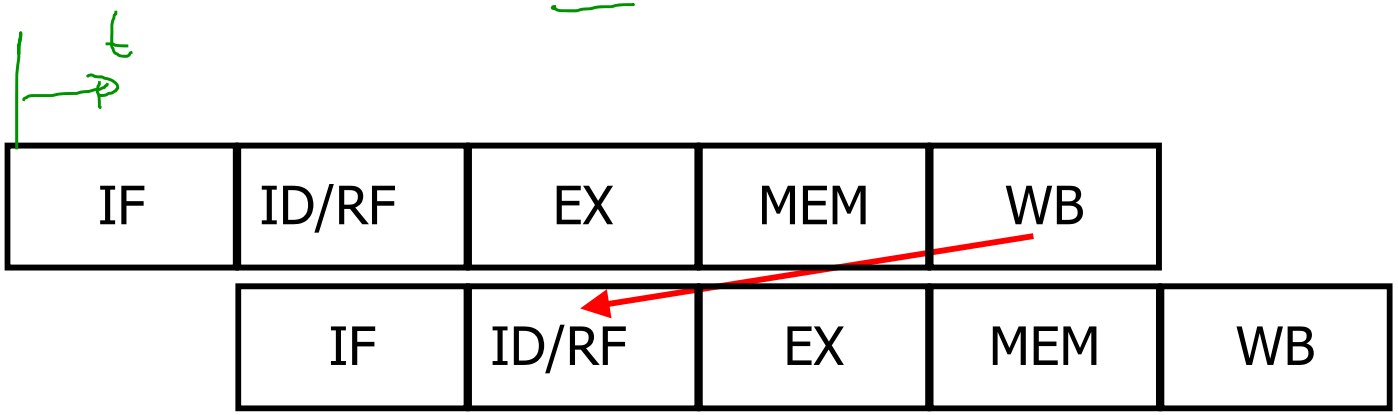
```
S1:           add $t1, $zero, $zero
S2:           lw $t3, 1500($zero)
S3:    loop:  lw $t4, 5000($t1)
S4:           add $t5, $t4, $t3
S5:           sw $t5, 400($t1)
S6:           addi $t1, $t1, 4
S7:           subi $t2, $t1, 400
S8:           bnez $t2, loop
S9:    end:   srl $t1, $t1, 2
S10:    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

RAW nach load

Steuerflussabhängigkeit wegen branch

RAW nach load

```
S3:    loop:  lw $t4, 5000($t1)
S4:           add $t5, $t4, $t3
```

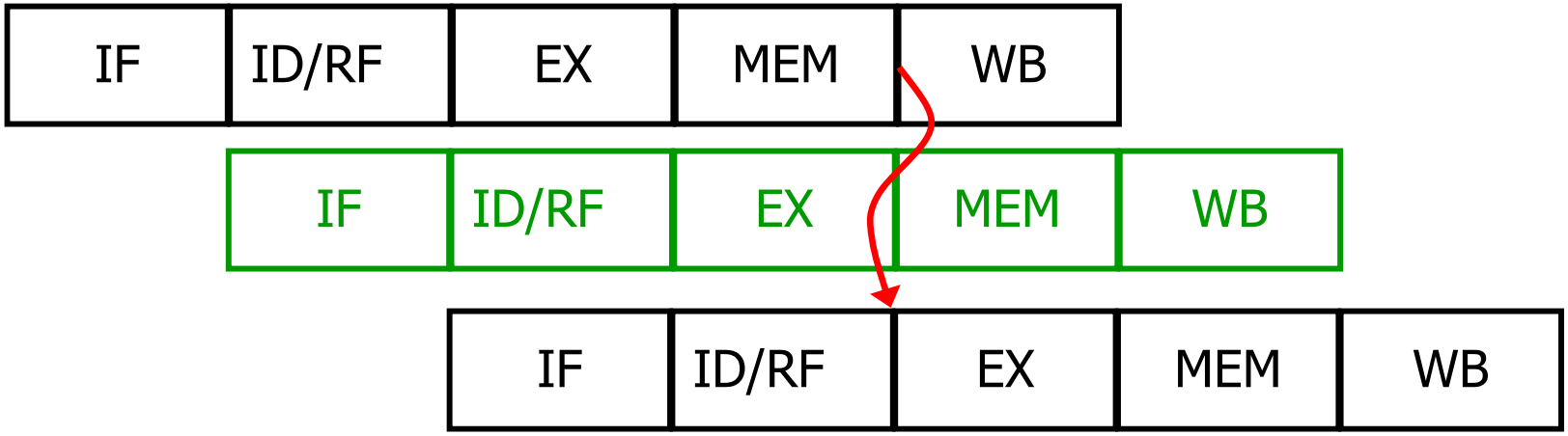


- lw schreibt $t4 nachdem add $t4 bereits ausgelesen hat.
- Forwarding: EX-Phase von add muss um einen Takt verschoben werden

RAW nach load

```
S3:    loop:  lw $t4, 5000($t1)
              nop
S4:           add $t5, $t4, $t3
```

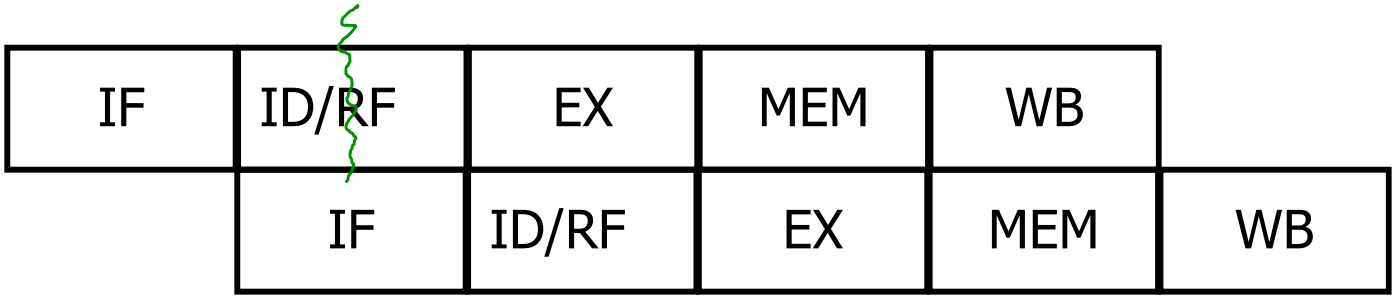
Kann durch unabhängigen Befehl ersetzt werden



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

Steuerflussabhängigkeit nach branch

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



Annahme:

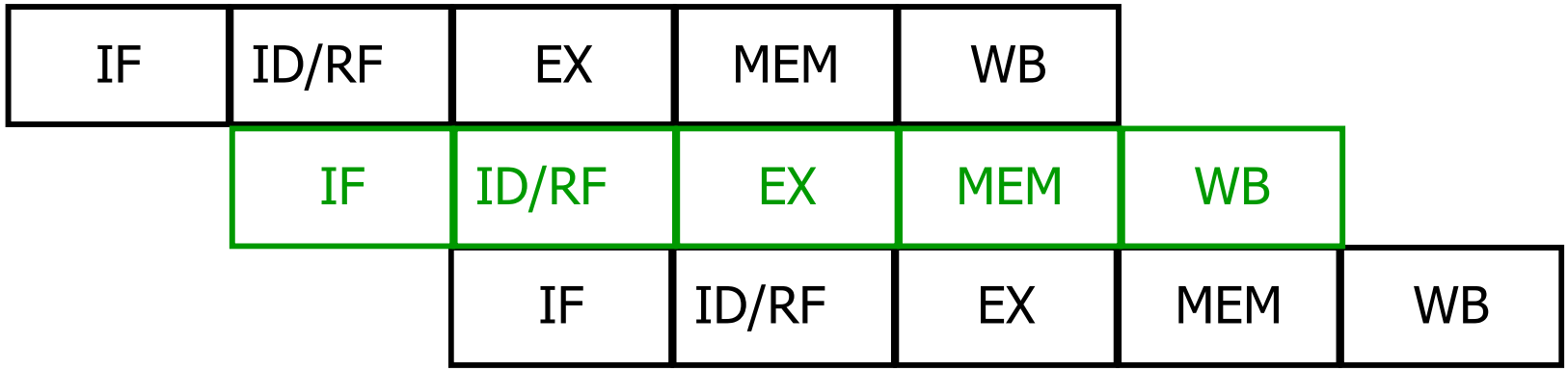
Dekodierung, Berechnung der Sprungzieladresse und Rückschreibung des PCs in der ID-Stufe (modifizierte DLX Pipeline)

- Bei bnez wird in ID/RF-Stufe der Kontrollfluss geändert
- Nächster Befehl ist bereits in der Pipeline
- Immer ein nop unabhängig vom Folgebefehl

Steuerflussabhängigkeit nach branch

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

Kann durch unabhängigen Befehl ersetzt werden



- Bei bnez wird in ID/RF-Stufe der Kontrollfluss geändert
- Nächster Befehl ist bereits in der Pipeline
- Immer ein nop unabhängig vom Folgebefehl

Code mit nops

```
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:  srl $t1, $t1, 2
S10:         sw $t1, 2000($zero)
```

8 Taktzyklen pro Schleifendurchlauf (Mindestens)

Vermeidung von *nops* durch Umorganisieren der Schleife

```
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:   srl $t1, $t1, 2
S10:         sw $t1, 2000($zero)
```

Vermeidung von *nops* 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
              nop
S9:    end:   srl $t1, $t1, 2
S10:          sw $t1, 2000($zero)
```

Vermeidung von *nops* 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
              nop
S9:    end:   srl $t1, $t1, 2
S10:          sw $t1, 2000($zero)
```

Vermeidung von *nops* 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:   srl $t1, $t1, 2
S10:          sw $t1, 2000($zero)
```

Vermeidung von *nops* durch Umorganisieren der Schleife

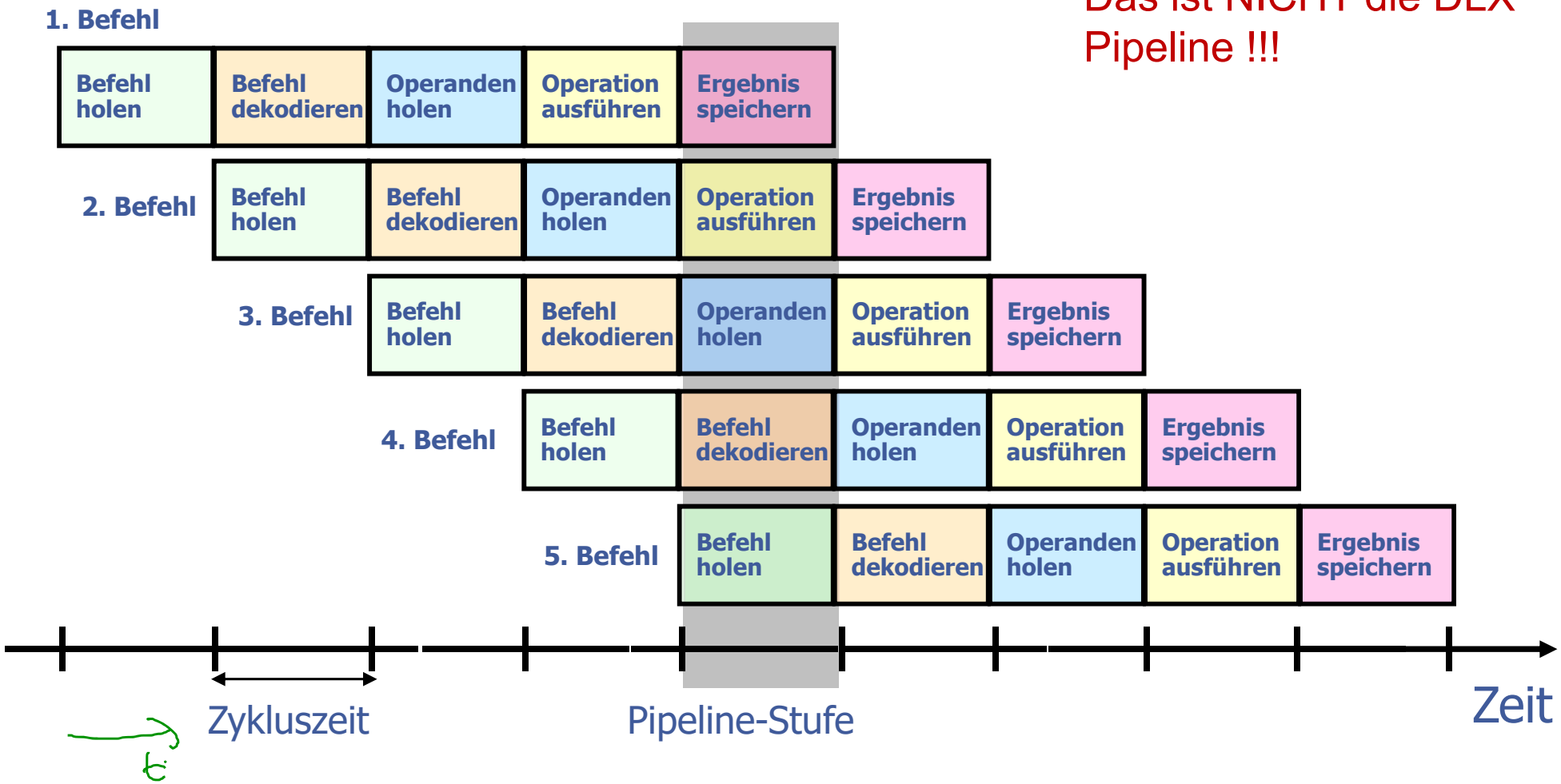
```
S1:          add $t1, $zero, $zero
S2:          lw $t3, 1500($zero)
S3:    loop: lw $t4, 5000($t1)
S6:          addi $t1, $t1, 4
S4:          add $t5, $t4, $t3
S7:          subi $t2, $t1, 400
S8:          bnez $t2, loop
S5:          sw $t5, 396($t1)
S9:    end:  srl $t1, $t1, 2
S10:         sw $t1,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

Alle nops wurden beseitigt

6 Taktzyklen pro Schleifendurchlauf

Aufgabe 4

Gegeben sei eine 5-stufige Pipeline:



Aufgabe 4

Die Kontrolle der Befehlspipeline wird vollständig dem Compiler übertragen. Die einzelnen Befehle werden in einer fünfstufigen Befehlspipeline verarbeitet. Erst am Ende der Ergebnis-speichern-Phase ist ein Schreibvorgang in das entsprechende Zielregister abgeschlossen.

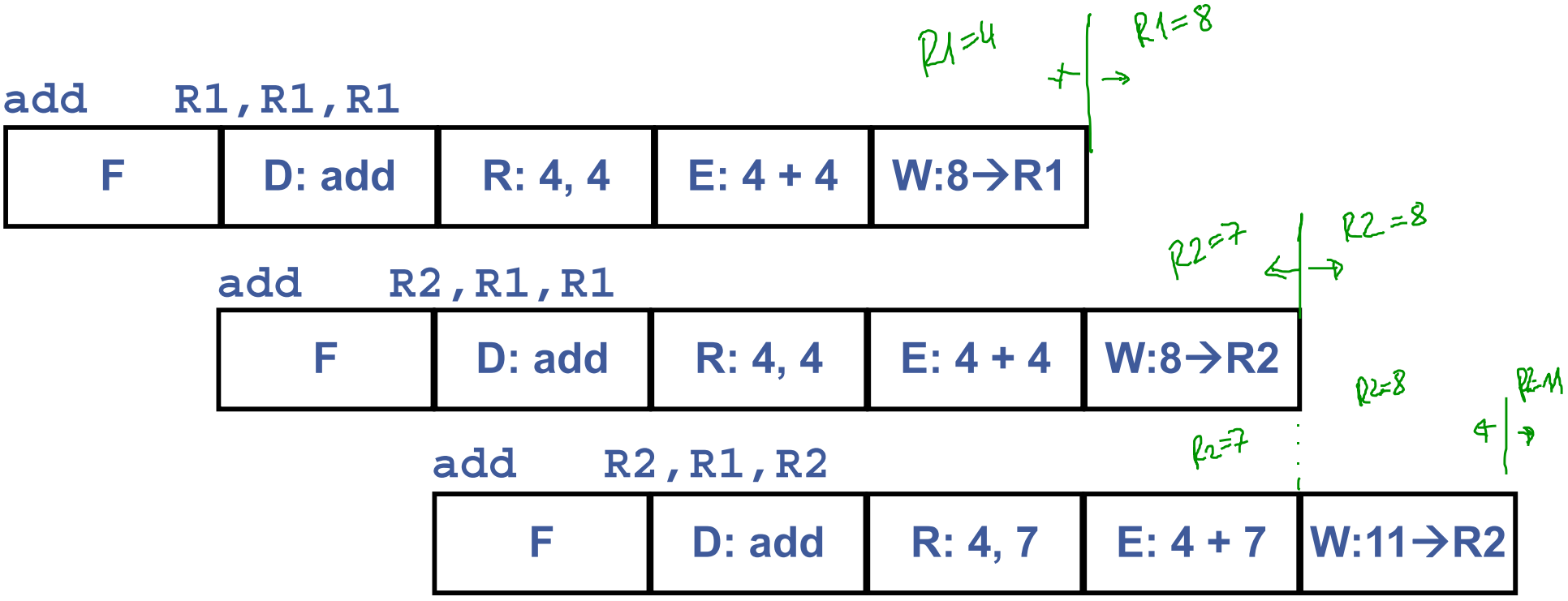
Betrachten Sie das folgende sequentielle Programmstück:

```
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 4.1

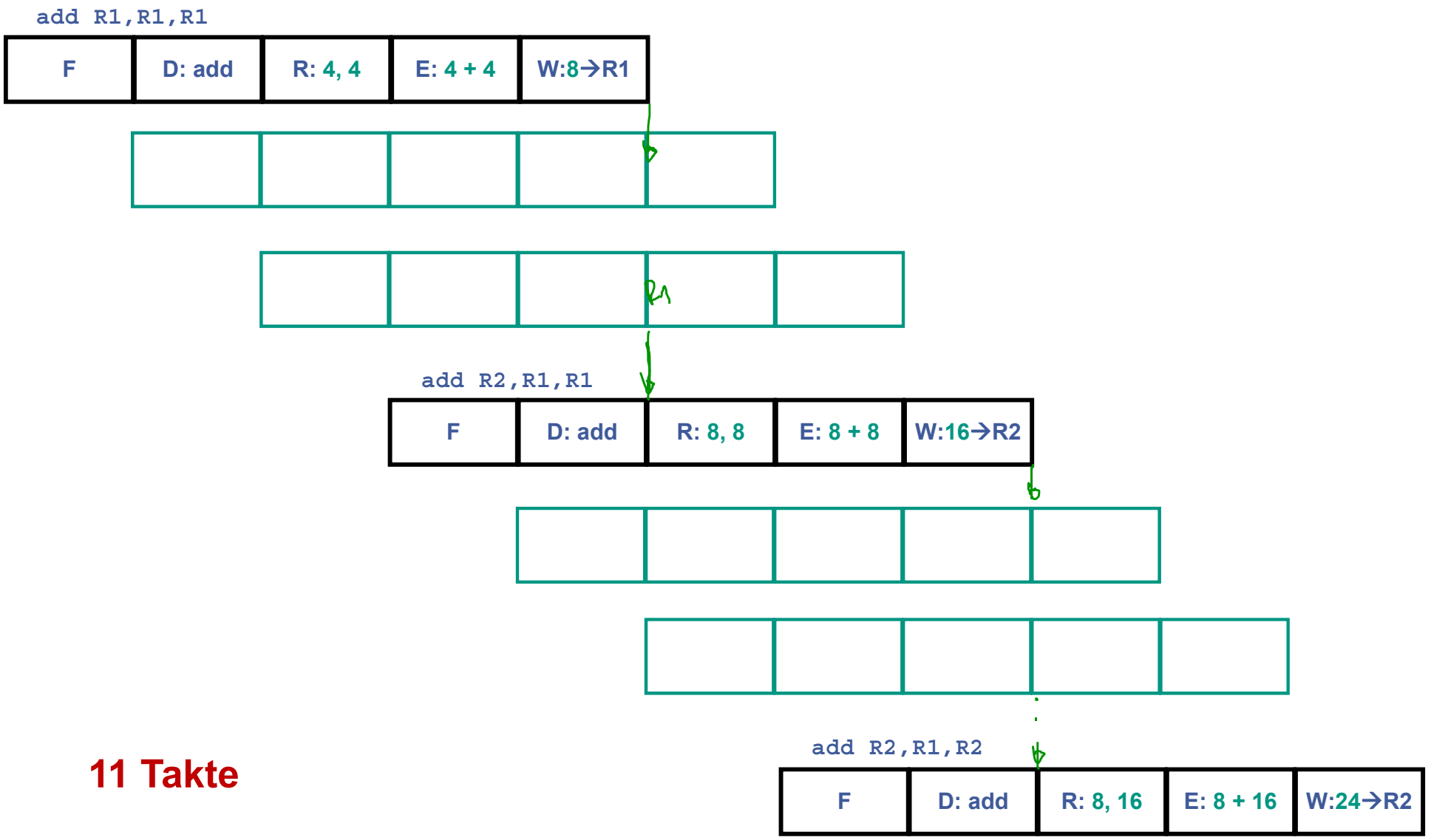
```
add     R1,R1,R1
add     R2,R1,R1
add     R2,R1,R2
```

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



R2 = 11

Korrekte Ausführung



11 Takte

Korrekte Ausführung

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

F	D: add	R: 4, 4	E: 4 + 4	W:8→R1

2 nops

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

F	D: add	R: 8, 8	E: 8 + 8	W:16→R2

2 nops

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

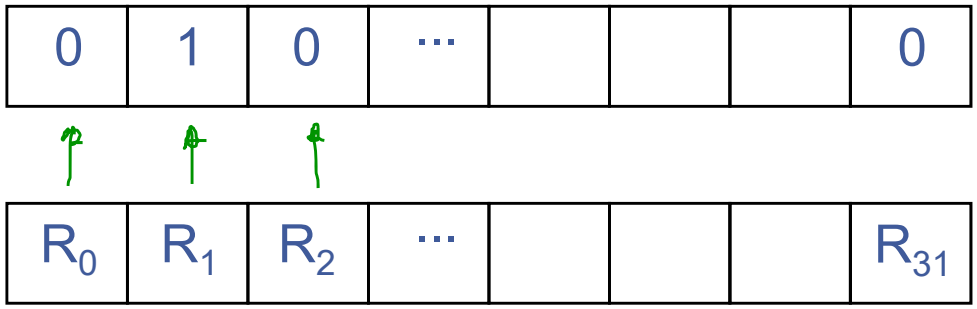
F	D: add	R: 8, 16	E: 8 + 16	W:24→R2

Aufgabe 4.2

2. Wie viele 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.

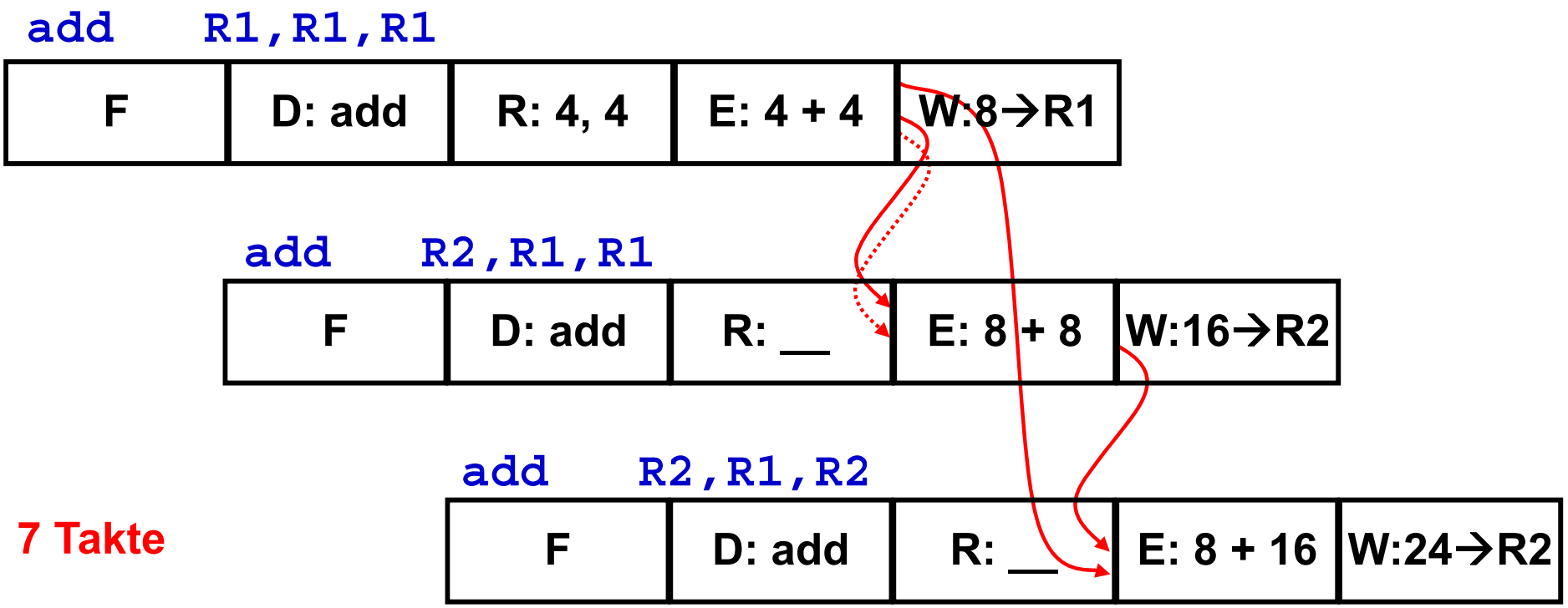
Scoreboarding-Technik

- 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 Daten-abhängigkeiten durch die Hardware erkannt und im einfachsten Fall durch Verzögerungen behandelt werden können.

Aufgabe 4.2

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



7 Takte

Aufgabe 5

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 5

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 `mi, mj, mk, …` 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 5

- 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.