

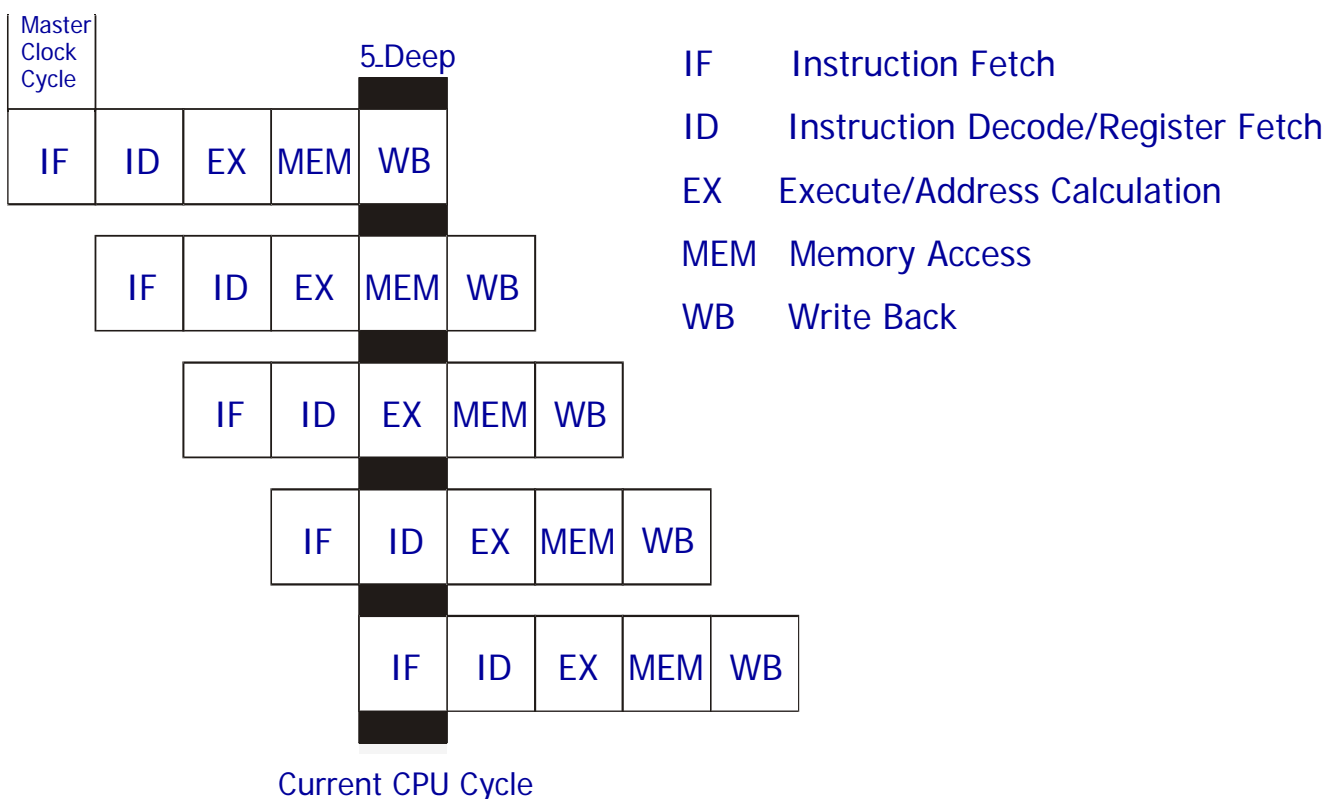
# Pipelining „Fließband-Bearbeitung“

„Pipelines beschleunigen die Ausführungsgeschwindigkeit eines Rechners in gleicher Weise wie Henry Ford die Autoproduktion mit der Einführung des Fließbandes revolutionierte.“

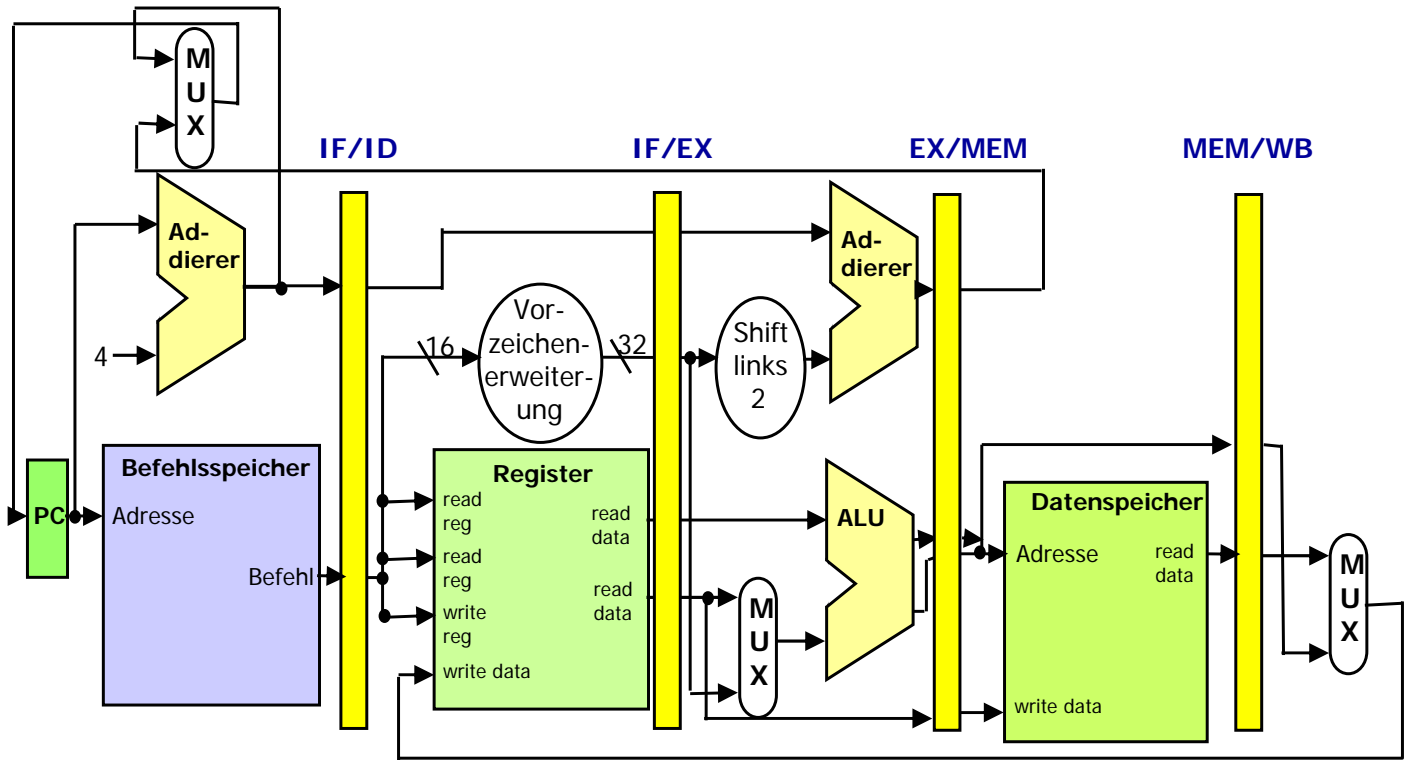
(Peter Wayner 1992)



## DLX Pipeline

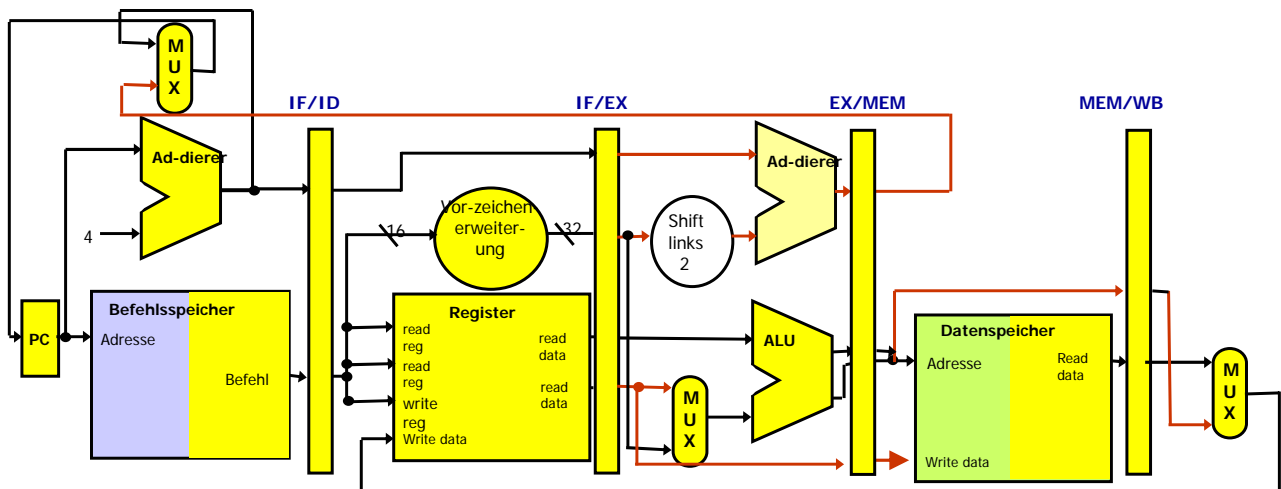


# Pipelining in MIPS-Architektur



# Pipeline Konflikte

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



Zyklus 6				
Befehl 6	Befehl 5	Befehl 4	Befehl 3	Befehl 2

# 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 Befehlsbereitstellungsphase 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.



## Datenabhängigkeiten

- Zwischen zwei aufeinander folgenden Befehle  $Inst_1$  und  $Inst_2$  besteht eine **echte Datenabhängigkeit** (*true dependence*)  $\delta^t$  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 Befehle  $Inst_1$  und  $Inst_2$  besteht eine **Gegenabhängigkeit (*antidependence*)**  $\delta^a$  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 Befehle  $Inst_1$  und  $Inst_2$  besteht eine **Ausgabeabhängigkeit (*output dependence*)**  $\delta^o$  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.



# Pipelinekonflikte

- Ressourcenkonflikte: DLX-Pipeline entsprechend angepasst
- 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

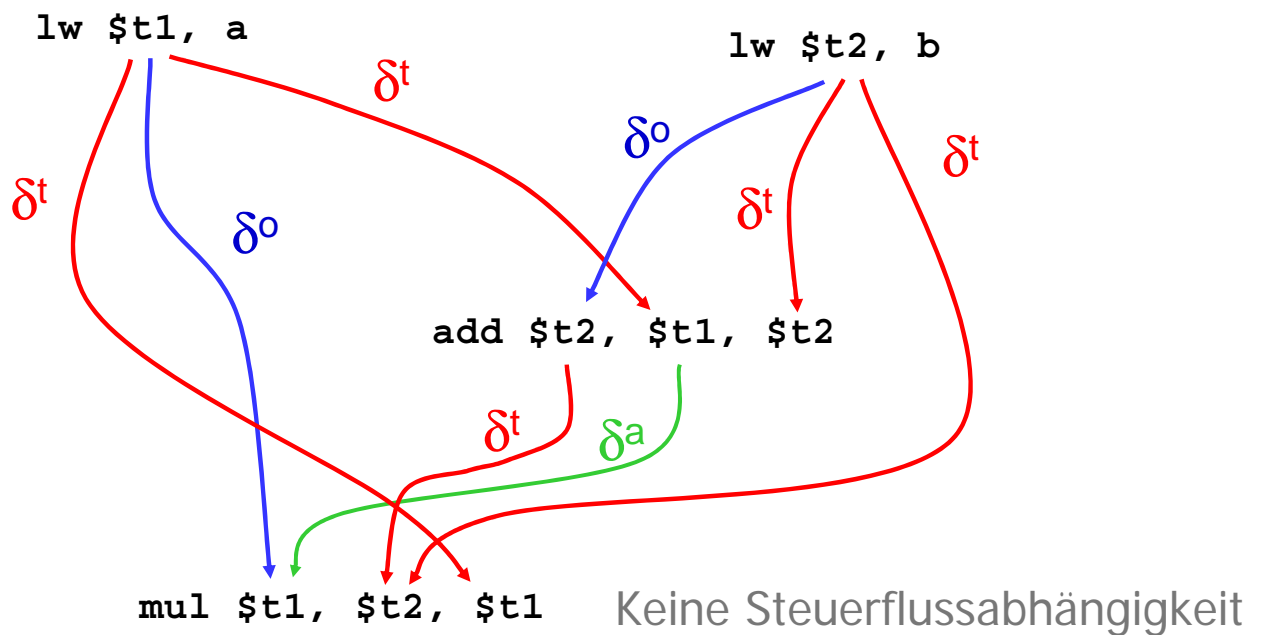
Betrachten Sie das folgende sequentielle Programmstück, in dem die Konstanten **a** und **b** Speicheradressen darstellen:

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



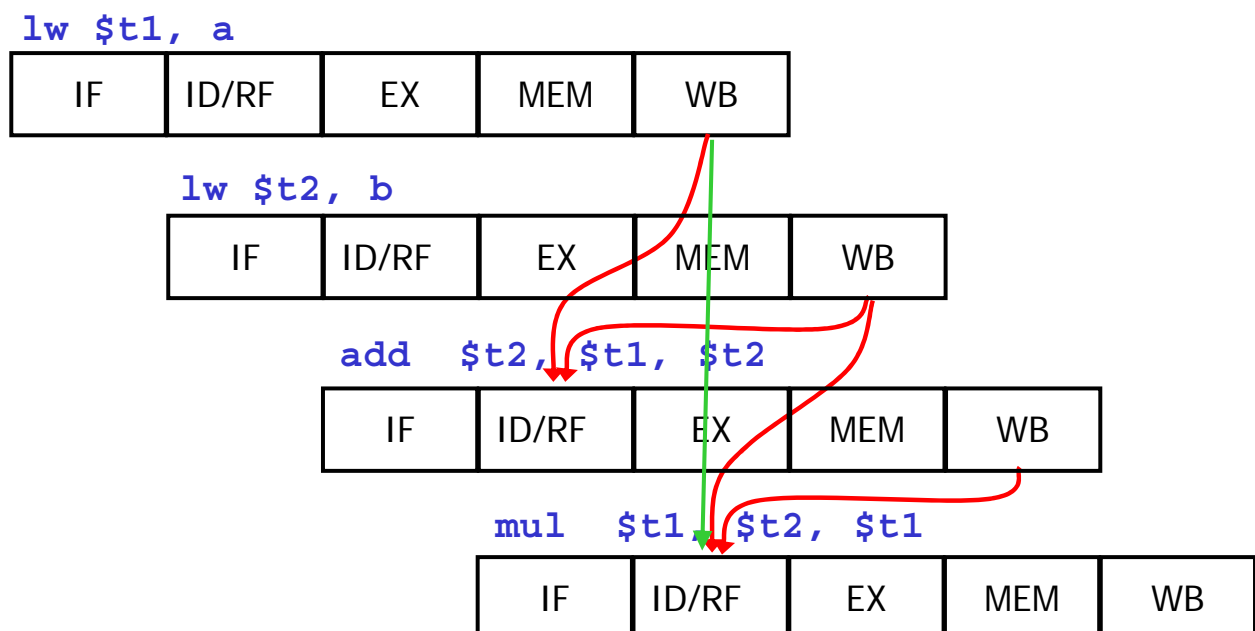
# Aufgabe 1.1

- Bestimmen Sie alle Daten- und Steuerflussabhängigkeiten in diesem Programmstück



# Aufgabe 1.2

- Wieviele Pipelinekonflikte treten auf?



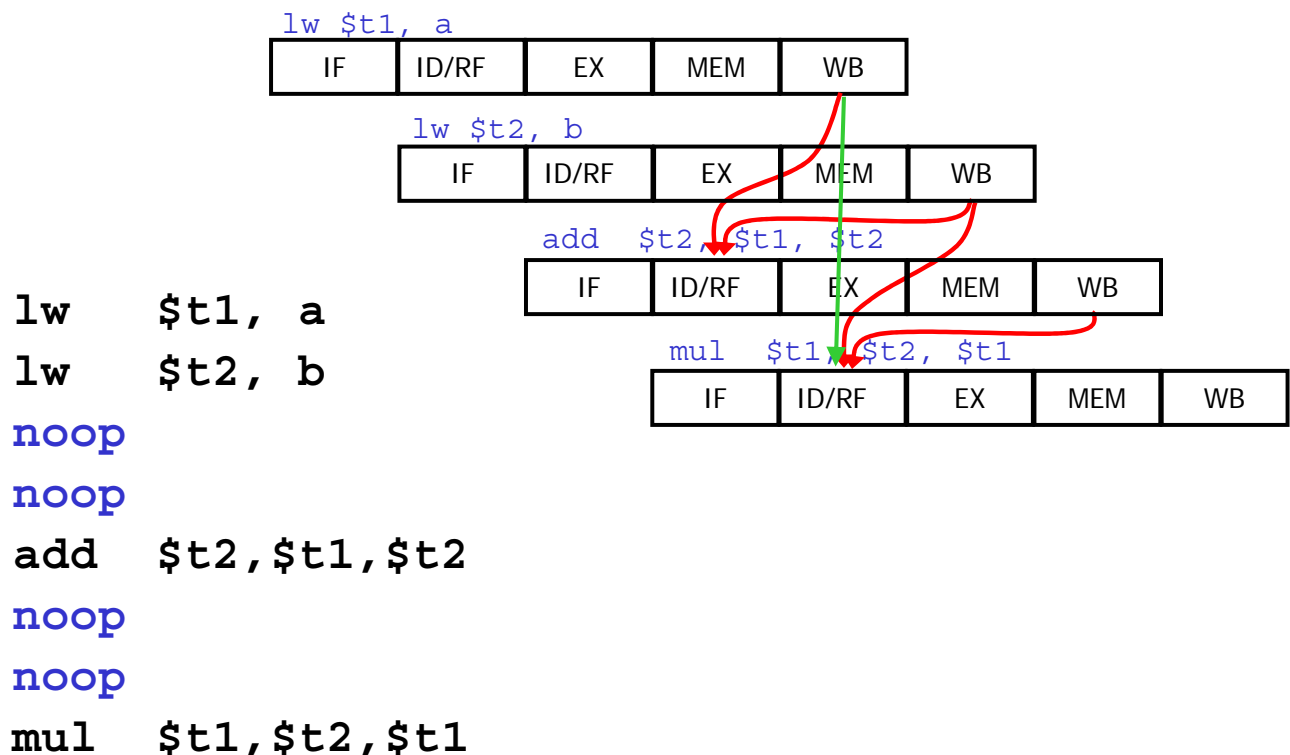
# Aufgabe 1

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

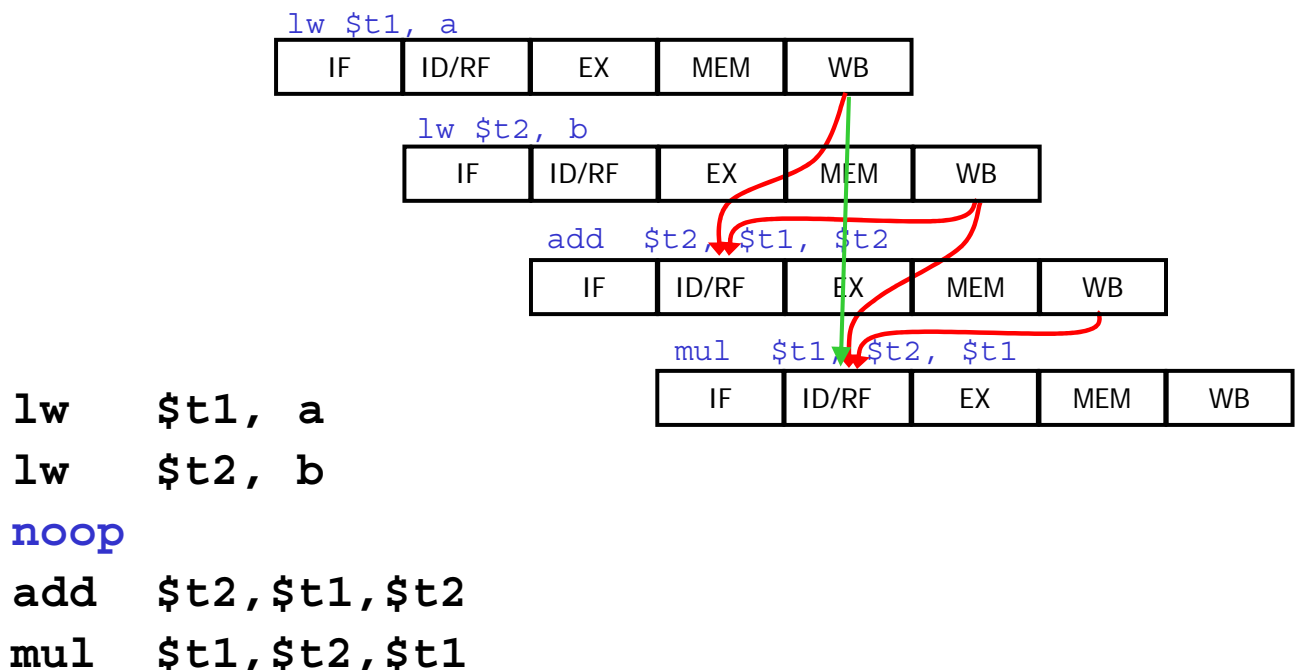


# Aufgabe 1

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

Gegeben sei das folgende DLX-Programm

```
S1:      add $t1, $zero, $zero
S2:      lw  $t3, 0x1500($zero)
S3:  loop: lw  $t4, 0x5000($t1)
S4:      add $t5, $t4, $t3
S5:      sw  $t5, 0x400($t1)
S6:      addi $t1, $t1, 4
S7:      subi $t2, $t1, 0x400
S8:      bnez $t2, loop
S9:  end:  srli $t1, $t1, 2
S10:     sw  $t1, 0x2000($zero)
```

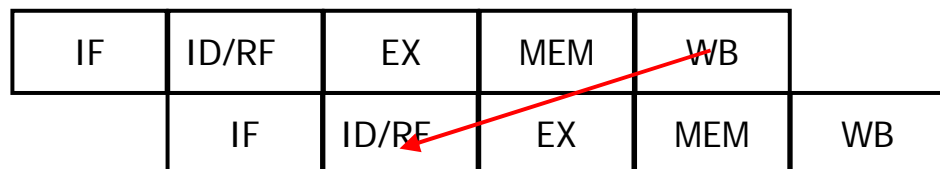
RAW nach load

Steuerflussabhängigkeit  
wegen branch



## RAW nach load

```
S3:  loop: lw  $t4, 0x5000($t1)
S4:      add $t5, $t4, $t3
```



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



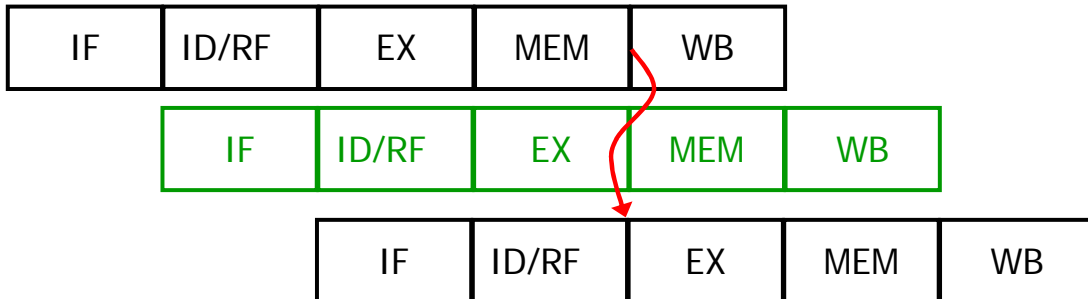
# RAW nach load

S3:      loop:    lw \$t4, 0x5000(\$t1)

          noop

S4:              add \$t5, \$t4, \$t3

Kann durch  
unabhängigen Befehl  
ersetzt werden



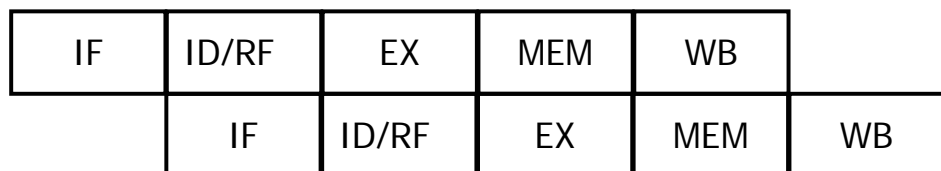
- lw schreibt \$t4 nachdem add \$t4 bereits ausgelesen hat.
- Forwarding: EX-Phase von add um einen Takt verschoben werden



## Steuerflussabhängigkeit nach branch

S8:              bnez \$t2, loop

S9:      end:      srli \$t1, \$t1, 2



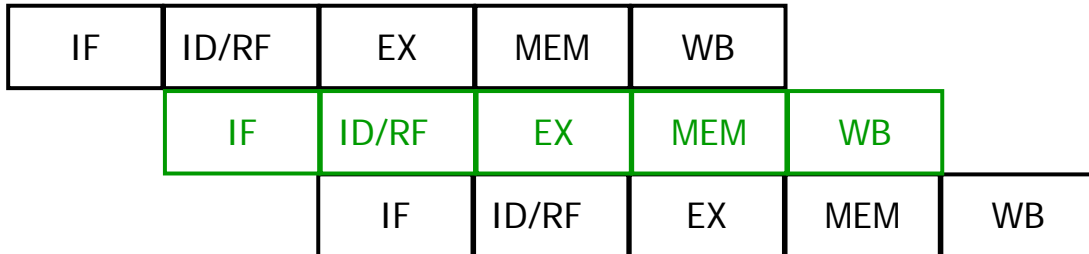
- Bei branch wird in ID der Kontrollfluss geändert
- Nächster Befehl ist bereits in der Pipeline
- Immer ein noop unabhängig vom Folgebefehl



# 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  $t3, 0x1500($zero)
S3:  loop:   lw  $t4, 0x5000($t1)
             noop
S4:          add $t5, $t4, $t3
S5:          sw  $t5, 0x400($t1)
S6:          addi $t1, $t1, 4
S7:          subi $t2, $t1, 0x400
S8:          bnez $t2, loop
             noop
S9:  end:    srli $t1, $t1, 2
S10:         sw  $t1, 0x2000($zero)
```

8 Taktzyklen pro  
Schleifendurchlauf



## Vermeidung von noops durch Umorganisieren der Schleife

```
S1:      add $t1, $zero, $zero
S2:      lw  $t3, 0x1500($zero)
S3:  loop: lw  $t4, 0x5000($t1)
          noop
S4:      add $t5, $t4, $t3
S5:      sw  $t5, 0x400($t1)
S6:      addi $t1, $t1, 4
S7:      subi $t2, $t1, 0x400
S8:      bnez $t2, loop
          noop
S9:  end:  srli $t1, $t1, 2
S10:     sw  $t1, 0x2000($zero)
```



## Vermeidung von noops durch Umorganisieren der Schleife

```
S1:      add $t1, $zero, $zero
S2:      lw  $t3, 0x1500($zero)
S3:  loop: lw  $t4, 0x5000($t1)
S6:      addi $t1, $t1, 4
S4:      add $t5, $t4, $t3
S5:      sw  $t5, 0x396($t1)
          noop
S7:      subi $t2, $t1, 0x400
S8:      bnez $t2, loop
          noop
S9:  end:  srli $t1, $t1, 2
S10:     sw  $t1, 0x2000($zero)
```



## Vermeidung von noops durch Umorganisieren der Schleife

```
S1:      add $t1, $zero, $zero
S2:      lw $t3, 0x1500($zero)
S3:  loop: lw $t4, 0x5000($t1)
S6:      addi $t1, $t1, 4
S4:      add $t5, $t4, $t3
S5:      sw $t5, 0x396($t1)
          

S7:      subi $t2, $t1, 0x400
S8:      bnez $t2, loop
          

S9:  end: srli $t1, $t1, 2
S10:     sw $t1, 0x2000($zero)
```



## Vermeidung von noops durch Umorganisieren der Schleife

```
S1:      add $t1, $zero, $zero
S2:      lw $t3, 0x1500($zero)
S3:  loop: lw $t4, 0x5000($t1)
S6:      addi $t1, $t1, 4
S4:      add $t5, $t4, $t3
          

S7:      subi $t2, $t1, 0x400
S8:      bnez $t2, loop
S5:      sw $t5, 0x396($t1)
S9:  end: srli $t1, $t1, 2
S10:     sw $t1, 0x2000($zero)
```



# Vermeidung von noops durch Umorganisieren der Schleife

```
S1:      add $t1, $zero, $zero
S2:      lw  $t3, 0x1500($zero)
S3:      loop: lw $t4, 0x5000($t1)
S6:      addi $t1, $t1, 4
S4:      add $t5, $t4, $t3
S7:      subi $t2, $t1, 0x400
S8:      bnez $t2, loop
S5:      sw  $t5, 0x396($t1)
S9:      end: srli $t1, $t1, 2
S10:     sw  $t1, 0x2000($zero)
```

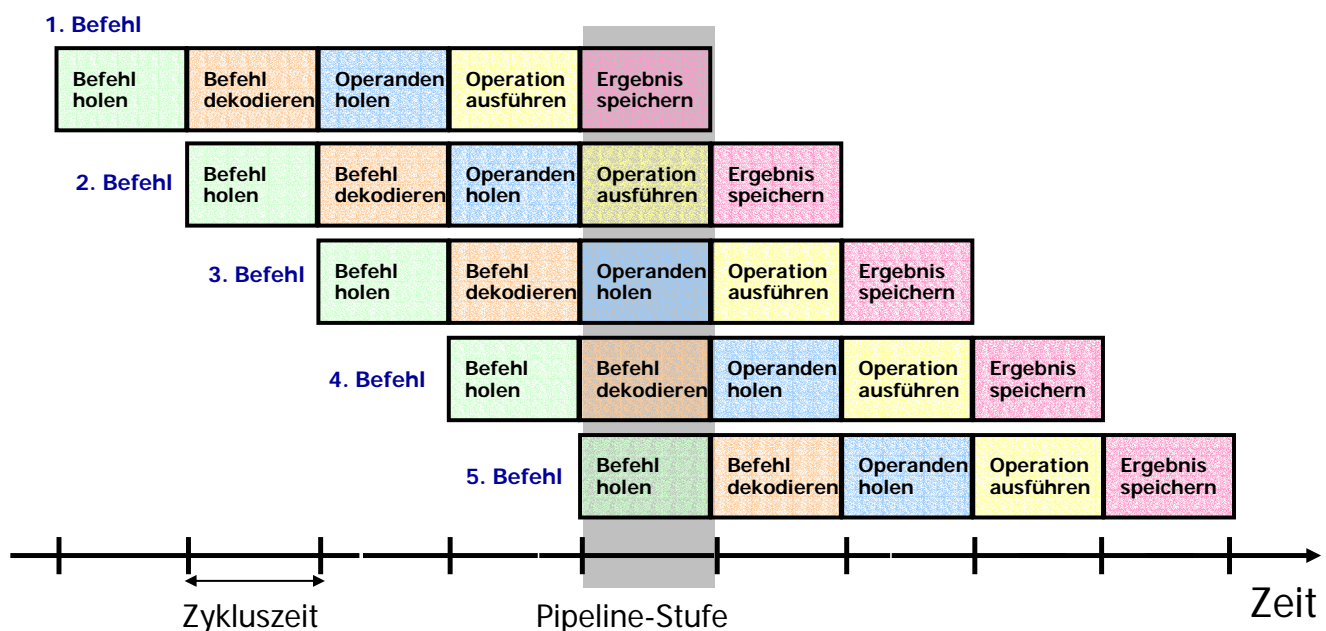
Alle noops wurden  
beseitigt

6 Taktzyklen pro  
Schleifendurchlauf



## Aufgabe 3

Gegeben sei eine 5-stufige Pipeline:



# Aufgabe 3

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

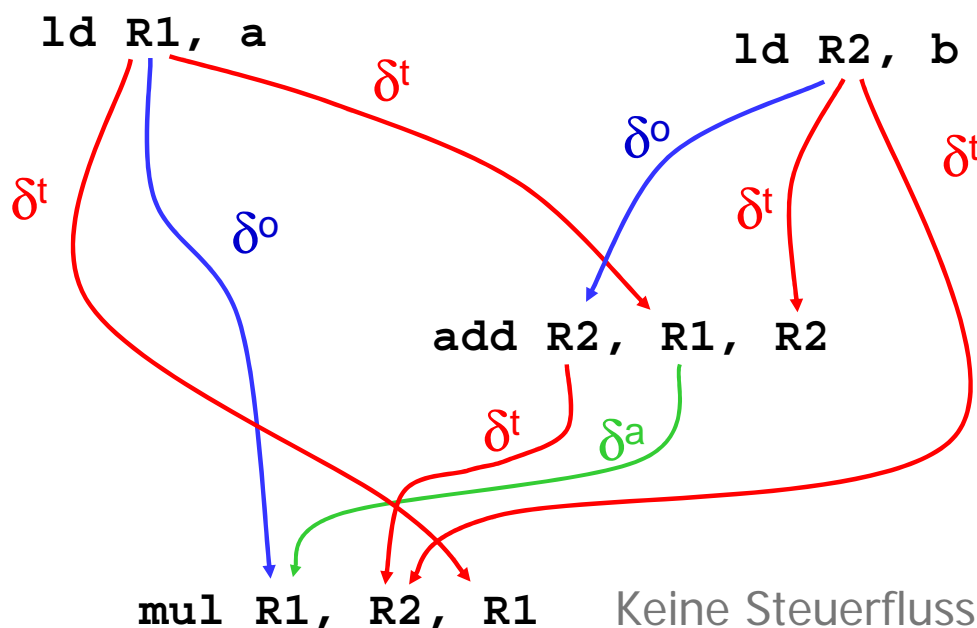
```
m1:  ld    R1, a      ; R1 := [a]
m2:  ld    R2, b      ; R2 := [b]
m3:  add   R2, R1, R2  ; R2 := R1+R2
m4:  mul   R1, R2, R1  ; R1 := R1*R2
```

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



## Aufgabe 3.1

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



Keine Steuerflussabhängigkeit



## Aufgabe 3.2

2. Wieviele Pipelinekonflikte treten auf?

`ld R1, a`



`ld R2, b`



`add R2, R1, R2`



`mul R1, R2, R1`



## Aufgabe 3.3

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 3.3

ld R1, a

ld R2, b

noop

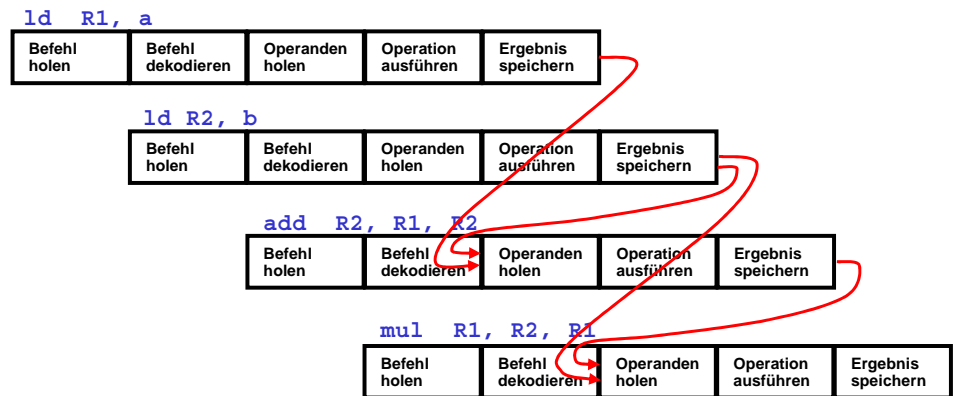
noop

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

Befehl holen	Befehl dekodieren	Operanden holen	Operation ausführen	Ergebnis speichern
-----------------	----------------------	--------------------	------------------------	-----------------------

ld R2, b

Befehl holen	Befehl dekodieren	Operanden holen	Operation ausführen	Ergebnis speichern
-----------------	----------------------	--------------------	------------------------	-----------------------

add R2, R1, R2

Befehl holen	Befehl dekodieren	Operanden holen	Operation ausführen	Ergebnis speichern
-----------------	----------------------	--------------------	------------------------	-----------------------

mul R1, R2, R1

Befehl holen	Befehl dekodieren	Operanden holen	Operation ausführen	Ergebnis speichern
-----------------	----------------------	--------------------	------------------------	-----------------------



# Aufgabe 4.4

*Keine !!!*

ld R1, a

Befehl holen	Befehl dekodieren	Operanden holen	Operation ausführen	Ergebnis speichern
-----------------	----------------------	--------------------	------------------------	-----------------------

ld R2, b

Befehl holen	Befehl dekodieren	Operanden holen	Operation ausführen	Ergebnis speichern
-----------------	----------------------	--------------------	------------------------	-----------------------

add R2, R1, R2

Befehl holen	Befehl dekodieren	Operanden holen	Operation ausführen	Ergebnis speichern
-----------------	----------------------	--------------------	------------------------	-----------------------

mul R1, R2, R1

Befehl holen	Befehl dekodieren	Operanden holen	Operation ausführen	Ergebnis speichern
-----------------	----------------------	--------------------	------------------------	-----------------------



# 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**



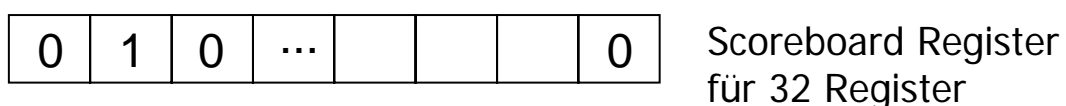
## Aufgabe 5.2

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.



- 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

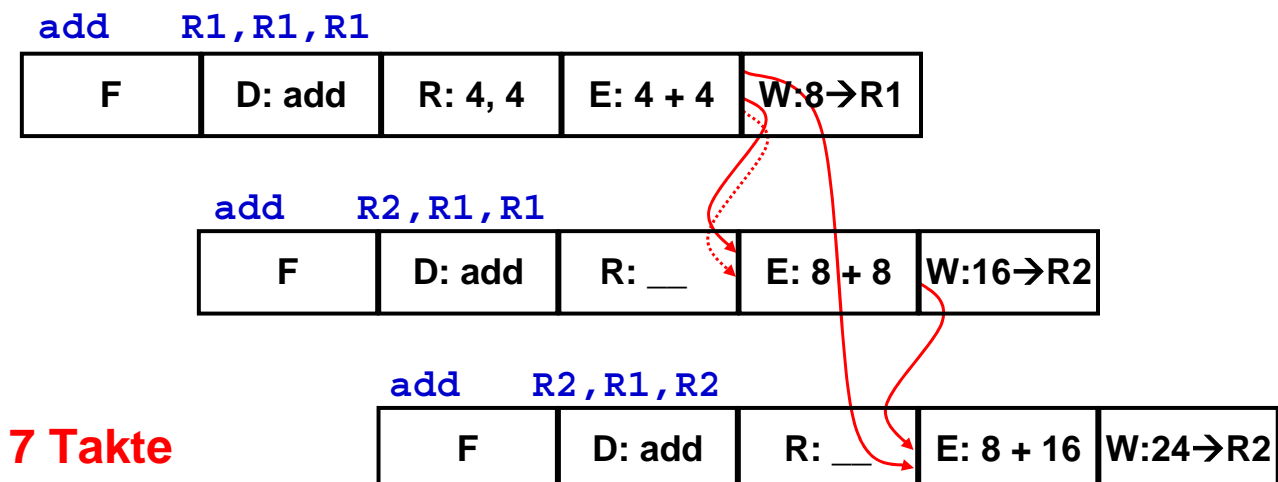
- Durch Prüfung der Scoreboard-Bits kann für jeden Befehl ein durch Datenabhängigkeit drohender Pipelinekonflikt erkannt werden.
- Behandlung des Konflikts durch die Verzögerung der Ausführung des Befehls, dessen Operandenregister ein gesetztes Scoreboard-Bit besitzt, bis zum Rücksetzen des Bits.

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?



# 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<sub>i</sub>**, **m<sub>j</sub>**, **m<sub>k</sub>**, ... 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.

