

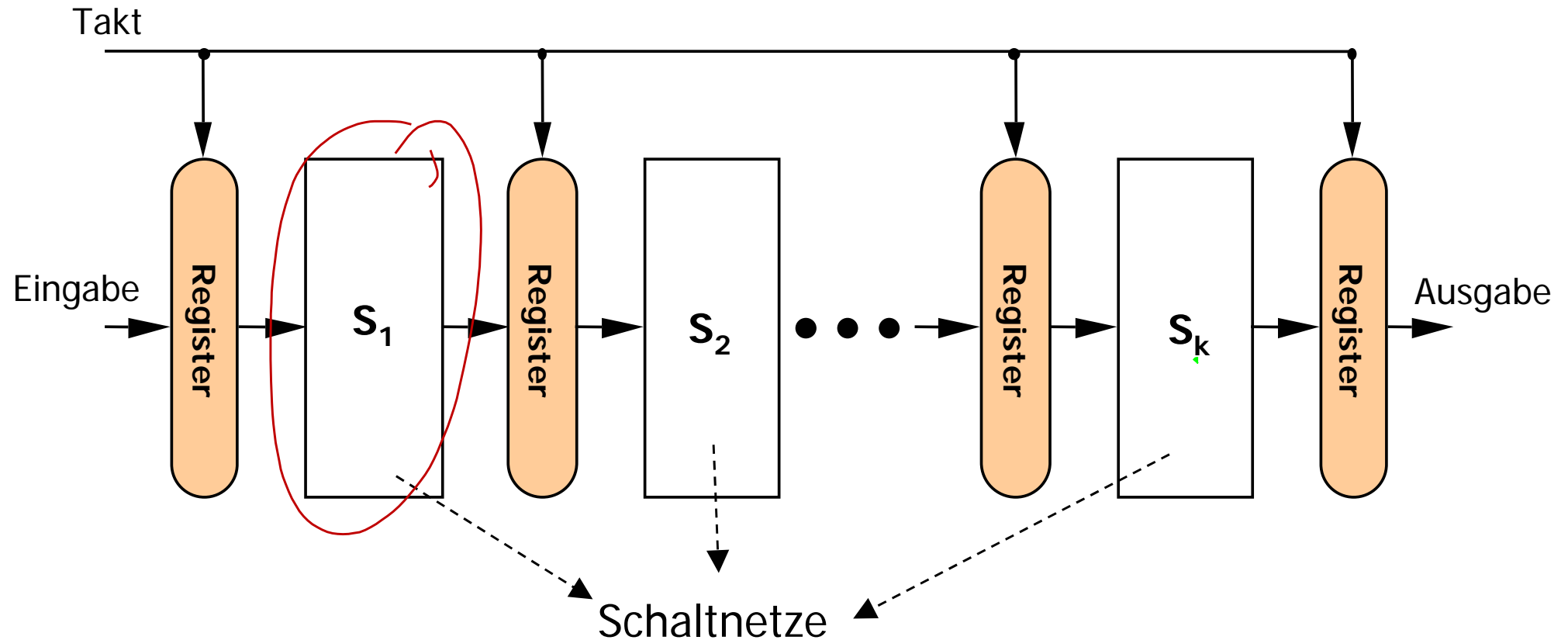
# Kapitel 5

---

## Pipeline-Verarbeitung



## 5.2 Pipeline-Stufen und Pipeline-Register



Verzögerungszeiten:

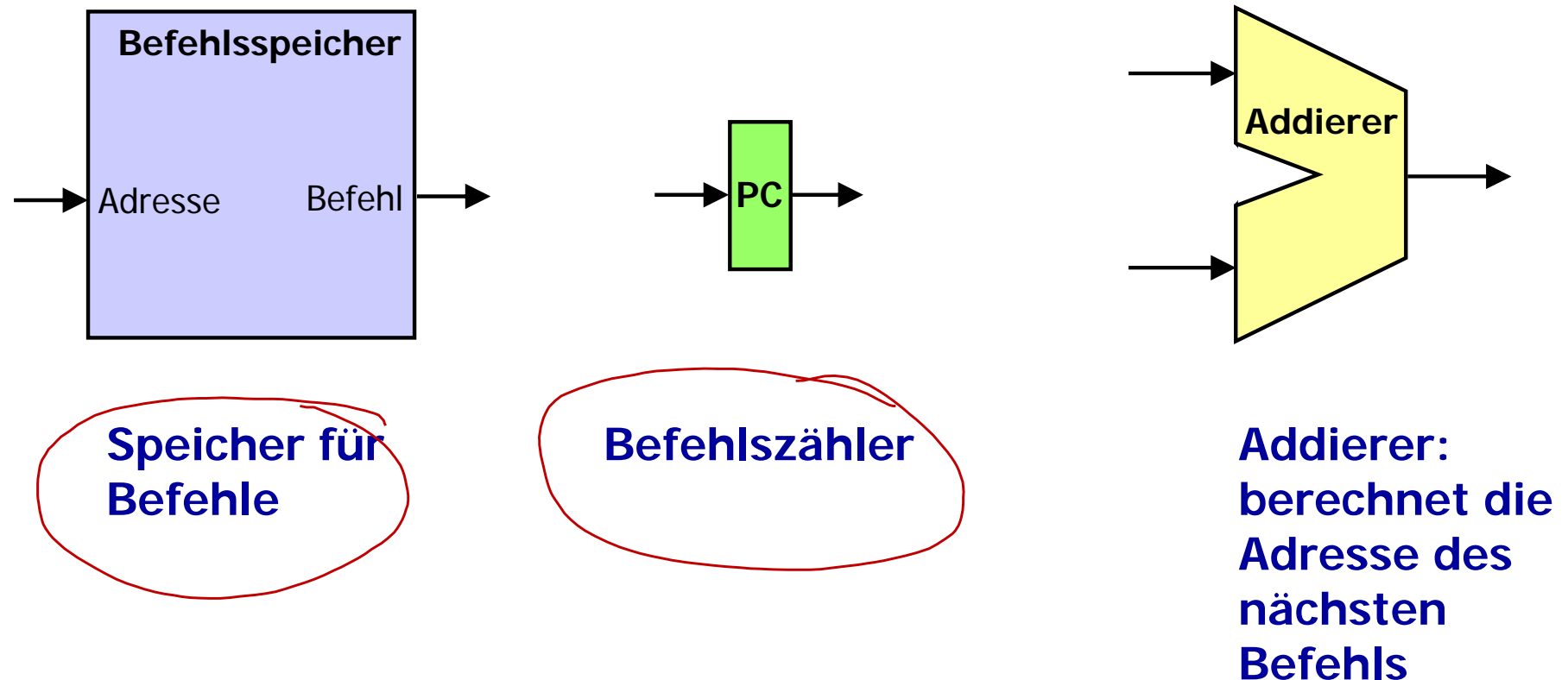
- der Schaltnetze:  $\tau_i$  ( $i = 1, \dots, k$ )
- der Pipeline-Register:  $\tau_{reg}$

Länge eines Taktzyklus:

$$\tau = \max\{\tau_1, \tau_2, \dots, \tau_k\} + \tau_{reg}$$

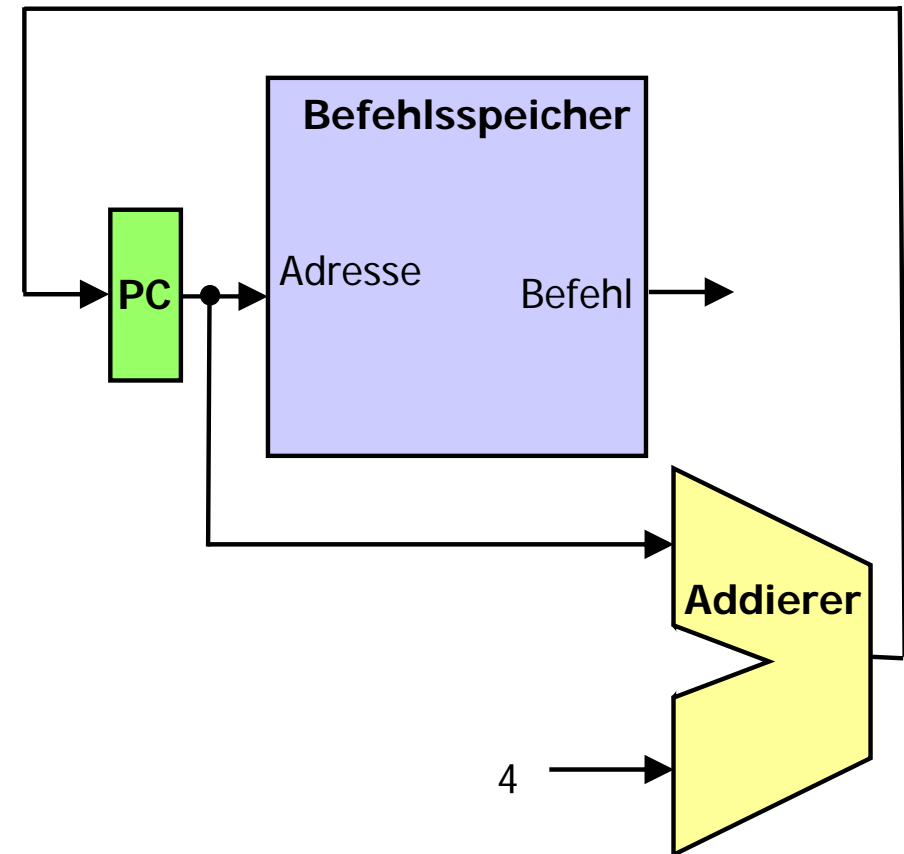
## 5.3 Befehlsabarbeitung und Datenpfade der MIPS-Befehle

- Welche Hardware-Komponenten sind zur Ausführung der **MIPS-Befehle** notwendig?
  - Für alle Befehlsklassen werden folgende Komponenten benötigt:



## 5.3 Befehlsabarbeitung und Datenpfade

- Befehl aus dem Befehlsspeicher holen
  - Befehl im Befehlsspeicher adressieren
  - Befehlszähler um 4 inkrementieren

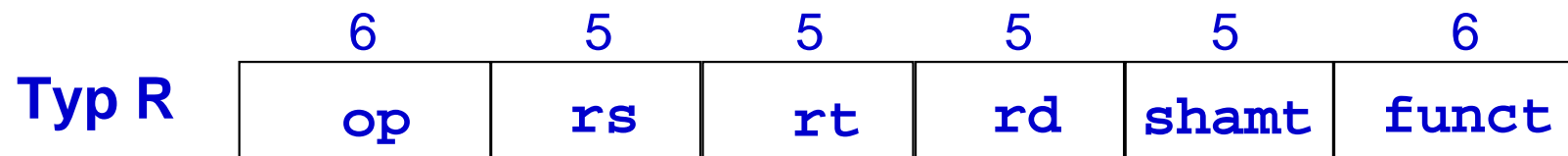


**Einheit für das Holen von Befehlen**

## 5.3 Befehlsabarbeitung und Datenpfade

□ Befehle vom R-Typ: **opcode**  $r_z$ ,  $r_m$ ,  $r_n$

- Arithmetisch logische Befehle: **add**, **sub**, **and** **or**
- Vergleichsbefehle: **slt**

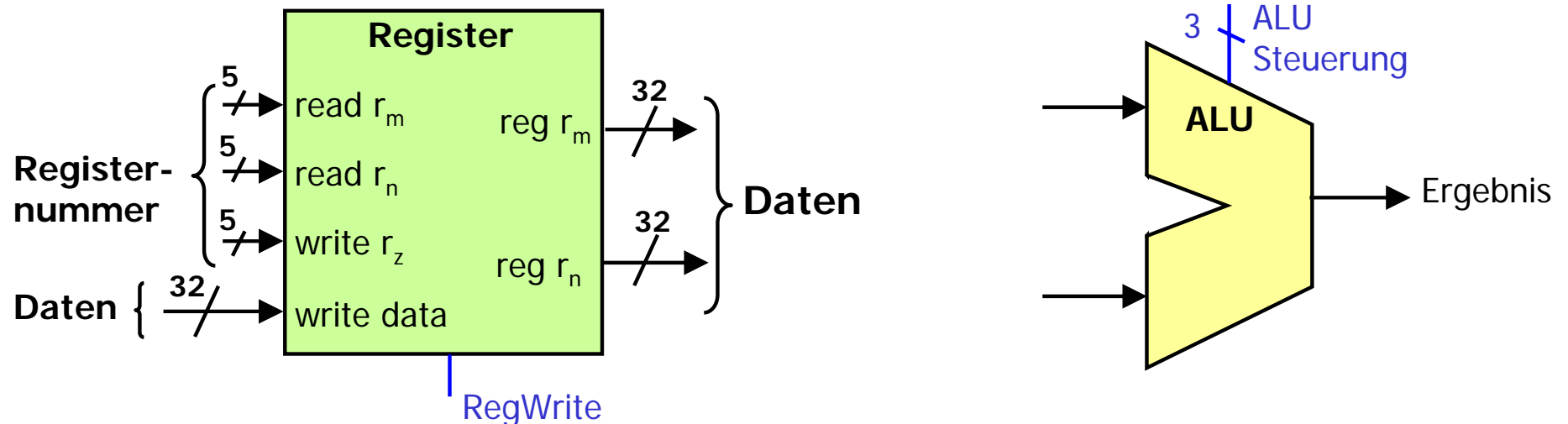


- Befehle haben 3 Operanden
- Operanden stehen in Registern
- Lesezugriff auf beiden Operanden-Register und
- ein Schreibzugriff auf das Zielregister

## 5.3 Befehlsabarbeitung und Datenpfade

### □ Befehle vom R-Typ:

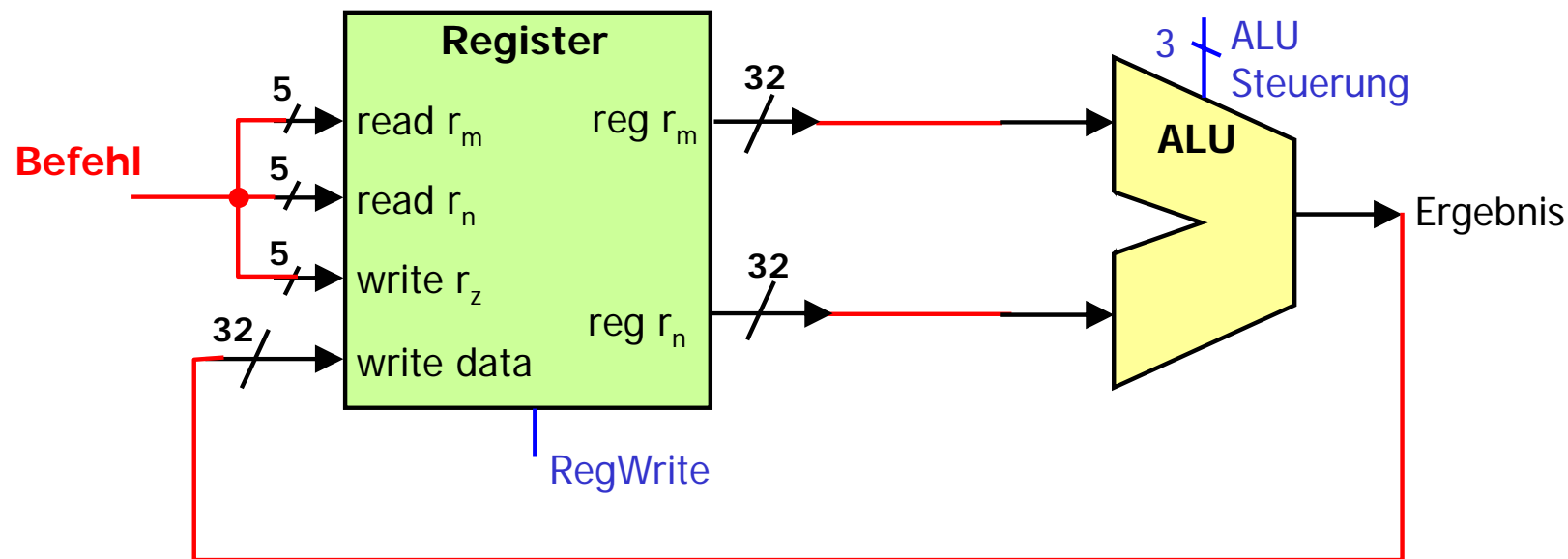
**opcode**  $r_z$ ,  $r_m$ ,  $r_n$



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

## 5.3 Befehlsabarbeitung und Datenpfade

### □ Befehle vom R-Typ:



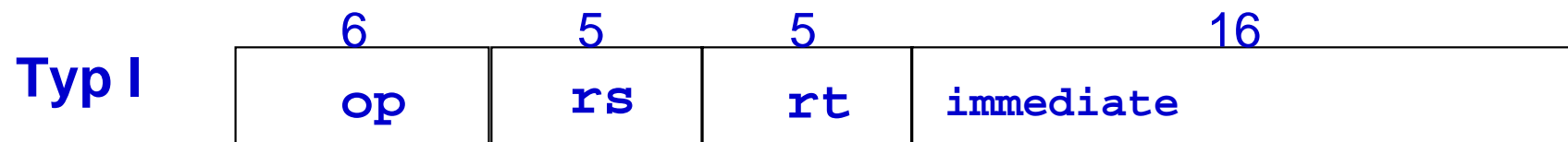
- Lesezugriff auf beiden Operanden-Register und
- ein Schreibzugriff auf das Zielregister

## 5.3 Befehlsabarbeitung und Datenpfade

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

**lw**  $r_z$ , **offset**( $r_m$ )

**sw**  $r_n$ , **offset**( $r_m$ )



- Speicheradresse wird durch die Addition einer Basisadresse im Register  $r_m$  zu einem vorzeichenbehafteten 16-bit offset berechnet
- Bei Speicherbefehlen wird das zu speichernde Wort aus dem Register  $r_n$  gelesen. Bei Ladebefehlen wird das geladene Wort ins Register  $r_z$  geladen

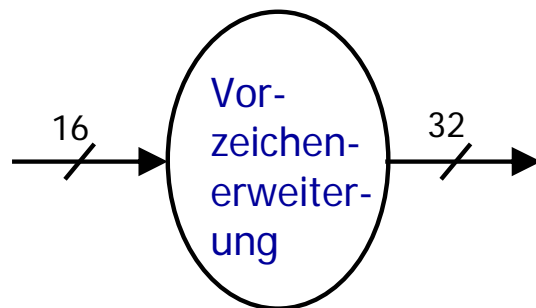
## 5.3 Befehlsabarbeitung und Datenpfade

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

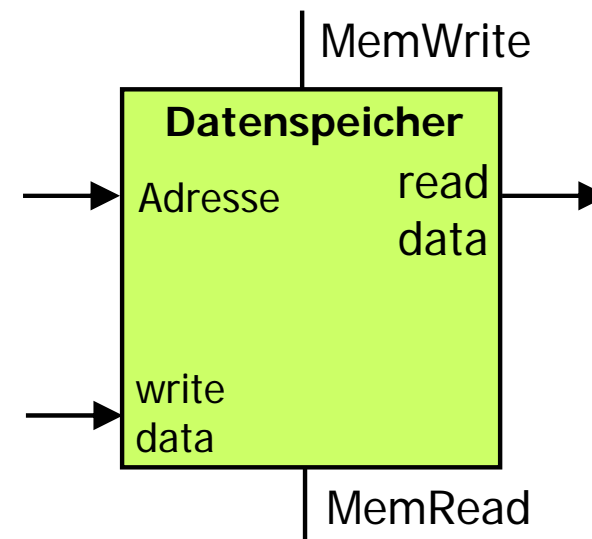
**lw  $r_z$ , offset( $r_m$ )**

**sw  $r_n$ , offset( $r_m$ )**

➤ Weitere benötigte Komponenten:



**Vorzeichen-  
erweiterungs-  
einheit**

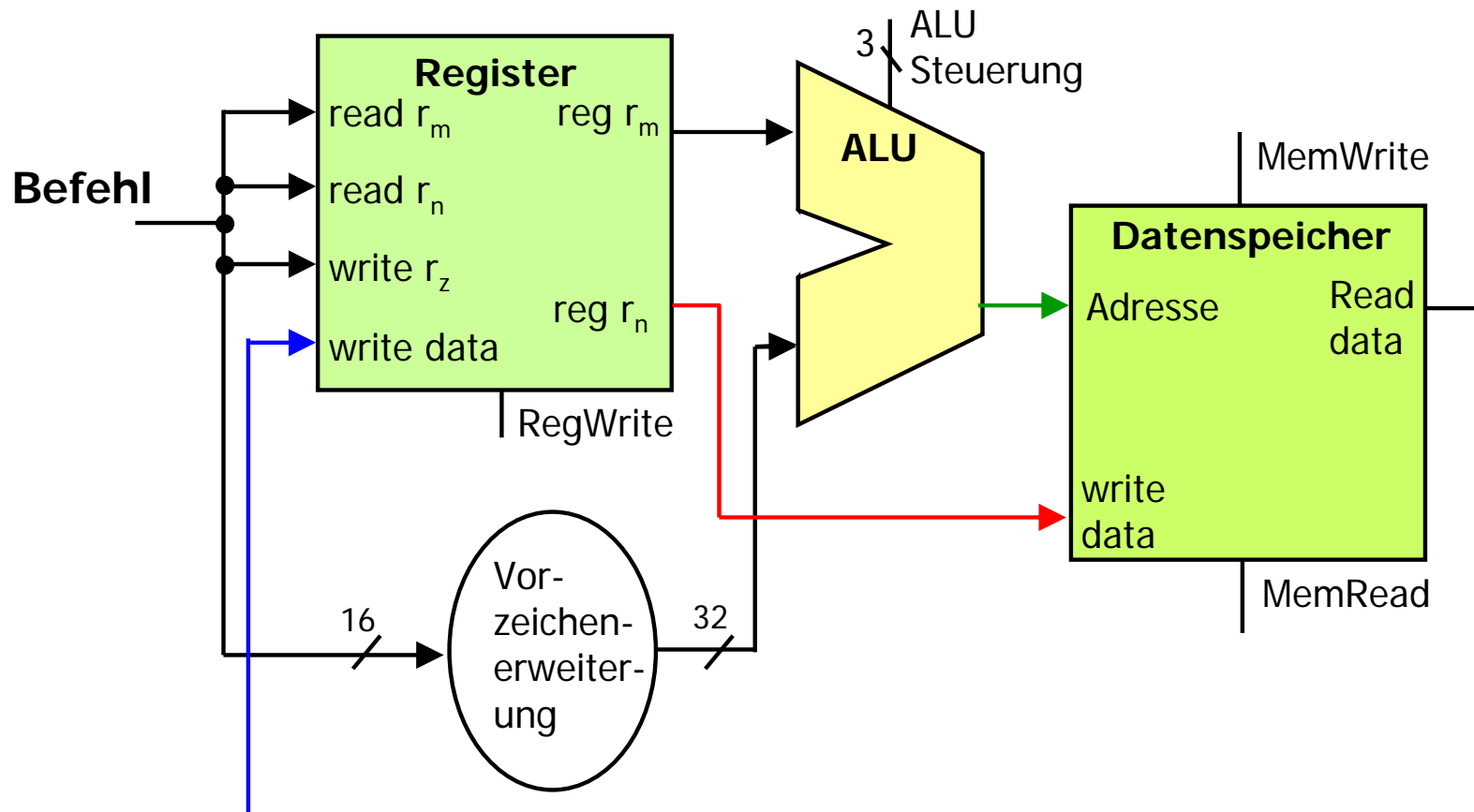


**Datenspeicher:**

Steuersignale für Lese- (read data)  
und Schreibzugriffe (write data)

## 5.3 Befehlsabarbeitung und Datenpfade

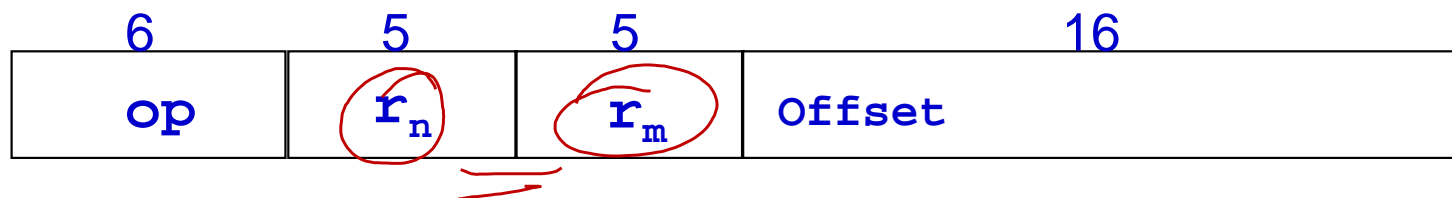
### □ Lade- und Speicherbefehle (*load and store*)



## 5.3 Befehlsabarbeitung und Datenpfade

### □ Verzweigungsbefehle

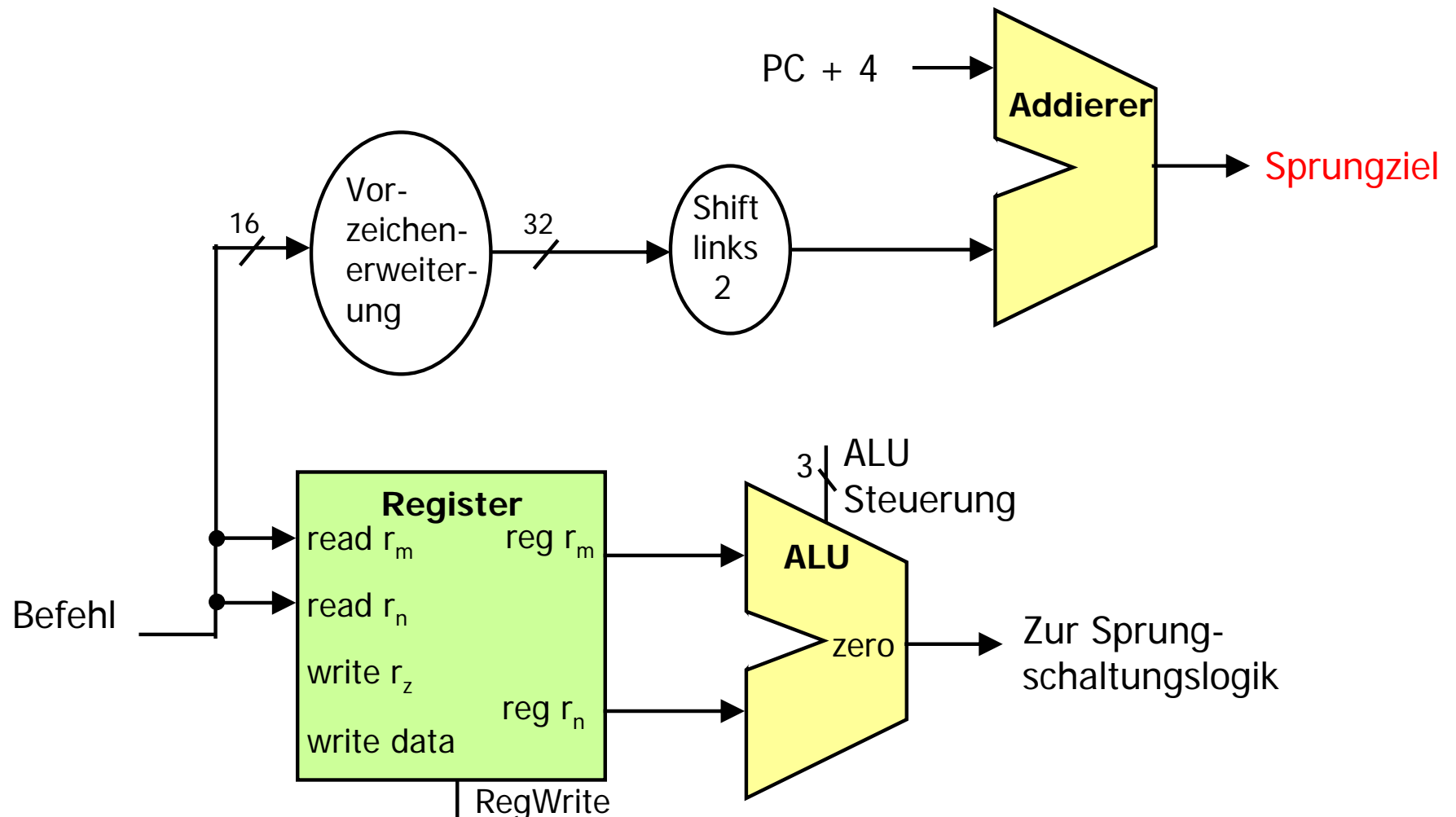
**beq  $r_n$ ,  $r_m$ , offset**



- 16-bit vorzeichenbehaftetes Offset
  - ➡  $2^{15}-1$  Befehle vorwärts und  $2^{15}$  rückwärts
- Basisadresse zur Berechnung der Sprungadresse ist die Adresse des Befehls hinter dem Verzweigungsbefehl, d. h. (PC+4)
  - Offset: um 2 Bits nach links verschieben, um Wörter zu adressieren
- Ergebnis des Vergleichs von  $r_n$  und  $r_m$  entscheidet, ob der Sprung „genommen“ bzw. „nicht genommen“ wird

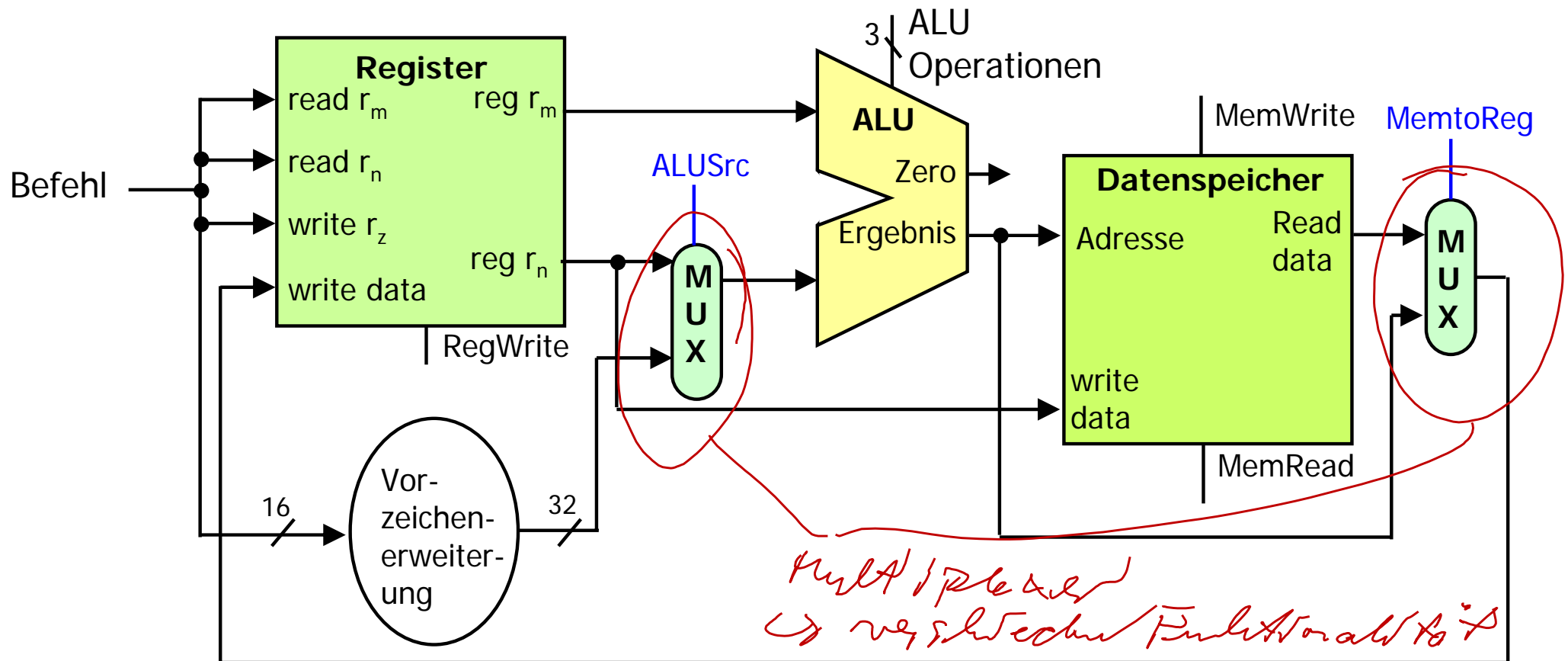
## 5.3 Befehlsabarbeitung und Datenpfade

### □ Verzweigungsbefehle

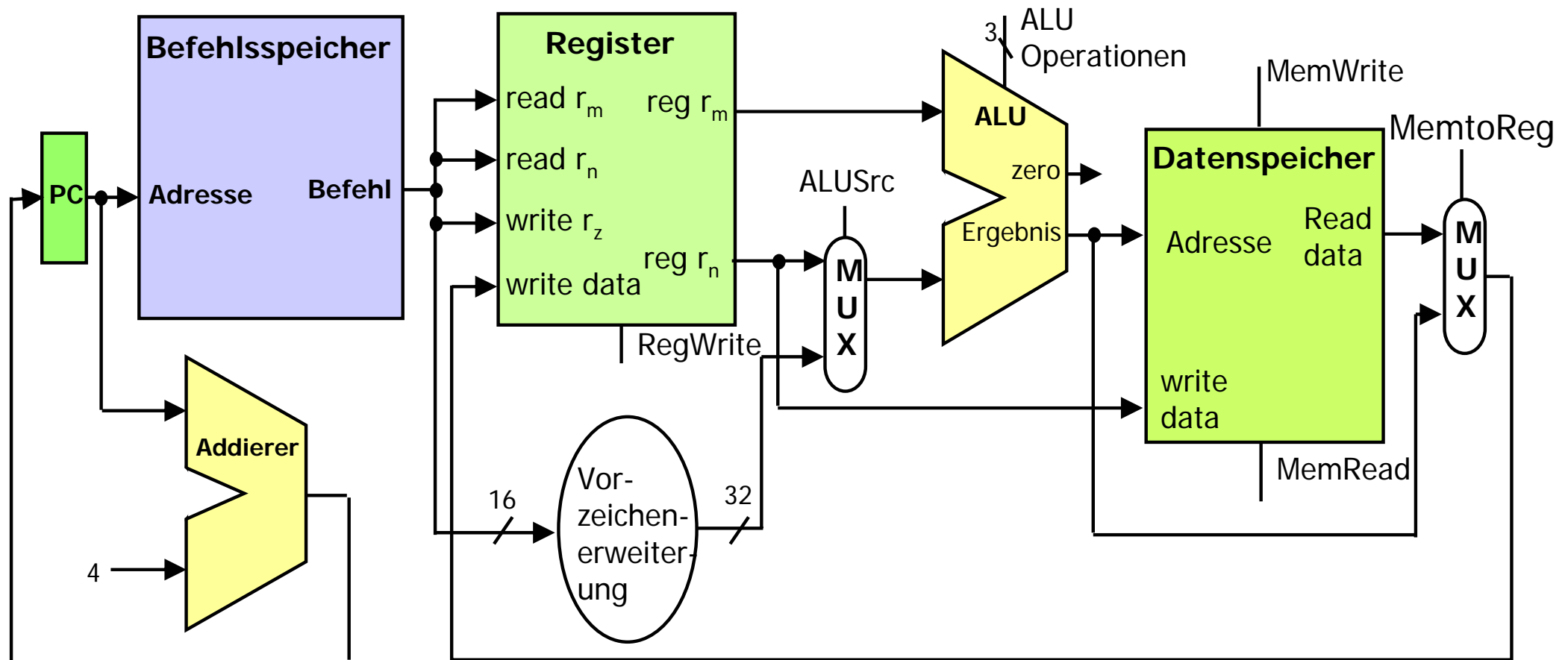


## 5.3 Befehlsabarbeitung und Datenpfade

### □ Datenpfad für Lade-Speicherbefehle und Befehle vom R-Typ



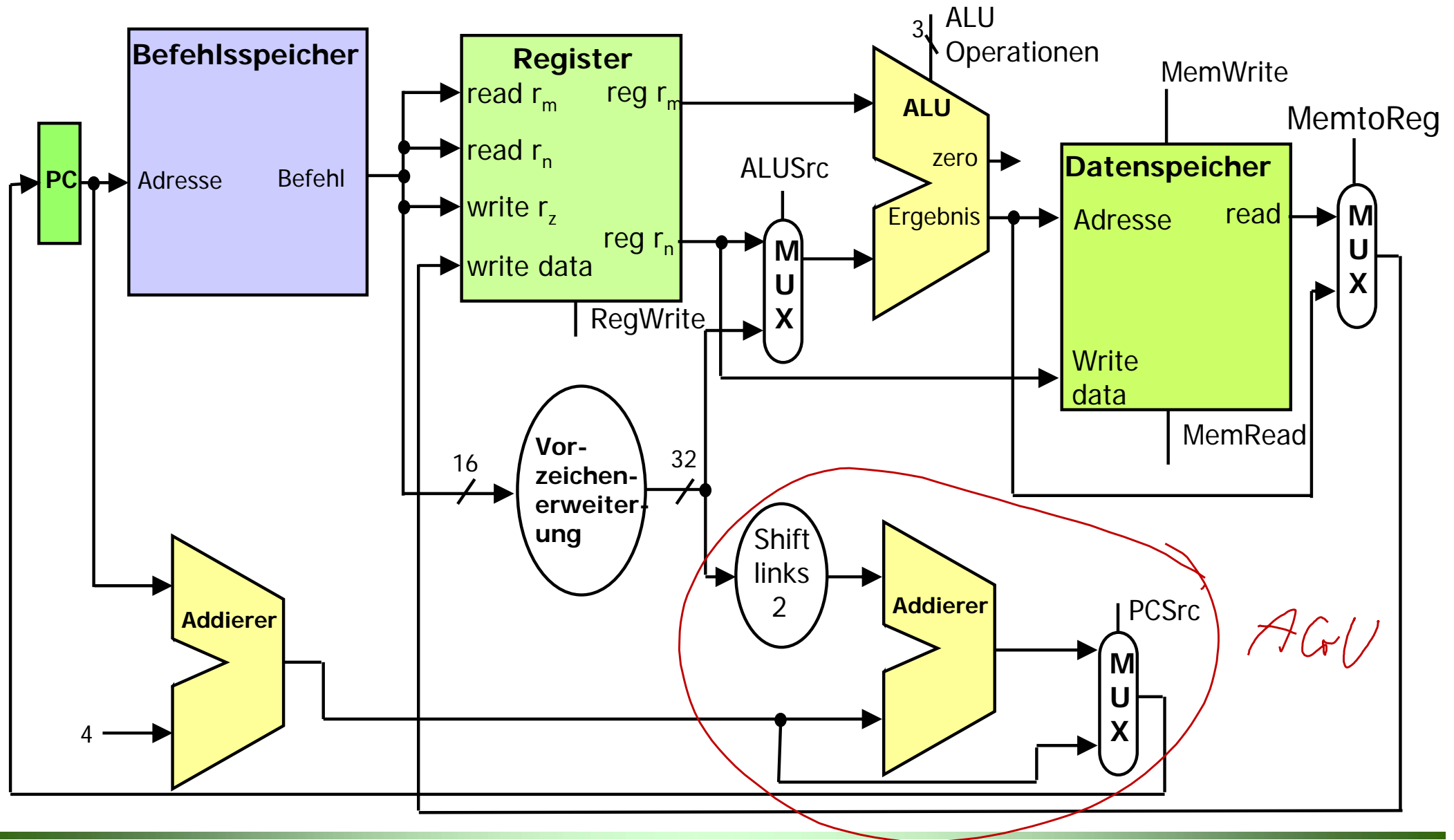
## 5.3 Befehlsabarbeitung und Datenpfade



Einheit für das  
Holen von Befehlen

Datenpfade für Lade-Speicherbefehle  
und Befehle vom R-Typ

# Datenpfad für die MIPS-Architektur



# Erinnerung: MIPS-Befehlsformate

## ➤ R-Typ Befehl

0	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

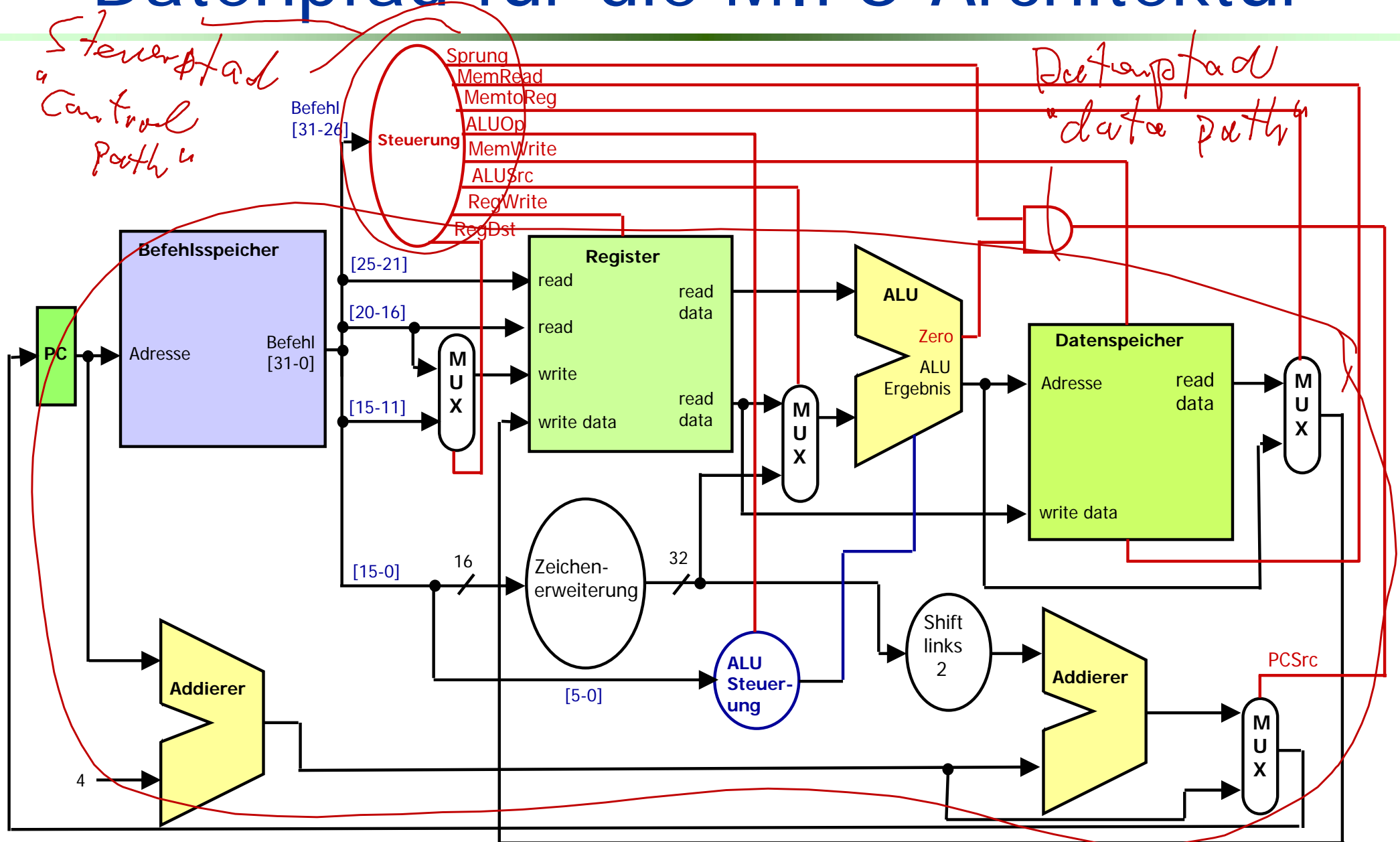
## ➤ Lade/Speicher Befehl

35 oder 43	rs	rt	Adresse
31-26	25-21	20-16	15-0

## ➤ Sprung Befehl

4	rs	rt	Adresse
31-26	25-21	20-16	15-0

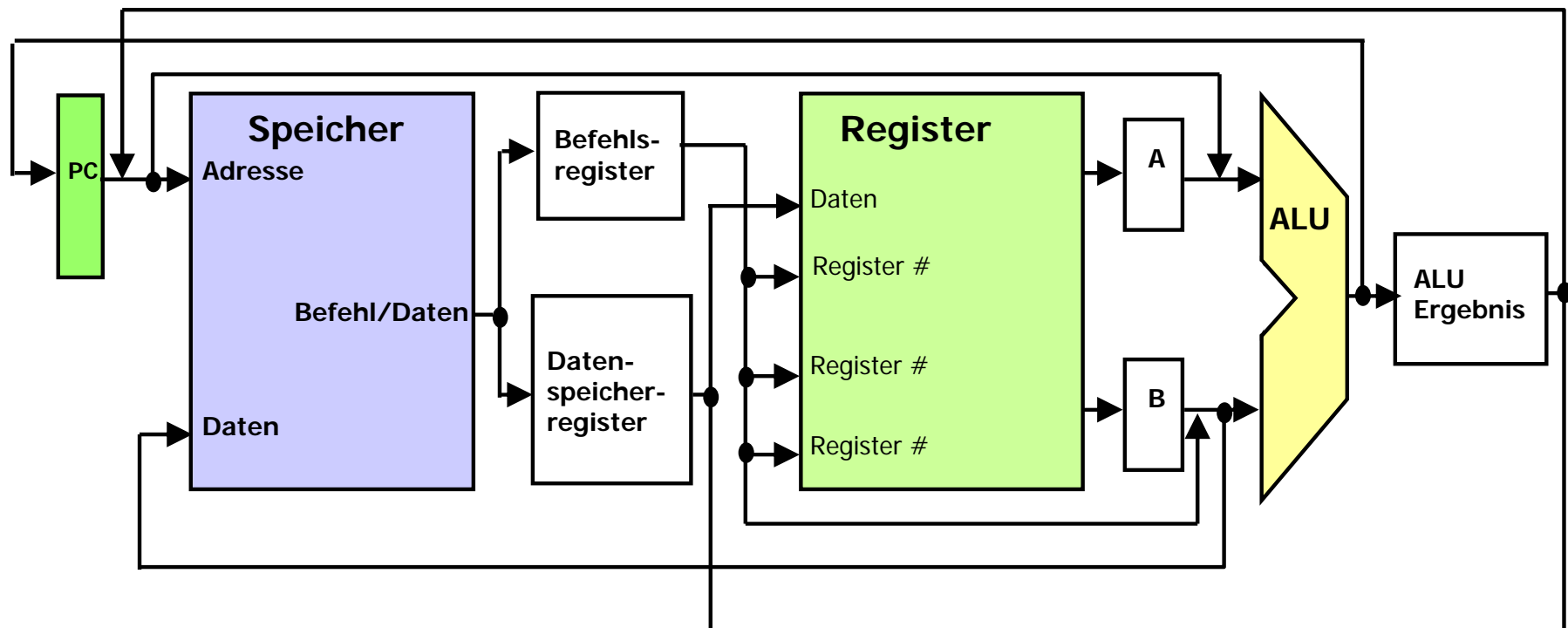
# Datenpfad für die MIPS-Architektur



# 5.4 Pipelining in MIPS-Architektur

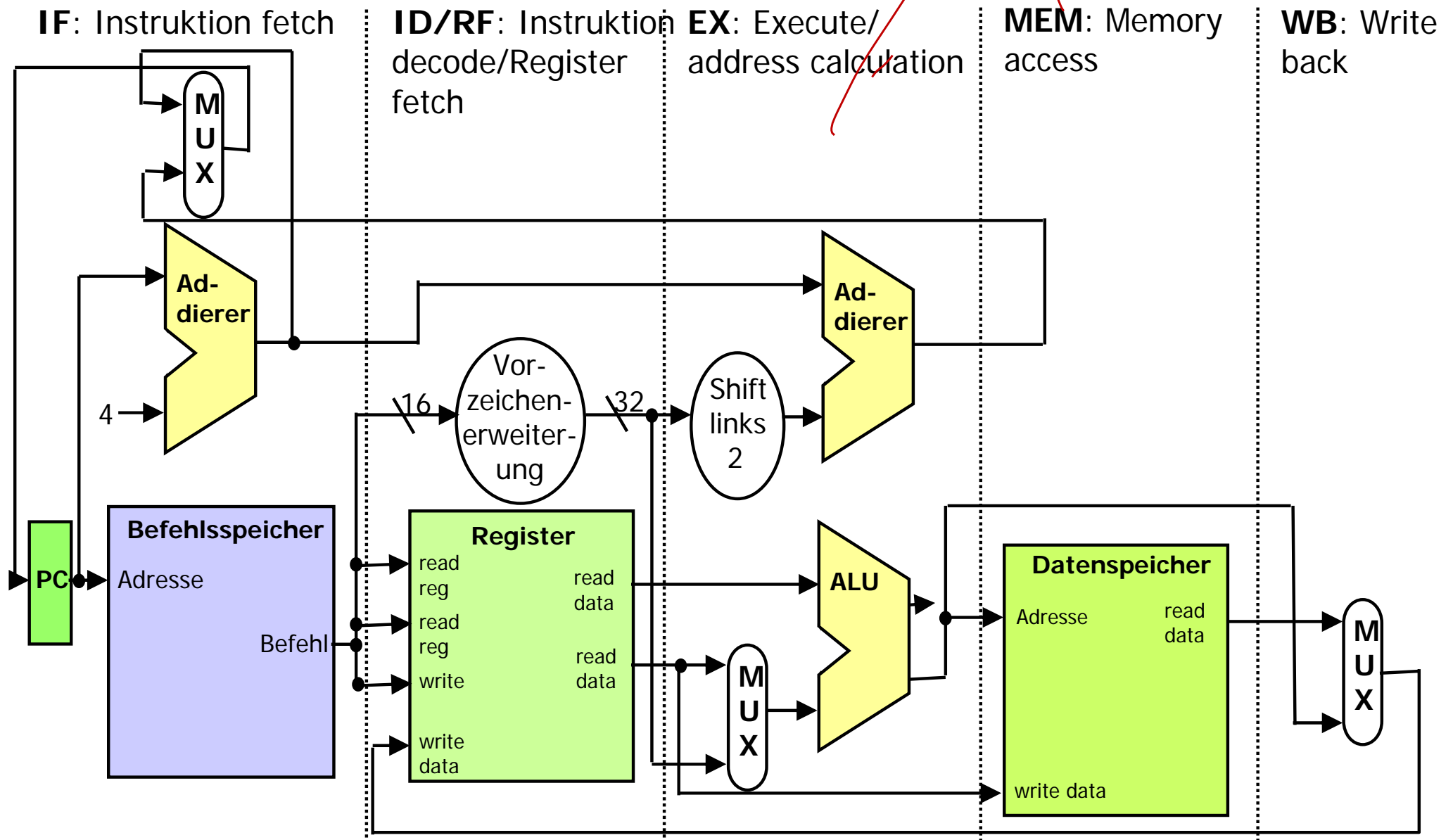
## □ Schlüsseleinheiten des Datenpfads

- Eine ALU
- Eine Speichereinheit für Daten und Befehle
- Register: Befehlsregister, Datenspeicherregister, A, B und ALU-Ergebnisregister

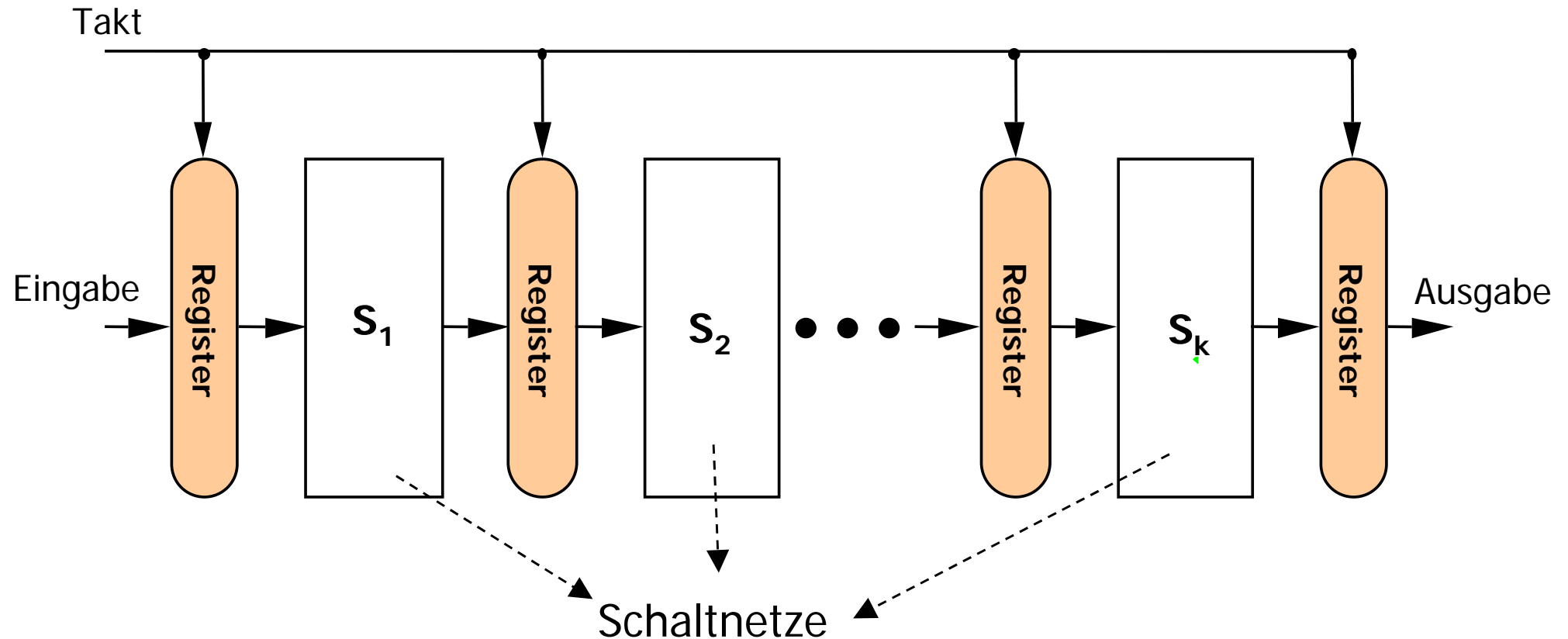


# 5.4 Pipelining in MIPS-Architektur

*am meisten intensiven!*



# Pipeline-Stufen und Pipeline-Register



Verzögerungszeiten:

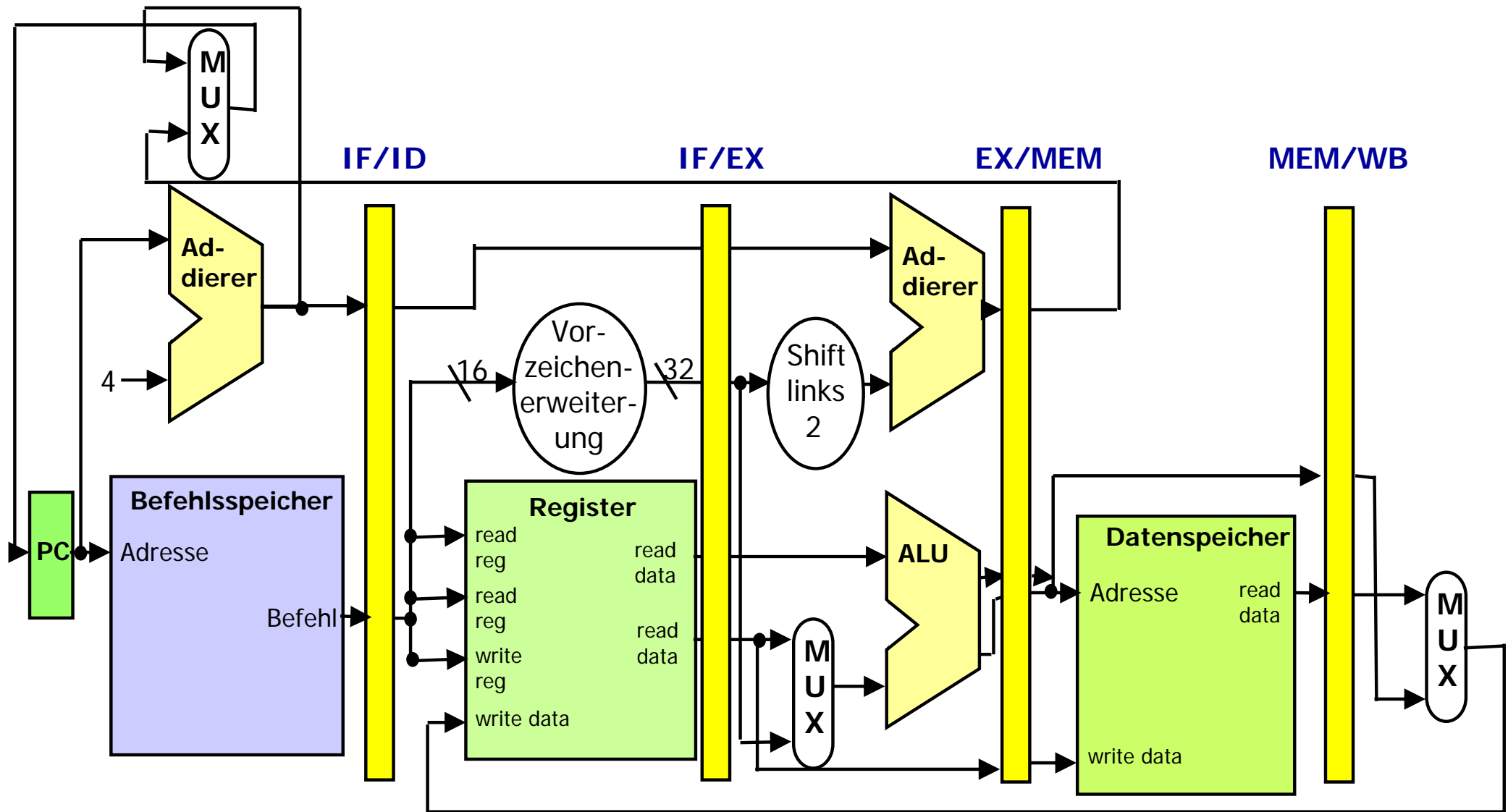
- der Schaltnetze:  $\tau_i$  ( $i = 1, \dots, k$ )
- der Pipeline-Register:  $\tau_{reg}$

Länge eines Taktzyklus:

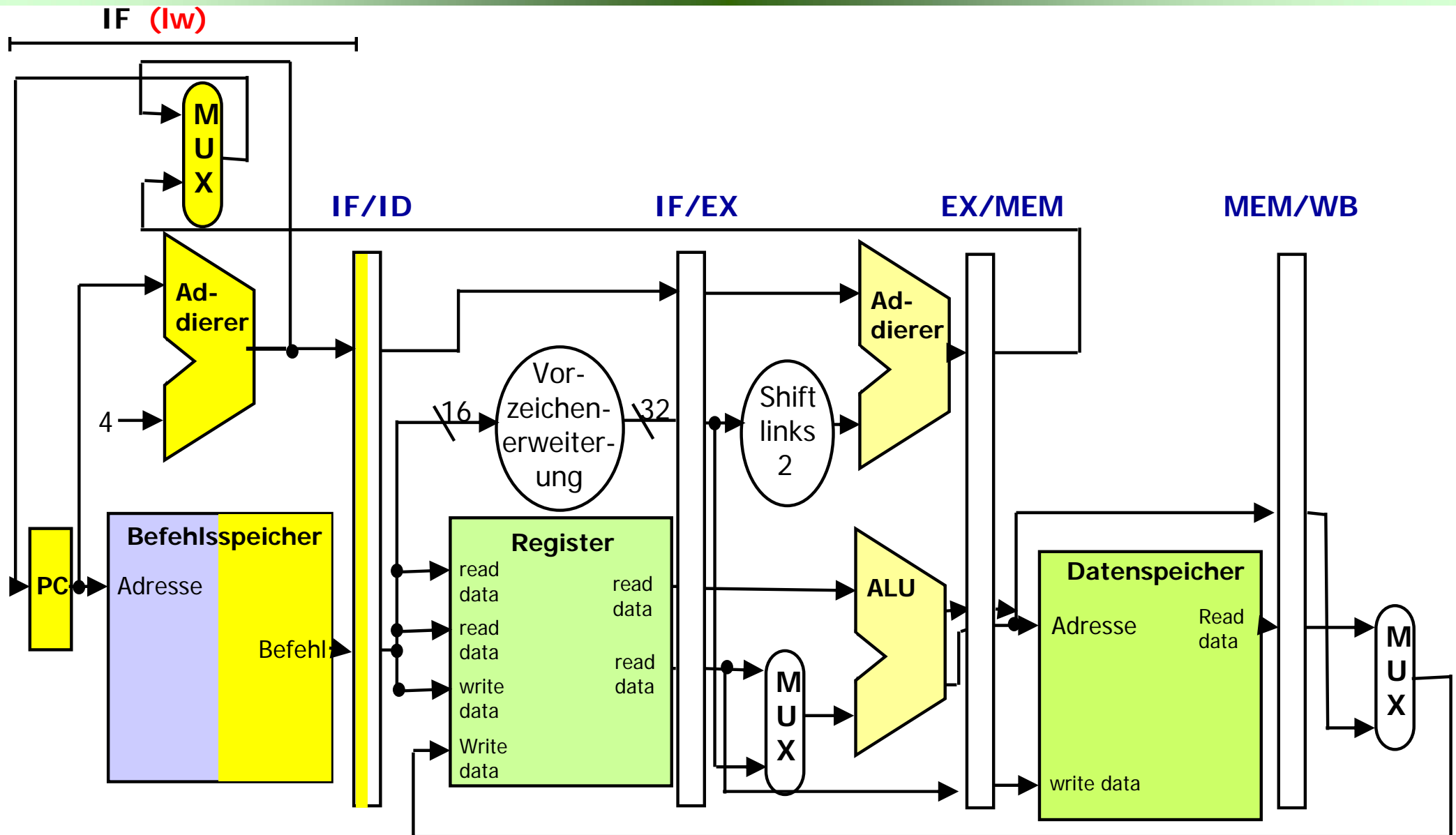
$$\tau = \max\{\tau_1, \tau_2, \dots, \tau_k\} + \tau_{reg}$$

*Pipelinstufen möglichst gleich groß gehalten/entworfen*

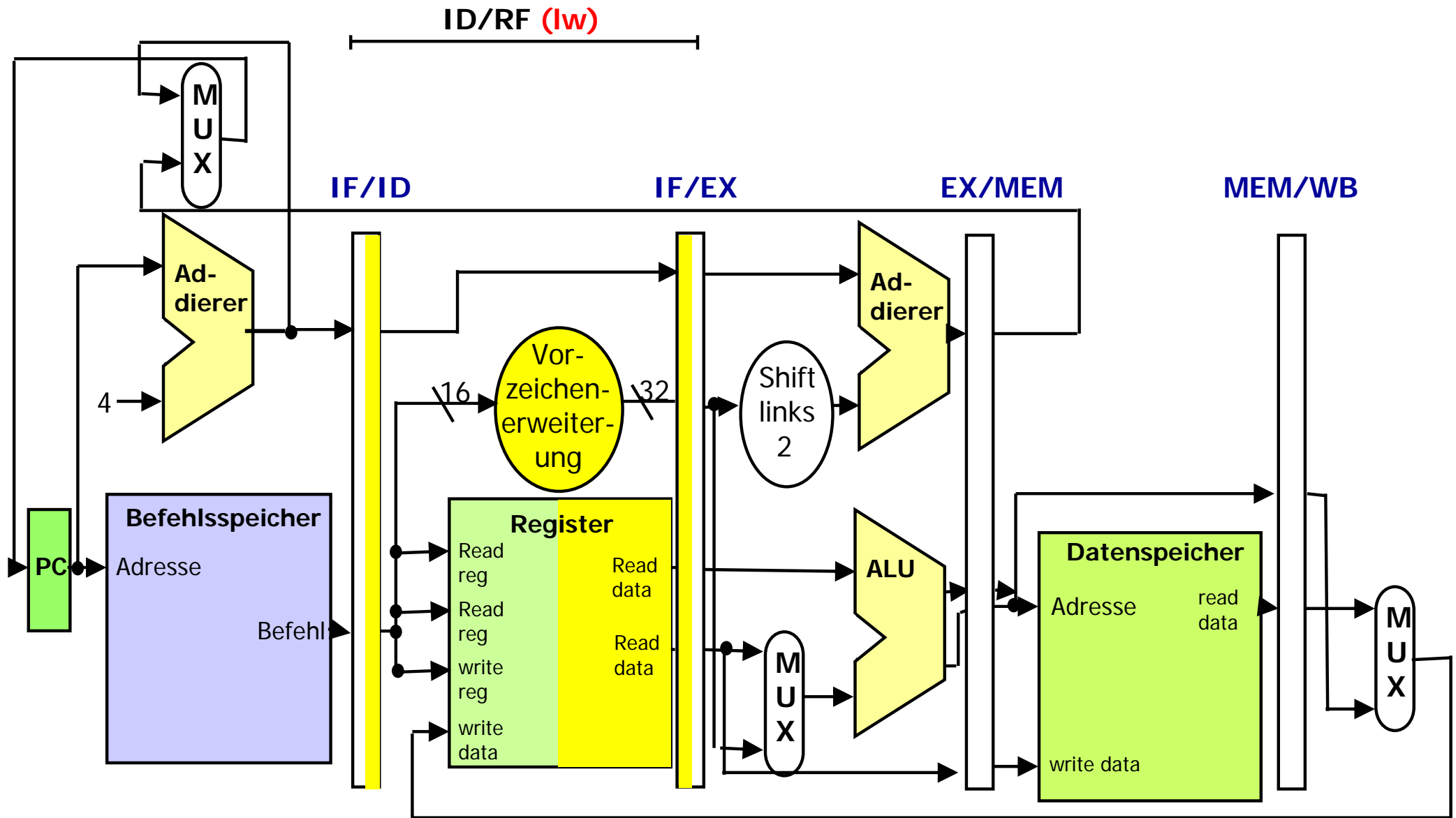
# 5.4 Pipelining in MIPS-Architektur



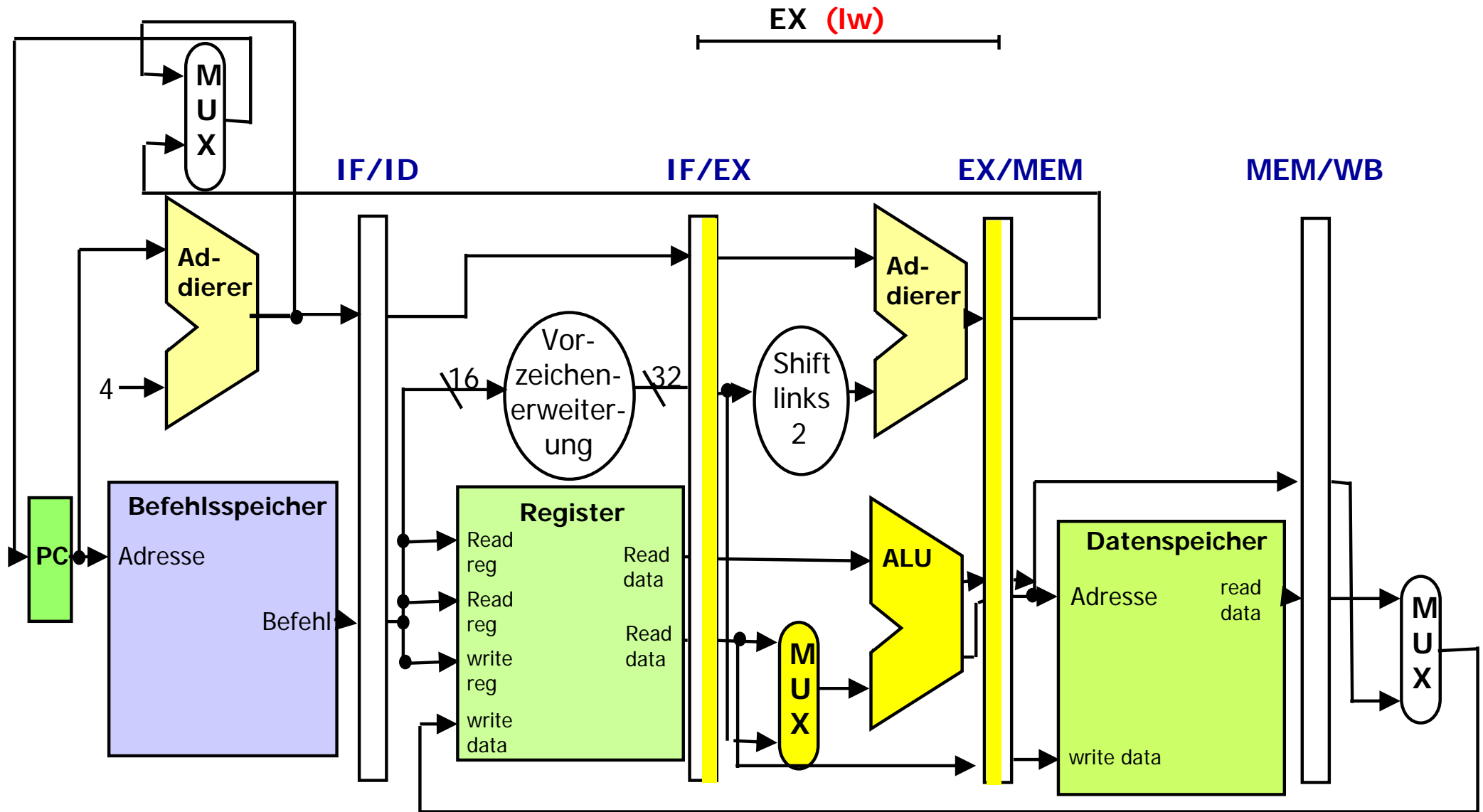
# 5.4 DLX-Pipelinstufen



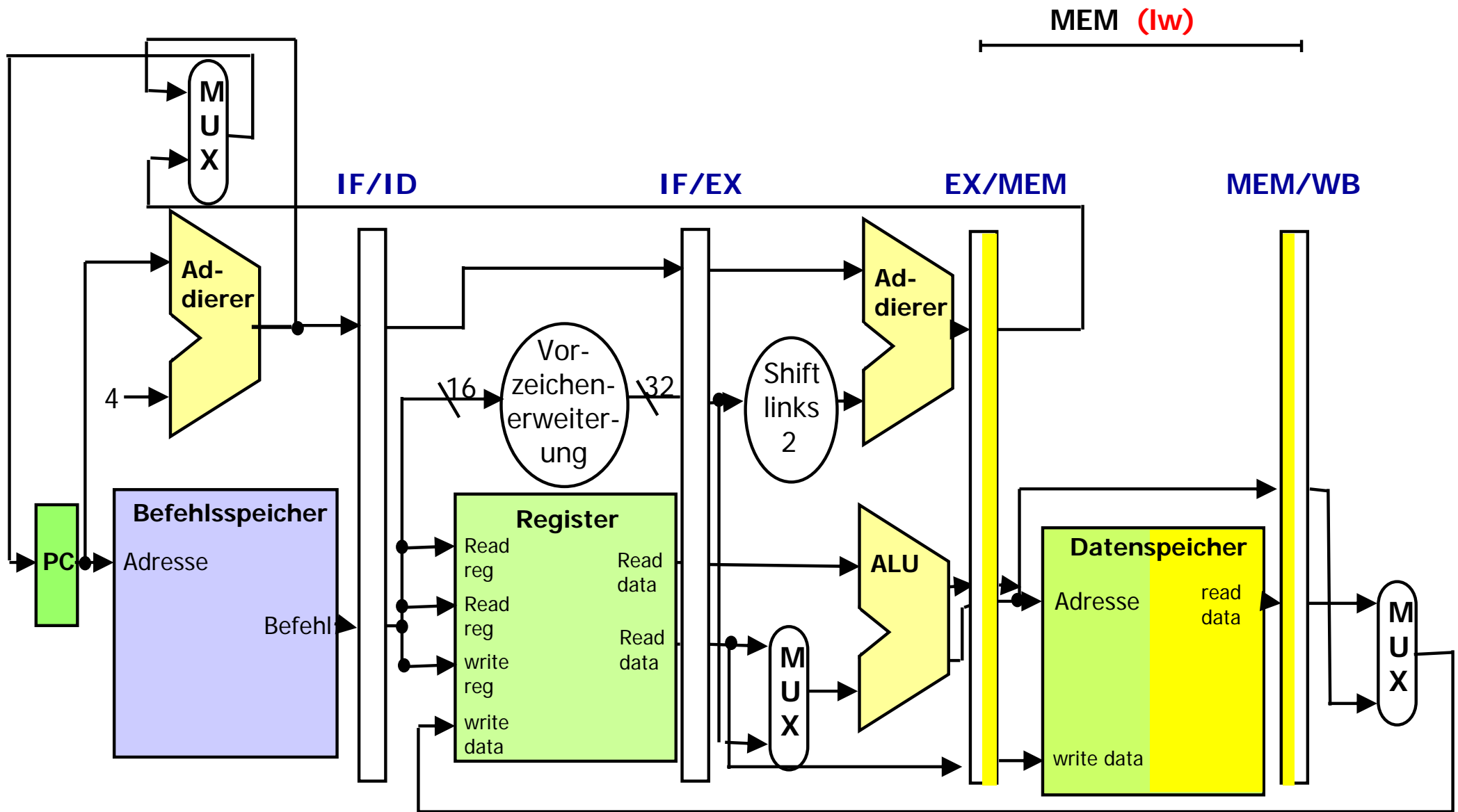
# 5.4 DLX-Pipelinstufen



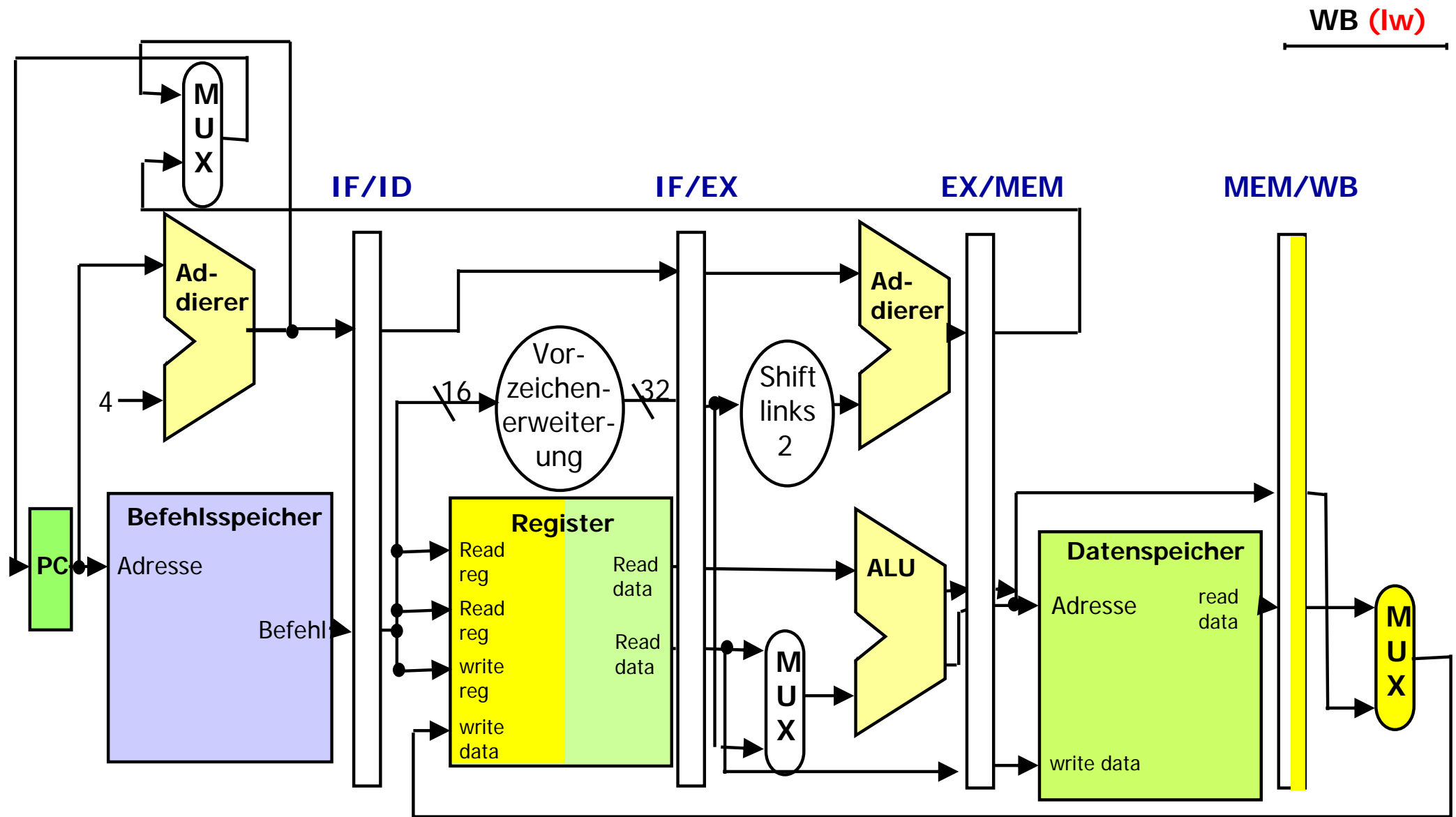
# 5.4 DLX-Pipelinstufen



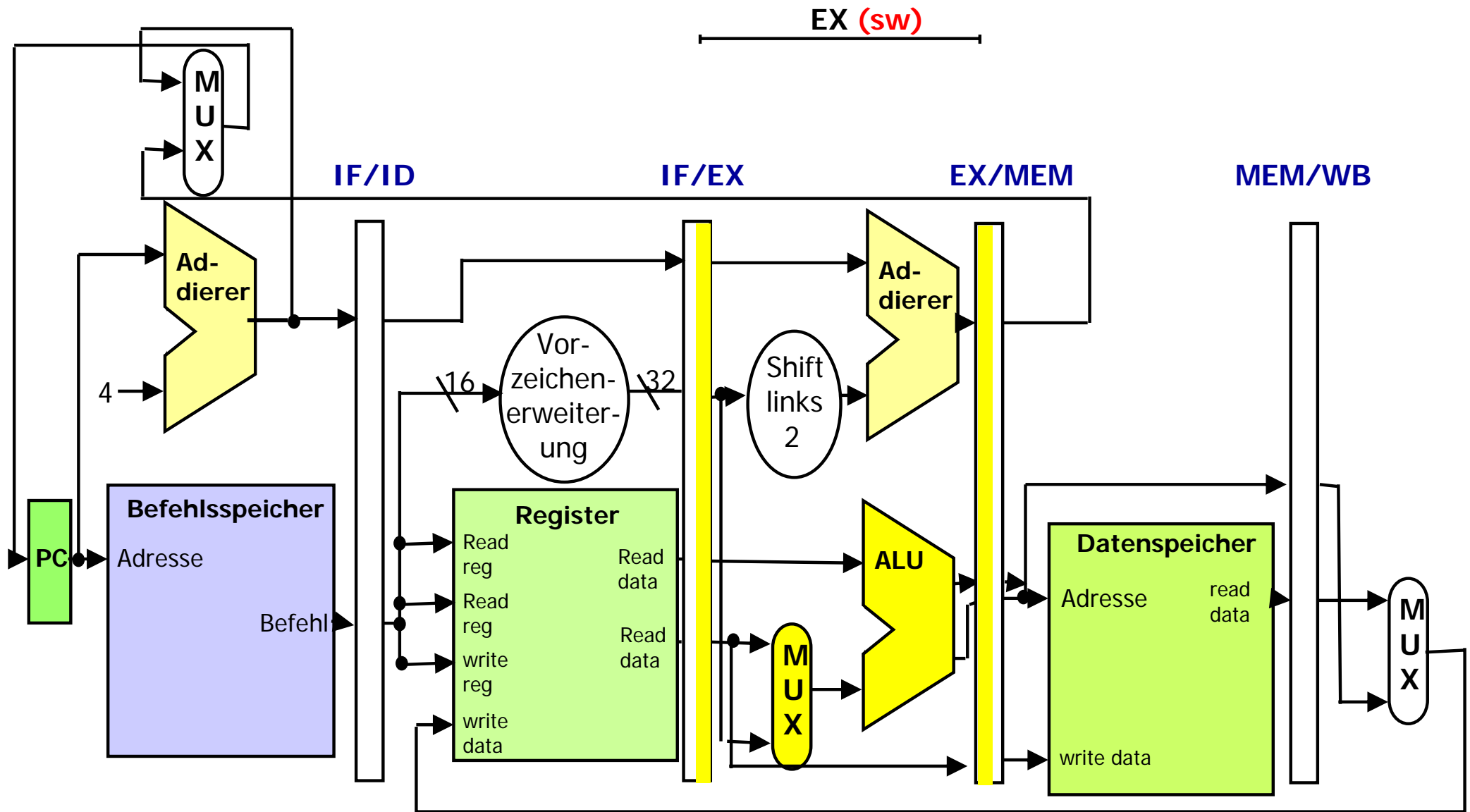
# 5.4 DLX-Pipelinstufen



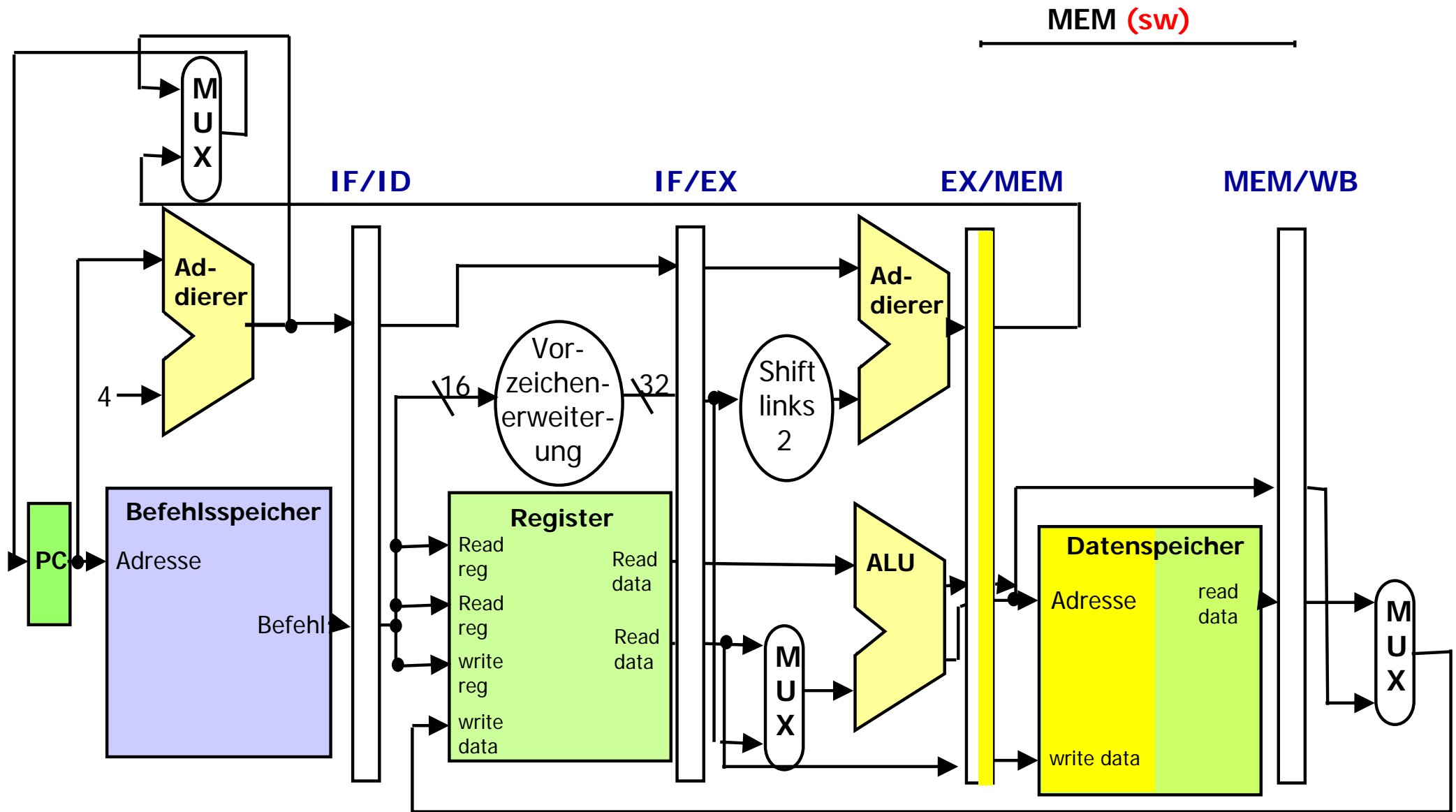
# 5.4 DLX-Pipelinstufen



# 5.4 DLX-Pipelinstufen

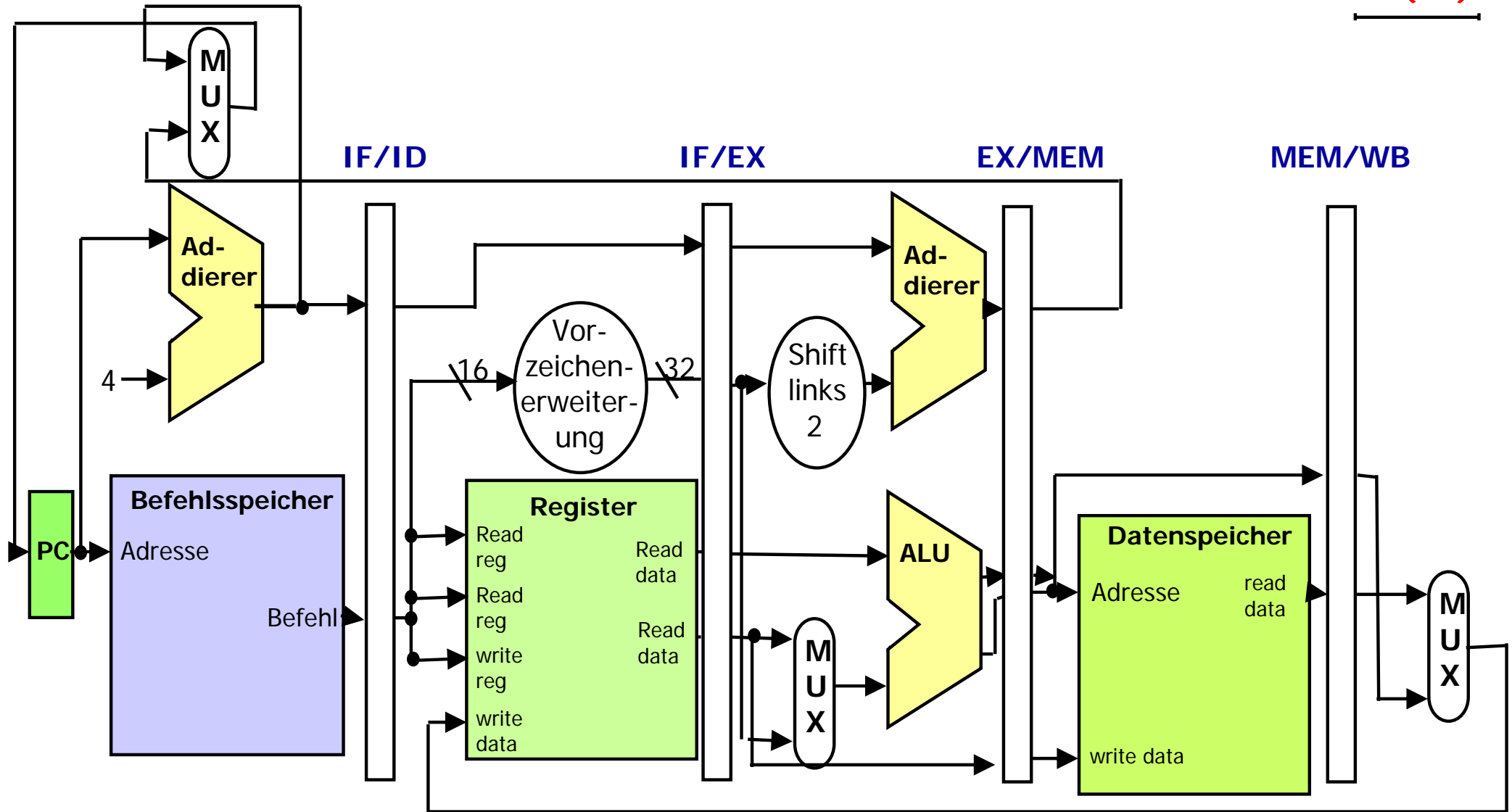


# 5.4 DLX-Pipelinstufen

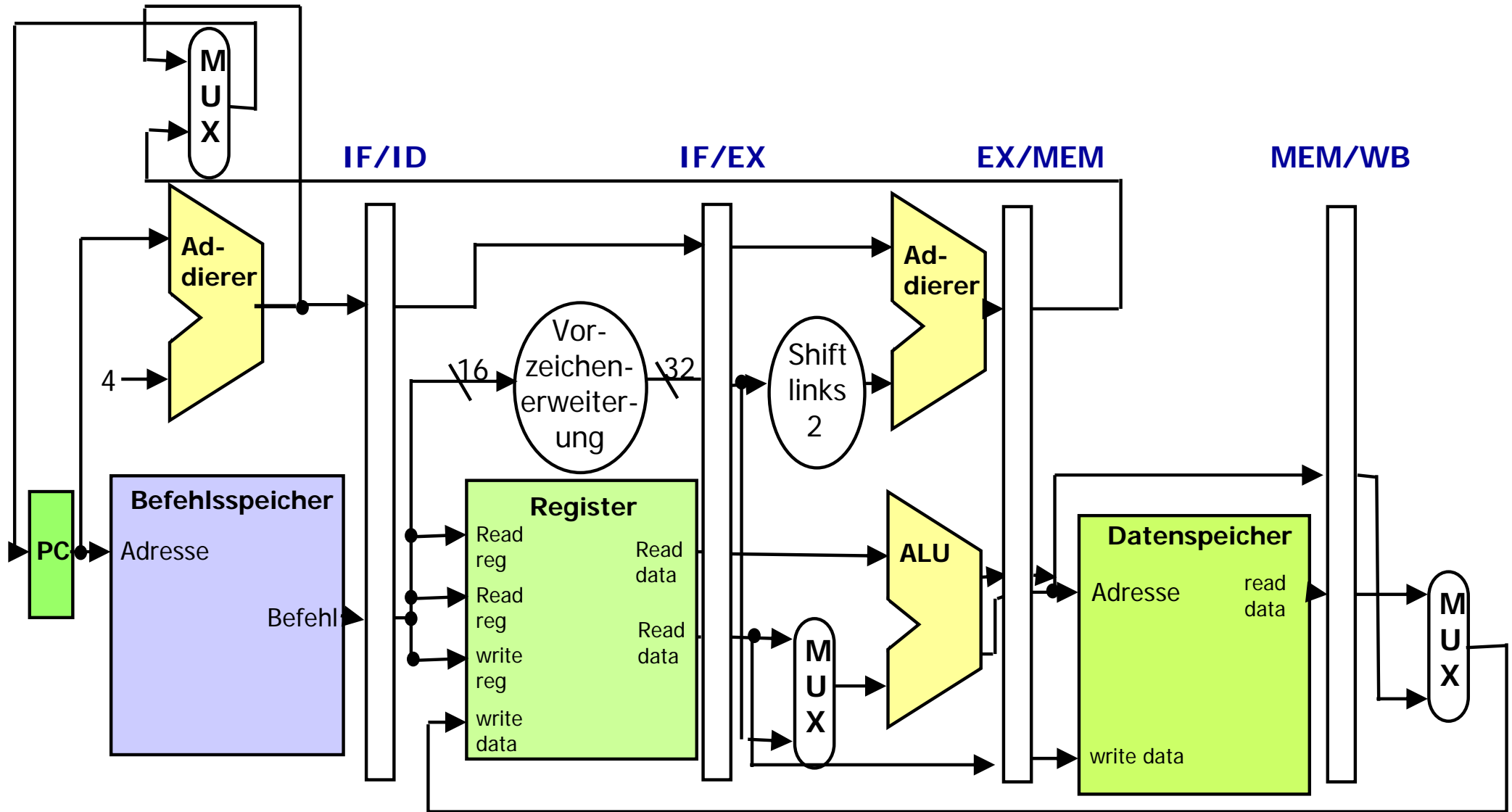


## 5.4 DLX-Pipelinstufen

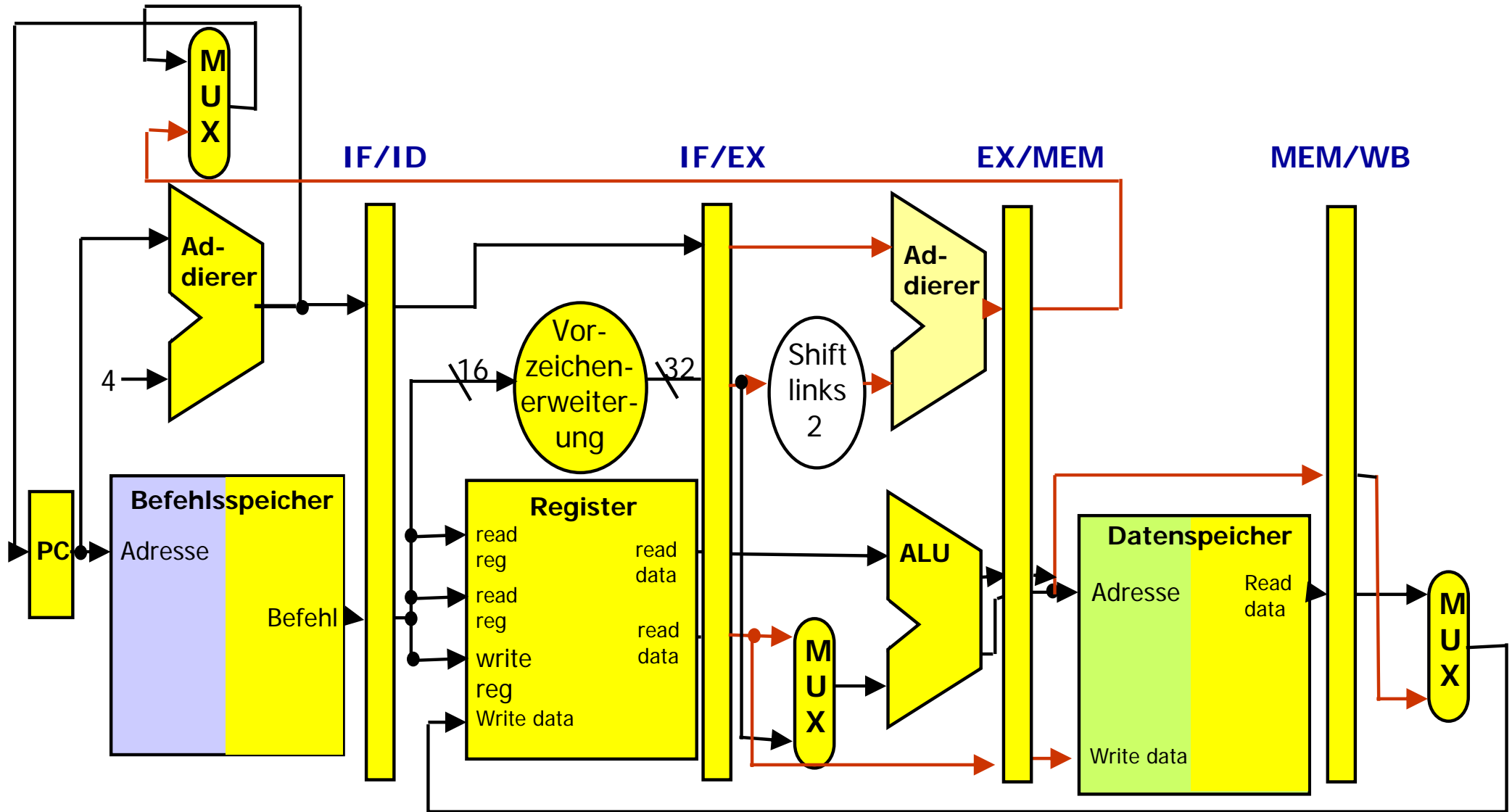
**WB (sw)**



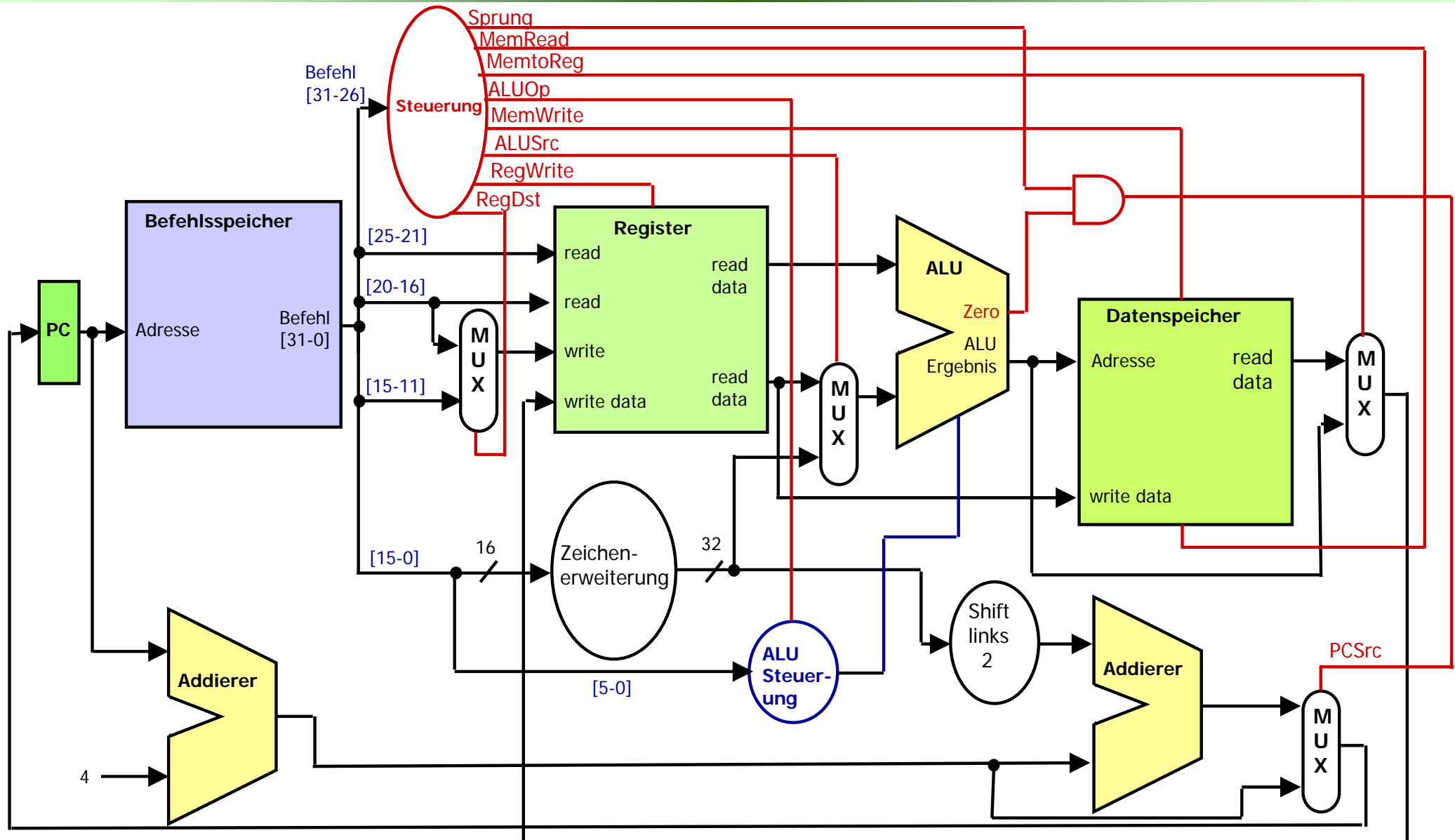
## 5.4 DLX-Pipelinstufen



## 5.4 DLX-Pipelinstufen

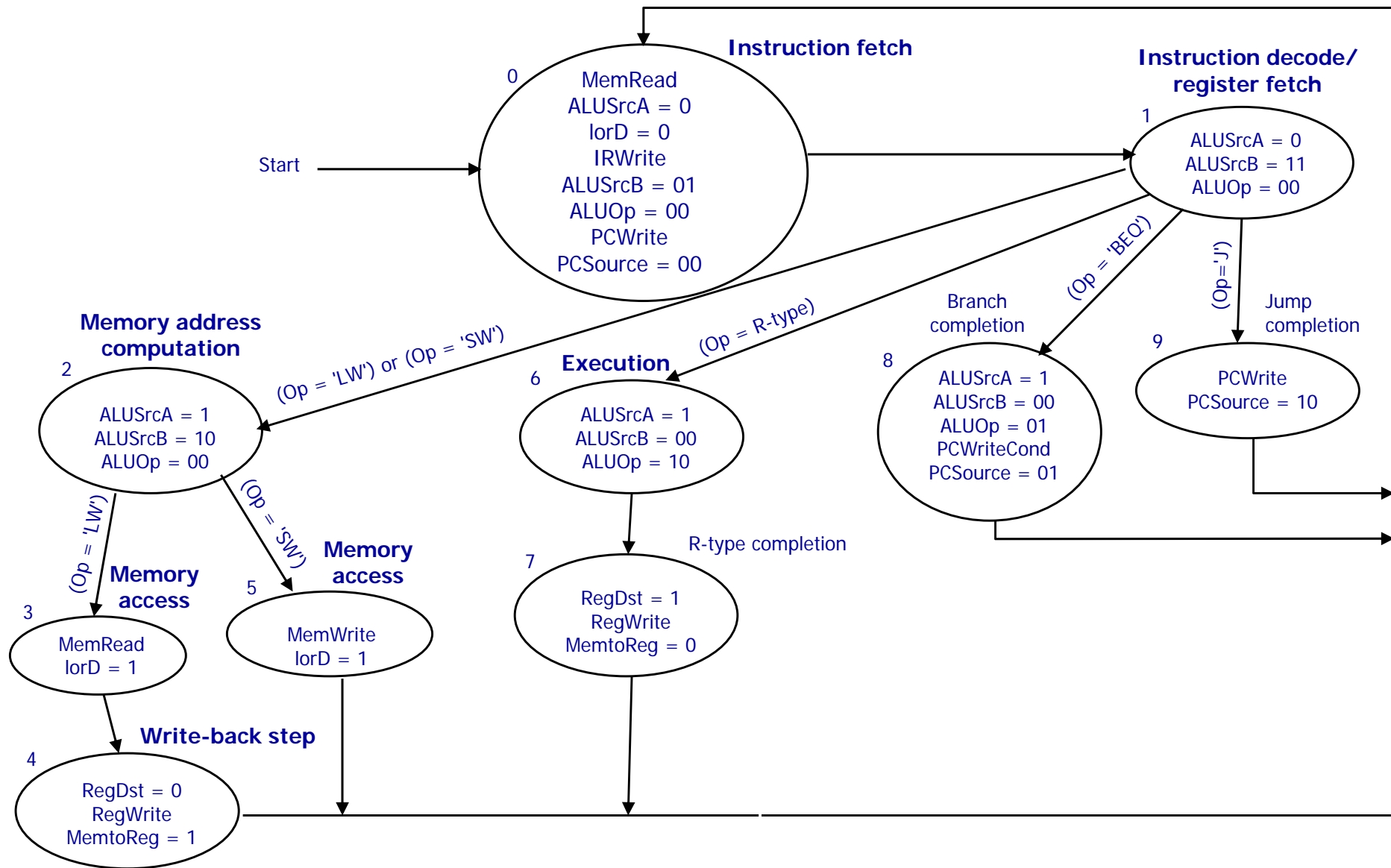


# Datenpfad für die MIPS-Architektur

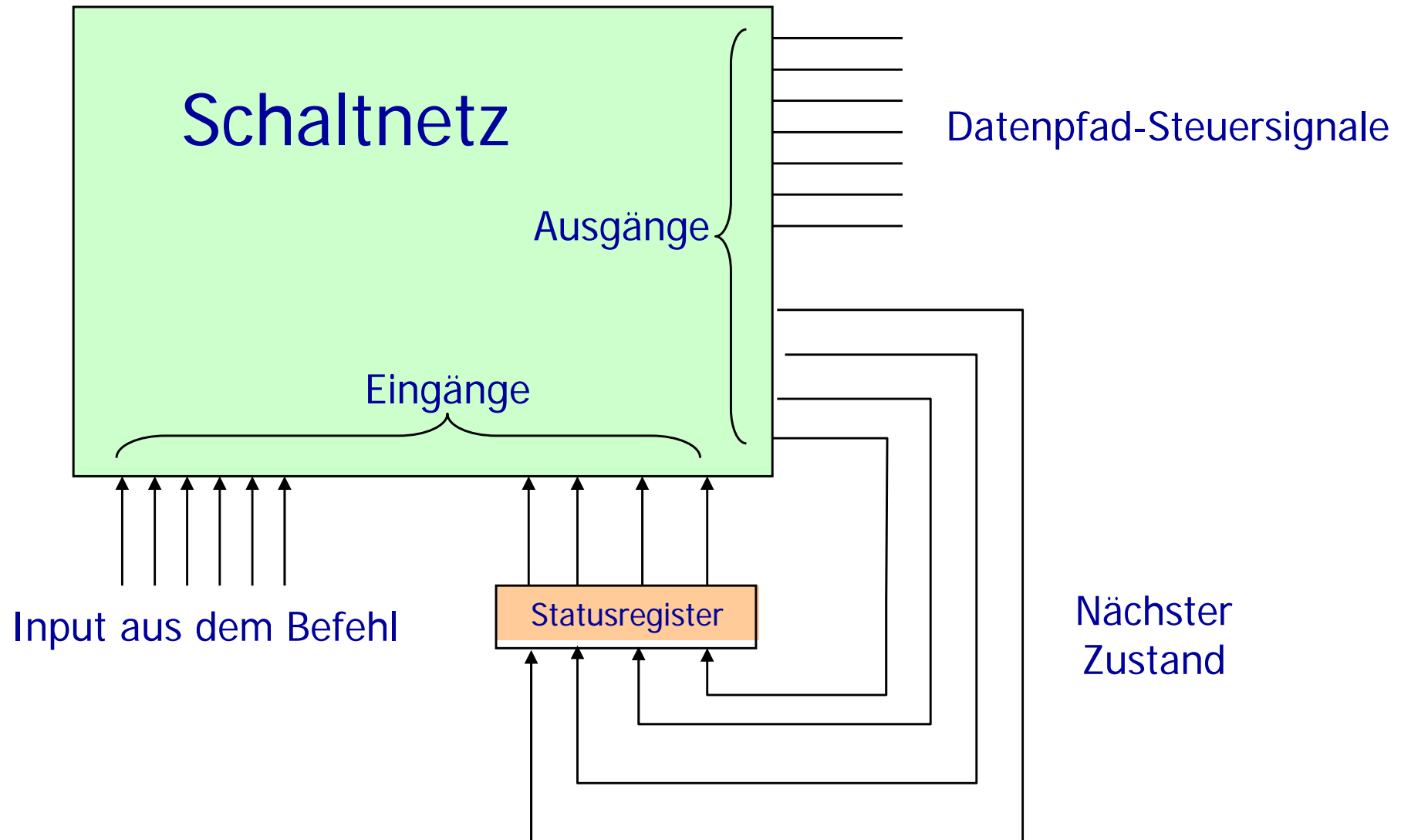


# Zustandsautomat

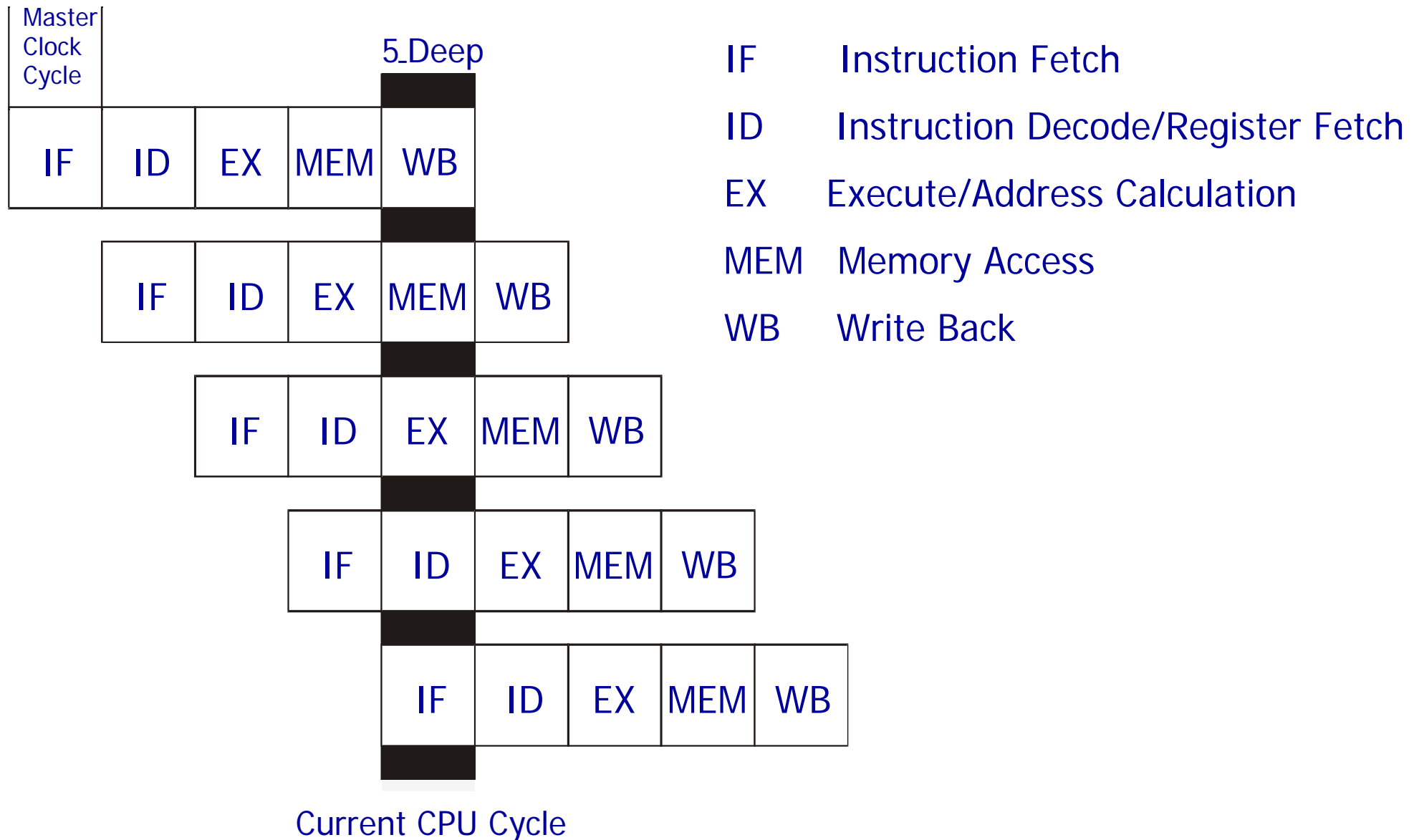
für S-ten 4/12



# Steuersignale für den Datenpfad



# 5.4. DLX-Pipelinstufen



# Phasen der Befehlsausführung in einer fünfstufigen Pipeline (DLX-Pipeline)

---

- ❑ **Befehlsbereitstellungs-Phase (IF-Phase: Instruction Fetch)**  
Der durch den Befehlszähler adressierte Befehl wird aus dem Arbeitsspeicher (bzw. dem Befehlscache) in einen Befehlspuffer geladen. Der Befehlszähler wird weitergeschaltet.
- ❑ **Dekodier- und Operandenbereitstellungsphase (ID-Phase: Instruction Decode & Register Fetch)**  
Aus dem Operationscode des Maschinenbefehls werden prozessorinterne Steuersignale erzeugt. Die Operanden werden aus (Universal)-Register bereit gestellt (2. Takthälfte).
- ❑ **Ausführungsphase / Berechnung der effektiven Adresse (EX-Phase: Execution/Effective Address Calculation)**  
Die Operation wird auf den Operanden ausgeführt.  
Bei Lade- und Speicherbefehlen oder Verzweigungen berechnet die ALU die effektive Adresse

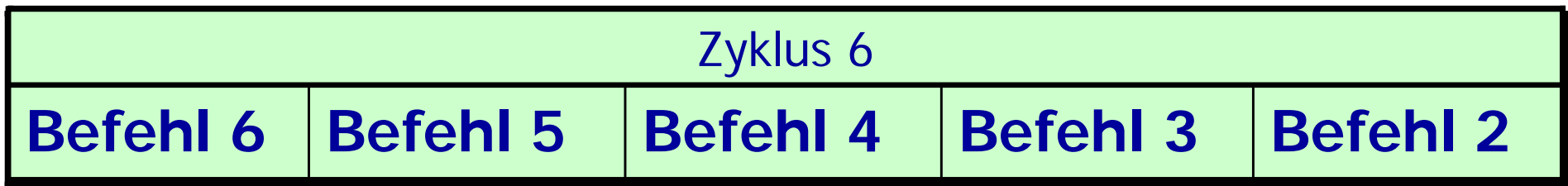
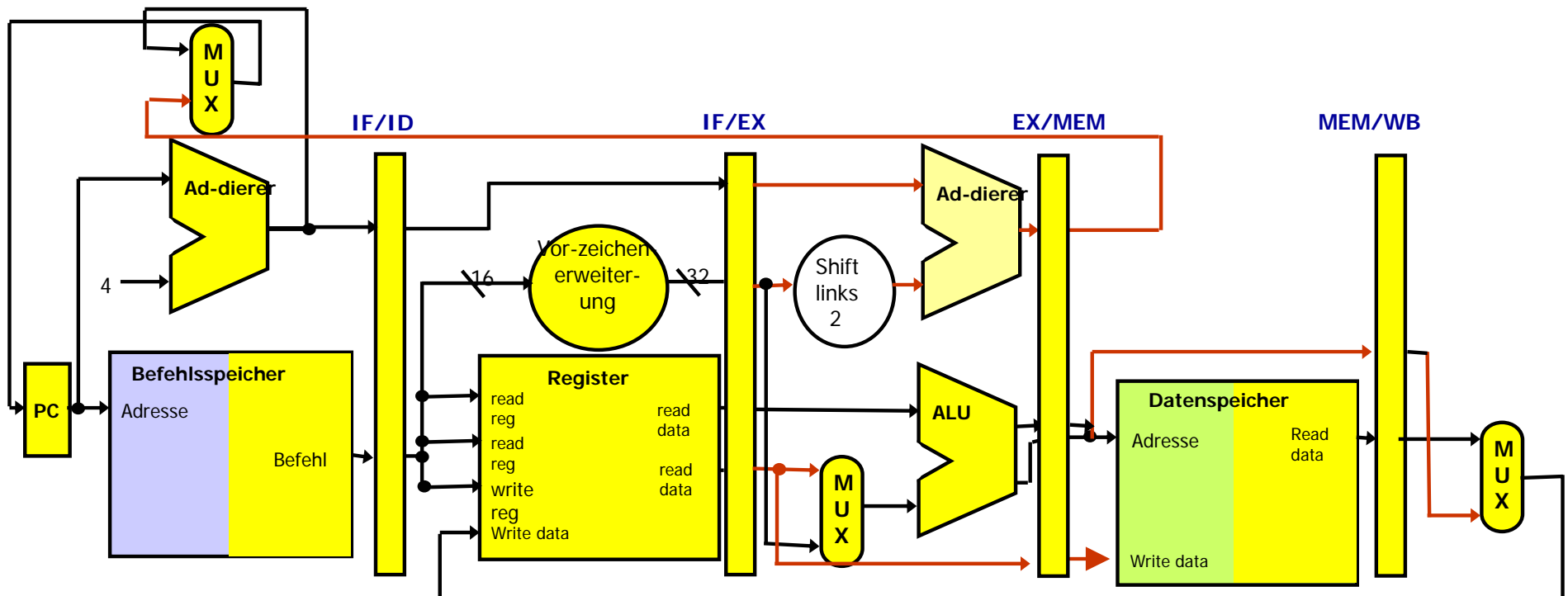
# Phasen der Befehlsausführung in einer fünfstufigen Pipeline (DLX-Pipeline)

---

- ❑ **Speicherzugriffsphase (MEM-Phase: memory access)**  
Der Speicherzugriff (bei Lade- und Speicherbefehlen) wird durchgeführt
- ❑ **Resultatspeicherphase (WB-Phase: write back):**  
Das Ergebnis wird in ein (Universal)-Register geschrieben (1. Takthälfte).  
Befehle ohne Ergebnis durchlaufen diese Phase passiv.

# 5.5 Pipeline-Konflikte

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



## 5.5 Pipeline-Konflikte

- Es gibt leider mehrere potentielle Probleme, die den Durchfluss durch die Pipeline hemmen bzw. verzögern. Man spricht von **Pipeline-Hemmnissen**

- Diese Verzögerungen entstehen durch **Daten-, Struktur- und Steuerflussabhängigkeiten**

# 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 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.

## 5.5.1 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.

## 5.5.1 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.

## 5.5.1 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.

# Beispiel: Datenabhängigkeiten

$S_1$ :    `add`   `r1,r2,2`    `#`    `r1 := r2 + 2`

$S_2$ :    `add`   `r4,r1,r3`    `#`    `r4 := r1 + r3`

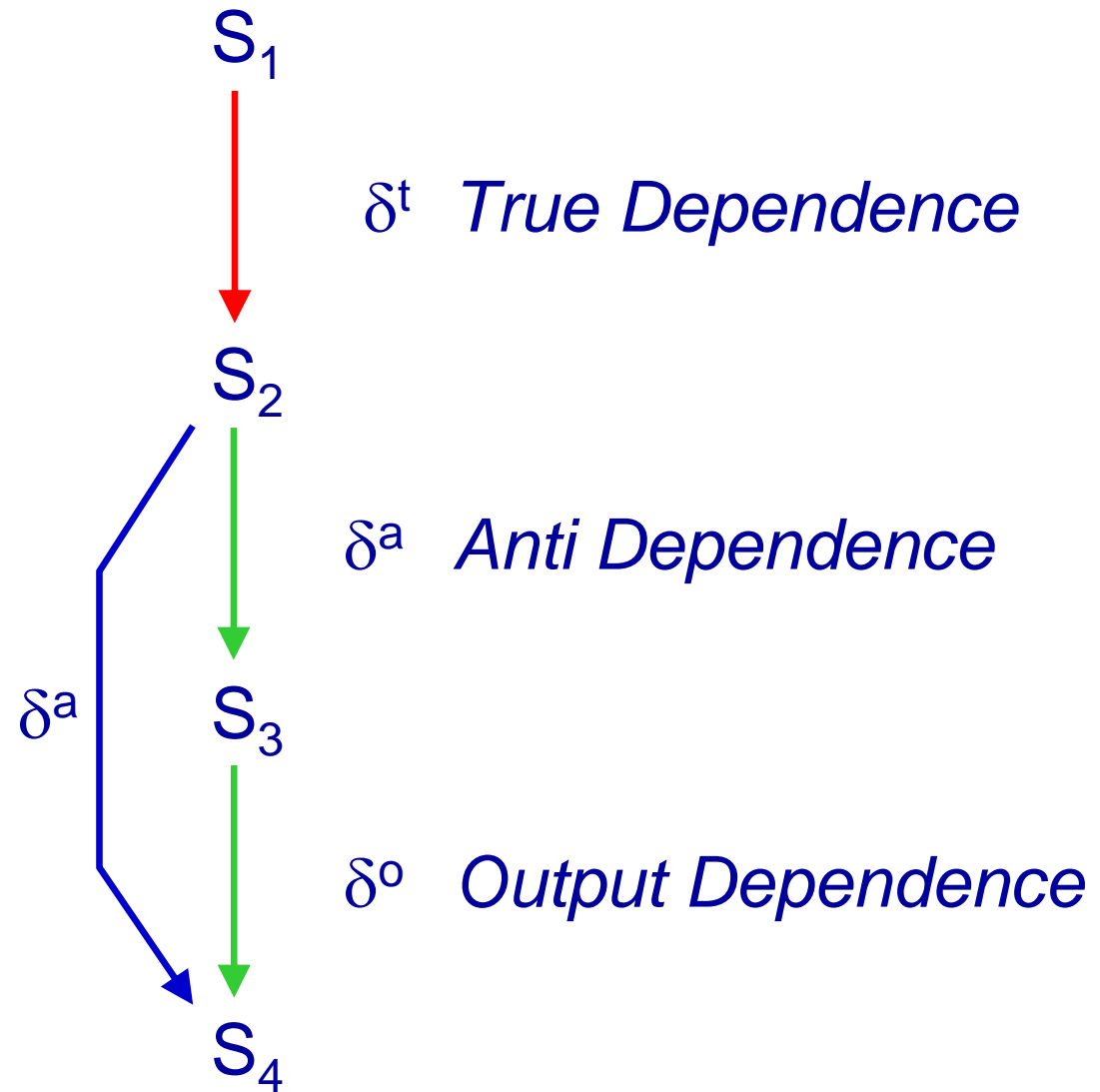
$S_3$ :    `mul`   `r3,r5,3`    `#`    `r3 := r5 * 3`

$S_4$ :    `mul`   `r3,r6,3`    `#`    `r3 := r6 * 3`

```
graph TD; S1["S1: add r1,r2,2 # r1 := r2 + 2"] -- "r1" --> S2["S2: add r4,r1,r3 # r4 := r1 + r3"]; S2 -- "r3" --> S3["S3: mul r3,r5,3 # r3 := r5 * 3"]; S3 -- "r3" --> S4["S4: mul r3,r6,3 # r3 := r6 * 3"];
```

# Beispiel: Datenabhängigkeiten

$S_1$ :    add    ~~r1~~, r2, 2  
 $S_2$ :    add    r4, ~~r1~~, r3  
 $S_3$ :    mul    ~~r3~~, r5, 3  
 $S_4$ :    mul    ~~r3~~, r6, 3



# 5.5.1 Datenabhängigkeiten

---

## Bemerkung:

Gegenabhängigkeiten und Ausgabeabhängigkeiten werden häufig auch **falsche Datenabhängigkeiten** oder entsprechend dem englischen Begriff „*Name Dependency*“ **Namensabhängigkeiten** genannt.

## 5.5.2 Datenkonflikte

- ❑ **Lese-nach-Schreibe-Konflikt** (read after write, RAW):  
Wird durch echte Abhängigkeit verursacht.
- ❑ **Schreibe-nach-Lese-Konflikt** (write after read, WAR):  
Wird durch Gegenabhängigkeit verursacht. Tritt dann auf, wenn in einer Pipeline die Schreibestufe der Lesestufe vorangeht.
- ❑ **Schreibe-nach-Schreibe-Konflikt** (write after write, WAW): Wird durch Ausgabeabhängigkeit verursacht. Tritt in Pipelines auf, die mehr als in einer Stufe schreiben, oder es erlauben, dass die Ausführung eines Befehls fortgesetzt werden darf, wenn ein vorhergehender Befehl angehalten worden ist.

# WAR und WAW

---

Können WAR und WAW in der fünfstufigen DLX-Pipeline auftreten?

Nein, weil:

- Alle Befehle werden in 5 Stufen ausgeführt,
- Lesen aus Registern immer in der Stufe 2, und
- Schreiben in Register immer in der Stufe 5.

WAR und WAW treten bei „komplexeren“ Pipelines auf

# Beispiel

load r1,A



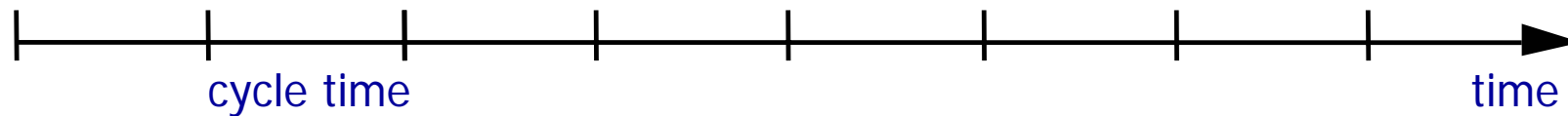
load r2,B



add r2,r1,r2



mul r1,r2,r1



# Fehlzuweisung durch einen Datenkonflikt

`add r2,r1,r2`

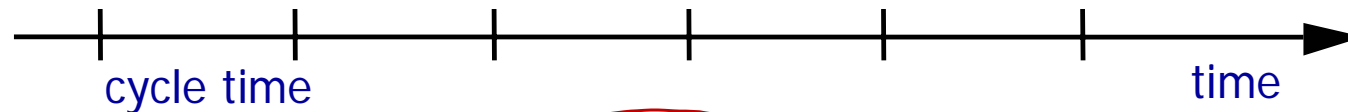


wrong register read!

Reg2 alt

Reg2 neu

`mul r1,r2,r1`



# 5.5.2 Lösungen für Datenkonflikte

## □ Software-Lösung

### ➤ Aufgabe des Compilers:

- Erkennen von Datenkonflikten
- Einfügen von Leeroperationen (noop) nach jedem Befehl, der einen Konflikt verursacht oder verursachen kann.

*No Operation*

### ➤ Statische Befehlsanordnung:

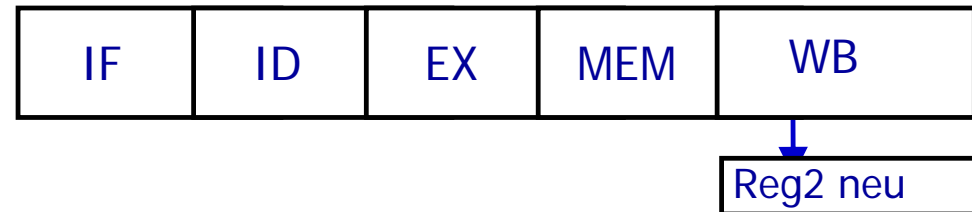
- Instruction Scheduling, Pipeline Scheduling
- Umordnen der Befehle des Programms (Code-Optimierung), um Leeroperationen zu eliminieren.

# 5.5.2 Lösungen für Datenkonflikte

## ❑ Software-Lösungen (*Compiler scheduling*)

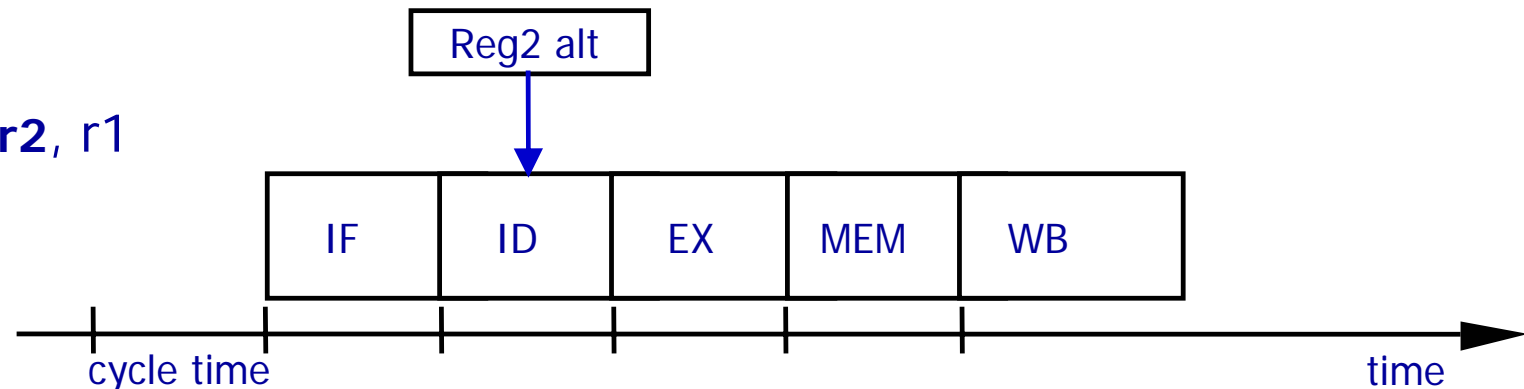
- noop Befehle einfügen

add r2, r1, r2



*2 x Noop einfügen*

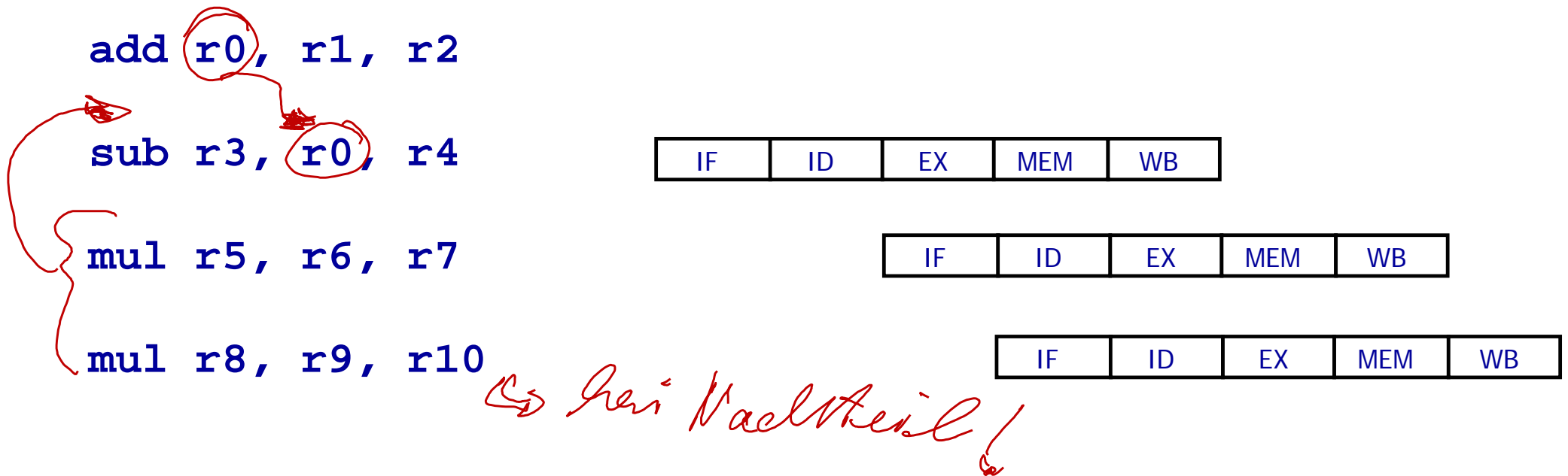
mul r1, r2, r1



# 5.5.2 Lösungen für Datenkonflikte

## □ Software-Lösungen (*Compiler scheduling*)

- **Befehlsanordnung** (*instruction scheduling* oder *pipeline scheduling*): Befehlsanordnungen, um noops zu entfernen



## 5.5.2 Lösungen für Datenkonflikte

- **Hardware-Lösungen:** Konflikt muss per HW entdeckt werden!!
  - **Leerlauf der Pipeline:** Die einfachste Art, mit solchen Datenkonflikten umzugehen ist es,  $\text{Inst}_2$  in der Pipeline für zwei Takte anzuhalten. Auch als **Pipeline-Sperrung** (*interlocking*) oder **Pipeline-Leerlauf** (*stalling*) bezeichnet.
  - **Forwarding:** Wenn ein Datenkonflikt erkannt wird, so sorgt eine Hardware-Schaltung dafür, dass der betreffende Operand nicht aus dem Universalregister, sondern direkt aus dem ALU-Ausgaberegister der vorherigen Operation in das ALU-Eingaberegister übertragen wird.
  - **Forwarding with interlocking:** Forwarding löst nicht alle möglichen Datenkonflikte auf.

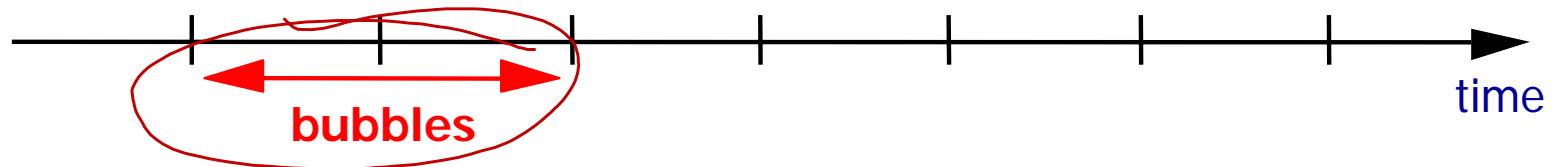
# Hardware-Lösung durch Interlocking

add r2, r1, r2



Register Reg2

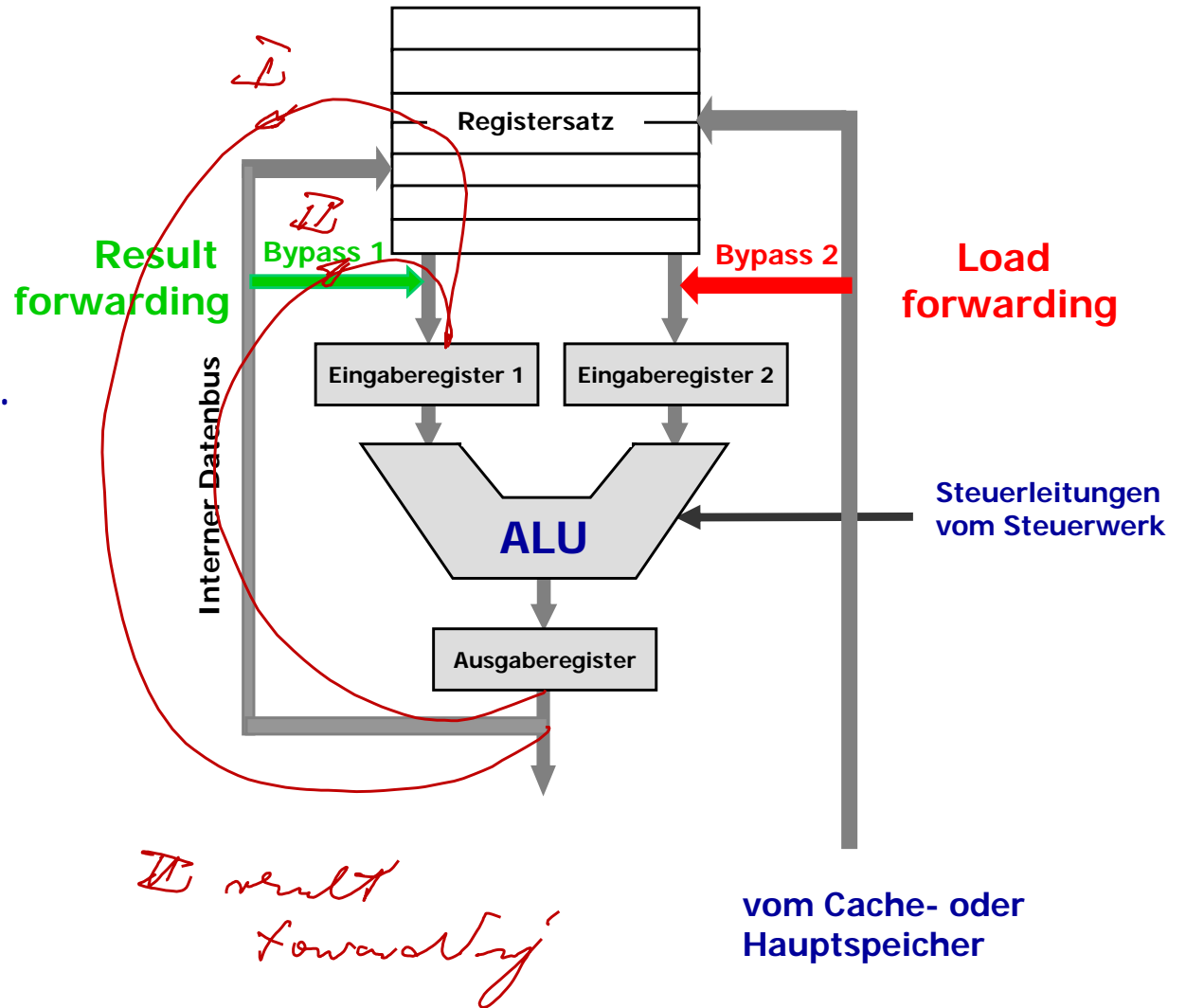
mul r1, r2, r1



# Forwarding-Techniken

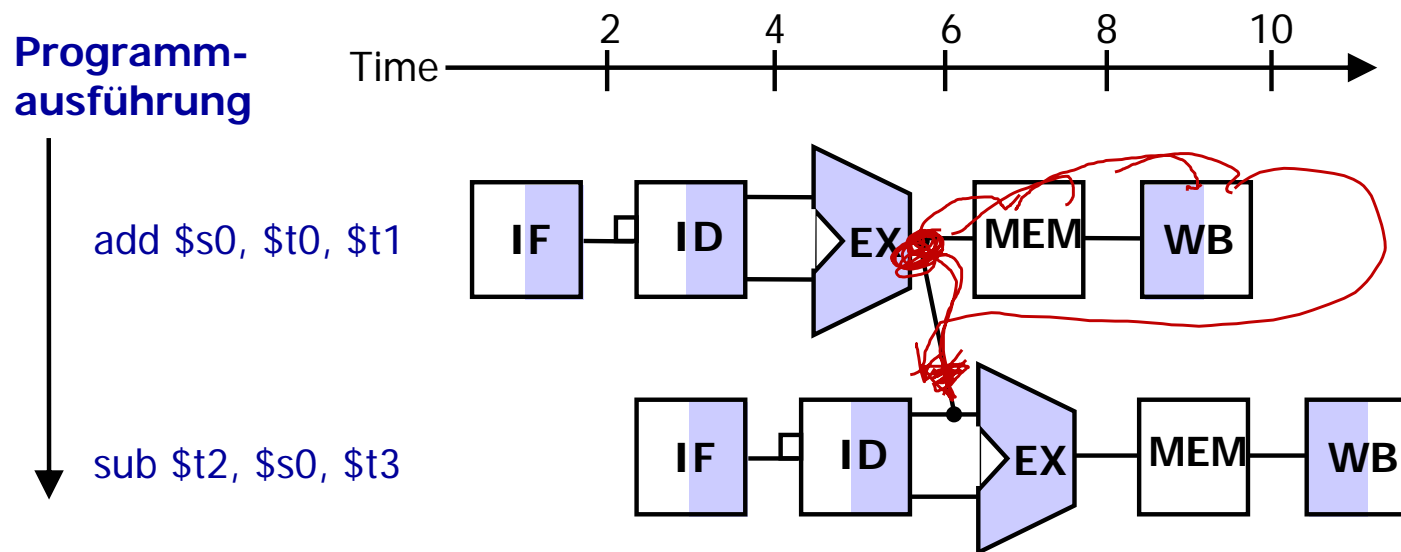
## Forwarding:

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



# Forwarding-Techniken

## Result Forwarding:



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

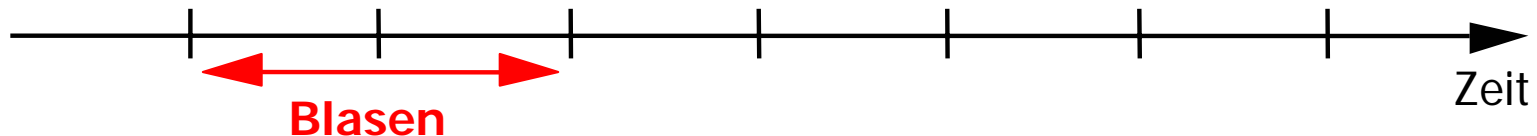
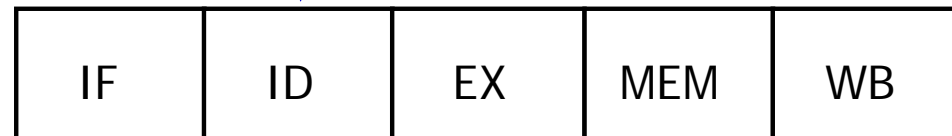
# Leerlauf der Pipeline: Interlocking

`add r2,r1,r2`



Register r2

`mul r1,r2,r1`



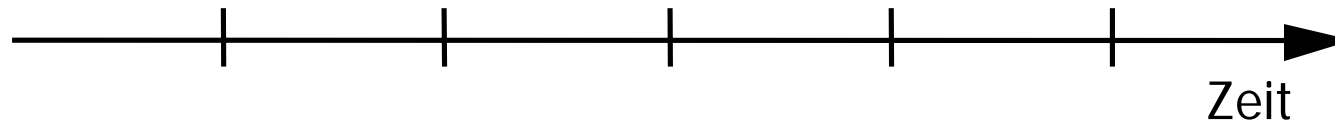
Anhalten des `mul`-Befehls für zwei Takte

# Hardware-Lösung durch *Forwarding*

`add r2,r1,r2`

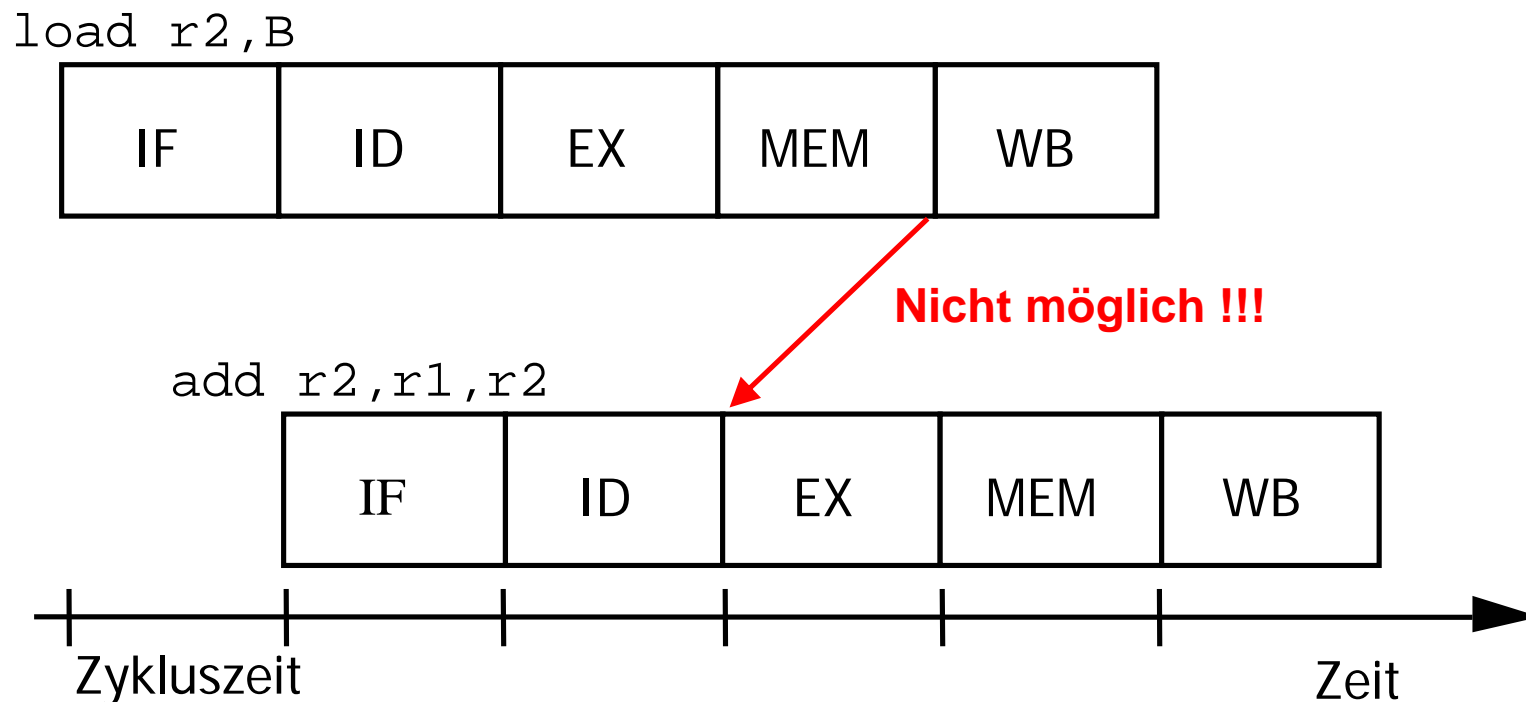


`mul r1, r2, r1`



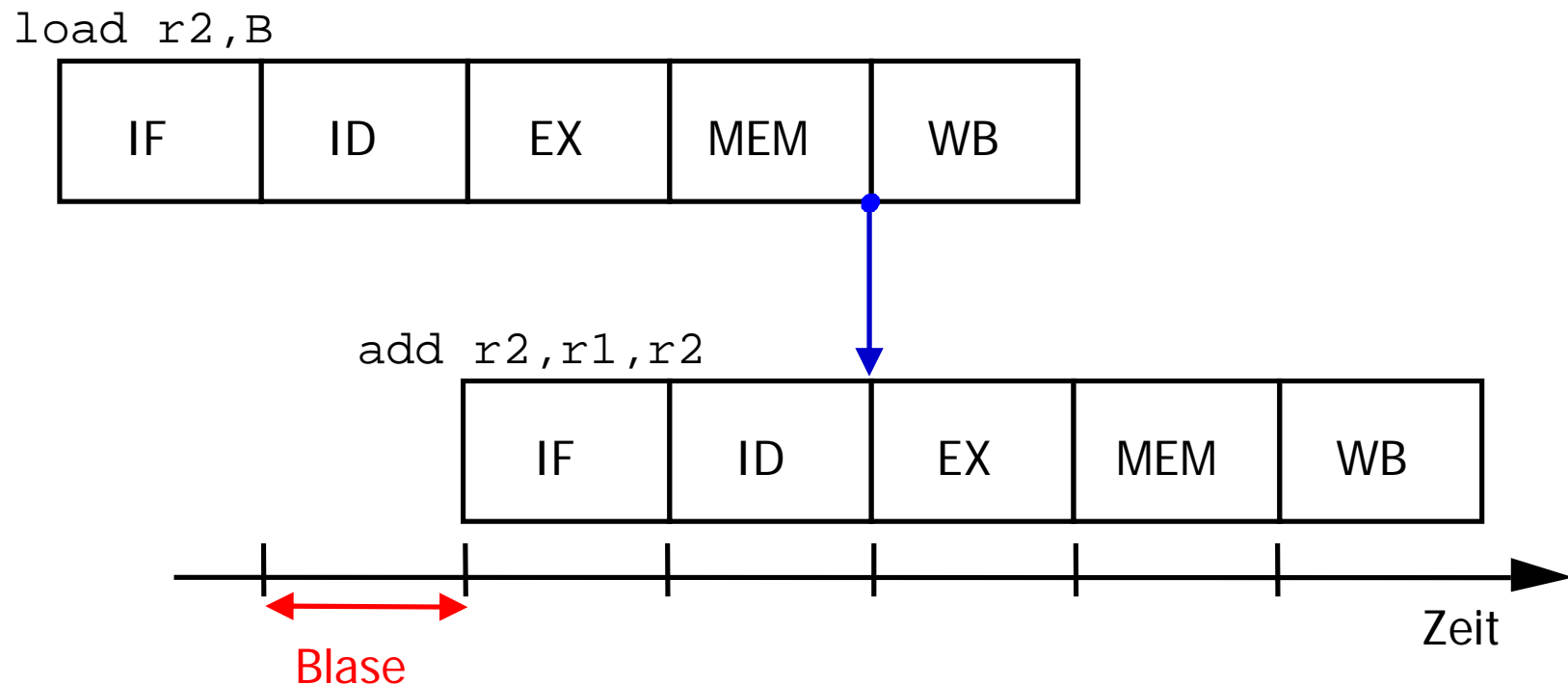
# Problem: Nicht alle Konflikte sind alleine durch Forwarding behebbar !!!

## Beispiel: Speicherzugriff (z. B. Ladebefehl)



- EX-Stufe berechnet die effektive Adresse
- Der `add`-Befehl muss angehalten werden, bis die geladenen Daten im Ladewertregister der MEM-Stufe verfügbar sind

# Lösung: Pipelineverzögerungen (bubble)



# 5.5.2 Lösungen für Datenkonflikte

---

## □ Software-Lösung

### ➤ Aufgabe des Compilers:

- Erkennen von Datenkonflikten
- Einfügen von Leeroperationen (`noop`) nach jedem Befehl, der einen Konflikt verursacht oder verursachen kann.

### ➤ Statische Befehlsanordnung:

- Instruction Scheduling, Pipeline Scheduling
- Umordnen der Befehle des Programms (Code-Optimierung), um Leeroperationen zu eliminieren.

# 5.5.2 Lösungen für Datenkonflikte

---

- **Hardware-Lösungen:** Konflikt muss per HW entdeckt werden!!
  - **Leerlauf der Pipeline:** Die einfachste Art, mit solchen Datenkonflikten umzugehen ist es,  $\text{Inst}_2$  in der Pipeline für zwei Takte anzuhalten. Auch als **Pipeline-Sperrung** (*interlocking*) oder **Pipeline-Leerlauf** (*stalling*) bezeichnet.
  - **Forwarding:** Wenn ein Datenkonflikt erkannt wird, so sorgt eine Hardware-Schaltung dafür, dass der betreffende Operand nicht aus dem Universalregister, sondern direkt aus dem ALU-Ausgaberegister der vorherigen Operation in das ALU-Eingaberegister übertragen wird.
  - **Forwarding with interlocking:** Forwarding löst nicht alle möglichen Datenkonflikte auf.

## 5.5.3 Ressourcenkonflikte

---

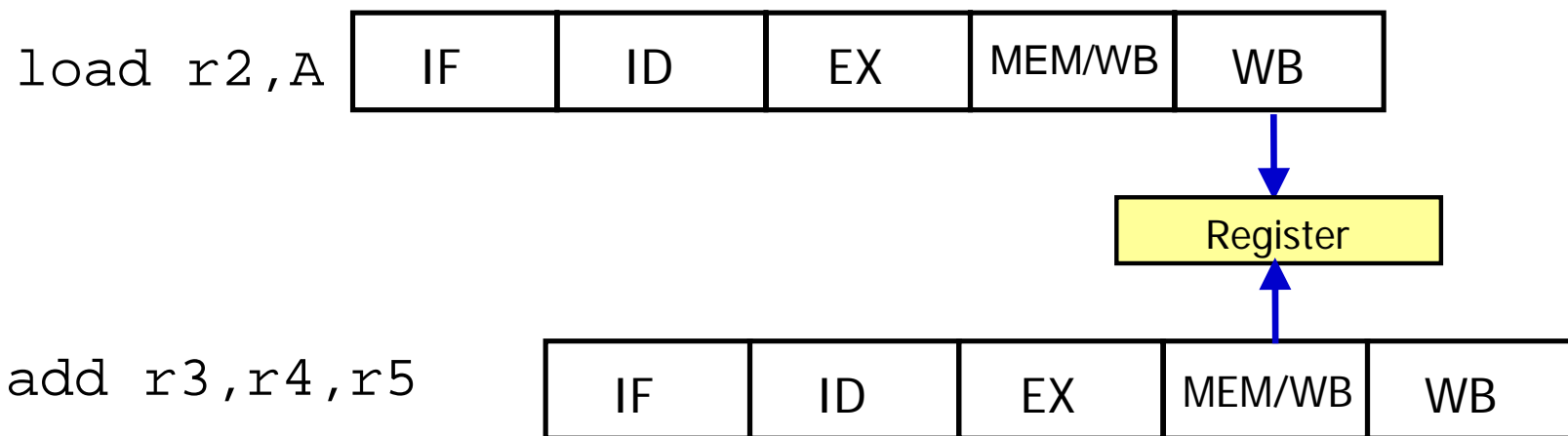
### □ Ressourcenkonflikt:

- Treten auf, wenn zwei oder mehr Befehle gleichzeitig dieselbe Ressource benötigen, auf die aber nur einmal zugegriffen werden kann
- Treten bei einer einfachen Pipeline, wie die DLX-Pipeline, nicht auf
- **Ziel beim Pipeline-Entwurf:** Ressourcenkonflikte möglichst zu vermeiden und dort, wo sie nicht vermeidbar sind, zu erkennen und behandeln

## 5.5.3 Ressourcenkonflikte

### □ Ressourcenkonflikt:

- **Beispiel:** Prozessor mit veränderter DLX-Pipeline
  - Die MEM-Stufe ist in der Lage, bei einem Register-Register-Befehl die Ausgabe der ALU durch einen Schreibkanal auf den Registersatz zurückzuschreiben.
  - Im Beispiel greifen zwei Befehle gleichzeitig auf einen nur einfach vorhandenen Schreibeingang zu.



## 5.5.4 Lösungen von Ressourcenkonflikte

---

### □ Lösungen (Hardware):

- **Arbitrierung mit Interlocking**

Arbitrierungslogik hält den Befehl, der den Konflikt verursacht → Verzögerung durch Leerzyklen

- **Übertaktung:**

Die Ressource, die den Konflikt hervorruft schneller als die übrigen Pipeline-Stufen takten

- **Ressourcenreplizierung:** Vervielfachung der Ressourcen

**Beispiel:** Registersatz mit mehreren Schreibkanälen

# 5.5.5 Steuerflusskonflikte

---

## ➤ Programmsteuerbefehle:

- die bedingten und die unbedingten Sprungbefehle,
- die Unterprogrammaufruf- und -rückkehrbefehle sowie
- die Unterbrechungsbefehle

## ➤ Steuerflussabhängigkeiten verursachen **Steuerflusskonflikte** in der DLX-Pipeline, da

- der Programmsteuerbefehl erst in der ID-Stufe als solcher erkannt und damit bereits ein Befehl des falschen Programmpfades in die Pipeline geladen wurde.
- die Sprungzieladresse von der ALU erst in der EX-Stufe berechnet wird und
- der PC am Ende der MEM-Stufe ersetzt wird

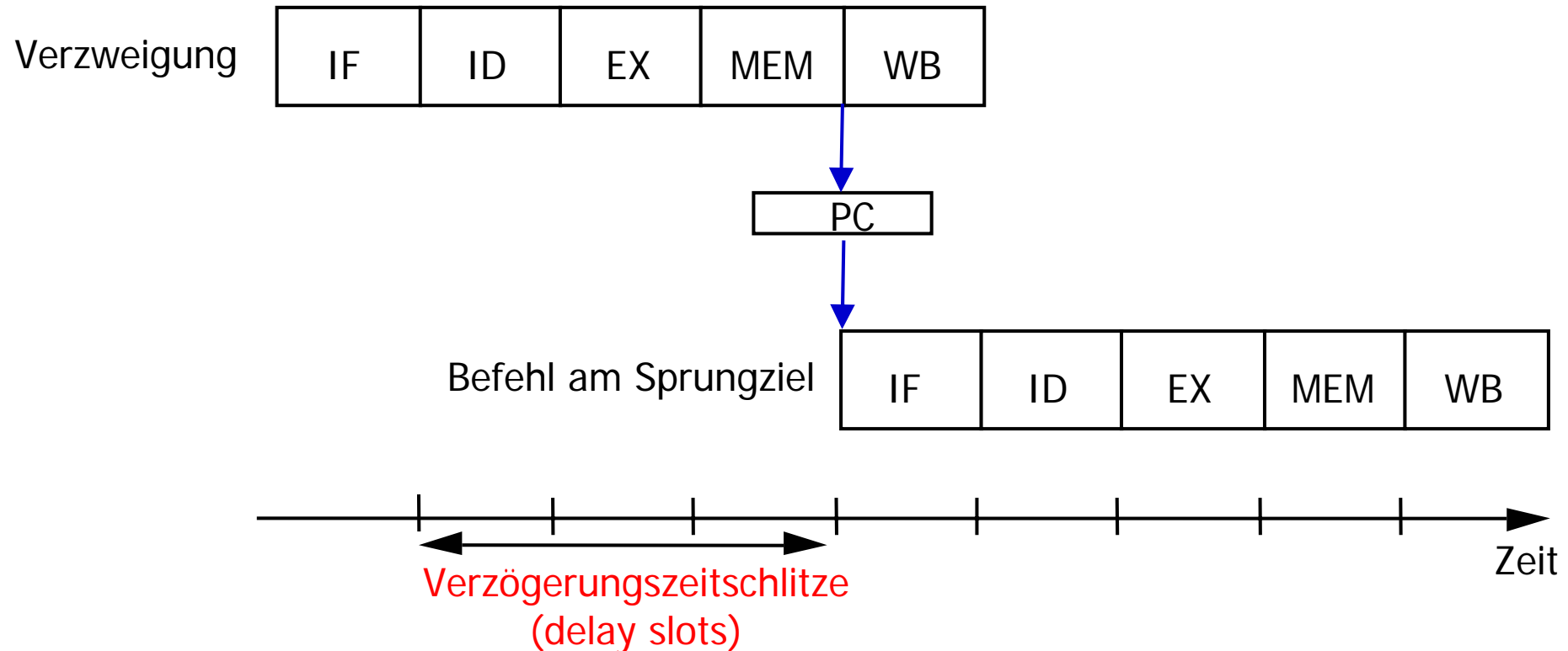
# Steuerflusskonflikte durch Verzweigung

Sprungbefehl



- In der Pipeline befinden sich drei Befehle, die bereits geladen worden sind und wieder gelöscht werden müssen, wenn der Sprung ausgeführt wird.
- Bei einer Verzweigung kann erst nach **drei Taktzyklen** mit der Ausführung der korrekten Befehlsfolge gestartet werden.

# Steuerflusskonflikte durch Verzweigung



- Nach einem genommenen bedingten Sprung ist eine Pipeline-Verzögerung durch Einfügen von Wartezyklen notwendig

# Steuerflusskonflikte durch Verzweigung

---

## Verringerung der Anzahl der Wartezyklen:

- Sprungrichtung so schnell wie möglich entscheiden und Zieladresse so schnell wie möglich berechnen.
- Berechnung in der ID-Stufe → nur noch ein Wartezyklus, aber dann → **Strukturkonflikt**
  - Die ALU kann nicht zur Berechnung der Sprungzieladresse benutzt werden, da sie vom vorhergehenden Befehl benötigt wird
- Dekodierung, Berechnung der Sprungzieladresse und Rückschreibung des PCs in einer Pipeline-Stufe → Kritischer Pfad in der Dekodierphase → Verlängerung der Zykluszeit!

# Steuerflusskonflikte durch Verzweigung

---

- Trotz der vorgeschlagenen Pipeline-Reorganisation bleibt ein Wartezyklus (Verzögerungsphase).
- Bei der DLX-Pipeline werden Steuerflusskonflikte durch die Hardware **weder erkannt noch behandelt**, d. h. die drei Befehle hinter einem Sprungbefehl werden immer ausgeführt.
- **Techniken zur Konfliktauflösung**
  - Software-Techniken
  - Hardware-Techniken

# Steuerflusskonflikte durch Verzweigung

---

## □ Software-Lösung

### ➤ Verzögerte Sprungtechnik (delayed branch technique)

- Ausfüllen der Verzögerungsschlitze (delay slots) mit Leerbefehlen (noop)
- DLX-Pipeline: Drei Leerbefehle nach jedem Programmsteuerbefehl

# Steuerflusskonflikte durch Verzweigung

---

## □ Software-Lösung

### ➤ Statische Befehlsanordnung:

- Der Compiler verschiebt Befehle, die in der logischen Programmreihenfolge vor dem Sprungbefehl liegen, in die Verzögerungszeitschlitze
- Nur dann möglich, wenn die verschobenen Befehle keinen Einfluss auf die Sprungrichtung haben
- Gibt es keine Befehle, die in die Verzögerungszeitschlitze verschoben werden können, müssen Leerbefehle eingefügt werden
- **Nachteil:** Code wird abhängig von der Pipeline-Struktur

# Steuerflusskonflikte durch Verzweigung

---

## □ Hardware-Lösung (Dynamische Techniken)

### ➤ Pipeline-Leerlauf:

- Einfachste und ineffizienteste Methode
- Hardware erkennt Verzweigungsbefehle (in der ID-Stufe) und lädt keine weiteren Befehle in die Pipeline, bis die Zieladresse berechnet und im Falle bedingter Sprungbefehle die Sprungentscheidung getroffen ist
- Der eine Befehl, der bereits ins Pipeline-Register der IF-Stufe geladen wurde, muss gelöscht werden.

# Steuerflusskonflikte durch Verzweigung

---

## □ Hardware-Lösung (Dynamische Techniken)

### ➤ Spekulation auf nicht genommene bedingte Sprünge:

- Einfachste Technik der statischen Sprungvorhersagen
- Annahme: Sprung wird **nicht** „genommen“
- Nachfolgende Befehle werden in die Pipeline geladen
- Falls der Sprung „genommen“ wird, müssen die drei Befehle gelöscht werden (Pipeline flushing).
- Falls der Sprung nicht „genommen“ wird, so können die drei Befehle ausgeführt werden.
- **Nachteil:** Ineffizient, da bedingt durch die in Programmen häufig auftretenden Schleifen die Mehrzahl der Sprünge genommen wird

# Steuerflusskonflikte durch Verzweigung

---

## □ Hardware-Lösung (Dynamische Techniken)

- Wie können „genommene“ bedingte Sprünge und alle anderen Programmsteuerbefehle, die immer zu Steuerflussänderung führen, behandelt werden?
  - Nur möglich, wenn in der IF-Stufe der richtige nachfolgende Befehl geladen werden kann.
  - Kleiner Pufferspeicher (Sprungzieladress-Cache, *branch target address cache*) in der IF-Stufe:
    - Speichert nach dem ersten Durchlauf der Befehlsfolge die Adresse des Programmsteuerbefehls und die Sprungzieladresse
    - Beim nächsten Auftreten desselben Programmsteuerbefehls kann die Pipeline sofort mit dem Befehl an der Zieladresse geladen werden.

# Hyper-Pipeline beim Pentium 4

---

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive Alloc		Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

- 20 Stufen
- Bis zu 126 Instruktion werden gleichzeitig bearbeitet, davon 48 Load- und 24 Store-Operationen
- Kurze Stufen → Weniger Transistoren müssen geschaltet werden → Schneller

# Pentium 4 Prozessor-Architektur

