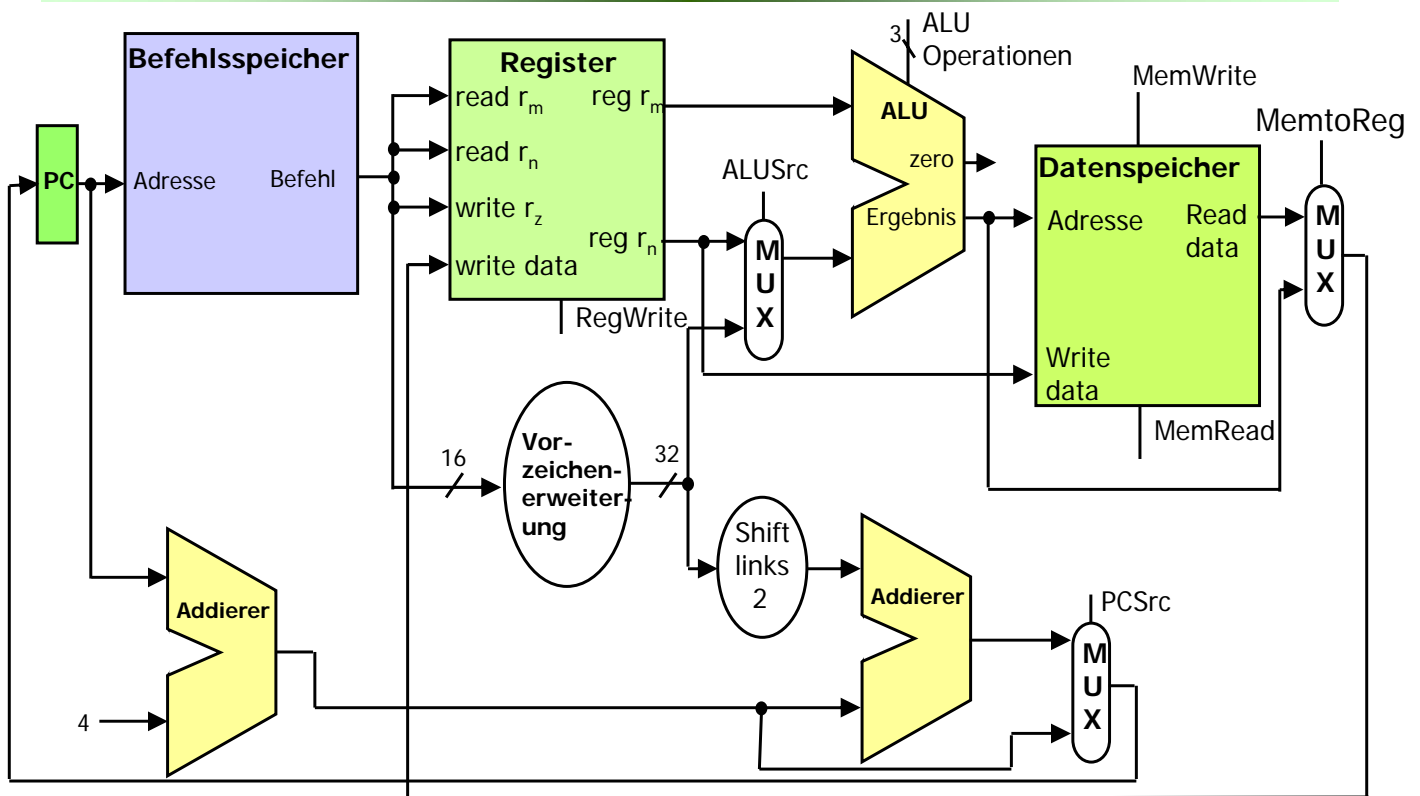


5. 3 Datenpfad für die MIPS-Architektur



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

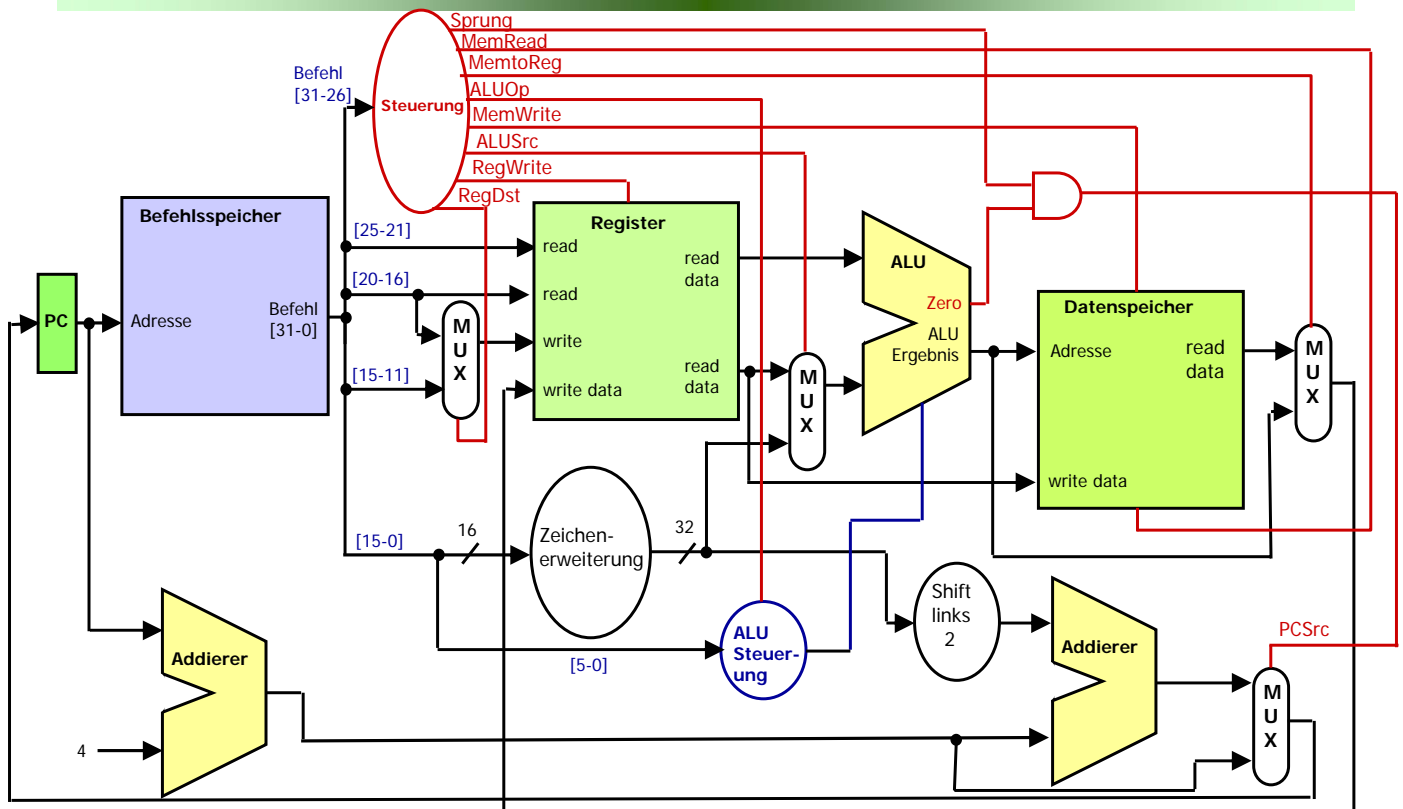
35 oder 43	rs	rt	Adresse
31-26	25-21	20-16	15-0

➤ Sprung Befehl

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



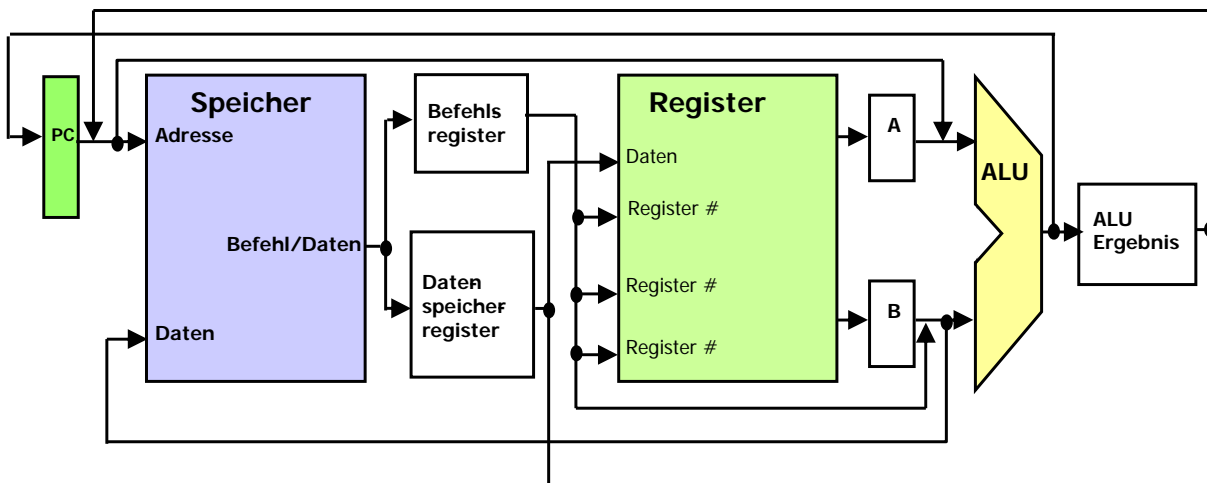
5.3 Datenpfad für die MIPS-Architektur



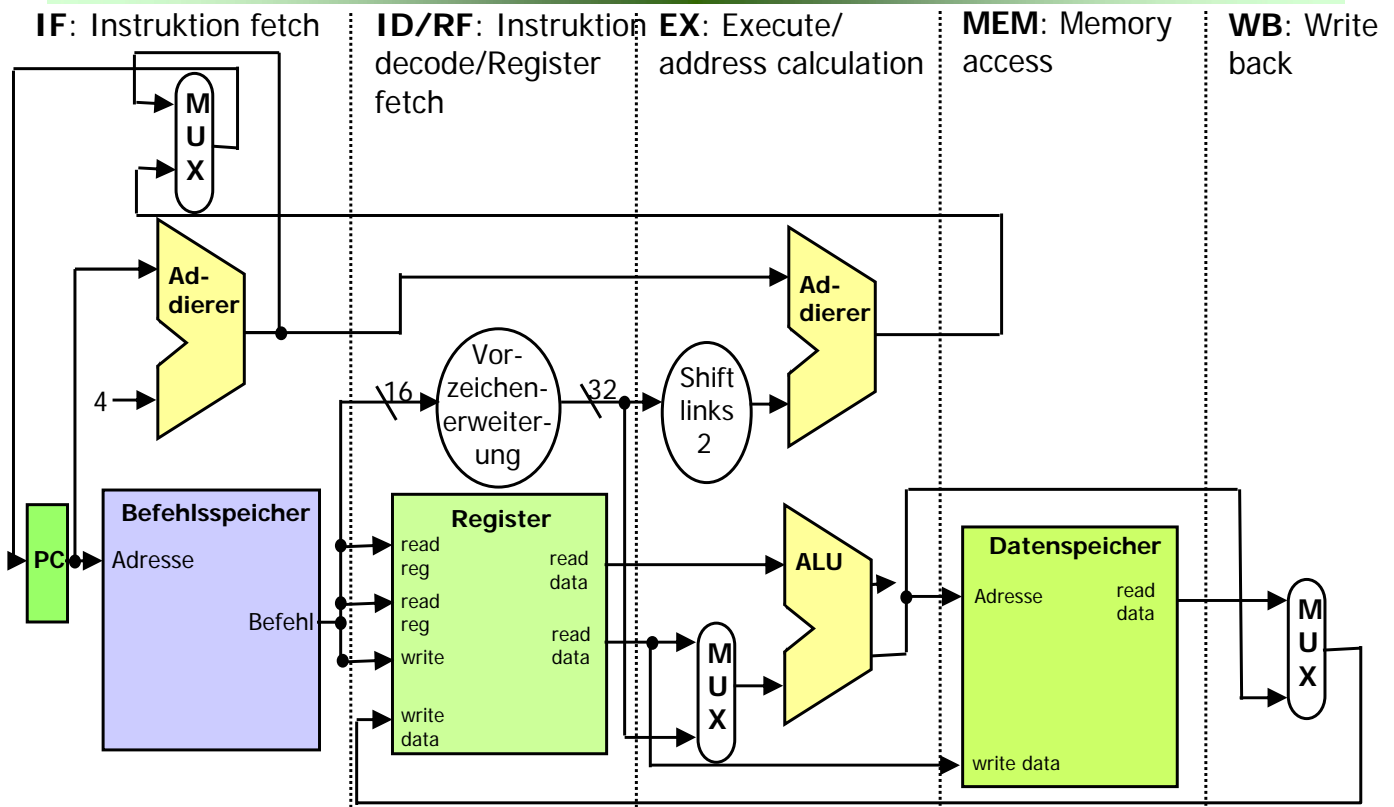
5.4. Pipelining in MIPS Architektur

□ Schlüsseleinheiten des Datenpfads

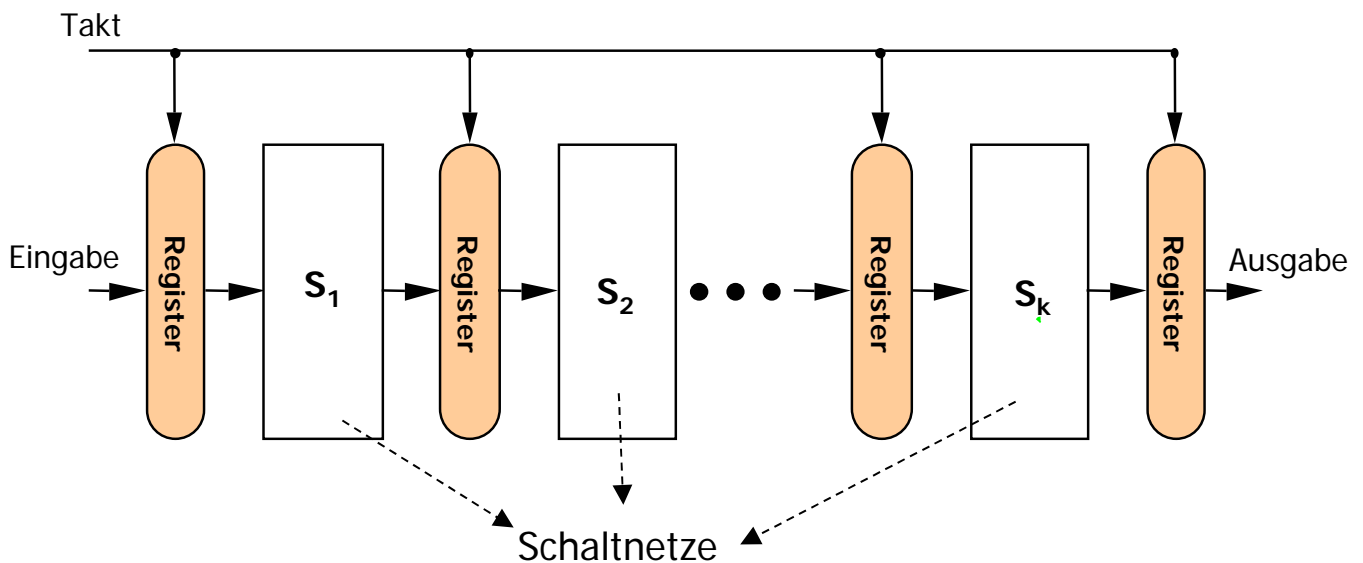
- Eine ALU
- Eine Speichereinheit für Daten und Befehle
- Register: Befehlsregister, Datenspeicherregister, A, B und ALU-Ergebnisregister



5.4. Pipelining in MIPS Architektur



Pipeline-Stufen und Pipeline-Register



Verzögerungszeiten:

➤ der Schaltnetze: τ_i ($i = 1, \dots, k$)

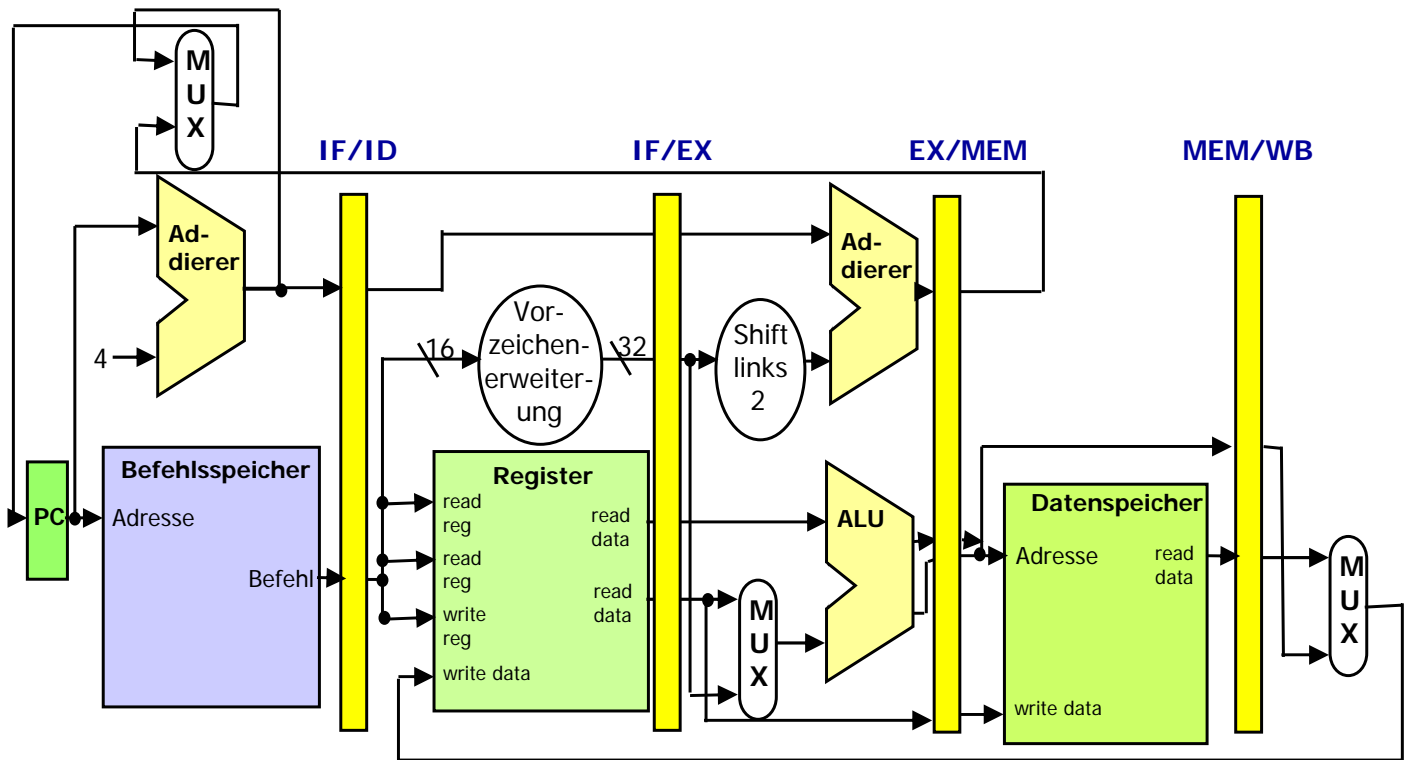
➤ der Pipeline-Register: τ_{reg}

Länge eines Taktzyklus:

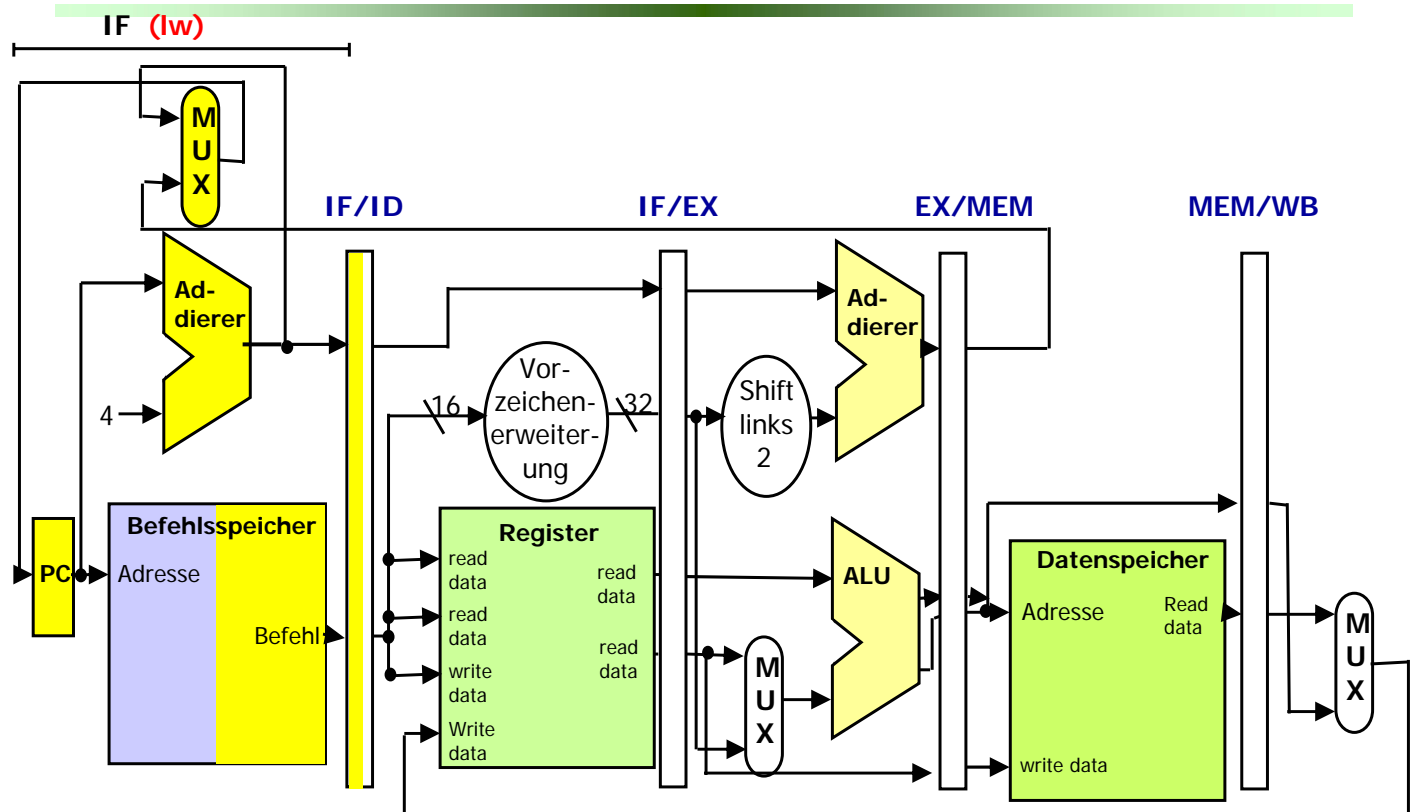
$$\tau = \max\{\tau_1, \tau_2, \dots, \tau_k\} + \tau_{reg}$$



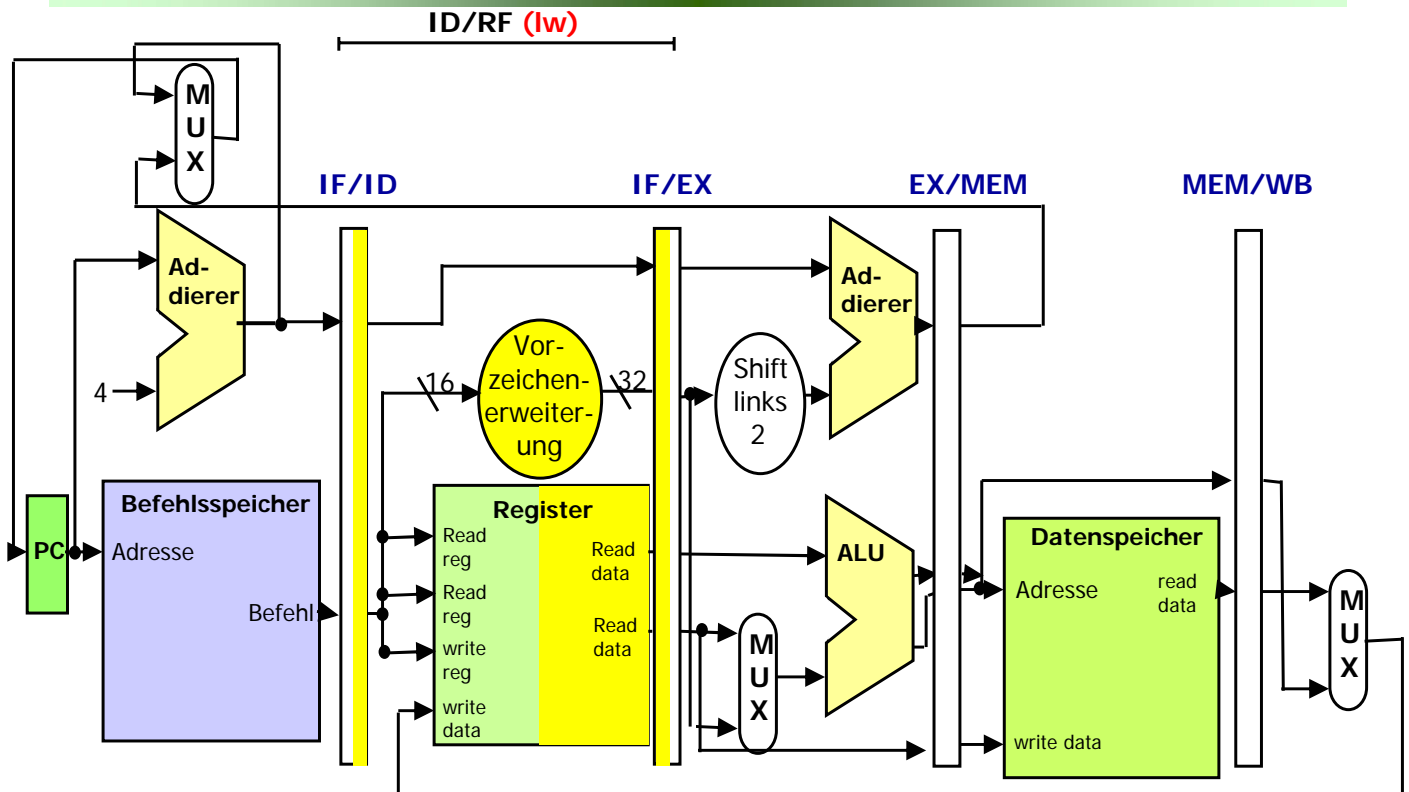
5.4. Pipelining in MIPS Architektur



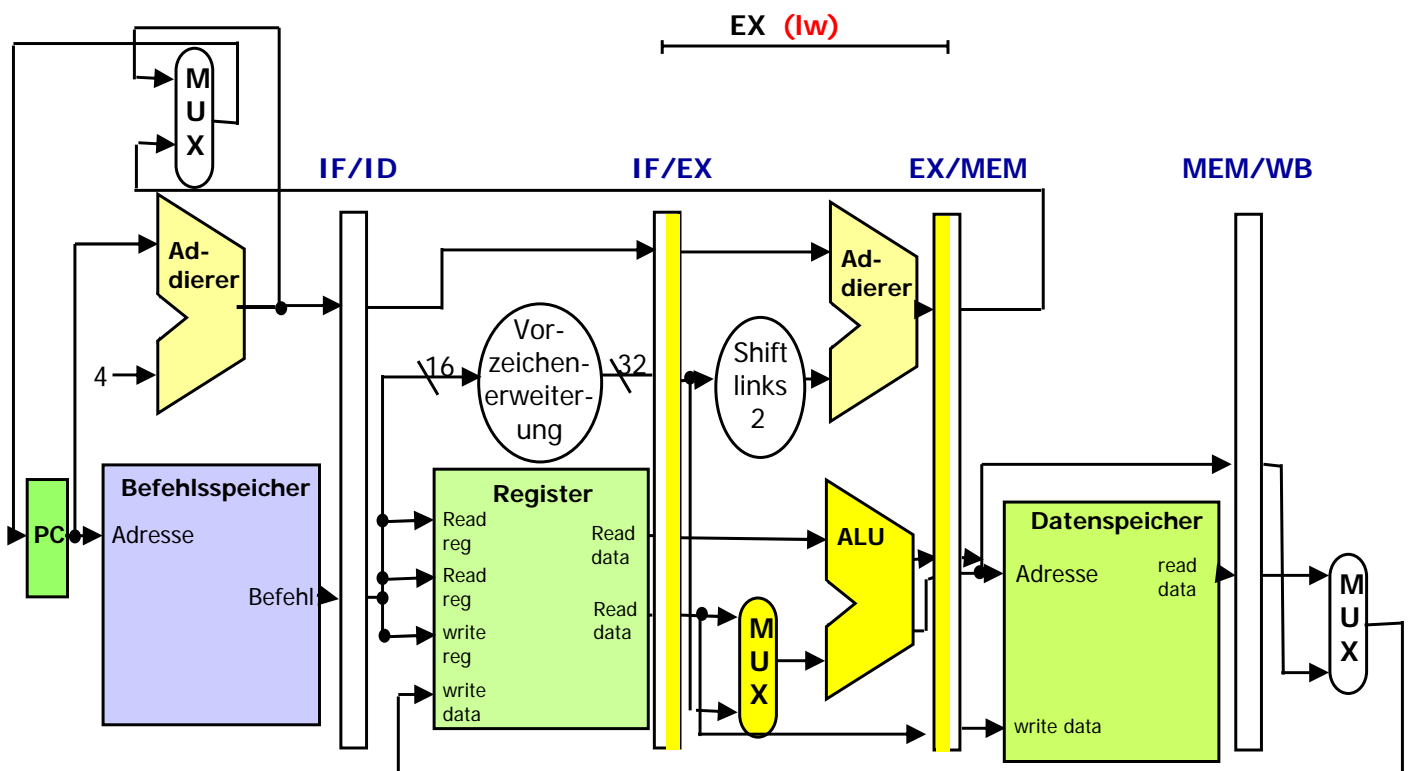
5.4. DLX Pipelinestufen



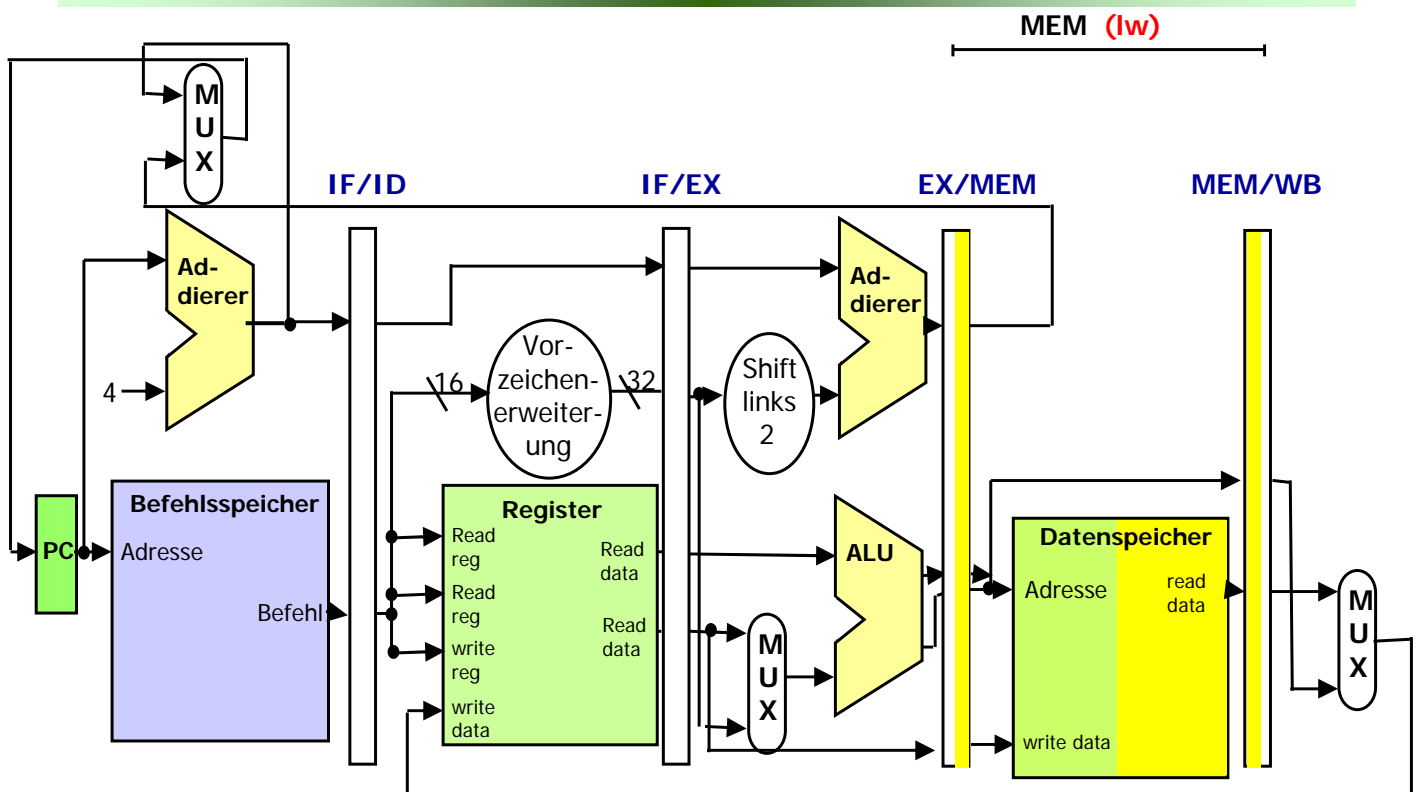
5.4. DLX Pipelinestufen



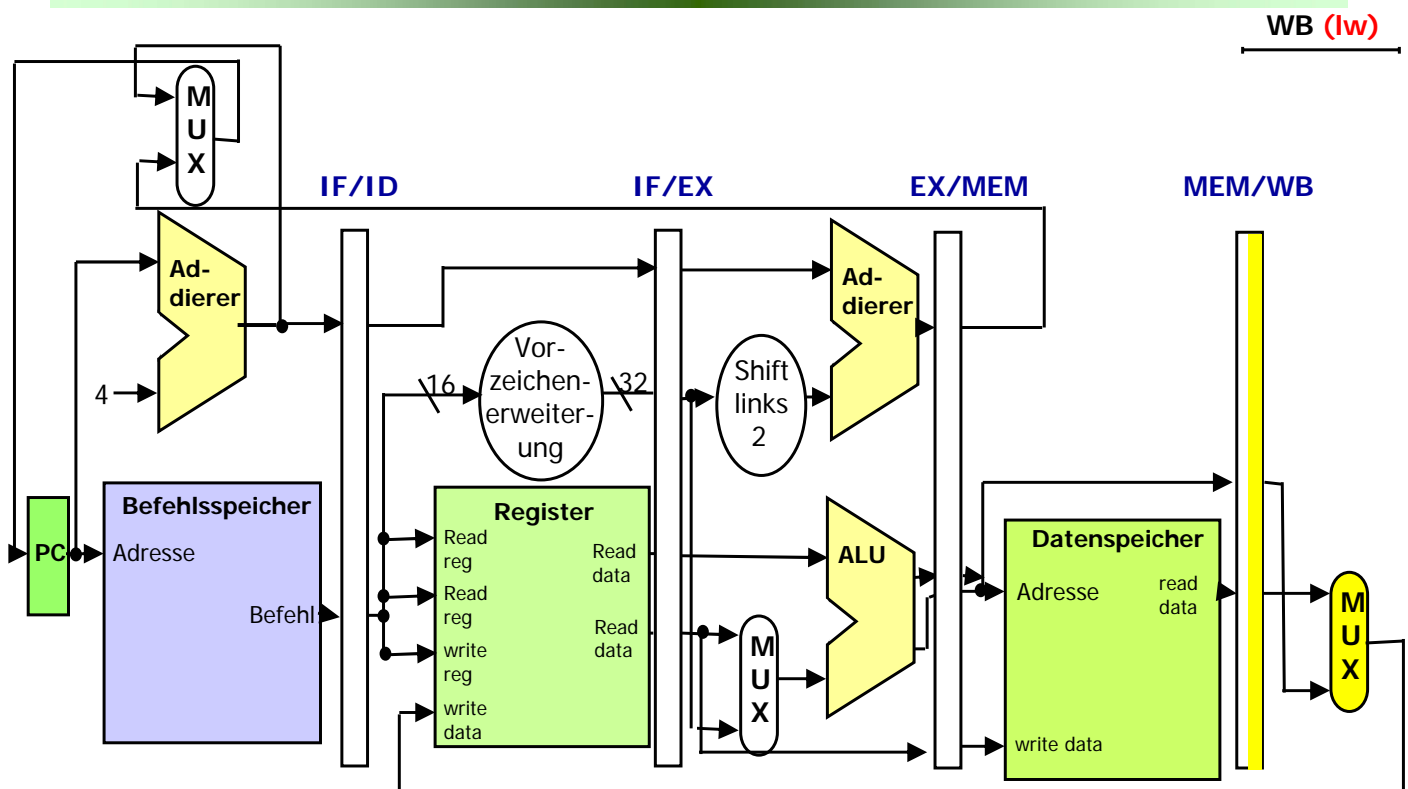
5.4. DLX Pipelinestufen



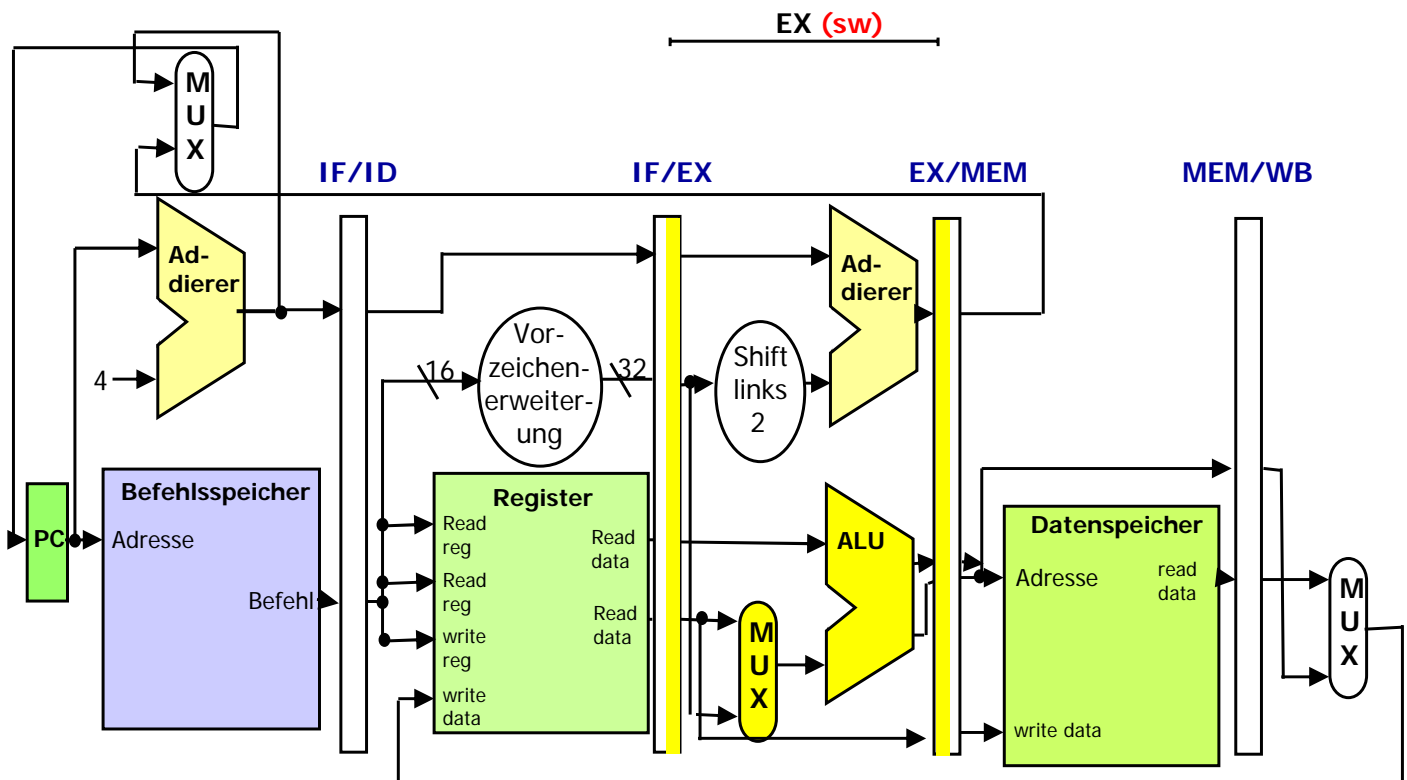
5.4. DLX Pipelinestufen



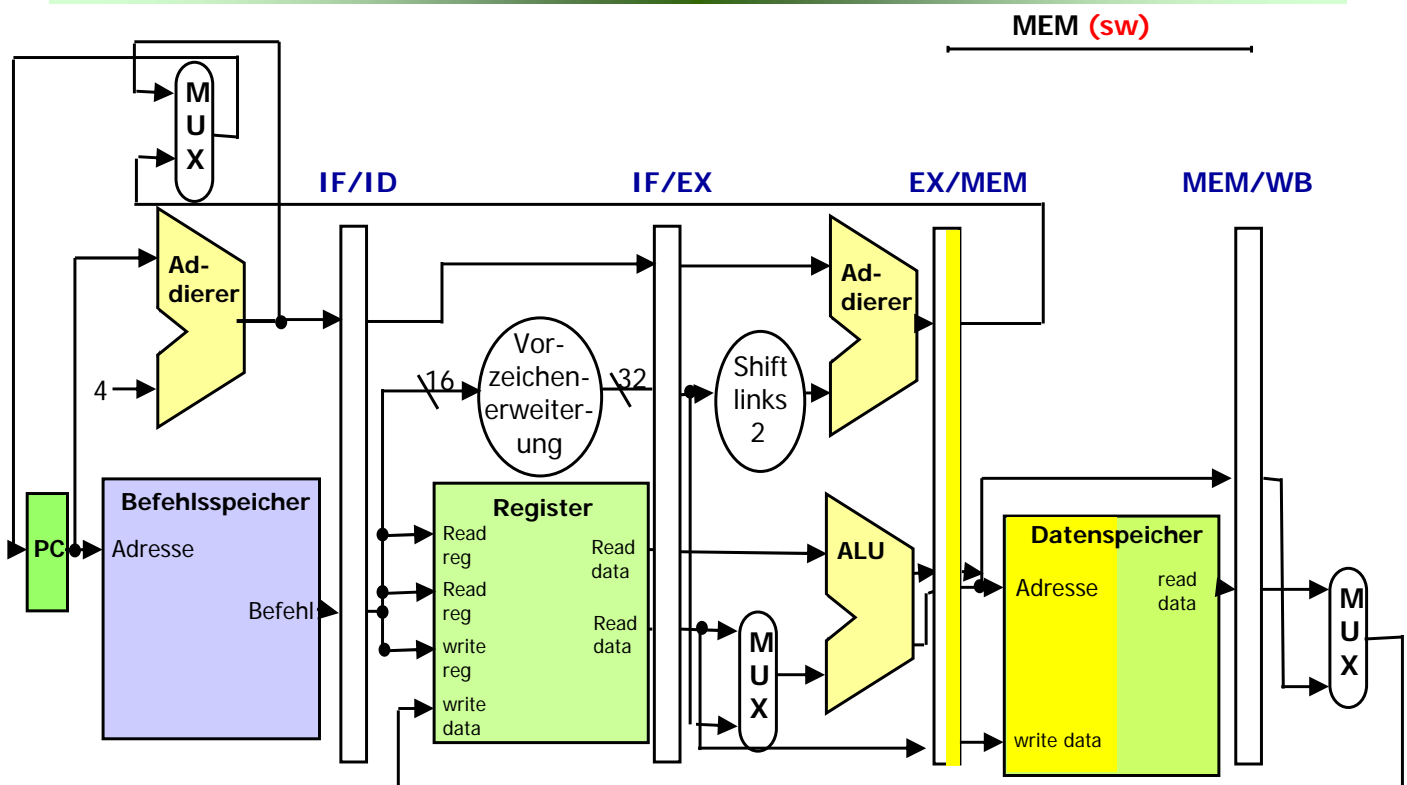
5.4. DLX Pipelinestufen



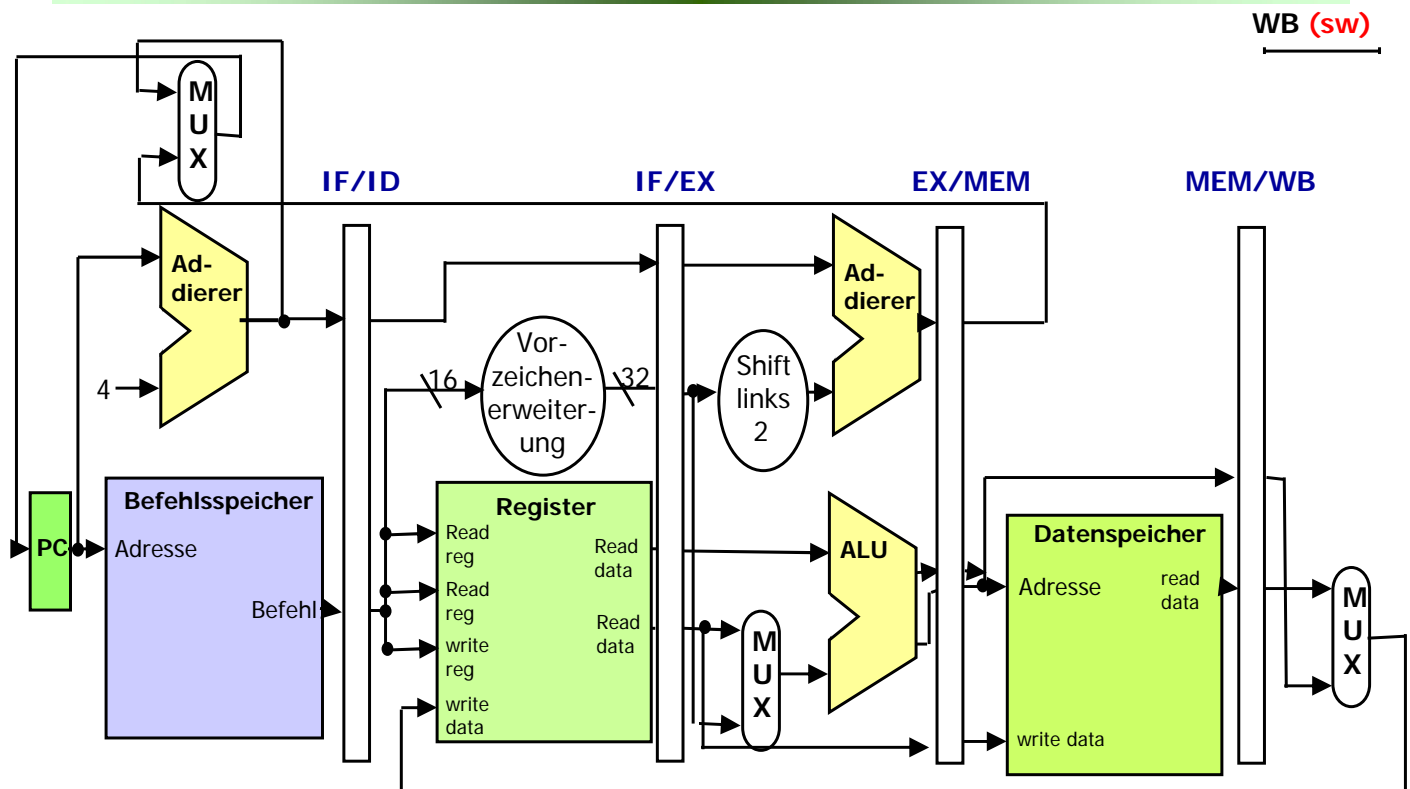
5.4. DLX Pipelinestufen



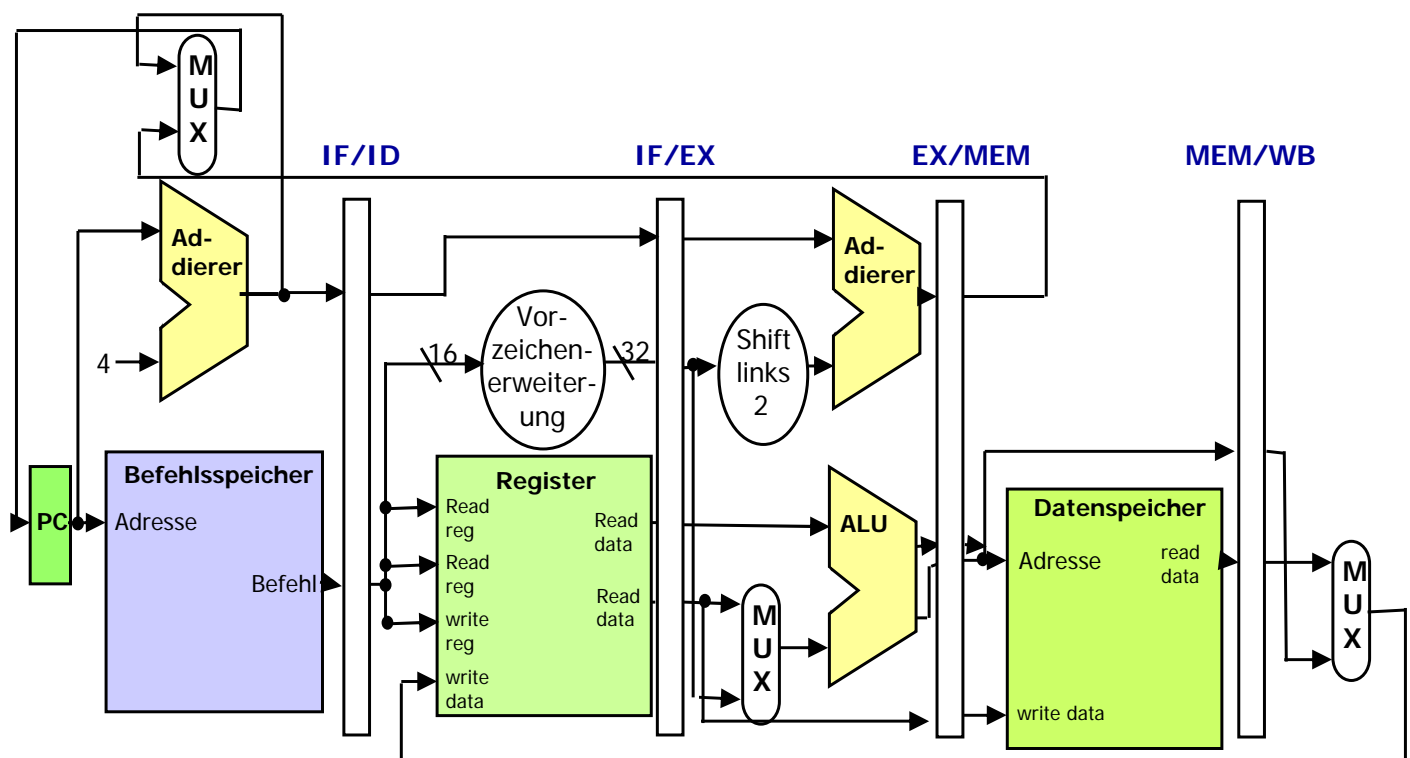
5.4. DLX Pipelinestufen



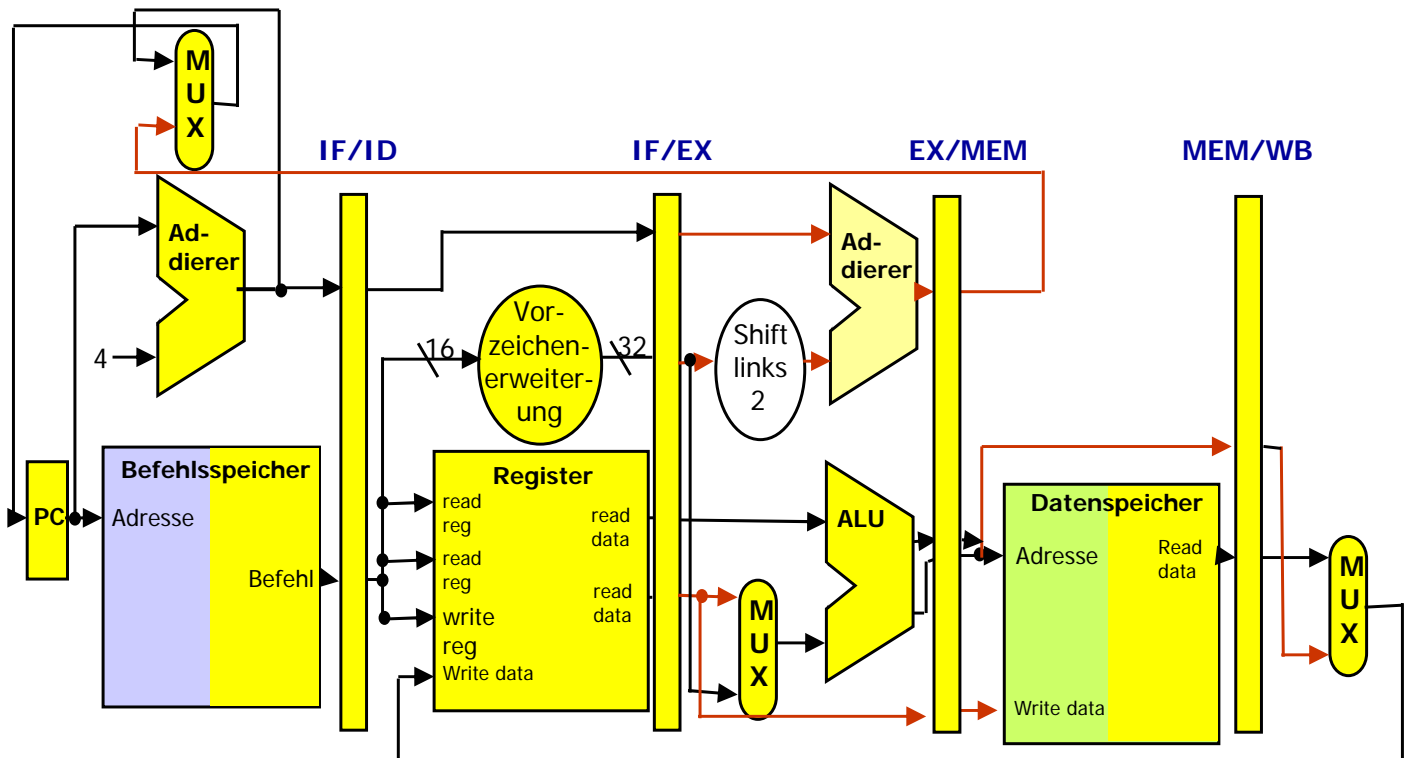
5.4. DLX Pipelinestufen



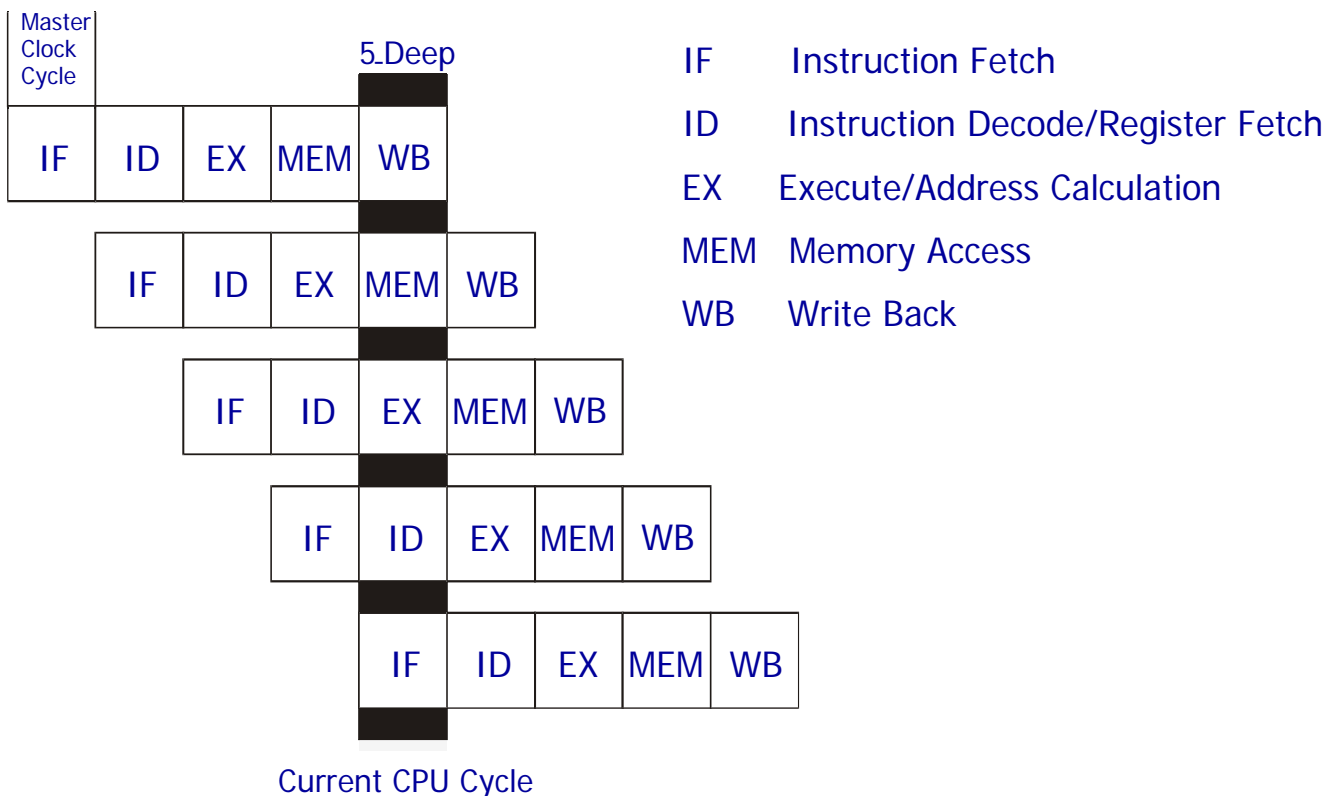
5.4. DLX Pipelinestufen



5.4. DLX Pipelinestufen



5.4. DLX Pipelinestufen



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



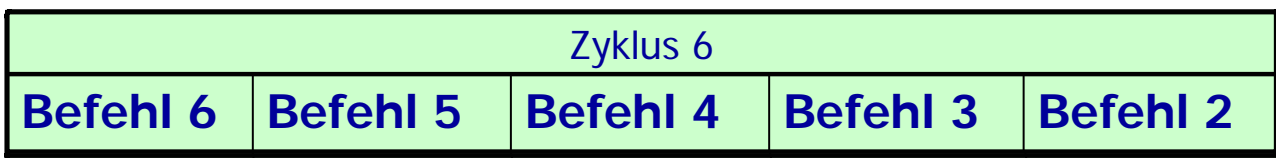
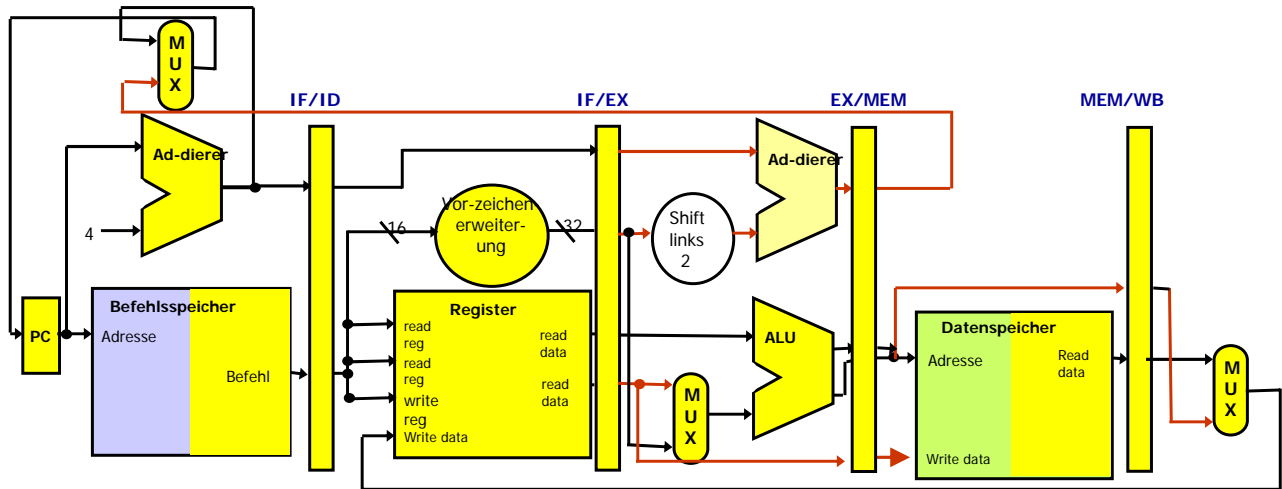
Phasen der Befehlsausführung in einer fünfstufigen Pipeline (DLX-Pipeline)

- ❑ **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 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.



5.5.1 Datenabhängigkeiten

- ❑ Zwischen zwei aufeinander folgenden Befehle $Inst_1$ und $Inst_2$ besteht eine **echte Datenabhängigkeit** (*true dependence*) δ^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*)** δ^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*)** 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.



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`



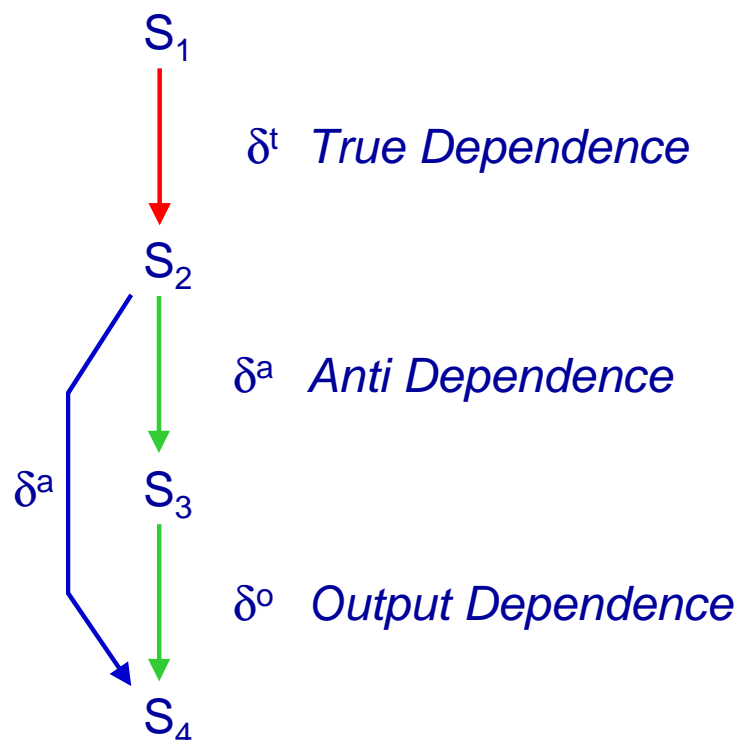
Beispiel: Datenabhängigkeiten

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?

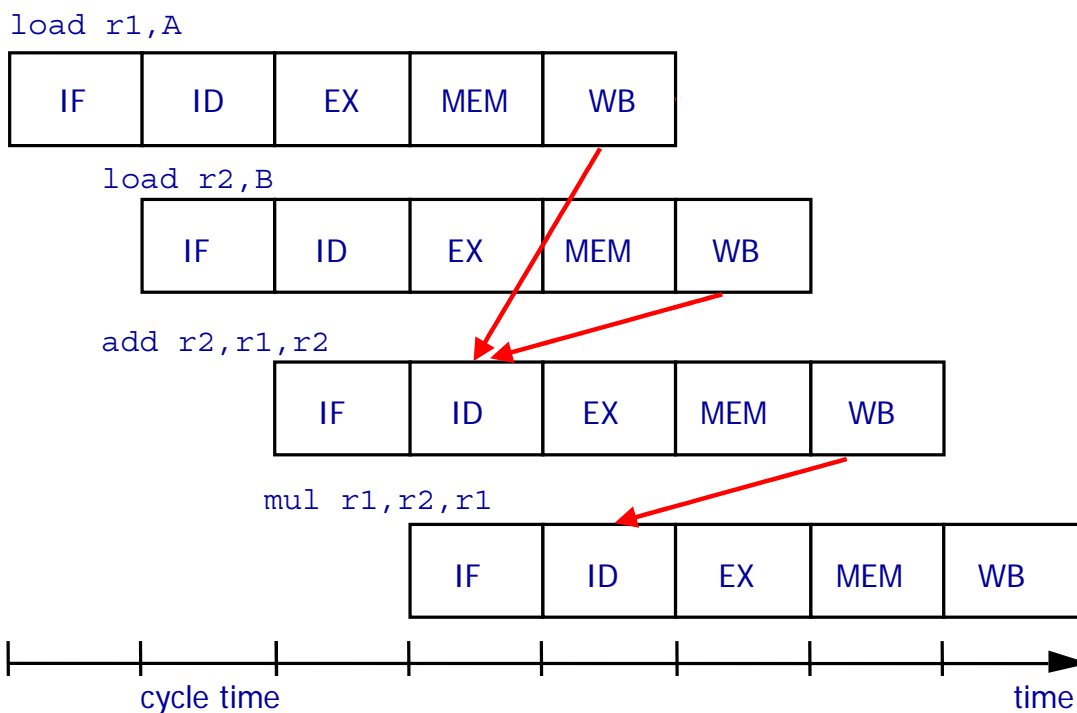
Nein, weil:

- 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



Beispiel

load r1,A



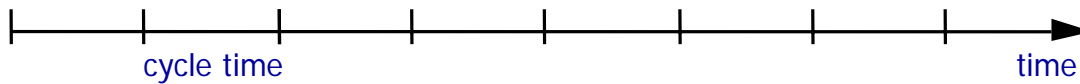
load r2,B



add r2 , r1 , r2

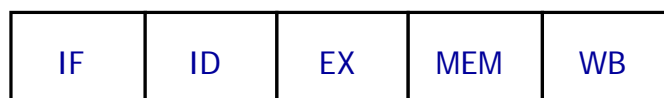


mul r1 , r2 , r1



Fehlzuweisung durch einen Datenkonflikt

add r2,r1,r2

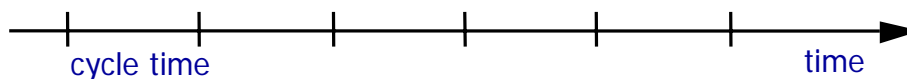
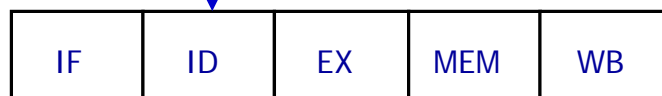


wrong register read!

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.

➤ Statische Befehlsanordnung:

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

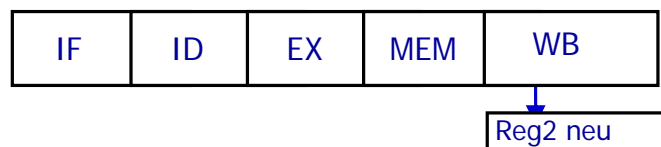


5.5.2 Lösungen für Datenkonflikte

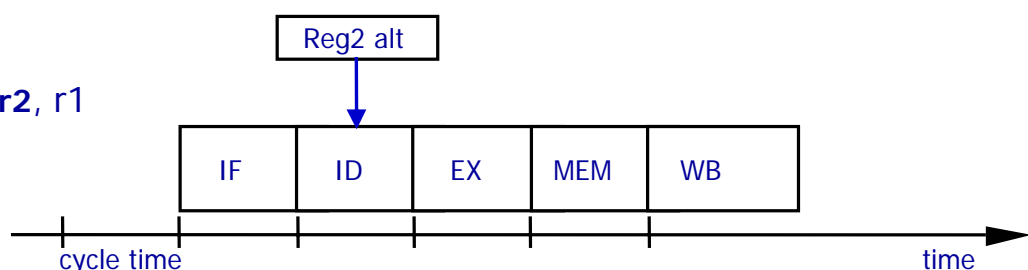
□ Software-Lösungen (*Compiler scheduling*)

➤ `noop` Befehle einfügen

`add r2, r1, r2`



`mul r1, r2, r1`



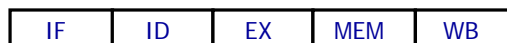
5.5.2 Lösungen für Datenkonflikte

□ Software-Lösungen (*Compiler scheduling*)

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

add r0, r1, r2

sub r3, r0, r4



mul r5, r6, r7



mul r8, r9, r10



5.5.2 Lösungen für Datenkonflikte

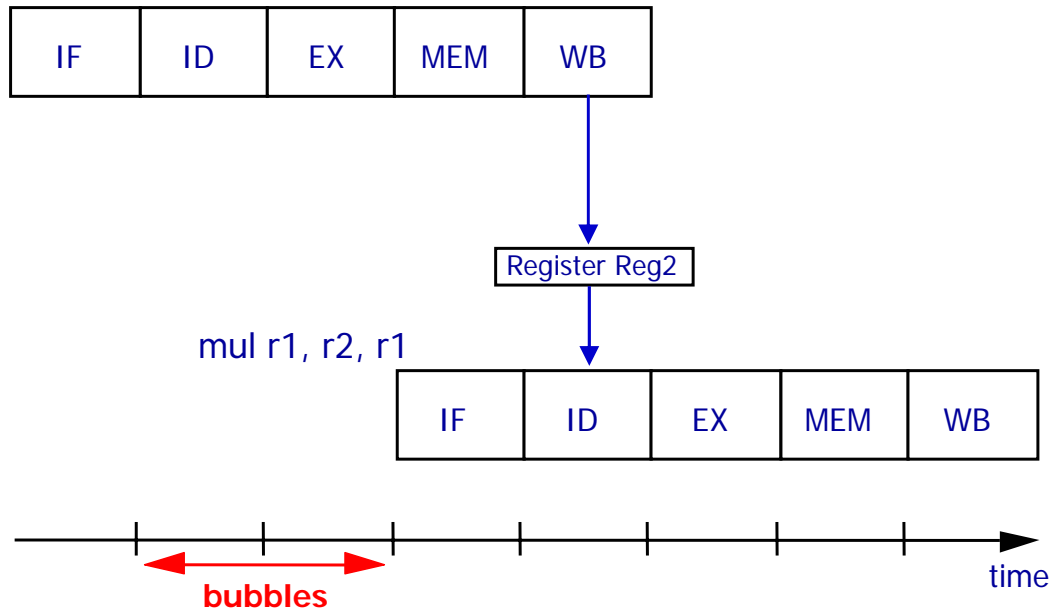
□ Hardware-Lösungen: Konflikt muss per HW entdeckt werden!!

- **Leerlauf der Pipeline:** Die einfachste Art, mit solchen Datenkonflikten umzugehen ist es, Inst₂ 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

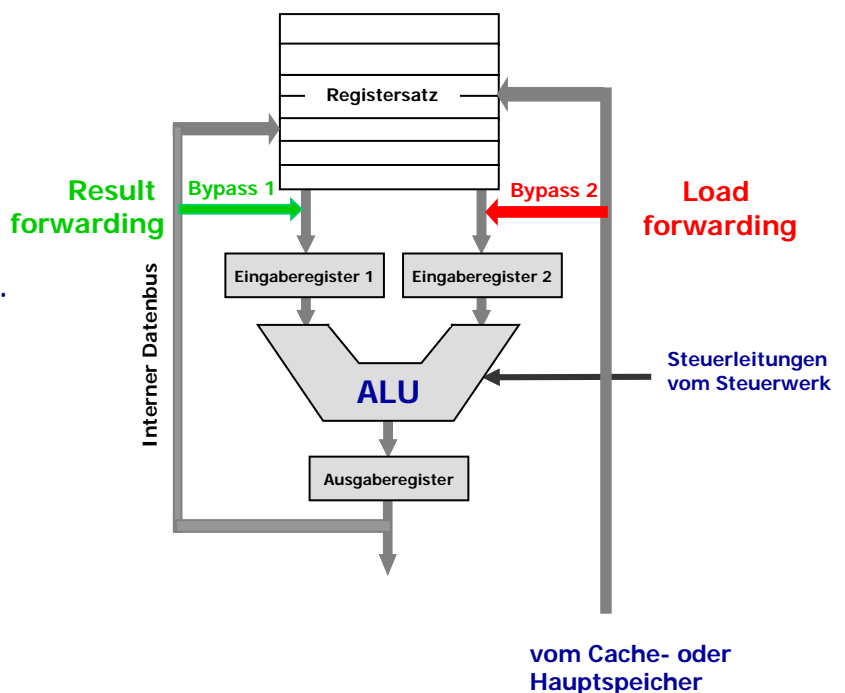
add r2, r1, r2



Forwarding-Techniken

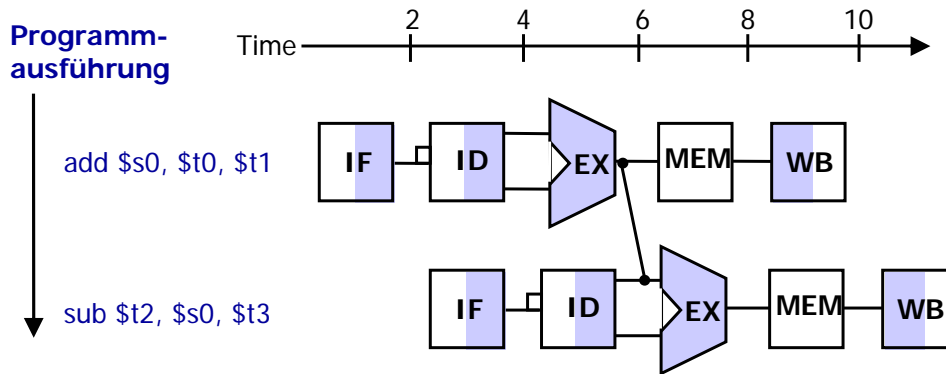
Forwarding:

- 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

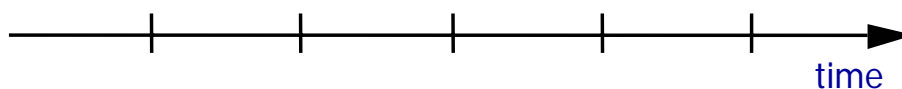
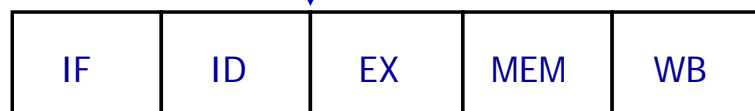


Hardware-Lösung durch *Forwarding*

add \$s0, \$t0, \$t1

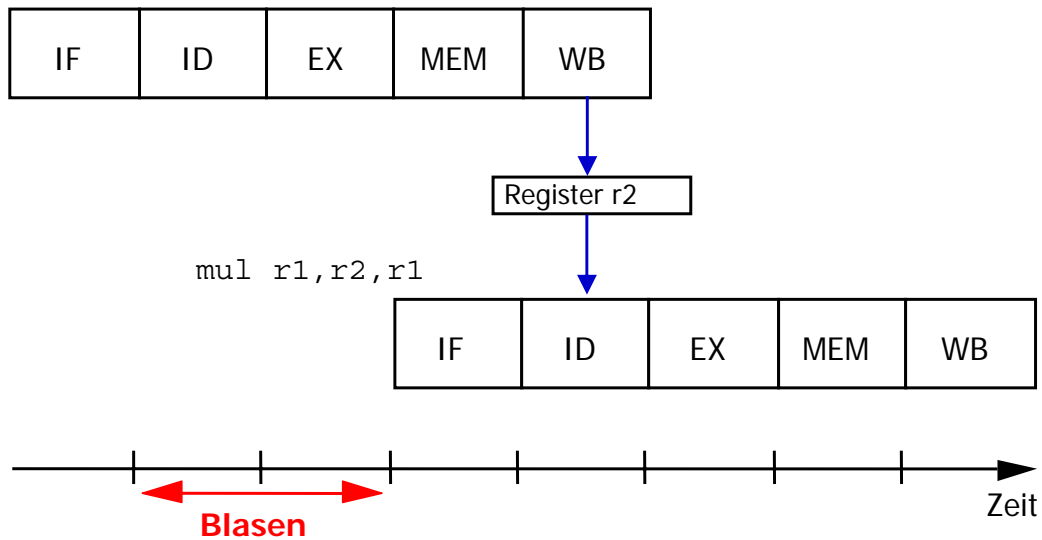


mul r1,r2,r1



Leerlauf der Pipeline: Interlocking

add r2,r1,r2

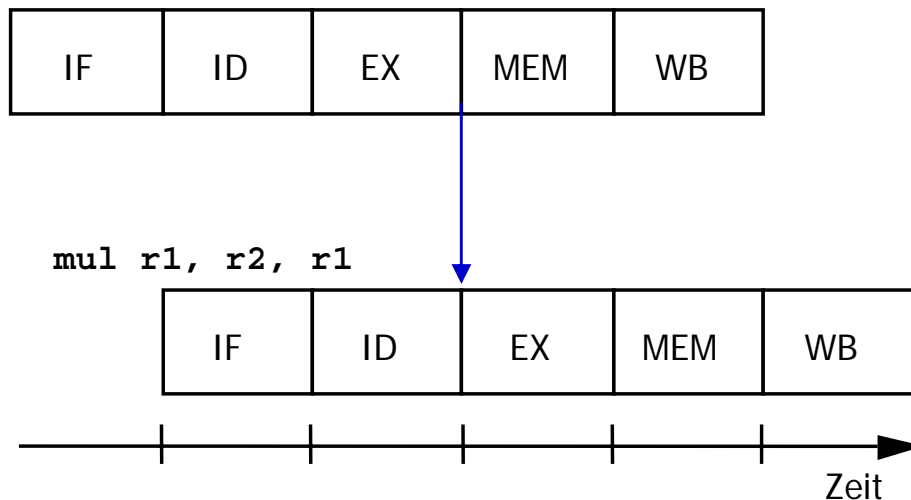


Anhalten des `mul`-Befehls für zwei Takte



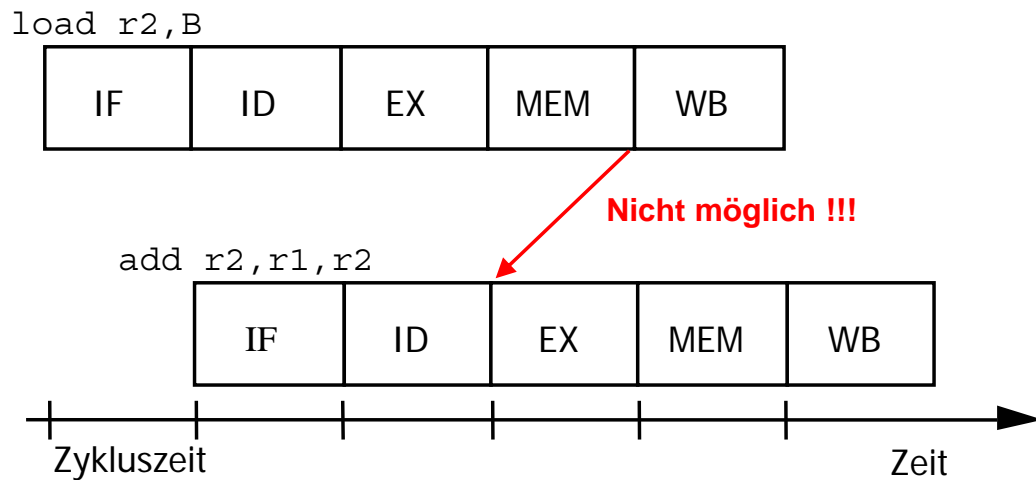
Hardware-Lösung durch *Forwarding*

add r2,r1,r2



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)

