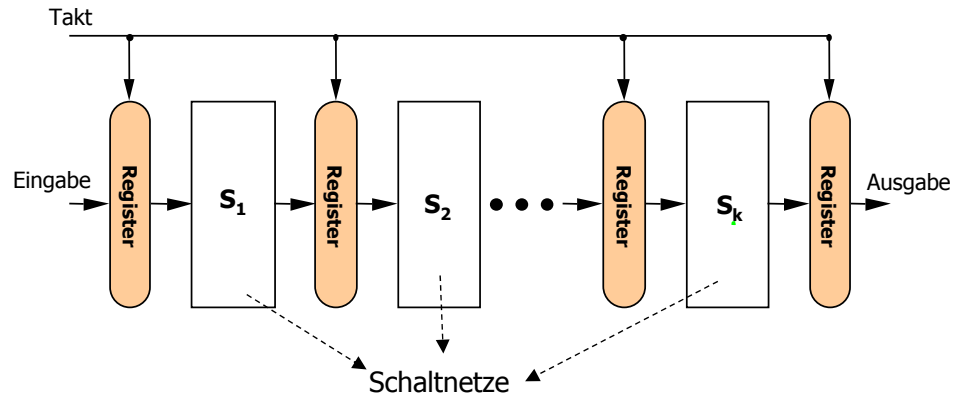


2.2 Pipeline-Stufen und Pipeline-Register



Verzögerungszeiten:

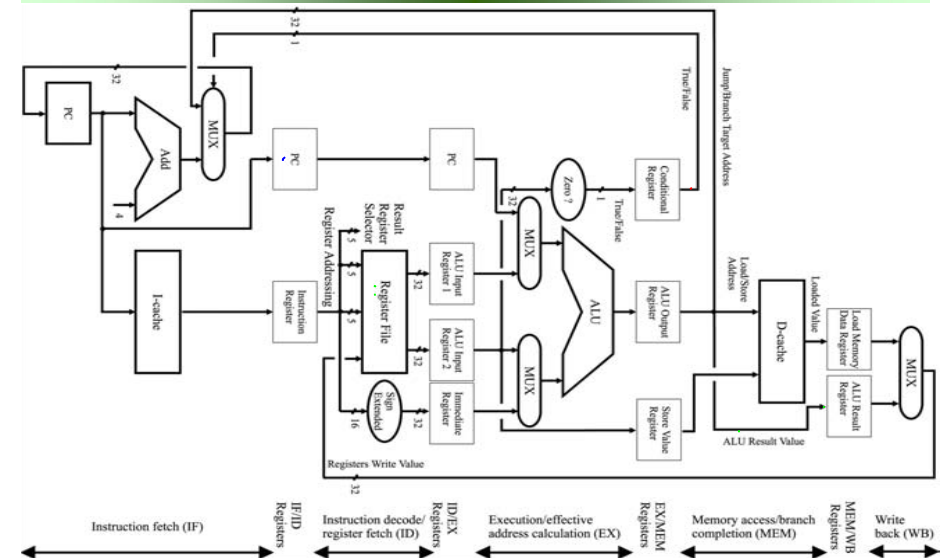
- der Schaltnetze: τ_i ($i = 1, \dots, k$)
- der Pipeline-Register: τ_{reg}

Länge eines Taktzyklus:

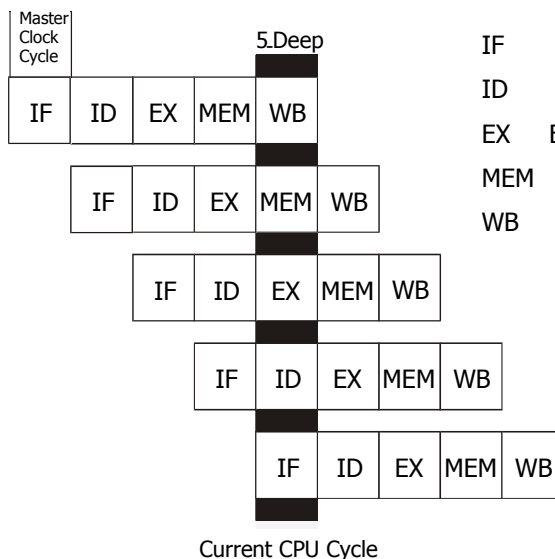
$$\tau = \max\{\tau_1, \tau_2, \dots, \tau_k\} + \tau_{reg}$$



Pipeline (Übersicht)



2.3 Grundlegendes Befehlspipelining

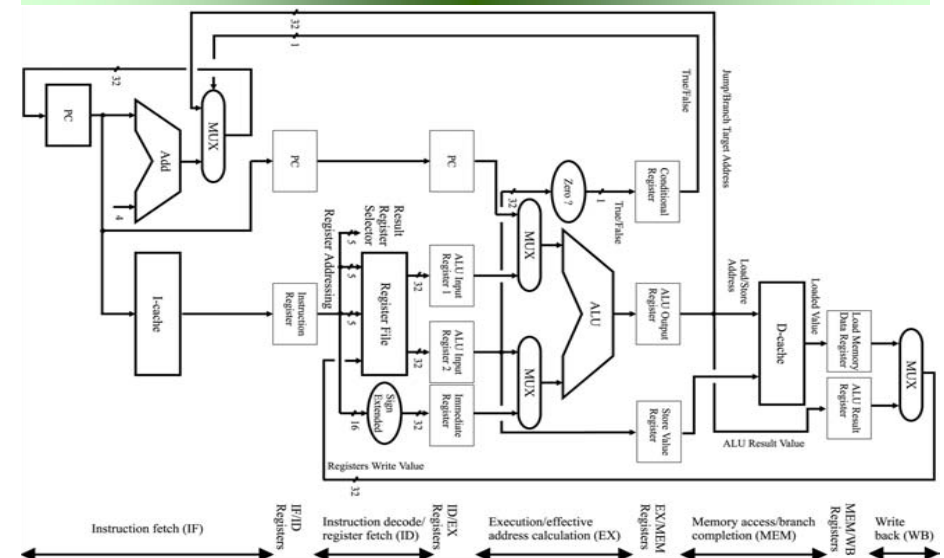


IF Instruction Fetch
ID Instruction Decode/Register Fetch
EX Execute/Address Calculation
MEM Memory Access
WB Write Back

Current CPU Cycle



Pipeline (Übersicht)



2.4 Pipeline-Konflikte

Bei einer gefüllten Pipeline wird pro Taktzyklus ein Befehl beendet.

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

$\text{add } r_0, r_1, r_2 \quad r_0 = r_1 + r_2$
 $\text{add } r_3, r_0, r_4 \quad r_3 = r_0 + r_4$

□ **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.



Datenkonflikte und deren Lösungsmöglichkeiten

- 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.

δ^t $Inst_1:$ $a = b + c$
 $Inst_2:$ $d = a + e$



Datenkonflikte und deren Lösungsmöglichkeiten

- 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.

$Inst_1:$ $a = b + c$
 $Inst_2:$ $b = d + e$



Datenkonflikte und deren Lösungsmöglichkeiten

- 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.

$Inst_1: \delta^o \textcircled{a} = b + c$
 $Inst_2: \textcircled{a} = d + e$



Beispiel: Datenabhängigkeiten

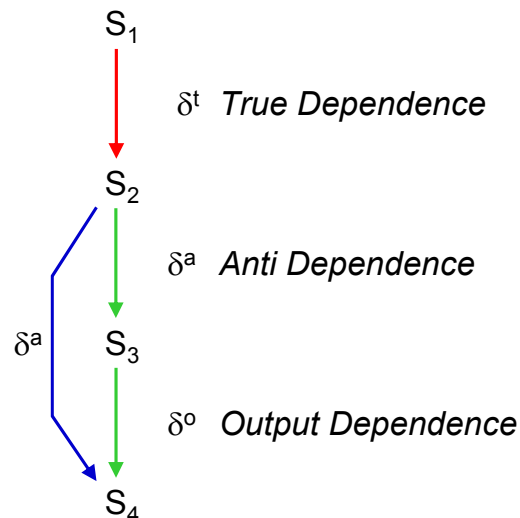
$S_1: \text{ add } r1, r2, 2 \quad \# \quad \textcircled{r1} := r2 + 2$
 $S_2: \text{ add } r4, r1, r3 \quad \# \quad r4 := \textcircled{r1} + \textcircled{r3}$
 $S_3: \text{ mul } r3, r5, 3 \quad \# \quad \textcircled{r3} := r5 * 3$
 $S_4: \text{ mul } r3, r6, 3 \quad \# \quad \textcircled{r3} := r6 * 3$

Diagram illustrating data dependencies between instructions S_1 through S_4 . Red arrows indicate δ^t (True) dependencies, and green arrows indicate δ^a (Anti) and δ^o (Output) dependencies.



Beispiel: Datenabhängigkeiten

$S_1: \text{ add } r1, r2, 2$
 $S_2: \text{ add } r4, r1, r3$
 $S_3: \text{ mul } r3, r5, 3$
 $S_4: \text{ mul } r3, r6, 3$



Datenkonflikte und deren Lösungsmöglichkeiten

Bemerkung:

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



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 Schreibstufe der Lesestufe vorangeht.
- **Schreibe-nach-Schreibe-Konflikt** (write after read, WAR): 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?

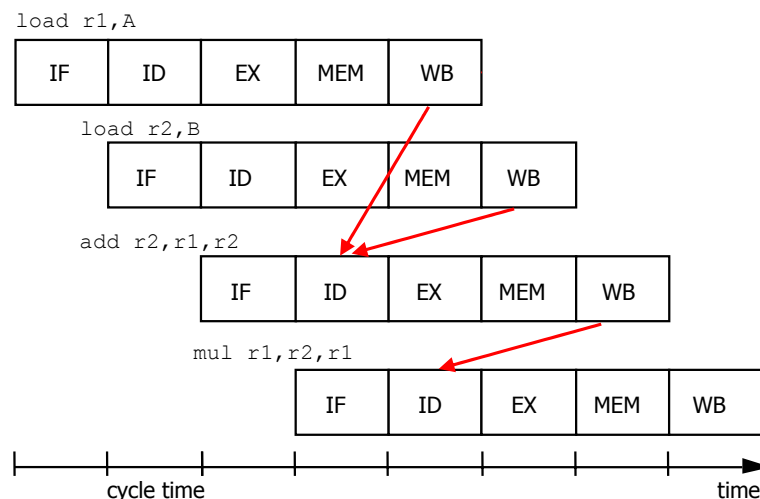
Nein, weil:

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

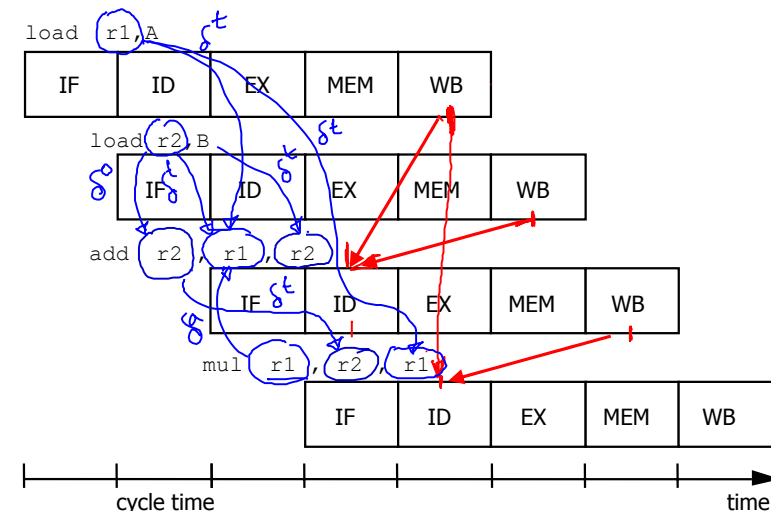
WAR und WAW treten bei "komplexeren" Pipelines auf



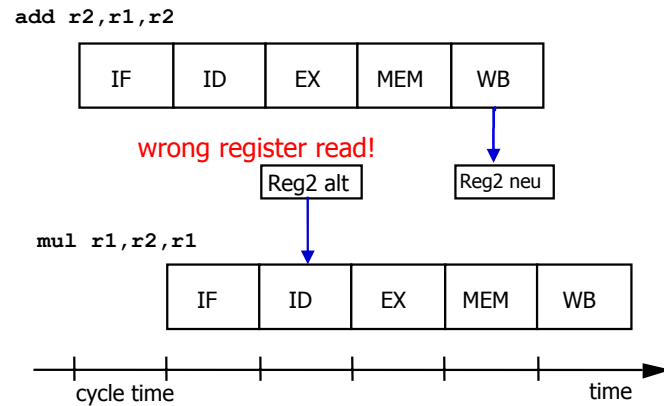
Datenkonflikte in einer Befehls-Pipeline



Datenkonflikte in einer Befehls-Pipeline



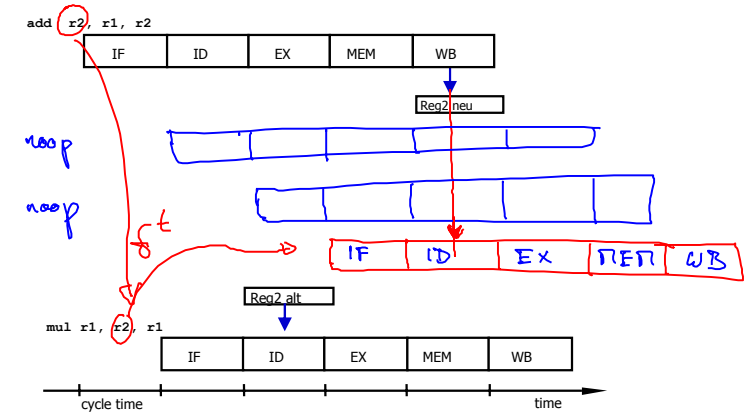
Fehlzuweisung durch einen Datenkonflikt



Lösungen für Datenkonflikte

Software-Lösungen (Compiler scheduling)

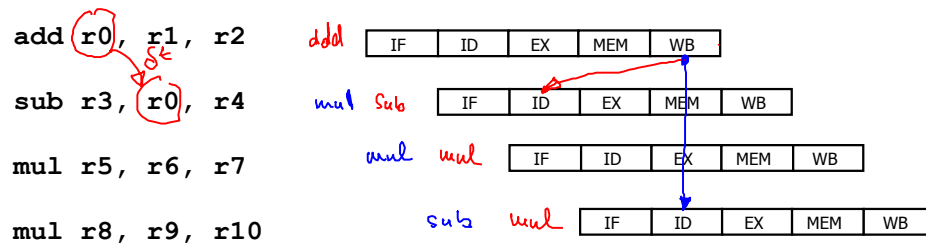
- noop Befehle einfügen



Lösungen für Datenkonflikte

Software-Lösungen (Compiler scheduling)

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



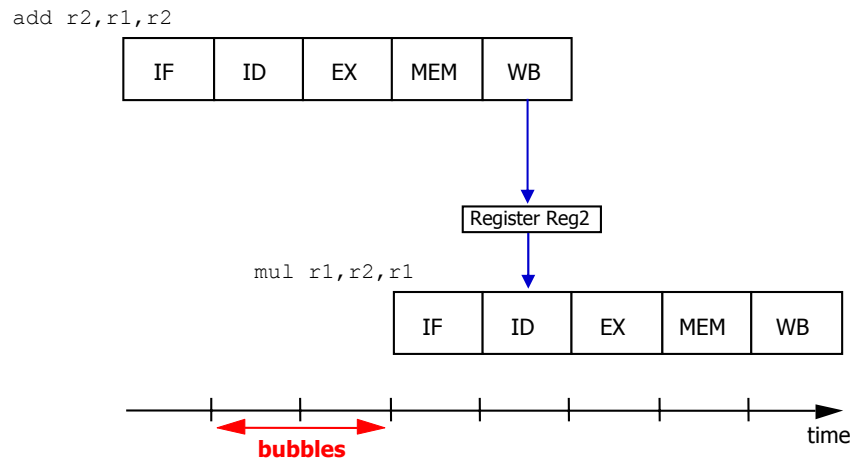
Lösungen für Datenkonflikte

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

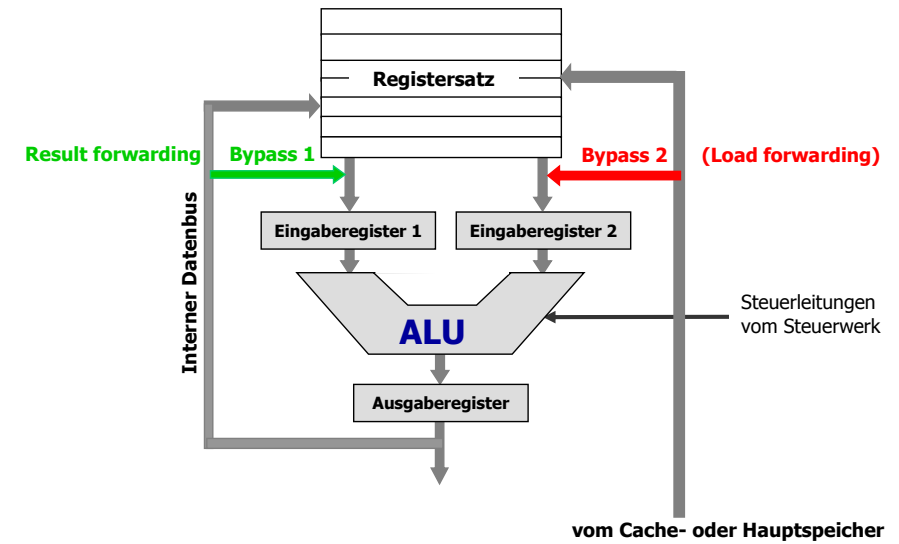
- Leerlauf der Pipeline:** Die einfachste Art, mit solchen Datenkonflikten umzugehen ist es, Inst₂ in der Pipeline für zwei Takte anzuhalten. Auch als **Pipeline-Sperrung** (interlocking) oder **Pipeline-Leerlauf** (stalling) bezeichnet.
- Forwarding:** Wenn ein Datenkonflikt erkannt wird, so sorgt eine 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



Forwarding-Techniken



Hardware-Lösung durch Forwarding

