

2.4.1 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.



Lösungen für Datenkonflikte

□ Hardware-Lösungen (Dynamische Verfahren):

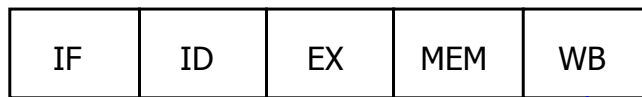
Konflikt muss per Hardware entdeckt werden!!

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



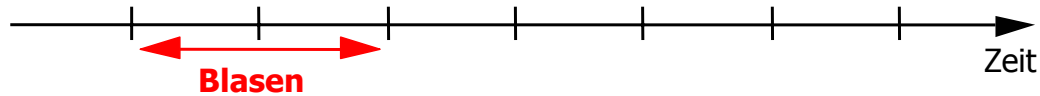
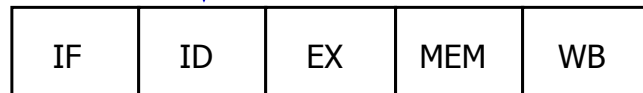
Leerlauf der Pipeline: Interlocking

add r2,r1,r2



Register r2

mul r1,r2,r1



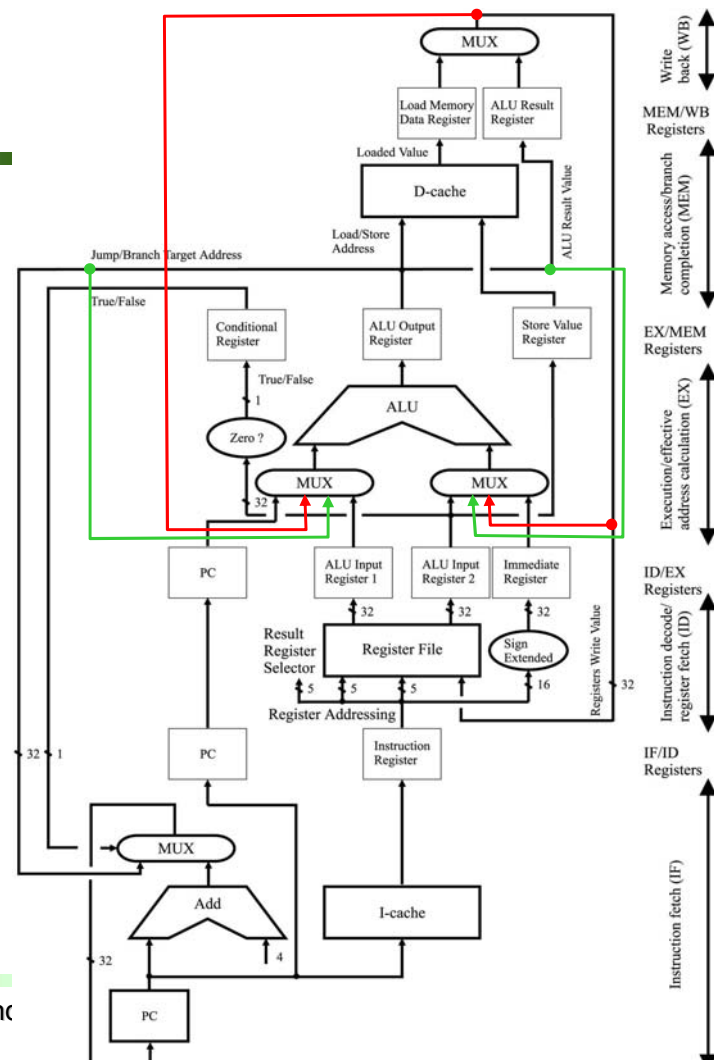
Anhalten des `mul`-Befehls für zwei Takte



Forwarding

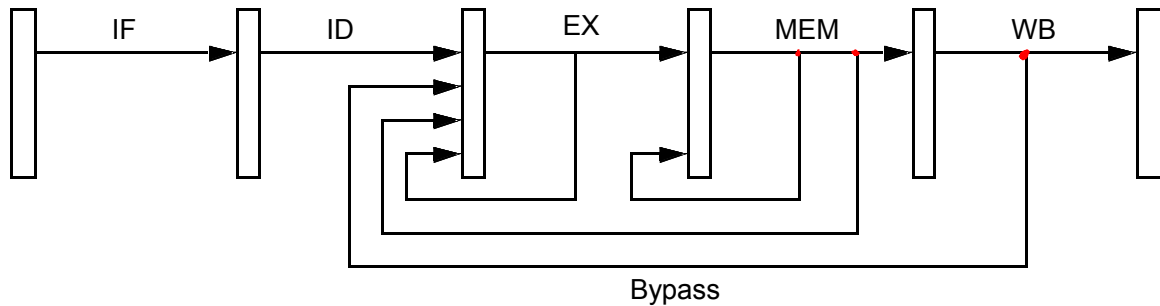
Forwarding:

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



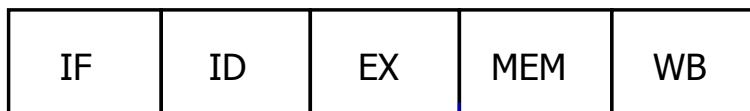
Forwarding Techniken

- Hardware-Aufwand
- Forwarding-Logik und zusätzliche Datenpfade

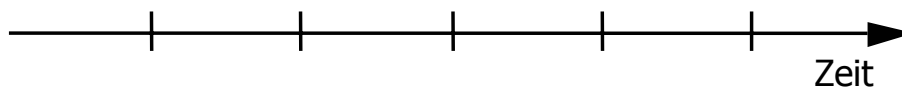
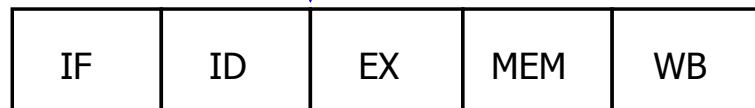


Hardware-Lösung durch *Forwarding*

`add r2, r1, r2`



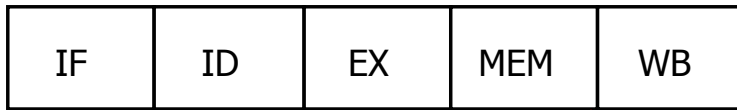
`mul r1, r2, r1`



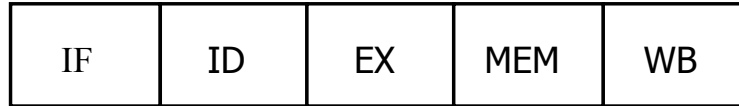
Problem: Nicht alle Konflikte sind alleine durch Forwarding behebbar !!!

Beispiel: Speicherzugriff (z. B. Ladebefehl)

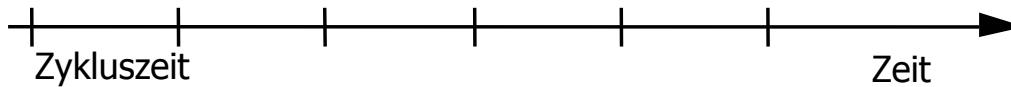
load r2,B



add r2,r1,r2



Nicht möglich !!!



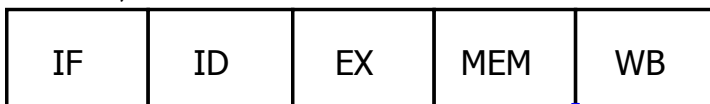
- EX-Stufe berechnet die effektive Adresse
- Der add-Befehl muss angehalten werden, bis die geladenen Daten im Ladewertregister der MEM-Stufe verfügbar sind



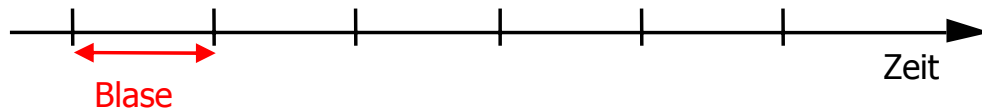
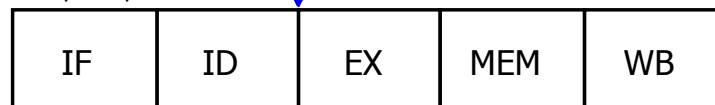
Lösung: Pipelineverzögerungen (bubble)

mit Forwarding

load r2,B



add r2,r1,r2



2.4.2 Ressourcenkonflikte

□ Ressourcenkonflikt:

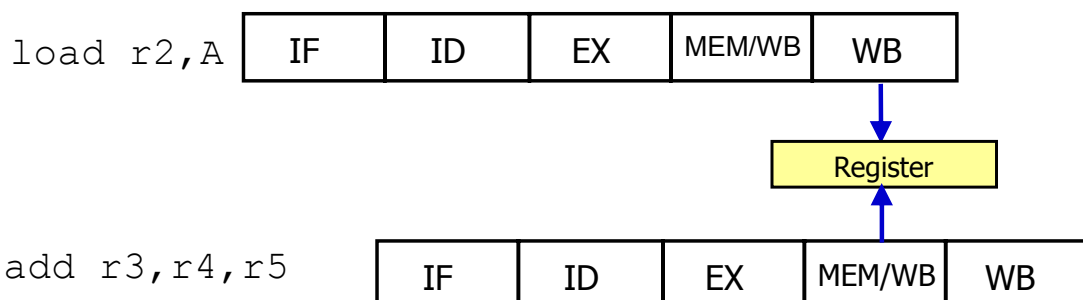
- Treten auf, wenn zwei oder mehr Befehle gleichzeitig dieselbe Ressource benötigen, auf die aber nur einmal zugegriffen werden kann
- Treten bei einer einfachen Pipeline, wie die DLX-Pipeline, nicht auf
- **Ziel beim Pipeline-Entwurf:** Ressourcenkonflikte möglichst zu vermeiden und dort, wo sie nicht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



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.



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



2.4.3 Steuerflusskonflikte

➤ Programmsteuerbefehle:

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

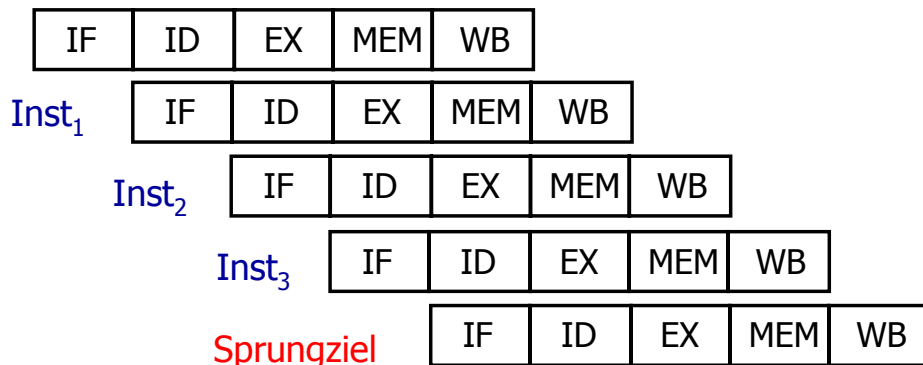
➤ Steuerflussabhängigkeiten verursachen **Steuerflusskonflikte** in der DLX-Pipeline, da

- 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

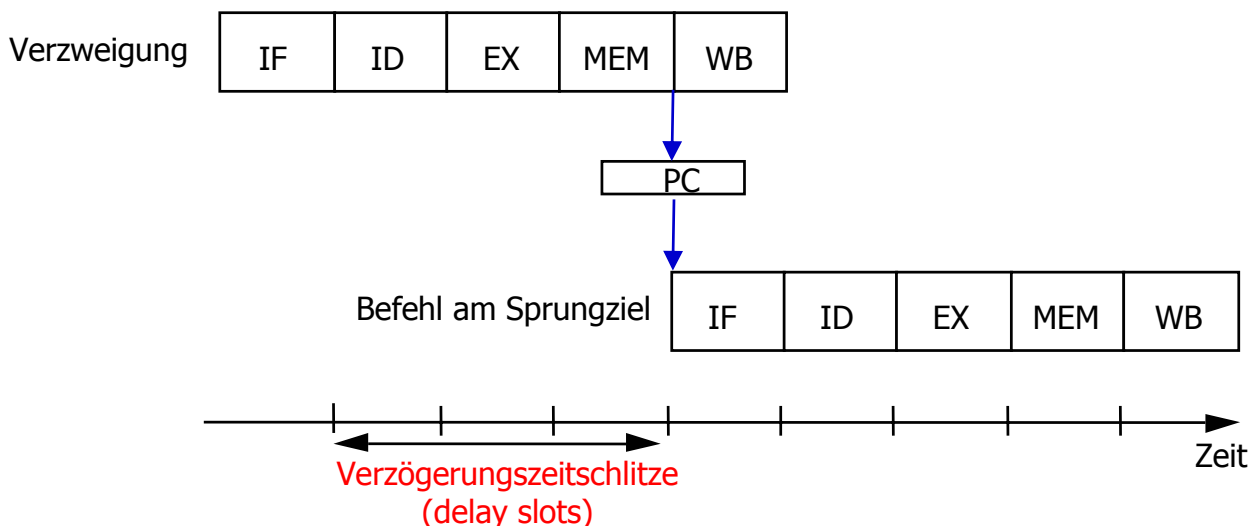
Sprungbefehl



- 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 Verzögerungszeitschlitze (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 (Sprungzieladdress-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

