Rechnerstrukturen

Vorlesung im Sommersemester 2006

Prof. Dr. Wolfgang Karl

Universität Karlsruhe (TH)

Fakultät für Informatik

Institut für Technische Informatik

Kapitel 1: Grundlagen

1.5 Einführung: Klassifizierung von Rechnerarchitekturen

- Klassifikationen
  - Aufspannen von Entwurfsräumen
  - Aufzeigen von Entwurfsalternativen
  - Klassifikationsschemata versuchen, der Vielfalt von Rechnerarchitekturen eine Ordnungsstruktur zu geben
  - Frühe Klassifikationen konzentrieren sich auf die Hardware-Struktur
    - Anordnung und Organisation der Verarbeitungselemente
    - Operationsprinzip

- Klassifizierung nach M. Flynn
  - Zweidimensionale Klassifizierung
    - Hauptkriterien:
      - Zahl der Befehlsströme und
      - Zahl der Datenströme sind.
    - Merkmale:
      - Ein Rechner bearbeitet zu einem gegebenen Zeitpunkt einen oder mehr als einen Befehl.
      - Ein Rechner bearbeitet zu einem gegebenen Zeitpunkt einen oder mehr als einen Datenwert.

- Klassifizierung nach M. Flynn
  - Vier Klassen von Rechnerarchitekturen
    - SISD Single Instruction – Single Data
      - Uniprozessor
    - SIMD Single Instruction – Multiple Data
      - Vektorrechner, Feldrechner
    - MISD Multiple Instructions – Single Data
      - ?
    - MIMD Multiple Instructions – Multiple Data
      - Multiprozessor

- Diskussion der Flynnschen Klassifizierung
  - Schwachpunkte:
    - Sehr hohes Abstraktionsniveau ==> sehr unterschiedliche Rechnerarchitekturen fallen in die gleiche Klasse.
    - Viele moderne Parallelarbeitstechniken (z.B. Superskalar, Befehlspipelining, VLIW) lassen sich überhaupt nicht einordnen.
    - In heutigen Rechnern findet man die Kombination mehrerer Techniken.
      Beispiel: Vektorrechner-Multiprozessoren fallen in eine als MIMD/SIMD zu bezeichnende Klasse.
    - Die Klasse MISD wurde nur der Systematik wegen aufgeführt.

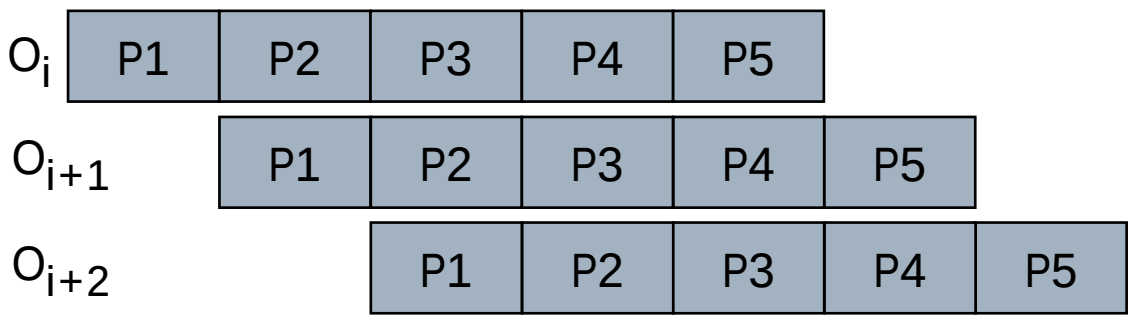
- Diskussion von Klassifikationen
  - Kennzeichnend für moderne Rechnerstrukturen:
    - Prinzip der Virtualität:
      - Mit verschiedenen Techniken und Mechanismen in der Hardware können auf einer Maschine verschiedene parallele Programmiermodelle unterstützt werden
      - Die zugrunde liegende Organisation und Architektur ist weitgehend transparent
    - Allgemeiner Trend in der Rechnerarchitektur
      - Verwendung von Standardkomponenten
      - Verständnis über die Implementierungstechniken zur Virtualität
      - Keine allgemeingültige Klassifikation!

- **Kapitel 2: Zentraleinheiten**

2.1: Parallelismus auf Befehlsebene

- Überblick
  - Pipelining
    - Überlappte Ausführung der Phasen des Maschinenbefehlszyklus
      - Nützen alle Prozessoren seit 1985 aus
  - Nebenläufigkeit
    - Zu einem Zeitpunkt gleichzeitige Ausführung mehrerer Maschinenbefehle zu
      - Dynamische Ansätze
        - » Superskalare Mikroprozessoren
      - Statische Ansätze
        - » VLIW, EPIC

- Pipelining
  - *Pipelining auf einer Maschine liegt dann vor, wenn die Bearbeitung eines Objektes in Teilschritte zerlegt und diese in einer sequentiellen Folge (Phasen der Pipeline) ausgeführt werden. Die Phasen der Pipeline können für verschiedene Objekte überlappt abgearbeitet werden. (Bode 95)*



- Pipelining
  - Befehlspipelining (Instruction Pipelining):
    - Zerlegung der Ausführung einer Maschinenoperation in Teilphasen, die dann von hintereinander geschalteten Verarbeitungseinheiten taktsynchron bearbeitet werden, wobei jede Einheit genau eine spezielle Teiloperation ausführt.
  - Pipeline:
    - Gesamtheit der Verarbeitungseinheiten
  - Pipeline-Stufe:
    - Stufen der Pipeline, die jeweils durch Pipeline-Register getrennt sind

- RISC (Reduced Instruction Set Computers)
  - Einfache Maschinenbefehle
    - Einheitliches und festes Befehlsformat
  - Load/Store Architektur
    - Befehle arbeiten auf Registeroperanden
    - Lade- und Speicherbefehle greifen auf Speicher zu
  - Einzyklus-Maschinenbefehle
    - Effizientes Pipelining des Maschinenbefehlszyklus
    - Einheitliches Zeitverhalten der Maschinenbefehle, wovon nur Lade- und Speicherbefehle sowie die Verzweigungsbefehle abweichen
  - Optimierende Compiler
    - Reduzierung der Befehle im Programm

- Implementierung eines RISC-Befehlssatzes
  - Befehlsbereitstellungsphase (Instruction Fetch, Befehl holen)
    - Der Befehl, der durch den Befehlszähler adressiert ist, wird aus dem Speicher (Befehls-Cache) geholt. Der Befehlszähler wird um 4 weitergeschaltet.

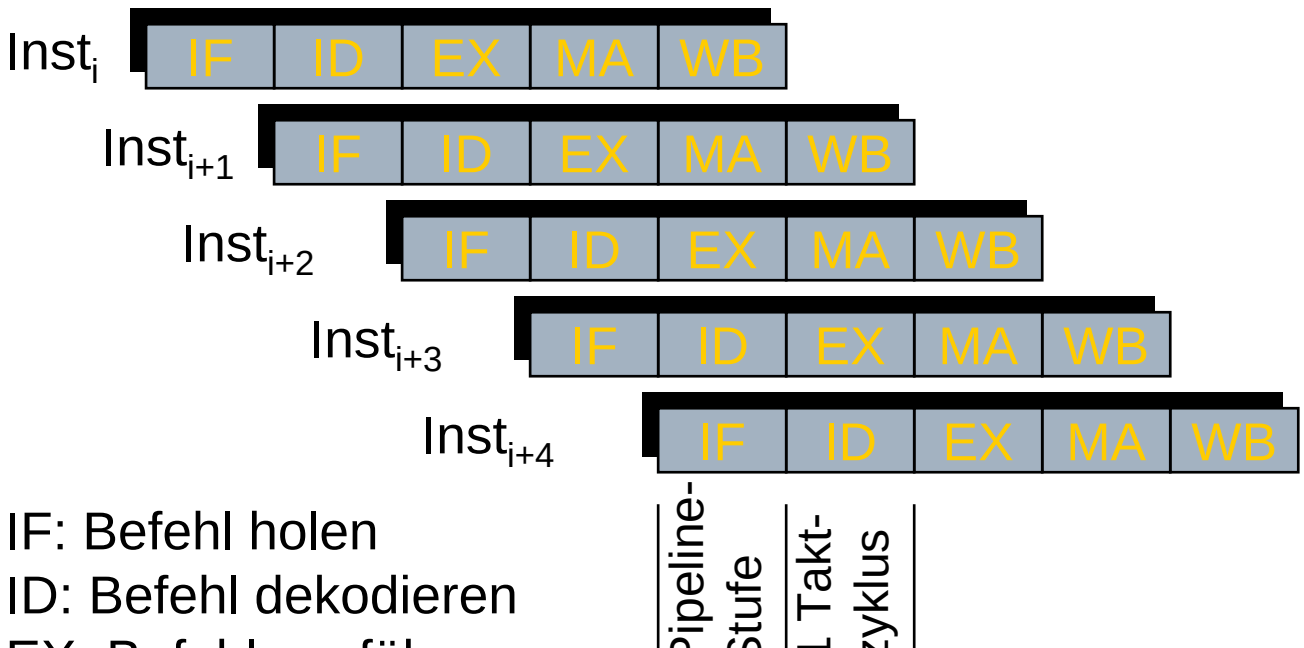
- Implementierung eines RISC-Befehlssatzes
  - Dekodier- und Operandenbereitstellungsphase (Instruction Decode and Register Fetch)
    - Aus dem Opcode werden prozessorinterne Steuersignale erzeugt
    - Die Inhalte der im Befehl adressierten Register werden gelesen
    - Fixed-Field Decoding
      - Gleichzeitiges Dekodieren und Lesen der Register
    - Weitere Aktionen
      - Vergleich der Registerinhalte bei einer Verzweigung
      - Berechnung der Verzweigungsadresse

- Implementierung eines RISC-Befehlssatzes
  - Ausführungsphase / Berechnung der effektiven Adresse (Execution /Effective Address Computation)
    - Register-Register-ALU-Operation
      - Ausführung der im Opcode spezifizierten ALU-Operation mit den aus den Registern gelesenen Werten
    - Register-Immediate-ALU-Operation
      - Ausführung der ALU-Operation mit einem aus einem Register gelesenen Wert und der Vorzeichen-erweiterten Konstanten
    - Speicherzugriff
      - Berechnung der effektiven Adresse des Speicheroperanden

- Implementierung eines RISC-Befehlssatzes
  - Speicherzugriffsphase (Memory Access)
    - Lade-Operartion
      - Lesen der Daten aus dem Speicher unter Verwendung der im vorangegangen Zyklus berechneten effektiven Adresse
    - Speicher-Operation
      - Schreiben der Daten in den Speicher unter Verwendung der effektiven Adresse

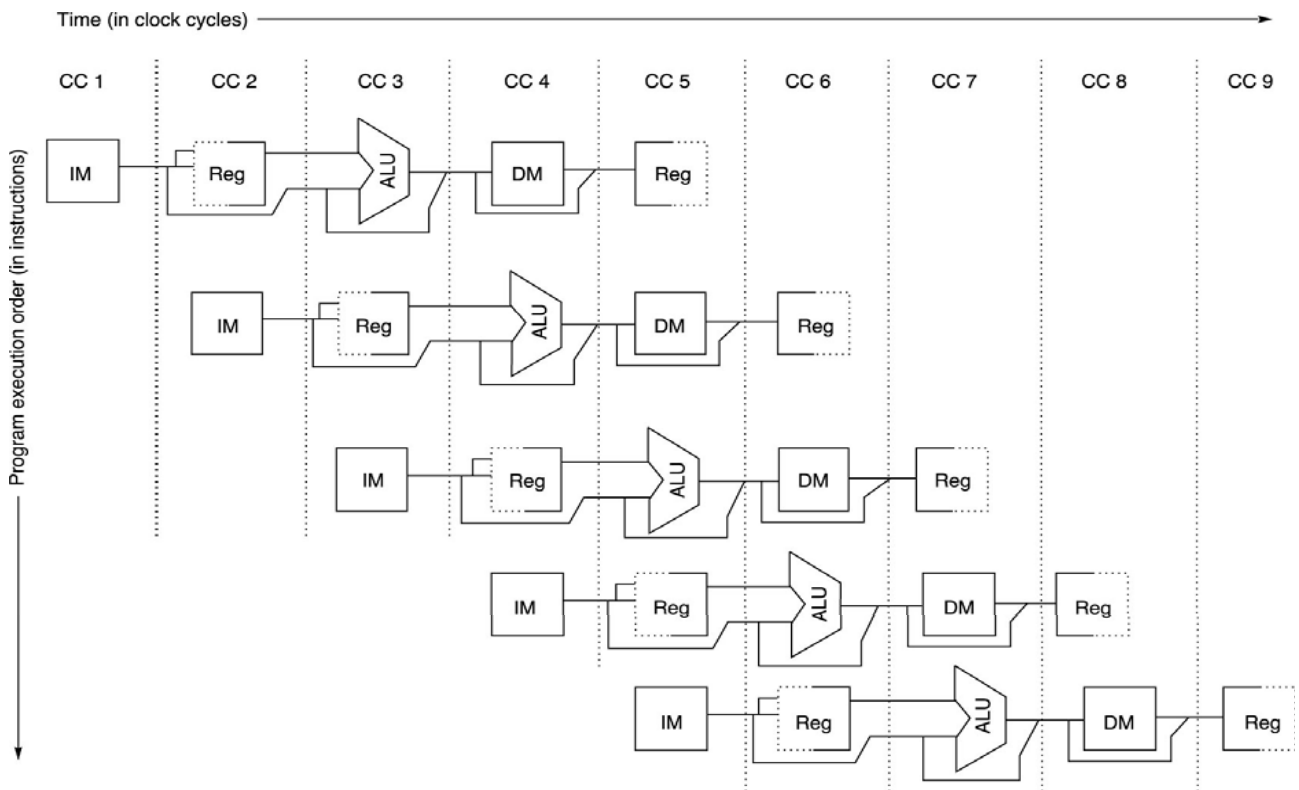
- Implementierung eines RISC-Befehlssatzes
  - Resultatsspeicherphase (Write-Back, Rückschreibphase)
    - Bei Register-Register-ALU und bei Lade-Befehlen:
      - Schreiben des Ergebnisses in das Zielregister, je nachdem ob es von der ALU oder vom Speichersystem kommt

- Implementierung eines RISC-Befehlssatzes
  - k-stufige Befehlspipeline (k=5)



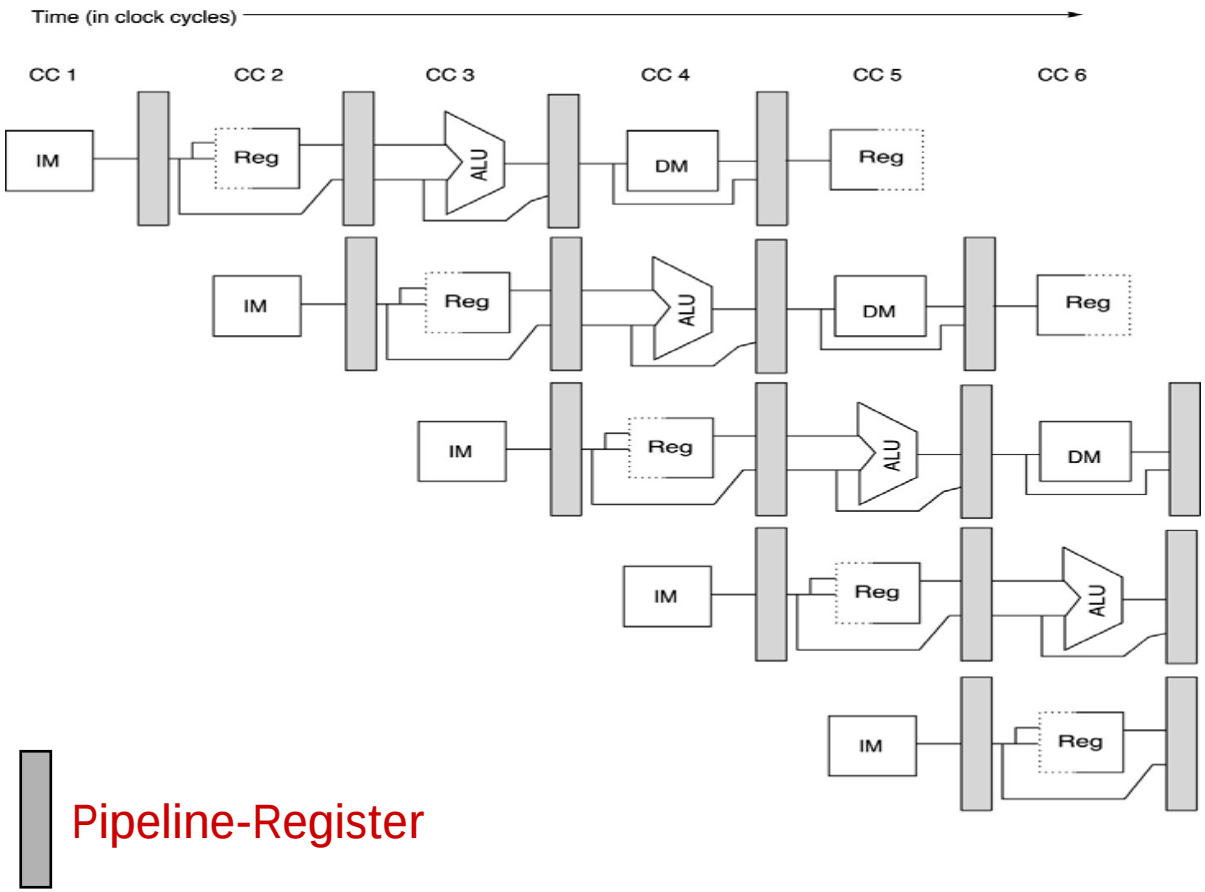
IF: Befehl holen
ID: Befehl dekodieren
EX: Befehl ausführen
MA: Speicherzugriff
WB: Zurückschreiben

- **k-stufige Befehlspipeline**



IM: Instruction Memory
DM: Data Memory

- **k-stufige Befehlspipeline**



- Leistungsaspekte
  - Ausführungszeit eines Befehls:
    - Zeit, die zum Durchlaufen der Pipeline benötigt wird
    - Ausführung eines Befehls in k Taktzyklen (ideale Verhältnisse)
    - Gleichzeitige Behandlung von k Befehlen (ideale Verhältnisse)
  - Latenz:
    - Anzahl der Zyklen zwischen einer Operation, die ein Ergebnis produziert, und einer Operation, die das Ergebnis verwendet

- Leistungsaspekte
  - Durchsatz T
    - Anzahl der Befehle, die eine Pipeline pro Takt verlassen können

$$T = n+k-1$$

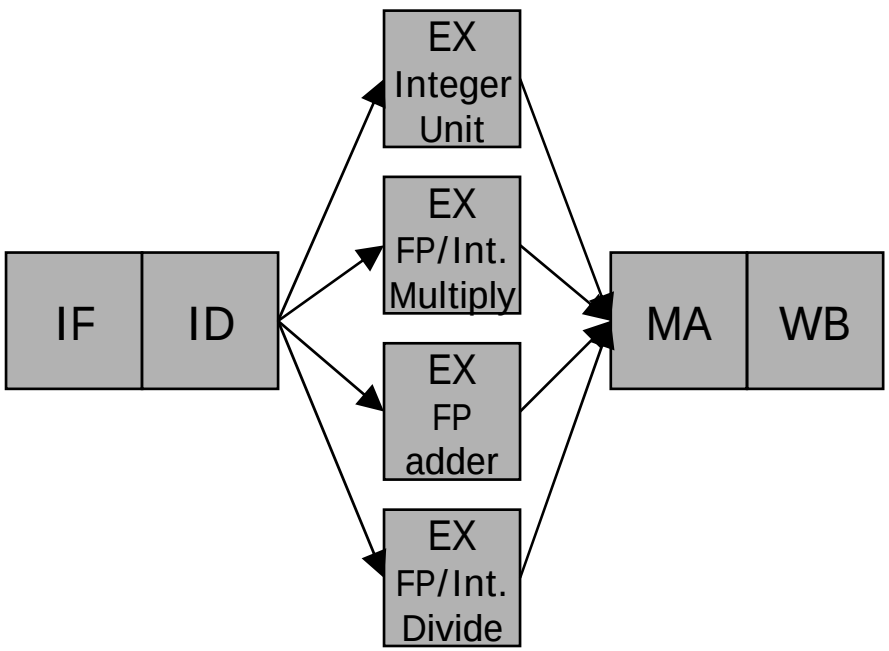
      - N: Anzahl der Befehle in einem Programm
      - Annahme: ideale Verhältnisse!
  - Beschleunigung S

$$S = n * k / (k + n - 1)$$

- Diskussion
  - Alle Pipelinestufen benützen unterschiedliche Ressourcen
  - Pipelining erhöht den Durchsatz
    - Mit jedem Takt wird unter Annahme idealer Verhältnisse ein Befehl geholt bzw. beendet.
    - Im eingeschwungenen Zustand der Pipeline:
      - Durchsatz = 1 Befehl / Taktzyklus
    - Aber, reduziert nicht die Ausführungszeit einer individuellen Instruktion
  - Zykluszeit, Taktzyklus:
    - Abhängig vom kritischen Pfad, der langsamsten Pipelinestufe

- Diskussion
  - Ausführungsphase
    - Integer-Verarbeitung
      - Ausführung von arithmetischen und logischen Befehlen dauert einen Taktzyklus (Ausnahme: Division)
    - Gleitkomma-Verarbeitung:
      - Zerlegung in weitere Stufen
      - Eingliederung an der Stelle der Ausführungsstufe in der Befehlspipeline
      - Mehrere Gleitkommarechenwerke (Floating-Point Units)

- Diskussion
  - Ausführungsphase
    - Gleitkomma-Verarbeitung: weitere Rechenwerke

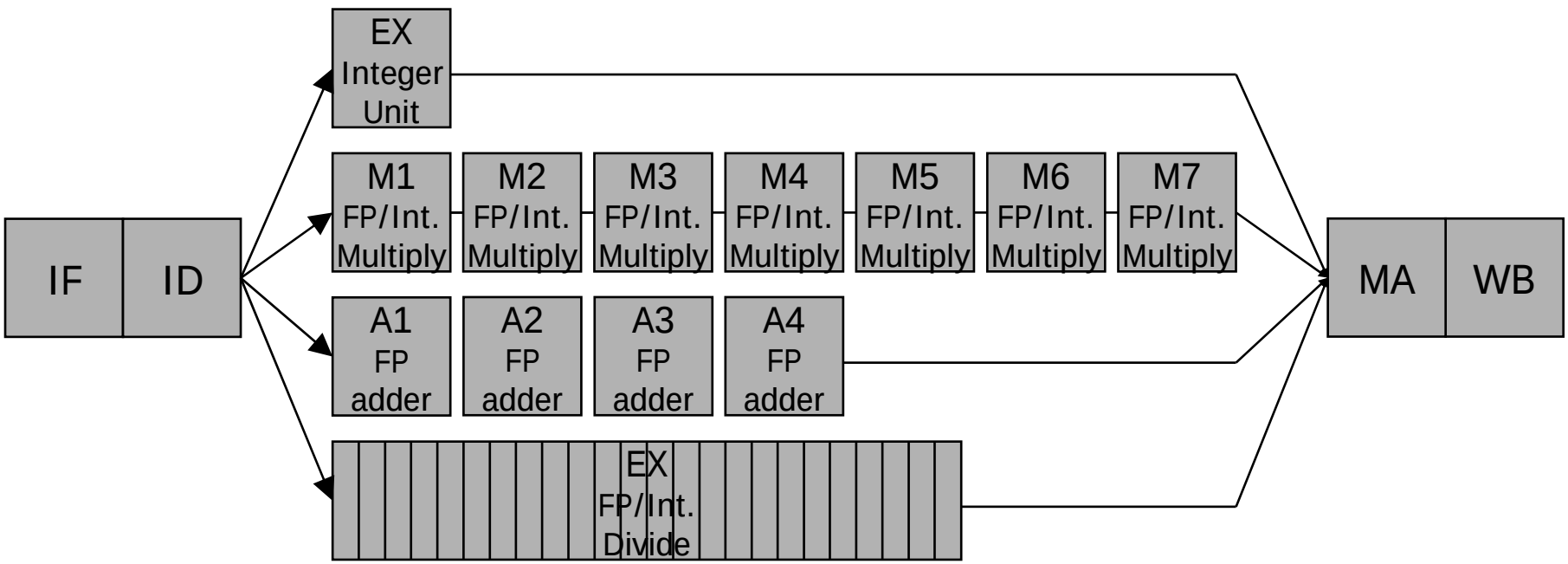


Rechenwerk	Latenz	Initiierungs-intervall
Integer ALU	0	1
FP Add	3	1
FP Multiply	6	1
FP Divide	24	25

Latenz: Anzahl der Zyklen zwischen einer Operation, die ein Ergebnis produziert und einer Operation, die das Ergebnis verwendet

Initiierungsintervall: Anzahl der Zyklen zwischen zwei Operationen

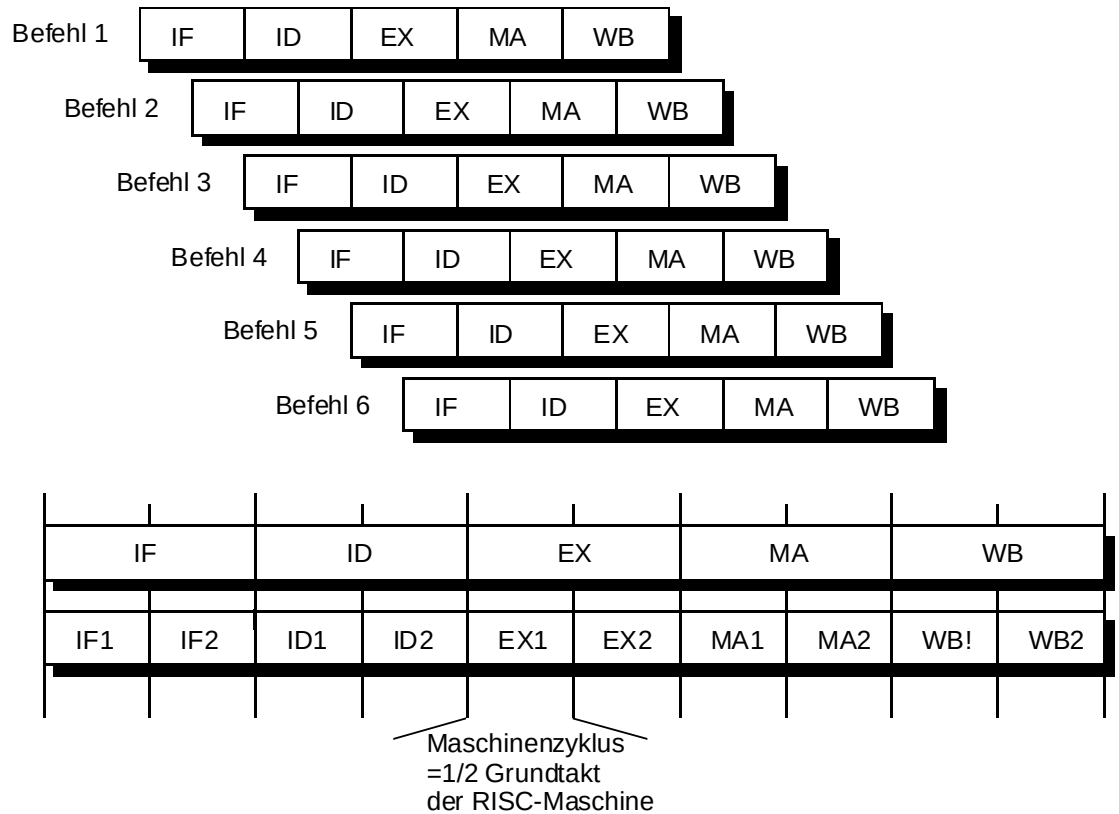
- Diskussion
  - Ausführungsphase
    - Gleitkomma-Verarbeitung: Pipelining der Rechenwerke



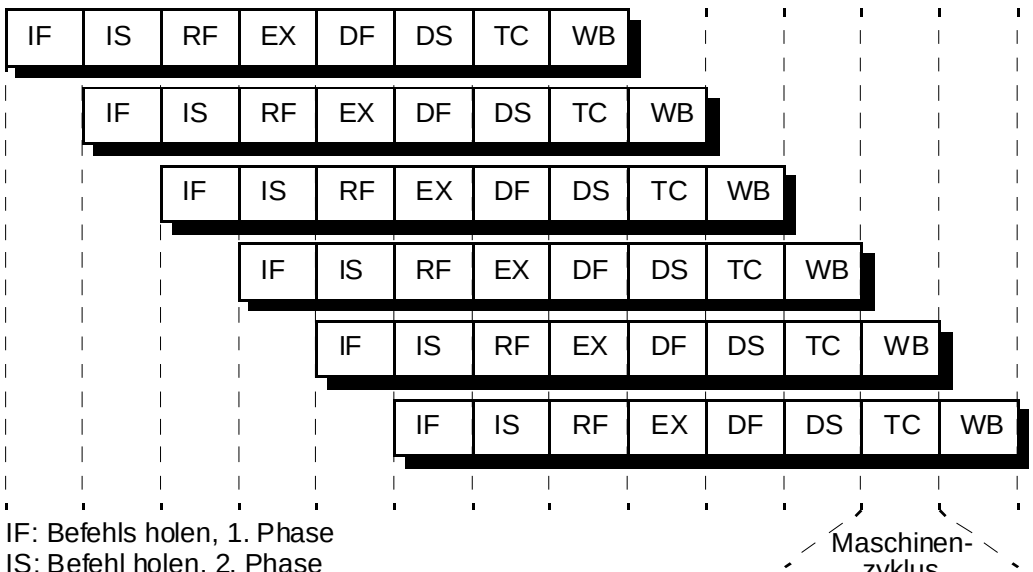
- Diskussion
  - Ausführungsphase
    - Gleitkomma-Verarbeitung: Pipelining der Rechenwerke
      - Latenz: 1 Zyklus weniger als die Anzahl der Pipelinestufen
      - Beispiel:
        - » 4 ausstehende FP add Operationen
        - » 7 ausstehende FP multiply Operationen
        - » 1 FP Divide Operation, da kein Pipelining

- Diskussion
  - Verfeinerung der Pipeline-Stufen
    - Weitere Unterteilung der Pipeline-Stufen
      - Weniger Logik-Ebenen pro Pipeline-Stufe
    - Erhöhung der Taktrate
      - Führt aber auch zu einer Erhöhung der Ausführungszeit pro Instruktion
  - „Superpipelining“

- Diskussion
  - Verfeinerung der Befehlspipeline (k=10)



- Diskussion
  - Verfeinerung der Befehlspipeline: Beispiel MIPS R4000 (~1991)



IF: Befehls holen, 1. Phase
IS: Befehl holen, 2. Phase
RF: Holen der Daten aus der Registerdatei
EX: Befehl ausführen
DF: Holen der Daten, 1.Zyklus (für Load- und Store-Befehle,
DS: Holen der Daten, 2. Zyklus
TC Tag-Check
WB: Ergebnis zurückschreiben

- Pipeline-Konflikte (Pipeline Hazards, Pipeline-Hemmnisse)
  - Situationen, die verhindern, dass die nächste Instruktion im Befehlsstrom im zugewiesenen Taktzyklus ausgeführt wird
    - Unterbrechung des taktsynchronen Durchlaufs durch die einzelnen Stufen der Pipeline
  - Verursachen Leistungseinbußen im Vergleich zum idealen Speedup
    - Erfordern ein Anhalten der Pipeline (Pipeline stall)
      - Bei einfacher Pipeline:
        - » Wenn eine Instruktion angehalten wird, werden auch alle Befehle, die nach dieser Instruktion zur Ausführung angestoßen wurden, angehalten
        - » Alle Befehle, die vor dieser Instruktion zur Ausführung angestoßen wurden, durchlaufen weiter die Pipeline

- Pipeline-Konflikte
  - Strukturkonflikte
    - Ergeben sich aus Ressourcenkonflikten
    - Die Hardware kann nicht alle möglichen Kombinationen von Befehlen unterstützen, die sich in der Pipeline befinden können
    - Beispiel:
      - Gleichzeitiger Schreibzugriff zweier Befehle auf eine Registerdatei mit nur einem Schreibeingang

- Pipeline-Konflikte
  - Datenkonflikte
    - Ergeben sich aus Datenabhängigkeiten zwischen Befehlen im Programm
    - Instruktion benötigt das Ergebnis einer vorangehenden und noch nicht abgeschlossenen Instruktion in der Pipeline
      - D.h. ein Operand ist noch nicht verfügbar
  - Steuerkonflikte
    - Treten bei Verzweigungsbefehlen und anderen Instruktionen auf, die den Befehlszähler verändern

- Pipeline-Konflikte
  - Auflösung der Pipeline-Konflikte
    - Einfache Lösung: Anhalten der Pipeline
    - Einfügen eines Leerzyklus (Pipeline Bubble)
      - Führt zu Leistungseinbußen
      - Verschiedene Maßnahmen in der Hardware und in der Software, um Auswirkungen auf die Leistungsfähigkeit möglichst zu vermeiden

- Pipeline-Konflikte
  - Ursachen für Datenkonflikte:
    - Datenabhängigkeiten
      - zwischen Befehlen im Programm
      - Beispiel:

```
add R1,R2,R3
sub R4,R5,R6
and R6,R1,R8
xor R9,R1,R11
```

- Pipeline-Konflikte
  - Ursachen für Datenkonflikte:
    - Datenabhängigkeiten
      - sind Eigenschaften des Programms!
      - Es hängt von der Pipeline-Organisation ab, ob eine gegebene Anhängigkeit zu einem Konflikt führt und ob Konflikte zu einem Anhalten der Pipeline führen!
      - in einem Programm zeigen die Möglichkeit eines Konflikts an!
      - Legen die Programmordnung fest, d.h. die Reihenfolge in der die Ergebnisse berechnet werden müssen.
      - Legen eine obere Grenze für den Grad des Parallelismus fest, der ausgenützt werden kann

- Pipeline-Konflikte
  - Ursachen für Datenkonflikte:
    - Echte Datenabhängigkeit (true dependence, flow dependence)
      - Ein Befehl *j* ist datenabhängig von einem Befehl *i*, wenn eine der folgenden Bedingungen gilt:
        - » Befehl *i* produziert ein Ergebnis, das von Befehl *j* verwendet wird, oder
        - » Befehl *j* ist datenabhängig von Befehl *k* und Befehl *k* ist datenabhängig von Befehl i (Abhängigkeitskette)

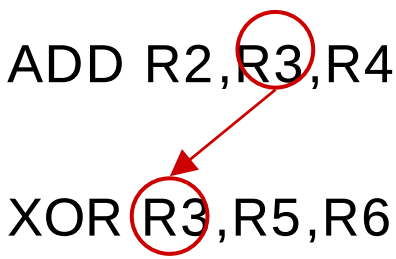
- Pipeline-Konflikte
  - Ursachen für Datenkonflikte:
    - Echte Datenabhängigkeit (true dependence, flow dependence)
      - Beispiel (MIPS Assembler):

```
– LOOP:   L.D      F0,0(R1)
          ADD.D    F4,F0,F2
          S.D      F4,0(R1)
          D.ADDUI  R1,R1,#-8
          BNE      R1,R2,LOOP
```

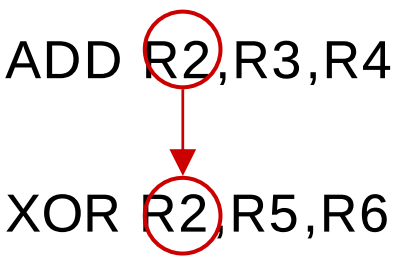
Abhängigkeit o tritt in Pipeline auf, in der der Vergleich in der ID Phase stattfindet

- Pipeline-Konflikte
  - Ursachen für Datenkonflikte:
    - Namensabhängigkeiten
      - Treten auf, wenn zwei Instruktionen dasselbe Register dieselbe Speicherzelle (den Namen) verwenden, aber kein Datenfluss zwischen den Befehlen mit dem Namen verbunden ist.
      - Es gibt zwei Arten von Namensabhängigkeiten zwischen zwei Befehlen *i* und *j* :
        - » Gegenabhängigkeit (Anti dependence)
        - » Ausgabeabhängigkeit (Output dependence)

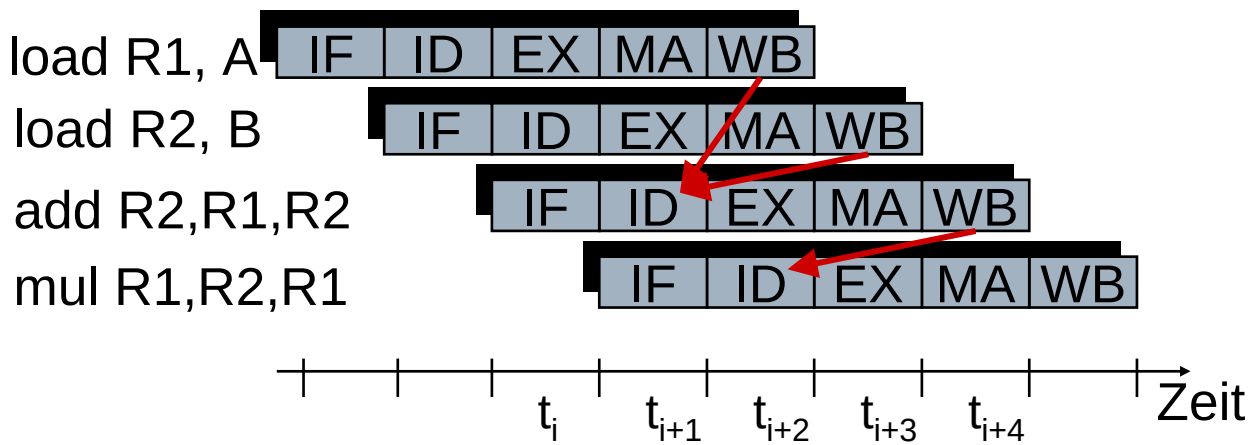
- Pipeline-Konflikte
  - Ursachen für Datenkonflikte:
    - Namensabhängigkeiten
      - Gegenabhängigkeit (Anti dependence)
        - » Der Befehl *i* liest einen Operanden aus einem Register, (Speicher), das von einem Befehl *j* anschließend überschrieben wird.



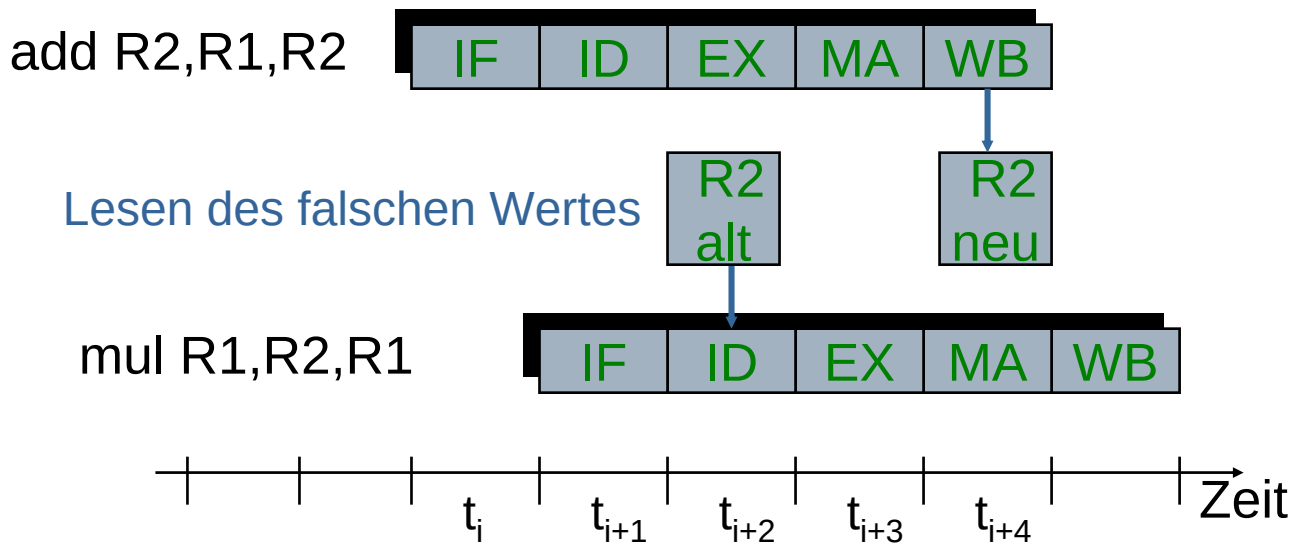
- Pipeline-Konflikte
  - Ursachen für Datenkonflikte:
    - Namensabhängigkeiten
      - Ausgabeabhängigkeit (Output dependence)
        - » Der Befehl *i* und der Befehl *j* schreiben in dasselbe Register oder in dieselbe Speicherzelle:



- Pipeline-Konflikte
  - Datenkonflikte:
    - Zwischen datenabhängigen Befehlen können Datenkonflikte auftreten, wenn sie so nahe im Programm stehen, dass ihre Überlappung innerhalb der Pipeline die Zugriffsreihenfolge auf die Register verändern würde.
      - » Beispiel: Echte Datenabhängigkeit



- Pipeline-Konflikte
  - Datenkonflikte:



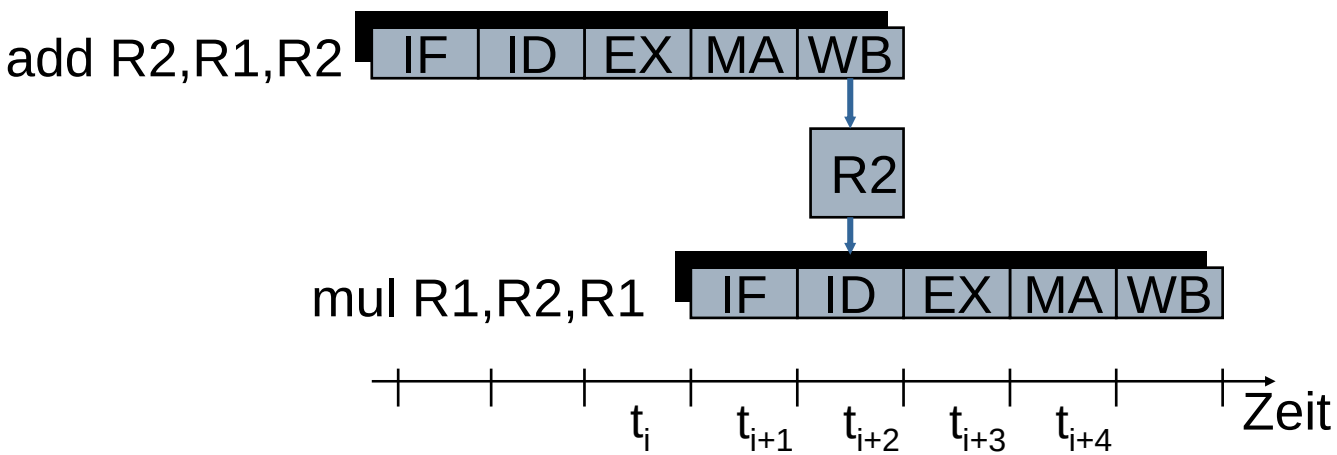
- Pipeline-Konflikte
  - Datenabhängigkeiten können folgende Konflikte verursachen:
    - Lese-nach-Schreib-Konflikt (Read-After-Write, RAW)
    - Lese-nach-Schreib-Konflikt (Write-After-Read, WAR)
    - Schreib-nachSchreib-Konflikt (Write-After-Write, WAW)

- Pipeline-Konflikte
  - Datenabhängigkeiten können folgende Konflikte verursachen:
    - Lese-nach-Schreib-Konflikt (Read-After-Write, RAW)
      - Tritt auf, wenn Befehl j sein Quellregister liest, bevor Befehl i das Ergebnis geschrieben hat.
    - Lese-nach-Schreib-Konflikt (Write-After-Read, WAR)
      - Tritt auf, wenn Befehl j sein Zielregister beschreibt, bevor Befehl i den Operanden gelesen hat.
      - D.h. der Befehl i liest einen falschen Wert
    - Schreib-nach-Schreib-Konflikt (Write-After-Write, WAW)
      - Tritt auf, wenn Befehl j sein Zielregister beschreibt, bevor Befehl i das Ergebnis geschrieben hat.
      - D.h. Der Befehl i liefert den Wert für das Zielregister, anstelle von j

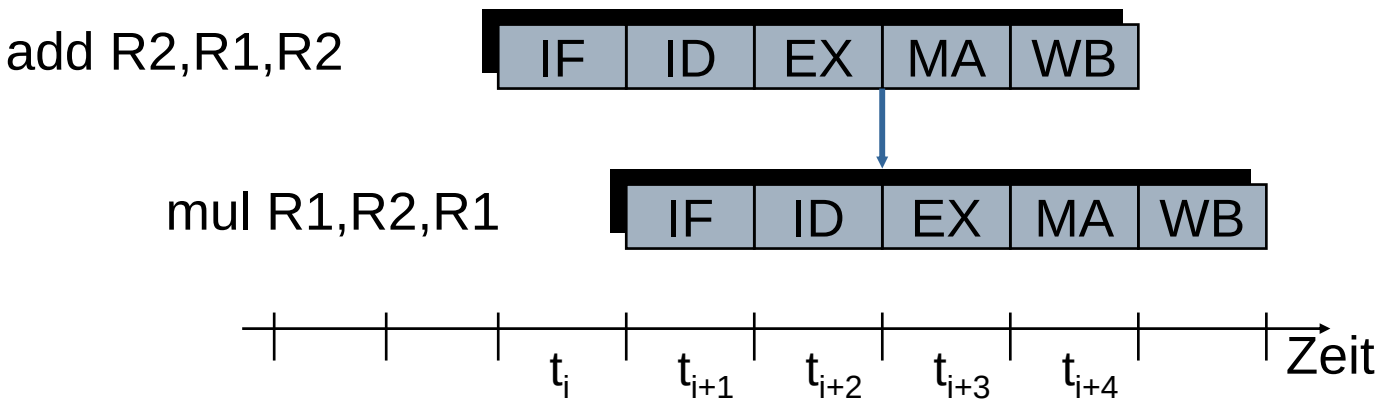
- Pipeline-Konflikte
  - Auflösen von Konflikten
    - Software-Lösungen
      - Aufgabe des Compilers:
        - » Erkennen von Datenkonflikten
        - » Einfügen von Leeroperationen nach jedem Befehl, der einen Konflikt verursacht oder verursachen kann.
    - Statische Verfahren:
      - Instruction Scheduling, Pipeline Scheduling
      - Eliminieren von Leeroperationen
      - Umordnen der Befehle des Programms (Code-Optimierung)

- Pipeline-Konflikte
  - Auflösen von Konflikten
    - Hardware-Lösungen (Dynamische Verfahren)
      - Erkennen von Konflikten
        - » Entsprechende Konflikterkennungslogik notwendig!
      - Techniken:
        - » Leerlauf der Pipeline (Interlocking, Stalling)
        - » Forwarding
        - » Forwarding mit Interlocking

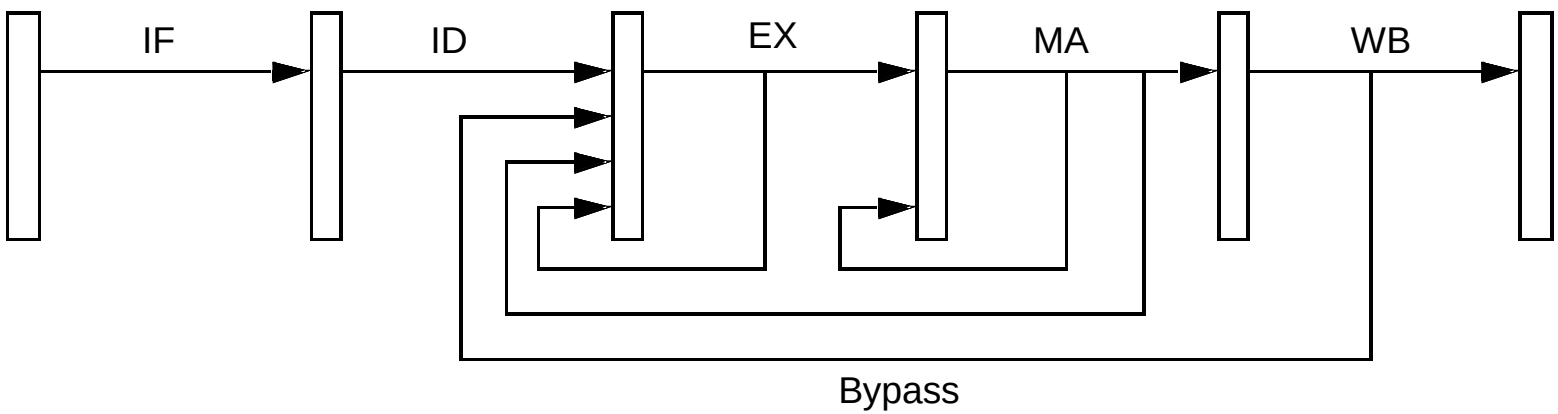
- Auflösen von Konflikten
  - Dynamische Verfahren
    - Anhalten der Pipeline (Pipeline Interlock)
      - Erkennen von Konflikten und Auflösen des Konflikts durch Anhalten der Pipeline
      - Anhalten des Befehls j und nachfolgender Befehle in der Pipeline für mehrere Takte



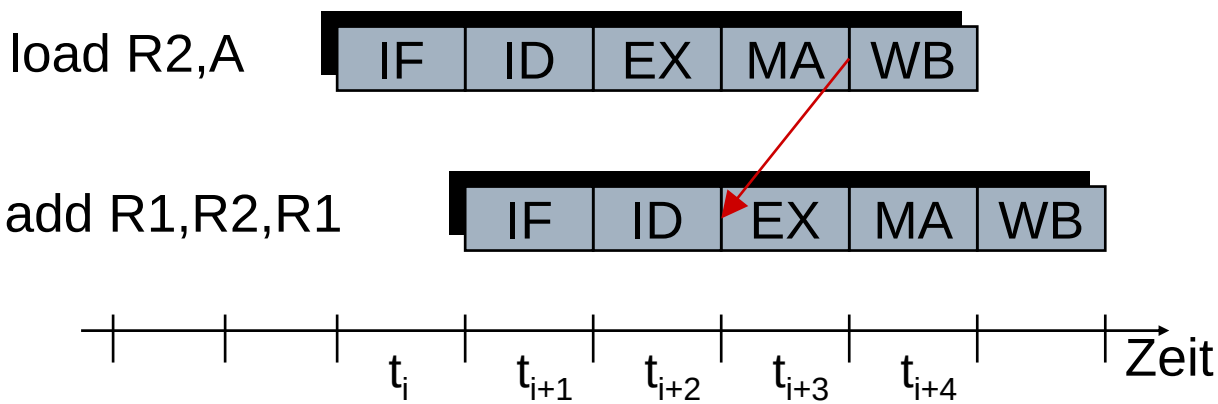
- Auflösen von Konflikten
  - Dynamische Verfahren
    - Forwarding
      - Erhöhter Hardware-Aufwand
      - Rückführung des ALU-Ergebnisses zur Eingabe
      - Kein Warten notwendig!



- Auflösen von Konflikten
  - Dynamische Verfahren
    - Forwarding
      - Zusätzlicher Hardware-Aufwand:
        - » Forwarding-Logik und zusätzliche Datenpfade



- Auflösen von Konflikten
  - Dynamische Verfahren
    - Forwarding mit Interlock (Result Forwarding)
      - Problem: Speicherzugriff, z.B Ladeoperation
        - » Nicht alle Konflikte lassen sich mit Forwarding allein auflösen



- Auflösen von Konflikten
  - Dynamische Verfahren
    - Forwarding mit Interlock (Result Forwarding)
      - Lösung: Anhalten der Pipeline