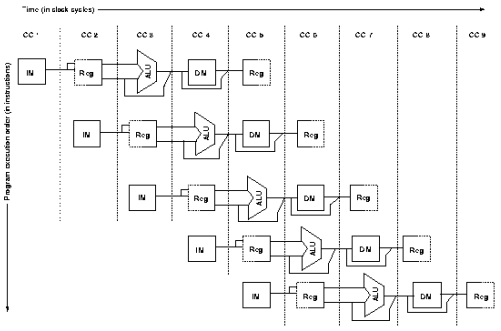
# 9. Fließbandverarbeitung




**Nachteile der Mikroprogrammierung**
**Von CISC zu RISC**
**Fließbandverarbeitung**
**Pipeline-Hazards**
**Behandlung von Daten- und Kontroll-Hazards**

# Nachteile mikroprogrammierter CPUs

**Zusätzliche Laufzeit durch LIE-Interpretierer**

**Unterschiedliche Taktzeiten verschiedener Befehle**

**Unterschiedliche Befehlsstruktur erschwert Parallelisierung**

**Taktzeiten einiger 8086 Maschinenbefehle :**

| | | |
|---|---|---|
| **NOP** | **3 Takte** | |
| **MUL BX** | **118-133 Takte** | |
| **STOSB** | **8 Takte** | |
| **REP STOSB** | **8 + (n-1)*4 Takte** | **)*** |

**Empirische Untersuchungen zeigten, dass komplexe Befehle in der Praxis relativ selten eingesetzt wurden.**

)* n = Anzahl der Iterationen



# Mikroprogrammierte CPUs

**leistungsfähige Maschinenbefehle**

**kurze ausführbare Programme**

**wenige, aber vielseitige Register**

**einfache Erweiterbarkeit durch zusätzliche Befehle**

**Aufwärtskompatibilität einfach zu gewährleisten**

**zusätzlicher Zeitbedarf durch Dekodier- und LIE-Zyklus**

**viele Befehle kaum genutzt**

**uneinheitliche Befehlsstruktur**

**schwierig zu parallelisieren**

# Ursachen für RISC

- Halbleiterspeicher ersetzen Kernspeicher
  - > Hauptspeicher werden schneller und größer
- Einführung von Cache-Speichern
  - > Beschleunigung der Speicherschnittstelle
- Compiler nutzen CISC-Befehlssätze nicht aus
  - 5-10% Befehle in 60-80% der Ausführungszeit
- virtuelle Speicherkonzepte erschweren Mikroprogrammierung
- Korrektheitsprobleme mit Microcode
- uneinheitliche Befehlsstruktur erschwert Parallelverarbeitung



# RISC-Generationen

- 1. Generation (1975-85): Pionierprojekte
  - RISC I & II, David Patterson et al., Univ. of Berkeley (1980-1985)
  - MIPS, John Hennessy et al., Univ. of Stanford (1981- 1985)
    -> Microprocessor without Interlocking Pipeline Stages
- 2. Generation (1985-90):
  - erste kommerzielle Weiterentwicklungen
  - Klassifizierung: Stanford- oder Berkeley-Ansatz
- 3. Generation (ab 1985):
  - superskalare RISC: CPI < 1

    *Verarbeitung mehrerer unabhängiger Instruktionen in einem Takt unter Einsatz separater Funktionseinheiten*
  - superpipelined RISC: CPI > 1

    *Minimierung der Taktzeit, Einsatz einer "tiefen" Pipeline*
  - Multiprozessor-RISC

CPI = Clock cycles Per Instruction

# RISC-Architekturmerkmale

| | *typische RISC-Werte* |
|---|---|
| • reduzierter, einfacher Instruktionssatz | *64 Instruktionen* |
| • Einzyklus-Maschinenbefehle (CPI = 1) | |
| • Maschinenbefehle fester Länge, wenige Formate | *1 Adressier-modus* |
| • Hardwaredekodierung | |
| • großer Registersatz | *32 Register* |
| • ``load/store´´-Architektur | |
| • Pipelining mit “delayed branching” | *>= 4 Pipeline-stufen* |
| • Cache-Speicher | |

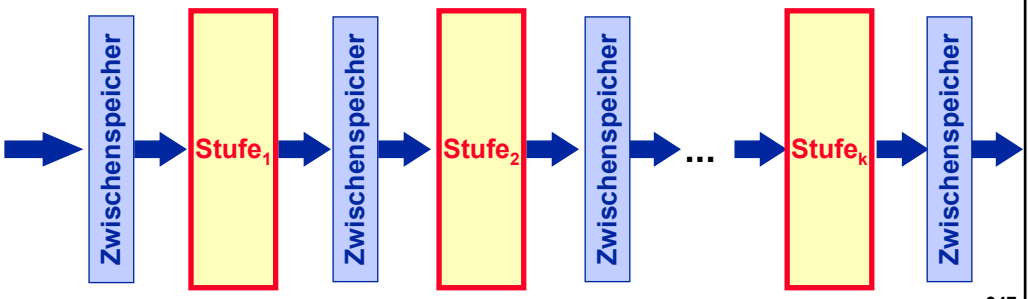


# Gegenüberstellung einiger RISC-Prozessoren

| | MIPS R3000 | HP PA (Precision Architecture) | Sparc (Scalable Processor Architecture) | IBM POWER (Performance Optimization with Enhanced Risc) | Pentium |
|---|---|---|---|---|---|
| Jahr | 1986 | 1986 | 1987 | 1990 | 1993 |
| Instruktionslänge | 32 | fest | 32 | fest | variabel |
| Befehlsanzahl | 64 | 64 | 64 | >= 128 | >= 256 |
| Adresslänge | 32 | 32 | 32 | 32 | 32 |
| Technologie | CMOS | NMOS | CMOS | CMOS | CMOS |
| Transistorzahl | 110.000 | 115.000 | 75.000 | 2.040.000 | 3.100.000 |
| Pipeline-Tiefe | 5 | 5 | 4 | 4 | 5 |
| Registerzahl | 32 | 32 | 40-520 | 32 | 8 |

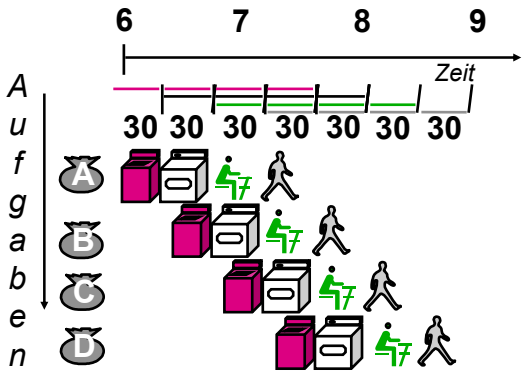
# Fließbandverarbeitung

- zentrales Merkmal von RISC-Prozessoren
- überlappende Ausführung mehrerer Instruktionen
- aufeinanderfolgende Phasen der Instruktionsabarbeitung werden in separaten Verarbeitungseinheiten durchgeführt.





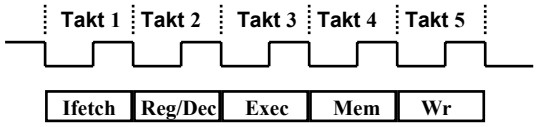

# Pipeline Lektionen (Wh aus Einführungskapitel)




- Fließbandverarbeitung verbessert nicht die Laufzeit einer einzelnen Aufgabe, sondern den Durchsatz der gesamten Arbeitslast.
- Mehrere Aufgaben werden mit unterschiedlichen Ressourcen simultan verarbeitet.
- Max. Beschleunigung = Anzahl der Verarbeitungsstufen
- Pipeline-Geschwindigkeit ist durch langsamste Stufe begrenzt.
- Unbalancierte Pipeline-Stufen reduzieren die Beschleunigung.
- Anlauf- und Leerlaufzeiten reduzieren die Beschleunigung.
- Leerlaufzyklen bei Abhängigkeiten

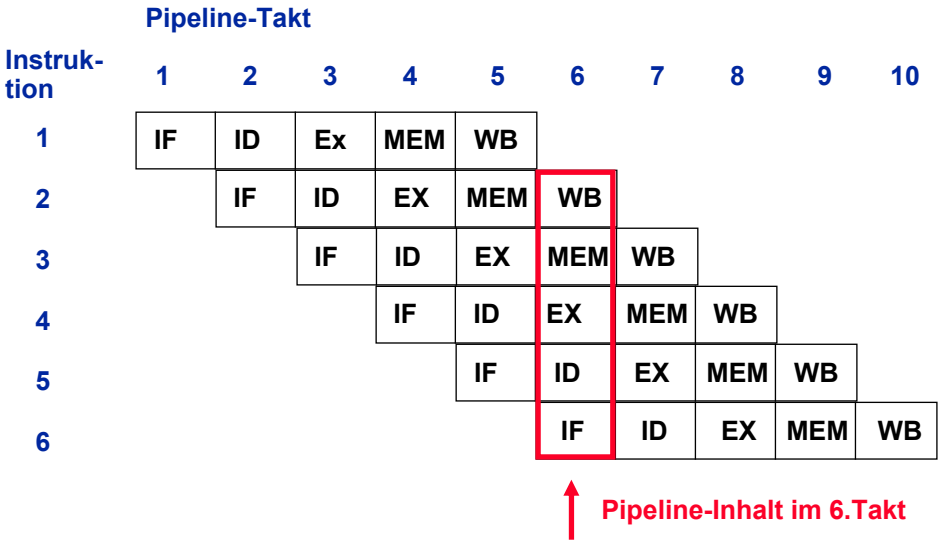
# Phasen der MIPS-Befehlsausführung

- **IF - Instruction Fetch, Befehlsbereitstellung**
  - Laden der Instruktion aus dem Speicher (Cache)
- **ID - Instruction Decode, Befehlsdekodierung**
  - Analyse der Instruktion, Laden der Operandenregister
- **EX - Execution, ALU-Operation**
  - Ausführung der Instruktion im Rechenwerk
- **MEM - Memory Access Speicherzugriff**
  - Lade- und Speichervorgänge
- **WB - Write Back, Zurückschreiben**
  - Rückschreiben des Ergebnisses in Zielregister

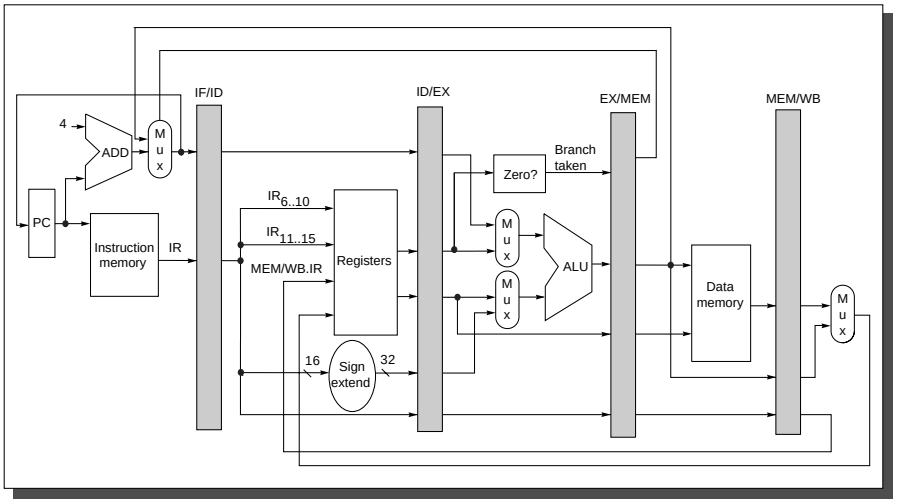





# MIPS-Pipeline

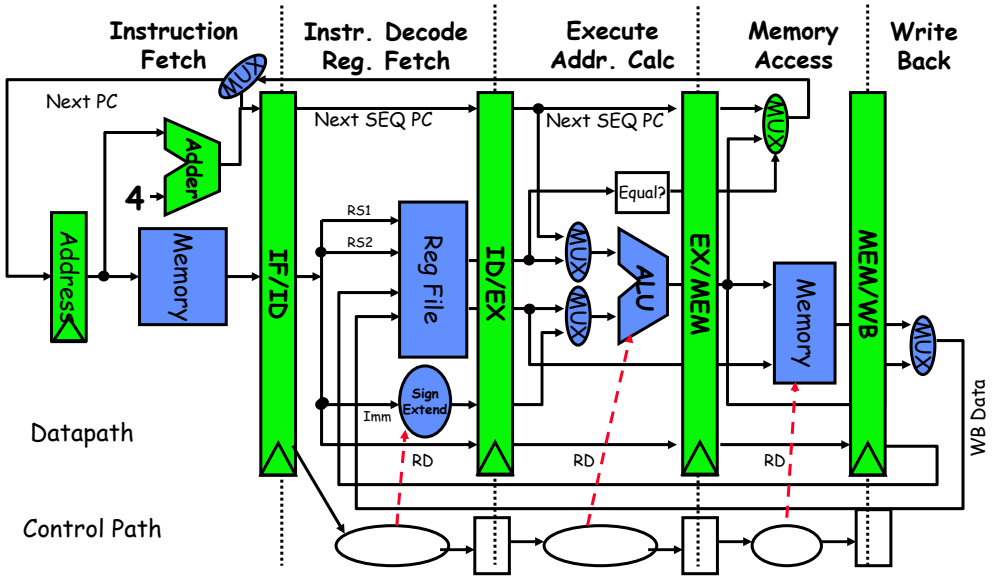


# MIPS-Pipeline-Datenpfad

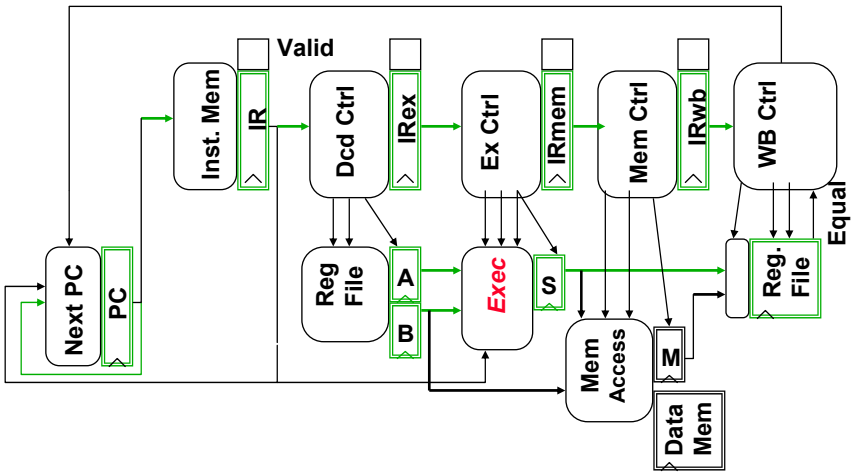





# 5 Steps of MIPS Datapath

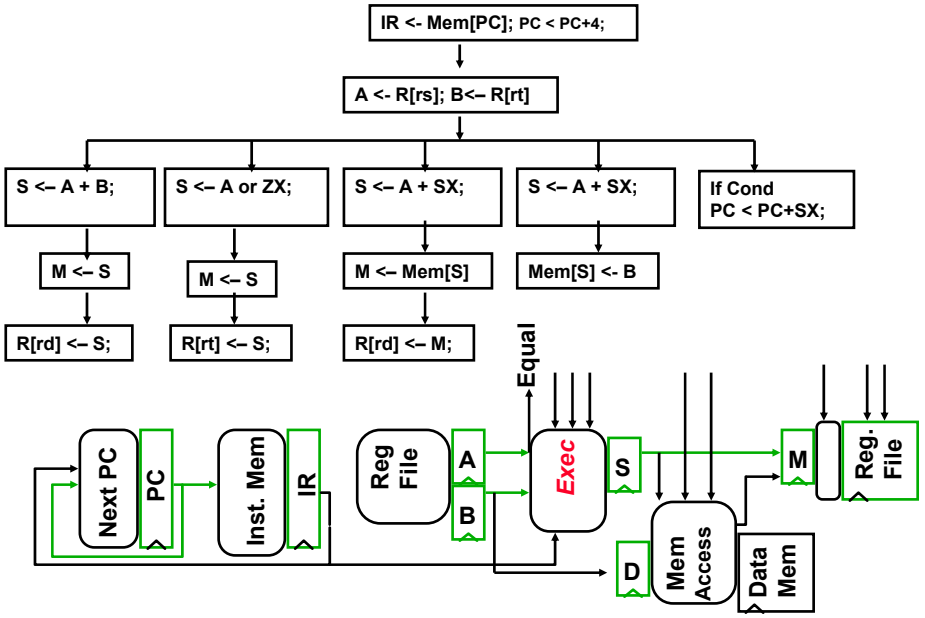


# Vereinfachte Sicht

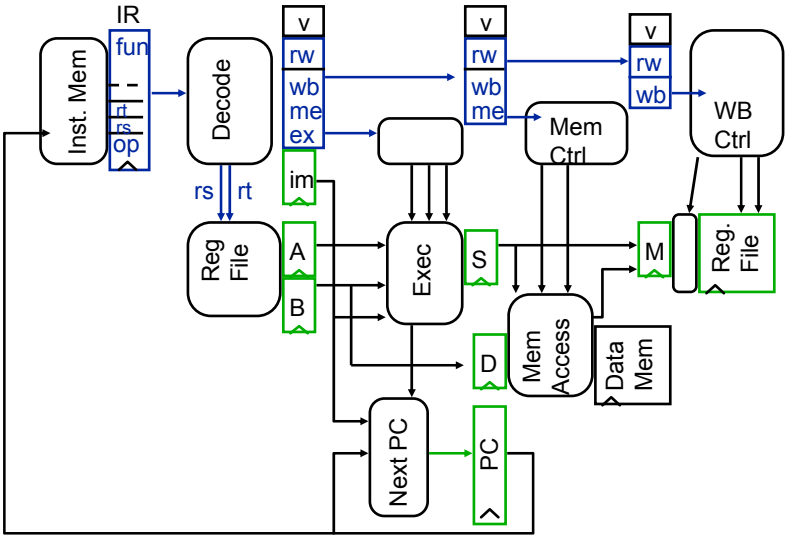





# Kontrolldiagramm



# Datenpfad und Kontrolle

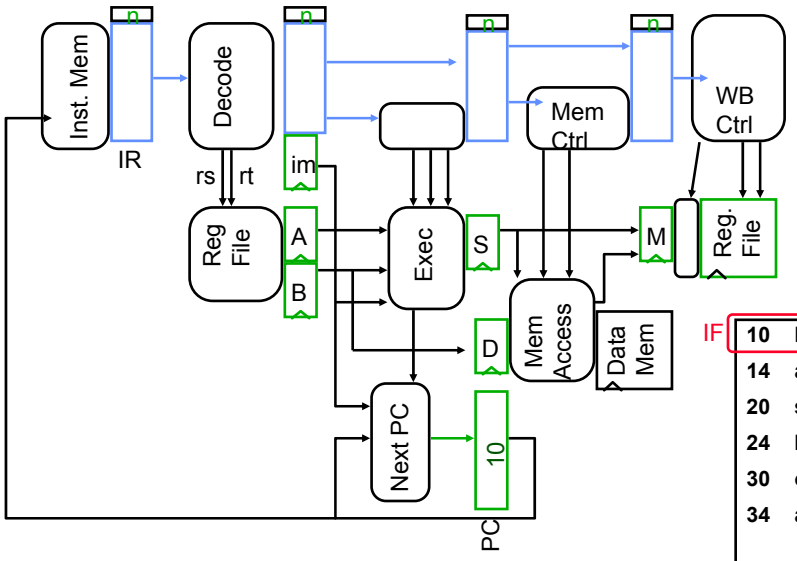





# Beispielverarbeitung

| Oktal-adressen | | |
|---|---|---|
| **10** | **lw** | **r1, 35(r2)** |
| **14** | **addi** | **r2, r2, 3** |
| **20** | **sub** | **r3, r4, r5** |
| **24** | **beq** | **r6, r7, 100** |
| **30** | **ori** | **r8, r9, 17** |
| **34** | **add** | **r10, r11, r12** |
| | | |
| **100** | **and** | **r13, r14, r15** |

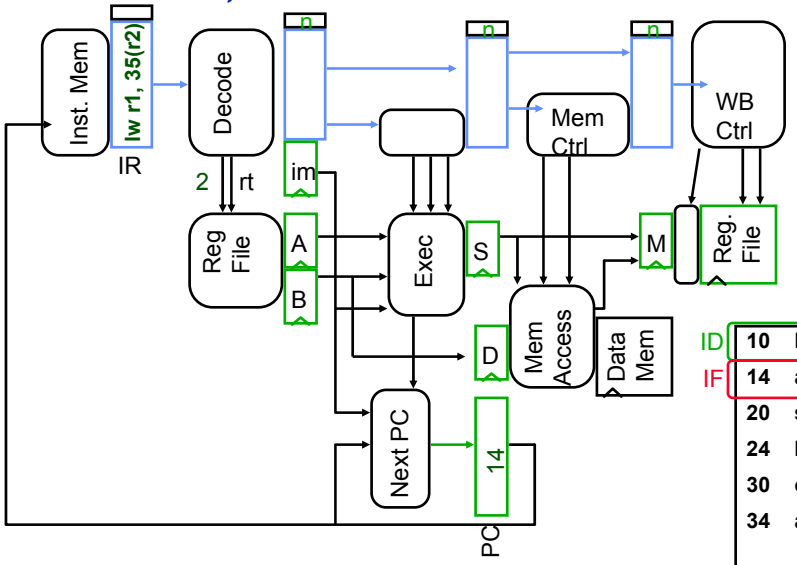
# Start: Fetch 10




| | | | |
|---|---|---|---|
| IF | 10 | lw | r1, 35(r2) |
| | 14 | addi | r2, r2, 3 |
| | 20 | sub | r3, r4, r5 |
| | 24 | beq | r6, r7, 100 |
| | 30 | ori | r8, r9, 17 |
| | 34 | add | r10, r11, r12 |
| | 100 | and | r13, r14, r15 |

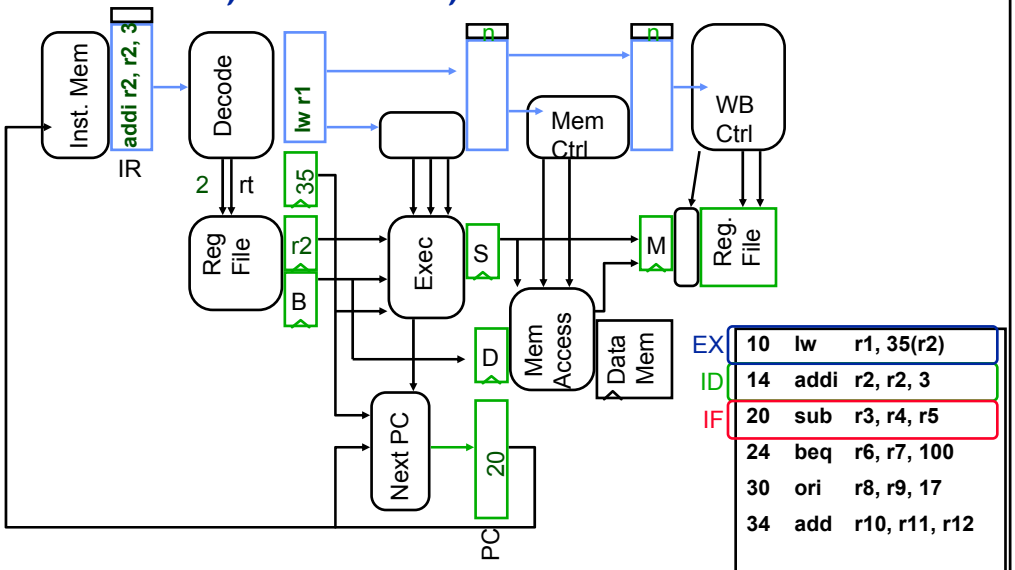


# Fetch 14, Decode 10



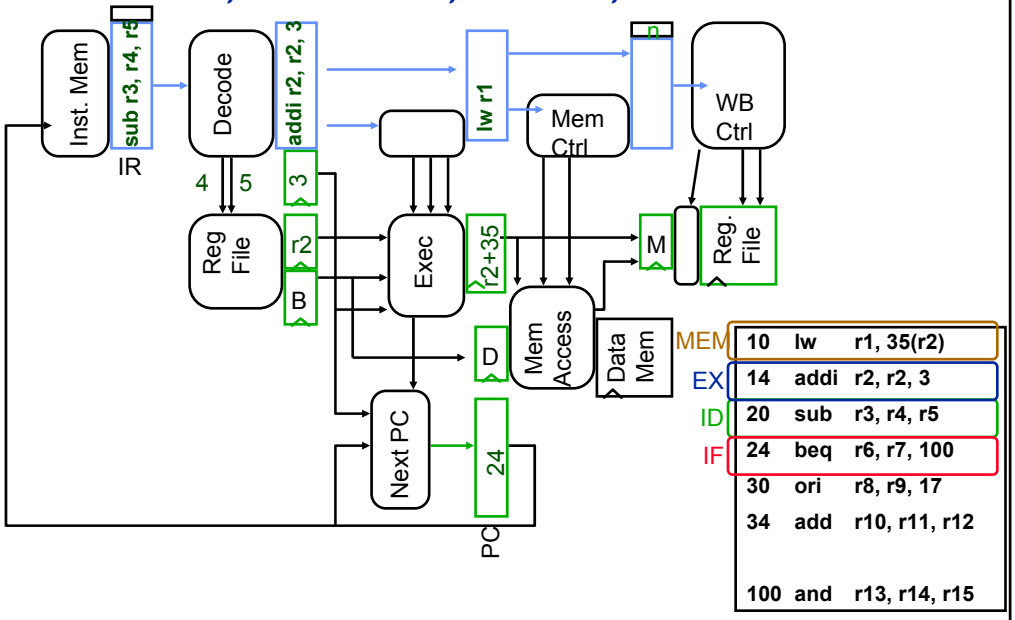

| | | | |
|---|---|---|---|
| ID | 10 | lw | r1, 35(r2) |
| IF | 14 | addi | r2, r2, 3 |
| | 20 | sub | r3, r4, r5 |
| | 24 | beq | r6, r7, 100 |
| | 30 | ori | r8, r9, 17 |
| | 34 | add | r10, r11, r12 |
| | 100 | and | r13, r14, r15 |

# Fetch 20, Decode 14, Exec 10

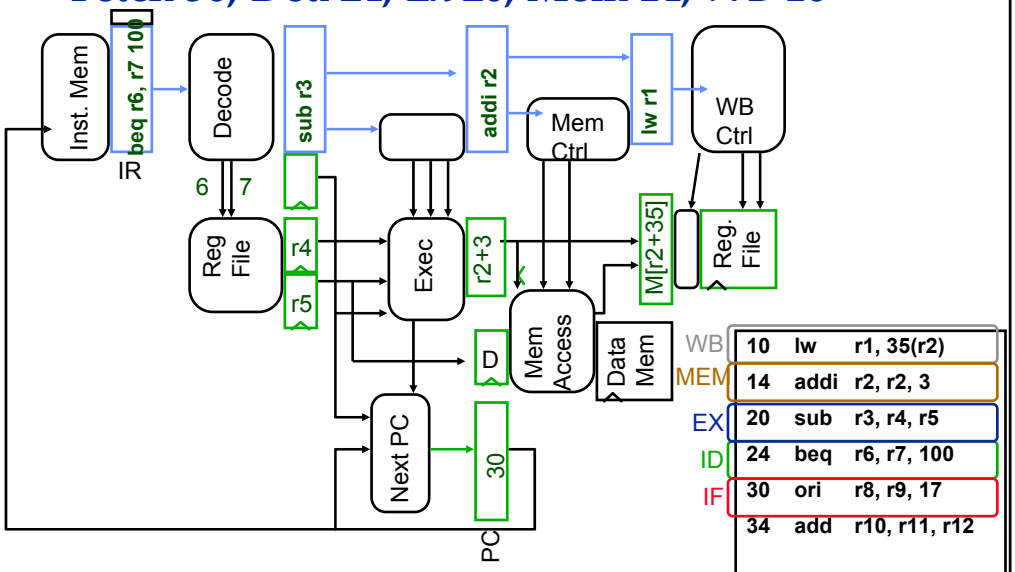





# Fetch 24, Decode 20, Exec 14, Mem 10

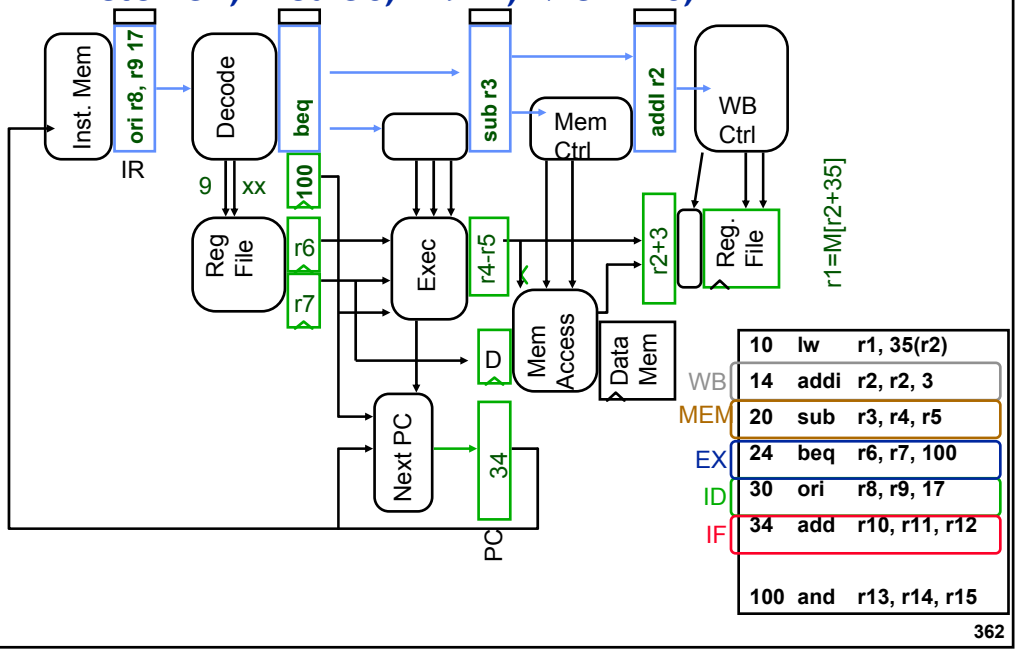


# Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10



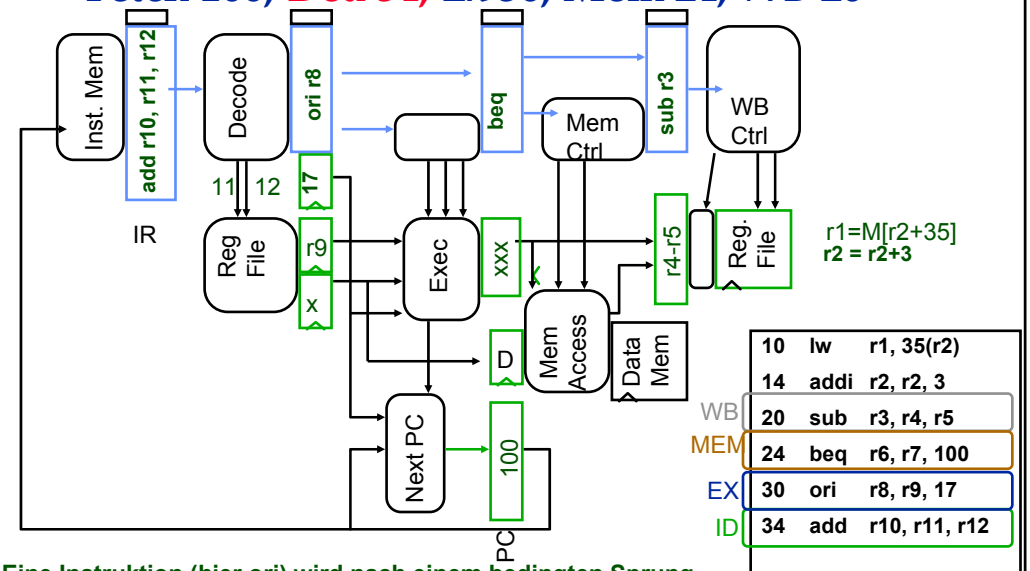

361

# Fetch 34, Dcd 30, Ex 24, Mem 20, WB 14




362

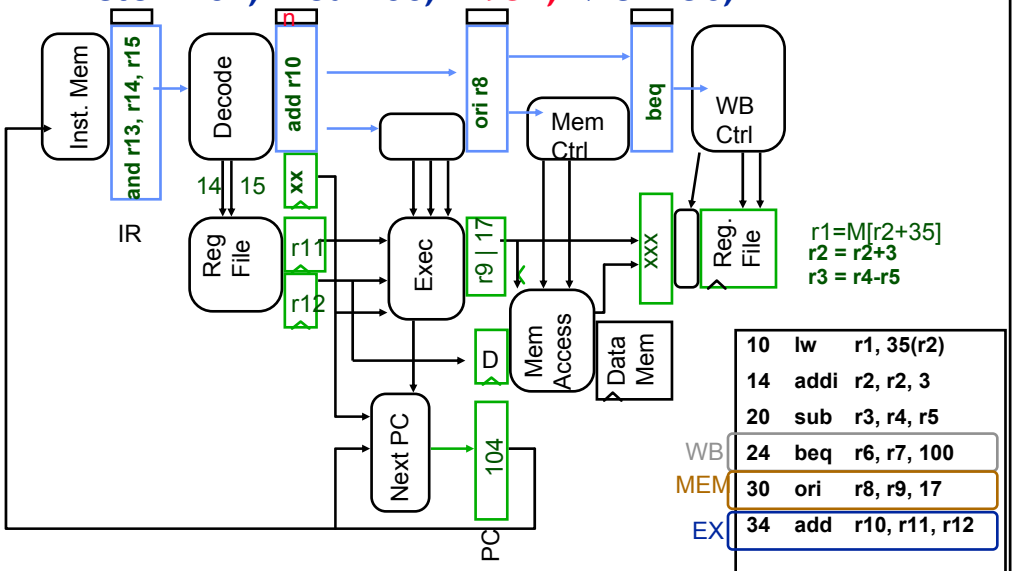
# Fetch 100, Dcd 34, Ex 30, Mem 24, WB 20



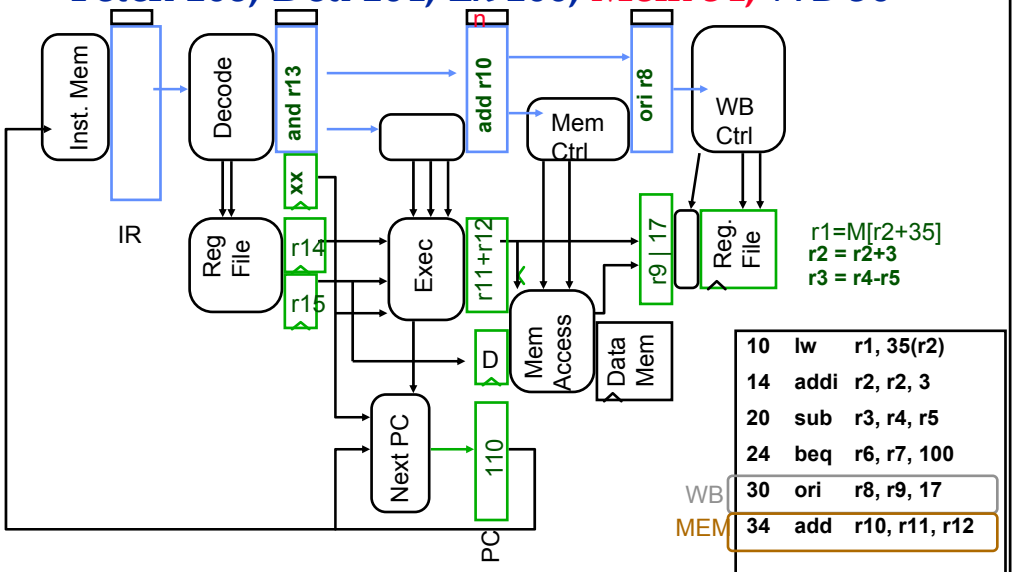

Eine Instruktion (hier ori) wird nach einem bedingten Sprung auf jeden Fall ausgeführt (-> delayed branching (siehe später)), die nachfolgende (add) wird als ungültig markiert.



# Fetch 104, Dcd 100, Ex 34, Mem 30, WB 24

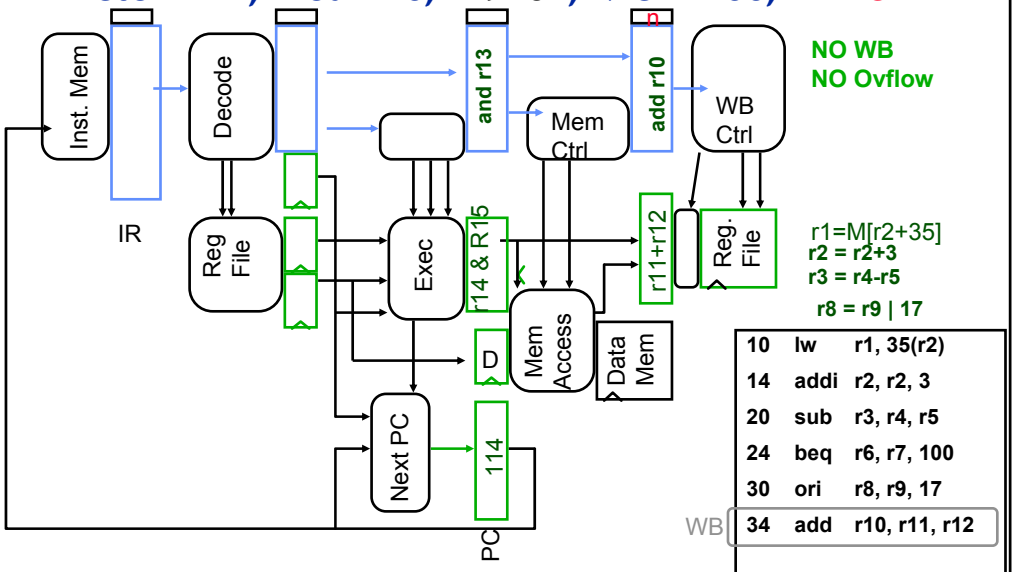


# Fetch 108, Dcd 104, Ex 100, Mem 34, WB 30






# Fetch 114, Dcd 110, Ex 104, Mem 100, WB 34



# Pipeline-Hazards

**Situationen, in denen die regelmäßige Abarbeitung in einer Pipeline unterbrochen werden muss**

- **Struktur-Hazards:**

  **aufeinanderfolgende Instruktionen benötigen dieselbe Hardware und können nicht überlappend ausgeführt werden.**

- **Daten-Hazards:**

  **Datenabhängigkeiten zwischen aufeinanderfolgenden Instruktionen**

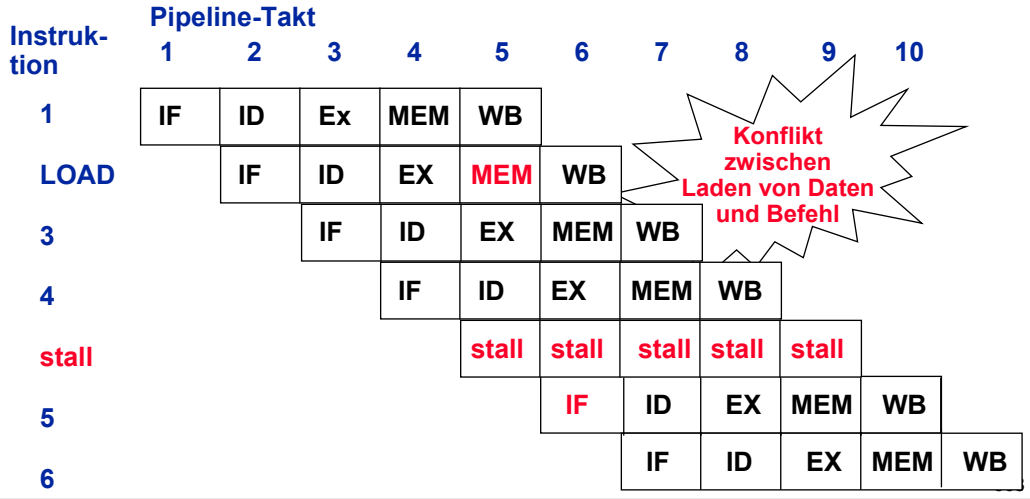
- **Kontroll-Hazards:**

  **Änderung der sequentiellen Abarbeitung von Befehlsfolgen durch bedingte Sprünge**

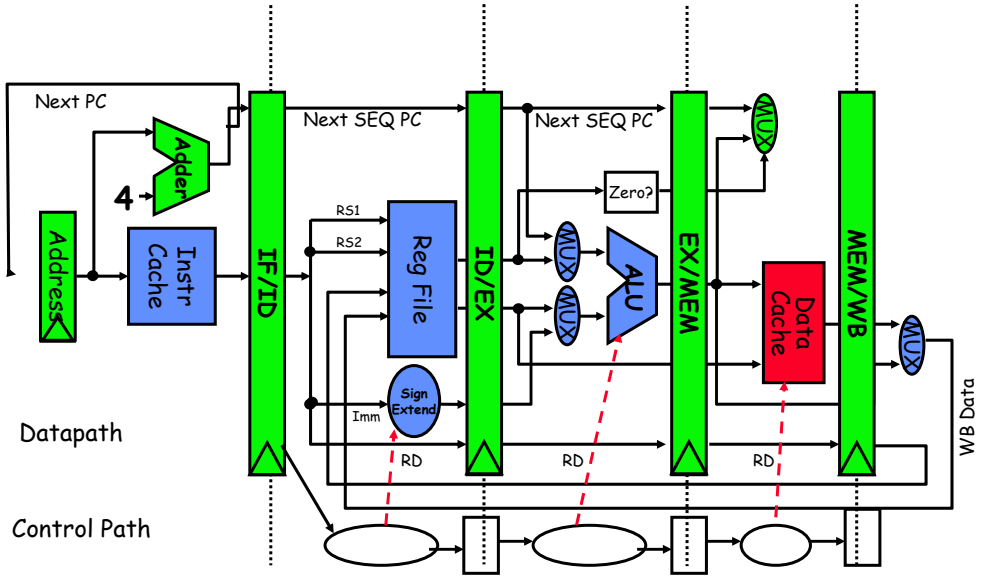


# Struktur-Hazards

**Beispiel: Bei nur einem Speicherport zum Holen von Befehlen und Daten führt ein Speicherzugriffsbefehl zu einem Leertakt in der Pipeline:**



# Eliminating Structural Hazards at Design Time

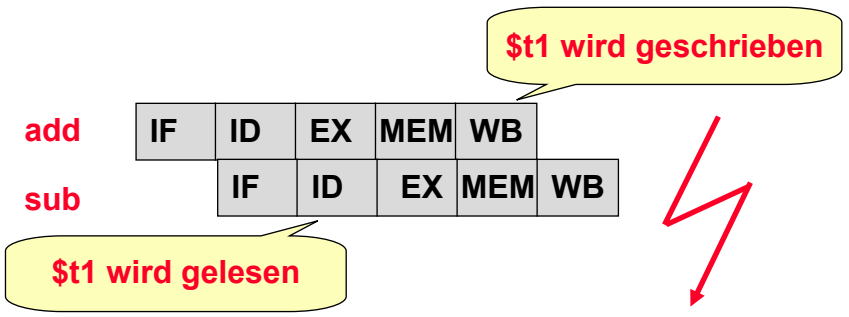





# Beispiel: Daten-Hazard

**Betrachte die Instruktionsfolge**

```
add  $t1, $t2, $t3   (* $t1 := $t2 + $t3 *)
sub  $t4, $t1, $t5   (* $t4 := $t1 - $t5 *)
```



# Daten-Hazards

- **RAW (read after write)**

  **Befehl i+j versucht Datum zu lesen, bevor Befehl i dieses geschrieben hat**

- **WAR (write after read)**

  **Befehl i+j versucht Datum zu schreiben, bevor es von Befehl i gelesen wurde**

  **-> in MIPS-Pipeline nicht möglich, da frühes Lesen in ID und spätes Schreiben in WB**

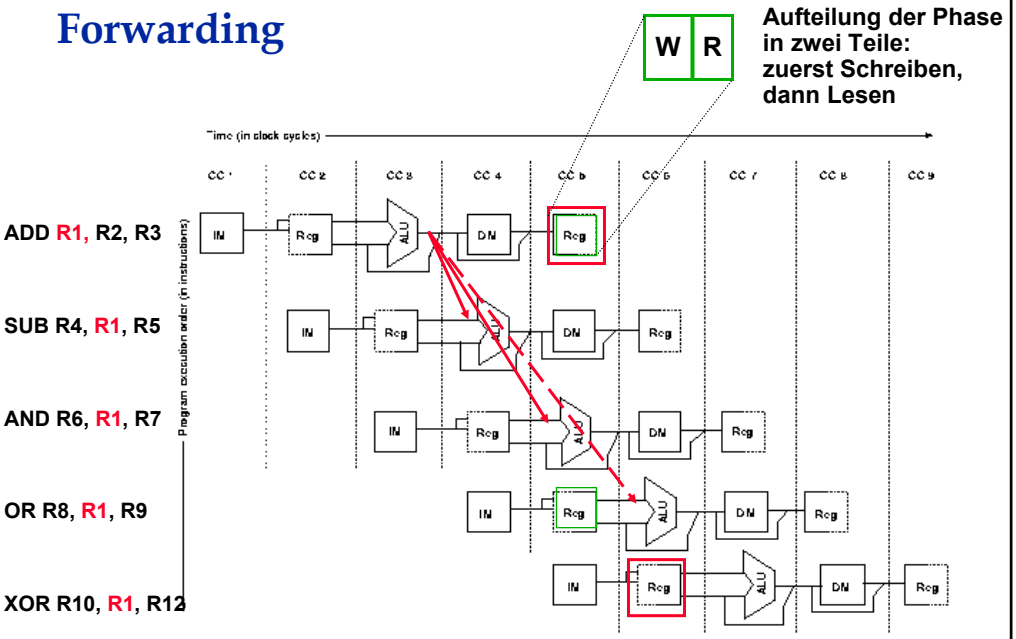
- **WAW (write after write)**

  **Befehl i+j versucht Datum zu schreiben, bevor es durch Befehl i geschrieben wurde**

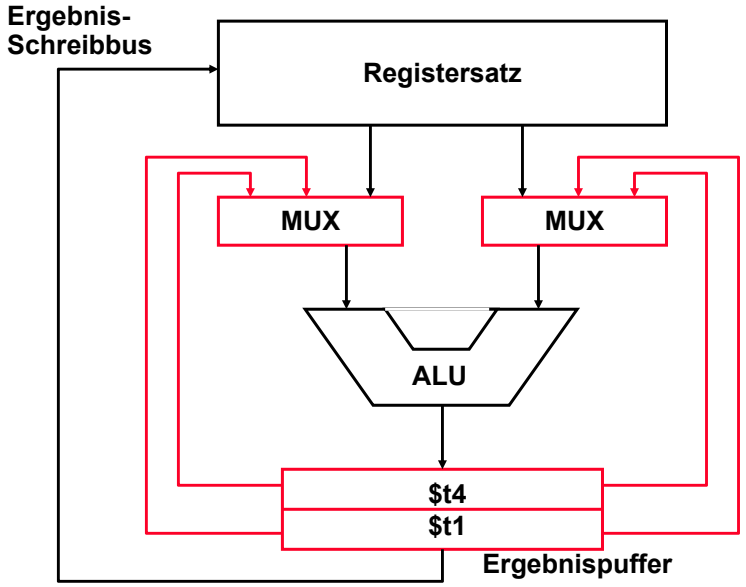
  **-> in MIPS-Pipeline nicht möglich, da nur in Phase WB geschrieben wird**



# Forwarding

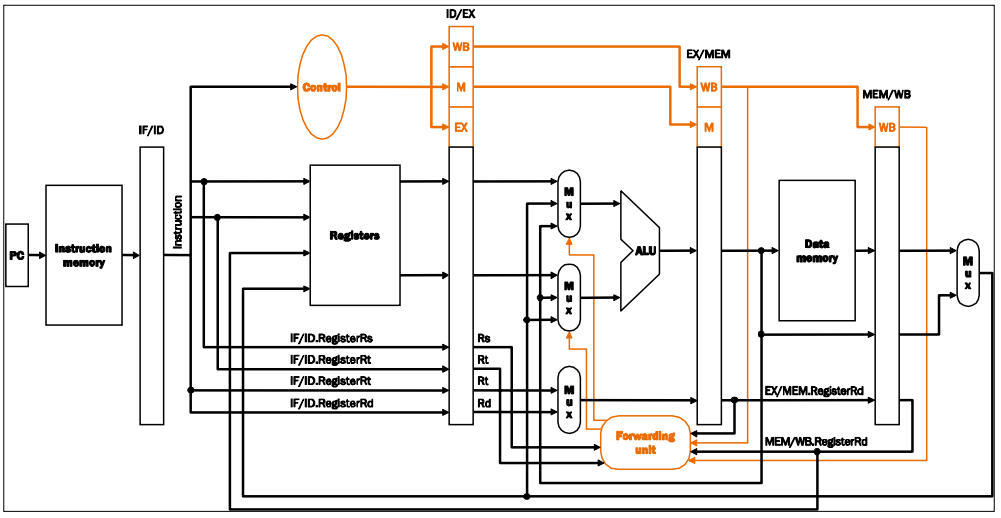


# ALU mit Forwarding-Einheit

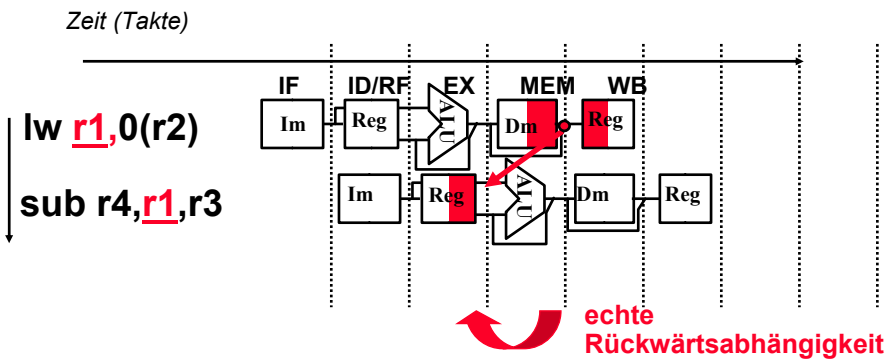





# Forwarding in MIPS-Pipeline



# Unvermeidbare Leerzyklen (stalls)




- **Forwarding nicht möglich!**
- **Von load abhängige Instruktionen müssen verzögert werden.**



# Compiler Scheduling

**Durch geschicktes Umordnen von Instruktionen können Daten-Hazards oft vermieden werden.**

**Bsp: A = B + C wird durch folgende Instuktionssequenz realisiert:**

```
lw $t1, B
lw $t2, C
add $t0, $t1, $t2
sw $t0, A
```

**Diese enthält einen nicht durch forwarding lösbaren Datenhazard.**

**Aber: A = B+C; D = E-F kann durch folgende hazardfreie Sequenz implementiert werden:**

```
lw $t1, B
lw $t2, C
lw $t4, E
add $t0, $t1, $t2
lw $t5, F
sw $t0, A
sub $t3, $t4, $t5
sw $t3, D
```

# Kontroll-Hazards

**Bedingte Sprungbefehle können zu erheblichen Effizienzverlusten führen, weil im Normalfall erst am Ende der Bearbeitung des Befehls entschieden wird, ob ein Sprung erfolgt oder nicht.**

**=> Pipeline-Verzögerung (engl. stall) von 3 Takten**

| Instruktion \ Pipeline-Takt | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| bed. Sprung | IF | ID | EX | MEM | WB | | | | | |
| 2 | | IF | stall | stall | IF | ID | EX | MEM | WB | |
| 3 | | | stall | stall | stall | IF | ID | EX | MEM | WB |
| 4 | | | | stall | stall | stall | IF | ID | EX | MEM | WB |
| 5 | | | | | stall | stall | stall | IF | ID | EX | MEM |
| 6 | | | | | | | | | | |

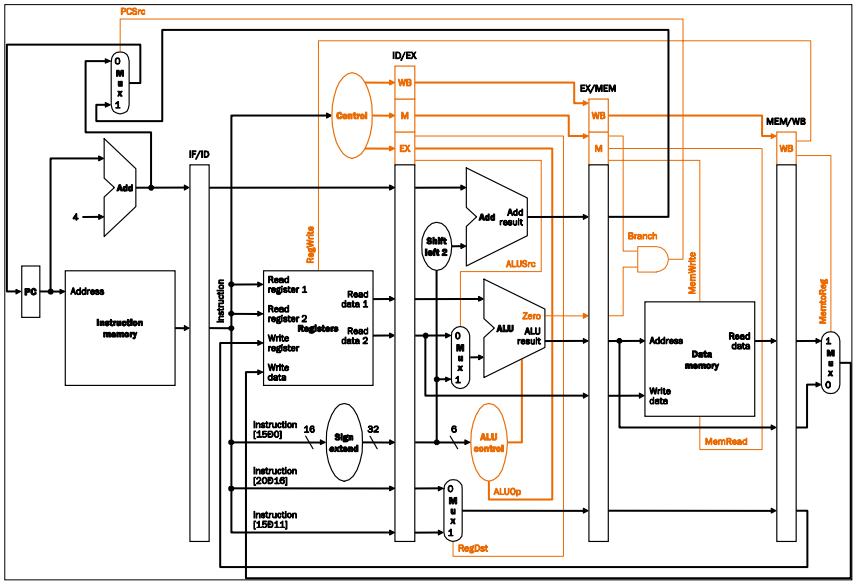
Erst wenn das Sprungziel bekannt ist, kann der nächste Befehl geladen werden.

**Abhilfe:**

- früheres Herausfinden, ob Sprung erfolgt
- früheres Berechnen der Sprungadresse



# MIPS-Pipeline mit Kontrolle



# Maßnahmen bei Kontrollhazards

- **Vorverlagerung der Bedingungsauswertung in ID-Phase**
  - Die Berechnung des Verzweigungsziels kann bereits in der ID-Phase beginnen. Es sind die zu vergleichenden Register bekannt. Der Vergleich kann durch Spezialhardware (EXOR und NAND) sehr schnell erfolgen.
  - Der korrekte Ziel-PC-Wert kann durch einen zusätzlichen Addierer aus der Konstante im Instruktionswort und dem aktuellen PC-Wert auch bereits in der ID-Phase berechnet werden.

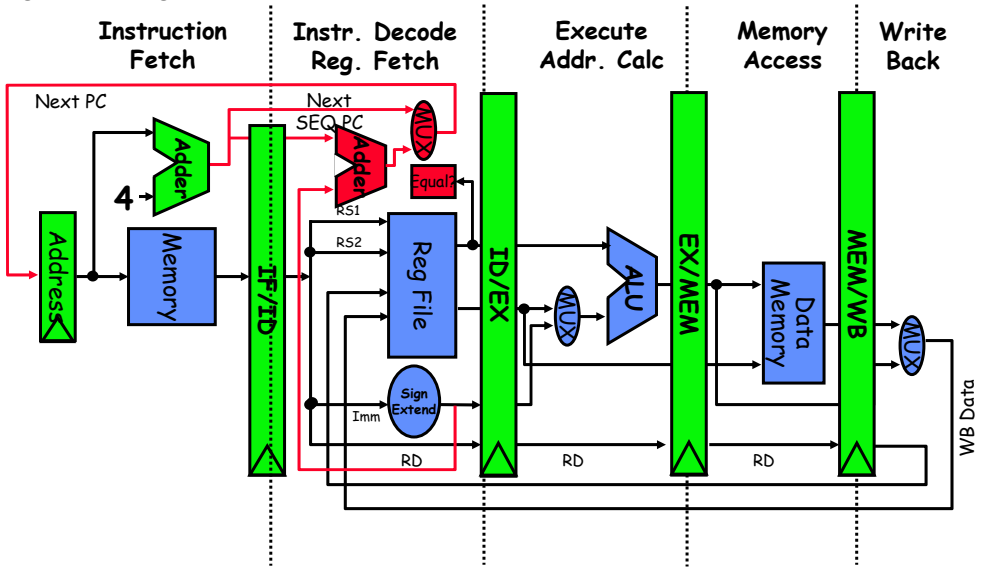
  => Somit ist nur noch ein Stall-Schritt erforderlich.

- **Predict Branch Not Taken**
  - Beginnen der Ausführung der Instruktionen, die dem Branch unmittelbar folgen
  - Falls sich herausstellt, dass tatsächlich eine Verzweigung erfolgen soll, müssen die begonnenen Instruktionen annulliert werden.



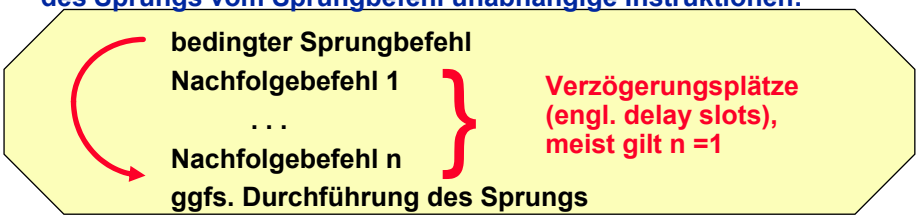
# Pipelined MIPS Datapath

Figure 3.22, page 163, CA:AQA 2/e



# ``Delayed Branching´´

**Idee:** Schiebe zwischen Berechnung des Sprungziels und Ausführung des Sprungs vom Sprungbefehl unabhängige Instruktionen:

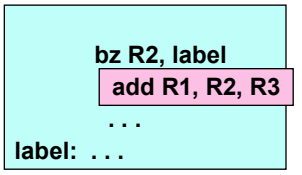



Die korrekte Verwendung verzögerter Sprünge liegt in der Verantwortung des Compiler-Entwicklers oder Assembler-programmierers.

**Beispiel:**

Transformiere

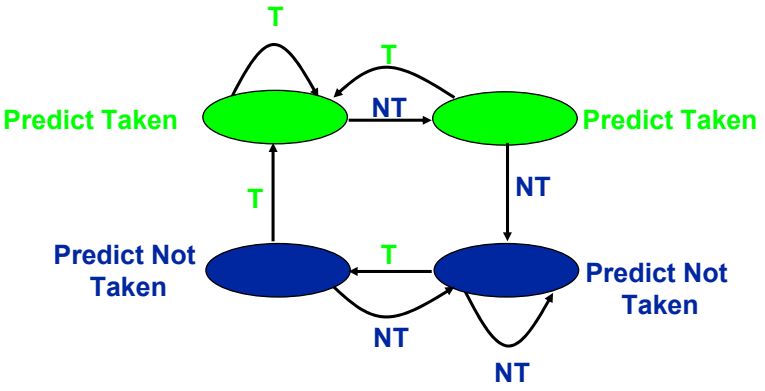

zu






# Dynamische Sprungvorhersage

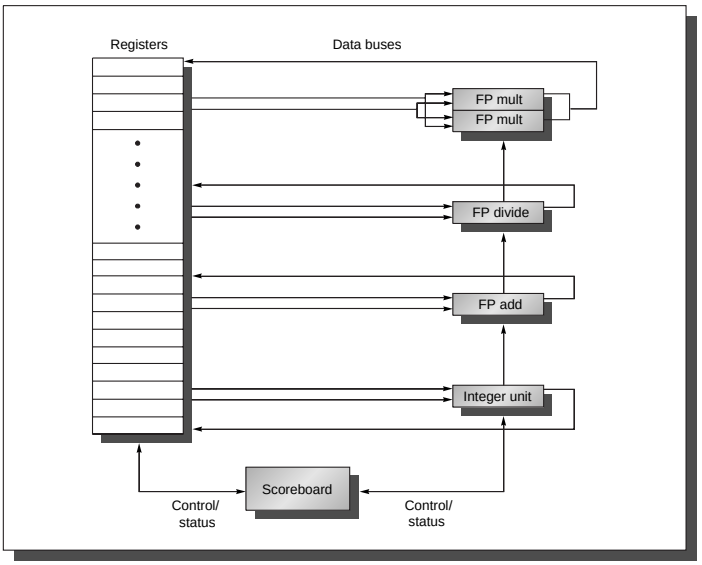
**2-Bit-Schema, das die Vorhersage bei zwei Fehlern ändert**




**Branch Target Buffer (BTB)** enthält Adresse des Sprungindexes, um Vorhersage zu ermitteln und die Sprungadresse,

# Dynamische Instruktionsanordnung

- mehrere Funktionseinheiten => mehrere Instruktionen können gleichzeitig bearbeitet werden (multiple issue)
- „out-of-order“ Ausführung von Instruktionen
- Verwaltung bspsweise über sogenanntes Scoreboard
- IF wird zerlegt in
  - Issue: Dekodiere, Test auf Strukturhazards
  - Operand Fetch: Test auf Datenhazards






# Beispiel für Scoreboard

**Pipelinephasen:**

- **Issue**: Instruktionsstart, falls
  - kein **Struktur-hazard**
  - kein **WAW**-Hazard
- **Read Operands**:
  - Test auf **RAW** Hazards
- **Execution**
- **Write Result**
  - Test auf **WAR**-Hazards

| | Instruction status | | | |
|---|---|---|---|---|
| Instruction | Issue | Read operands | Execution complete | Write result |
| LD F6,34(R2) | √ | √ | √ | √ |
| LD F2,45(R3) | √ | √ | √ | |
| MULTD F0,F2,F4 | √ | | | |
| SUBD F8,F6,F2 | √ | | | |
| DIVD F10,F0,F6 | √ | | | |
| ADDD F6,F8,F2 | | | | |

Ziel Operanden Quellen Operand bereits konsumiert?

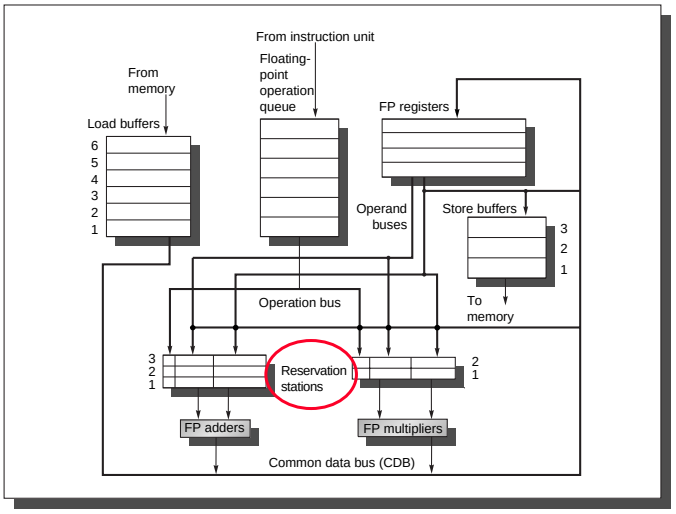
| | Functional unit status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Busy | Op | Fi | Fj | Fk | Qj | Qk | Rj | Rk |
| Integer | Yes | Load | F2 | R3 | | | | No | |
| Mult1 | Yes | Mult | F0 | F2 | F4 | Integer | | No | Yes |
| Mult2 | No | | | | | | | | |
| Add | Yes | Sub | F8 | F6 | F2 | | Integer | Yes | No |
| Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

Welche Einheit wird schreiben?

| | Register result status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
| FU | Mult1 | Integer | | | Sub | Divide | | | |

# Dynamische Registerumbenennung (nach Tomasulo)

- Einführung von virtuellen Registern (reservation stations) in jeder Funktionseinheit zur Speicherung von Operanden
- Resultate werden über den gemeinsamen Datenbus (CDB) verbreitet (Broadcast)
- Loads und Stores als funktionale Einheiten
- Vermeidung von WAR- und WAW-Hazards






# Verarbeitungsstufen

- Issue
  - Instruktion wartet, bis Platz in Zwischenpuffern frei ist (-> Register-umbenennung)
- Execute
  - Warte, bis alle Operanden verfügbar
  - Überwache CDB, um fehlende Operanden zu lesen
- Write Result
  - Resultat wird in CDB geschrieben und von dort überall gelesen, wo es benötigt wird

| | | Instruction status | | |
|---|---|---|---|---|
| Instruction | | Issue | Execute | Write result |
| LD | F6,34(R2) | √ | √ | √ |
| LD | F2,45(R3) | √ | √ | |
| MULTD | F0,F2,F4 | √ | | |
| SUBD | F8,F6,F2 | √ | | |
| DIVD | F10,F0,F6 | √ | | |
| ADDD | F6,F8,F2 | √ | | |

| | Reservation stations | | | | | |
|---|---|---|---|---|---|---|
| Name | Busy | Op | Vj | Vk | Qj | Qk |
| Add1 | Yes | SUB | Mem[34+Regs[R2]] | | | Load2 |
| Add2 | Yes | ADD | | | Add1 | Load2 |
| Add3 | No | | | | | |
| Mult1 | Yes | MULT | | Regs[F4] | Load2 | |
| Mult2 | Yes | DIV | | Mem[34+Regs[R2]] | Mult1 | |

| | Register status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Field | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
| Qi | Mult1 | Load2 | | Add2 | Add1 | Mult2 | | | |