



# 64-040 Modul InfB-RS: Rechnerstrukturen

[https://tams.informatik.uni-hamburg.de/  
lectures/2016ws/vorlesung/rs](https://tams.informatik.uni-hamburg.de/lectures/2016ws/vorlesung/rs)

– Kapitel 16 –

Andreas Mäder



Universität Hamburg  
Fakultät für Mathematik, Informatik und Naturwissenschaften  
Fachbereich Informatik  
**Technische Aspekte Multimodaler Systeme**

Wintersemester 2016/2017



## Pipelining

Motivation und Konzept

Befehlspipeline

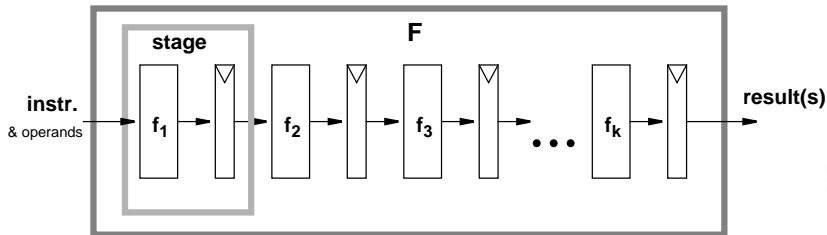
MIPS

Bewertung

Hazards

Literatur





## Grundidee

- ▶ Operation  $F$  kann in Teilschritte zerlegt werden
- ▶ jeder Teilschritt  $f_i$  braucht ähnlich viel Zeit
- ▶ Teilschritte  $f_1..f_k$  können parallel zueinander ausgeführt werden
- ▶ Trennung der Pipelinestufen („stage“) durch Register
- ▶ Zeitbedarf für Teilschritt  $f_i \gg$  Zugriffszeit auf Register ( $t_{FF}$ )

## Pipelining-Konzept

- ▶ Prozess in unabhängige Abschnitte aufteilen
- ▶ Objekt sequenziell durch diese Abschnitte laufen lassen
  - ▶ zu jedem Zeitpunkt werden zahlreiche Objekte bearbeitet
  - ▶ –"– sind alle Stationen ausgelastet

## Konsequenz

- ▶ Pipelining lässt Vorgänge gleichzeitig ablaufen
- ▶ reale Beispiele: Autowaschanlagen, Fließbänder in Fabriken

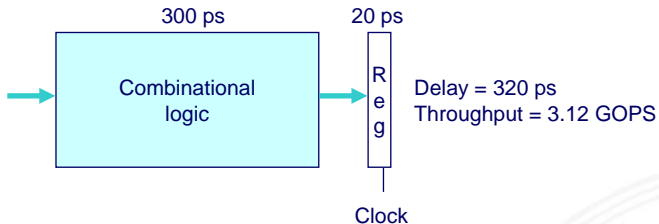
## Arithmetische Pipelines

- ▶ Idee: lange Berechnung in Teilschritte zerlegen  
wichtig bei komplizierteren arithmetischen Operationen
  - ▶ die sonst sehr lange dauern (weil ein großes Schaltnetz)
  - ▶ die als Schaltnetz extrem viel Hardwareaufwand erfordern
  - ▶ Beispiele: Multiplikation, Division, Fließkommaoperationen...
- + Erhöhung des Durchsatzes, wenn Berechnung mehrfach hintereinander ausgeführt wird

## Befehlspipeline im Prozessor

- ▶ Idee: die Phasen der von-Neumann Befehlsabarbeitung (Befehl holen, Befehl decodieren ...) als Pipeline implementieren

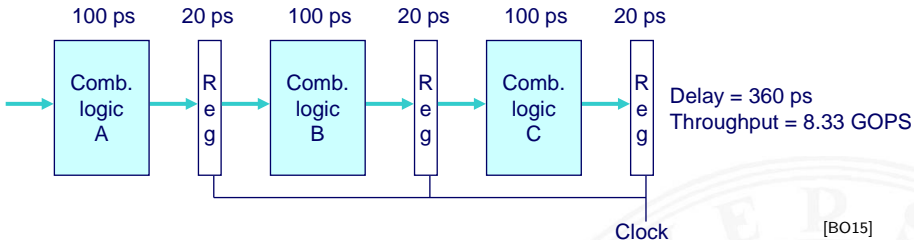
# Beispiel: Schaltnetz ohne Pipeline



[BO15]

- ▶ Verarbeitung erfordert 300 ps
- ▶ weitere 20 ps um das Resultat im Register zu speichern
- ▶ Zykluszeit: mindestens 320 ps

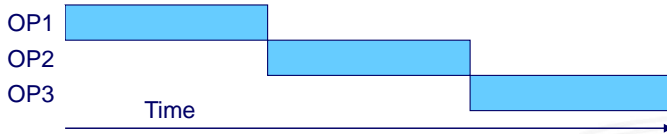
# Beispiel: Version mit 3-stufiger Pipeline



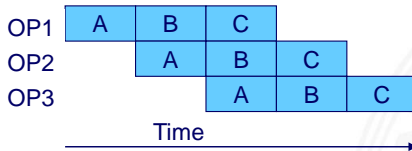
- ▶ Kombinatorische Logik in 3 Blöcke zu je 100 ps aufgeteilt
- ▶ neue Operation, sobald vorheriger Abschnitt durchlaufen wurde  
⇒ alle 120 ps neue Operation
- ▶ allgemeine Latenzzunahme  
⇒ 360 ps von Start bis Ende

# Prinzip: 3-stufige Pipeline

## ▶ ohne Pipeline



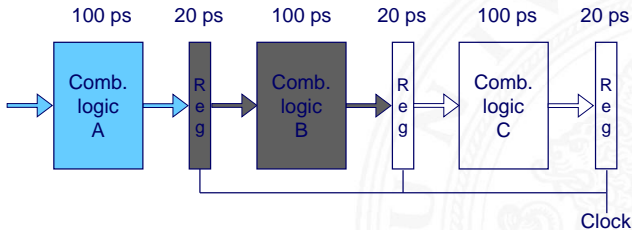
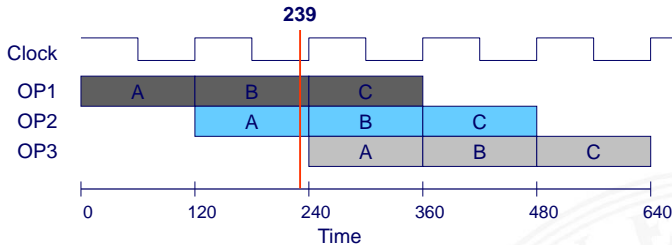
## ▶ 3-stufige Pipeline



[BO15]

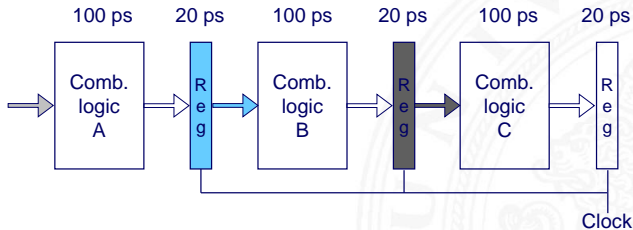
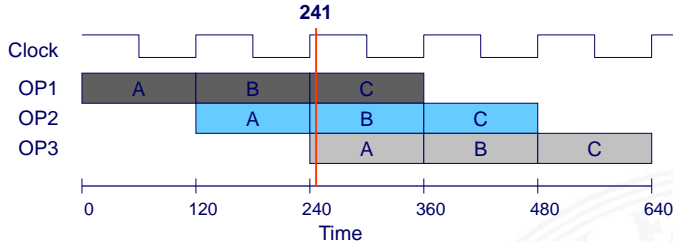


# Timing: 3-stufige Pipeline



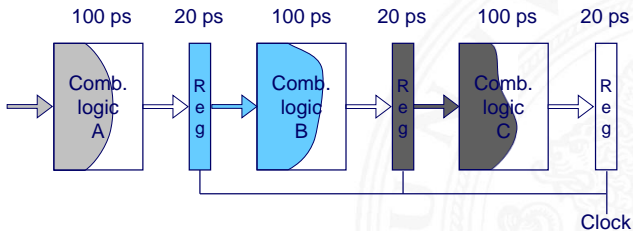
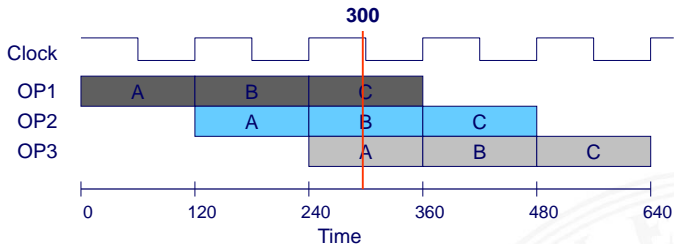
[BO15]

# Timing: 3-stufige Pipeline



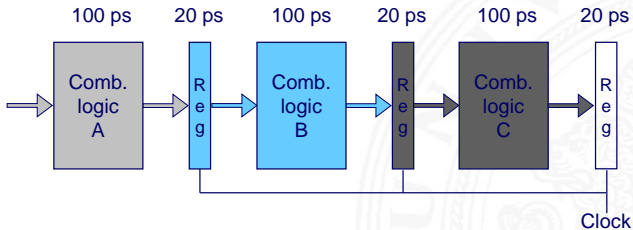
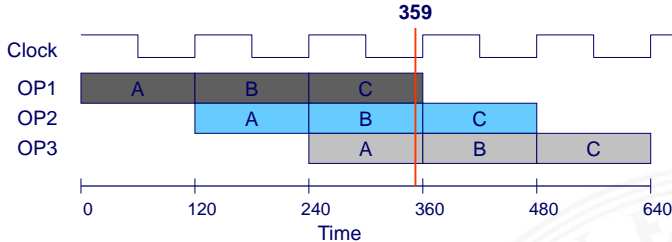
[BO15]

# Timing: 3-stufige Pipeline

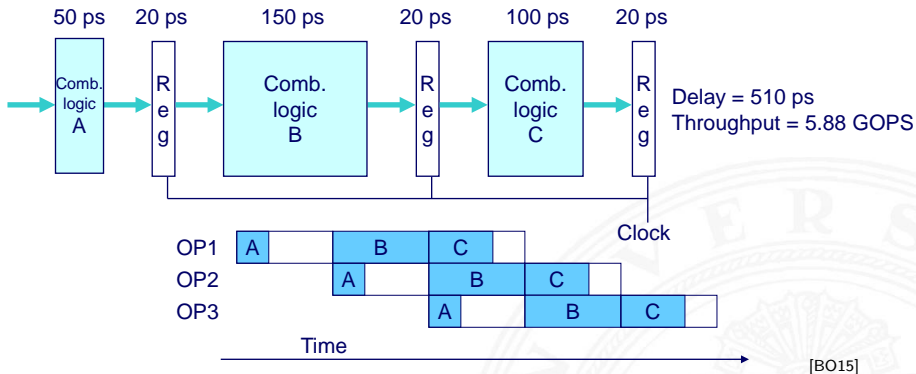


[BO15]

# Timing: 3-stufige Pipeline

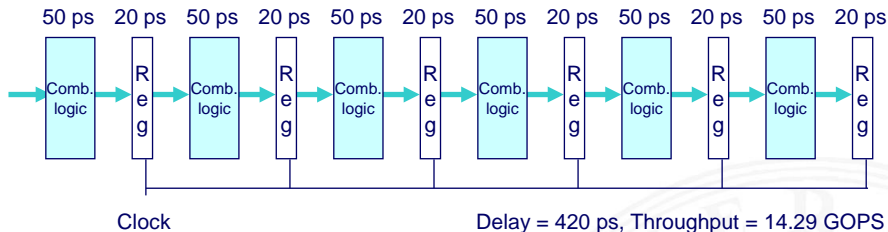


# Limitierungen: nicht uniforme Verzögerungen



- ▶ Taktfrequenz limitiert durch langsamste Stufe
- ▶ Schaltung in möglichst gleich schnelle Stufen aufteilen

# Limitierungen: Register „Overhead“

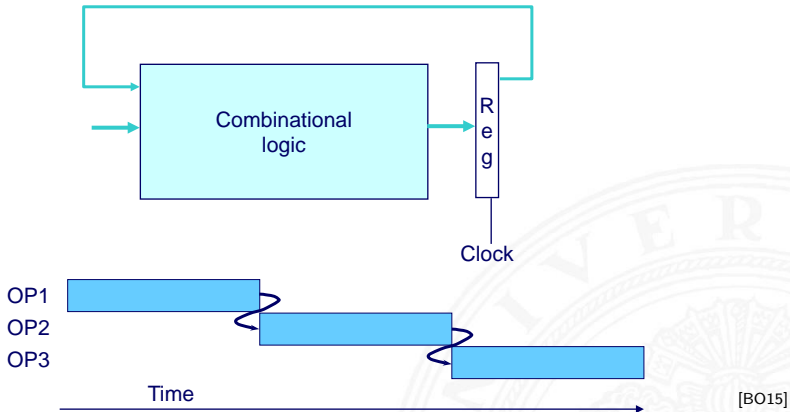


[BO15]

- ▶ registerbedingter Overhead wächst mit Pipelinelänge
- ▶ (anteilige) Taktzeit für das Laden der Register

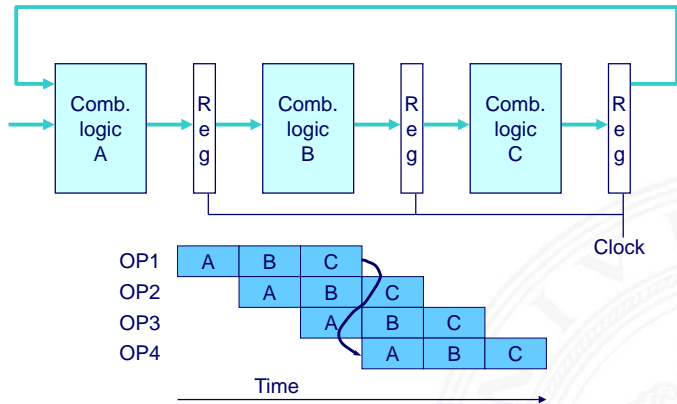
	Overhead	Taktperiode
1-Register:	6,25% 20 ps	320 ps
3-Register:	16,67% 20 ps	120 ps
6-Register:	28,57% 20 ps	70 ps

# Limitierungen: Datenabhängigkeiten



- ▶ jede Operation hängt vom Ergebnis der Vorhergehenden ab

# Limitierungen: Datenabhängigkeiten (cont.)



[BO15]

- ⇒ Resultat-Feedback kommt zu spät für die nächste Operation
- ⇒ Pipelining ändert Verhalten des gesamten Systems





typische Schritte der Befehlsabarbeitung (von ISA abhängig)

- ▶ **IF**      **I**nstruction **F**etch  
Instruktion holen, in Befehlsregister laden

---

- ID**      **I**nstruction **D**ecode  
Instruktion decodieren

---

- OF**      **O**perand **F**etch  
Operanden aus Registern holen

---

- EX**      **E**xecute  
ALU führt Befehl aus

---

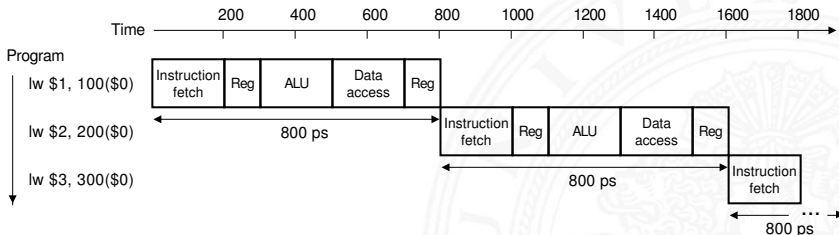
- MEM**    **M**emory access  
Speicherzugriff: Daten laden/abspeichern

---

- WB**      **W**rite **B**ack  
Ergebnis in Register zurückschreiben

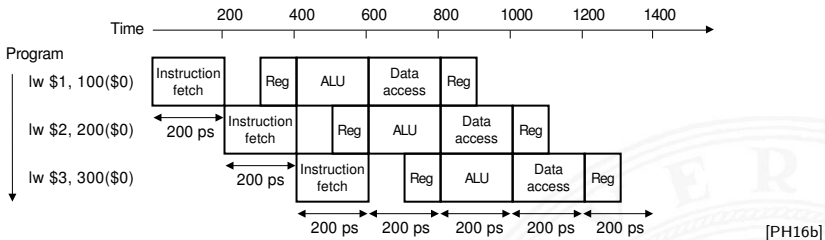
- ▶ je nach Instruktion sind 3-5 dieser Schritte notwendig
  - ▶ *nop*: nur Instruction-Fetch
  - ▶ *jump*: kein Speicher-/Registerzugriff
- ▶ Schritte können auch feiner unterteilt werden (mehr Stufen)

## serielle Bearbeitung ohne Pipelining



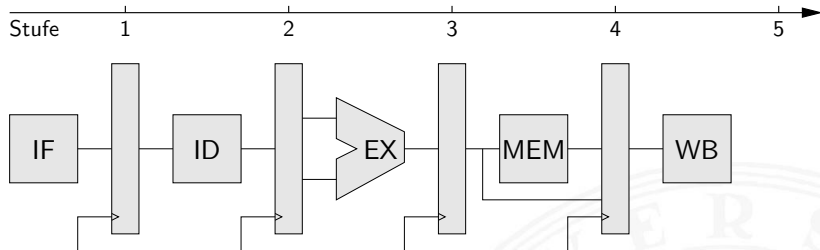
[PH16b]

## Pipelining für die einzelnen Schritte der Befehlsausführung



- ▶ Befehle überlappend ausführen: neue Befehle holen, dann dekodieren, während vorherige noch ausgeführt werden
- ▶ Register trennen Pipelinestufen

# Klassische 5-stufige Pipeline



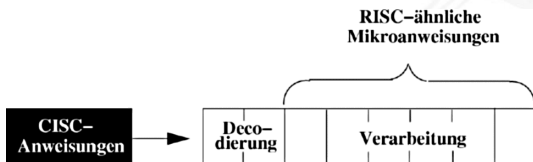
- ▶ Grundidee der ursprünglichen RISC-Architekturen
- + Durchsatz ca.  $3 \dots 5 \times$  besser als serielle Ausführung
- + guter Kompromiss aus Leistung und Hardwareaufwand
  
- ▶ MIPS-Architektur (aus Patterson, Hennessy [PH16b])

▶ MIPS ohne Pipeline

▶ MIPS Pipeline

▶ Pipeline Schema

- ▶ RISC ISA: Pipelining wird direkt umgesetzt
  - ▶ Befehlssätze auf diese Pipeline hin optimiert
  - ▶ IBM-801, MIPS R-2000/R-3000 (1985), SPARC (1987)
- ▶ CISC-Architekturen heute ebenfalls mit Pipeline
  - ▶ Motorola 68020 (zweistufige Pipeline, 1984), Intel 486 (1989), Pentium (1993)...
  - ▶ Befehle in Folgen RISC-ähnlicher Anweisungen umsetzen

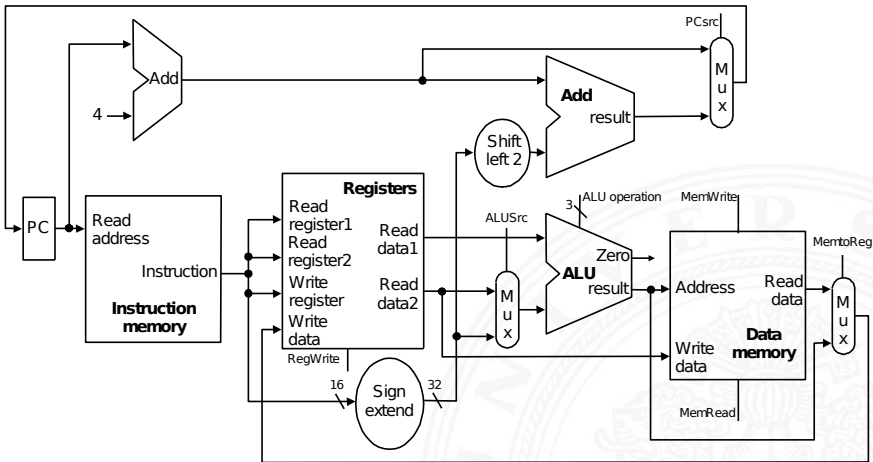


- + CISC-Software bleibt lauffähig
- + Befehlssatz wird um neue RISC Befehle erweitert

# MIPS: serielle Realisierung ohne Pipeline

16.3 Pipelining - MIPS

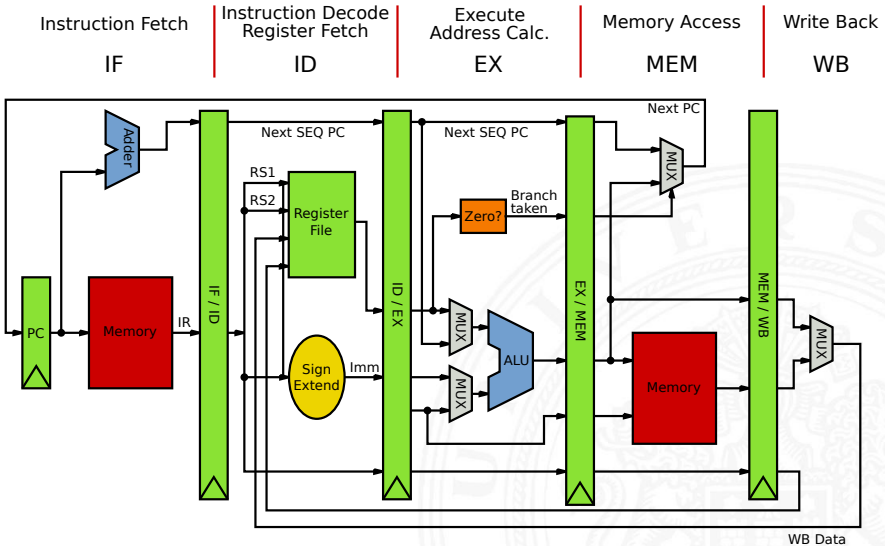
64-040 Rechnerstrukturen



längster Pfad: PC - IM - REG - MUX - ALU - DM - MUX - PC/REG [PH16b]

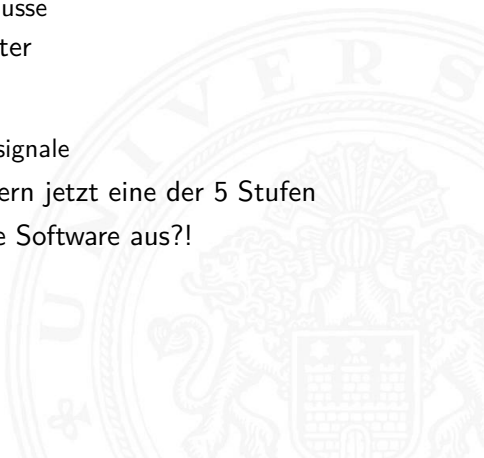
◀ RISC Pipelining

# MIPS: mit 5-stufiger Pipeline





- ▶ die Hardwareblöcke selbst sind unverändert
  - ▶ PC, Addierer fürs Inkrementieren des PC
  - ▶ Registerbank
  - ▶ Rechenwerke: ALU, sign-extend, zero-check
  - ▶ Multiplexer und Leitungen/Busse
- ▶ vier zusätzliche Pipeline-Register
  - ▶ die (dekodierten) Befehle
  - ▶ alle Zwischenergebnisse
  - ▶ alle intern benötigten Statussignale
- ▶ längster Pfad zwischen Registern jetzt eine der 5 Stufen
- ▶ aber wie wirkt sich das auf die Software aus?!

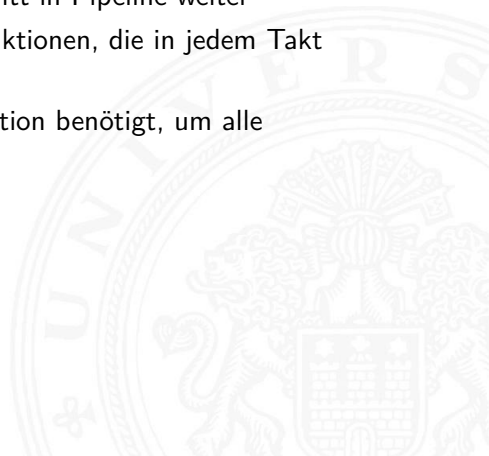






## Begriffe

- ▶ **Pipeline-Stage:** einzelne Stufe der Pipeline
- ▶ **Pipeline Machine Cycle:**  
Instruktion kommt einen Schritt in Pipeline weiter
- ▶ **Durchsatz:** Anzahl der Instruktionen, die in jedem Takt abgeschlossen werden
- ▶ **Latenz:** Zeit, die eine Instruktion benötigt, um alle Pipelinestufen zu durchlaufen



## Vor- und Nachteile

- + Schaltnetze in kleinere Blöcke aufgeteilt  $\Rightarrow$  höherer Takt
- + im Idealfall ein neuer Befehl pro Takt gestartet  $\Rightarrow$  höherer Durchsatz, bessere Performance
- + geringer Zusatzaufwand an Hardware
- + Pipelining ist für den Programmierer nicht direkt sichtbar!
  - Achtung: Daten-/Kontrollabhängigkeiten (s.u.)
- Latenz wird nicht verbessert, bleibt bestenfalls gleich
- Pipeline Takt limitiert durch langsamste Pipelinestufe  
unausgewogene Pipelinestufen reduzieren den Takt und damit die Performanz
- zusätzliche Zeiten, um Pipeline zu füllen bzw. zu leeren

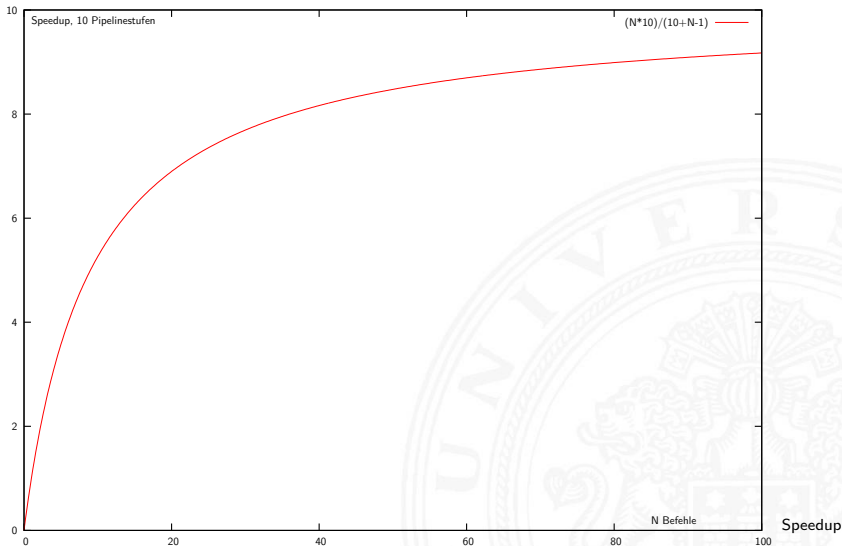
## Analyse

- ▶  $N$  Instruktionen;  $K$  Pipelinestufen
- ▶ ohne Pipeline:  $N \cdot K$  Taktzyklen
- ▶ mit Pipeline:  $K + N - 1$  Taktzyklen
  
- ▶ „Speedup“  $S = \frac{N \cdot K}{K + N - 1}$ ,  $\lim_{N \rightarrow \infty} S = K$

⇒ ein großer Speedup wird erreicht durch

- ▶ große Pipelintiefe:  $K$
- ▶ lange Instruktionssequenzen:  $N$
  
- ▶ wegen Daten- und Kontrollabhängigkeiten nicht erreichbar
- ▶ außerdem: Register-Overhead nicht berücksichtigt

# Prozessorpipeline – Bewertung (cont.)



- ▶ größeres  $K$  wirkt sich direkt auf den Durchsatz aus
- ▶ weniger Logik zwischen den Registern, höhere Taktfrequenzen
- ▶ zusätzlich: technologischer Fortschritt (1985...2010)
- ▶ Beispiele

CPU	Pipelinstufen	Taktfrequenz [MHz]
80386	1	33
Pentium	5	300
Motorola G4	4	500
Motorola G4e	7	1000
Pentium II/III	12	1400
Athlon XP	10/15	2500
Athlon 64, Opteron	12/17	$\leq 3000$
Pentium 4	20	$\leq 5000$

Architekturentscheidungen, die sich auf das Pipelining auswirken

gut für Pipelining

- ▶ gleiche Instruktionslänge
- ▶ wenige Instruktionsformate
- ▶ Load/Store Architektur

## BASIC INSTRUCTION FORMATS

<b>R</b>	opcode	rs	rt	rd	shamt	funct	
	31	26 25	21 20	16 15	11 10	6 5	0
<b>I</b>	opcode	rs	rt	immediate			
	31	26 25	21 20	16 15	0		
<b>J</b>	opcode	address					
	31	26 25					0

## FLOATING-POINT INSTRUCTION FORMATS

<b>FR</b>	opcode	fmt	ft	fs	fd	funct	
	31	26 25	21 20	16 15	11 10	6 5	0
<b>FI</b>	opcode	fmt	ft	immediate			
	31	26 25	21 20	16 15	0		

MIPS-Befehlsformate [PH16b]

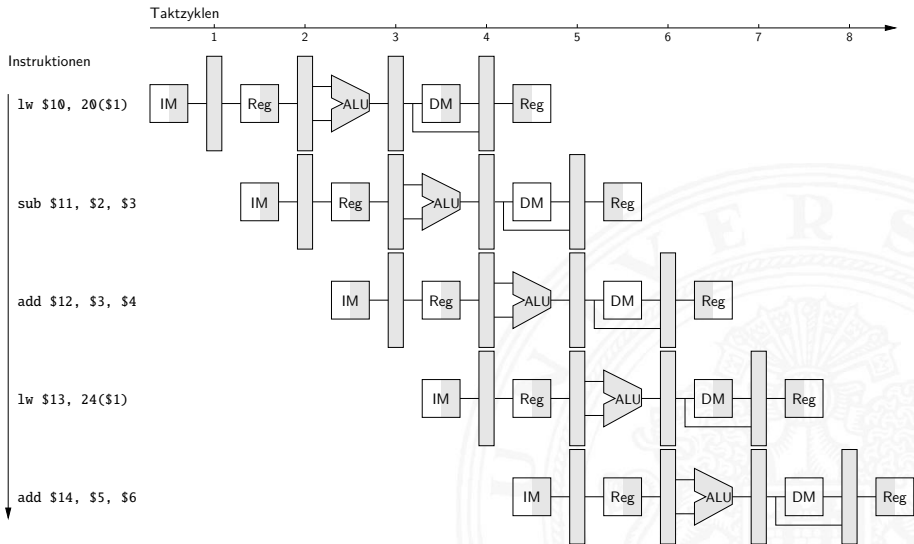
**schlecht** für Pipelining: *Pipelinekonflikte / -Hazards*

- ▶ Strukturkonflikt: gleichzeitiger Zugriff auf eine Ressource durch mehrere Pipelinestufen
- ▶ Datenkonflikt: Ergebnisse von Instruktionen werden innerhalb der Pipeline benötigt
- ▶ Steuerkonflikt: Sprungbefehle in der Pipelinesequenz

**sehr schlecht** für Pipelining

- ▶ Unterbrechung des Programmkontexts: Interrupt, System-Call, Exception. . .
- ▶ (Performanz-) Optimierungen mit „Out-of-Order Execution“ etc.

# Pipeline Schema



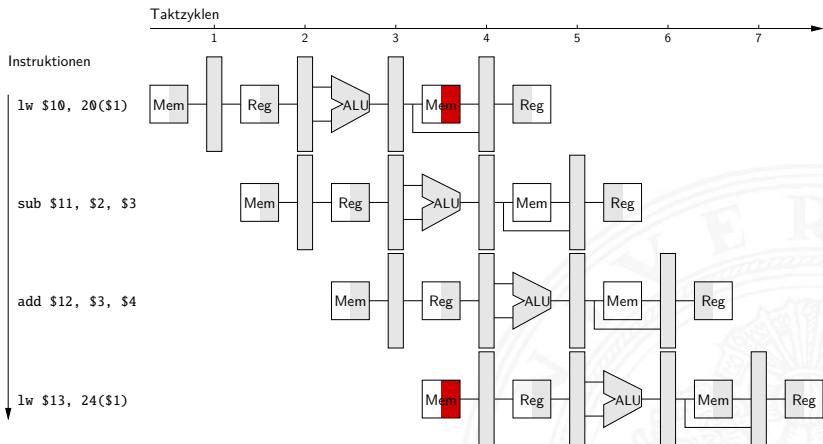
◀ RISC Pipelining



## Strukturkonflikt / Structural Hazard

- ▶ mehrere Stufen wollen gleichzeitig auf eine Ressource zugreifen
- ▶ Beispiel: gleichzeitiger Zugriff auf Speicher ▶ Beispiel
- ⇒ Mehrfachauslegung der betreffenden Ressourcen
  - ▶ Harvard-Architektur vermeidet Strukturkonflikt aus Beispiel
  - ▶ Multi-Port Register
  - ▶ mehrfach vorhandene Busse und Multiplexer...

# Beispiel: Strukturkonflikt



◀ Strukturkonflikte

gleichzeitigen Laden aus *einem* Speicher, zwei verschiedene Adressen

## Datenkonflikt / Data Hazard

- ▶ eine Instruktion braucht die Ergebnisse einer vorhergehenden, diese wird aber noch in der Pipeline bearbeitet
  - ▶ Datenabhängigkeiten aufeinanderfolgender Befehle
    - ▶ Operanden während ID-Phase aus Registerbank lesen
    - ▶ Resultate werden erst in WB-Phase geschrieben
- ⇒ aber: Resultat ist schon nach EX-/MEM-Phase bekannt

▶ Beispiel

## Forwarding

- ▶ zusätzliche Hardware („*Forwarding-Unit*“) kann Datenabhängigkeiten auflösen
- ▶ Änderungen in der Pipeline Steuerung
- ▶ neue Datenpfade und Multiplexer

▶ ohne Forwarding

▶ mit Forwarding

## Rückwärtsabhängigkeiten

- ▶ spezielle Datenabhängigkeit ▶ Beispiel
- ▶ Forwarding-Technik funktioniert nicht, da die Daten erst *später* zur Verfügung stehen
  - ▶ bei längeren Pipelines
  - ▶ bei Load-Instruktionen (s.u.)

## Auflösen von Rückwärtsabhängigkeiten

1. Softwarebasiert, durch den Compiler, Reihenfolge der Instruktionen verändern ▶ Beispiel
  - ▶ andere Operationen (ohne Datenabhängigkeiten) vorziehen
  - ▶ nop-Befehl(e) einfügen

## 2. „Interlocking“

► Beispiel

- ▶ zusätzliche (Hardware) Kontrolleinheit: komplexes Steuerwerk
- ▶ automatisches Stoppen der Pipeline, bis die benötigten Daten zur Verfügung stehen – Strategien:
  - ▶ in Pipeline werden keine neuen Instruktionen geladen
  - ▶ Hardware erzeugt: Pipelineleerlauf / „*pipeline stall*“

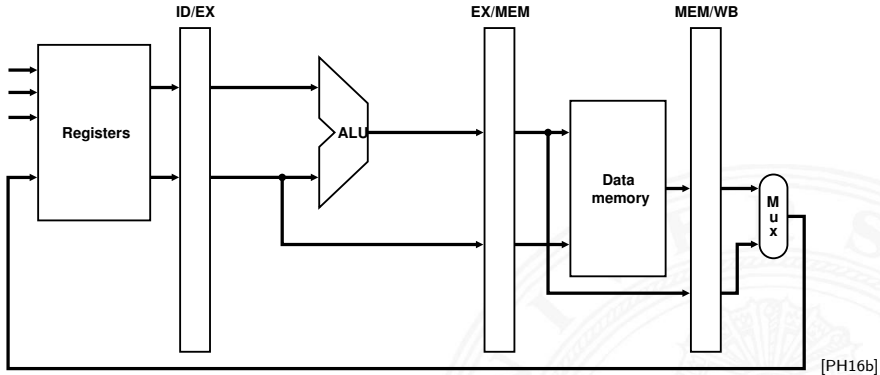
## „Scoreboard“

- ▶ Hardware Einheit zur zentralen Hazard-Erkennung und -Auflösung
- ▶ Verwaltet Instruktionen, benutzte Einheiten und Register der Pipeline (siehe „*Superskalare Rechner*“, ab Folie 1194)

# Beispiel: MIPS Datenpfad

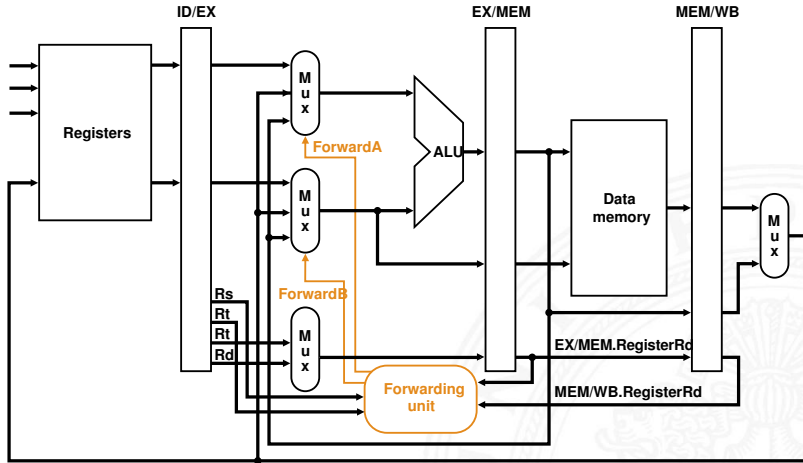
16.5 Pipelining - Hazards

64-040 Rechnerstrukturen



◀ Forwarding

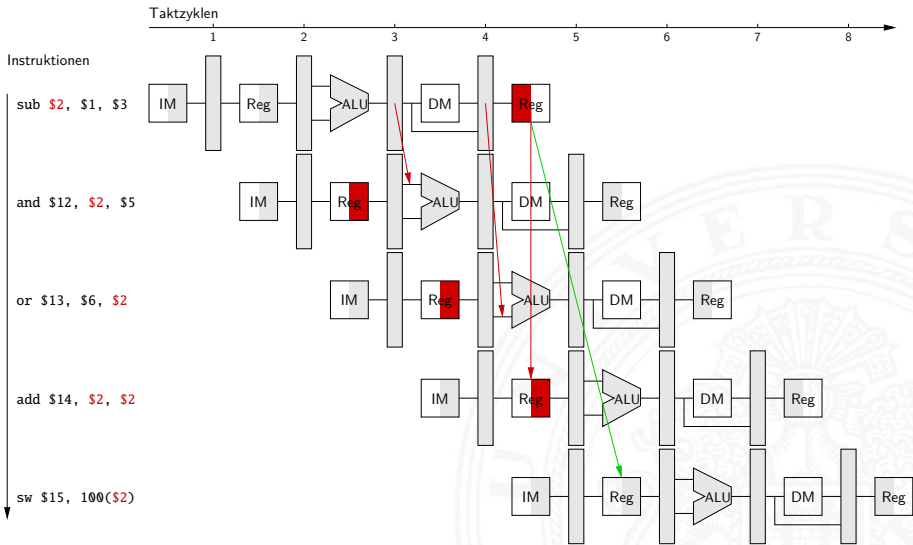
# Beispiel: MIPS Forwarding



[PH16b]

◀ Forwarding

# Beispiel: Datenkonflikt

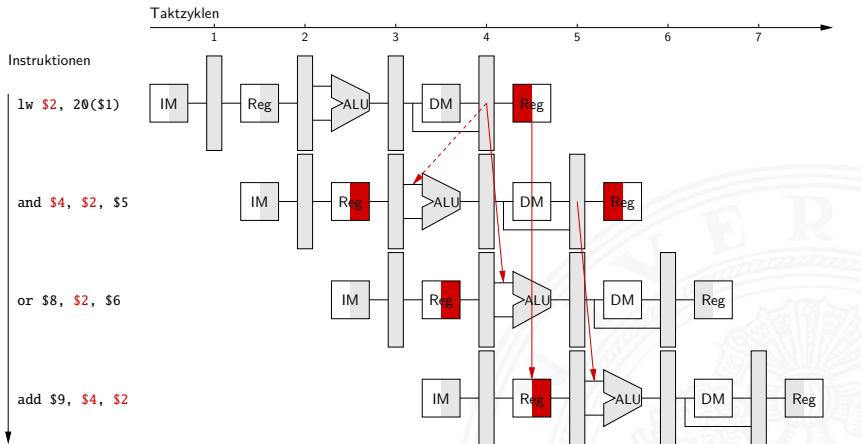


◀ Datenkonflikte

Befehle wollen R2 lesen, während es noch vom ersten Befehl berechnet wird



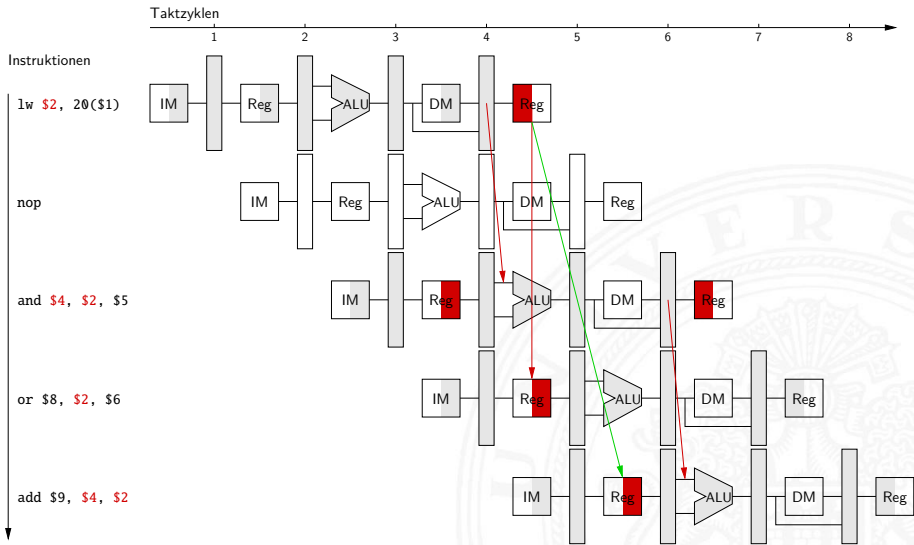
# Beispiel: Rückwärtsabhängigkeit



◀ Datenkonflikte

Befehle wollen R2 lesen, bevor es aus Speicher geladen wird

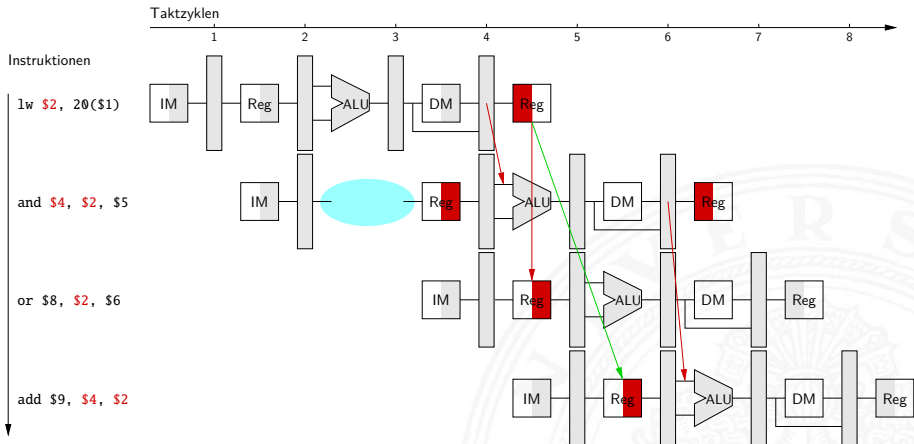
# Beispiel: nop



◀ Datenkonflikte

Compiler kennt Hardware und hat einen nop-Befehl eingefügt

# Beispiel: „bubbles“



◀ Datenkonflikte

Hardware verzögert Bearbeitung, bis Konflikte beseitigt sind („bubbles“)

## Steuerkonflikt / Control Hazard

- ▶ Unterbrechung des sequenziellen Ablaufs durch Sprungbefehle und Unterprogrammaufrufe: `call` und `ret`
  - ▶ Instruktionen die auf (bedingte) Sprünge folgen, werden bereits in die Pipeline geschoben
  - ▶ Sprungadresse und Status (*taken/untaken*) sind aber erst am Ende der EX-Phase bekannt
  - ▶ einige Befehle wurden bereits teilweise ausgeführt, Resultate eventuell „ge-forwarded“
- alle Zwischenergebnisse müssen verworfen werden
  - ▶ inklusive aller Forwarding-Daten
  - ▶ Pipeline an korrekter Zieladresse neu starten
  - ▶ erfordert sehr komplexe Hardware
- jeder (ausgeführte) Sprung kostet enorm Performance

▶ Beispiel

## Lösungsmöglichkeiten für Steuerkonflikte

- ▶ „*Interlocking*“: Pipeline prinzipiell bei Sprüngen leeren
  - ineffizient: ca. 19% der Befehle sind Sprünge
- 1. Annahme: nicht ausgeführter Sprung („*untaken branch*“)
  - + kaum zusätzliche Hardware
  - im Fehlerfall
    - ▶ Pipeline muss geleert werden („*flush instructions*“)
- 2. Sprungentscheidung „vorverlegen“
  - ▶ Software: Compiler zieht andere Instruktionen vor  
Verzögerung nach Sprungbefehl („*delay slots*“)
  - ▶ Hardware: Sprungentscheidung durch Zusatz-ALU  
(nur Vergleiche) während Befehlsdecodierung (z.B. MIPS)

## 3. Sprungvorhersage („branch prediction“)

- ▶ Beobachtung: ein Fall tritt häufiger auf; Schleifendurchlauf, Datenstrukturen durchsuchen etc.
- ▶ mehrere Vorhersageverfahren; oft miteinander kombiniert
- + hohe Trefferquote: bis 90 %

### Statische Sprungvorhersage (softwarebasiert)

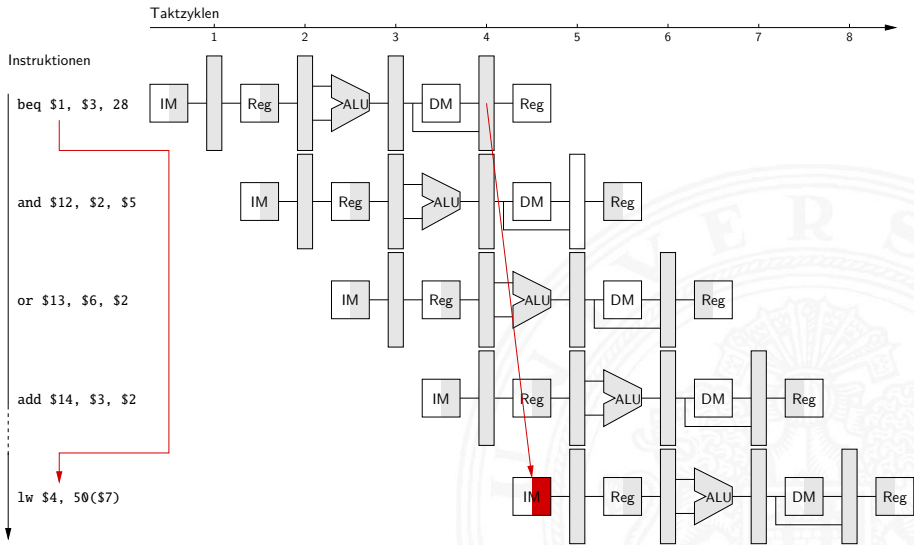
- ▶ Compiler erzeugt extra Bit in Opcode des Sprungbefehls
- ▶ Methoden: Codeanalyse, Profiling...

### Dynamische Sprungvorhersage (hardwarebasiert)

- ▶ Sprünge durch Laufzeitinformation vorhersagen:  
*Wie oft wurde der Sprung in letzter Zeit ausgeführt?*
- ▶ viele verschiedene Verfahren:  
History-Bit, 2-Bit Prädiktor, korrelationsbasierte Vorhersage, Branch History Table, Branch Target Cache...

- ▶ Schleifen abrollen / „Loop unrolling“
  - ▶ zusätzliche Maßnahme zu allen zuvor skizzierten Verfahren
  - ▶ bei statischer Schleifenbedingung möglich
  - ▶ Compiler iteriert Instruktionen in der Schleife (teilweise)
  - längerer Code
  - + Sprünge und Abfragen entfallen
  - + erzeugt sehr lange Codesequenzen ohne Sprünge
    - ⇒ Pipeline kann optimal ausgenutzt werden

# Beispiel: Steuerkonflikt



◀ Steuerkonflikte



- ▶ von-Neumann Zyklus auf separate Phasen aufteilen
- ▶ überlappende Ausführung von mehreren Befehlen
  - ▶ einfachere Hardware für jede Phase  $\Rightarrow$  höherer Takt
  - ▶ mehrere Befehle in Bearbeitung  $\Rightarrow$  höherer Durchsatz
  - ▶ 5-stufige RISC-Pipeline: IF $\rightarrow$ ID/OE $\rightarrow$ Exe $\rightarrow$ Mem $\rightarrow$ WB
  - ▶ mittlerweile sind 9...20 Stufen üblich
- ▶ Struktur-, Daten- und Steuerkonflikte
  - ▶ Lösung durch mehrfache/bessere Hardware
  - ▶ Data-Forwarding umgeht viele Datenabhängigkeiten
  - ▶ Sprungbefehle sind ein ernstes Problem
- ▶ Pipelining ist prinzipiell unabhängig von der ISA
  - ▶ einige Architekturen basieren auf Pipelining (MIPS)
  - ▶ Compiler/Tools/Programmierer sollten CPU Pipeline kennen

- [PH16a] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface: ARM Edition*. Morgan Kaufmann Publishers Inc., 2016.  
ISBN 978-0-12-801733-3
- [PH16b] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle*. 5. Auflage, Oldenbourg, 2016. ISBN 978-3-11-044605-0
- [HP12] J.L. Hennessy, D.A. Patterson: *Computer architecture – A quantitative approach*. 5th edition, Morgan Kaufmann Publishers Inc., 2012.  
ISBN 978-0-12-383872-8

[BO15] R.E. Bryant, D.R. O'Hallaron:

*Computer systems – A programmers perspective.*

3rd global ed., Pearson Education Ltd., 2015.

ISBN 978-1-292-10176-7. [csapp.cs.cmu.edu](http://csapp.cs.cmu.edu)

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur –*

*Von der digitalen Logik zum Parallelrechner.*

6. Auflage, Pearson Deutschland GmbH, 2014.

ISBN 978-3-86894-238-5