64-040 Modul IP7: Rechnerstrukturen

http://tams.informatik.uni-hamburg.de/lectures/2011ws/vorlesung/rs

Kapitel 20

Andreas Mäder

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

Wintersemester 2011/2012

Kapitel 20

Computerarchitektur

- Befehlssätze / ISA
- Sequenzielle Befehlsabarbeitung
- Pipelining
- Superskalare Prozessoren
- Beispiele

Bewertung der ISA

Kriterien für einen *guten* Befehlssatz

- vollständig: alle notwendigen Instruktionen verfügbar
- orthogonal: keine zwei Instruktionen leisten das Gleiche
- symmetrisch: z.B. Addition $\Leftrightarrow$ Subtraktion
- adäquat: technischer Aufwand entsprechend zum Nutzen
- effizient: kurze Ausführungszeiten

Statistiken zeigen: Dominanz der einfachen Instruktionen

Bewertung der ISA (cont.)

- x86-Prozessor

	Anweisung	Ausführungshäufigkeit %
1.	load	22 %
2.	conditional branch	20 %
3.	compare	16 %
4.	store	12 %
5.	add	8 %
6.	and	6 %
7.	sub	5 %
8.	move reg-reg	4 %
9.	call	1 %
10.	return	1 %
Total		96 %

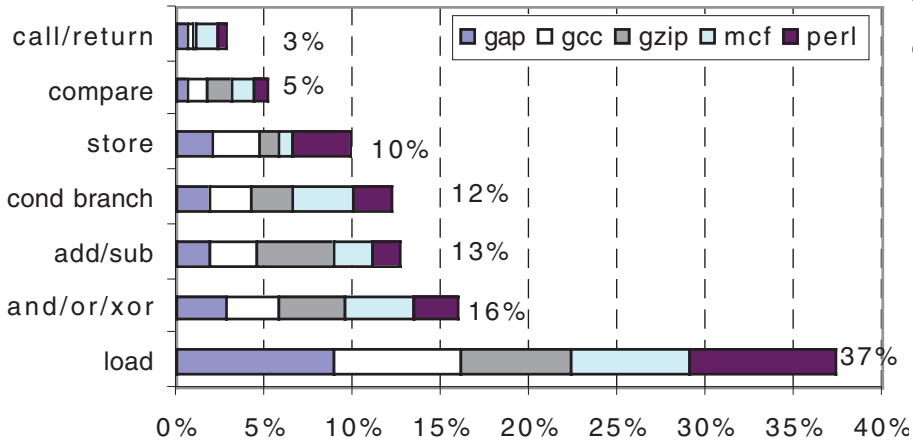
Bewertung der ISA (cont.)

Instruction	compress	eqntott	espresso	gcc (cc1)	li	Int. average
load	20.8%	18.5%	21.9%	24.9%	23.3%	22%
store	13.8%	3.2%	8.3%	16.6%	18.7%	12%
add	10.3%	8.8%	8.15%	7.6%	6.1%	8%
sub	7.0%	10.6%	3.5%	2.9%	3.6%	5%
mul				0.1%		0%
div						0%
compare	8.2%	27.7%	15.3%	13.5%	7.7%	16%
mov reg-reg	7.9%	0.6%	5.0%	4.2%	7.8%	4%
load imm	0.5%	0.2%	0.6%	0.4%		0%
cond. branch	15.5%	28.6%	18.9%	17.4%	15.4%	20%
uncond. branch	1.2%	0.2%	0.9%	2.2%	2.2%	1%
call	0.5%	0.4%	0.7%	1.5%	3.2%	1%
return, jmp indirect	0.5%	0.4%	0.7%	1.5%	3.2%	1%
shift	3.8%		2.5%	1.7%		1%
and	8.4%	1.0%	8.7%	4.5%	8.4%	6%
or	0.6%		2.7%	0.4%	0.4%	1%
other (xor, not, . . .)	0.9%		2.2%	0.1%		1%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
compare FP						0%
mov reg-reg FP						0%
other (abs, sqrt, . . .)						0%

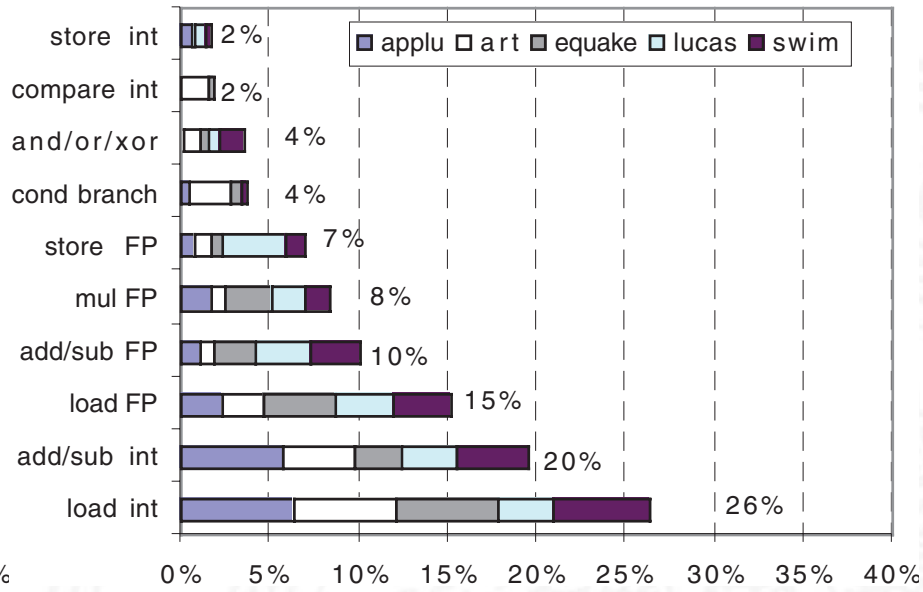
Figure D.15 80x86 instruction mix for five SPECint92 programs.

Bewertung der ISA (cont.)

- MIPS-Prozessor



SPECint2000 (96%)



SPECfp2000 (97%)

Bewertung der ISA (cont.)

- ca. 80 % der Berechnungen eines typischen Programms verwenden nur ca. 20 % der Instruktionen einer CPU
- am häufigsten gebrauchten Instruktionen sind einfache Instruktionen: load, store, add...

$\Rightarrow$ Motivation für RISC

CISC – Befehlssätze

Complex **I**nstruction **S**et **C**omputer

- aus der Zeit der ersten Großrechner, 60er Jahre
- Programmierung auf Assemblerebene
- Komplexität durch sehr viele (mächtige) Befehle umgehen

CISC Befehlssätze

- Instruktionssätze mit mehreren hundert Befehlen (> 300)
- sehr viele Adressierungsarten, -Kombinationen
- verschiedene, unterschiedlich lange Instruktionsformate
- fast alle Befehle können auf Speicher zugreifen
  - mehrere Schreib- und Lesezugriffe pro Befehl
  - komplexe Adressberechnung

CISC – Befehlssätze (cont.)

- Stack-orientierter Befehlssatz
  - Übergabe von Argumenten
  - Speichern des Programmzählers
  - explizite „Push“ und „Pop“ Anweisungen
- Zustandscodes („Flags“)
  - gesetzt durch arithmetische und logische Anweisungen

Konsequenzen

\+ nah an der Programmiersprache, einfacher Assembler

\+ kompakter Code: weniger Befehle holen, kleiner I-Cache

– Pipelining schwierig

– Ausführungszeit abhängig von: Befehl, Adressmodi. . .

– Instruktion holen schwierig, da variables Instruktionsformat

– Speicherhierarchie schwer handhabbar: Adressmodi

CISC – Mikroprogrammierung

- ein Befehl kann nicht in einem Takt abgearbeitet werden

$\Rightarrow$ Unterteilung in Mikroinstruktionen ($\varnothing$ 5...7)

- Ablaufsteuerung durch endlichen Automaten
- meist als ROM (RAM) implementiert, das *Mikroprogammworte* beinhaltet

1. horizontale Mikroprogrammierung
   - langes Mikroprogrammwort (ROM-Zeile)
   - steuert direkt alle Operationen
   - Spalten entsprechen: Kontrolleitungen und Folgeadressen

CISC – Mikroprogrammierung (cont.)

2. vertikale Mikroprogrammierung
   - kurze Mikroprogrammworte
   - Spalten enthalten Mikrooperationscode
   - mehrstufige Decodierung für Kontrolleitungen

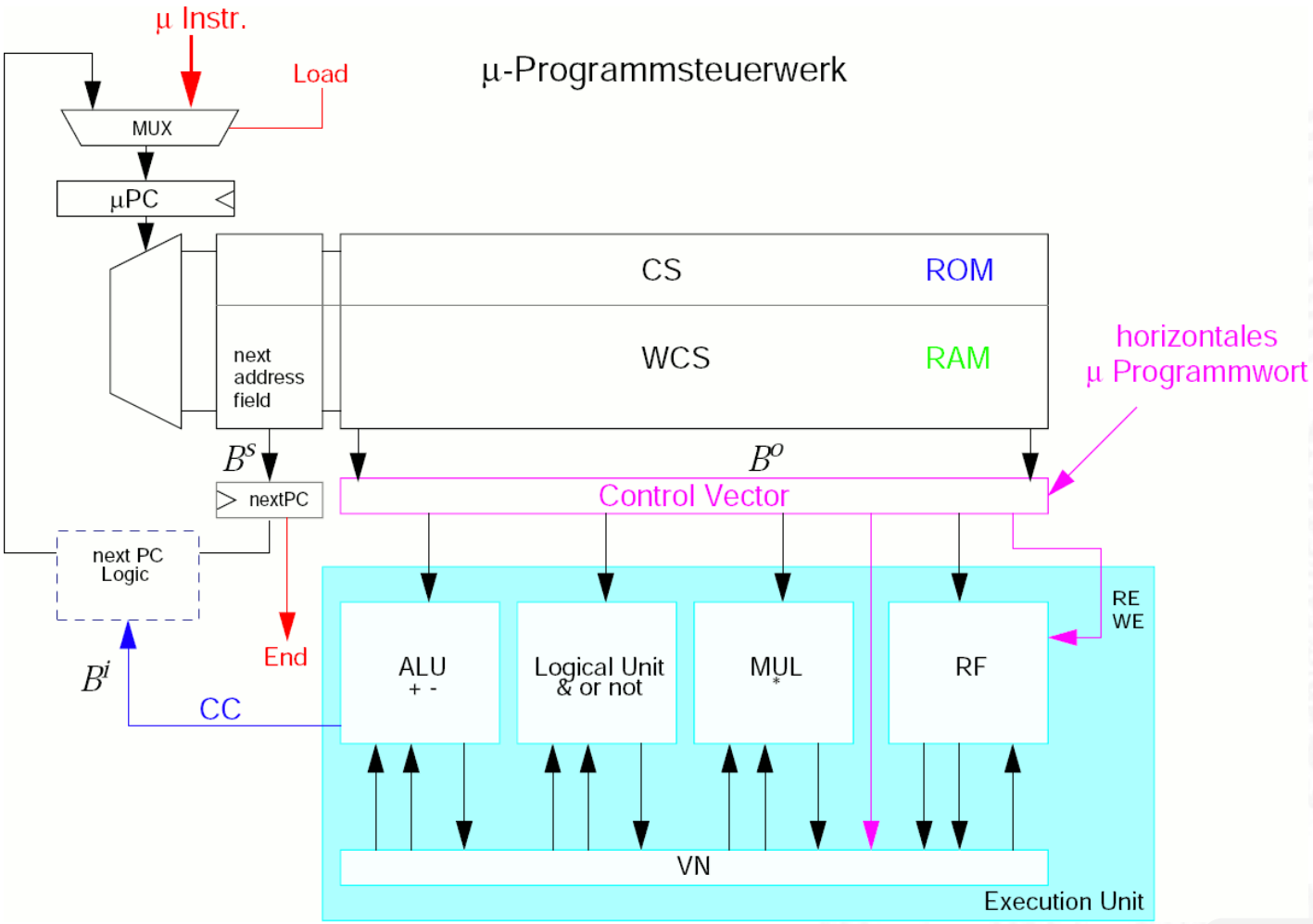
\+ CISC-Befehlssatz mit wenigen Mikrobefehlen realisieren

\+ bei RAM: Mikrobefehlssatz austauschbar

– (mehrstufige) ROM/RAM Zugriffe: zeitaufwändig

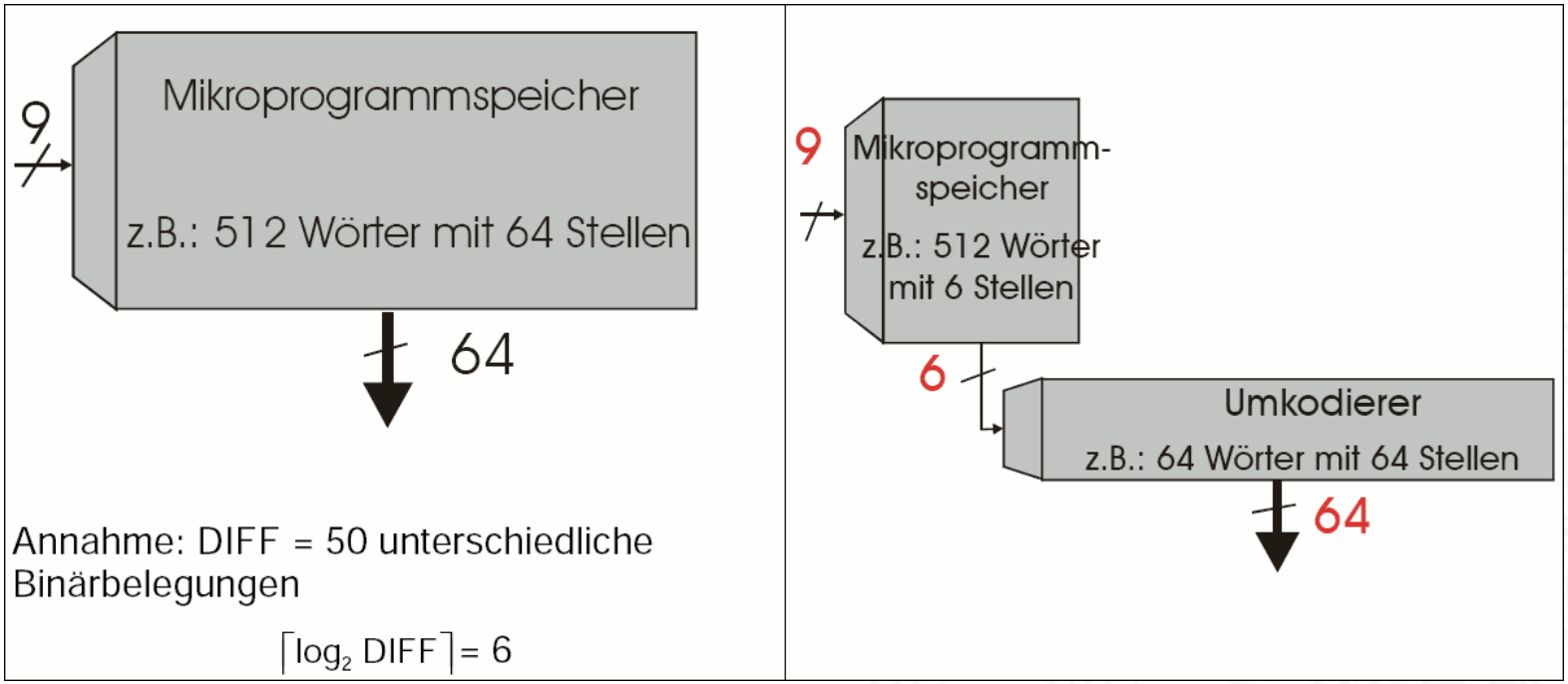
▶ ▸ horizontale Mikroprog. ▸ vertikale Mikroprog.

horizontale Mikroprogrammierung



◂ Mikroprogrammierung

vertikale Mikroprogrammierung



◂ Mikroprogrammierung

RISC – Befehlssätze

Reduced **I**nstruction **S**et **C**omputer

- Grundidee: Komplexitätsreduktion in der CPU
- internes Projekt bei IBM, seit den 80er Jahren: „RISC-Boom“
  - von Hennessy (Stanford) und Patterson (Berkeley) publiziert
- Hochsprachen und optimierende Compiler

$\Rightarrow$ kein Bedarf mehr für mächtige Assemblerbefehle

$\Rightarrow$ pro Assemblerbefehl muss nicht mehr „möglichst viel“ lokal in der CPU gerechnet werden (CISC Mikroprogramm)

RISC – Befehlssätze (cont.)

RISC Befehlssätze

- reduzierte Anzahl einfacher Instruktionen (z.B. 128)
  - benötigen in der Regel mehr Anweisungen für eine Aufgabe
  - werden aber mit kleiner, schneller Hardware ausgeführt
- Register-orientierter Befehlssatz
  - viele Register (üblicherweise ≥ 32)
  - Register für Argumente, „Return“-Adressen, Zwischenergebnisse
- Speicherzugriff *nur* durch „Load“ und „Store“ Anweisungen
- alle anderen Operationen arbeiten auf Registern
- keine Zustandscodes (Flag-Register)
  - Testanweisungen speichern Resultat direkt im Register

RISC – Befehlssätze (cont.)

Konsequenzen

- \+ fest-verdrahtete Logik, kein Mikroprogramm
- \+ einfache Instruktionen, wenige Adressierungsarten
- \+ Pipelining gut möglich
- \+ Cycles per Instruction = 1
  in Verbindung mit Pipelining: je Takt (mind.) ein neuer Befehl
- – längerer Maschinencode
- – viele Register notwendig
- ▶ optimierende Compiler nötig / möglich
- ▶ High-performance Speicherhierachie notwendig

CISC vs. RISC

ursprüngliche Debatte

- streng geteilte Lager
- pro CISC: einfach für den Compiler; weniger Code Bytes
- pro RISC: besser für optimierende Compiler;
  schnelle Abarbeitung auf einfacher Hardware

aktueller Stand

- Grenzen verwischen
  - RISC-Prozessoren werden komplexer
  - CISC-Prozessoren weisen RISC-Konzepte oder gar RISC-Kern auf
- für Desktop Prozessoren ist die Wahl der ISA kein Thema
  - Code-Kompatibilität ist sehr wichtig!
  - mit genügend Hardware wird alles schnell ausgeführt
- eingebettete Prozessoren: eindeutige RISC-Orientierung
  + kleiner, billiger, weniger Leistung

ISA Design heute

- Restriktionen durch Hardware abgeschwächt
- Code-Kompatibilität leichter zu erfüllen
  - Emulation in Firm- und Hardware
- Intel bewegt sich weg von IA-32
  - erlaubt nicht genug Parallelität
- hat IA-64 eingeführt („Intel Architecture 64-bit“)

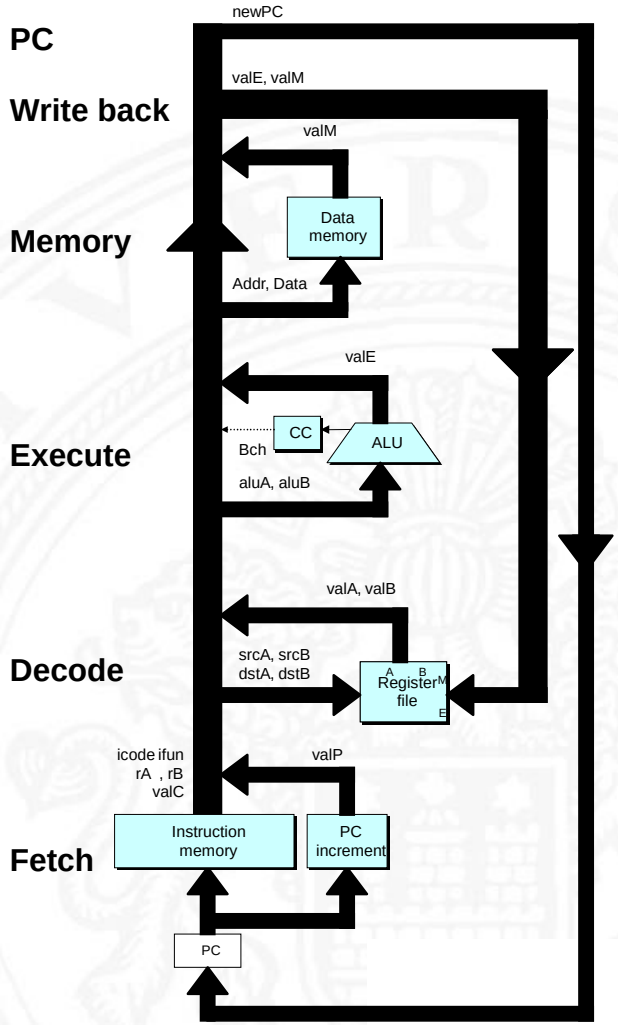
  $\Rightarrow$ neuer Befehlssatz mit expliziter Parallelität (EPIC)

  $\Rightarrow$ 64-bit Wortgrößen (überwinden Adressraumlimits)

  $\Rightarrow$ benötigt hoch entwickelte Compiler

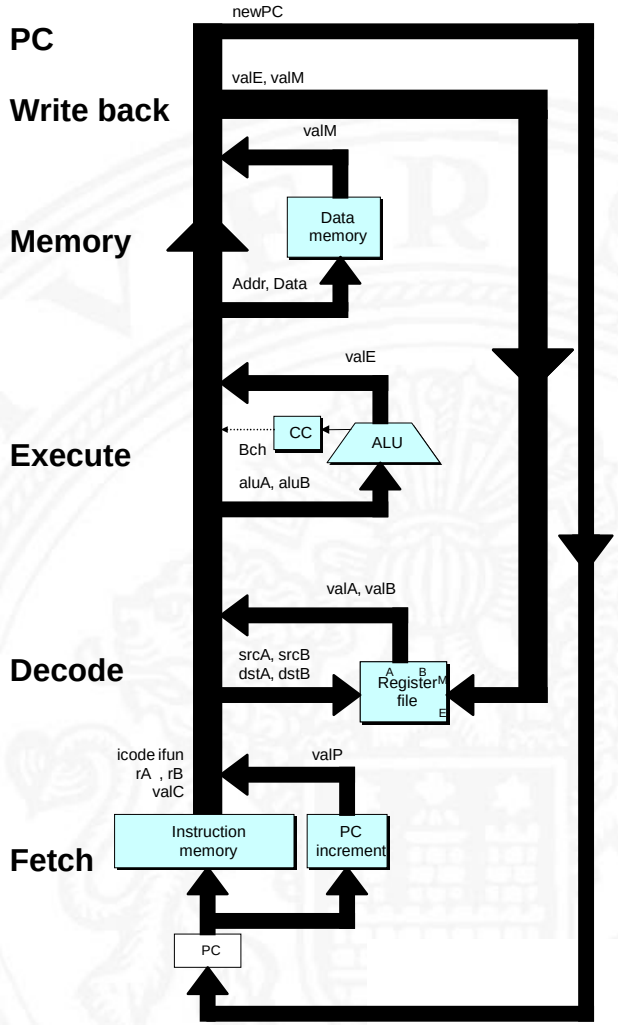
Sequenzielle Hardwarestruktur

- interner Zustand
  - Programmzähler Register PC
  - Zustandscode Register CC
  - Registerbank
  - Speicher
    - gemeinsamer Speicher für Daten und Anweisungen
- von-Neumann Abarbeitung
  - Befehl aus Speicher laden
    PC enthält Adresse
  - Verarbeitung durch die Stufen
  - Programmzähler aktualisieren

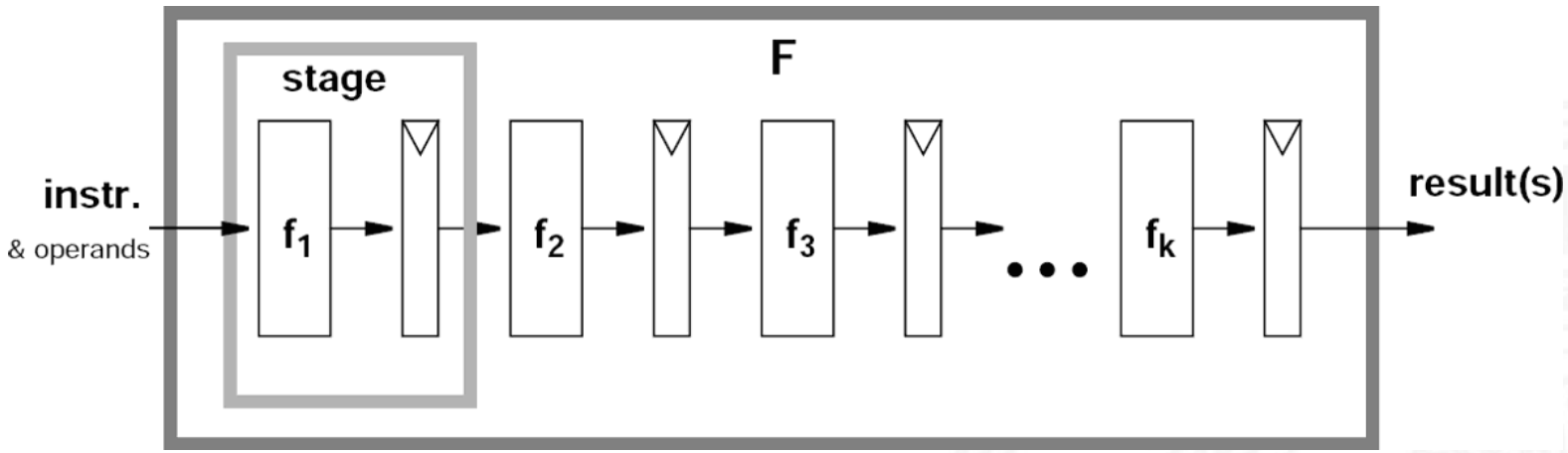


Sequenzielle Befehlsabarbeitung

- Befehl holen „Fetch“
  - Anweisung aus Speicher lesen
- Befehl decodieren „Decode“
  - Befehlsregister interpretieren
  - Operanden holen
- Befehl ausführen „Execute“
  - berechne Wert oder Adresse
- Speicherzugriff „Memory“
  - Daten lesen oder schreiben
- Registerzugriff „Write Back“
  - in Registerbank schreiben
- Programmzähler aktualisieren
  - inkrementieren –oder–
  - Spreicher-/Registerinhalt bei Sprung



Pipelining / Fließbandverarbeitung



Grundidee

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

Pipelining / Fließbandverarbeitung (cont.)

Pipelining-Konzept

- Prozess in unabhängige Abschnitte aufteilen
- Objekt sequenziell durch diese Abschnitte laufen lassen
- zu jedem gegebenen Zeitpunkt werden zahlreiche Objekte bearbeitet

Konsequenz

- lässt Vorgänge gleichzeitig ablaufen
- „Real-World Pipelines“: Autowaschanlagen

Pipelining / Fließbandverarbeitung (cont.)

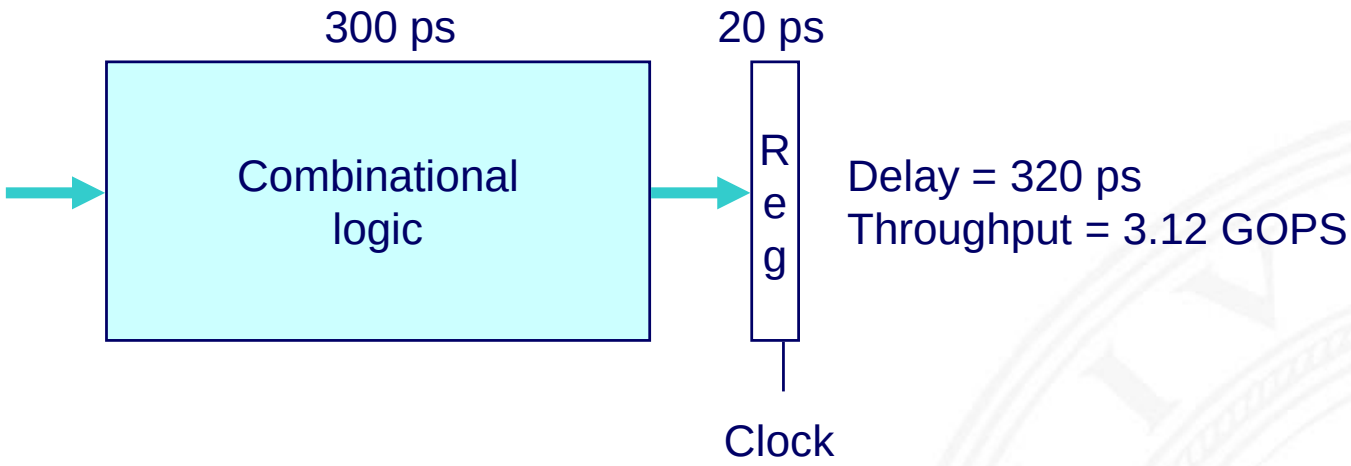
Arithmetische Pipelines

- 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

(RISC) Prozessorpipelines

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

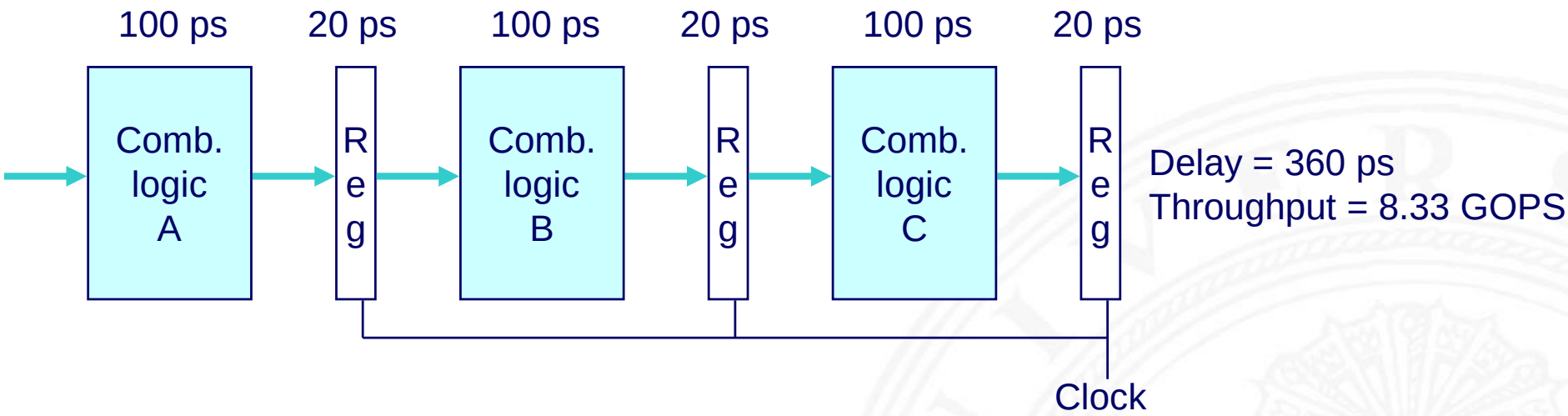
Berechnungsbeispiel: ohne Pipeline



System

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

Berechnungsbeispiel: Version mit 3-stufiger Pipeline

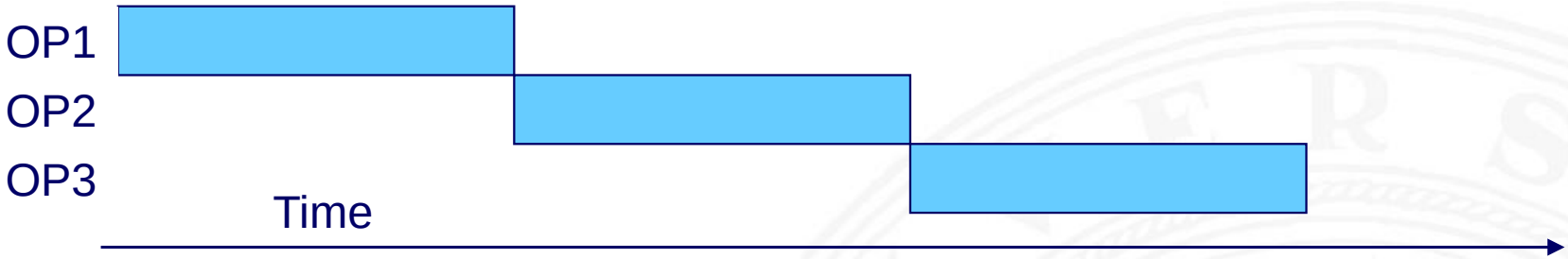


System

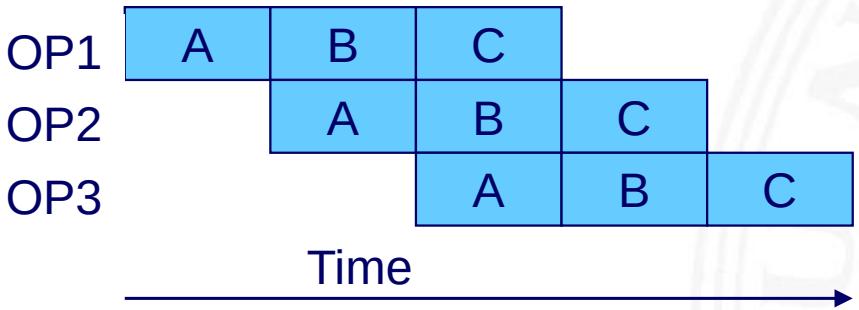
- Kombinatorische Logik in 3 Blöcke zu je 100 ps aufgeteilt
- neue Operation, sobald vorheriger Abschnitt durchlaufen wurde
  $\Rightarrow$ alle 120 ps neue Operation
- allgemeine Latenzzunahme
  $\Rightarrow$ 360 ps von Start bis Ende

Funktionsweise der Pipeline

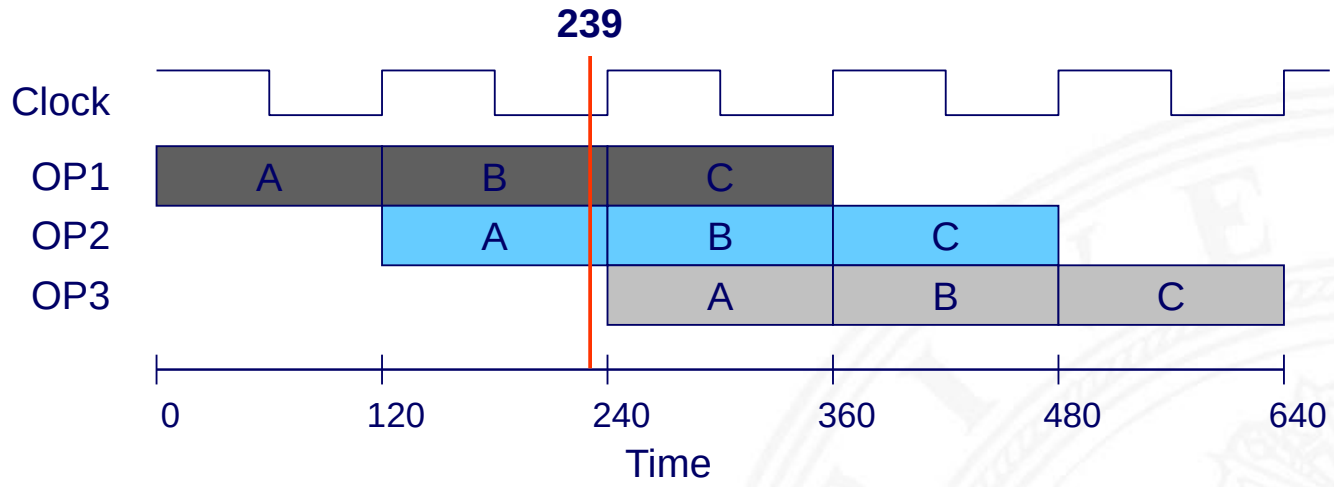
- ohne Pipeline

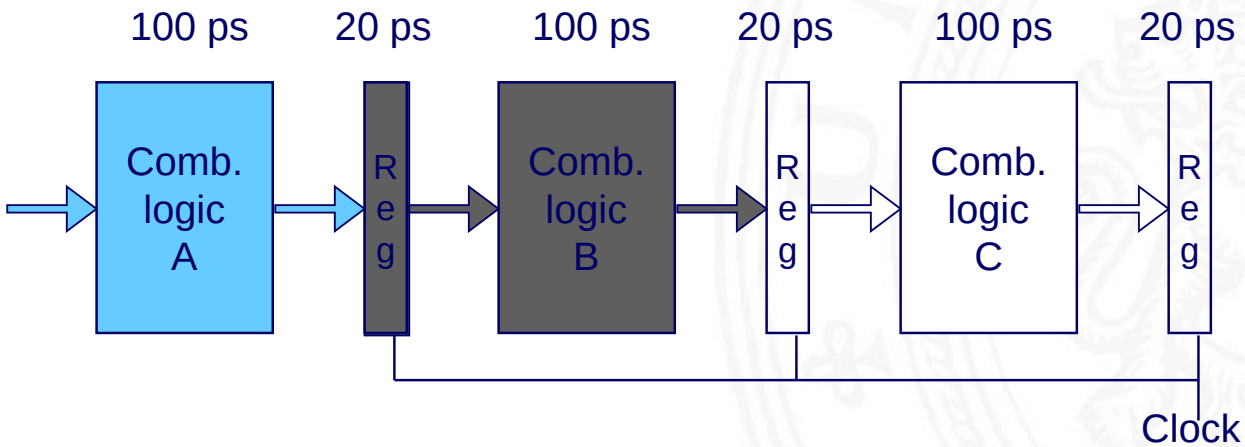


- 3-stufige Pipeline

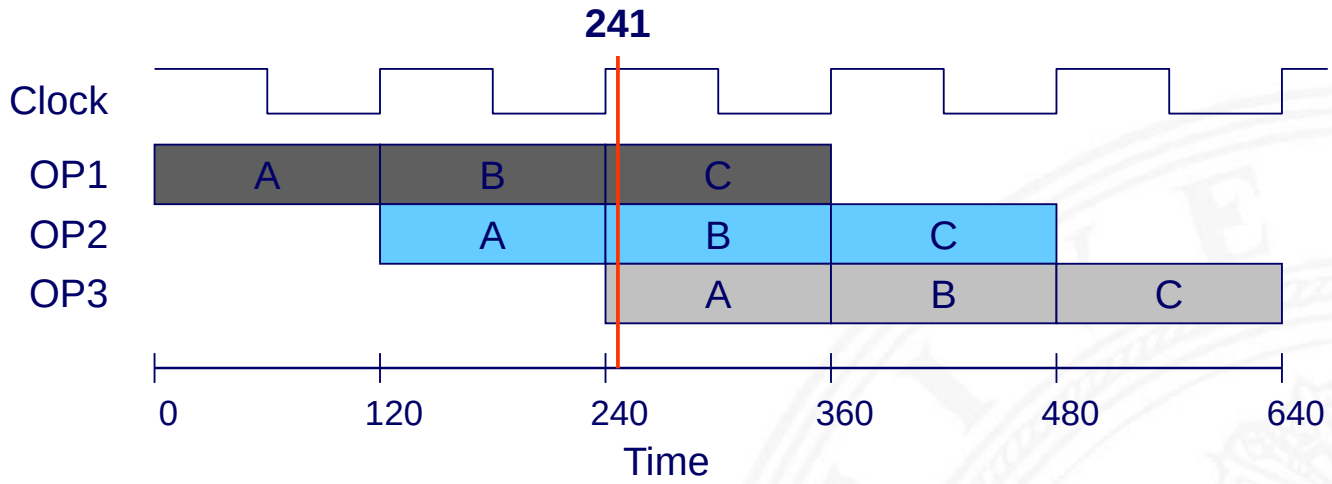


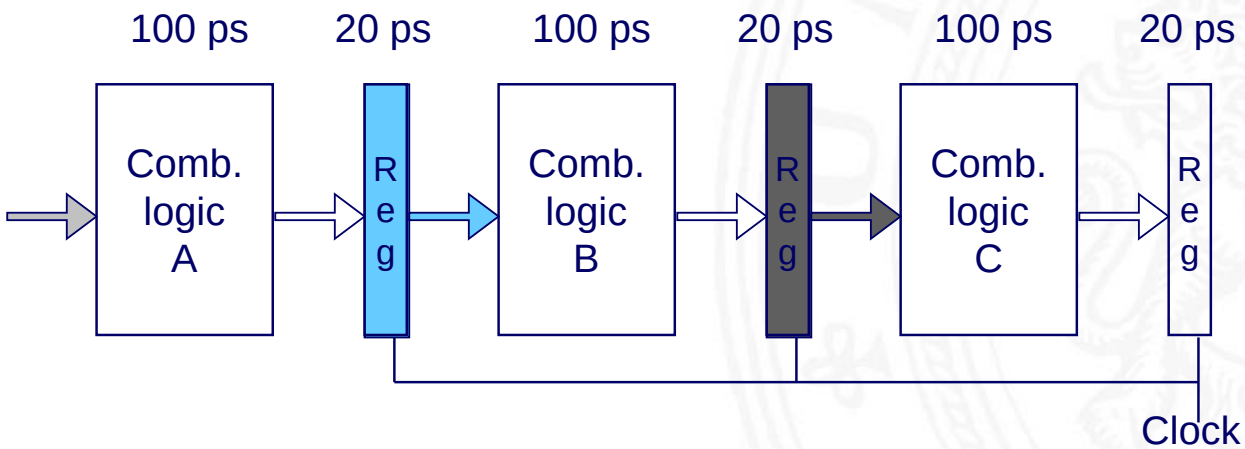
Beispiel: 3-stufige Pipeline



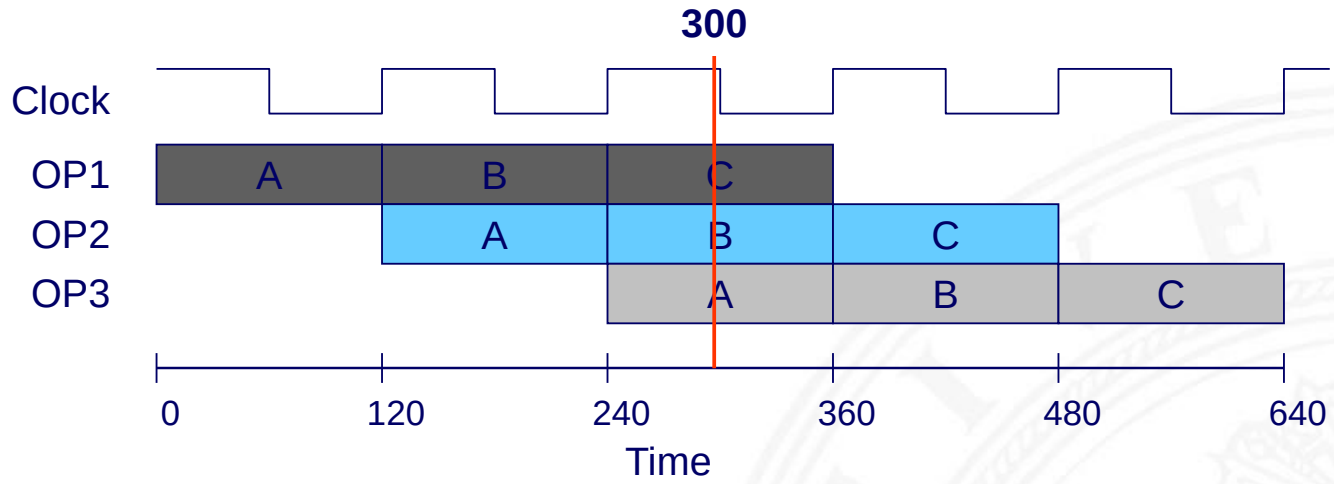


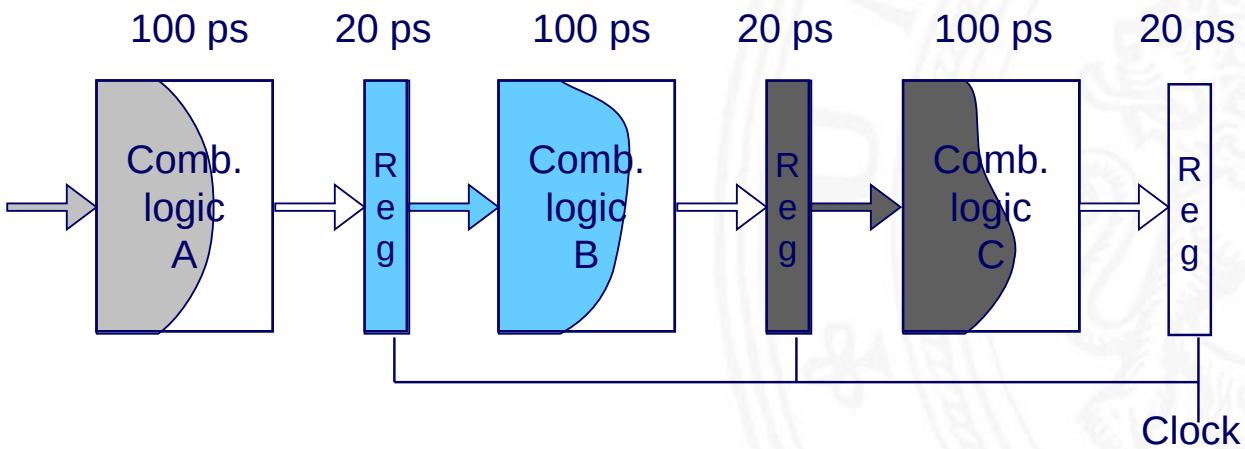
Beispiel: 3-stufige Pipeline



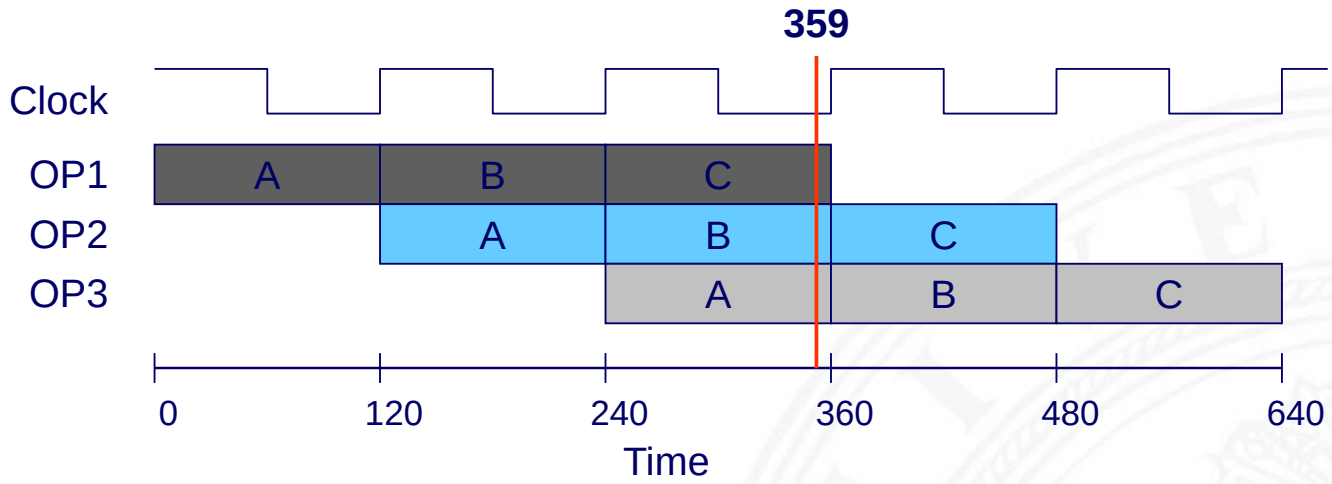


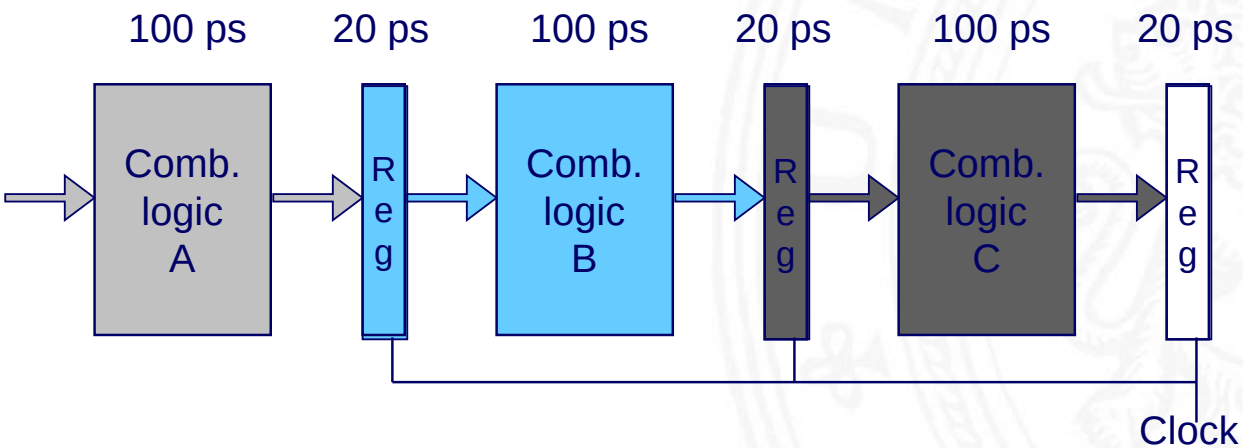
Beispiel: 3-stufige Pipeline



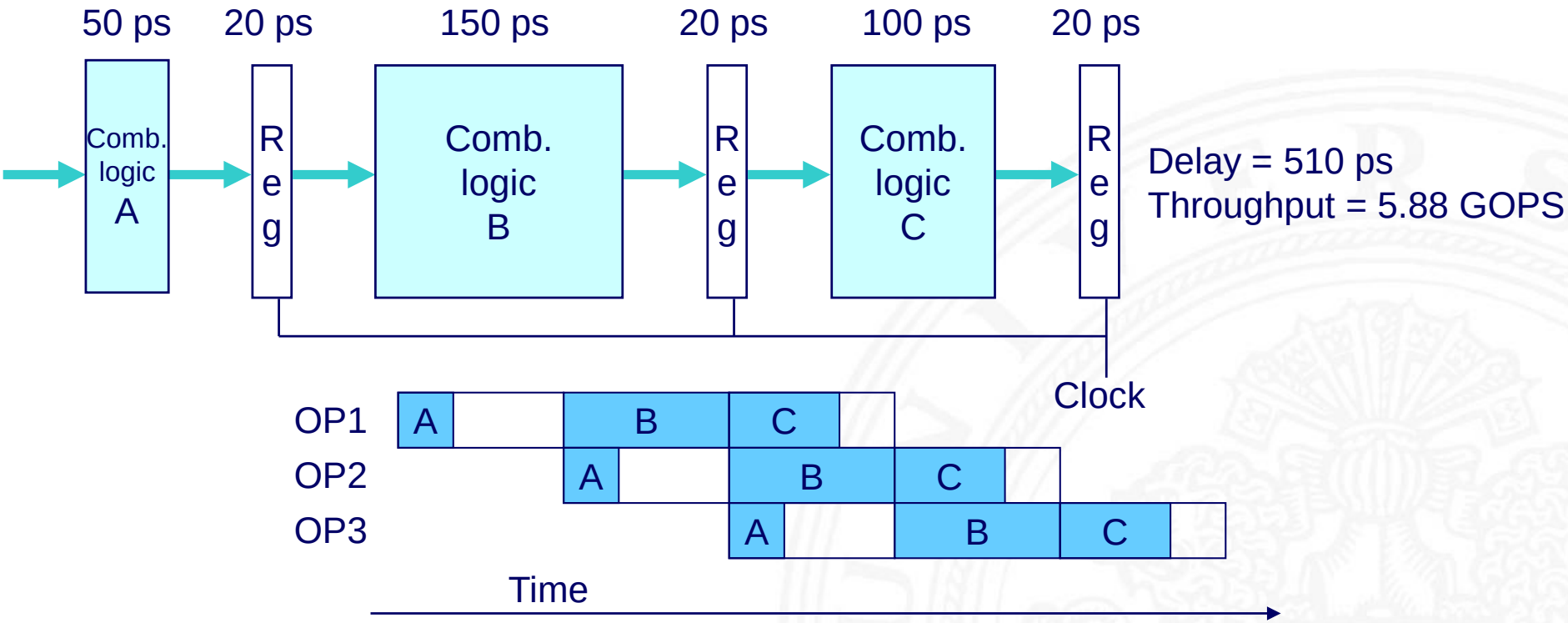


Beispiel: 3-stufige Pipeline



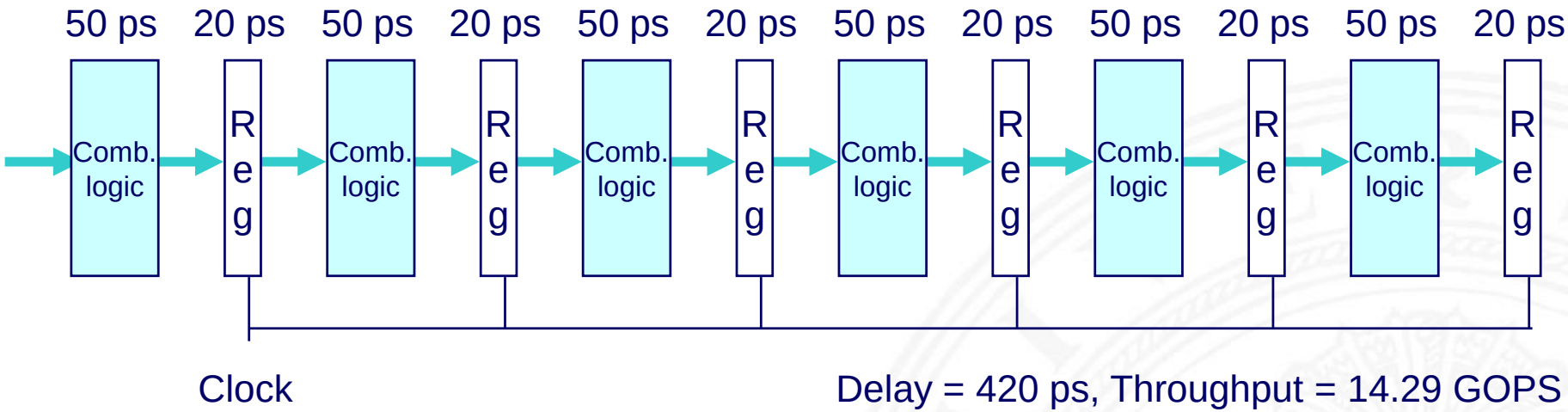


Probleme: nichtuniforme Verzögerungen



- größte Verzögerung bestimmt Taktfrequenz

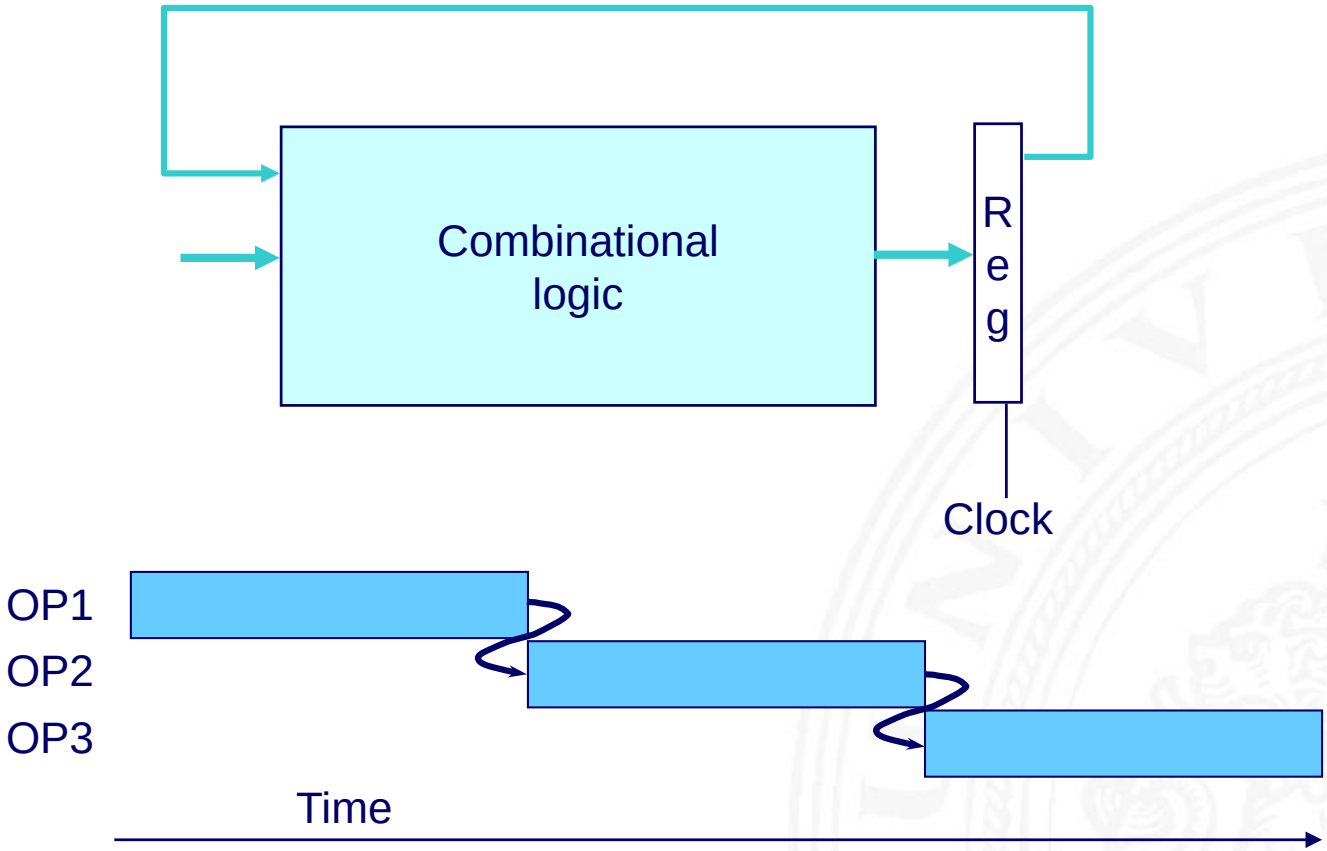
Probleme: Register „Overhead“



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

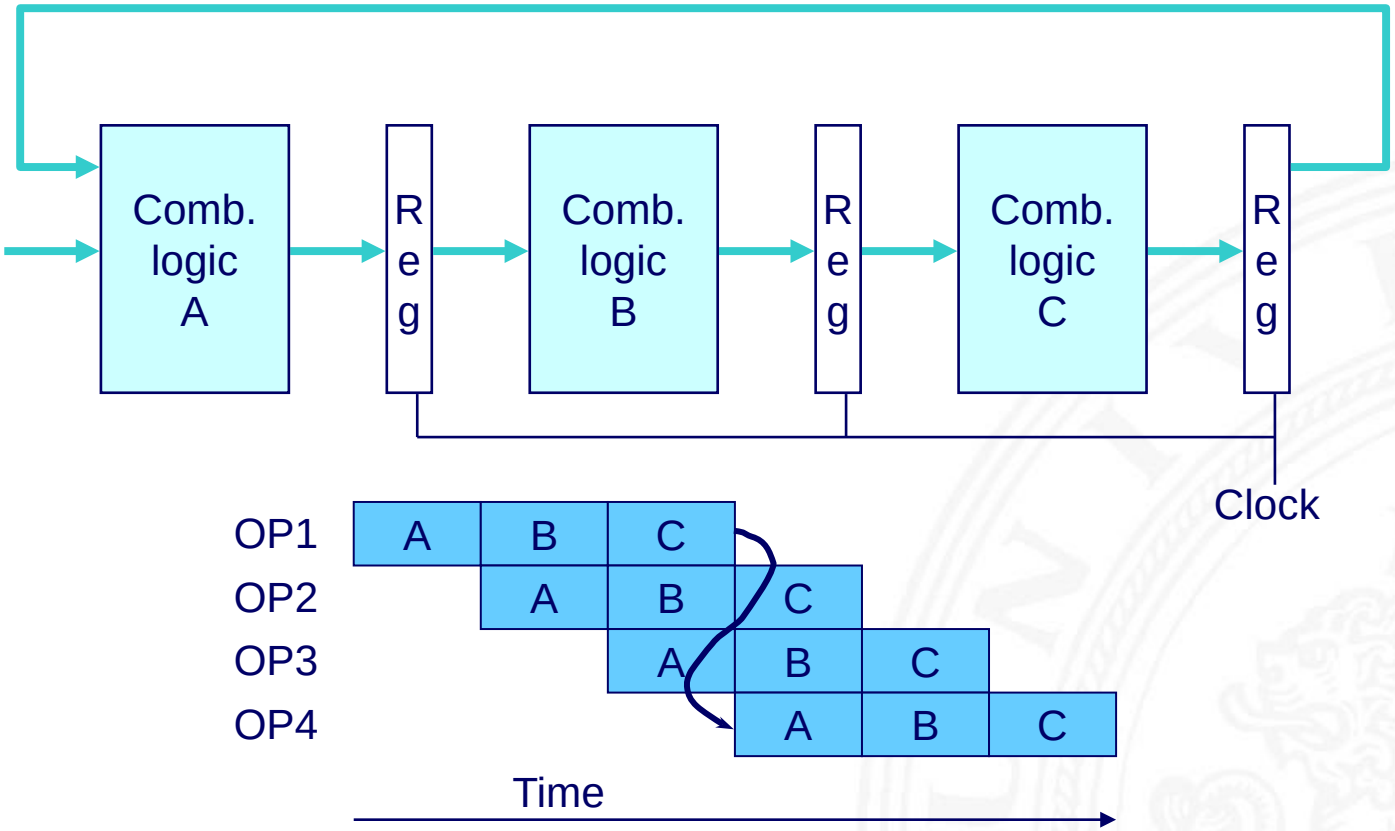
	Overhead		Taktperiode
1-Register:	6,25%	20 ps	320 ps
3-Register:	16,67%	20 ps	120 ps
6-Register:	28,57%	20 ps	70 ps

Probleme: Datenabhängigkeiten / „Daten Hazards“



- jede Operation hängt vom Ergebnis der Vorhergehenden ab

Probleme: Datenabhängigkeiten / „Daten Hazards“ (cont.)



⇒ Resultat-Feedback kommt zu spät für die nächste Operation

⇒ Pipelining ändert Verhalten des gesamten Systems

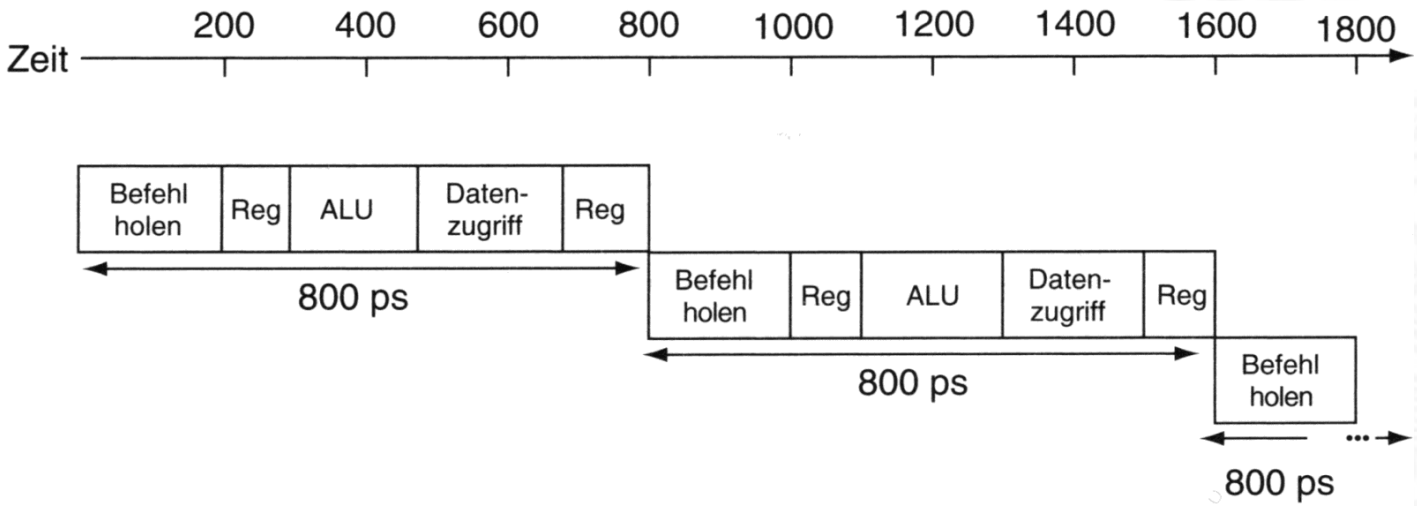
RISC Pipelining

Schritte der RISC 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 — **Ex**ecute
  ALU führt Befehl aus
- MEM — **Mem**ory access
  Speicherzugriff bei Load-/Store-Befehlen
- WB — **W**rite **B**ack
  Ergebnisse in Register zurückschreiben

RISC Pipelining (cont.)

- je nach Instruktion sind 3-5 dieser Schritte notwendig
- Beispiel *ohne* Pipelining:

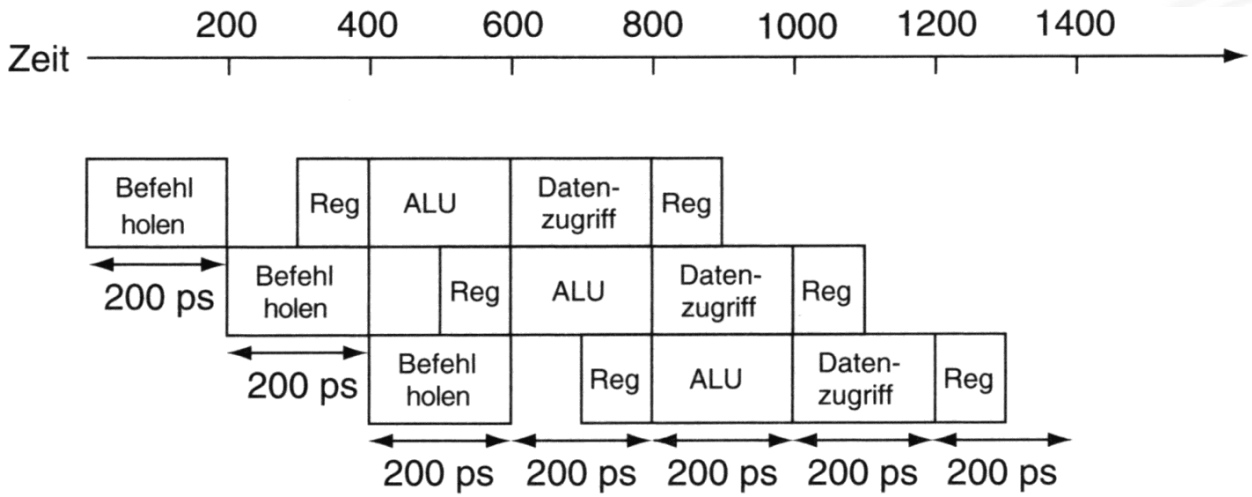


Patterson, Hennessy, *Computer Organization and Design*

RISC Pipelining (cont.)

Pipelining in Prozessoren

- Beispiel *mit* Pipelining:



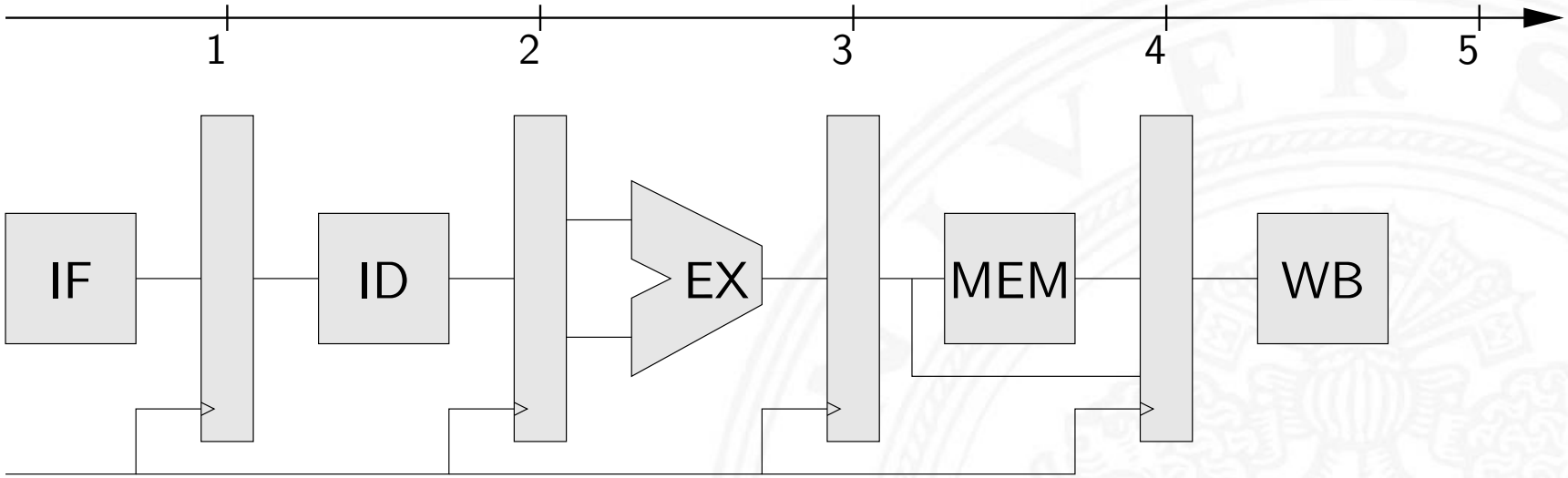
Patterson, Hennessy, *Computer Organization and Design*

- Befehle überlappend ausführen
- Register trennen Pipelinestufen

RISC Pipelining (cont.)

- RISC ISA: Pipelining wird direkt umgesetzt

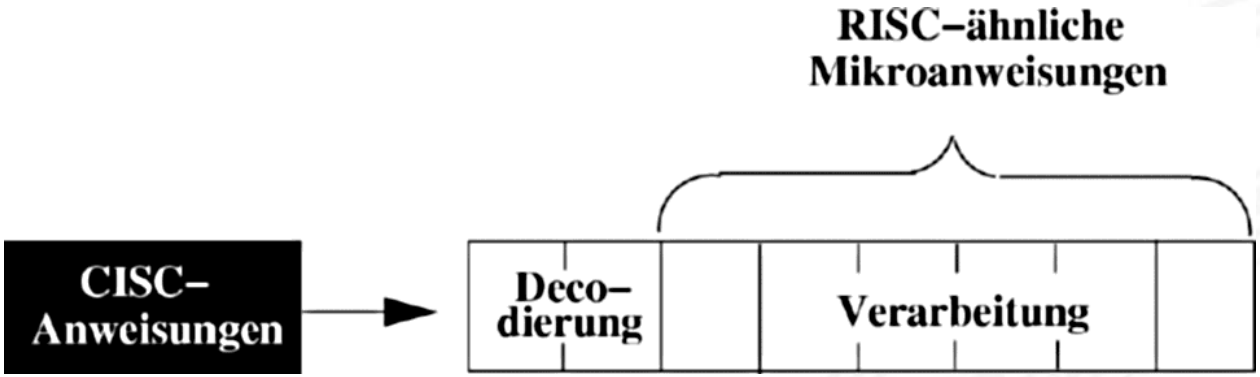
  Pipelinestufen



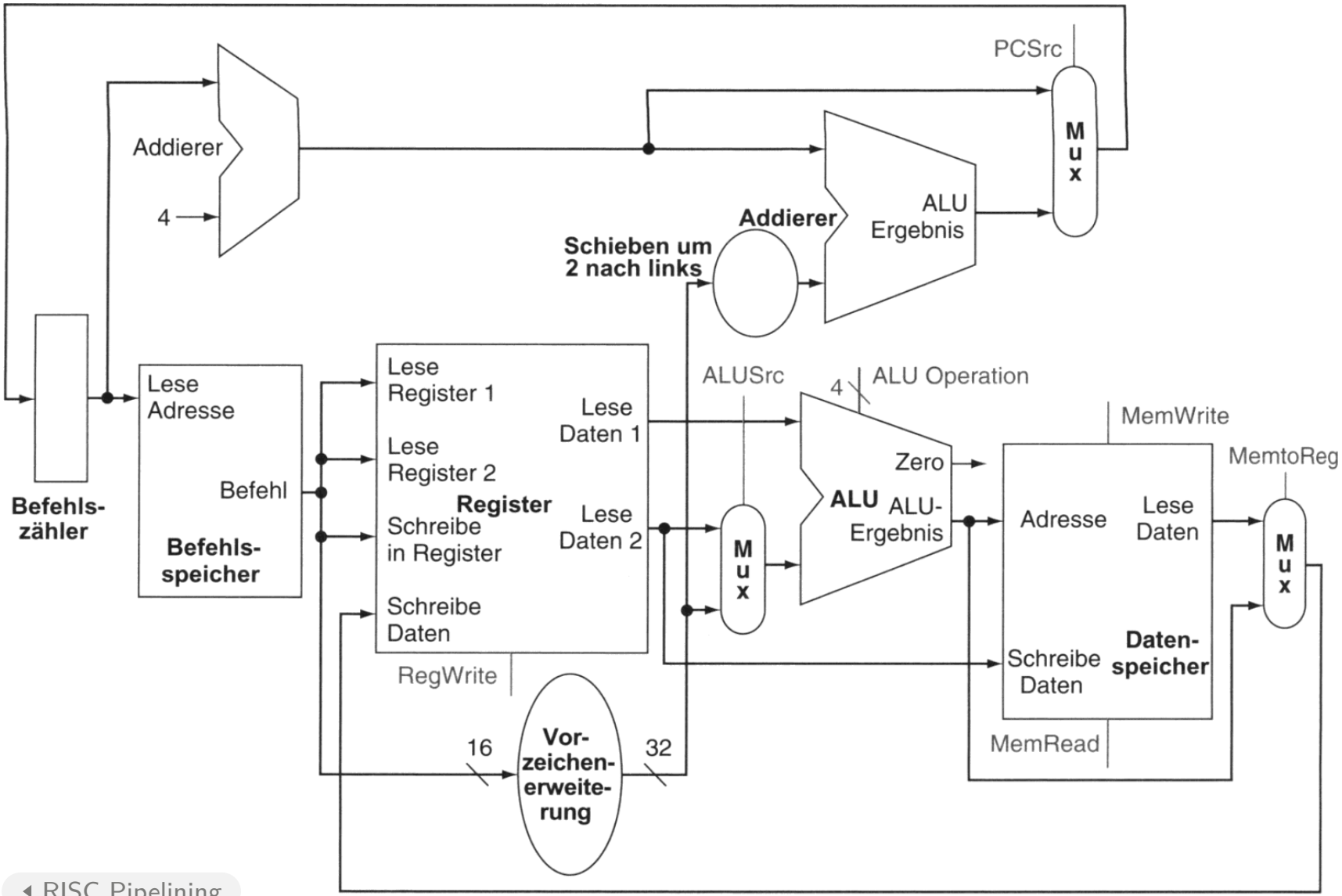
- MIPS-Architektur (aus Patterson, Hennessy)
  - MIPS ohne Pipeline
  - MIPS Pipeline
  - Pipeline Schema
- Bryant, O'Hallaron, *Computer systems*
  - Pipeline – Register
  - Pipeline – Architektur

RISC Pipelining (cont.)

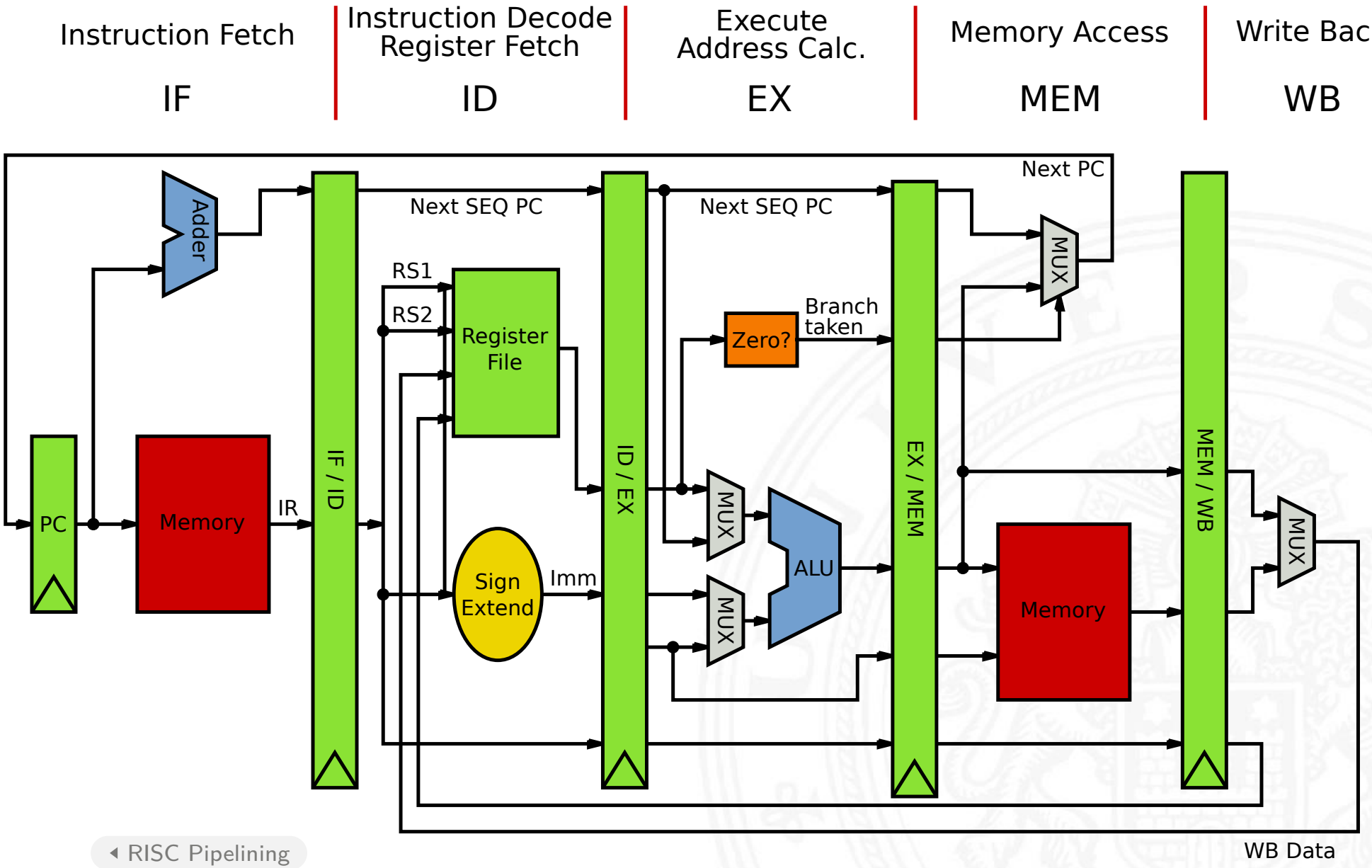
- CISC ISA (x86): Umsetzung der CISC Befehle in Folgen RISC-ähnlicher Anweisungen

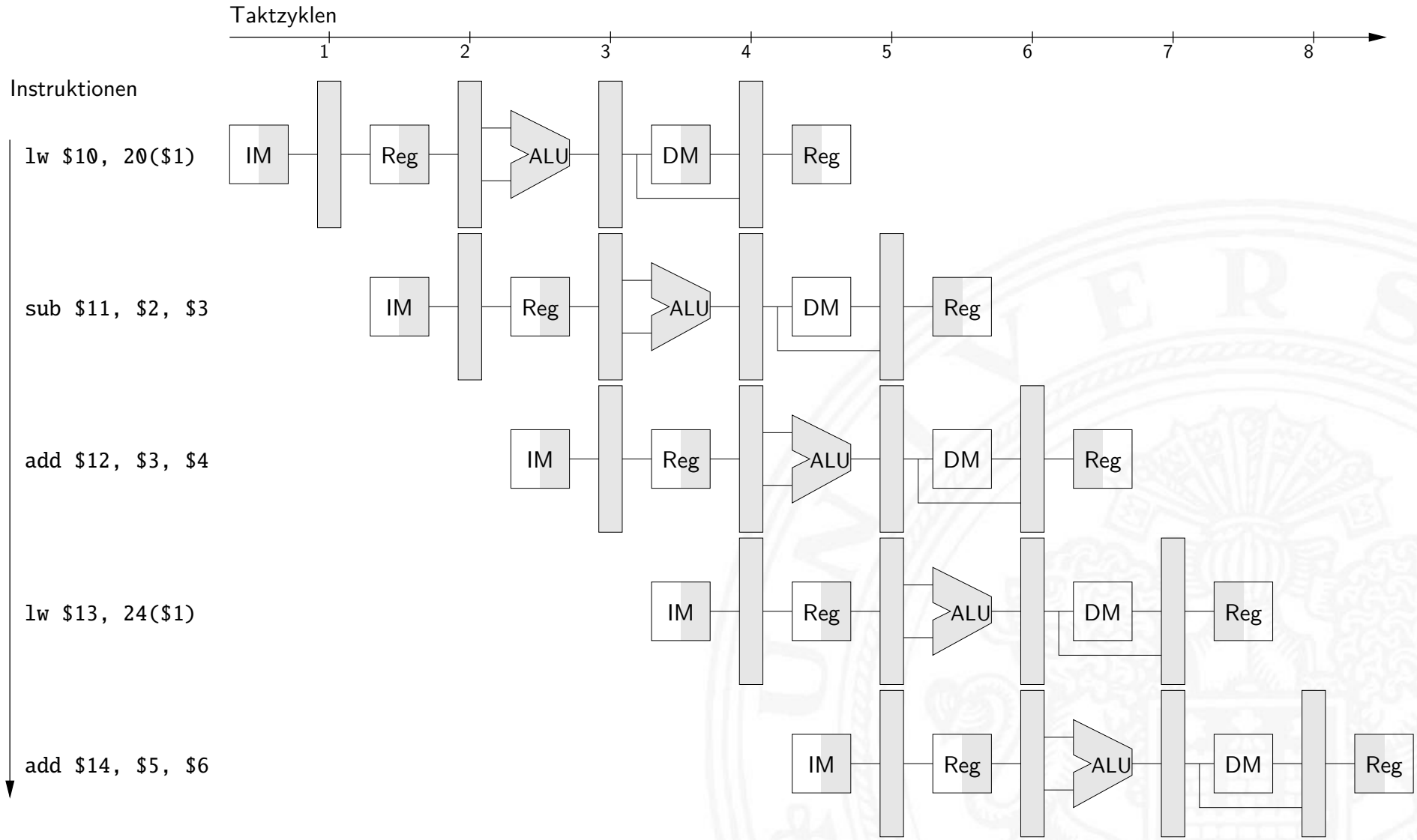


+ CISC-Software bleibt lauffähig
+ Befehlssatz wird um neue RISC Befehle erweitert

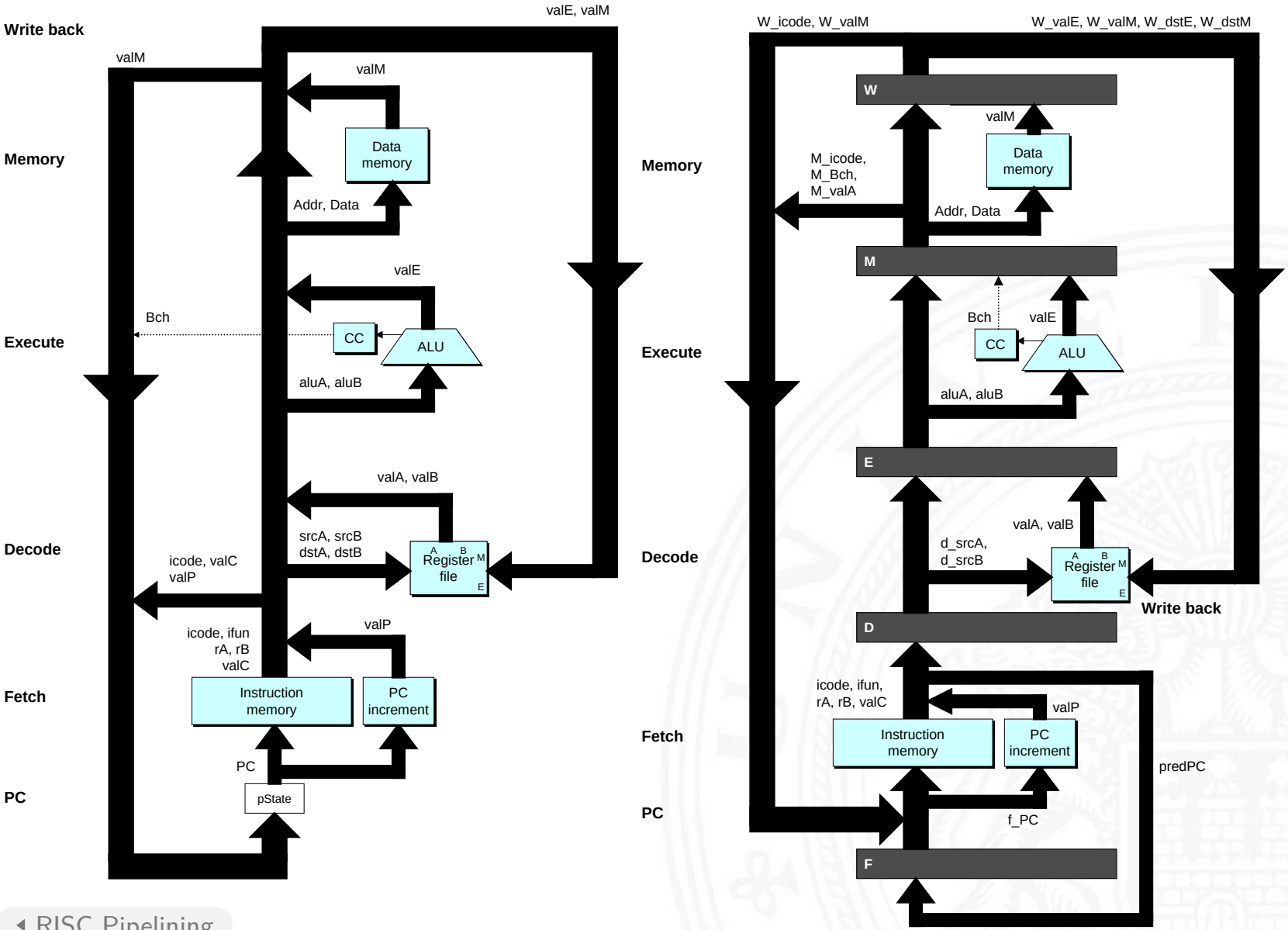


RISC Pipelining

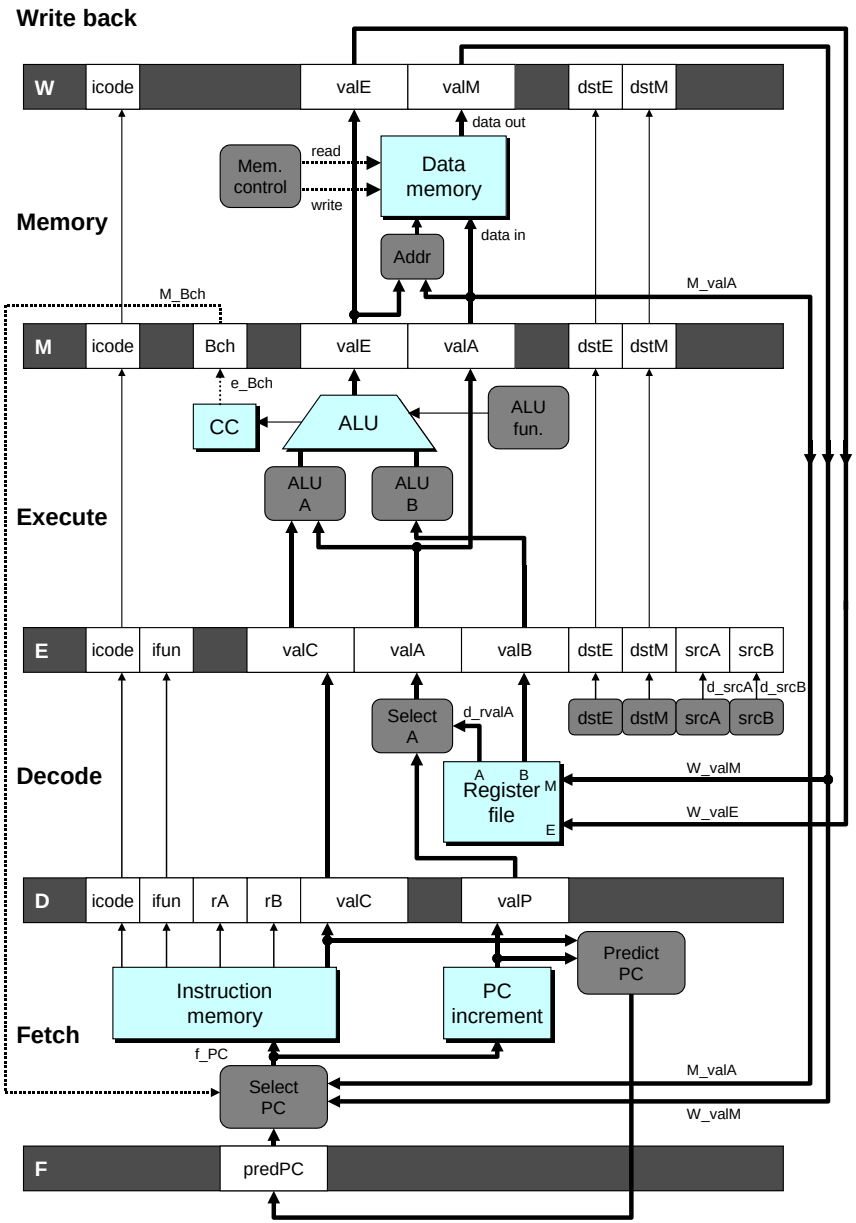

Instruction Fetch
IF
Instruction Decode
Register Fetch
ID
Execute
Address Calc.
EX
Memory Access
MEM
Write Back
WB
Adder
Next SEQ PC
Next SEQ PC
Next PC
MUX
RS1
RS2
Register
File
Zero?
Branch
taken
PC
Memory
IR
IF / ID
ID / EX
EX / MEM
MEM / WB
MUX
MUX
ALU
MUX
Sign
Extend
Imm
Memory
WB Data



◂ RISC Pipelining



RISC Pipelining


Write back
W
icode
valE
valM
dstE
dstM
data out
Mem. control
read
write
Data memory
Memory
data in
Addr
M_valA
M_Bch
M
icode
Bch
valE
valA
dstE
dstM
e_Bch
CC
ALU
ALU fun.
ALU A
ALU B
Execute
E
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
Select A
d_rvalA
dstE
dstM
srcA
srcB
W_valM
A
B
Register file
M
E
W_valE
Decode
D
icode
ifun
rA
rB
valC
valP
Predict PC
Instruction memory
PC increment
Fetch
f_PC
M_valA
Select PC
W_valM
F
predPC

Prozessorpipeline – Begriffe

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

Prozessorpipeline – Bewertung

Vor- und Nachteile

+ Pipelining ist für den Programmierer nicht sichtbar!
+ höherer Instruktionsdurchsatz $\Rightarrow$ bessere Performanz
– 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

Prozessorpipeline – Speed-Up

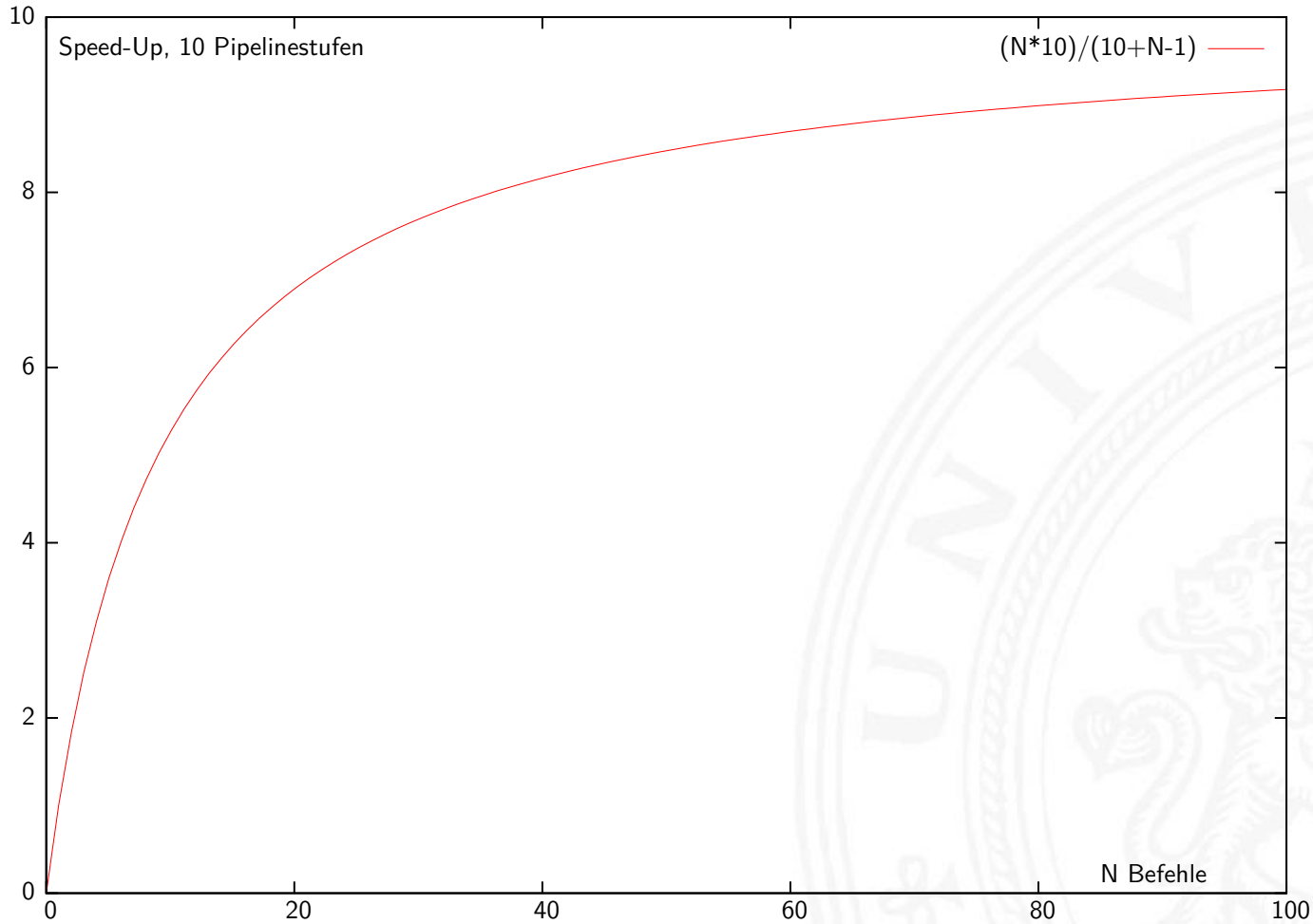
Pipeline Speed-Up

- N Instruktionen; K Pipelinestufen
- ohne Pipeline: $N \cdot K$ Taktzyklen
- mit Pipeline: $K + N - 1$ Taktzyklen
- Speed-Up $= \frac{N \cdot K}{K+N-1}$, $\lim_{N \to \infty} S = K$

$\Rightarrow$ ein großer Speed-Up wird erreicht durch

1. große Pipelinetiefe: K
2. lange Instruktionssequenzen: N

Prozessorpipeline – Speed-Up (cont.)



Prozessorpipeline – Dimensionierung

Dimensionierung der Pipeline

- Längere Pipelines
- Pipelinestufen in den Einheiten / den ALUs (*superskalar*)

$\Rightarrow$ größeres K wirkt sich direkt auf den Durchsatz aus

$\Rightarrow$ weniger Logik zwischen den Registern, höhere Taktfrequenzen

- Beispiele

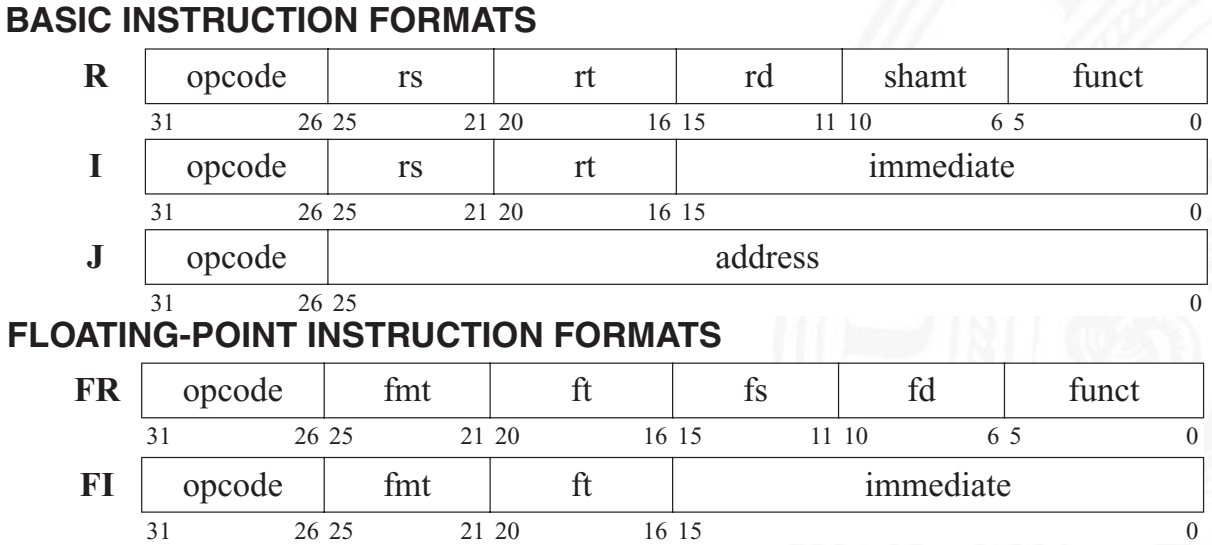
CPU	Pipelinestufen	Taktfrequenz [*MHz*]
Pentium	5	300
Motorola G4	4	500
Motorola G4e	7	1000
Pentium II/III	12	1400
Athlon XP	10/15	2500
Athlon 64, Opteron	12/17	$\leq$ 3000
Pentium 4	20	$\leq$ 5000

Prozessorpipeline – Auswirkungen

Architekturentscheidungen, die sich auf das Pipelining auswirken

gut für Pipelining

- gleiche Instruktionslänge
- wenige Instruktionsformate
- Load/Store Architektur



MIPS-Befehlsformate

Prozessorpipeline – Auswirkungen (cont.)

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.

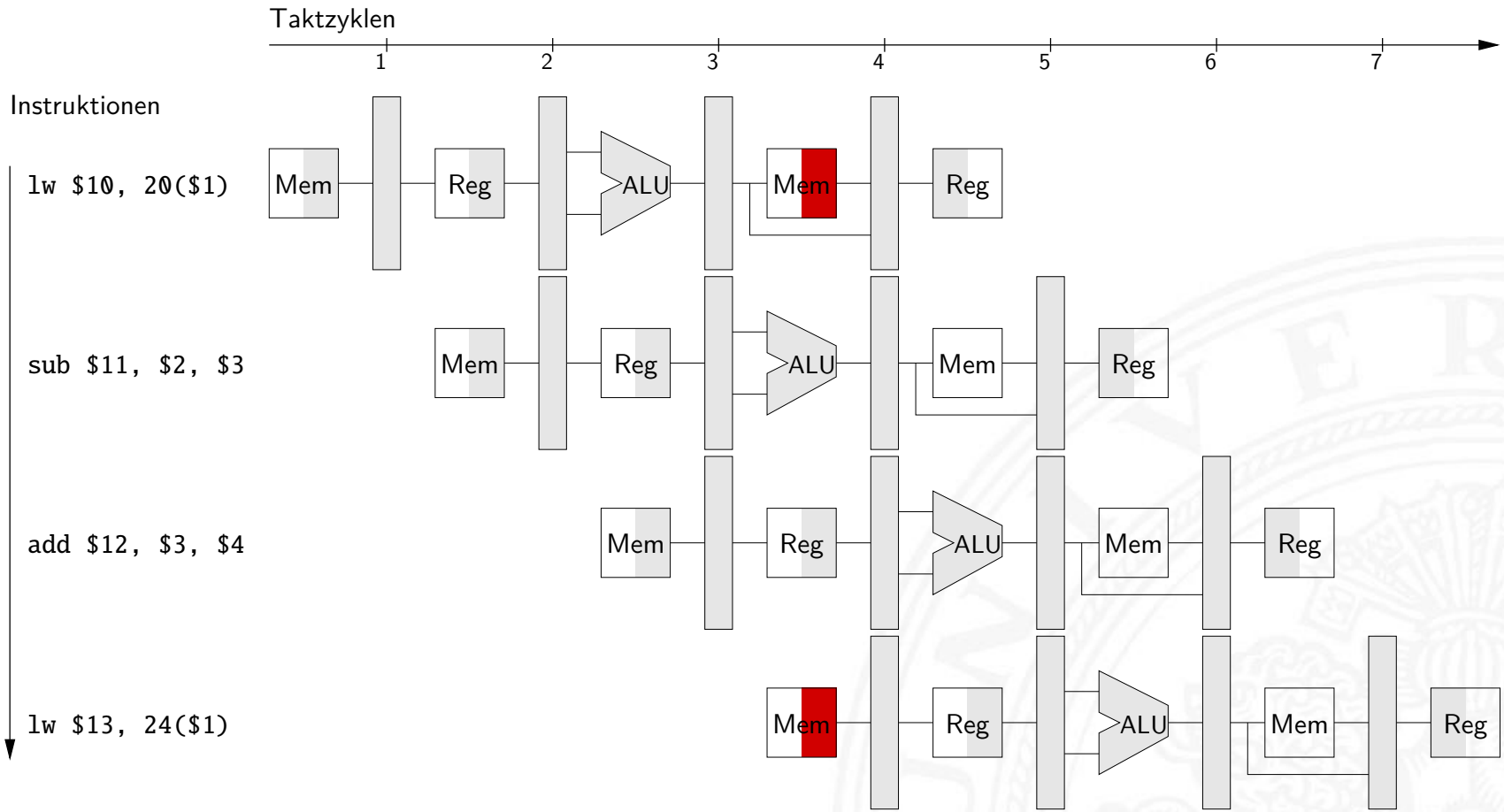
Pipeline Strukturkonflikte

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...



◂ Strukturkonflikte

Pipeline Datenkonflikte

Datenkonflikt / Data Hazard

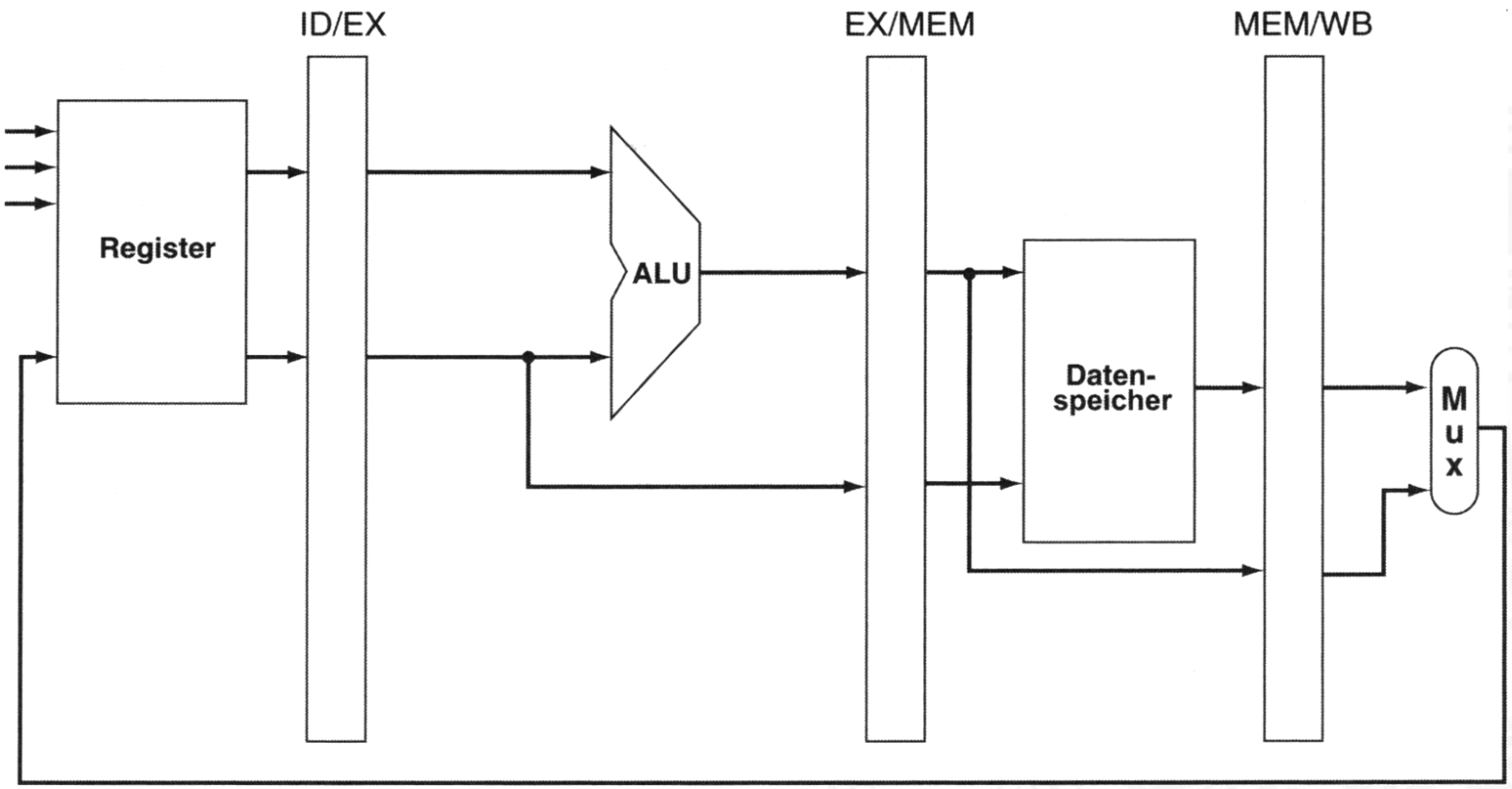
- eine Instruktion braucht die Ergebnisse einer vorhergehenden, diese wird aber noch in der Pipeline bearbeitet
- Datenabhängigkeiten der Stufe „Befehl ausführen“

▸ Beispiel

Forwarding

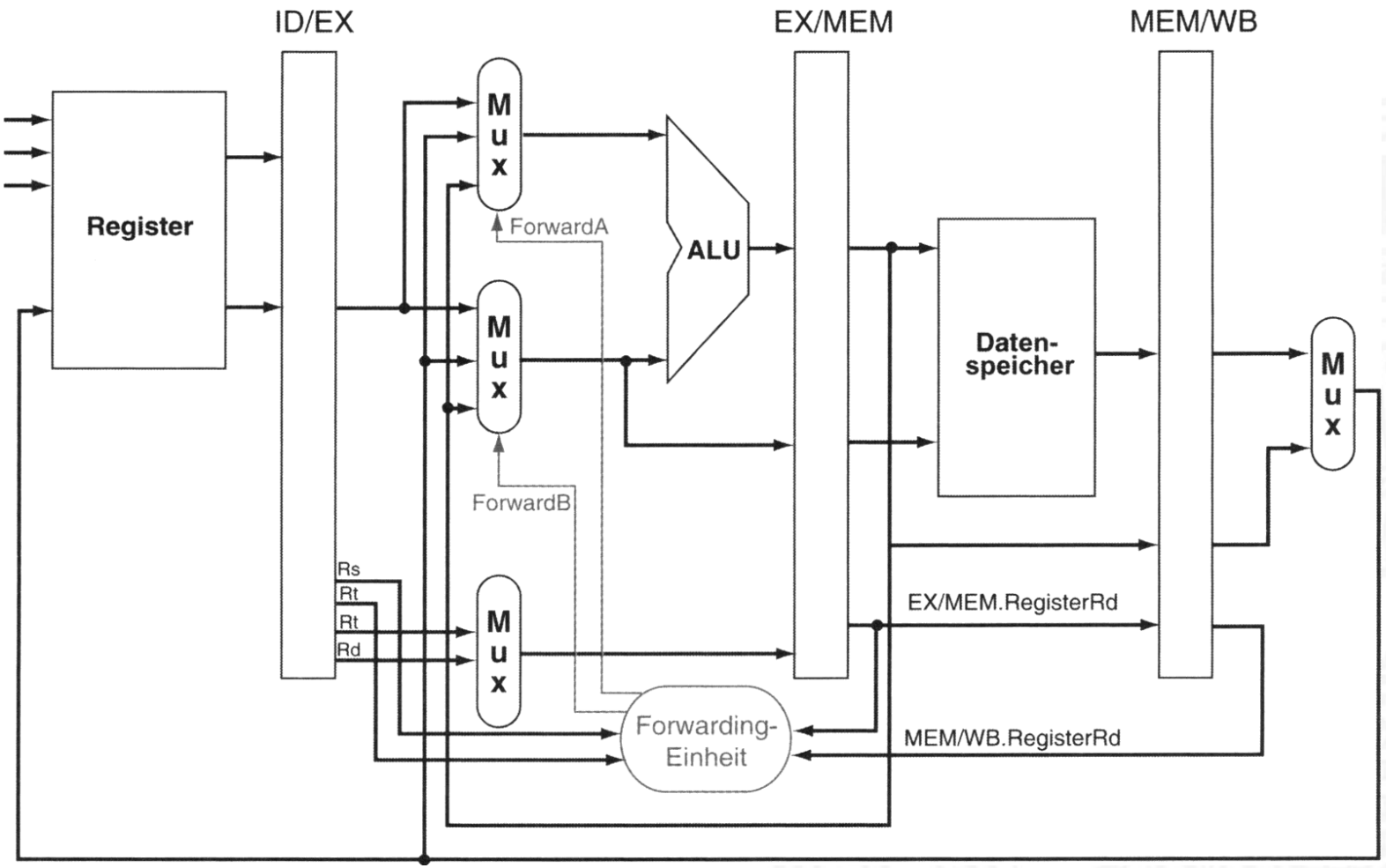
- kann Datenabhängigkeiten auflösen, s. Beispiel
- extra Hardware: „*Forwarding-Unit*“
- Änderungen in der Pipeline Steuerung
- neue Datenpfade und Multiplexer

Pipeline Datenkonflikte (cont.)



MIPS
Datenpfad

Pipeline Datenkonflikte (cont.)



MIPS Forwarding

Pipeline Datenkonflikte (cont.)

Rückwärtsabhängigkeiten

- spezielle Datenabhängigkeit ▸ Beispiel
- Forwarding-Technik funktioniert nicht, da die Daten erst *später* zur Vefügung 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

Pipeline Datenkonflikte (cont.)

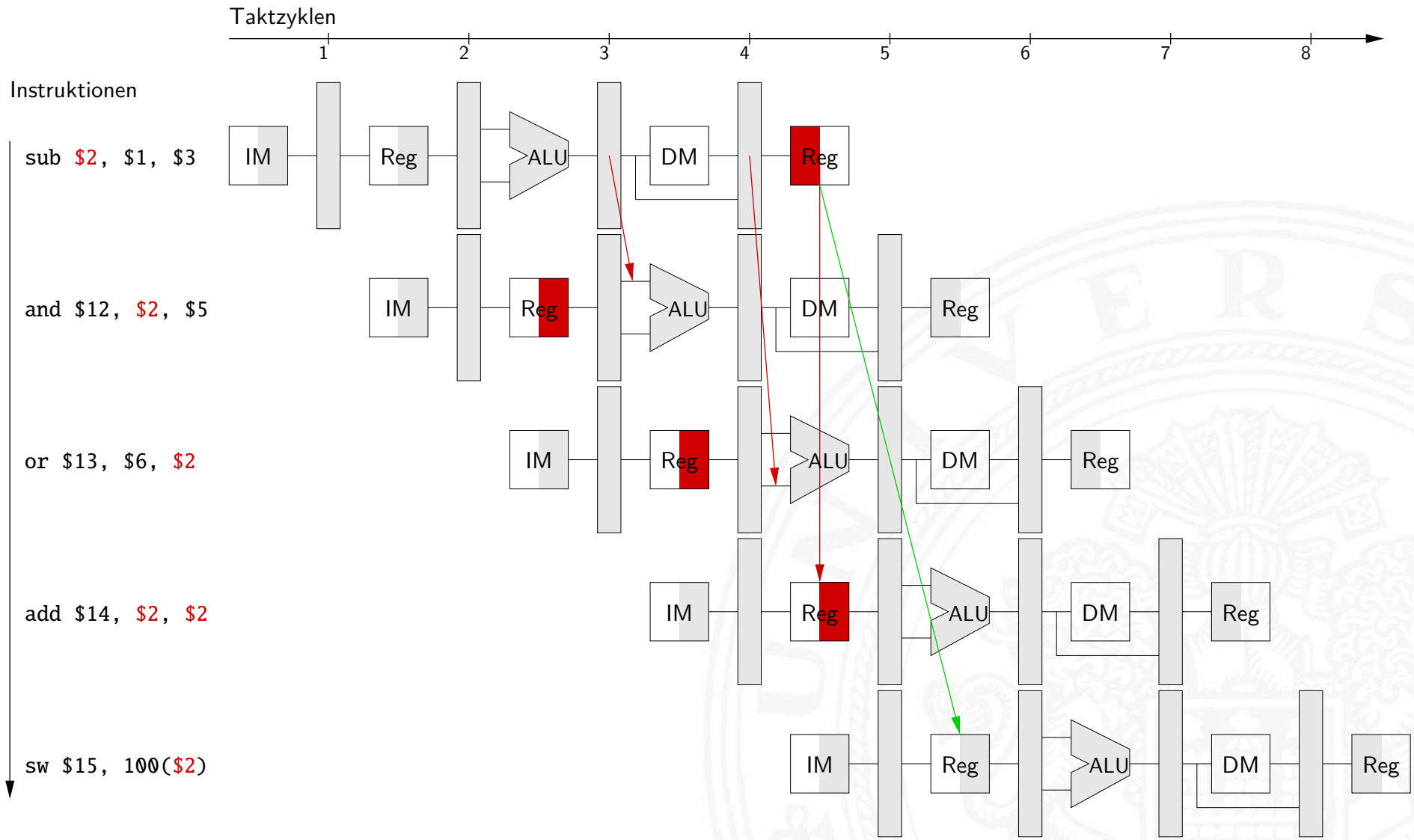
2. *„Interlocking“*

▸ Beispiel

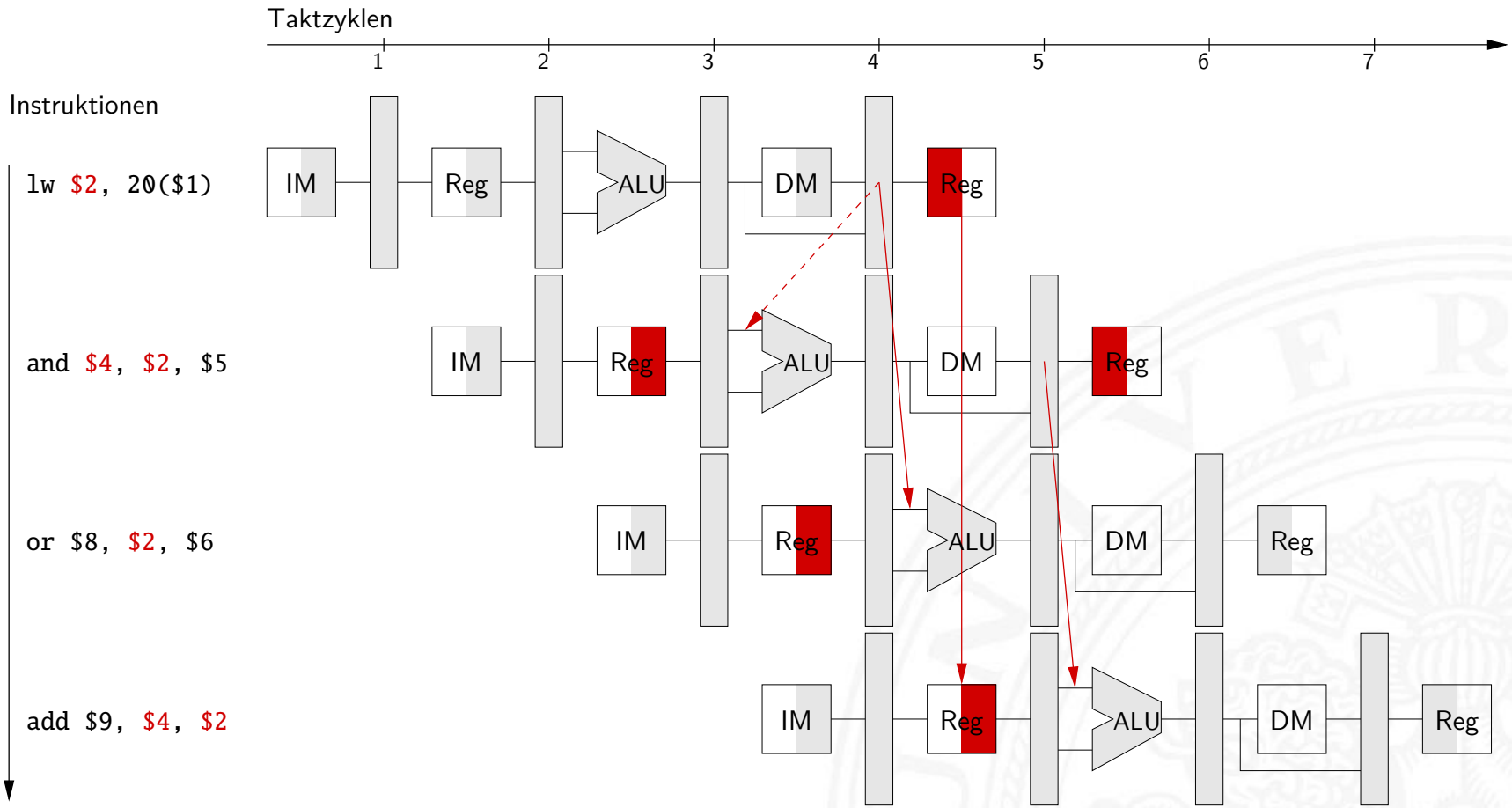
   - zusätzliche (Hardware) Kontrolleinheit
   - verschiedene Strategien
   - in Pipeline werden keine neuen Instruktionen geladen
   - Hardware erzeugt: Pipelineleerlauf / *„pipeline stall“*

„Scoreboard“

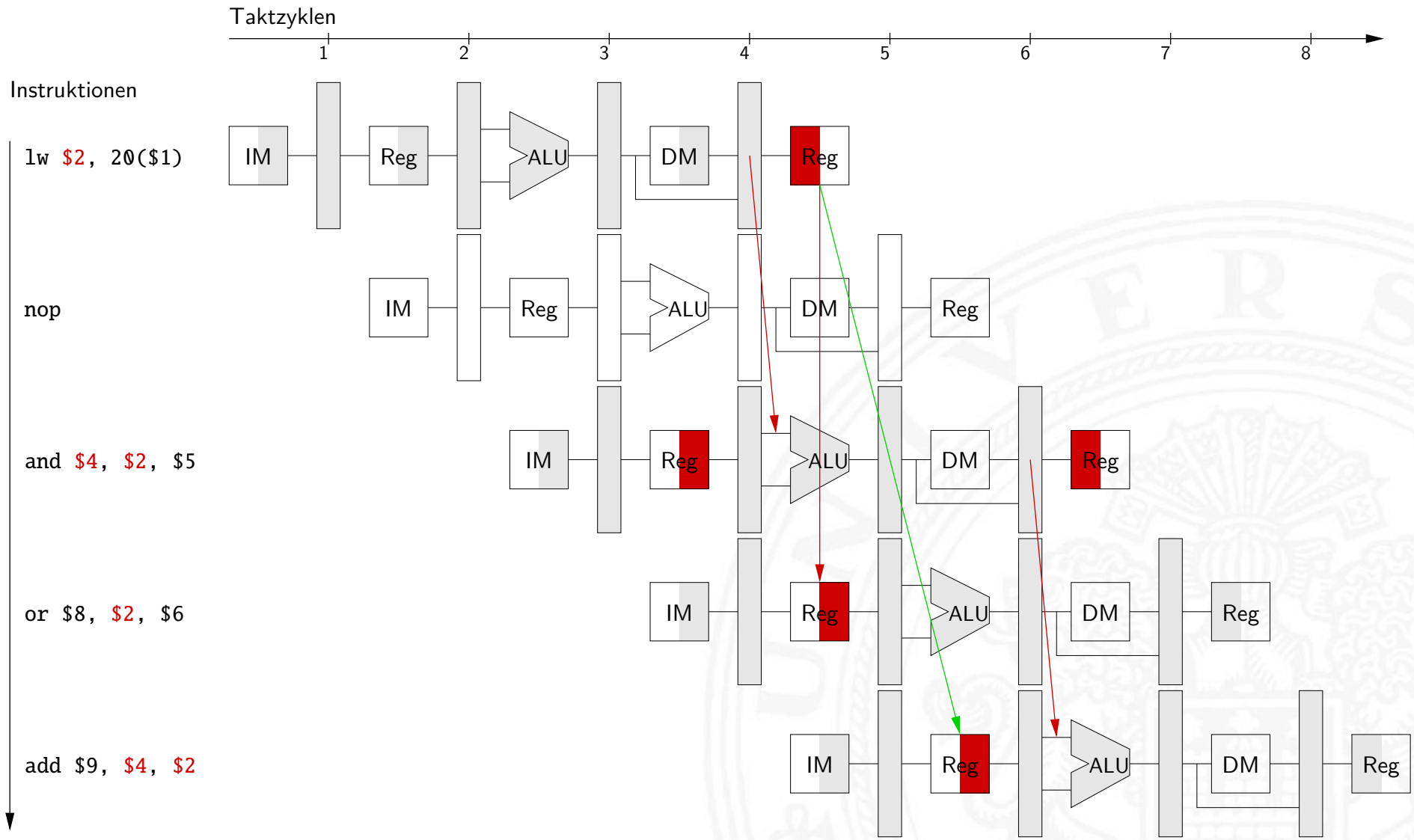
- Hardware Einheit zur zentralen Hazard-Erkennung und -Auflösung
- Verwaltet Instruktionen, benutzte Einheiten und Register der Pipeline



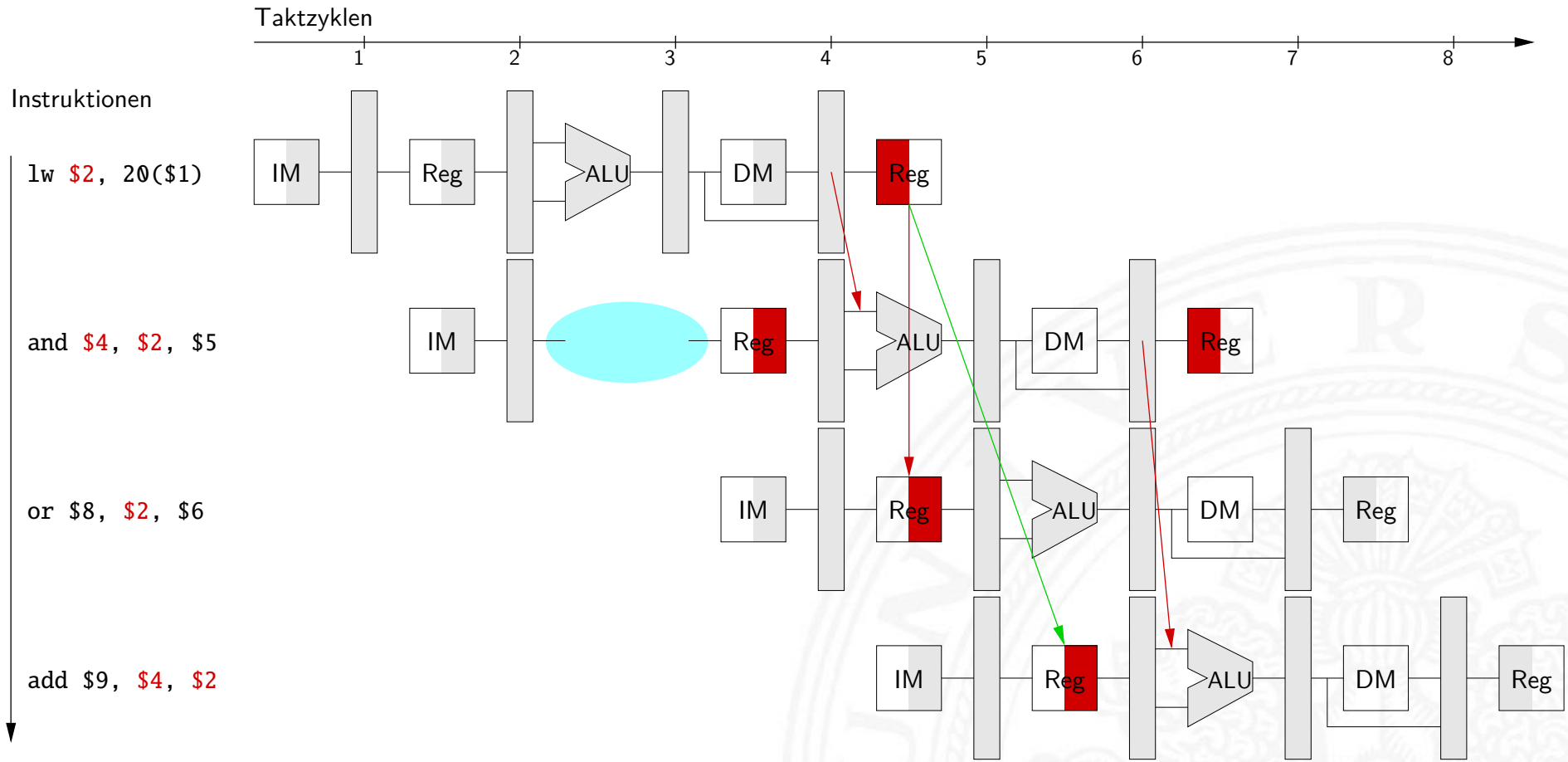
◂ Datenkonflikte



◂ Datenkonflikte



◂ Datenkonflikte



◂ Datenkonflikte

Pipeline Steuerkonflikte

Steuerkonflikt / Control Hazard

- Sprungbefehle unterbrechen den sequenziellen Ablauf der Instruktionen
- Problem: Instruktionen die auf (bedingte) Sprünge folgen, werden in die Pipeline geschoben, bevor bekannt ist, ob verzweigt werden soll
- Beispiel: bedingter Sprung

▸ Beispiel

Pipeline Steuerkonflikte (cont.)

Lösungsmöglichkeiten für Steuerkonflikte

- ad-hoc Lösung: „Interlocking“ erzeugt Pipelineleerlauf
  - – ineffizient: ca. 19 % der Befehle sind Sprünge

1. Annahme: nicht ausgeführter Sprung / „untaken branch“
   - \+ kaum zusätzliche Hardware
   - – im Fehlerfall
     - Pipelineleerlauf
     - 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)

Pipeline Steuerkonflikte (cont.)

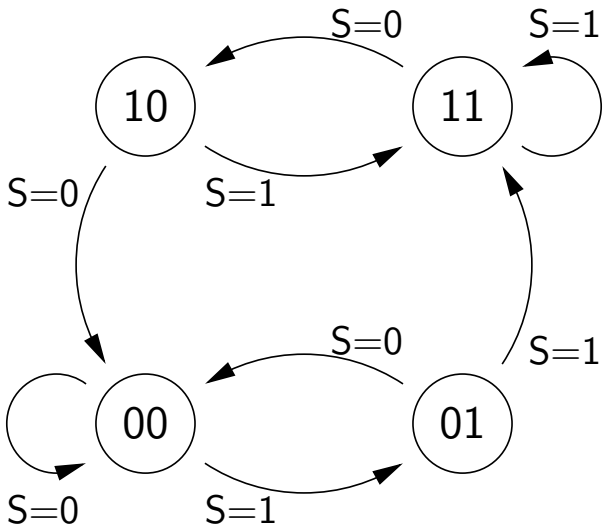
3. Sprungvorhersage / *„branch prediction“*
   - Beobachtung: ein Fall tritt häufiger auf:
     Schleifendurchlauf, Datenstrukturen durchsuchen etc.
   - mehrere Vorhersageverfahren; oft miteinander kombiniert
   + hohe Trefferquote: bis 90 %

   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...

Pipeline Steuerkonflikte (cont.)

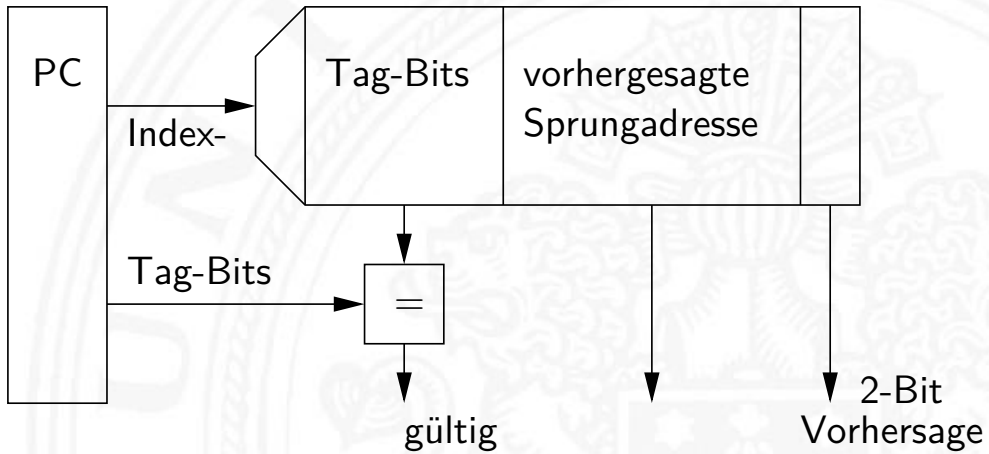
- Beispiel: 2-Bit Sprungvorhersage + Branch Target Cache



VH **V**orhersage bit

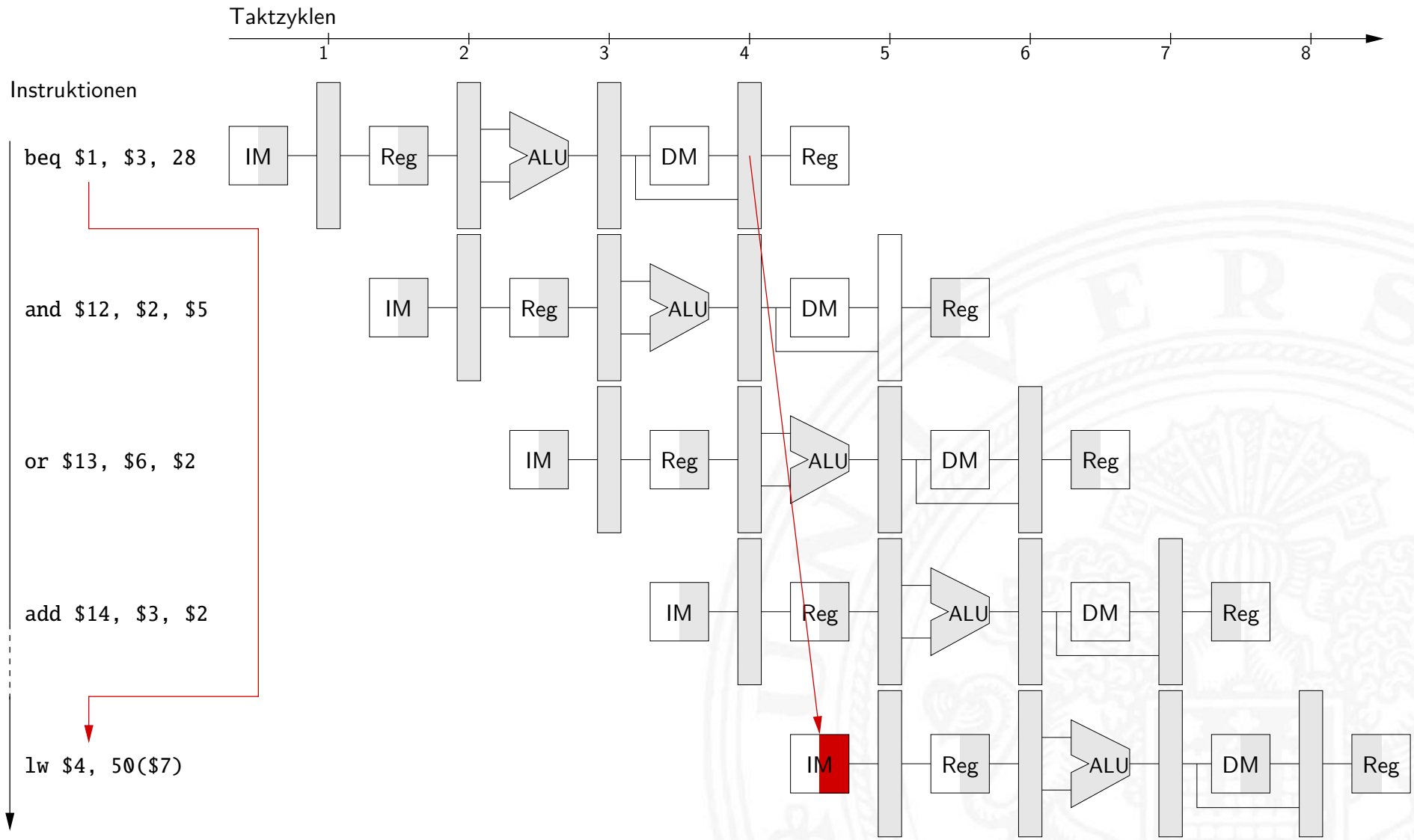
Historie bit

S=0/1 **S**prung ausgeführt



Pipeline Steuerkonflikte (cont.)

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



◂ Steuerkonflikte

Superskalare Prozessoren

- Superskalare CPUs besitzen mehrere Recheneinheiten: 4...10
- In jedem Takt werden (dynamisch) mehrere Instruktionen eines konventionell linearen Instruktionsstroms abgearbeitet: $CPI < 1$
  ILP (**I**nstruction **L**evel **P**arallelism) ausnutzen!
- Hardware verteilt initiierte Instruktionen auf Recheneinheiten
- Pro Takt kann *mehr als eine* Instruktion initiiert werden
  Die Anzahl wird dynamisch von der Hardware bestimmt:
  0...„*Instruction Issue Bandwidth*“

\+ sehr effizient, alle modernen CPUs sind superskalar

– Abhängigkeiten zwischen Instruktionen sind der Engpass, das Problem der Hazards wird verschärft

Superskalar – Datenabhängigkeiten

Datenabhängigkeiten

- RAW – **R**ead **A**fter **W**rite
  Instruktion I_x darf Datum erst lesen, wenn I_{x-n} geschrieben hat
- WAR – **W**rite **A**fter **R**ead
  Instruktion I_x darf Datum erst schreiben, wenn I_{x-n} gelesen hat
- WAW – **W**rite **A**fter **W**rite
  Instruktion I_x darf Datum erst überschreiben, wenn I_{x-n} geschrieben hat

Superskalar – Datenabhängigkeiten (cont.)

Datenabhängigkeiten superskalarer Prozessoren

- RAW: echte Abhängigkeit; Forwarding ist kaum möglich und in superskalaren Pipelines extrem aufwändig
- WAR, WAW: „*Register Renaming*“ als Lösung

„*Register Renaming*“

- Hardware löst Datenabhängigkeiten innerhalb der Pipeline auf
- Zwei Registersätze sind vorhanden
  1. Architektur-Register: „logische Register“ der ISA
  2. viele Hardware-Register: „Rename Register“
     - dynamische Abbildung von ISA- auf Hardware-Register

Superskalar – Datenabhängigkeiten (cont.)

- Beispiel

Originalcode

```
tmp  = a + b;
res1 = c + tmp;
tmp  = d + e;
res2 = tmp - f;
```

nach Renaming

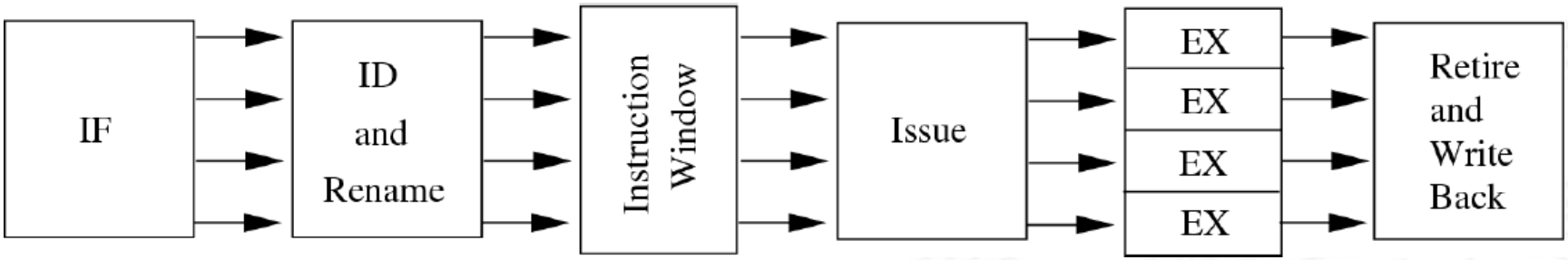
```
tmp1 = a + b;
res1 = c + tmp1;
tmp2 = d + e;
res2 = tmp2 - f;
tmp  = tmp2;
```

Parallelisierung des modifizierten Codes

```
tmp1 = a + b;      tmp2 = d + e;
res1 = c + tmp1;   res2 = tmp2 - f;   tmp = tmp2;
```

Superskalar – Pipeline

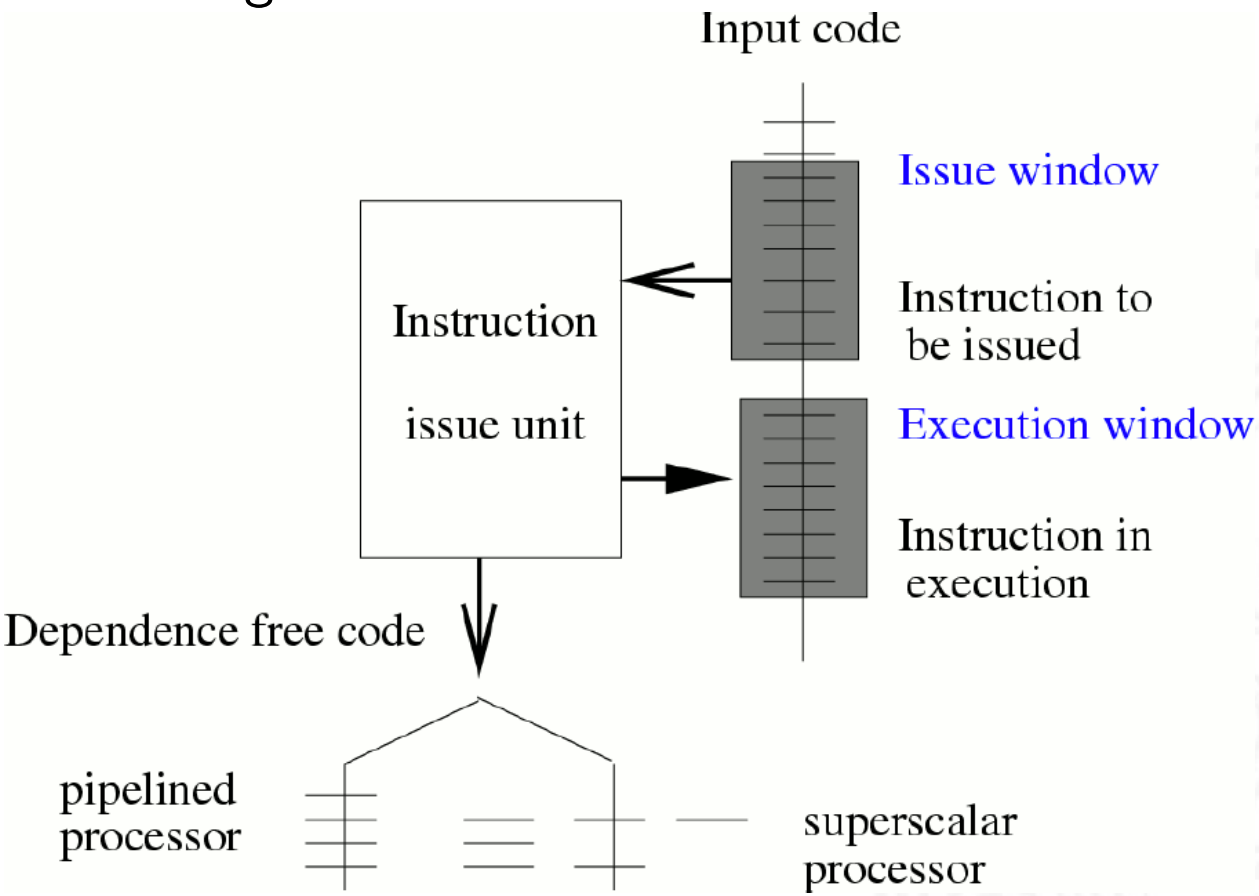
Aufbau der superskalaren Pipeline



- lange Pipelines mit vielen Phasen: Fetch (Prefetch, Predecode), Decode / Register-Renaming, Issue, Dispatch, Execute, Retire (Commit, Complete / Reorder), Write-Back
- je nach Implementation unterschiedlich aufgeteilt
- entscheidend für superskalare Architektur sind die Schritte
  vor den ALUs: Issue, Dispatch $\Rightarrow$ *out-of-order* Ausführung
  nach -"- : Retire $\Rightarrow$ *in-order* Ergebnisse

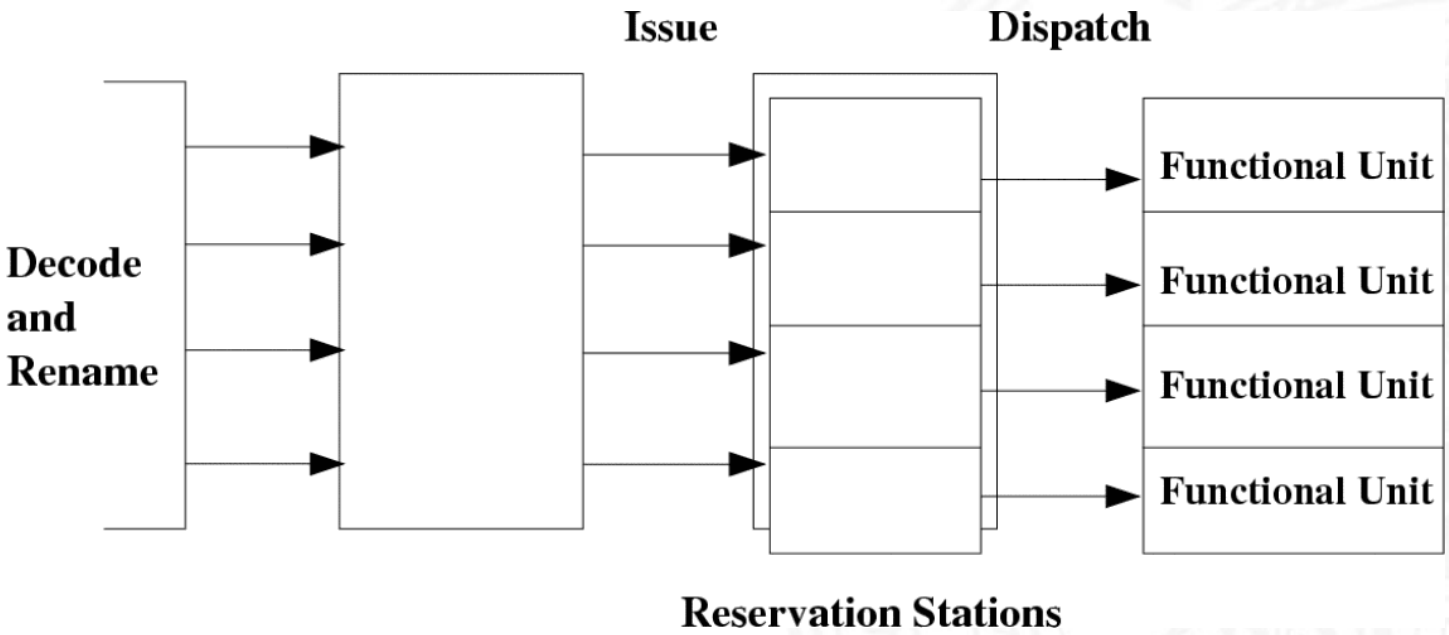
Superskalar – Pipeline (cont.)

Scheduling der Instruktionen



Superskalar – Pipeline (cont.)

- Dynamisches Scheduling erzeugt *out-of-order* Reihenfolge der Instruktionen
- Issue: globale Sicht
  Dispatch: getrennte Ausschnitte in „Reservation Stations“



Superskalar – Pipeline (cont.)

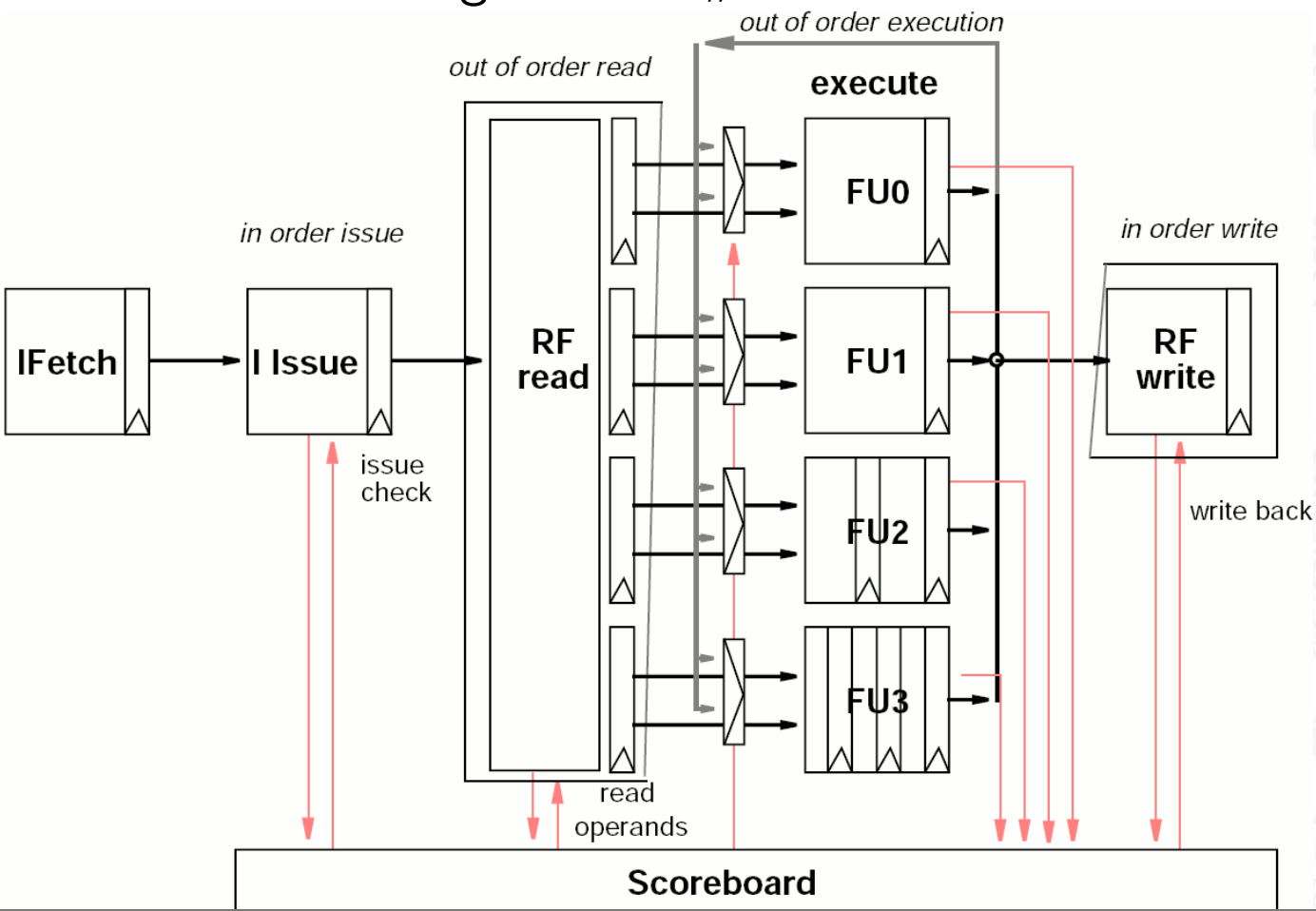
Reservation Station für jede Funktionseinheit

- speichert: initiierte Instruktionen die auf Recheneinheit warten
- –"– zugehörige Operanden
- –"– ggf. Zusatzinformation
- Instruktion bleibt blockiert, bis alle Parameter bekannt sind und wird dann an die zugehörige ALU weitergeleitet

- Dynamisches Scheduling: zuerst '67 in IBM 360 (Robert Tomasulo)
  - Forwarding
  - Registerumbenennung und Reservation Stations

Superskalar – Scoreboard

Zentrale Verwaltungseinheit: „Scoreboard“



Superskalar – Scoreboard (cont.)

Scoreboard erlaubt das Management mehrerer Ausführungseinheiten

- out-of-order Ausführung von Mehrzyklusbefehlen
- Auflösung aller Struktur- und Datenkonflikte: RAW, WAW, WAR

Einschränkungen

- single issue (nicht superskalar)
- in-order issue
- keine Umbenennungen; also Leerzyklen bei WAR- und WAW-Konflikten
- kein Forwarding, daher Zeitverlust bei RAW-Konflikten

Superskalar – Retire-Stufe

„Retire“

- erzeugt wieder *in-order* Reihenfolge
- FIFO: Reorder-Buffer
- commit: „richtig ausgeführte“ Instruktionen gültig machen
- abort: Sprungvorhersage falsch
  Instruktionen verwerfen

Probleme superskalarer Pipelines

Spezielle Probleme superskalarer Pipelines

- weitere Hazard-Möglichkeiten
  - die verschiedenen ALUs haben unterschiedliche Latenzzeiten
  - Befehle „warten“ in den Reservation Stations

  $\Rightarrow$ Datenabhängigkeiten können sich mit jedem Takt ändern
- Kontrollflussabhängigkeiten: Anzahl der Instruktionen zwischen bedingten Sprüngen limitiert Anzahl parallelisierbarer Instruktion

$\Rightarrow$ „Loop Unrolling“ wichtig

$+$ optimiertes (dynamisches) Scheduling: Faktor 3 möglich

Software Pipelining

Softwareunterstützung für Pipelining superskalarer Prozessoren

- Codeoptimierungen beim Compilieren: Ersatz für, bzw. Ergänzend zu der Pipelieunterstützung durch Hardware
- Compiler hat „globalen“ Überblick
  $\Rightarrow$ zusätzliche Optimierungsmöglichkeiten
- symbolisches Loop Unrolling
- Loop Fusion
- ...

Superskalar – Interrupts

Exceptions, Interrupts und System-Calls

- Interruptbehandlung ist wegen der Vielzahl paralleler Aktionen und den Abhängigkeiten innerhalb der Pipelines extrem aufwändig
  - da unter Umständen noch Pipelineaktionen beendet werden müssen, wird *zusätzliche Zeit* bis zur Interruptbehandlung benötigt
  - wegen des Register-Renaming muss sehr viel *mehr Information* gerettet werden als nur die ISA-Register
- Prinzip der Interruptbehandlung
  - keine neuen Instruktionen mehr initiieren
  - warten bis Instruktionen des Reorder-Buffers abgeschlossen sind

Superskalar – Interrupts (cont.)

- Verfahren ist von der „Art“ des Interrupt abhängig
  - Precise-Interrupt: Pipelineaktivitäten komplett Beenden
  - Imprecise-Interrupt: wird als verzögerter Sprung (Delayed-Branching) in Pipeline eingebracht
    Zusätzliche Register speichern Information über Instruktionen die in der Pipeline nicht abgearbeitet werden können (z.B. weil sie den Interrupt ausgelöst haben)
- Definition: Precise-Interrupt
  - Programmzähler (PC) zur Interrupt auslösenden Instruktion ist bekannt
  - Alle Instruktionen bis zur PC-Instruktion wurden vollständig ausgeführt
  - Keine Instruktion nach der PC-Instruktion wurde ausgeführt
  - Ausführungszustand der PC-Instruktion ist bekannt

Ausnahmebehandlung

Ausnahmebehandlung („Exception Handling“)

- Pipeline kann normalen Ablauf nicht fortsetzen
- Ursachen
  - „Halt“ Anweisung
  - ungültige Adresse für Anweisung oder Daten
  - ungültige Anweisung
  - Pipeline Kontrollfehler
- erforderliches Vorgehen
  - einige Anweisungen vollenden
    Entweder aktuelle oder vorherige (hängt von Ausnahmetyp ab)
  - andere verwerfen
  - „Exception handler“ aufrufen: spez. Prozeduraufruf

Pentium 4 / NetBurst Architektur

- superskalare Architektur (mehrere ALUs)
- CISC-Befehle werden dynamisch in „μOPs“ (1...3) umgesetzt
- Ausführung der μOPs mit „Out of Order“ Maschine, wenn
  - Operanden verfügbar sind
  - funktionelle Einheit (ALU) frei ist
- Ausführung wird durch „Reservation Stations“ kontrolliert
  - beobachtet die Datenabhängigkeiten zwischen μOPs
  - teilt Ressourcen zu
- „Trace“ Cache
  - ersetzt traditionellen Anweisungscache
  - speichert Anweisungen in decodierter Form: Folgen von μOPs
  - reduziert benötigte Rate für den Anweisungsdecoder

Pentium 4 / NetBurst Architektur (cont.)

- „Double pumped“ ALUs (2 Operationen pro Taktzyklus)
- große Pipelinelänge ⇒ sehr hohe Taktfrequenzen

Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

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

- umfangreiches Material von Intel unter:
  `ark.intel.com`, `techresearch.intel.com`

Pentium 4 / NetBurst Architektur (cont.)

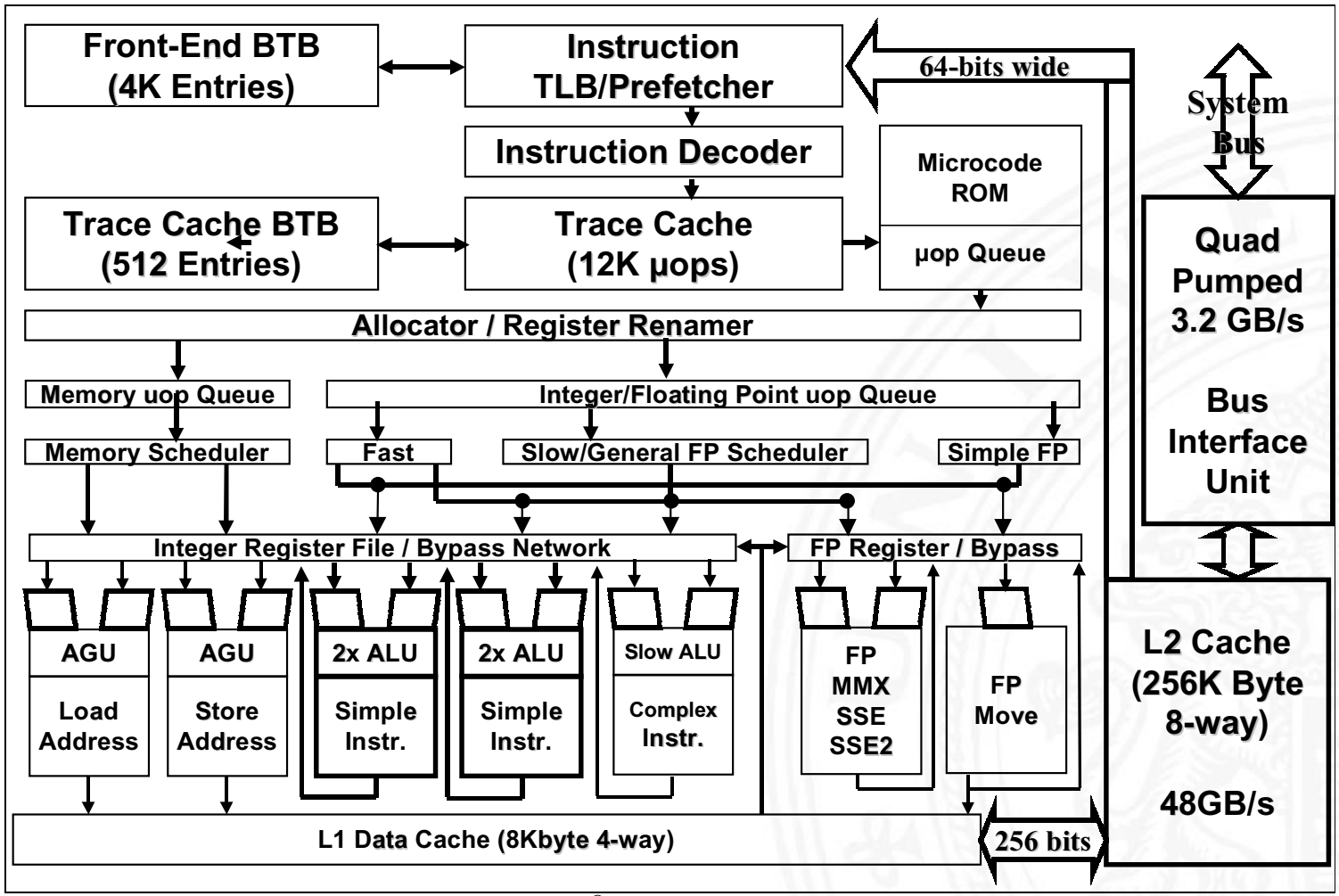
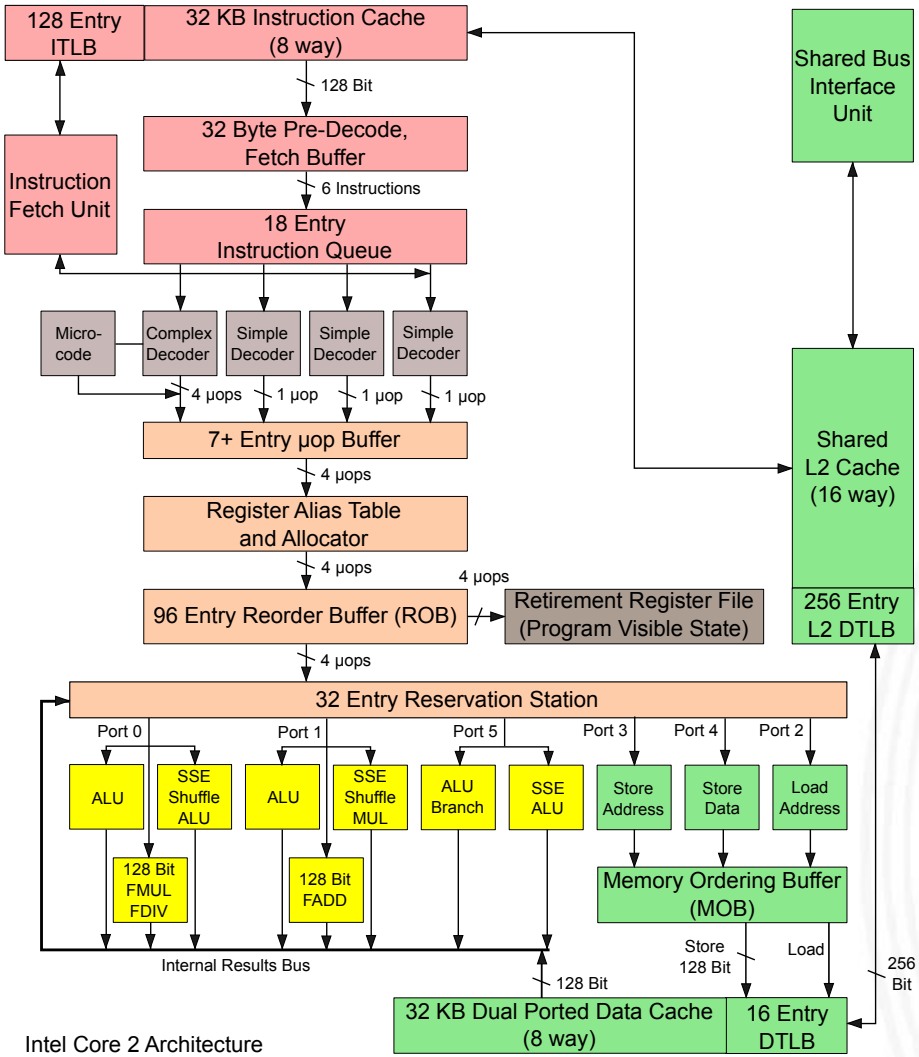


Figure 4: Pentium® 4 processor microarchitecture

Intel
Q1, 2001

Core 2 Architektur



Intel Core 2 Architecture