64-040 Modul InfB-RS: Rechnerstrukturen

https://tams.informatik.uni-hamburg.de/lectures/2015ws/vorlesung/rs

– Kapitel 14 –

Norman Hendrich

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

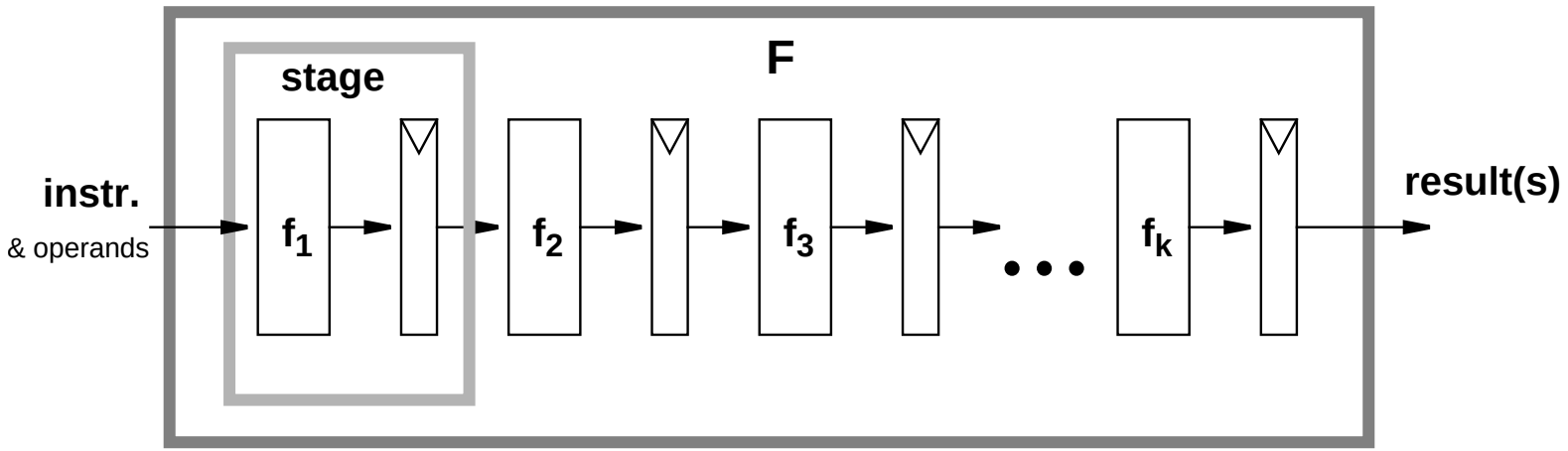
Wintersemester 2015/2016

Kapitel 14

Pipelining

- Motivation und Konzept
- Befehlspipeline
- MIPS
- Bewertung
- Hazards
- Literatur

Pipelining / Fließbandverarbeitung



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 / Fließbandverarbeitung (cont.)

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

Pipelining / Fließbandverarbeitung (cont.)

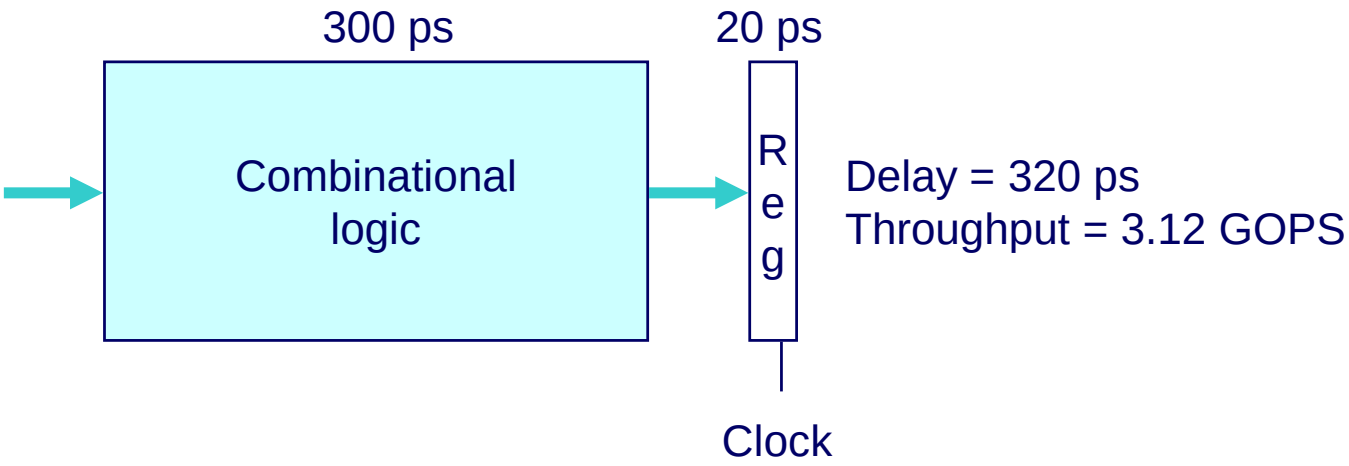
Arithmetische Pipelines

- 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

Befehlspipeline im Prozessor

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

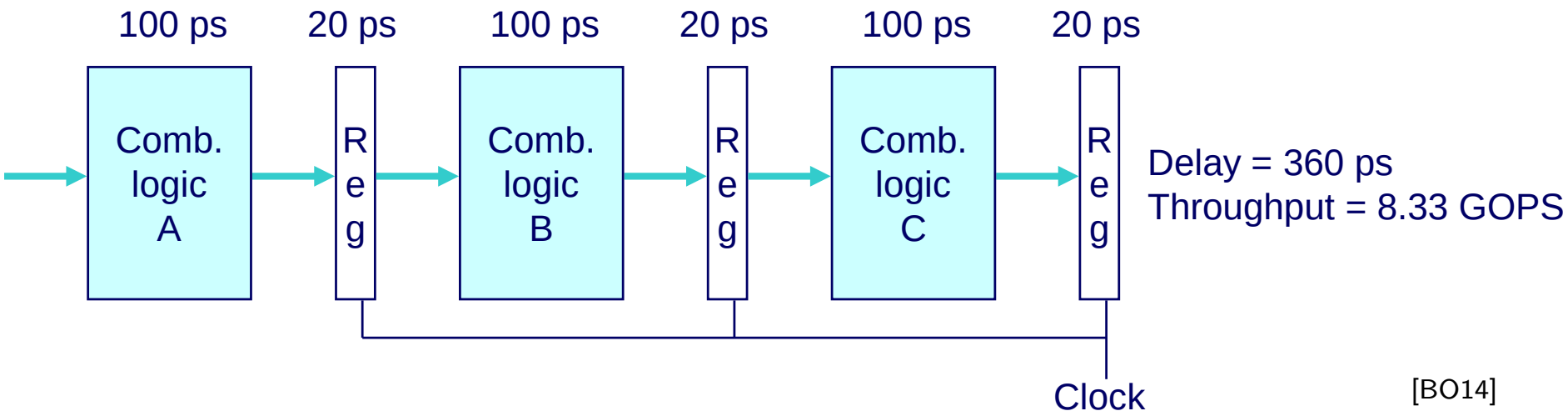
Beispiel: Schaltnetz ohne Pipeline



[BO14]

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

Beispiel: Version mit 3-stufiger Pipeline

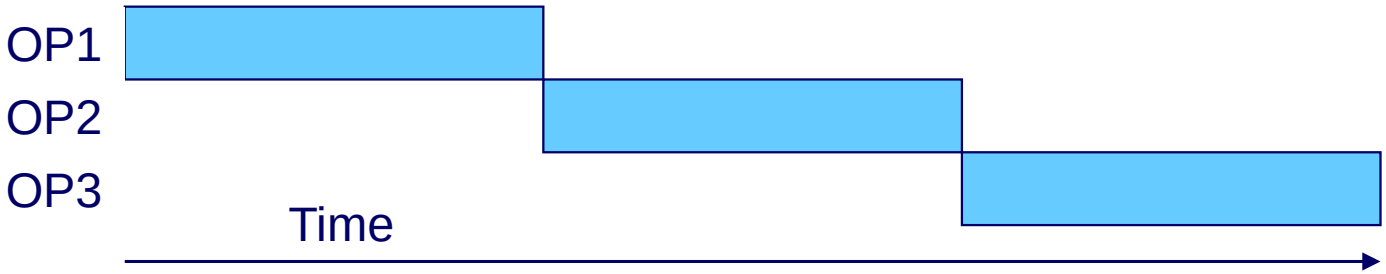


[BO14]

- 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

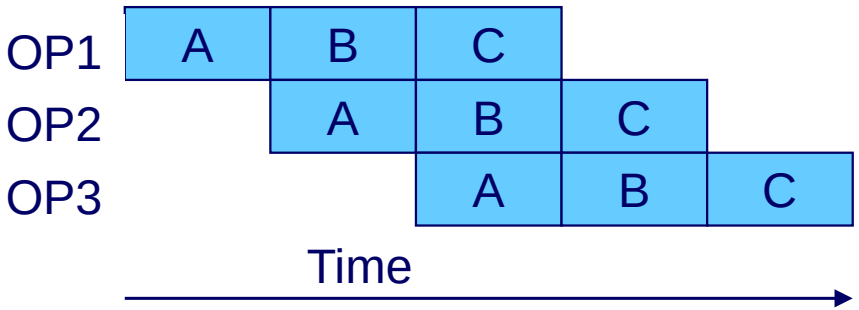
Prinzip: 3-stufige Pipeline

- ohne Pipeline

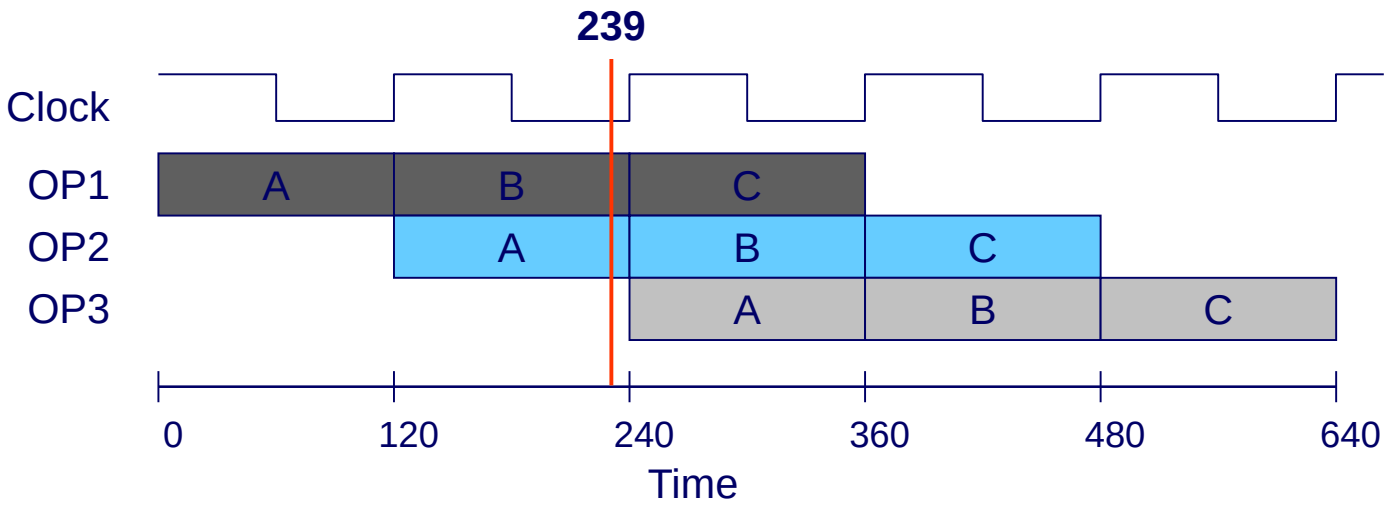


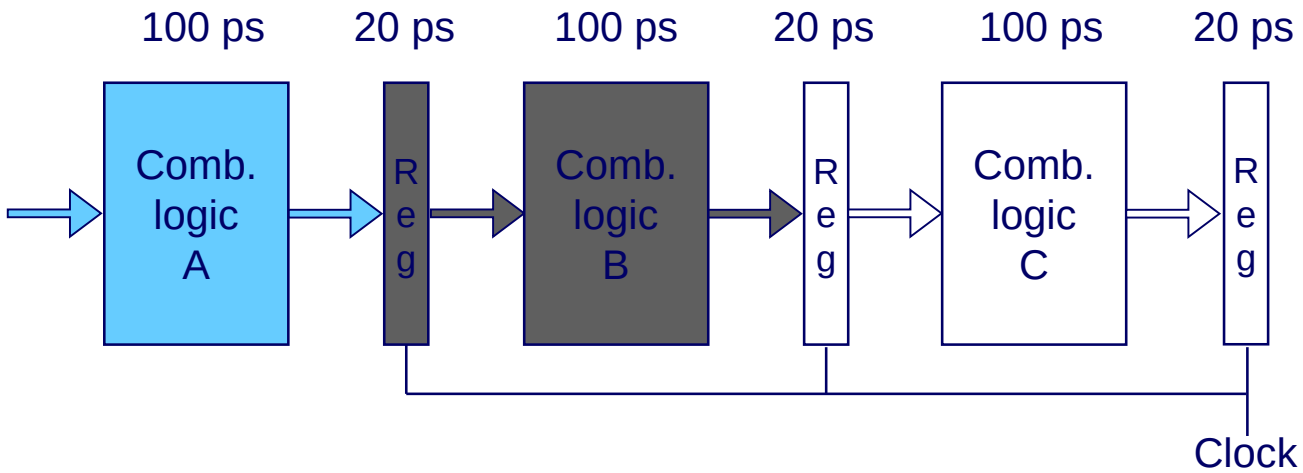
- 3-stufige Pipeline

[BO14]



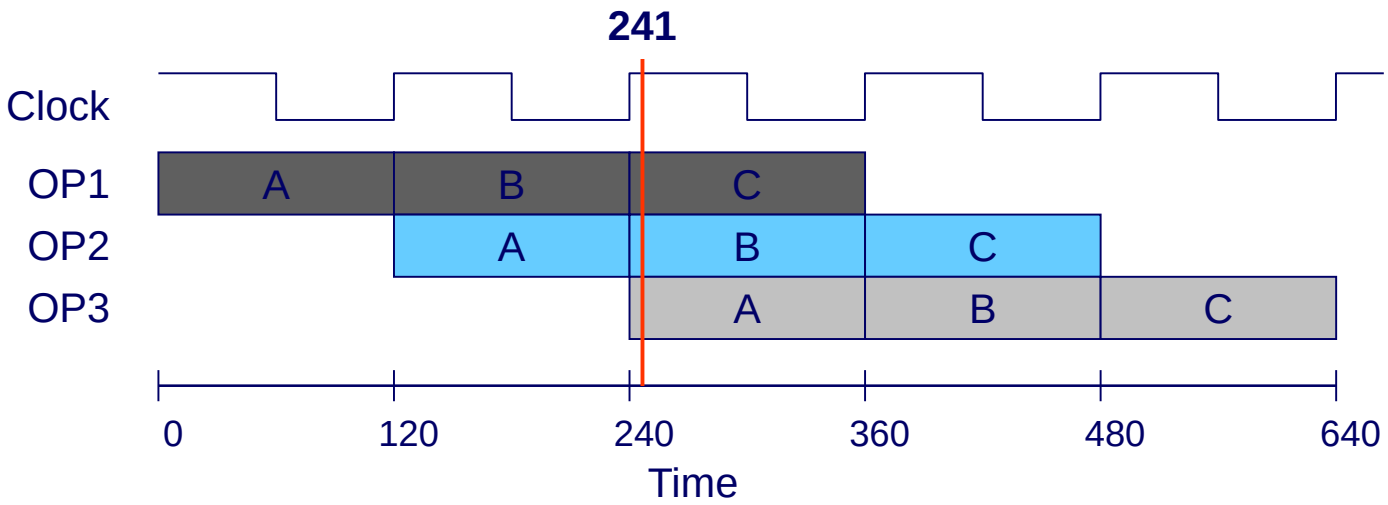
Timing: 3-stufige Pipeline

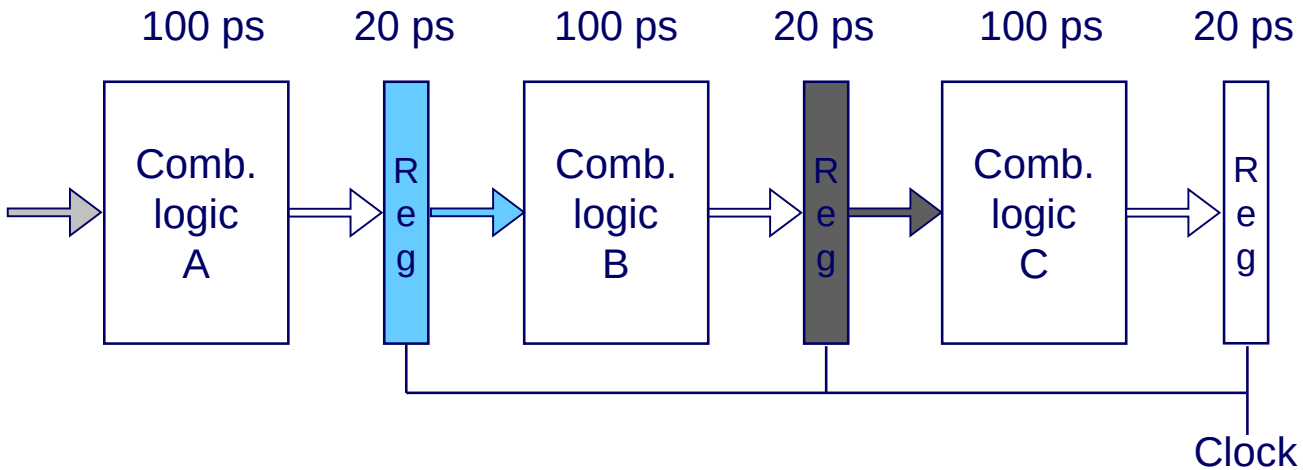




[BO14]

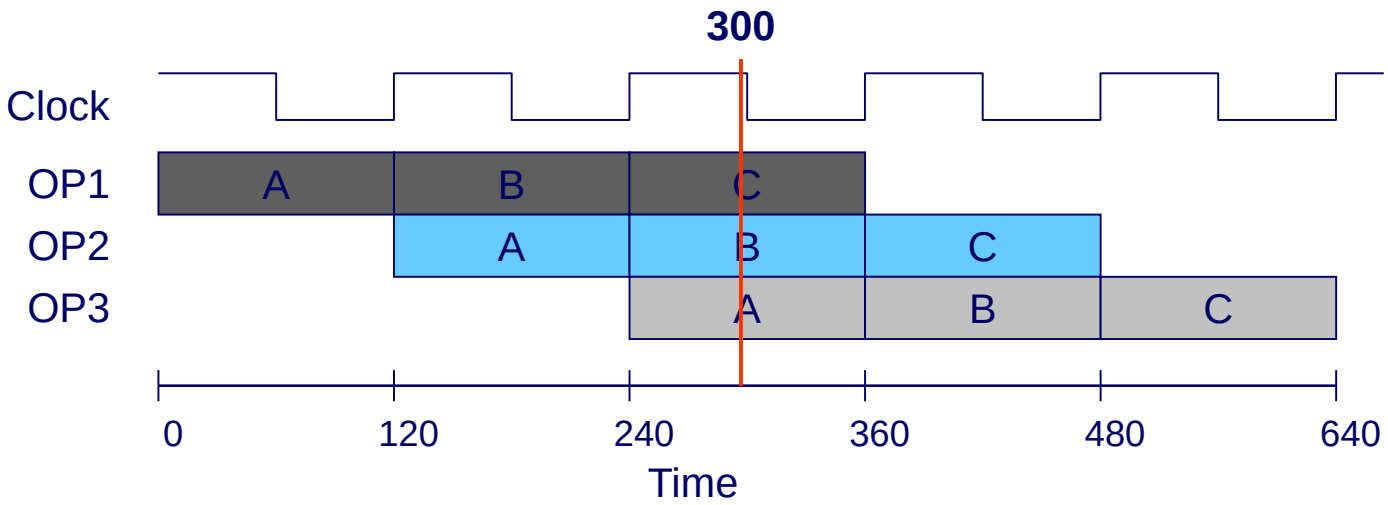
Timing: 3-stufige Pipeline

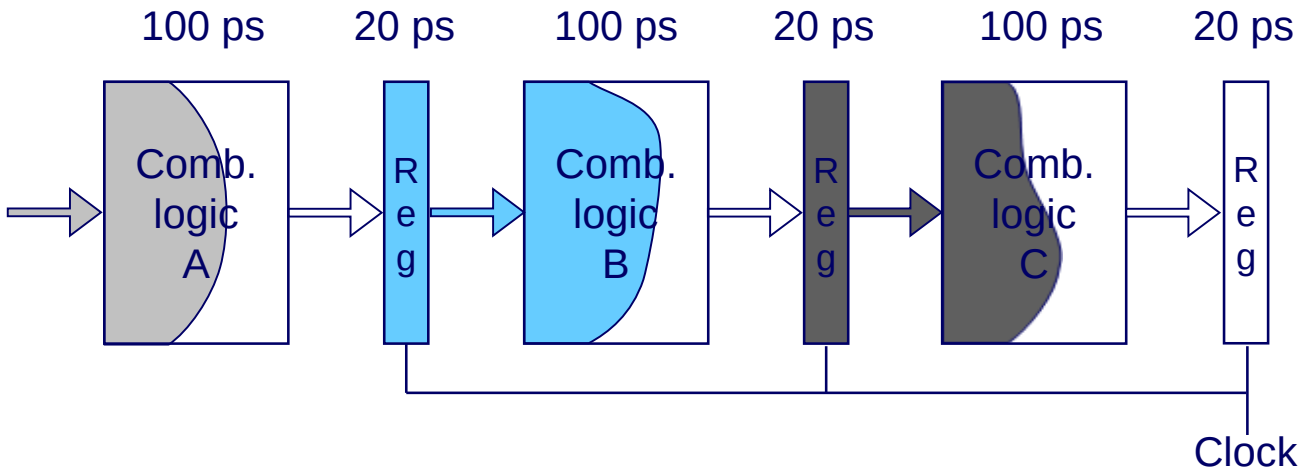




[BO14]

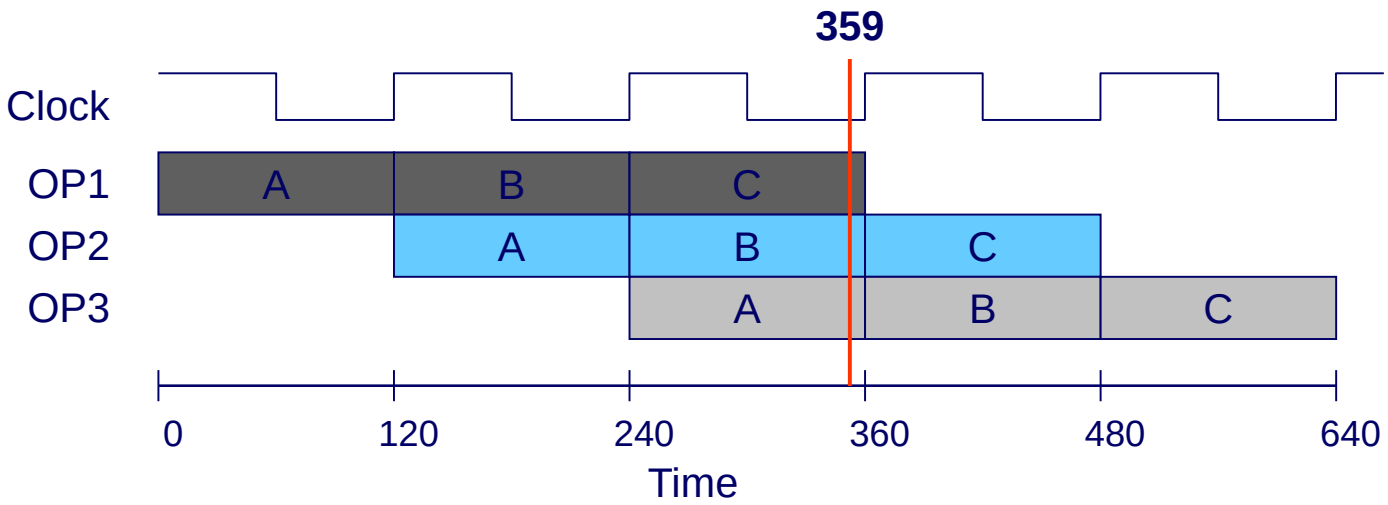
Timing: 3-stufige Pipeline

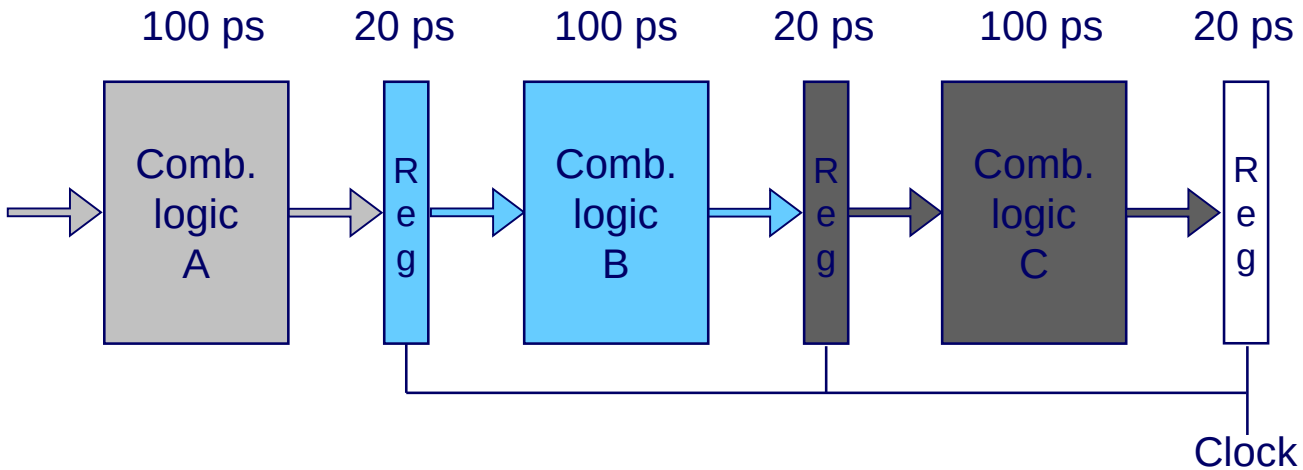




[BO14]

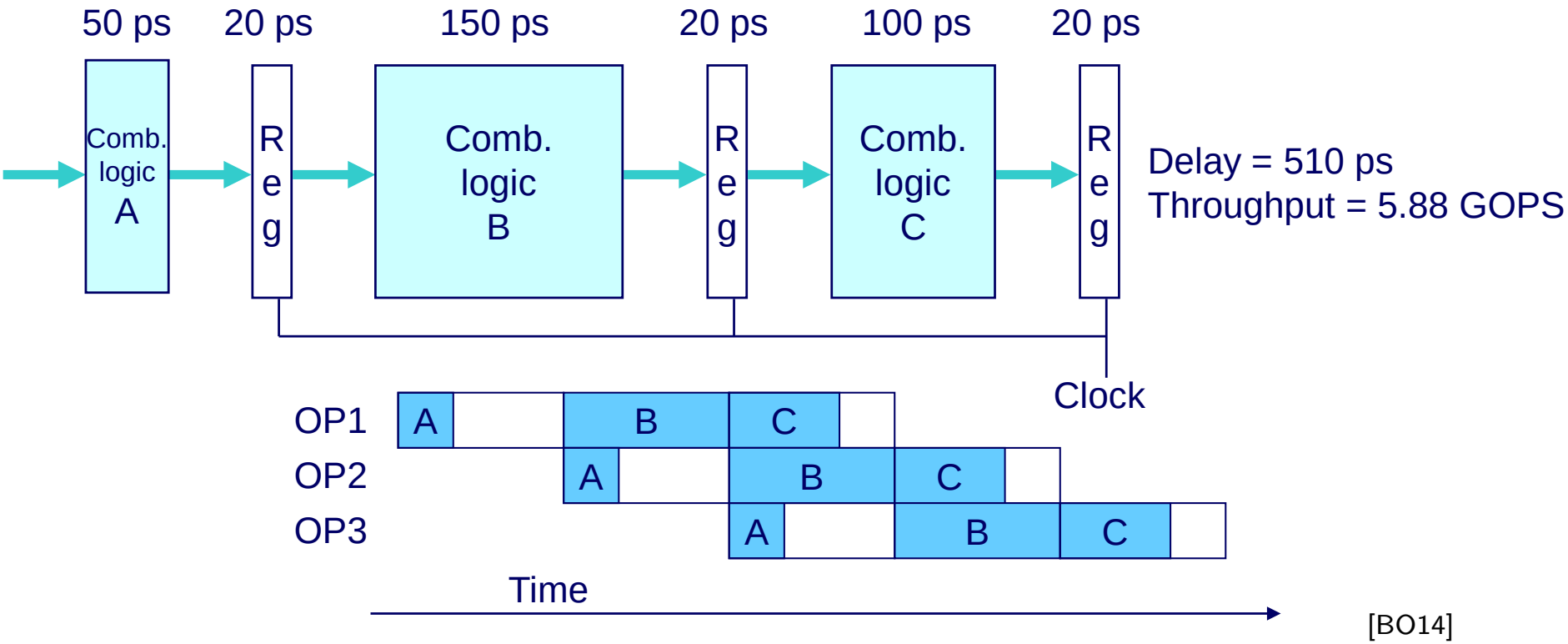
Timing: 3-stufige Pipeline





[BO14]

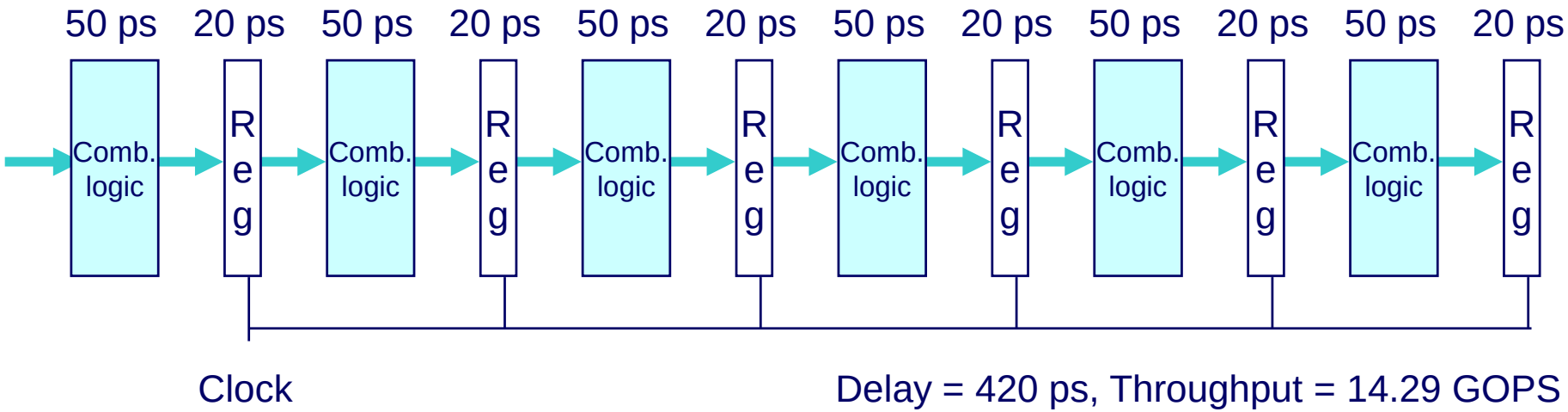
Limitierungen: nicht uniforme Verzögerungen



[BO14]

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

Limitierungen: Register „Overhead“

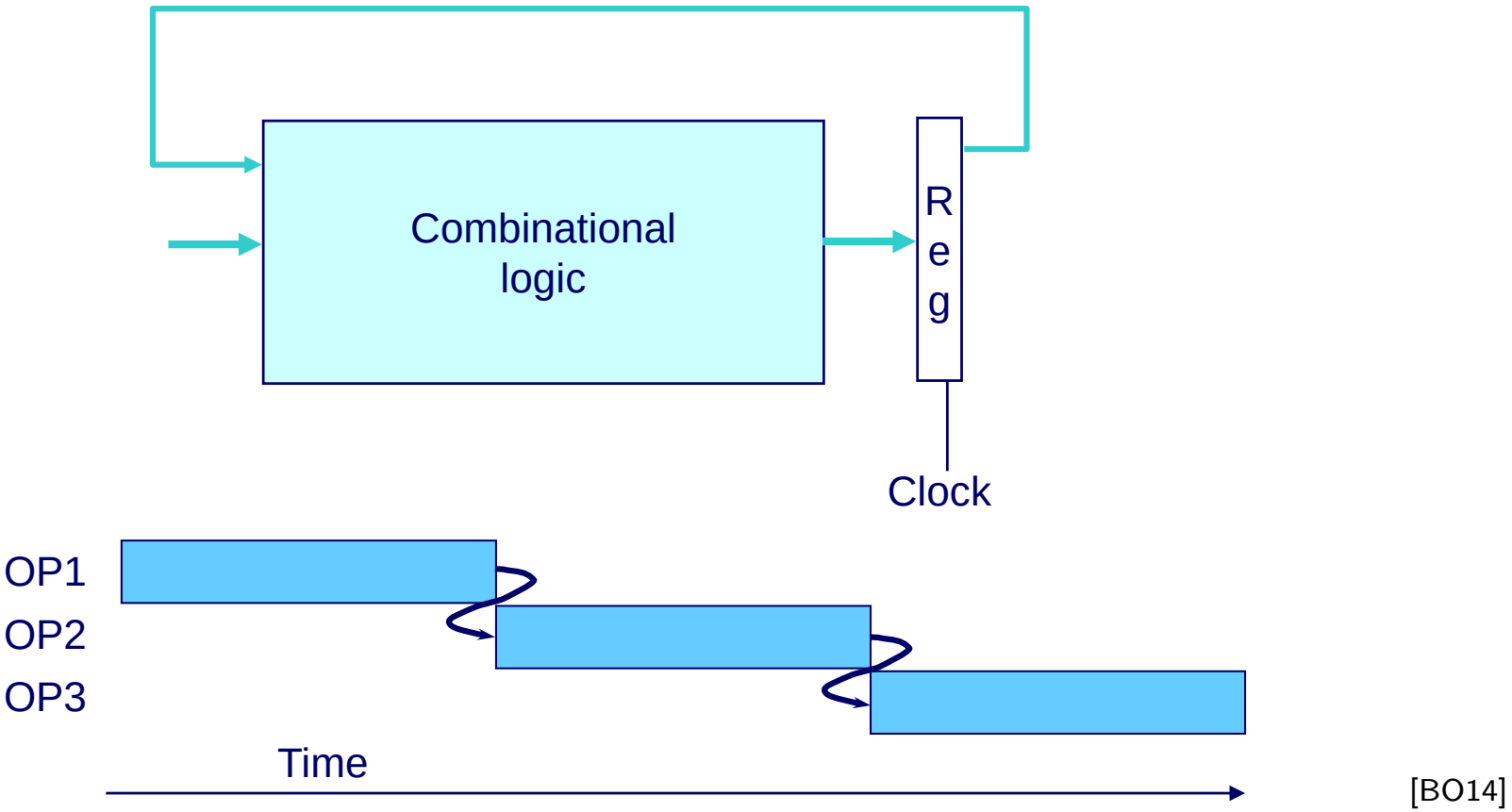


[BO14]

- 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

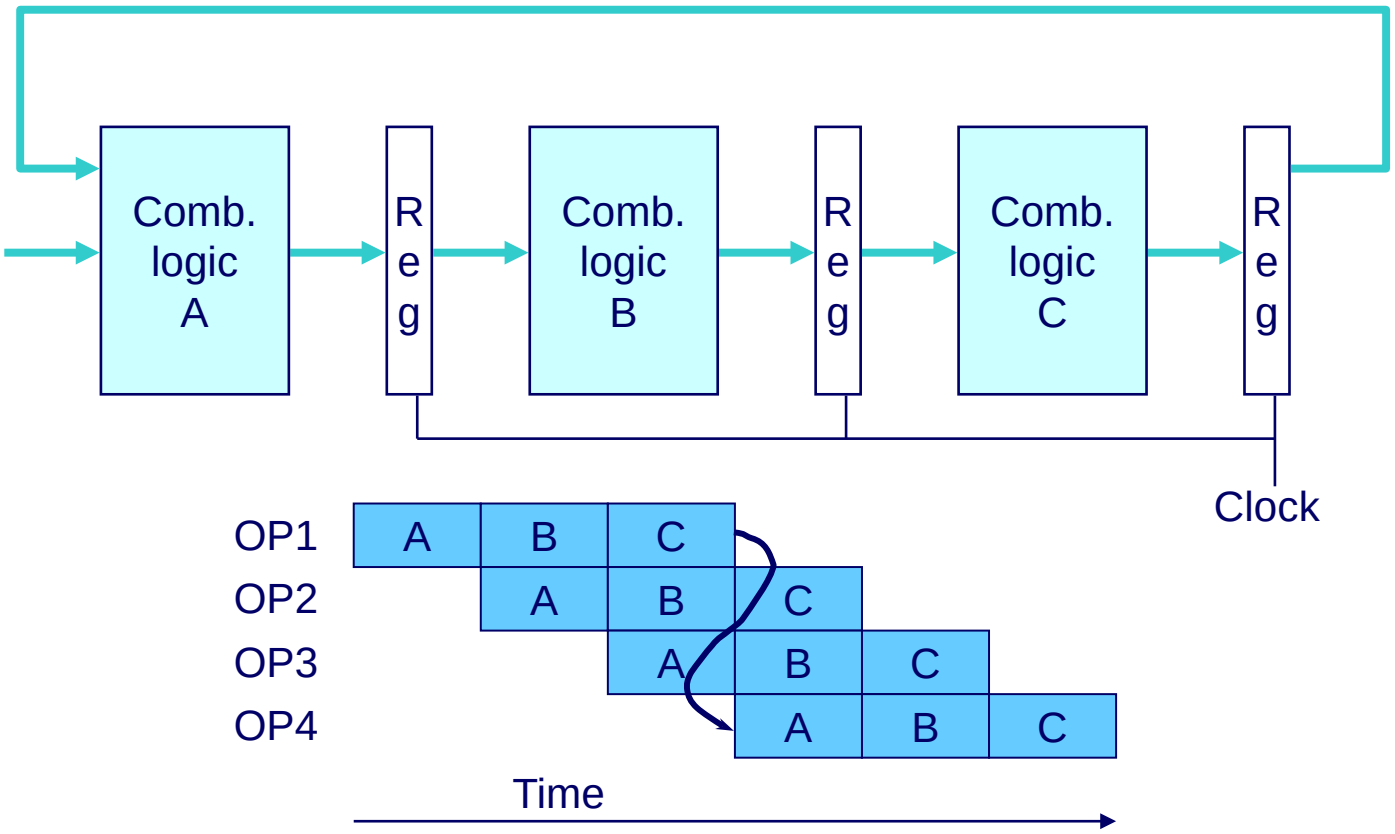
Limitierungen: Datenabhängigkeiten



[BO14]

- jede Operation hängt vom Ergebnis der Vorhergehenden ab

Limitierungen: Datenabhängigkeiten (cont.)



[BO14]

⇒ Resultat-Feedback kommt zu spät für die nächste Operation

⇒ Pipelining ändert Verhalten des gesamten Systems

von-Neumann Befehlszyklus

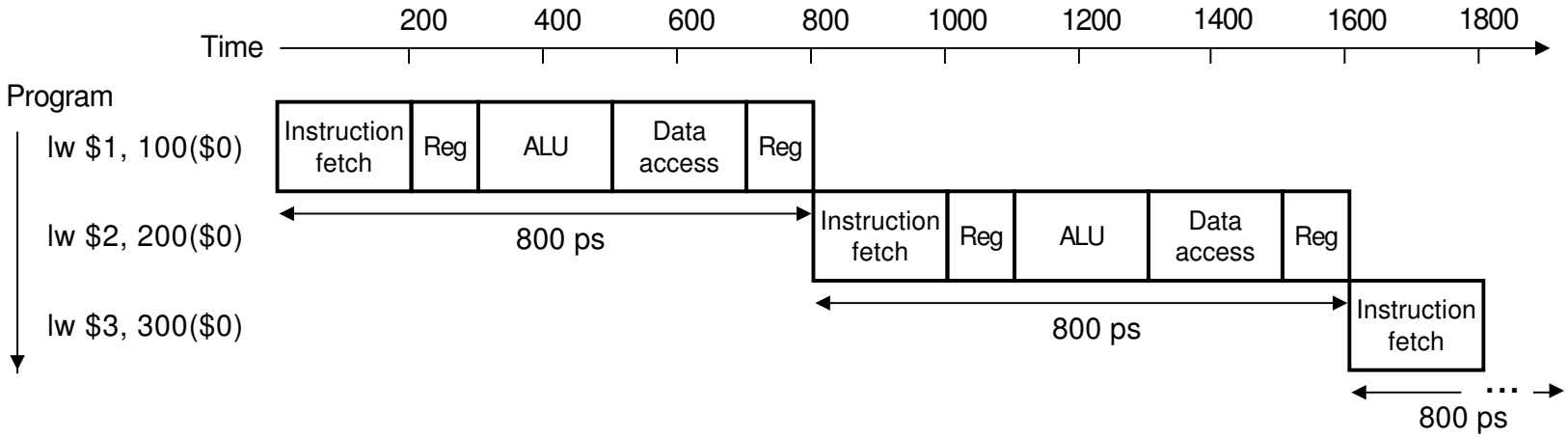
typische Schritte der Befehlsabarbeitung (von ISA abhängig)

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	**Ex**ecute ALU führt Befehl aus
MEM	**Mem**ory access Speicherzugriff: Daten laden/abspeichern
WB	**W**rite **B**ack Ergebnis in Register zurückschreiben

von-Neumann Befehlszyklus (cont.)

- 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)

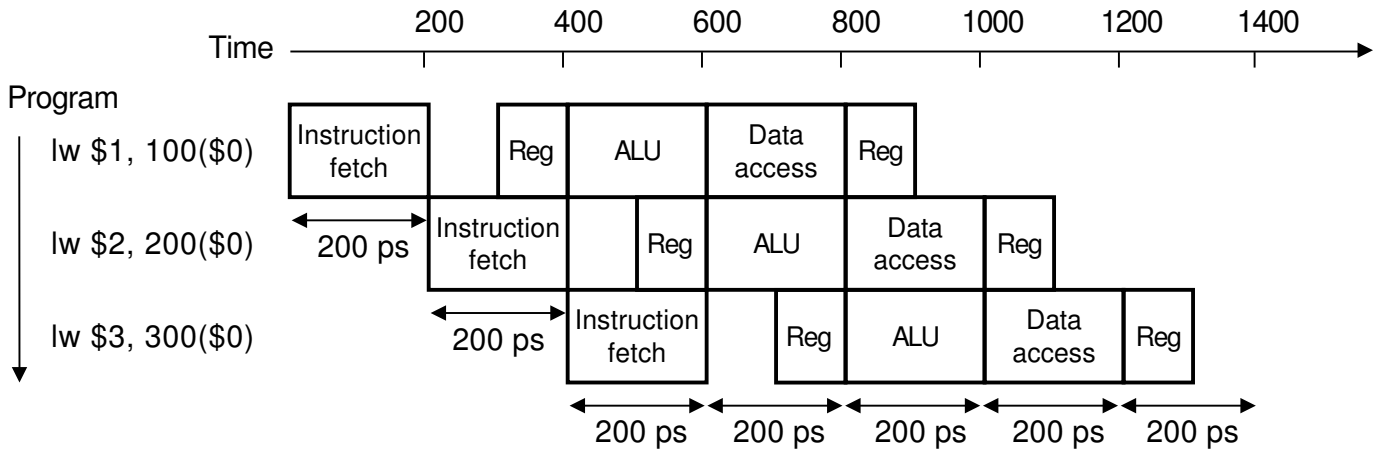
serielle Bearbeitung ohne Pipelining



[PH14]

von-Neumann Befehlszyklus (cont.)

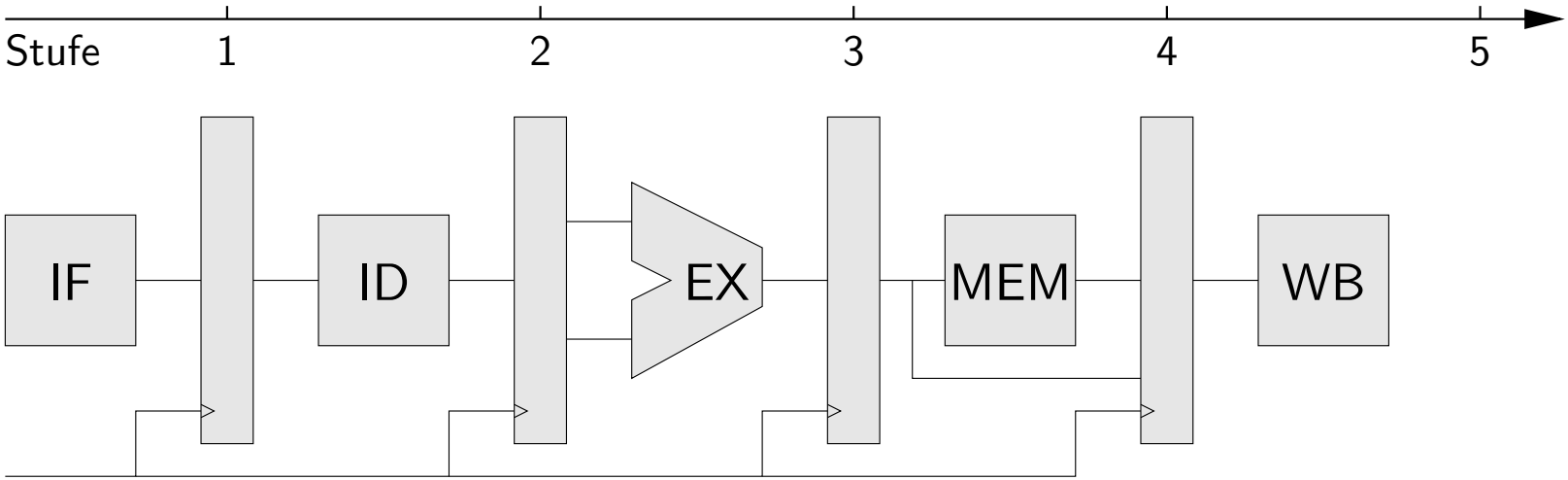
Pipelining für die einzelnen Schritte der Befehlsausführung



[PH14]

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

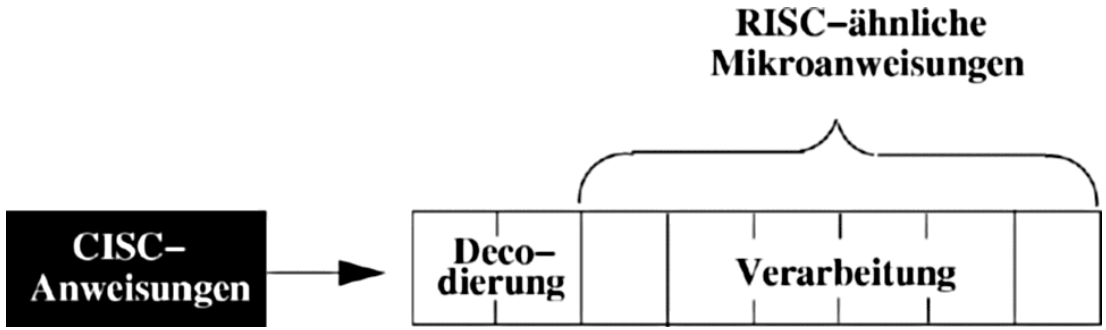
Klassische 5-stufige Pipeline



- Grundidee der ursprünglichen RISC-Architekturen
- \+ Durchsatz ca. 3...5× besser als serielle Ausführung
- \+ guter Kompromiss aus Leistung und Hardwareaufwand

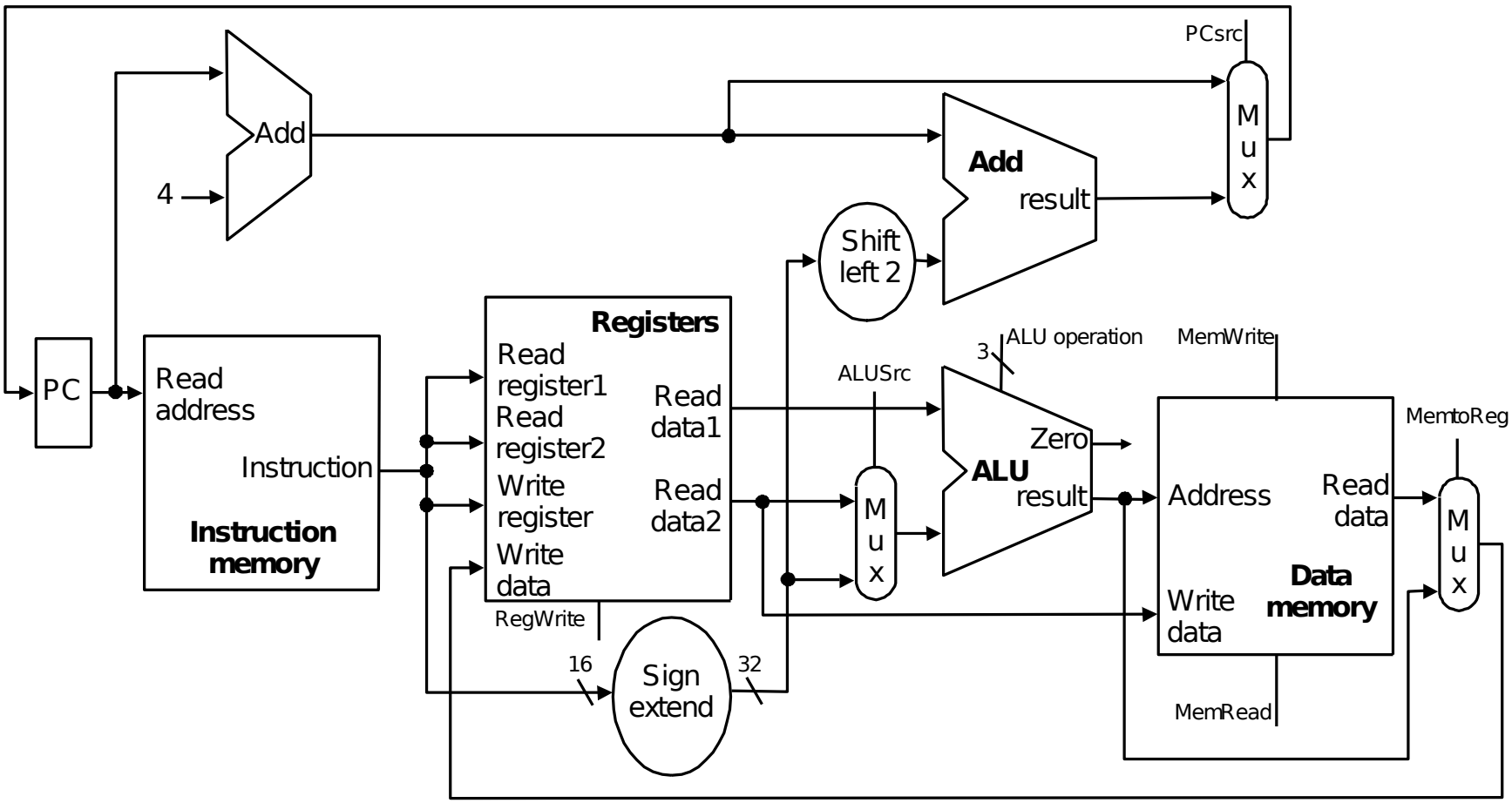
Klassische 5-stufige Pipeline (cont.)

- 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



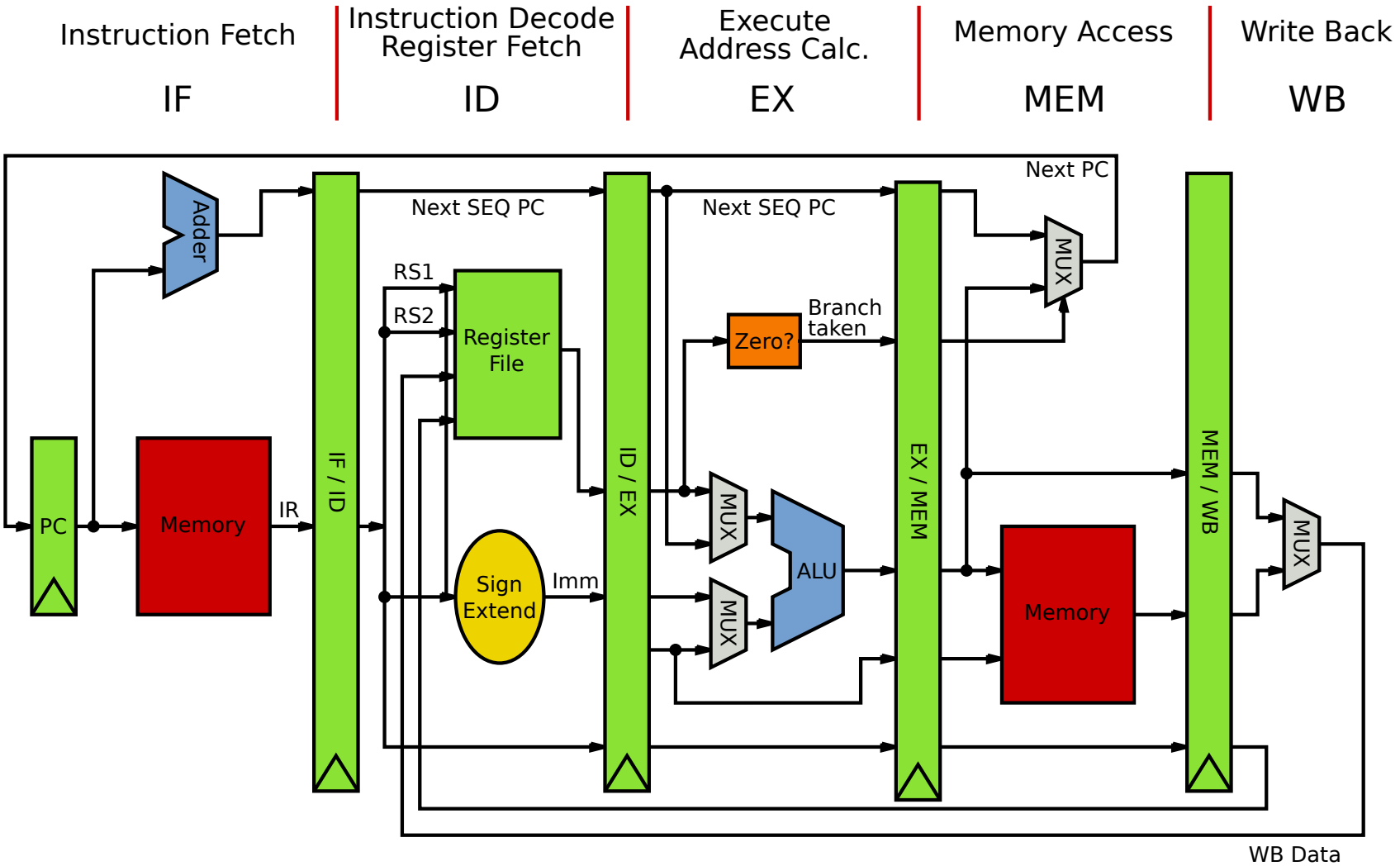
\+ CISC-Software bleibt lauffähig
\+ Befehlssatz wird um neue RISC Befehle erweitert

MIPS: serielle Realisierung ohne Pipeline



längster Pfad: PC - IM - REG - MUX - ALU - DM - MUX - PC/REG [PH14]

MIPS: mit 5-stufiger Pipeline



MIPS: mit 5-stufiger Pipeline (cont.)

- 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?!

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

+ Schaltnetze in kleinere Blöcke aufgeteilt ⇒ höherer Takt
+ im Idealfall ein neuer Befehl pro Takt gestartet ⇒ 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

Prozessorpipeline – Bewertung (cont.)

Analyse

- N Instruktionen; K Pipelinestufen
- ohne Pipeline: $N \cdot K$ Taktzyklen
- mit Pipeline: $K + N - 1$ Taktzyklen

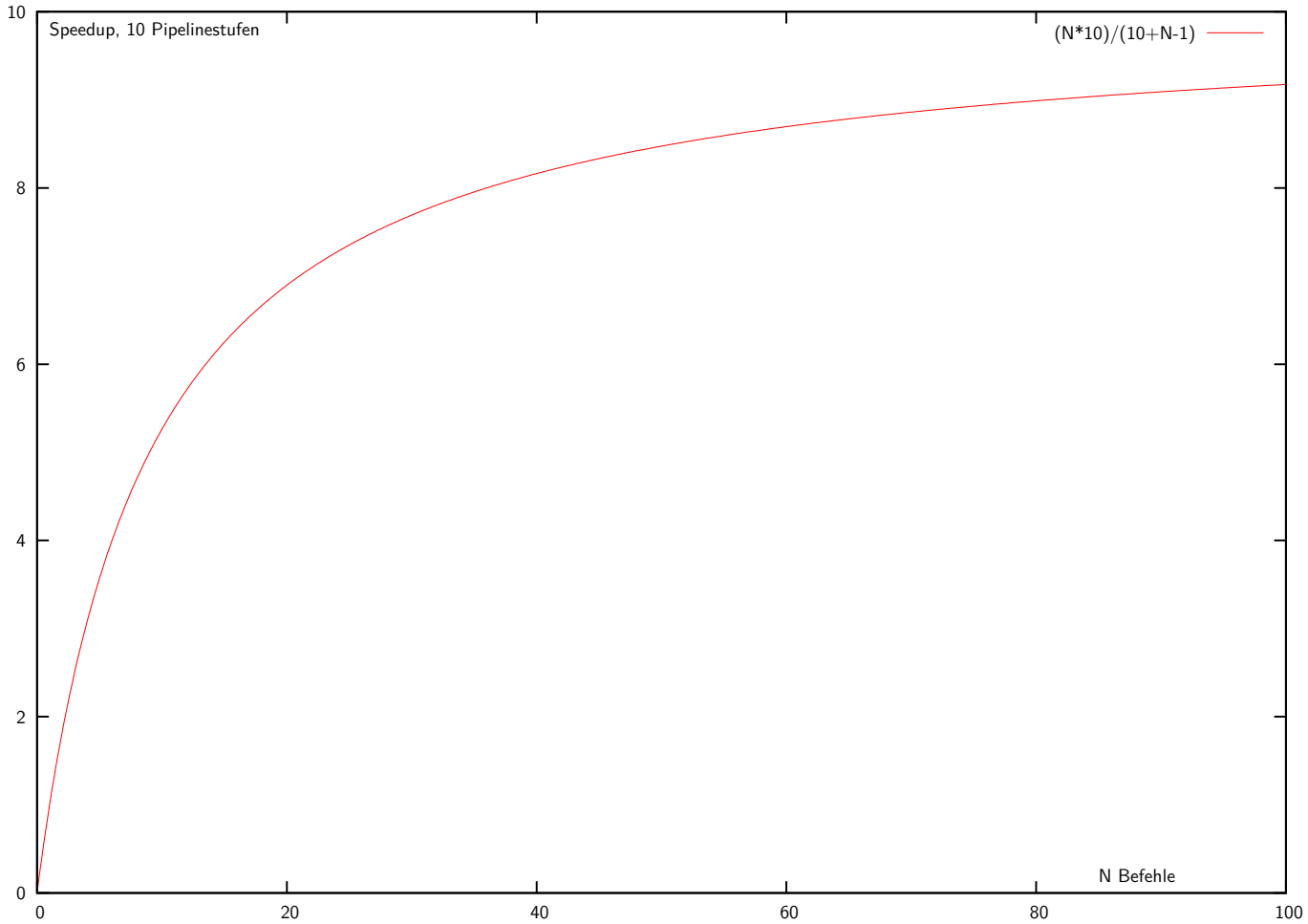
- „Speedup“ $S = \frac{N \cdot K}{K+N-1}$, $\lim_{N \to \infty} S = K$

$\Rightarrow$ ein großer Speedup wird erreicht durch

- große Pipelinetiefe: K
- lange Instruktionssequenzen: N

- wegen Daten- und Kontrollabhängigkeiten nicht erreichbar
- außerdem: Register-Overhead nicht berücksichtigt

Prozessorpipeline – Bewertung (cont.)



Speedup

Prozessorpipeline – Dimensionierung

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

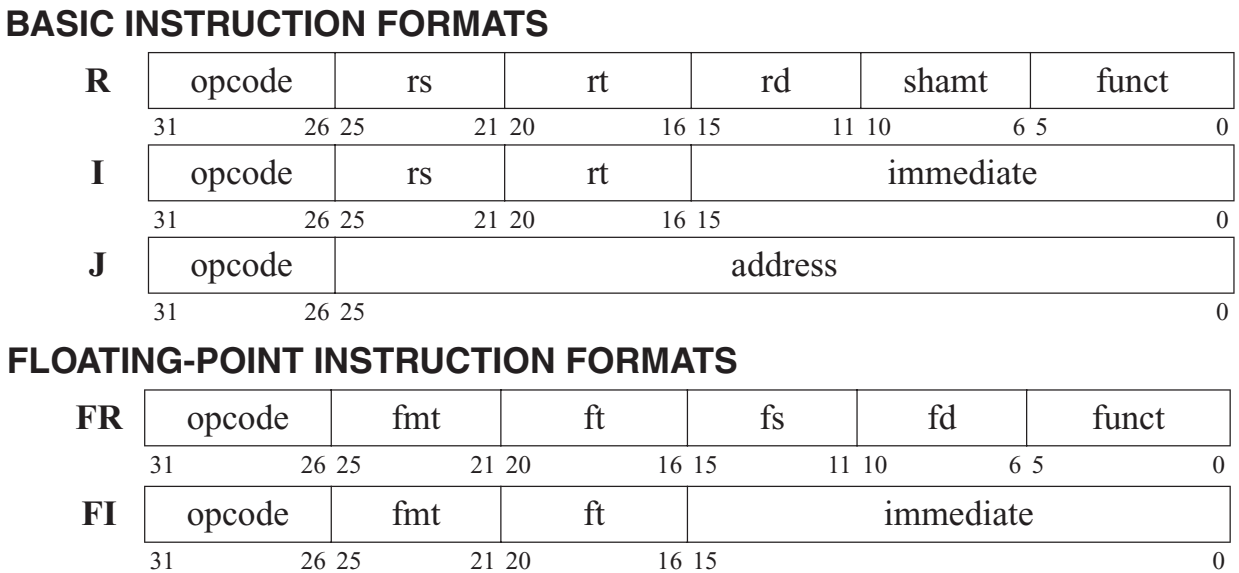
CPU	Pipelinestufen	Taktfrequenz [MHz]
80 386	1	33
Pentium	5	300
Motorola G4	4	500
Motorola G4e	7	1000
Pentium II/III	12	1400
Athlon XP	10/15	2500
Athlon 64, Opteron	12/17	$\leq$ 3000
Pentium 4	20	$\leq$ 5000

Prozessorpipeline – Auswirkungen

Architekturentscheidungen, die sich auf das Pipelining auswirken

gut für Pipelining

- gleiche Instruktionslänge
- wenige Instruktionsformate
- Load/Store Architektur



MIPS-Befehlsformate [PH14]

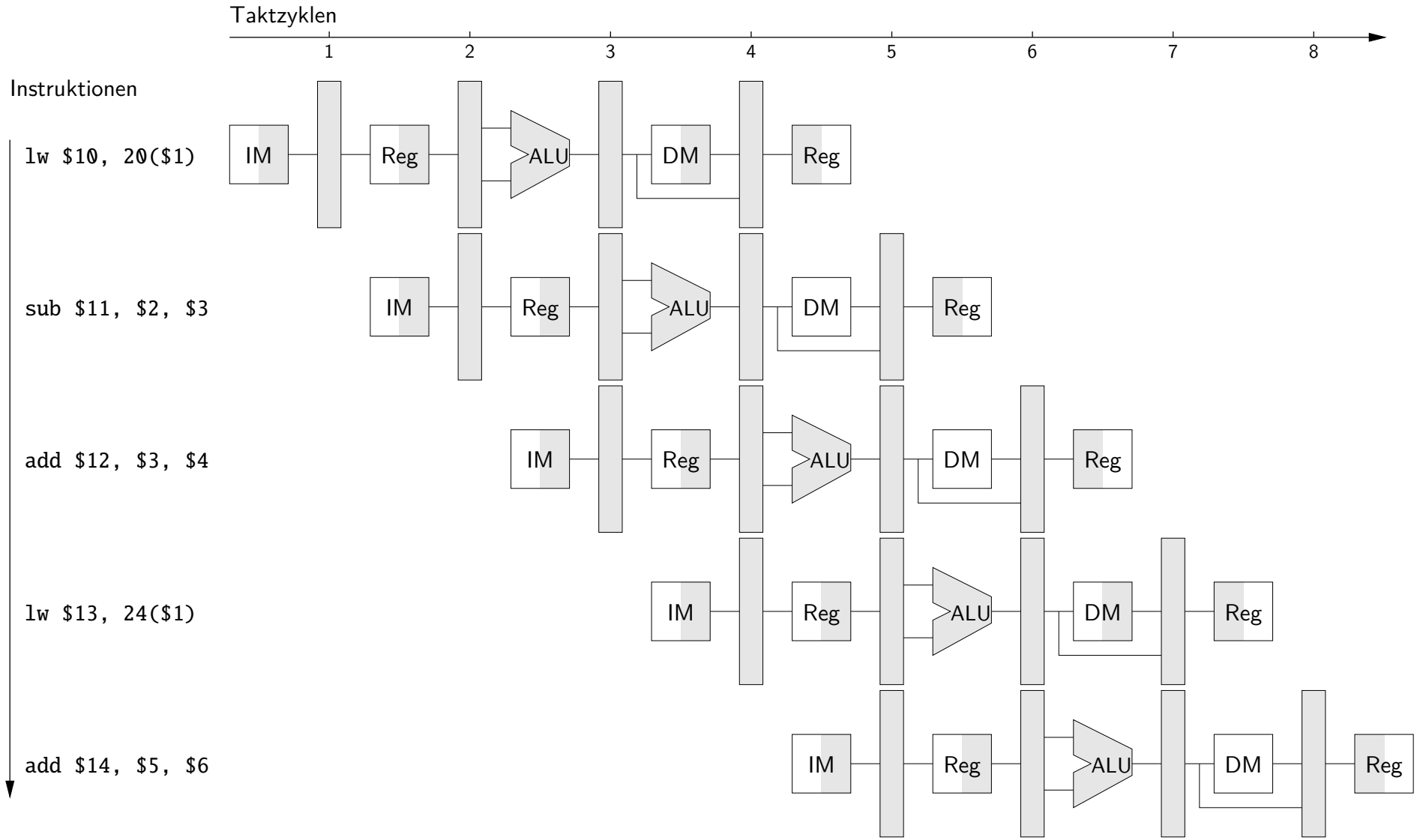
Prozessorpipeline – Auswirkungen (cont.)

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.


Taktzyklen
1
2
3
4
5
6
7
8
Instruktionen
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
IM
Reg
ALU
DM
Reg

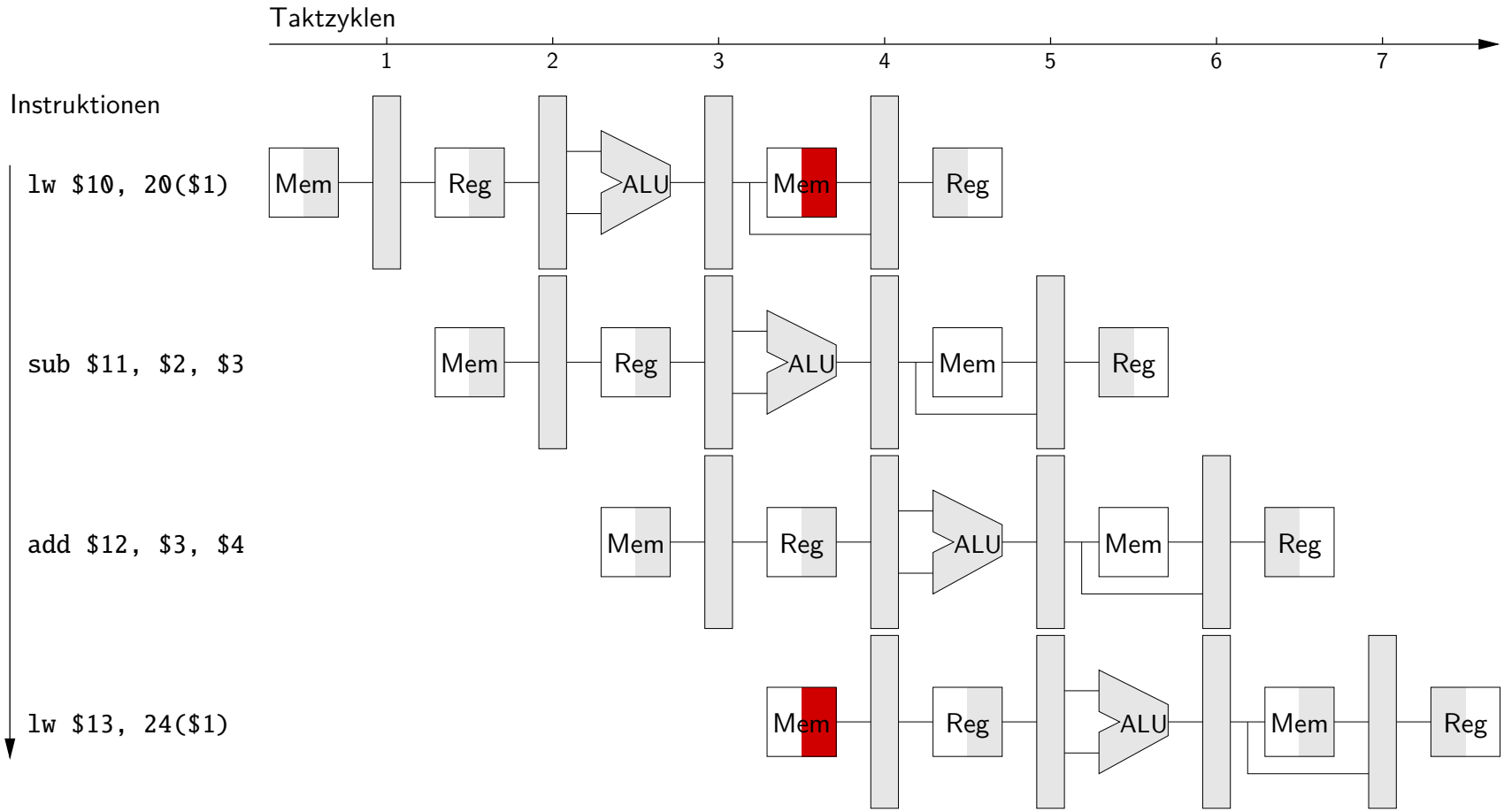
Pipeline Strukturkonflikte

Strukturkonflikt / Structural Hazard

- mehrere Stufen wollen gleichzeitig auf eine Ressource zugreifen
- Beispiel: gleichzeitiger Zugriff auf Speicher

$\Rightarrow$ Mehrfachauslegung der betreffenden Ressourcen

- Harvard-Architektur vermeidet Strukturkonflikt aus Beispiel
- Multi-Port Register
- mehrfach vorhandene Busse und Multiplexer...



gleichzeitiges Laden aus *einem* Speicher, zwei verschiedene Adressen

Pipeline Datenkonflikte

Datenkonflikt / Data Hazard

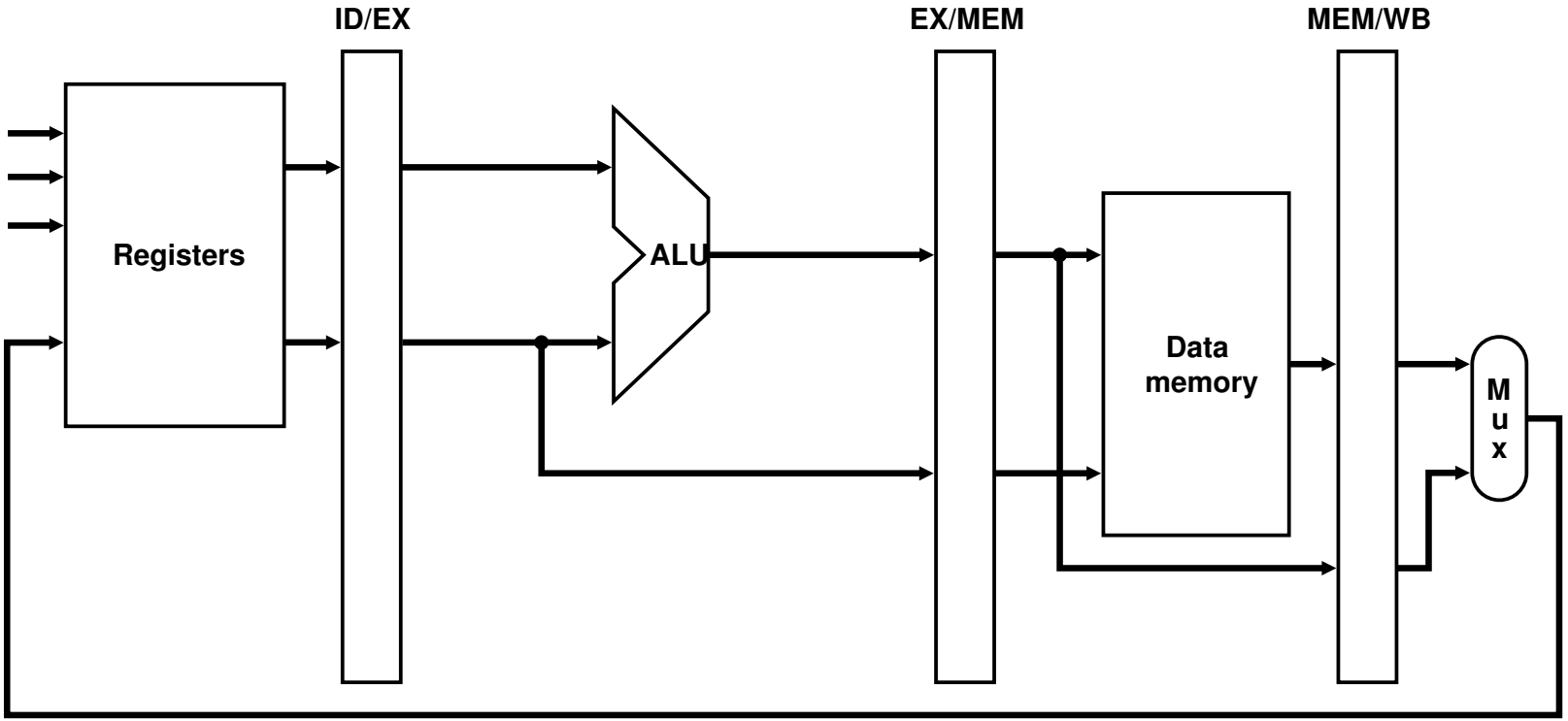
- eine Instruktion braucht die Ergebnisse einer vorhergehenden, diese wird aber noch in der Pipeline bearbeitet
- Datenabhängigkeiten aufeinanderfolgender Befehle
  - Operanden während ID-Phase aus Registerbank lesen
  - Resultate werden erst in WB-Phase geschrieben

  $\Rightarrow$ aber: Resultat ist schon nach EX-/MEM-Phase bekannt

Forwarding

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

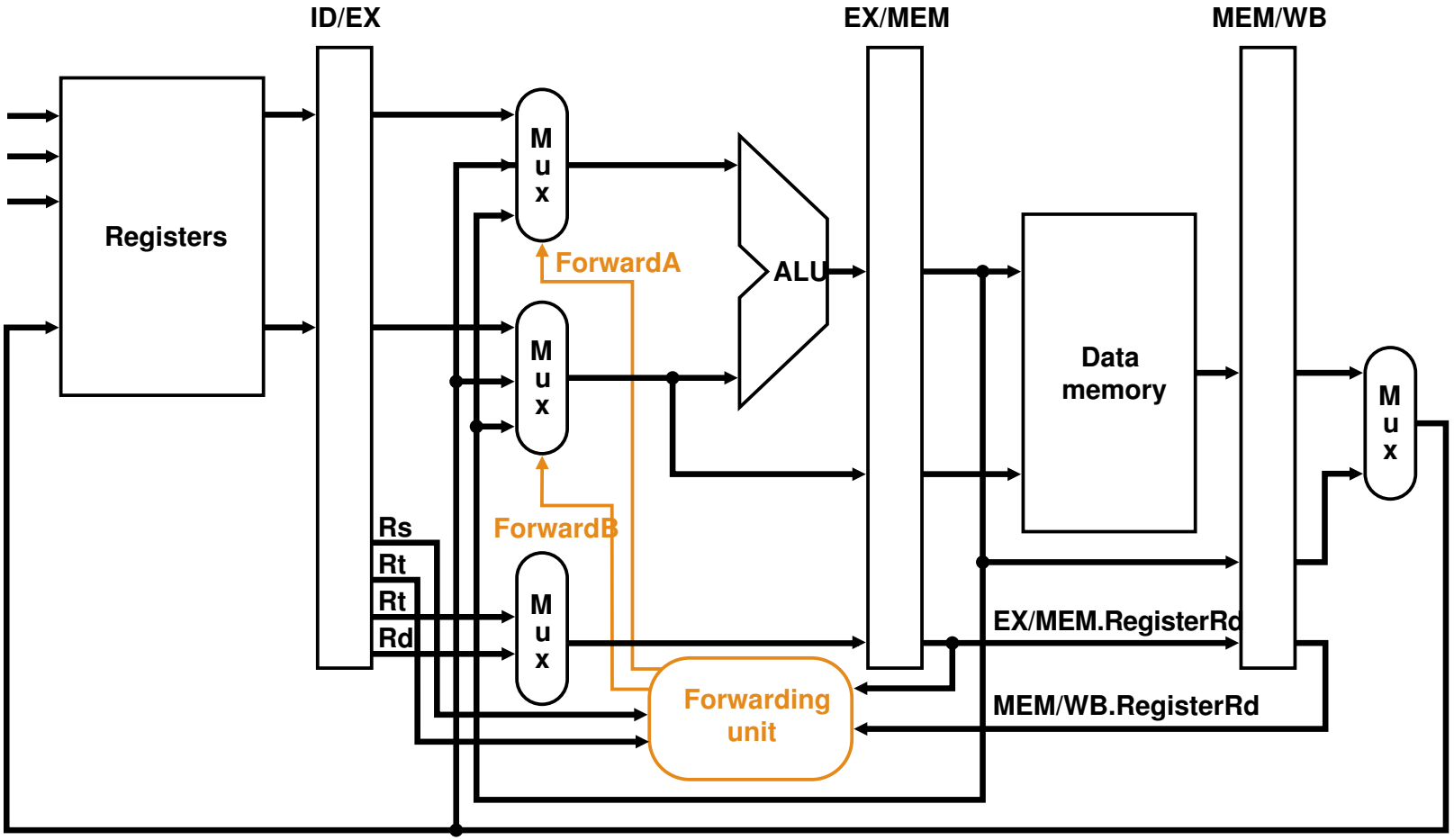
Pipeline Datenkonflikte (cont.)



MIPS Datenpfad

[PH14]

Pipeline Datenkonflikte (cont.)



MIPS Forwarding

[PH14]

Pipeline Datenkonflikte (cont.)

Rückwärtsabhängigkeiten

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

Auflösen von Rückwärtsabhängigkeiten

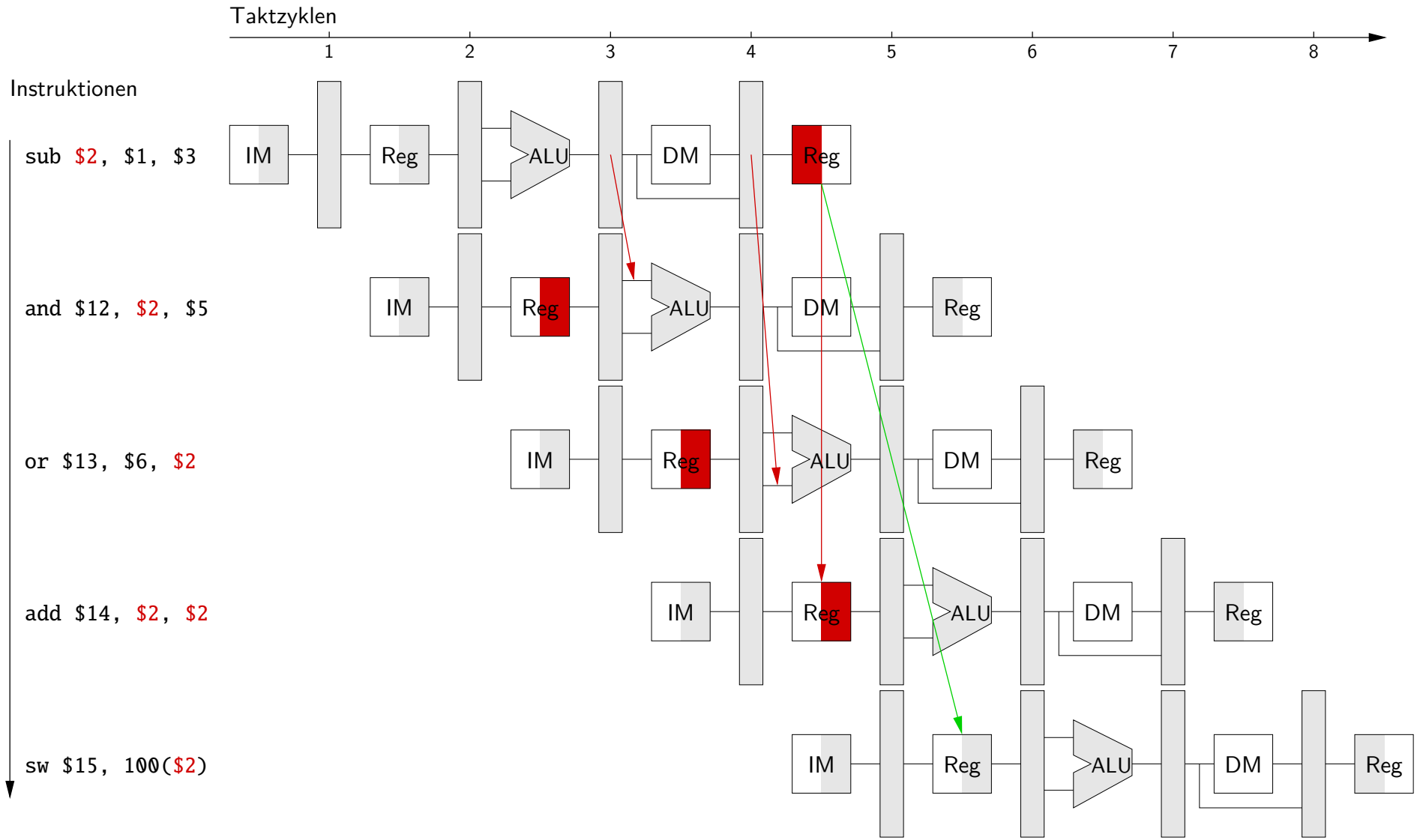
1. Softwarebasiert, durch den Compiler, Reihenfolge der Instruktionen verändern
   - andere Operationen (ohne Datenabhängigkeiten) vorziehen
   - nop-Befehl(e) einfügen

Pipeline Datenkonflikte (cont.)

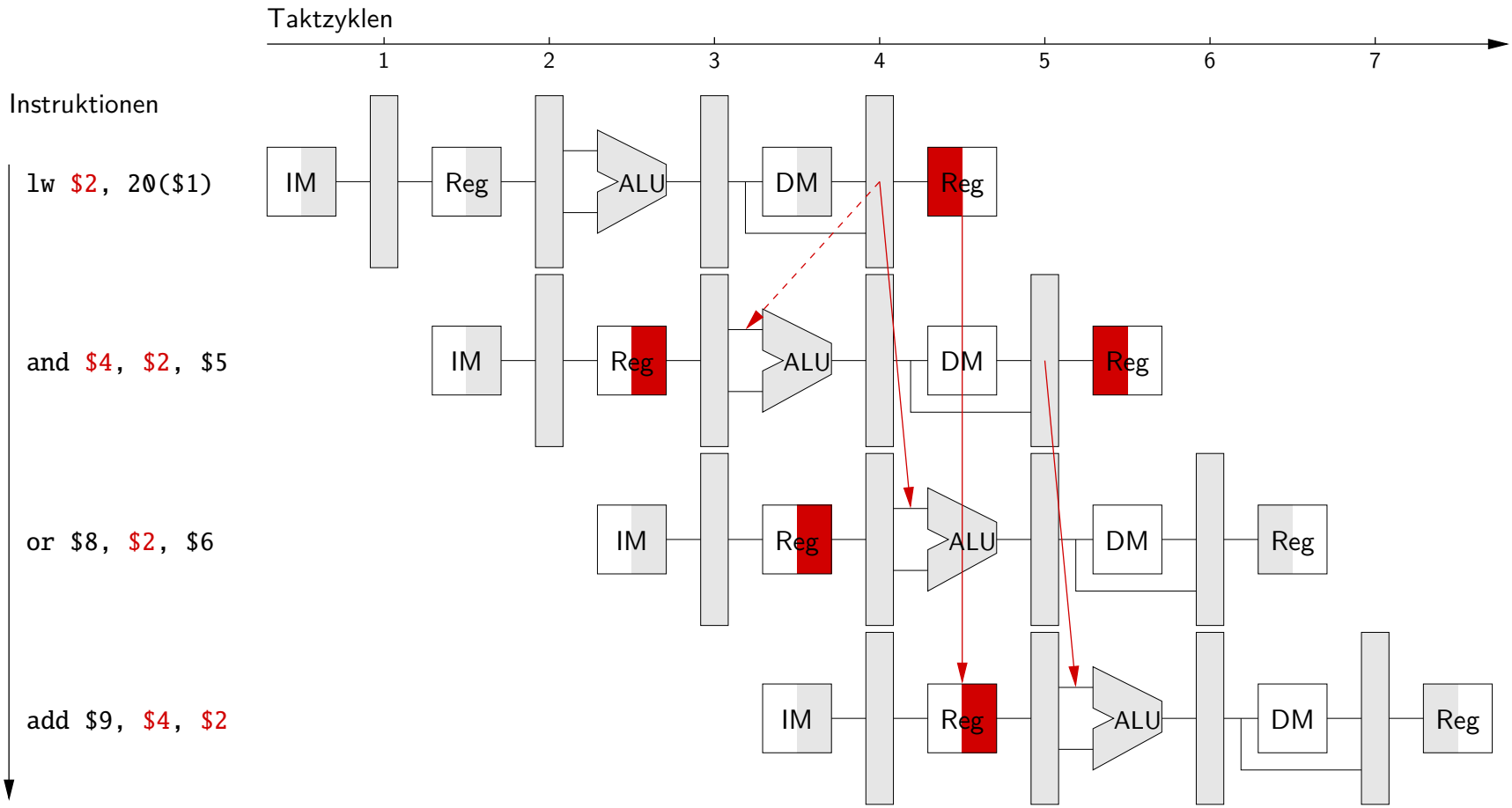
2. *„Interlocking“*
   - zusätzliche (Hardware) Kontrolleinheit: komplexes Steuerwerk
   - automatisches Stoppen der Pipeline, bis die benötigten Daten zur Verfügung stehen – Strategien:
     - in Pipeline werden keine neuen Instruktionen geladen
     - Hardware erzeugt: Pipelineleerlauf / *„pipeline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

„Scoreboard“

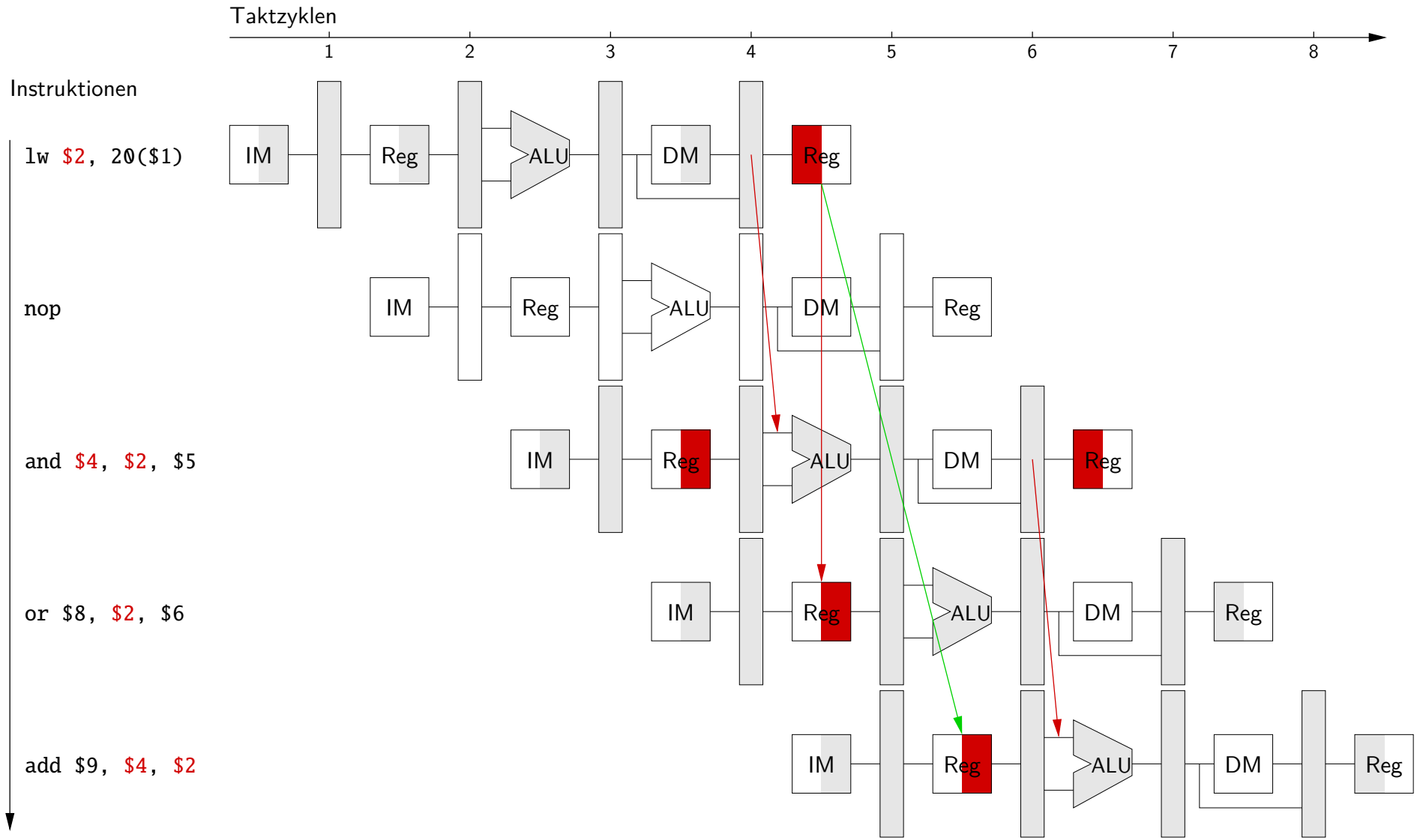
- Hardware Einheit zur zentralen Hazard-Erkennung und -Auflösung
- Verwaltet Instruktionen, benutzte Einheiten und Register der Pipeline (siehe *„Superskalare Rechner“*, ab Folie 1223)



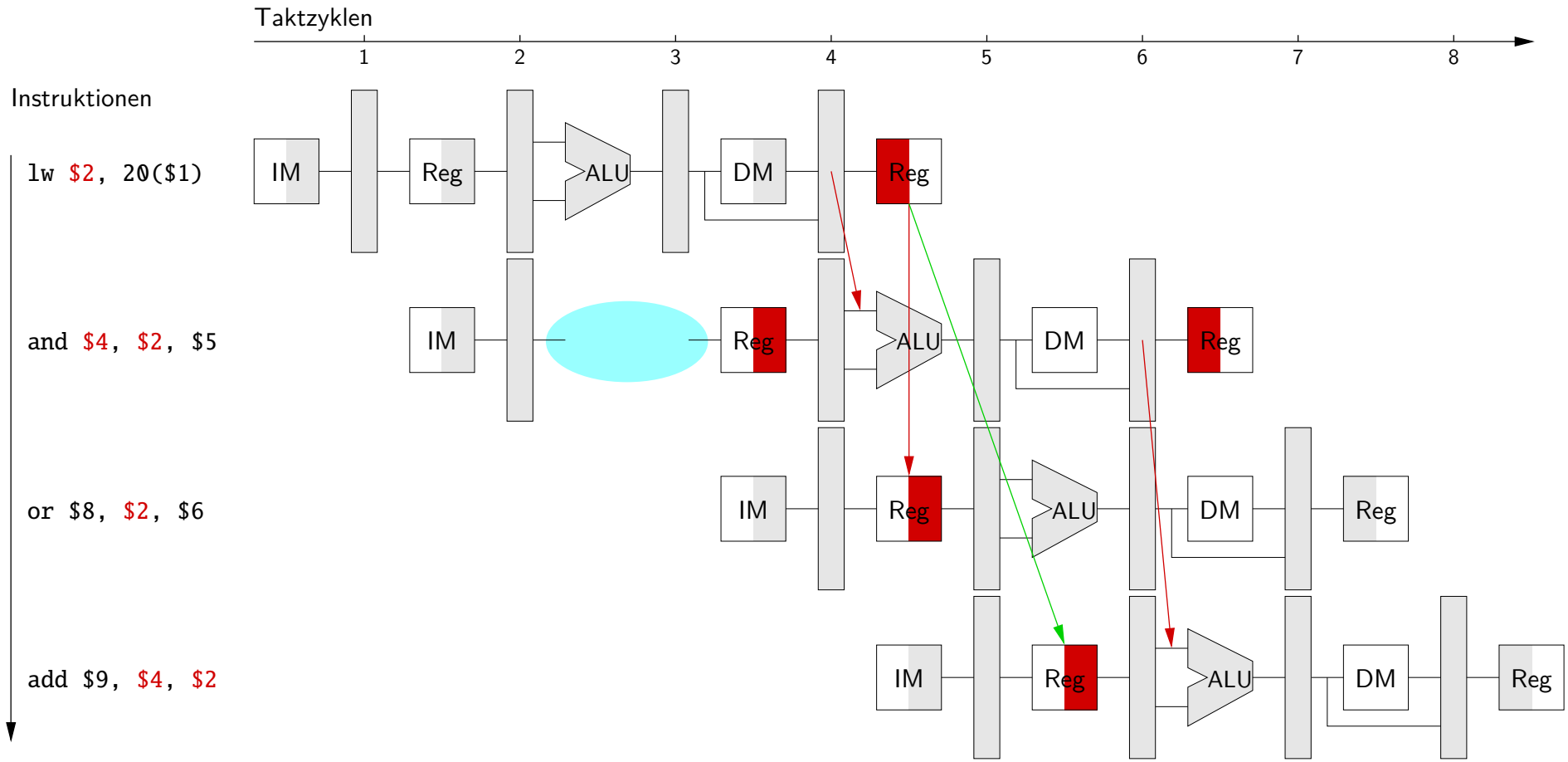
Befehle wollen R2 lesen, während es noch vom ersten Befehl berechnet wird



Befehle wollen R2 lesen, bevor es aus Speicher geladen wird



Compiler kennt Hardware und hat einen nop-Befehl eingefügt



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

Pipeline Steuerkonflikte

Steuerkonflikt / Control Hazard

- Unterbrechung des sequenziellen Ablaufs durchSprungbefehle 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

Pipeline Steuerkonflikte (cont.)

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)

Pipeline Steuerkonflikte (cont.)

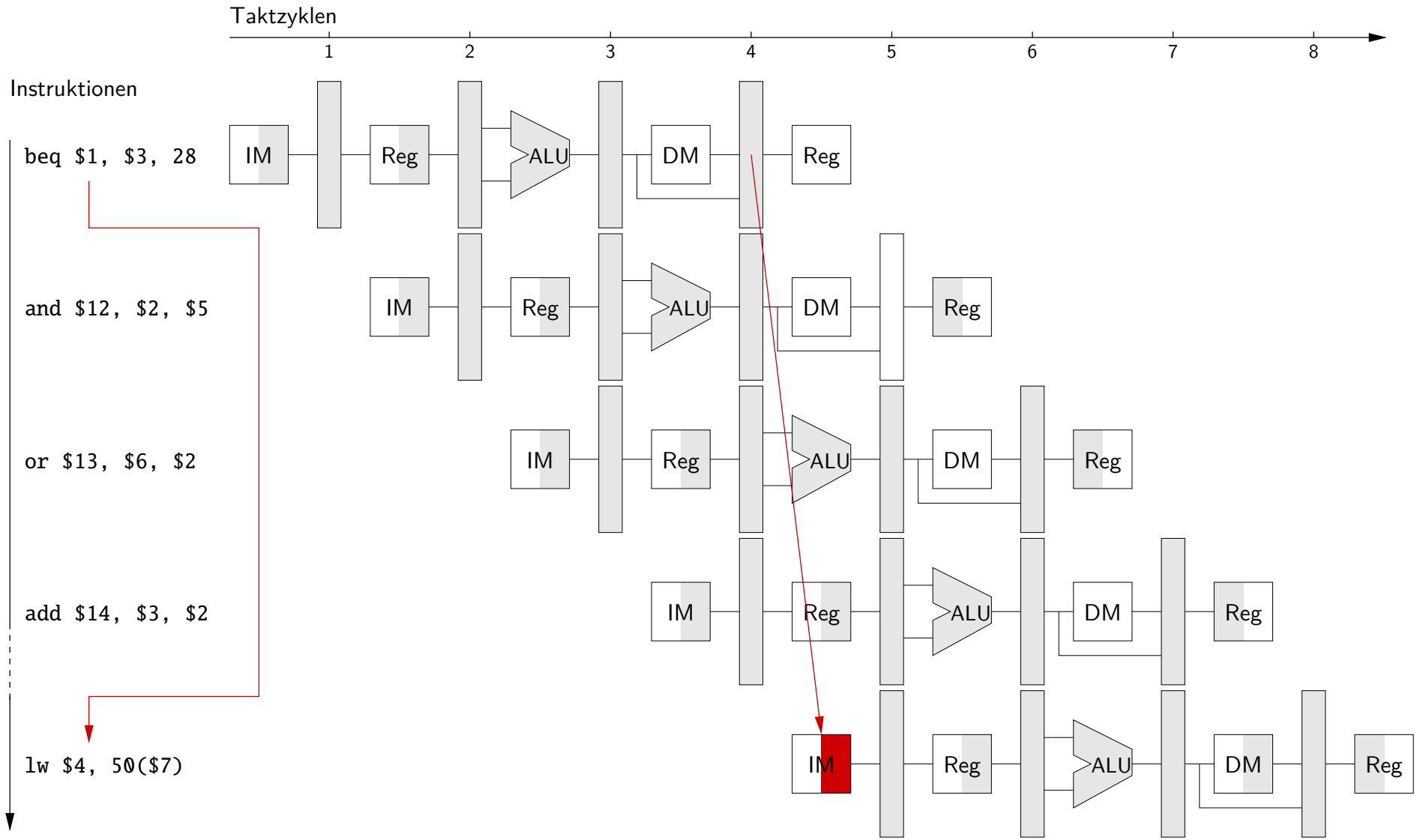
3. Sprungvorhersage („*branch prediction*“)
   - Beobachtung: ein Fall tritt häufiger auf;
     Schleifendurchlauf, Datenstrukturen durchsuchen etc.
   - mehrere Vorhersageverfahren; oft miteinander kombiniert
   + hohe Trefferquote: bis 90 %

   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.)

- 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
    $\Rightarrow$ Pipeline kann optimal ausgenutzt werden


Taktzyklen
1
2
3
4
5
6
7
8
Instruktionen
beq $1, $3, 28
and $12, $2, $5
or $13, $6, $2
add $14, $3, $2
lw $4, 50($7)
IM
Reg
ALU
DM
Reg

Pipeline – Zusammenfassung

- 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/OF$\rightarrow$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/Progammierer sollten CPU Pipeline kennen

Literatur

[PH14] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware/Software Interface*.
5th edition, Morgan Kaufmann Publishers Inc., 2014.
ISBN 978–0–12–407726–3

[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

Literatur (cont.)

[BO14] R.E. Bryant, D.R. O'Hallaron:
Computer systems – A programmers perspective.
2nd new intl. ed., Pearson Education Ltd., 2014.
ISBN 978–1–292–02584–1. `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