Rechnerarchitekturen 1*

Einführung

Prof. Dr. Alexander Auch

*Teilweise entnommen aus “Mikrocomputercomputertechnik 1” von Prof.Dr-Ing. Ralf Stiehler, sowie Patterson & Hennessy

Ziele der Veranstaltung

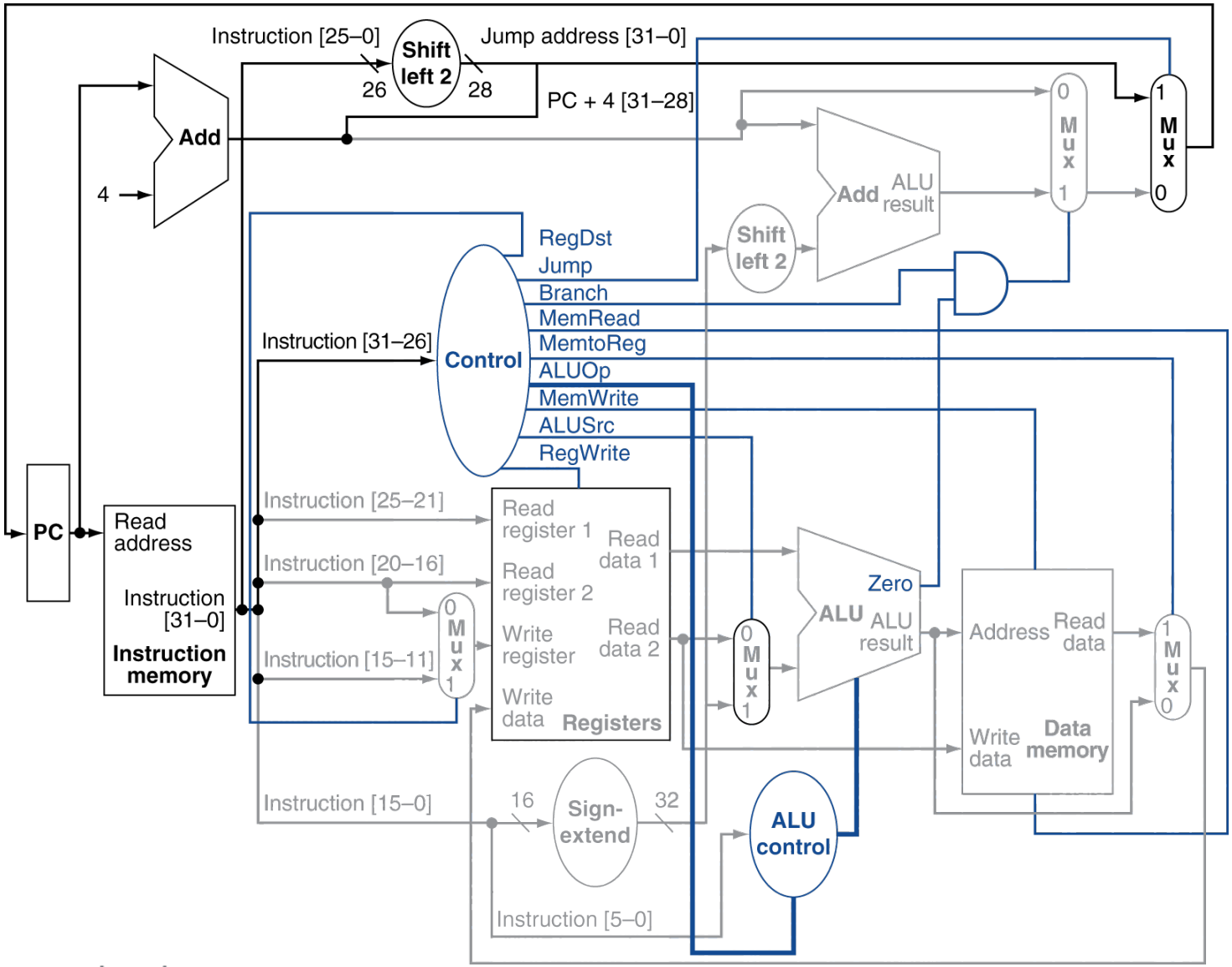
- **Rechnerentwurf**:
  - Prozessor, Speicher, Ein-/Ausgabe
  - Entwurfs- und Optimierungsmöglichkeiten

- **Prozessorentwurf**:
  - Befehlsverarbeitung
  - Entwurfs- und Optimierungsmöglichkeiten
    Multi-Cycle, Pipelining, Superskalare Architekturen

- **Assemblerprogrammierung**:
  - im MIPS-Simulator MARS

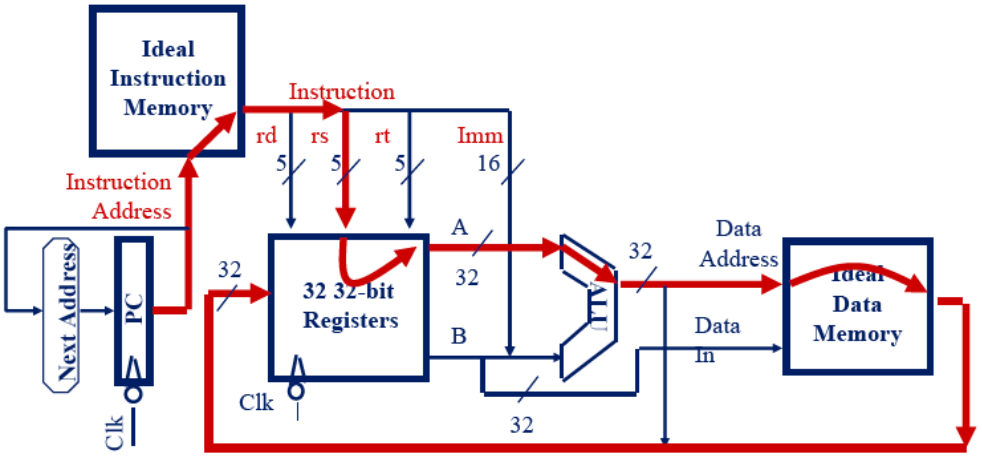
Mikrocomputertechnik 1 – Multi Cycle

Vollständiger Single-Cycle- Datenpfad mit Steuerwerk



Nachteile der Single Cycle CPU

⇒ Taktzykluszeit muss sich am sog. „kritischen Pfad“ der kombinatorischen Einheiten orientieren, darunter versteht man den längsten Pfad, entlang dessen sich die Gatterlaufzeiten addieren



⇒ Kritischer Pfad hier : Gatterlaufzeiten für Load Word sind am längsten

⇒ Load-Befehl verhindert höhere Taktfrequenz

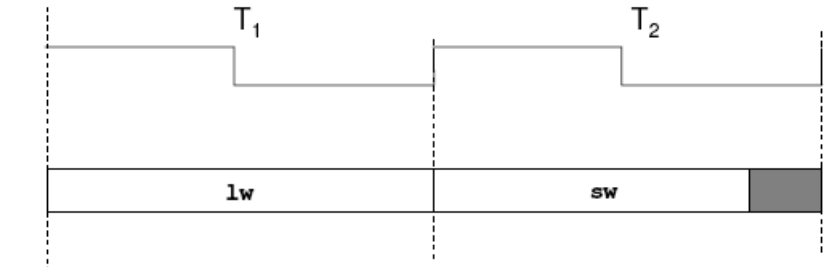
⇒ einige Hardwareblöcke müssen mehrfach vorhanden sein, da jede Einheit in einem Taktzyklus nur einmal verwendet werden kann.

Multi-Cycle CPUs

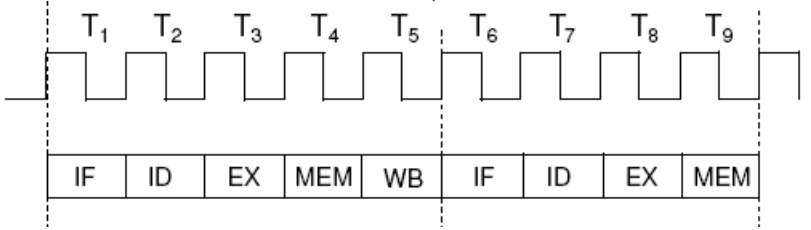
Cycle und Multi Cycle CPU

Man unterscheidet
⇒ Single Cycle CPU (jeder Befehl ein Taktzyklus) und
⇒ Multi Cycle CPU (mehrere Taktzyklen pro Befehl bei höherer Taktfrequenz)
⇒ CPU mit Pipeline-Strukturen

Single Cycle CPU =>



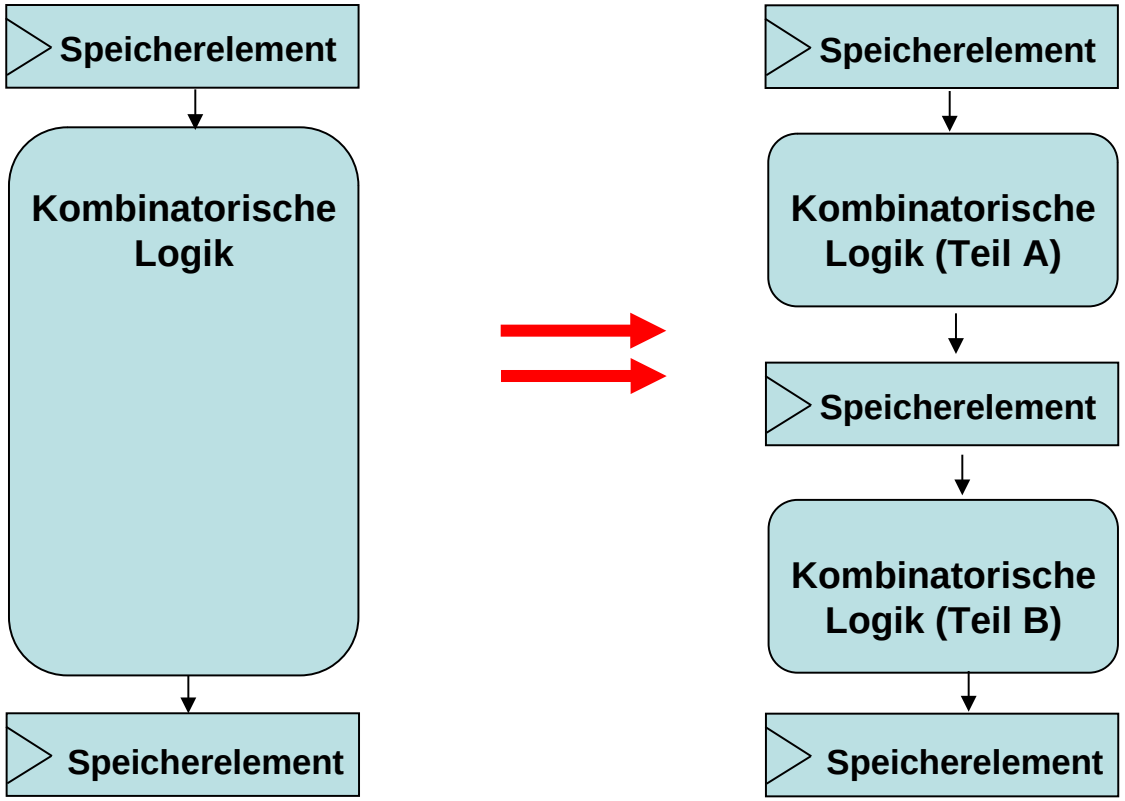
Multi Cycle CPU =>



Übergang vom Single-Cycle zum Multi-Cycle-Datenpfad:

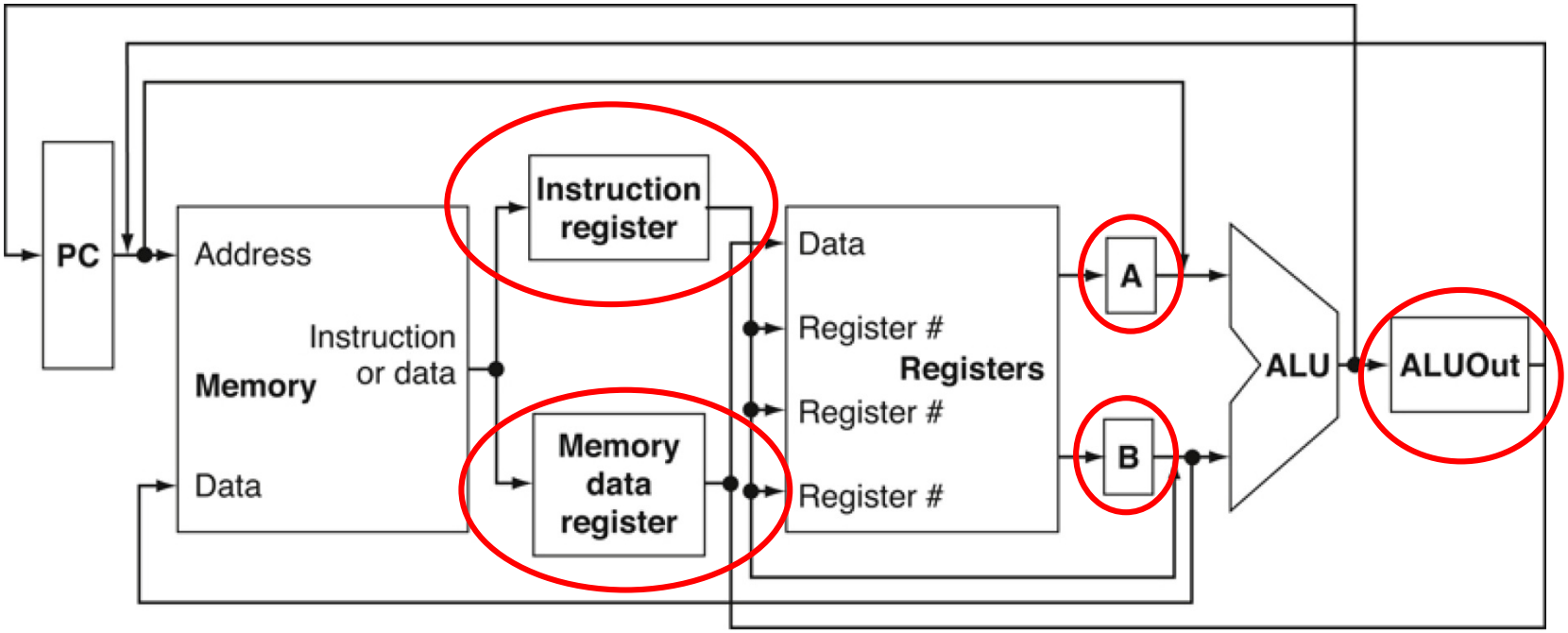
⇒ Aufbrechen kombinatorischer Pfade und Einsetzen von Registern

⇒ Durchführung eines Befehls in mehreren kurzen anstelle in einem langen Taktzyklus

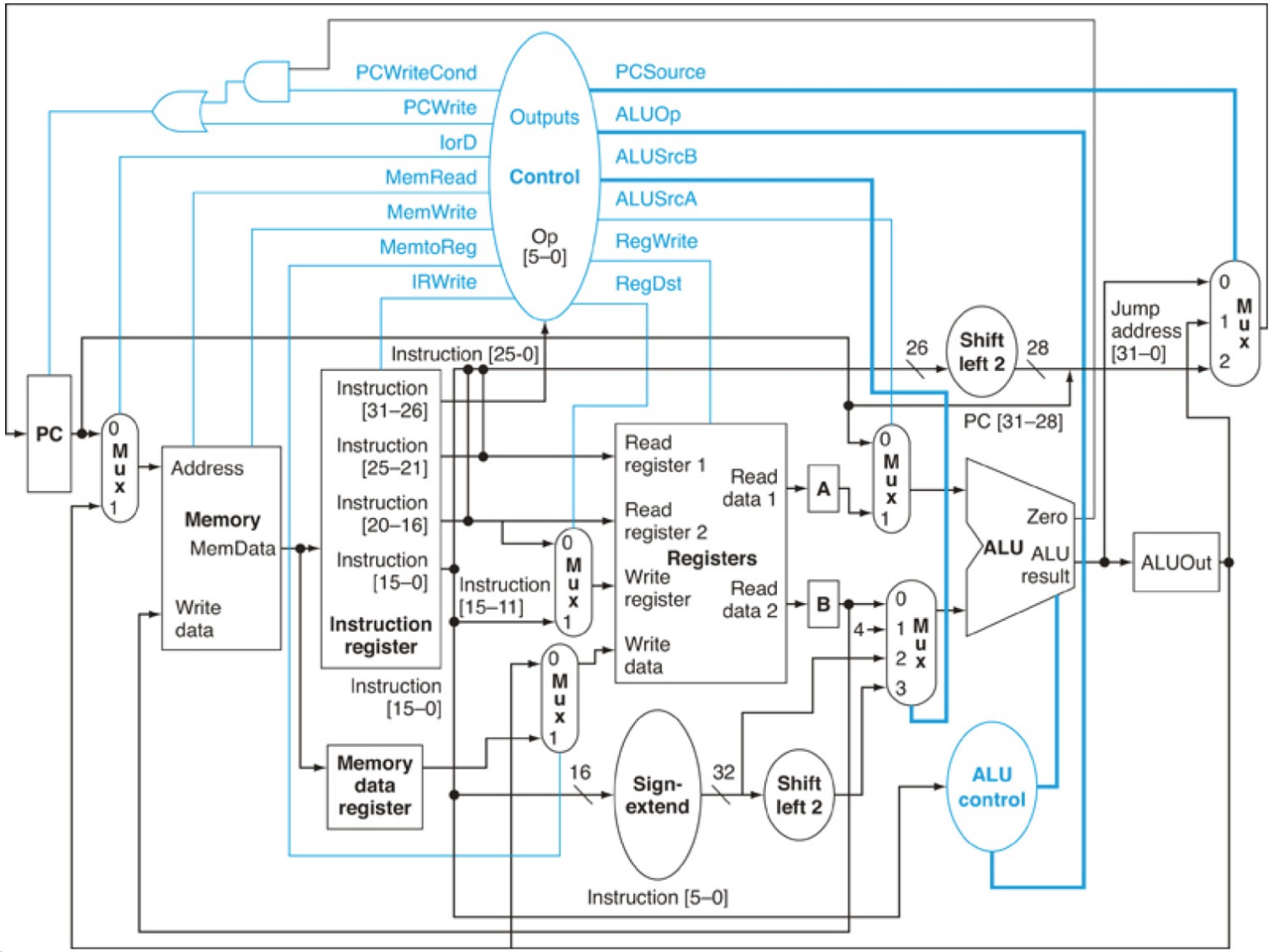


Mikrocomputertechnik 1 – Multi Cycle

- Speicherelemente um Status zwischen Taktzyklen zu speichern
- Eine einzige ALU, die für alle Berechnungen verwendet wird
- Eine einzige Memory-Unit für Instruktionen und Daten
- Befehle umfassen nun mehrere Zyklen – ein Zyklus pro Ausführungsphase



Vollständiger Datenpfad für Multi Cycle

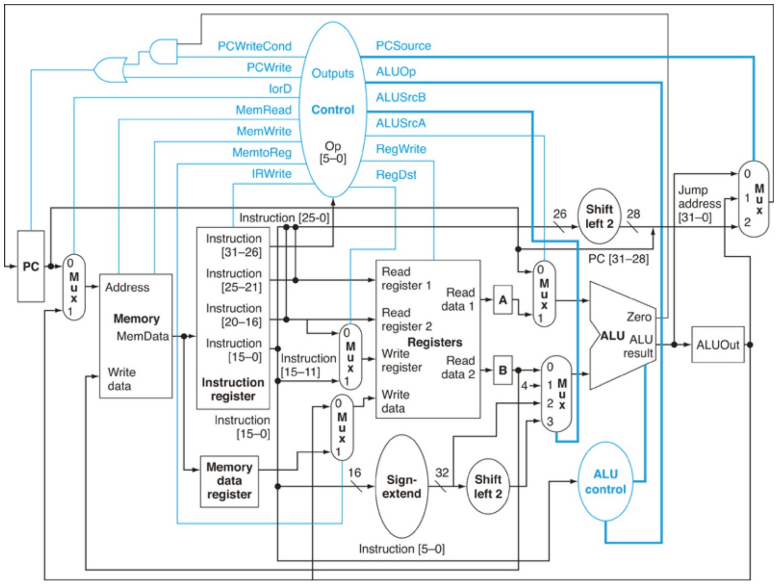


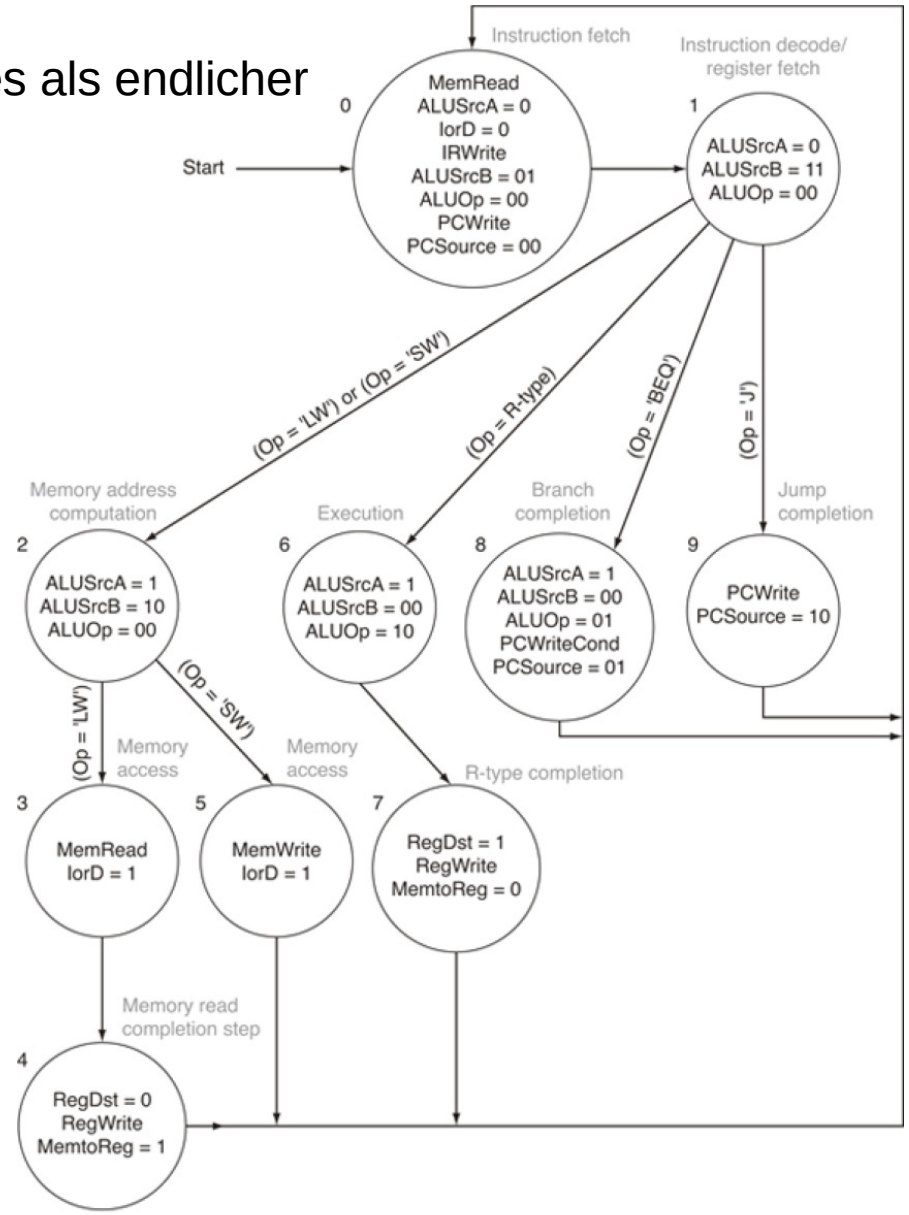
Mikrocomputertechnik 1 – Multi Cycle

Das Steuerwerk ist nun so komplex, dass es als endlicher Automat dargestellt werden muss.

Die Zustände entsprechen den Phasen

- Instruction Fetch
- Instruction Decode
- Execute
- Memory Access
- Write Back



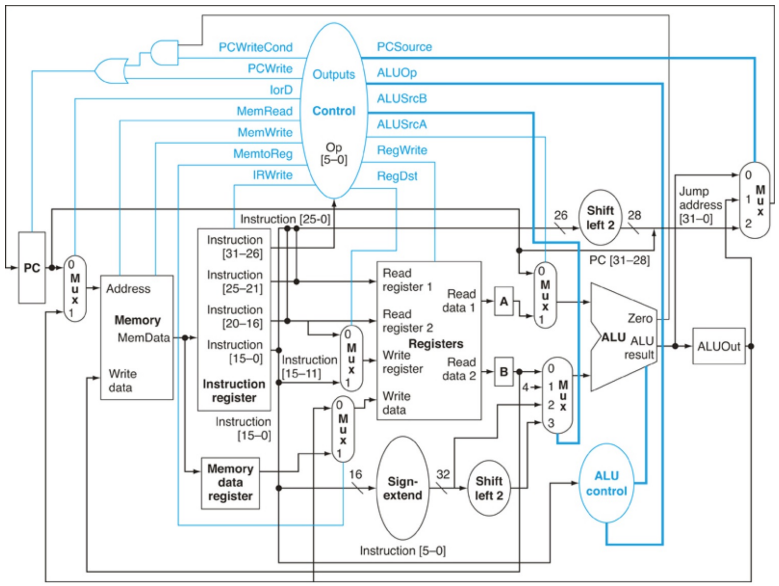


Zusammenfassung Multi-Cycle

- Hardware kann eingespart werden (ALU, Datenzugriff → teurer als Register)
- Taktzykluszeiten können nun verringert werden

- Aber: nach Moore's Law können wir mit der Zeit immer mehr Transistoren unterbringen – spielt also keine große Rolle mehr

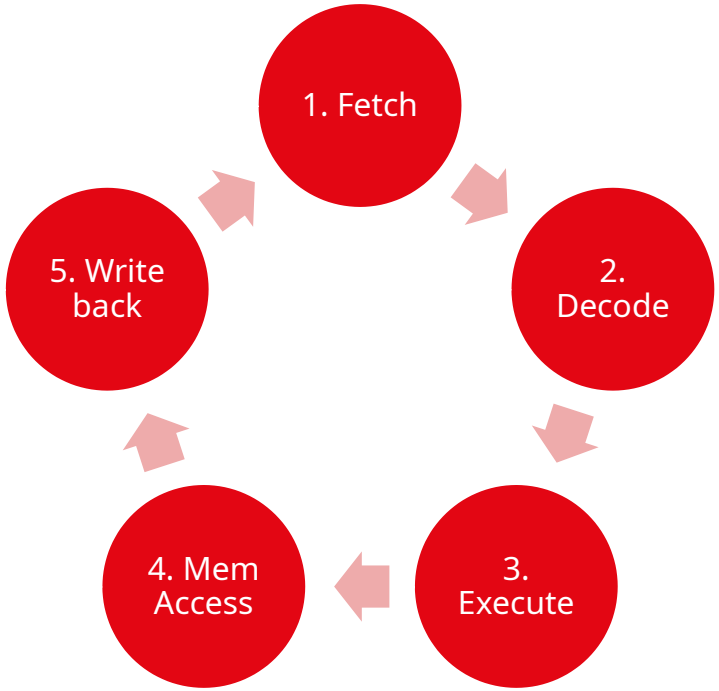
- Besser: mehr Hardware, Auslastung der Hard-ware optimieren
- → Pipelining
- → Superskalare Architekturen



Pipelining

Steuerwerk: Fetch-Decode-Execute

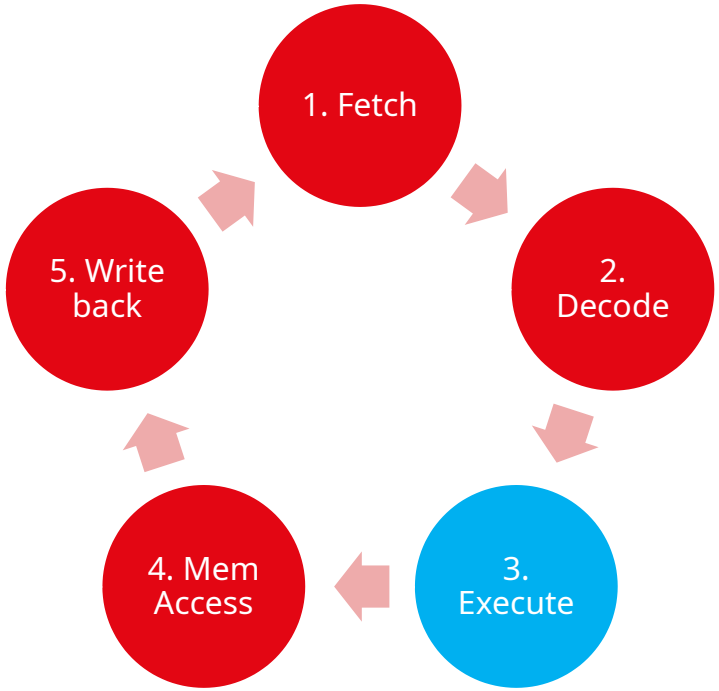
1. Hole Instruktion aus Speicher; inkrementiere Instruction-Pointer
2. Dekodiere Instruktion
3. Führe die Instruktion aus
4. Hole zugehörige Daten aus Speicher oder Schreibe in Speicher
5. Schreibe Ergebnis zurück in Register



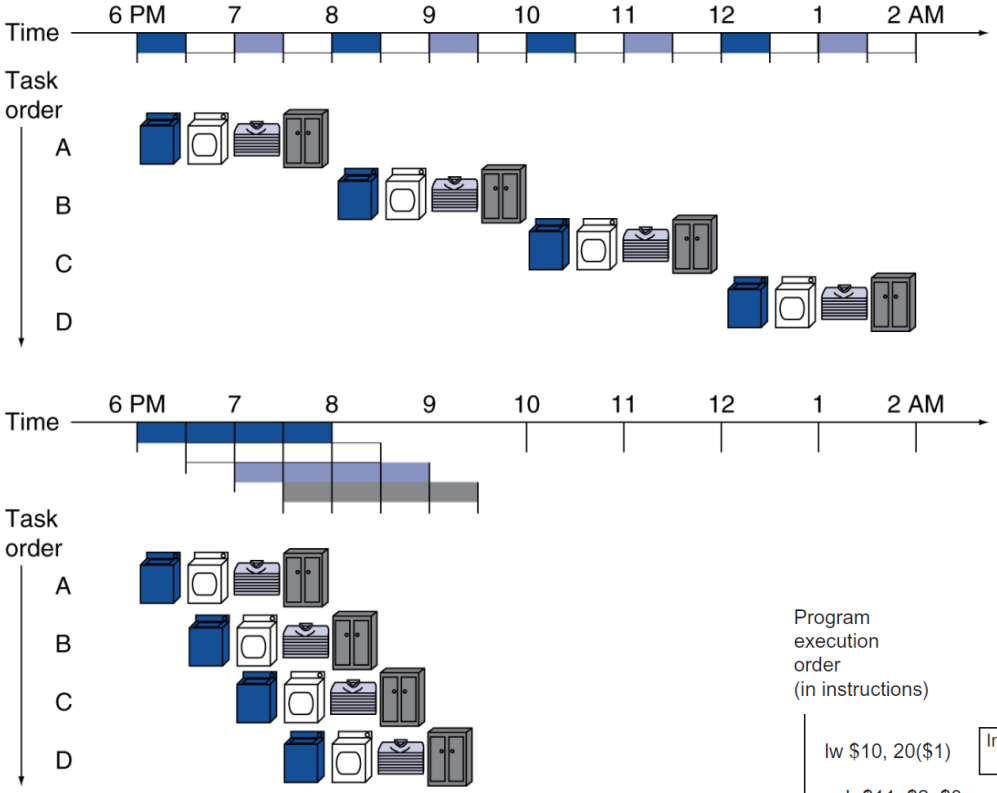
Frage: Was macht das Rechenwerk in der Zwischenzeit?

Nur in der Execute-Phase hat das Rechenwerk richtig was zu tun

Aber: Während der Execute-Phase braucht es keinen Hauptspeicherzugriff



Die Lösung: Pipelining

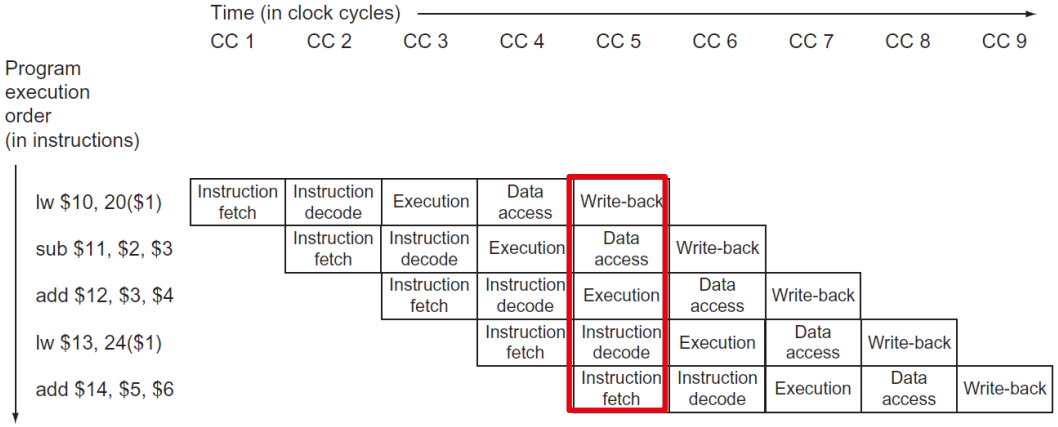


Wie bei Waschmaschine und Trockner – braucht viel mehr Hardware!

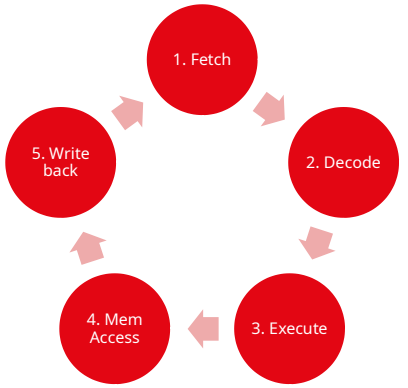
Steigerung: 16h/7h ≈ 2,3

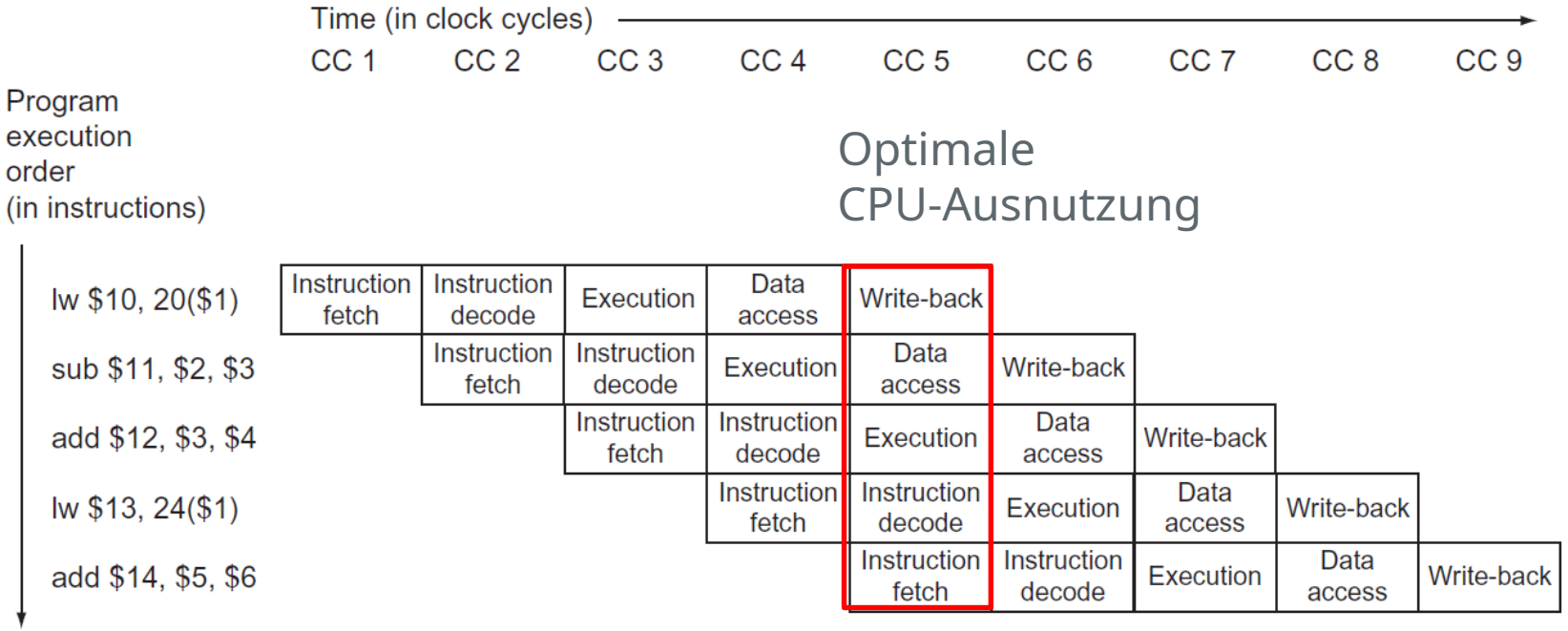
Für große n: Faktor 4

Ebenso mit CPU:



Pipelining

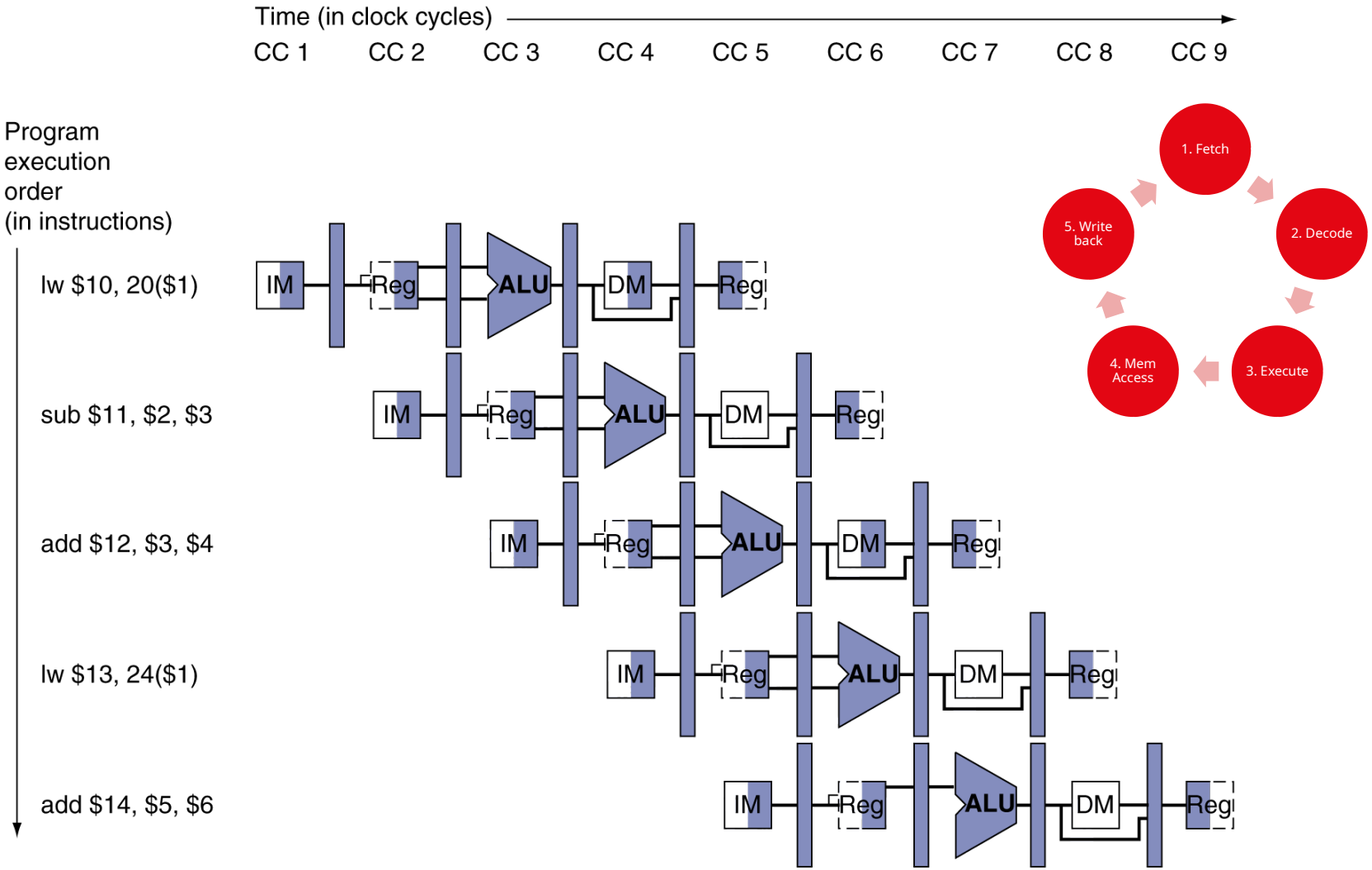




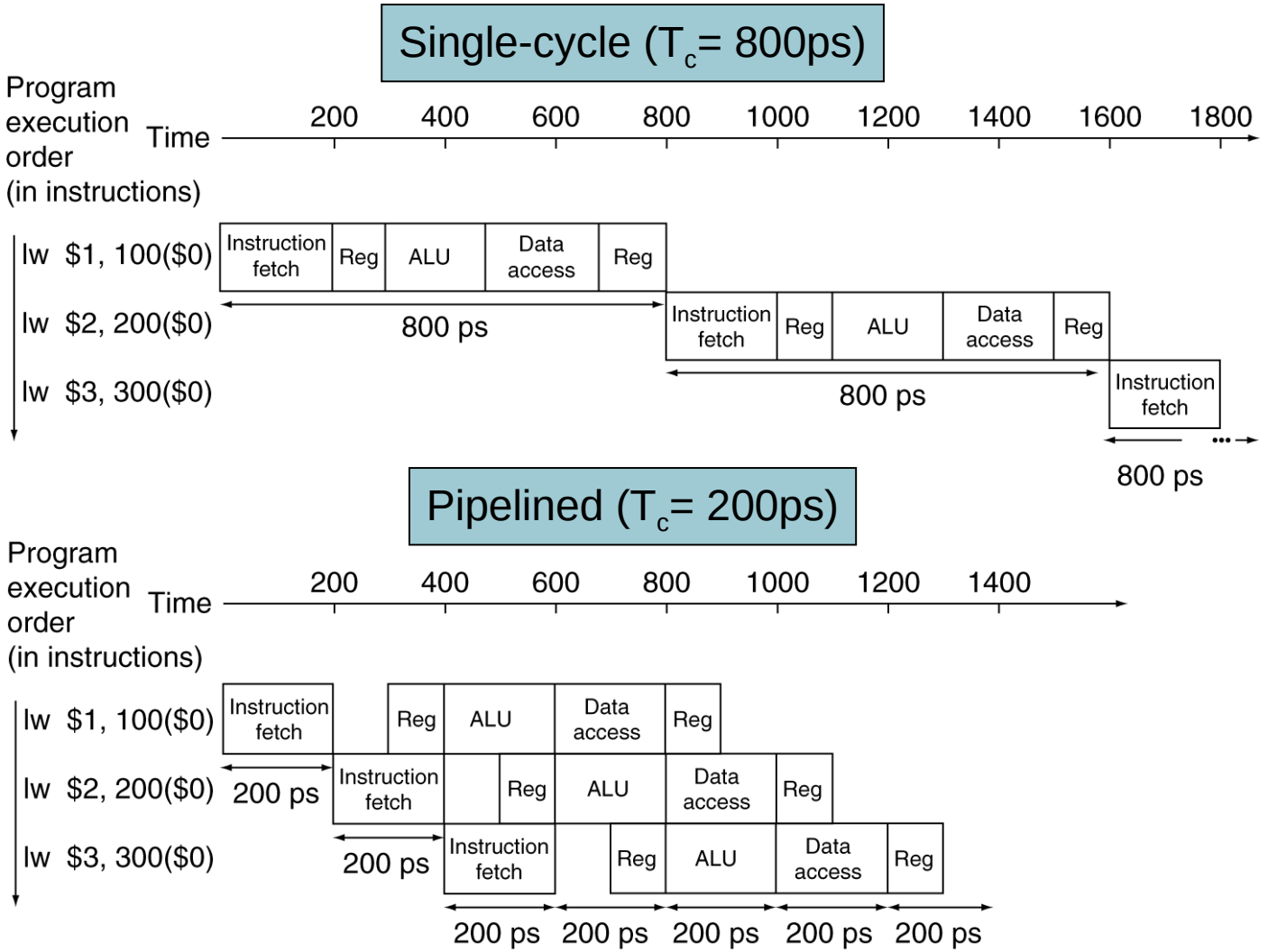
Optimale CPU-Ausnutzung

Patterson/Hennessy

Pipelining

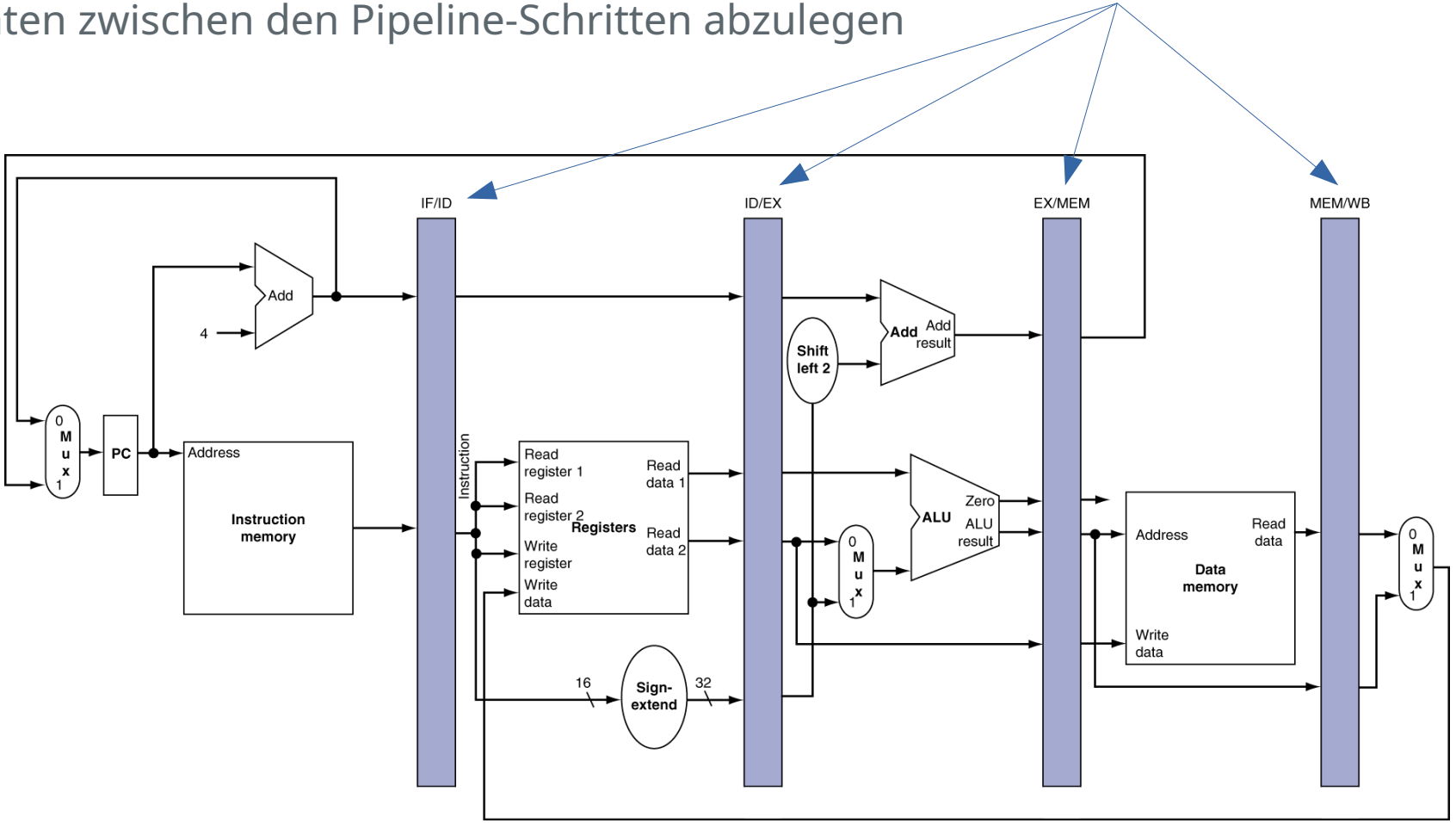

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
IM
Reg
ALU
DM
Reg
1. Fetch
2. Decode
3. Execute
4. Mem Access
5. Write back

Pipelining Performance

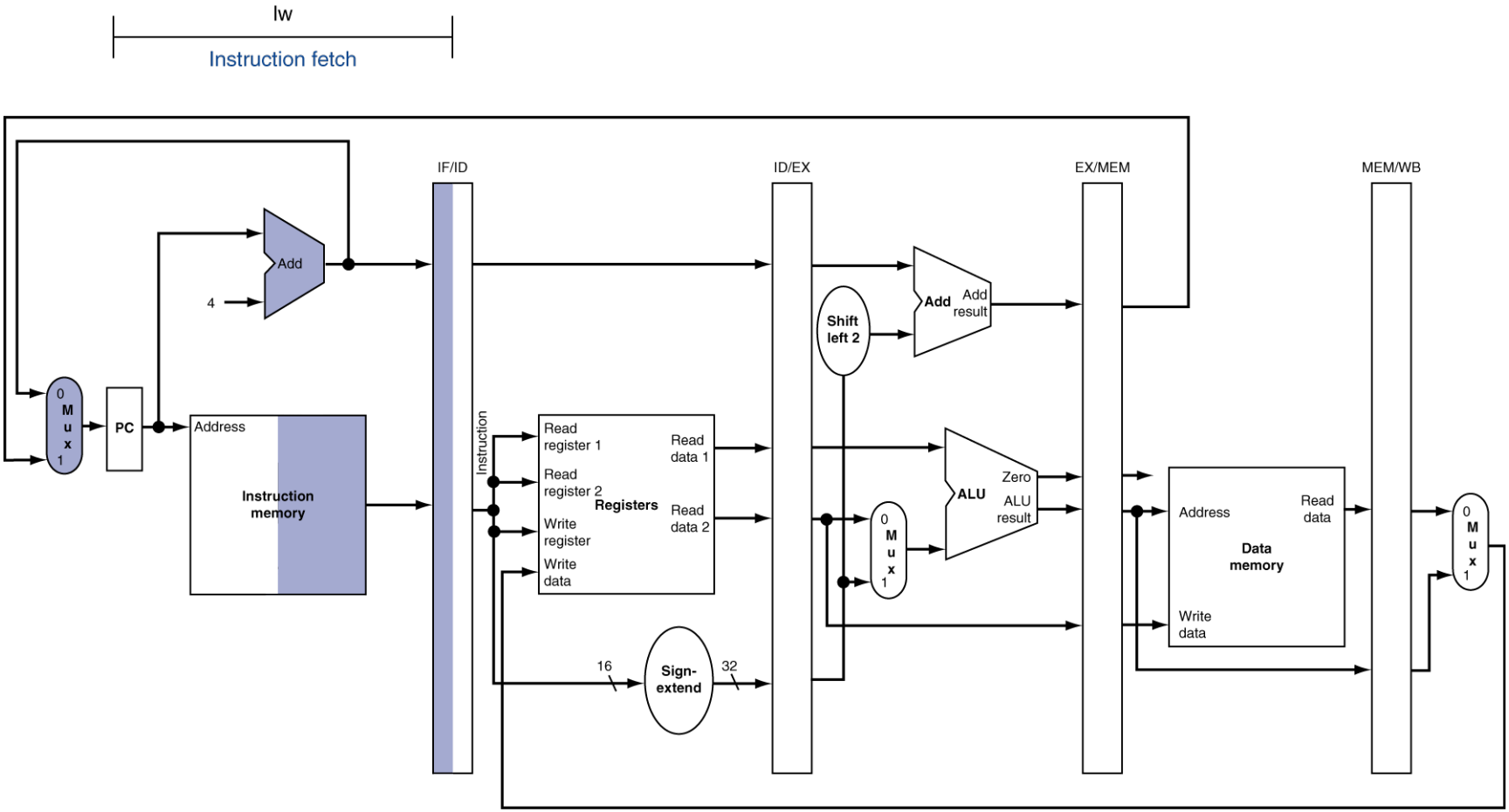


Datenpfad für Pipelined CPU

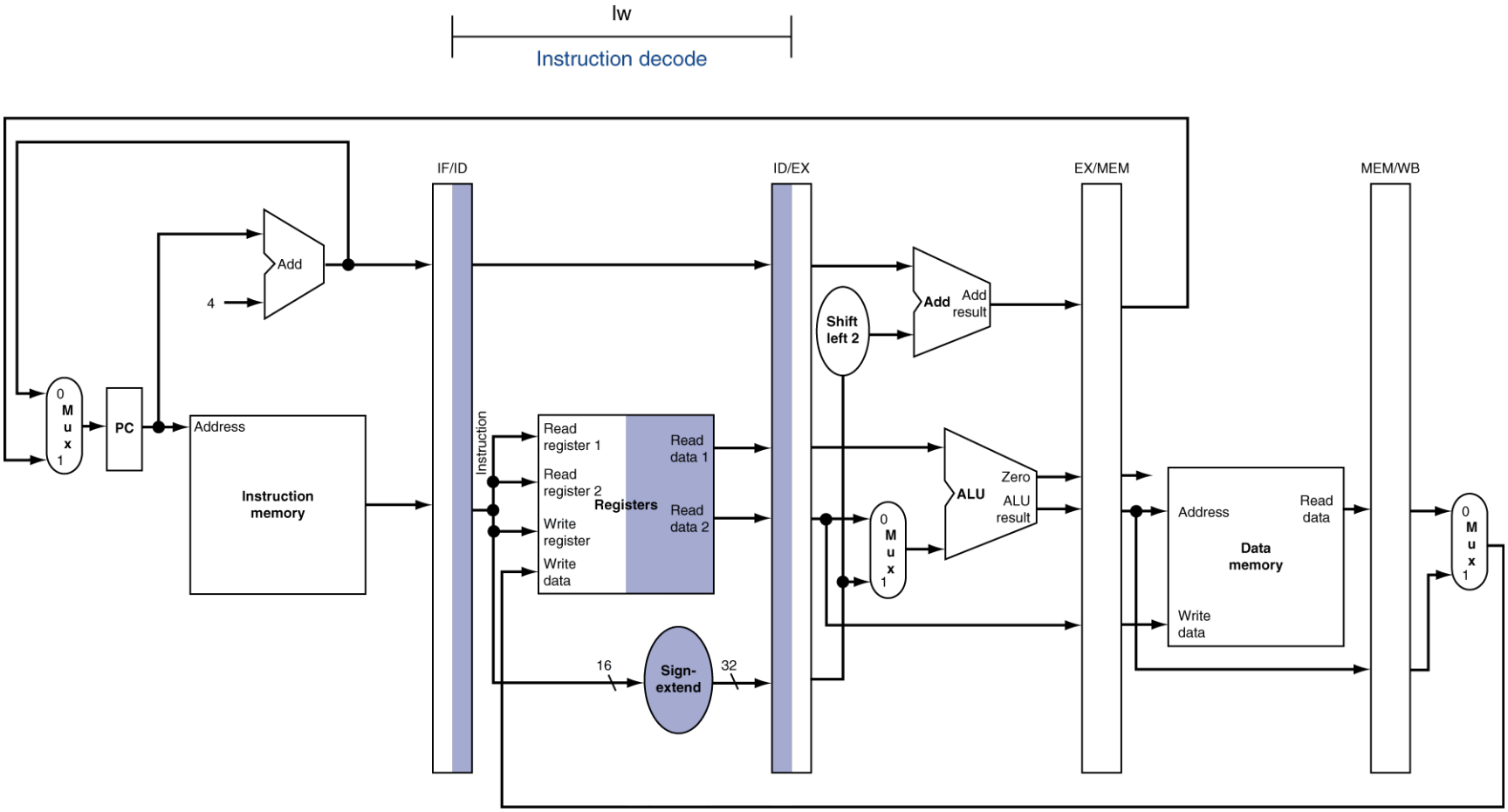
Analog zur Multi-Cycle-CPU benötigen wir wieder spezielle Register um Daten zwischen den Pipeline-Schritten abzulegen



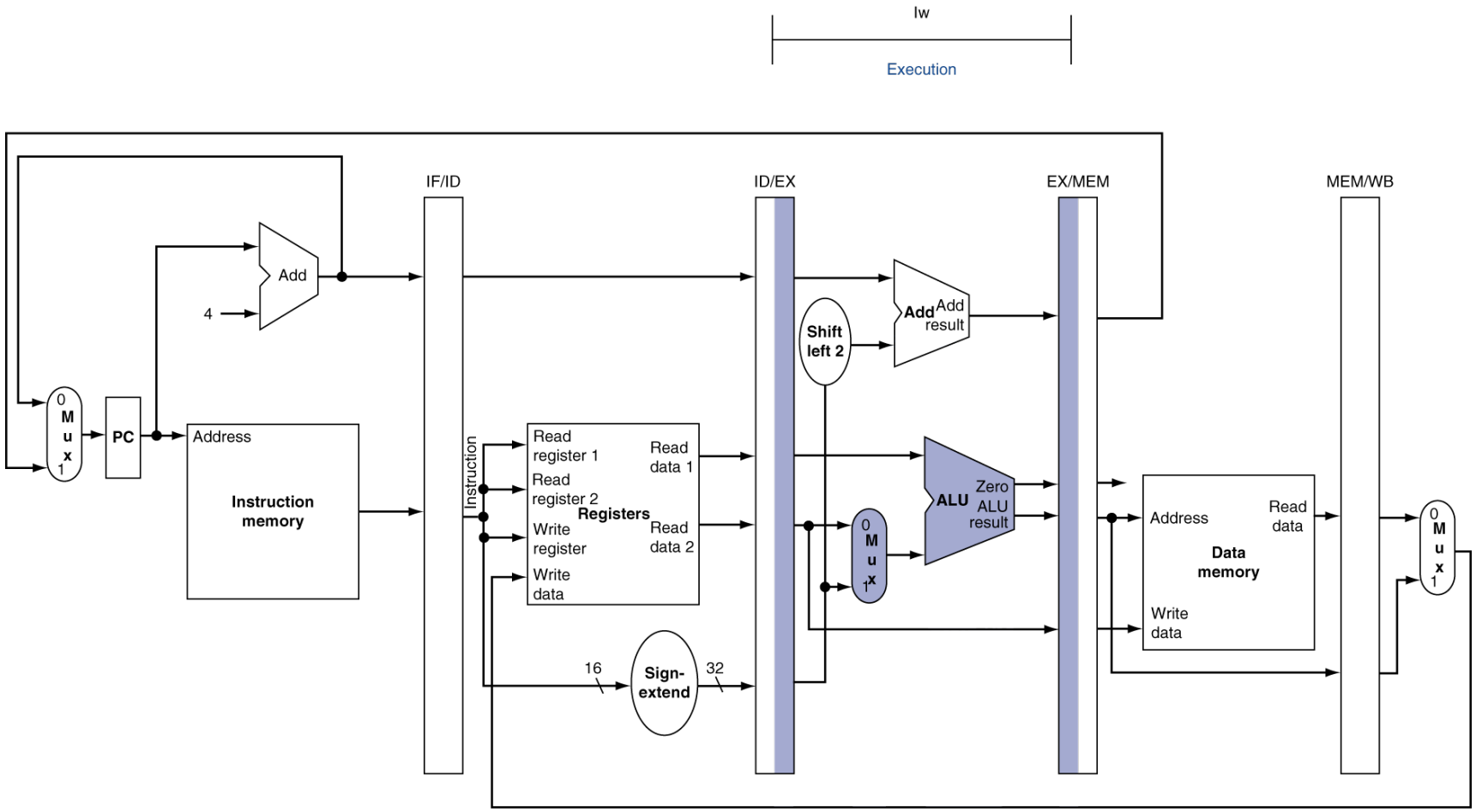
Beispiel-Durchgang für lw/sw Instruction Fetch



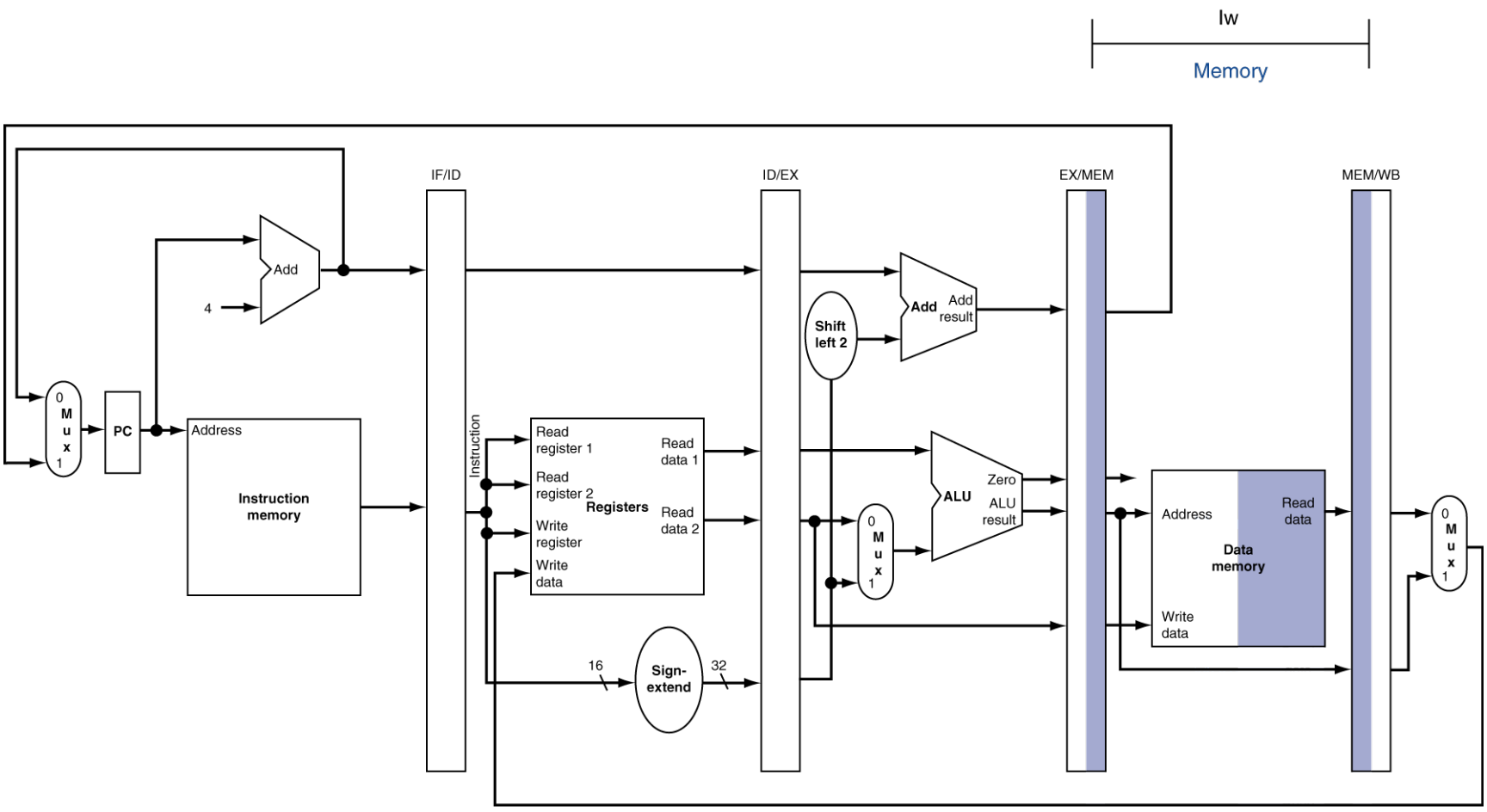
Beispiel-Durchgang für lw/sw Instruction Decode



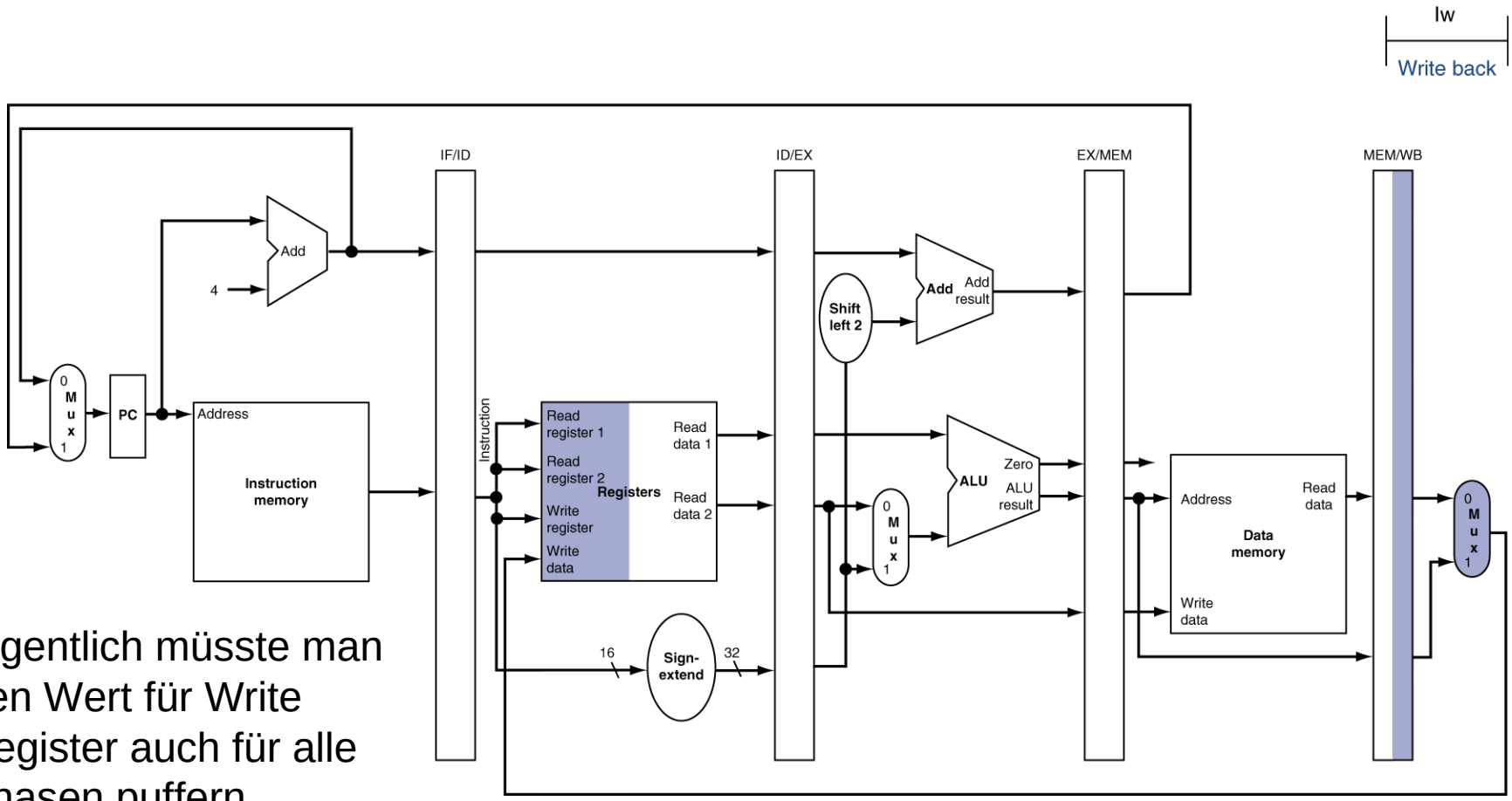
Beispiel-Durchgang für lw/sw Execute



Beispiel-Durchgang für lw/sw Memory Access

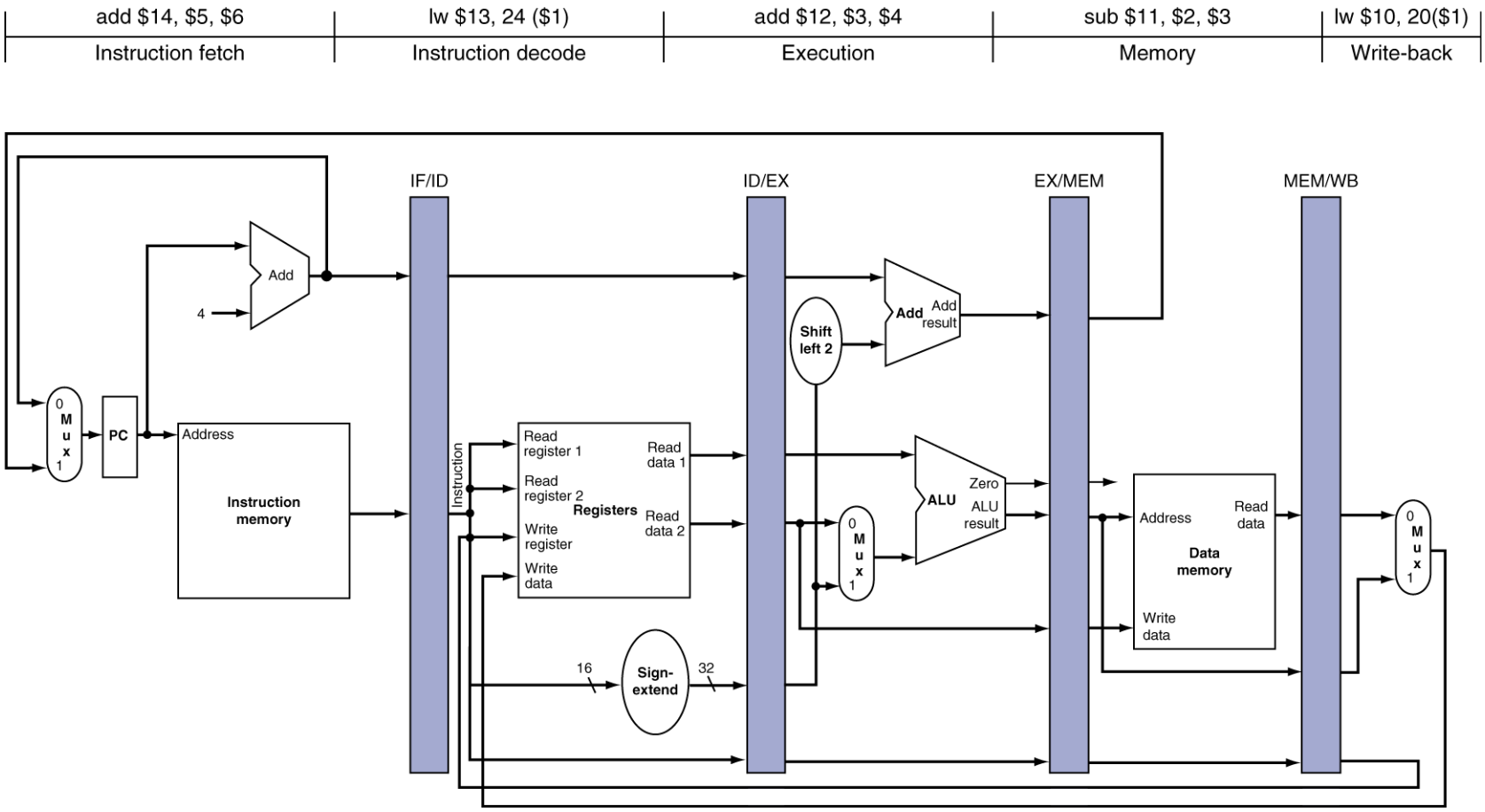


Beispiel-Durchgang für lw/sw Write Back



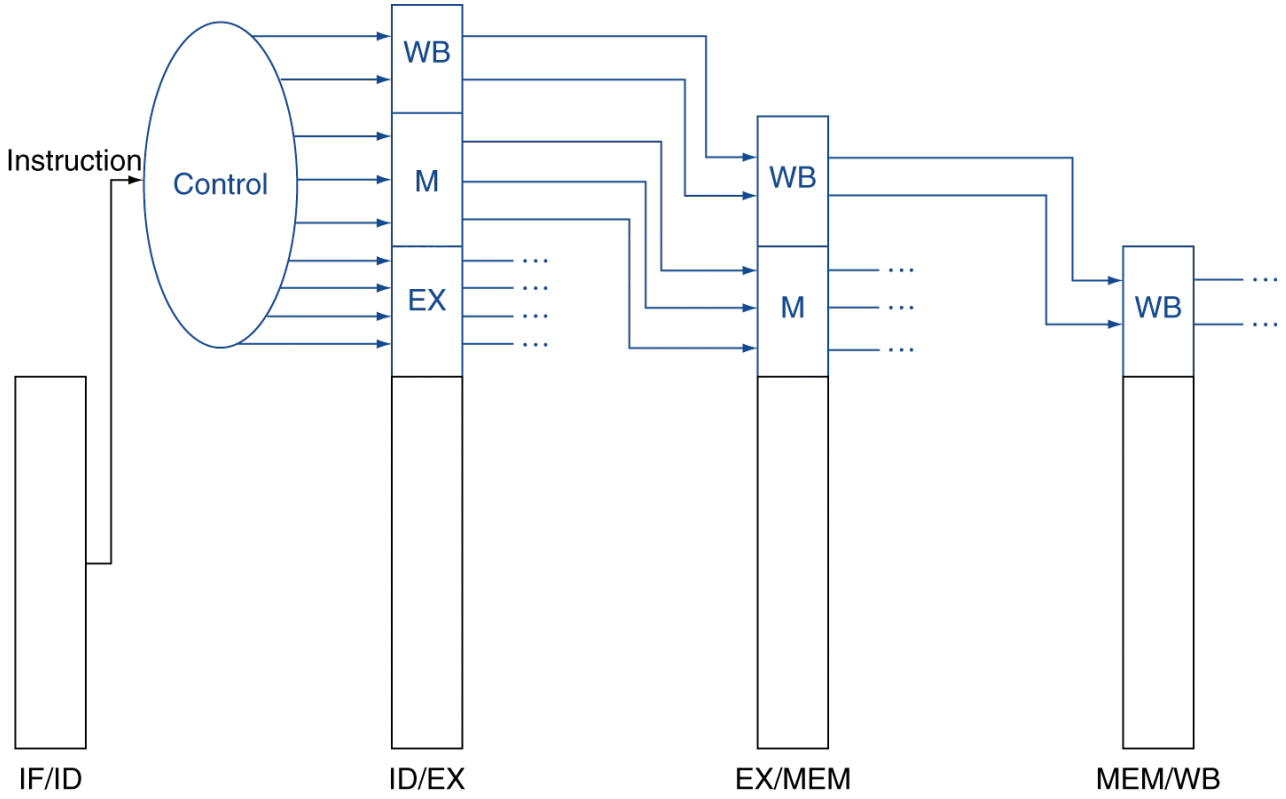
Eigentlich müsste man den Wert für Write Register auch für alle Phasen puffern...

Beispiel-Durchgang mit mehreren Befehlen

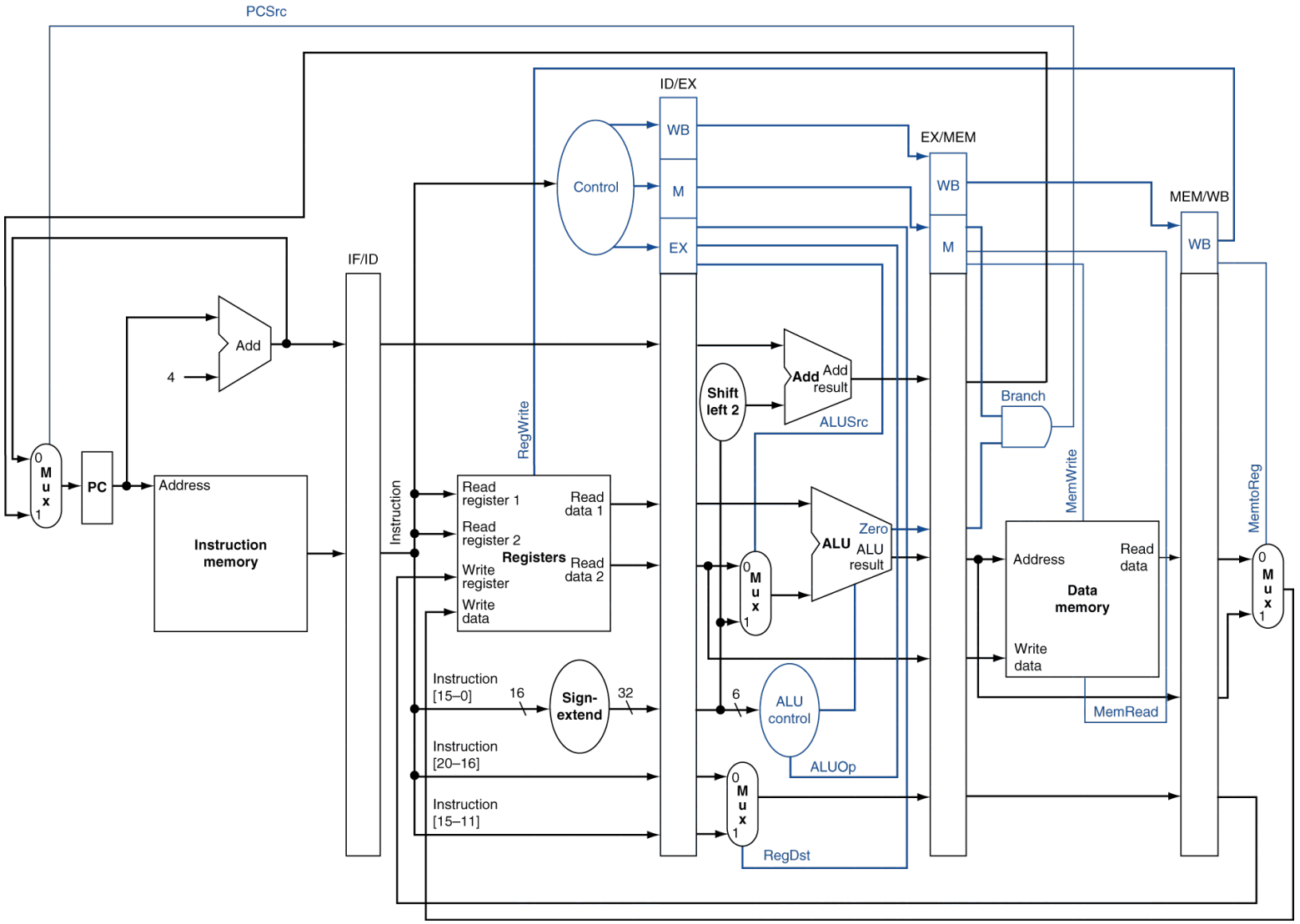


Steuerwerk

Die Signale für spätere Stages müssen mitgeschleppt werden…
Auch das Steuerwerk muss wieder für die nächste Instruktion „frei“ werden



Steuerwerk + Datenpfad



Probleme

Structure Hazard:
Ressource ist belegt, z.B. ALU oder Mem
→ braucht separaten Bus für Daten / Instruktionen, oder zumindest Cache

Data Hazard:
Anweisungen hängen voneinander ab:

- `x1 = x2 + x3`
- `x4 = x1 + x5`
- `x6 = x1 + x4`

Die CPU „ratet“, was als nächstes ausgeführt wird
→ Sprungvorhersage

Control Hazard:
Sprünge im Kontrollfluss:

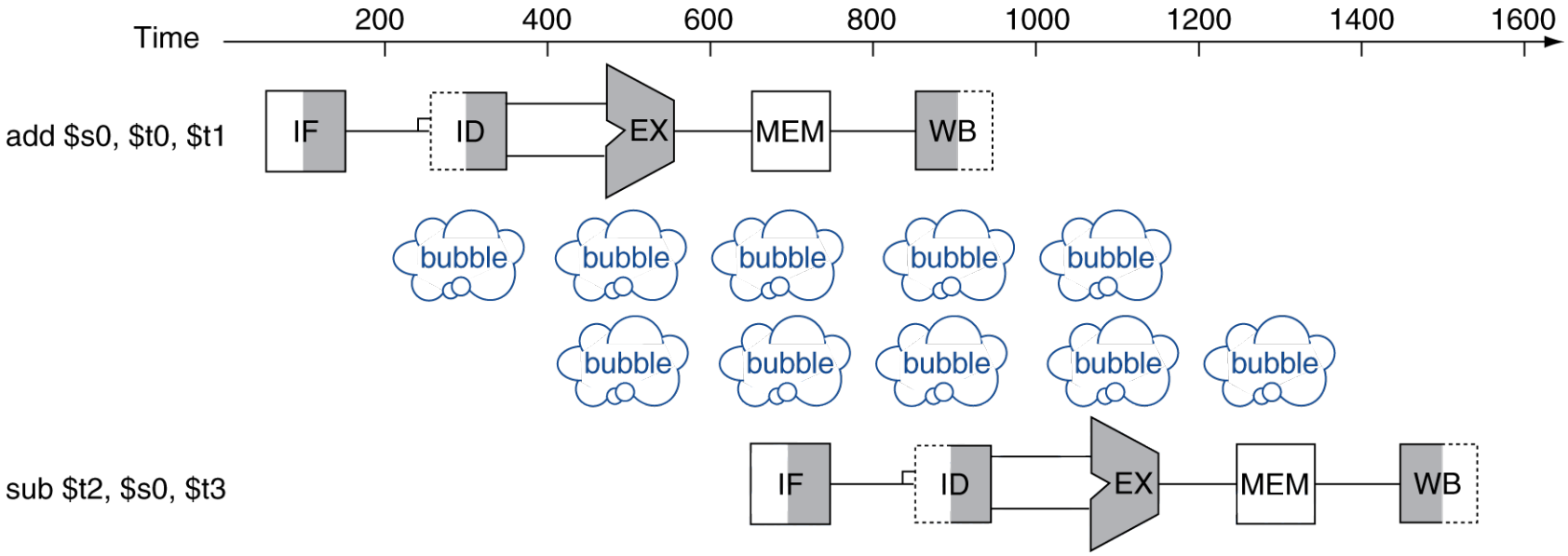
```
if(x1 % 2 == 0)
  { … }
else
  {… }
```

Welcher Zweig wird ausgeführt?

Data Hazards

Instruktion hängt von der Ausführung einer vorherigen Instruktion ab:

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

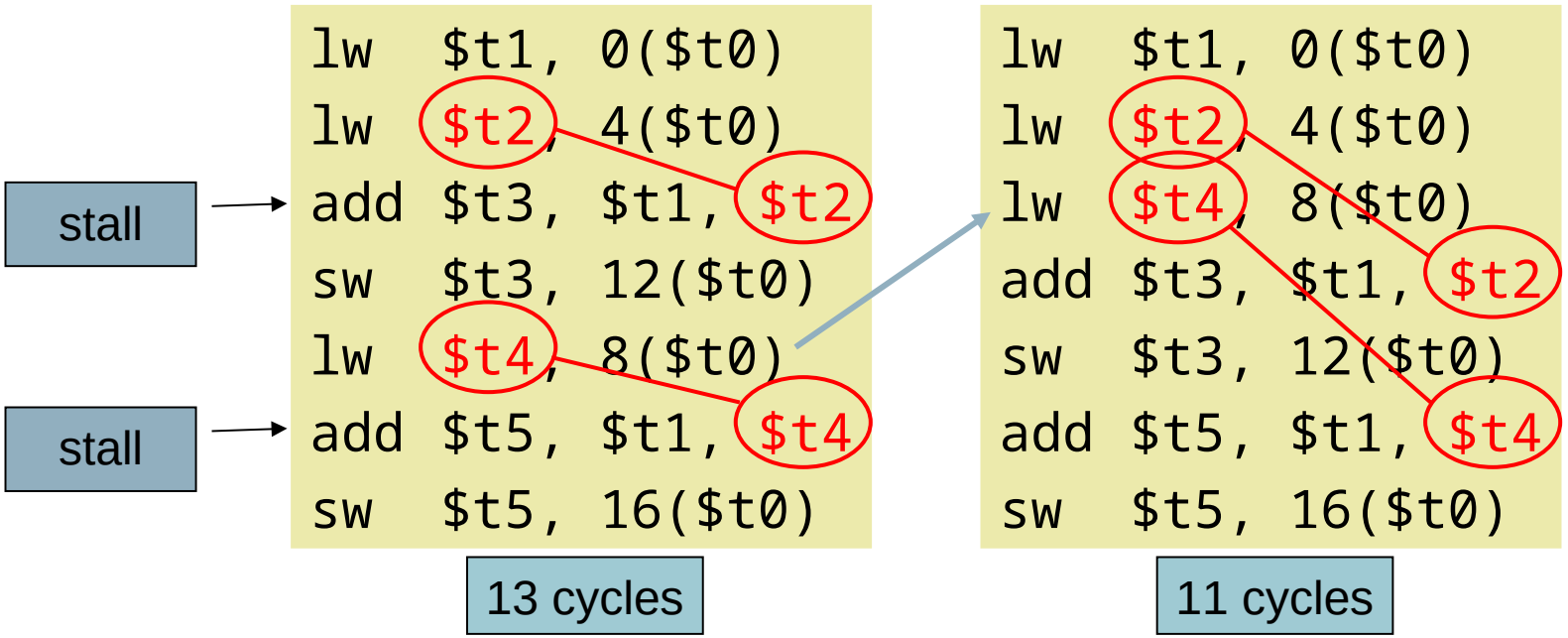


Data Hazards durch den Compiler vermeiden

Durch Ändern der Anweisungsreihenfolge können „Stalls“ vermieden oder zumindest verringert werden

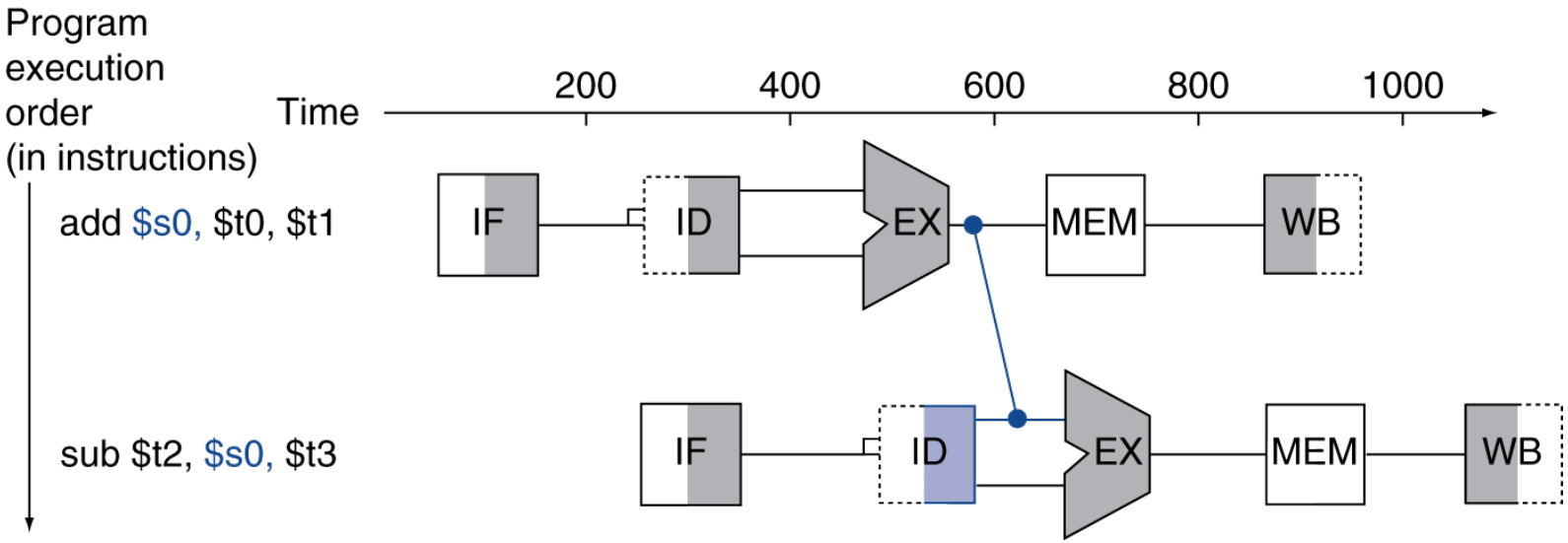
Beispiel:

```
A = B + E; C = B + F;
```



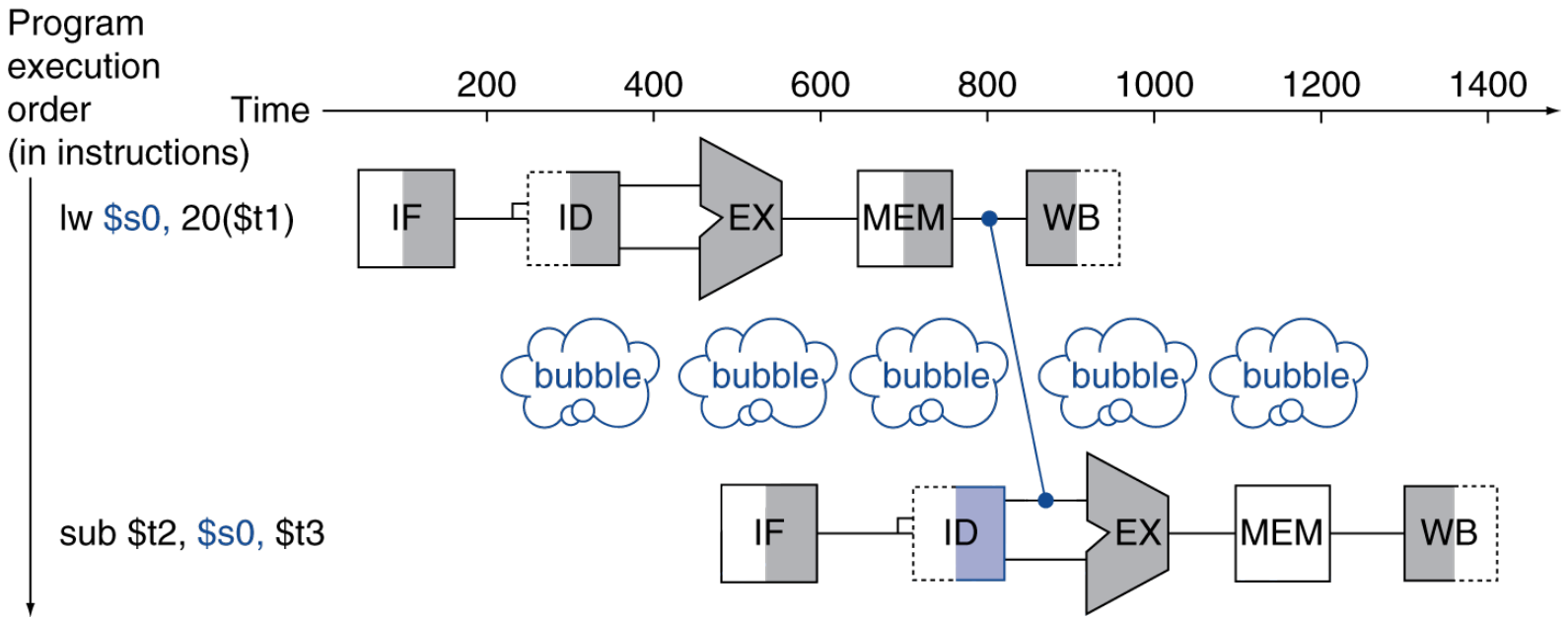
Data Hazards beheben durch Forwarding

Das Ergebnis wird weitergereicht (eigentlich „zurückgereicht“) an eine Einheit weiter vorne im Datenpfad, die aktuell schon eine andere Anweisung ausführt
→ braucht zusätzliche Verbindungen im Datenpfad



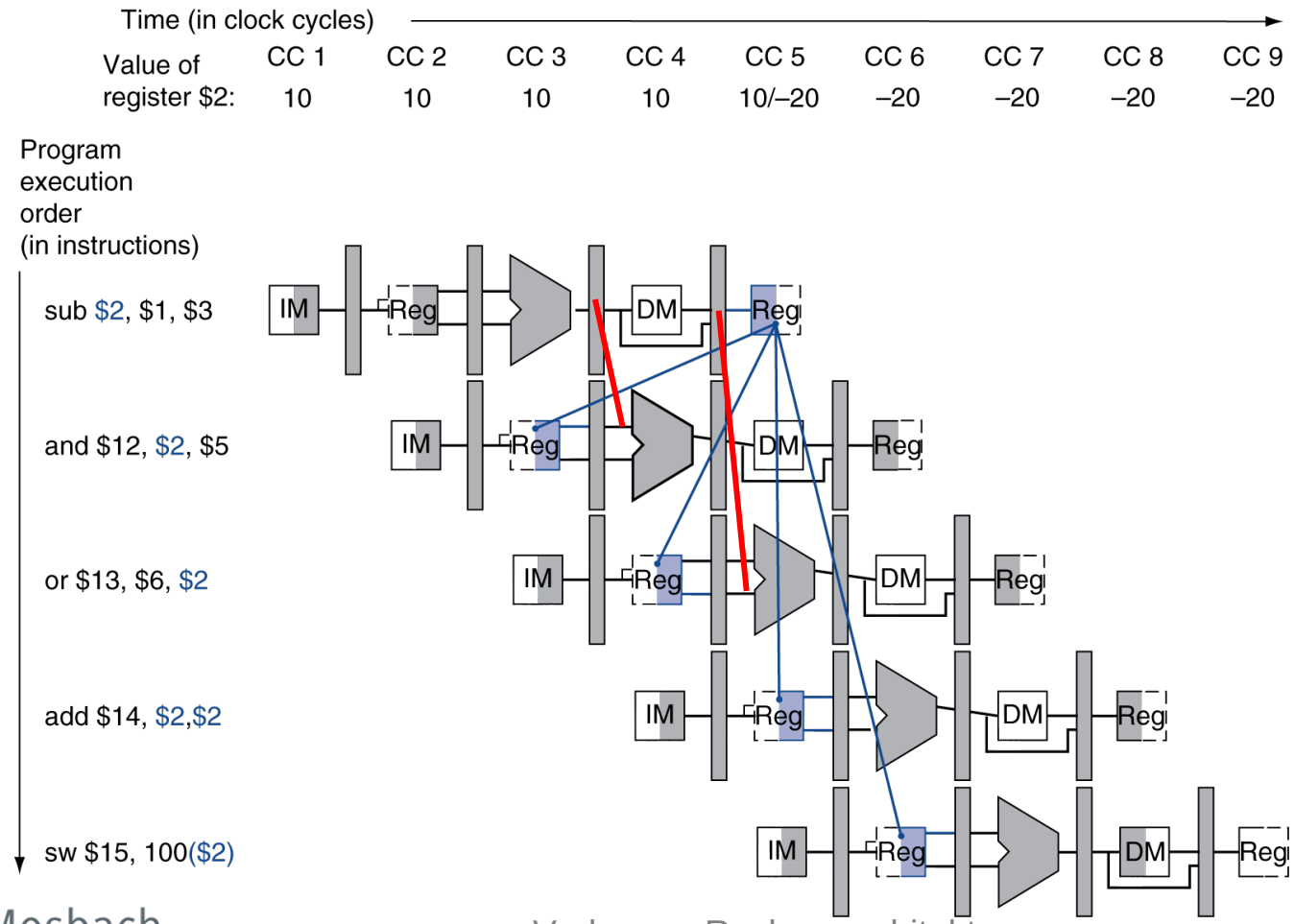
Data Hazards beheben durch Forwarding II

Ein Anhalten der Pipeline (Stall) kann aber nicht immer vermieden werden!



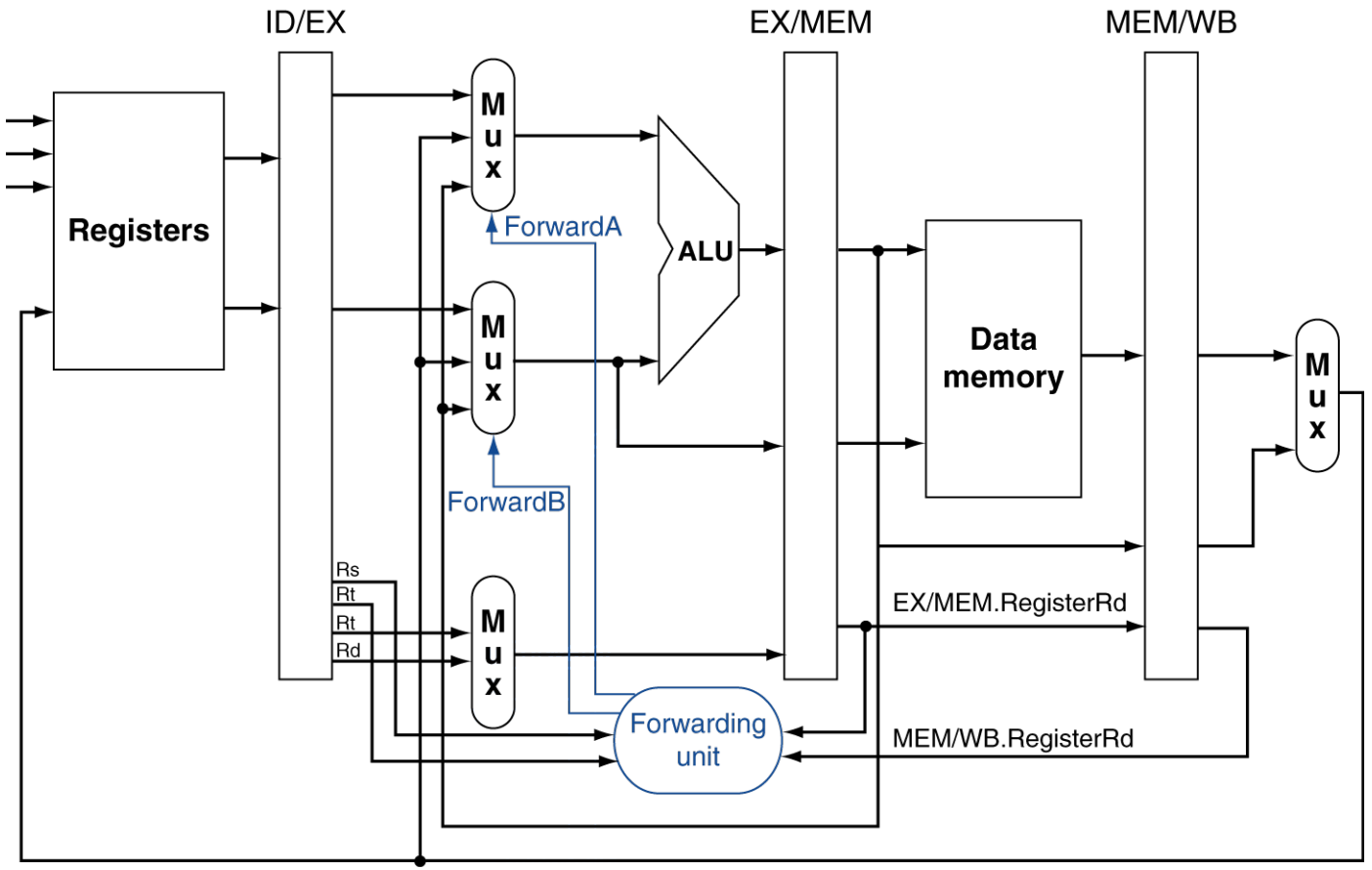
Data Hazards beheben durch Forwarding III

Problem: Ergebnis ist erst in Stage 5 im Register File.
Wir können aber die Werte aus den Pipeline-Registern „forwarden“



Forwarding-Logik im Datenpfad

Hierfür wird viel zusätzliche Kontroll-Logik (u.a. Multiplexer) benötigt...

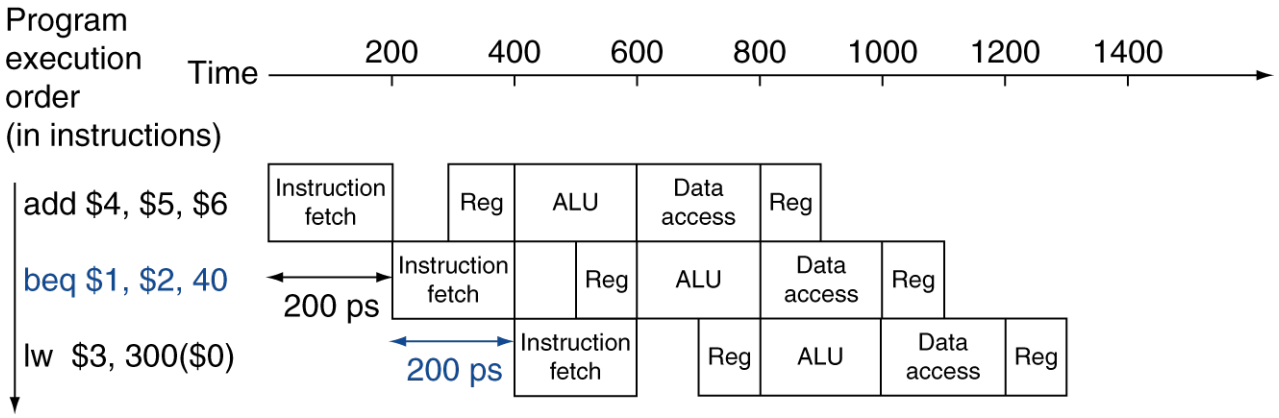


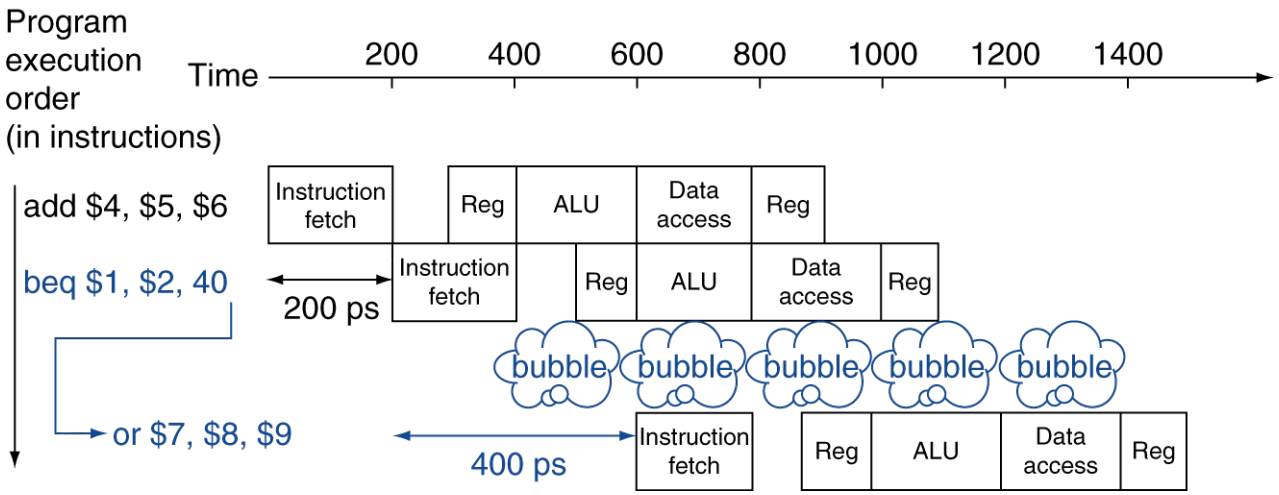
b. With forwarding

Control Hazards

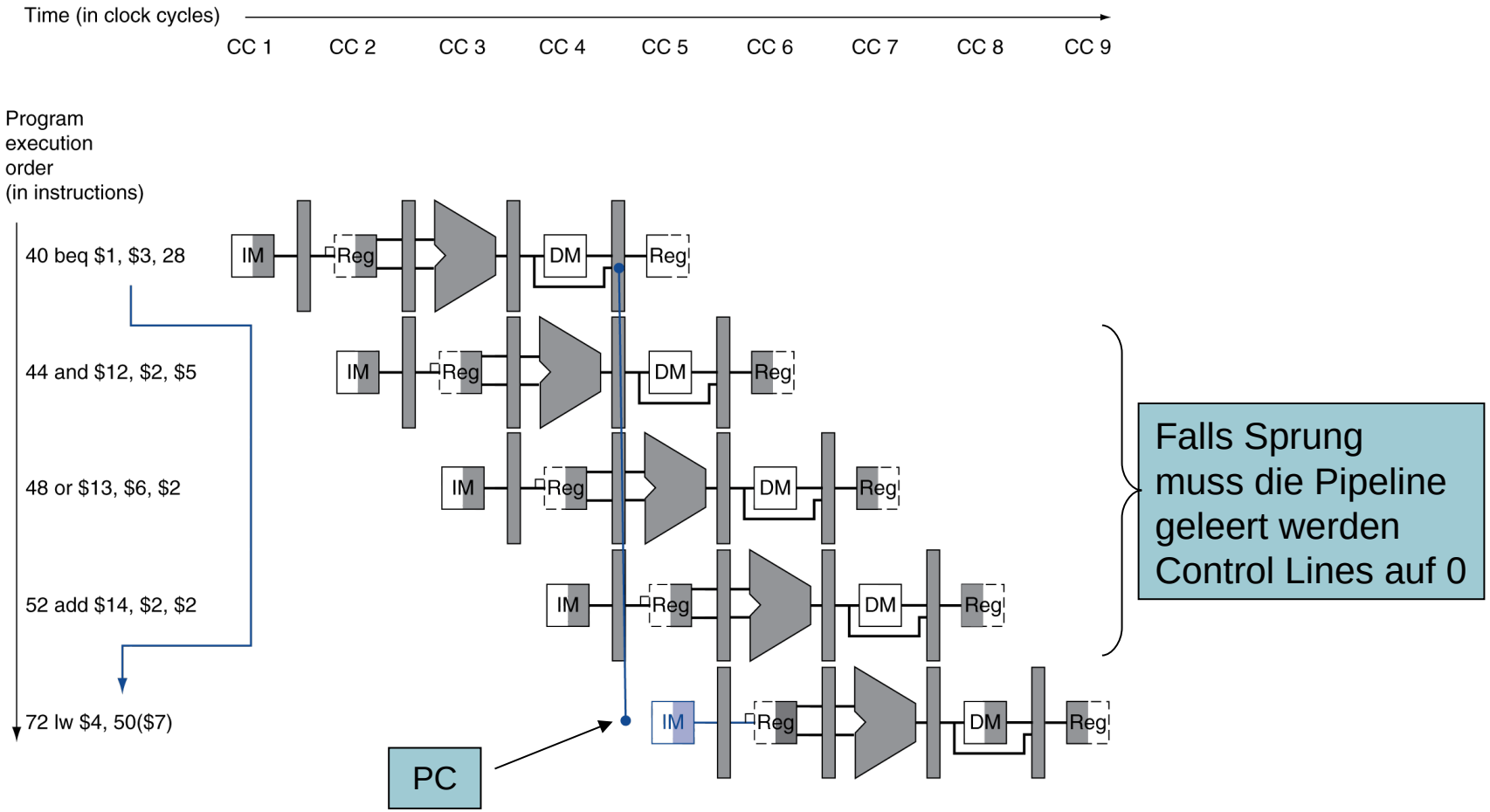
Sprunganweisungen ändern die Reihenfolge der Ausführung

Mögliche Lösung: Die Pipeline muss warten, bis Entscheidung klar ist → schlecht für Performance



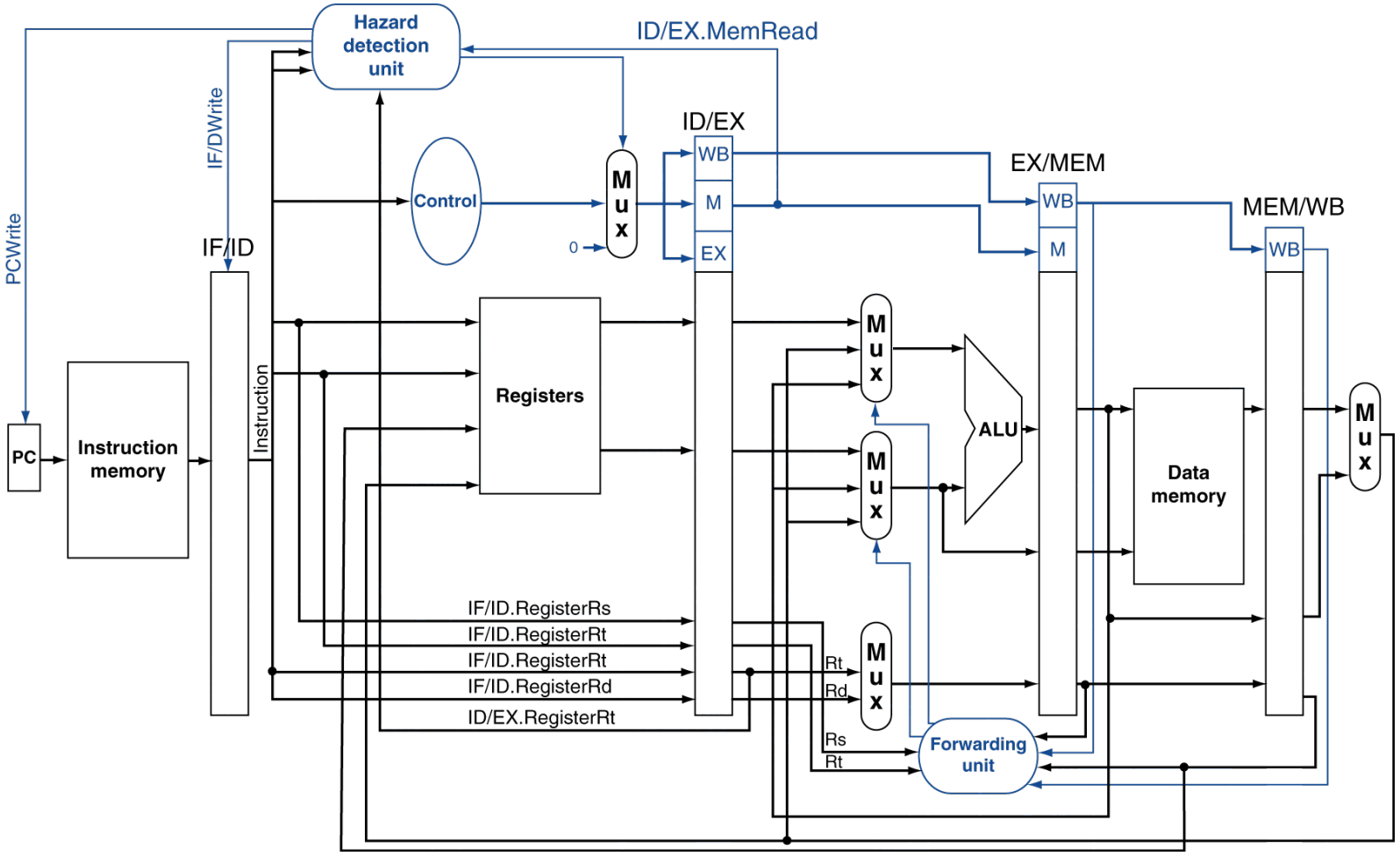


Control Hazards



Hazard-Erkennung

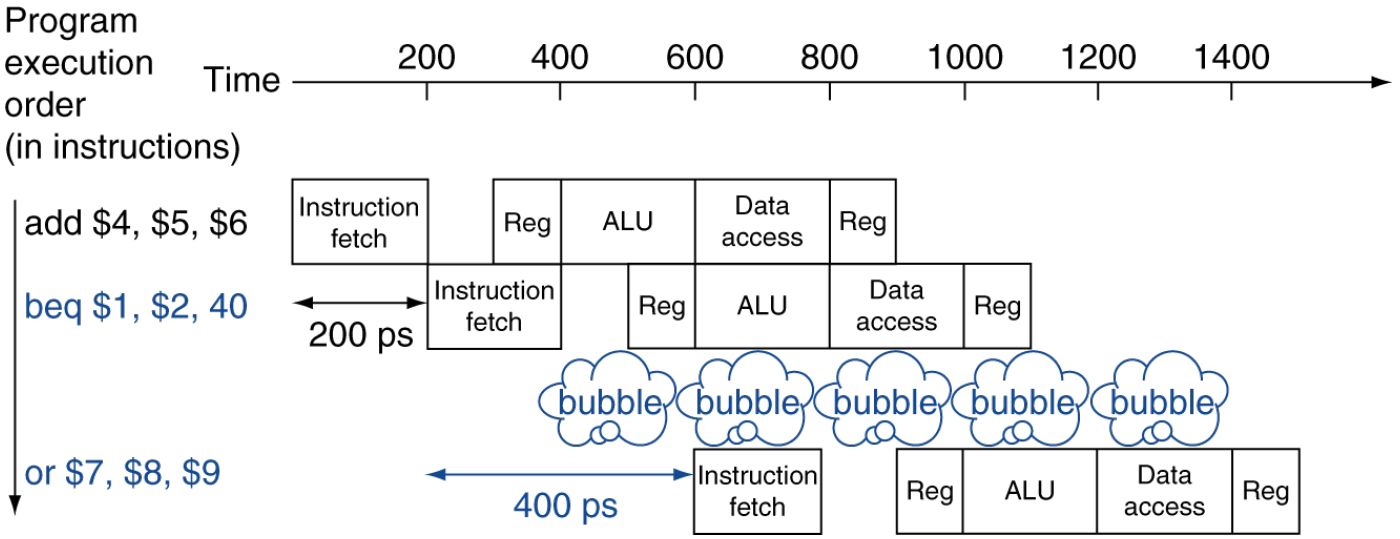
Wir brauchen dafür eine eigene Hazard Detection Unit...



Wie wird die Pipeline angehalten (Stall)?

- Steuerwerk-Werte für Daten schreiben / Mem write auf 0
  Damit verhält sich die Pipeline bei Execute, Mem und Write Back wie bei einem N0P-Befehl

- PC und IF/ID-Register werden *nicht* aktualisiert
  - Beim nächsten Taktzyklus wird somit wieder dieselbe Anweisung verarbeitet



Mögliche Lösung: Sprungvorhersage

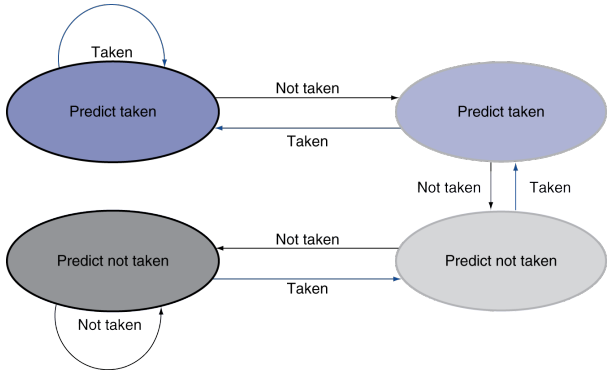
Zwei Möglichkeiten:

Statische Sprungvorhersage
- Einfachste Lösung wie bei Schleifen: wir gehen davon aus, dass ein Rücksprung öfter genommen wird

Dynamische Sprungvorhersage
- Die Hardware merkt sich die letzten Sprünge, z.B. durch die Zustände eines Endlichen Automaten (letzte x mal Branch taken / not taken)

Falls wir falsch geraten haben, muss die Pipeline geleert werden

→ Pipelining braucht einiges an Hardware



Superskalare Architekturen / Multiple Issue

Instruction-Level Parallelism (ILP)

Es gibt prinzipiell mehrere Möglichkeiten, die Ausführung von Instruktionen in einem Single-Core-System zu parallelisieren

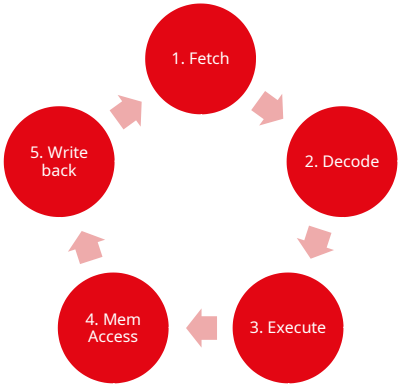
1) Pipelining
   Parallele Pipeline-Stufen für die Verarbeitung der Befehle
   Anzahl an Pipeline-Stages kann erhöht werden, allerdings Probleme mit Hazards

2) Multiple Issue / Superskalare Architekturen
   Hardware-Einheiten in den Pipeline-Stages duplizieren → mehr ALUs, Steuerwerke, etc...
   Damit können mehrere Befehle pro Taktzyklus parallel abgearbeitet werden → Hazards müssen auch hier vermieden werden!

3) Vector Instructions / SIMD (Single Instruction Multiple Data)
   Verbreiterung des Datenbusses, ISA mit Befehlen zur Verarbeitung vieler Daten parallel (z.B. MMX, SSE, AVX, ...)

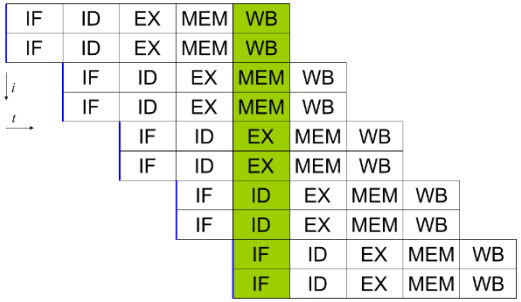
Superskalare Architektur

Durch Pipelining werden Einheiten einer CPU optimal ausgelastet



Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Duplikation von Steuer-/Rechenwerk: Mehrere Befehle können gleichzeitig verarbeitet werden



Wikipedia

Multiple Issue / Superskalare Architekturen

Static Scheduling / Multiple Issue

- Der Compiler gruppiert parallel ausführbare Instruktionen (ohne Abhängigkeiten / Data/Control-Hazards) zusammen in einen „Issue Slot“
- Issue Slot wird dann von der CPU parallel ausgeführt
- Der Compiler ist zuständig für Hazard-Vermeidung (z.B. durch Einfügen von `NOP`s oder Ändern der Reihenfolge)
- Zusätzlich kann der Compiler generell durch Loop-Unrolling, Inlining etc., helfen (auch ohne Multiple Issue)
- Kann also bestehenden Assembler-Code nicht beschleunigen

Dynamic Scheduling / Multiple Issue → **Superskalare Prozessoren**

- Die CPU entscheidet jeweils, ob die Instruktionen zusammen ausgeführt werden können, sie darf auch die Reihenfolge ändern (wenn ohne Folgen)!
- Die CPU ist für die Hazard-Detektion zuständig → aufwendig
- Der Compiler darf gerne mithelfen, muss aber nicht
- Auch „alter“ Code wird damit evtl. beschleunigt abgearbeitet

Dynamic Scheduling

Bei superskalaren Architekturen (mit Dynamic Scheduling) darf die CPU auch die Ausführungsreihenfolge verändern, solange das keine Auswirkung auf das Ergebnis hat

→ Out of Order Execution

Beim Rückspeicher ins Registerfile muss aber die ursprüngliche Reihenfolge wieder eingehalten werden

Beispiel

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

sub kann vorgezogen werden, da addu auf lw warten muss

Dynamic Scheduling und Register

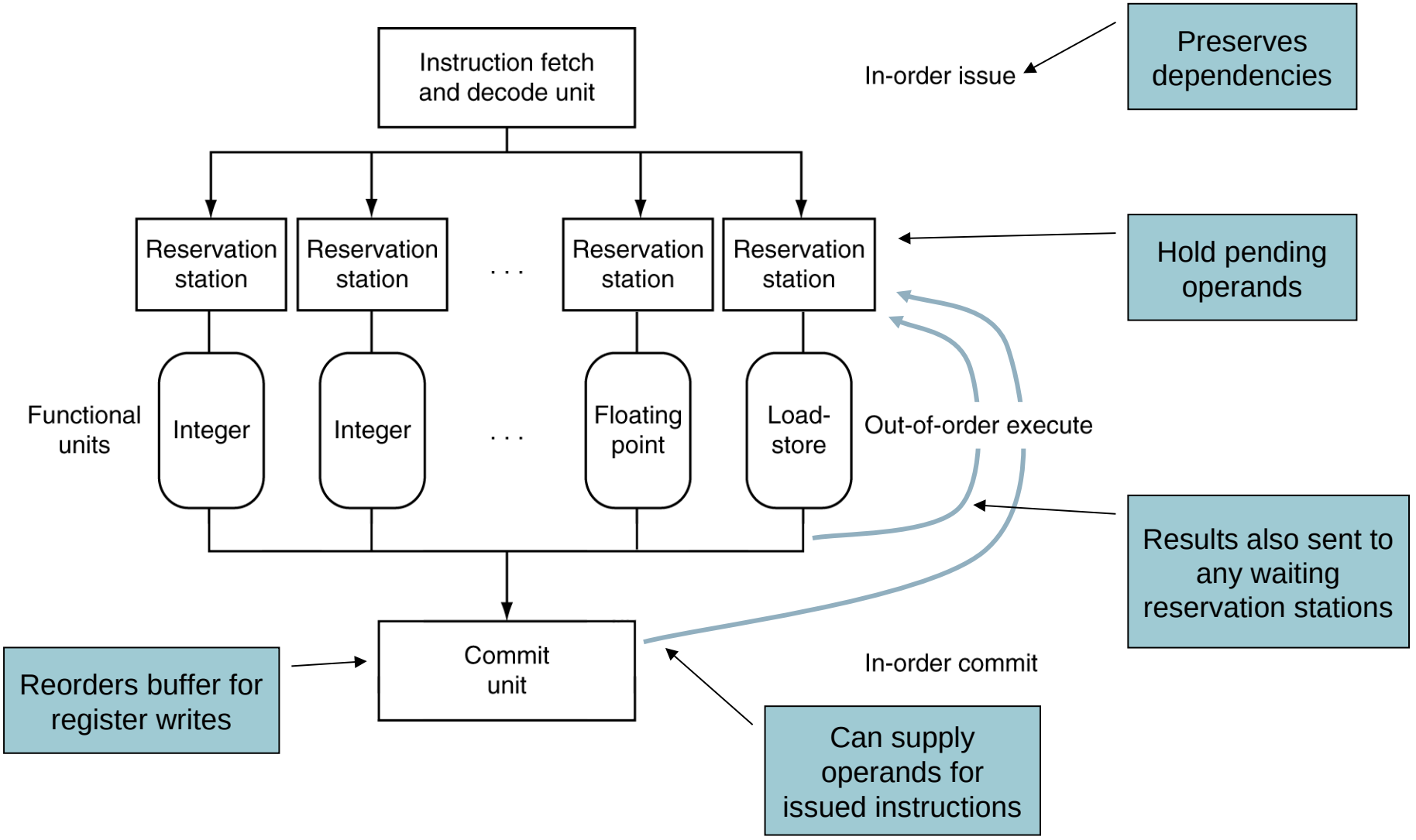
Durch Out of Order Execution können Ergebnisse vorher feststehen, sie dürfen aber, um die Konsistenz zu garantieren, nur in originaler Reihenfolge gespeichert werden

→ wir brauchen dafür wieder eigene Register

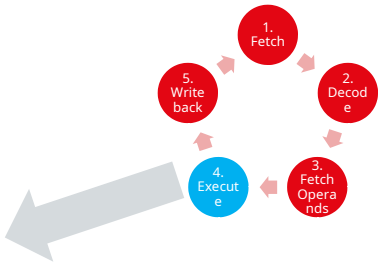
- Reservation Station: speichert die Operanden (Kopie der Register-Werte) und den Befehl für die Ausführung
- Reorder Buffer: hält Kopie der geänderten, aber noch nicht zurückgeschriebenen Register

Wichtig: In Order commit, in korrekter Reihenfolge Ergebnisse zurückschreiben in Registerfile für den Unterbrechungsfall (z.B. Exception)

Superskalare Architektur



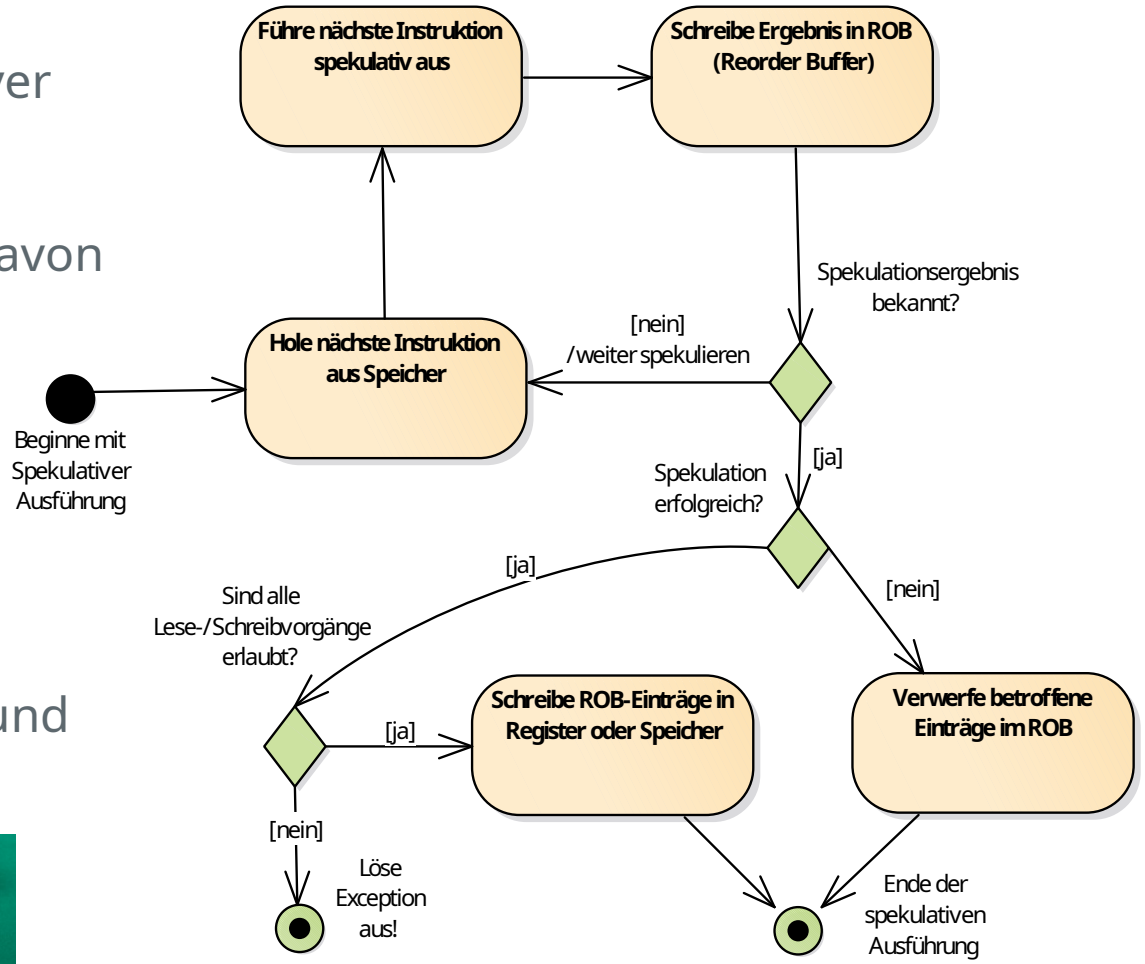
Falsch geraten?



Kein Problem bei spekulativer Ausführung!

Die CPU geht einfach mal davon aus, dass die Anweisungen ausgeführt werden können

Ergebnisse werden in Zwischenspeicher gehalten (Reorder Buffer), und können verworfen werden

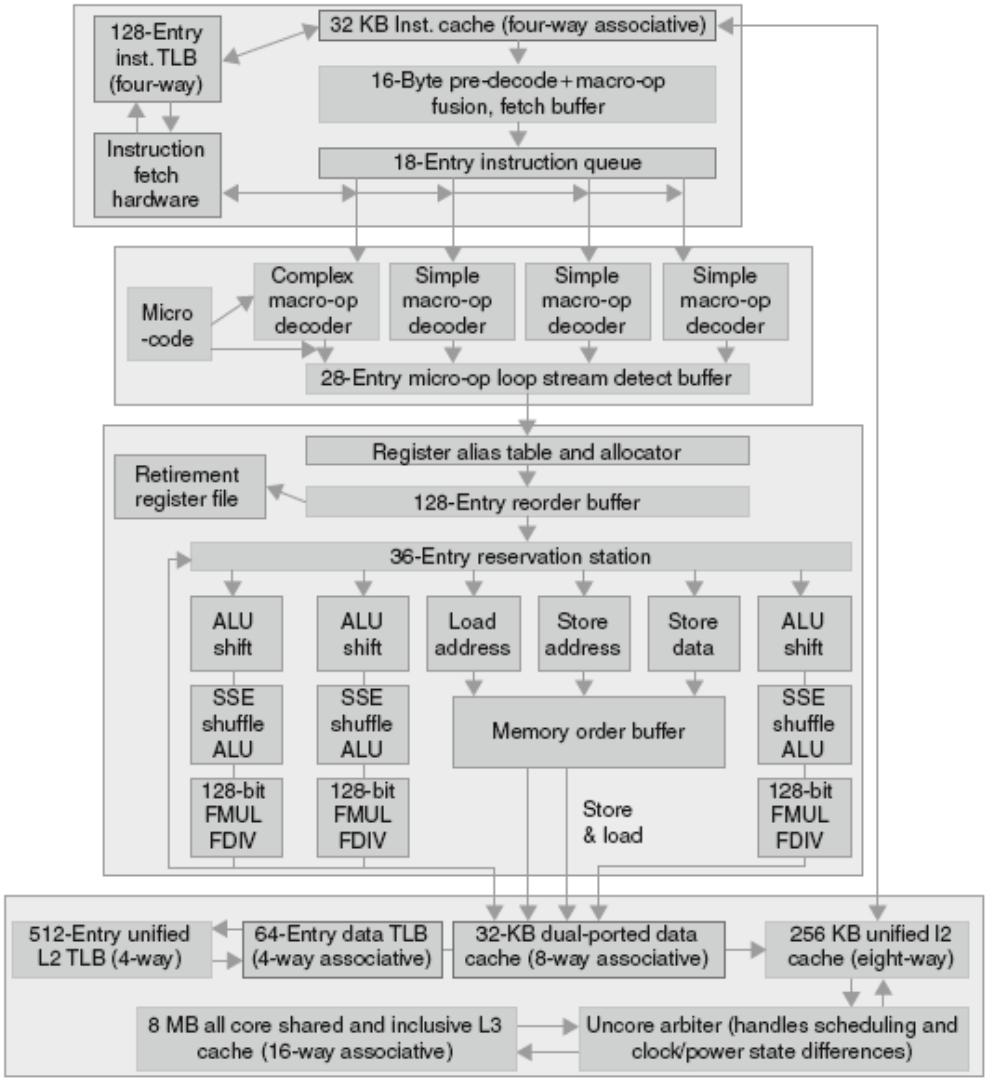


Probleme mit Superskalaren Architekturen

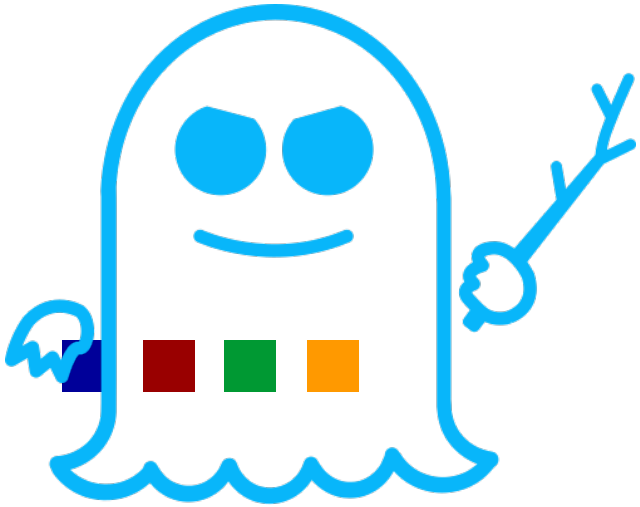
Hohe Komplexität des dynamisches Schedulings und der Sprungvorhersage
→ höherer Stromverbrauch
Daher Übergang zu Multicore-Architekturen

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	3000 MHz	14	4	Yes	2	75 W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87 W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130 W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130 W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140 W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165 W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185 W

Beispiel: Core i7 Pipeline



Backup / Spectre, Meltdown



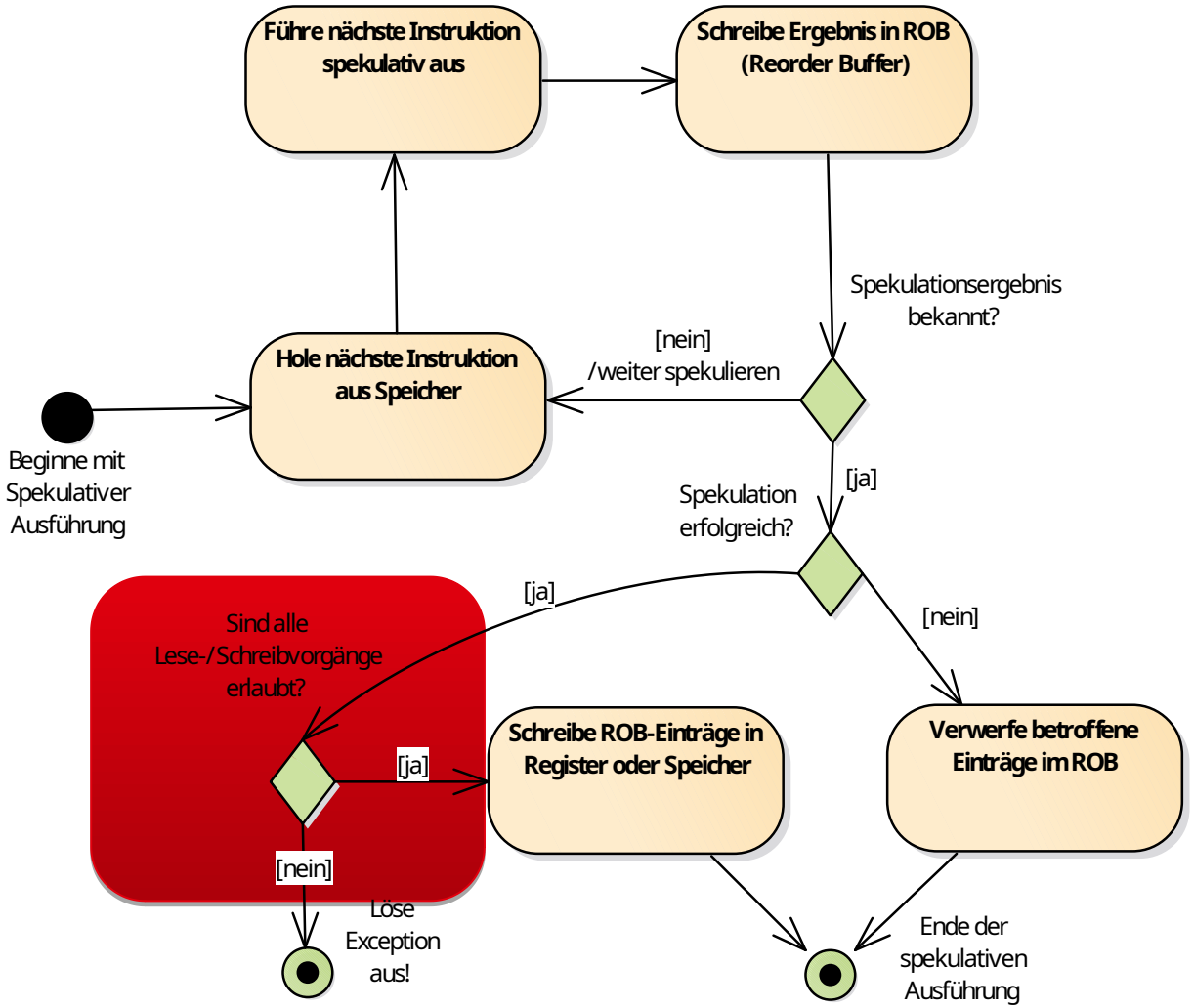
SPECTRE

MELTDOWN

Was hat dies mit Spectre zu tun?

Rechte-Überprüfung findet erst spät statt. Spekulative Instruktionen können unerlaubt auf Daten zugreifen

Wie kann man das ausnutzen?



Wie kann man das ausnutzen?

Wir können auf einfache Weise keine Daten herausschmuggeln

```
char * illegal_address = ...;
char my_illegal_data;

 // Precondition: uncached_address=false & not in cache
if(uncached_address == true)
{
  my_illegal_data = *illegal_address; // Speculative Execution
}
// use my_illegal_data
```

→ nach my_illegal_data wird nicht geschrieben!

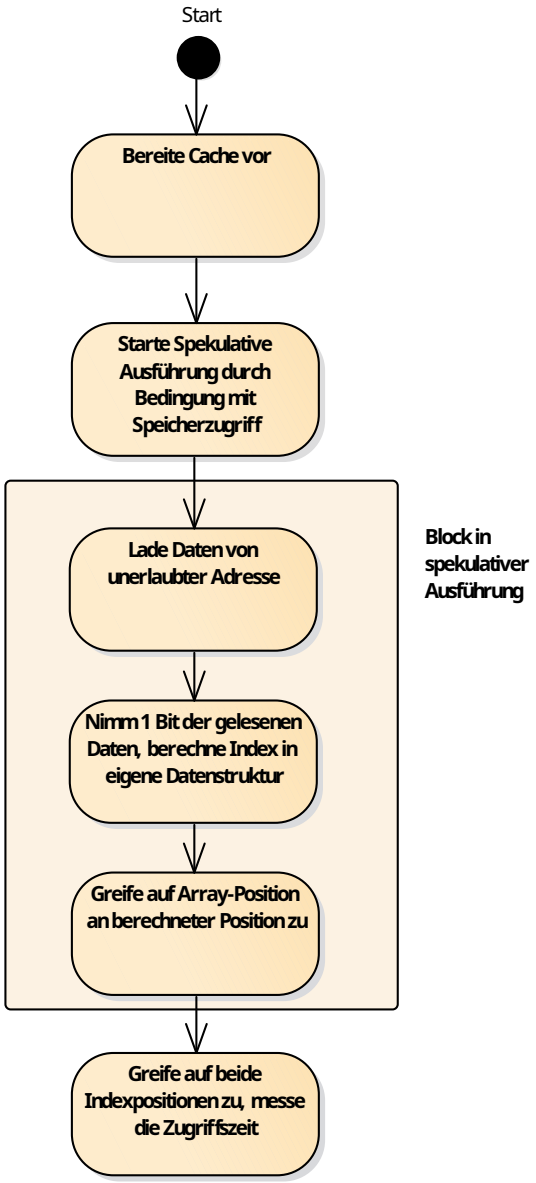
Lösung: Wir (miss-/)gebrauchen den Memory-Cache!

Spectre V1

```
char * illegal_address = ...;
char my_illegal_data;
char arr[0x400] = {0};

// Precondition: uncached_address=false & not
// in cache
if(uncached_address == true)
{
  // Speculative Execution
  my_illegal_data = *illegal_address;
  unsigned long index = ((my_illegal_data & 1)
*
    0x100) + 0x200;
  char testread = arr[index];
}

// measure time for accessing arr[0x200]
// and arr[0x300]
```



Meltdown, Spectre V2, …

Basieren schlussendlich alle auf der selben Idee:
Benutze Spekulative Ausführung, schmuggle die Daten durch Cache-Zugriff heraus!

Referenzen

- Patterson, D; Hennessy , J; 2017; „Rechnerorganisation und Rechnerentwurf - Die Hardware/Software Schnittstelle”; Spektrum Akademischer Verlag; ISBN 9783110446050
- Wikipedia - The Free Enzyklopädie; wikipedia.com
- Wikimedia Commons; commons.wikimedia.org
- Google LLC; google.com
- ARM®; 2016; „ARM® Cortex®-A72 MPCore Processor - Technical Reference Manual“
- ARM®; 2015; „ARM® Cortex®-A Series - Programmer’s Guide for ARMv8-A“