

Technische Grundlagen der Informatik – Kapitel 7



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Prof. Dr.-Ing. Andreas Koch
Fachgebiet Eingebettete Systeme und ihre Anwendungen (ESA)
Fachbereich Informatik

WS 13/14



Kapitel 7: Themen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Einführung in die Mikroarchitektur**
- **Analyse der Rechenleistung**
- **Ein-Takt-Prozessor**
- **Mehrtakt-Prozessor**
- **Pipeline-Prozessor**
- **Ausnahmebehandlung**
- **Weiterführende Themen**

Einleitung

- Mikroarchitektur
 - Hardware-Implementierung einer Architektur
- Prozessor:
 - Datenpfad: funktionale Blöcke
 - Steuerwerk: Steuersignale

Anwendungs-Software	Programme
Betriebs-systeme	Gerätetreiber
Architektur	Instruktionen Register
Mikro-architektur	Datenpfade Steuerwerke
Logik	Addierer Speicher
Digital-schaltungen	AND Gatter NOT Gatter
Analog-schaltungen	Verstärker Filter
Bauelemente	Transistoren Dioden
Physik	Elektronen



- **Mehrere** Implementierungen für eine Architektur
 - **Ein-Takt**
 - Jede Instruktion wird in einem Takt ausgeführt
 - **Mehrtakt**
 - Jede Instruktion wird in Teilschritte zerlegt
 - **Pipelined**
 - Jede Instruktion wird in Teilschritte zerlegt
 - Mehrere Instruktionen werden gleichzeitig ausgeführt

Rechenleistung eines Prozessors



- **Ausführungszeit** eines Programms

$$\text{Ausführungszeit} = (\# \text{ Instruktionen}) \cdot (\text{Takte/Instruktion}) \cdot (\text{Sekunden/Takt})$$

- Definitionen:
 - Takte/Instruktion = CPI (*cycles per instruction*)
 - Sekunden/Takt = Taktperiode
 - $1/\text{CPI} = \text{Instruktionen/Takt} = \text{IPC}$ (*instructions per cycle*)
- Herausforderung: Einhalten **zusätzlicher** Anforderungen
 - Kosten
 - Energiebedarf
 - Rechenleistung

Unser erster MIPS Prozessor



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Zunächst **Untermenge** des MIPS Befehlssatzes:
 - R-Typ Befehle: `and`, `or`, `add`, `sub`, `slt`
 - Speicherbefehle: `lw`, `sw`
 - Bedingte Verzweigungen: `beq`
- **Später** hinzunehmen: `addi` und `j`

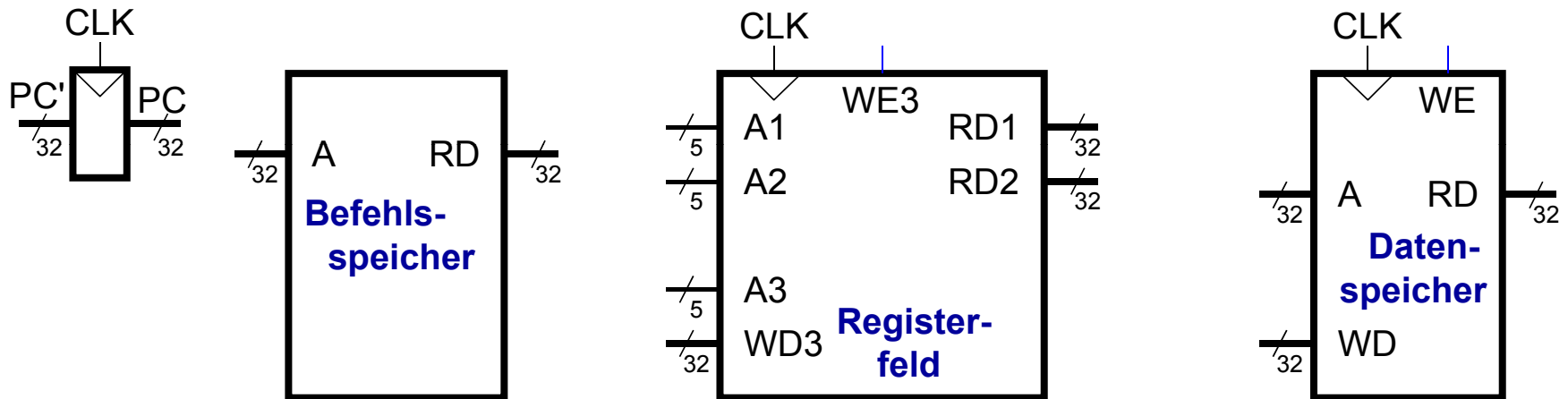
Architekturzustand



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Auf Ebene der **Architektur** sichtbare Daten
 - Für den Programmierer **zugänglich**
- Bestimmen **vollständigen** Zustand der Architektur
 - PC
 - 32 Register
 - Speicher

Elemente des MIPS Architekturzustands



Ein-Takt MIPS Prozessor

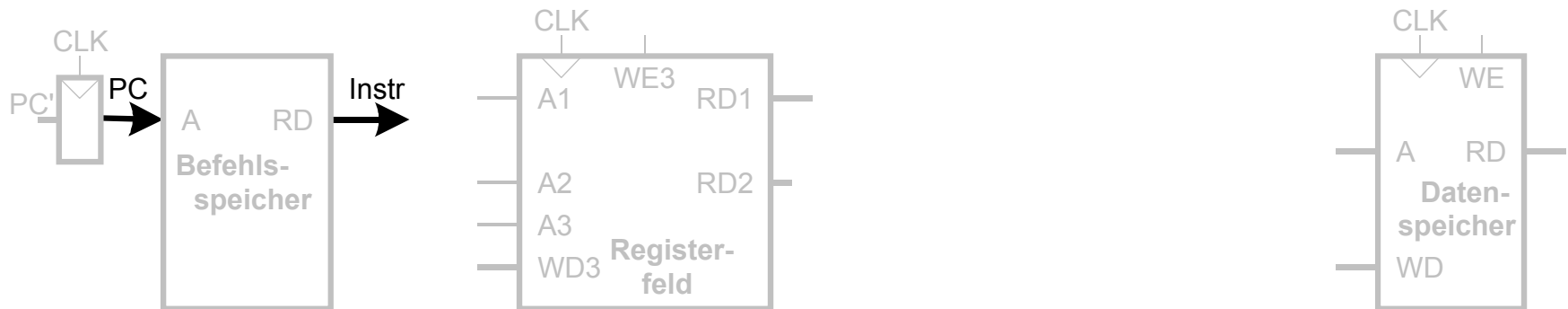


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Datenpfad
- Steuerwerk

Ein-Takt Datenpfad: Holen eines $1w$ Befehls

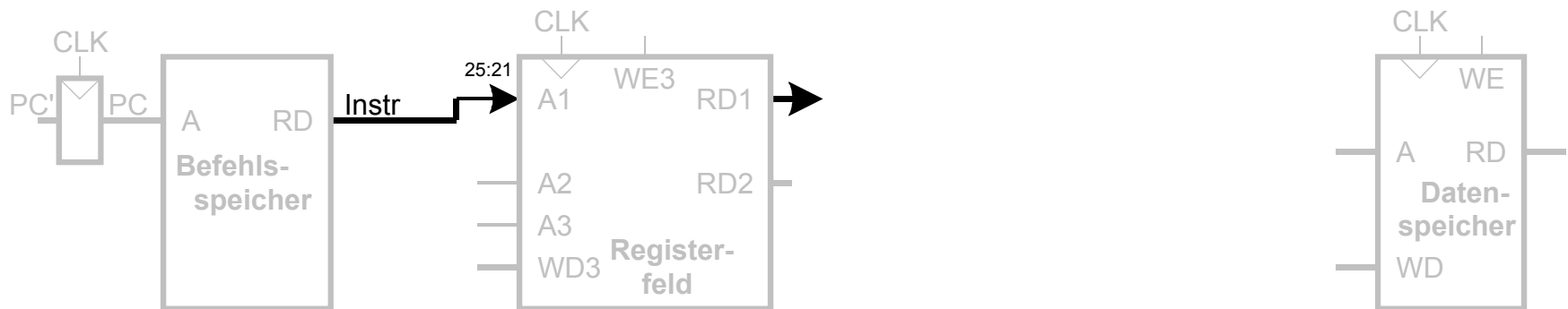
- Ein *load word* Befehl ($1w$) soll ausgeführt werden
- **Schritt 1:** Hole Instruktion



Ein-Takt Datenpfad: Lesen des Registers für 1w

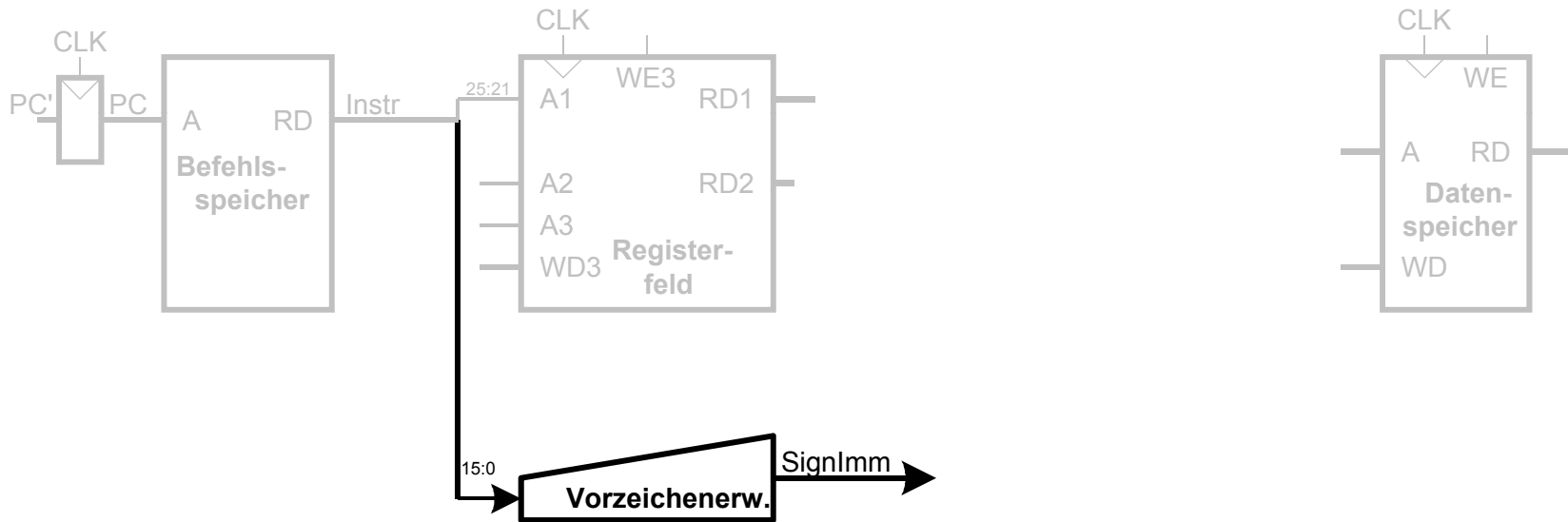


- **Schritt 2:** Lese Quelloperand aus Registerfeld



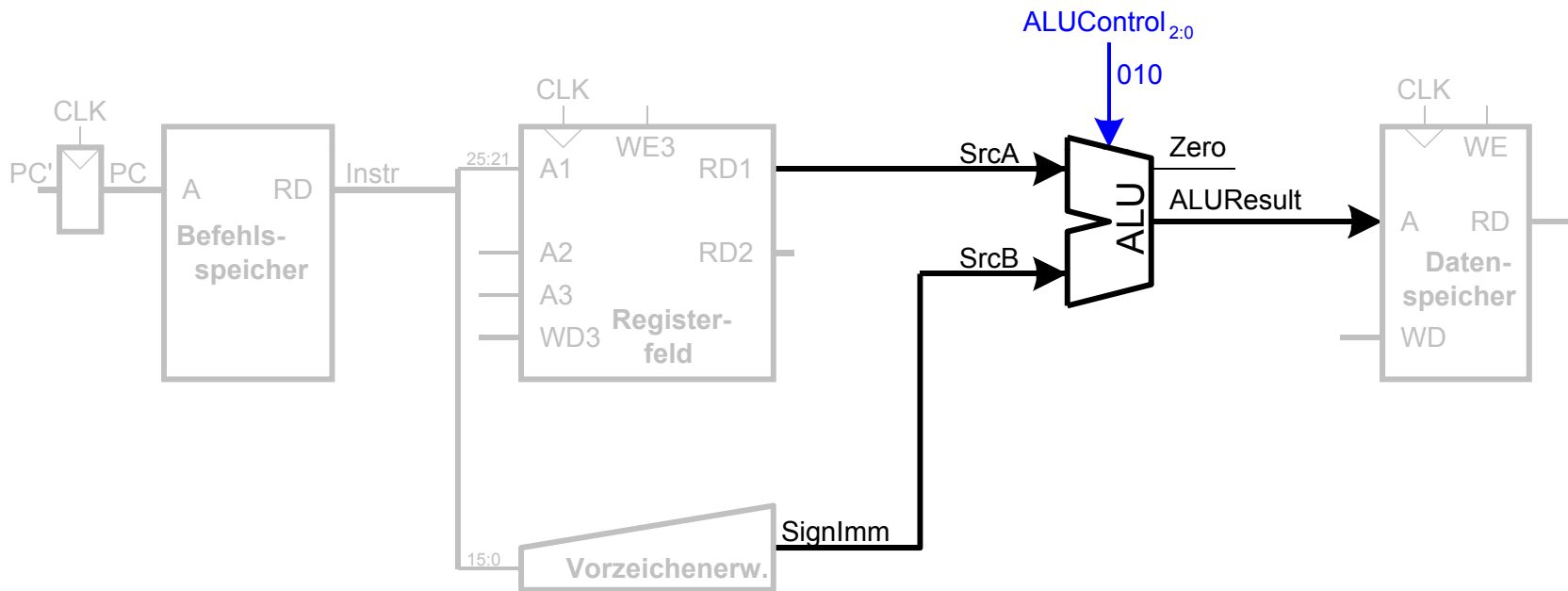
Ein-Takt Datenpfad: Behandle 1w Direktwert

- **Schritt 3:** Vorzeichenerweiteren den 16b Direktwert auf 32b Signal `SignImm`



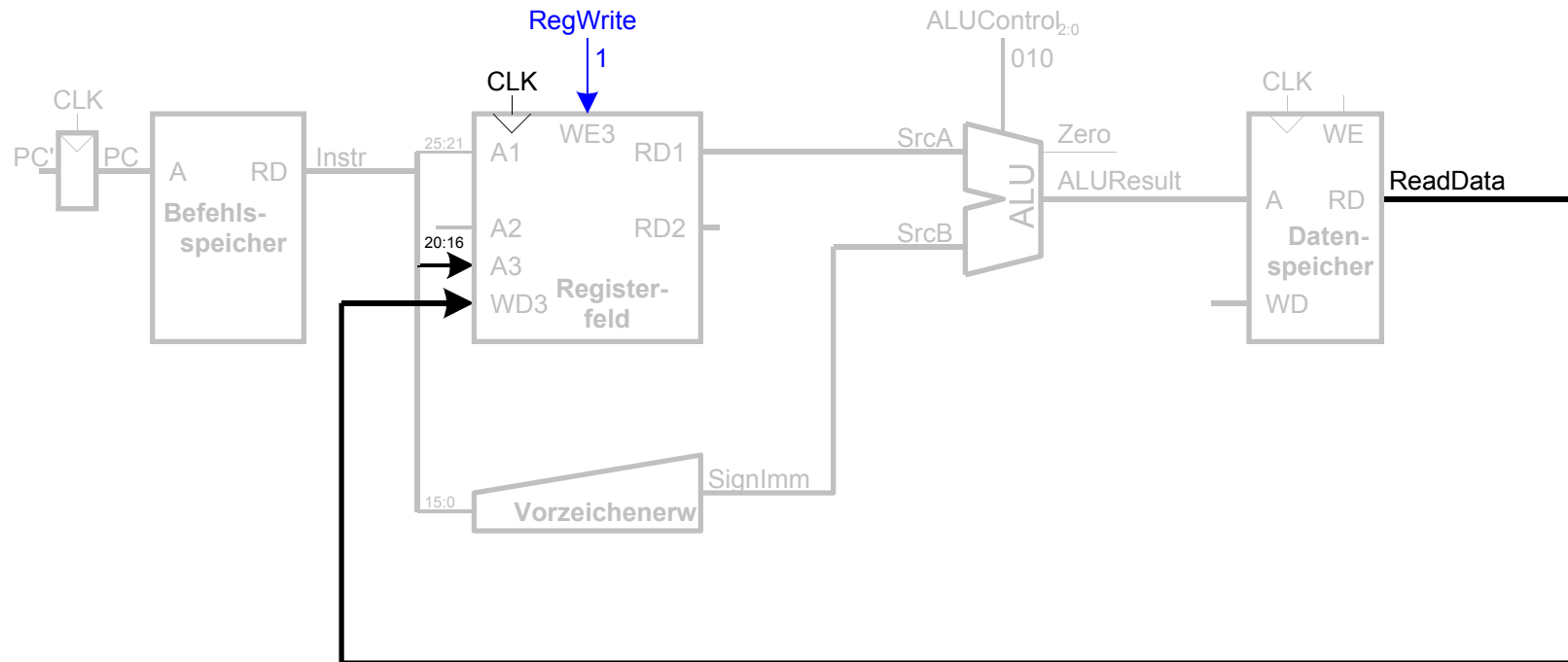
Ein-Takt Datenpfad: Berechne 1w Zieladresse

- **Schritt 4:** Berechne die effektive Speicheradresse



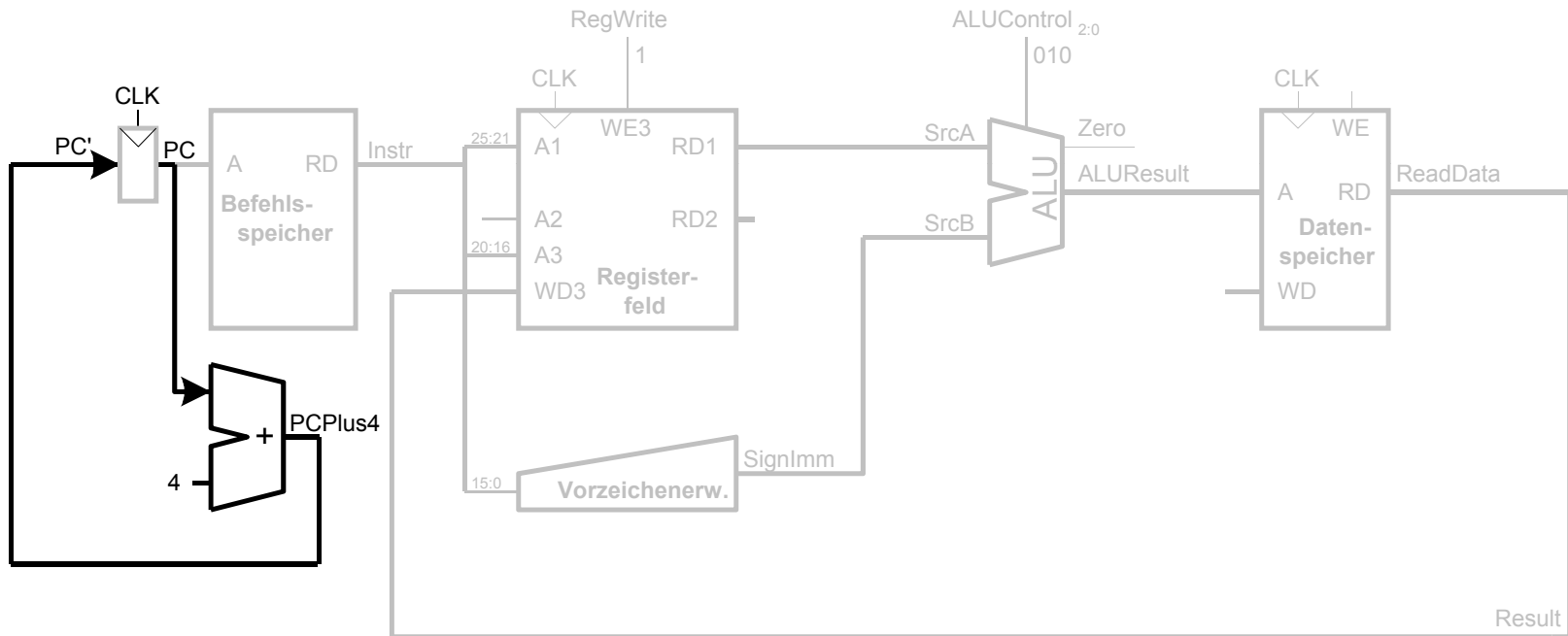
Ein-Takt Datenpfad: Lese Speicher mit 1w

- **Schritt 5:** Lese Daten aus Speicher und schreibe sie ins passende Register



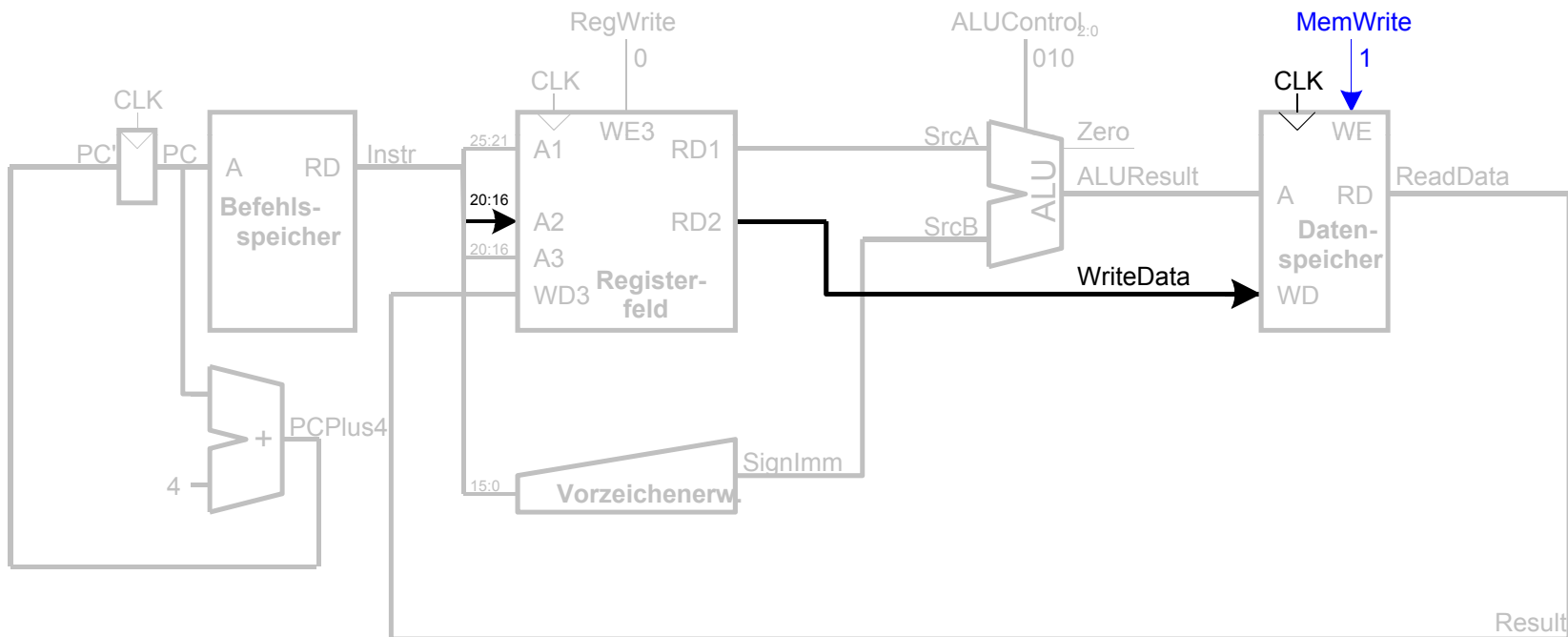
Ein-Takt Datenpfad : Erhöhe PC nach 1w

- **Schritt 6:** Bestimme Adresse des nächsten Befehls



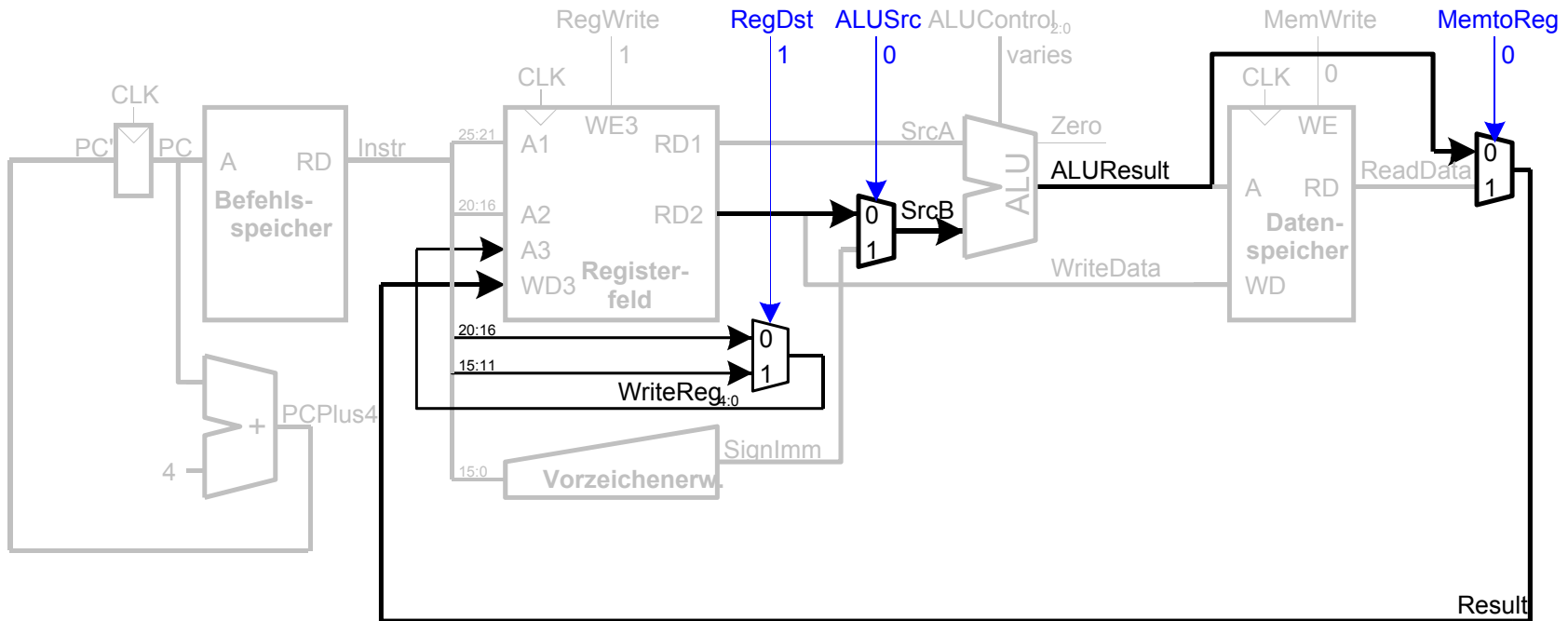
Ein-Takt Datenpfad: sw

- Schreibe Daten aus r_t in den Speicher



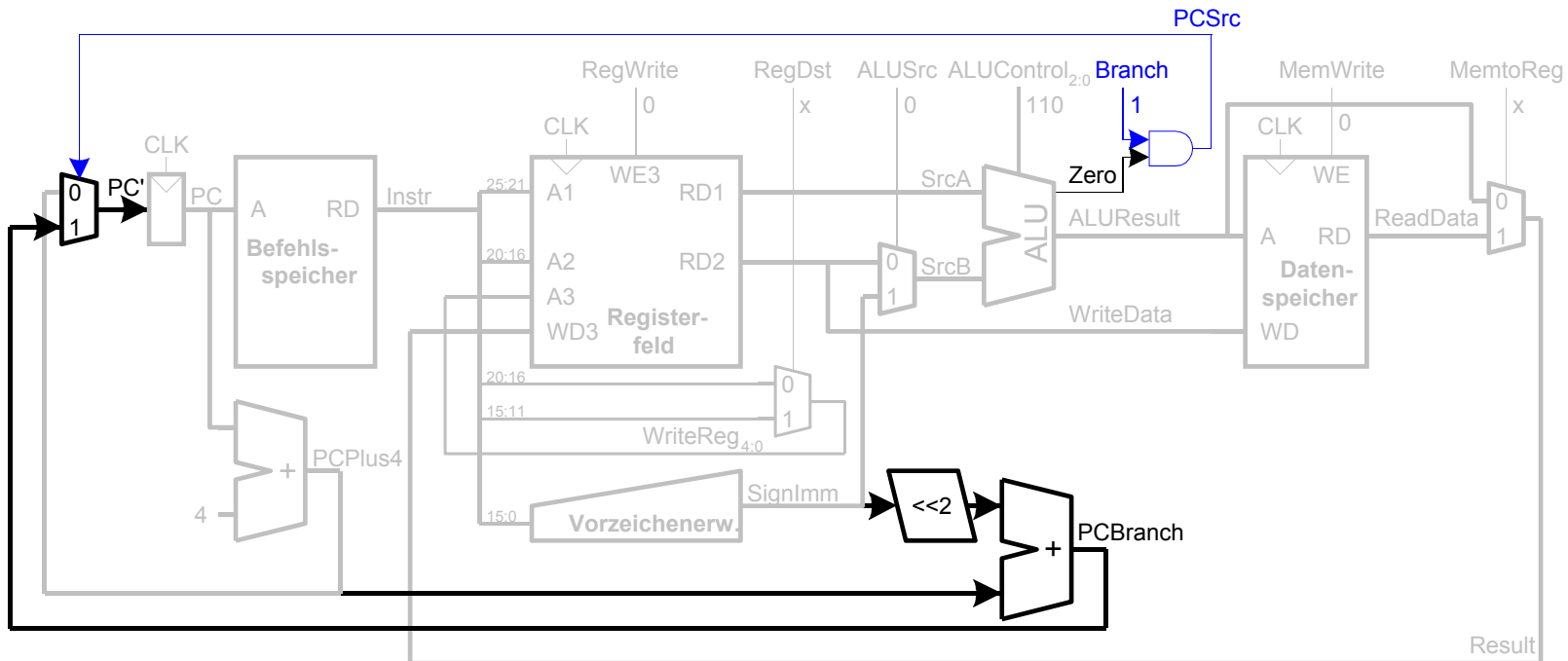
Ein-Takt Datenpfad: Instruktionen vom R-Typ

- Lese aus `rs` und `rt`
- Schreibe *ALUResult* ins Registerfeld
- Schreibe nach `rd` (statt nach `rt` wie bei `sw`)

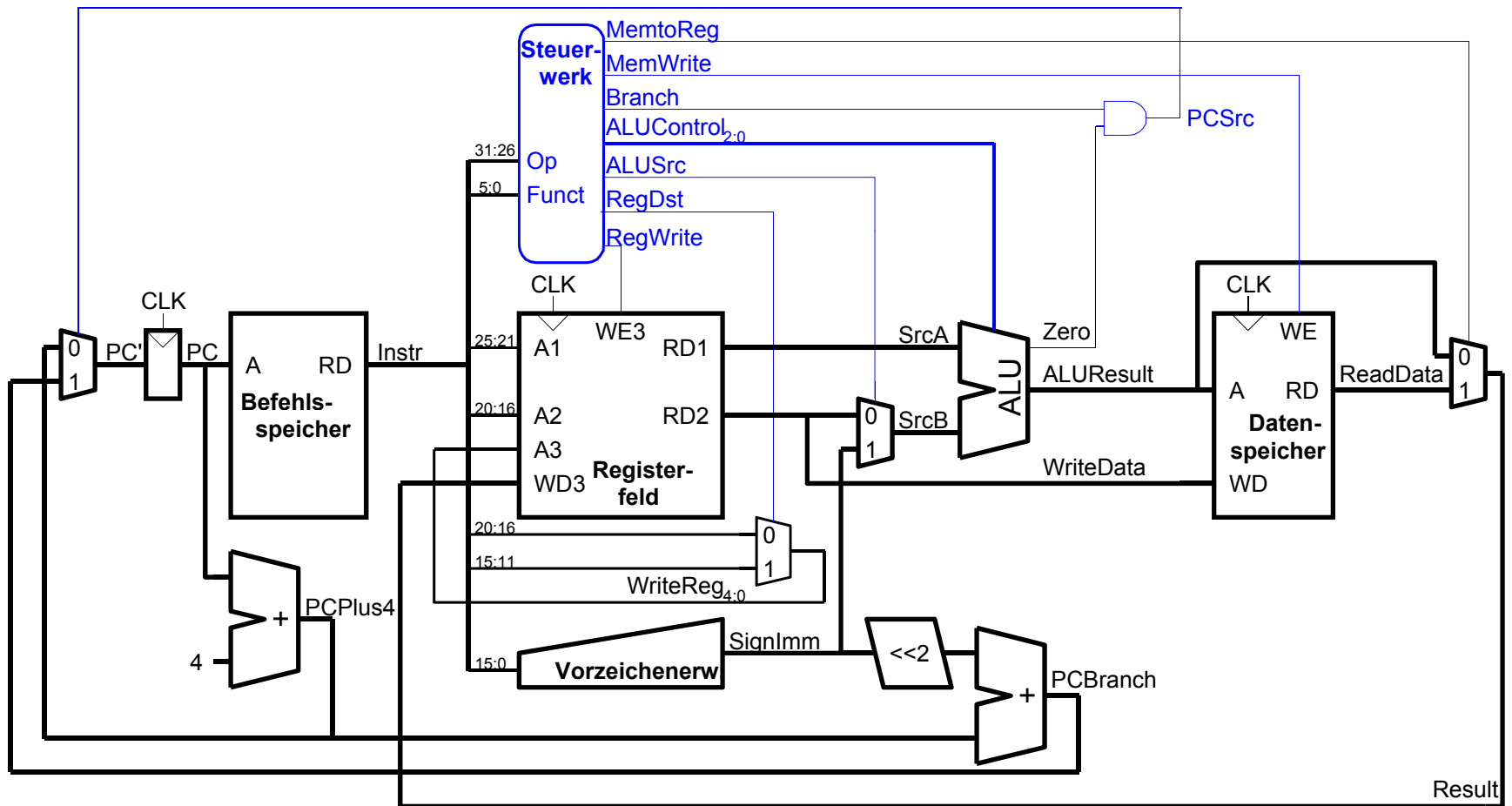


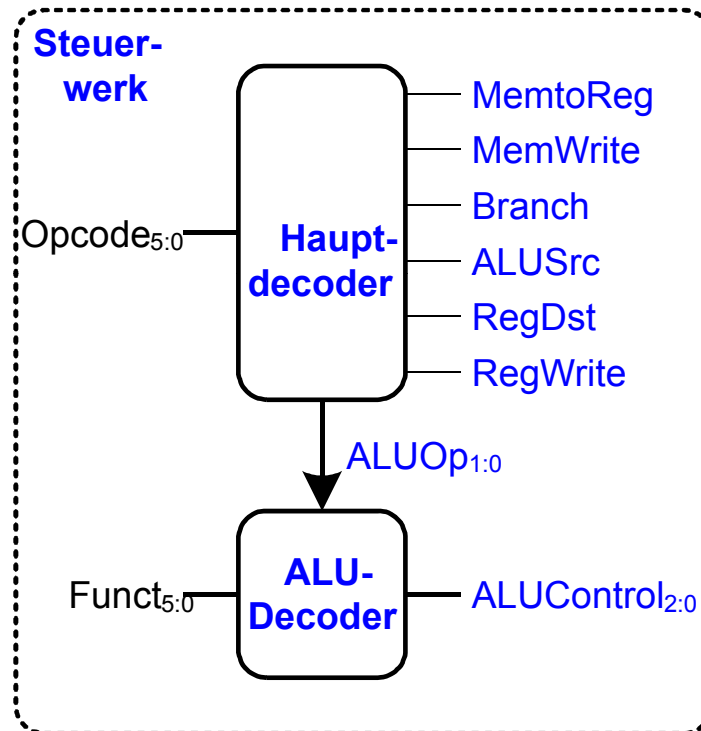
Ein-Takt Datenpfad: beq

- Prüfe ob Werte in rs und rt gleich sind
- Bestimme Adresse von Sprungziel (*branch target adress, BTA*):
 $BTA = (\text{vorzeichenerweiterter Direktwert} \ll 2) + (PC+4)$

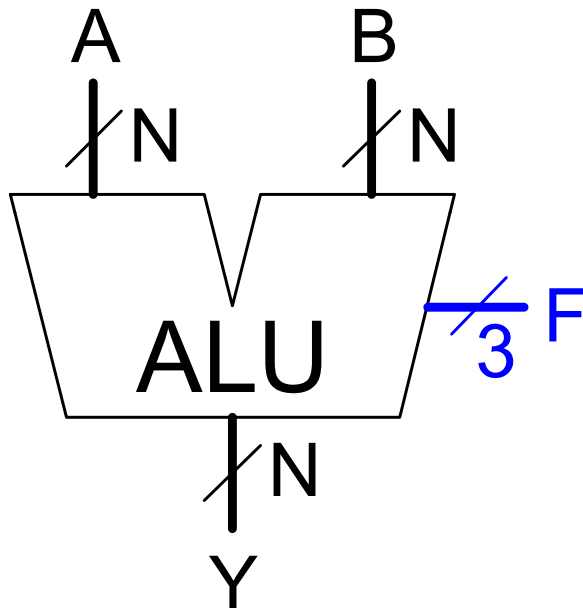


Vollständiger Ein-Takt-Prozessor



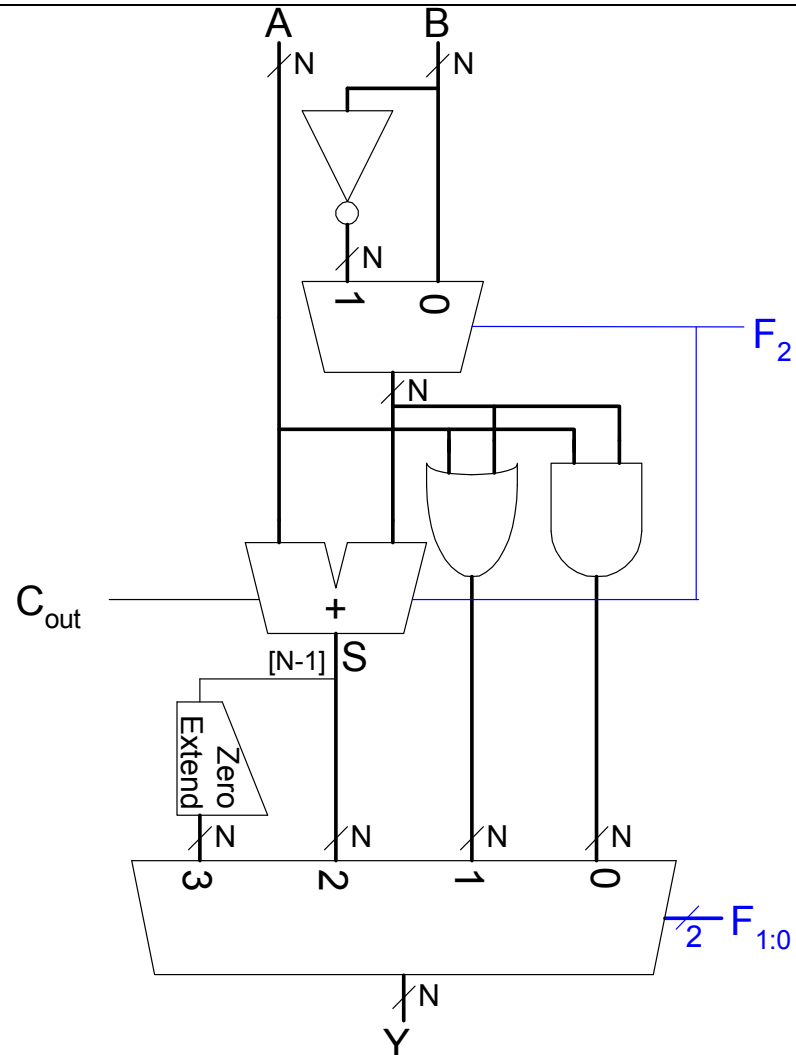


Zur Erinnerung: ALU



$F_{2:0}$	Funktion
000	A & B
001	A B
010	A + B
011	unbenutzt
100	A & ~B
101	A ~B
110	A - B
111	SLT

Zur Erinnerung: ALU



Steuerwerk: ALU-Decoder

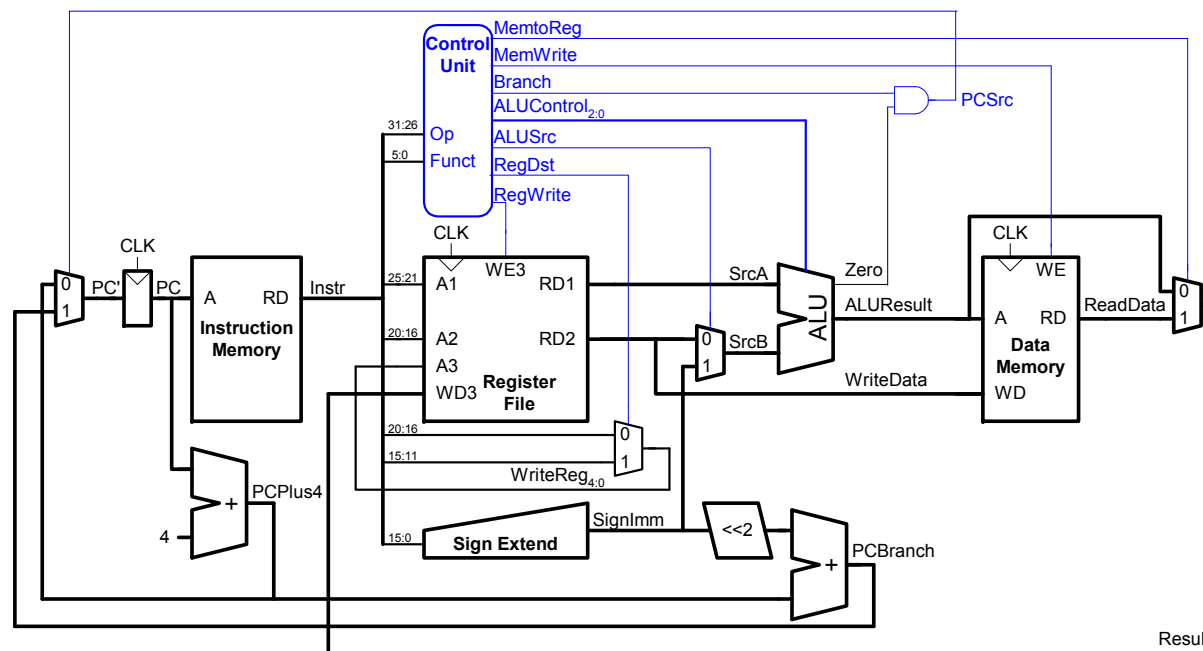


ALUOp_{1:0}	Bedeutung
00	Addiere
01	Subtrahiere
10	Werte Funct-Feld aus
11	unbenutzt

ALUOp_{1:0}	Funct	ALUControl_{2:0}
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

Steuerwerk: Hauptdecoder

Instruktion	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-Typ	000000							
lw	100011							
sw	101011							
beq	000100							

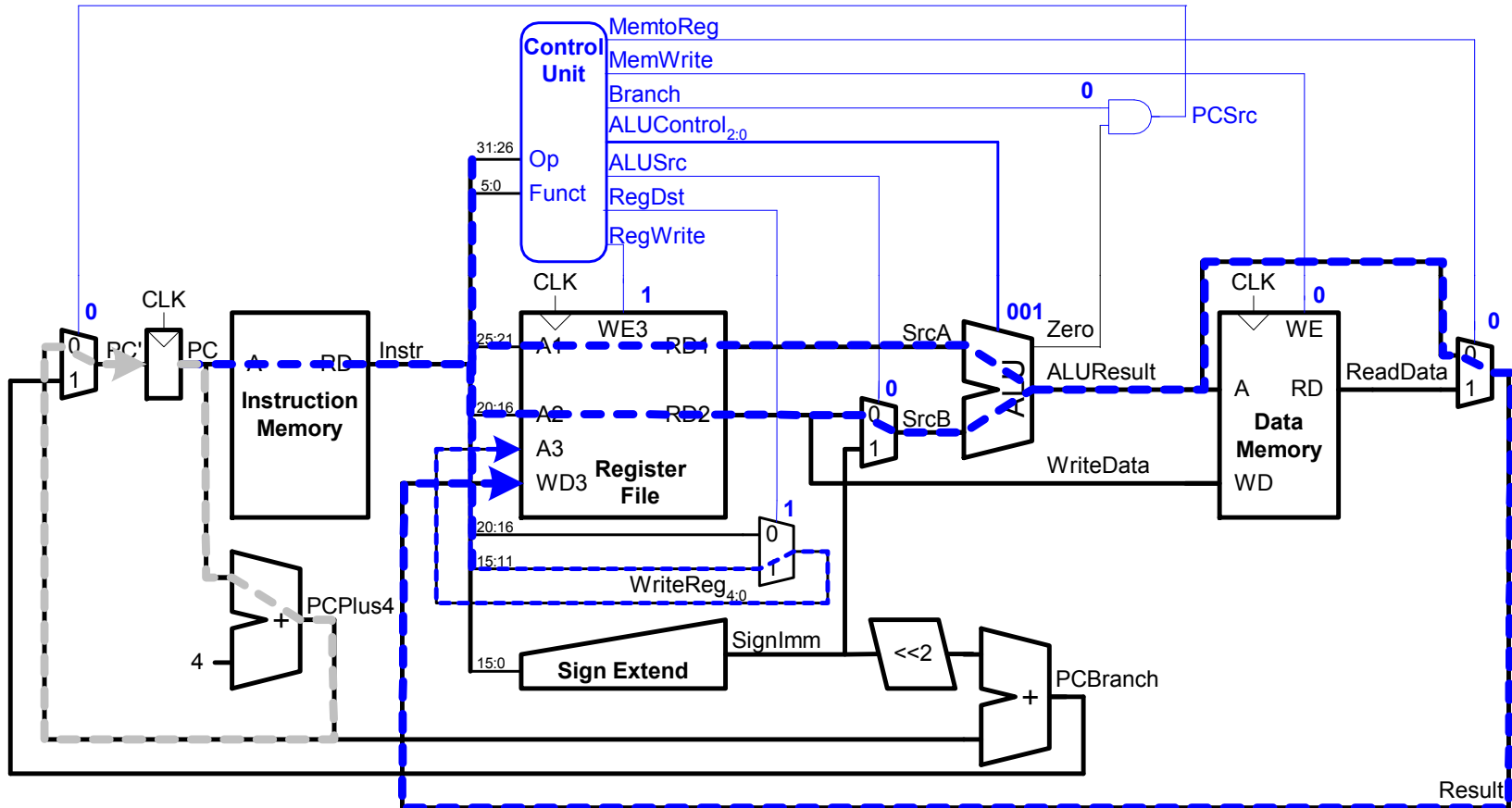


Steuerwerk: Hauptdecoder



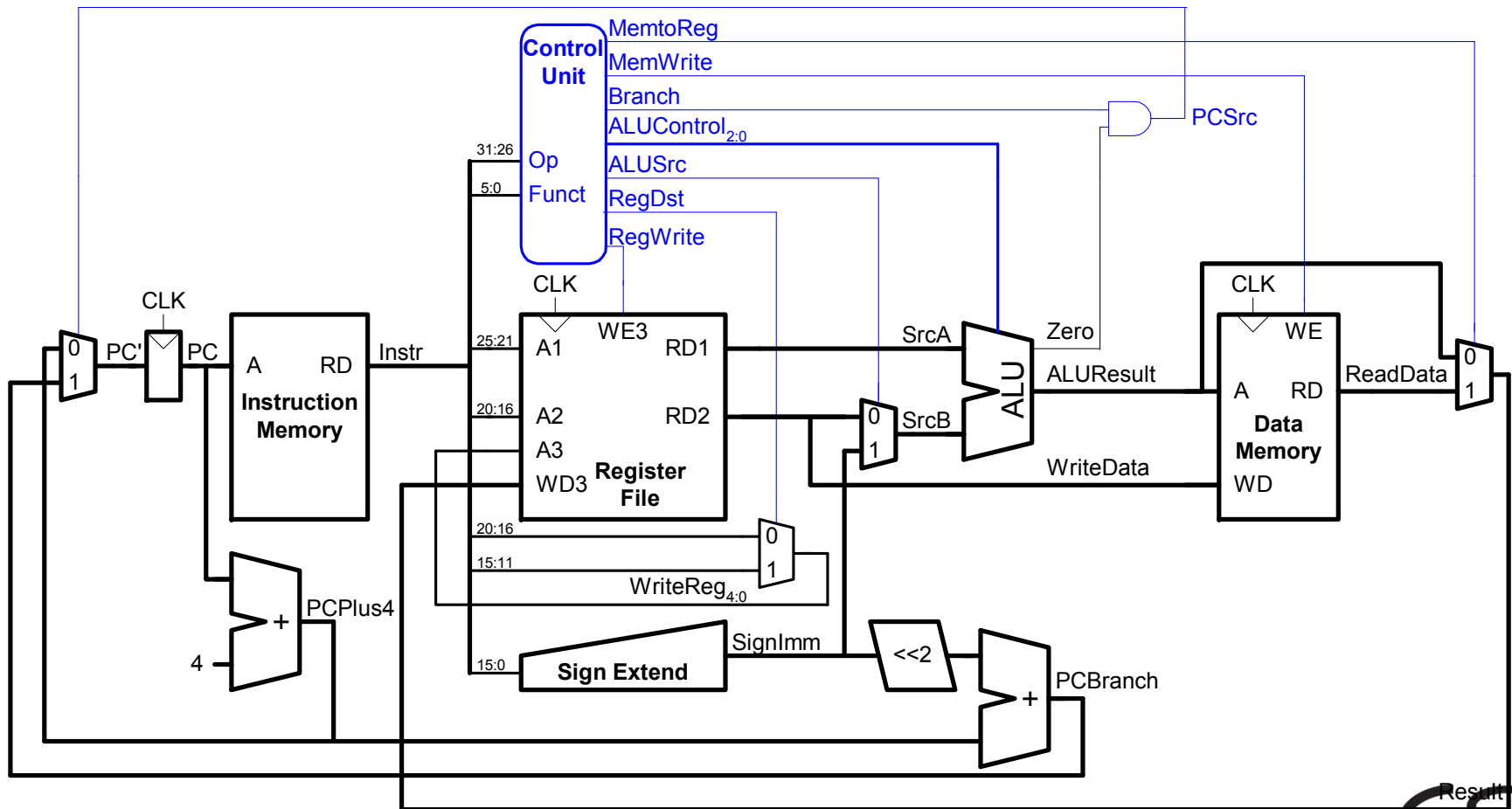
Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

Beispiel im Ein-Takt Datenpfad: `or`



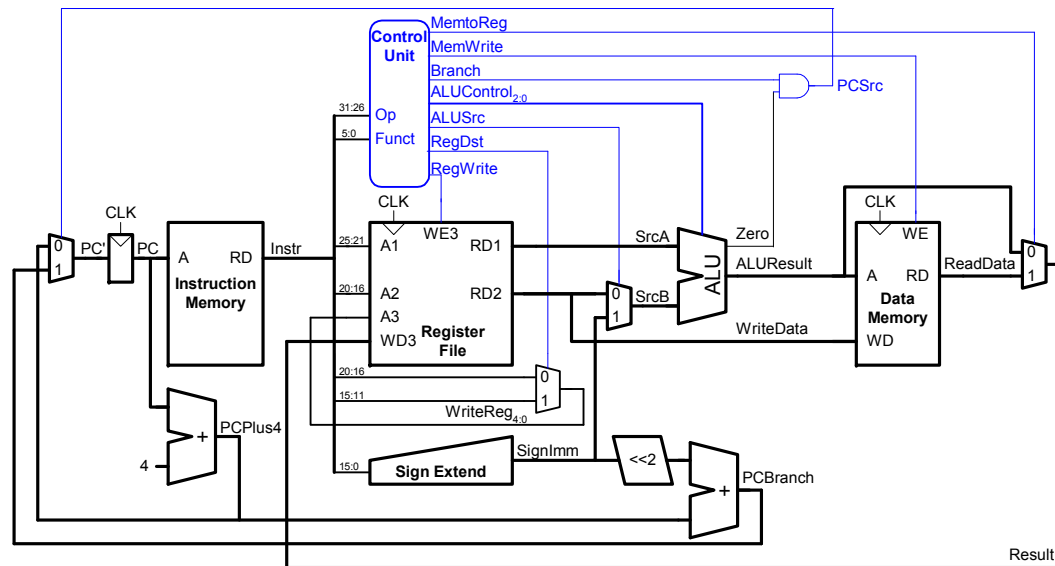
Erweitere Funktionalität: addi

- Keine Änderung am Datenpfad nötig



Erweiterte Steuerwerk: addi

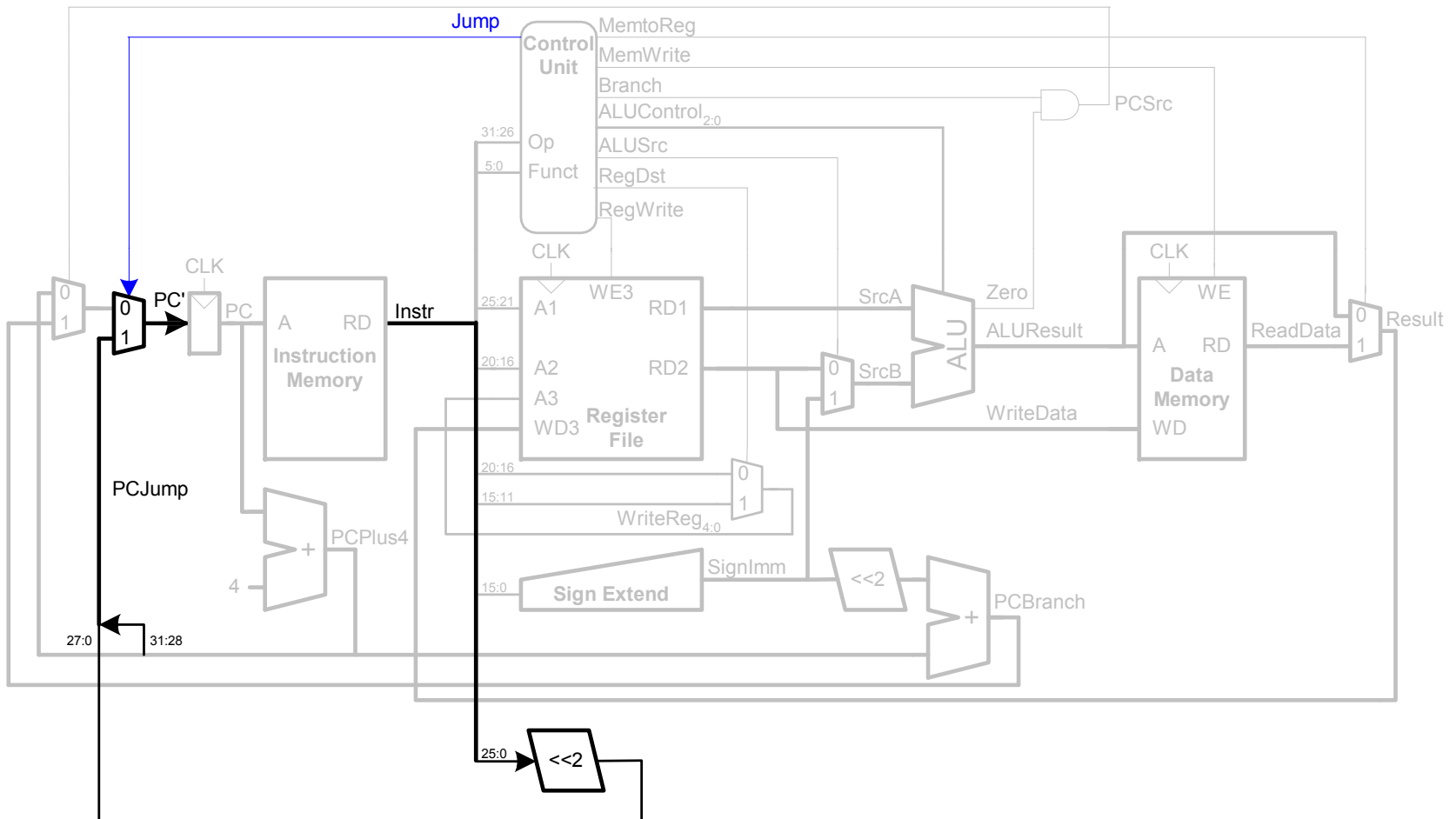
Instruktion	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-Typ	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000							



Erweitere Steuerwerk: addi

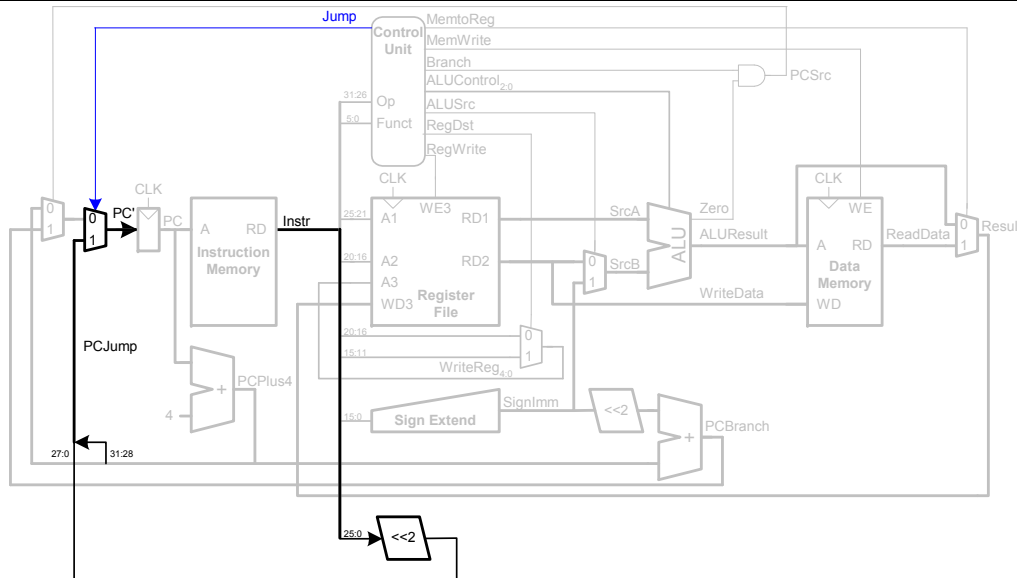
Instruktion	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-Typ	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00

Erweitere Funktionalität: j



Steuerwerk: Hauptdecoder

Instruktion	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}	Jump
R-Typ	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000010								



Steuerwerk: Hauptdecoder

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000010	0	X	X	X	0	X	XX	1

Wiederholung: Rechenleistung des Prozessors



TECHNISCHE
UNIVERSITÄT
DARMSTADT

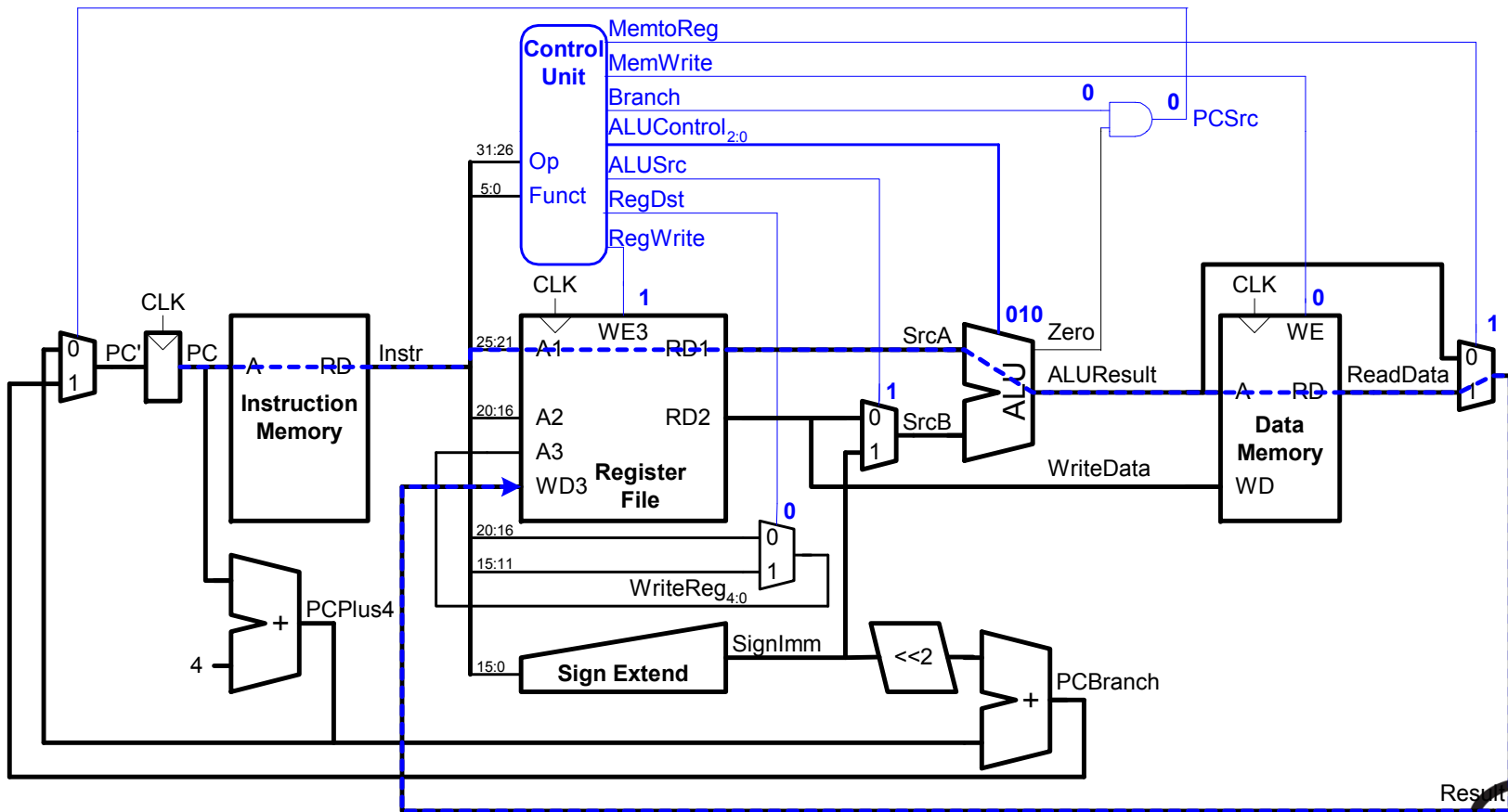
Programmausführungszeit

= (# Instruktionen) (Takte/Instruktion) (Sekunden/Takt)

= # Instruktionen CPI T_c

Rechenleistung des Ein-Takt-Prozessors

- T_C wird durch längsten Pfad bestimmt (\perp_w)



Rechenleistung des Ein-Takt-Prozessors



- **Kritischer Pfad:**

$$T_c = t_{pcq_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

- In **vielen** Implementierungen: Kritischer Pfad durch

- Speicher, ALU, Registerfeld

- **Damit:**

- $T_c = t_{pcq_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$

Ein-Takt Prozessor Rechenleistung: Beispiel



Element	Parameter	Verzögerung (ps)
Register Clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Speicher lesen	t_{mem}	250
Registerfeld lesen	t_{RFread}	150
Registerfeld setup	$t_{RFsetup}$	20

$T_c =$

Ein-Takt Prozessor Rechenleistung: Beispiel



Element	Parameter	Verzögerung (ps)
Register Clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Speicher lesen	t_{mem}	250
Registerfeld lesen	t_{RFread}	150
Registerfeld setup	$t_{RFsetup}$	20

$$\begin{aligned}T_c &= t_{pcq_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup} \\ &= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps} \\ &= 925 \text{ ps}\end{aligned}$$

Ein-Takt Prozessor Rechenleistung: Beispiel



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Auszuführen: Programm mit 100 Milliarden Instruktionen
 - Auf Ein-Takt MIPS Prozessor

Ausführungszeit =

Ein-Takt Prozessor Rechenleistung: Beispiel



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Auszuführen: Programm mit 100 Milliarden Instruktionen
 - Auf Ein-Takt MIPS Prozessor

$$\begin{aligned} \text{Ausführungszeit} &= \# \text{ Instruktionen} \cdot \text{CPI} \cdot T_C \\ &= (100 \times 10^9) (1) (925 \times 10^{-12} \text{ s}) \\ &= 92,5 \text{ Sekunden} \end{aligned}$$



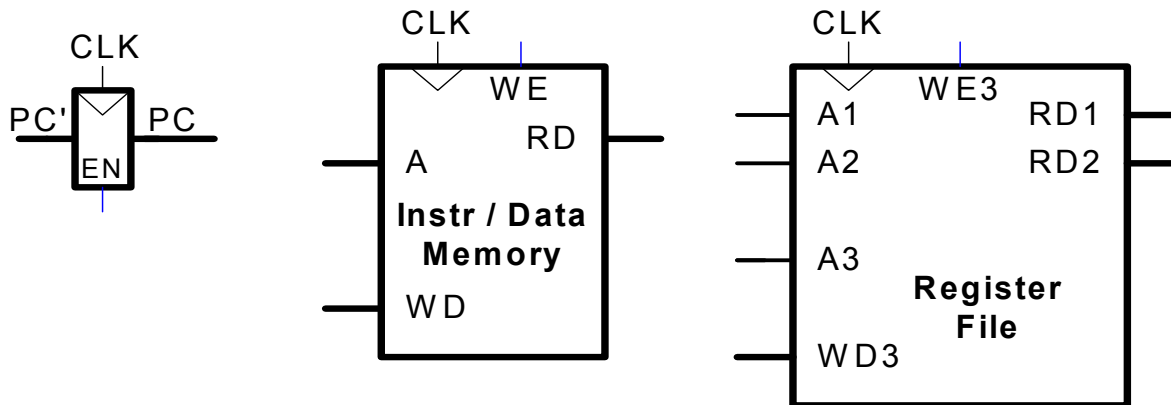
- **Ein-Takt-Mikroarchitektur:**
 - + einfach
 - Taktfrequenz wird durch langsamste Instruktion bestimmt ($\perp w$)
 - Zwei Addierer / ALUs und zwei Speicher

- **Mehrtaktmikroarchitektur:**
 - + höhere Taktfrequenz
 - + einfachere Instruktionen laufen schneller
 - + bessere Wiederverwendung von Hardware in verschiedenen Takten
 - aufwendigere Ablaufsteuerung

- **Gleiche Grundkomponenten**
 - Datenpfad
 - Steuerwerk

Zustandselemente im Mehrtaktprozessor

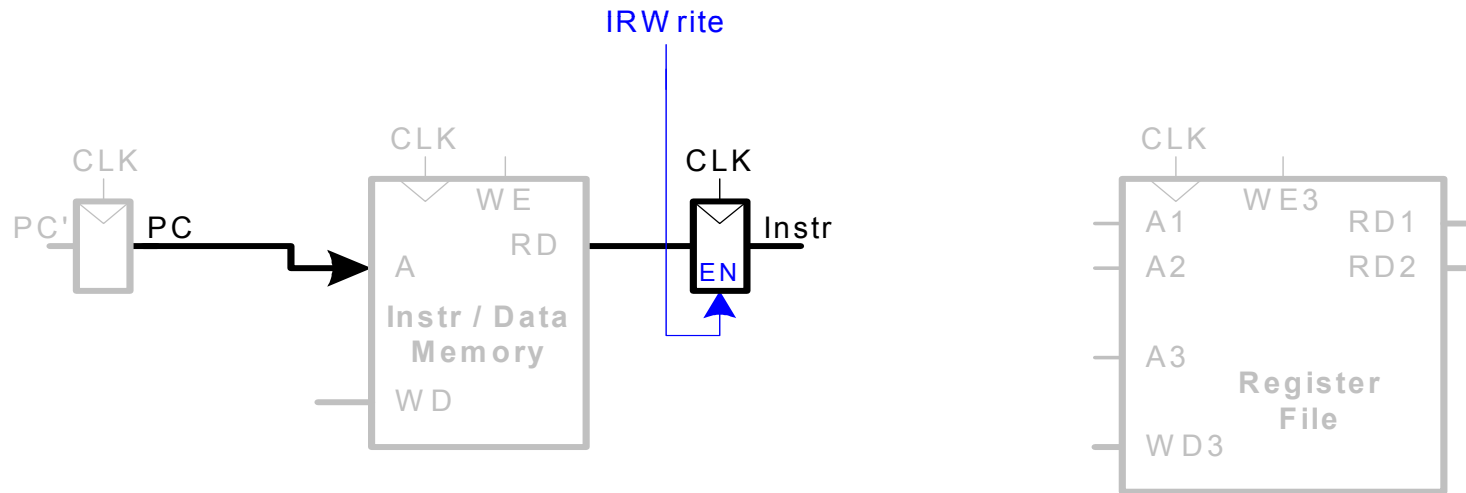
- Ersetze getrennte Instruktions- und Datenspeicher
 - Harvard-Architektur
- Durch einen gemeinsamen Speicher
 - Von Neumann-Architektur
 - Heute weiter verbreitet



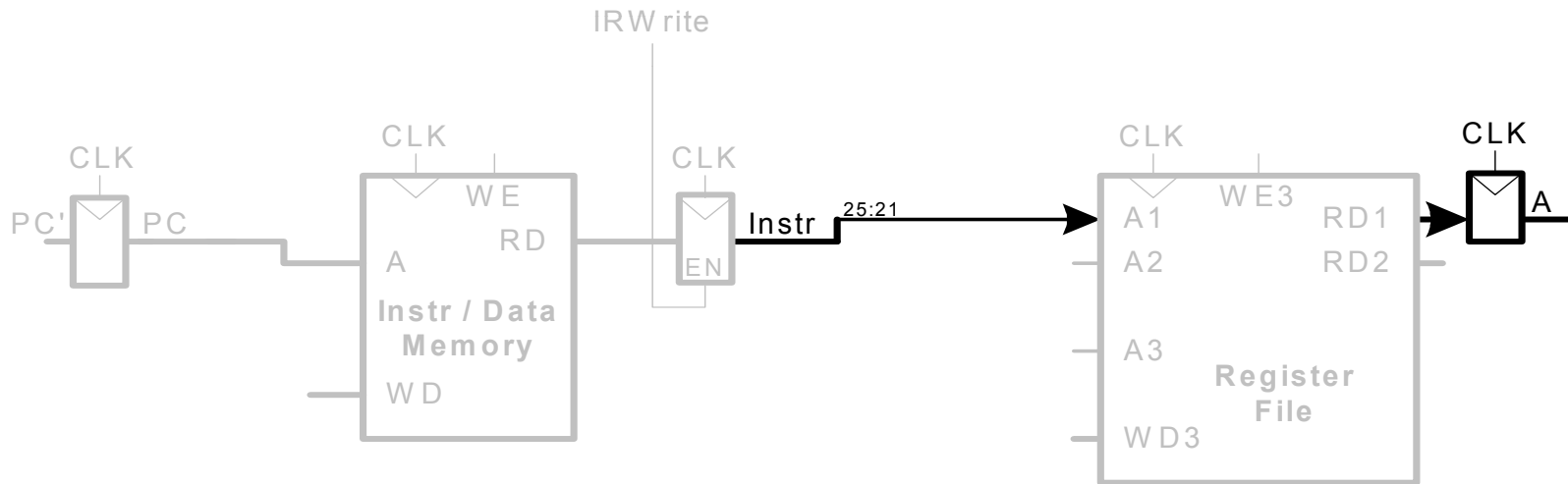
Mehrtaktdatenpfad: Instruktionen holen (*fetch*)



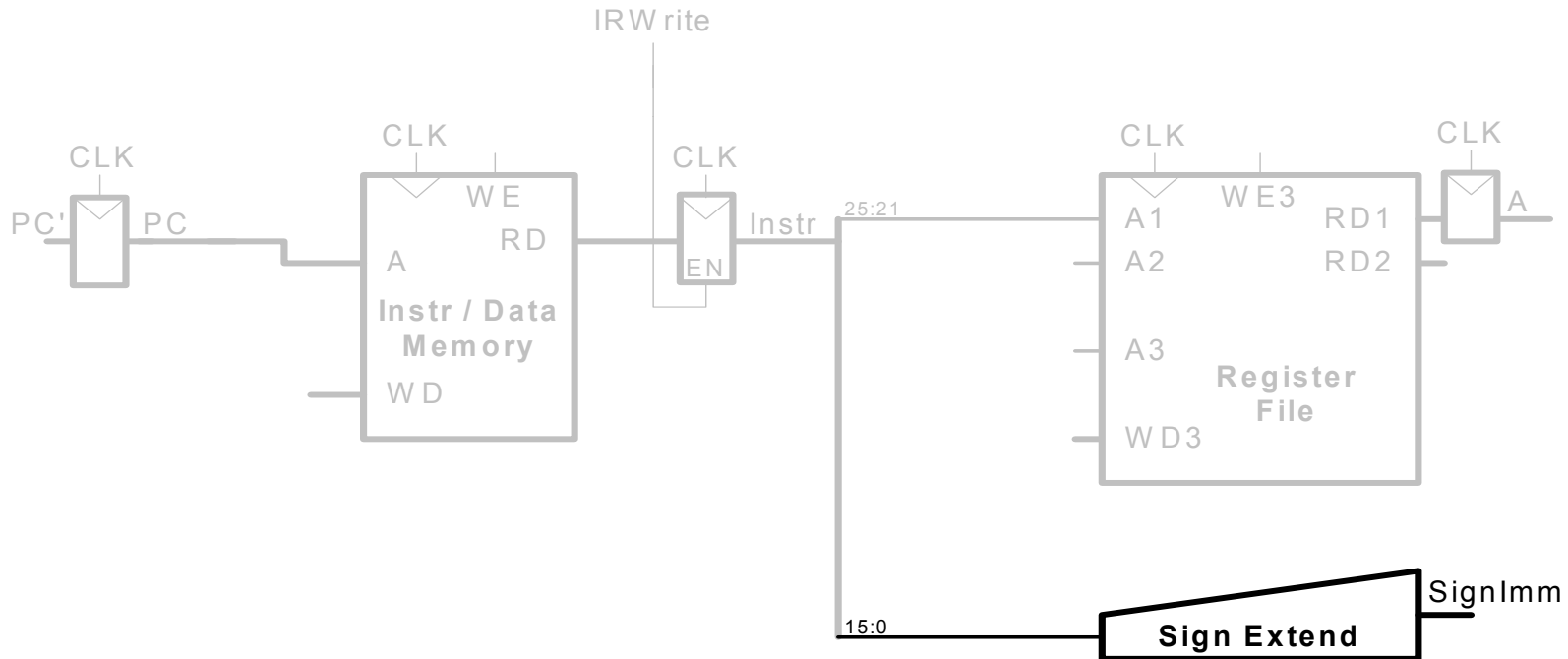
- Beispiel: Ausführung von l_w
- **Schritt 1:** Hole Instruktion



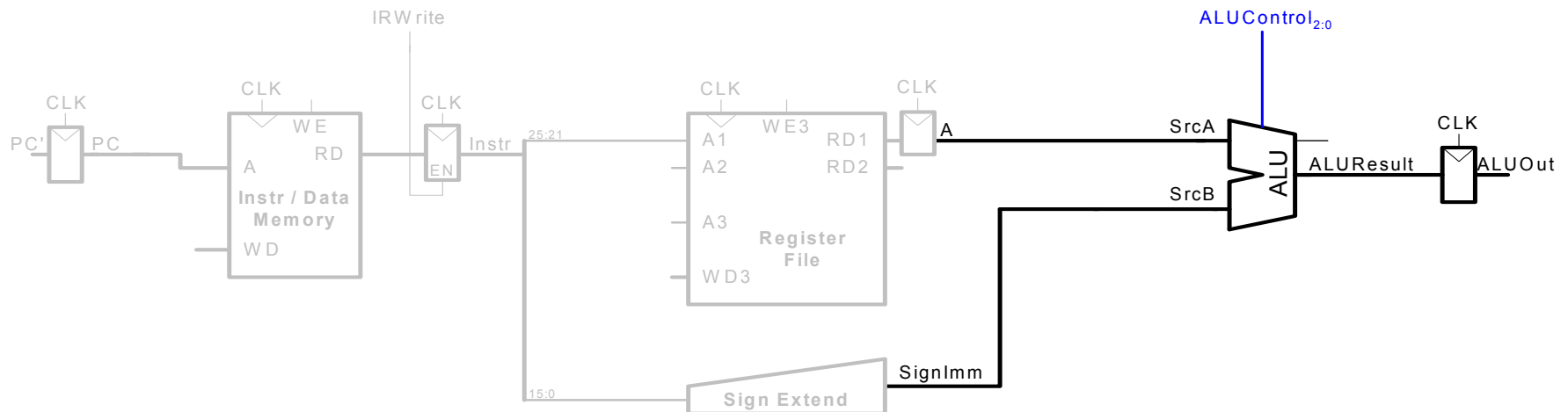
Mehrtaktdatenpfad: Lese Register für 1w



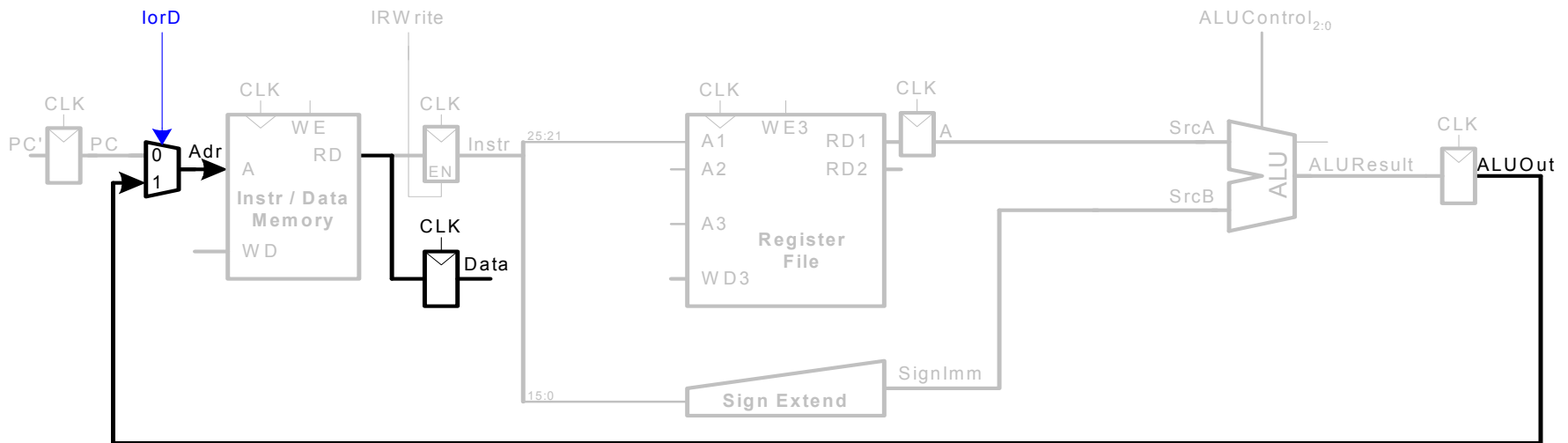
Mehrtaktdatenpfad: Werte 1w Direktwert aus



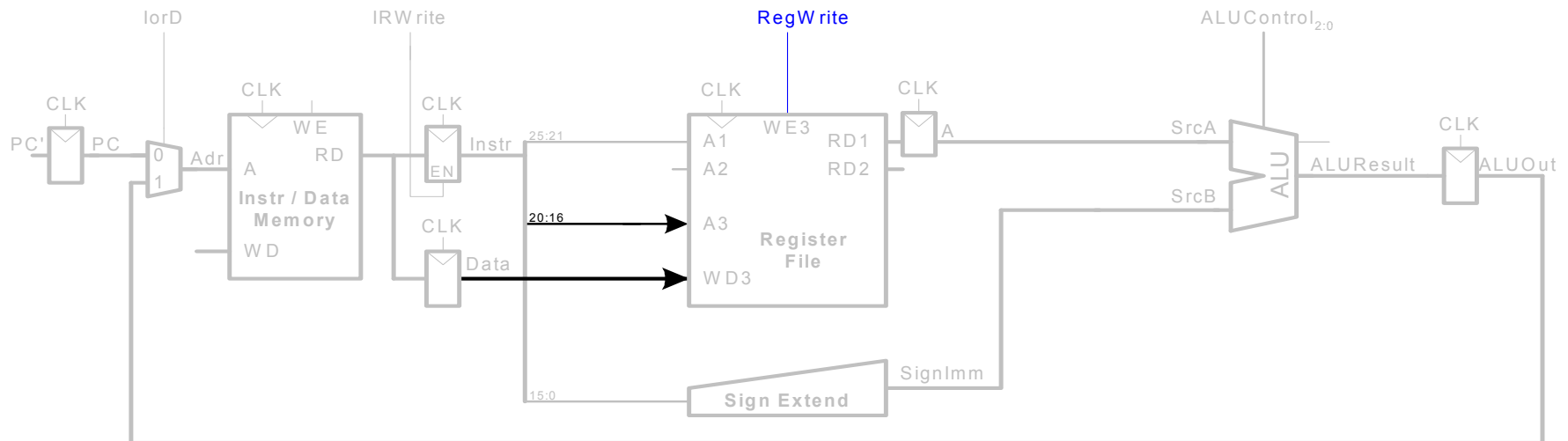
Mehrtaktdatenpfad: Bestimme effektive Adresse für 1w



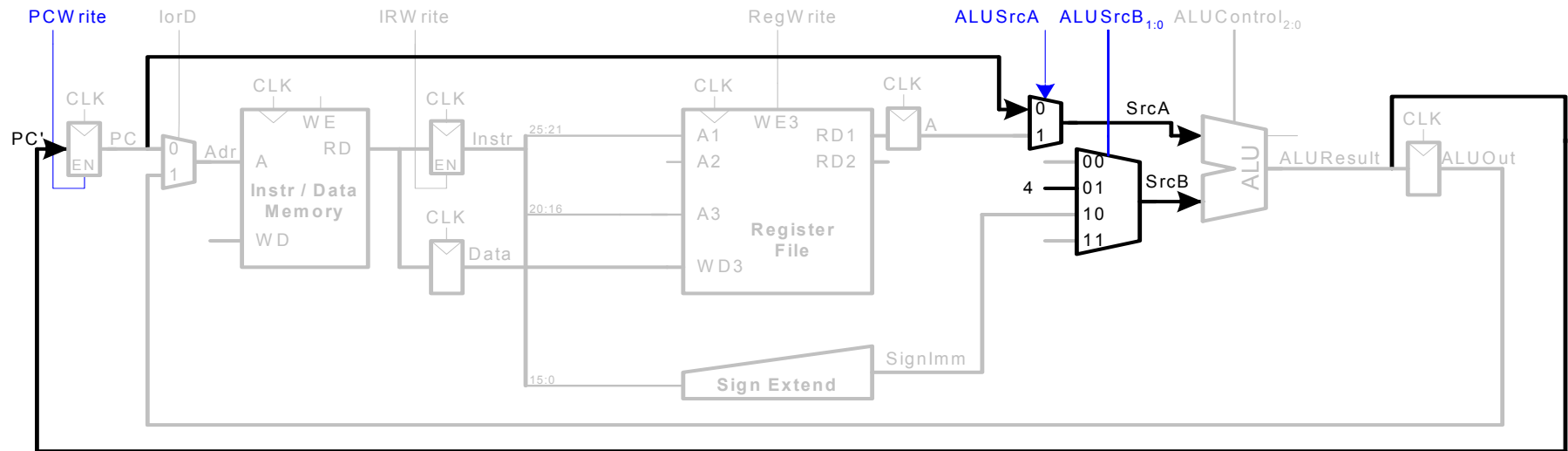
Mehrtaktdatenpfad: Lesezugriff von 1w



Mehrtaktdatenpfad: Schreibe Register in 1w

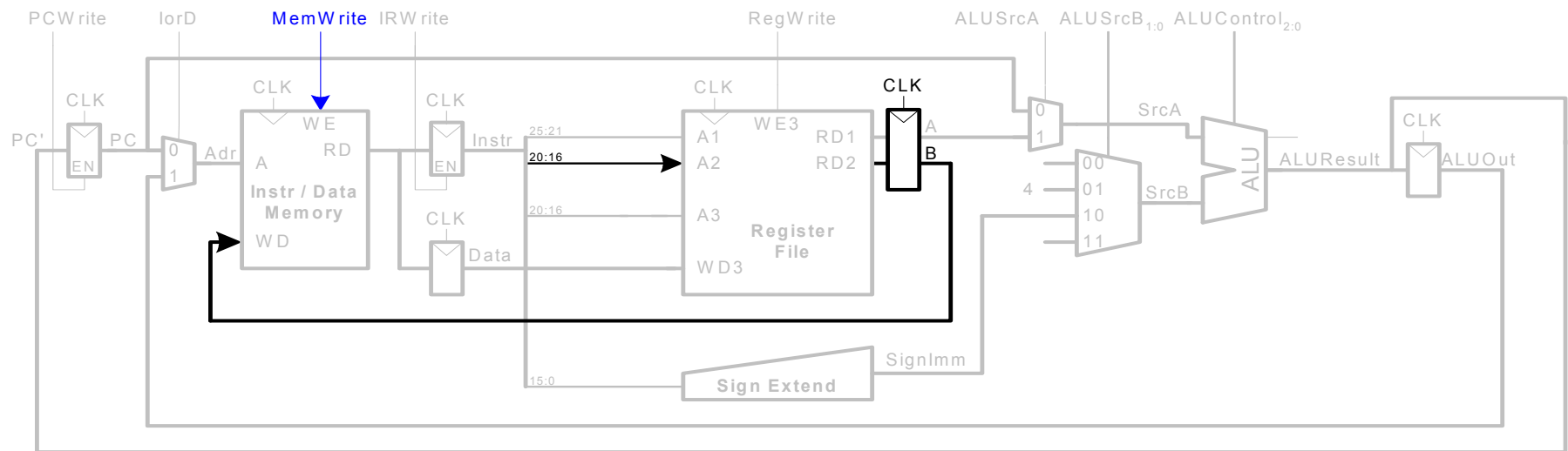


Mehrtaktdatenpfad: Erhöhe PC



Mehrtaktdatenpfad: Nun Ausführung von `sw`

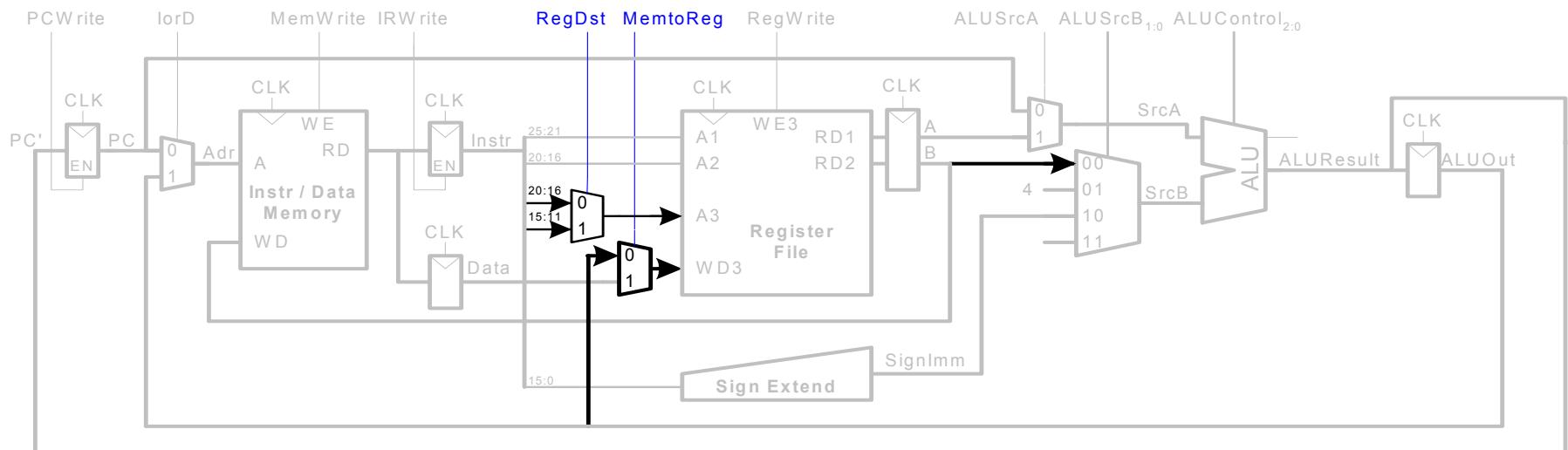
- Schreibe Daten aus `rt` in Speicher



Mehrtaktdatenpfad: Instruktion vom R-Type

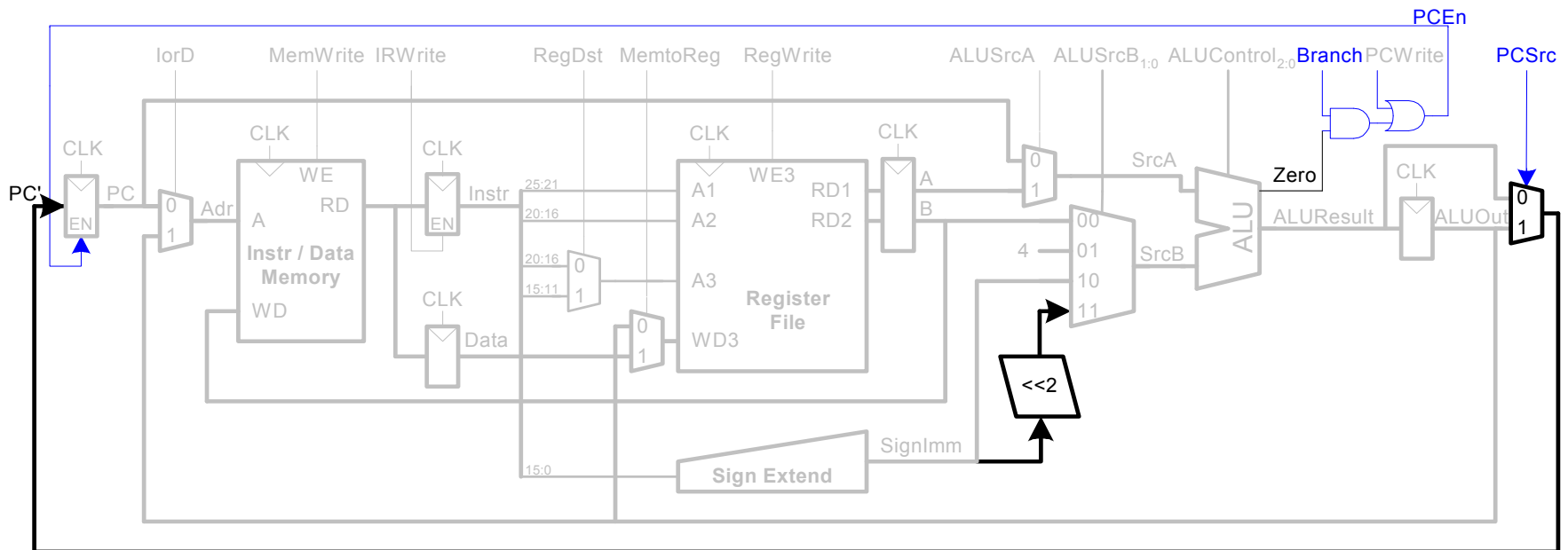


- Lese Werte aus rs und rt
- Schreibe $ALUResult$ ins Registerfeld
- Schreibe Wert nach rd (statt nach rt)

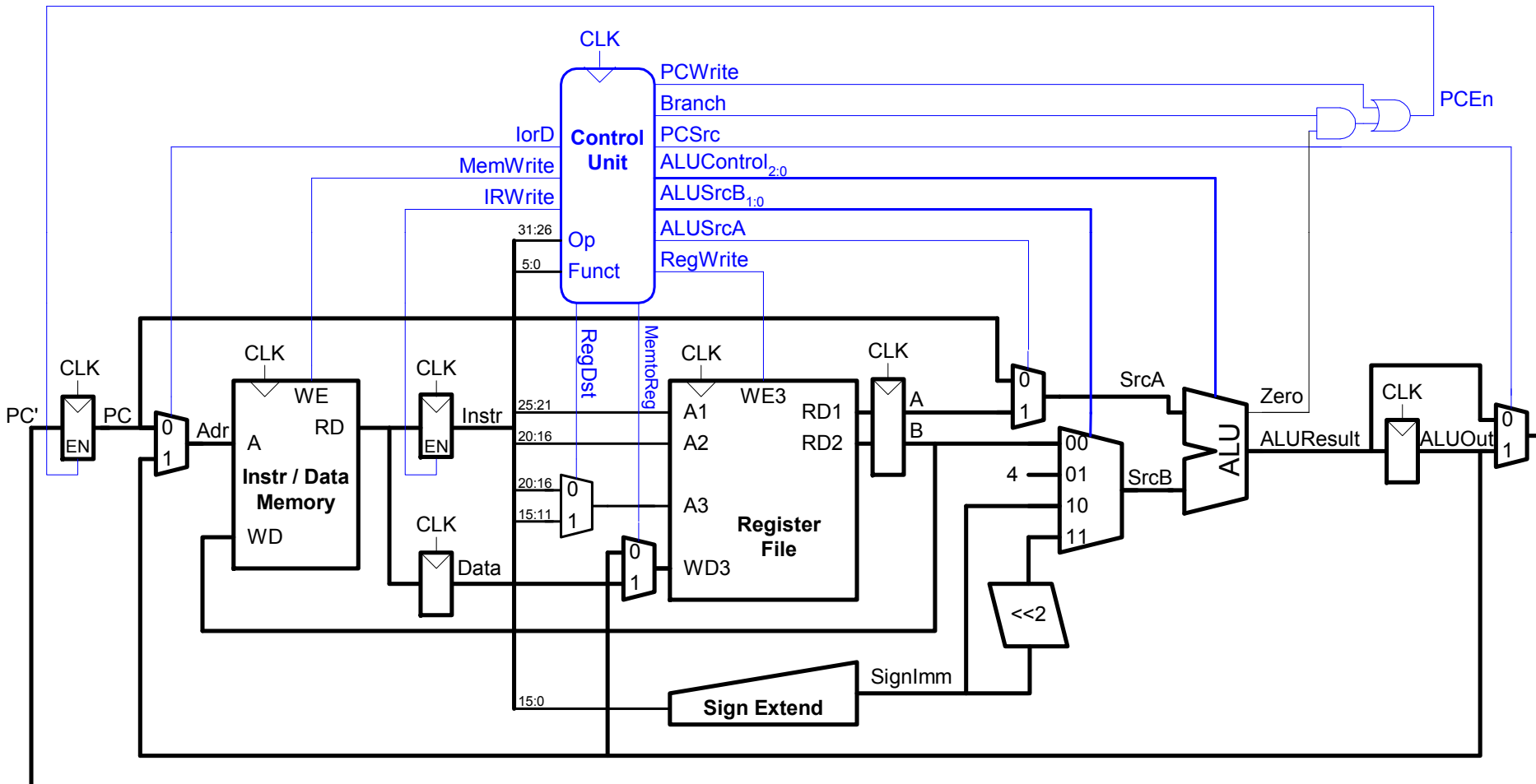


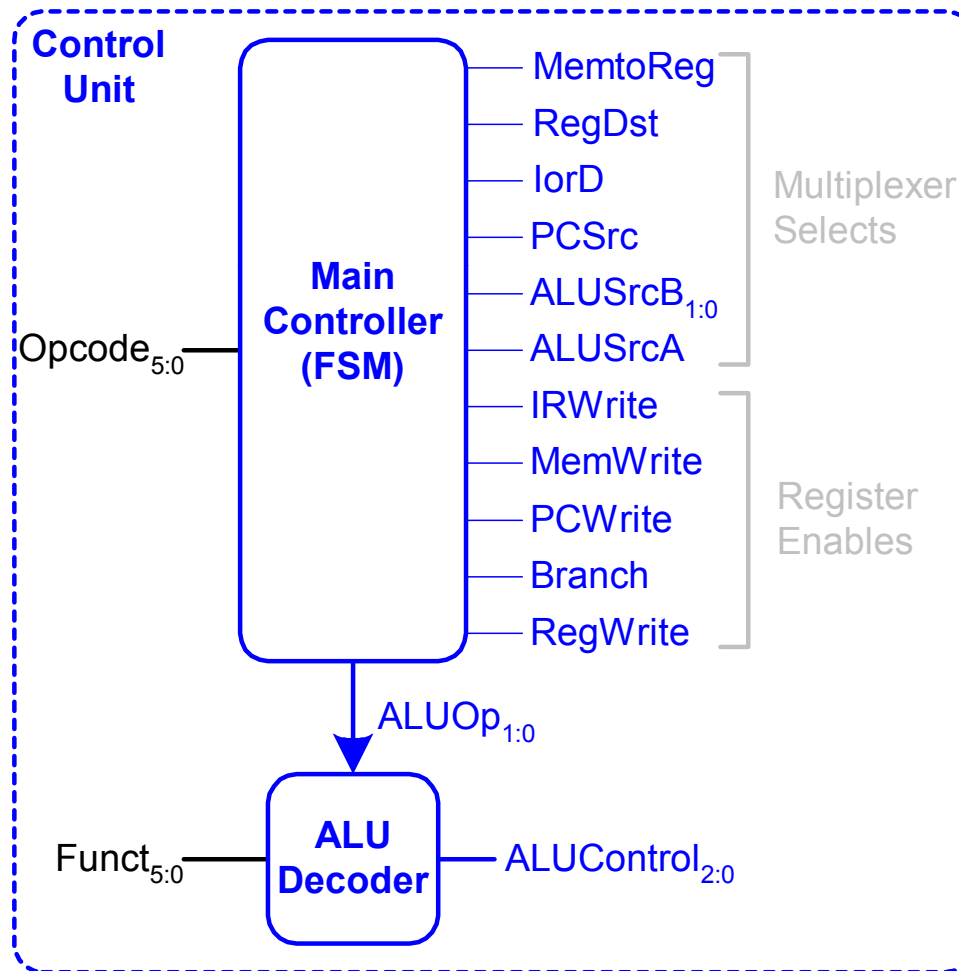
Mehrtaktdatenpfad: beq

- Prüfe, ob Werte in r_s und r_t gleich sind
- Bestimme Adresse des Sprungziels (*branch target address*):
 $BTA = (\text{vorzeichenerweiterter Direktwert} \ll 2) + (PC+4)$

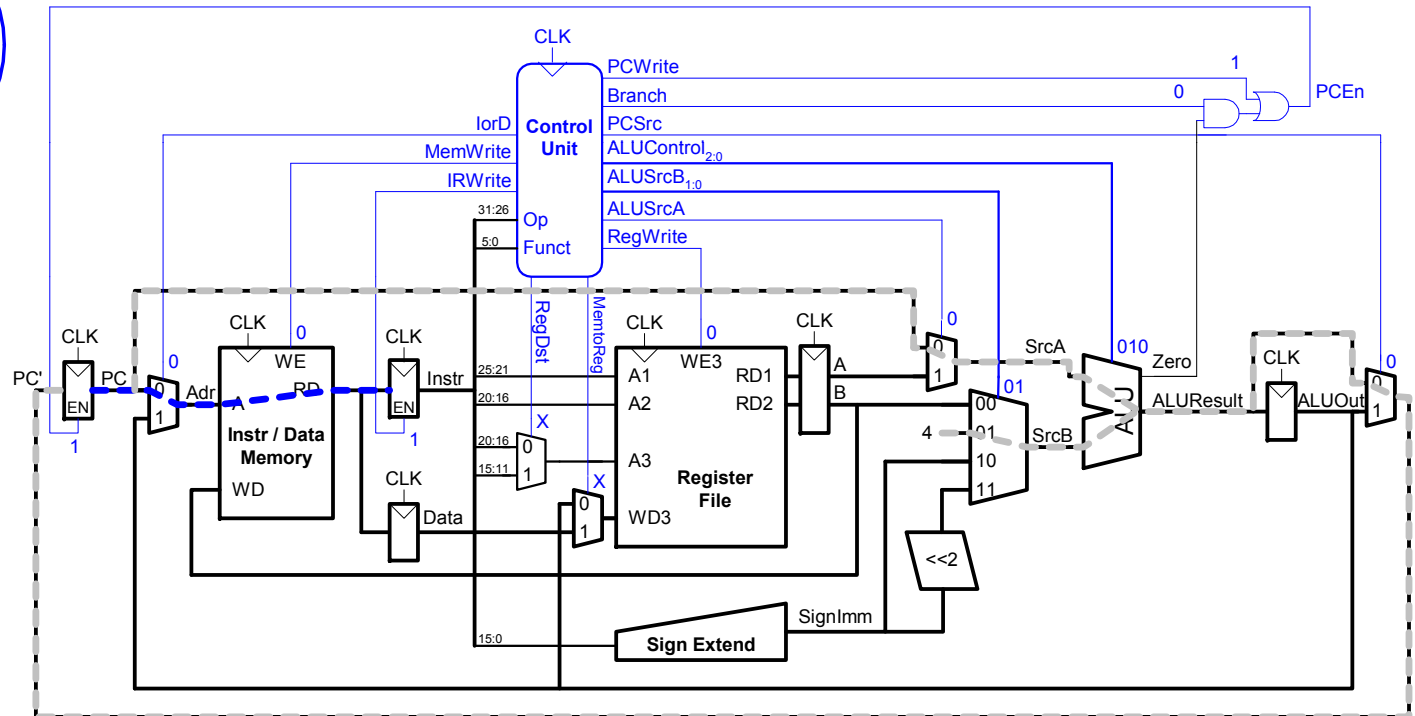
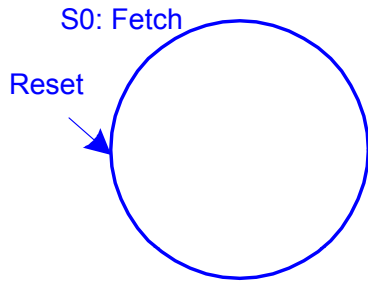


Vollständiger Mehrtaktprozessor





Hauptsteuerwerk: Holen eines Befehls

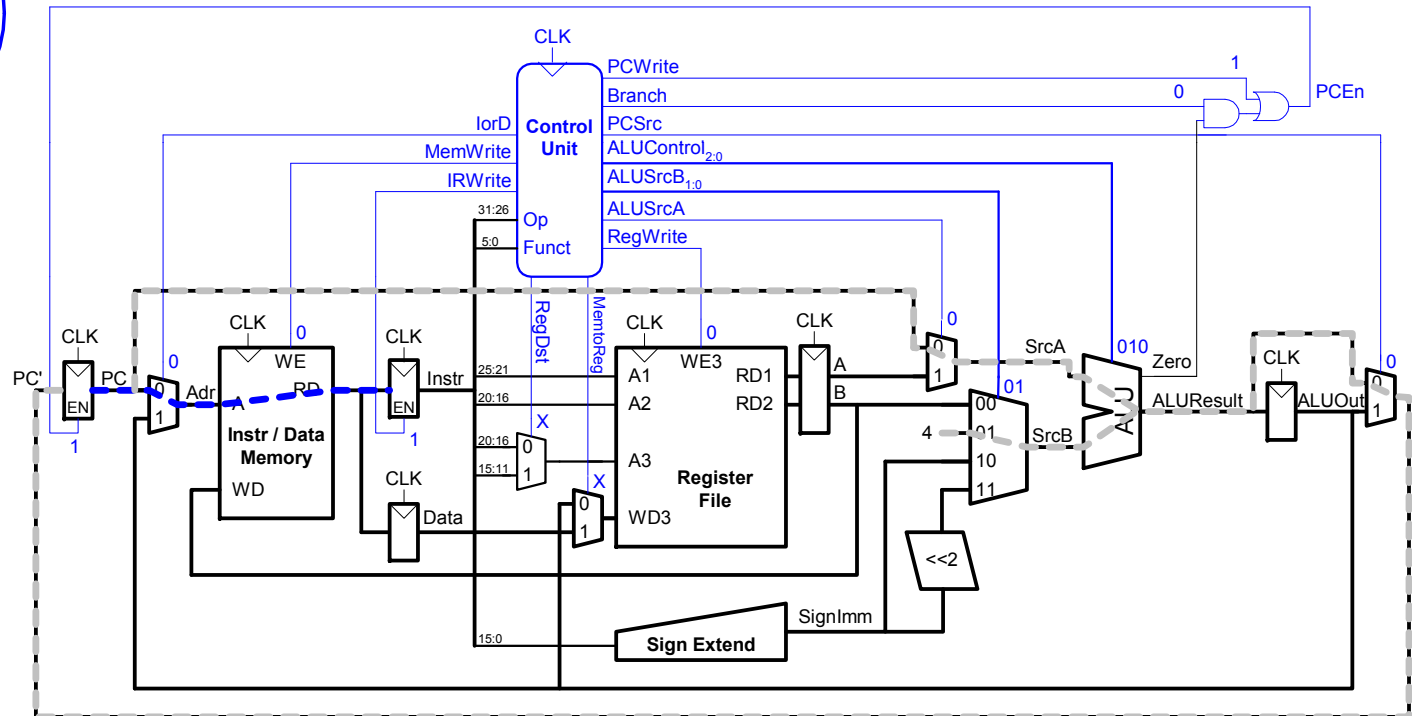


Hauptsteuerwerk: Holen eines Befehls

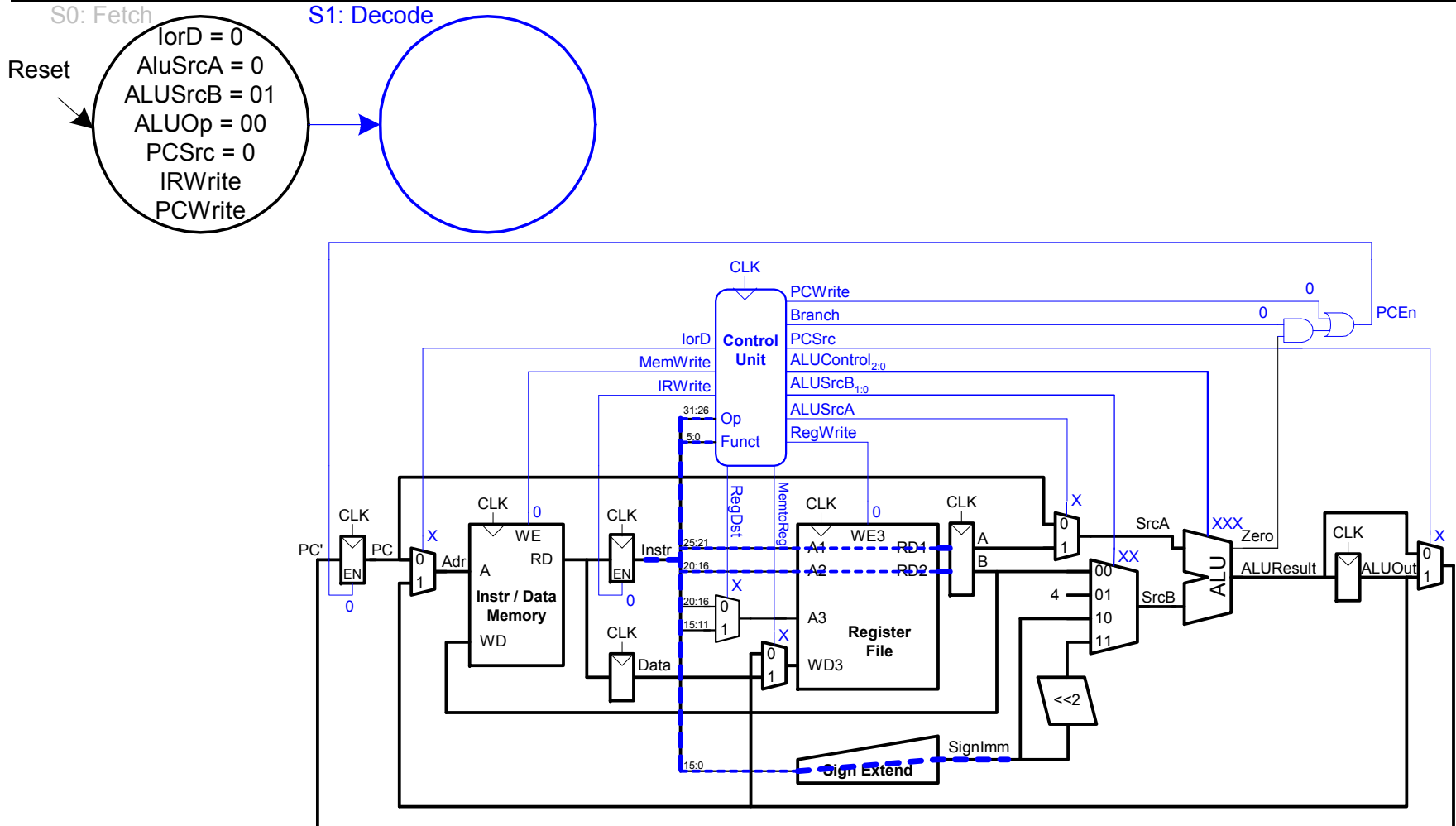
S0: Fetch

Reset

lorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite



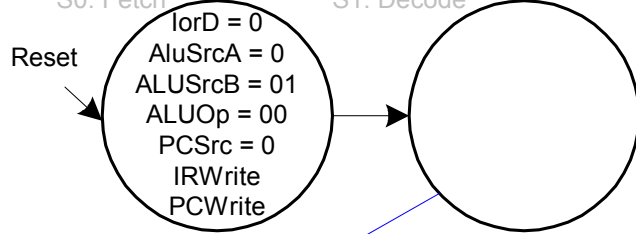
Hauptsteuerwerk: Dekodieren eines Befehls



Hauptsteuerwerk: Adressberechnung

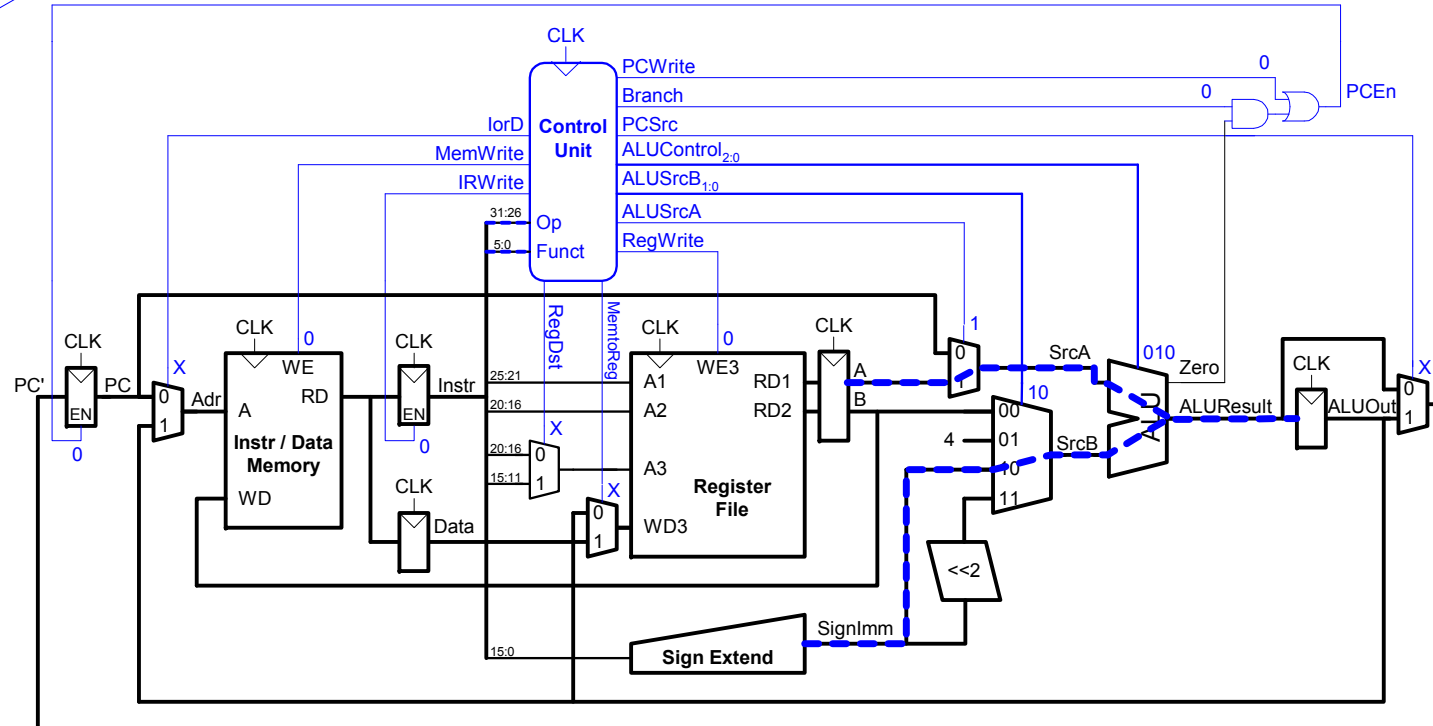
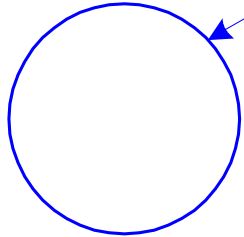
S0: Fetch

S1: Decode



Op = LW
or
Op = SW

S2: MemAdr

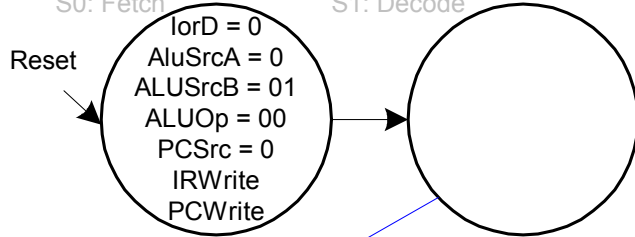


Hauptsteuerwerk: Adressberechnung



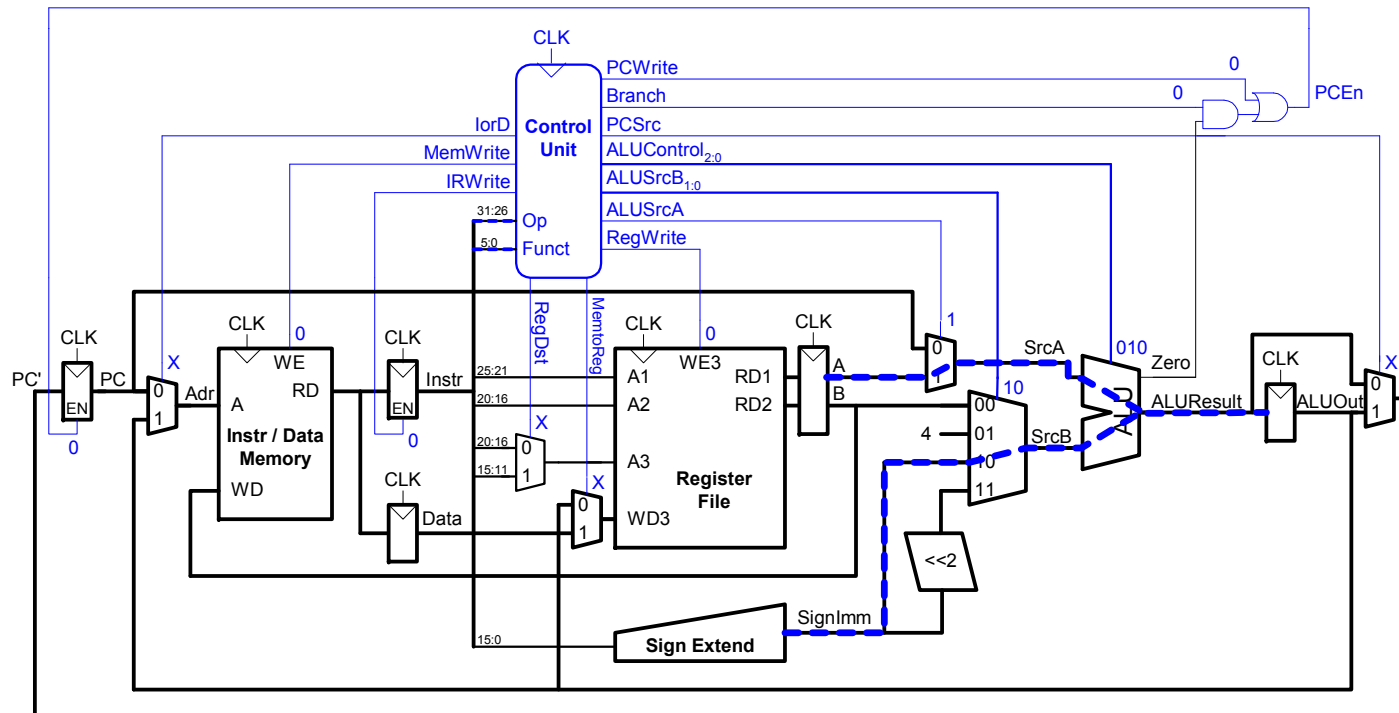
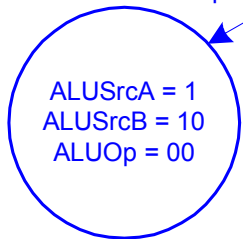
S0: Fetch

S1: Decode

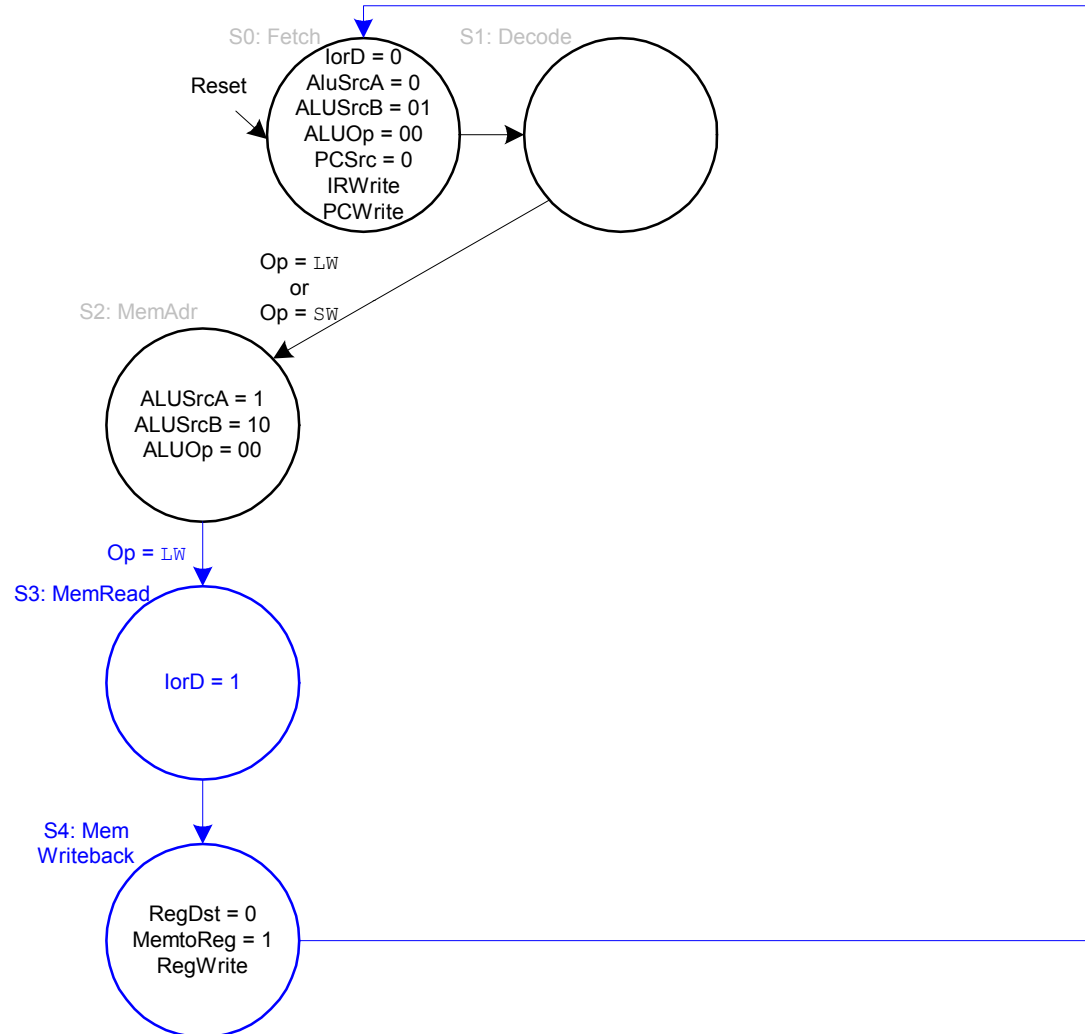


Op = LW
or
Op = SW

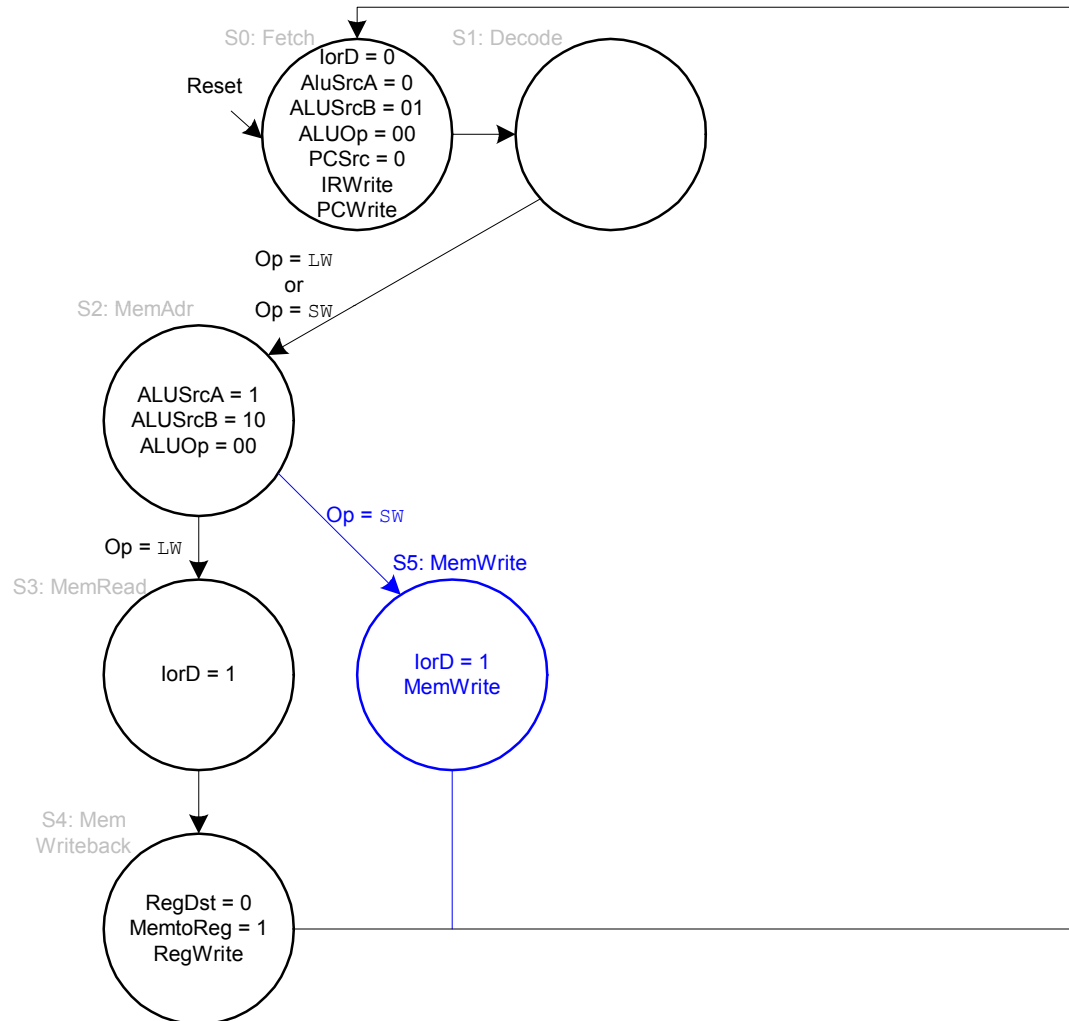
S2: MemAdr



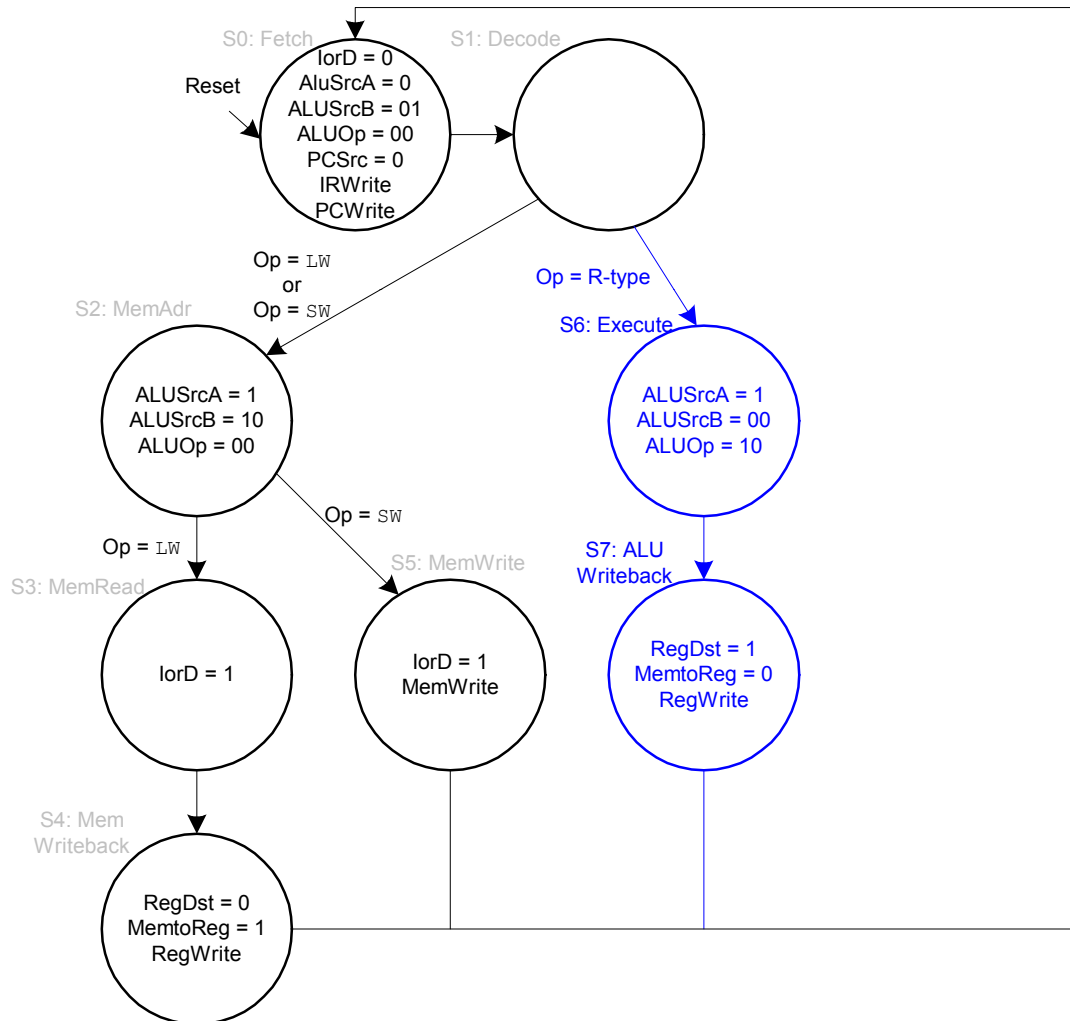
Hauptsteuerwerk: FSM für lw



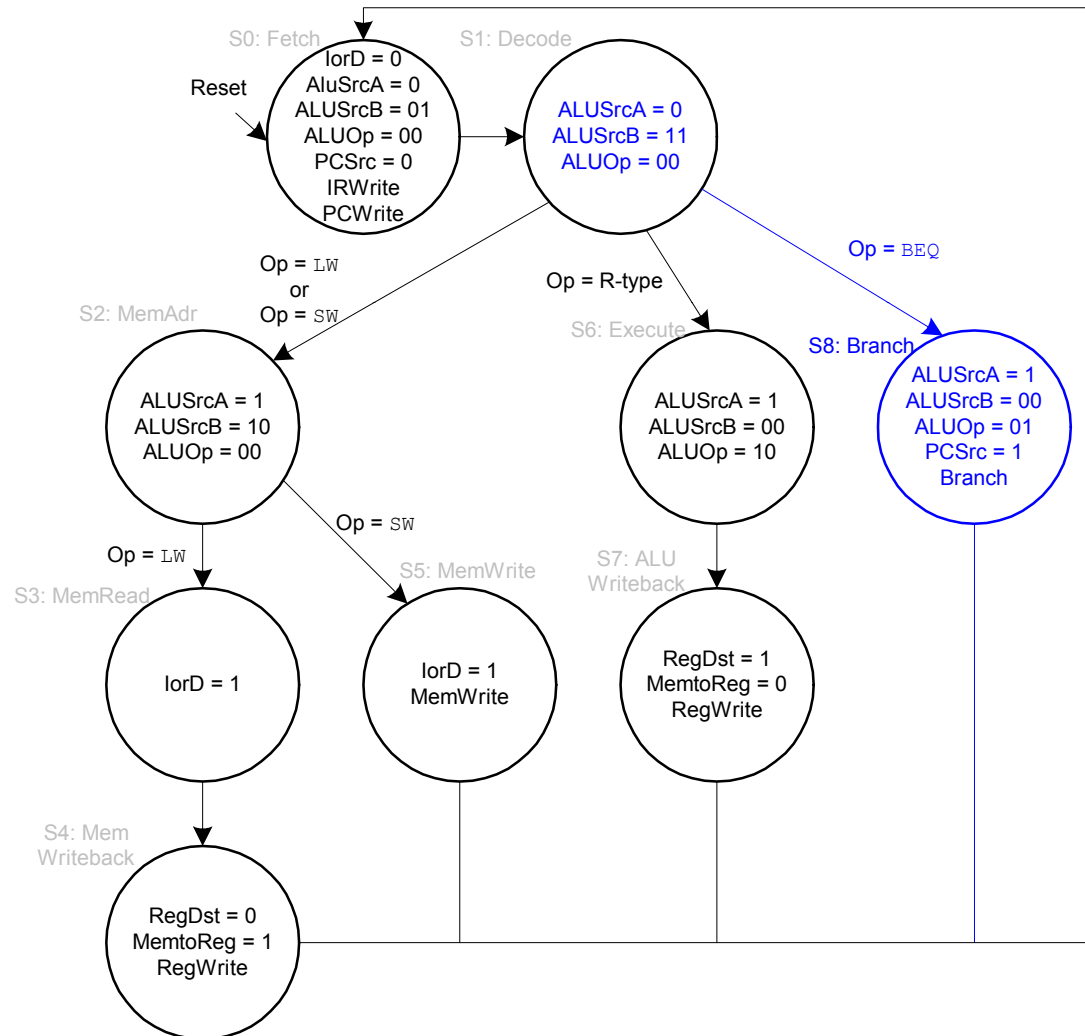
Hauptsteuerwerk: FSM für sw



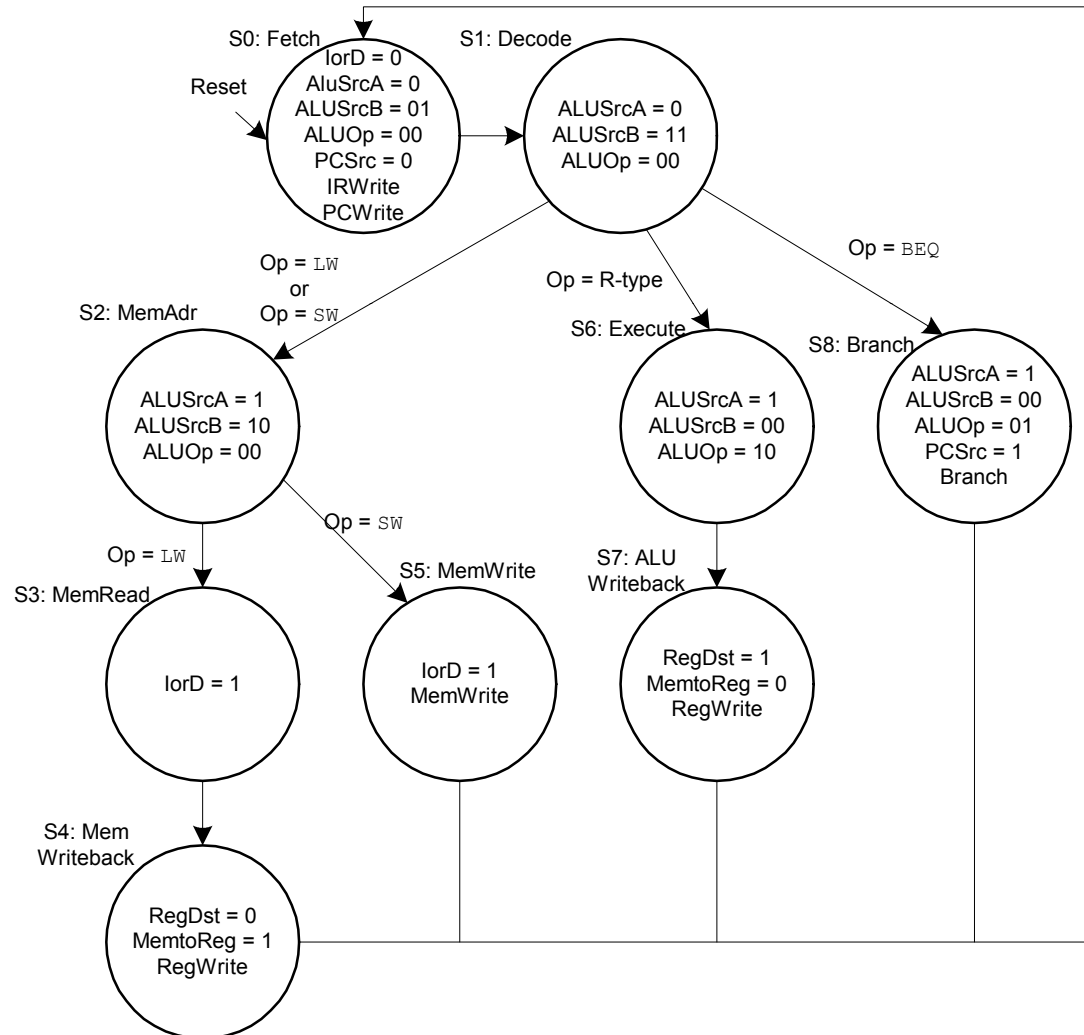
Hauptsteuerwerk: FSM für R-Typ



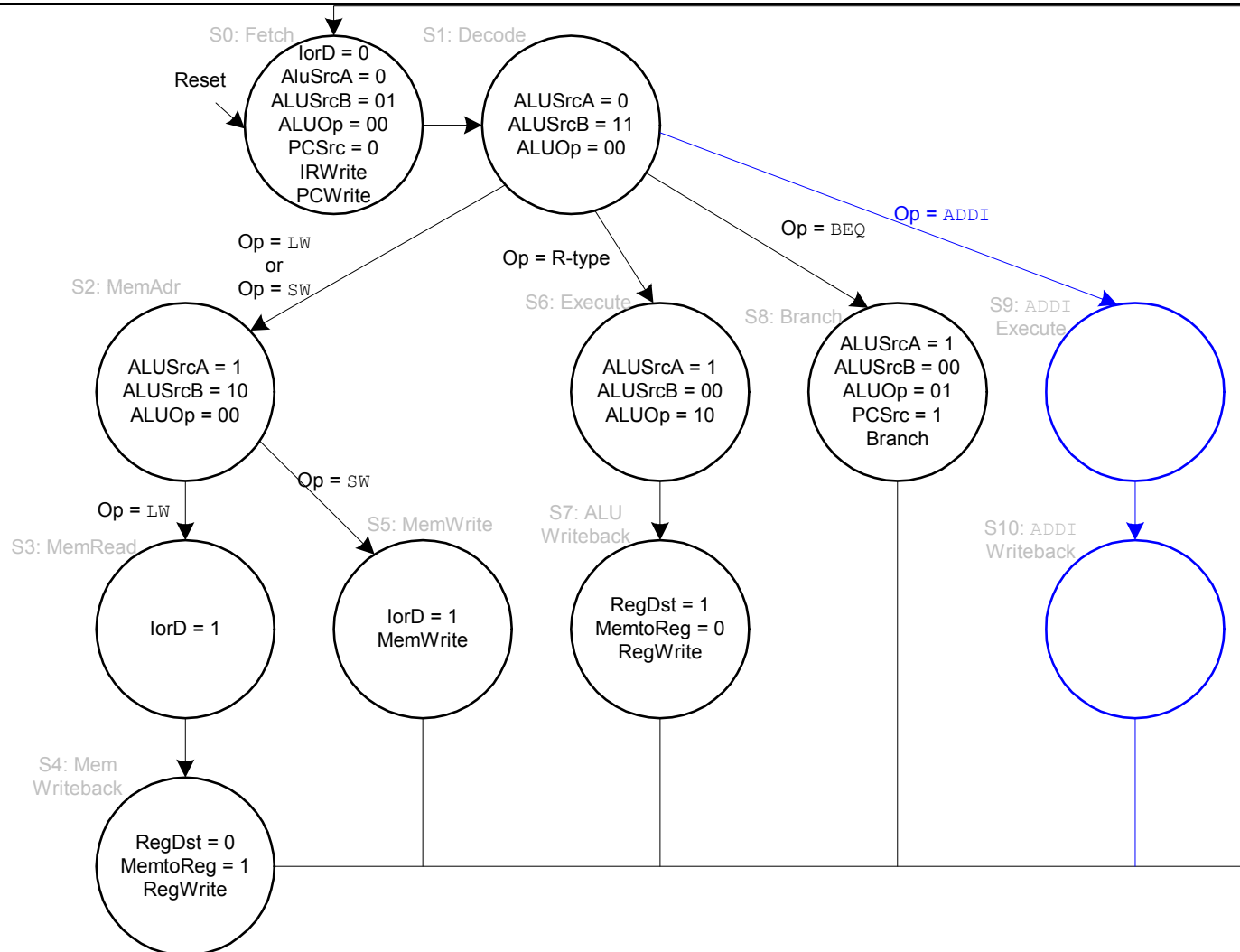
Hauptsteuerwerk: FSM für beq



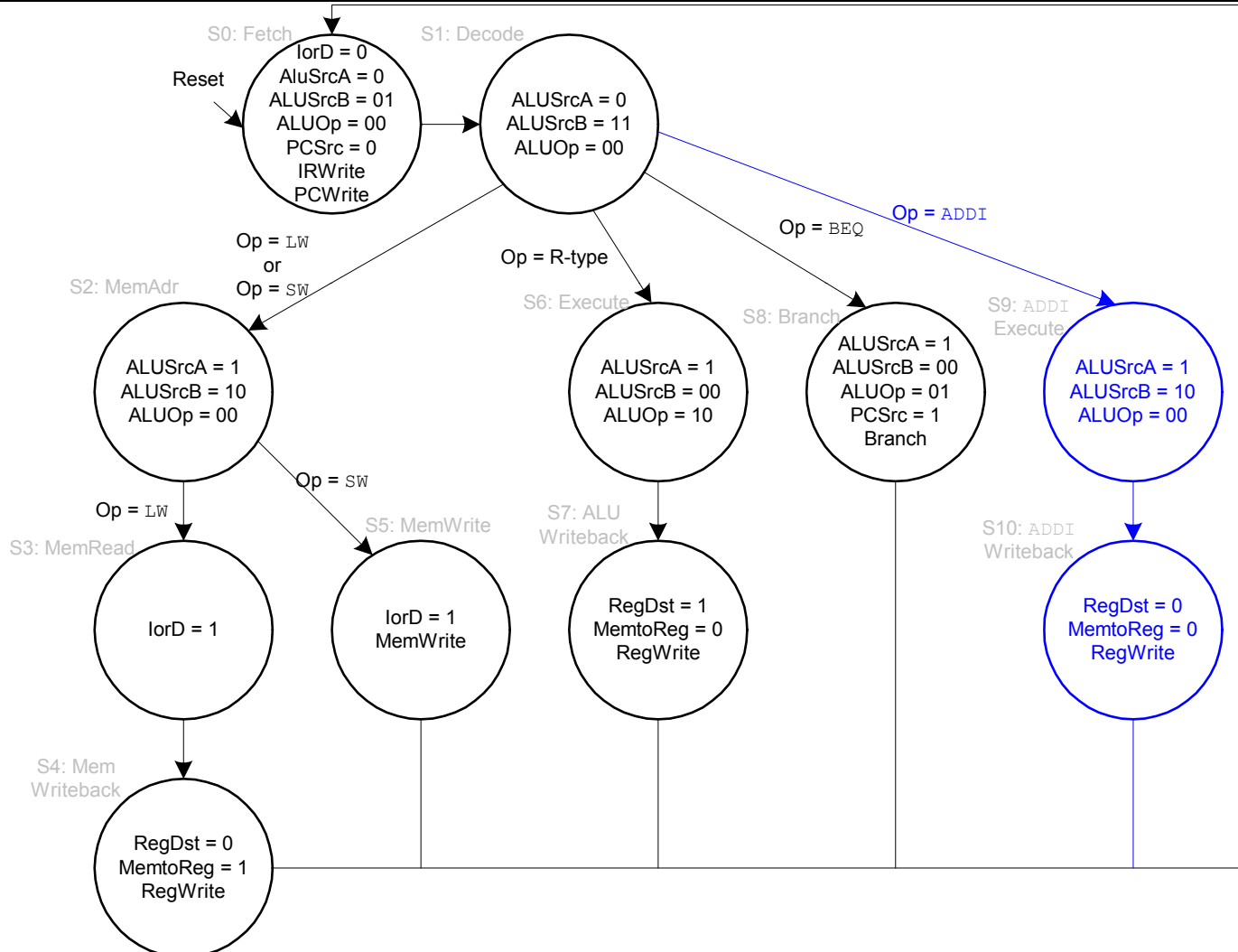
Vollständiges Hauptsteuerwerk für Mehrtakt-CPU



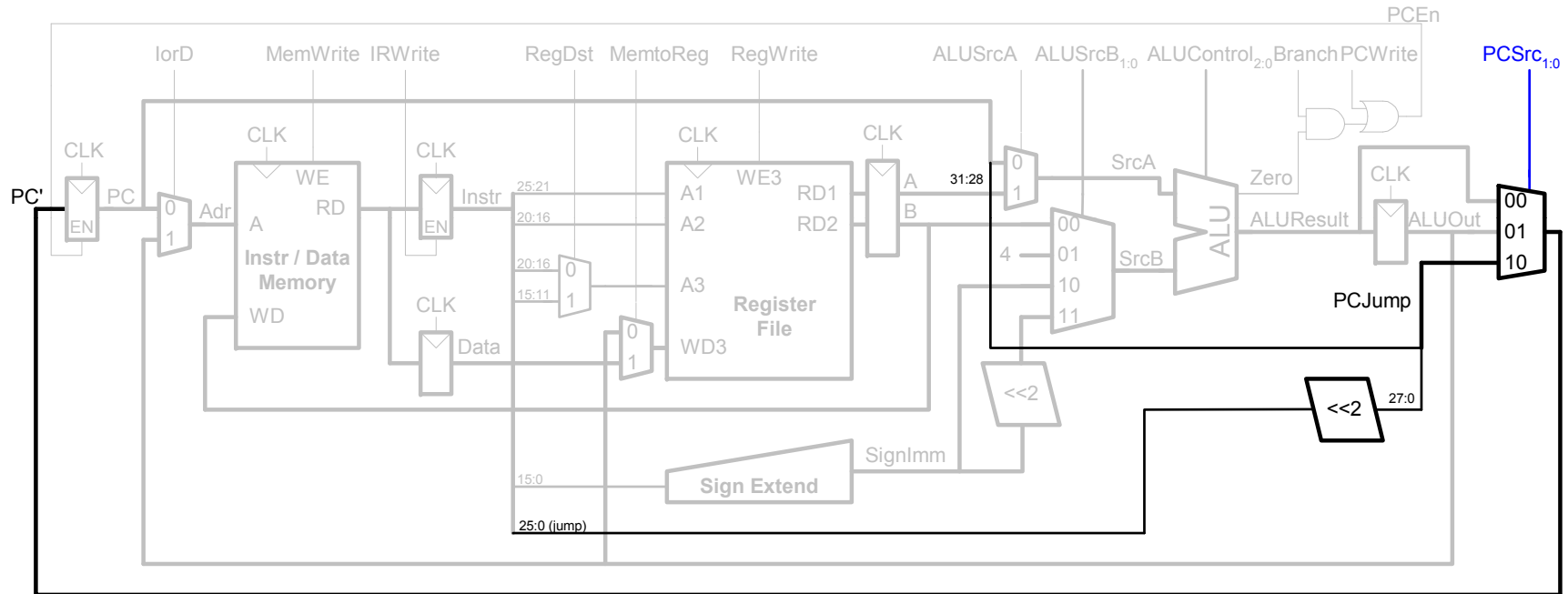
Erweiterung des Hauptsteuerwerks: addi



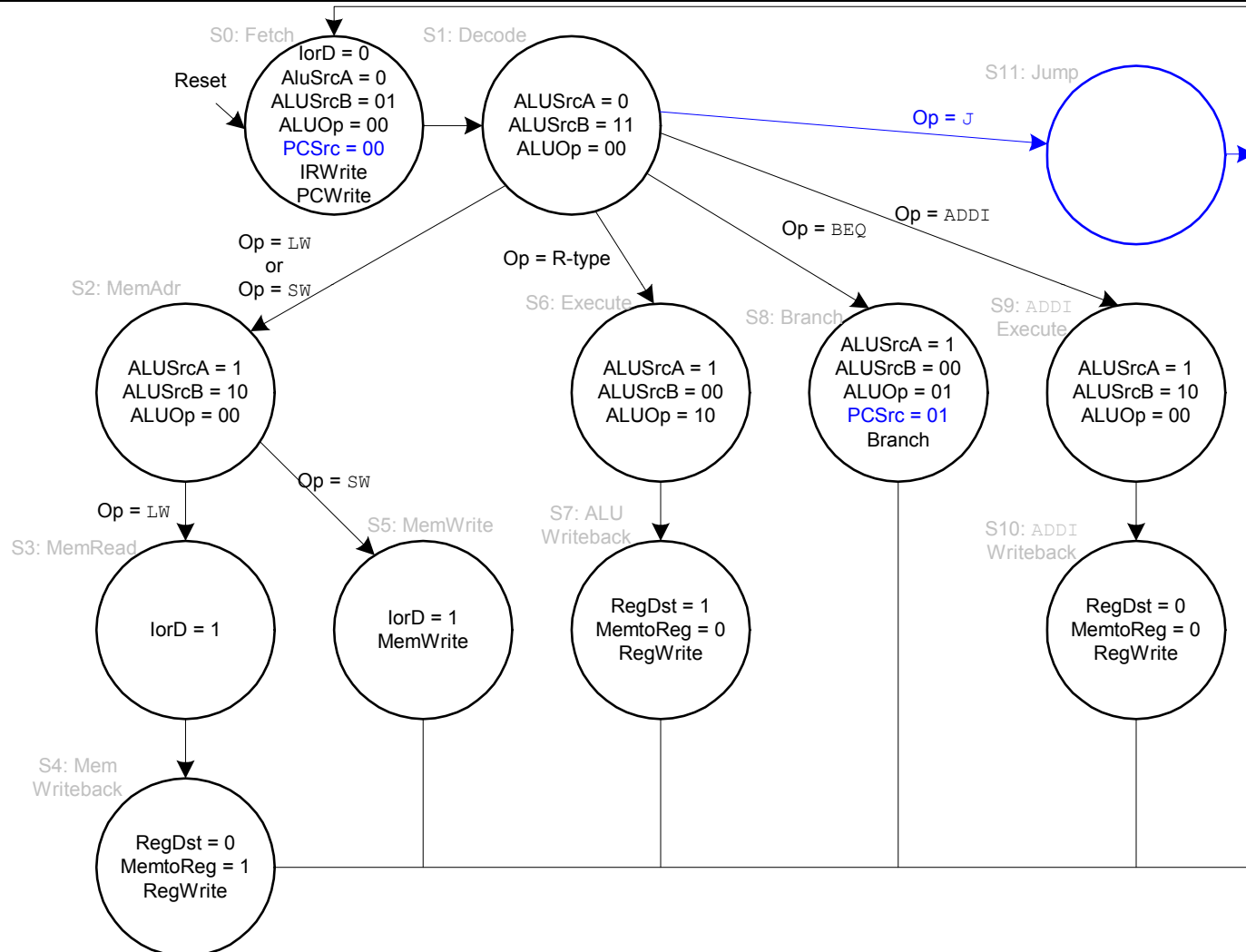
Erweiterung des Hauptsteuerwerks: addi



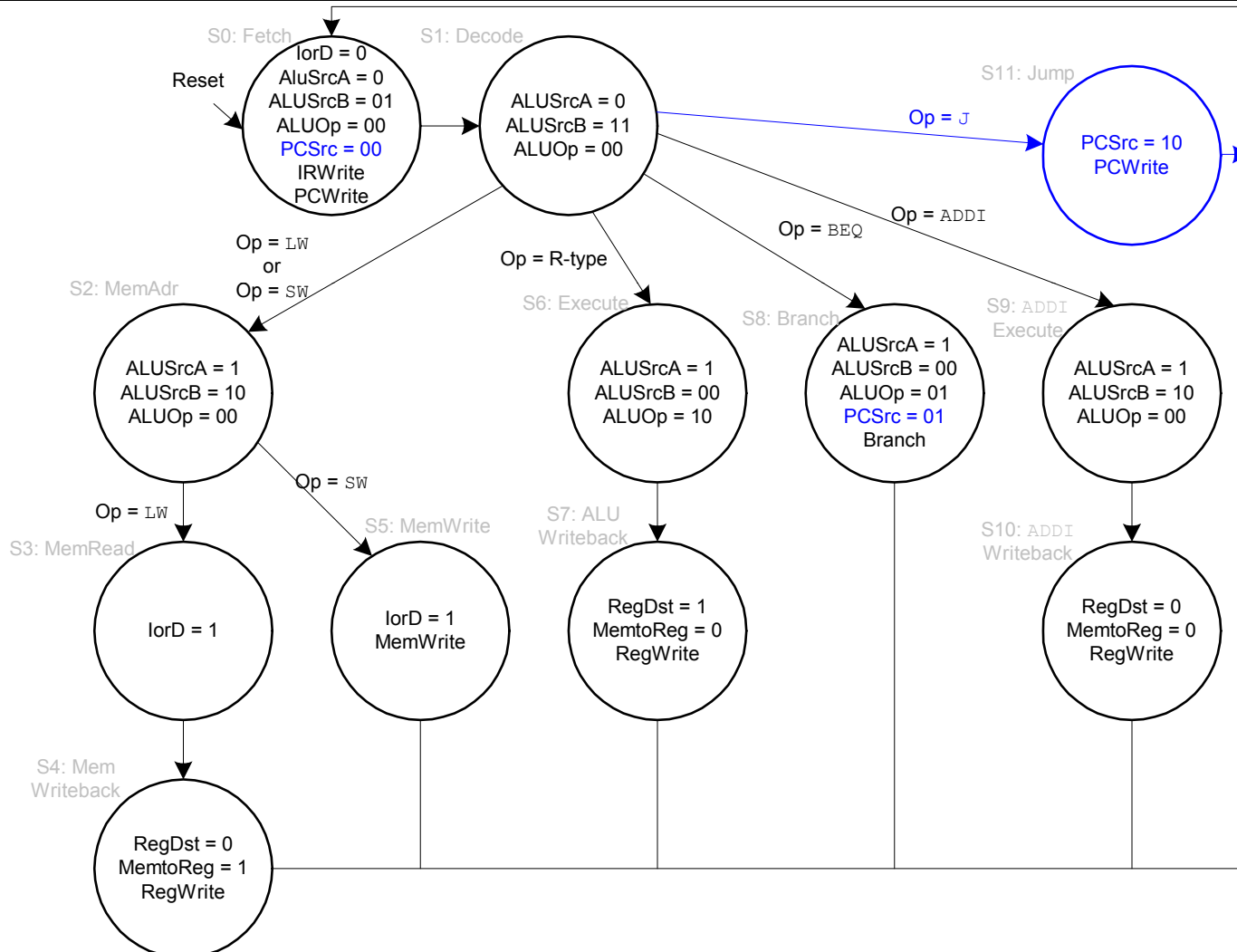
Erweiterung des Datenpfads für j



Erweiterung des Hauptsteuerwerks um j



Erweiterung des Hauptsteuerwerks um j



Rechenleistung des Mehrtaktprozessors



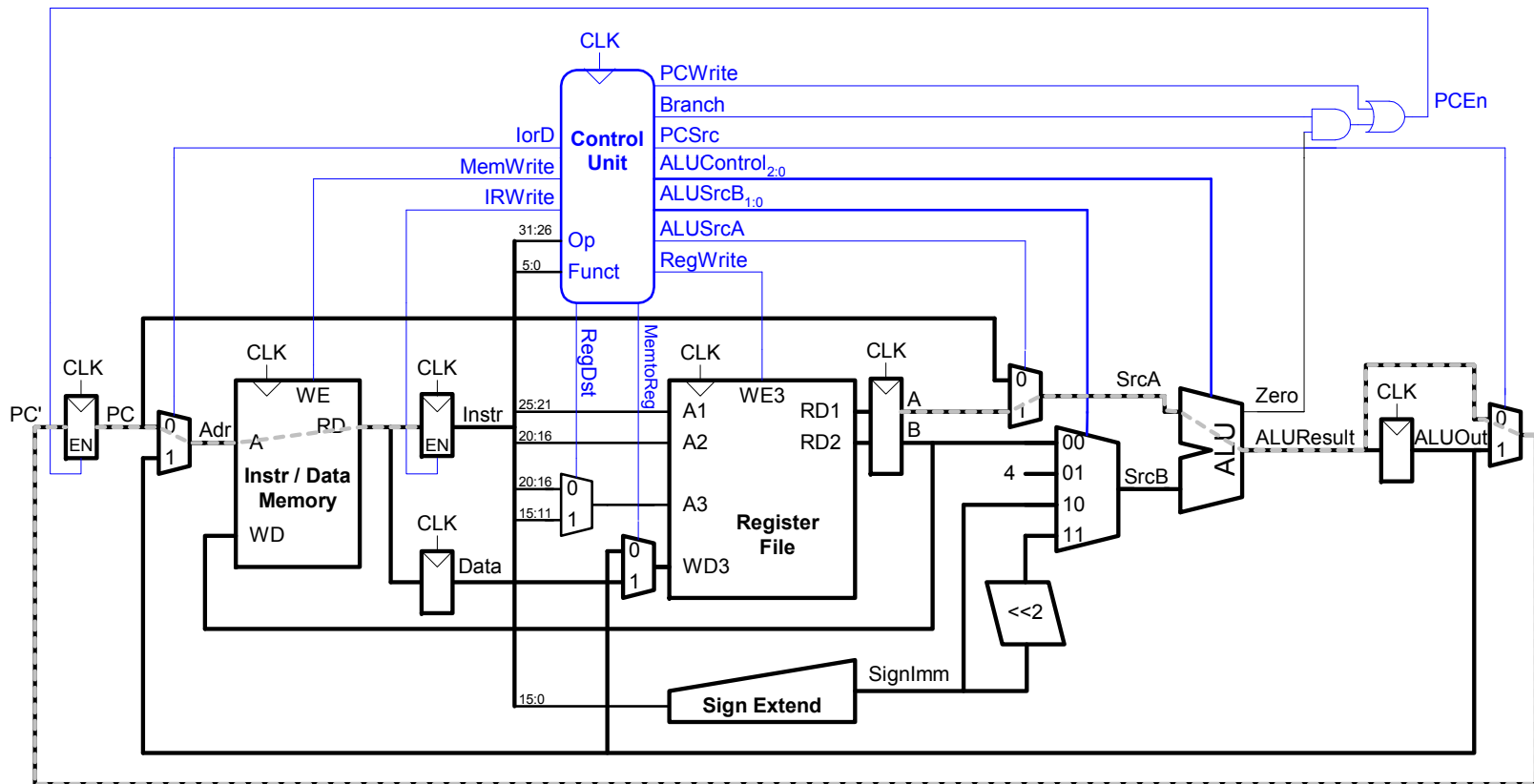
- Instruktionen benötigen unterschiedliche viele Takte:
 - 3 Takte : `beq, j`
 - 4 Takte : R-Typ, `sw, addi`
 - 5 Takte : `lw`
- CPI wird bestimmt als gewichteter Durchschnitt
- SPECint 2000 Benchmark:
 - 25% Laden
 - 10% Speichern
 - 11% Verzweigungen
 - 2% Sprünge
 - 52% R-Typ

Durchschnittliche CPI = $(0,11 + 0,02)(3) + (0,52 + 0,10)(4) + (0,25)(5) = 4,12$

Rechenleistung des Mehrtaktprozessors

- Kritischer Pfad:

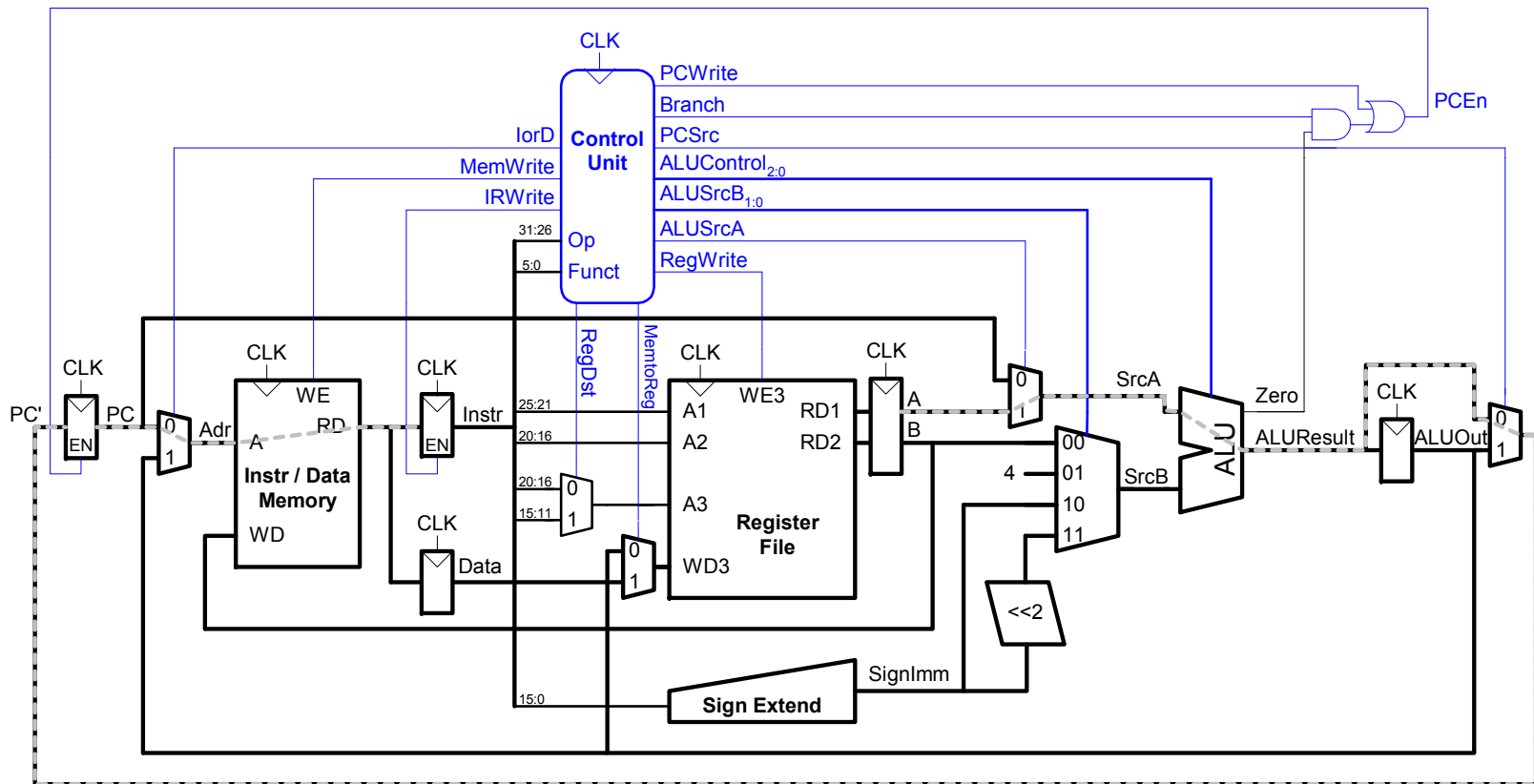
$$T_c =$$



Rechenleistung des Mehrtaktprozessors

- Kritischer Pfad :

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$



Beispiel: Rechenleistung Mehrtaktprozessor



Element	Parameter	Verzögerung (ps)
Register Clock-to-Q	t_{pcq}	30
Register Setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Speicher Lesen	t_{mem}	250
Registerfeld Lesen	t_{RFread}	150
Registerfeld Setup	$t_{RFsetup}$	20

$$T_c =$$

Beispiel: Rechenleistung Mehrtaktprozessor



Element	Parameter	Verzögerung (ps)
Register Clock-to-Q	t_{pcq}	30
Register Setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Speicher Lesen	t_{mem}	250
Registerfeld Lesen	t_{RFread}	150
Registerfeld Setup	$t_{RFsetup}$	20

$$\begin{aligned}T_c &= t_{pcq_PC} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup} \\ &= t_{pcq_PC} + t_{mux} + t_{mem} + t_{setup} \\ &= [30 + 25 + 250 + 20] \text{ ps} \\ &= 325 \text{ ps}\end{aligned}$$

Beispiel: Rechenleistung Mehrtaktprozessor



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Führe Programm mit 100 Milliarden Instruktionen auf Mehrtaktprozessor aus
 - $CPI = 4,12$
 - $T_c = 325 \text{ ps}$

Ausführungszeit =

Beispiel: Rechenleistung Mehrtaktprozessor



- Führe Programm mit 100 Milliarden Instruktionen auf Mehrtaktprozessor aus
 - $CPI = 4,12$
 - $T_c = 325 \text{ ps}$

$$\begin{aligned} \text{Ausführungszeit} &= (\# \text{ Instruktionen}) \times CPI \times T_c \\ &= (100 \times 10^9) (4,12) (325 \times 10^{-12}) \\ &= 133,9 \text{ Sekunden} \end{aligned}$$

- **Langsamer** als Ein-Takt-Prozessor (brauchte 92,5 Sekunden).
 - Warum?

Beispiel: Rechenleistung Mehrtaktprozessor

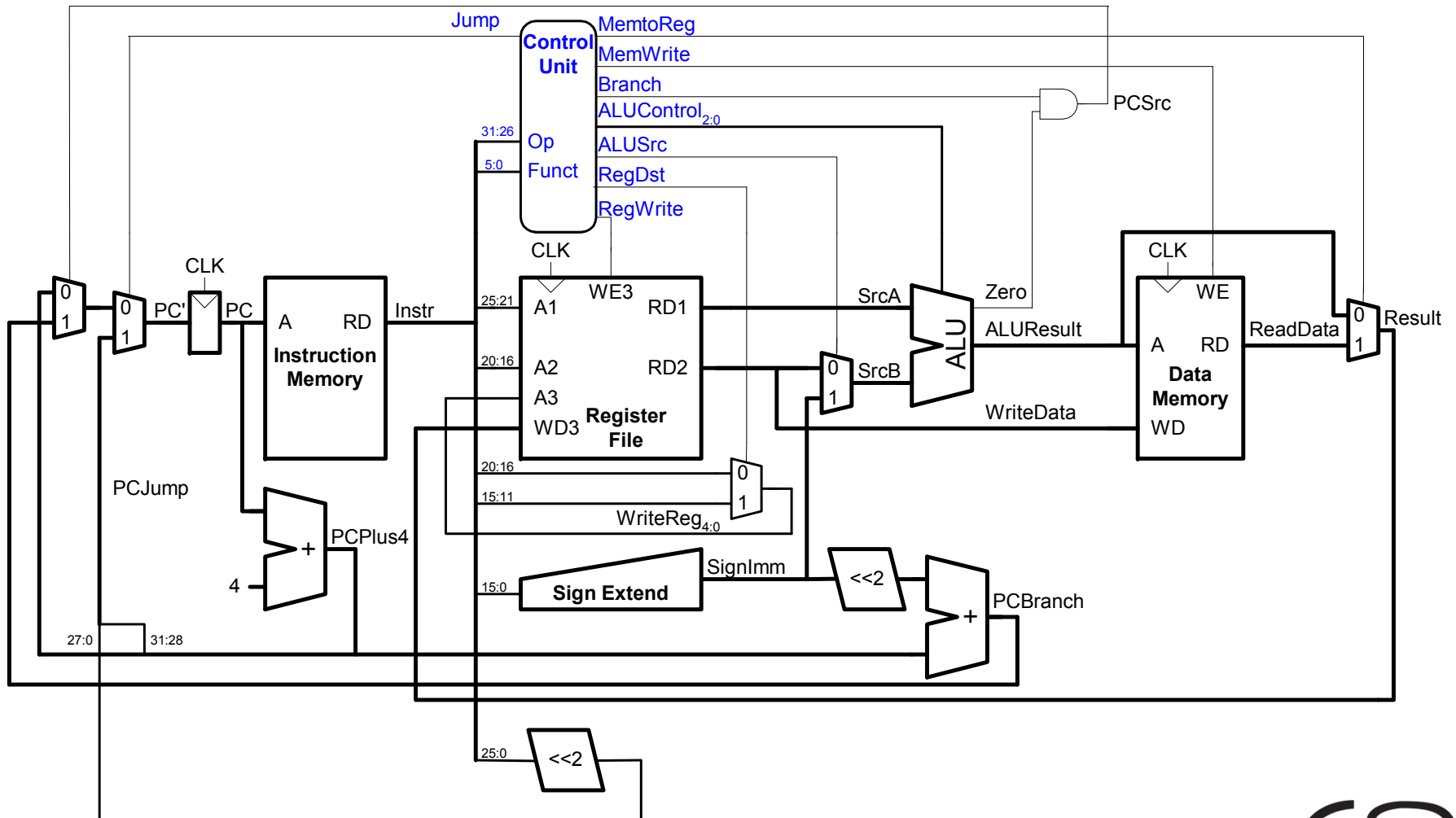


- Führe Programm mit 100 Milliarden Instruktionen auf Mehrtaktprozessor aus
 - $CPI = 4,12$
 - $T_c = 325 \text{ ps}$

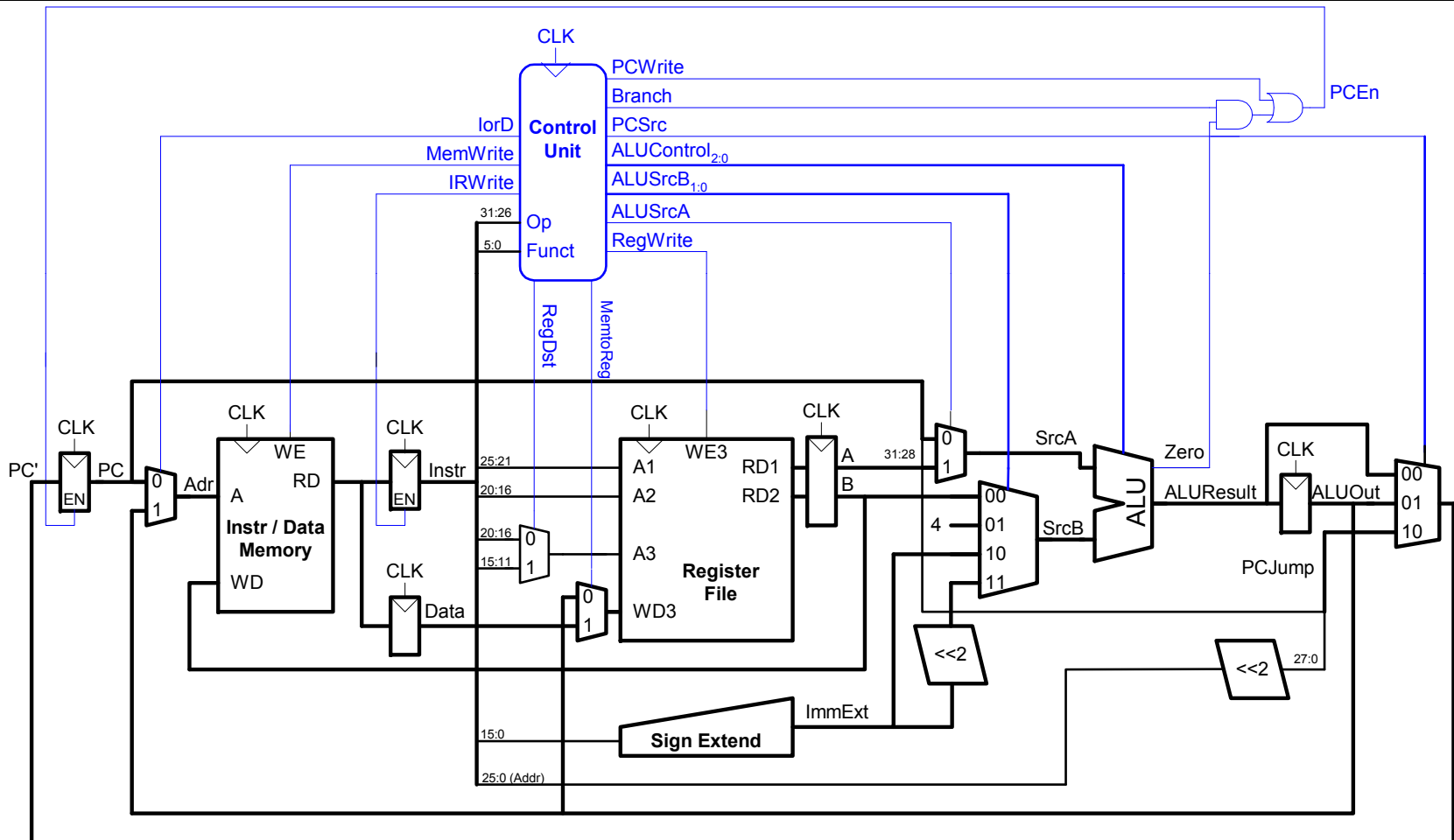
$$\begin{aligned} \text{Ausführungszeit} &= (\# \text{ Instruktionen}) \times CPI \times T_c \\ &= (100 \times 10^9) (4,12) (325 \times 10^{-12}) \\ &= 133,9 \text{ Sekunden} \end{aligned}$$

- **Langsamer** als Ein-Takt-Prozessor (brauchte 92,5 Sekunden).
 - Unterschiedlich lange Anzahl von Ausführungstakten (bis zu 5 für lw)
 - Aber nicht 5x schnellere Taktfrequenz
 - Nun zusätzliche Verzögerungen für sequentielle Logik mehrfach je Befehl
 - $t_{pcq} + t_{setup} = 50 \text{ ps}$

Rückblick: Ein-Takt MIPS Prozessor



Rückblick: Mehrtakt-MIPS-Prozessor



Potentiell etwas kleiner.

MIPS Prozessor mit Pipelining



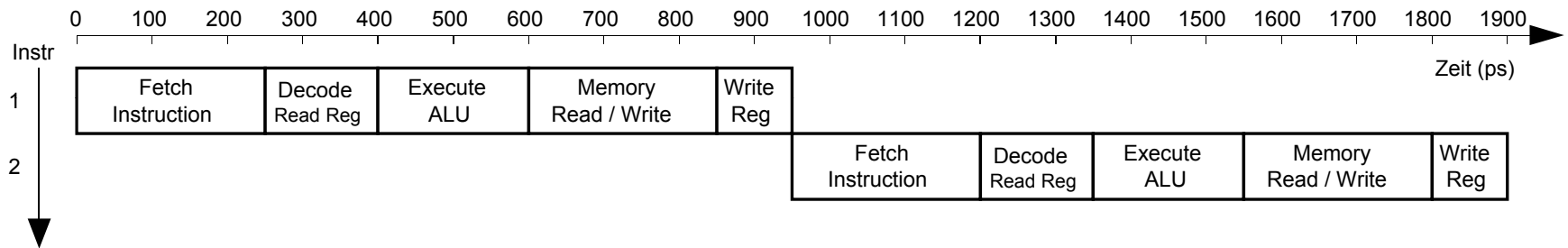
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Zeitliche** Parallelität
- **Teile** Ablauf im Ein-Takt-Prozessor in fünf Stufen:
 - Hole Instruktion (*Fetch*)
 - Dekodiere Bedeutung von Instruktion (*Decode*)
 - Führe Instruktion aus (*Execute*)
 - Greife auf Speicher zu (*Memory*)
 - Schreibe Ergebnisse zurück (*Writeback*)
- Füge **Pipeline-Register** zwischen den Stufen ein

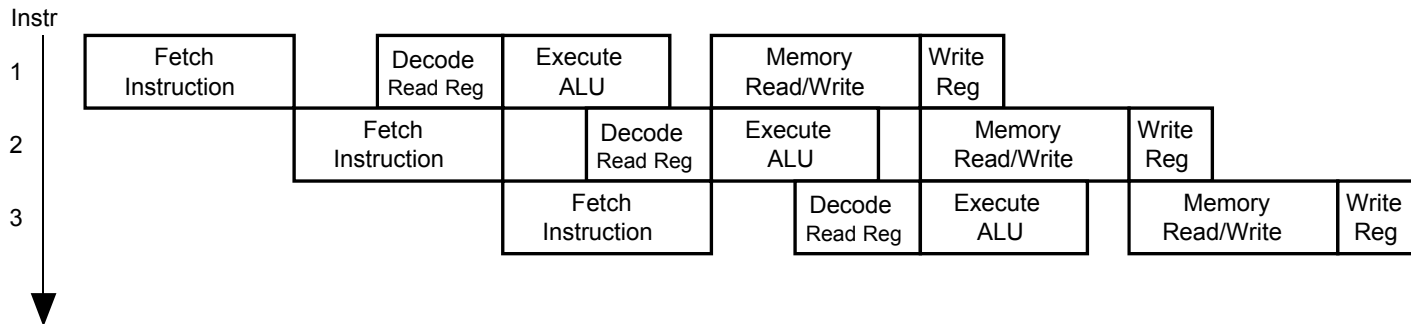
Rechenleistung: Ein-Takt und Pipelined



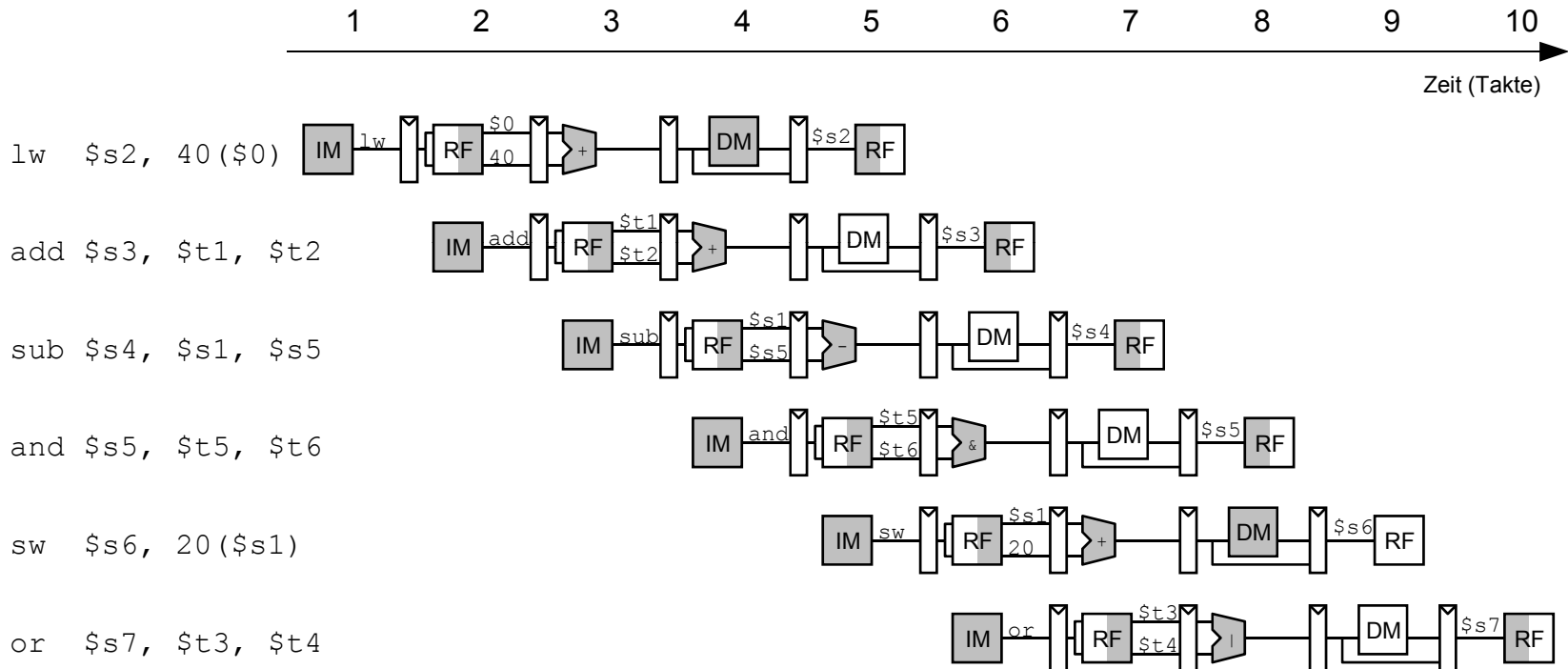
Ein-Takt



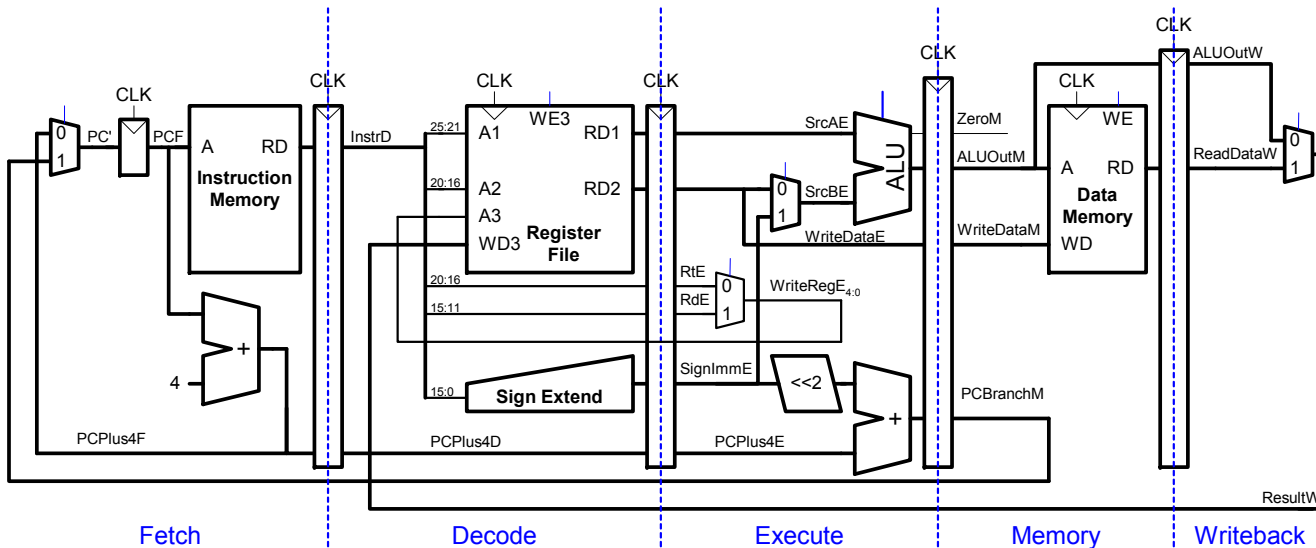
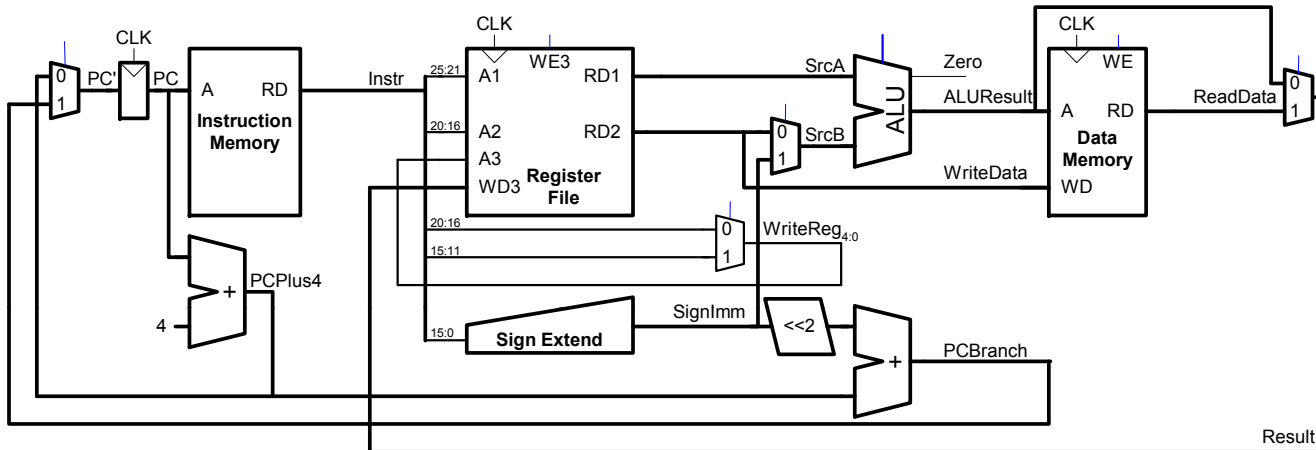
Pipelined



Abstraktere Darstellung des Pipelinings

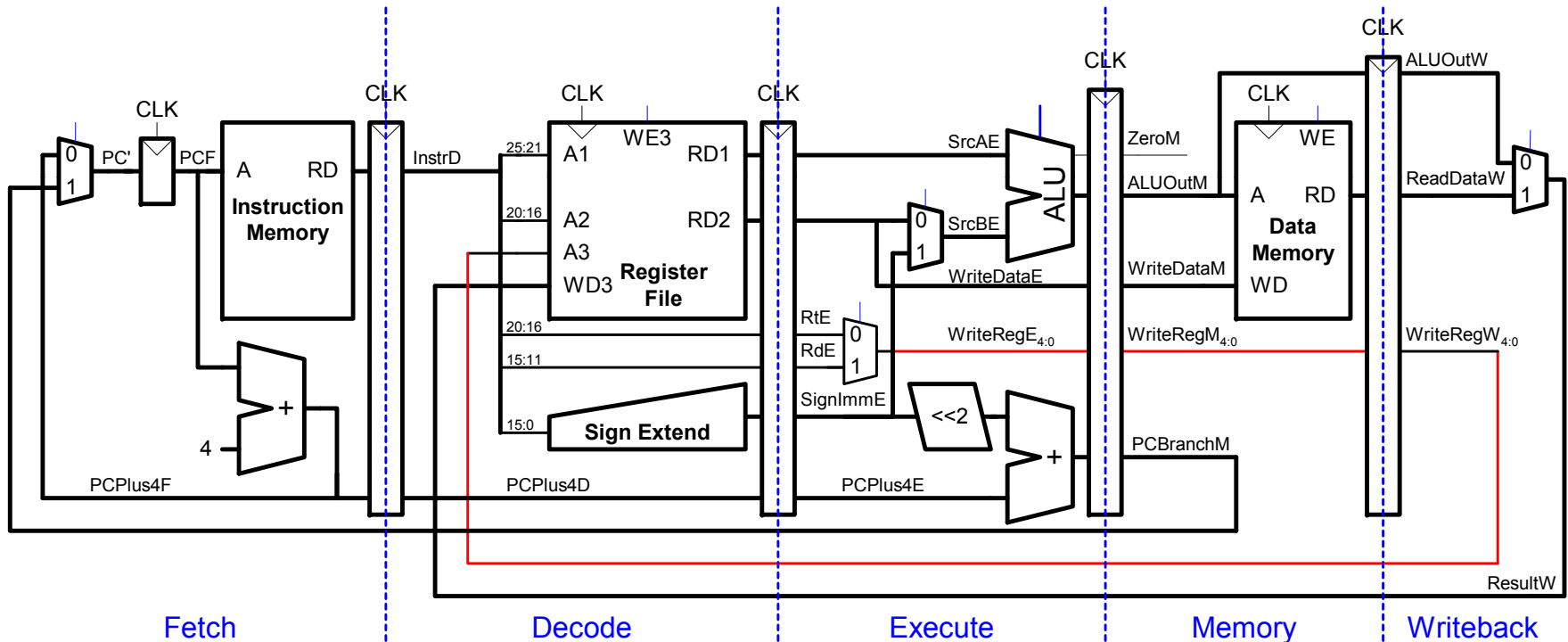


Ein-Takt- und Pipelined-Datenpfad

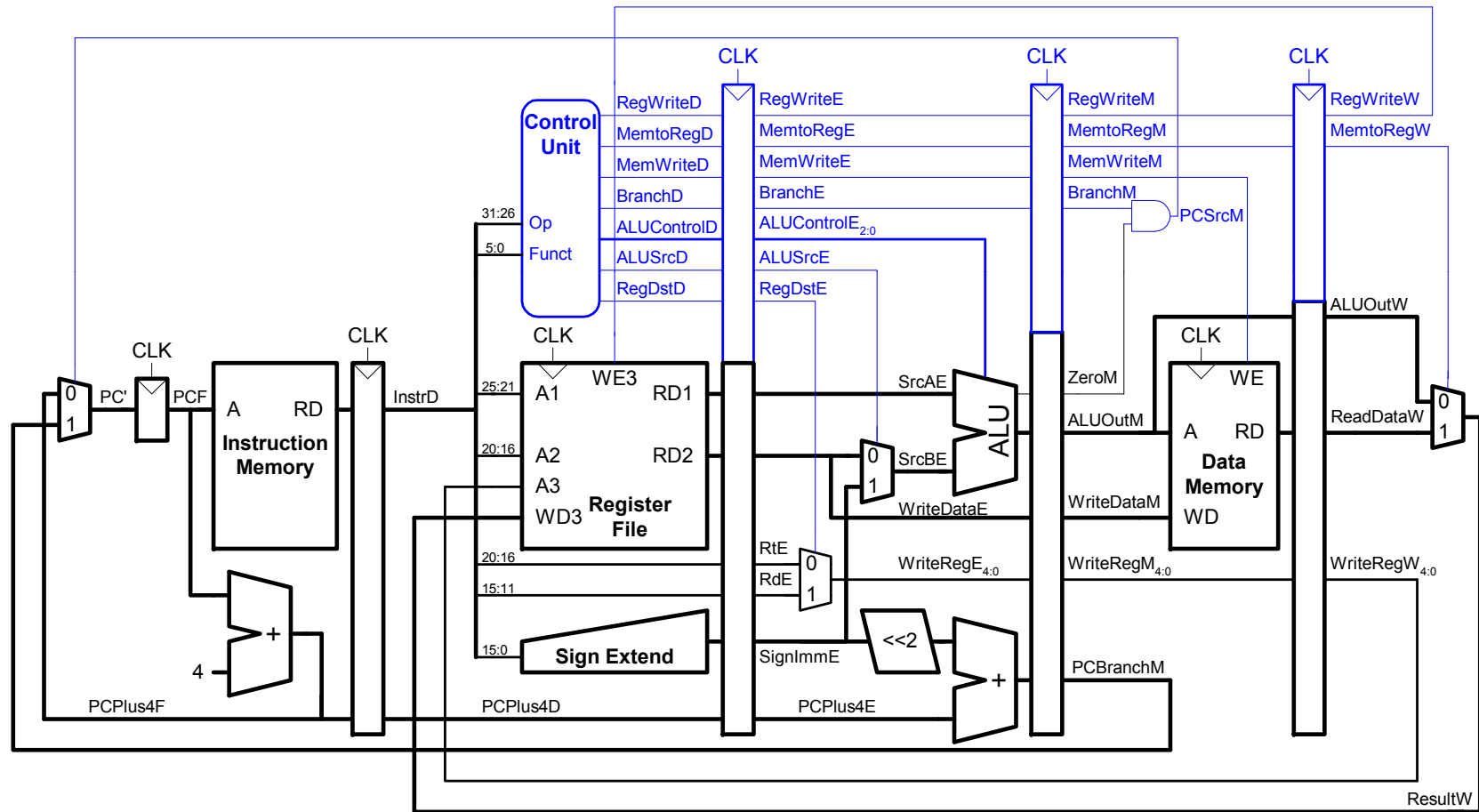


Korrigierter Pipelined-Datenpfad

- WriteReg muss zur gleichen Zeit am Registerfeld ankommen wie Result



Steuersignale für Pipelined-Datenpfad



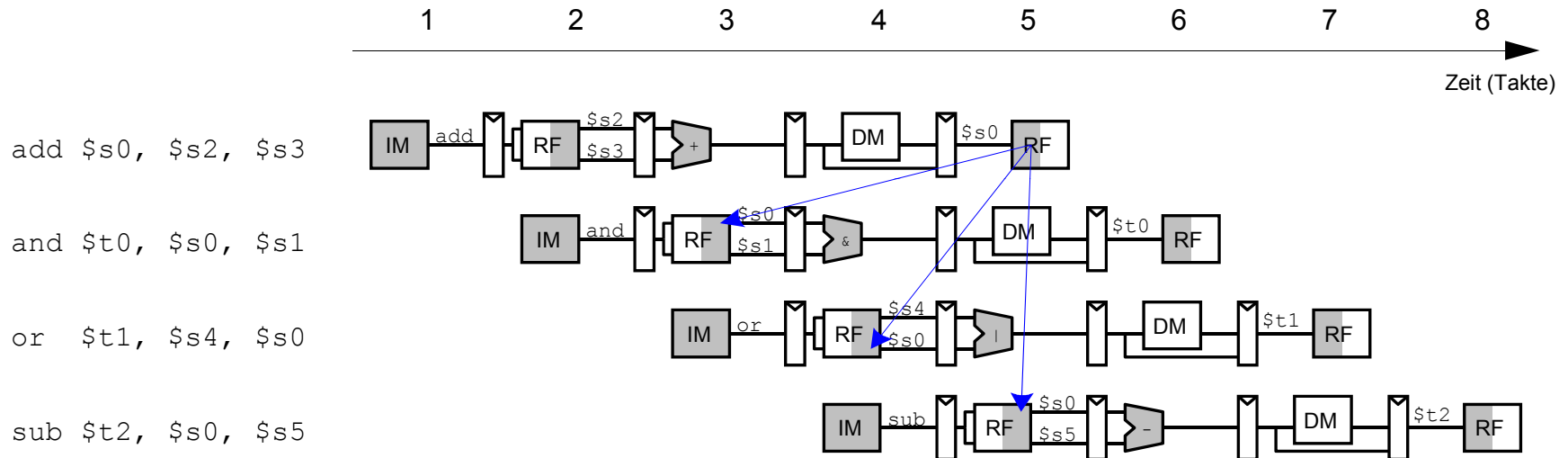
Identisch zu Ein-Takt-Steuerwerk, aber Signale verzögert über Pipeline-Stufen

Abhängigkeiten zwischen Pipeline-Stufen (*hazards*)

- Treten auf wenn eine
 - Instruktion vom Ergebnis einer vorhergehenden abhängt
 - ... diese aber noch kein Ergebnis geliefert hat

- Arten von Hazards
 - **Data Hazard:** z.B. Neuer Wert von Register noch nicht in Registerfeld eingetragen
 - **Control Hazard:** Unklar welche Instruktion als nächstes ausgeführt werden muss
 - Tritt bei Verzweigungen auf

Data Hazard



Hier: Read-after-Write Hazard (RAW)
- \$s0 „muss vor Lesen geschrieben werden“

Umgang mit Data Hazards

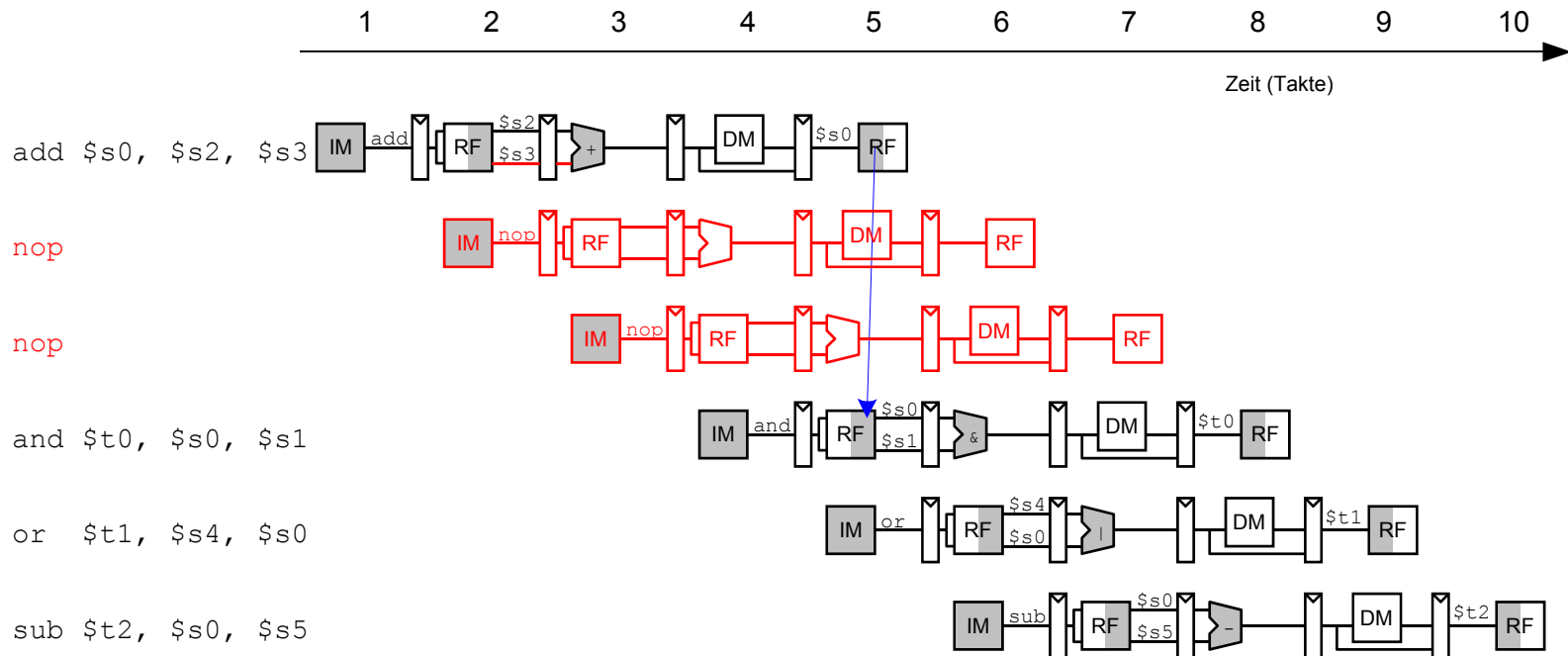


- Plane **Wartezeiten** von Anfang an ein
 - Füge `nops` zur Compile-Zeit ein
 - *scheduling*
- **Stelle** Maschinencode zur Compile-Zeit **um**
 - *scheduling / reordering*
- Leite Daten zur Laufzeit schneller über **Abkürzungen** weiter
 - *bypassing / forwarding*
- **Halte** Prozessor zur Laufzeit **an** bis Daten da sind
 - *stalling*

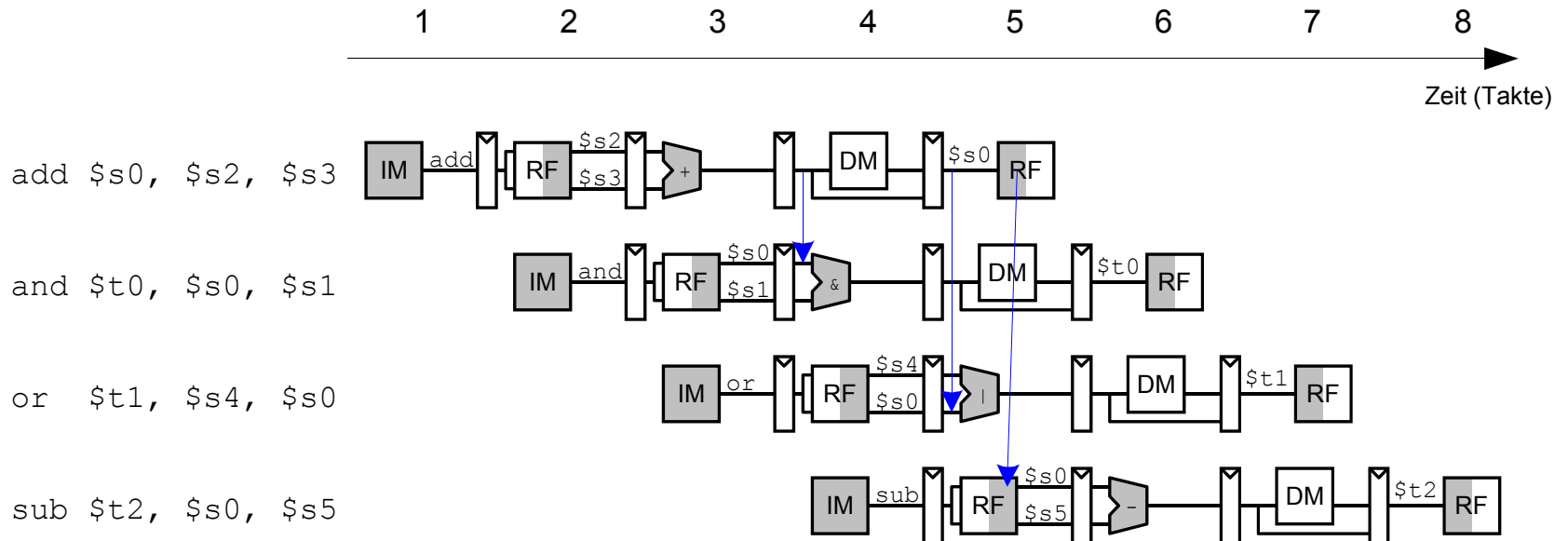
Beseitigung von Data Hazards zur Compile-Zeit



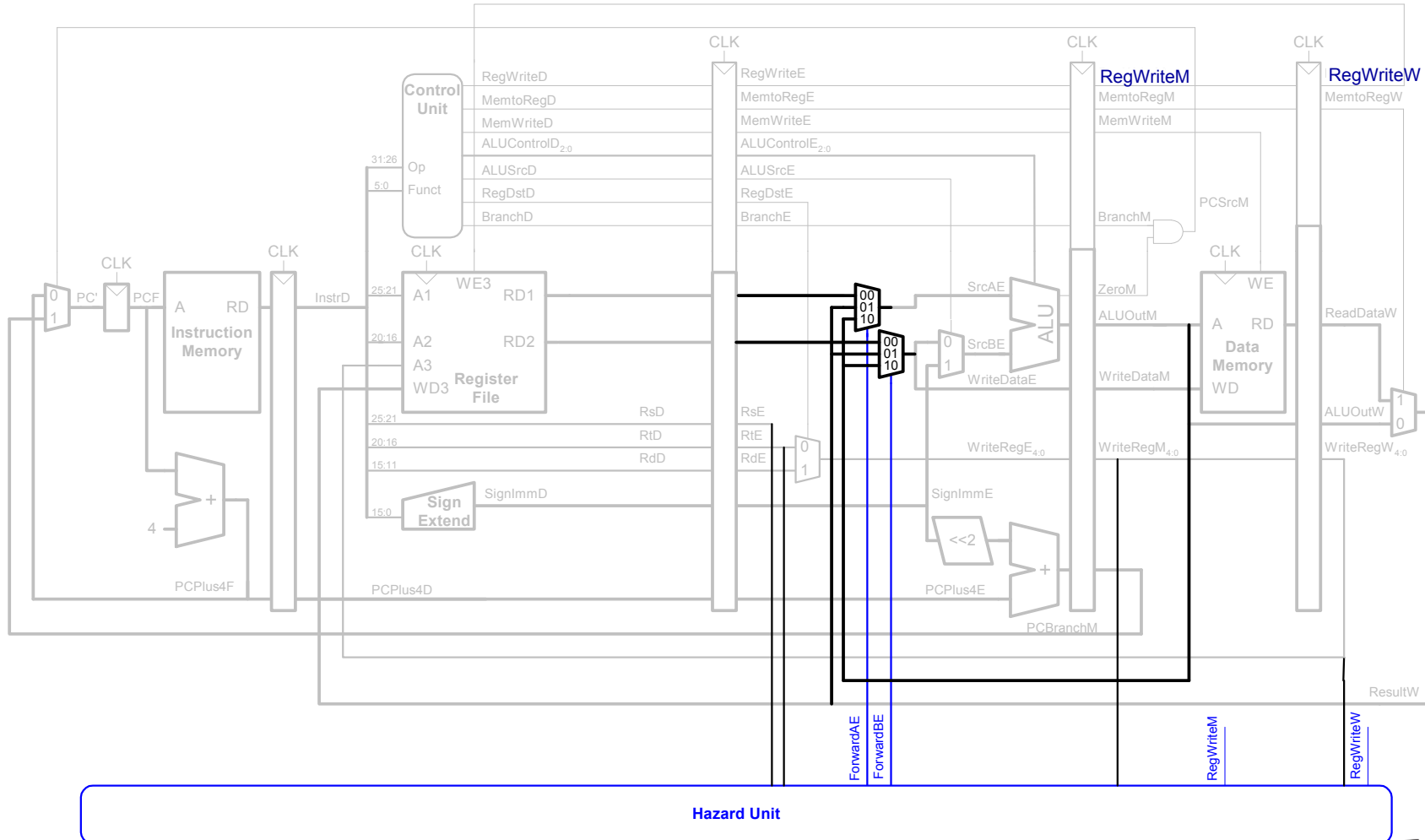
- Füge ausreichend viele `nops` ein bis Ergebnis bereitsteht
- Oder schiebe unabhängige Instruktionen nach vorne (statt `nops`)



Data Forwarding: "Abkürzungen" einbauen



Data Forwarding: "Abkürzungen" einbauen



Data Forwarding: “Abkürzungen” einbauen

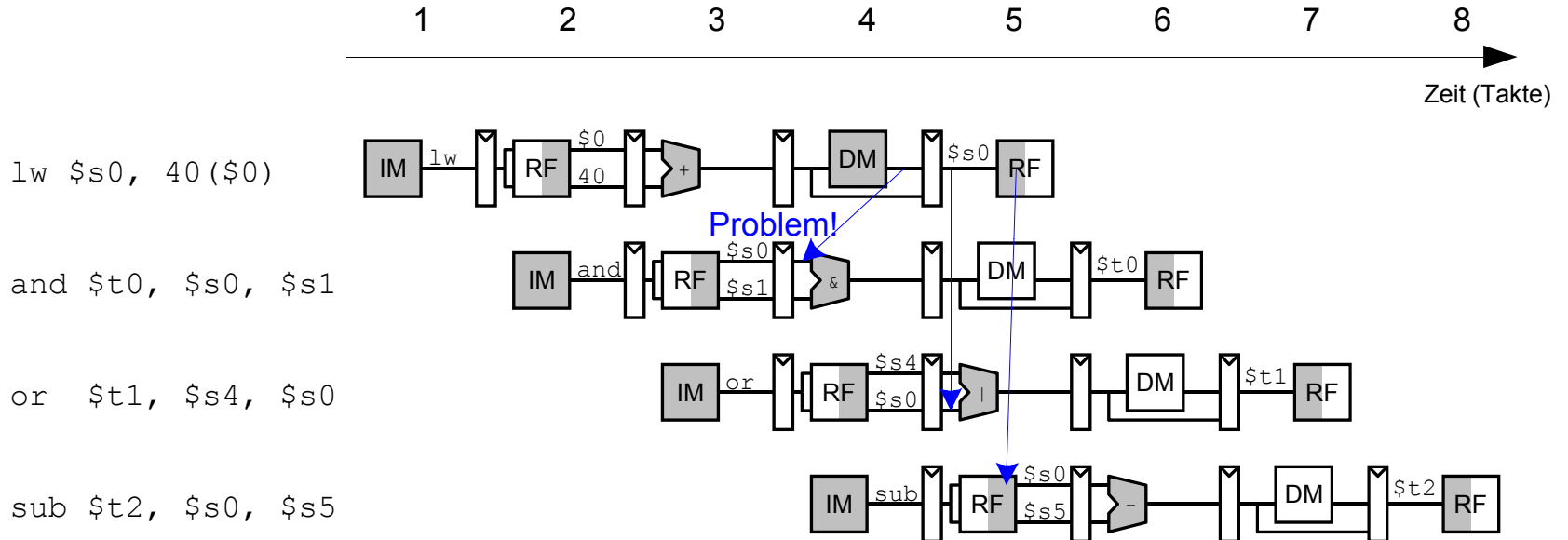


- “Abkürzung” zur Execute-Stufe von
 - Memory-Stufe oder
 - Writeback-Stufe
- Forwarding-Logik für Signal *ForwardAE* (Weiterleiten von Operand A):

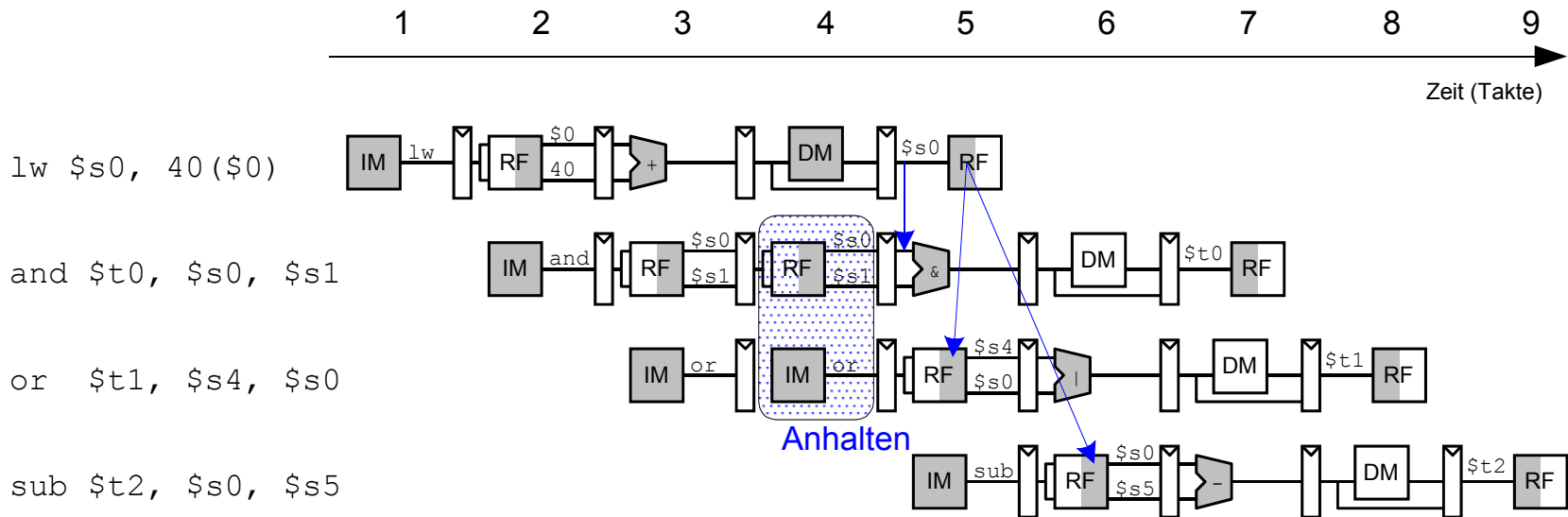
```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
    ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
    ForwardAE = 01
else
    ForwardAE = 00
```

- Forwarding-Logik für Signal *ForwardBE* (Weiterleiten von Operand B) analog
 - Ersetze *rsE* durch *rtE*

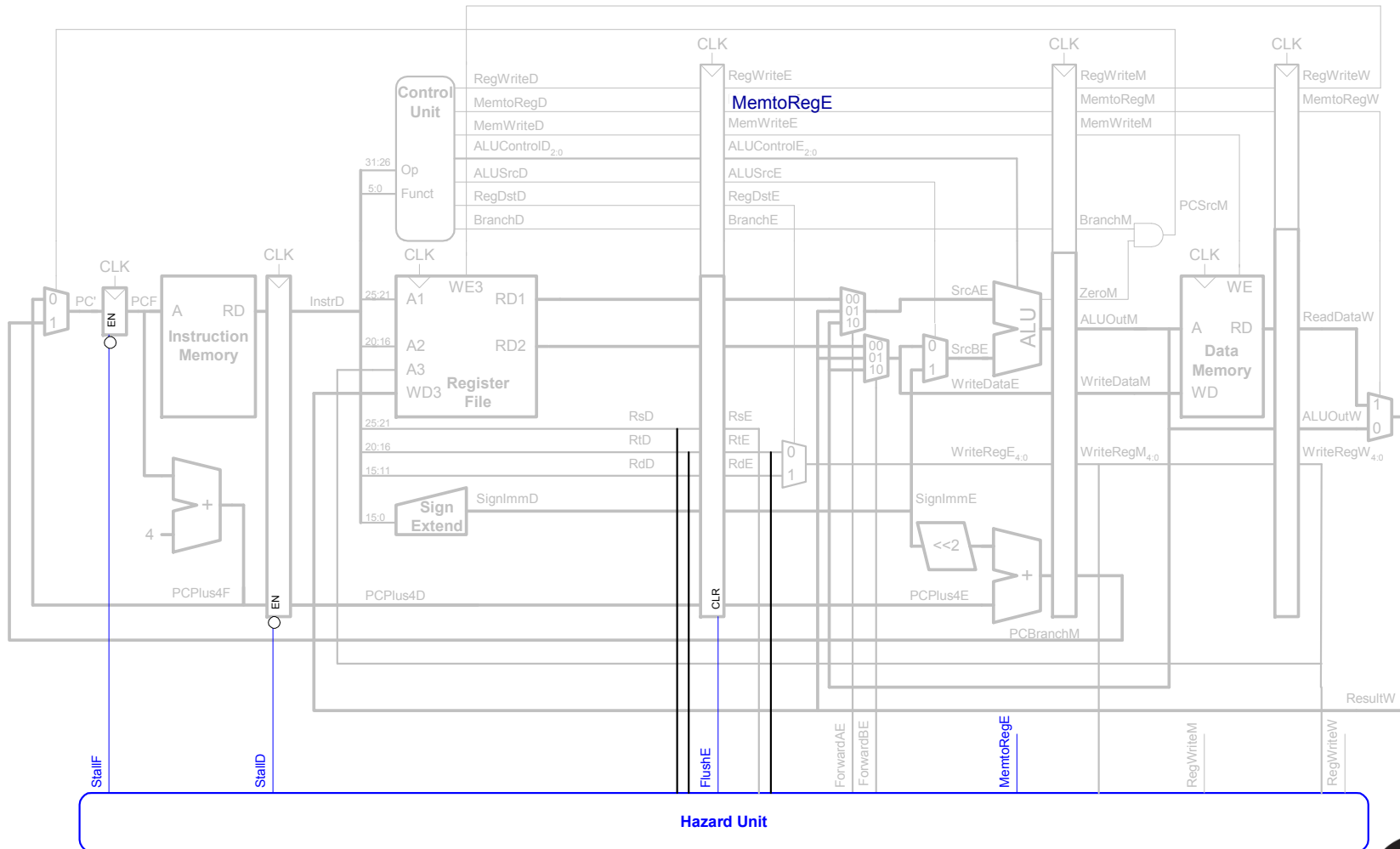
Anhalten des Prozessors (*stalling*)



Anhalten des Prozessors (*stalling*)



Erweiterung der Hazard-Einheit für Stalling



Behandlung von Stalling in Hazard-Einheit



- Stalling-Logik:

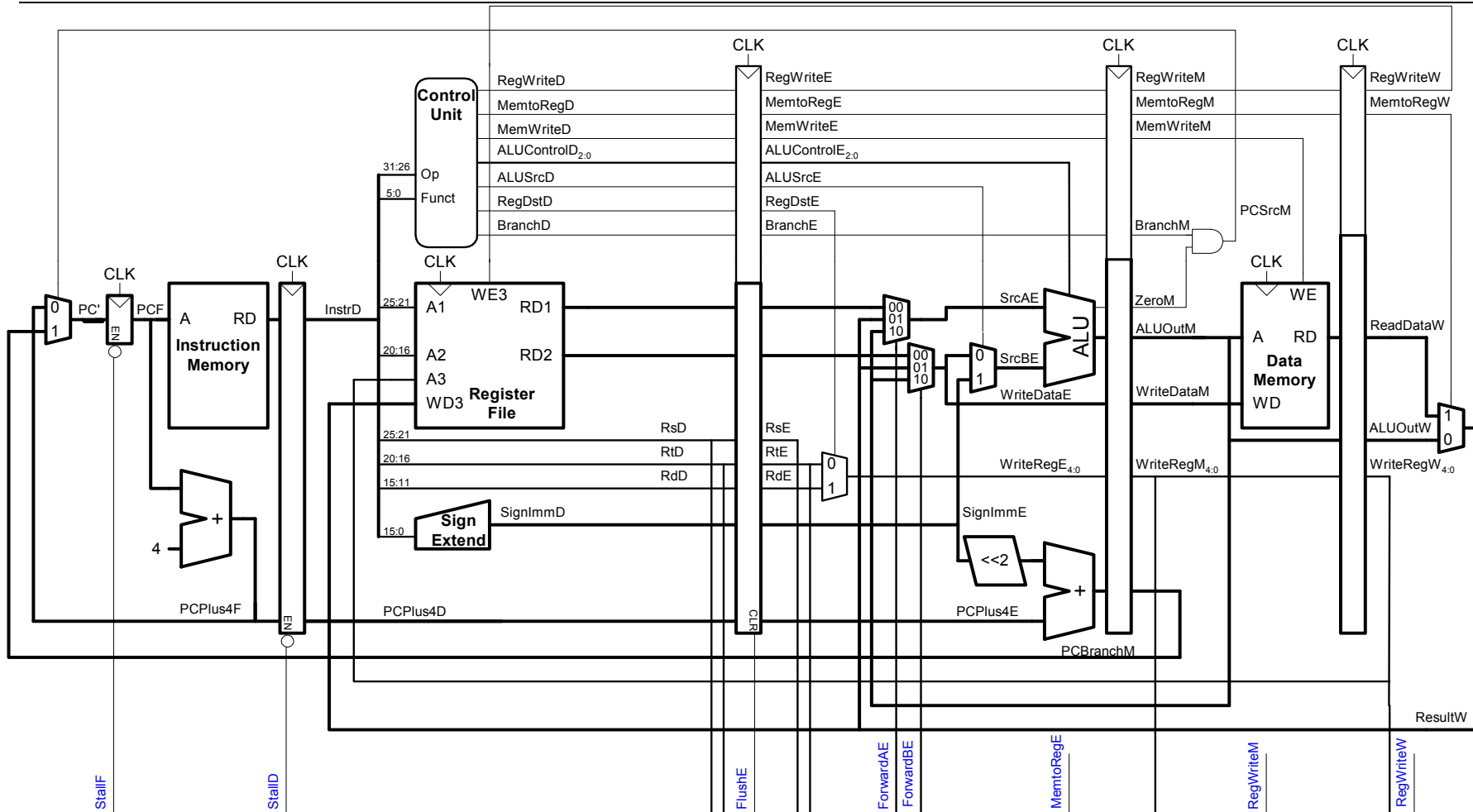
lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE

StallF = StallD = FlushE = lwstall

Control Hazards

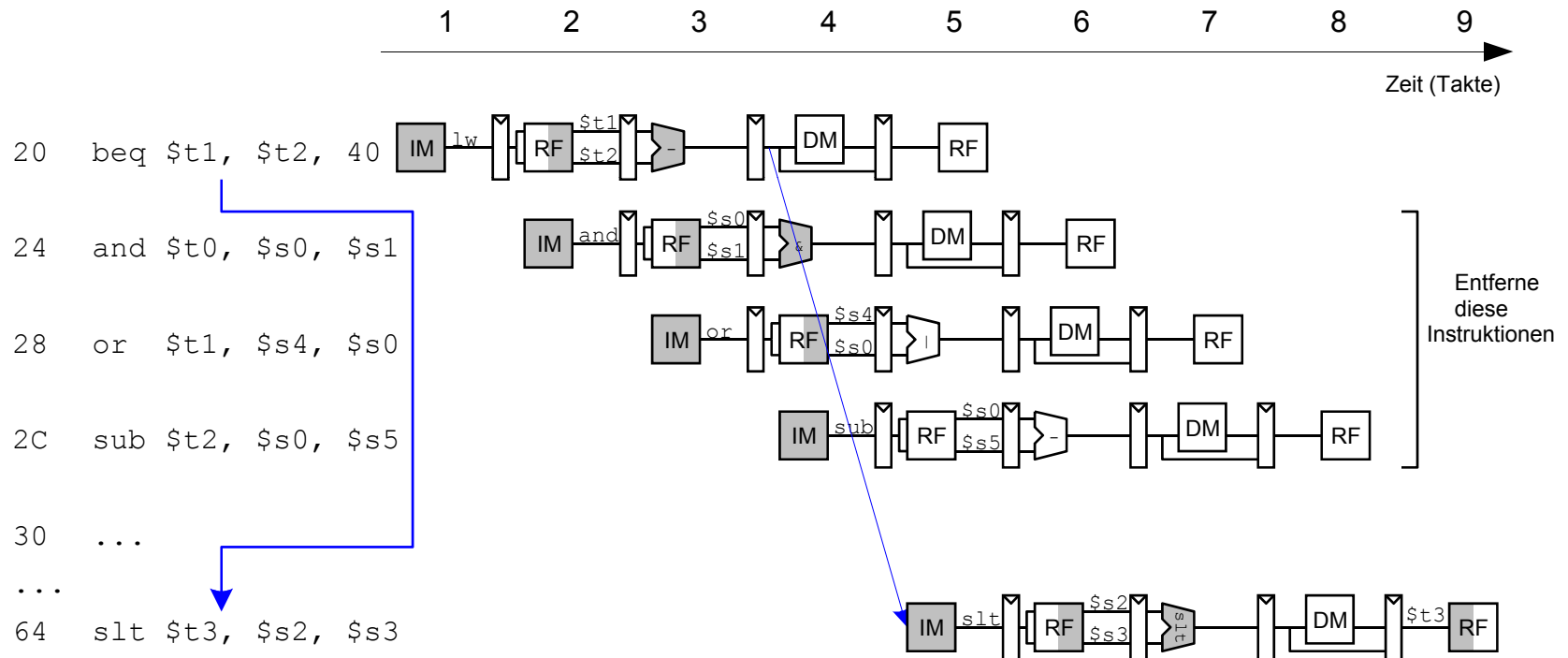
- `beq`:
 - **Entscheidung** zu Springen wird erst in vierter Stufe der Pipeline (M) getroffen
 - Neue Instruktionen werden aber bereits **geholt**
 - Im einfachsten Fall: Von PC+4, +8, +12, ...
 - Falls zu springen ist, müssen diese Instruktionen aus der Pipeline **entfernt** werden
 - ... das Programm wäre ja **woanders** (am Sprungziel) weitergegangen
 - “Spülen” (*flush*)
- Kosten eines solchen **falsch vorhergesagten** Sprunges:
 - Anzahl von zu entfernenden Instruktion falls Sprung genommen
- Könnte reduziert werden, wenn Sprung in **früherer** Pipeline-Stufe entschieden würde

Control Hazards: Ursprüngliche Pipeline

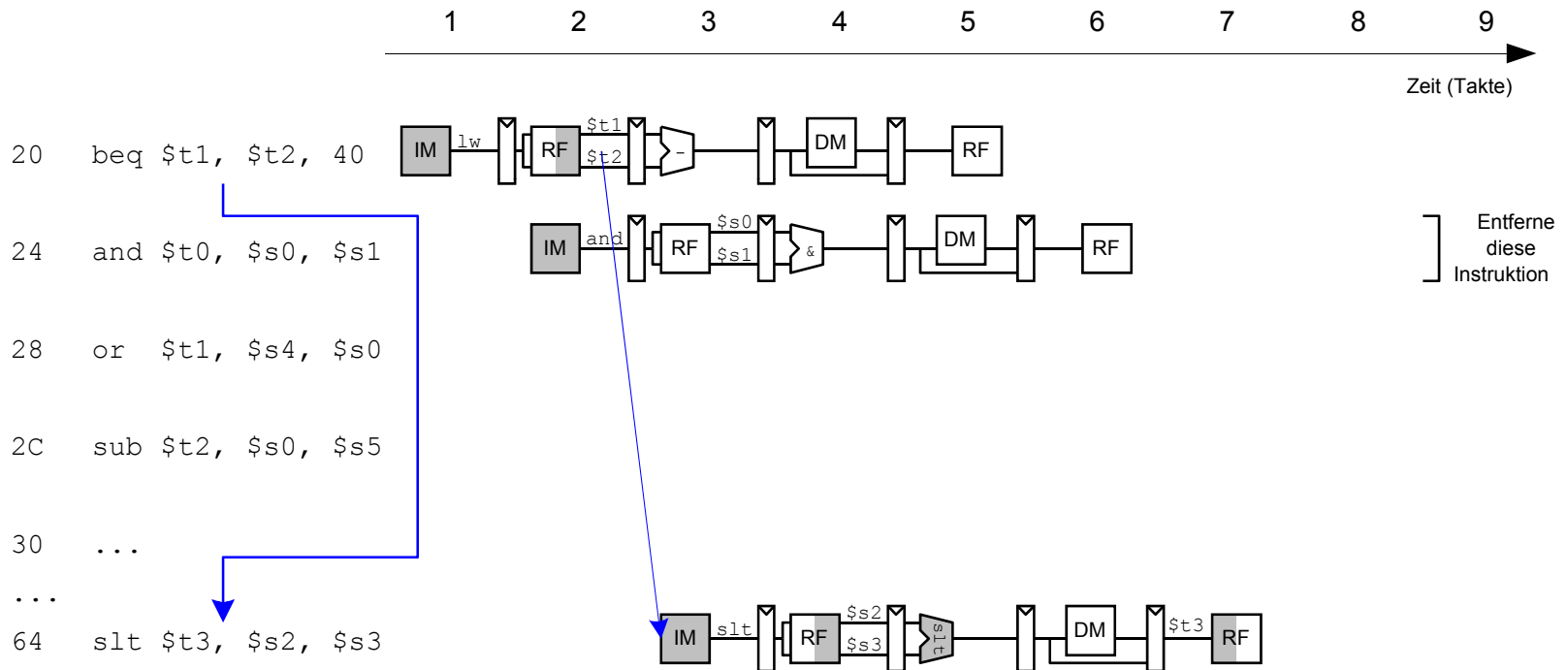


Hazard Unit

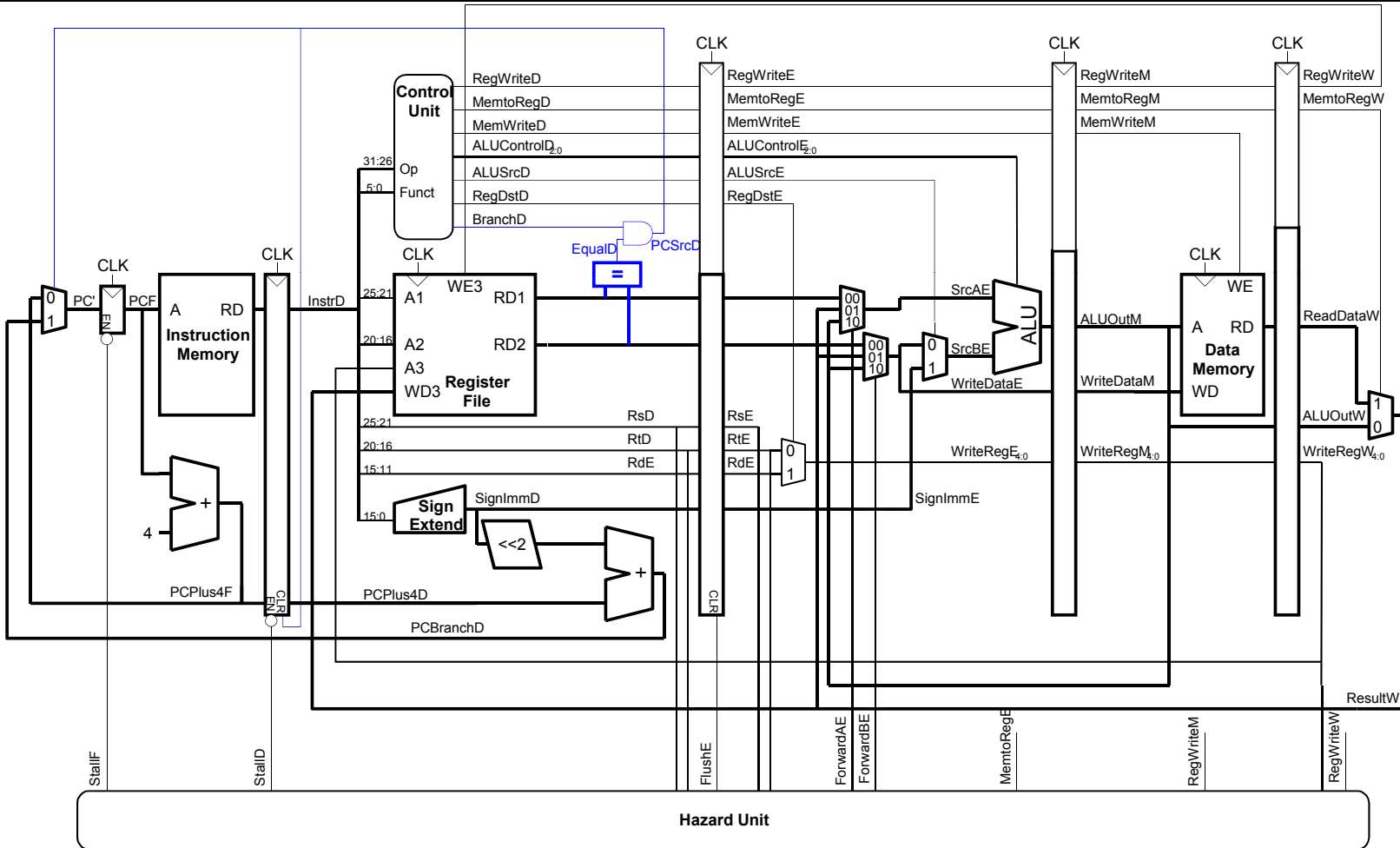
Beispiel: Control Hazards



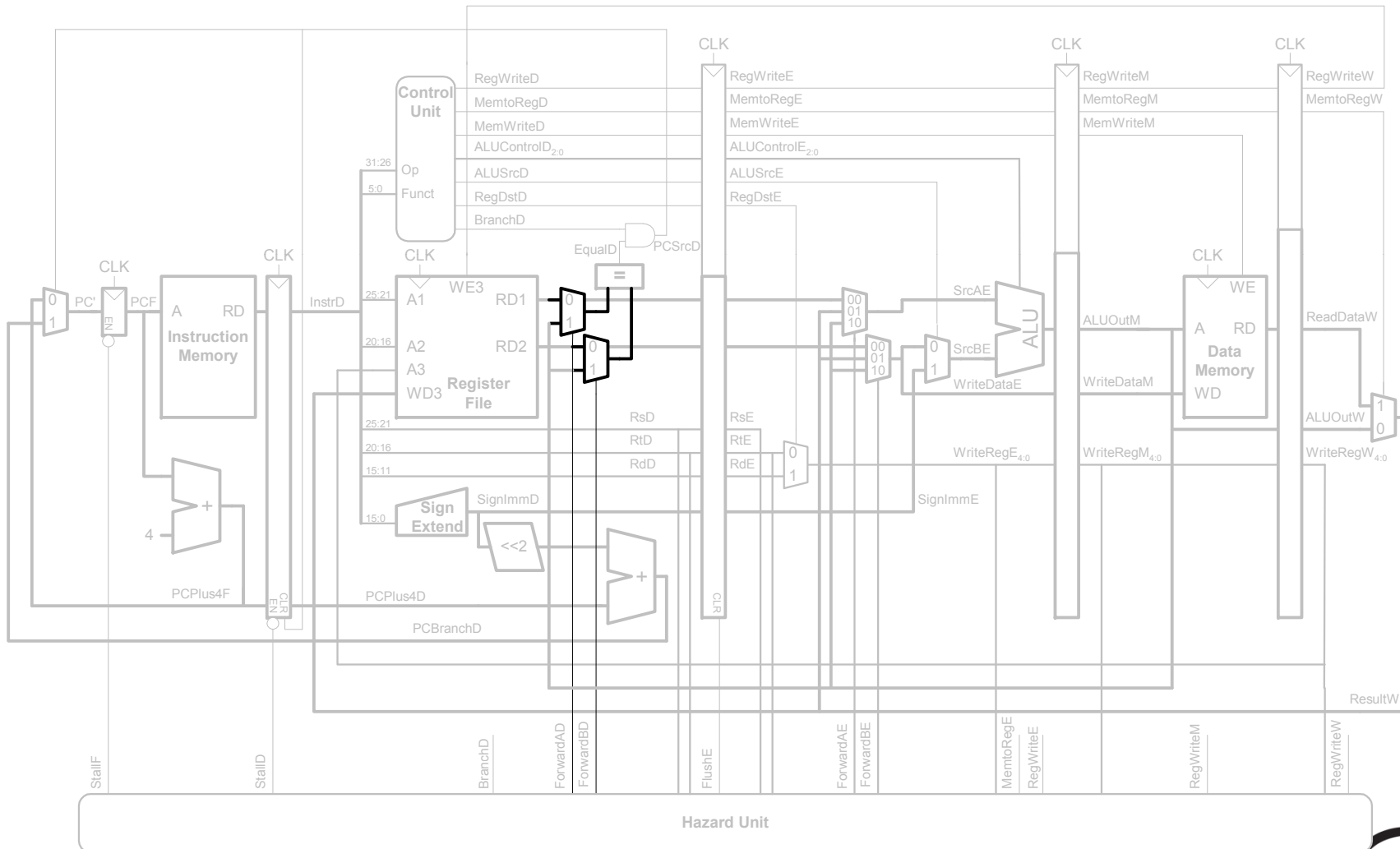
Auflösen von Control Hazards durch frühere Sprungentscheidung



Control Hazards: Ansatz "Frühere Sprungentscheidung"



Berücksichtige neue Data Hazards



Frühe Sprungentscheidung: Benötigte Logik für Forwarding und Stalling



▪ Forwarding-Logik:

```
ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND RegWriteM  
ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND RegWriteM
```

▪ Stalling-Logik:

```
branchstall = BranchD AND RegWriteE AND  
              (WriteRegE == rsD OR WriteRegE == rtD)  
OR  
              BranchD AND MemtoRegM AND  
              (WriteRegM == rsD OR WriteRegM == rtD)
```

```
StallF = StallD = FlushE = lwstall OR branchstall
```

Orthogonaler Ansatz: Sprungvorhersage



- Versuche **vorherzusagen**, ob ein Sprung genommen wird
 - Dann können Instruktionen von der **richtigen** Stelle geholt werden
 - **Rückwärtssprünge** werden üblicherweise genommen (Schleifen!)
 - Genauer: Für jeden Sprung **Historie** führen, ob er die letzten Male genommen wurde
 - ... dann wird jetzt vermutlich auch wieder genommen
- Eine gute Vorhersage **reduziert** die Zahl der Sprünge, die ein Flush der Pipeline erforderlich machen

Beispiel: Rechenleistung des Pipelined-Prozessors



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Idealerweise wäre $CPI = 1$
- Manchmal treten aber **Stalls** auf (wegen Lade- und Verzweigungsbefehlen)
- SPECint 2000 benchmark:
 - 25% loads
 - 10% stores
 - 11% branches
 - 2% jumps
 - 52% R-type
- Annahmen:
 - 40% der geladenen Daten werden gleich in der **nächsten** Instruktion gebraucht
 - 25% aller Verzweigungen werden **falsch** vorhergesagt
 - Alle Sprünge erzeugen eine zu entfernende (*flush*) Instruktion
- **Wie hoch ist der durchschnittliche CPI-Wert?**

Beispiel: Rechenleistung des Pipelined-Prozessors



- SPECint 2000 benchmark:
 - 25% loads
 - 10% stores
 - 11% branches
 - 2% jumps
 - 52% R-type
- Annahmen:
 - 40% der geladenen Daten werden gleich in der **nächsten** Instruktion gebraucht
 - 25% aller Verzweigungen werden **falsch** vorhergesagt
 - Alle Sprünge erzeugen eine zu entfernende (*flush*) Instruktion
- **Wie hoch ist der durchschnittliche CPI-Wert?**
 - Lade/Verzweigungsinstruktionen haben CPI = 1 ohne Stall, = 2 mit Stall. Daher:
 - $CPI_{lw} = 1 (0,6) + 2 (0,4) = 1,4$
 - $CPI_{beq} = 1 (0,75) + 2 (0,25) = 1,25$
 - Also:

$$\text{Durchschnittliche CPI} = (0,25) (1,4) + (0,1) (1,0) + (0,11)(1,25) + (0,02) (2,0) + (0,52)(1,0) \\ = 1,15$$

Beispiel: Rechenleistung des Pipelined-Prozessors



- Kritischer Pfad des Pipelined-Prozessors:

$$T_c = \max \left\{ \begin{array}{l} t_{pcq} + t_{mem} + t_{setup}, \\ 2 (t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}), \\ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup}, \\ t_{pcq} + t_{memwrite} + t_{setup}, \\ 2 (t_{pcq} + t_{mux} + t_{RFwrite}) \end{array} \right\}$$

Fetch
Decode
Execute
Memory
Writeback

Beispiel: Rechenleistung des Pipelined-Prozessors

Element	Parameter	Verzögerung (ps)
Register Clock-to-Q	t_{pcq_PC}	30
Register Setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Speicher Lesen	t_{mem}	250
Registerfeld Lesen	t_{RFread}	150
Registerfeld Setup	$t_{RFsetup}$	20
Vergleich auf Gleichheit	t_{eq}	40
AND Gatter	t_{AND}	15
Speicher Schreiben	$T_{memwrite}$	220
Registerfeld Schreiben	$t_{RFwrite}$	100

$$T_c = 2 (t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$$
$$= 2 [150 + 25 + 40 + 15 + 25 + 20] \text{ ps} = \mathbf{550 \text{ ps}}$$

Beispiel: Rechenleistung des Pipelined-Prozessors

- Führe Programm mit 100 Milliarden Instruktionen auf Pipelined-MIPS-Prozessor aus
- $CPI = 1,15$
- $T_c = 550 \text{ ps}$

$$\begin{aligned} \text{Ausführungszeit} &= (\# \text{ Instruktionen}) \times CPI \times T_c \\ &= (100 \times 10^9) (1,15) (550 \times 10^{-12}) \\ &= 63 \text{ Sekunden} \end{aligned}$$

Prozessor	Ausführungszeit (Sekunden)	Beschleunigungsfaktor (im Vergleich zu Ein-Takt-CPU)
Ein-Takt	95	1,00
Mehrtakt	133	0,71
Pipelined	63	1,51

Wiederholung: Ausnahmebehandlung (*exceptions*)

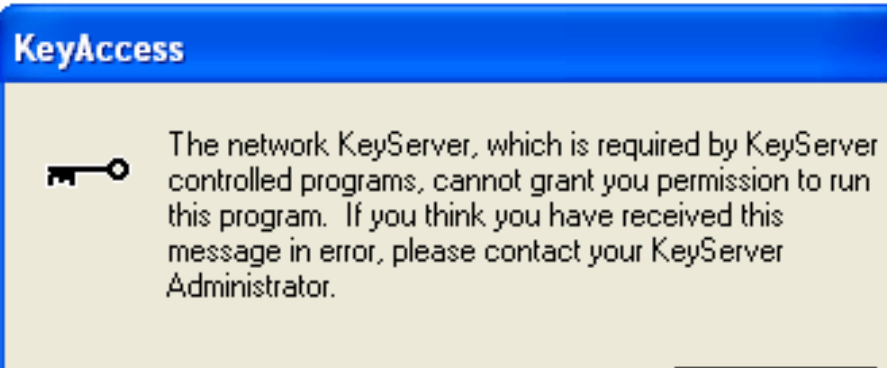
- Außerplanmäßiger Aufruf der Ausnahmebehandlungsroutine
- Verursacht durch:
 - Hardware, auch genannt *Interrupt*, z.B. Tastatur, Netzwerk, ...
 - Software, auch genannt *Traps*, z.B. unbekannte Instruktion, Überlauf, Teilen-durch-Null, ...
- Beim Auftreten einer Ausnahme:
 - Abspeichern der Ursache für Ausnahme im `Cause Register`
 - Sprung zu Ausnahmebehandlungsroutine bei `0x80000180`
 - Rückkehr zum Programm (über `EPC Register`)

Beispiel für Ausnahme

sequential circuits.¶

Can we design a spiff

Figure 2.11 shows a box indicates that in this case, the function is



Visio.exe - Application Error



The exception unknown software exception (0xc06d007e) occurred in the application at location 0x7c81eb33.

OK

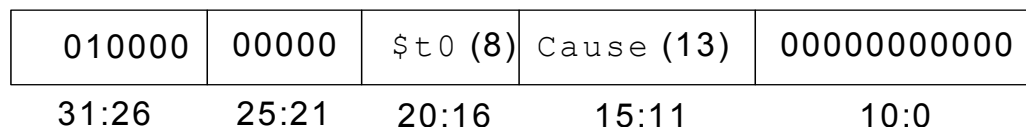
words, we say the output Y is a function of the two inputs A and B where the function performed is $A \text{ OR } B$.¶

The *implementation* of the combinational circuit is independent of its functionality. Figure 2.1 and Figure 2.2 show two possible implementa-

Register für Ausnahmebehandlung

- Nicht Teil des regulären MIPS Registersfelds
 - Cause
 - Speichert die Ursache der Ausnahme
 - Koprozessor 0, Register 13
 - EPC (Exception PC)
 - Speichert den PC-Stand, an dem die Aufnahme auftrat
 - Koprozessor 0, Register 14
- Befehl: “Move from Coprocessor 0”
 - `mfc0 $t0, Cause`
 - Überträgt aktuellen Wert von `Cause` nach `$t0`

mfc0



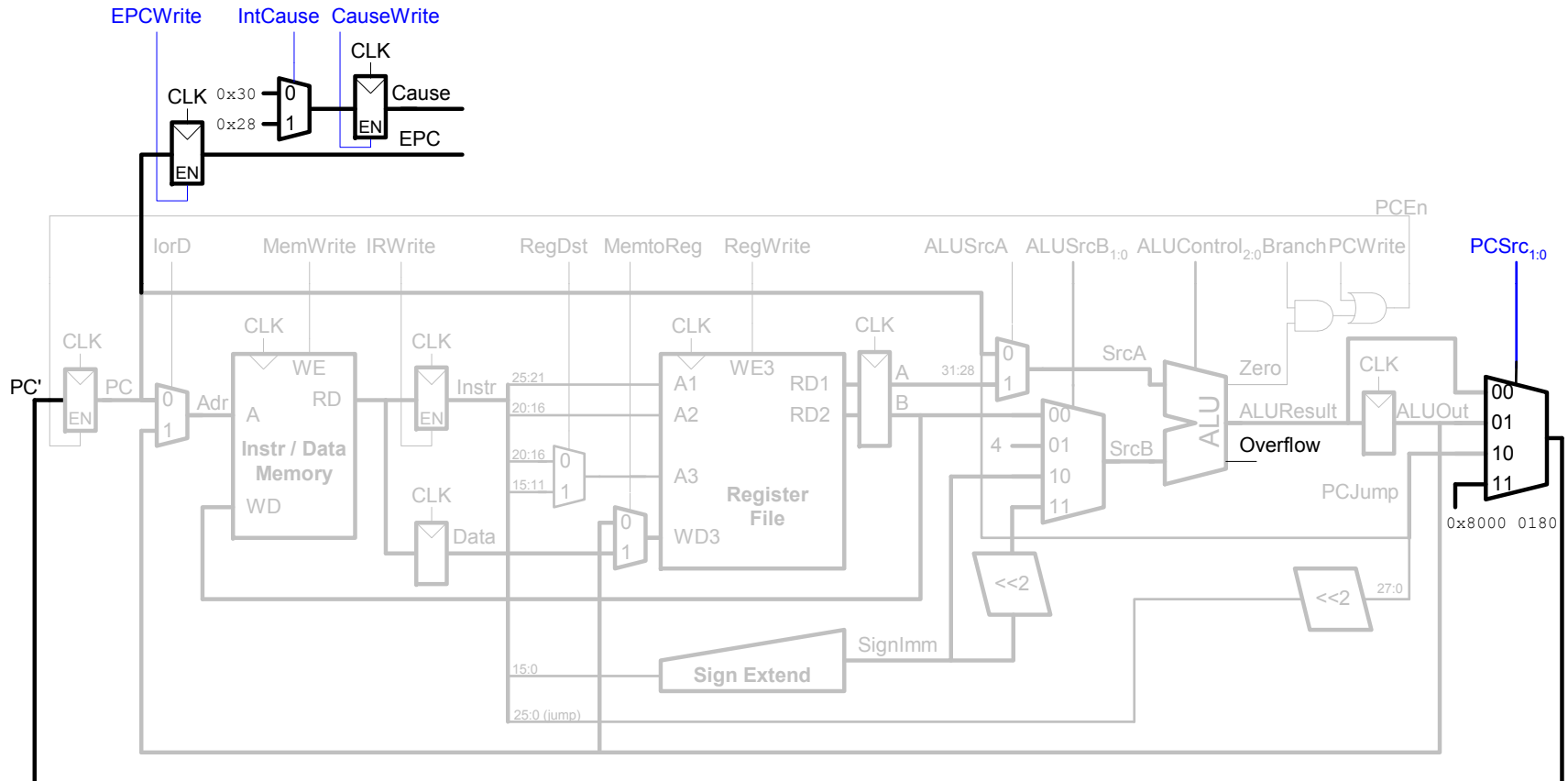
Auswahl von Ausnahmeursachen



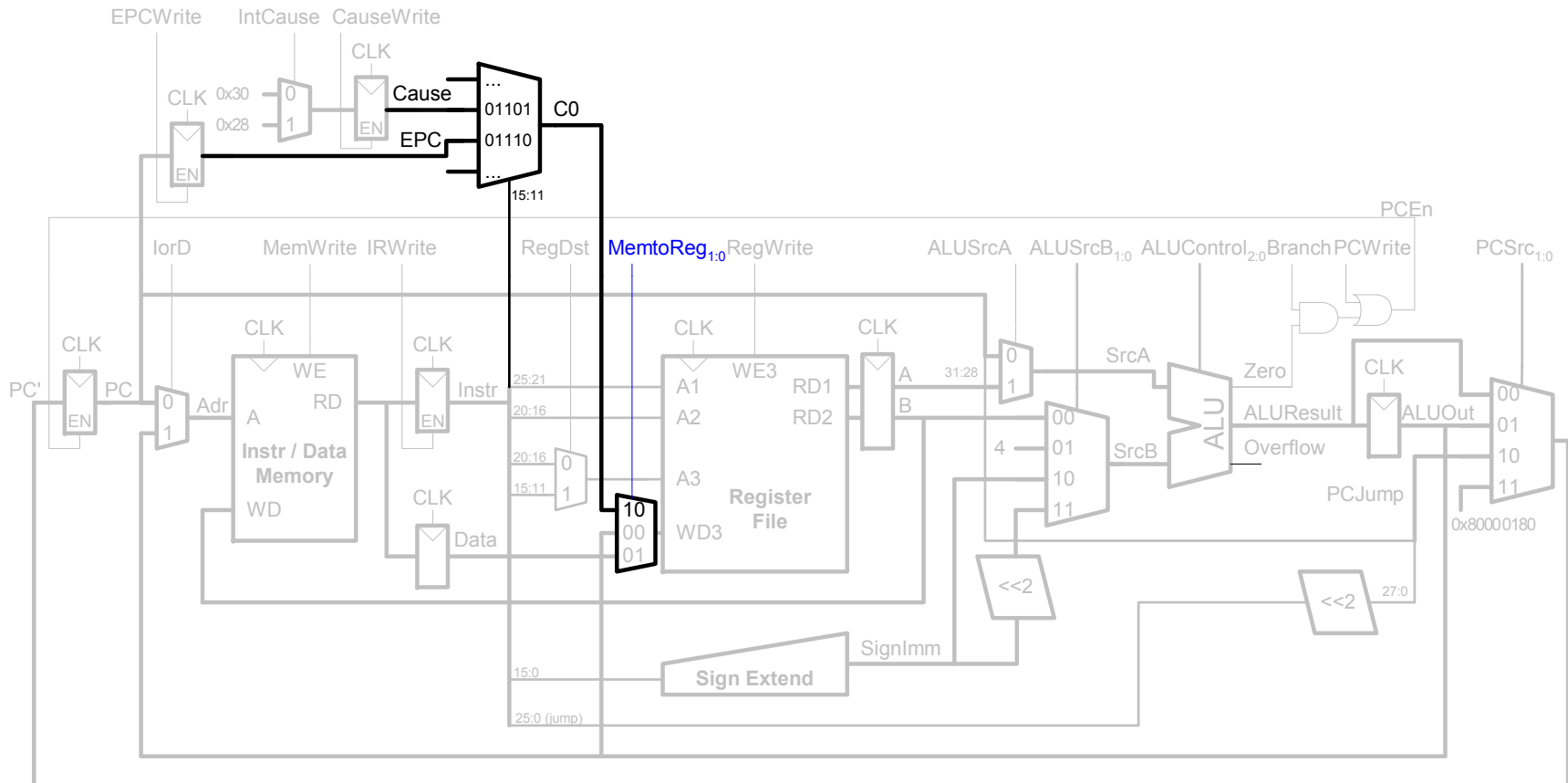
Ausnahme	Ursache
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Division durch 0	0x00000024
Unbekannte Instruktion	0x00000028
Arithmetischer Überlauf	0x00000030

**Ziel: Erweitere den Mehrtaktprozessor um
Behandlung der letzten beiden Ausnahmen**

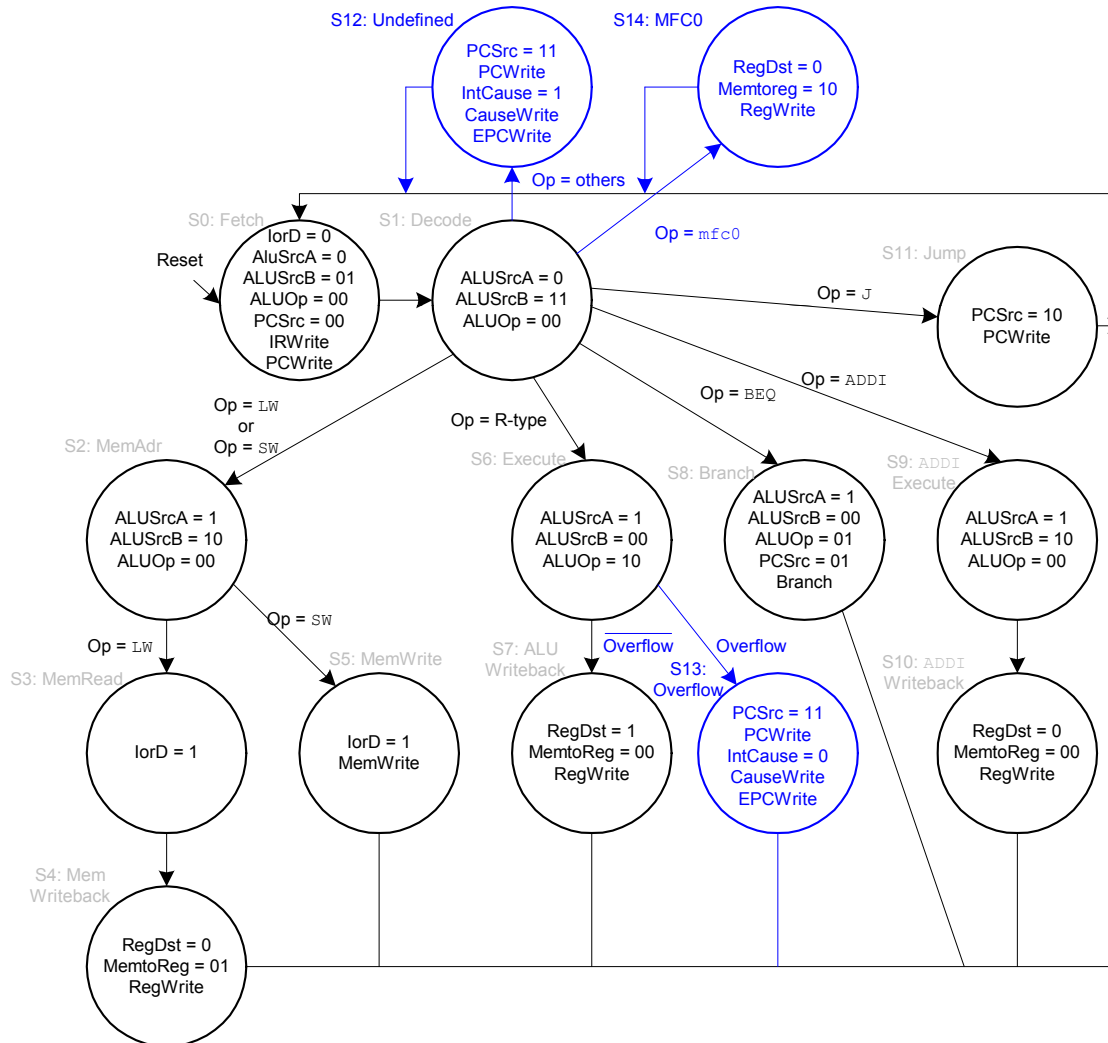
Hardware für Ausnahmebehandlung: EPC und Cause



Hardware für Ausnahmebehandlung : mfc0



Steuerwerk-FSM erweitert um Ausnahmen



Weiterführende Themen der Mikroarchitektur



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Tiefe Pipelines
- Sprungvorhersage
- Superskalare Prozessoren
- Out of Order-Prozessoren
- Umbenennen von Registern
- SIMD
- Multithreading
- Multiprozessoren



- Üblicherweise 10-20 Stufen
 - **Ausnahmen**
 - Fehlkonstruktionen (Intel P4, >30 Stufen)
 - Anwendungsspezifische Spezialprozessoren (ggf. Hunderte von Stufen)
- **Grenzen** für Pipeline-Tiefe
 - Pipeline Hazards
 - Zusätzlicher Zeitaufwand für sequentielle Schaltungen
 - Elektrische Leistungsaufnahme und Energiebedarf
 - Kosten

- **Idealer** Pipelined-Prozessor: $CPI = 1$
- Fehler der Sprungvorhersage **erhöht** CPI

- **Statische** Sprungvorhersage:
 - Prüfe Sprungrichtung (vorwärts oder rückwärts)
 - Falls rückwärts: Sage “Springen” vorher
 - Sonst: Sage “Nicht Springen” vorher

- **Dynamische** Sprungvorhersage:
 - Führe Historie der letzten (einige Hundert) Verzweigungen in *Branch Target Buffer*, speichert:
 - Sprungziel
 - Wurde Sprung das letzte Mal / die letzten Male genommen?



Beispiel: Sprungvorhersage

```
add  $s1, $0, $0      # sum = 0
add  $s0, $0, $0      # i = 0
addi $t0, $0, 10      # $t0 = 10
for:
beq  $s0, $t0, done  # falls i == 10, springe
add  $s1, $s1, $s0    # sum = sum + i
addi $s0, $s0, 1      # inkrementiere i
j    for
done:
```

1-Bit Sprungvorhersage



- Speichert, ob die Verzweigung das letzte Mal **genommen** wurde
 - ... und sagt **genau** dieses Verhalten für das aktuelle Mal vorher
- **Fehlvorhersagen**
 - Einmal bei Austritt aus der Schleife bei Schleifenende
 - Dann wieder bei erneutem Eintritt in Schleife

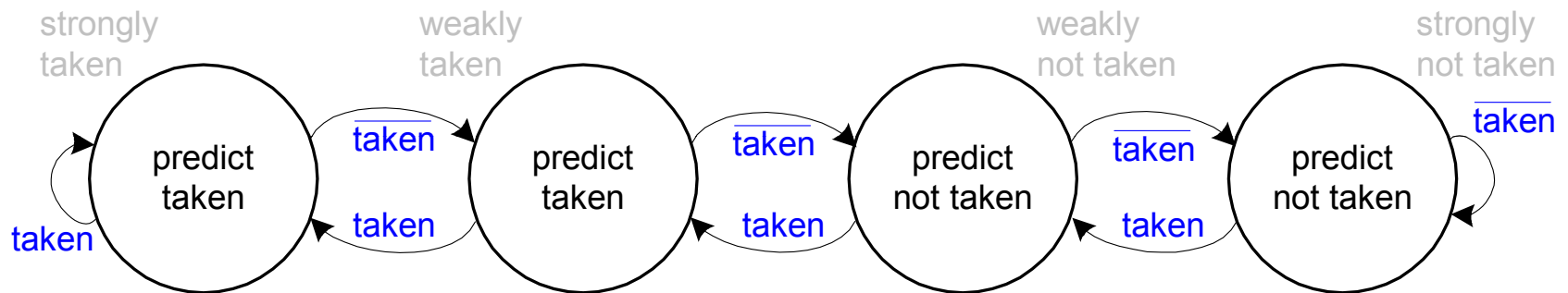
```
add  $s1, $0, $0      # sum = 0
add  $s0, $0, $0      # i   = 0
addi $t0, $0, 10      # $t0 = 10

for:
    beq  $s0, $t0, done # falls i == 10, springe
    add  $s1, $s1, $s0   # sum = sum + i
    addi $s0, $s0, 1     # inkrementiere i
    j    for

done:
```


2-Bit Sprungvorhersage

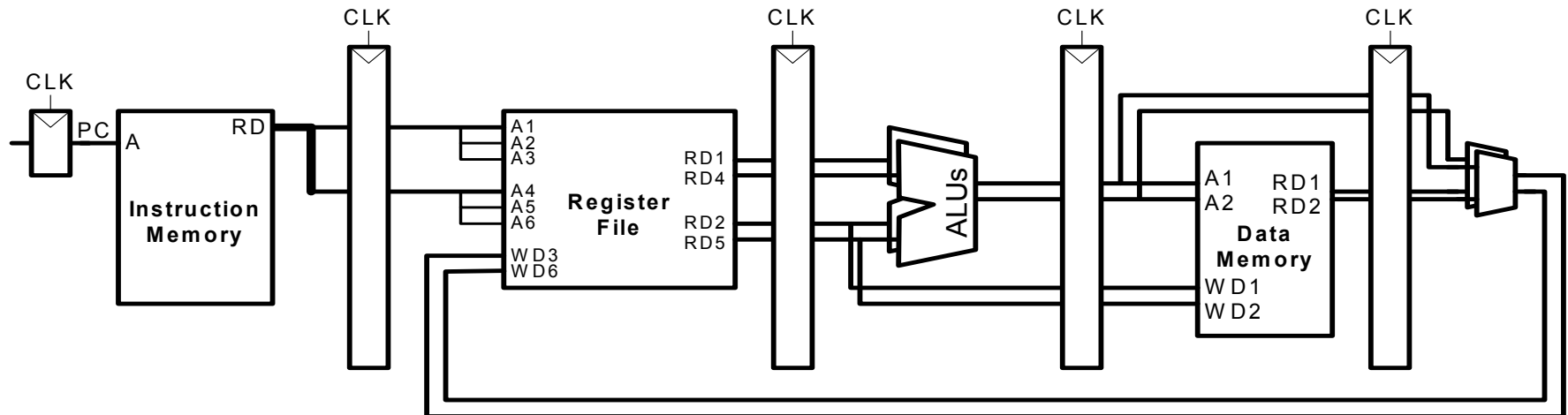
- Falsche Vorhersage nur beim **letzten** Sprung aus Schleife heraus



```
add $s1, $0, $0      # sum = 0
add $s0, $0, $0      # i = 0
addi $t0, $0, 10     # $t0 = 10
for:
  beq $s0, $t0, done # falls i == 10, springe
  add $s1, $s1, $s0  # sum = sum + i
  addi $s0, $s0, 1   # inkrementiere i
  j for
done:
```

Superskalare Mikroarchitektur

- Mehrere **Instanzen** des Datenpfades führen mehrere Instruktionen gleichzeitig aus
- Abhängigkeiten zwischen Instruktionen **erschweren** parallele Ausführung

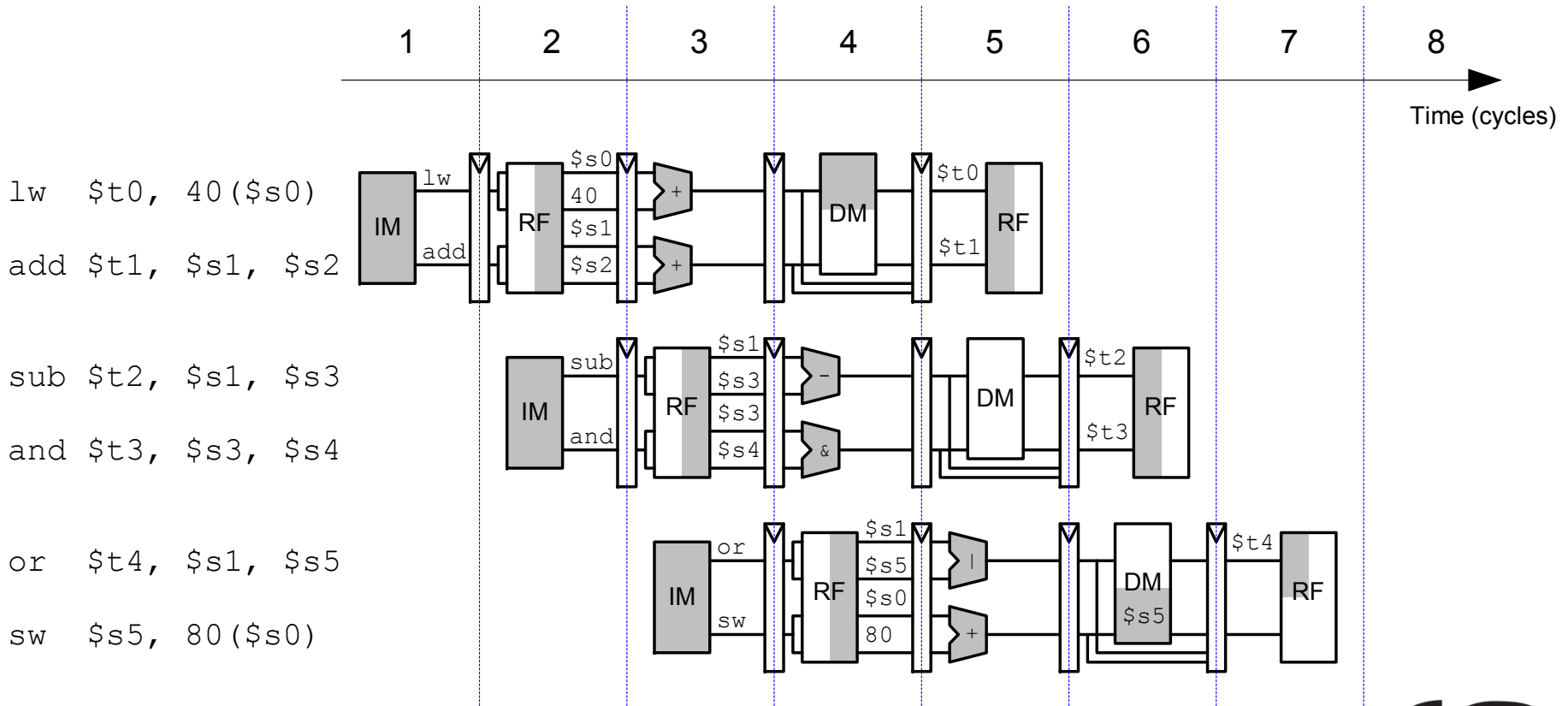


Beispiel: Superskalare Ausführung



Idealer IPC-Wert: 2

Erreichter IPC-Wert: 2

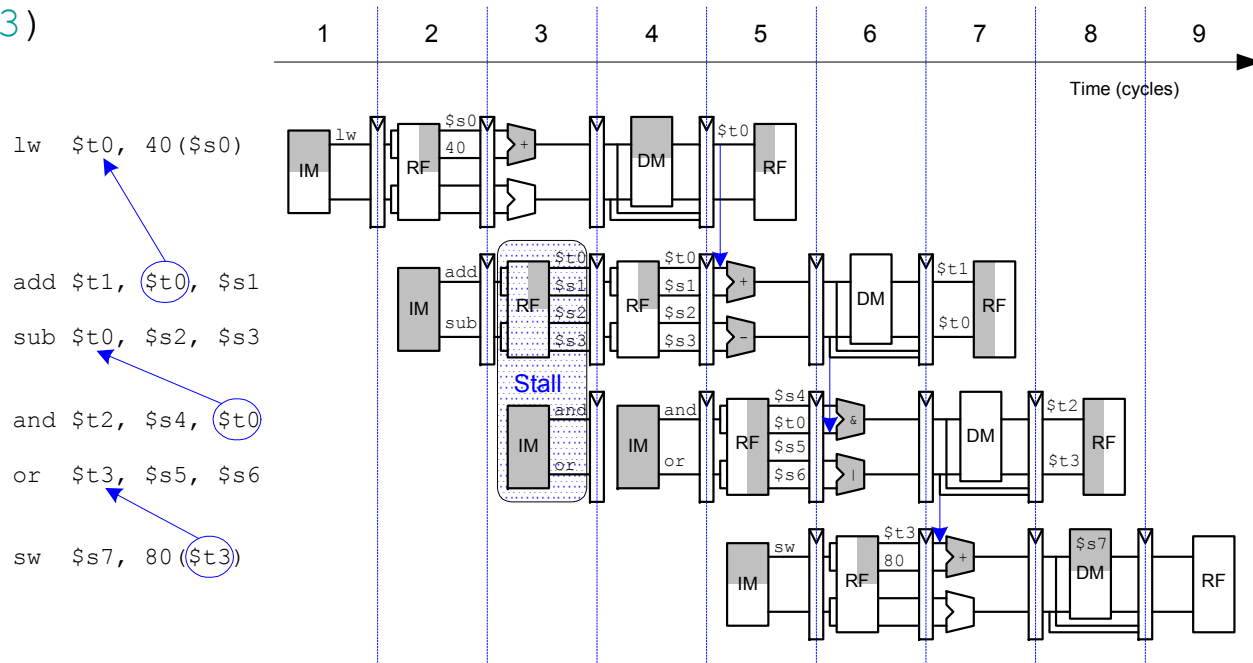


Beispiel: Superskalare Ausführung mit Abhängigkeiten

```
lw $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

Idealer IPC-Wert: 2,00

Erreichter IPC-Wert: $6/5 = 1,20$





- Kann Ausführungsreihenfolge von Instruktion **umsortieren**
- Sucht im voraus nach **parallel** startbaren Instruktionen
- Startet Instruktionen in **beliebiger** Reihenfolge
 - Solange **keine** Abhängigkeiten verletzt werden!
- **Abhängigkeiten**
 - **RAW** (read after write)
 - Spätere Instruktion darf Register erst lesen, nachdem es vorher geschrieben wurde
 - **WAR** (write after read, anti-dependence)
 - Spätere Instruktion darf Register erst schreiben, nachdem es vorher gelesen wurde
 - **WAW** (write after write, output dependence)
 - Reihenfolge von in Register schreibenden Instruktionen muss eingehalten werden

Out of Order-Mikroarchitektur



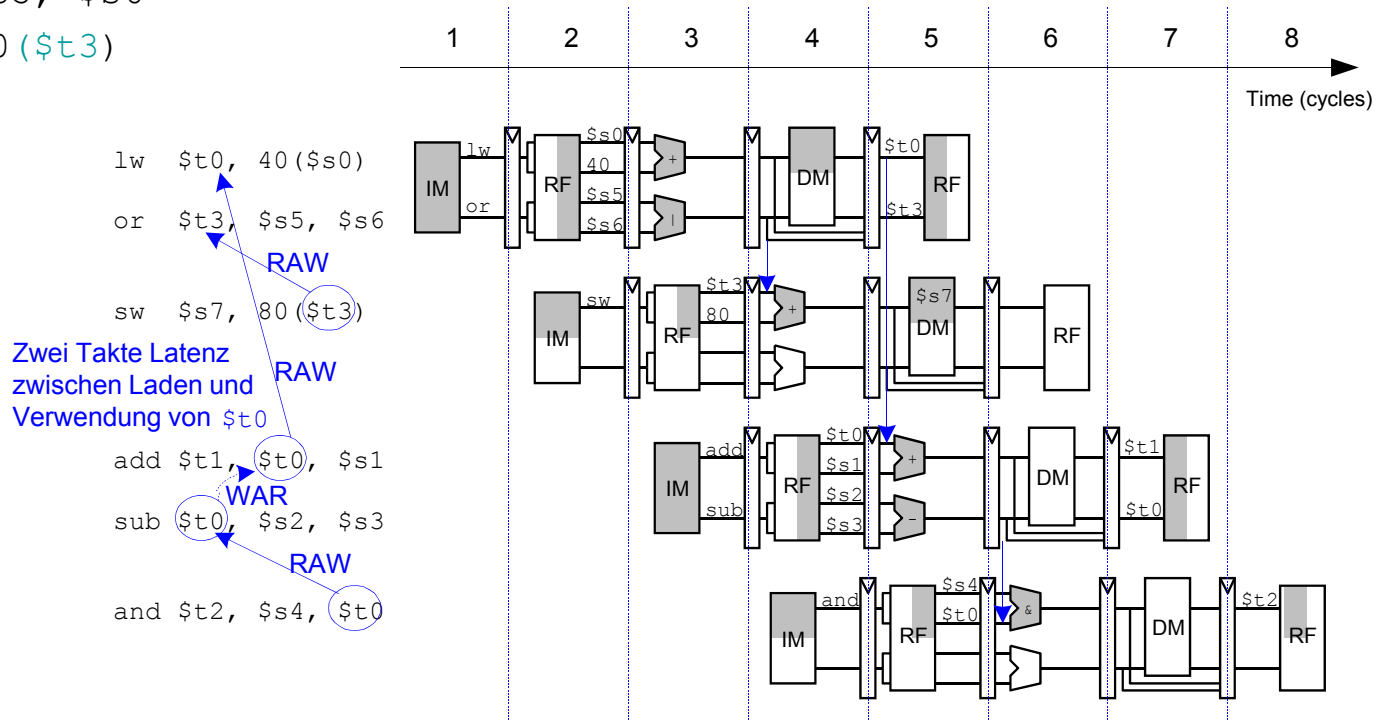
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Parallelismus auf Instruktionsebene (*instruction level parallelism, ILP*)**
 - Anzahl von parallel startbaren Instruktionen (i.d.R. < 3)
- **Scoreboard**
 - Tabelle im Prozessor
 - Verwaltet
 - Auf Start wartende Instruktionen
 - Verfügbare Recheneinheiten (z.B. ALUs)
 - Abhängigkeiten

Beispiel: Out of Order-Mikroarchitektur

```
lw $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

Idealer IPC-Wert: 2,0
Erreichter IPC-Wert: $6/4 = 1,5$



Umbenennen von Registern

```
lw $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

Idealer IPC-Wert: 2,0

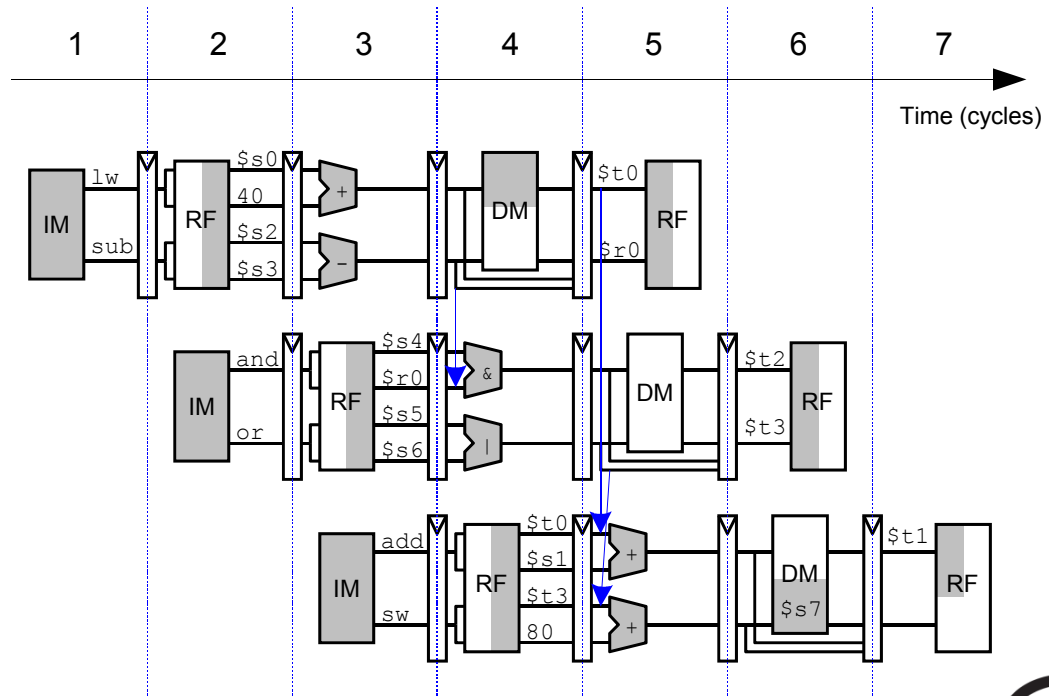
Erreichter IPC-Wert: $6/3 = 2,0$

```
lw $t0, 40($s0)
sub $r0, $s2, $s3
and $t2, $s4, $r0
or $t3, $s5, $s6
add $t1, $t0, $s1
sw $s7, 80($t3)
```

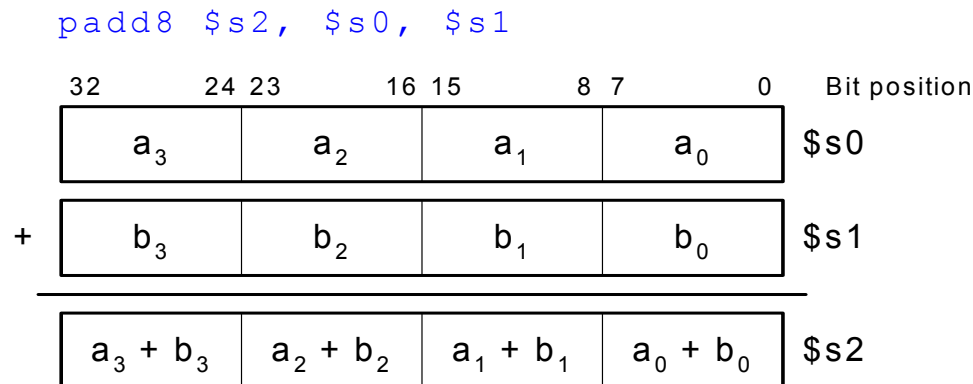
2 Takte RAW

RAW

RAW



- Single Instruction Multiple Data (SIMD)
 - Eine Instruktion wird auf **mehrere** Datenelemente gleichzeitig angewandt
 - Häufige Anwendung: Graphik, Multimedia
 - Oft: Führe **schmale** arithmetische Operatione aus
 - Auch genannt: gepackte Arithmetik
 - Beispiel: Addiere **gleichzeitig** vier Bytes
- ALU muss **verändert** werden
 - Kein Übertrag mehr zwischen einzelnen Bytes



■ Multithreading

- Beispiel: Textverarbeitung
- Threads (parallel laufende, weitgehend unabhängige Instruktionsfolgen)
 - Texteingabe
 - Rechtschreibprüfung
 - Drucken

■ Multiprozessoren

- Viele weitgehend unabhängige Prozessoren auf einem Chip
- Am weitesten verbreitet heute in Grafikkarten (Hunderte von Prozessoren)
 - Aber auch in Spezialprozessoren, z.B. für UMTS Nachfolger LTE

Genauer: Multithreading



- **Prozesse:** Auf dem Computer gleichzeitig laufende Programme
 - z.B. Web-Browser, Musik im Hintergrund, Textverarbeitung
- **Thread:** Parallele Ausführung als Teil eines Programmes
- Ein Prozess kann **mehrere** Threads enthalten
- In konventionellem Prozessor
 - Jeweils **ein** Thread wird ausgeführt
 - Wenn eine Thread-Ausführung einen **Stall** hat (z.B. Warten auf Speicher)
 - **Sichere** Architekturzustand des Threads
 - **Lade** Architekturzustand eines anderen, derzeit inaktiven aber lauffähigen Threads
 - **Starte** neuen Thread
 - Vorgang wird **Kontextumschaltung** (*context switching*) genannt
 - Alle Threads laufen **scheinbar** gleichzeitig

Multithreading auf Mikroarchitekturebene



- Mehrere Instanzen des **Architekturzustandes** im Prozessor
- Mehrere Threads nun **gleichzeitig** aktiv
 - Sobald ein Thread *stalled* wird **sofort** ein anderer gestartet
 - **Kein** Sichern/Laden von Architekturzustand mehr
 - Falls ein Thread nicht alle Recheneinheiten **ausnutzt**, kann dies ein anderer Thread tun
- Erhöht **nicht** den Grad an ILP innerhalb eines Threads
- Erhöht aber **Durchsatz** des Gesamtsystems mit mehreren Threads

- Mehrere unabhängige Prozessorkerne mit einem dazwischenliegenden Kommunikationsnetz
- Arten von Multiprocessing:
 - **Symmetric multiprocessing (SMT)**: mehrere gleiche Kerne mit einem gemeinsamen Speicher
 - **Asymmetric multiprocessing**: unterschiedliche Kerne für unterschiedliche Aufgaben
 - Beispiel: CPU in Handy für GUI, DSP für Funksignalverarbeitung
 - **Clusters**: Jeder Kern hat seinen eigenen Speicher

Weiterführende Informationen

- **Patterson & Hennessy**
Computer Architecture: A Quantitative Approach
- **Konferenzen:**
 - www.cs.wisc.edu/~arch/www/
 - ISCA (International Symposium on Computer Architecture)
 - HPCA (International Symposium on High Performance Computer Architecture)