

All about Processors

A closer look on what is inside a processing unit and how it works

Philip Lukert

Universität des Saarlandes



Übersicht

- Motivation: Vom Transistor zum Computer

Übersicht

- Motivation: Vom Transistor zum Computer
- Addierer aus Transistoren
- MIPS
- Prozessor
- Pipelining, ...
- Fragen / „Diskussion“

Addierer

Can you imagine how transistors could be used in a circuit to add two numbers?

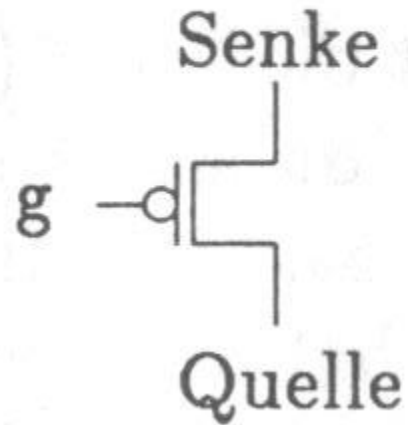
Transistoren

- “Elektronische Schalter”

Transistoren

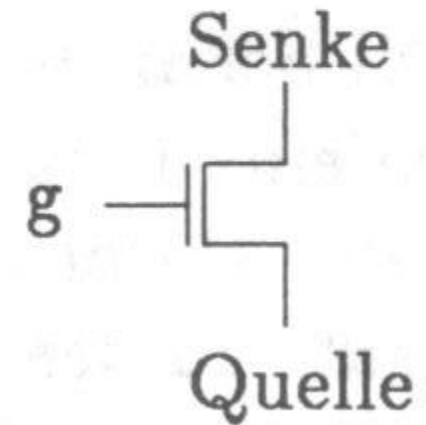
- “Elektronische Schalter”

p-Kanal Transistor



sperrt
wenn an g eine 1 anliegt

n-Kanal Transistor



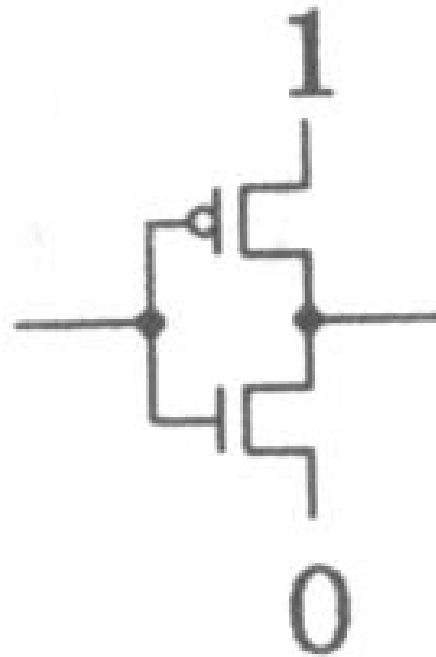
leitet
wenn an g eine 1 anliegt

Transistoren

- Konstruktion von logischen Gattern

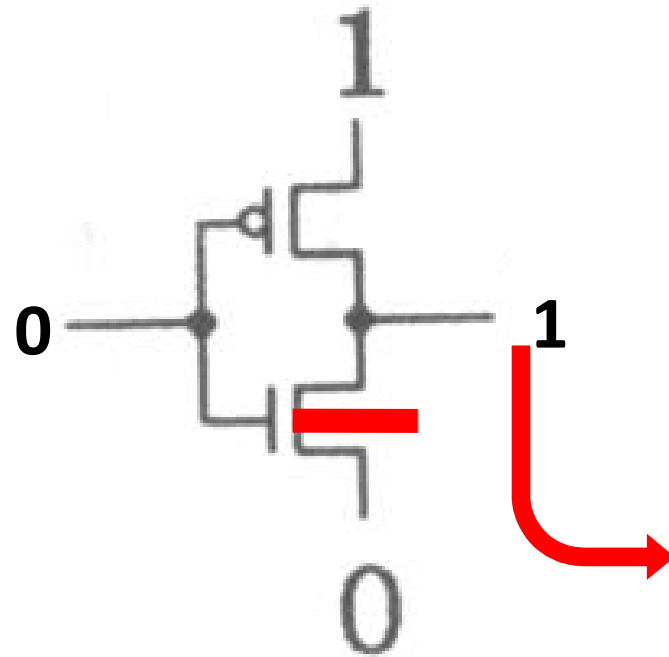
Transistoren

- Konstruktion von logischen Gattern
- Beispiel: NOT



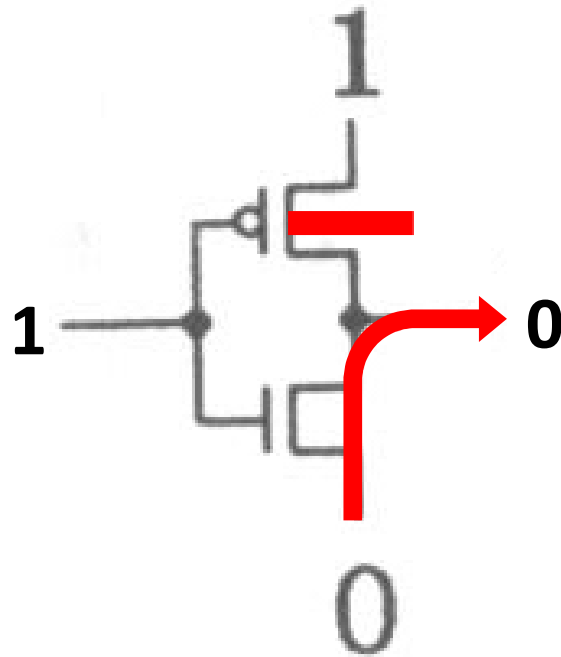
Transistoren

- Konstruktion von logischen Gattern
- Beispiel: NOT



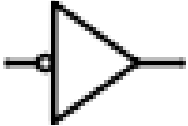

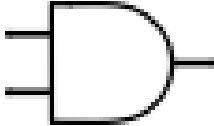


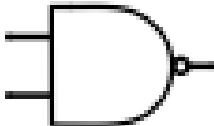
Transistoren

- Konstruktion von logischen Gattern
- Beispiel: NOT



Transistoren

- Konstruktion von logischen Gattern
- Beispiel: NOT
- Analog können weitere Gatter konstruiert werden:

NOT 	OR 	AND 
XOR 	NOR 	NAND 

Addierer

- Ziel: Schaltkreis, der zwei Zahlen addiert

Addierer

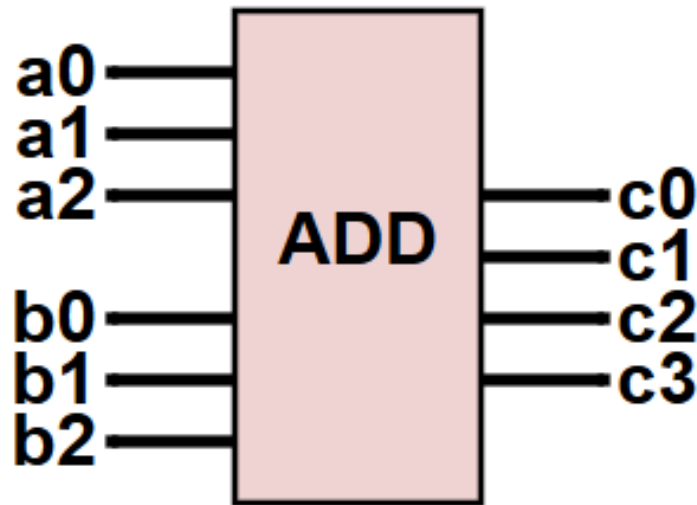
- Ziel: Schaltkreis, der zwei Zahlen addiert
- wie werden Zahlen repräsentiert?

Addierer

- Ziel: Schaltkreis, der zwei Zahlen addiert
- wie werden Zahlen repräsentiert?
- können mit 0/1 kodiert werden (Binärdarstellung)
- Beispiel: $9 = 8+0+0+1 \Rightarrow 1001$

Addierer

- Ziel: Schaltkreis, der zwei Zahlen addiert
- wie werden Zahlen repräsentiert?
- können mit 0/1 kodiert werden (Binärdarstellung)
- Beispiel: $9 = 8 + 0 + 0 + 1 \Rightarrow 1001$
- gesucht:



Addierer

- Ansatz: Addierer für mit 1Bit konstruieren

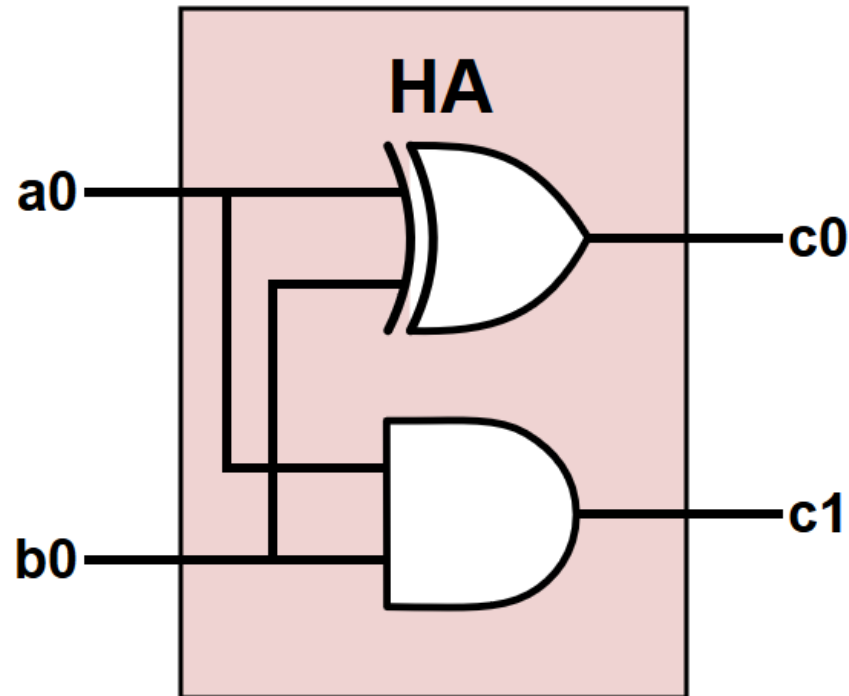
Addierer

- Ansatz: Addierer für mit 1Bit konstruieren
⇒ Halbaddierer

a	b	c	dezimal
0	0	00	0+0=0
0	1	01	0+1=1
1	0	01	1+0=1
1	1	10	1+1=2

Addierer

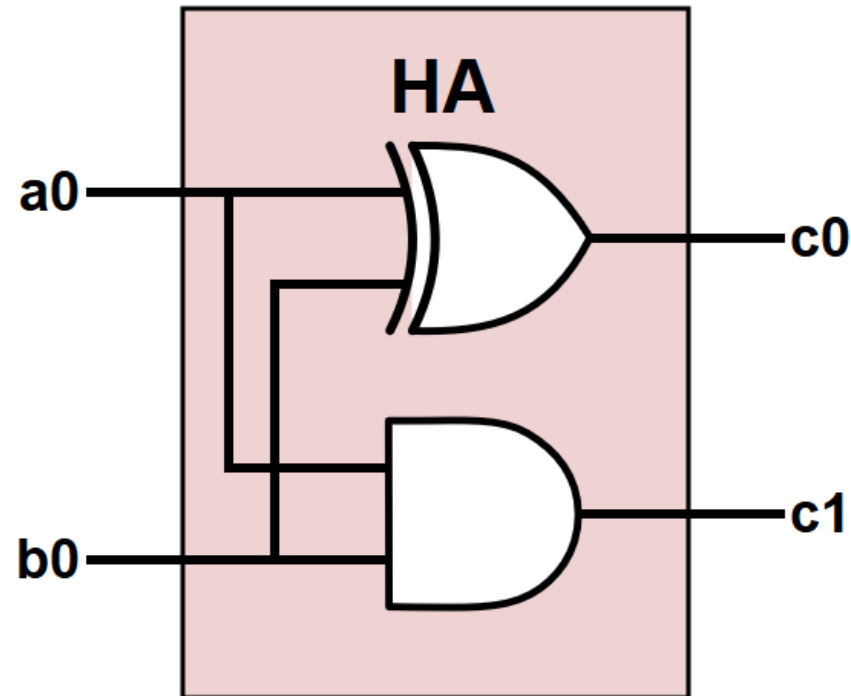
- Ansatz: Addierer für mit 1Bit konstruieren
⇒ Halbaddierer



a	b	c	dezimal
0	0	00	0+0=0
0	1	01	0+1=1
1	0	01	1+0=1
1	1	10	1+1=2

Addierer

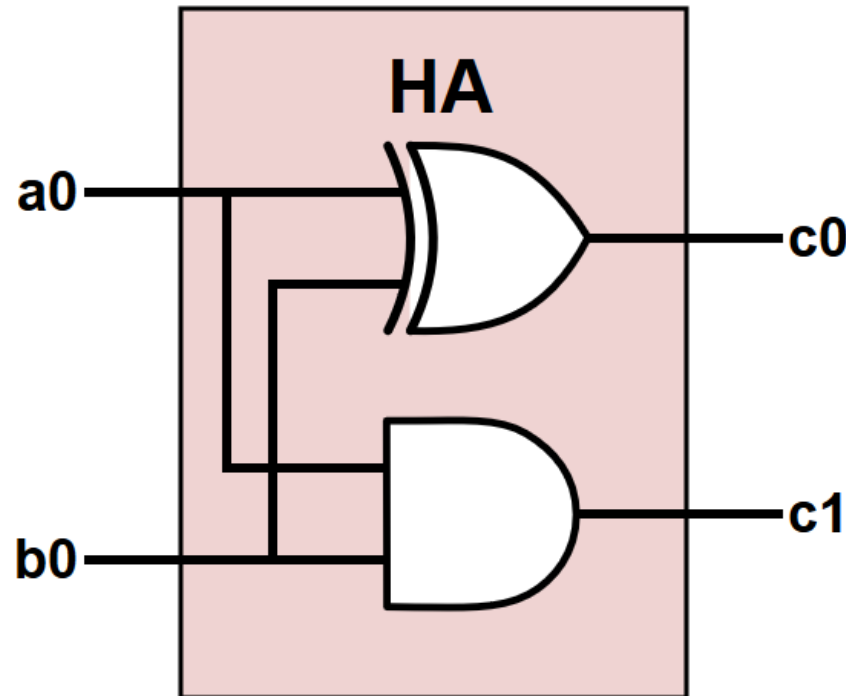
- Ansatz: Addierer für mit 1Bit konstruieren
⇒ Halbaddierer
- c braucht 2 Bit!



a	b	c	dezimal
0	0	00	0+0=0
0	1	01	0+1=1
1	0	01	1+0=1
1	1	10	1+1=2

Addierer

- Ansatz: Addierer für mit 1Bit konstruieren
⇒ Halbaddierer
- c braucht 2 Bit!
- c_1 ist "Übertrag"



a	b	c	dezimal
0	0	00	$0+0=0$
0	1	01	$0+1=1$
1	0	01	$1+0=1$
1	1	10	$1+1=2$

Addierer

- Ansatz: Addierer für mit 1-Bit konstruieren
⇒ Halbaddierer
- c braucht 2 Bit!
- c_1 ist “Übertrag”
- hintereinander-schalten von 1-Bit Addierern

Addierer

- Ansatz: Addierer für mit 1-Bit konstruieren
⇒ Halbaddierer
- c braucht 2 Bit!
- c_1 ist “Übertrag”
- hintereinander-schalten von 1-Bit Addierern
- ein Eingangsbit mehr benötigt

Addierer

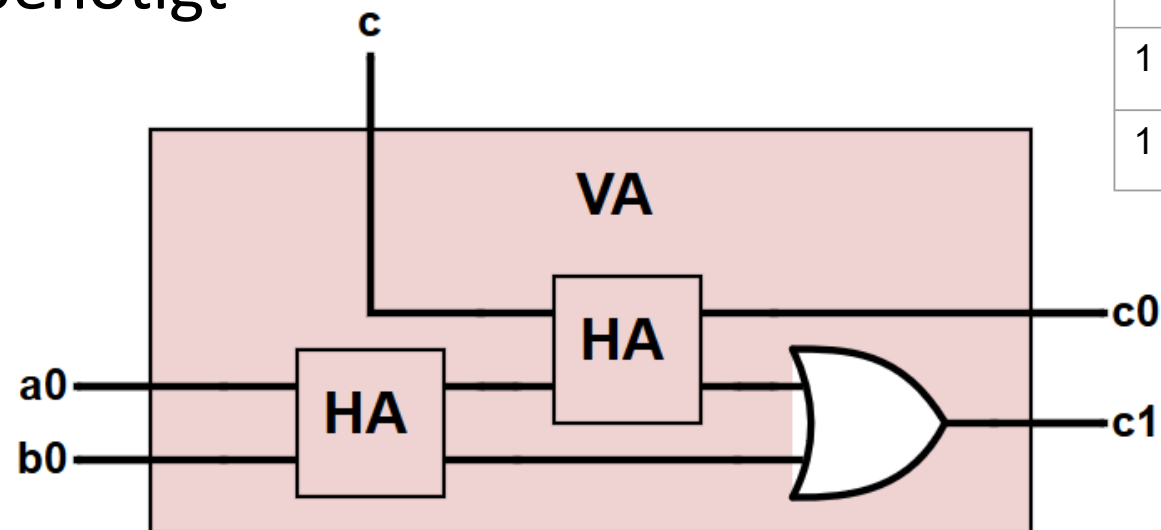
- Ansatz: Addierer für mit 1-Bit konstruieren
⇒ Halbaddierer
- c braucht 2 Bit!
- c_1 ist “Übertrag”
- hintereinander-schalten von 1-Bit Addierern
- ein Eingangsbit mehr benötigt
⇒ Volladdierer

a	b	c	out	dezimal
0	0	0	00	$0+0+0=0$
0	0	1	01	$0+0+1=1$
0	1	0	01	$0+1+0=1$
0	1	1	10	$0+1+1=2$
1	0	0	01	$1+0+0=1$
1	0	1	10	$1+0+1=2$
1	1	0	10	$1+1+0=2$
1	1	1	11	$1+1+1=3$

Addierer

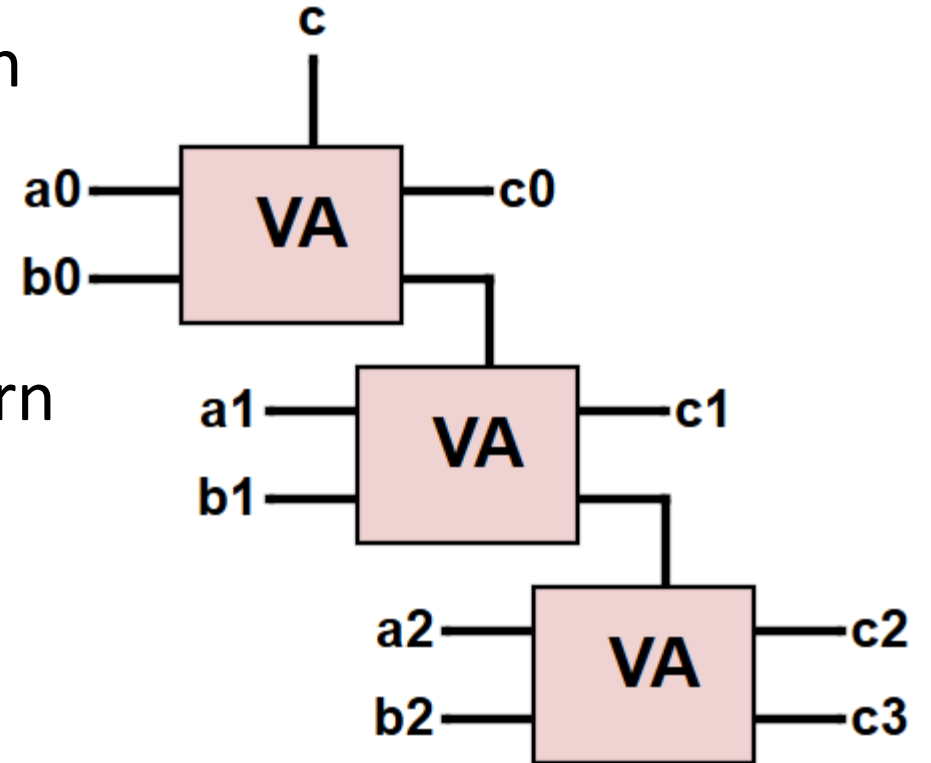
- Ansatz: Addierer für mit 1-Bit konstruieren
⇒ Halbaddierer
- c braucht 2 Bit!
- c_1 ist “Übertrag”
- hintereinander-schalten von 1-Bit Addierern
- ein Eingangsbit mehr benötigt
⇒ Volladdierer

a	b	c	out	dezimal
0	0	0	00	$0+0+0=0$
0	0	1	01	$0+0+1=1$
0	1	0	01	$0+1+0=1$
0	1	1	10	$0+1+1=2$
1	0	0	01	$1+0+0=1$
1	0	1	10	$1+0+1=2$
1	1	0	10	$1+1+0=2$
1	1	1	11	$1+1+1=3$



Addierer

- Ansatz: Addierer für mit 1-Bit konstruieren
⇒ Halbaddierer
- c braucht 2 Bit!
- c_1 ist “Übertrag”
- hintereinander-schalten von 1-Bit Addierern
- ein Eingangsbit mehr benötigt
⇒ Volladdierer
- n Volladdierer \Rightarrow n -Bit Addierer



MIPS

kurze Einführung in die Programmiersprache

MIPS

Code:

```
1 main:
2  li $t1, 4          #load immediate
3  li $t2, 5
4  add $t0, $t1, $t2  #sub/mult/div
5  sw $t0 ($sp)       #store word
6  beq $t0 $t1 main   #branch if equal
```

MIPS

Code:

```
1 main:
2  li $t1, 4          #load immediate
3  li $t2, 5
4  add $t0, $t1, $t2  #sub/mult/div
5  sw $t0 ($sp)       #store word
6  beq $t0 $t1 main   #branch if equal
```

32 Register:

Name	Wert
\$t0	0
\$t1	0
\$t2	0
...	...
\$sp (Stack Pointer)	0x7ffefffc (Adresse)

MIPS

Code:

```
1 main:
2  li $t1, 4           #load immediate
3  li $t2, 5
4  add $t0, $t1, $t2   #sub/mult/div
5  sw $t0 ($sp)        #store word
6  beq $t0 $t1 main    #branch if equal
```

32 Register:

Name	Wert
\$t0	0
\$t1	0
\$t2	0
...	...
\$sp (Stack Pointer)	0x7ffeffc (Adresse)

Arbeitsspeicher:

Adresse	...	0x7ffeff8	0x7ffeffc	0x7fff000	...
Wert	...	0	0	0	...

MIPS

Code:

```
1 main:
2  li $t1, 4           #load immediate
3  li $t2, 5
4  add $t0, $t1, $t2    #sub/mult/div
5  sw $t0 ($sp)         #store word
6  beq $t0 $t1 main     #branch if equal
```

32 Register:

Name	Wert
\$t0	0
\$t1	4
\$t2	0
...	...
\$sp (Stack Pointer)	0x7ffeffc (Adresse)

Arbeitsspeicher:

Adresse	...	0x7ffeff8	0x7ffeffc	0x7fff000	...
Wert	...	0	0	0	...

MIPS

Code:

```
1 main:
2  li $t1, 4          #load immediate
3  li $t2, 5
4  add $t0, $t1, $t2  #sub/mult/div
5  sw $t0 ($sp)       #store word
6  beq $t0 $t1 main   #branch if equal
```

32 Register:

Name	Wert
\$t0	0
\$t1	4
\$t2	5
...	...
\$sp (Stack Pointer)	0x7ffeffc (Adresse)

Arbeitsspeicher:

Adresse	...	0x7ffeff8	0x7ffeffc	0x7fff000	...
Wert	...	0	0	0	...

MIPS

Code:

```
1 main:
2  li $t1, 4          #load immediate
3  li $t2, 5
4  add $t0, $t1, $t2  #sub/mult/div
5  sw $t0 ($sp)       #store word
6  beq $t0 $t1 main   #branch if equal
```

32 Register:

Name	Wert
\$t0	9
\$t1	4
\$t2	5
...	...
\$sp (Stack Pointer)	0x7ffeffc (Adresse)

Arbeitsspeicher:

Adresse	...	0x7ffeff8	0x7ffeffc	0x7fff000	...
Wert	...	0	0	0	...

MIPS

Code:

```
1 main:
2  li $t1, 4          #load immediate
3  li $t2, 5
4  add $t0, $t1, $t2  #sub/mult/div
5  sw $t0 ($sp)       #store word
6  beq $t0 $t1 main   #branch if equal
```

32 Register:

Name	Wert
\$t0	9
\$t1	4
\$t2	5
...	...
\$sp (Stack Pointer)	0x7ffeffc (Adresse)

Arbeitsspeicher:

Adresse	...	0x7ffeff8	0x7ffeffc	0x7fff000	...
Wert	...	0	9	0	...

MIPS

Code:

```
1 main:
2  li $t1, 4          #load immediate
3  li $t2, 5
4  add $t0, $t1, $t2  #sub/mult/div
5  sw $t0 ($sp)       #store word
6  beq $t0 $t1 main   #branch if equal
```

32 Register:

Name	Wert
\$t0	9
\$t1	4
\$t2	5
...	...
\$sp (Stack Pointer)	0x7ffeffc (Adresse)

Arbeitsspeicher:

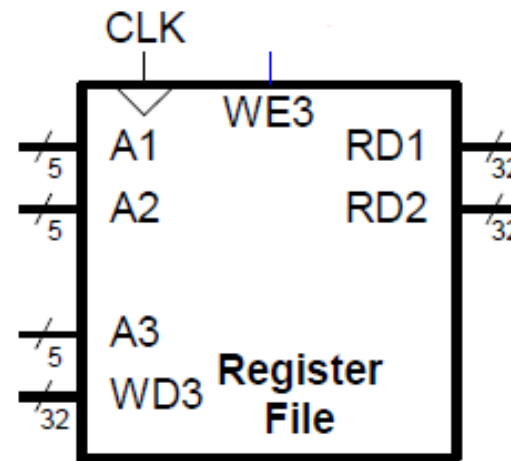
Adresse	...	0x7ffeff8	0x7ffeffc	0x7fff000	...
Wert	...	0	9	0	...

Prozessor

wie wird MIPS ausgeführt?

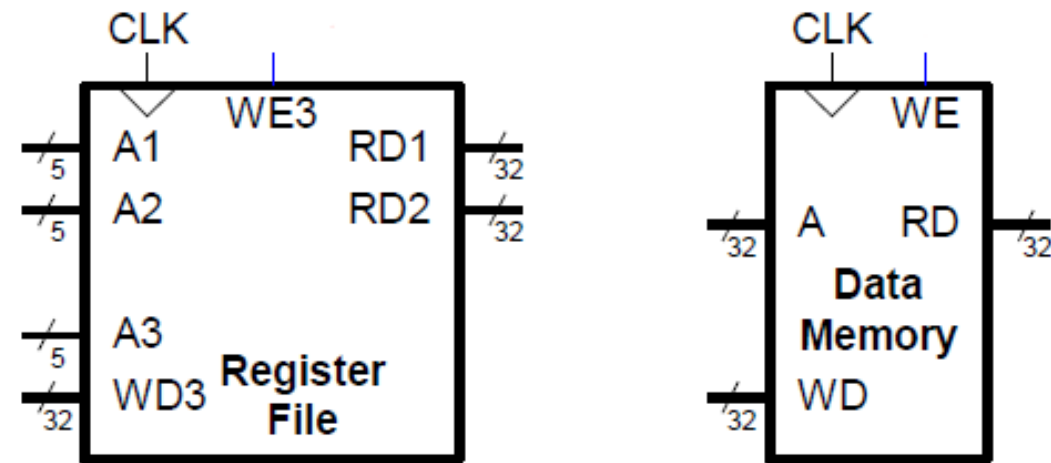
Prozessor

- Register File (32 Mips-Register)



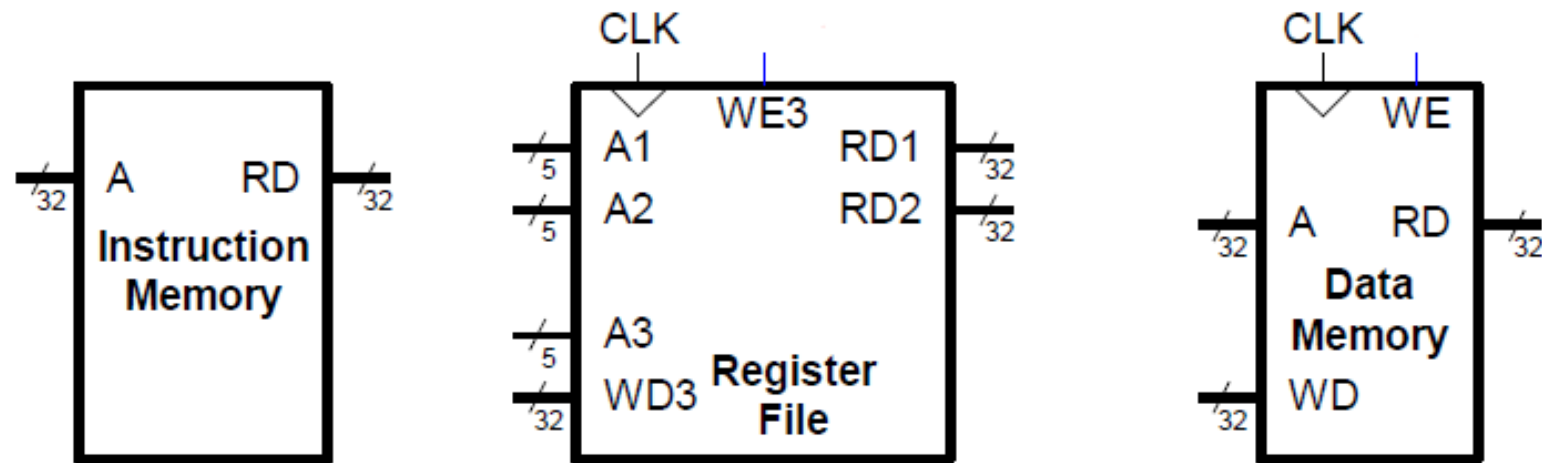
Prozessor

- Register File (32 Mips-Register)
- Arbeitsspeicher:
 - Data-Memory



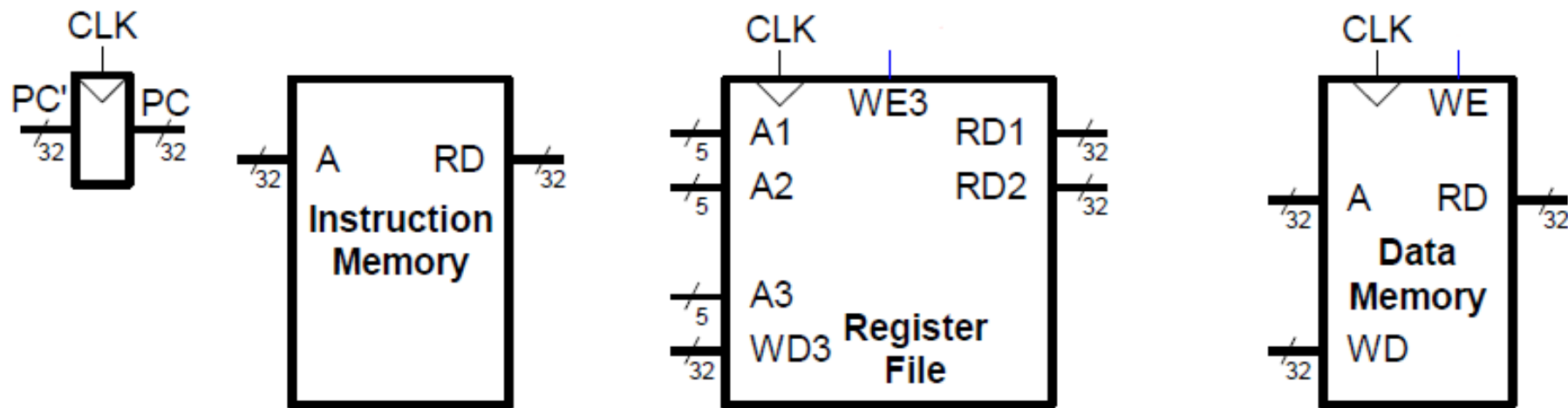
Prozessor

- Register File (32 Mips-Register)
- Arbeitsspeicher:
 - Data-Memory
 - Instruction-Memory



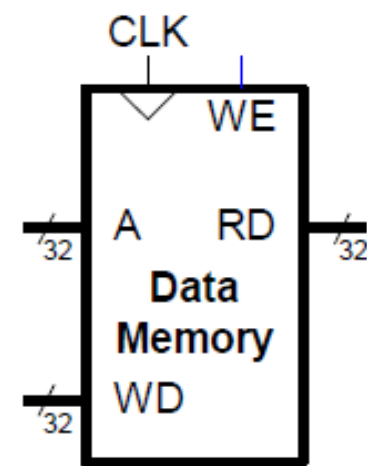
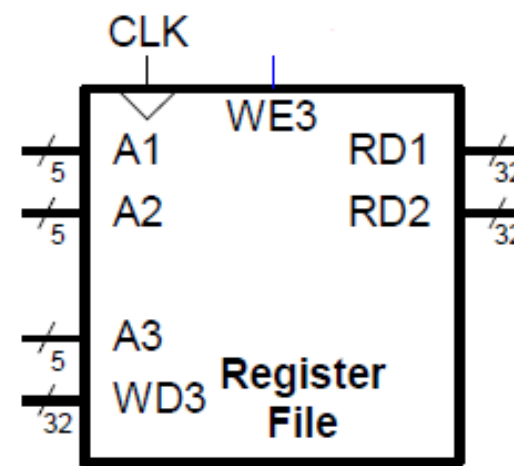
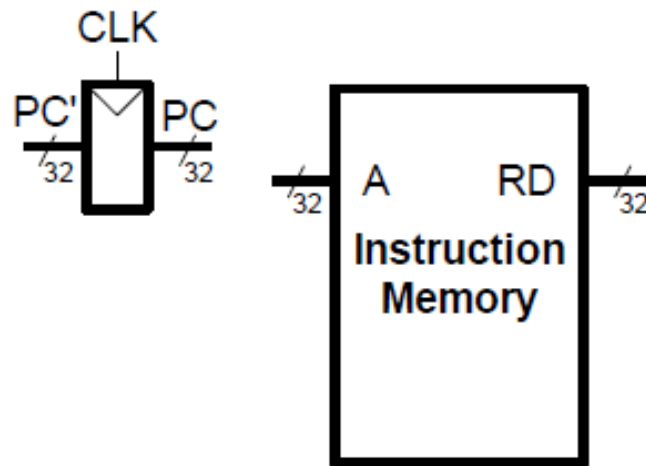
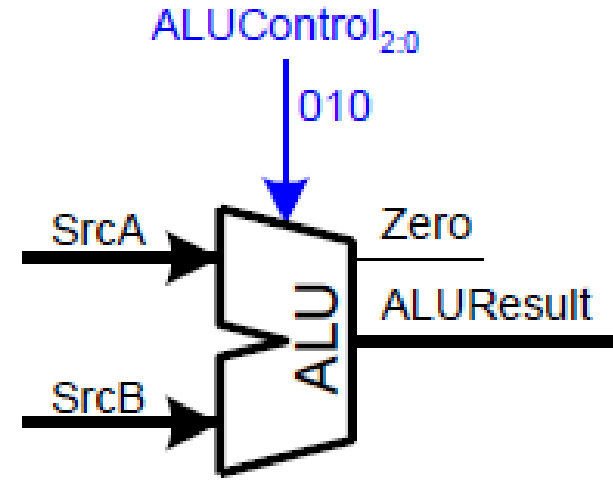
Prozessor

- Register File (32 Mips-Register)
- Arbeitsspeicher:
 - Data-Memory
 - Instruction-Memory
- PC (Program Counter)



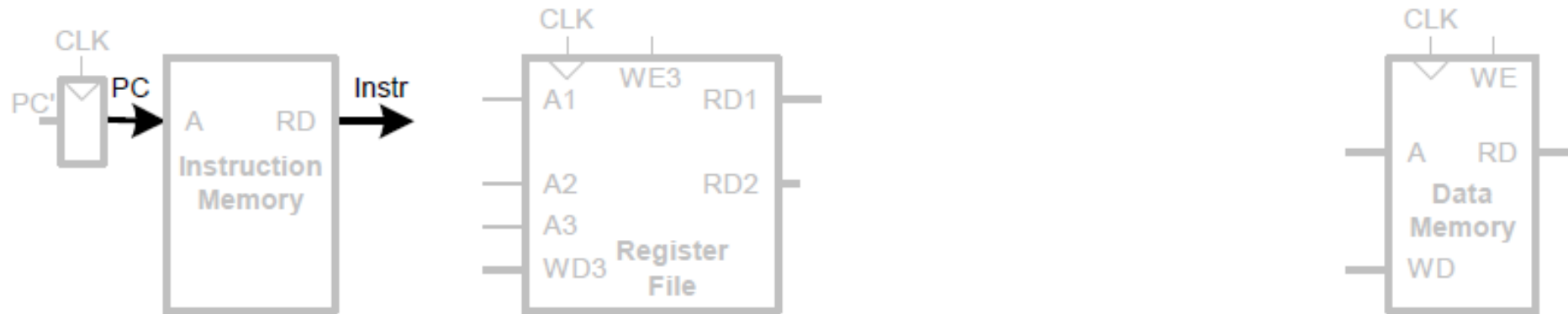
Prozessor

- Register File (32 Mips-Register)
- Arbeitsspeicher:
 - Data-Memory
 - Instruction-Memory
- PC (Program Counter)
- ALU (Arithmetic Logical Unit)



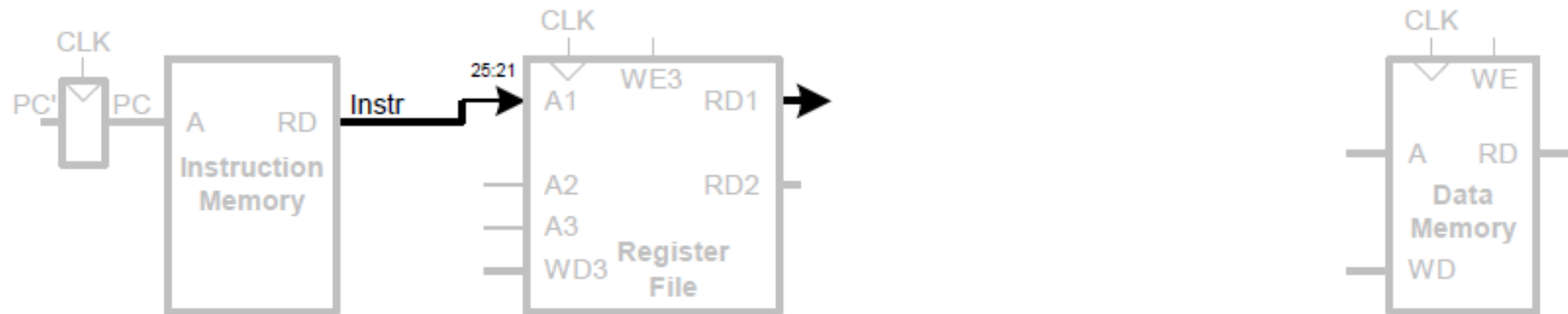
Prozessor

- Instruction Fetch



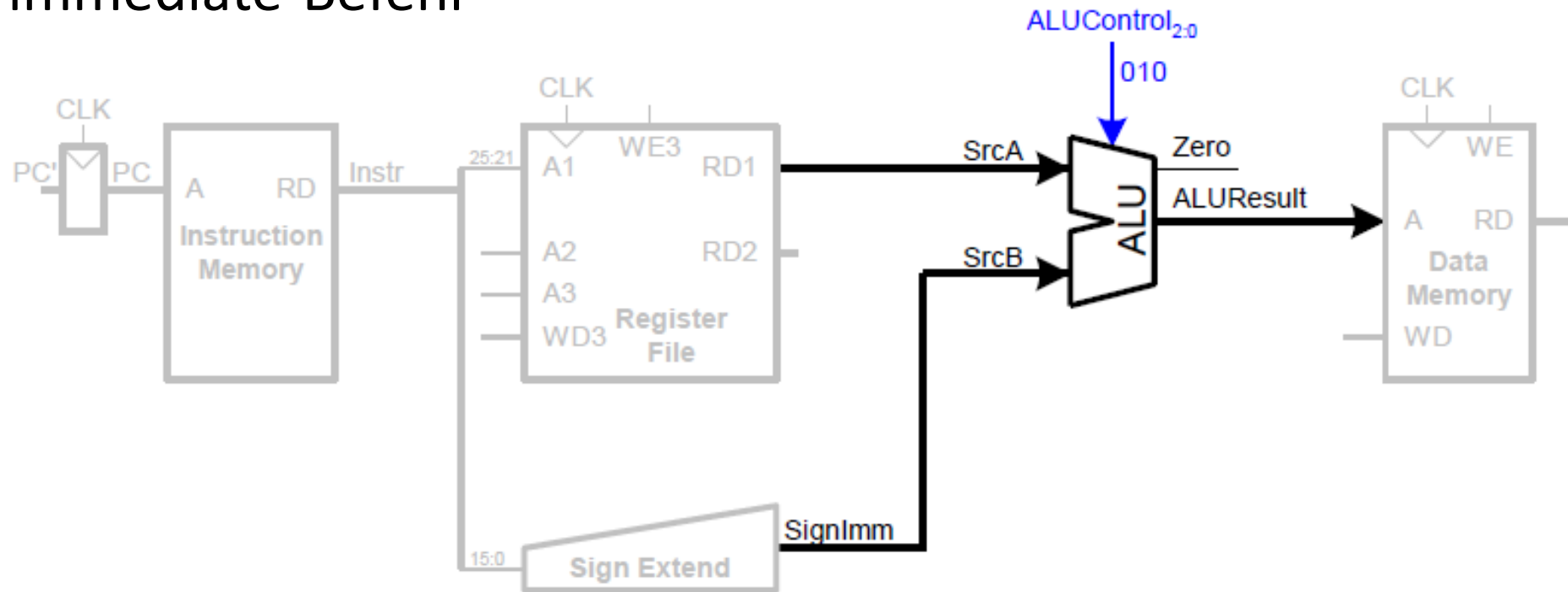
Prozessor

- Register Fetch



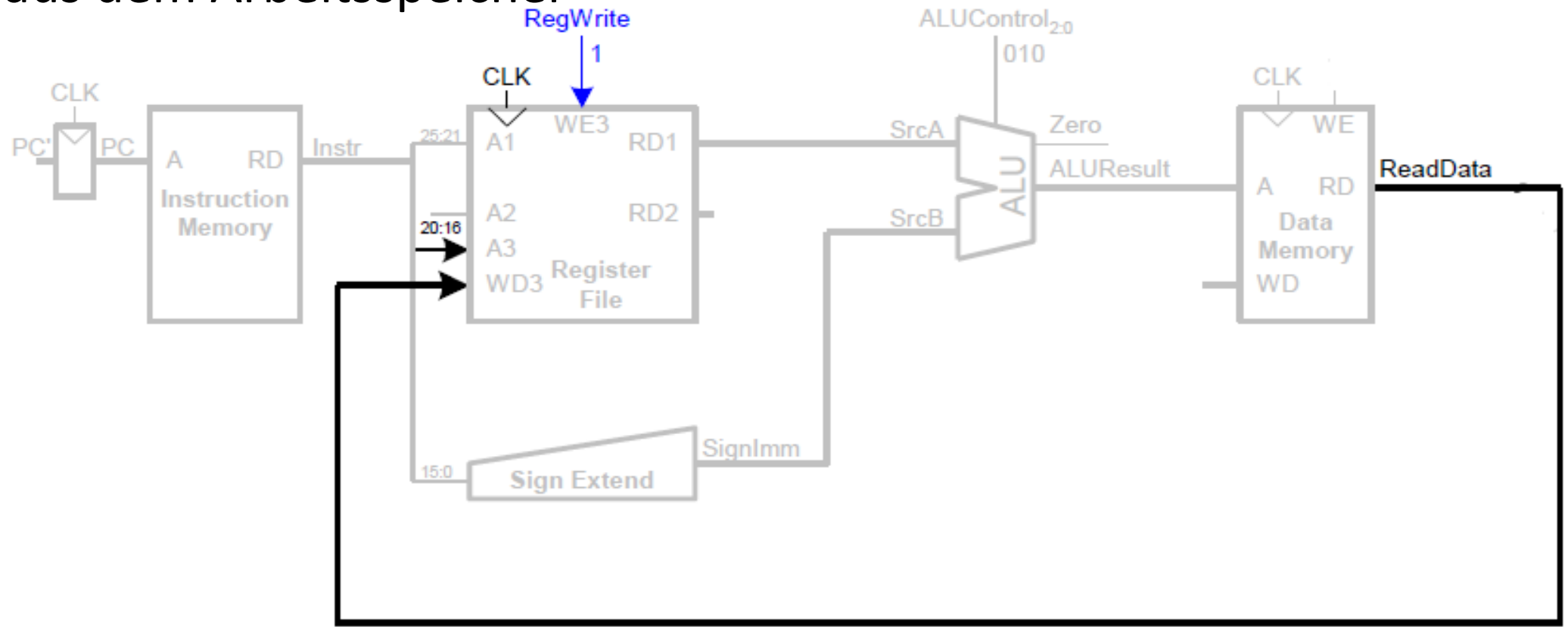
Prozessor

- Execute
- hier: immediate-Befehl



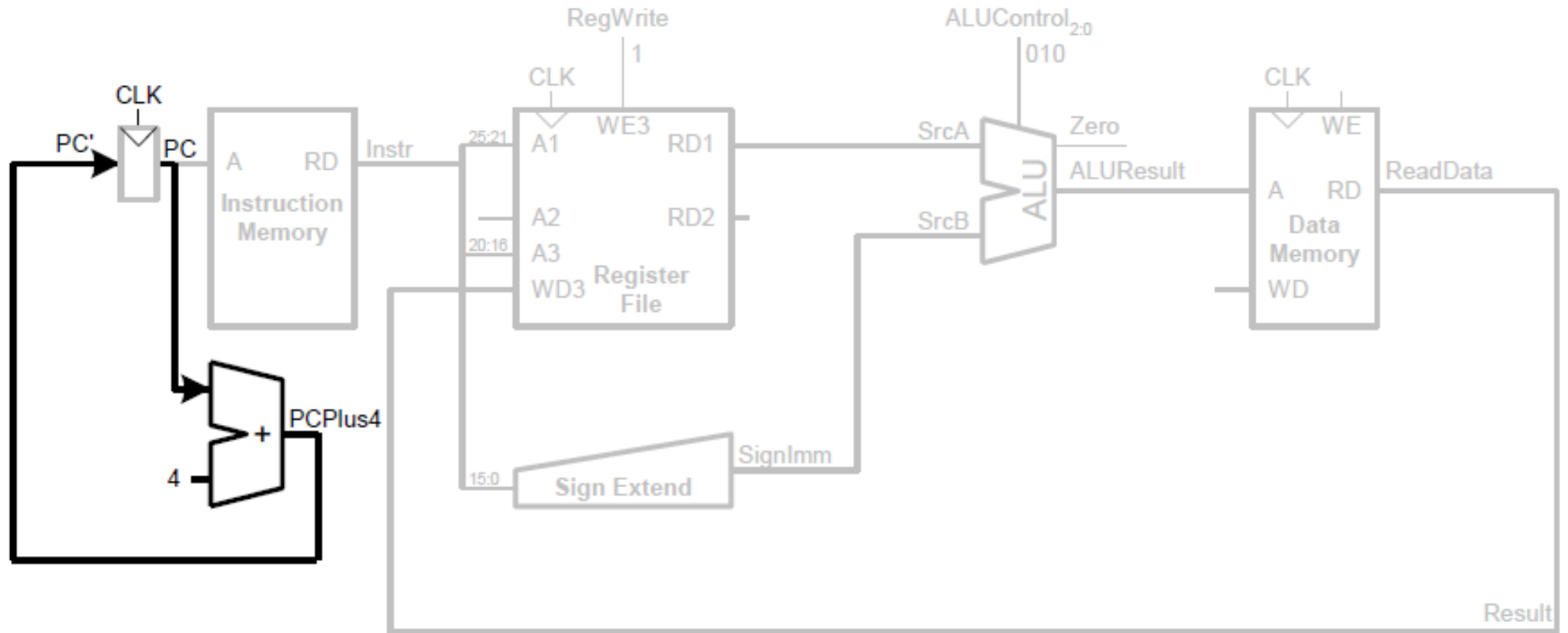
Prozessor

- Register write back
- hier: aus dem Arbeitsspeicher



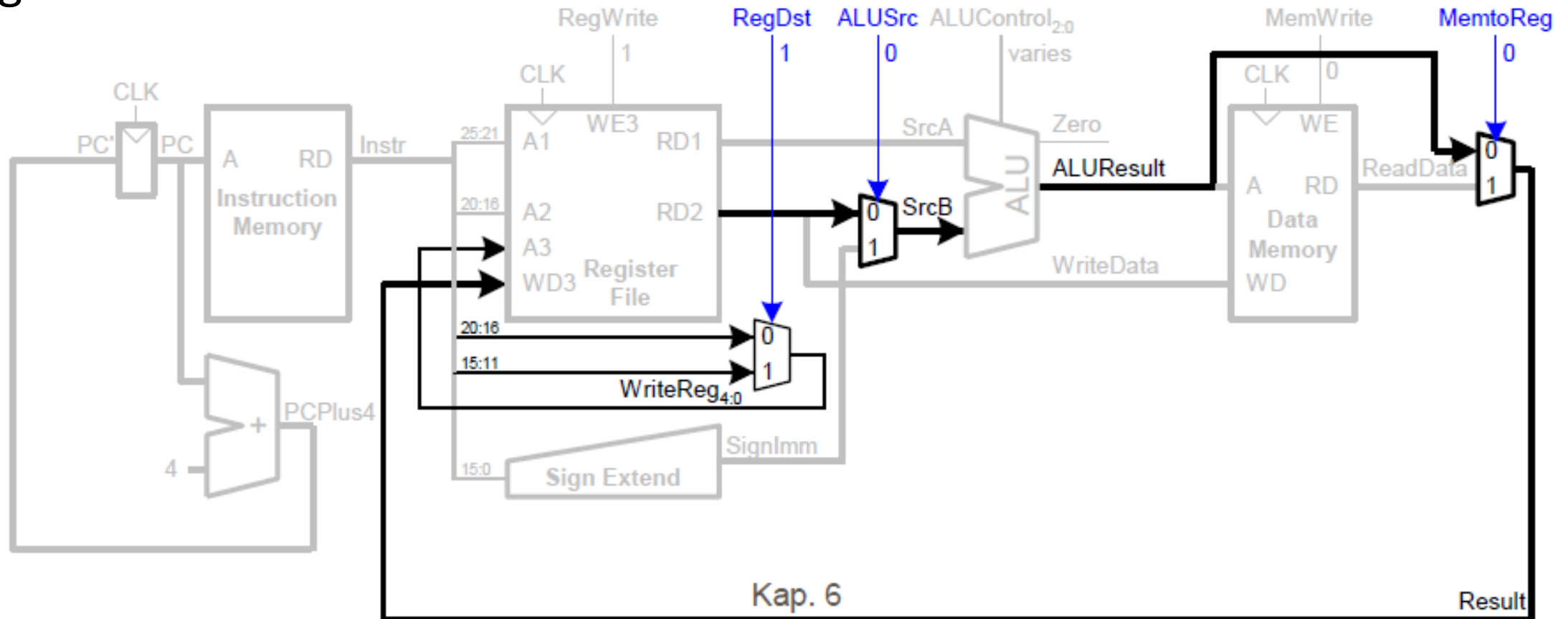
Prozessor

- PC erhöhen



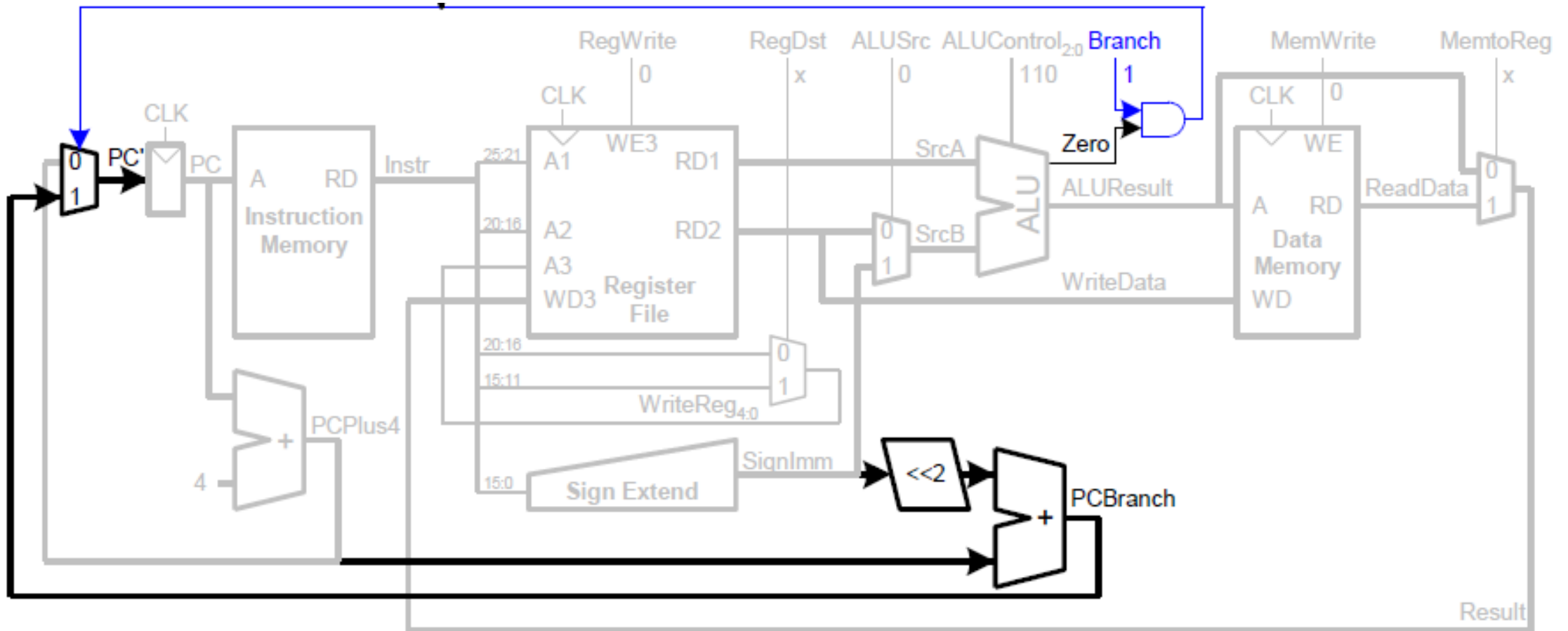
Prozessor

- zweites Register verwenden
- Ergebnis direkt abrufen



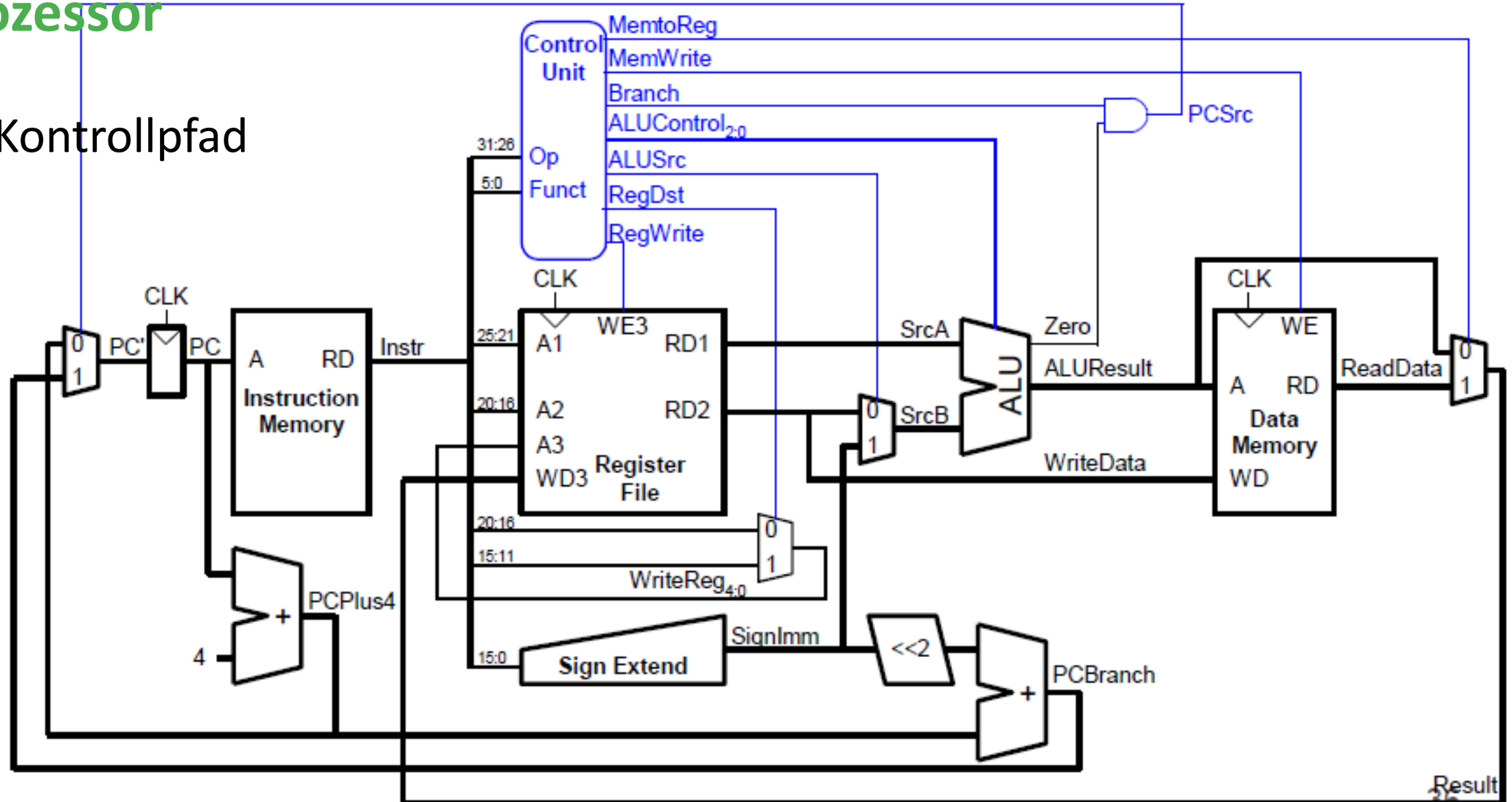
Prozessor

- Branch (an andere Programmstelle springen)



Prozessor

- Kontrollpfad



Pipelining

wie kann die Effizienz eines Prozessors erhöht werden?

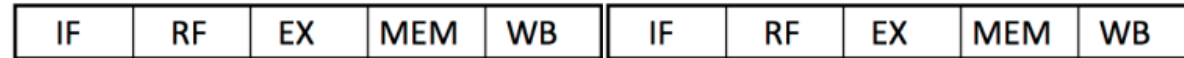
Pipelining

- 5 Ausführungsstufen:
 - Instruction-Fetch
 - Register-Fetch
 - Execute
 - Memory-Access
 - Write-Back

Pipelining

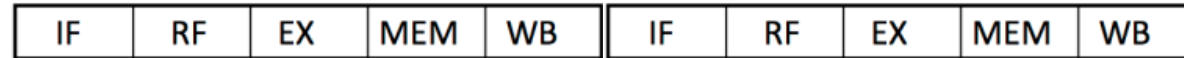
- 5 Ausführungsstufen:

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



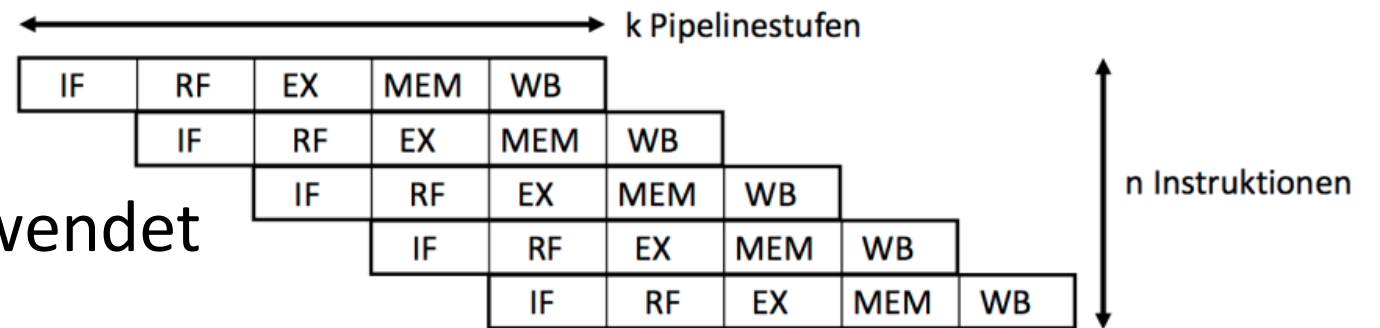
Pipelining

- 5 Ausführungsstufen:
 - Instruction-Fetch
 - Register-Fetch
 - Execute
 - Memory-Access
 - Write-Back
- Ressourcen werden nicht verwendet



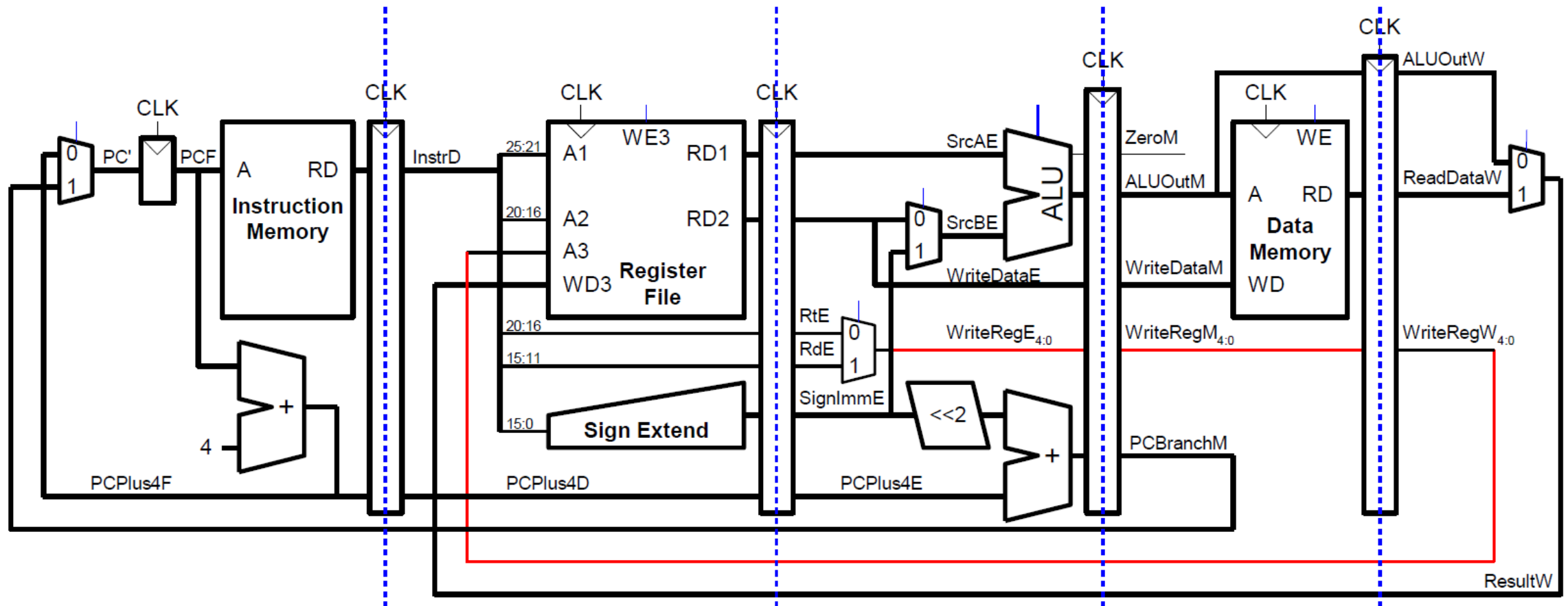
Pipelining

- 5 Ausführungsstufen:
 - Instruction-Fetch
 - Register-Fetch
 - Execute
 - Memory-Access
 - Write-Back
- Ressourcen werden nicht verwendet



Pipelining

- Register zur Speicherung benötigt



Pipelining

- Hazards beachten!!

Pipelining

- Hazards beachten!!
 - Read after write

Pipelining

- Hazards beachten!!
 - Read after write
 - Branch oder nicht?

Pipelining

- Hazards beachten!!
 - Read after write
 - Branch oder nicht?
- Lösungen:
 - no-operations einfügen

Pipelining

- Hazards beachten!!
 - Read after write
 - Branch oder nicht?
- Lösungen:
 - no-operations einfügen
 - einzelne Ergebnisse „vorschleusen“
 -

Pipelining

- Hazards beachten!!
 - Read after write
 - Branch oder nicht?
- Lösungen:
 - no-operations einfügen
 - einzelne Ergebnisse „vorschleusen“
 - branch-prediction

Fragen? Diskussion?

eventuell noch weitere Optimierungen

Literatur

- Alle Abbildungen: Vorlesung Systemarchitektur Uni Saarland, Jan Reineke
- Computer Organization and Design, The Hardware/Software Interface
- Computer Architecture: A Quantitative Approach
- Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture
- logische Schaltungen wurden mit www.digikey.de/schemeit gezeichnet