Grundlagen der Prozessorarchitektur und der Ein-/Ausgabe

Prof. Sarah Harris, Ph.D.
Fachgebiet Eingebettete Systeme und ihre Anwendungen (ESA)
Fachbereich Informatik

TECHNISCHE UNIVERSITÄT DARMSTADT

SS 16

Professorin Sarah Harris

TECHNISCHE UNIVERSITÄT DARMSTADT

- Technische Universität Darmstadt
  2015 – 2016

- **Fachgebiet:** Eingebettete Systeme, Rechnerarchitekturen

- **Kontakt Informationen:**
  Sprechstunden: Mittwochs, 13:30 Uhr – 14:30 Uhr
  Email: harris@esa.informatik.tu-darmstadt.de
  Büro: S2/02 (Piloty Gebäude), Raum E102

Wo ich herkomme

TECHNISCHE UNIVERSITÄT DARMSTADT

- Stanford University
  Ph.D. (≈Doktor)
  Electrical and Computer Engineering
  (2005)

- Harvey Mudd College
  Assistant/Associate Professorin
  (2004-2014)

- University of Nevada, Las Vegas
  Associate Professorin
  (2014 -)

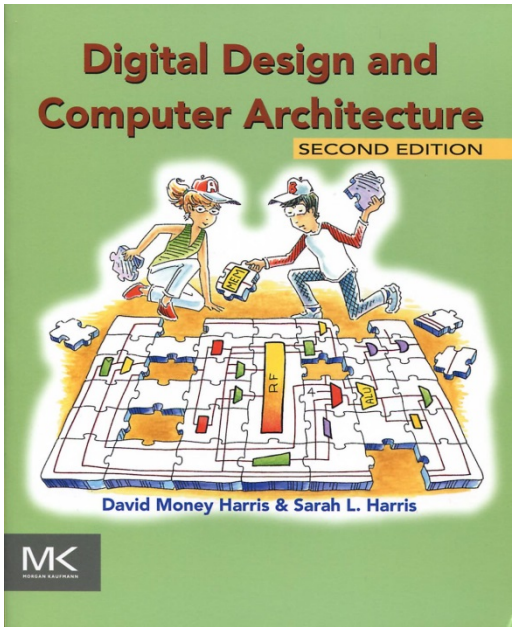
Embedded Systems & Applications

Lehr- und Anschauungsmaterial

TECHNISCHE UNIVERSITÄT DARMSTADT

- Aus dem Lehrbuch

Digital Design and Computer Architecture, zweite Auflage



Embedded Systems & Applications

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

Themen zu diesem Semester:

- **Architektur und Mikroarchitektur**
  - MIPS Processors
  - Pipelined Processor
- **Entwurfsmethoden für Rechnerarchitektur**
- **Entwurf mit HDL und Kommerziellen Werkzeuge:**
  - SystemVerilog
  - Xilinx's Vivado
  - Commercial MIPS Processor
  - u.s.w.
- **Entwickeln und Umgehen mit Ein- und Ausgabe Geräte**

Organisatorisch

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorlesungen:

- Donnerstags, 11:40 – 13:10 (Piloty C120)

Labzeiten (freiwillig):

- Dienstags, 11:30 – 13:00 Uhr (Piloty E104)

Assistent:

- Magnus Gärtner
- Sprechstunde: Di, 11:30-13:00 Uhr (E104)

Webseite:

- *http://www.esa.informatik.tu-darmstadt.de*

(klicke auf **Lehre** und dann **Grundlagen der Prozessorarchitektur and der Ein-/Ausgabe**)

Embedded Systems & Applications

Organisatorisch

TECHNISCHE UNIVERSITÄT DARMSTADT

- **TUCaN Anmeldefrist:** 21.04.16, 23:59
- **Moodle:** ich werde Sie nach dem TUCaN Anmelden einschreiben

Embedded Systems & Applications

Aufgaben

TECHNISCHE UNIVERSITÄT DARMSTADT

Nr.	Titel	Abgabefrist	Punkte
1	MIPS Assemblersprache	21. April	100
2	MIPS Eintaktprozessor	28. April	100
3	MIPS Pipeline-Prozessor	12. Mai	200
4	MIPSfpga: Tutorial	19. Mai	100
5	MIPSfpga: Memory-Mapped I/O – Rauschgerät	26. Mai	100
6	MIPSfpga: Memory-Mapped I/O – LCD	9. Juni	200
7	MIPSfpga: DMA Engine	23. Juni	200
8	MIPSfpga: DES Encryption	7. Juli	200

Embedded Systems & Applications

Aufgaben

TECHNISCHE UNIVERSITÄT DARMSTADT

- Durch Moodle eingereicht
- Abgabefrist: 10 Uhr
- Mit wenigstens 50% der insgesamt erreichbaren Punkte von den Aufgaben wird die Veranstaltung bestanden.

Prior Knowledge

TECHNISCHE UNIVERSITÄT DARMSTADT

- Verilog, SystemVerilog, other HDL?
- C?
- MIPS Architecture? Other Architecture?
- Embedded Systems?

MIPS Registerfeld

TECHNISCHE UNIVERSITÄT DARMSTADT

Name	Registernummer	Verwendungszweck
`$0`	0	Konstante Null
`$at`	1	Temporäre Variable für Assembler
`$v0-$v1`	2-3	Rückgabe von Werten aus Prozedur
`$a0-$a3`	4-7	Aufrufparameter in Prozedur
`$t0-$t7`	8-15	Temporäre Variablen
`$s0-$s7`	16-23	Gesicherte Variablen
`$t8-$t9`	24-25	Mehr temporäre Variablen
`$k0-$k1`	26-27	Temporäre Variablen für Betriebssystem
`$gp`	28	Zeiger auf globale Variablen im Speicher
`$sp`	29	Stapelzeiger im Speicher
`$fp`	30	Zeiger auf aktuellen Aufruf-Frame im Speicher
`$ra`	31	Rücksprungadresse aus Prozedur

Embedded Systems & Applications

Lesen aus byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

- Lesen geschieht durch Ladebefehle (*load)*
- Befehlsname: *load word* (`lw`)
- **Format:** `lw $t0, 8($s2)`

Embedded Systems & Applications

Lesen aus byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

- Lesen geschieht durch Ladebefehle (*load)*
- Befehlsname: *load word* (`lw`)
- **Format:** `lw $t0, 8($s2)`

Lese ein Datenwort von der Speicheradresse (`$s2` + 8) nach `$t0`

Adressarithmetik: Adressen werden relativ zu einem Register angegeben

- Basisadresse (`$s2`) plus Distanz (*offset*) (8)
- Adresse = (`$s2` + 8)

Ergebnis: `$t0` enthält das Datawort von Speicheradresse (`$s2` + 8)

Jedes Register darf als Basisadresse verwendet werden

Embedded Systems & Applications

Schreiben in byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

- Schreiben geschieht durch Speicherbefehle (*store)*
- Befehlsname: *store word* (`sw`)
- **Format:** `sw $t4, 0x1c($0)`

Beispiel: Schreibe (speichere) den Wert aus `$t4` in Speicherwort 7 (bzw. Speicheradrese 7x4=28 = 0x1C)

Adressarithmetik:

- Basisadresse (`$0`) plus Distanz (*offset*) (0x1c)
- Adresse = (`$0` + 28)

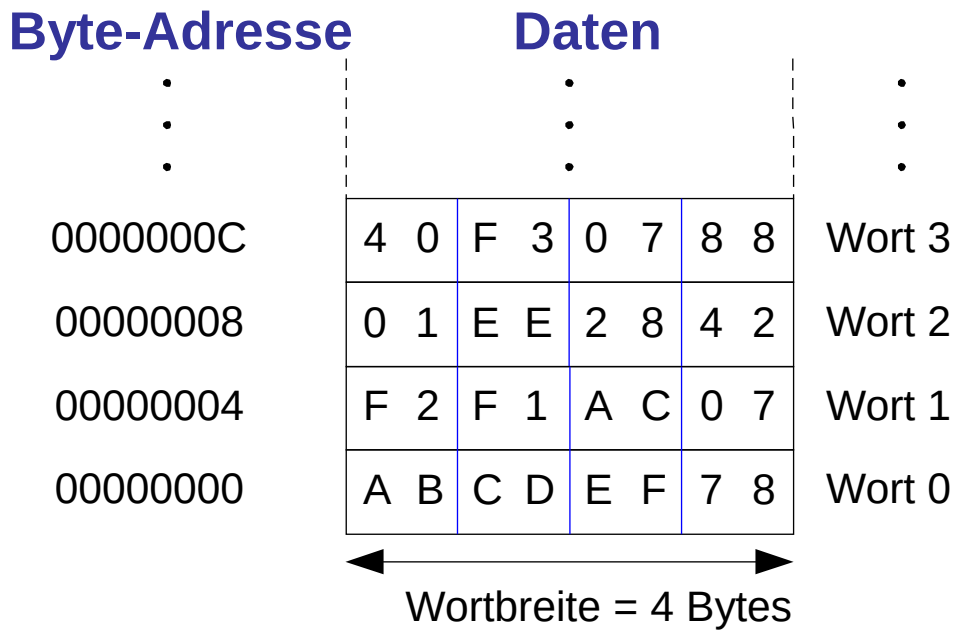
Ergebnis: Nach Abarbeiten des Befehls enthält Speicheradresse (`$0` + 28) das Datawort von `$t4`

Byte-addressierbarer Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

Speicherbefehle können auf Worten oder Bytes arbeiten

Worte: `lw` / `sw` **Bytes:** `lb` / `sb`



Embedded Systems & Applications

Speicherorganisation: Big-Endian und Little-Endian

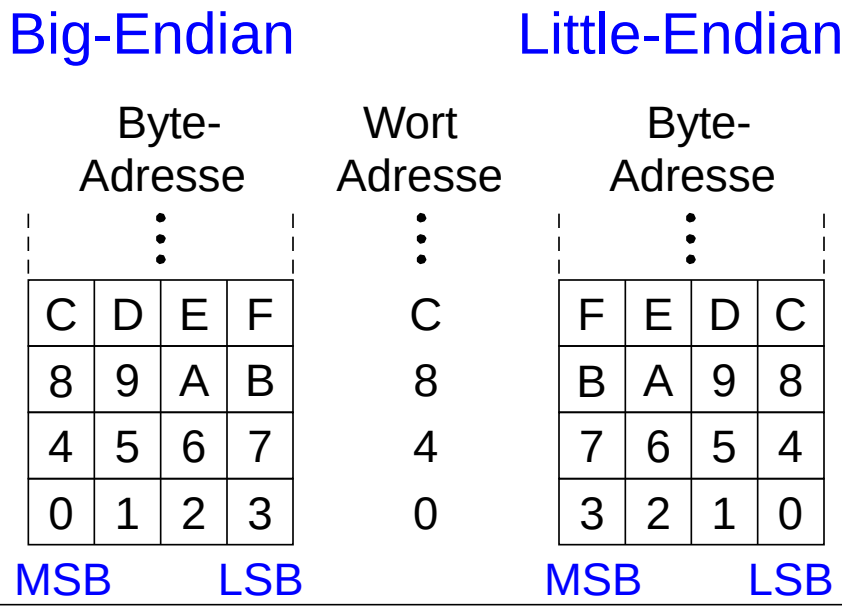
TECHNISCHE UNIVERSITÄT DARMSTADT

Schemata für Nummerierung von Bytes in einem Wort

Wort-Adresse ist bei beiden **gleich**

Little-endian: Bytes werden vom niederstwertigen Ende an gezählt

Big-endian: Bytes werden vom höchstwertigen Ende an gezählt



Embedded Systems & Applications

Beispiel: Big-Endian und Little-Endian

TECHNISCHE UNIVERSITÄT DARMSTADT

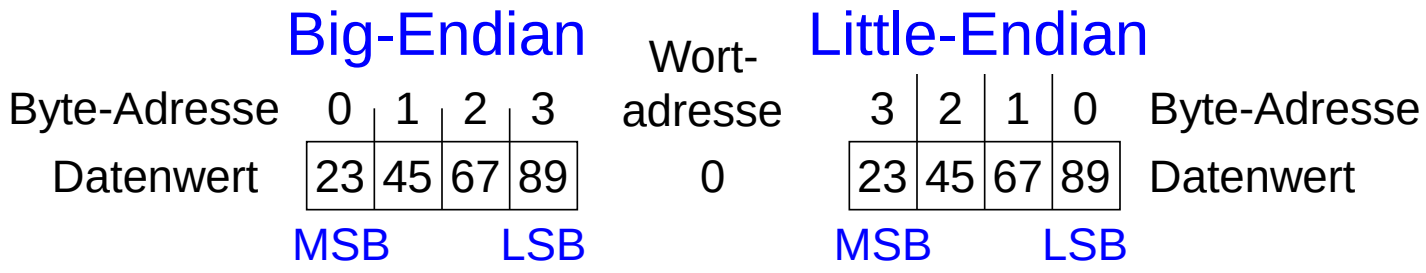
Annahme: `$t0` enthält den Wert 0x23456789

Programm:

```
sw $t0, 0($0)
lb $s0, 1($0)
```

Fragen: Welchen Wert hat `$s0` nach Ausführung auf einem…

- … Big-Endian Prozessor? 0x00000045
- … Little-Endian Prozessor? 0x00000067



Embedded Systems & Applications

Maschinensprache

TECHNISCHE UNIVERSITÄT DARMSTADT

- Computer verstehen nur 0'en und 1'en
- **Maschinensprache:** Binärdarstellung von Befehlen
- 32b Befehle
  - **Regularität vereinfacht Entwurf:** Daten und Befehle sind beides 32b Worte

Drei Befehlsformate

- R-Typ: Operanden sind nur Register
- I-Typ: Register und ein Direktwert
- J-Typ: für Programmsprünge (kommt noch)

Übersicht über Befehlsformate

TECHNISCHE UNIVERSITÄT DARMSTADT

R-Typ

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Typ

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Typ

op	addr
6 bits	26 bits

Embedded Systems & Applications

Beispiele für Befehle vom R-Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

Assemblersprache

```
add $s0, $s1, $s2

sub $t0, $t3, $t5
```

Felder in Befehlswort

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Maschinensprache

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Beachte andere Reihenfolge der Register in Assembler-Sprache:

```
add rd, rs, rt
```


Embedded Systems & Applications

Beispiel für Befehle vom I-Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

Assemblersprache

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1,  4($t1)
```

Felder im Befehlswort

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

Maschinensprache

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

Embedded Systems & Applications

Beispiel für Befehle vom I-Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

Assemblersprache

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1,  4($t1)
```

Felder im Befehlswort

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

Beachte unterschiedliche Reihenfolge von Registern in Assembler- und Maschinensprache

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

Maschinensprache

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

Embedded Systems & Applications

Adressierungsarten

TECHNISCHE UNIVERSITÄT DARMSTADT

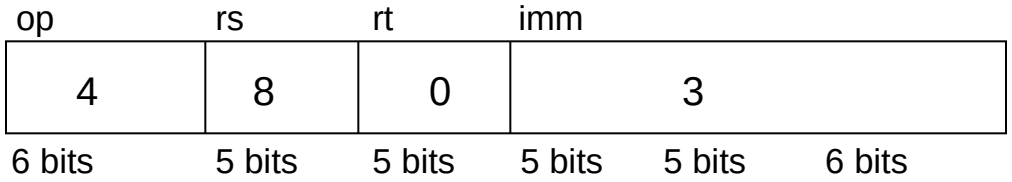
Relativ zur nächsten Adresse im Programmzähler

```
0x10              beq   $t0, $0, else
0x14              addi  $v0, $0, 1
0x18              addi  $sp, $sp, i
0x1C              jr          $ra
0x20      else:   addi  $a0, $a0, -1
0x24              jal   fakultaet
```

Assemblersprache

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

Bitfelder in Instruktion



Embedded Systems & Applications

Befehlsformat J-Typ (Jump Typ)

TECHNISCHE UNIVERSITÄT DARMSTADT

- 26b Adressoperand (`addr`)
- Verwendet für Sprungbefehle (`j`)

J-Typ

Embedded Systems & Applications

Adressierungsarten

TECHNISCHE UNIVERSITÄT DARMSTADT

Pseudodirekte Operanden

Auffüllen von entfallenen Bits (mit Nullen und (PC+4)[31:28])

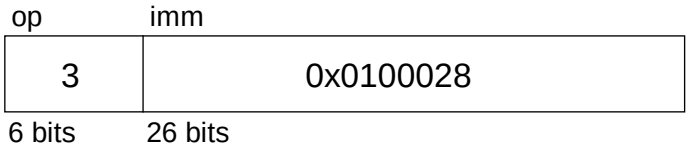
```
0x0040005C          jal   sum
...
0x004000A0  sum:    add   $v0, $a0, $a1
```

32b Sprungzieladresse 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

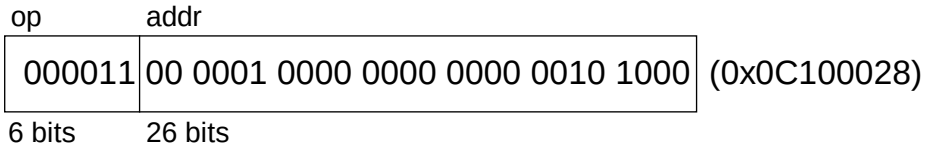
26b Feld in J-Instruktion 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

Bitfelder in Instruktion



Maschinencode



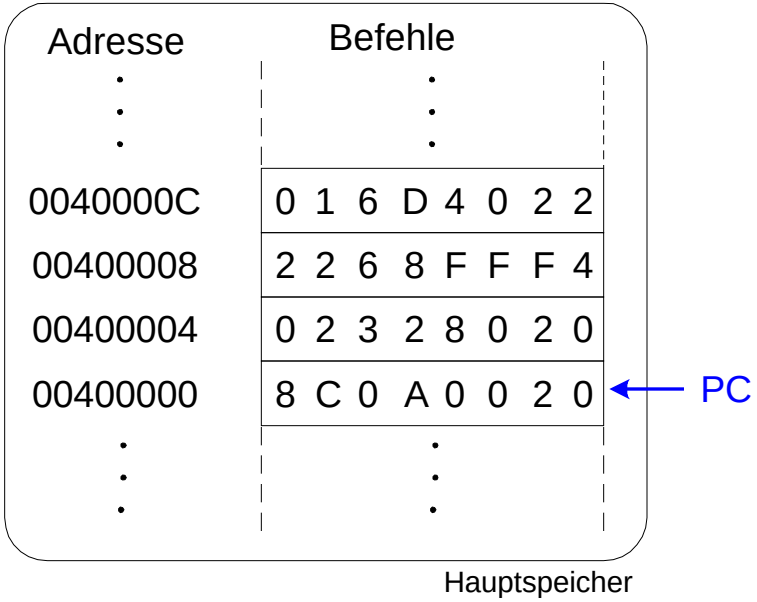
Embedded Systems & Applications

Im Speicher abgelegtes Programm

TECHNISCHE UNIVERSITÄT DARMSTADT

Assemblersprache	Maschinensprache
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

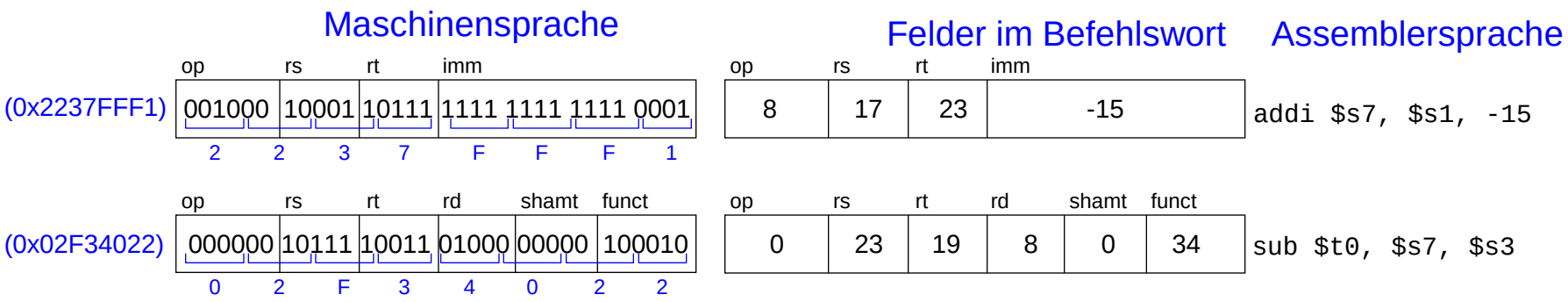
Programm im Speicher



Embedded Systems & Applications

Maschinensprache verstehen

- Beginn mit Entschlüsseln des Opcodes
- Opcode bestimmt Bedeutung der anderen Bits
- Wenn Opcode Null ist
  - … liegt ein Befehl im R-Format vor
  - Die Operation wird durch das Funktionsfeld bestimmt
- Sonst
  - Bestimmt Opcode alleine die Operation, siehe Anhang B im Buch



Programmierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Hochsprachen:

- z.B. C, Java, Python, Scheme
- Auf einer abstrakteren Ebene programmieren

Häufige Konstrukte in Hochsprachen:

- if/else-Anweisungen
- for-Schleifen
- while-Schleifen
- Feld (Array) zugriffe
- Prozeduraufrufe\

Andere nützliche Anweisungen:

- Arithmetische/logische Ausdrücke
- Verzweigungen

Logische Befehle

TECHNISCHE UNIVERSITÄT DARMSTADT

and, or, xor, nor

- and: nützlich zum Maskieren von Bits

  Beispiel: Ausmaskieren aller Bits außer dem LSB:

  0xF234012F AND 0x000000FF = 0x0000002F

- or: Nützlich zum Vereinigen von Bitfeldern

  Beispiel: Vereinige 0xF2340000 mit 0x000012BC:

  0xF2340000 OR 0x000012BC = 0xF23412BC

- nor: nützlich zur Invertierung von Bits:

  Beispiel: A NOR $0 = NOT A

Embedded Systems & Applications

Logische Befehle mit Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

`andi`, `ori`, `xori`

- 16-bit Direktwert wird erweitert mit führenden **Nullbits** (nicht *vorzeichenerweitert*)
- `nori` wird nicht benötigt

Embedded Systems & Applications

Beispiele: Logische Befehle

TECHNISCHE UNIVERSITÄT DARMSTADT

Quellregister

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assemblersprache

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Ergebnisse

$s3								
$s4								
$s5								
$s6								

Embedded Systems & Applications

Beispiele: Logische Befehle

TECHNISCHE UNIVERSITÄT DARMSTADT

Quellregister

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assemblersprache		Ergebnisse							
`and $s3, $s1, $s2`	$s3	0100	0110	1010	0001	0000	0000	0000	0000
`or $s4, $s1, $s2`	$s4	1111	1111	1111	1111	1111	0000	1011	0111
`xor $s5, $s1, $s2`	$s5	1011	1001	0101	1110	1111	0000	1011	0111
`nor $s6, $s1, $s2`	$s6	0000	0000	0000	0000	0000	1111	0100	1000

Embedded Systems & Applications

Beispiele: Logische Befehle

Operanden

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

Null-erweitert

Assemblersprache

```
andi $s2, $s1, 0xFA34
ori  $s3, $s1, 0xFA34
xori $s4, $s1, 0xFA34
```

Ergebnisse

$s2								
$s3								
$s4								

Beispiele: Logische Befehle

Operanden

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

Null-erweitert

Assemblersprache		Ergebnisse							
`andi $s2, $s1, 0xFA34`	$s2	0000	0000	0000	0000	0000	0000	0011	0100
`ori $s3, $s1, 0xFA34`	$s3	0000	0000	0000	0000	1111	1010	1111	1111
`xori $s4, $s1, 0xFA34`	$s4	0000	0000	0000	0000	1111	1010	1100	1011

Schiebebefehle

TECHNISCHE UNIVERSITÄT DARMSTADT

sll: shift left logical

- **Beispiel:**
```
sll $t0, $t1, 5
        # $t0 <= $t1 << 5
```

srl: shift right logical

- **Beispiel:**
```
srl $t0, $t1, 5
        # $t0 <= $t1 >> 5
```

sra: shift right arithmetic

- **Beispiel:**
```
sra $t0, $t1, 5
        # $t0 <= $t1 >>> 5
```

Schiebebefehle

TECHNISCHE UNIVERSITÄT DARMSTADT

Schieben mit variabler Distanz:

`sllv:` shift left logical variable

- **Beispiel :**
```
sllv $t0, $t1, $t2
# $t0 <= $t1 << $t2
```

`srlv:` shift right logical variable

- **Beispiel :**
```
srlv $t0, $t1, $t2
# $t0 <= $t1 >> $t2
```

`srav:` shift right arithmetic variable

- **Beispiel** :
```
srav $t0, $t1, $t2
# $t0 <= $t1 >>> $t2
```

Schiebebefehle

TECHNISCHE UNIVERSITÄT DARMSTADT

Assemblersprache

```
sll $t0, $s1, 2
srl $s2, $s1, 2
sra $s3, $s1, 2
```

Felder in Instruktion

	op	rs	rt	rd	shamt	funct
sll $t0, $s1, 2	0	0	17	8	2	0
srl $s2, $s1, 2	0	0	17	18	2	2
sra $s3, $s1, 2	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Maschinensprache

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Embedded Systems & Applications

Handhabung von Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

16-Bit Konstante mit `addi`:

Hochsprache	**MIPS Assemblersprache**
`// int ist ein`	`# $s0 = a`
`// vorzeichenbehaftetes`	
`// 32b Wort`	
`int a = 0x4f3c;`	`addi $s0, $0, 0x4f3c`

32-Bit Konstante mit Load Upper Immediate (`lui`) und `ori`:

(`lui` lädt den 16-Bit Direktwert in obere Registerhälfte und setzt untere Hälfte auf 0.)

Hochsprache	**MIPS Assemblersprache**
	`# $s0 = a`
`int a = 0xFEDC8765;`	`lui $s0, 0xFEDC`
	`ori $s0, $s0, 0x8765`

Embedded Systems & Applications

Multiplikation und Division

TECHNISCHE UNIVERSITÄT DARMSTADT

Spezialregister: `lo`, `hi`

- 32b × 32b Multiplikation, 64b Produkt

  ```
  mult $s0, $s1
  ```

  Ergebnis in {`hi`, `lo`}

- 32b Division, 32b Quotient, 32b Rest

  ```
  div $s0, $s1
  ```

  Quotient in `lo`

  Rest in `hi`

- Lesen von Daten aus Spezialregistern („*move from …*“)

  ```
  mflo $s2
  mfhi $s3
  ```


Embedded Systems & Applications

Verzweigungen und Sprünge

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ändern der Ausführungsreihenfolge von Befehlen
- Arten von Verzweigungen:
  - **Bedingte**
    - branch if equal (`beq`): Verzweige, wenn gleich
    - branch if not equal (`bne`): Verzweige, wenn ungleich
  - **Unbedingte Verzweigungen**
    - jump (`j`): Springe
    - jump register (`jr`): Springe auf Adresse aus Register
    - jump and link (`jal`): Springe und merke Adresse des nächsten Befehls

Embedded Systems & Applications

Bedingte Verzweigungen (beq)

TECHNISCHE UNIVERSITÄT DARMSTADT

MIPS Assemblersprache

```
 addi $s0, $0, 4              # $s0 = 0 + 4 = 4
 addi $s1, $0, 1              # $s1 = 0 + 1 = 1
 sll  $s1, $s1, 2             # $s1 = 1 << 2 = 4
 beq  $s0, $s1, target        # Verzweigung wird genommen
 addi $s1, $s1, 1             # nicht ausgeführt
 sub  $s1, $s1, $s0           # nicht ausgeführt

target:                       # Positionsmarkierung (label)
 add  $s1, $s1, $s0           # $s1 = 4 + 4 = 8
```

Label sind Namen für Stellen (Adressen) im Programm. Sie müssen anders als Mnemonics heißen und haben einen Doppelpunkt am Ende.

Embedded Systems & Applications

Nicht genommene Sprünge (bne)

TECHNISCHE UNIVERSITÄT DARMSTADT

MIPS Assemblersprache

```
  addi  $s0, $0, 4              # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1              # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2             # $s1 = 1 << 2 = 4
  bne   $s0, $s1, target        # Verzweigung nicht genommen
  addi  $s1, $s1, 1             # $s1 = 4 + 1 = 5
  sub   $s1, $s1, $s0           # $s1 = 5 – 4 = 1

target:
  add   $s1, $s1, $s0           # $s1 = 1 + 4 = 5
```


Embedded Systems & Applications

Unbedingte Verzweigungen / Springen (j)

TECHNISCHE UNIVERSITÄT DARMSTADT

MIPS Assemblersprache

```
  addi $s0, $0, 4              # $s0 = 4
  addi $s1, $0, 1              # $s1 = 1
  j    target                  # Springe zu target
  sra  $s1, $s1, 2             # nicht ausgeführt
  addi $s1, $s1, 1             # nicht ausgeführt
  sub  $s1, $s1, $s0           # nicht ausgeführt

target:
  add  $s1, $s1, $s0           # $s1 = 1 + 4 = 5
```


Embedded Systems & Applications

Unbedingte Verzweigungen (jr)

TECHNISCHE UNIVERSITÄT DARMSTADT

MIPS Assemblersprache

```
0x00002000        addi $s0, $0, 0x2010
0x00002004        jr   $s0
0x00002008        addi $s1, $0, 1
0x0000200C        sra  $s1, $s1, 2
0x00002010        lw   $s3, 44($s1)
```

Konstrukte in Hochsprachen

TECHNISCHE UNIVERSITÄT DARMSTADT

- `if`-Anweisungen
- `if/else`-Anweisungen
- `while`-Schleifen
- `for`-Schleifen

If-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

Hochsprache

```
if (i == j)
  f = g + h;

f = f – i;
```

MIPS Assemblersprache

```
# $s0=f, $s1=g, $s2=h
# $s3=i, $s4=j
    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

Beachte: Im Assembler wird auf **entgegengesetzte** Bedingung geprüft (`i != j`) als in der Hochsprache (`i == j`).

Embedded Systems & Applications

If / Else-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

Hochsprache

```
if (i == j)
  f = g + h;
else
  f = f – i;
```

MIPS Assemblersprache

```
# $s0=f, $s1=g, $s2=h
# $s3=i, $s4=j
      bne $s3, $s4, L1
      add $s0, $s1, $s2
      j   done
L1:   sub $s0, $s0, $s3
done:
```


Embedded Systems & Applications

While-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

Hochsprache

```
// berechnet
// x = log2(128)

int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS Assemblersprache

```
# $s0 = pow, $s1 = x

        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128
while:  beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j    while
done:
```

Auch hier: Assemblersprache prüft auf **entgegengesetzte** Bedingung (`pow == 128`) als Hochsprache (`pow != 128`).

Embedded Systems & Applications

For-Schleifen

TECHNISCHE UNIVERSITÄT DARMSTADT

Hochsprache

```
// addiere Zahlen
// von 0 to 9 auf
int sum = 0;
int i;

for (i = 0; i != 10; i = i+1) {
  sum = sum + i;
}
```

MIPS Assemblersprache

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

Auch hier: Prüfen auf **entgegengesetzte** Bedingung in Assemblersprache (`i == 10`) als in Hochsprache (`i != 10`).

Embedded Systems & Applications

Kleiner-als Vergleiche

TECHNISCHE UNIVERSITÄT DARMSTADT

Hochsprache

```
// addiere Zweierpotenzen
// kleiner gleich  100

int sum = 0;
int i;

for (i=1; i<101; i=i*2) {
  sum = sum + i;
}
```

MIPS Assemblersprache

```
# $s0 = i, $s1 = sum

        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

$t1 = 1 if i < 101.

Embedded Systems & Applications

Bearbeite Array in for-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Hochsprache
  int array[1000];
  int i;

  for (i=0; i < 1000; i = i + 1)
            array[i] = array[i] * 8;
```

```
# MIPS Assemblersprache
# $s0 = Basisadresse von Array, $s1 = i

# Initialisierung
   lui  $s0, 0x23B8        # $s0 = 0x23B80000
   ori  $s0, $s0, 0xF000   # $s0 = 0x23B8F000
   addi $s1, $0, 0         # i = 0
   addi $t2, $0, 1000      # $t2 = 1000
loop:
   slt  $t0, $s1, $t2      # i < 1000?
   beq  $t0, $0, done      # if not then done
   sll  $t0, $s1, 2        # $t0 = i * 4 (byte offset)
   add  $t0, $t0, $s0      # address of array[i]
   lw   $t1, 0($t0)        # $t1 = array[i]
   sll  $t1, $t1, 3        # $t1 = array[i] * 8
   sw   $t1, 0($t0)        # array[i] = array[i] * 8
   addi $s1, $s1, 1        # i = i + 1
   j    loop               # repeat
done:
```


Embedded Systems & Applications

Prozedur- und Funktionsaufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

Definitionen

- Aufrufer: Ursprung des Prozeduraufrufs (hier `main`)
- Aufgerufener: aufgerufene Prozedur (hier `sum`)

Hochsprache

```
void main()
{
  int y;
  y = sum (42, 7);
  ...
}

int sum (int a, int b)
{
  return (a + b);
}
```

Prozedur- und Funktionsaufruf Konventionen für MIPS

TECHNISCHE UNIVERSITÄT DARMSTADT

- Prozeduraufruf: “jump and link (`jal`)“
- Rücksprung: “jump register (`jr`)”
- Register für Argumente: `$a0 - $a3`
- Register für Ergebnis: `$v0`

Prozedur- und Funktionsaufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

Hochsprache

```
int main() {
  simple ();
  a = b + c;
}

void simple () {
  return;
}
```

MIPS Assemblersprache

```
0x00400200 main: jal  simple
0x00400204       add  $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

`void` bedeutet, dass `simple` keinen Rückgabewert hat.

- Also eine Prozedur und keine Funktion ist

Embedded Systems & Applications

Prozedur- und Funktionsaufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

Hochsprache

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

MIPS Assemblersprache

```
0x00400200 main: jal  simple
0x00400204       add  $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

jal: springt zu simple
speichert PC+4 im Spezialregister $ra “return address register”
Hier: $ra = 0x00400204 nach Ausführung von jal

jr $ra: springt zur Adresse in $ra, hier also 0x00400204.

Embedded Systems & Applications

Aufrufargumente und Rückgabewert

TECHNISCHE UNIVERSITÄT DARMSTADT

MIPS Konventionen:

- Argumentwerte (aktuelle Parameter): `$a0` - `$a3`
- Rückgabewert (Funktionswert, Ergebnis): `$v0`

Aufrufargumente und Rückgabewert

TECHNISCHE UNIVERSITÄT DARMSTADT

Hochsprache

```
int main()
{
  int y;
  ...
  y = diffofsums (2, 3, 4, 5);  // 4 Argumente
  ...
}

int diffofsums (int f, int g, int h, int i)
// 4 formale Parameter
{
  int result;
  result = (f + g) - (h + i);
  return result;                  // Rückgabewert
}
```


Embedded Systems & Applications

Aufrufargumente und Rückgabewert

TECHNISCHE UNIVERSITÄT DARMSTADT

MIPS Assemblersprache

```
# $s0 = y

main:
  ...
  addi $a0, $0, 2    # Argument 0 = 2
  addi $a1, $0, 3    # Argument 1 = 3
  addi $a2, $0, 4    # Argument 2 = 4
  addi $a3, $0, 5    # Argument 3 = 5
  jal  diffofsums    # Prozeduraufruf
  add  $s0, $v0, $0  # y = Rückgabewert
  ...

# $s0 = Rückgabewert
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # Lege Rückgabewert in $v0 ab
  jr  $ra            # Rücksprung zum Aufrufer
```


Embedded Systems & Applications

Aufrufargumente und Rückgabewert

TECHNISCHE UNIVERSITÄT DARMSTADT

MIPS Assemblersprache

```
# $s0 = result
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # Lege Rückgabewert in $v0 ab
  jr  $ra            # Rücksprung zum Aufrufer
```

- `diffofsums` überschreibt drei Register: `$t0`, `$t1` und `$s0`
- `diffofsums` kann benötigte Register temporär auf Stack sichern

Sicherungskonventionen für Register

TECHNISCHE UNIVERSITÄT DARMSTADT

Erhalten *Gesichert vom Aufgerufenen*	Nicht erhalten *Gesichert vom Aufrufer*
`$s0 - $s7`	`$t0 - $t9`
`$ra`	`$a0 - $a3`
`$sp`	`$v0 - $v1`
Stack oberhalb von `$sp`	Stack unterhalb von `$sp`

Embedded Systems & Applications

Mehrfache Prozeduraufrufe: Sichern von $ra

TECHNISCHE UNIVERSITÄT DARMSTADT

```
proc1:
  addi $sp, $sp, -4    # Platz auf Stack anlegen
  sw   $ra, 0($sp)     # sichere $ra auf Stack
  jal  proc2
  ...
  lw   $ra, 0($sp)     # stelle $ra vom Stack wieder her
  addi $sp, $sp, 4     # Stapelspeicher wieder freigeben
  jr  $ra              # Rückkehr zum Aufrufer von proc1

proc2:
   ...
```


Embedded Systems & Applications

Erhalten von Registern mittels Stack

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4  # Platz auf Stack für 4 Bytes anlegen
                     # reicht zum Sichern eines Registers
  sw  $s0, 0($sp)    # sichere $s0 auf Stack
                     # $t0 und $t1 brauchen nicht erhalten zu werden!
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # Lege Rückgabewert in $v0 ab
  lw  $s0, 0($sp)    # stelle $s0 vom Stack wieder her
  addi $sp, $sp, 4   # Gebe nicht mehr benötigten Speicher auf Stack frei
  jr  $ra            # Rücksprung zum Aufrufer
```


Embedded Systems & Applications

Zusammenfassung: Prozeduraufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

Aufrufer

- Lege Aufrufparameter (aktuelle Parameter) in `$a0-$a3` ab
- Sichere zusätzlich benötigte Register auf Stack (`$ra`, manchmal auch `$t0-t9`) - entsprechend Konvention über Erhaltung von Registern
- `jal aufgerufener`
- Stelle gesicherte Register wieder her
- Hole evtl. Rückgabewert aus `$v0` (bei Funktionen)

Aufgerufener

- Sichere zu erhaltende verwendete Register auf Stack (üblicherweise `$s0-$s7`)
- Führe Berechnungen der Prozedur aus
- Lege Rückgabewert in ab `$v0` (bei Funktionen)
- Stelle gesicherte Register wieder her
- `jr $ra`

Embedded Systems & Applications

Mikroarchitektur

TECHNISCHE UNIVERSITÄT DARMSTADT

Mehrere Implementierungen für eine Architektur

- **Ein-Takt**

  Jede Instruktion wird in einem Takt ausgeführt

- **Pipelined**

  Jede Instruktion wird in Teilschritte zerlegt

  Mehrere Instruktionen werden gleichzeitig ausgeführt

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

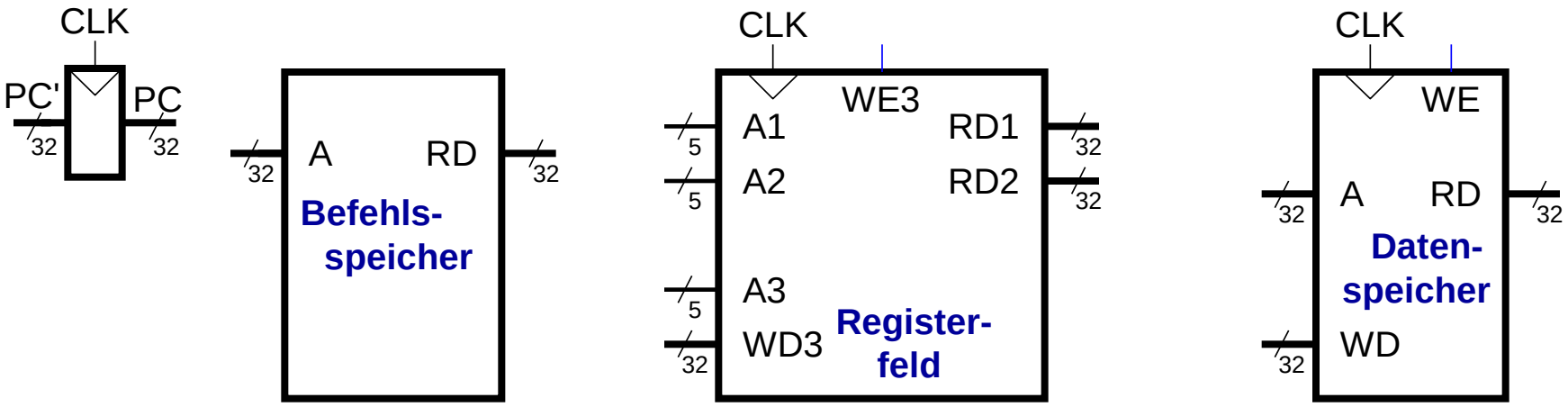
- Für den Programmierer zugänglich

Bestimmen vollständigen Zustand der Architektur

- PC
- 32 Register
- Speicher

Embedded Systems & Applications

Elemente des MIPS Architekturzustands



Ein-Takt MIPS Prozessor

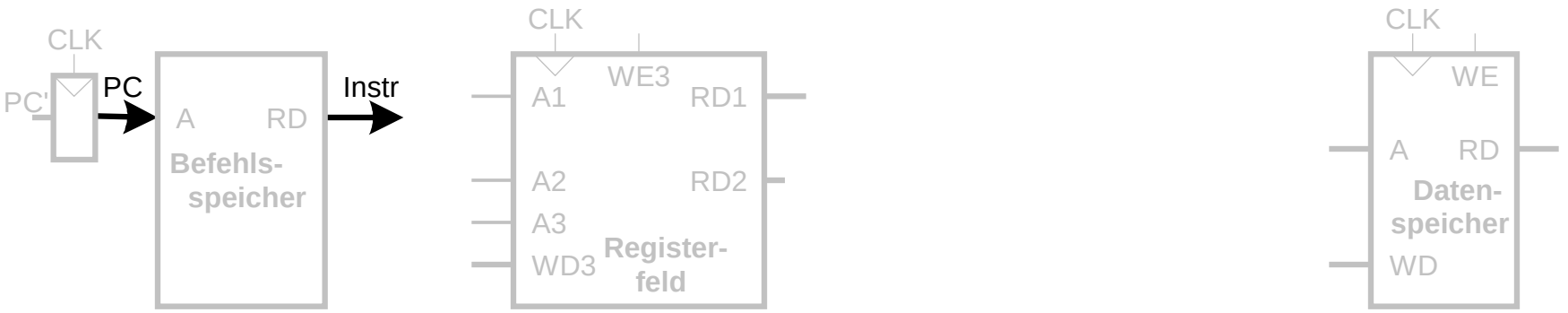
TECHNISCHE UNIVERSITÄT DARMSTADT

- Datenpfad
- Steuerwerk

Ein-Takt Datenpfad: Holen eines `lw` Befehls

Ein *load word* Befehl (`lw`) soll ausgeführt werden

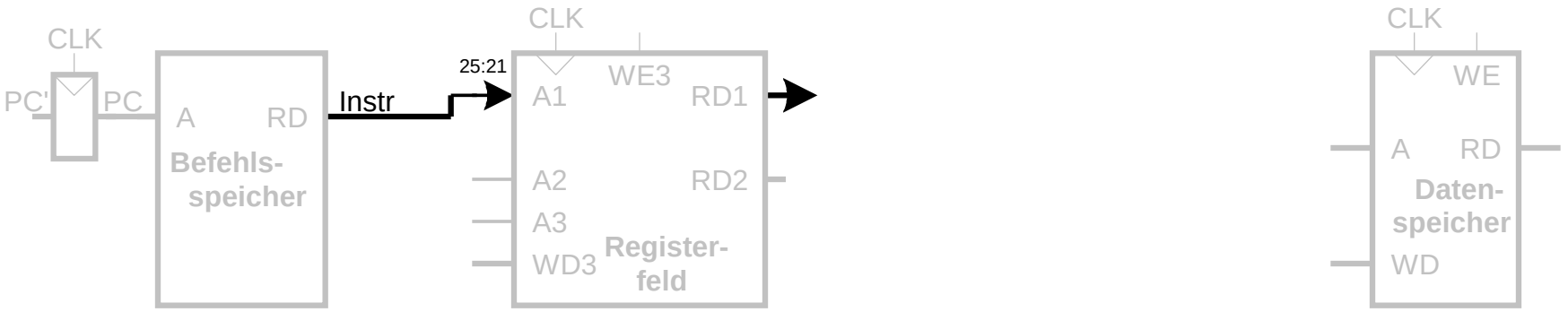
Schritt 1: Hole Instruktion



Ein-Takt Datenpfad: Lesen des Registers für `lw`

TECHNISCHE UNIVERSITÄT DARMSTADT

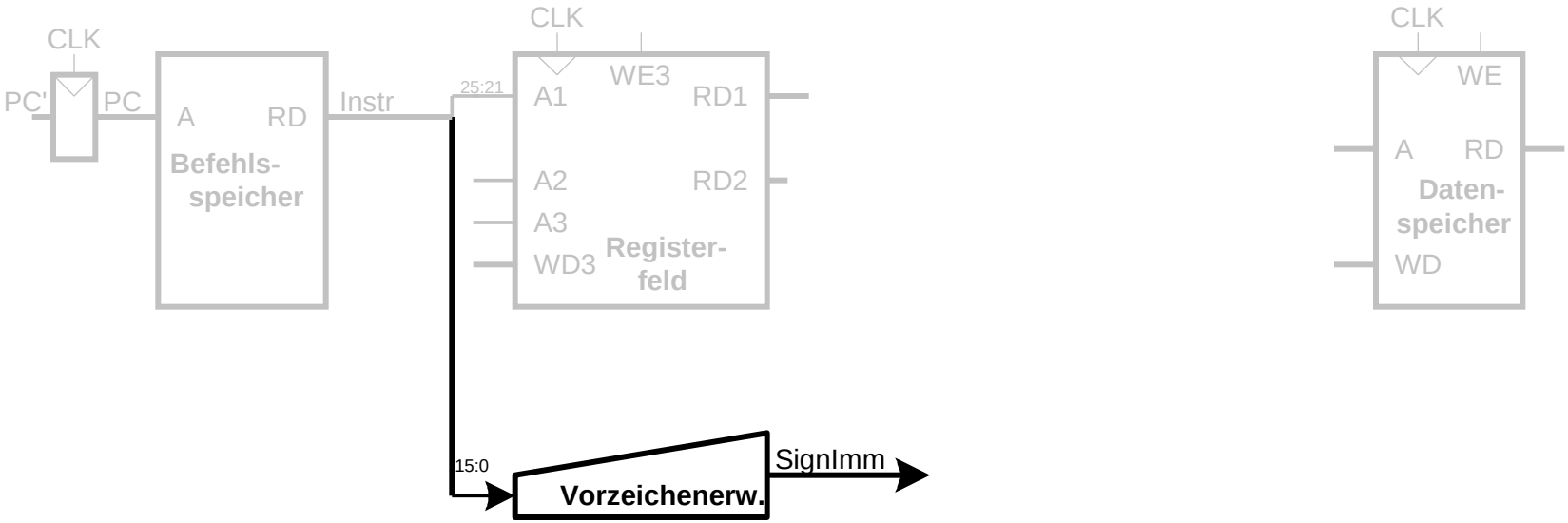
Schritt 2: Lese Quelloperand aus Registerfeld



Embedded Systems & Applications

Ein-Takt Datenpfad: Behandle `lw` Direktwert

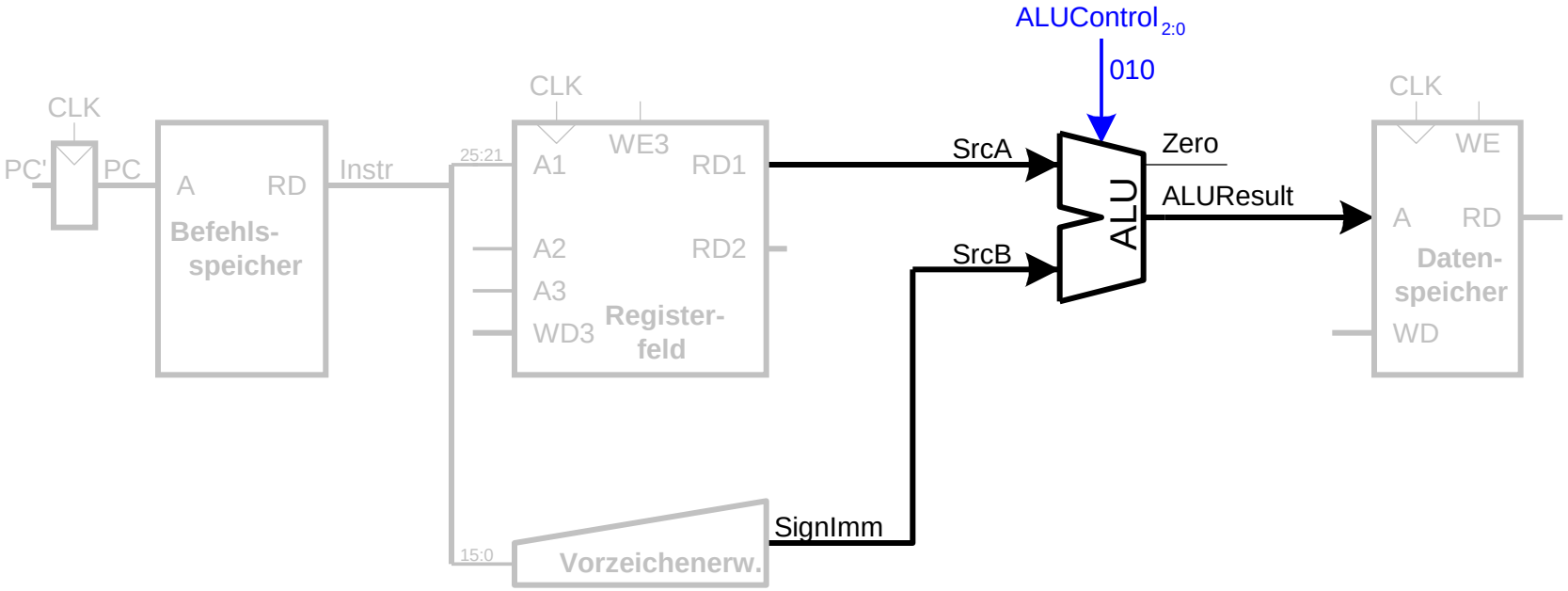
Schritt 3: Vorzeichenerweitere den 16b Direktwert auf 32b Signal `SignImm`



Ein-Takt Datenpfad: Berechne `lw` Zieladdresse

TECHNISCHE UNIVERSITÄT DARMSTADT

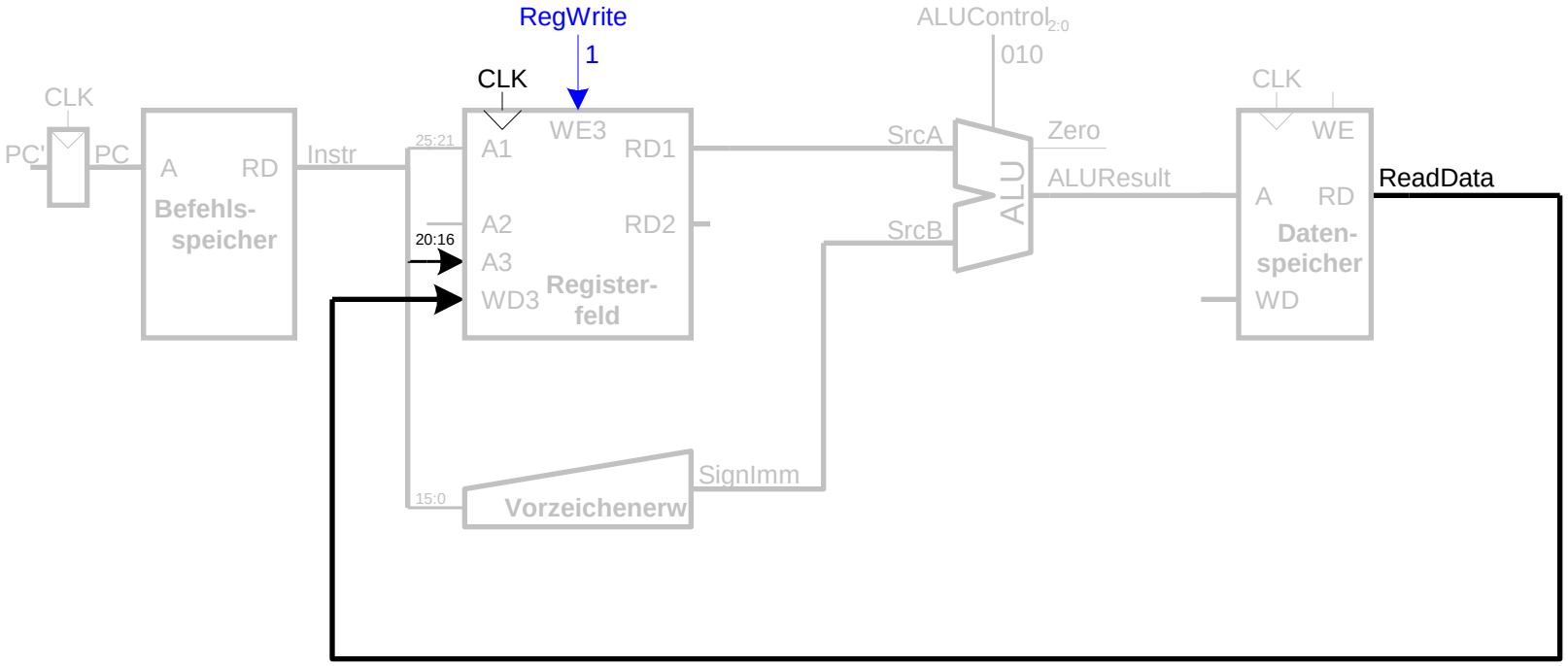
Schritt 4: Berechne die effektive Speicheradresse



Embedded Systems & Applications

Ein-Takt Datenpfad: Lese Speicher mit `lw`

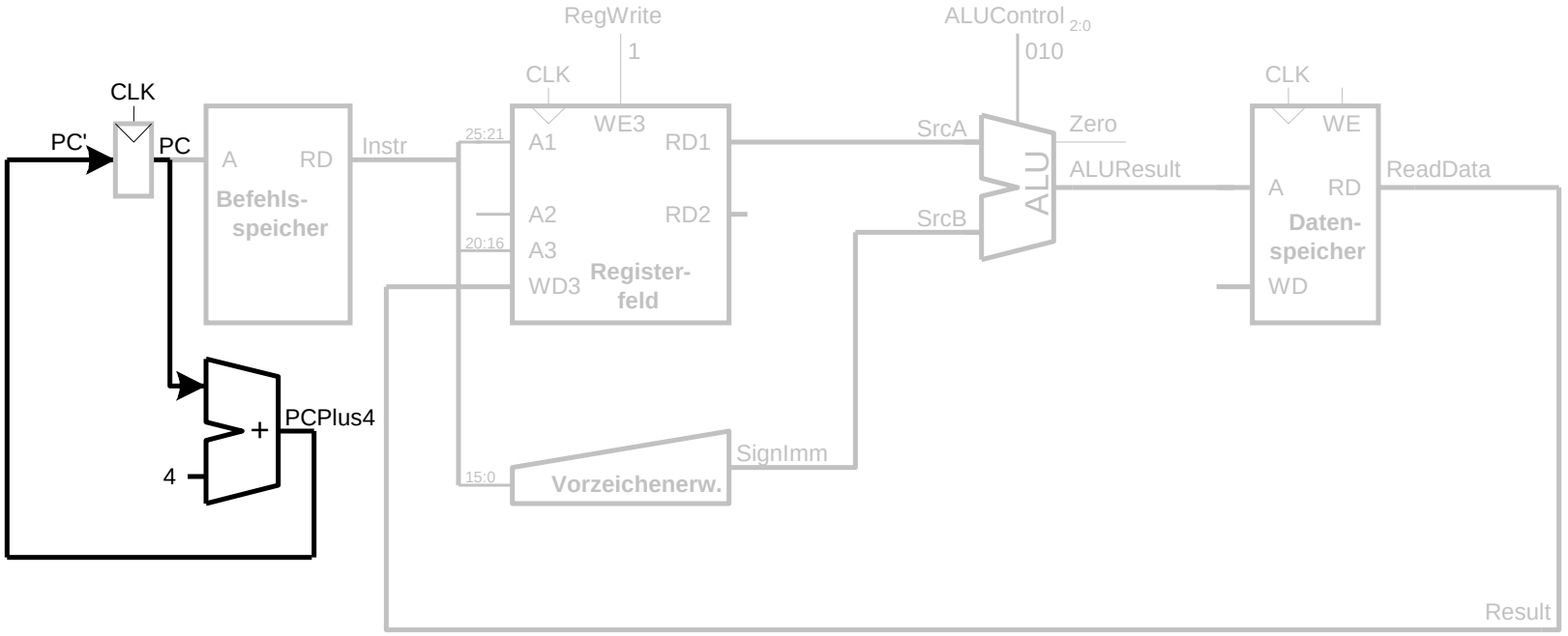
Schritt 5: Lese Daten aus Speicher und schreibe sie ins passende Register



Ein-Takt Datenpfad : Erhöhe PC nach `lw`

TECHNISCHE UNIVERSITÄT DARMSTADT

Schritt 6: Bestimme Adresse des nächsten Befehls

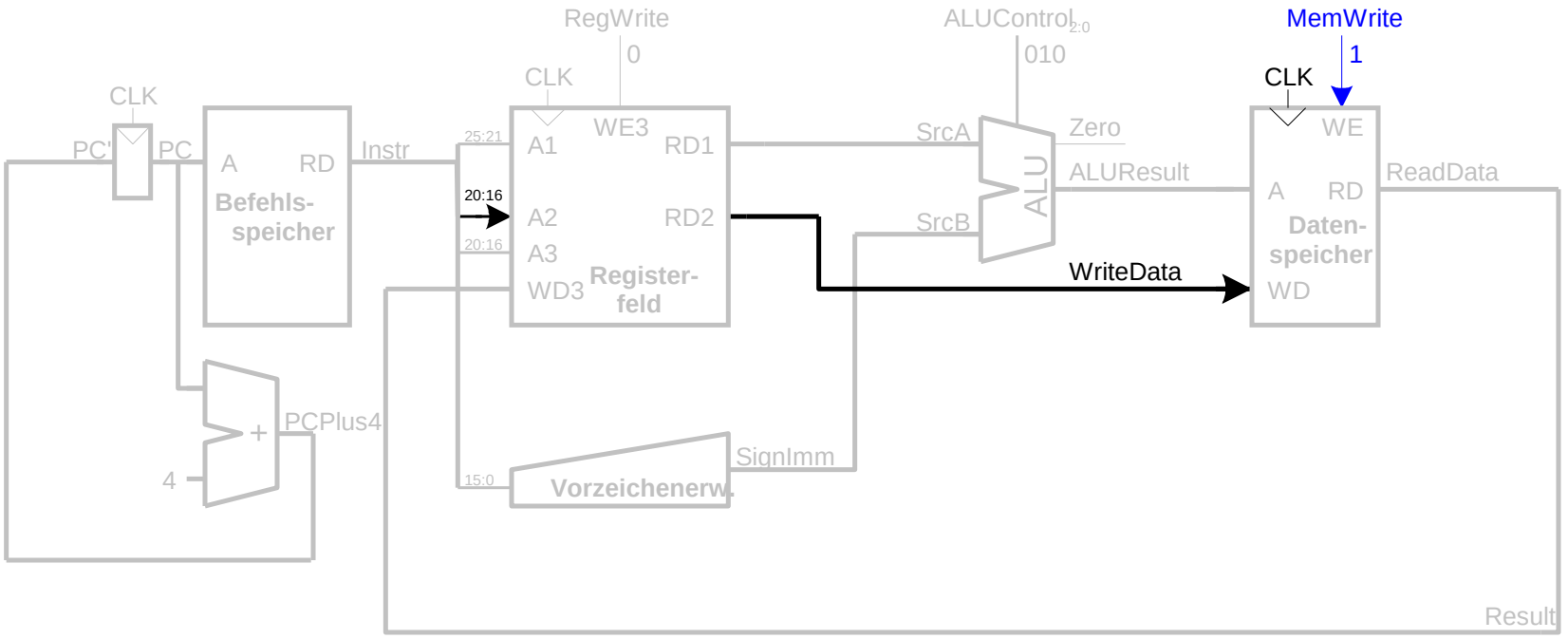


Embedded Systems & Applications

Ein-Takt Datenpfad: sw

TECHNISCHE UNIVERSITÄT DARMSTADT

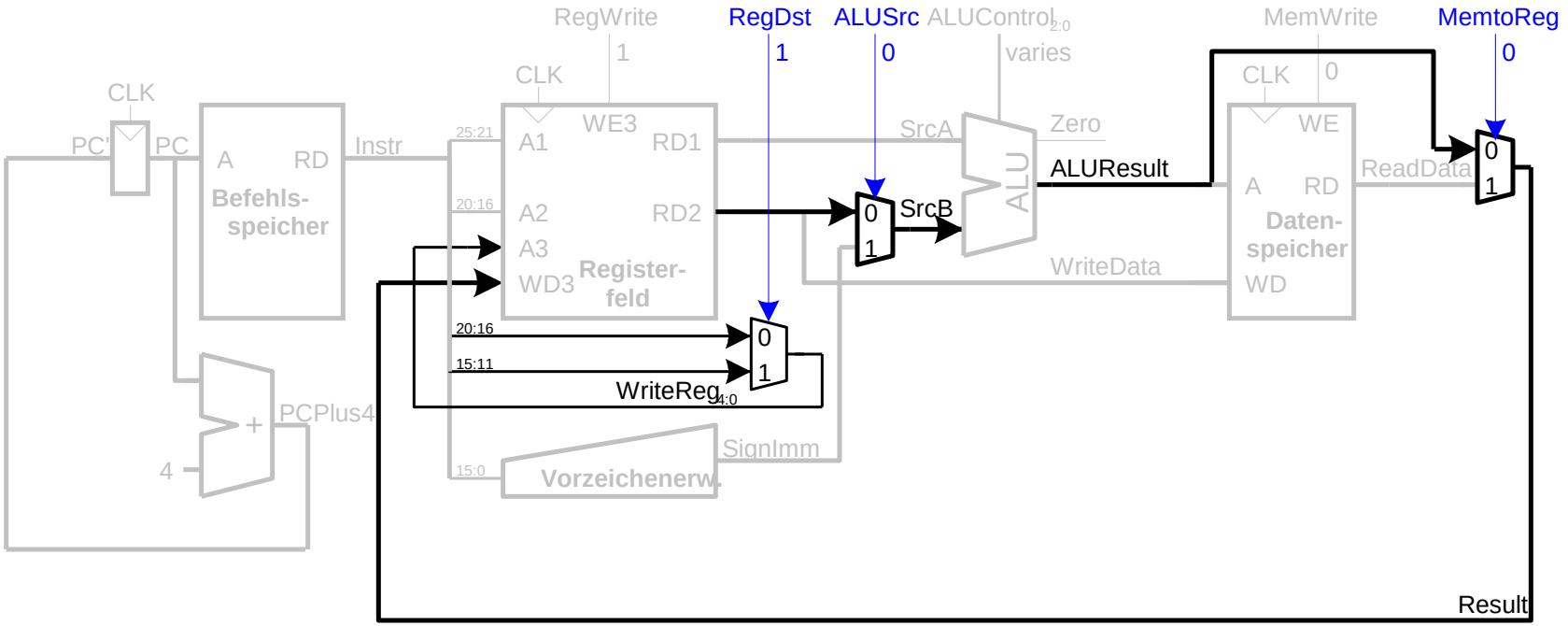
Schreibe Daten aus `rt` in den Speicher



Embedded Systems & Applications

Ein-Takt Datenpfad: Instruktionen vom R-Typ

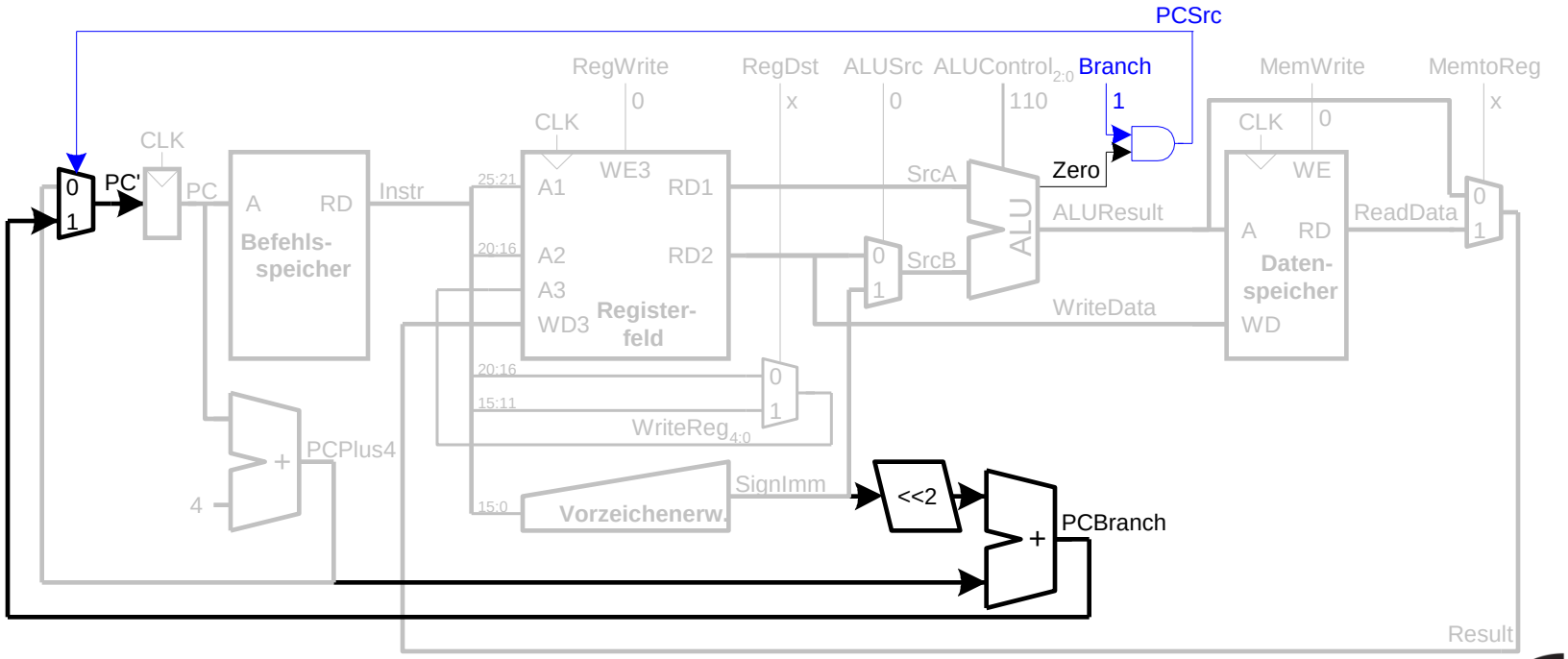
- Lese aus `rs` und `rt`
- Schreibe *ALUResult* ins Registerfeld
- Schreibe nach `rd` (statt nach `rt` wie bei `sw`)



Ein-Takt Datenpfad: beq

- Prüfe ob Werte in `rs` und `rt` gleich sind
- Bestimme Adresse von Sprungziel *(branch target adress, BTA):*

  BTA = (vorzeichenerweiterter Direktwert << 2) + (PC+4)



Vollständiger Ein-Takt-Prozessor