Technische Informatik

2 - Instruktionssatz

© Lothar Thiele

Computer Engineering and Networks Laboratory

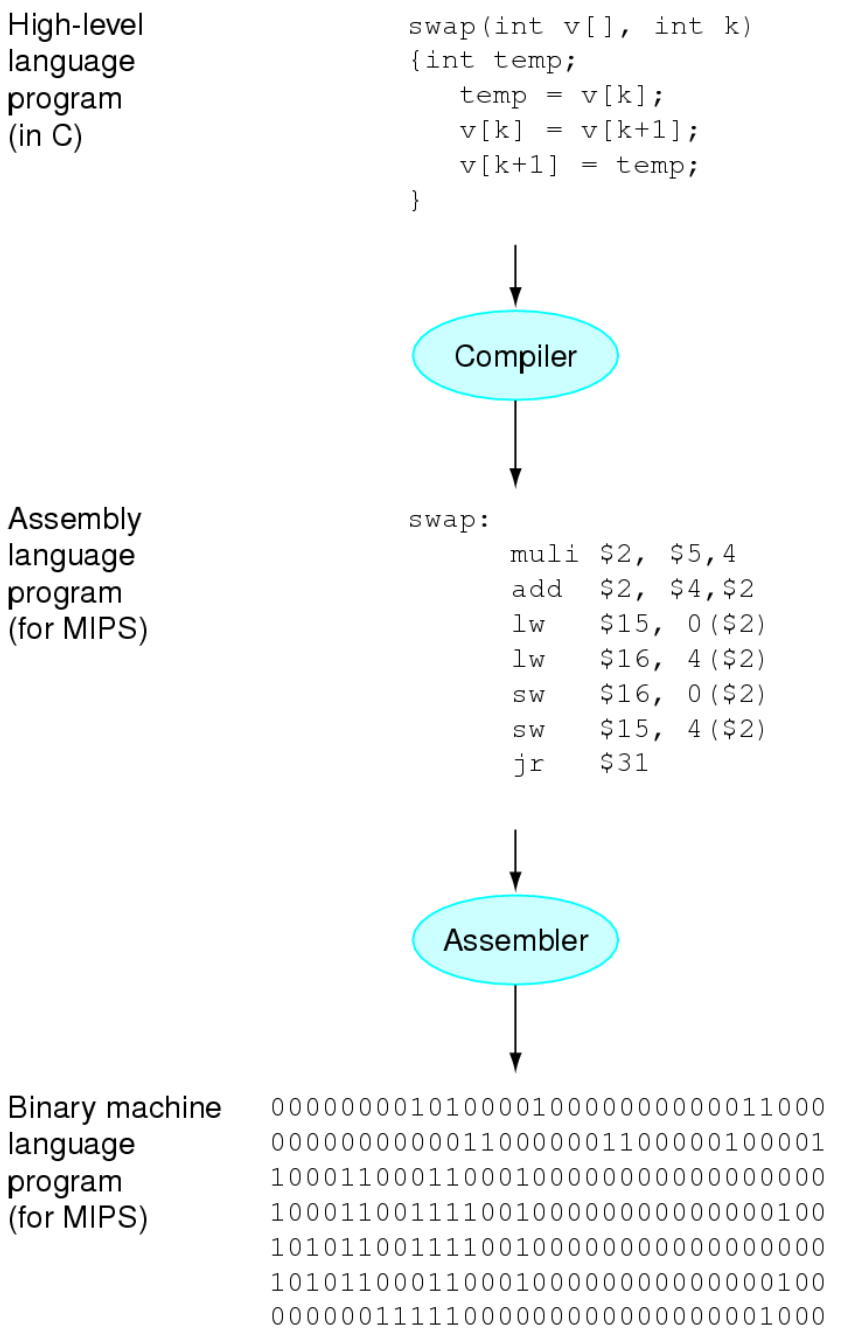
ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Instruktionsverarbeitung

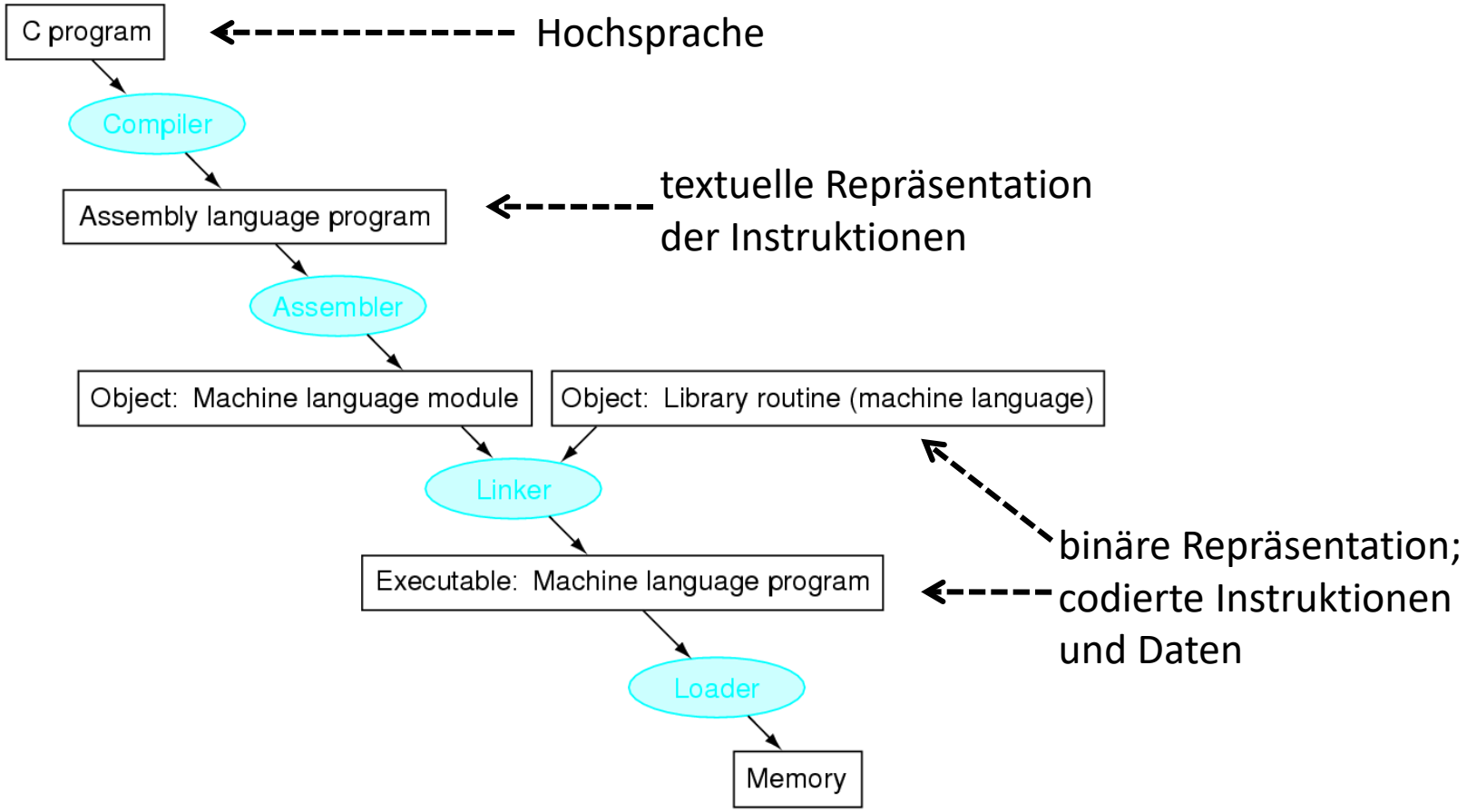
Übersetzung

- Das Kapitel 2 der Vorlesung setzt sich mit der Maschinensprache und der Assemblersprache auseinander.
- Die Instruktionen und die zugehörige Maschinensprache definieren die Hardware-Software Schnittstelle einer Rechnerarchitektur.
- Die «Instruction Set Architecture» ist demzufolge der Teil einer Rechnerarchitektur, der mit ihrer Programmierung zusammenhängt:
  - Instruktionen und Adressierungsarten,
  - Register und Speicherarchitektur,
  - Datenformate und Instruktionskodierung
  - Unterbrechnungs- und Ausnahmebehandlung,
  - Ein- und Ausgabe.

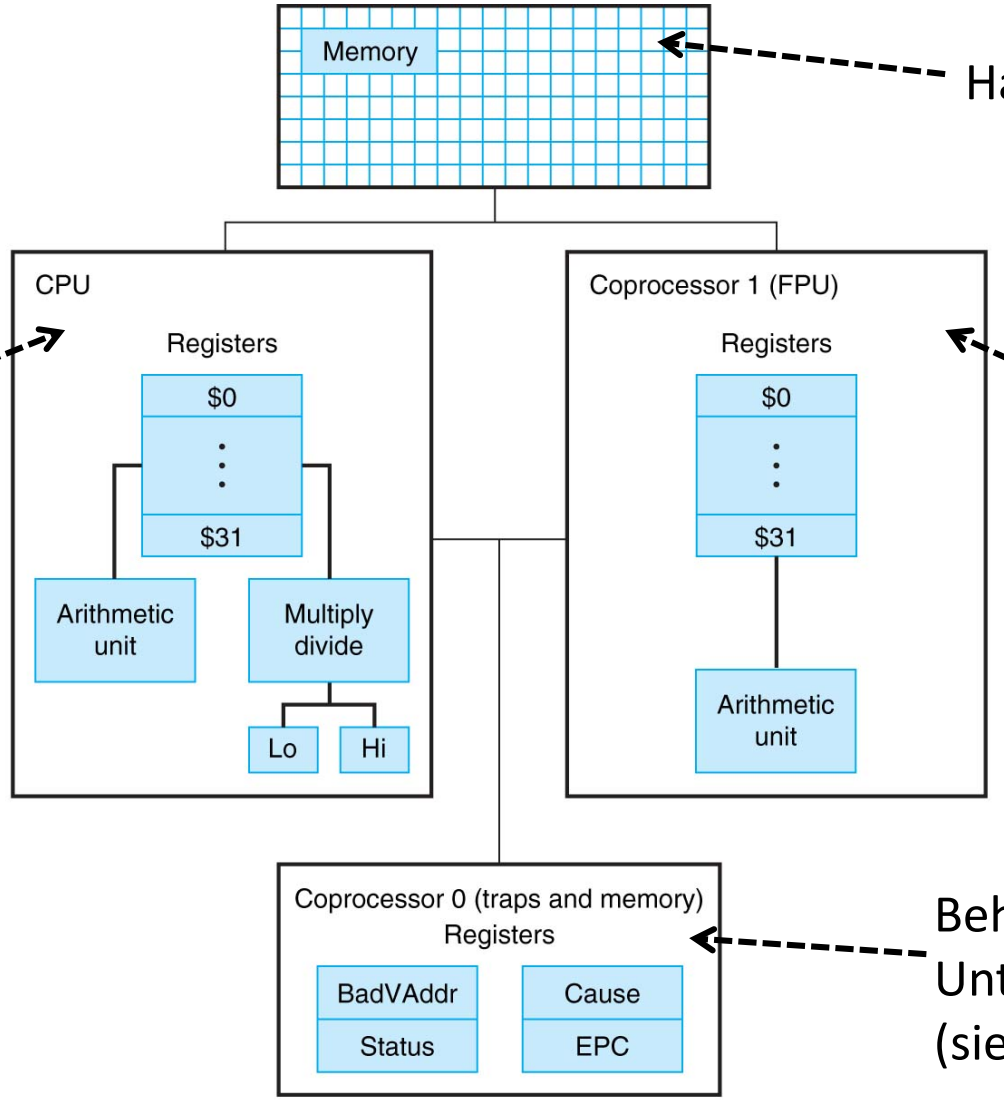


Schichtenmodell

Übersetzung eines C-Programms in Maschinensprache:



Logische Struktur eines MIPS Prozessors



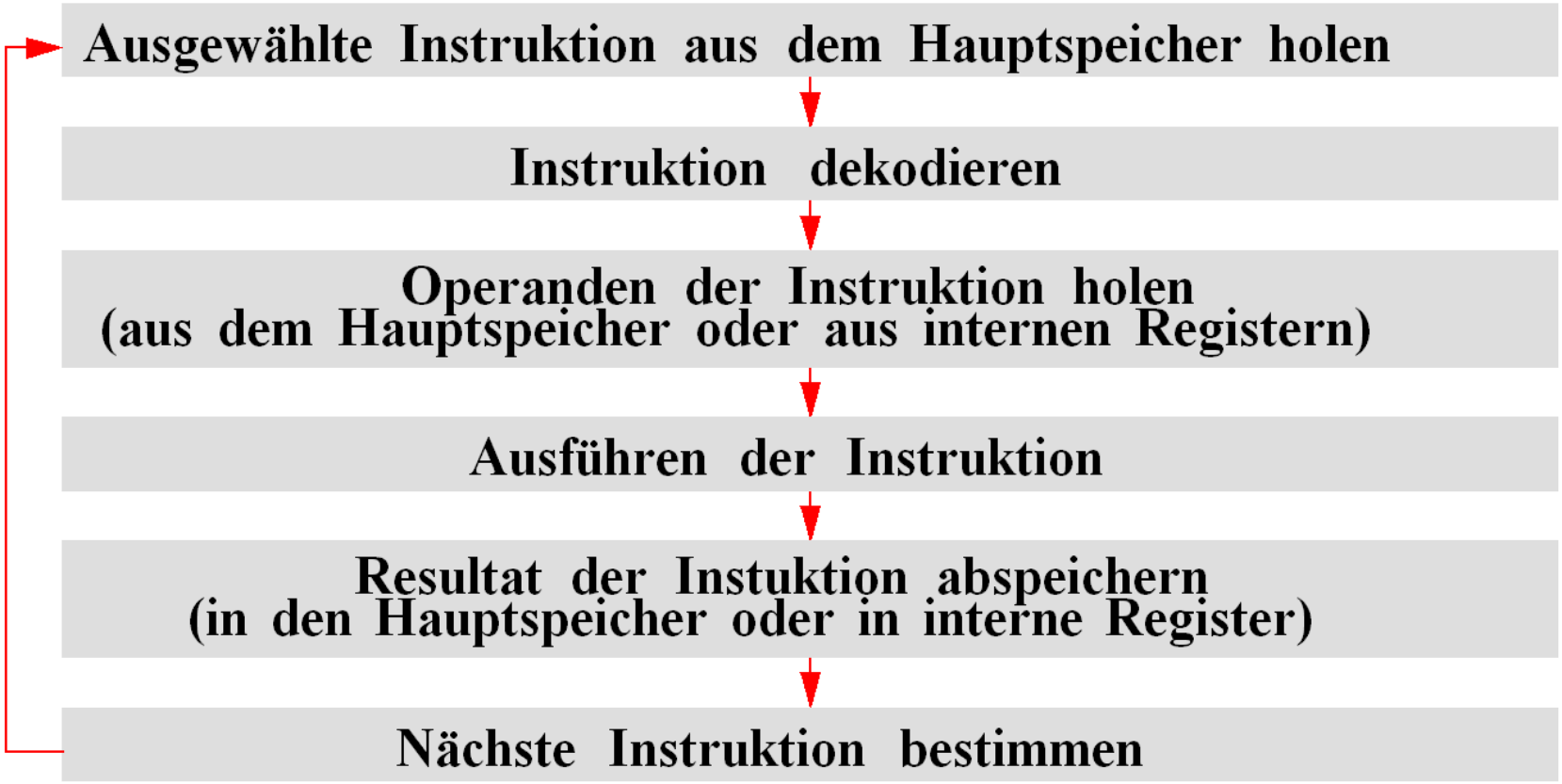
Hauptspeicher

Prozessorkern (Central Processing Unit)

Gleitkomma-Recheneinheit (wird nicht in der Vorlesung behandelt)

Behandlung von Unterbrechungen (siehe Kapitel 3)

Abarbeitung eines Maschinenprogramms



Je nach Instruktion können auch einzelne Schritte übersprungen werden.

Instruktionssatz

MIPS Assembler Instruktionen

- Es folgt eine kurze Übersicht über die verschiedenen Operationen des MIPS Instruktionssatzes. Eine vollständigere Aufstellung finden Sie im Buch zur Vorlesung.

- Ein Prozessor besitzt eine üblicherweise relativ kleine Zahl von Registern, die er zum Zwischenspeichern von Daten verwenden kann. Das auszuführende Programm und umfangreichere Daten werden im Hauptspeicher abgelegt.

symbolische Namen der Register

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.

MIPS Assembler Instruktionen

***Arithmetische und logische Instruktionen (Register)*:**

- Arithmetische Instruktionen führen arithmetische und logische Operationen auf Registern aus.
- Registerinstruktionen verwenden die ***Registeradressierung***. Alle Operanden sind also Registerinhalte.
- *Beispiele:*

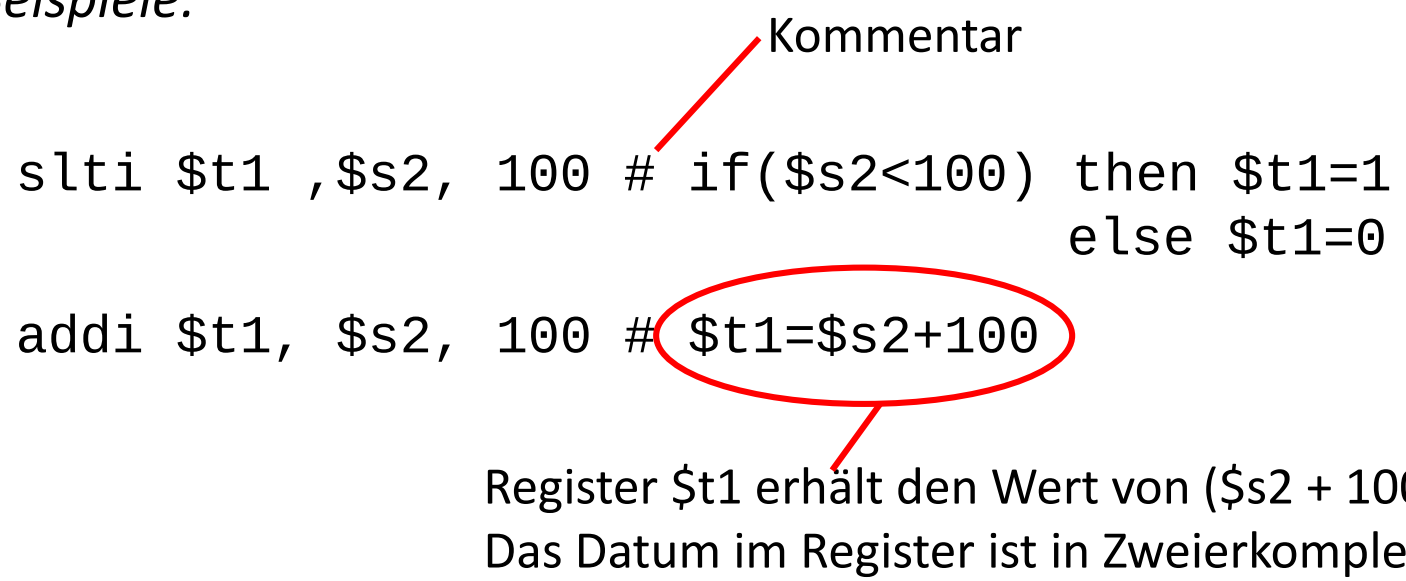
```
add $s1, $s2, $s3   # $s1=$s2+$s3

slt $s1 ,$s2, $s3   # if($s2<$s3) then $s1=1
                                  else $s1=0

and $s1, $s2, $s3   # $s1=$s2 & $s3
```

bitweise logische UND Verknüpfung

MIPS Assembler Instruktionen

***Arithmetische und logische Instruktionen (direkt)*:**

- Arithmetische Instruktionen führen arithmetische und logische Operationen auf Registern aus.
- Die *direkten* Instruktionen verwenden die so genannte ***direkte Adressierung***.
- *Beispiele:*



MIPS Assembler Instruktionen

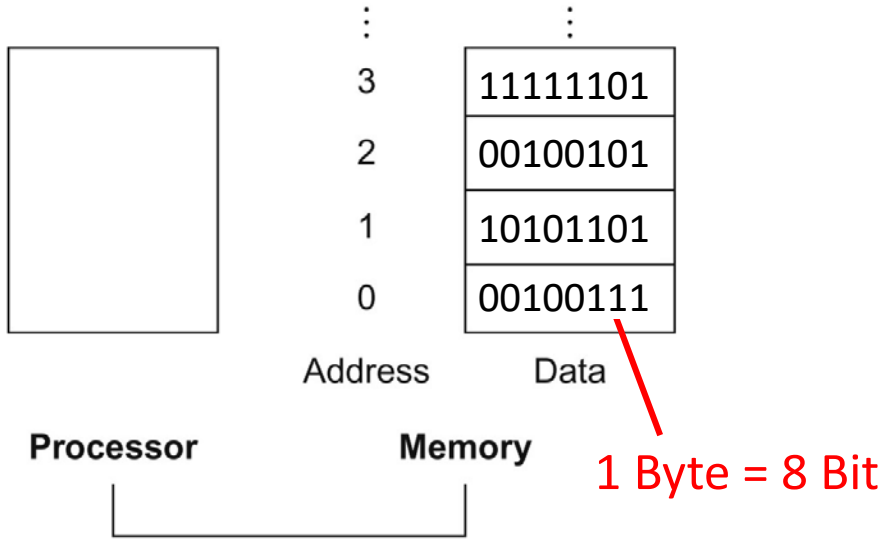
***Speicherinstruktionen*:**

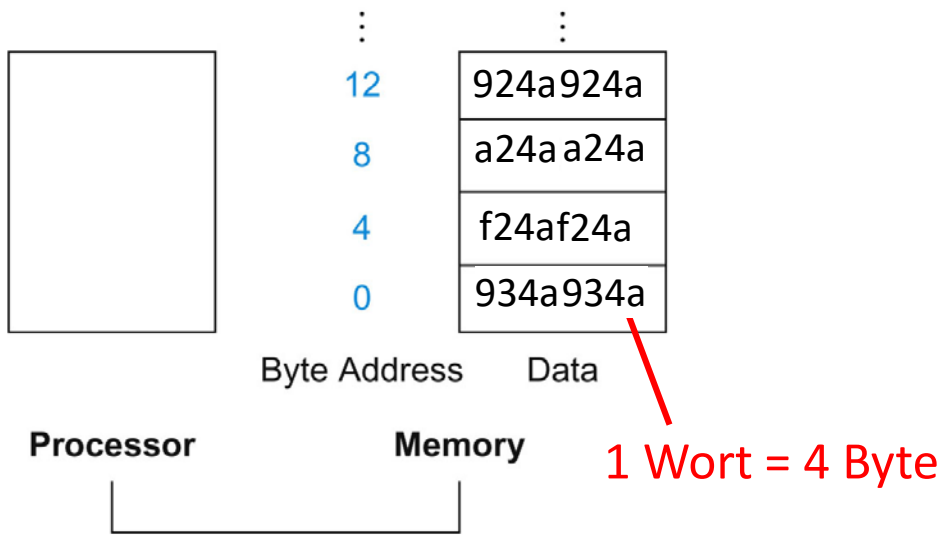
- Speicherinstruktionen bewegen Daten zwischen Speicher und Registern. Sie verwenden die so genannte ***Basisadressierung***.
- Beispiele:

```
lw $t0, 100($s2)      # $t0 = Speicher[100+$s2]
sw $t0, 100($s2)      # Speicher[100+$s2]=$t0
```

(100 + $s2) **Bytes** vom Beginn des Hauptspeichers

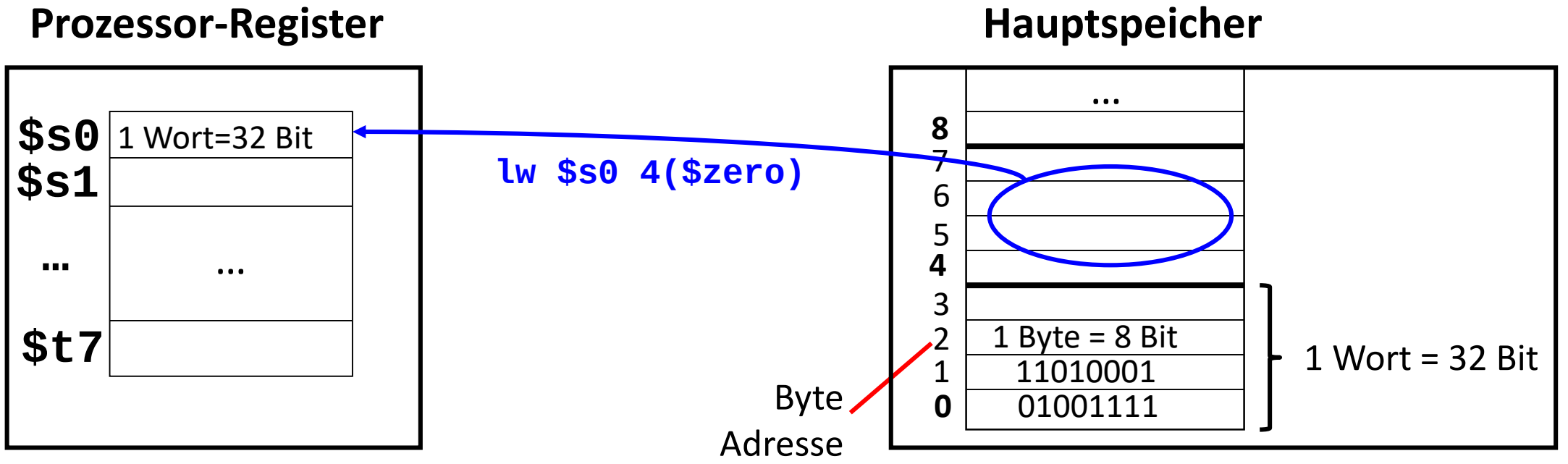
- Der Hauptspeicher (Speicher, Memory) wird byteweise adressiert:





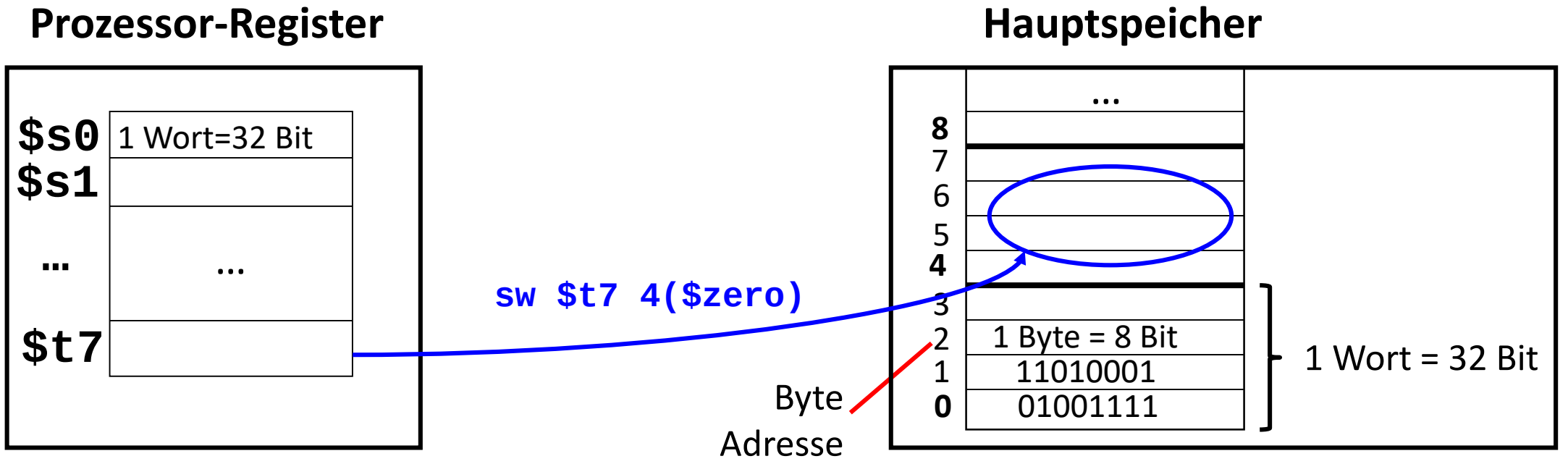
Speicherorganisation

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.



Speicherorganisation

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.



MIPS Assembler Instruktionen

Sprung- und Verzweigungsinstruktionen:

- Sprung- und Verzweigungsinstruktionen ändern den Kontrollfluss eines Programms.
- Bei Sprunginstruktionen wird in jedem Fall der Programmzähler auf eine neue Adresse gesetzt.

- Sprunginstruktionen verwenden entweder eine ***pseudodirekte Adressierung*** (`j`, `jal`) oder eine ***Registeradressierung*** (`jr, jalr`).

- Verzweigungsinstruktionen (`beq`, `bne`) verwenden eine ***PC-relative Adressierung***. Bei Verzweigungsinstruktionen wird der Programmzähler nur dann neu gesetzt, falls eine bestimmte Bedingung erfüllt ist.

MIPS Assembler Instruktionen

Beispiele Sprunginstruktion:

```
jr $ra          # PC=$ra (Adresse der nächsten Instruktion
                   ist in Register $ra)

j Label1        # go to Label1

jal Label2      # $ra=PC+4 ; go to Label2
```

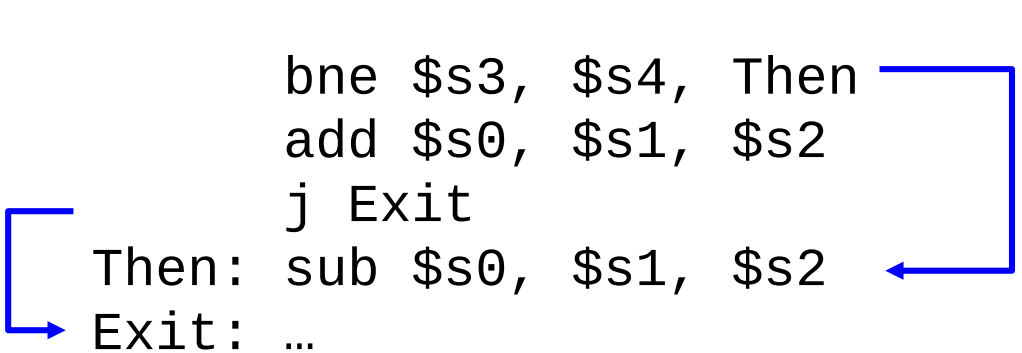
- Die Instruktionen verändern den Programmzähler. Da Instruktionen immer ein ganzes Wort (4 Byte) belegen, der Hauptspeicher aber byteweise adressiert wird, ***müssen*** die beiden niedrigwertigsten Bits des Registers mit der Sprungadresse immer den Wert 0 enthalten.
- Bei der pseudodirekten Adressierung (`j`, `jal`) wird im Assemblercode üblicherweise eine symbolische Marke als Sprungadresse angegeben.

MIPS Assembler Instruktionen

Beispiele Verzweigungsinstruktionen:

```
beq $s1, $s2, Label3 # if($s1==$s2) then go to Label3
bne $s1, $s2, Label4 # if($s1!=$s2) then go to Label4
```

- Auch bei der PC-relativen Adressierung (`beq, bne`) wird im Assemblercode üblicherweise eine symbolische Marke als Sprungadresse angegeben:




```
if ($s3 != $s4){
   $s0 = $s1 - $s2;
} else {
   $s0 = $s1 + $s2;
}
```

MIPS Assembler Instruktionen

- Falls bei Sprung- und Verzweigungsinstruktionen im Assemblercode ***ausnahmsweise*** keine symbolischen Marken sondern numerische Werte verwendet werden, richten wir uns nach den Konventionen im Buch!
- Hier werden im Assemblercode als Einheit „Instruktionen“ verwendet. Da sie immer ein ganzes Wort (4 Byte) belegen, der Hauptspeicher aber byteweise adressiert wird, müssen die jeweiligen Adressen also mit 4 multipliziert werden.

Allgemein (imm und target sind ganze Zahlen):

```
beq $s1, $s2, imm # if($s1==$s2) then PC=PC+4+4*imm
j target          # PC=4*target
```

direkter Wert: Wort

Programmzähler: Byte

nächste Instruktion

Beispiel:

```
beq $s1, $s2, 25 # if($s1==$s2) then PC=PC+4+100
j 2500           # PC=10000
```

MIPS Assembler Instruktionen

- Zusammenfassung der Bedeutung aller MIPS-Register

Name	Register number	Usage	Preserved on call?
$zero	0	the constant value 0	n.a.
$v0–$v1	2–3	values for results and expression evaluation	no
$a0–$a3	4–7	arguments	no
$t0–$t7	8–15	temporaries	no
$s0–$s7	16–23	saved	yes
$t8–$t9	24–25	more temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

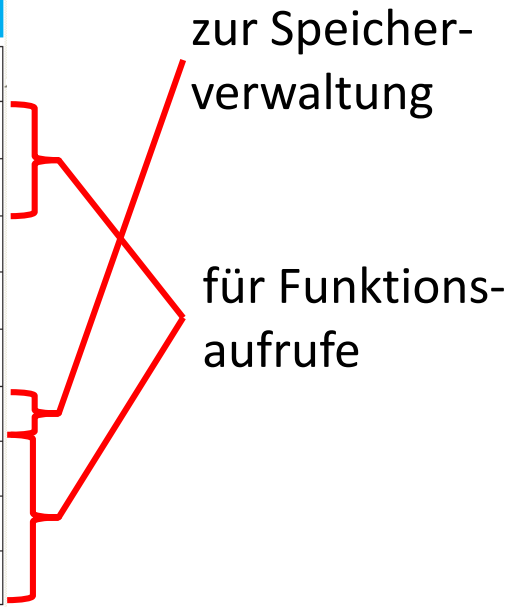


FIGURE 2.18 MIPS register conventions. Register 1, called $at, is reserved for the assembler (see Section 2.10), and registers 26–27, called $k0–$k1, are reserved for the operating system.

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

arithmetische und logische Instruktionen

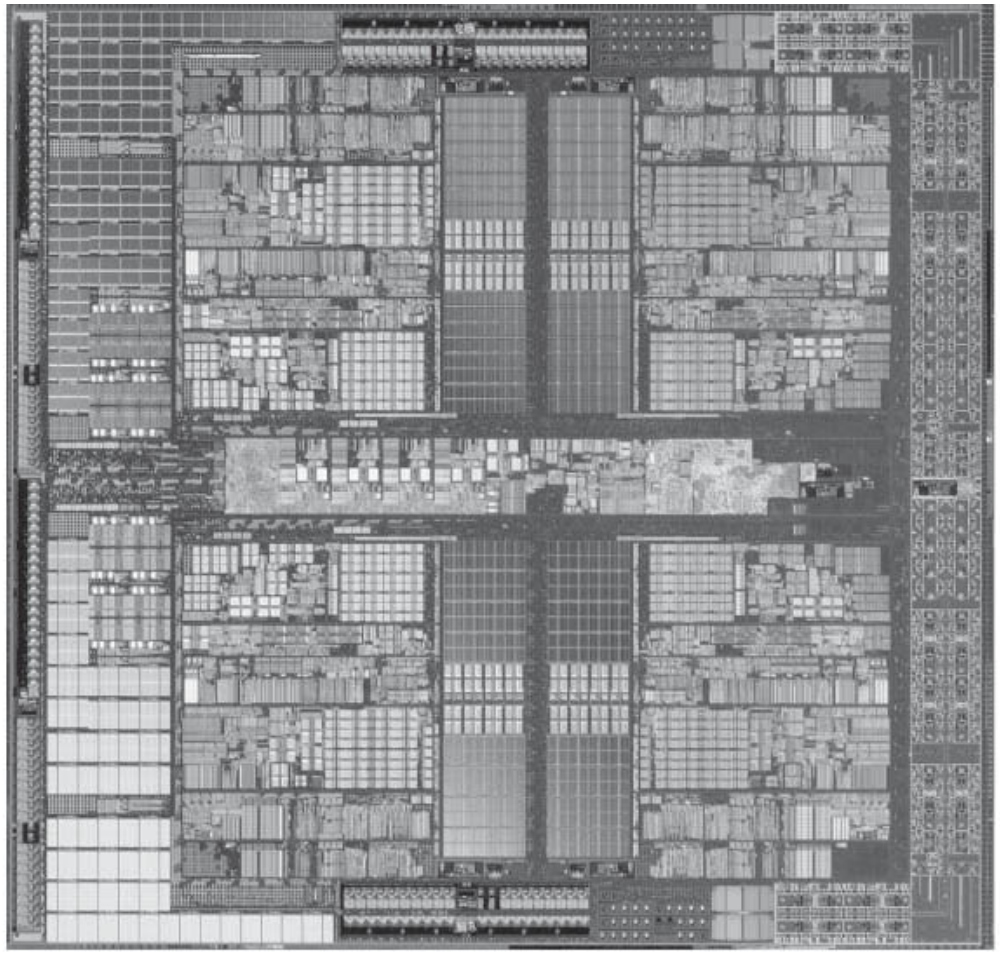
Speicher-Instruktionen

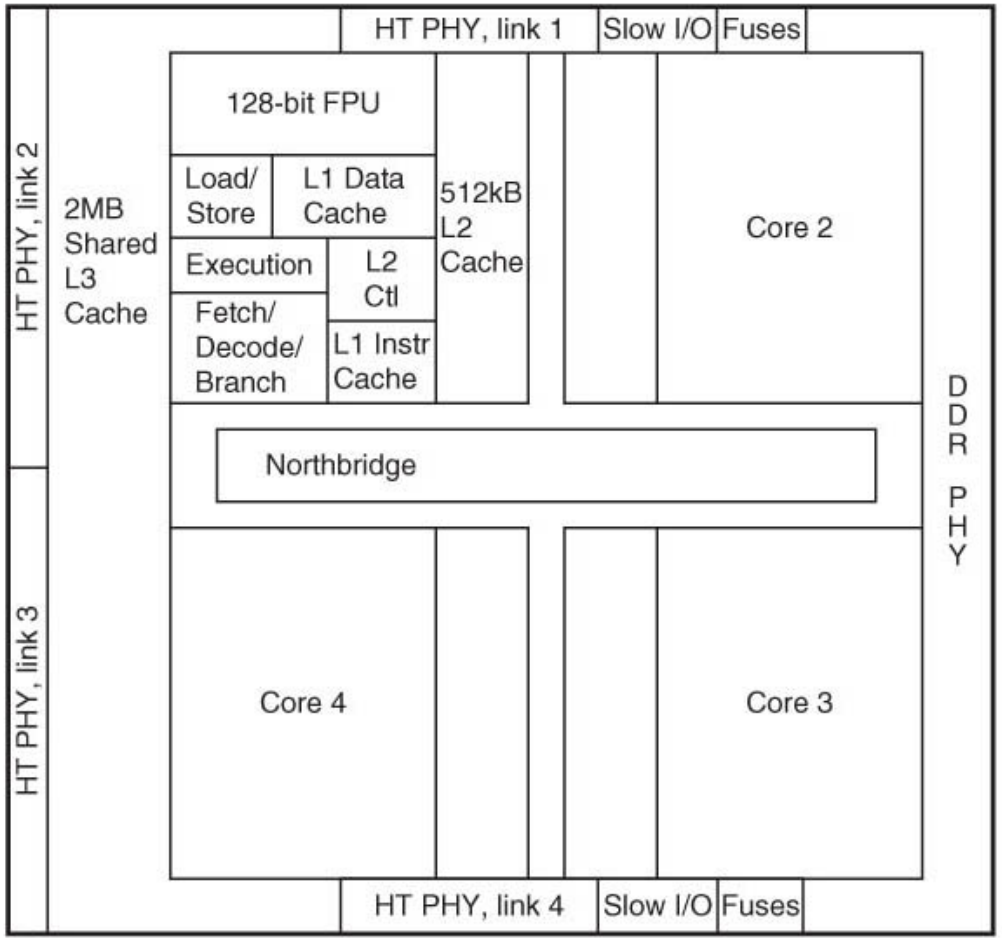
Sprung- und Verzweigungs-Instruktionen

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Synchronisation

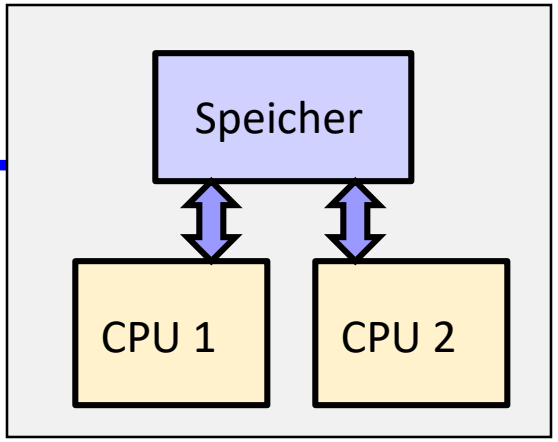
Multi-Core Prozessor





AMD Barcelona microprocessor

Synchronisation



- Parallele Zugriffe auf einen gemeinsam genutzten Speicher müssen oft koordiniert werden, z.B. beim Schreiben und Lesen einer gemeinsamen Datenstruktur.
- *Beispiel*: Zwei gleichzeitig ausgeführte Programme wollen exklusiv auf den Speicher zugreifen (*entweder* Zugriff von CPU1 *oder* von CPU2).
- Zur Koordination unter den beiden Programmen wird ein gemeinsam genutztes Datum im Speicher verwendet (Variable `lock`):
  Falls `lock` den Wert 1 hat, benutzt bereits ein Programm den Speicher, falls 0, dann ist der Zugriff freigegeben. Ein Programm führt also z.B. folgendes aus:

```
...
if (lock == 0) {
       lock = 1;
       <benutze Speicher>;
       lock = 0; }
...
```

Problem gelöst ?

Synchronisation

Was passiert, wenn nach dem erfolgreichen Test auf `lock == 0` von einem Programm das andere Programm die Variable auf `lock = 1` setzt ?

- *Erste Lösungsmöglichkeit:* atomarer Austausch, d.h. ein Registerwert wird mit einem Speicherwert atomar mit einer einzelnen Instruktion ausgetauscht.
- *Alternative:* Einführen zweier spezieller Instruktionen

```
ll $t1, offset($s1) # load linked
sc $t0, offset($s1) # store conditional
```

Falls der Inhalt der von `ll` spezifizierten Speicherstelle `Speicher[offset+$s1]` geändert wird ***vor*** der Ausführung von `sc` auf die gleiche Speicherstelle, so wird

1. `$t0` **nicht** in den Speicher geschrieben und
2. `$t0` mit 0 überschrieben.

Synchronisation

im Speicher: `Speicher[$s1]`

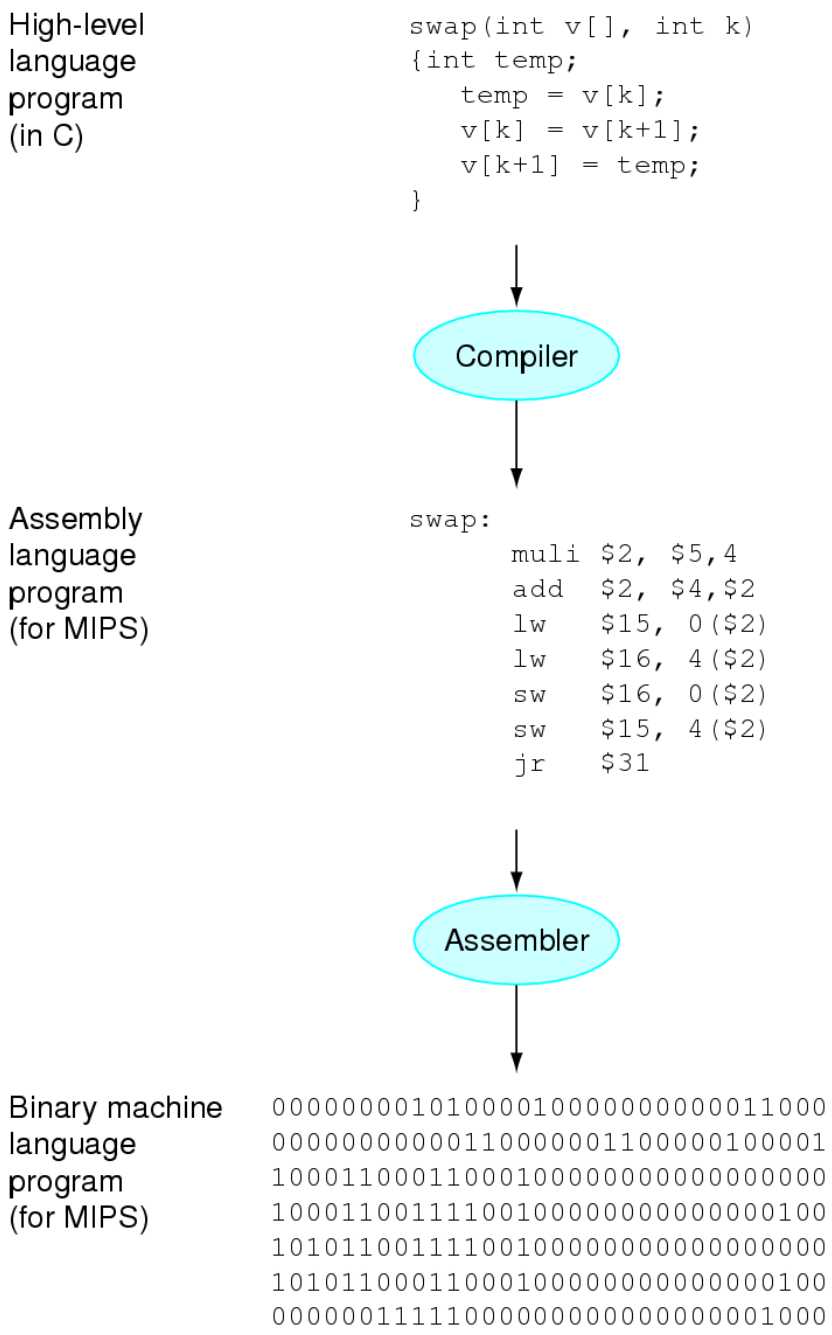
```
...
if (lock == 0) {
        lock = 1;
        <benutze Speicher>;
        lock = 0; }
...
```

```
try:   addi $t0, $zero, 1      # $t0 = 1
       ll $t1, 0($s1)          # ll auf die Variable lock
       bne $t1, $zero, exit    # falls lock != 0 -> exit
       sc $t0, 0($s1)          # sc auf die Variable lock
       beq $t0, $zero, try     # falls sc nicht erfolgreich -> try
       ...                     # benutze Speicherbereich
       sw $zero, 0($s1)        # lock = 0
exit:  ...
```

Instruktionskodierung

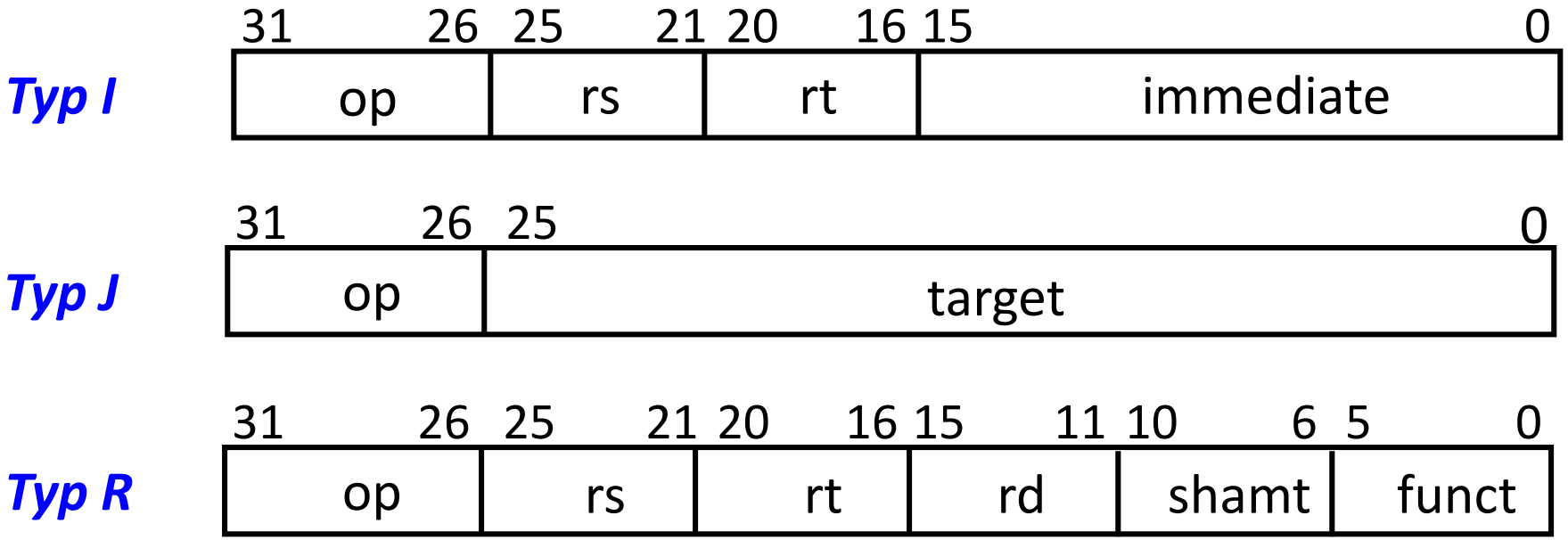
Übersetzung (Wiederholung)

- Das Kapitel 2 der Vorlesung setzt sich mit der Maschinensprache und der Assemblersprache auseinander.
- Die Instruktionen und die zugehörige Maschinensprache definieren die Hardware-Software Schnittstelle einer Rechnerarchitektur.
- Die «Instruction Set Architecture» ist demzufolge der Teil einer Rechnerarchitektur, der mit ihrer Programmierung zusammenhängt:
  - Instruktionen und Adressierungsarten, ✓
  - Register und Speicherarchitektur, ✓
  - Datenformate und Instruktionskodierung
  - Unterbrechnungs- und Ausnahmebehandlung,
  - Ein- und Ausgabe.



Instruktionskodierung

- Unter ***Instruktionskodierung*** versteht man die Umsetzung einer Instruktion in ein Maschinenwort.
- Der MIPS Prozessor benutzt ausschliesslich 32-Bit Kodierungen (alle Instruktionen besitzen eine feste Länge).
- Man unterscheidet die 3 Typen I, J und R:



Instruktionskodierung

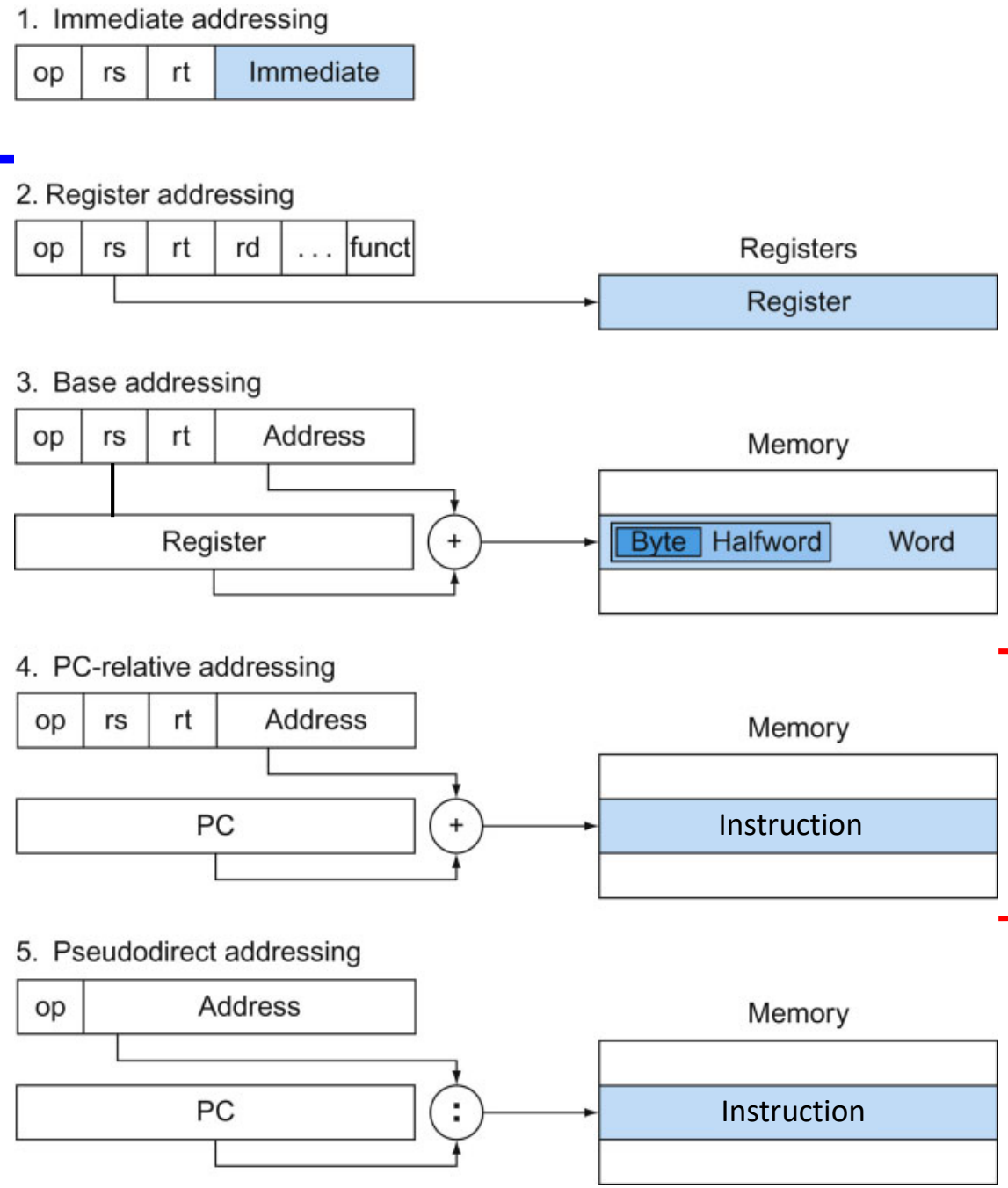
Abkürzung	Bedeutung
I	immediate (direkt)
J	jump (Sprung)
R	register (Register)
op	6 Bit Kodierung der Operation
rs	5 Bit Kodierung eines Quellenregister
rt	5 Bit Kodierung eines Quellenregisters oder Zielregisters
immediate	16 Bit direkter Wert oder Adressverschiebung
target	26 Bit Sprungadresse
rd	5 Bit Kodierung des Zielregisters
shamt	5 Bit Grösse einer Verschiebung
funct	6 Bit Kodierung der Funktion (Ergänzung des Feldes op)

Adressierungsarten

- Unter Adressierungsarten versteht man die verschiedenen Möglichkeiten, die ein Instruktionssatz bietet, um Daten zu adressieren.
- Der MIPS Prozessor unterstützt 5 Adressierungsarten:
  - ***Direkte Adressierung***: Der Operand ist eine Konstante in der Instruktion.
  - ***Registeradressierung***: Der Operand ist in einem Register.
  - ***Basisadressierung***: Der Operand ist im Speicher an der Adresse, die sich durch die Summe eines Registerinhaltes und einer Konstanten in der Instruktion ergibt.
  - ***PC-relative Adressierung***: Die Adresse ergibt sich aus der Summe des Programmzählers (PC + 4) und einer Konstanten in der Instruktion.
  - ***Pseudodirekte Adressierung***: Der neue Programmzähler (PC) ergibt sich aus einer Konstanten (26 Bit) und den oberen 4 Bit des alten Programmzählers (PC + 4).

Adressierungsarten

Instruktionskodierung



Kapitel 4

Instruktionskodierung - Beispiele

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10
beq	I	4	17	18	25			beq $s1,$s2, 25
bne	I	5	17	18	25			bne $s1,$s2, 25
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 2500 (see Section 2.9)
jr	R	0	31	0	0	0	8	jr $ra
jal	J	3	2500					jal 2500 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

Instruktionskodierung

Beispiel für Instruktionskodierung:

- `add $t0, $s1, $s2`

  `$t0` ist Register 8, `$s1` ist Register 17, `$s2` ist Register 18
  `add` hat Operationskodierung 0 und Funktionskodierung 32

  Dezimale Schreibweise:

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32

  Binäre Schreibweise:

000000	10001	10010	01000	00000	100000

`0000 0010 0011 0010 0100 0000 0010 0000`

  Hexadezimale Schreibweise:

02324020_{16} oder 02324020_{hex} oder `0x02324020`

Umrechnungstabelle

dezimal	hexadezimal	binär
0_{10}	0_{16}	0000_2
1_{10}	1_{16}	0001_2
2_{10}	2_{16}	0010_2
3_{10}	3_{16}	0011_2
4_{10}	4_{16}	0100_2
5_{10}	5_{16}	0101_2
6_{10}	6_{16}	0110_2
7_{10}	7_{16}	0111_2

dezimal	hexadezimal	binär
8_{10}	8_{16}	1000_2
9_{10}	9_{16}	1001_2
10_{10}	a_{16}	1010_2
11_{10}	b_{16}	1011_2
12_{10}	c_{16}	1100_2
13_{10}	d_{16}	1101_2
14_{10}	e_{16}	1110_2
15_{10}	f_{16}	1111_2

Instruktionskodierung

Beispiel für Instruktionskodierung:

- Assemblerinstruktion umwandeln in eine binäre Darstellung:

The machine language version of `lui $t0, 255` `# $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing `lui $t0, 255`:

0000 0000 1111 1111	0000 0000 0000 0000

- `00af8020`$_{hex}$ umwandeln in eine Assemblerinstruktion:
  - Binäre Darstellung: `0000 0000 1010 1111 1000 0000 0010 0000`
  - Dekodierung des Instruktionstyps: Typ R da op=`000000`
  - Unterteilung der binären Darstellung

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

  - Resultat: `add $s0, $a1, $t7`

Datenformate

Die folgenden ***Datenformate*** sind definiert:

- Byte (8 Bit)
- Halbwort (16 Bit)
- Wort (32 Bit)

Verwendet wird die *"**big-endian**"* Konvention für die Ordnung von Bytes in Worten und die von Worten in mehrfachen Wortstrukturen (Blöcken):

- Das höchstwertige Byte eines Wortes ist an seiner niedrigsten Adresse.
- Ein Wort wird adressiert mit der Byteadresse seines höchstwertigen Bytes (niedrigste Adresse).

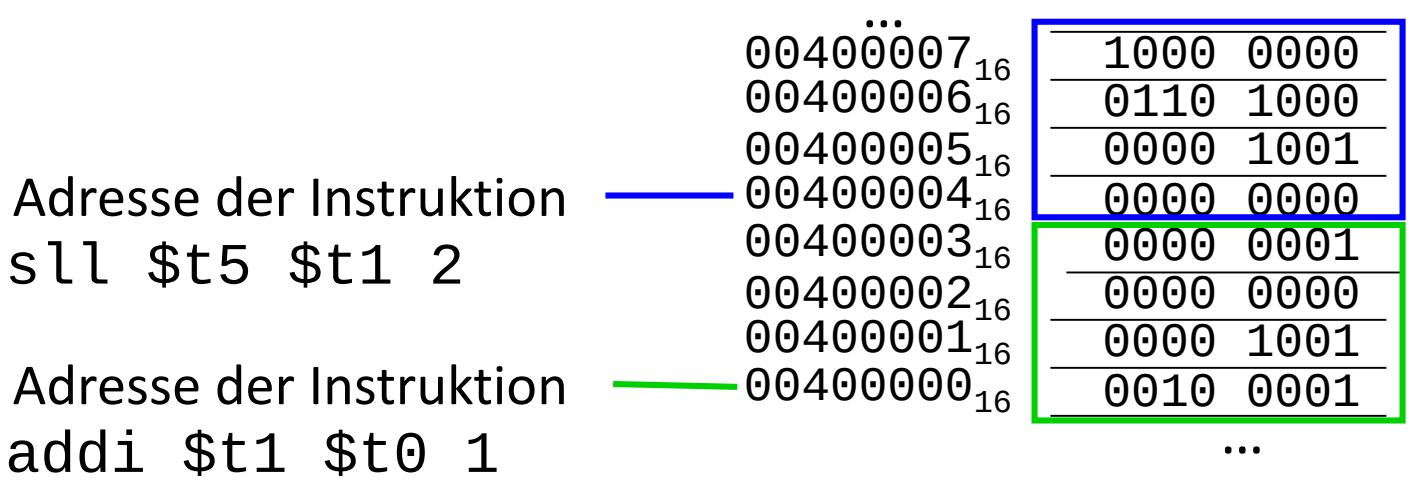
Datenformate

Beispiel *"**big-endian**"* Konvention:

- Programm:

```
addi $t1, $t0, 1          #   21090001hex
sll $t5, $t1, 2           #   00096880hex
```

- Programmspeicher:



Datenformate

Ganze Zahlen werden im MIPS Prozessor entweder vorzeichenlos (“***unsigned***“) oder vorzeichenbehaftet (“***signed***“) im Zweierkomplement dargestellt:

- Eine n-Bit *vorzeichenlose* Binärzahl hat den Wert:

$$B = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

Die *Erweiterung* zu einer m-Bit-Binärzahl erfolgt durch Auffüllen mit 0:

$$B = \sum_{i=n}^{m-1} 0 \cdot 2^i + \sum_{i=0}^{n-1} b_i \cdot 2^i$$

Datenformate

Eine n-Bit *vorzeichenbehaftete* Binärzahl wird im *Zweierkomplement* dargestellt und hat den Wert:

$$B = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

- Das *Vorzeichen* kann leicht wie folgt bestimmt werden:

$$(B < 0) \Leftrightarrow (b_{n-1} = 1)$$

Negieren erfolgt durch Addition von 1 zur dualen Binärzahl:

$$-B = -(1 - b_{n-1}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (1 - b_i) \cdot 2^i + 1$$

Datenformate

- Die *Erweiterung* zu einer m Bit Binärzahl im Zweierkomplement erfolgt durch Auffüllen mit dem höchstwertigen Bit:

$$B = -b_{n-1} \cdot 2^{m-1} + \sum_{i=n}^{m-2} b_{n-1} \cdot 2^i + \sum_{i=0}^{n-1} b_i \cdot 2^i$$

- *Beispiel:* Registerinhalte:

```
r1 = 0..0001, r2 = 0..0010, r3 = 1..1111
```

Assemblerinstruktionen:

```
slt  r4,r2,r1  # if(r2<r1)then r4=1 else r4=0
slt  r5,r3,r1  # if(r3<r1)then r5=1 else r5=0
sltu r6,r2,r1 # if(r2<r1)then r6=1 else r6=0
sltu r7,r3,r1 # if(r3<r1)then r7=1 else r7=0
```

unsigned (vorzeichenlos)