```
FUN:    sw $ra,-4($sp)
        sw $s0,-8($sp) # $s0 für C
        sw $s1,-12($sp) # $s1 für j
        sw $s2,-16($sp) # $s2 für i
        sw $s3,-20($sp) # $s3 für m
        addi $sp,$sp,-20
        move $s0,$a0
        move $s1,$a1
        move $s2,$a3
        addi $s3,$a2,9
        move $a0,$s2
        move $a1,$s1
        jal V
        add $s3,$s3,$v0
        move $a0,$s1
        move $a1,$s2
        jal X
        add $s3,$s3,$v0
        sll $t0,$s3,2
        add $t0,$t0,$s0
        lw $v0,0($t0)
        addi $sp,$sp,20
        lw $ra,-4($sp)
        lw $s0,-8($sp) # $s0 für C
        lw $s1,-12($sp) # $s1 für j
        lw $s2,-16($sp) # $s2 für i
        lw $s3,-20($sp) # $s3 für m
        jr $ra
```

Rechnerarchitektur (AIN 2)

SoSe 2021

Kapitel 2

Befehle: Die Sprache des Rechners

Prof. Dr.-Ing. Michael Blaich
mblaich@htwg-konstanz.de

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

2.1 Befehlssatz: Was ist das?

2.2 Befehle des MIPS Befehlssatzes

2.3 Darstellungen von Befehlen im Rechner

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.6 MIPS Assembler und MARS Simulator

2.7 Weitere MIPS-Befehle

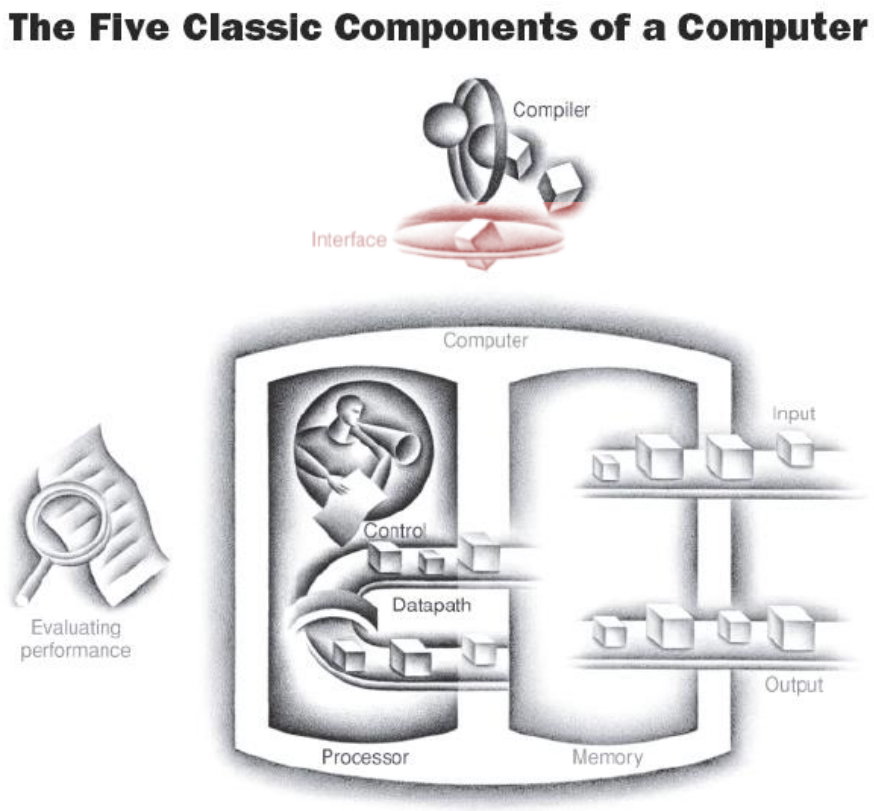
2.8 Compiler, Assembler, Linker, Loader

2.9 Andere Befehlssätze

2.10 Zusammenfassung

Befehlssatz (Instruction Set)

ISA (Instruction Set Architecture): Interface zwischen Compiler und Prozessor



Befehlssatz (Instruction Set)

Befehlssatz=Repertoire an Befehlen, die eine bestimmte Rechnerarchitektur versteht

- Grundlegende Befehlssätze sehr ähnlich
- Grundlegende Befehlssätze können durch spezielle Befehlssätze ergänzt werden
  - Multimedia
  - Kryptographie
- Frühe und moderne Befehlssätze sehr einfach
  - RISC (Reduced Instruction Set Computer)
- Zwischenzeitlich umfangreiche Befehlssätze
  - CISC (Complete Instruction Set Computer)

Arithmetische Operationen	
ADD	Addition von Operanden
SUB	Subtraktion von Operanden
MUL	Multiplikation von Operanden
AND	Logische AND-Operation
Vergleichsoperationen	
EQ	Tested ganzahlige Operanden
GT	Ermittelt den größeren Operanden
LT	Ermittelt den kleineren Operanden
GEQ	Ob ein Operand größer/gleich ist
Steuerungsoperationen	
JMP	Setzen des Programmzählers auf die nächste Anweisung
BLT	Setzen des Progr.-Zählers auf den ersten Operanden, wenn der zweite kleiner ist als der dritte

MIPS Befehlssatz

- Beispiel Befehlssatz dieser Vorlesung
- Keine Theorie sondern real genutzter Befehlssatz

- MIPS Technologies (-> Wikipedia, www.imgtec.com):
  - 1984 u.a. von John l. Hennessy gegründet
  - Entwickler der MIPS Architektur, erster kommerzieller Anbieter von RISC CPUs; 2012 von Imagination Technologies übernommen; 2018 von Wave Computing (KI Spezialist) erworben
  - Beträchtlicher Markanteil im Embedded Bereich
  - Typisch für einen modernen Befehlssatz

Aktuelle MIPS Architektur

	SIMD	Virtualization	Multi-threading	DSP	MIPS16e™ ASE	MCU ASE
MIPS32	●	●	●	●	●	●
MIPS64	●	●	●	●	●	●
microMIPS32	●	●	●	●		●
microMIPS64	●	●	●	●		●

Weitere weitverbreitete Befehlssätze

ARMv7 Befehlssatzarchitektur

- ähnlich zu MIPS
- mit mehr als 9 Milliarden Chips im Jahr 2011 verbreitetste Befehlssatzarchitektur

ARMv8 Befehlssatzarchitektur

- Erweiterung von ARMv7 auf 64bit
- Noch ähnlicher zu MIPS als ARMv7

Intel x86 Befehlssatzarchitektur

- Einsatz für PC und PostPC-Ära
- Interne Umsetzung eines CISC-Befehlssatzes auf einen modernen RISC-Befehlssatz

Java Bytecode

- Befehlssatz des ursprünglichen Java Interpreters
- mittlerweile übersetzen Just-in-time Compiler den Java-Bytecode in MIPS ähnlichen Befehlssatz

Atmel AVR Befehlssatz

- RISC Befehlssatz für Atmel 8-Bit Microcontroller
- sehr einfache 16bit Befehle, die auf 2 Registern arbeiten

Vorlesungsinhalt

Erstes Beispiel

Alle arithmetischen Operationen haben 3 Operanden (3-Adressen-Maschine)

- Befehlssatz mit fester Anzahl von Operanden ist einfacher in Hardware zu realisieren als Befehlssatz mit variabler Anzahl von Operanden

Design-Prinzip 1 des Hardware-Entwurfs:

- Simplicity favors regularity (Einfachheit begünstigt Regelmäßigkeit)

Addiere die Werte der Variablen „b“ und „c“ und weise das Ergebnis der Variablen „a“ zu

```
add     a, b, c     # a = b+c
```

Operator Operanden Kommentar

„Kompilieren“ einer Code-Sequenz

- Hochsprachenbefehle werden teilweise in mehrere Assemblerbefehle umgesetzt

Hochsprache wie z.B. C		Assemblersprache
Beispiel 1:	`a=b+c;` `d=a-e;`	`add a, b, c` `sub d, a, e`

Beispiel 2:	`f=(g+h)-(i+j);`	`add t0, g, h`
		`add t1, i, j`
		`sub f, t0, t1`

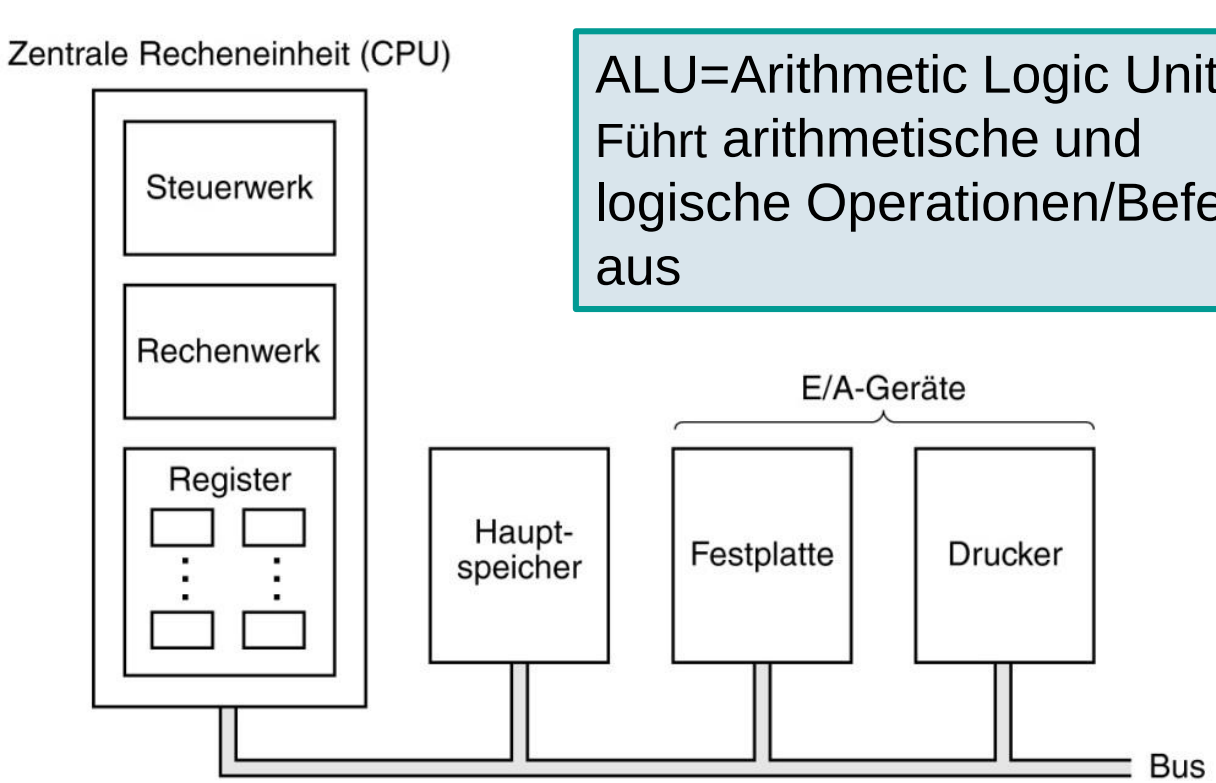
Nächste Frage:
Wo kommen die Operanden her?

temporäre Variablen t0 und t1

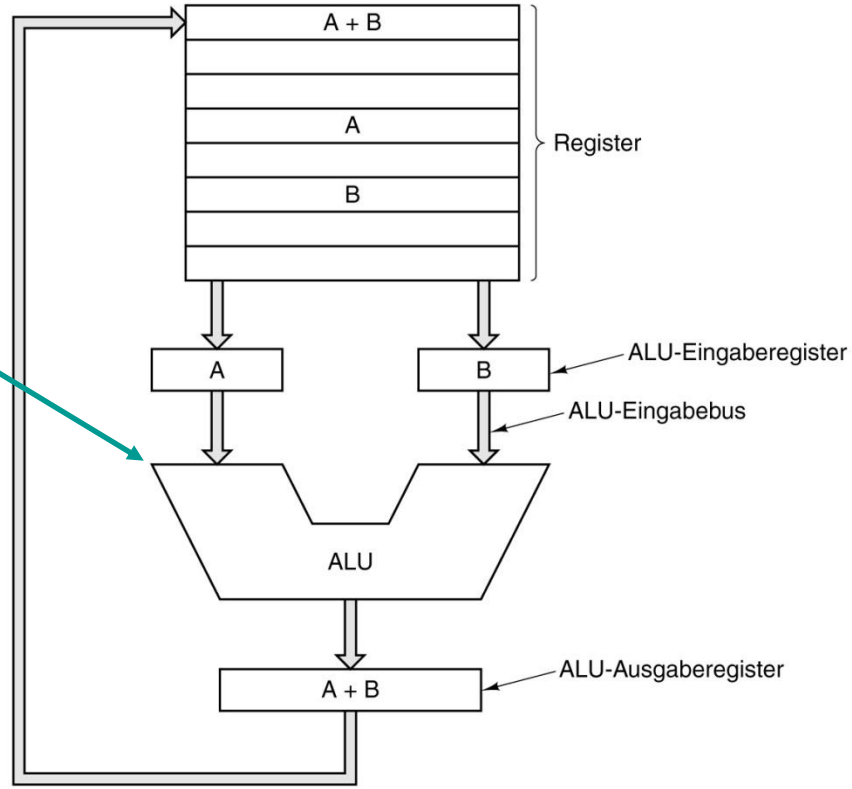
Rechnen in einer CPU

Die Operanden arithmetischer Operationen stehen in Registern und auch das Ergebnis wird in einem Register gespeichert.

- Register sind wenige spezielle Speicherplätze in der CPU
- Zugriff auf Register erfolgt **nicht** über den Bus



ALU=Arithmetic Logic Unit
Führt arithmetische und logische Operationen/Befehle aus



Zusammenfassung: Operanden und Register

CPU rechnet mit Variablen, die in Registern gespeichert sind

- Eine 64-bit CPU benötigt dementsprechend Register mit 64 Bit

Arithmetische Befehle haben drei Operanden

- zwei Input-Operanden
- ein Output-Operand

Diese Operanden sind in Registern gespeichert

Registern:

- Speicherplätze als Teil der CPU in der unmittelbaren Nähe des Rechenwerks.
- Zugriff erfolgt direkt und nicht über einen Bus.
- Die einzigen Speicherplätze, die Operanden enthalten können. Operanden können nicht im Hauptspeicher liegen.
- Register sind typischerweise 32-bit oder 64-bit breit.
- Größe des Registerinhalts wird als ein Wort (word) bezeichnet
  - Vorsicht: nicht zu verwechseln mit ein (Daten-)Word=2 Byte

Register in MIPS

Design-Prinzip 2 des Hardware-Entwurfs

- Smaller is Faster (Kleiner ist schneller)
- mehr Register bedeuten längeren Taktzyklus und mehr Bits für Adressierung

MIPS benutzt 32 Register mit 32 Bit

- ein „Wort“ in MIPS hat 32 Bits
- nummeriert: 0-31

32 Register bilden Kompromiss zwischen Anzahl und Schnelligkeit

- mehr Register benötigen größere Schaltkreise und würden Taktgeschwindigkeit senken
- Hauptspeicher größer und langsamer

Register haben oft spezielle Funktionen und bekommen spezielle Namen

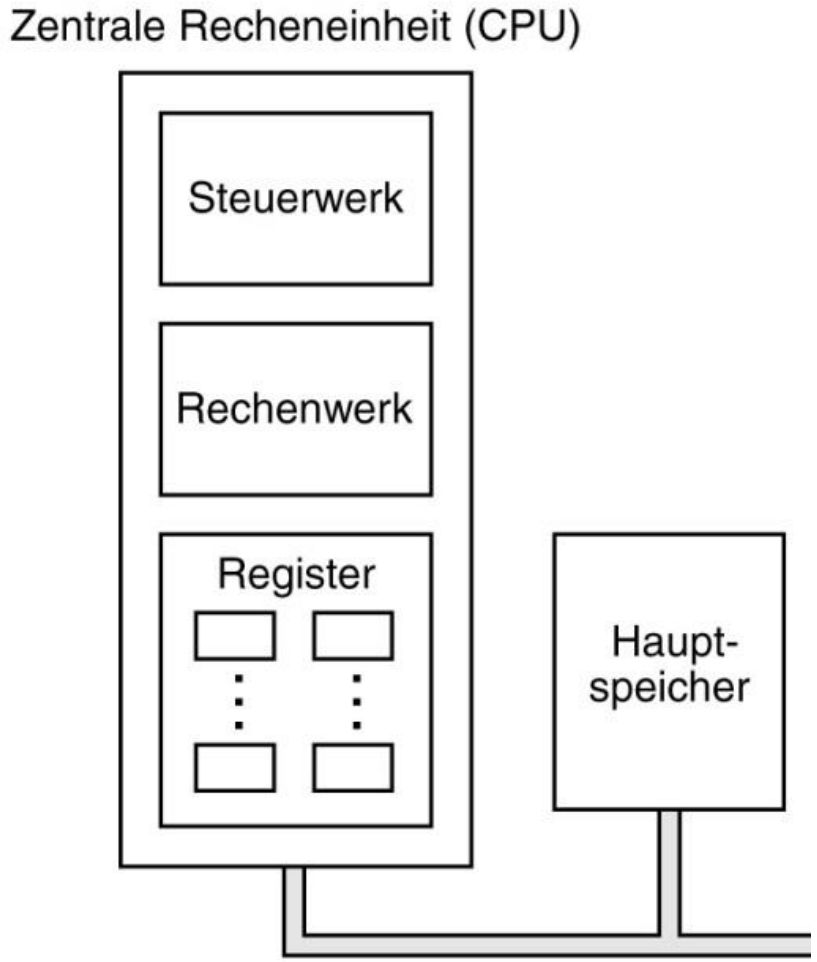
- $t0-$t9: temporäre Variablen
- $s0-$s7: gesicherte Variablen

Viele Register haben spezielle Funktionen z.B. im Zusammenhang mit Prozeduren.

- $a0 ... $a3: Argumente für Prozeduraufruf
- $v0 ... $v1: Result value Register

Die 32 MIPS Register

Name	Nummer	Funktion
$zero	0	constant 0
$at	1	assembler
$v0 ... $v1	2-3	result value registers
$a0 ... $a3	4-7	arguments
$t0 ... $t7	8-15	temporary variables
$s0 ... $s7	16-23	saved
$t8 ... $t9	24-25	temporary variables
$k0 ... $k1	26-27	operating system
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address



Voriges Beispiel mit Registern statt Variablen

- 32 Register ausreichend, um Befehle direkt umzusetzen
- Abstraktion: Compiler verbirgt „Register“

Hochsprache
wie z.B. C

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

Assemblersprache
mit Variablen:

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

Variablen stehen in Registern $s0-$s4

Assemblersprache

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Ablauf:

	0	1	2	3
$s0				(g+h)-(i+j)
$s1	g			
$s2	h			
$s3	i			
$s4	j			
$t0		g+h		
$t1			i+j	

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

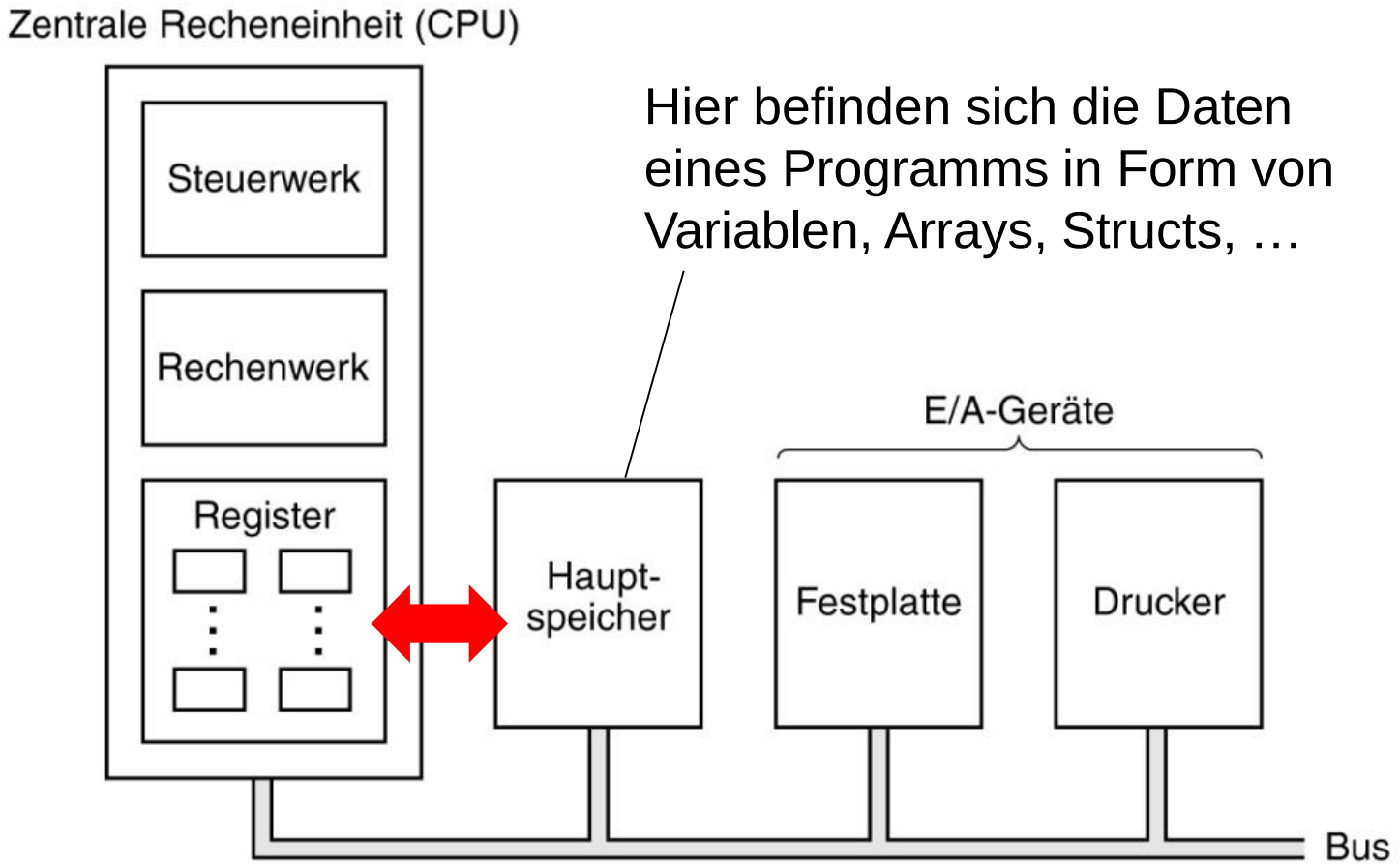
Kapitel 2: Befehle: Die Sprache des Rechners

- 2.1 Befehlssatz: Was ist das?
- **2.2 Befehle des MIPS Befehlssatzes**
  - 2.2.1 Arithmetische Befehle
  - **2.2.2 Grundlegende Datentransferbefehle**
  - 2.2.3 Konstanten als Operanden
  - 2.2.4 Vorzeichen in Binärzahlendarstellung
  - 2.2.5 Weitere Datentransferbefehle
- 2.3 Darstellungen von Befehlen im Rechner
- 2.4 Logische Operationen
- 2.5 Kontrollstrukturen
- 2.6 MIPS Assembler und MARS Simulator
- …

Datentransferbefehle (Data Transfer Instructions)

Datentransferbefehle: Daten vom Hauptspeicher in Register kopieren und umgekehrt

- 32 Register können nicht alle Variablen halten
- ALU kann nicht auf Hauptspeicherinhalten rechnen
- Spezielle Datentransferbefehle um Daten in zu kopieren



Aufbau Hauptspeicher

Der Inhalts des Speichers wird in der Vorlesung und im MARS-Simulator meist nicht linear sondern in Gruppen von 4 Bytes, in Worten, dargestellt.

Bytes im Hauptspeicher ab Adresse 0

Adresse	Byte
…	…
7	111
6	13
5	255
4	254
3	0
2	0
1	14
0	7

Darstellung des Speicherinhalts in Worten (little endian)

				12
			…	8
111	13	255	254	4
0	0	14	7	0

Wort

Byte 0: 0000 0111

Byte-Adressen

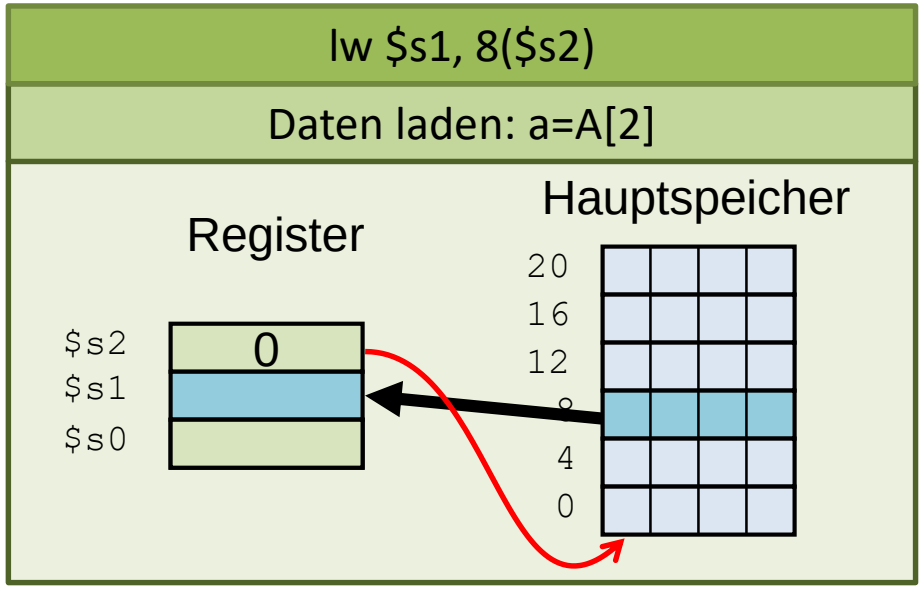
			…
7	6	5	4
3	2	1	0

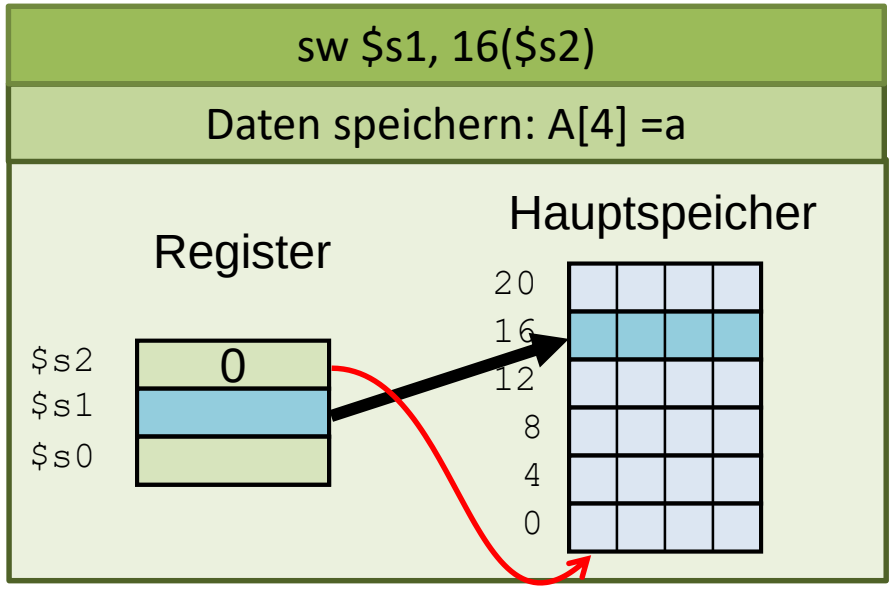
In der Vorlesung und im MARS-Simulator wird die little-endian Notation verwendet. Das am wenigsten signifikante Bit eines Bytes oder Worts steht an der niedrigsten Speicheradresse.

Datentransferbefehle: Zugriff auf Hauptspeicher

Datentransferbefehle: load, store

- Speicheradresse: Registerinhalt + Konstante
- Hauptspeicher enthält Array „int A[6]“ in Speicherplätzen 0-23
- Register $s2 enthält Basisadresse von Array A
  - Basisadresse ist die Speicheradresse von Byte 0
- Register $s1 wird für Variable „a“ verwendet

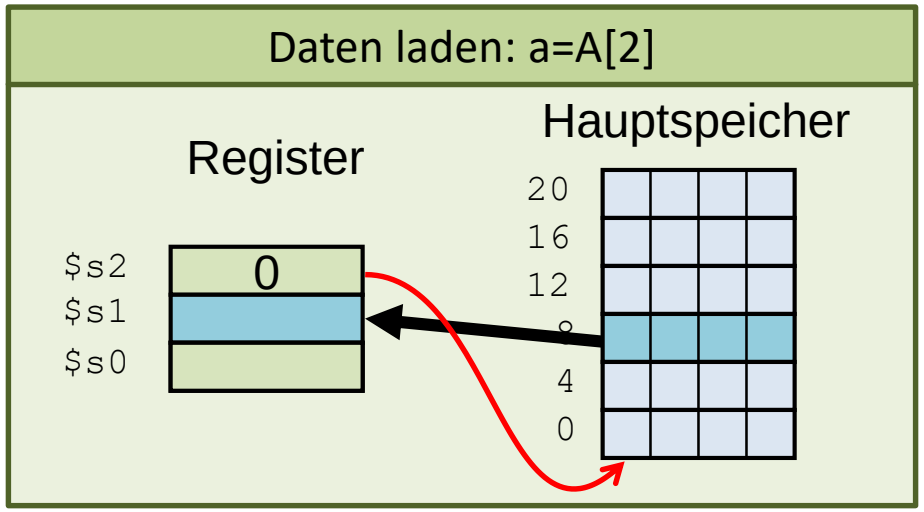


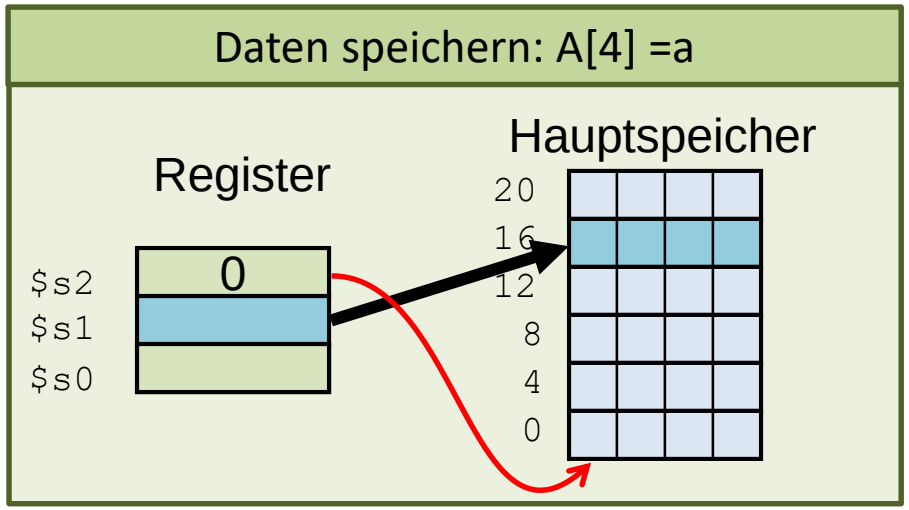


Datentransferbefehle: Syntax

Speicheradresse: Registerinhalt + Konstante

- Format: Konstante(Registerinhalt)
- Wichtig: Zugriff auf Hauptspeicher **NUR** über Datentransferbefehle





Zusammensetzung eines Datentransferbefehls:

```
lw $s1,8($s2)      $s1=A[$s2+8]
sw $s1,16($s2)     A[$s2+16]=$s1
```

Basisadresse von Array A
konstanter Offset

lw: load word
sw: store word
$s1: Zielregister
$s2+16: Speicheradresse von A[4]

Datentransferbefehle – Aufbau Hauptspeicher

Der Zugriff auf den Hauptspeicher erfolgt über Byteadressen. Das Laden oder Speichern eines Worts bezieht sich auf vier Bytes ab der angegebenen Adresse.

Bytes im Hauptspeicher ab Adresse 0

Adresse	Byte
...	...
7	111
6	13
5	255
4	254
3	0
2	0
1	14
0	7

Bytes im Hauptspeicher ab Adresse 48

Adresse	Byte
...	...
55	111
54	13
53	255
52	254
51	0
50	0
49	14
48	7

in Register $s2 sei 48 gespeichert

lw $t0, 0($s2) ✓

lw $t0, 2($s2) ✗

Darstellung des Speicherinhalts in Worten

				Adresse
			60	
		...	56	
111	13	255	254	52
0	0	14	7	48

Wort

Das Laden von Worten über Wortgrenzen hinweg wird in MIPS nicht unterstützt. Die Wortadresse $s2+offset(hier 2) muss ein Vielfaches on 4 Bytes sein.

Datentransferbefehle

Hauptspeicheradresse basiert auf Bytes

- Array A besteht aus Words

Beispiel:

```
g=h+A[8];
```

```
lw  $t0, 8($s3)
add $s0, $s1, $t0
```

Was ist falsch?

Offset 8 adressiert 8. Bytes also 2. Wort

```
g=h+A[8];
```

```
lw  $t0, 32($s3)
add $s0, $s1, $t0
```

	Instruktionsnummer		
	0	1	2
$s0			h+A[2]
$s1	g		
$s2	h		
$s3	Basisadresse von A		
$t0		A[2]	

	Instruktionsnummer		
	0	1	2
$s0			h+A[8]
$s1	g		
$s2	h		
$s3	Basisadresse von A		
$t0		A[8]	

Übung: Datentransfer

- Tragen Sie die Änderungen ein, die durch die folgenden Instruktionen in Registern und Speicher erfolgen:

```
lw $t0,5($s0)
lw $t1,-16($s1)
sw $t2,8($s2)
```

Register	Inhalt	Änderung
$s0	7	
$s1	20	
$s2	56	
$t0	1	4
$t1	2	256
$t2	257	

Speicherinhalt (Bytes)				Adresse
0	0	0	0	60
0	0	0	0	56
111	13	255	254	52
0	0	14	7	48
				…
0	0	0	4	12
0	0	0	6	8
0	0	1	0	4
255	255	255	255	0
Änderung				
Speicherinhalt (Bytes)				Adresse
0	0	1	1	64

Quiz: Berechnung mit Wert aus Array und Zurückspeichern

- Hauptspeicher enthält Array „`int A[12]`“
- Register `$s3` enthält Basisadresse von Array `A`
- Register `$s2` enthält Wert der Variablen „h“
- „Kompiliere“ folgenden C Code: `A[12]=h+A[8];`

- Assembler-Code:

```
lw   $t0, 32($s3)     % Speicheradresse ist Basisadresse
                      % von Array A in Register $s3
                      % plus 8 Words oder 32 Bytes
add  $t0, $s2, $t0
sw   $t0, 48($s3)     % Speicheradresse ist Basisadresse
                      % von Array A in Register $s3
                      % plus 12 Words oder 48 Bytes
```

Vorsicht bei der Adressierung

- Hauptspeicheradresse: Bytes
- Array A besteht aus Words

Vorlesungsinhalt

Konstanten als Operanden

Konstanten als Operanden ersparen das Schreiben von Konstanten in Register

- Neben Registern können auch Konstanten in arithmetischen Befehlen als Operanden vorkommen
  - aber: der Befehl heißt dann nicht *add* sondern *addi*

- Verwendung:

```
addi $s3, $s3, 4   # $s3=$s3+4
                   # Konstante 4 zum Register $s3 addieren und in Register $s3
                   # schreiben
```

Frage: Es gibt keinen Befehl *subi* ! Warum nicht?

- negative Konstanten werden unterstützt *addi* kann verwendet werden

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

- 2.1 Befehlssatz: Was ist das?
- **2.2 Befehle des MIPS Befehlssatzes**
  - 2.2.1 Arithmetische Befehle
  - 2.2.2 Grundlegende Datentransferbefehle
  - 2.2.3 Konstanten als Operanden
  - **2.2.4 Vorzeichen in Binärzahlendarstellung**
  - 2.2.5 Weitere Datentransferbefehle
- 2.3 Darstellungen von Befehlen im Rechner
- 2.4 Logische Operationen
- 2.5 Kontrollstrukturen
- 2.6 MIPS Assembler und MARS Simulator
- …

Zahlenformate: „Signed“ und „Unsigned“

Unterschiedliche Verwendung von „signed“ und „unsigned“ Zahlendarstellungen

- Zahlenformate:
  - „signed“: mit Vorzeichen
  - „unsigned“: ohne Vorzeichen (positiv)

- Warum brauchen wir Zahlenformate mit und ohne Vorzeichen?
  - Beispiel Speicheradresse als Registerinhalt:
    - es gibt keine negativen Adressen
    - „unsigned“ Format verdoppelt adressierbaren Speicherbereich
  - Beispiel Konstante:
    - durch Zahlenformat „signed“ für Konstanten fallen die Befehle *addi* und *subi* zusammen

Wiederholung: „Signed“

Erstes Bit ist Vorzeichenbit und unterscheidet positive und negative Zahlen

- Unsigned Word (32 bit): $0 \ldots 2^{32} - 1$
- Signed Word (32 bit): Zweierkomplementdarstellung
  - binäre Zahlen mit führenden Nullen: positiv
  - binäre Zahlen mit führenden Einsen: negativ

$$-2^{31} \ldots 2^{31} - 1$$

```
1111 1111 1111 1111 1111 1111 1111 1110 = -2
1111 1111 1111 1111 1111 1111 1111 1111 = -1
0000 0000 0000 0000 0000 0000 0000 0000 =  0
0000 0000 0000 0000 0000 0000 0000 0001 =  1
```

Was ist -14 (Signed Byte)?

```
 14 = 0000 1110
-14 = 1111 0010
```

Umrechnung Binär-Dezimal

Wie lautet der Dezimalwert der folgenden 32-Bit-Zweikomplementzahlen?

1111	1111	1111	1111	1111	1111	1001	0011	?
?	?	?	?	?	?	?	?	-112

- Berechnungsvorschrift

$$X_{dec} = X_{bin,31} \cdot \left(-2^{31}\right) + \sum_{i=0}^{30} X_{bin,i} \cdot 2^{i}$$

- Pragmatische Rechnung per Hand:
  - „-“ -> „+“: erst invertieren, dann „1“ addieren
  - „+“ -> „-“: erst „1“ subtrahieren, dann invertieren

Umrechnung Binär-Dezimal

Wie lautet der Dezimalwert der folgenden 32-Bit-Zweikomplementzahlen?

Original	1111	1111	1111	1111	1111	1111	1001	0011		-109
Invertiert	0000	0000	0000	0000	0000	0000	0110	1100		
+ „1“	0000	0000	0000	0000	0000	0000	0110	1101	64+32+8+4+1	109
+ „1“	1111	1111	1111	1111	1111	1111	1001	0000		-112
Invertiert	1111	1111	1111	1111	1111	1111	1000	1111		
Original	0000	0000	0000	0000	0000	0000	0111	0000	64+32+16	112

- Berechnungsvorschrift

$$X_{dec} = X_{bin,31} \cdot \left(-2^{31}\right) + \sum_{i=0}^{30} X_{bin,i} \cdot 2^{i}$$

- Pragmatische Rechnung per Hand:
  - „-“ -> „+“: erst invertieren, dann „1“ addieren
  - „+“ -> „-“: erst „1“ subtrahieren, dann invertieren

Test: http://www.binaryconvert.com/result_signed_int.html?decimal=045049052

Vorlesungsinhalt

Weitere Datentransferbefehle

Datentransferbefehle für Bytes und Halb-Worte

	Instruktion	Bedeutung
Laden	`lb rt, address`	Lade Byte von der Addresse *address* in das Register *rt*
	`lh rt, address`	Lade Half-Word von der Addresse *address* in das Register *rt*
Speichern	`sb rt, address`	Speichere unterstes Byte des Registers *rt* and die Addresse *address*
	`sh rt, address`	Speichere unterstes Half-Word des Registers *rt* and die Addresse *address*

Weitere Datentransferbefehle

Datentransferbefehle für Bytes und Halb-Worte

	Instruktion	Bedeutung
Laden	`lb rt, address`	Lade Byte von der Addresse *address* in das Register *rt*
	`lh rt, address`	Lade Half-Word von der Addresse *address* in das Register *rt*
Speichern	`sb rt, address`	Speichere unterstes Byte des Registers *rt* and die Addresse *address*
	`sh rt, address`	Speichere unterstes Half-Word des Registers *rt* and die Addresse *address*

Datentransferbefehle für „Unsigned/Signed“

- Hochwertige Bits werden beim Laden als „Unsigned“ immer mit Nullen gefüllt
- Beim Laden als „Signed“ hängt das vom Vorzeichen ab

Weitere Datentransferbefehle

Datentransferbefehle für Bytes und Halb-Worte

Datentransferbefehle für „Unsigned/Signed“

- Hochwertige Bits werden beim Laden als „Unsigned“ immer mit Nullen gefüllt
- Beim Laden als „Signed“ hängt das vom Vorzeichen ab

	Instruktion	Bedeutung
Laden	**lb** *rt, address*	Lade Byte von der Addresse *address* in das Register *rt*. Das Byte ist sign-extended.
	lbu *rt, address*	Lade Byte von der Addresse *address* in das Register *rt*.
	lh *rt, address*	Lade Half-Word von der Addresse *address* in das Register *rt*. Das Byte ist sign-extended.
	lhu *rt, address*	Lade Half-Word von der Addresse *address* in das Register *rt*.
	lw *rt, address*	Lade Word von der Addresse *address* in das Register *rt*.
Speichern	**sb** *rt, address*	Speichere unterstes Byte des Registers *rt* and die Addresse *address*.
	sh *rt, address*	Speichere unterstes Half-Word des Registers *rt* and die Addresse *address*.
	sw *rt, address*	Speichere Inhalt des Registers *rt* and die Addresse *address*.

Laden von Signed und Unsigned Bytes

Inhalt des Byte Arrays A, dessen Speicheradresse in Register $s2 steht:

```
A[3]=14;
A[6]=-14
```

Laden der Bytes (**lb**) von A[3] und A[6] in Register $t1 und $t2:

```
lb  $t1,3($s2)
lb  $t2,6($s2)
```

Was steht in den Registern?

$t1	0000	0000	0000	0000	0000	0000	0000	1110
$t2	1111	1111	1111	1111	1111	1111	1111	0010

Laden als Unsigned Bytes (**lbu**) :

```
lbu $t1,3($s2)
lbu $t2,6($s2)
```

Was steht in den Registern?

$t1	0000	0000	0000	0000	0000	0000	0000	1110
$t2	0000	0000	0000	0000	0000	0000	1111	0010

Übersicht der Instruktionen

	Instruktion	Bedeutung
Arithmetik	**add** *rd, rs, rt*	Register *rd* = Register *rs* + Register *rt*
	addi *rd, rs, imm*	Register *rd* = Register *rs* + Konstante *imm*
	sub *rd, rs, rt*	Register *rd* = Register *rs* - Register *rt*
Laden	**lb** *rt, address*	Lade Byte von der Addresse *address* in das Register *rt*. Das Byte ist sign-extended.
	lbu *rt, address*	Lade Byte von der Addresse *address* in das Register *rt*.
	lh *rt, address*	Lade Half-Word von der Addresse *address* in das Register *rt*. Das Byte ist sign-extended.
	lhu *rt, address*	Lade Half-Word von der Addresse *address* in das Register *rt*.
	lw *rt, address*	Lade Word von der Addresse *address* in das Register *rt*.
Speichern	**sb** *rt, address*	Speichere unterstes Byte des Registers *rt* and die Addresse *address*.
	sh *rt, address*	Speichere unterstes Half-Word des Registers *rt* and die Addresse *address*.
	sw *rt, address*	Speichere Inhalt des Registers *rt* and die Addresse *address*.

Quiz

```
addi $s0, $zero, 4   # s0=4
lw $s1, 0($s0)       # s1=18   s1=Mem[4-7]
lw $s2, 4($s0)       # s2=6    s2=Mem[8-11]
add $s1, $s1, $s1    # s1=36
add $s1, $s1, $s2    # s1=42
addi $s1, $s1, 1     # s1=43
sw $s1, 0($s0)       # Mem[4-7]=43
```

Hauptspeicher zu Beginn:

Adresse	Inhalt (Word)
	⋮
12	33
8	6
4	18
0	97563

Hauptspeicher am Ende:

Adresse	Inhalt (Word)
	⋮
12	33
8	6
4	43
0	97563