```
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

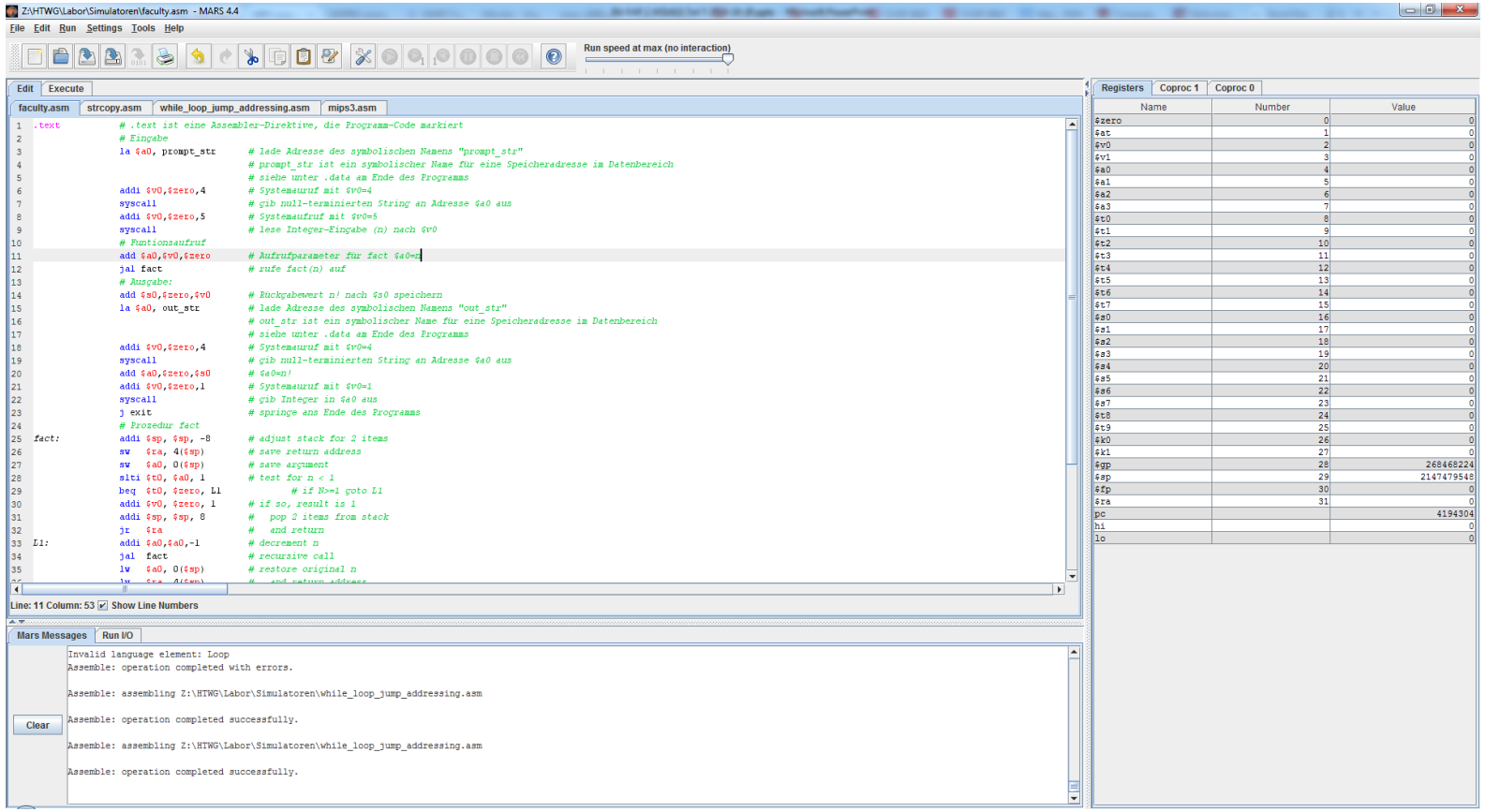
Befehle: Die Sprache des Rechners

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

Vorlesungsinhalt

MARS – MIPS Assembler and Runtime Simulator

- MARS: http://courses.missouristate.edu/KenVollmar/MARS/
  - freier MIPS Simulator
  - wird in Vorlesung und Laborübungen verwendet



Assembler

Assemblersprache

- Symbolische Repräsentation der Maschineninstruktionen
- Vereinfachte Anwendung durch
  - Pseudoinstruktionen: Instruktionen, die der Befehlssatz nicht enthält, und die vom Assembler in gültige Maschineninstruktionen übersetzt werden
    - Beipiele:

```
Pseudo-Instruktion        Maschinensprache
move $t0, $t1       ⟹    add $t0, $zero, $t1

blt $s0, $s1, L1    ⟹    slt $t0, $s0, $s1
                          bne $t0, $zero, L1
```

    - Assemblersprache kann z.B. in `addi` „große Konstanten“ zulassen, die der Assembler durch `lui` und `ori` ersetzt
  - Zahlendarstellungen: Neben der binären und dezimalen Zahlen-repräsentationen ist auch die hexadezimale Darstellung gebräuchlich, die v.a. bei Adressberechnungen vorteilhaft ist.

Vorlesungsinhalt

Assembler: Direktiven

Direktiven (Directives)

- Anweisung, die dem Assembler helfen Code zu interpretieren, ohne selbst in Instruktionen übersetzt zu werden
- Übersicht der MIPS Assembler-Direktiven auf Moodle

Data Layout Directives

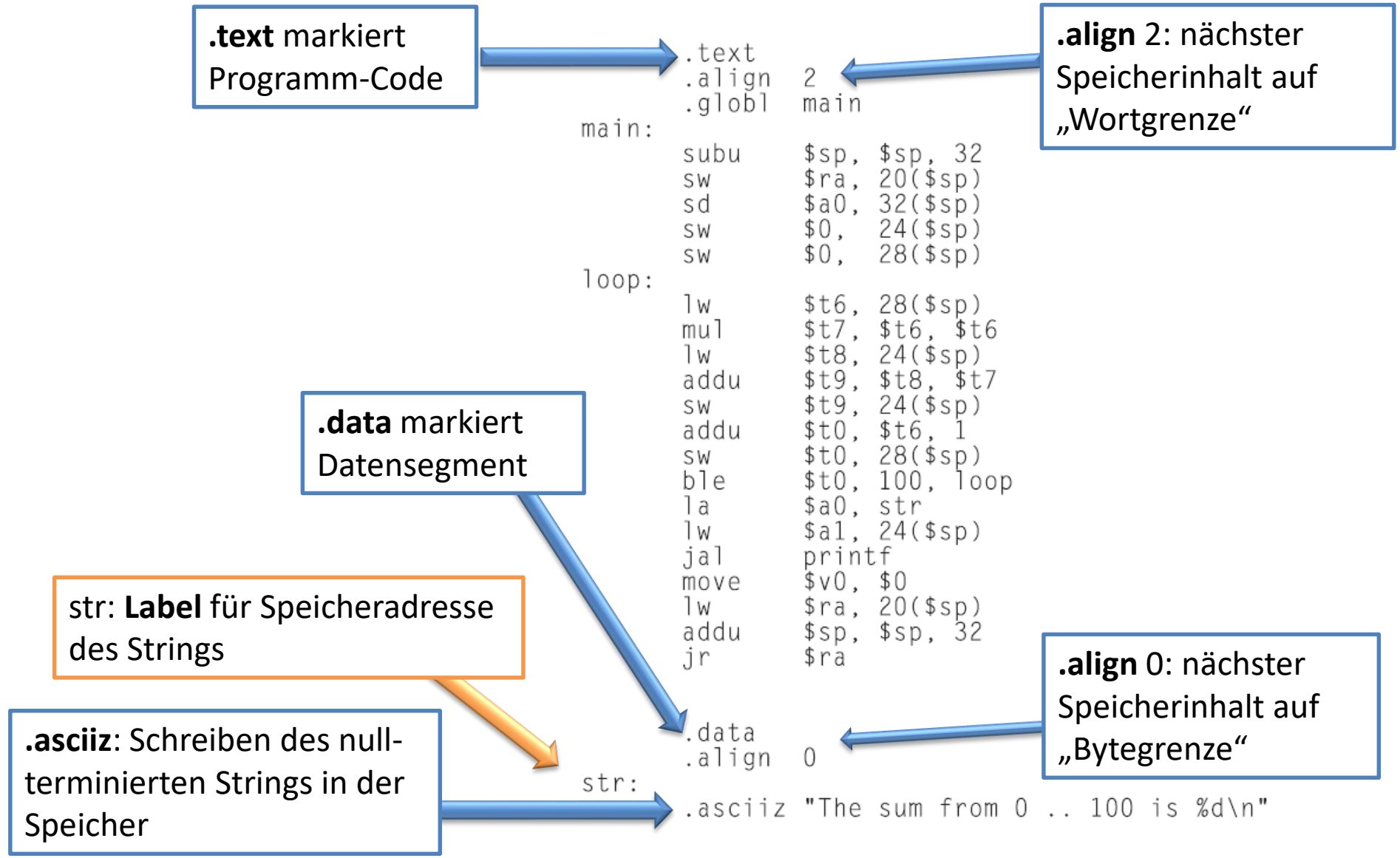
- einfache und kompakte Beschreibung von Daten
- Beispiel:

```
.asciiz „The sum from 0 .. 100 ist %d\n“
```

legt Byte der ASCII-Zeichen im Speicher ab

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

Weitere Direktiven und Labels für Daten



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
  - 2.6.1 MARS Simulator
  - 2.6.2 Direktiven
  - **2.6.3 Makros**
  - 2.6.4 Systemaufrufe

Makros

In der Assemblersprache können Makros definiert werden.

- Makros sind „Abkürzungen„ für eine bestimmte Code-Sequenz und werden vom Assembler „eins-zu-eins“ in Code übersetzt
- Arten von Makros sind
  - einfache Programmsequenzen
  - Programmsequenzen mit Parameterübergabe

Beispiel eines Makros

```
            .data
int_str:    .asciiz "%d"
            .text
            la $a0, int_str       # Load string address
                                  # into first arg
            mov $a1, $7           # Load value into second arg
            jal printf            # Call the printf routine
```

als Makro

```
            .data
int_str:    .asciiz "%d"
            .text
            .macro print_int($arg)
                 la $a0, int_str       # Load string address
                                       # into 1st arg
                 mov $a1, $arg         # Load value into 2nd arg
                 jal printf            # Call printf routine
            .end_macro
print_int($7)
```

Assembler: Makros expandieren

```
            .data
int_str:    .asciiz "%d"
            .text
            .macro print_int($arg)
                 la $a0, int_str        # Load string address
                                        # into 1st arg
                 mov $a1, $arg          # Load value into 2nd arg
                 jal printf             # Call printf routine
            .end_macro
print_int($7)
```

`print_int($7)` expandiert zu:

```
la $a0, int_str
mov $a1, $7
jal printf
```

`print_int($t0)` expandiert zu:

```
la $a0, int_str
mov $a1, $t0
jal printf
```

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
  - 2.6.1 MARS Simulator
  - 2.6.2 Direktiven
  - 2.6.3 Makros
  - **2.6.4 Systemaufrufe**

Eingabe und Ausgabe mit System-Calls

- Systemaufrufe in MIPS (MARS Simulator)
  - Systemaufrufe erfolgen über den Befehl `syscall`
  - Register `$v0` legt die Art des Systemaufrufs fest
  - Register `$a0-$a3` enthalten Parameter
  - Register `$v0` enthält Rückgabewerte
- Einige wichtige Systemaufrufe
  - Eingabe:
    - Integer: Aufruf mit $v0=5, Rückgabe der Eingabe in $v0
    - String: Aufruf mit
      - $v0=8
      - $a0=Adresse, an die der eingegebene String gespeichert werden soll
      - $a1=maximale String-Länge
  - Ausgabe:
    - Integer: Aufruf mit $v0=1, Integer in $a0
    - String: Aufruf mit $v0=4, Adresse des null-terminierten Strings in $a0
- Übersicht der System-Calls:
  - http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html

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**
  - 2.7.1 Unterstützung für Strings in MIPS
  - 2.7.2 Zusätzliche Adressierungsarten
  - 2.7.3 Instruktionen zur Synchronisierung

Zeichensätze

8-Bit Zeichensätze (ein Zeichen in einem Byte kodiert)

- ASCII: 128 Zeichen
  - 95 dargestellte Zeichen, 33 Steuerzeichen
- Latin-1: 256 Zeichen
  - ASCII +96 zusätzliche dargestellte Zeichen

Unicode: 32-Bit Zeichensatz

- Java, C++ wide characters, …
- enthält die meisten Alphabete, weitere Symbole, etc.
- UTF-8, UTF-16: Kodierungen unterschiedlicher Länge
  - UTF: Universal Character Set (UCS) Transmission Format
  - 1 Byte: 128 ASCII-Zeichen
  - 2 Bytes: z.B. Umlaute

Strings in MIPS

Byteweise Zugriffe und Zeichen

- Daten werden nicht immer als Zahlen interpretiert. In der heutigen Praxis ist die Verarbeitung von Texten (Zeichenketten) mindestens genau so wichtig.
- Zeichen werden durch einzelne Bytes repräsentiert. Instruktionssätze verfügen daher über Byte-Lade- und Speicheroperationen.
- Ladeoperation

  ```
  lb $t0, 0($sp)
  ```

  Lädt ein Byte aus dem Speicher und plaziert es in die rechts gelegenen (niederwertigen) Bits eines Registers
- Speicheroperation

  ```
  sb $to, 0($a1)
  ```

  speichert die niederwertigen acht Bits eines Registers in die bezeichnete Speicherposition

Beispiel: Text kopieren

C-Sourcecode:

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != 0) /* copy and test byte */
        i = i + 1;
}
```

Prozedur strcpy:

- Adressen von x und y in $a0, $a1
- i in $s0
- Beginn der Prozedur: Stackpointer und Register sichern

```
strcpy:   addi $sp, $sp, -4      # adjust stack for 1 item
          sw   $s0, 0($sp)       # save $s0
```

While-Schleife:

Adressen von x und y in $a0, $a1
i in $s0

```
i = 0;
while ((x[i] = y[i]) != 0) /* copy and test byte */
        i = i + 1;
```

	add $s0, $zero, $zero	# i = 0
L1:	add $t1, $s0, $a1	# addr of y[i] in $t1
	lbu $t2, 0($t1)	# $t2 = y[i]
	add $t3, $s0, $a0	# addr of x[i] in $t3
	sb $t2, 0($t3)	# x[i] = y[i]
	beq $t2, $zero, L2	# exit loop if y[i] == 0
	addi $s0, $s0, 1	# i = i + 1
	j L1	# next iteration of loop
L2:	lw $s0, 0($sp)	# restore saved $s0
	addi $sp, $sp, 4	# pop 1 item from stack
	jr $ra	# and return

Vorlesungsinhalt

Mehr Komfort beim Datenzugriff: Adressierungsarten

Mängel der derzeitigen Instruktionen

- Die Verwendung von Konstanten ist unhandlich
  - Konstante sind auf 16 Bit beschränkt -32768 bis 32767

- Die Handhabung von Verzweigungsadressen ist unzulänglich
  - Das ist bisher nicht aufgefallen, da wir immer auf Assembler-Ebene mit symbolischen Sprungzielen operiert haben!
  - Für Verzweigungsziele sind nur Adressbereiche im Rahmen von 16 Bits erreichbar
  - Als Sprungziele nur 16 Bit lange Adressen möglich

Große Konstanten

Wie haben wir bislang Konstanten (Zahlen) in ein Register bekommen?

- addi, ori, etc. (maximal 16 Bit)
- Alternative: Laden aus dem Speicher
  - umständlich zu laden
  - müssen erst in den Speicher geschrieben werden?

Neuer Befehl:

- **`lui`**: load upper [half word] immediate:

```
lui reg, constant      # lädt eine Konstante in „obere“ 16 Bits  eines Registers
```

- Maschinensprache:

OP-Code	Register	Unused	Constant
6 Bits	5 Bits	5 Bits	16 Bits

Beispiel: Große Konstante

- Wie laden wir jetzt z.B. 98304 in Register $t0?

0000	0000	0000	0001	1000	0000	0000	0000	=98304

- „Upper half“ mit **`lui`**

```
lui $t0, 1
```

$t0:	0000	0000	0000	0001	0000	0000	0000	0000	=65536

- „Lower half“ mit **`ori`**

```
ori $t0, $t0, 32768
```

```
ori $t0,$t0, -32767
```

Weite bedingte Sprünge???

- … und was, wenn es doch mal weitergehen soll?

```
0x0400000 beq $s0, Ss1, L1
…
0x0800000 L1:
```

- … dann springen wir mittels Vergleich und „jump“

```
0x0400000 bne $s0, Ss1, L2
0x0400004 j   L1
L2:
…
0x0800000 L1:
```

- … darum müssen wir uns beim Programmieren nicht kümmern, das macht der Assembler

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**
  - 2.7.1 Unterstützung für Strings in MIPS
  - 2.7.2 Zusätzliche Adressierungsarten
  - **2.7.3 Instruktionen zur Synchronisierung**

Synchronisation – Data Race

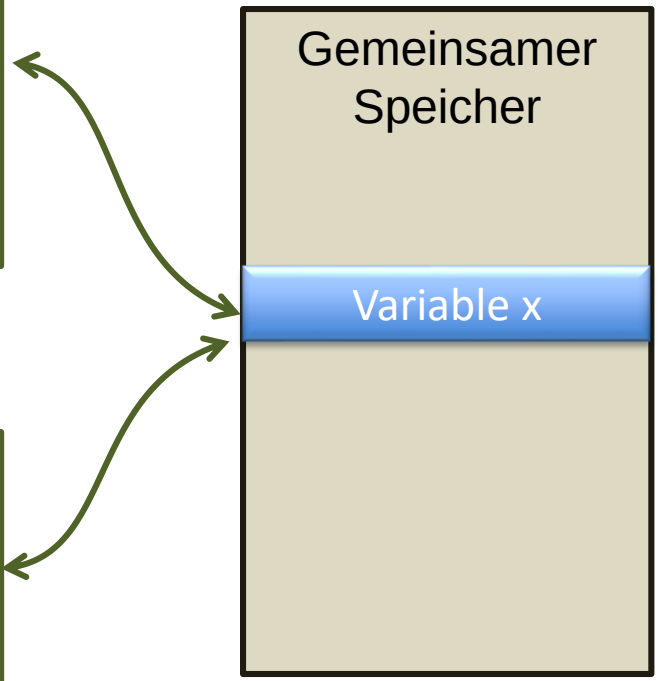
Prozessor 1: berechne x=x+2

```
lw  $t0, 0($s0)        # lade x nach $t0
addi   $t0, $t0, 2     # $t0=$t0+2
sw  $t0, 0($s0)        # speichere $t0 nach x
```

nicht synchronisiert

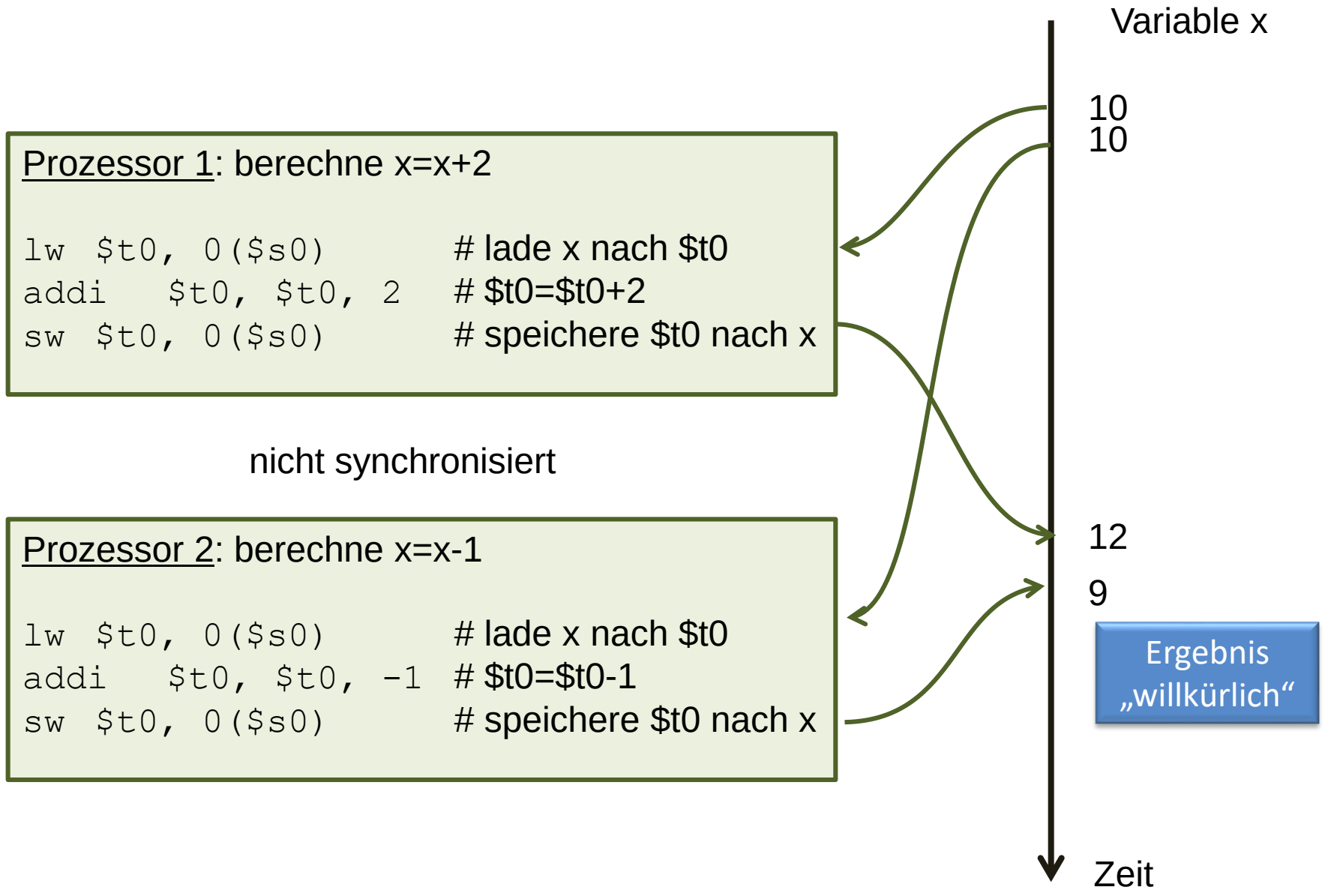
Prozessor 2: berechne x=x-1

```
lw  $t0, 0($s0)        # lade x nach $t0
addi   $t0, $t0, -1    # $t0=$t0-1
sw  $t0, 0($s0)        # speichere $t0 nach x
```



x sei zu Beginn=10
Kommt am Ende immer x=11 raus?

Synchronisation – Data Race



Atomare Befehle

Zwei Prozessoren greifen auf gleichen Speicherbereich zu

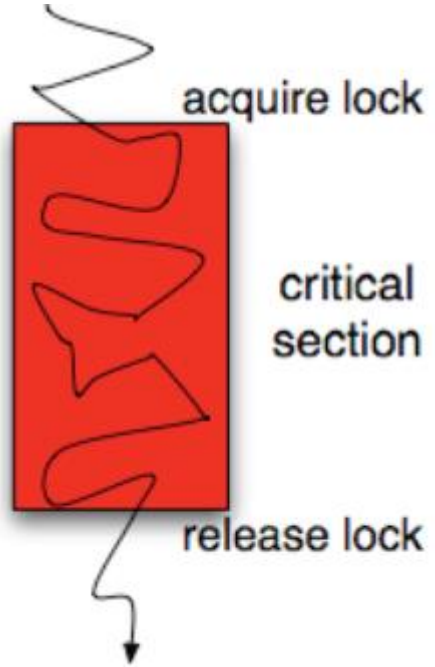
- wenn P1 und P2 nicht synchronisiert sind, kommt es zu einem “Data Race”
- Ergebnis hängt von der “willkürlichen” Reihenfolge der Zugriffe ab
- Definition einer Variable „Lock“, die Speicherzugriff regelt
  - wer Lock auf „1“ setzen kann (acquire lock), kann auf Daten zugreifen

Hardware Unterstützung für „Lock“ benötigt

- Atomarer Lese-/Schreib-Zugriff auf „Lock“
- Kein anderer Prozess kann „Lock“ zwischen Lesen und Schreiben ändern
- „Lock“ kann nur gesetzt werden, wenn „Lock“ nicht gesetzt ist

Umsetzung als

- atomarer Austausch (atomic swap):
  - register ↔ memory
  - schwierig, da Prozessor gleichzeitiges Lesen- und Schreiben nicht unterstützt
- atomares Befehlspaar (atomic pair of instructions)

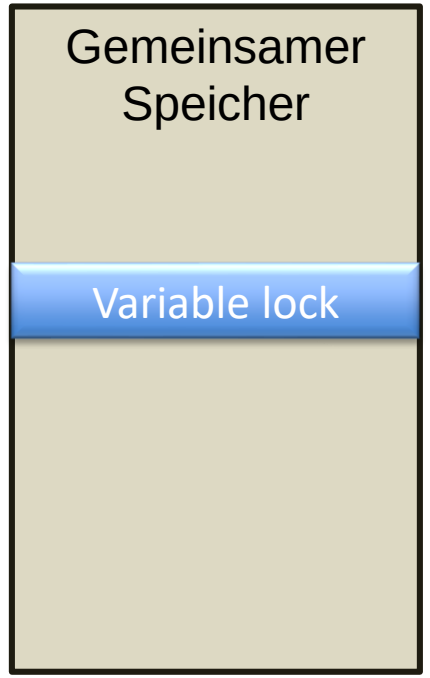


Atomic Swap

MIPS ISA enthält keinen Atomic Swap Befehl

- Hier wird das Problem über ein „atomic pair“ gelöst.

```
swap $t1, lock    # kopiere Inhalt von lock nach $t1 und kopiere Inhalt
                  # von $t1 nach lock
```



- Swap ist atomar, d.h. während des swap wird jeglicher Speicherzugriff anderer Prozesse verzögert bis swap vollständig ausgeführt wurde
- Zugriff gewährt bei $t1=0

Beispiel		
	$t1	lock
Vor Ausführung von swap	1	0
Nach Ausführung von swap	0 ✓	1

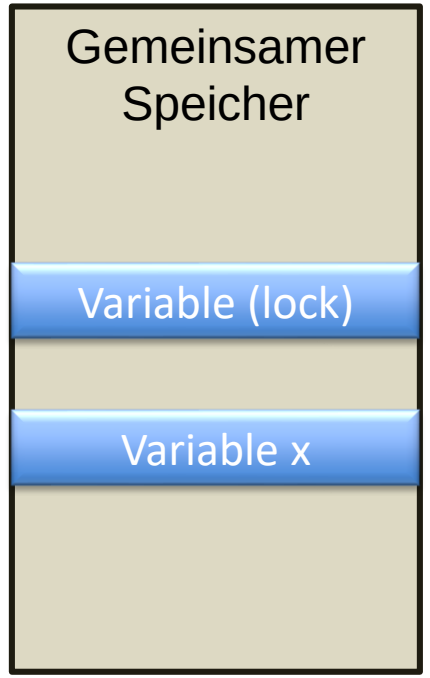
Atomic Swap - Beispiel

Prozessor 1: berechne x=x+2

```
        addi   $t1, $zero, 1     # $t1=1
loop:   swap $t1, lock           # tausche $t1 und lock
        bne $t1, zero, loop      # nochmal wenn lock
                                 # gesetzt war
        lw  $t0, 0($s0)          # lade x nach $t0
        addi   $t0, $t0, 2       # $t0=$t0+2
        sw  $t0, 0($s0)          # speichere $t0 nach x
        swap   $t1, lock         # lock freigeben
```

Prozessor 2: berechne x=x-1

```
        addi   $t1, $zero, 1     # $t1=1
loop:   swap $t1, lock           # tausche $t1 und lock
        bne $t1, zero, loop      # nochmal wenn lock
                                 # gesetzt war
        lw  $t0, 0($s0)          # lade x nach $t0
        addi   $t0, $t0, -1      # $t0=$t0-1
        sw  $t0, 0($s0)          # speichere $t0 nach x
        swap   $t1, lock         # lock freigeben
```



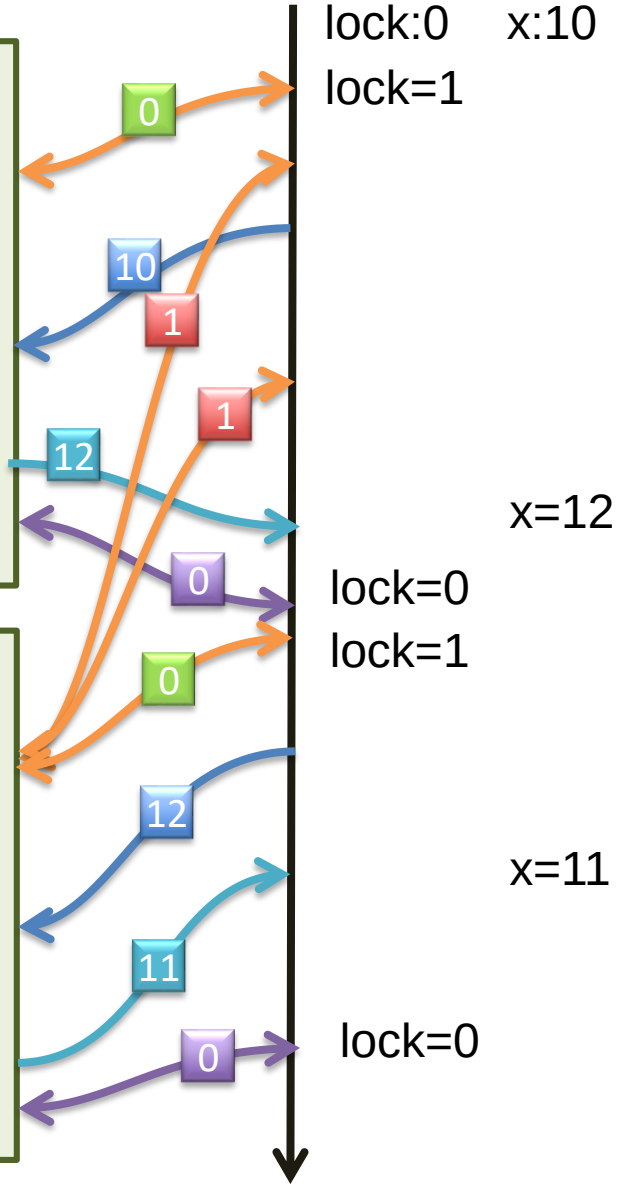
Atomic Swap - Beispiel

Prozessor 1: berechne x=x+2

```
        addi   $t1, $zero, 1     # $t1=1
loop:   swap $t1, lock           # tausche $t1 und lock
        bne $t1, zero, loop      # nochmal wenn lock
                                 # gesetzt war
        lw  $t0, 0($s0)          # lade x nach $t0
        addi   $t0, $t0, 2       # $t0=$t0+2
        sw  $t0, 0($s0)          # speichere $t0 nach x
        swap   $t1, lock         # lock freigeben
```

Prozessor 2: berechne x=x-1

```
        addi   $t1, $zero, 1     # $t1=1
loop:   swap $t1, lock           # tausche $t1 und lock
        bne $t1, zero, loop      # nochmal wenn lock
                                 # gesetzt war
        lw  $t0, 0($s0)          # lade x nach $t0
        addi   $t0, $t0, -1      # $t0=$t0-1
        sw  $t0, 0($s0)          # speichere $t0 nach x
        swap   $t1, lock         # lock freigeben
```



MIPS Lösung: Load Linked and Store Conditional

Load Linked

- Inhalt der Speicherstelle `0($s1)` wird in Register $t1 geladen und Hardware „notiert“ Speicherzugriffe auf 0($s1)

```
ll $t1, 0($s1)    # load linked
```

Store conditional

- Fallunterscheidung:
  1. Keine Veränderung seit „load linked“: Speichere Inhalt von $t0 nach 0($s1) und setze $t0 auf 1
  2. Veränderung seit „load linked“: $t0 wird nicht gespeichert, aber auf 0 gesetzt

```
sc $t0, 0($s1)    # store conditional
```

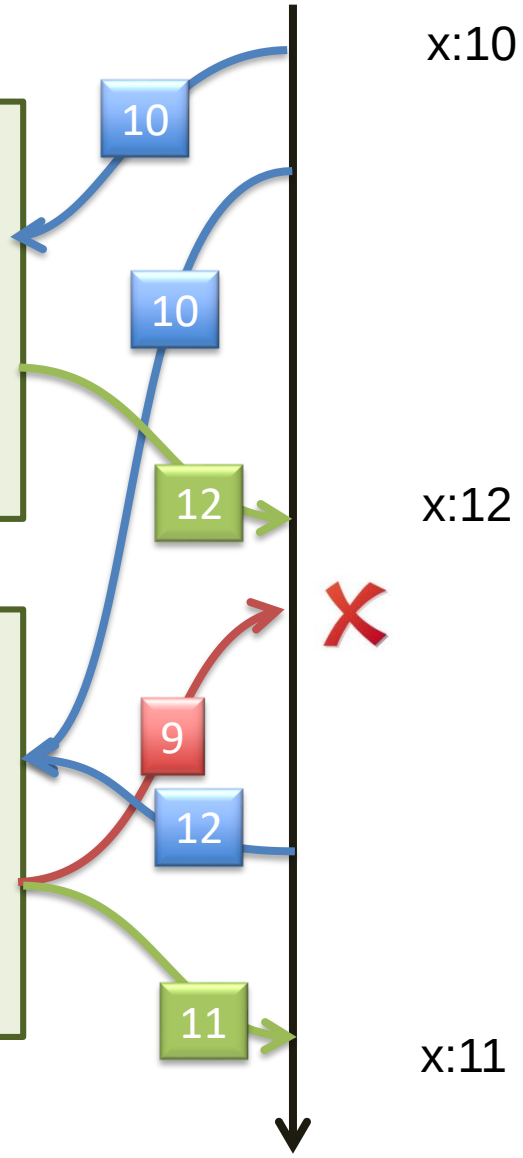
Linked Loaded / Store Conditional - Beispiel

Prozessor 1: berechne x=x+2

```
loop:  ll $t0, 0($s0)        # $t0=x
       addi   $t0, $t0, 2    # $t0=$t0+2
       sc $t0, 0($s0)        # x=$t0
       beq $t0, zero, loop   # nochmal bei Fehler
```

Prozessor 2: berechne x=x-1

```
loop:  ll $t0, 0($s0)        # $t0=x
       addi   $t0, $t0, -1   # $t0=$t0-1
       sc $t0, 0($s0)        # x=$t0
       beq $t0, zero, loop   # nochmal bei Fehler
```



Quiz

Wir realisieren „swap“ mit `ll` und `sc`

- Swap: tausche Registerinhalt mit Speicherinhalt

```
swap $s0, 0($s1)
```