



LUDWIG-  
MAXIMILIANS-  
UNIVERSITÄT  
MÜNCHEN



 mobile and  
distributed systems group



# Grundlagen der Assemblerprogrammierung unter SPIM

Grundlagen Technische Informatik  
im Sommersemester 2015

Lorenz Schauer  
01. Juli 2015

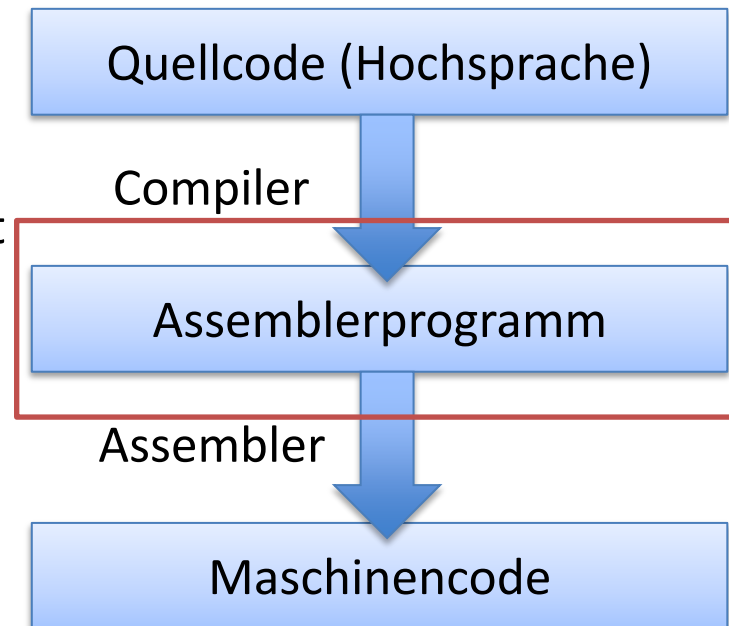


## Theoretische Grundlagen

- Motivation
- Register
- Aufbau eines Programms
  - Daten-Segment:
    - Direktiven: Word, asciiz,...
  - Text-Segment
- Befehle
  - Ladebefehle
  - Speicherbefehle
  - Arithmetische Befehle
- SPIM-Betriebssystem: Ein- und Ausgabe
- Kontroll-Strukturen

## Vereinfachter Ablauf:

- Hochsprache (C++, Java,...) wird mittels Compiler in eine Assemblersprache übersetzt
- Compiler analysiert das Programm und erzeugt Assemblercode
  - Für Menschen verständlicher Maschinencode
- Assembler übersetzt Assemblercode in Maschinencode
  - Maschinsprache bestehen aus 0 und 1



**Fokus nun auf Maschinenebene &  
Assemblerprogrammierung!**

## Assemblersprache:

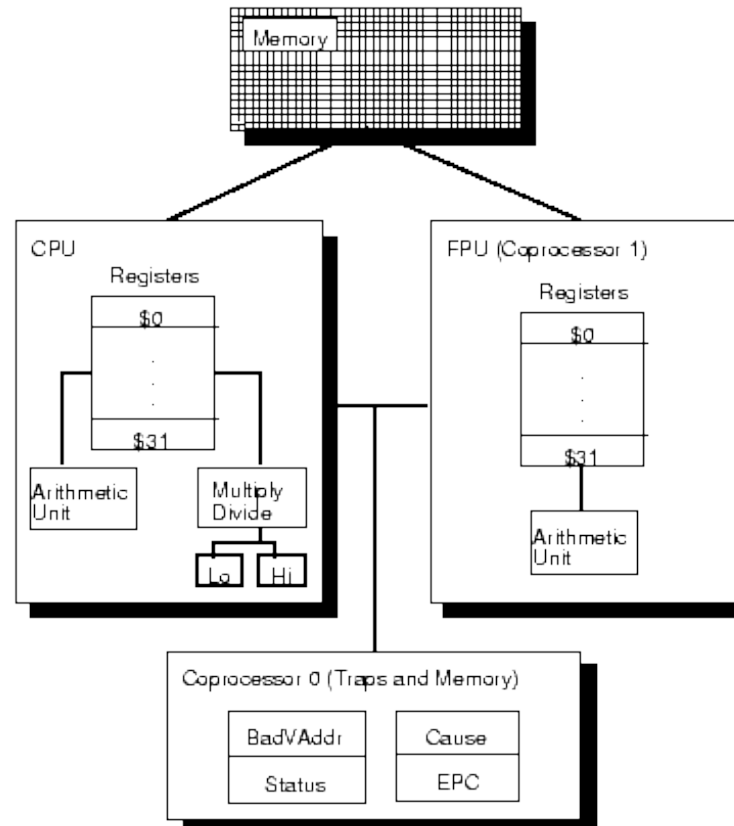
- Hardwarenahe Programmiersprache
- Wird von Assembler direkt in ausführbaren Maschinencode umgewandelt
- Alle Verarbeitungsmöglichkeiten des Mikrokontrollers werden genutzt
- Hardwarekomponenten können direkt angesteuert werden
- Erlauben Namen für Instruktionen, Speicherplätze, Sprungmarken, etc.
- I.d.R. effizient, geringer Speicherplatzbedarf
- Anwendung:
  - Gerätetreiber
  - Eingebettete Systeme
  - Echtzeitsysteme
  - Neue Hardware (Keine Bibliotheken vorhanden)
  - Programmierung von Mikroprozessoren (Bsp.: MIPS)

```
lw    $t0, ($a0)
add   $t0, $t1, $t2
sw    $t0, ($a0)
jr    $ra
```

Beispiel: Assemblercode

## MIPS R2000: RISC-Architektur

- Nicht alle Funktionen sind im Prozessor selbst realisiert, sondern in Coprozessoren ausgelagert.



- MIPS verfügt über 32 Register:
  - Jedes Register enthält ein 32-bit Wort (4 Byte)
  - Jedes Register kann für jeden Zweck verwendet werden (außer \$zero)
- Spezialregister in Koprozessor 0 (hier nicht weiter relevant)
- Register haben Nummern, aber auch Namen
- Konvention für Registernutzung:

Name	Nummer	Verwendung
\$zero	0	Enthält den Wert 0, kann nicht verändert werden.
\$at	1	temporäres Assemblerregister. (Nutzung durch Assembler)
\$v0	2	Funktionsergebnisse 1 und 2 auch für Zwischenergebnisse
\$v1	3	
\$a0	4	Argumente 1 bis 4 für den Prozeduraufruf
\$a1	5	
\$a2	6	
\$a3	7	
\$t0,...,\$t7	8-15	temporäre Variablen 1-8. Können von aufgerufenen Prozeduren verändert werden.

Name	Nummer	Verwendung
\$s0,..., \$s7	16 ... 23	langlebige Variablen 1-8. Dürfen von aufgerufenen Prozeduren nicht verändert werden.
\$t8,\$t9	24,25	temporäre Variablen 9 und 10. Können von aufgerufenen Prozeduren verändert werden.
\$k0,k1	26,27	Kernel-Register 1 und 2. Reserviert für Betriebssystem, wird bei Unterbrechungen verwendet.
\$gp	28	Zeiger auf Datensegment (global pointer)
\$sp	29	Stackpointer Zeigt auf das erste freie Element des Stacks.
\$fp	30	Framepointer, Zeiger auf den Prozedurrahmen
\$ra	31	Return Adresse



SPIM hat eine Load-Store Architektur

- Daten müssen erst aus dem Hauptspeicher in Register geladen werden (load), bevor sie verarbeitet werden können.
- Ergebnisse müssen aus Registern wieder in den Hauptspeicher geschrieben werden (store).

Es gibt keine Befehle, die Daten direkt aus dem Hauptspeicher verarbeiten.

Grundprinzip (Von-Neumann):

- Gemeinsamer Speicher für Daten und Programme

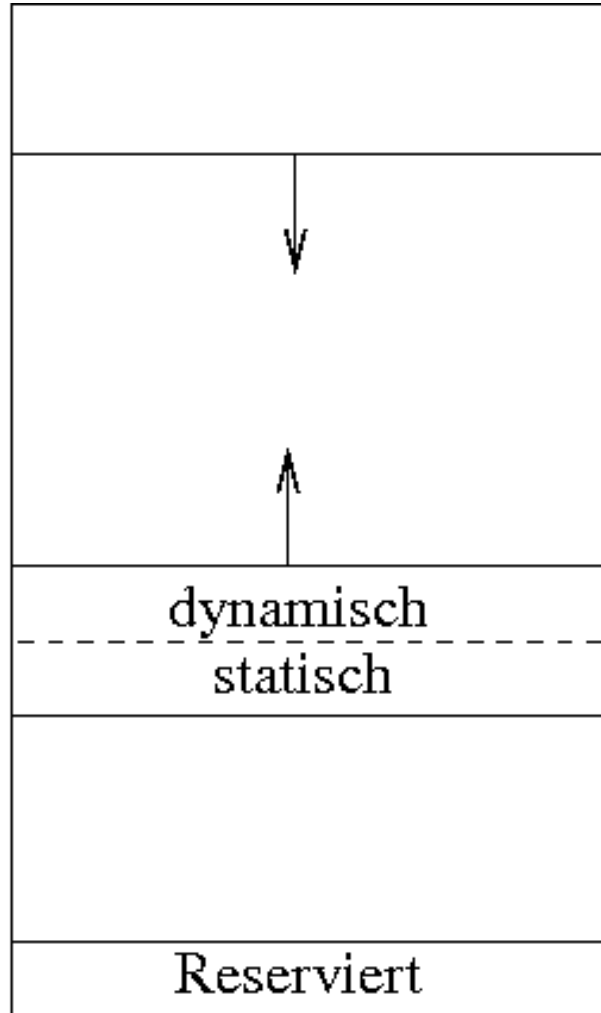
SPIM teilt den Hauptspeicher in **Segmente**, um Konflikte zu vermeiden:

- **Datensegment**
  - Speicherplatz für Programmdaten (Konstanten, Variablen, Zeichenketten, ...)
- **Textsegment**
  - Speicherplatz für das **Programm**.
- **Stacksegment**
  - Speicherplatz für den Stack.

Es gibt auch noch jeweils ein Text- und Datensegment für das Betriebssystem:

- Unterscheidung zwischen User- und Kernel- Text/Data Segment

7FFF FFFF



Stacksegment

1000 0000

Datensegment

0040 0000

Textsegment

0000 0000

Reserviert

Schreiben Sie ein Programm, das  $x+y$  berechnet. Die beiden Variablen sollen bereits im Datensegment initialisiert sein.

`.data` ← Datensegment

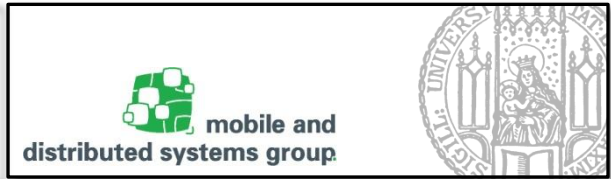
`.text` ← Textsegment

`main:` ← Marke, die vom SPIM  
Betriebssystem angesprungen wird.



LUDWIG-  
MAXIMILIANS-  
UNIVERSITÄT  
MÜNCHEN

# Programmaufbau



Schreiben Sie ein Programm, das den Umfang des Dreiecks mit den Kanten  $x$ ,  $y$ ,  $z$  berechnet!

Das Programm berechnet den Umfang des Dreiecks mit den Kanten x, y, z

```

    .data ←————— Datensegment
x:   .word 12
y:   .word 14
z:   .word 5
U:   .word 0

    .text ←————— Textsegment
main: lw $t0, x      # $t0 := x
      lw $t1, y      # $t1 := y
      lw $t2, z      # $t2 := z
      add $t0, $t0, $t1 # $t0 := x+y
      add $t0, $t0, $t2 # $t0 := x+y+z
      sw $t0, U      # U := x+y+z
      li $v0, 10     # EXIT
      syscall
  
```

←————— Kommentarzeichen

## Direktiven:

- `.data (.text):`
  - Kennzeichnet den Start des Datensegments (Textsegments)
- `.word:`
  - sorgt für Reservierung von Speicherplatz
  - hier für die Variablen `x,y,z,U`. Jeweils ein Wort (32 Bit) wird reserviert.
  - Inhalt wird mit den Zahlen 12, 14, 5 und 0 initialisiert.

## (Pseudo-) Befehle:

- `lw $t0, x` lädt den Inhalt von `x` in das Register `$t0`. (SPIM realisiert Load-Store Architektur)
- `add $t0, $t0, $t1` addiert den Inhalt von `$t0` zu `$t1` und speichert das Resultat wieder in `$t0`.
- `sw $t0, U` speichert den Inhalt von `$t0` in den Speicherplatz, der `U` zugewiesen ist.
- `li $v0, 10` und `syscall` halten das Programm an.

SPIM hat drei verschiedene Integertypen

Folgende Direktiven dienen zur Reservierung für den notwendigen Speicher

- `.word` (32 Bit Integer)
- `.half` (16 Bit Integer)
- `.byte` ( 8 Bit Integer)



Mit der Direktive

```
.word Wert1 Wert2 ...
```

werden Folgen von 32-Bit Integern angelegt (z.B. nützlich zur Speicherung von Feldern...)

Beispiel:

```
x: .word 256 0x100
```

- reserviert im Speicher  $2 \cdot 32$  Bit und schreibt in beide den Wert 256 hinein (0x... bedeutet hexadezimal).
- **x** ist eine *Marke*. Man kann damit auf den ersten Wert zugreifen.
- Mit **x+4** kann man auf den **zweiten** Wert zugreifen.

**x: .word 10 20 30**

**y: .half 3 4**

**z: .byte 5 6 7**

reserviert insgesamt 19 Bytes

- 12 Bytes mit den Zahlen 10, 20 und 30  
(zugreifbar über  $x$ ,  $x+4$  und  $x+8$ )
- 4 Bytes mit den Zahlen 3 und 4  
(zugreifbar über  $y$  und  $y+2$ )
- 3 Bytes mit den Zahlen 5,6 und 7  
(zugreifbar über  $z$ ,  $z+1$  und  $z+2$ )

Jeder Prozessor arbeitet nur vernünftig in einem Betriebssystem (Unix, Linux, Windows usw.)

Betriebssystem darf privilegierte Operationen durchführen:

- E/A-Operationen
- ...

Kontrollierten Zugang zu den Funktionen des Betriebssystems notwendig.

Betriebssystemfunktionen werden durchnummeriert, und über spezielle Funktion **syscall** aufgerufen.

**syscall** kann vor der Ausführung einer Betriebssystemfunktion überprüfen, ob das Programm die Rechte dazu hat.

Befehl **syscall** im SPIM erwartet die Nummer der auszuführenden Betriebssystemfunktion im Register **\$v0**.

Um eine Betriebssystemfunktion aufzurufen, lädt man deren Nummer in **\$v0** (z.B. **li \$v0,4**) und ruft dann **syscall** auf (4 ist die Druckfunktion für Strings).

**Laden:** `li $v0, <Code>`

**Ausführen:** `syscall`

Funktion	Code	Argumente	Ergebnis
print_int	1	Wert in \$a0 wird dezimal ausgegeben	
print_float	2	Wert in \$f12 wird als 32-Bit-Gleitkommazahl	
print_double	3	Wert in \$f12 und \$f13 wird als 64-Bit-Gleitkommazahl ausgegeben	
print_string	4	Die mit Chr \0 terminierte Zeichenkette, die an der Stelle (\$a0) beginnt, wird ausgegeben	
read_int	5		Die auf der Konsole dezimal eingegebene ganze Zahl in \$v0

Funktion	Code	Argumente	Ergebnis
read_float	6		Die auf der Konsole dezimal eingegebene 32-Bit-Gleitkommazahl in \$f0
read_double	7		Die auf der Konsole dezimal eingegebene 64-Bit- Gleitkommazahl in \$f0/1
read_string	8	Adresse, ab der die Zeichenkette abgelegt werden soll in \$a0, maximale Länge der Zeichenkette in \$a1	Speicher von (\$a0) bis (\$a0)+\$a1 wird mit der eingelesenen Zeichenkette belegt. Es wird "\n" mit eingelesen!
sbrk	9	Größe des Speicherbereichs in Bytes in \$a0	Anfangsadresse eines freien Blocks der geforderten Größe in \$v0
exit	10		

```
string1: .ascii "Hallo Welt"
```

```
string2: .asciiz "Hallo Welt"
```

Die Direktiven `.ascii` und `.asciiz` reservieren beide 10 Bytes für die ASCII-Darstellung von "Hallo Welt".

`.asciiz` hängt zusätzlich noch ein Null-Byte `\0` an (Ende der Zeichenkette) und verbraucht insgesamt 11 Bytes.

Die Zeichenketten sind über die Marken `string1` bzw. `string2` zugreifbar. (`string1` greift auf 'H' zu, `string1+1` auf 'a' usw.)

Schreiben Sie ein Programm, das „Servus Welt“ ausgibt.

Schreiben Sie ein Programm, das „Welt“ ausgibt.

Bonus:

- Schreiben Sie ein Programm, das einen String vom Benutzer einliest und ihn anschließend auf der Konsole ausgibt.



Innerhalb eines Strings sind folgende Kombinationen erlaubt:

`\n` (neue Zeile)

`\t` (Sprung zum nächsten Tabulator)

`\"` Das doppelte Anführungszeichen.

**Beispiel:**

```
a: .ascii "ab\n cd\t ef\"gh\""
```

könnte ausgedruckt so aussehen:

```
ab
cd   ef"gh"
```

(`\` ist das sog. "escape Zeichen")

4-Byte Integer könnte an den Adressen 0x3, 0x4,0x5,0x6 abgelegt werden (Adressierung geschieht byteweise).

Aber: Das ist *nicht ausgerichtet* (engl. aligned).

Ausgerichtete Speicherung wäre z.B. an den Adressen 0x0,0x1,0x2,0x3 oder 0x4,0x5,0x6,0x7.

Viele SPIM Befehle erwarten ausgerichtete Daten

- Anfangsadressen der Daten ist ein Vielfaches ihrer Länge.
- Die `.word`, `.half` und `.byte` Direktiven machen das automatisch richtig.

**Beispiel:**

**x: `.half 3`**

**y: `.word 55`**

würde nach dem x 2 Byte frei lassen, damit y ausgerichtet ist.

Befehl	Argumente	Wirkung
add	Rd, Rs1, Rs2	Rd := Rs1 + Rs2 (mit Überlauf)
sub	Rd, Rs1, Rs2	Rd := Rs1 - Rs2 (mit Überlauf)
addu	Rd, Rs1, Rs2	Rd := Rs1 + Rs2 (ohne Überlauf)
subu	Rd, Rs1, Rs2	Rd := Rs1 - Rs2 (ohne Überlauf)
addi	Rd, Rs1, Imm	Rd := Rs1 + Imm
addiu	Rd, Rs1, Imm	Rd := Rs1 + Imm (ohne Überlauf)
div	Rd, Rs1, Rs2	Rd := Rs1 DIV Rs2
rem	Rd, Rs1, Rs2	Rd := Rs1 MOD Rs2
mul	Rd, Rs1, Rs2	Rd := Rs1 × Rs2
b	label	unbedingter Sprung nach label
j	label	unbedingter Sprung nach label
jal	label	unbed.Sprung nach label, Adresse des nächsten Befehls in \$ra
jr	Rs	unbedingter Sprung an die Adresse in Rs
beq	Rs1, Rs2, label	Sprung, falls Rs1 = Rs2
beqz	Rs, label	Sprung, falls Rs = 0
bne	Rs1, Rs2, label	Sprung, falls Rs1 ≠ Rs2
bnez	Rs1, label	Sprung, falls Rs1 ≠ 0
bge	Rs1, Rs2, label	Sprung, falls Rs1 ≥ Rs2
bgeu	Rs1, Rs2, label	Sprung, falls Rs1 ≥ Rs2
bgez	Rs, label	Sprung, falls Rs ≥ 0
bgt	Rs1, Rs2, label	Sprung, falls Rs1 > Rs2
bgtu	Rs1, Rs2, label	Sprung, falls Rs1 > Rs2
bgtz	Rs, label	Sprung, falls Rs > 0
ble	Rs1, Rs2, label	Sprung, falls Rs1 ≤ Rs2
bleu	Rs1, Rs2, label	Sprung, falls Rs1 ≤ Rs2
blez	Rs, label	Sprung, falls Rs ≤ 0
blt	Rs1, Rs2, label	Sprung, falls Rs1 < Rs2
bltu	Rs1, Rs2, label	Sprung, falls Rs1 < Rs2
bltz	Rs, label	Sprung, falls Rs < 0
not	Rd, Rs1	Rd := ¬ Rs1 (bitweise Negation)
and	Rd, Rs1, Rs2	Rd := Rs1 & Rs2 (bitweises UND)
or	Rd, Rs1, Rs2	Rd := Rs1   Rs2 (bitweises ODER)
xori	Rd, Rs1, Imm	Rd := Rs1 ↔ Imm (bitweises XOR)
syscall		führt Systemfunktion aus
move	Rd, Rs	Rd := Rs
la	Rd, label	Adresse des Labels wird in Rd geladen
lb	Rd, Adr	Rd := MEM[Adr]
lw	Rd, Adr	Rd := MEM[Adr]
li	Rd, Imm	Rd := Imm
sw	Rs, Adr	MEM[Adr] := Rs (Speichere ein Wort)
sh	Rs, Adr	MEM[Adr] MOD 2 <sup>16</sup> := Rs (Speichere ein Halbwort)
sb	Rs, Adr	MEM[Adr] MOD 256 := Rs (Speichere ein Byte)

Funktion	Code in \$v0	Funktion	Code in \$v0
print_int	1	read_float	6
print_float	2	read_double	7
print_double	3	read_string	8
print_string	4	sbrk	9
read_int	5	exit	10

**<Marke>: <Befehl> <Arg 1> <Arg 2> <Arg 3> #<Kommentar>**

Oder mit Kommas

**<Marke>: <Befehl> <Arg 1>, <Arg 2>, <Arg 3> #<Kommentar>**

In der Regel 1 – 3 Argumente:

- Fast alle arithm. Befehle 3: 1 Ziel + 2 Quellen
- Befehle für Datenübertragung zw. Prozessor und HS: 2 Arg
- Treten *in folgender Reihenfolge auf*:
  - 1.) Register des Hauptprozessors, zuerst das Zielregister,
  - 2.) Register des Coprozessors,
  - 3.) Adressen, Werte oder Marken

Befehl	Argumente	Wirkung	Erläuterung
lw	Rd, Adr	RD=MEM[Adr]	Load word

**lw** lädt die Daten aus der angegebenen Adresse **Adr** in das Zielregister **Rd**.

**Adr** kann auf verschiedene Weise angegeben werden:

- **(Rs)** : Der Wert steht im Hauptspeicher an der Adresse, die im Register **Rs** steht (Register-indirekt)
- **label** oder **label+Konstante**: Der Wert steht im Hauptspeicher an der Stelle, die für **label** reserviert wurde, bzw. nachdem Konstante dazu addiert wurde (direkt).
- **label (Rs)** oder **label+Konstante (Rs)** : Der Wert steht im Hauptspeicher an der Stelle, die für **label** reserviert wurde + **Konstante** + **Inhalt** von Register **Rs** (indexiert).

Befehl	Argumente	Wirkung	Erläuterung
la	Rd, Label	RD=Adr(Label)	Load address

**la** lädt die **Adresse** auf die das Label **label** zeigt in das Zielregister Rd.

Zum Vergleich: **lw** lädt die **Daten** aus der angegebenen Adresse **Adr** in das Zielregister **Rd**.

**lb** und **lh** müssen aus 8 bzw. 16 Bit ein 32 Bit Integer machen.

Bei negativer Zahl muss mit 1en aufgefüllt werden!

**lbu** und **lhu** füllen **immer** mit 0en auf.

Befehl	Argumente	Wirkung	Erläuterung
lb	Rd,Adr	RD=MEM[Adr]	Load byte
lbu	Rd,Adr	RD=MEM[Adr]	Load unsigned byte
lh	Rd,Adr	RD=MEM[Adr]	Load halfword
lhu	Rd,Adr	RD=MEM[Adr]	Load unsigned halfword
ld	Rd,Adr	Lädt das Doppelword an der Stelle Adr in die Register Rd und Rd+1	Load double word

speichern Registerinhalte zurück in den Hauptspeicher.

Befehl	Argumente	Wirkung	Erläuterung
sw	Rs,Adr	MEM[Adr]:=Rs	store word
sb	Rs,Adr	MEM[Adr]:=Rs MOD 256	store byte (die letzten 8 Bit)
sh	Rs,Adr	MEM[Adr]:=Rs MOD $2^{16}$	store halfword(die letzten 16 Bit)
sd	Rs,Adr	sw Rs,Adr sw Rd+1,Adr+4	Store double word



**move:** Kopieren zwischen Registern.

**li:** Direktes laden des Wertes in ein Register

**lui:** Lädt den Wert in die oberen 16 Bits des Registers (und macht die unteren 16 Bits zu 0).

Befehl	Argumente	Wirkung	Erläuterung
move	Rd,Rs	$Rd := Rs$	move
li	Rd, Imm	$Rd := Imm$	load immediate
lui	Rd,Imm	$Rd := Imm * 2^{16}$	Load upper immediate

Ein Überlauf (Overflow) bewirkt den Aufruf eines Exception Handlers (ähnlich catch in Java).

Es gibt auch arithmetische Befehle, die Überläufe ignorieren.

## Weitere arithmetische Befehle:

- **div, mult** (in Versionen mit und ohne overflow, sowie mit und ohne Vorzeichen)
- **neg** (Zahl negieren), **abs** (Absolutbetrag), **rem** (Rest)

Befehl	Argumente	Wirkung	Erläuterung
add	Rd, Rs1, Rs2	$Rd := Rs1 + Rs2$	addition (mit overflow)
addi	Rd, Rs1, Imm	$Rd := Rs1 + Imm$	addition immediate (mit overflow)
sub	Rd, Rs1, Rs2	$Rd := Rs1 - Rs2$	subtract (mit overflow)

Erstellen Sie ein vollständiges SPIM-Programm, das folgendes durchführt:

- Es werden 2 positive Integer-Zahlen von der Konsole eingelesen
- Es wird der Durchschnitt dieser beiden Zahlen auf eine Nachkommastelle berechnet
- Das Ergebnis wird auf der Konsole ausgegeben
- Verwenden Sie für die Vorkommazahl das Register  $\$s0$  und für die Nachkommazahl das Register  $\$s1$  und ansonsten nur die temporären Register
- Sowohl die Eingabe als auch die Ausgabe soll mit einem Anweisungstext versehen werden
- Die Ausgabe muss logischerweise das Komma enthalten.

Geben Sie eine Sequenz von MIPS-Instruktionen für eine neue Anweisung „swap“ an, die die Werte der beiden Register  $\$s0$  und  $s1$  vertauscht.

Sie dürfen dazu genau ein weiteres Register (z.B  $\$t0$ ) verwenden.



LUDWIG-  
MAXIMILIANS-  
UNIVERSITÄT  
MÜNCHEN

Literatur



Deutschsprachiges SPIM-Tutorial von Reinhard Nitzsche (97) zu finden unter:

[http://www.mobile.ifi.uni-muenchen.de/studium\\_lehre/sose15/rechnerarchitektur/spim\\_tutorial\\_de.pdf](http://www.mobile.ifi.uni-muenchen.de/studium_lehre/sose15/rechnerarchitektur/spim_tutorial_de.pdf)