



LUDWIG-  
MAXIMILIANS-  
UNIVERSITÄT  
MÜNCHEN



mobile and  
distributed systems group



# Vorlesung Rechnerarchitektur

Sommersemester 2017

Carsten Hahn

1. Juni 2017



## **Kapitel 1:** Motivation, Einleitung & Hinführung zum Assembler

- Compiler, Interpreter, Assembler
- MIPS Architektur
  - CISC vs. RISC

## **Kapitel 2:** Einführung in die Assemblerprogrammierung mit SPIM

- Load-Store-Architektur
- Adressierung und Wörter
  - Byte-Reihenfolge
- Register
- Daten & Zeichenketten
- SPIM-Befehle
- Sprünge, IF, SWITCH, Schleifen

## Grundproblem:

**Menschen** besitzen komplexe Sprache (Subjekt – Prädikat – Objekt). Das Verstehen von Maschinencode ist umständlich

```
0011 1100 0001 1101 1100 1000
0110 0011 1010 1011 1001 1100
0010 0101 0011 1110 0110 0011
      0011 01.....
```

???

**Maschinen** besitzen eine sehr primitive Sprache (Befehlsfolge) und können nur **Maschinensprache** verstehen:

```
0011 1100 0001 1101 1100 1000
0110 0011 1010 1011 1001 1100
0010 0101 0011 1110 0110 0011
      0011 01.....
```

## Lösung:

- Menschen nutzen einen anderen Befehlssatz als Sprache L1 (Bsp.: C++, Java, Python, usw.)
- Maschinen verarbeiten eine Übersetzung (Compiler) bzw. Interpretation (Interpreter) von L1, hier als L0 bezeichnet.

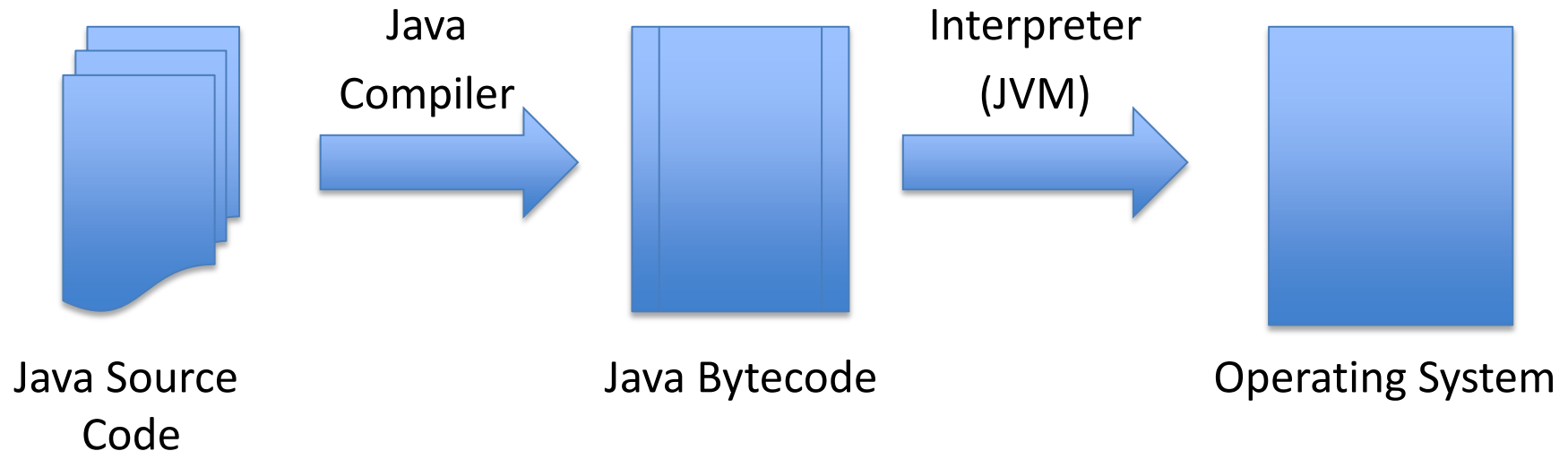
### ▪ **Compiler:**

- Vollständige Übersetzung des Programms von L1 zu L0
- Quellprogramm in L1 wird verworfen
- Zielprogramm in L0 wird in Speicher geladen und ausgeführt

### ▪ **Interpreter:**

- Jede L1 Anweisung wird analysiert, dekodiert und unmittelbar in L0 ausgeführt
- Quellcode + Interpreter auf ausführender Maschine benötigt
- I.d.R. langsamer als kompilierte Programme

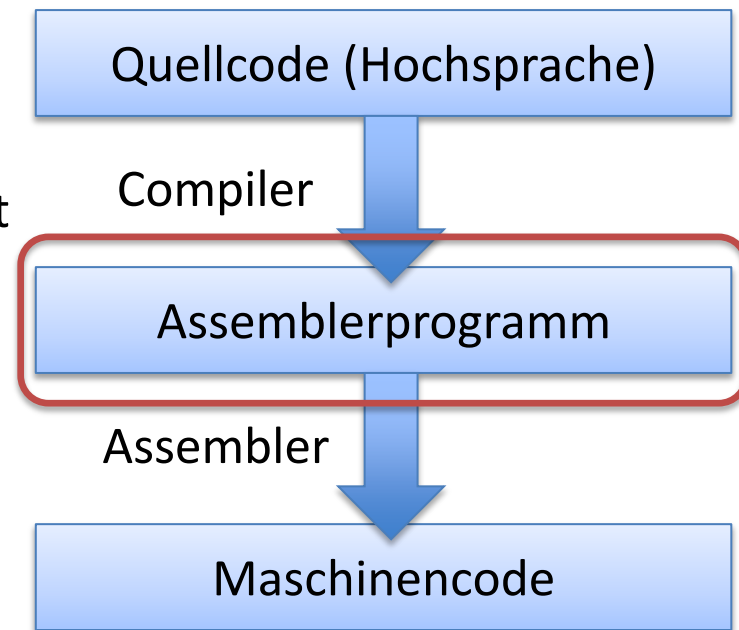
Auch hybride Ansätze möglich: Beispiel Java



- Quellcode wird von Java Compiler in Bytecode übersetzt
- JVM interpretiert den Bytestream als nativen Maschinencode, der vom OS ausgeführt werden kann

Vereinfachter, abstrakter Ablauf:

- Hochsprache (Bsp.: C++, C#)
- Compiler analysiert das Programm und erzeugt Assemblercode
  - Für Menschen verständlicher Maschinencode
- Assembler übersetzt Assemblercode in Maschinencode
  - Maschinsprache bestehend aus 0 und 1
  - Für Menschen schwer verständlich



**Fokus nun auf Maschinenebene &  
Assemblerprogrammierung!**

## Assemblersprache:

- Extrem hardwarenah
- Alle Verarbeitungsmöglichkeiten des Mikrokontrollers werden genutzt
- Hardwarekomponenten (Register) 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
```

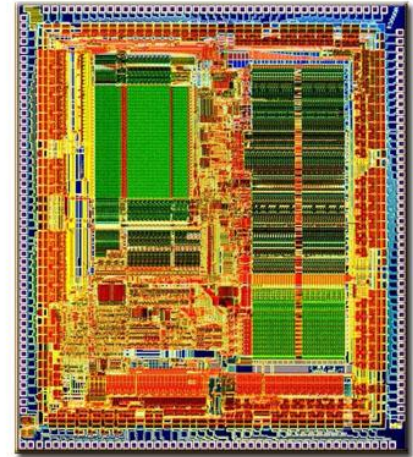
Beispiel: Assemblercode

## MIPS-Architektur (Microprocessor without Interlocked Pipeline Stages)

- Mikroprozessor ohne Pipeline-Sperren
  - Vereinfachtes Design: Eine Ausführung => Ein Zyklus
- Haupteinsatzbereich heute: Eingebettete Systeme
- RISC-Prozessorarchitektur

Unterscheidung zwischen:

- **RISC** = Reduced Instruction Set Computer
  - Mikroprozessoren waren früher alle RISC Prozessoren.
  - schnellere Ausführung von Befehlen (keine Interpretation nötig)
- **CISC** = Complex Instruction Set Computer
  - haben oft einen RISC Kern
  - Komplexe CISC-Instruktionen werden in Folge von RISC-Instruktionen übersetzt.



MIPS R3000



## CISC CPUs:

- Motorola 68000, Zilog Z80, Intel x86 Familie
- ab Pentium 486 allerdings mit RISC Kern und vorgeschaltetem Übersetzer in RISC Befehle.

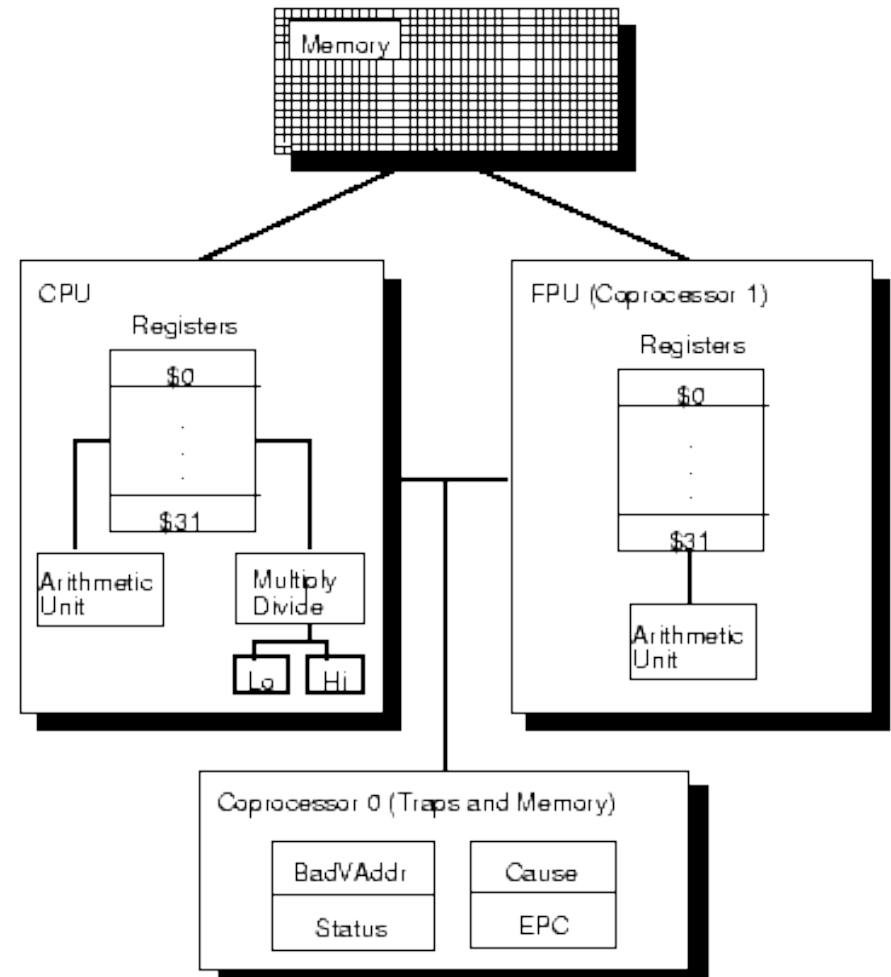
## RISC CPUs:

- Leistungsstarke eingebetteten Systemen (Druckern, Router)
- Workstations
- Supercomputern der Spitzenklasse
- ARM Prozessoren (**A**dvanced **R**ISC **M**achines)
- Milliarden ARM-CPU's im Einsatz in
  - hohe Leistung, geringen Stromverbrauch, niedrige Kosten
  - Apple: iPods (ARM7TDMI SoC), iPhone (Samsung ARM1176JZF), iPod touch (ARM11)
  - Google: Bsp.: LG Nexus (bspw.: ARM Cortex-A57)
  - Canon: IXY Digital 700 Kamera (ARM-basiert)
  - Hewlett-Packard: HP-49/50 Taschenrechner (ARM9TDMI)

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

Der SPIM Simulator simuliert 2:

- Koprozessor 0:
  - Ausnahme- und Unterbrechungs-Routinen
    - Status, Cause, ...
- Koprozessor 1:
  - FPU: **F**loating **P**oint **U**nit
  - Für Berechnungen mit Gleitkommazahlen (floating points)





Die Assemblersprache für den **MIPS**-Prozessor heißt: **SPIM**

Ein deutschsprachiges Tutorial von Reinhard Nitzsche (1997) ist auf der Vorlesungswebseite verlinkt:

- <http://www.mobile.ifi.lmu.de/lehrveranstaltungen/rechnerarchitektur-rose17/>
- Wird sehr empfohlen, da übersichtlich, kompakt und mit eigenen Übungen!
- Sehr gut für den Einstieg geeignet
- Umfasst den behandelten Stoff für SPIM

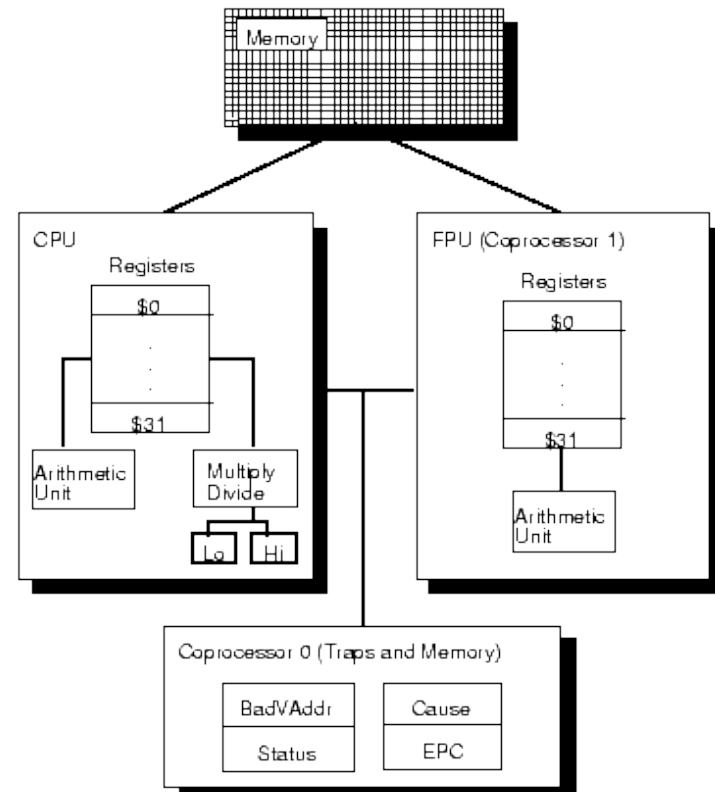
Zur Assemblersprache gibt es auch einen Assembler und einen Simulator für den MIPS-Prozessor. Der heißt ebenfalls **SPIM**.

- Auch dieser ist auf der Vorlesungsseite verlinkt:
- Bsp.: QtSpim: <http://spimsimulator.sourceforge.net/>
- Bsp.: MARS: <http://courses.missouristate.edu/KenVollmar/MARS/>

## SPIM hat eine **Load-Store-Architektur**

- Daten müssen zuerst aus dem Hauptspeicher in die 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!



Um Daten aus dem Hauptspeicher überhaupt laden zu können, müssen wir bestimmen, welche Daten wir laden wollen.

- Das geschieht über Adressen!
- Speicherzellen in denen Daten liegen können müssen adressierbar sein.
  - Adressierung der Daten im Hauptspeicher bei SPIM: **byteweise**!

Dem gegenüber definiert man ein Wort (**word**):

- Bezeichnet die natürliche Einheit (Grundverarbeitungsgröße) einer zugrundeliegenden Architektur
  - Hat eine feste Größe in bits/bytes (Wortlänge)
  - Ist die max. Datengröße, die in einem Rechenschritt verarbeitet werden kann.
    - ⇒ Abhängig von Bus- und Registerbreite.
    - ⇒ Ist also charakteristisch für eine Computer Architektur

**Achtung:** Nicht verwechseln mit Adressen!

Bei SPIM:

- **Byteweise** Adressierung:
  - D.h.: Jedes Byte im Speicher hat seine eigene Adresse!
    - I.d.R. fortlaufend nummeriert, wie bei einer Straße
- Wörter:
  - MIPS besitzt Register von je 32 Bit Breite = 4 Byte (entspricht auch der Wortlänge)
  - D.h.: Jedes **Wort ist 4 Byte** groß und hat somit (theoretisch) 4 Adressen  
 => Will man vom aktuellen Wort mit Adresse x zum nächsten Wort, so muss man  $x+4$  rechnen



|            |     |             |             |             |             |             |             |             |             |     |
|------------|-----|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-----|
| Adresse    | ... | 0xA8        | 0xA9        | 0xAA        | 0xAB        | 0xAC        | 0xAD        | 0xAE        | 0xAF        |     |
| Bytegrenze | ... | byte<br>168 | byte<br>169 | byte<br>170 | byte<br>171 | byte<br>172 | byte<br>173 | byte<br>174 | byte<br>175 |     |
| Wortgrenze | ... | word 42     |             |             |             | word 43     |             |             |             | ... |

Ein Wort umfasst also mehrere Bytes (hier: 4)

- Bytes können in
  - Aufsteigender oder
  - absteigender

Reihenfolge aneinander gehängt werden.

**Big-Endian** (wörtlich „Großes Ende“):

- Byte mit den höchstwertigen Bits (signifikantesten Stellen) an der kleinsten Speicheradresse.

**Little-Endian** (wörtlich „Kleines Ende“):

- Byte mit den niederwertigsten Bits (wenigsten signifikanten Stellen) an der kleinsten Speicheradresse

**Achtung:**

- Der SPIM-Simulator benutzt die Byte-order des Rechners, auf dem er läuft.



Speicherung der Dezimalzahl 1.296.650.323 als 32-Bit-Wert:

- **Binär:**                    01001101    01001001    01010000    01010011
- **Hexadezimal:**        0x4D            0x49            0x50            0x53

Interpretation der Zahl als Zeichenkette in einem 32-Bit Wort:

- **ASCII (1 Byte/Zeichen):** M I P S

| Adresse       |       | ... | 0xA8     | 0xA9     | 0xAA     | 0xAB     | ... |
|---------------|-------|-----|----------|----------|----------|----------|-----|
| Big Endian    | Binär | ... | 01001101 | 01001001 | 01010000 | 01010011 | ... |
|               | Hex   | ... | 0x4D     | 0x49     | 0x50     | 0x53     | ... |
|               | ASCII | ... | M        | I        | P        | S        | ... |
| Little Endian | Hex   | ... | 0x53     | 0x50     | 0x49     | 0x4D     | ... |
|               | Binär | ... | 01010011 | 01010000 | 01001001 | 01001101 | ... |
|               | ASCII | ... | S        | P        | I        | M        | ... |

Beispiel:

- Konversion einer Zwei-Byte- in eine Vier-Byte-Zahl
  - Bsp.: 0x23 0xA1 -> **0x00 0x00** 0x23 0xA1

Little-Endian-Maschine:

- Anfügen von zwei Null Bytes am Ende
- Speicheradresse bleibt gleich

| Adresse     | 1    | 2    | 3           | 4           |
|-------------|------|------|-------------|-------------|
| Wert zuvor  | 0xA1 | 0x23 |             |             |
| Wert danach | 0xA1 | 0x23 | <b>0x00</b> | <b>0x00</b> |

Big-Endian-Maschine:

- Wert muss zunächst im Speicher um zwei Byte verschoben werden.

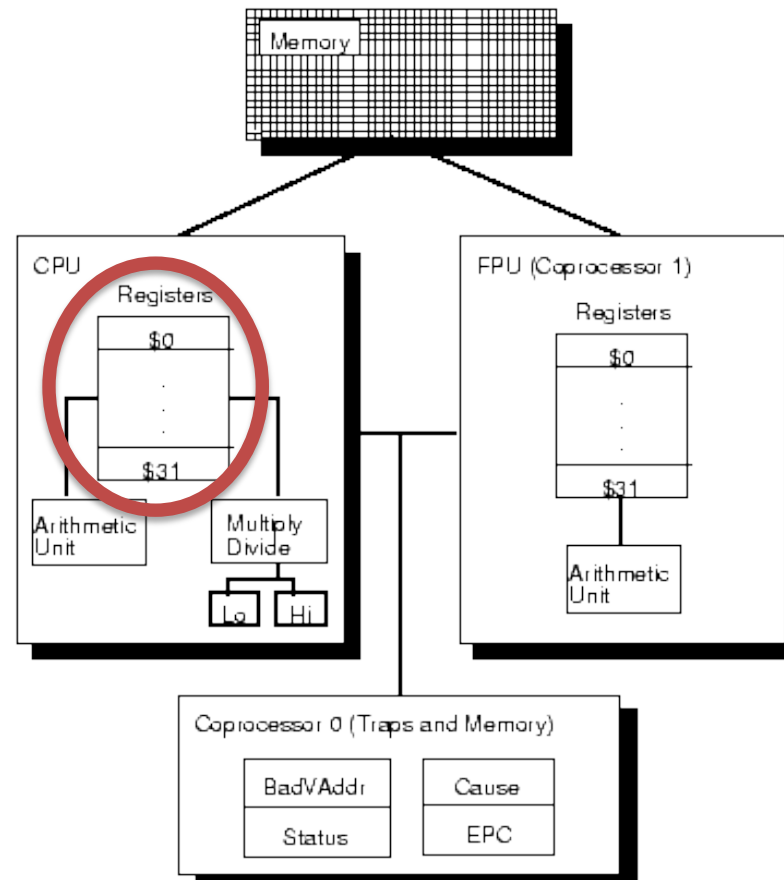
| Adresse     | 1           | 2           | 3           | 4           |
|-------------|-------------|-------------|-------------|-------------|
| Wert zuvor  | 0x23        | 0xA1        |             |             |
| Wert danach | <b>0x00</b> | <b>0x00</b> | <b>0x23</b> | <b>0xA1</b> |

Umgekehrte Umwandlung:

- Einfacher auf Little-Endian-Maschine:
  - höherwertige Bytes werden verworfen
  - Speicheradresse bleibt gleich
- Bei Big-Endian: Rückgängige Verschiebung

## SPIM hat eine **Load-Store-Architektur**

- Daten müssen zuerst aus dem Hauptspeicher in die **Register** geladen werden (load), bevor sie verarbeitet werden können.



MIPS verfügt über 32 Register:

- General Purpose Register
  - Können im Prinzip für jeden Zweck genutzt werden (Ausnahme: \$zero)
  - **ABER:** Konvention sollte unbedingt beachtet werden!

| 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,...,<br>\$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        | Global Pointer: Zeiger auf Datensegment                                                     |
| \$sp              | 29        | Stackpointer Zeigt auf das erste freie Element des Stacks.                                  |
| \$fp              | 30        | Framepointer, Zeiger auf den Prozedurrahmen                                                 |
| \$ra              | 31        | Return Adresse                                                                              |

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:

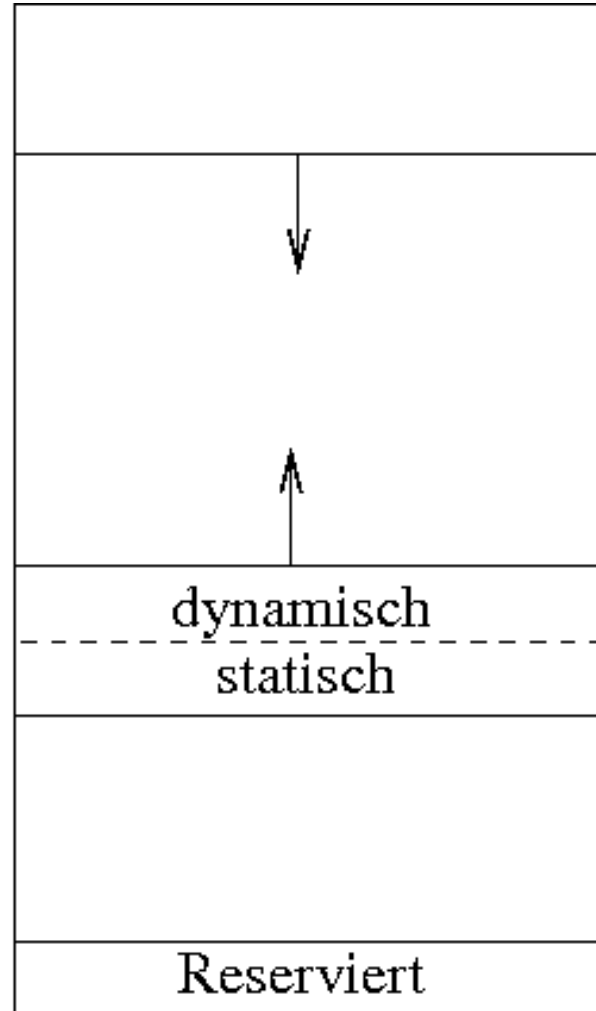
- Unterscheidung zwischen User- und Kernel- Text/Data Segment

7FFF FFFF

1000 0000

0040 0000

0000 0000



Stacksegment

Datensegment

Textsegment

Reserviert

Wir wollen uns ein erstes Assemblerprogramm in SPIM anschauen:

- Das Programm berechnet den Umfang des Dreiecks mit den Kanten x, y, z
- Legt das Ergebnis in die Variable U

```
.data  
x:    .word 12  
y:    .word 14  
z:    .word 5  
U:    .word 0
```

Direktive **.data** kennzeichnet  
den Beginn des Datensegments

Definition der Variablen im  
Datensegment

```
.text  
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
```

Direktive **.text** kennzeichnet  
den Beginn des Textsegments

Ladebefehle

Arithmetische Befehle

Speicherbefehle

Programm beenden

Marke **main:** als Einstiegspunkt  
erforderlich!

Kommentare



## 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.

File Edit Run Settings Tools Help



**Edit Execute**

**Text Segment**

| Bkpt | Address    | Code       | Basic                    | Source                                           |
|------|------------|------------|--------------------------|--------------------------------------------------|
|      | 0x00400000 | 0x24080000 | addiu \$8,\$0,0x00000000 | 8: main: li \$t0, 0 # i := 0                     |
|      | 0x00400004 | 0x24090004 | addiu \$9,\$0,0x00000004 | 9: byte: li \$t1, 4 # k := 4                     |
|      | 0x00400008 | 0x0109082a | slt \$1,\$8,\$9          | 10: loop1: bge \$t0, \$t1, half # WHILE i < k DO |
|      | 0x0040000c | 0x10200008 | beq \$1,\$0,0x00000008   |                                                  |
|      | 0x00400010 | 0x24020001 | addiu \$2,\$0,0x00000001 | 11: li \$v0, 1 #                                 |
|      | 0x00400014 | 0x3c011001 | lui \$1,0x00001001       | 12: lb \$a0, a(\$t0) # num := a[i]               |
|      | 0x00400018 | 0x00280821 | addu \$1,\$1,\$8         |                                                  |
|      | 0x0040001c | 0x80240000 | lb \$4,0x00000000(\$1)   |                                                  |
|      | 0x00400020 | 0x0000000c | syscall                  | 13: syscall #                                    |
|      | 0x00400024 | 0x21080001 | addi \$8,\$8,0x00000001  | 14: addi \$t0, \$t0, 1 # i := i+1                |
|      | 0x00400028 | 0x0c10001e | jal 0x00400078           | 15: jal nl # println()                           |
|      | 0x0040002c | 0x08100002 | j 0x00400008             | 16: j loop1 # DONE                               |
|      | 0x00400030 | 0x24080000 | addiu \$8,\$0,0x00000000 | 21: half: li \$t0, 0 # i := 0                    |
|      | 0x00400034 | 0x0109082a | slt \$1,\$8,\$9          | 22: loop2: bge \$t0, \$t1, word # WHILE i < k DO |
|      | 0x00400038 | 0x10200008 | beq \$1,\$0,0x00000008   |                                                  |
|      | 0x0040003c | 0x24020001 | addiu \$2,\$0,0x00000001 | 23: li \$v0, 1 #                                 |
|      | 0x00400040 | 0x3c011001 | lui \$1,0x00001001       | 24: lhu \$a0, a(\$t0) # num := a[i]              |
|      | 0x00400044 | 0x00280821 | addu \$1,\$1,\$8         |                                                  |
|      | 0x00400048 | 0x94240000 | lhu \$4,0x00000000(\$1)  |                                                  |

**Data Segment**

| Address    | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) |
|------------|------------|------------|------------|------------|-------------|
| 0x10010000 | 0x000000ff | 0x0000000a | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x10010020 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x10010040 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x10010060 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x10010080 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x100100a0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x100100c0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x100100e0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x10010100 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x10010120 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x10010140 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x10010160 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |
| 0x10010180 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  |

**Mars Messages** **Run I/O**

Clear 255

| Coprocc 0 |        |            | Coprocc 1 |  |  |
|-----------|--------|------------|-----------|--|--|
| Registers |        |            |           |  |  |
| Name      | Num... | Value      |           |  |  |
| \$zero    | 0      | 0x00000000 |           |  |  |
| \$at      | 1      | 0x10010000 |           |  |  |
| \$v0      | 2      | 0x00000001 |           |  |  |
| \$v1      | 3      | 0x00000000 |           |  |  |
| \$a0      | 4      | 0xffffffff |           |  |  |
| \$a1      | 5      | 0x00000000 |           |  |  |
| \$a2      | 6      | 0x00000000 |           |  |  |
| \$a3      | 7      | 0x00000000 |           |  |  |
| \$t0      | 8      | 0x00000000 |           |  |  |
| \$t1      | 9      | 0x00000004 |           |  |  |
| \$t2      | 10     | 0x00000000 |           |  |  |
| \$t3      | 11     | 0x00000000 |           |  |  |
| \$t4      | 12     | 0x00000000 |           |  |  |
| \$t5      | 13     | 0x00000000 |           |  |  |
| \$t6      | 14     | 0x00000000 |           |  |  |
| \$t7      | 15     | 0x00000000 |           |  |  |
| \$s0      | 16     | 0x00000000 |           |  |  |
| \$s1      | 17     | 0x00000000 |           |  |  |
| \$s2      | 18     | 0x00000000 |           |  |  |
| \$s3      | 19     | 0x00000000 |           |  |  |
| \$s4      | 20     | 0x00000000 |           |  |  |
| \$s5      | 21     | 0x00000000 |           |  |  |
| \$s6      | 22     | 0x00000000 |           |  |  |
| \$s7      | 23     | 0x00000000 |           |  |  |
| \$t8      | 24     | 0x00000000 |           |  |  |
| \$t9      | 25     | 0x00000000 |           |  |  |
| \$k0      | 26     | 0x00000000 |           |  |  |
| \$k1      | 27     | 0x00000000 |           |  |  |
| \$gp      | 28     | 0x10008000 |           |  |  |
| \$sp      | 29     | 0x7fffffc0 |           |  |  |
| \$fp      | 30     | 0x00000000 |           |  |  |
| \$ra      | 31     | 0x00000000 |           |  |  |
| pc        |        | 0x00400020 |           |  |  |
| hi        |        | 0x00000000 |           |  |  |
| lo        |        | 0x00000000 |           |  |  |

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 diesen Direktiven lassen sich Variablen (Werte) anlegen
- Mit einer Marke kann man auf den entsprechenden Wert mit Hilfe eines Namens leichter zugreifen
  - **Achtung:** Dürfen nicht so heißen wie ein Befehl

Beispiele:

```
x:  .word  0x22    # 32-bit int x = 34 (in Hexadezimal: 0x22)
```

```
y:  .half    22    # 16-bit int y = 22
```

```
z:  .byte    4     #  8-bit int z = 4
```

Marke

Direktive

Wert

Mit **marke: .word Wert1 Wert2 ...** können Folgen von 32-Bit Integern angelegt werden z.B. nützlich zur Speicherung von Feldern (Arrays)

- Beispiel:

```
x: .word 10 20 30
```

```
y: .half 3 4
```

```
z: .byte 5 6 7
```

- Mit der Marke kann auf den ersten Wert zugreifen.
- Mit **marke+SpeicherGröße** kann man auf den **zweiten** Wert zugreifen usw.

Im Beispiel werden insgesamt 19 Bytes reserviert:

- 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

```
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)
  - verbraucht insgesamt 11 Bytes.

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

Darstellung im Speicher von `string2`:

| Adresse       | 0xA8 | 0xA9 | 0xAA | 0xAB | 0xAC | 0xAD | 0xAE | 0xAF | 0xB0 | 0xB1 | 0xB2 | 0xB3 |
|---------------|------|------|------|------|------|------|------|------|------|------|------|------|
| Big Endian    | H    | a    | l    | l    | o    |      | W    | e    | l    | t    | \0   |      |
| Little Endian | l    | l    | a    | H    | e    | W    |      | o    |      | \0   | t    | l    |

Innerhalb eines Strings sind folgende Kombinationen erlaubt:

- `\n` neue Zeile
- `\t` Sprung zum nächsten Tabulator
- `\"` Das doppelte Anführungszeichen.

`\` ist das sog. "escape Zeichen"

Beispiel:

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

könnte ausgedruckt so aussehen:

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

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.

Allgemeiner Aufbau einer Assembler-Befehlszeile:

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

Oder mit Kommata:

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

In der Regel 1 – 3 Argumente:

- Fast alle arithmetischen Befehle 3: 1 Ziel + 2 Quellen
- Transferbefehle 2: Ziel + Quelle
- Treten *in folgender Reihenfolge auf*:
  - 1.) Register des Hauptprozessors, zuerst das Zielregister,
  - 2.) Register des Koprozessors,
  - 3.) Adressen, Werte oder Marken



| Befehl | Argumente    | Wirkung                    | Erläuterung    |
|--------|--------------|----------------------------|----------------|
| div    | Rd, Rs1, Rs2 | $Rd = \text{INT}(Rs1/Rs2)$ | divide         |
| li     | Rd, Imm      | $Rd = \text{Imm}$          | Load Immediate |

## Erläuterungen:

- Rd = destination register (Zielregister)
- Rs1, Rs2 = source register (Quellregister)
- Imm = irgendeine Zahl

## Beispiel:

div \$t0,\$t1,\$t2

Dividiere den Inhalt von \$t1 durch den Inhalt von \$t2 und speichere das Ergebnis ins Zielregister \$t0.

| 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:

- **Register-indirekt:**

- **(Rs):** Der Wert steht im Hauptspeicher an der Adresse, die im Register **Rs** steht
  - Bsp.: **lw \$t0, (\$t1)**
    - Lädt den **Wert** ins Register \$t0, der an der **Adresse** steht, die im Register \$t1 gespeichert ist.

- **Direkt:**

- **label** oder **label+Konstante**: Der Wert steht im Hauptspeicher an der Stelle, die für **label** reserviert wurde, bzw. nachdem **Konstante** dazu addiert wurde
  - Bsp.: **lw \$t0, x+4**
    - Lädt den **Wert** ins Register \$t0, der an der Stelle des **Labels** x + 4 Byte steht (nächstes Wort nach x)

- **Indexiert:**

- **label(Rs)** oder **label+Konstante(Rs)**: Der Wert steht im Hauptspeicher an der Stelle, die für **label** reserviert wurde + **Konstante** + **Wert** von Register **Rs**
  - Bsp.: **lw \$t0, x+12(\$t1)**
    - Lädt den **Wert** ins Register \$t0, der an der **Stelle** des **Labels** x + 12 Byte + **Wert** in \$t1 steht.

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

Dem gegenüber steht:

- **la** lädt die **Adresse** auf die das Label **label** zeigt in das Zielregister **Rd**.
  - Bsp.: **la \$t0, y**
    - Lädt die **Adresse nicht den Wert** vom Label y!

Zum Vergleich: **lw** lädt die **Daten (Wert)** an der angegebenen Adresse **Adr** in das Zielregister **Rd**.

```
.data
```

```
var: .word 20, 4, 22, 25, 7
```

```
.text
```

```
main: lw $t1, var
```

\$t1=20 -> Direkte Adressierung

```
lw $t1, var+4
```

\$t1=4 -> Direkte Adressierung

```
lw $t2, var($t1)
```

\$t2=4 -> Indexierte Adressierung

```
lw $t2, var+8($t1)
```

\$t2=25 -> Indexierte Adressierung

```
la $t1, var
```

\$t1=Adr(var) -> load address

```
lw $t2, ($t1)
```

\$t2=20 -> Indirekte Adressierung

| 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       |

- **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.

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<br>sw Rd+1,Adr+4 | Store double word                  |

Für die Adressierung stehen wieder die bekannten Modi wie bei den Ladebefehlen zur Verfügung

- Bsp.: **sw \$t0, var**
  - Speichert den **Wert** im Register \$t0 an die Stelle des **Labels** **var** im Hauptspeicher
  - Man beachte: Als Argumente kommen nun zuerst die Quelle und dann das Ziel
    - Bei Lade-Befehlen genau andersrum

Manipulieren Daten in Registern **ohne** Zugriff auf den Hauptspeicher:

| 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 |

- **move**: Kopieren zwischen Registern.
  - Bsp.: `move $t0, $t1`
    - Kopiere Inhalt von Register \$t1 ins Register \$t0
- **li**: Direktes laden eines Wertes in ein Register
  - Bsp.: `li $t0, 2`
    - Lädt den konstanten Wert 2 ins Register \$t0
- **lui**: Lädt den Wert in die oberen 16 Bits des Registers (und macht die unteren 16 Bits zu 0).

| 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)           |

Operationen können nur auf den Registern erfolgen!

- **Kein** arithmetischer Befehl hat als Parameter eine Hauptspeicheradresse!
- Arithmetische Befehle
  - greifen direkt auf Register zu (Register-direkte Adressierung)
  - oder verwenden konstante Werte (unmittelbare Adressierung)

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)



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

- Betriebssystem darf privilegierte Operationen durchführen:
  - Bsp.: E/A-Operationen
- Kontrollierten Zugang zu den Funktionen des Betriebssystems notwendig!

SPIM implementiert ein ganz einfaches Betriebssystem, das simple Methoden zur Ein- bzw. Ausgabe auf der Konsole mitliefert.

- SPIM Betriebssystemfunktionen werden durchnummeriert, und über die spezielle Funktion **syscall** aufgerufen
  - **syscall** hat selbst keine Parameter
    - Bedient sich aber einem Registerwert in \$v0
  - **syscall** kann vor der Ausführung einer Betriebssystemfunktion überprüfen, ob das Programm die Rechte dazu hat.

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

- Um eine bestimmte Betriebssystemfunktion aufzurufen,
  - schaut man in einer Tabelle nach, welche Nummer welche Funktion erfüllt
  - und lädt dann diese Nummer in das Register **\$v0**
    - Bsp.: Programm Ende: **li \$v0, 10**
  - anschließend ruft man **syscall** auf
- **syscall** schaut also nach, welcher Wert im Register **\$v0** liegt und führt dann die entsprechende Betriebssystemfunktion aus.
- Braucht eine Betriebssystemfunktion wiederum einen Wert (Bsp.: ein bestimmter String soll auf der Konsole ausgegeben werden)
  - dann muss dieser Wert im Argumentenregister **\$a0** stehen!

Laden: `li $v0,<Code>`

Ausführen: `syscall`

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

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

```
.data
txt1: .asciiz "Zahl= "
txt2: .asciiz "Text= "
input: .ascii "Dieser Text wird nachher ueber"
      .asciiz "geschrieben!"

.text
      # Eingabe...
main: li $v0, 4      # 4: print_str
      la $a0, txt1   # Adresse des ersten Textes in $a0
      syscall

      li $v0, 5      # 5: read_int
      syscall

      move $s0,$v0   # gelesenen Wert aus $v0 in $s0 kopieren
      li $v0, 4      # 4: print_str
      la $a0, txt2   # Adresse des zweiten Textes in $a0
      syscall

      li $v0, 8      # 8: read_str
      la $a0, input  # Adresse des einzugebenden Textes
      li $a1, 256    # maximale Länge
      syscall        # Eingelesener Text in input
```

*# Ausgabe...*

```
li $v0, 1      # 1: print_int
move $a0, $s0
syscall
```

```
li $v0, 4      # 4: print_str
la $a0, input
syscall
```

```
li $v0, 10     # Exit
syscall
```

Es werden immer alle Zeichen mit ausgegeben, also auch \n

```
.data
txt1: .asciiz "Dieser\nText\n\nwird\n\n\nausgegeben\n"
txt2: .asciiz "Und dieser auch"

.text
main: li    $v0, 4      # 4: print_str
      la    $a0, txt1   # Adresse von txt1 in $a0
      syscall

      la    $a0, txt2   # Adresse von txt2 in $a0
      syscall

      li    $v0, 10     # Exit
      syscall
```

Ausgabe:

```
1 prompt$ spim -file print_str.s
2 Dieser
3 Text
4
5 wird
6
7
8 ausgegeben
9 Und dieser auchprompt$
```

Es wird das Zeichen \n von der Konsole mit eingelesen!

```
.data
txt:    .asciiz    "Text="
input:  .asciiz    "xxxxx" # Das hier wird überschrieben

.text
main:   li $v0, 4      # 4: print_str
        la $a0, txt    # Adresse von txt in $a0
        syscall

        li $v0, 8      # 5: read_str
        la $a0, input  # Adresse des einzugebenden Textes
        li $a1, 4      # maximale Laenge
        syscall

        li $v0, 4      # 4: print_str
        la $a0, input
        syscall

        li $v0, 10     # Exit
        syscall
```

Ausgabe:

```
1 prompt$ spim -file read_str.s
2 Text=a
3 a
4 prompt$
```



## Speicherlayout vor dem Drücken der <Enter>-Taste

| Adresse       | 0xA8 | 0xA9 | 0xAA | 0xAB | 0xAC | 0xAD | 0xAE | 0xAF | 0xB0 | 0xB1 | 0xB2 | 0xB3 |
|---------------|------|------|------|------|------|------|------|------|------|------|------|------|
| Big Endian    | T    | e    | x    | t    | =    | \0   | x    | x    | x    | x    | x    | \0   |
| Little Endian | t    | x    | e    | T    | x    | x    | \0   | =    | \0   | x    | x    | x    |

## Speicherlayout nach dem Drücken der <Enter>-Taste

| Adresse       | 0xA8 | 0xA9 | 0xAA | 0xAB | 0xAC | 0xAD | 0xAE | 0xAF | 0xB0 | 0xB1 | 0xB2 | 0xB3 |
|---------------|------|------|------|------|------|------|------|------|------|------|------|------|
| Big Endian    | T    | e    | x    | t    | =    | \0   | a    | \n   | \0   | x    | x    | \0   |
| Little Endian | t    | x    | e    | T    | \n   | a    | \0   | =    | \0   | x    | x    | \0   |

| Befehl | Argumente    | Wirkung           | Erläuterung           |
|--------|--------------|-------------------|-----------------------|
| and    | Rd, Rs1, Rs2 | Rd := Rs1 and Rs2 | bitwise and           |
| andi   | Rd, Rs1, Imm | Rd := Rs1 and Imm | bitwise and immediate |
| or     | Rd, Rs1, Rs2 | Rd := Rs1 or Rs2  | bitwise or            |
| ori    | Rd, Rs1, Imm | Rd := Rs1 or Imm  | bitwise or immediate  |
| nor    | Rd, Rs1, Rs2 | Rd := Rs1 nor Rs2 | bitwise not or        |

+ Viele weitere logische Befehle:

- **xor**, **xori**, **not**
- **rol** (rotate left), **ror** (rotate right)
- **sll** (shift left logical), **sr1** (shift right logical), **sra** (shift right arithmetical)
- **seq** (set equal), **sne** (set not equal)
- **sge** (set greater than or equal), **sgt** (set greater than), ...

| Befehl | Argumente | Wirkung                       | Erläuterung          |
|--------|-----------|-------------------------------|----------------------|
| b      | label     | Unbedingter Sprung nach label | branch               |
| j      | label     | Unbedingter Sprung nach label | jump                 |
| beqz   | Rs,label  | Sprung nach label falls Rs=0  | Branch on equal zero |

+ weitere 20 bedingte branch Befehle.

- branch und jump Befehle unterscheiden sich auf Maschinenebene
- Der Unterschied wird von der Assemblersprache verwischt.

```
IF Betrag > 1000
    THEN Rabatt := 3
    ELSE Rabatt := 2
END;
```

### Assemblerprogramm:

- Annahme: Betrag ist in Register \$t0, Rabatt soll ins Register \$t1

```
    ble $t0, 1000, else # IF Betrag > 1000
    li  $t1, 3          # THEN Rabatt := 3

    b endif

else:
    li $t1, 2           # ELSE Rabatt := 2

endif:                  # FI
```

```

summe := 0;
i := 0;
WHILE summe <= 100
    i := i + 1;
    summe := summe + i
END;

```

## Assemblerprogramm:

```

        li    $t0, 0                # summe := 0;
        li    $t1, 0                # i := 0;
while:
        bgt    $t0, 100, elihw       # WHILE summe <= 100 DO
        addi   $t1, $t1, 1           # i := i + 1;
        add    $t0, $t0, $t1         # summe := summe + i
        b      while                # Schleife Wiederholen;

elihw:                                # DONE: Abbruchbedingung erfüllt

```

Mit der **.space** Direktive können wir eine bestimmte Größe an Speicher reservieren:

- Bsp.: **.space 52** reserviert 52 Byte (für 13 Integer)

Damit lassen sich auch Arrays anlegen und befüllen:

```
.data
feld: .space 52                # feld: ARRAY [0..12] OF INTEGER;

.text
main:
    li    $t0, 0
for:
    bgt   $t0, 48, rof         # FOR i := 0 TO 12 DO
    sw    $t0, feld($t0)       # feld[i] := i;
    addi  $t0, $t0, 4          # i += 1 (Nächstes Wort)
    b     for                  # Wiederholen
rof:
    # DONE
```

In vielen Programmiersprachen kennt man eine **switch** Anweisung.

- Beispiel Java;

```
switch(ausdruck) {  
    case konstante_1: anweisung_1;  
    case konstante_2: anweisung_2;  
    ...  
    case konstante_n: anweisung_n;  
}
```

Die Vergleiche `ausdruck==konstante_1`, `ausdruck==konstante_2`,... nacheinander zu testen wäre zu ineffizient.

| Befehl | Argumente | Wirkung                                 | Erläuterung   |
|--------|-----------|-----------------------------------------|---------------|
| jr     | Rs        | unbedingter Sprung an die Adresse in Rs | Jump Register |

**jr** ermöglicht uns den Sprung an eine erst zur Laufzeit ermittelte Stelle im Programm.

**switch** Konstrukt lässt sich über Sprungtabelle realisieren.

- Anlegen eines Feldes mit den Adressen der Sprungziele im Datensegment
- Adressen stehen schon zur Assemblierzeit fest
  - Zur Laufzeit muss nur noch die richtige Adresse geladen werden.



```
.data
jat: .word case0, case1, case2, case3, case4  # Sprungtabelle wird zur
                                              # Assemblerzeit belegt.

.text
main:

    li $v0, 5                # read_int
    syscall

    blt $v0, 0, error        # Eingabefehler abfangen: $v0 ist die Eingabe
    bgt $v0, 4, error

    mul $v0, $v0, 4          # 4-Byte-Adressen
    lw  $t0, jat($v0)        # $t0 enthält Sprungziel
    jr  $t0                  # springt zum richtigen Fall
```

```
case0: li    $a0, 0    # tu dies und das
      j      exit

case1: li    $a0, 1    # tu dies und das
      j      exit

case2: li    $a0, 2    # tu dies und das
      j      exit

case3: li    $a0, 3    # tu dies und das
      j      exit

case4: li    $a0, 4    # tu dies und das
      j      exit

error: li    $a0, 999  # tu dies und das

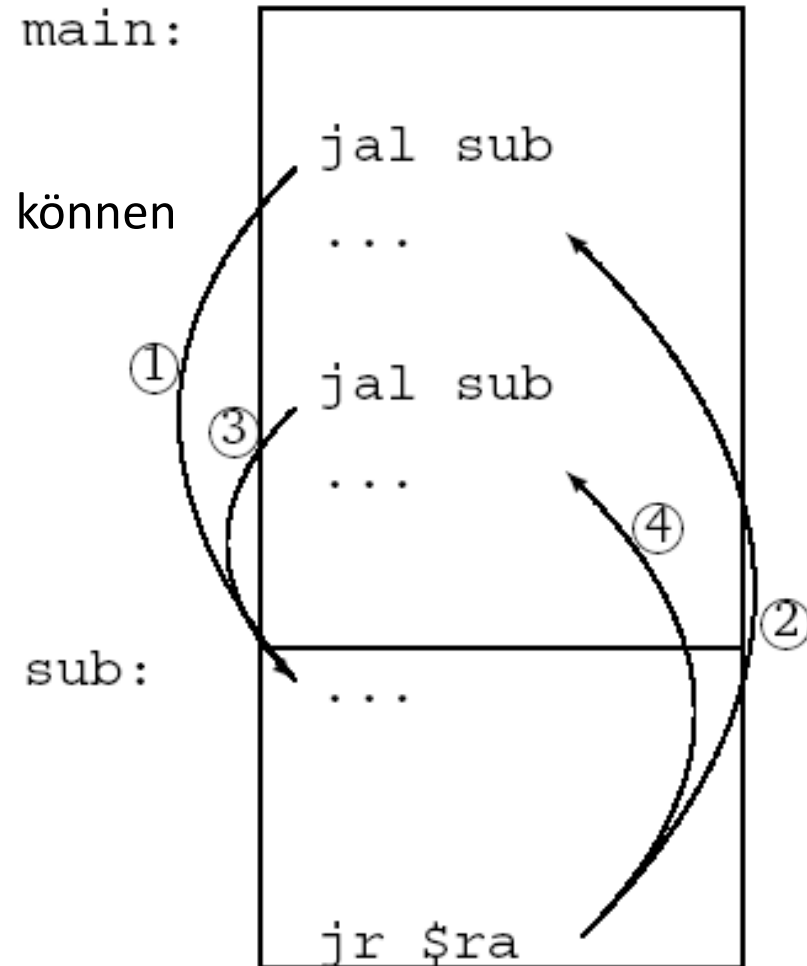
exit:  li    $v0, 1     # print_int
      syscall

      li    $v0, 10    # Exit
      syscall
```

## Unterprogramme:

- In Hochsprachen Prozeduren, Methoden
- Programmstücke, die von unterschiedlichen Stellen im Programm angesprungen werden können
- Dienen der Auslagerung wiederkehrender Berechnungen
- Nach deren Ausführung: Rücksprung zum Aufrufer

**jal** speichert richtige Rücksprungadresse (Adresse des nächsten Befehls im aufrufenden Programm) im Register **\$ra**.



## Einführung in die Assemblerprogrammierung mit dem MIPS Simulator SPIM

- Compiler, Interpreter, Assembler
- MIPS Prozessor
- CISC / RISC
- Little-endian, Big-endian
- Aufbau & Speicher (Daten, Text und Stack Segment)
- Daten & Zeichenketten (word, byte, strings, ...)
- SPIM-Befehle (`lw`, `sw`, `add`, ...)
- Sprünge, IF, SWITCH, Schleifen (`b`, `j`, `jal`, `beqz`, ...)

Der überwiegende Teil dieser Vorlesung ist dem SPIM-Tutorial von Reinhard Nitzsche entnommen:

- <http://www.mobile.ifi.lmu.de/lehrveranstaltungen/rechnerarchitektur-sole17/>