LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Vorlesung Rechnerarchitektur

Sommersemester 2015

Lorenz Schauer

30. April 2015

Agenda

Kapitel 1: Einleitung & Hinführung zum Assembler

- Compiler, Interpreter, Assembler
- MIPS Prozessor
  - Historie: CISC / RISC
- Einschub: Byte-Order

Kapitel 2: Einführung in die Assemblerprogrammierung mit SPIM

- Aufbau & Speicher
- Daten & Zeichenketten
- SPIM-Befehle
- Sprünge, IF, SWITCH, Schleifen
- Unterprogramme
- Call-by-value vs. Call-by-reference
- Unterbrechungen & Ausnahmen

Grundproblem:

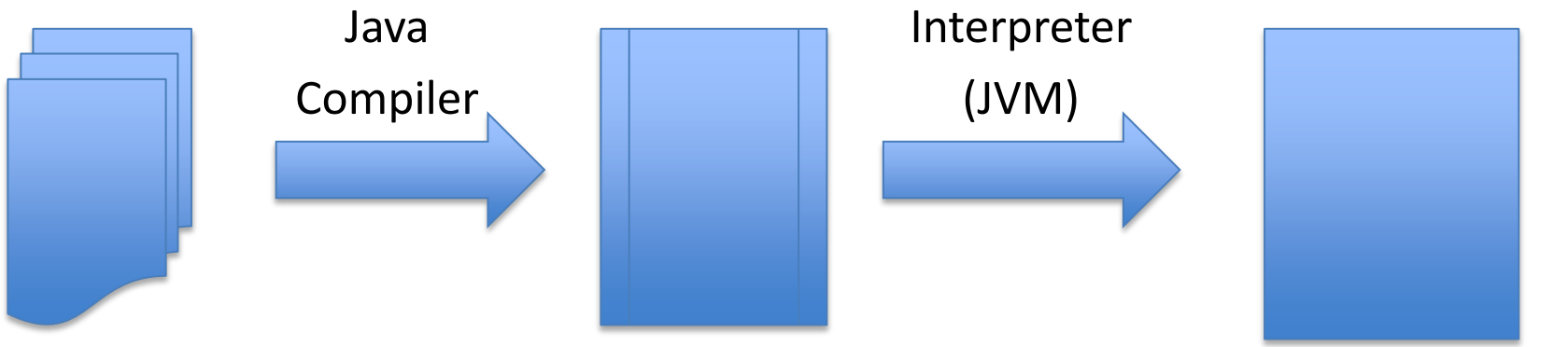
- Maschinensprache für den Menschen höchst umständlich
- Maschinen können aber nur Maschinensprache (primitive Befehle) verstehen

Lösung:

- Menschen nutzen einen anderen Befehlssatz als Sprache L1 (Bsp.: C++)
- Maschinen verarbeiten eine Übersetzung (Compiler) bzw. Interpretation (Interpreter) von L1, hier als L0 bezeichnet.
- **Compiler**:
  - Vollständige Übersetzung des Programms in 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

Beispiel: Java

Auch hybride Ansätze möglich: Beispiel Java

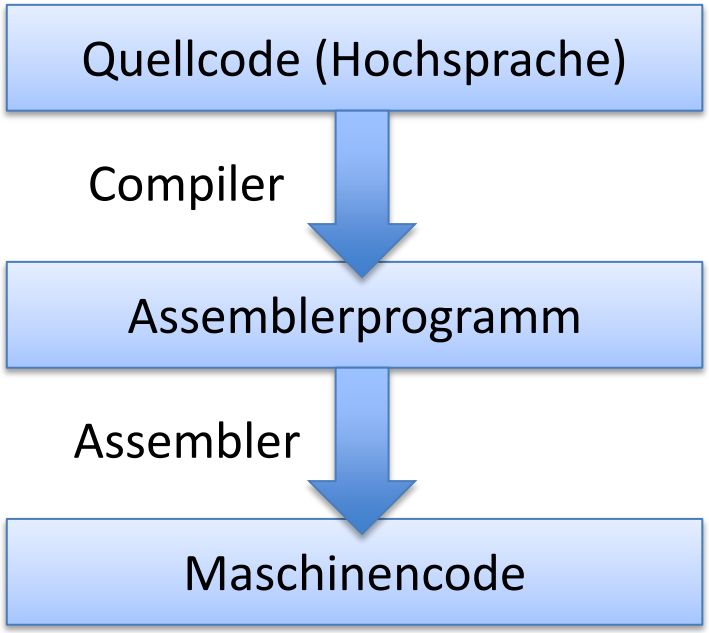


- Quellcode wird von Java Compiler in Bytecode übersetzt
- JVM interpretiert den Bytestream als nativen Maschinencode, der vom OS ausgeführt werden kann

Vom Programm zum Maschinenprogramm

Vereinfachter, abstrakter Ablauf:

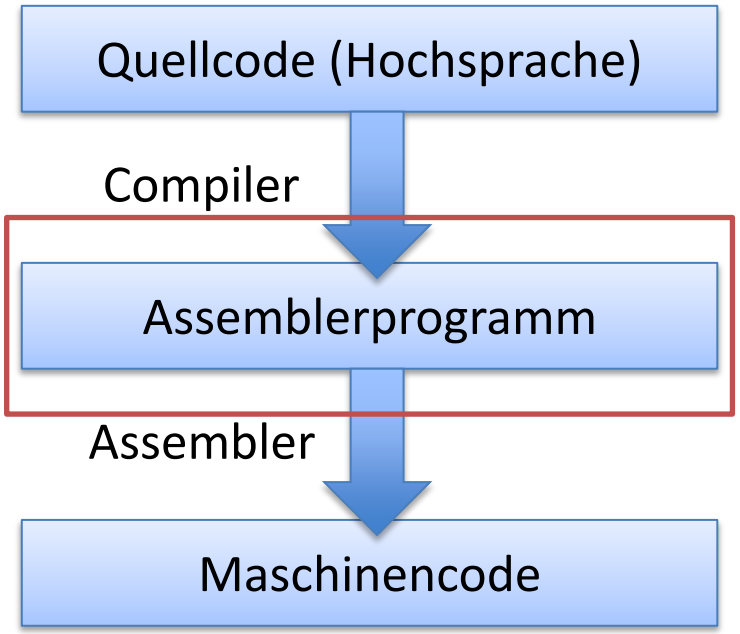
- Hochsprache (Bsp.: C++) wird mittels Compiler in eine Assemblersprache übersetzt
- Compiler analysiert das Programm und erzeugt Assemblercode
  - Für Menschen verständlicher Maschinencode
- Assembler übersetz Assemblercode in Maschinencode
  - Maschinensprache bestehend aus 0 und 1
  - Für Menschen schwer verständlich



Vom Programm zum Maschinenprogramm

Vereinfachter Ablauf:

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



Fokus nun auf Maschinenebene & Assemblerprogrammierung!

Assemblerprogrammierung

Assemblersprache:

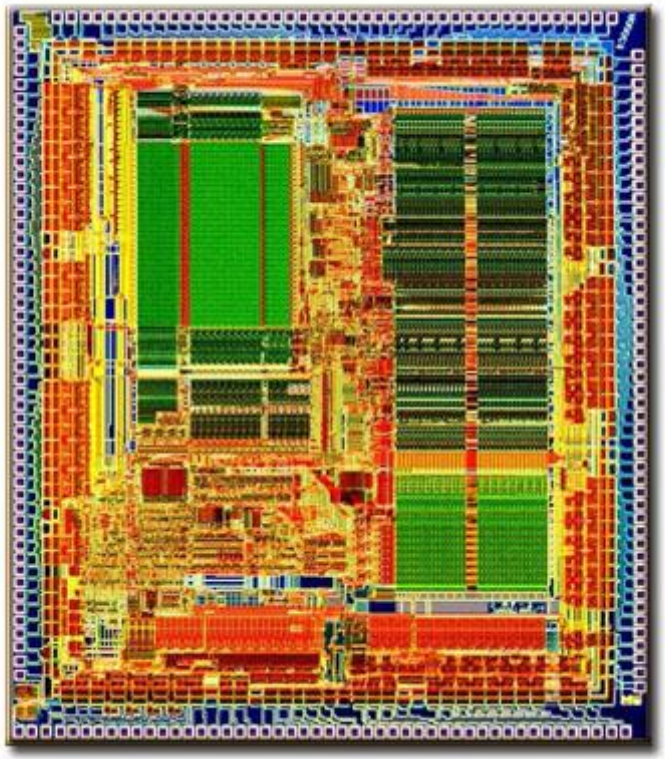
- Hardwarenahe Programmiersprache
- Wird von Assembler direkt in ausführbaren Maschinencode umgewandelt
- Alle Verarbeitungsmöglichkeiten des Mikrokontrollers werden genutzt
- Hardwarekomponenten können direkt angesteuert werden
- Erlauben Namen für Instruktionen, Speicherplätze, Sprungmarken, etc.
- 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-Architektur (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages)

- Mikroprozessor ohne Pipeline-Sperren
- RISC-Prozessorarchitektur
- 1981: von John Hennessy entwickelt (Standford-Universität)
- 1984: Weiterentwicklung durch MIPS Computer Systems Inc., heute MIPS Technologies.
- 1991: Mit R4000 auf 64 Bit erweitert (ursprünglich 32-Bit)
- Früher: vor allem in klassischen Workstations und Servern
- Heute: Haupteinsatzbereich im Bereich Eingebettete Systeme



MIPS R3000

RISC / CISC

Unterscheidung zwischen:

- **RISC** = Reduced Instruction Set Computer
- **CISC** = Complex Instruction Set Computer

Historie:

- Beginn: Mikroprozessoren waren alle RISC Prozessoren.
- Dann: Integration immer weiterer Funktionen
  => Immer komplexere Prozessor Designs
- Analyse zeigte:
  - ca. 95% aller Maschinenbefehle in Programmen sind einfache Befehle
- Also:
  - RISC: Entfernen komplexer Befehle (durch Software realisiert)
    - schnellere Ausführung von Befehlen (keine Interpretation nötig)
  - CISC: Prozessoren haben oft einen RISC Kern
    - Komplexe CISC-Instruktionen werden in Folge von RISC-Instruktionen übersetzt.

CISC vs. RISC

CISC CPUs:

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

Systeme mit RISC CPU:

- Leistungsstarke eingebetteten Systemen (Druckern, Router)
- Workstations
- Supercomputern der Spitzenklasse
- Beispiele:
  - Cisco: Router und Switches bis zur Enterprise-Klasse
  - IBM: Supercomputer, Mittelklasse-Server, Workstations
  - Nintendo: Gamecube, Wii Spielkonsolen
  - Microsoft: Xbox 360 Spielkonsole
  - Motorola: Bordcomputer für PKW und andere Fahrzeuge

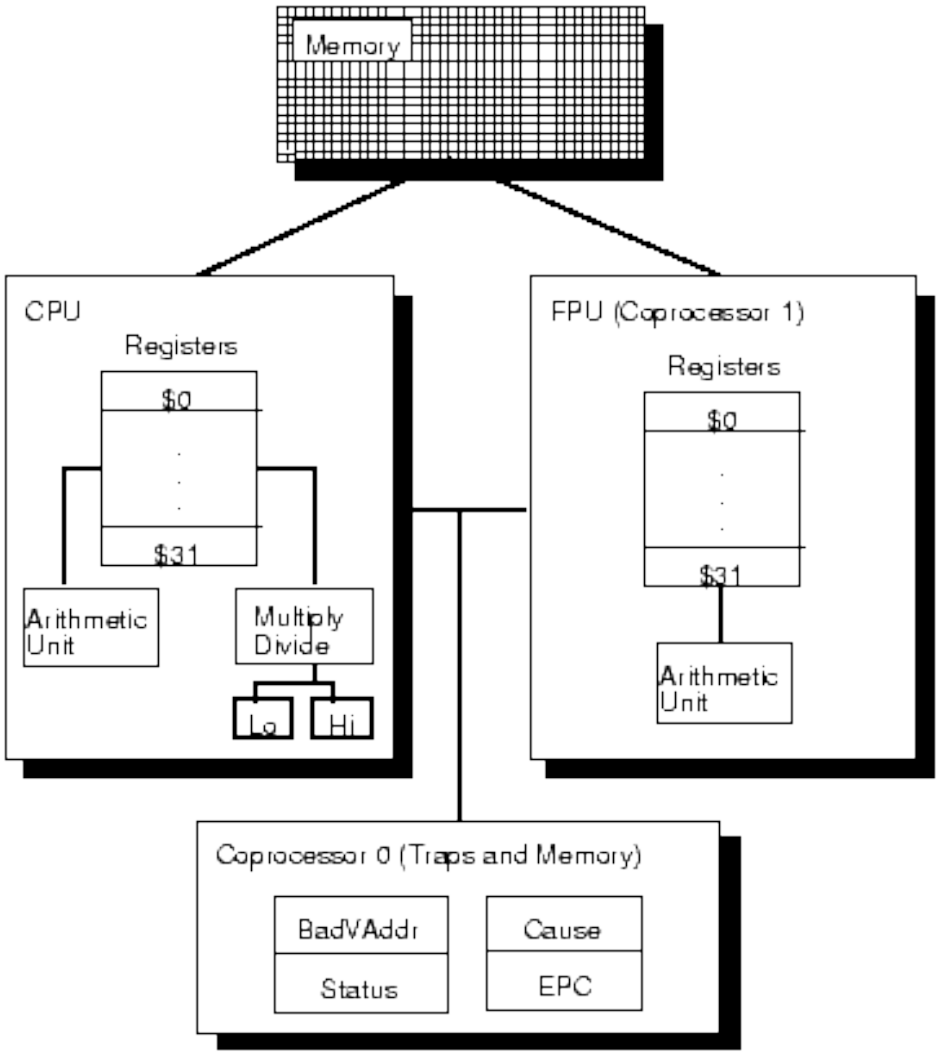
Systeme mit RISC CPU: ARM

ARM (Advanced RISC Machines)

- hohe Leistung, geringen Stromverbrauch, niedrige Kosten
- Typisch 400 MHz – 2 GHz (2011)
- Milliarden ARM-CPUs im Einsatz in
  - Apple: iPods (ARM7TDMI SoC), iPhone (Samsung ARM1176JZF), IPod touch (ARM11)
  - Canon: IXY Digital 700 Kamera (ARM-basiert)
  - Hewlett-Packard: HP-49/50 Taschenrechner (ARM9TDMI)
  - Linksys: NSLU2 Netzwerkspeicher/NAS [Intel XScale IXP420]
  - Nintendo: Game Boy Advance (ARM7), Nintendo DS (ARM7, ARM9)
  - Sony: verschiedene Mobiltelefone, Network Walkman (Eigenentwicklung, ARM-basiert)

Struktur des MIPS Prozessors

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



MIPS-Register (32 Bit breit)

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.

MIPS-Register 16-31

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
$sp	29	Stackpointer Zeigt auf das erste freie Element des Stacks.
$fp	30	Framepointer, Zeiger auf den Prozedurrahmen
$ra	31	Return Adresse

Adressierung

Adressierung: byteweise!

Wort mit Adresse n => Nächstes Wort: Adresse n+4

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

Einschub: Byte-Reihenfolge (Byte Order)

Mehr als ein Byte für die Speicherung einer Zahl:

- Bytes können in
  - aufsteigender
  - 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.

Beispiel

Speicherung der Dezimalzahl 1296650323 als 32-Bit-Werte:

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

Interpretation des 32-Bit-Wertes als Zeichenkette:

- 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	...

Unterschiede bei Konversionen

Beispiel:

- Konversion einer Zwei-Byte- in eine Vier-Byte-Zahl

Little-Endian-Maschine:

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

Big-Endian-Maschine:

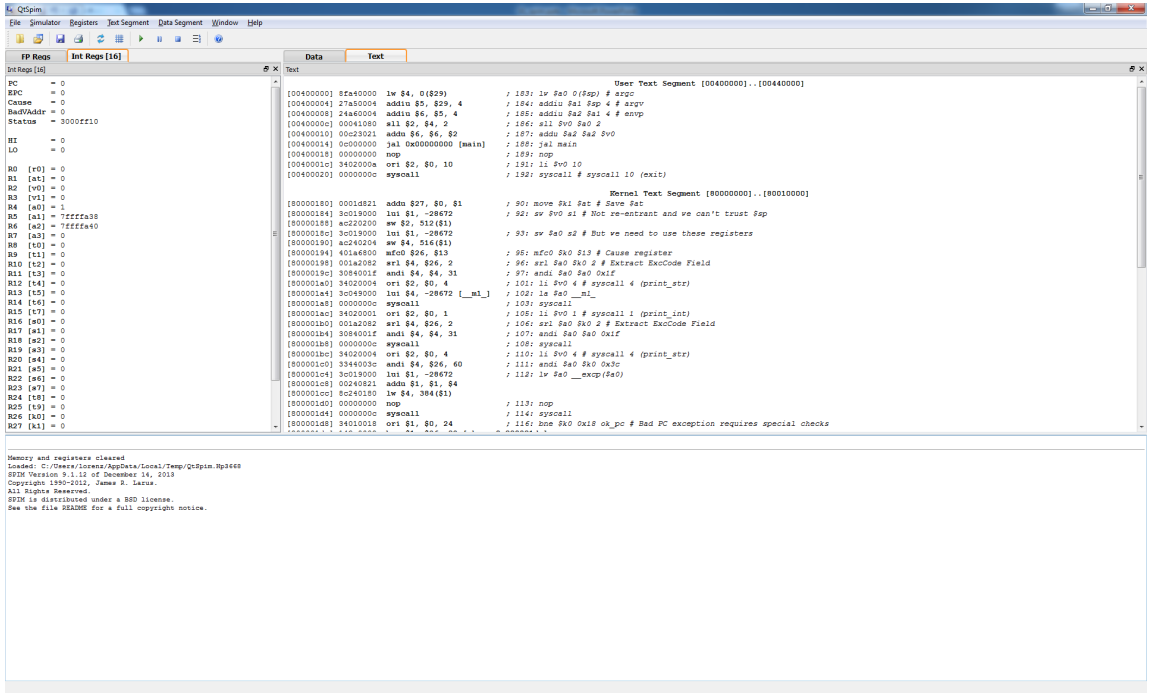
- Wert muss im Speicher um zwei Byte verschoben werden.

Umgekehrte Umwandlung:

- Einfacher auf Little-Endian-Maschine
- höherwertige Bytes werden verworfen
- Speicheradresse bleibt gleich

Einführung in die Assemblerprogrammierung

mit dem MIPS R2000 Simulator SPIM



Einleitung: SPIM

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

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

- http://www.mobile.ifi.uni-muenchen.de/studium_lehre/sose15/rechnerarchitektur/spim_tutorial_de.pdf

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://pages.cs.wisc.edu/~larus/spim.html#qtspim
- Bsp.: MARS: http://courses.missouristate.edu/KenVollmar/MARS/

Ausführung im Simulator

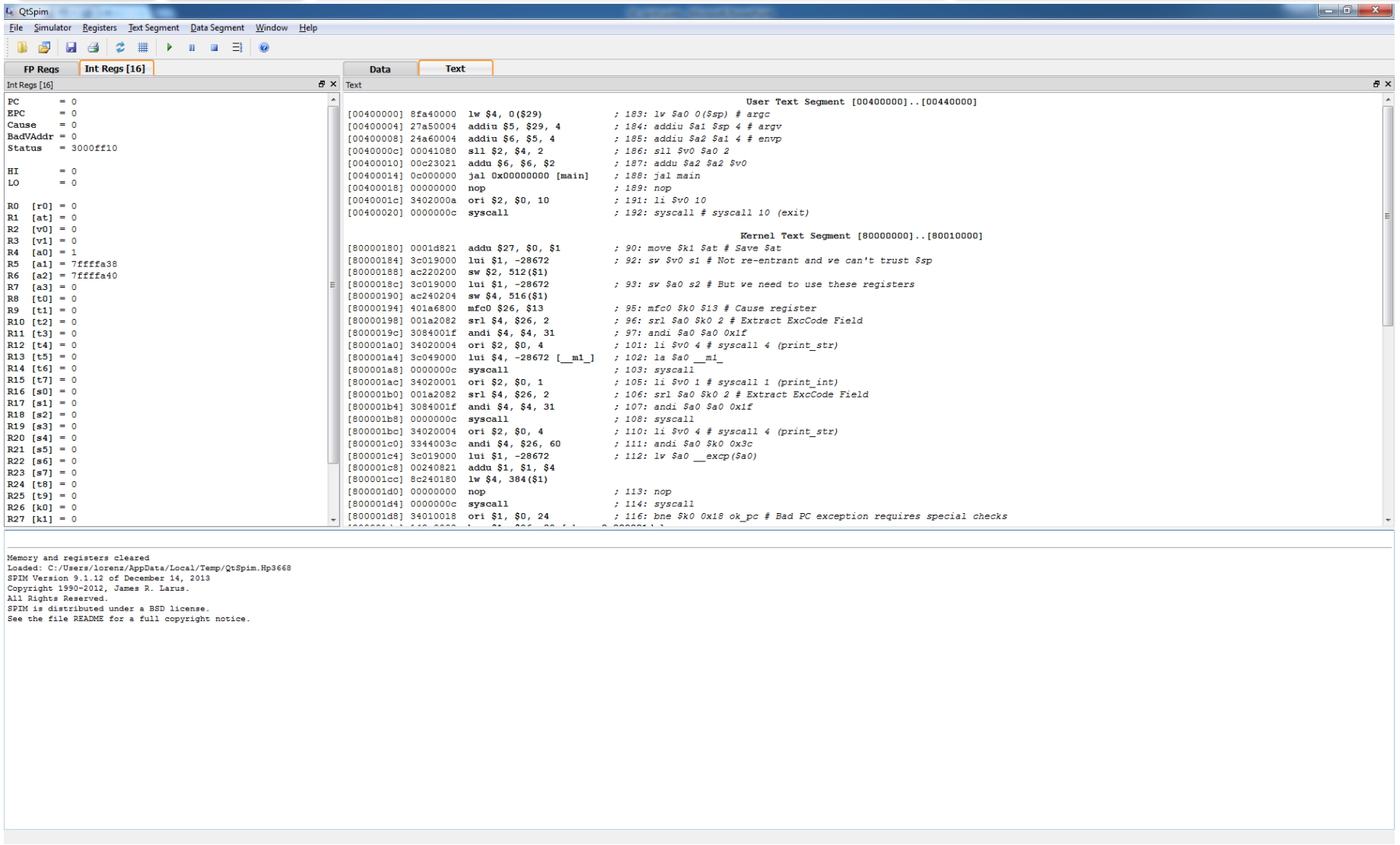
Verschiedene Simulatoren stehen zur Verfügung (siehe Website)

- Einfache Simulatoren:
  - Linux: **spim, xspim, qtspim**
  - Windows: **PCSpim, Qtspim**

- **MARS** (MIPS Assembler and Runtime Simulator)
  - in Java geschriebene Entwicklungsumgebung

  - MARS kann mehr als spim oder xspim!
    - Wir verwenden hier lediglich den Funktionsumfang den auch **spim** bietet!

Beispiel QtSpim



Load and Store Architektur

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.

Java: Stack Architektur

Zum Vergleich:

- Java hat eine Stack Architektur:
- Alle Argumente für die Java Maschinenbefehle werden vom Stack geholt und die Ergebnisse wieder auf den Stack geschrieben.
- Daher brauchen fast alle Java Maschinenbefehle keine Argumente (sie wissen, dass die auf dem Stack zu finden sind).
- Für die Java Maschinenbefehle genügt der op-code, und der passt in ein Byte.

Daher heißt die Java Maschinensprache auch **Bytecode**.

Trennung von Programm und Daten

Grundprinzip (Von-Neumann):

- Gemeinsamer Speicher für Daten und Programme

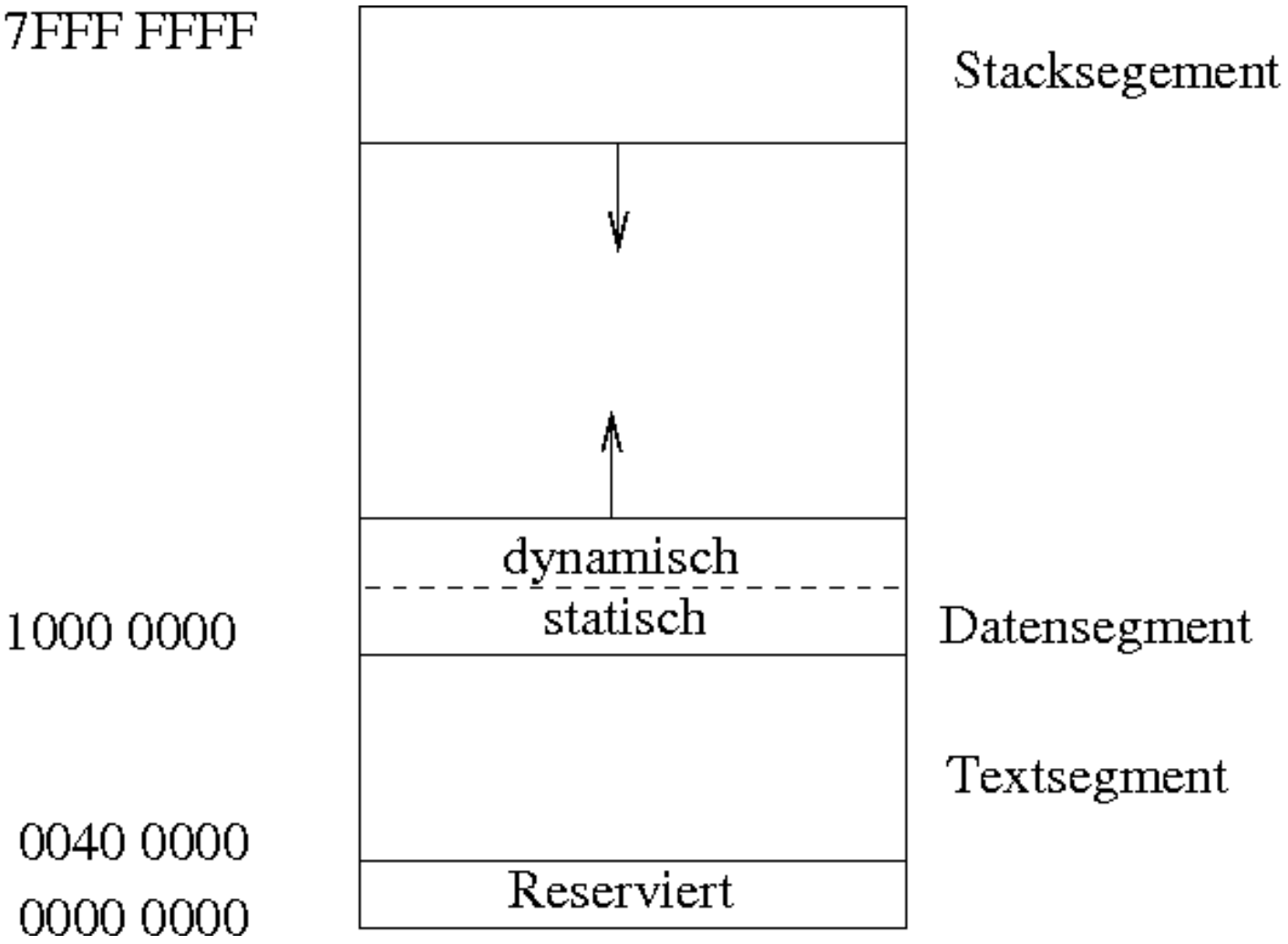
SPIM teilt den Hauptspeicher in **Segmente,** um Konflikte zu vermeiden:

- **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 Segement

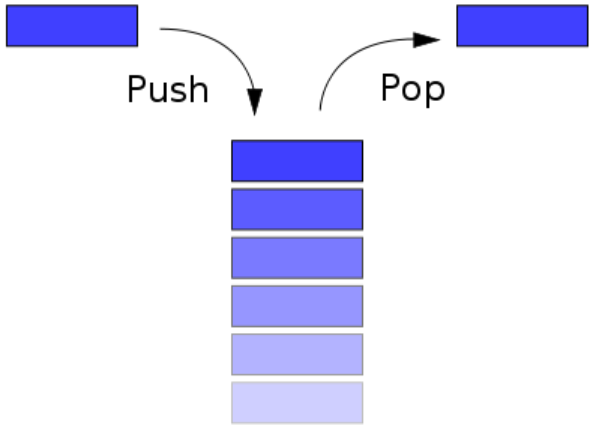
Speicherlayout



(Hardware-) Stack

Dient der Reservierung von und dem Zugriff auf Speicher

- Feste Startadresse (Meist am Ende des HS und wächst gegen 0)
- Variable Größe (nicht Breite!)
  - BS muss verhindern, dass Stack in das Daten-Segment wächst
- Arbeitet nach dem LIFO (Last In–First Out)-Prinzip



Zwei Basis-Operationen

- Push:
  - Ablegen eines Elements auf dem Stack
- Pop:
  - Entfernen des obersten Elements vom Stack

Verwendung bei MIPS (hauptsächlich)

- Sichern und Wiederherstellen von Registerinhalten vor bzw. nach einem Unterprogrammaufruf.

Assemblerprogramm, Beispiel

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

```
        .data <-------------- Datensegment
x:      .word 12
y:      .word 14
z:      .word 5                Textsegment
U:      .word 0
                                        Kommentarzeichen
        .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
```

Erklärung

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.

Ganzzahltypen

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)

Integer Daten anlegen

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*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.

Ein weiteres Beispiel:

```
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)

Zeichenketten


```
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.)

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	\0
Little Endian	l	l	a	H	e	W		o	\0	\0	t	l

Sonderzeichen

Innerhalb eines Strings sind folgende Kombinationen erlaubt:

\n (neue Zeile)
\t (Sprung zum nächsten Tabulator)
\" Das doppelte Anführungszeichen.

Beispiel:

```
a: .ascii "ab\ncd\tef\"gh\""
```

könnte ausgedruckt so aussehen:

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

(\ ist das sog. "escape Zeichen")

Datenausrichtung im Datensegment

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.

Aufbau einer Assembler-Befehlszeile

```
<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 i*n folgender Reihenfolge auf:*
  1.) Register des Hauptprozessors, zuerst das Zielregister,
  2.) Register des Coprozessors,
  3.) Adressen, Werte oder Marken

Notation der Befehle

Befehl	Argumente	Wirkung	Erläuterung
div	Rd,Rs1,Rs2	RD=INT(Rs1/Rs2)	divide
li	Rd,Imm	Rd=Imm	Load Immediate

Erläuterungen:

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

Beispiele:

```
div $t0,$t1,$t2
```

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

Ladebefehl und Adressierung

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

`lw` lädt die Daten aus der angegeben 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).

Ladebefehl und Adressierung

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 angegeben Adresse **`Adr`** in das Zielregister **`Rd`**.

Beispiel für Adressierung


```
   .data
var:  .word 20, 4, 22, 25, 7
   .text
main: lw $t1, var        # $t1 enthaelt "20"
                  # (direkte Adr.)
   lw $t1, var+4      # $t1 enthaelt "4"
                  # (direkte Adr.)
   lw $t2, var($t1)      # $t2 enthaelt "4"
                  # (indexierte Adr.)
   lw $t2, var+8($t1)    # $t2 enthaelt "25"
                  # (indexierte Adr.)
   la $t1, var        # Adr. von "20" in $t1
   lw $t2, ($t1)      # $t2 enthaelt "20"
                  # (indirekte Adr.)
```

Weitere Ladebefehle

`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

Speicherbefehle

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

Register-Transfer Befehle

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

Addition und Subtraktion

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)

Betriebssystemaufrufe: syscall

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.

syscall in SPIM

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).

Betriebssystemfunktionen des SPIM

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

Weitere Betriebssystemfunktionen

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- Gleit-kommazahl 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		

Beispiel für Betriebssystemaufrufe


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

    .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
```

Beispiel weiter


```
        # 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
```

Anmerkungen zu print_str

Es werden immer alle Zeichen \n mit ausgegeben!

Beispiel:

```
   .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
```

Anmerkungen zu print_str

Ausgabe:

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

Anmerkungen zu read_str

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

```
    .data
txt:    .asciiz "Text="
input:  .ascii  "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
```

Anmerkungen zu read_str

Ausgabe:

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

Anmerkungen zu read_str

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

Logische Befehle

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 Befehlte:

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

Sprungbefehle

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
- branch-Befehle erlaubt SCR-relative Adressierung.
- Der Unterschied wird von der Assemblersprache verwischt.

Beispiel für Sprünge

IF Betrag > 1000
 THEN Rabatt := 3
 ELSE Rabatt := 2
END;
Annahme: Betrag ist in Register $t0, Rabatt soll ins Register $t1

Assemblerprogramm:

```
main:
  ble $t0, 1000, else  # IF Betrag > 1000
  li  $t1, 3           # THEN Rabatt := 3
  b   endif
else:
  li  $t1, 2           # ELSE Rabatt := 2
endif:                 # FI
```

Beispiel für Schleifen


```
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, $t1, $t0           # summe := summe + i
   b       while                   # DONE;
elihw:
```

Felder (Arrays) mit .space


```
 .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
       b      for               # DONE
rof:
```

`.space 52` reserviert 52 Byte (für 13 Integer)

Mehrfache Fallunterscheidung (switch)

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.

Sprungtabellentechnik

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.

Sprungtabellenbeispiel


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

 .text
main:
    li  $v0, 5            # read_int
    syscall
    blt $v0, 0, error     # Eingabefehler abfangen
    bgt $v0, 4, error
    mul $v0, $v0, 4       # 4-Byte-Adressen
    lw  $t0, jat($v0)     # $t0 enthält Sprungziel
    jr  $t0               # springt zum richtigen Fall
```

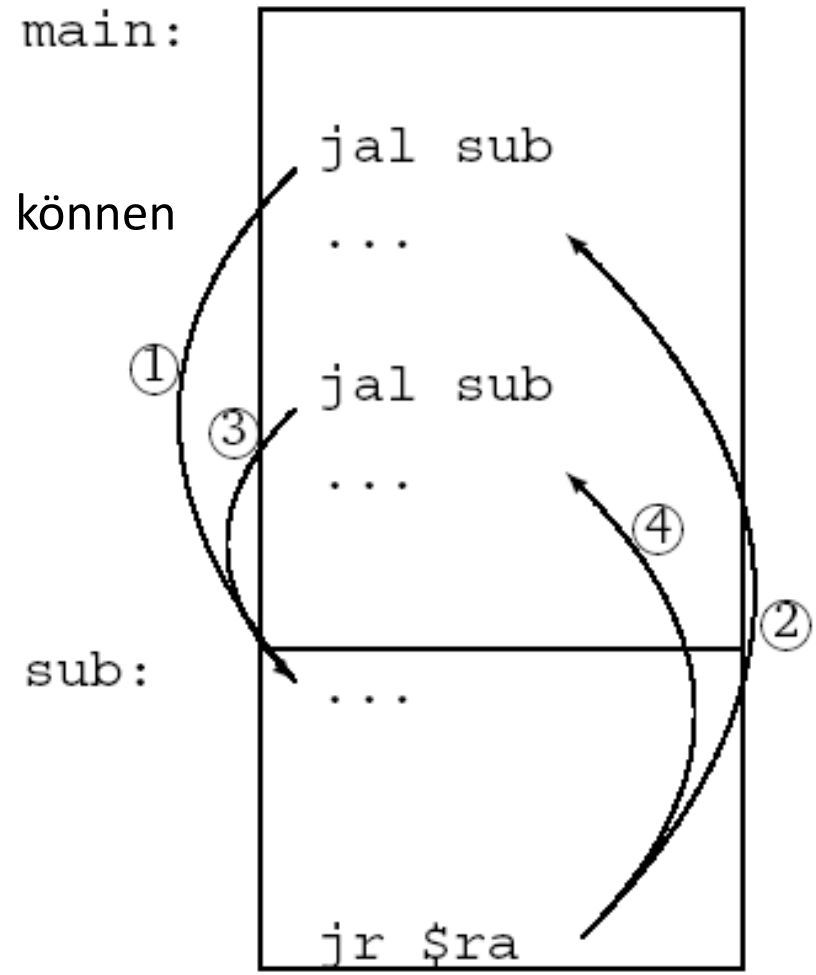
Sprungtabellenbeispiel (weiter)


```
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
```

Unterprogrammaufruf

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.

Parameter für das Unterprogramm

Die meisten Unterprogramme benötigen Eingaben (Parameter) und liefern Ergebnisse.

Bsp. Java:

```
public String myFunction(String param){
  return "Hallo: "+ param;
}
```

Wie erfolgt in SPIM die Übergabe von

- Parametern an das Unterprogramm
- Ergebnisse an das aufrufende Programm?

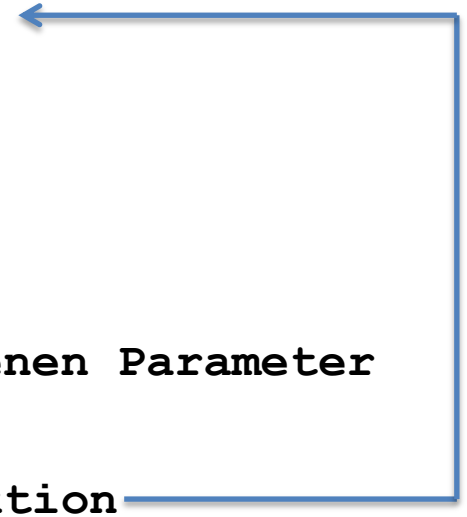
Methode 1:

- Aufrufendes Programm speichert Parameter in die Register $a0,$a1,$a2,$a3
- Unterprogramm holt sie dort ab
- Unterprogramm speichert Ergebnisse in die Register $v0,$v1
- Aufrufendes Programm holt sie dort ab

Beispiel: Dreiecksumfang

Die Prozedur Umfang berechnet den Umfang des Dreiecks mit den Kanten $a0,$a1 und $a2

```
li      $a0, 12     # Parameter für Übergabe an Unterprogramm
li      $a1, 14
li      $a2, 5
jal     uf          # Sprung zum Unterprogramm,
                    # Adresse von nächster Zeile (‘move‘) in $ra
move    $a0, $v0    # Ergebnis nach $a0 kopieren
li      $v0, 1      # 1: ausdrucken
syscall
...
uf:
add $v0,$a0,$a1     # Berechnung mittels übergebenen Parameter
add $v0,$v0,$a2
jr $ra              # Rücksprung zur move Instruktion
```



Parameter für das Unterprogramm

Methode 2:

- Parameter werden auf den Stack gepusht.
- Unterprogramm holt Parameter vom Stack
- Unterprogramm pusht Ergebnisse auf den Stack und springt zurück zum Aufrufer
- Aufrufendes Programm holt sich Ergebnisse vom Stack.

- Funktioniert auch für Unterprogramm das wiederum Unterprogramme aufruft (auch rekursiv).

Beide Methoden lassen sich kombinieren

- Teil der Werte Register
- Teil auf den Stack

Ein weiteres Problem: Register

Problem:

- Ein Unterprogramm benötigt u.U. Register, die das aufrufende Programm auch benötigt
- Inhalte könnten überschrieben werden!

Lösung:

- Vor Ausführung des Unterprogramms Registerinhalte auf dem Stack sichern
- Nach Ausführung des Unterprogramms vorherige Registerinhalte wieder vom Stack holen und wieder herstellen.

Prozedur-Konvention beim SPIM

Prolog des Callers (aufrufendes Programm):

Sichere alle *caller-saved* Register:

- Sichere Inhalt der Register $a0-$a3, $t0-$t9, $v0 und $v1.
- *Callee* (Unterprogramm) darf ausschließlich diese Register verändern ohne ihren Inhalt wieder herstellen zu müssen.

Übergebe die Argumente:

- Die ersten vier Argumente werden in den Registern $a0 bis $a3 übergeben
- Weitere Argumente werden in umgekehrter Reihenfolge auf dem Stack abgelegt (Das fünfte Argument kommt zuletzt auf den Stack)

Starte die Prozedur (jal)

Prozedur-Konvention beim SPIM

Prolog des Callee:

- **Schaffe Platz auf dem Stack (Stackframe)**
  - Stackframe: der Teil des Stacks, der für das Unterprogramm gebraucht wird
  - Subtrahiere die Größe des Stackframes vom Stackpointer:

    ```
    sub $sp, $sp, <Größe Stackframe>
    ```

- **Sichere alle *callee-saved* Register** (Register die in der Prozedur verändert werden)
  - Sichere Register $fp, $ra und $s0-$s7 (wenn sie innerhalb der Prozedur verändert werden)
  - Achtung: das Register $ra wird durch den Befehl jal geändert!

- **Erstelle den Framepointer:**
  - Addiere die Größe des Stackframe zum Stackpointer und lege das Ergebnis in $fp ab.

Prozedur-Konvention beim SPIM

Epilog des Callees:

- **Rückgabe des Funktionswertes:**
  - Ablegen des Funktionsergebnis in den Registern $v0 und $v1
- **Wiederherstellen der gesicherten Register:**
  - Vom Callee gesicherte Register werden wieder hergestellt.
  - Achtung: den Framepointer als letztes Register wieder herstellen!
- **Entferne den Stackframe:**
  - Addiere die Größe des Stackframes zum Stackpointer.
- **Springe zum Caller zurück:**
  - jr $ra

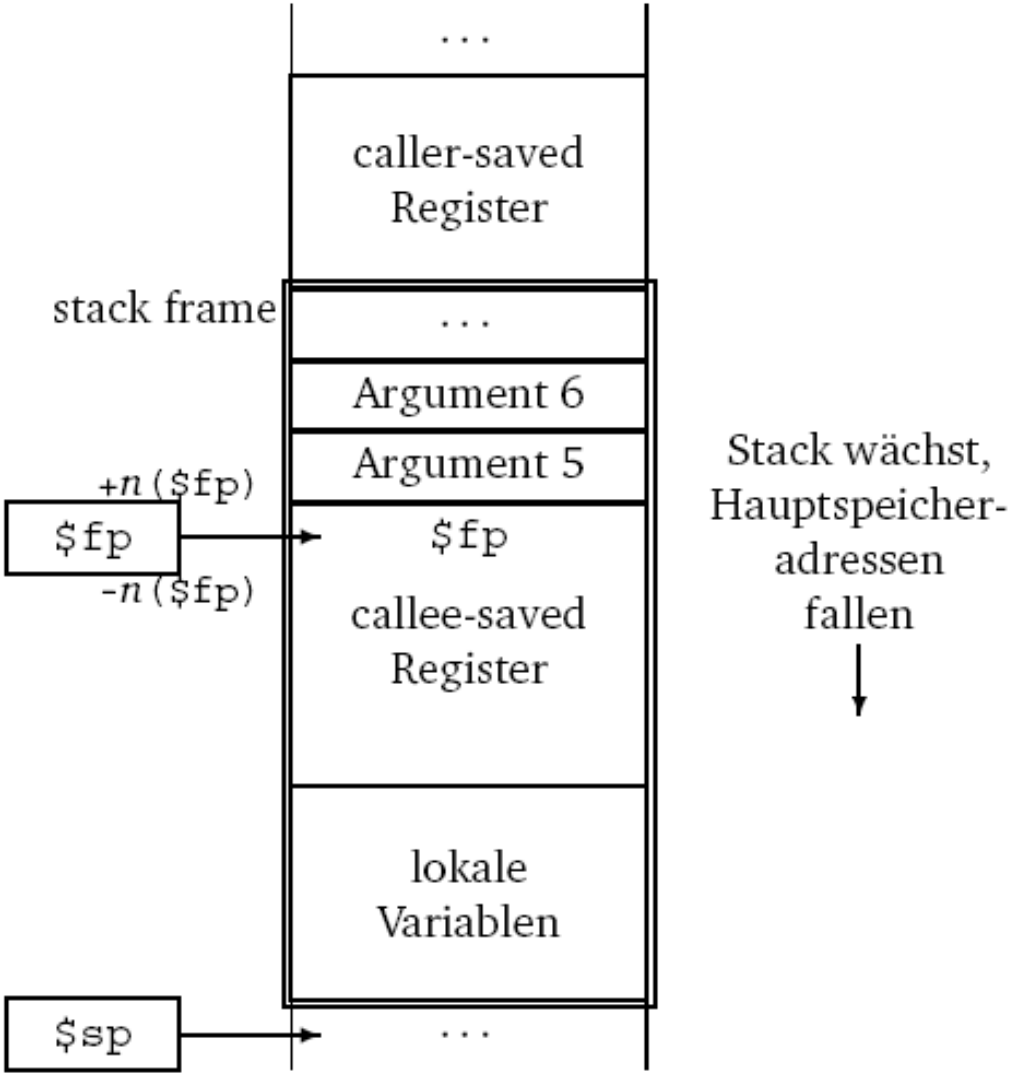
Prozedur-Konvention beim SPIM

Epilog des Callers:

- **Stelle gesicherte Register wieder her:**
  - Vom Caller gesicherte Register wieder herstellen
  - Achtung: Evtl. über den Stack übergebene Argumente bei der Berechnung des Abstandes zum Stackpointer beachten!

- **Stelle ursprünglichen Stackpointer wieder her:**
  - Multipliziere die Zahl der Argumente und gesicherten Register mit vier und addiere sie zum Stackpointer.

Stack-Layout



Call by Value / Call by Reference

Die Werte, die an ein Unterprogramm übergeben werden sind Bitfolgen.
Bitfolgen können sein:

- Daten (call by value) oder
- die Adressen von Daten (call by reference)

Beispiel:

```
.data
x:  .word 23
    .text
main:
    la      $a0,x           # lädt Adresse von x.
    jal     cbr
                            # Was ist jetzt der Wert von x?
cbr:
    lw      $t0,($a0)
    add     $t0, $t0, $t0
    sw      $t0,($a0)
    jr      $ra
```

Extrembeispiel: Übergabe von Arrays

Normalfall:

- Arrays werden an Unterprogramme übergeben, indem man die Anfangsadresse übergibt (call by reference).

Call by Value Übergabe:

- Eine call by value Übergabe eines Arrays bedeutet, das gesamte Array auf den Stack zu kopieren (nicht sinnvoll).

Unterbrechung (Interrupts)

- Ereignis, das asynchron zum Programmablauf eintritt
- Hat keine direkter Abhängigkeit zu bestimmten Befehlen
- Muss (sofort) vom Betriebssystem behandelt werden
- Beispiele:
  - Ein- und Ausgabegeräte, z.B. die Tastatur.
- Unterbrechungen sind nicht reproduzierbar!
- Unterbrechungen können jederzeit während der gesamten Programmausführung auftreten
- Unterbrechungen können wieder unterbrochen werden
  - Priorisierte Interrupts

Ausnahme (exceptions, traps)

- Ereignis, das synchron zum Programmablauf eintritt
- Steht in direktem Zusammenhang mit bestimmten Befehlen
- Muss vom Betriebssystem behandelt werden
- Beispiele:
  - Division durch Null
  - Überläufe
  - ...

Behandlung von Unterbrechungen und Ausnahmen

Auftreten eines Interrupts oder einer Exception

- aktueller Befehl wir abgearbeitet
- Abspeichern aller Informationen zum wiederherstellen des aktuellen Programms (PC, PSW, Register, …)
- Sprung an eine von der CPU-Hardware festgelegte Stelle
  - Beim SPIM: 0x8000 0080
  - Anfang der *Unterbrechungsbehandlungsroutine (ISR oder Interrupthandler)*
  - Interrupthandler behandelt Unterbrechung bzw. die Ausnahme
- Coprozessor 0 stellt dazu einige Register zur Verfügung
  - CPU schreibt dorthin Informationen über den Grund der Unterbrechung oder der Ausnahme
- Interrupthandler kann man direkt programmieren!
  - Aufgabe des Betriebsystems, daher ktext-Direktive an der Stelle 0x8000 0080 verwenden:

```
.ktext 0x8000 0080
```

Zusammenfassung

Struktur eines realen Prozessors & Assemblerprogrammierung mit 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`**,…)
- Unterprogramme ($**`a0`**, caller-saved, callee,…)
- Call-by-value vs. Call-by-reference
- Unterbrechungen & Ausnahmen (**`.ktext 0x8000 0080`**)

Der überwiegende Teil dieses Kapitels ist dem Spim Tutorial entnommen.

Folien teilweise adaptiert von Prof. Dr. Hans Jürgen Ohlbach