LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Vorlesung Rechnerarchitektur

Sommersemester 2016

Lorenz Schauer

12. Mai 2016

Agenda

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
- Unterprogramme
- Call-by-value vs. Call-by-reference

Motivation: Warum Programmieren?

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……

Motivation: Warum Programmieren?

Lösung:

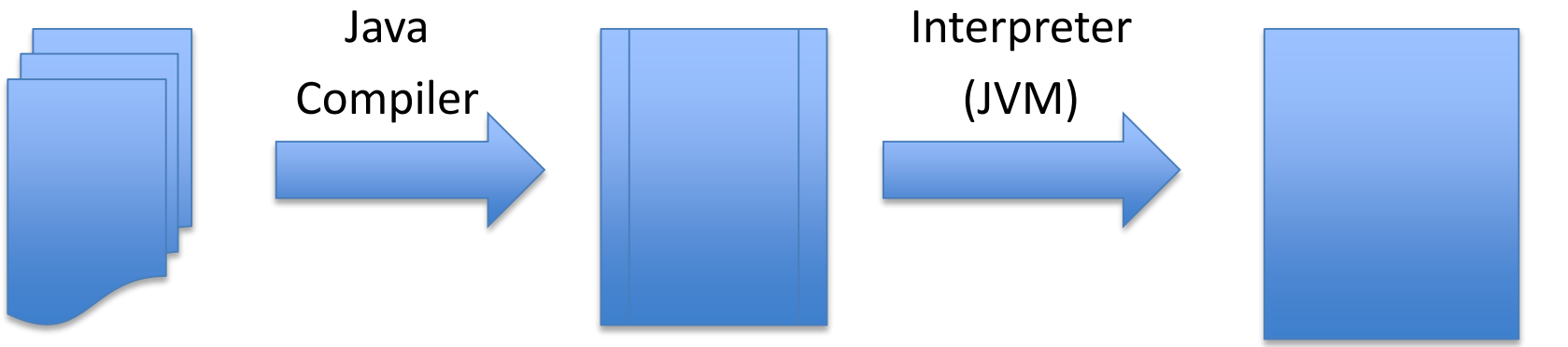
- Menschen nutzen einen anderen Befehlssatz als Sprache L1 (Bsp.: C++, Java, Python, usw.)
- Maschinen verarbeiten eine Übersetzung (Compiler) bzw. Interpretation (Interpreter) von L1, hier als L0 bezeichnet.

- **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

Beispiel: Java

Auch hybride Ansätze möglich: Beispiel Java

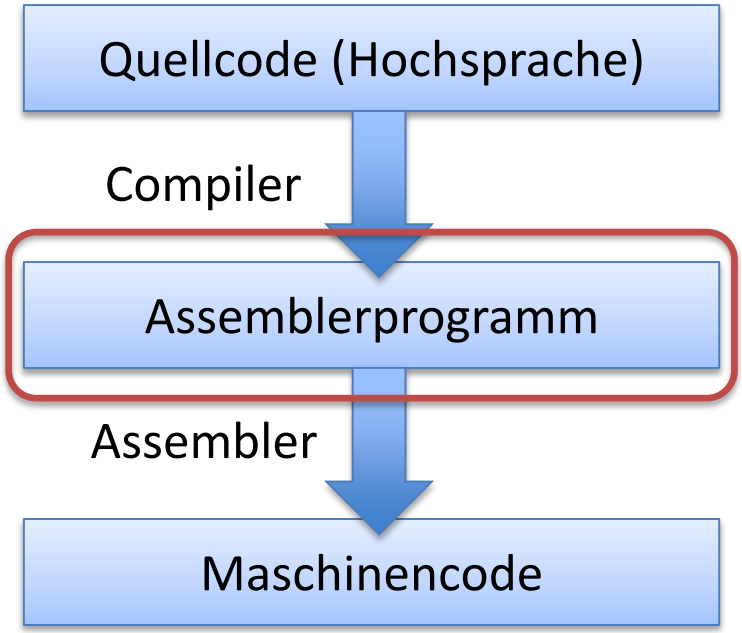


- Quellcode wird von Java Compiler in Bytecode übersetzt
- JVM interpretiert den Bytestream als nativen Maschinencode, der vom OS ausgeführt werden kann

Vom Programm zum Maschinenprogramm

Vereinfachter, abstrakter Ablauf:

- Hochsprache (Bsp.: C++, C#)
- 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

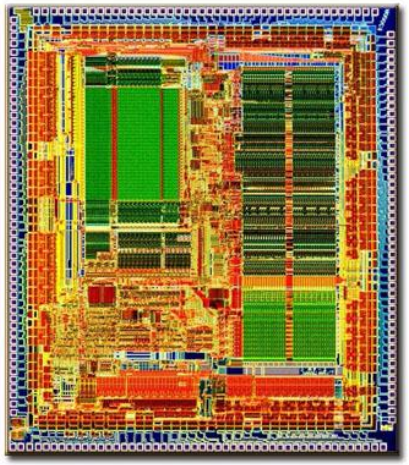


Fokus nun auf Assembler-Programm!

Warum Assembler-Programmierung?

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)



MIPS R3000

MIPS-Architektur (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages)

- 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.

Beispiele für CISC und RISC

CISC CPUs:

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

RISC CPUs:

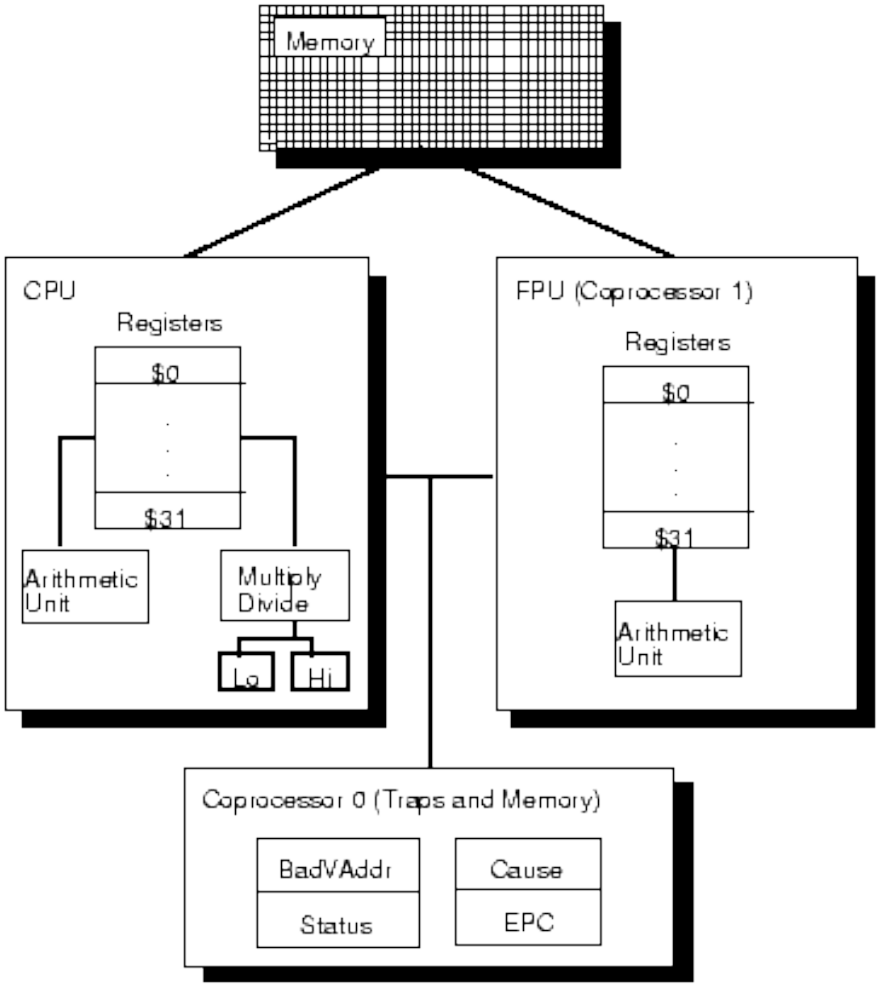
- Leistungsstarke eingebetteten Systemen (Druckern, Router)
- Workstations
- Supercomputern der Spitzenklasse
- ARM Prozessoren (**A**dvanced **R**ISC **M**achines)
- Milliarden ARM-CPUs im Einsatz in
  - hohe Leistung, geringen Stromverbrauch, niedrige Kosten
  - Apple: iPods (ARM7TDMI SoC), iPhone (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)

Struktur des MIPS Prozessors

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

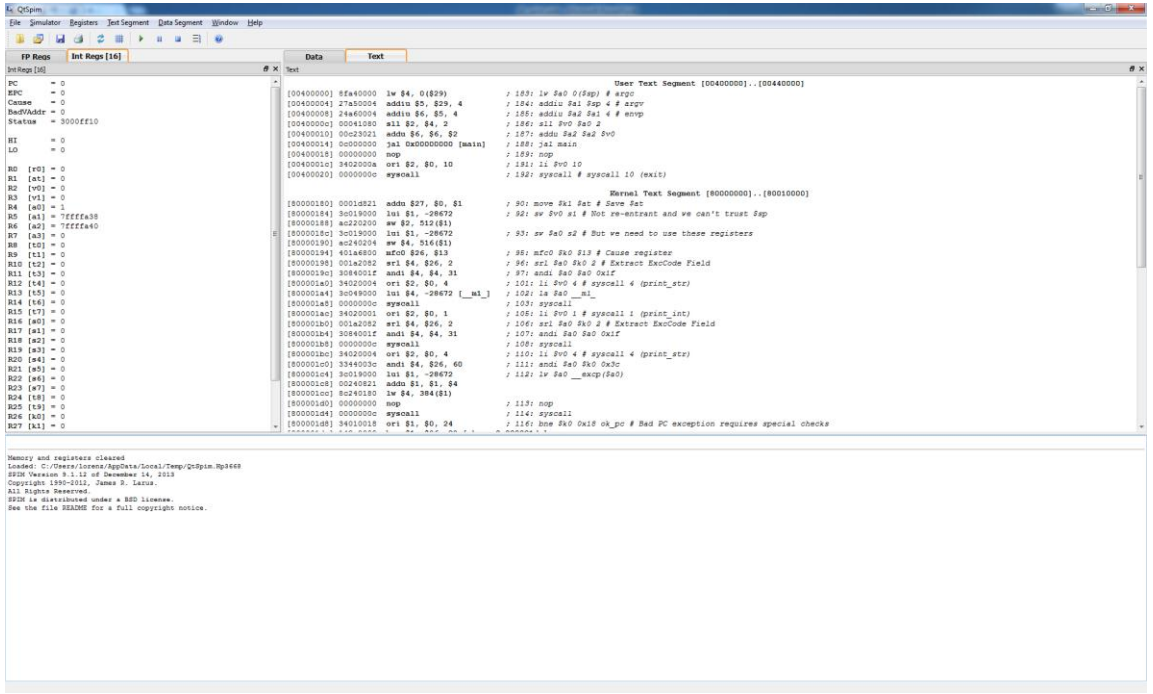
Der SPIM Simulator simuliert 2:

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



Einführung in die Assembler-Programmierung

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.lmu.de/lehrveranstaltungen/rechnerarchitektur-sose16/
- 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**.

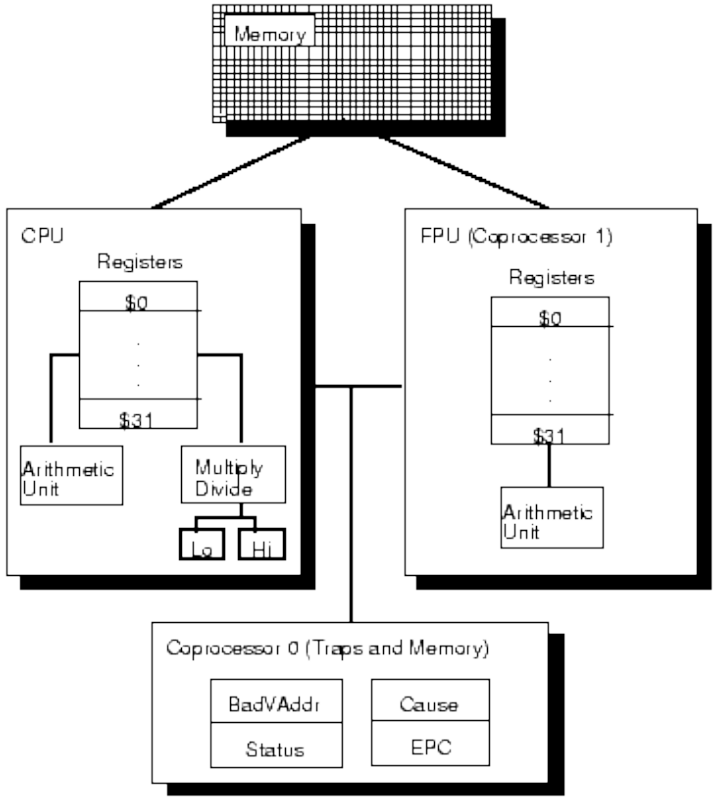
- Auch dieser ist auf der Vorlesungsseite verlinkt:
- Bsp.: QtSpim: http://spimsimulator.sourceforge.net/
- Bsp.: MARS: http://courses.missouristate.edu/KenVollmar/MARS/

Load-Store-Architektur

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!



Adressierung

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.

    $\Rightarrow$ Abhängig von Bus- und Registerbreite.

    $\Rightarrow$ Ist also charakteristisch für eine Computer Architektur

Achtung: Nicht verwechseln mit Adressen!

Adressierung und Wortlänge bei SPIM

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 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)

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.

Beispiel

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	...

Unterschiede bei Konversionen

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	**0x00**	**0x00**

Big-Endian-Maschine:

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

Adresse	1	2	3	4
Wert zuvor	0x23	0xA1		
Wert danach	**0x00**	**0x00**	**0x23**	**0xA1**

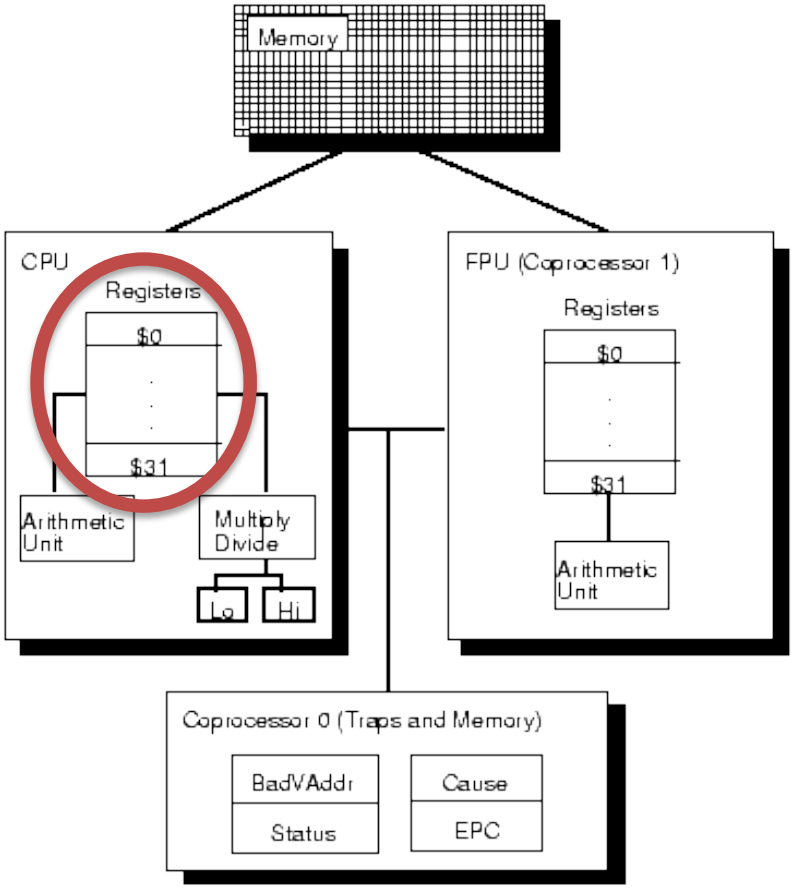
Umgekehrte Umwandlung:

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

Load-Store-Architektur

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

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



MIPS-Register (32 Bit breit)

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.

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	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

Trennung von Programm und Daten

Grundprinzip (Von-Neumann):

- Gemeinsamer Speicher für Daten und Programme

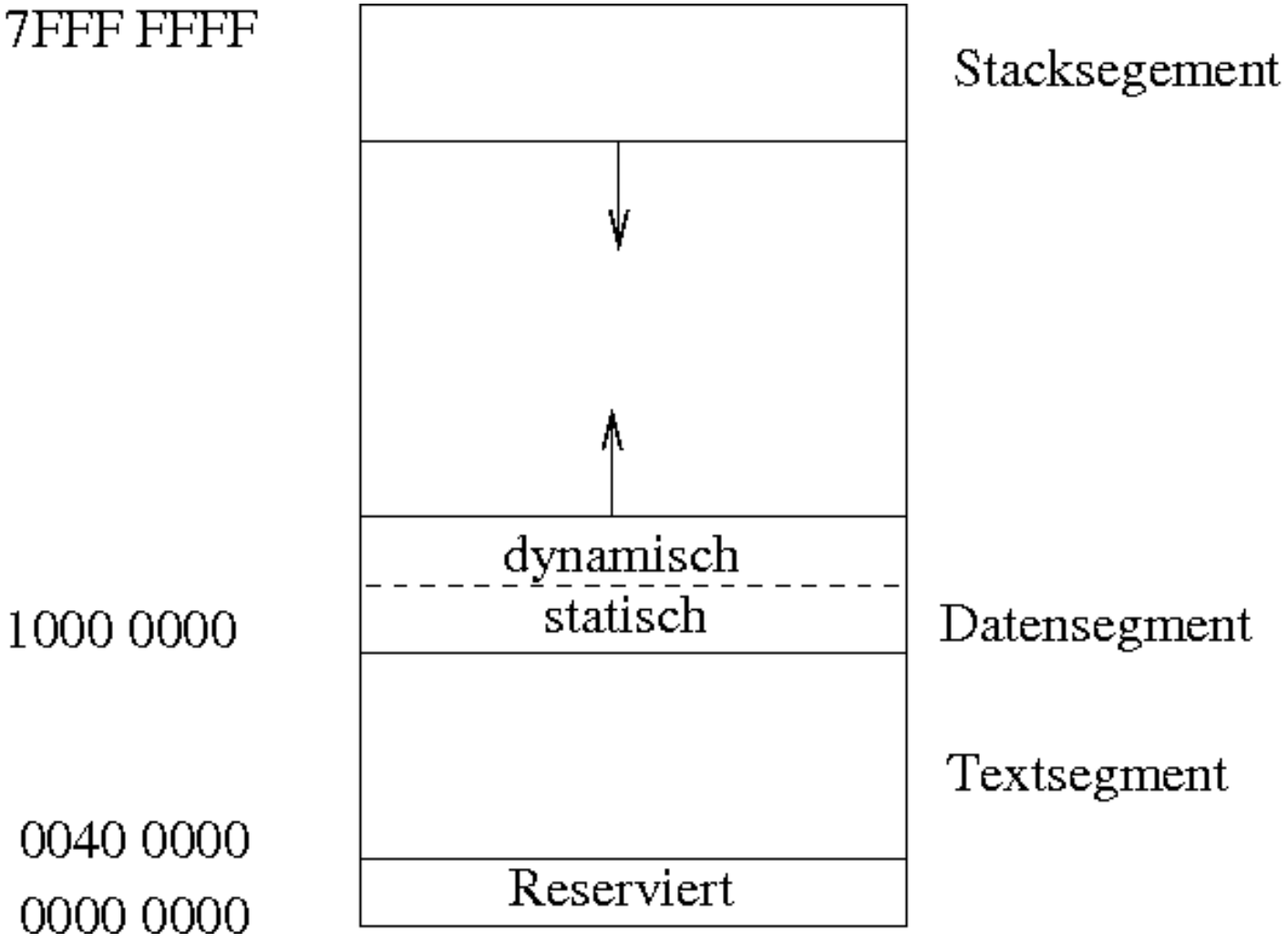
SPIM teilt den Hauptspeicher in **Segmente,** um Konflikte zu vermeiden:

- **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

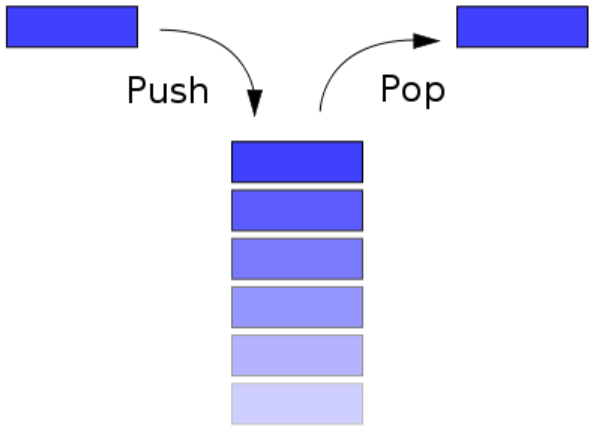
Speicherlayout



(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.
- Stack-Programmierung ist fehleranfällig und erfordert Einhaltung von Konventionen, sonst schwer zu debuggen!

Beispiel

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

        .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 `.data` kennzeichnet den Beginn des Datensegments

Definition der Variablen im Datensegment

Direktive `.text` kennzeichnet den Beginn des Textsegments

Ladebefehle

Arithmetische Befehle

Speicherbefehle

Programm beenden

Marke `main:` als Einstiegspunkt erforderlich!

Kommentare

Beispiel: QtSpim

QtSpim

File Simulator Registers Text Segment Data Segment Window Help

FP Regs | Int Regs [16] | Data | Text

Int Regs [16]

```
PC          = 0
EPC         = 0
Cause       = 0
BadVAddr    = 0
Status      = 3000ff10

HI          = 0
LO          = 0

R0   [r0] = 0
R1   [at] = 0
R2   [v0] = 0
R3   [v1] = 0
R4   [a0] = 1
R5   [a1] = 7ffffa38
R6   [a2] = 7ffffa40
R7   [a3] = 0
R8   [t0] = 0
R9   [t1] = 0
R10  [t2] = 0
R11  [t3] = 0
R12  [t4] = 0
R13  [t5] = 0
R14  [t6] = 0
R15  [t7] = 0
R16  [s0] = 0
R17  [s1] = 0
R18  [s2] = 0
R19  [s3] = 0
R20  [s4] = 0
R21  [s5] = 0
R22  [s6] = 0
R23  [s7] = 0
R24  [t8] = 0
R25  [t9] = 0
R26  [k0] = 0
R27  [k1] = 0
```

Text

```
                                    User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)           ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4        ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4         ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2           ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2         ; 187: addu $a2 $a2 $v0
[00400014] 0c000000  jal 0x00000000 [main]   ; 188: jal main
[00400018] 00000000  nop                     ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10          ; 191: li $v0 10
[00400020] 0000000c  syscall                 ; 192: syscall # syscall 10 (exit)

                                    Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1        ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672          ; 92: sw $v0 s1 # Not re-entrant and we can't trust $sp
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672          ; 93: sw $a0 s2 # But we need to use these registers
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13           ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2          ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f  andi $4, $4, 31         ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4           ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]  ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                 ; 103: syscall
[800001ac] 34020001  ori $2, $0, 1           ; 105: li $v0 1 # syscall 1 (print_int)
[800001b0] 001a2082  srl $4, $26, 2          ; 106: srl $a0 $k0 2 # Extract ExcCode Field
[800001b4] 3084001f  andi $4, $4, 31         ; 107: andi $a0 $a0 0x1f
[800001b8] 0000000c  syscall                 ; 108: syscall
[800001bc] 34020004  ori $2, $0, 4           ; 110: li $v0 4 # syscall 4 (print_str)
[800001c0] 3344003c  andi $4, $26, 60        ; 111: andi $a0 $k0 0x3c
[800001c4] 3c019000  lui $1, -28672          ; 112: lw $a0 __excp($a0)
[800001c8] 00240821  addu $1, $1, $4
[800001cc] 8c240180  lw $4, 384($1)
[800001d0] 00000000  nop                     ; 113: nop
[800001d4] 0000000c  syscall                 ; 114: syscall
[800001d8] 34010018  ori $1, $0, 24          ; 116: bne $k0 0x18 ok_pc # Bad PC exception requires special checks
```

```
Memory and registers cleared
Loaded: C:/Users/lorenz/AppData/Local/Temp/QtSpim.Hp3668
SPIM Version 9.1.12 of December 14, 2013
Copyright 1990-2012, James R. Larus.
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
```

Der Reihe nach: Ganzzahltypen im Datensegment

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
```

Ein weiteres Beispiel:

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

Zeichenketten im Datensegment


```
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

Sonderzeichen

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\ncd\tef\"gh\""
```

könnte ausgedruckt so aussehen:

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

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.

Kommen wir nun zum Textsegment

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 i*n folgender Reihenfolge auf:*

  1.) Register des Hauptprozessors, zuerst das Zielregister,

  2.) Register des Koprozessors,

  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, 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`.

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:

- **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.

Ladebefehl und Adressierung

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

Beispiele für Adressierung


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

.text
main: lw $t1, var
      lw $t1, var+4
      lw $t2, var($t1)
      lw $t2, var+8($t1)
      la $t1, var
      lw $t2, ($t1)
```

$t1=20 -> Direkte Adressierung

$t1=4 -> Direkte Adressierung

$t2=4 -> Indexierte Adressierung

$t2=25 -> Indexierte Adressierung

$t1=Adr(var) -> load address

$t2=20 -> Indirekte Adressierung

Weitere Ladebefehle

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.

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

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

Register-Transfer Befehle

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).

Arithmetische Befehle

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)

Betriebssystemaufrufe: `syscall`

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.

`syscall` in SPIM

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
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 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$
```

Anmerkungen zu read_str

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$
```

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 Befehle:

- `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
- Der Unterschied wird von der Assemblersprache verwischt.

Beispiel für Sprünge

```
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
```

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, $t0, $t1       # summe := summe + i
        b       while               # Schleife Wiederholen;

elihw:                              # DONE: Abbruchbedingung erfüllt
```

Felder (Arrays) mit .space

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
```

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: $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
```

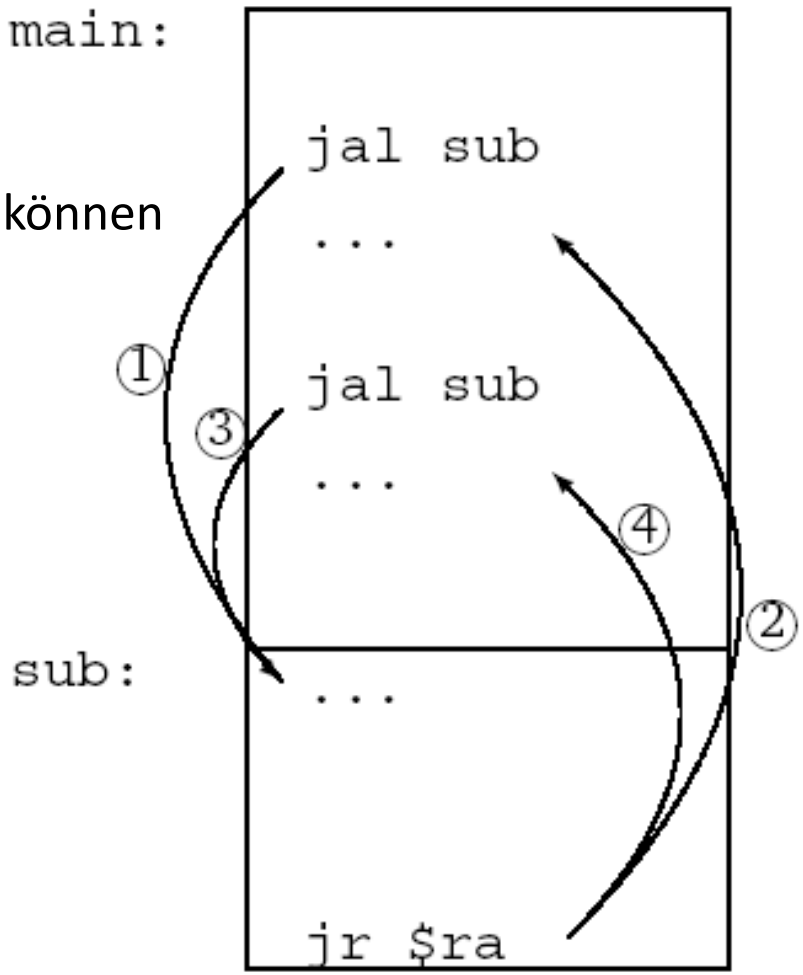
Sprungtabellenbeispiel (weiter)


```
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 nun in SPIM die Übergabe von

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

Methode 1:

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

Beispiel: Dreiecksumfang

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

#Unterprogramm:
uf:
add     $v0, $a0, $a1   # Berechnung mittels übergebenen Parameter
add     $v0, $v0, $a2   # $v0=$a0+$a1+$a2
jr      $ra             # Rücksprung zur move Instruktion
```

Parameter für das Unterprogramm

Was machen wir, wenn wir mehr Argumente übergeben wollen?

- bzw. mehr Ergebnisse bekommen wollen?

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 über Register
- Teil der Werte 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.
- **MIPS-Konvention für Unterprogrammaufrufe beachten!**

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)

Prolog des Callee (aufgerufenes Unterprogramm)

- **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>
```

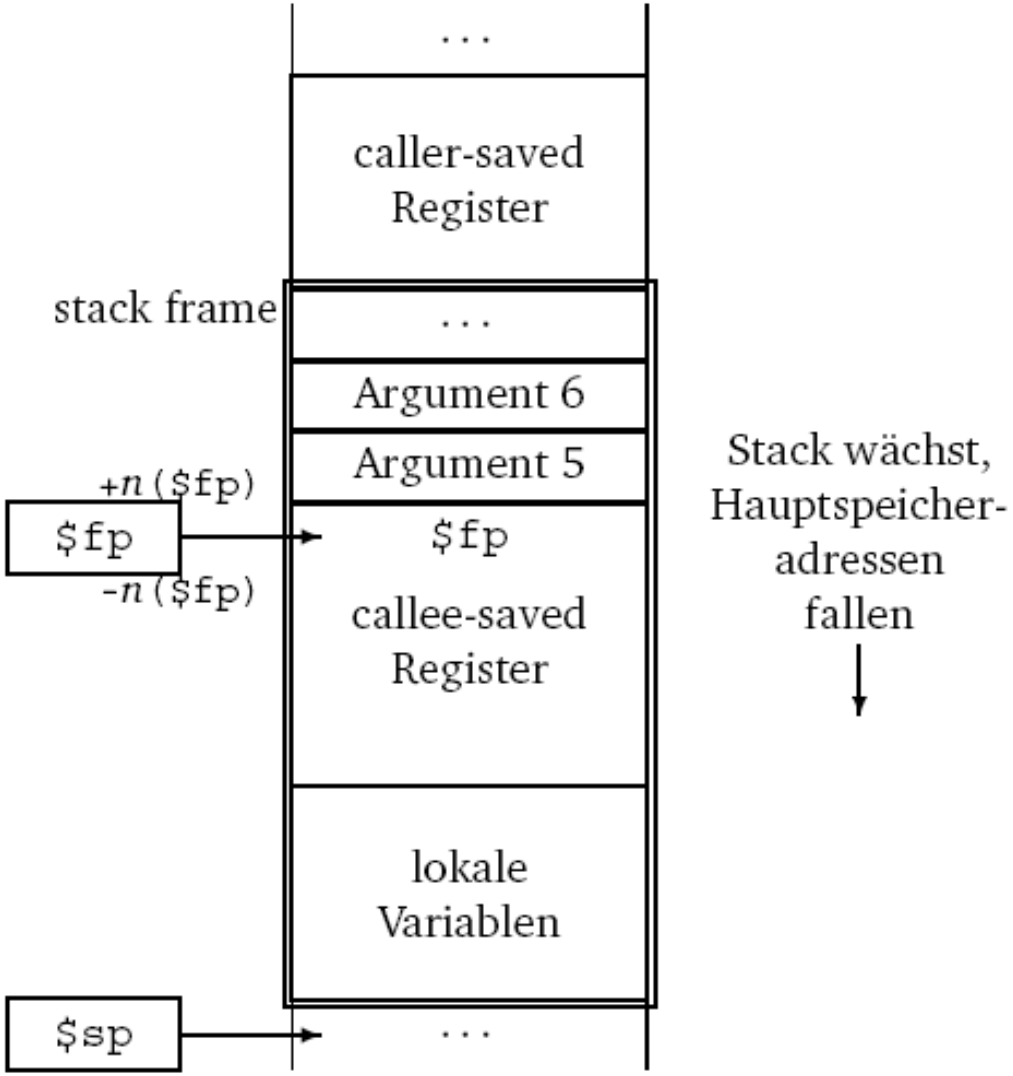
- **Sichere alle *callee-saved* Register** (Register die in der Prozedur verändert werden)
  - Sichere Register $fp, $ra und $s0-$s7 (wenn sie innerhalb der Prozedur verändert werden)
  - **Achtung**: das Register $ra wird durch den Befehl `jal` geändert!

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

Stack-Layout nach Callee-Prolog

Framepointer $fp markiert den Prozedurrahmen.

(Stelle des frühere Stackpointer.)



Stack wächst, Hauptspeicher-adressen fallen

Stackpointer $sp zeigt laut Konvention immer auf das **erste freie Element** des Stacks

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`

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.

Call by Value / Call by Reference

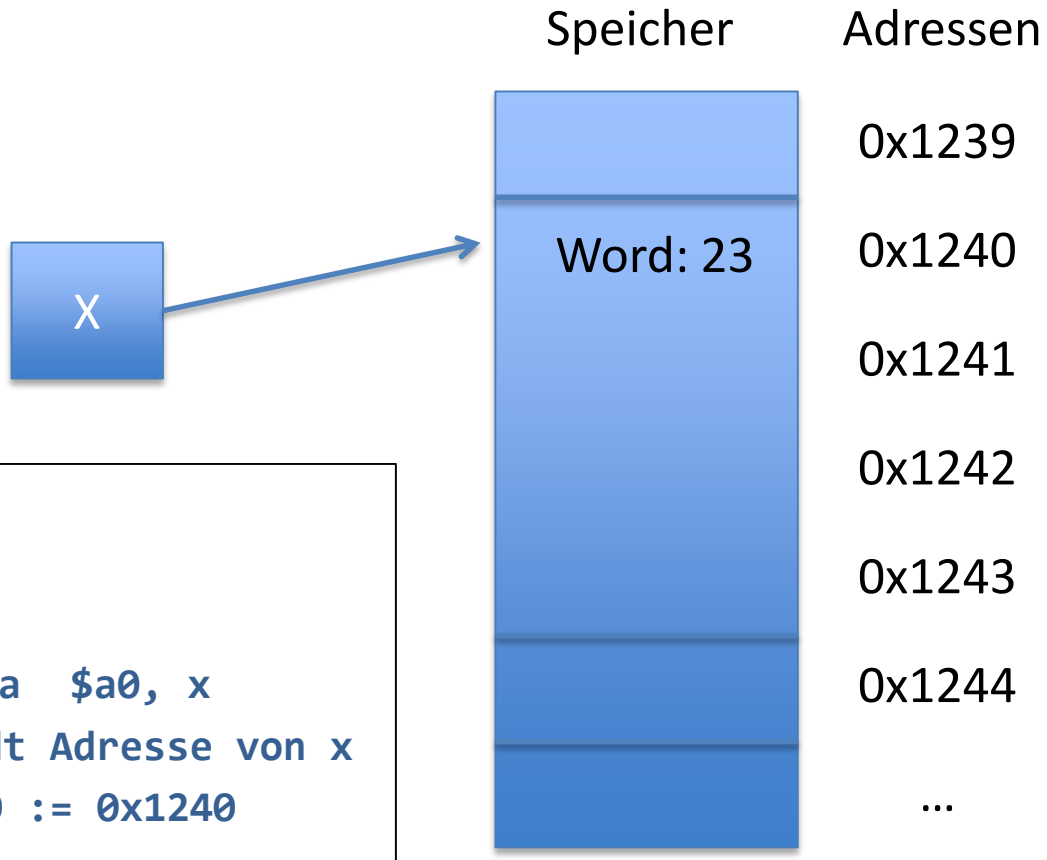
Variablen (Daten) können auf 2 Arten referenziert und an ein Unterprogramm übergeben werden:

- **Call by Value**: Der **Wert** einer Variablen wird referenziert
  - Bsp.: `lw $t0, var`
- **Call by Reference**: Die **Adresse** der Variablen wird referenziert
  - Bsp.: `la $t0, var`

Call by Value / Call by Reference

Beispiel:

```
.data
x:  .word 23
```



CBV

```
.text
main:
    lw  $a0, x
# lädt Wert von x
# $a0 := 23
```

CBR

```
.text
main:
    la  $a0, x
# lädt Adresse von x
# $a0 := 0x1240
```

Call by Value / Call by Reference


```
.data
x:   .word 23

.text
main:
     la     $a0, x       # lädt Adresse von x.
     lw     $a1, x       # lädt Wert von x
```

```
### Call by Value Übergabe ###

jal  cbv

cbv:
     move $t0, $a1
     add  $t0, $t0, $t0
     sw $t0, x
     jr $ra
```

```
### Call by Reference Übergabe ###

jal    cbr

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).

Zusammenfassung

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`,…)
- Unterprogramme (`$a0`, caller-saved, callee,…)
- Call-by-value vs. Call-by-reference

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

- http://www.mobile.ifi.lmu.de/lehrveranstaltungen/rechnerarchitektur-sose16/