



LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

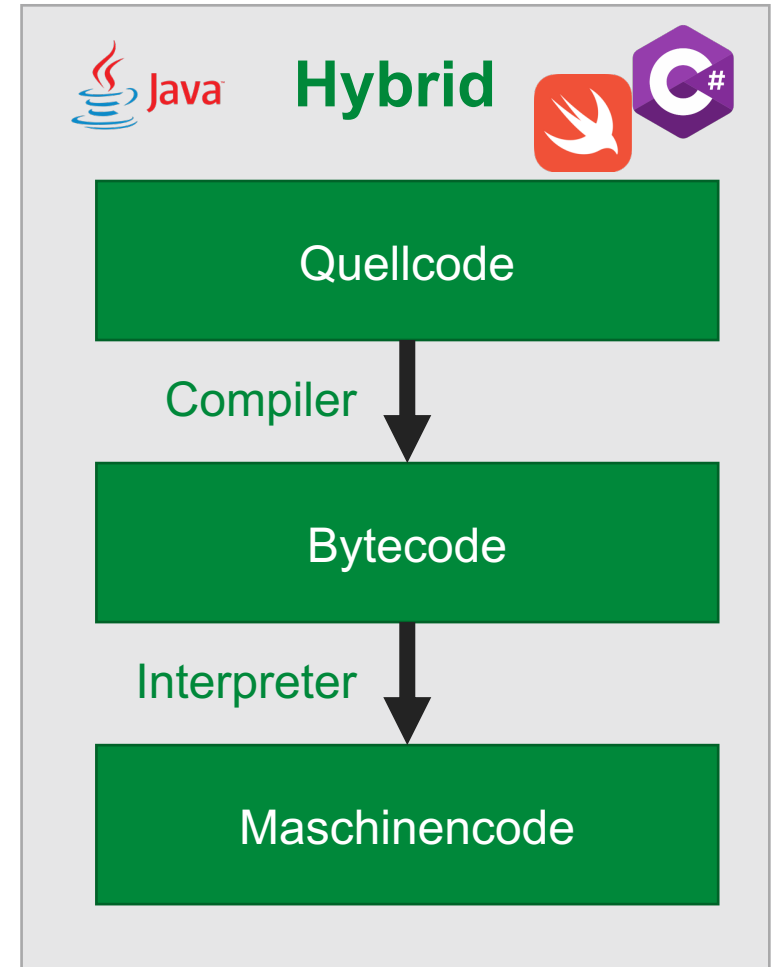
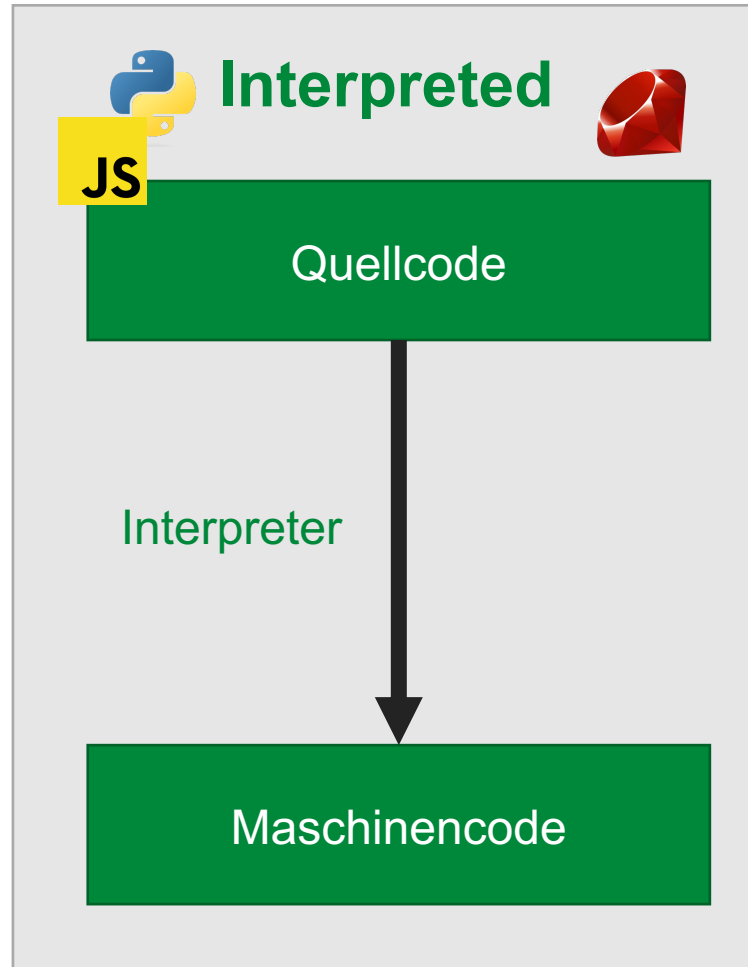
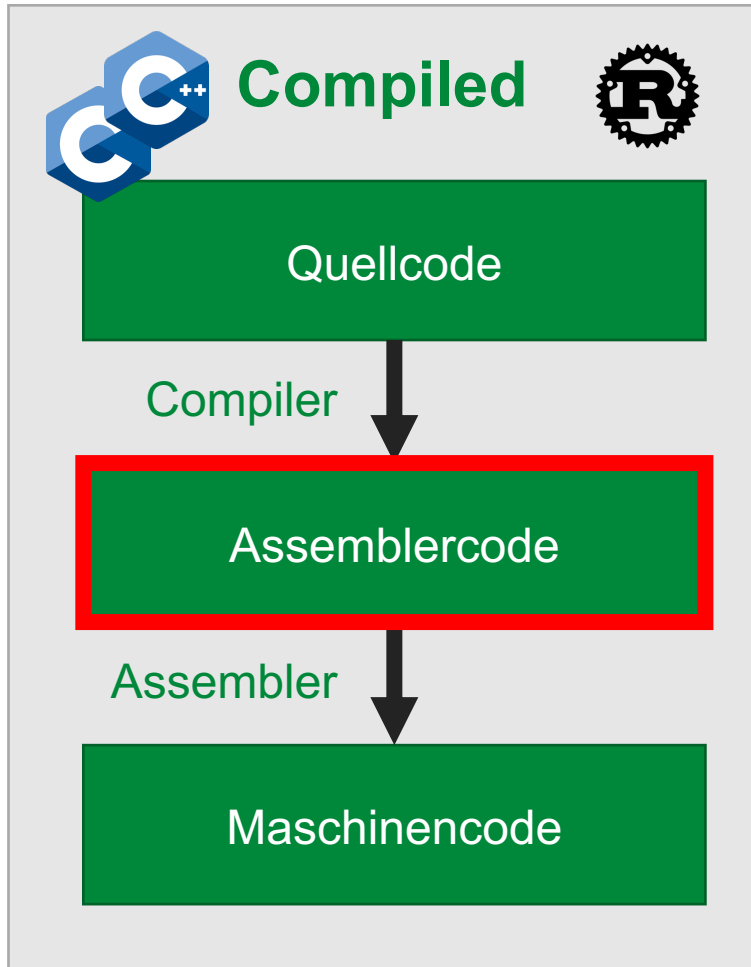
Vorlesung Rechnerarchitektur

Assemblerprogrammierung

Michael Kölle | Lehrstuhl für Mobile und Verteilte Systeme



Arten von Programmiersprachen



Warum sollte ich Assemblerprogrammierung lernen?

Verständnis von Low-Level-Computing:

Die Assembler-Programmierung vermittelt ein tieferes Verständnis dafür, wie Computer und Hardware auf der untersten Ebene funktionieren.

Code-Optimierung:

In einigen Fällen kann das Schreiben von Code in Assemblersprache im Vergleich zu Hochsprachen zu effizienterem und optimiertem Code führen. Dies kann besonders in Umgebungen mit eingeschränkten Ressourcen wichtig sein oder wenn jedes bisschen Leistung zählt.

Fehlersuche und Reverse Engineering:

Assembler-Kenntnisse sind beim Debuggen oder Reverse-Engineering von Low-Level-Software wie Firmware, Betriebssystemen oder eingebetteten Systemen nützlich.

Warum sollte ich Assemblerprogrammierung lernen?



Sicherheitsforschung:

Assembler-Kenntnisse sind entscheidend für die Erforschung von Sicherheitslücken, die Entwicklung von Exploits und die Analyse von Malware.



Echtzeitsysteme:

Echtzeitsysteme erfordern oft eine genaue Kontrolle über die Hardware und das Timing. Die Assemblersprache bietet das notwendige Maß an Kontrolle, um diese Anforderungen zu erfüllen.



Eingebettete Systeme und Mikrocontroller:

Assembler wird häufig bei der Entwicklung von Code für Mikrocontroller und andere eingebettete Systeme verwendet, bei denen Ressourcen wie Speicher und Verarbeitungsleistung begrenzt sind.

RISC (Reduced Instruction Set Computing)

- kleinerer, einfacherer Befehlssatz, der auf eine schnellere Ausführung abzielt.
- Befehle sind einzelne, einfache Operationen, wie Addition oder Laden/Speichern.
- Mehr Allzweckregister, um den Speicherzugriff zu reduzieren und die Leistung zu verbessern.
- Befehle mit fester Länge, leicht dekodierbar -
> vereinfacht Pipelining und parallele Ausführung.

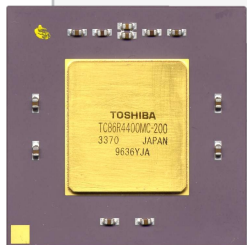
CISC (Complex Instruction Set Computing)

- Größerer, komplexerer Befehlssatz, der leistungsfähigere und vielseitigere Befehle bietet.
- Befehle können mehrere komplexe Operationen ausführen, z.B. die Manipulation von Zeichenketten.
- weniger Register, dafür häufigere Speicherzugriffe.
- Befehle mit variabler Länge -> komplizierte Dekodierung und Pipeline-Verarbeitung.

Wiederholung: Bekannte Vertreter von RISC/CISC

RISC (Reduced Instruction Set Computing)

- MIPS (Microprocessor without Interlocked Pipeline Stages)
- ARM (Advanced RISC Machine, previously Acorn RISC Machine)
- SPARC (Scalable Processor Architecture)
- PowerPC/Power ISA (used in Apple Macintosh, IBM, and others)
- RISC-V (open-source RISC architecture)



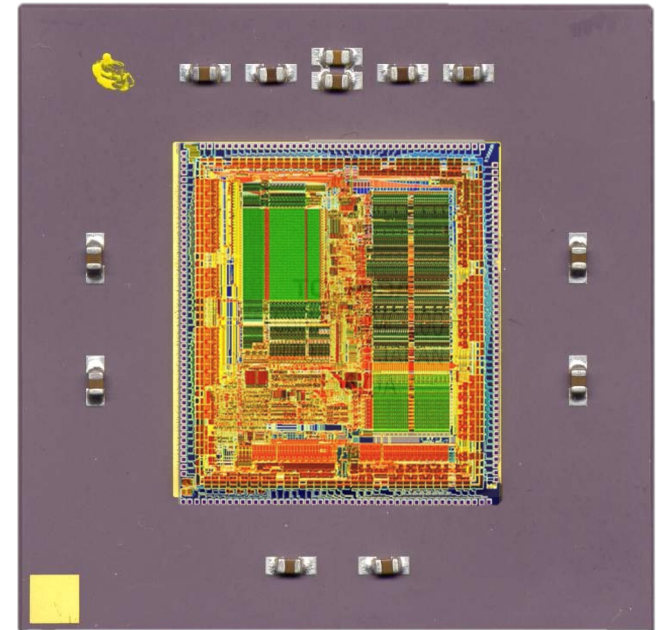
CISC (Complex Instruction Set Computing)

- x86 (Intel, AMD, and others; dominant architecture in personal computers)
- Motorola 68000 series (used in early Apple Macintosh, Sega Genesis, and others)
- IBM System/360 and System/370 (mainframe architectures)
- Zilog Z80 (used in early personal computers and game consoles)



MIPS (Microprocessor without Interlocked Pipeline Stages)

- **RISC-Prozessorarchitektur**
- **Pipeline Verarbeitung:** Befehle werden in Stufen unterteilt, die gleichzeitig verarbeitet werden
- **Load/Store Architektur:** nur Lade- und Speicherbefehle können auf den Speicher zugreifen
- **Registerbasiert:** große Anzahl von Allzweckregistern (normalerweise 32)
- **Verzögerte Verzweigung:** Der Befehl nach einer Verzweigung wird ausgeführt, bevor die Verzweigung ausgeführt wird → Verzweigungslatenz wird verborgen



1981: von John Hennessy entwickelt
(Stanford-Universität)

MIPS Register

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

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

- Byteweise Adressierung
- Ein Speicherwort (Word) entspricht 4 Byte

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				...

Byte-Reihenfolge (Byte Order)

Bytes können in aufsteigender oder absteigender Reihenfolge aneinander gehängt werden.



01001101 01001001 01010000 01010011

Big-Endian (wörtlich „Großes Ende“):

Byte mit den höchstwertigen Bits an der kleinsten Speicheradresse.

Little-Endian (wörtlich „Kleines Ende“):

Byte mit den niederwertigsten Bits an der kleinsten Speicheradresse

Adresse	Binär	00000000	00000001	00000010	00000011
Big Endian	Binär	01001101	01001001	01010000	01010011
	ASCII	M	I	P	S
Little Endian	Binär	01010011	01010000	01001001	01001101
	ASCII	S	P	I	M

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

Byte-Reihenfolge: Unterschiede der Konventionen

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

0101 1000 0101 1000

Big-Endian-Maschine:

- Wert muss im Speicher um zwei Byte verschoben werden.

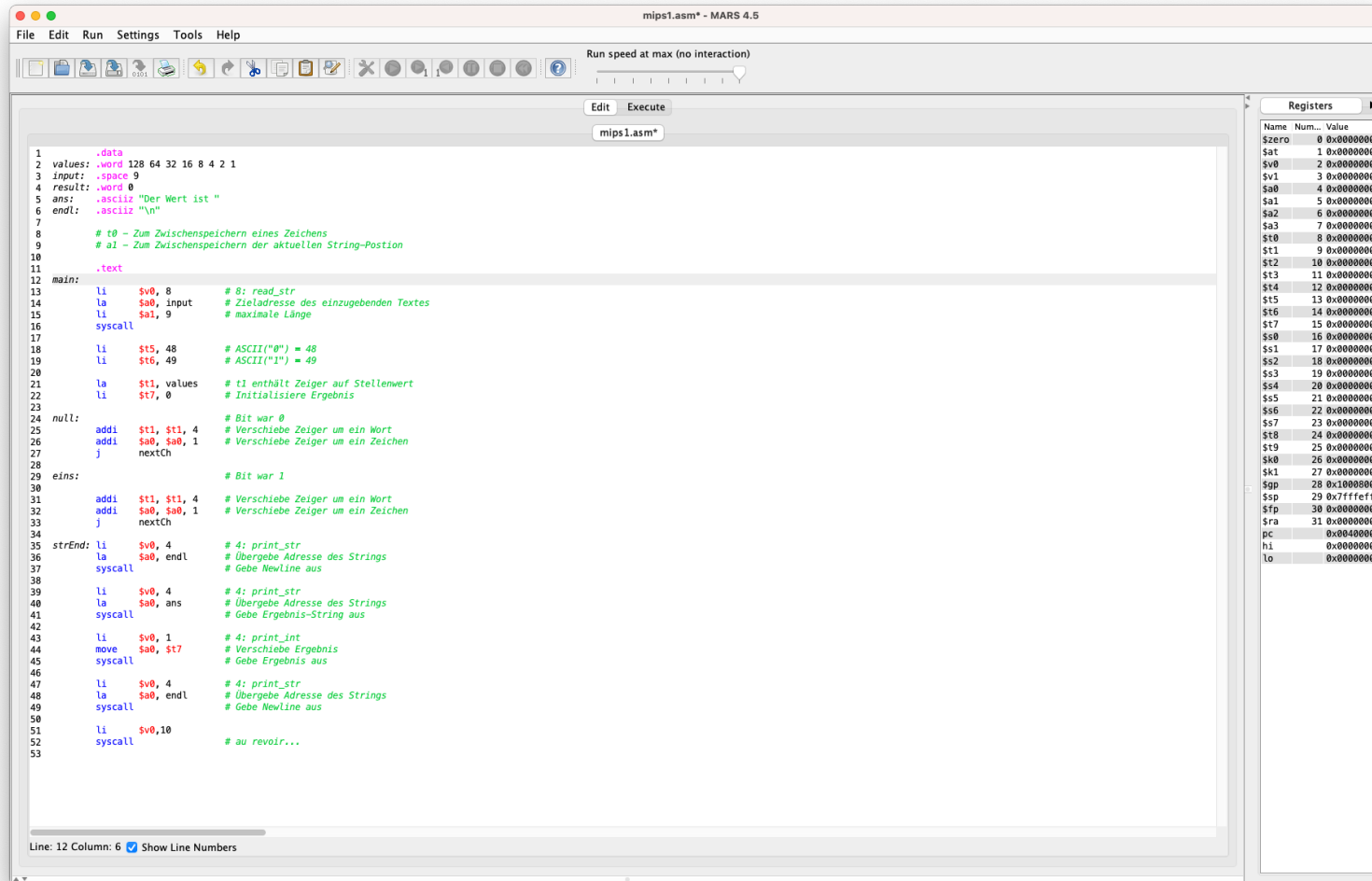

 0000 0000 0000 0000 0101 1000 0101 1000

Little-Endian-Maschine:

- Anfügen von zwei Null Bytes am Ende


 0101 1000 0101 1000 0000 0000 0000 0000

Einführung in die Assemblerprogrammierung Der MIPS Simulator



The screenshot shows the MIPS simulator MARS 4.5. The main window displays assembly code for a program that reads a string and prints it. The code is as follows:

```

1      .data
2  values: .word 128 64 32 16 8 4 2 1
3  input:  .space 9
4  result: .word 0
5  ans:    .asciiz "Der Wert ist "
6  endl:   .asciiz "\n"
7
8      # t0 - Zum Zwischenspeichern eines Zeichens
9      # a1 - Zum Zwischenspeichern der aktuellen String-Position
10
11     .text
12 main:
13     li   $v0, 8      # 8: read_str
14     la   $a0, input  # Zieladresse des einzugebenden Textes
15     li   $a1, 9      # maximale Länge
16     syscall
17
18     li   $t5, 48     # ASCII("0") = 48
19     li   $t6, 49     # ASCII("1") = 49
20
21     la   $t1, values # t1 enthält Zeiger auf Stellenwert
22     li   $t7, 0      # Initialisiere Ergebnis
23
24 null:
25     addi $t1, $t1, 4 # Verschiebe Zeiger um ein Wort
26     addi $a0, $a0, 1 # Verschiebe Zeiger um ein Zeichen
27     j    nextCh
28
29 eins:
30     # Bit war 1
31     addi $t1, $t1, 4 # Verschiebe Zeiger um ein Wort
32     addi $a0, $a0, 1 # Verschiebe Zeiger um ein Zeichen
33     j    nextCh
34
35 strEnd: li   $v0, 4      # 4: print_str
36         la   $a0, endl   # Übergebe Adresse des Strings
37         syscall          # Gebe NewLine aus
38
39         li   $v0, 4      # 4: print_str
40         la   $a0, ans    # Übergebe Adresse des Strings
41         syscall          # Gebe Ergebnis-String aus
42
43         li   $v0, 1      # 4: print_int
44         move $a0, $t7    # Verschiebe Ergebnis
45         syscall          # Gebe Ergebnis aus
46
47         li   $v0, 4      # 4: print_str
48         la   $a0, endl   # Übergebe Adresse des Strings
49         syscall          # Gebe NewLine aus
50
51         li   $v0, 10     # au revoir...
52         syscall
53

```

The right-hand pane shows the Register File:

Name	Num...	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$a8	8	0x00000000
\$t1	9	0x00000000
\$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	0x10000000
\$sp	29	0x7ffefffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

At the bottom of the window, it shows "Line: 12 Column: 6" and a checked "Show Line Numbers" option.

Assemblersprache: SPIM

Die Assemblersprache für den MIPS-Prozessor heißt **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://pages.cs.wisc.edu/~larus/spim.html#qtspim>
- Bsp.: MARS: <http://courses.missouristate.edu/KenVollmar/MARS/>

Empfehlung von mir: MARS

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.

Trennung von Programm und Daten

Grundprinzip (Von-Neumann): Gemeinsamer Speicher für Daten und Programme

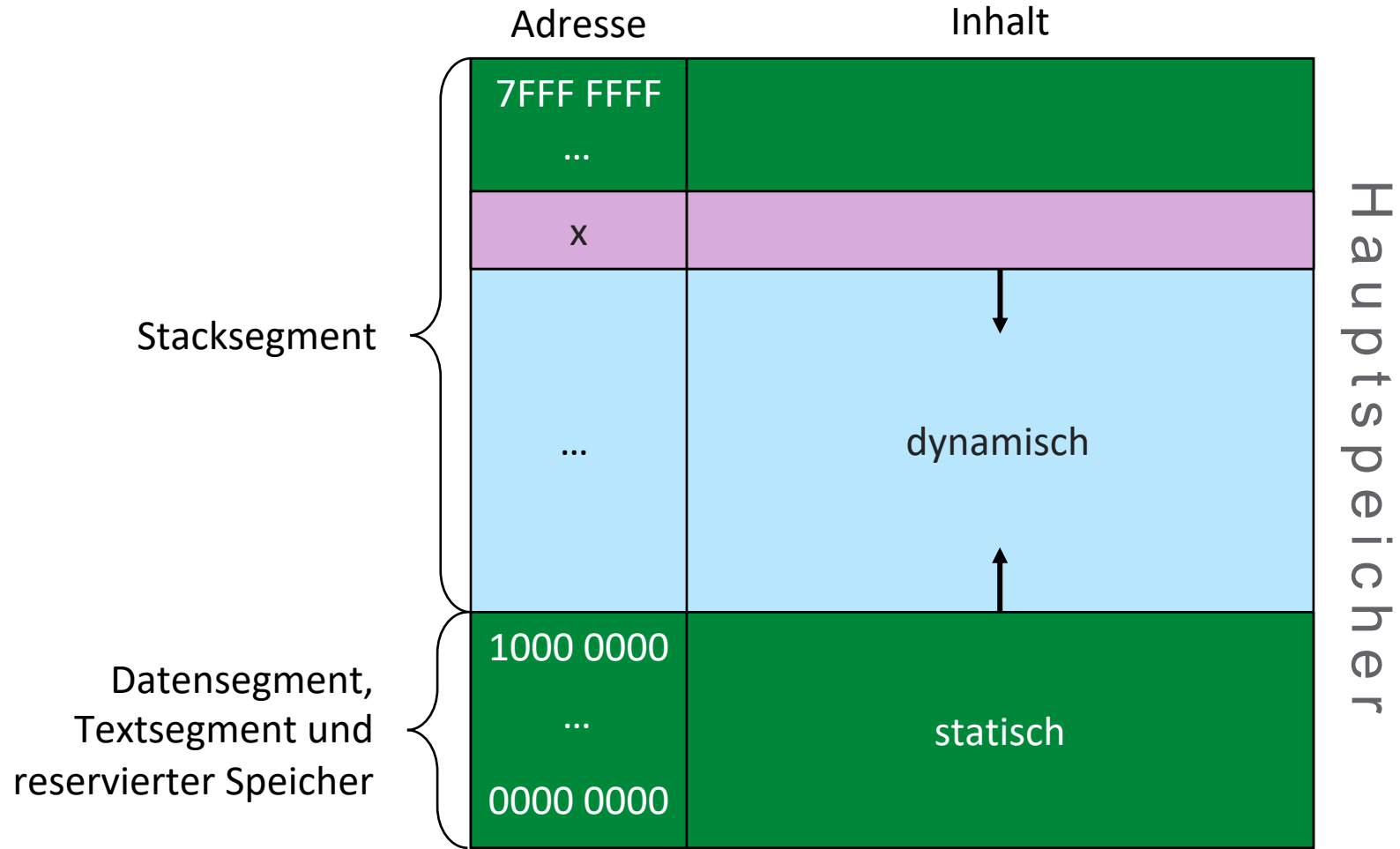
SPIM teilt den Hauptspeicher in **Segmente**, um Konflikte zu vermeiden:

- **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 Segment

Speicherlayout



Dient der Reservierung von und dem Zugriff auf Speicher

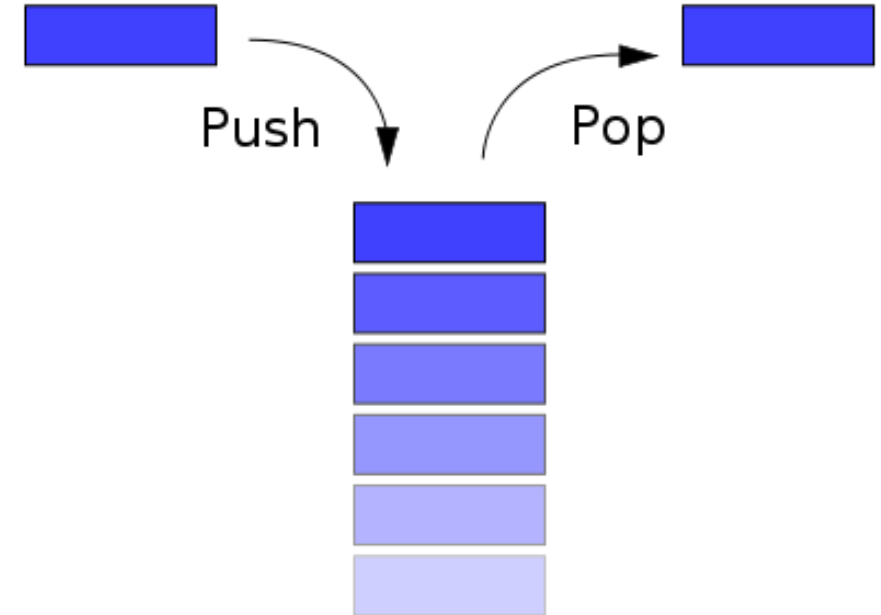
- Feste Startadresse (Meist am Ende des HS und wächst gegen 0)
- Variable Größe (nicht Breite!) BS muss verhindern, dass Stack in das Daten-Segment wächst
- Arbeitet nach dem LIFO (Last In–First Out)-Prinzip

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.



Beispiel: Assemblerprogramm

```

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

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

```

Datensegment

Textsegment

Kommentarzeichen

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

Erklärung zum Beispiel

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` beenden das Programm an.

SPIM hat drei verschiedene Integertypen

Folgende Direktiven dienen zur Reservierung für den notwendigen Speicher

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

Integer Daten anlegen

Mit der Direktive

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

werden Folgen von 32-Bit Integeren 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 (Big End)	0xA8	0xA9	0xAA	0xAB	0xAC	0xAD	0xAE	0xAF	0xB0	0xB1	0xB2
<code>.ascii</code>	H	a	l	l	o		W	e	l	t	
<code>.asciiz</code>	H	a	l	l	o		W	e	l	t	\0

Innerhalb eines Strings sind folgende Kombinationen erlaubt:

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

Beispiel:

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

könnte ausgedruckt so aussehen:

ab

cd ef"gh"

(\ ist das sog. "escape Zeichen")

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, 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.

Adresse (.asciiz)	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B
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

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 *in 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

Load Word

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

- **lw** lädt die Daten aus der angegebenen Adresse **Adr** in das Zielregister **Rd**.
- **Adr** kann auf verschiedene Weise angegeben werden:
 - **(Rs)** : Der Wert steht im Hauptspeicher an der Adresse, die im Register **Rs** steht (Register-indirekt)
 - **label** oder **label+Konstante**: Der Wert steht im Hauptspeicher an der Stelle, die für **label** reserviert wurde, bzw. nachdem Konstante dazu addiert wurde (direkt).
 - **label (Rs)** oder **label+Konstante (Rs)** : Der Wert steht im Hauptspeicher an der Stelle, die für **label** reserviert wurde + **Konstante** + **Inhalt** von Register **Rs** (indexiert).

Ladebefehl und Adressierung

Load Address

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

- **la** lädt die **Adresse** auf die das Label **label** zeigt in das Zielregister Rd.
- Zum Vergleich: **lw** lädt die **Daten** aus der angegebenen Adresse **Adr** in das Zielregister Rd.

```

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

    .text
main: lw $t1, var           # $t1 enthält "20" (direkte Adr.)

    lw $t1, var+4          # $t1 enthält "4" (direkte Adr.)

    lw $t2, var($t1)       # $t2 enthält "4" (indexierte Adr.)

    lw $t2, var+8($t1)     # $t2 enthält "25" (indexierte Adr.)

    la $t1, var            # Adr. von "20" in $t1
    lw $t2, ($t1)          # $t2 enthält "20" (indirekte Adr.)

```

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

Reigsterinhalt zurück in den Hauptspeicher speichern

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

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.
- **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).

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)

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)

Betriebssystem Aufruf: SYSCALL

- Jeder Prozessor arbeitet nur vernünftig in einem Betriebssystem (Windows, macOS, Linux, 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).

Betriebssystem Funktionen von 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

Betriebssystem Funktionen von SPIM

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 SYSCALL

```

.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 SYSCALL

```
                                # 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

```

Ausgabe:

```

1 Dieser
2 Text
3
4 wird
5
6
7 ausgegeben
8 Und dieser auch

```

Anmerkungen zu read_str

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

```

.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                # 8: read_str
      la    $a0, input            # Adresse des einzugebenden Textes
      li    $a1, 4                # maximale Länge
      syscall
      li    $v0, 4                # 4: print_str
      la    $a0, input
      syscall
      li    $v0, 10               # Exit
      syscall

```

Ausgabe:

```

1 Text=a
2 a

```

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.

- **jump**-Befehle können weiter springen, da ihre Instruction keine Bits für die Bedingung reservieren muss

Beispiel für IF-Bedingungen

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
```

Pseudocode:

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

Beispiel für Schleifen

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:

```

Pseudocode:

```

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

```


Beispiel: Schleifen und Arrays

.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:

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 aus *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 ermittelten 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 Assemblerzeit 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

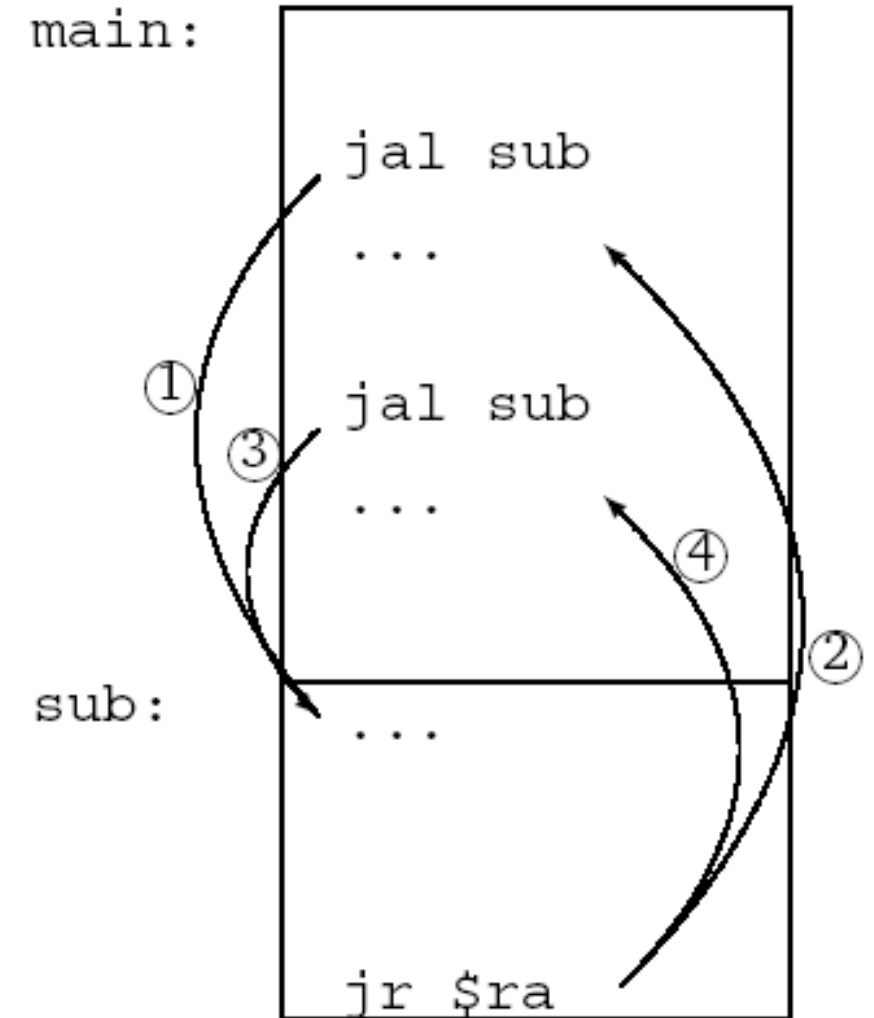
```

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: ausgeben
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.

Stackpointer und Framepointer

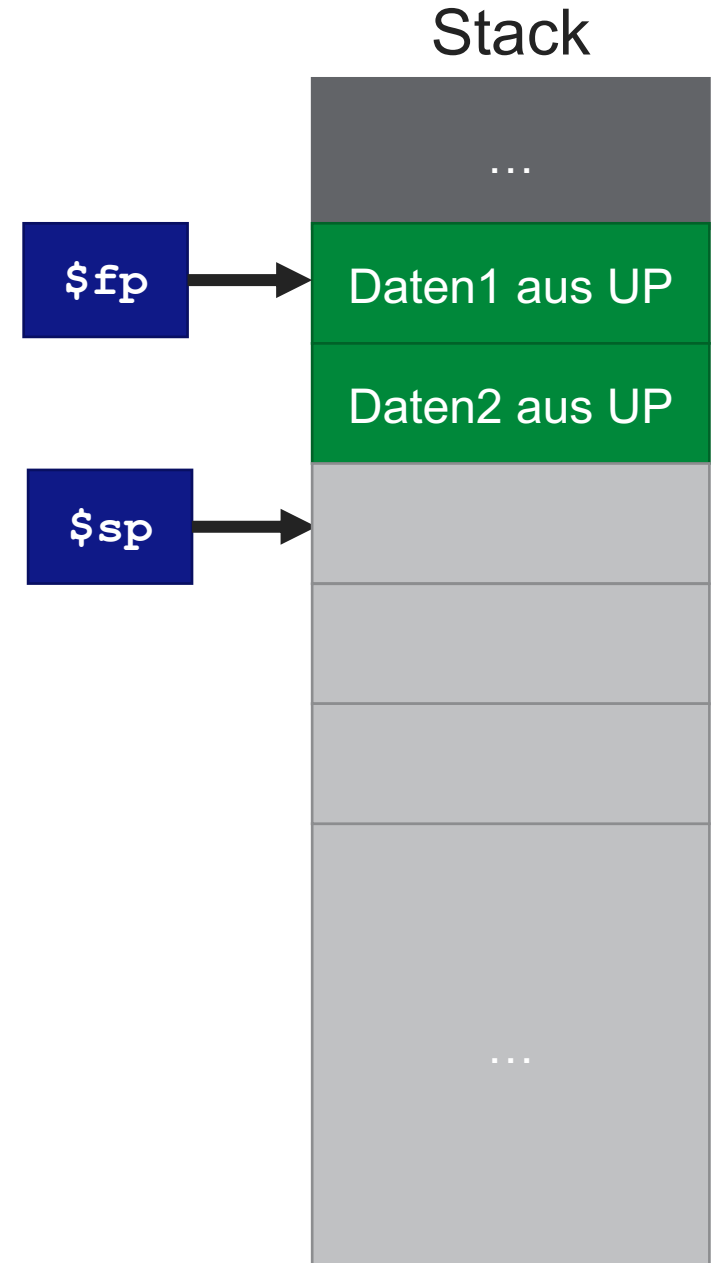
- Stackpointer `$sp` zeigt immer auf die erste freie Speicherzelle auf dem Stack
- Framepointer `$fp` gibt den Kontext an (z.B. Unterprogramm)

Speichern:

```
sub $sp, $sp, 4 # Reserviere 4 Byte
sw $t1, 4($sp) # Speichere $t1
```

Laden:

```
lw $t1, 4($sp) # Lade in $t1
add $sp, $sp, 4 # Gebe Speicher frei
```



Prozedur-Konvention beim SPIM

Prolog des Callers (aufrufendes Programm):

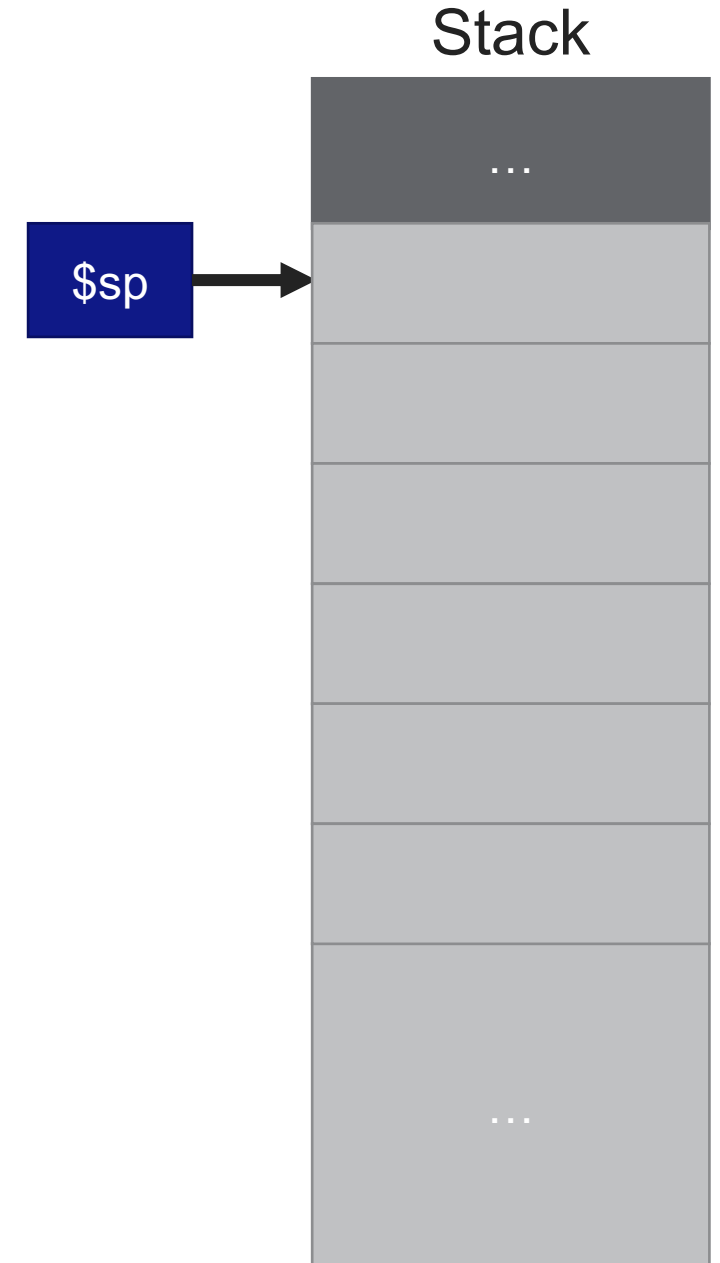
Sichere alle *caller-saved* Register:

- 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 Callers (aufrufendes Programm):

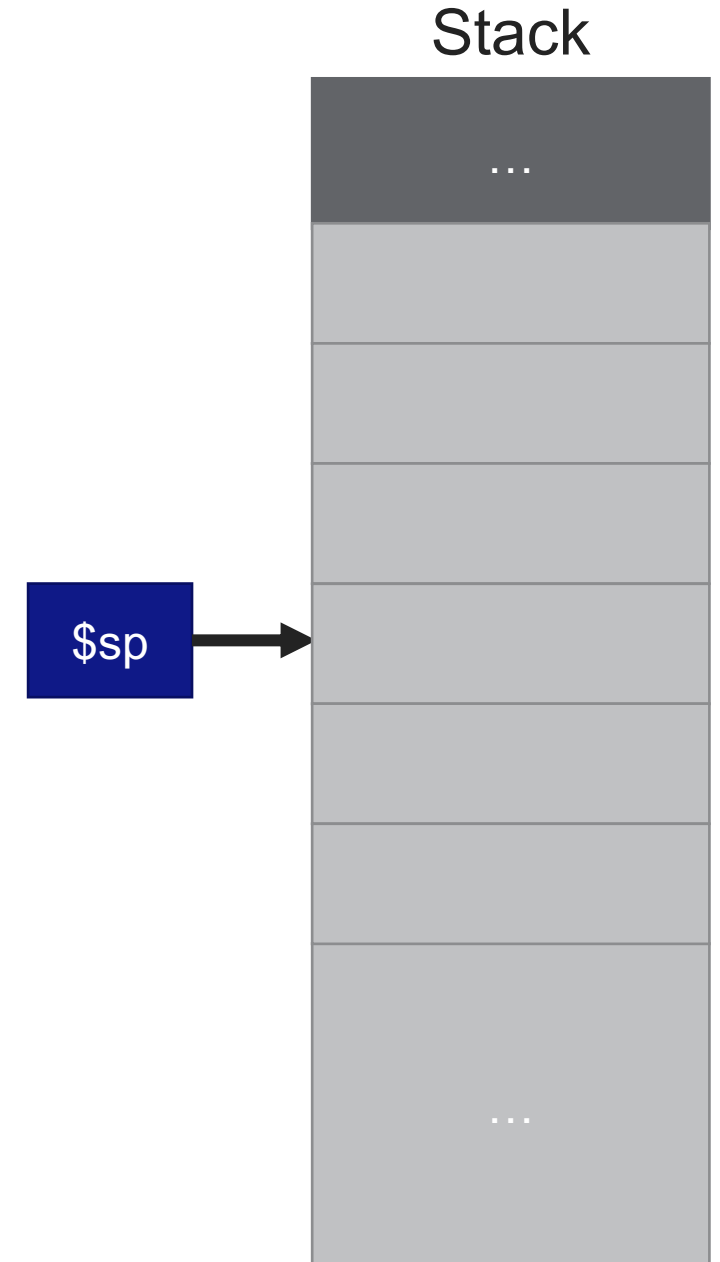
Sichere alle *caller-saved* Register:

- 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 Callers (aufrufendes Programm):

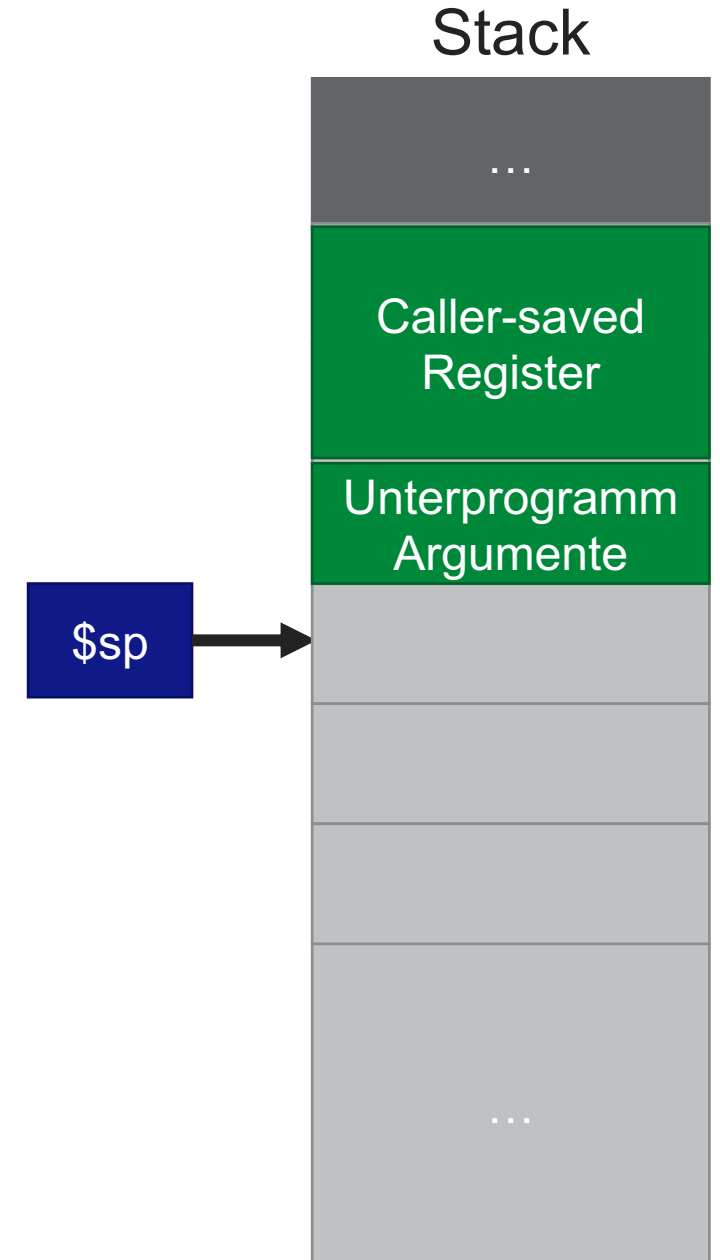
Sichere alle *caller-saved* Register:

- 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)

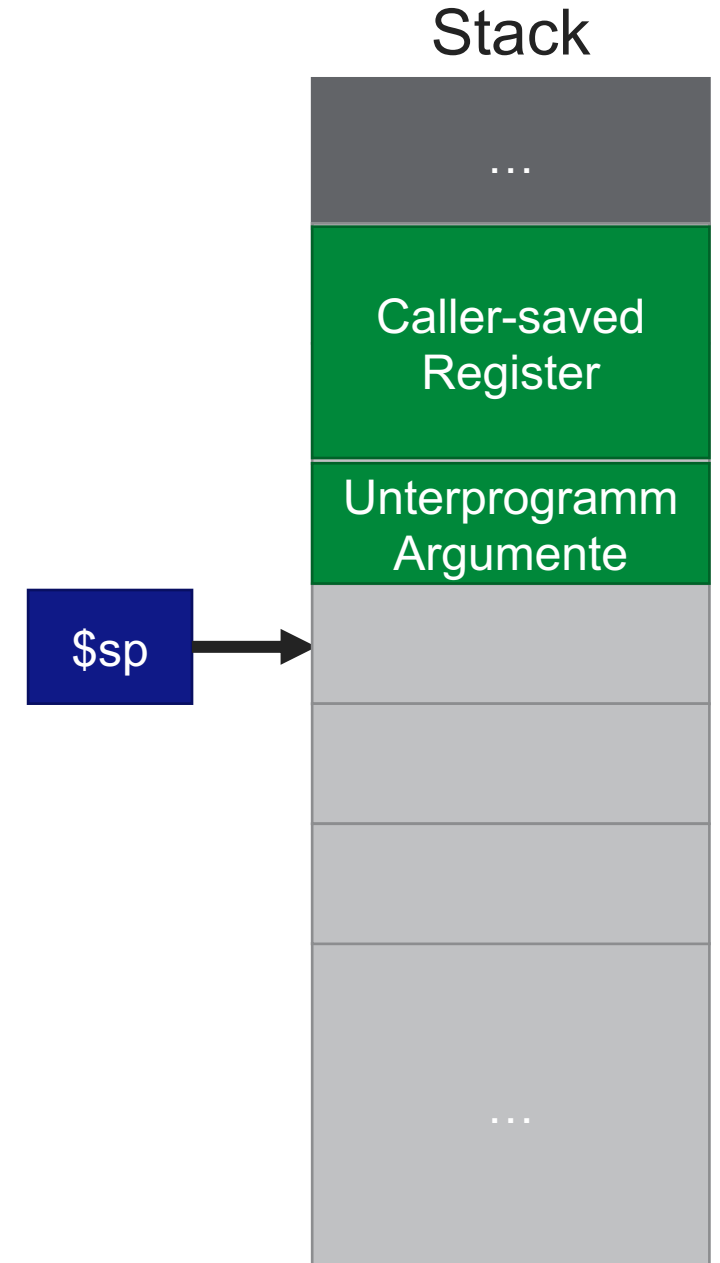
Starte die Prozedur (jal)



Prozedur-Konvention beim SPIM

Prolog des Callee:

- **Sichere alle *callee-saved* Register** (Register die in der Prozedur verändert werden)
 - Sichere Register \$fp, \$ra und \$s0-\$s7 (falls sie verändert werden, 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.

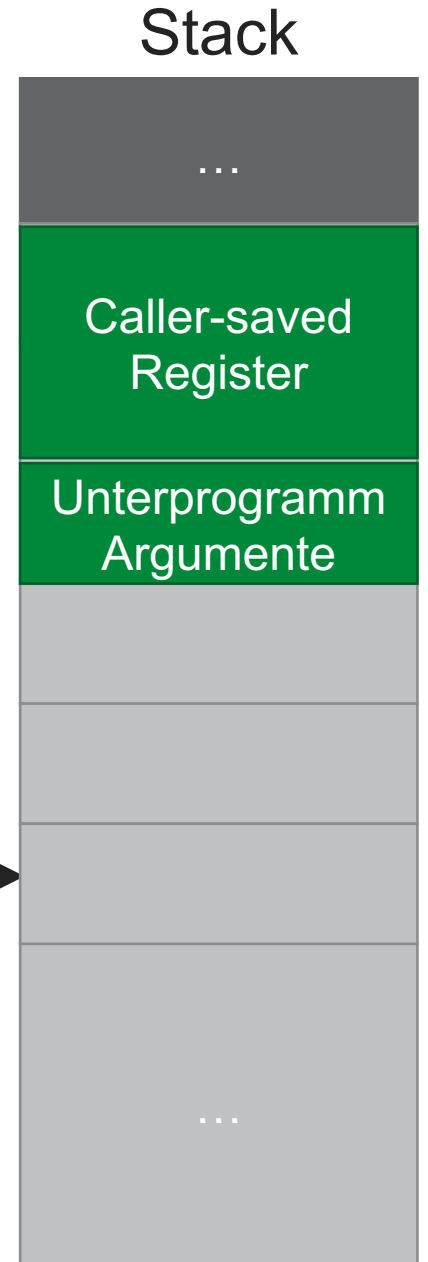


Prozedur-Konvention beim SPIM

Prolog des Callee:

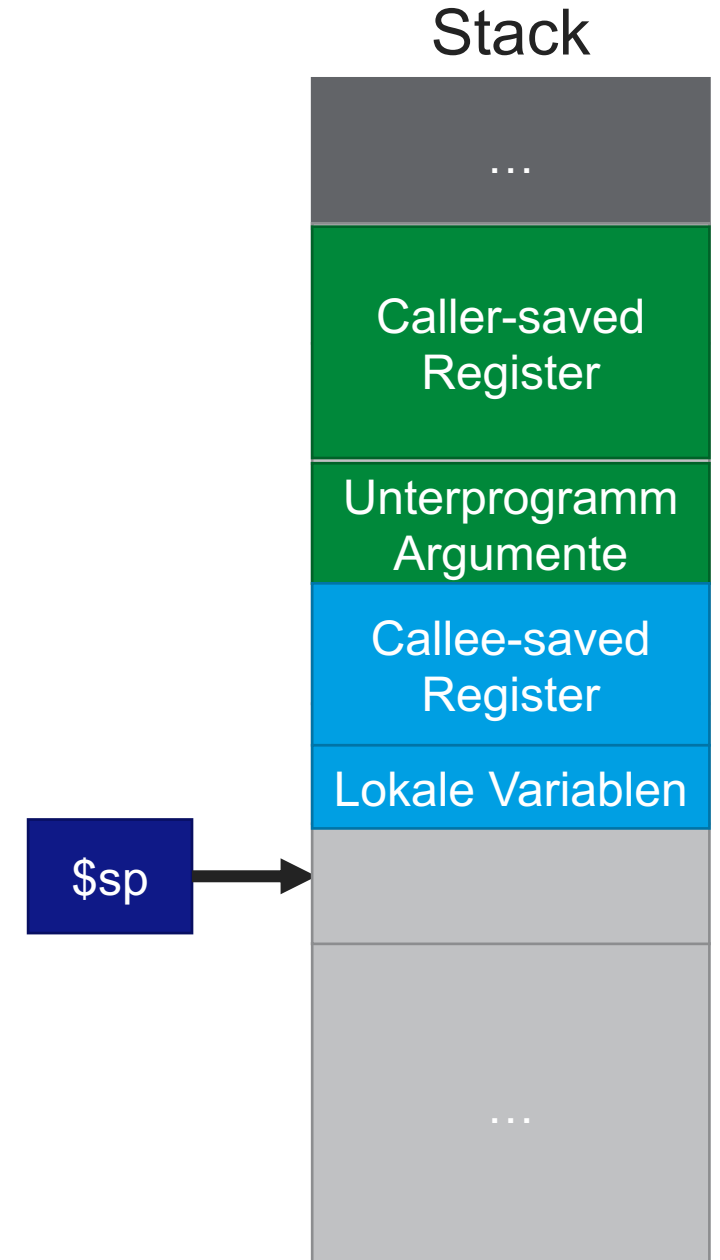
- **Sichere alle *callee-saved* Register** (Register die in der Prozedur verändert werden)
 - Sichere Register \$fp, \$ra und \$s0-\$s7 (falls sie verändert werden, Register \$ra wird durch den Befehl jal geändert)
- **Erstelle den Framepointer:**
 - Addiere die Größe des Stackframe zum Stackpointer und lege das Ergebnis in \$fp ab.

\$sp



Prolog des Callee:

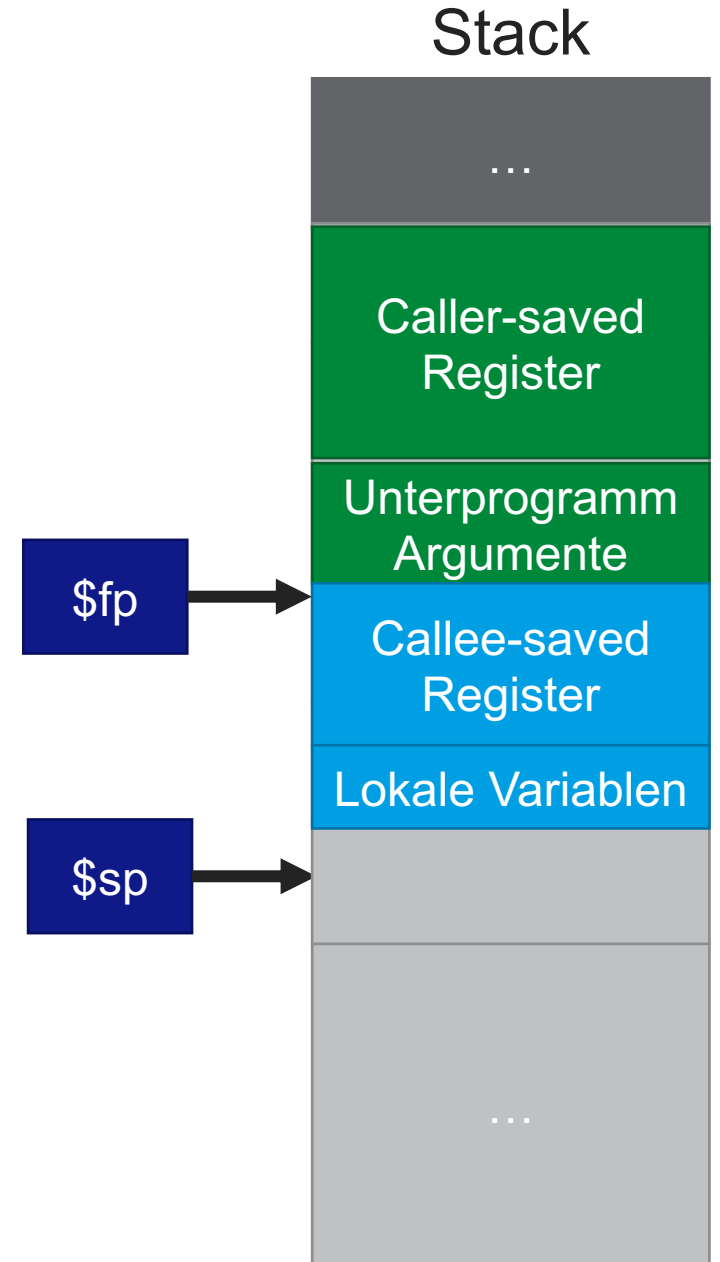
- **Sichere alle *callee-saved* Register** (Register die in der Prozedur verändert werden)
 - Sichere Register \$fp, \$ra und \$s0-\$s7 (falls sie verändert werden, Register \$ra wird durch den Befehl jal geändert)
- **Erstelle den Framepointer:**
 - Addiere die Größe des Stackframe zum Stackpointer und lege das Ergebnis in \$fp ab.



Prozedur-Konvention beim SPIM

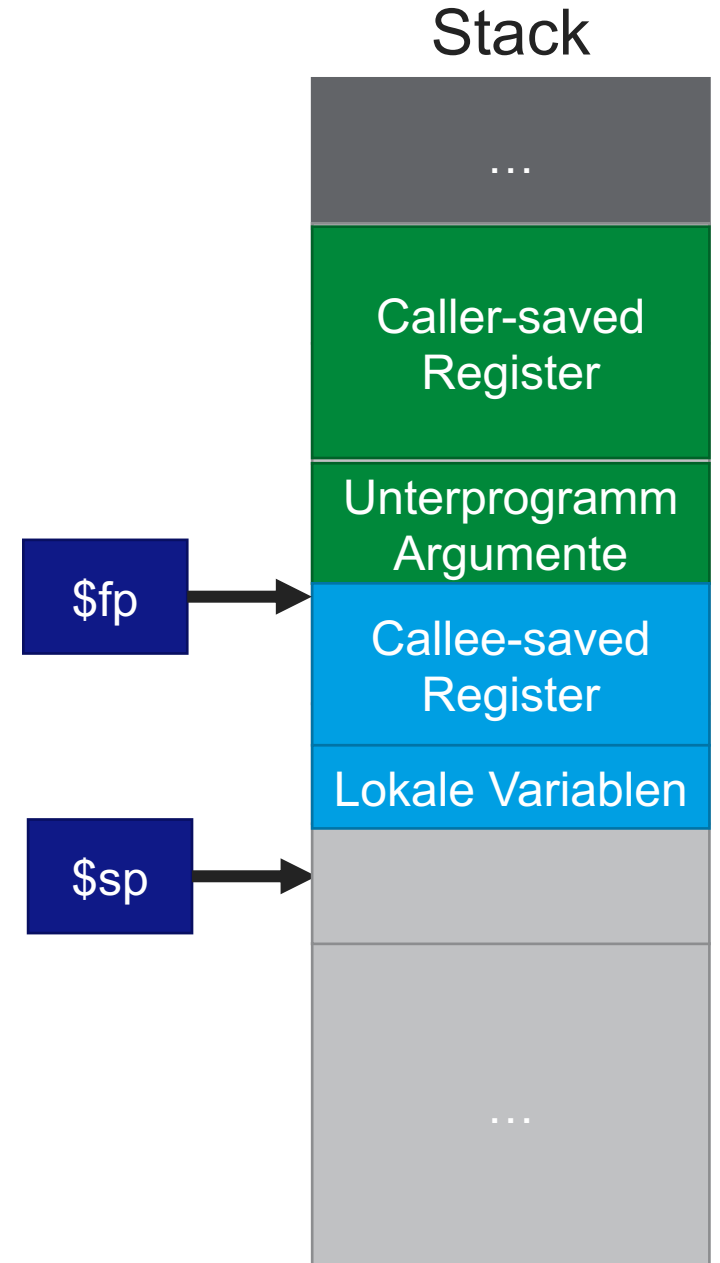
Prolog des Callee:

- **Sichere alle *callee-saved* Register** (Register die in der Prozedur verändert werden)
 - Sichere Register \$fp, \$ra und \$s0-\$s7 (falls sie verändert werden, 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.



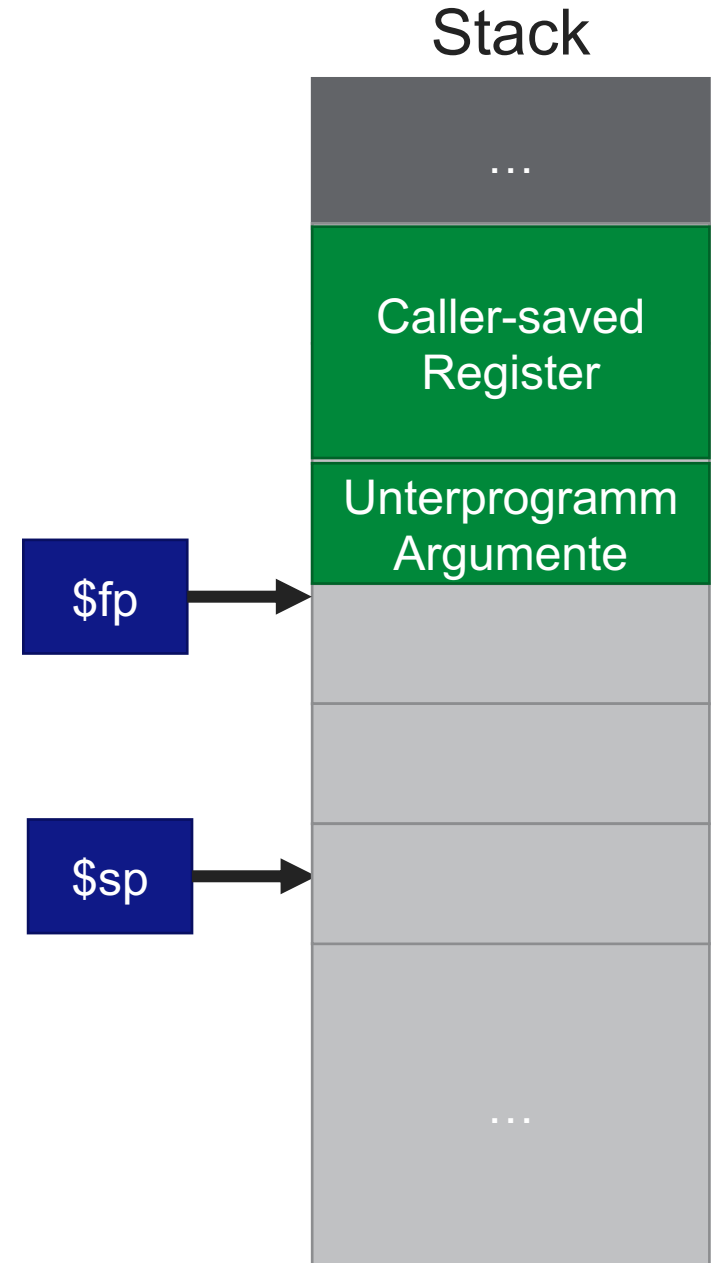
Epilog des Callees:

- **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!
- **Springe zum Caller zurück:**
 - jr \$ra



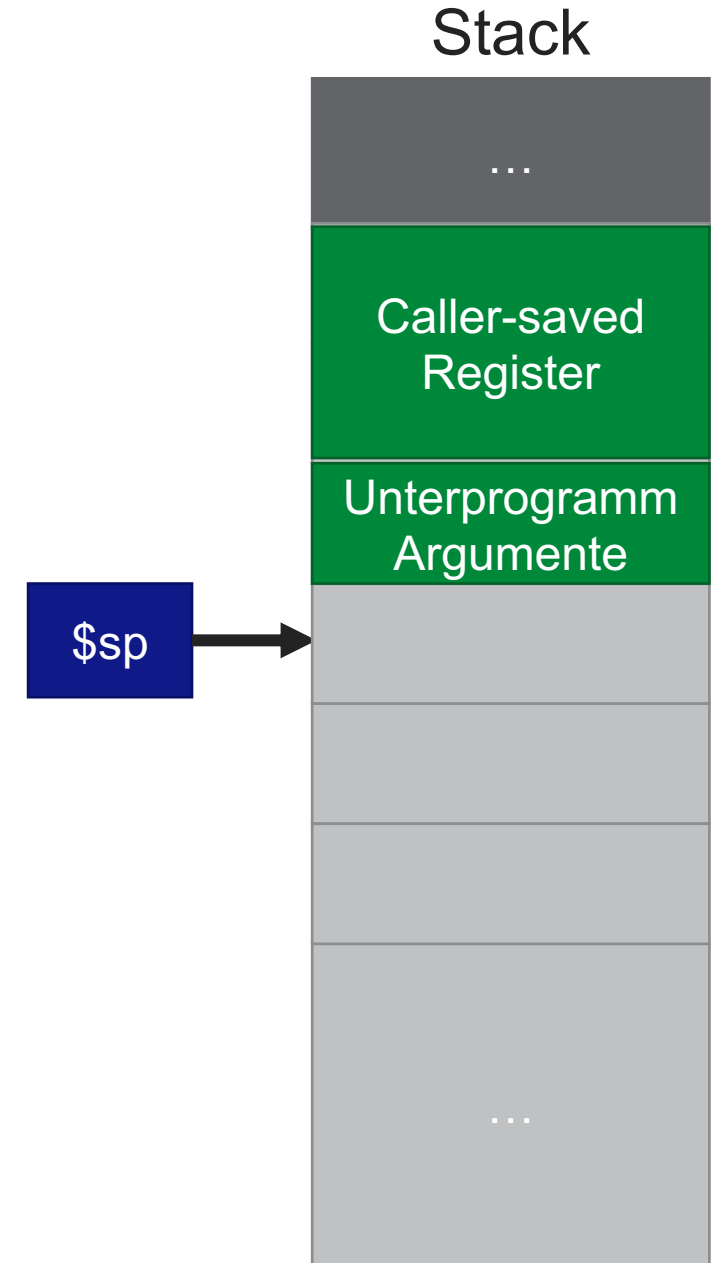
Epilog des Callees:

- **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!
- **Springe zum Caller zurück:**
 - jr \$ra



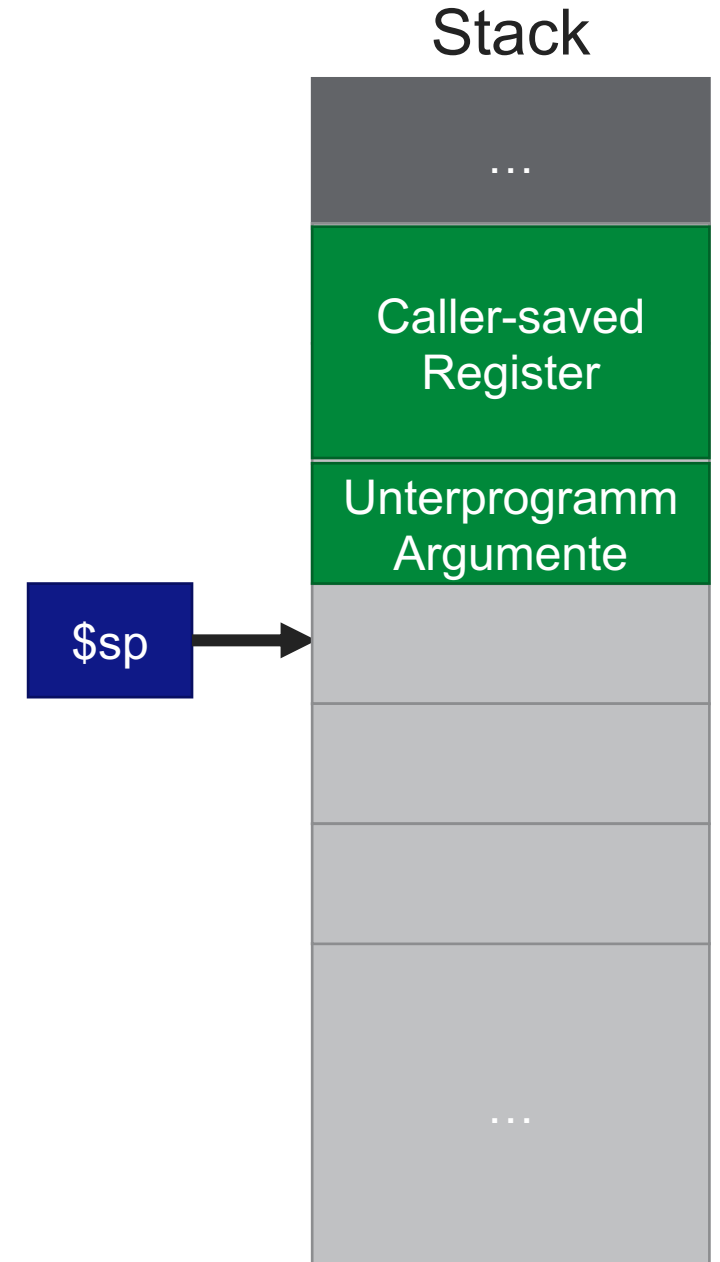
Epilog des Callees:

- **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!
- **Springe zum Caller zurück:**
 - jr \$ra



Epilog des Callers:

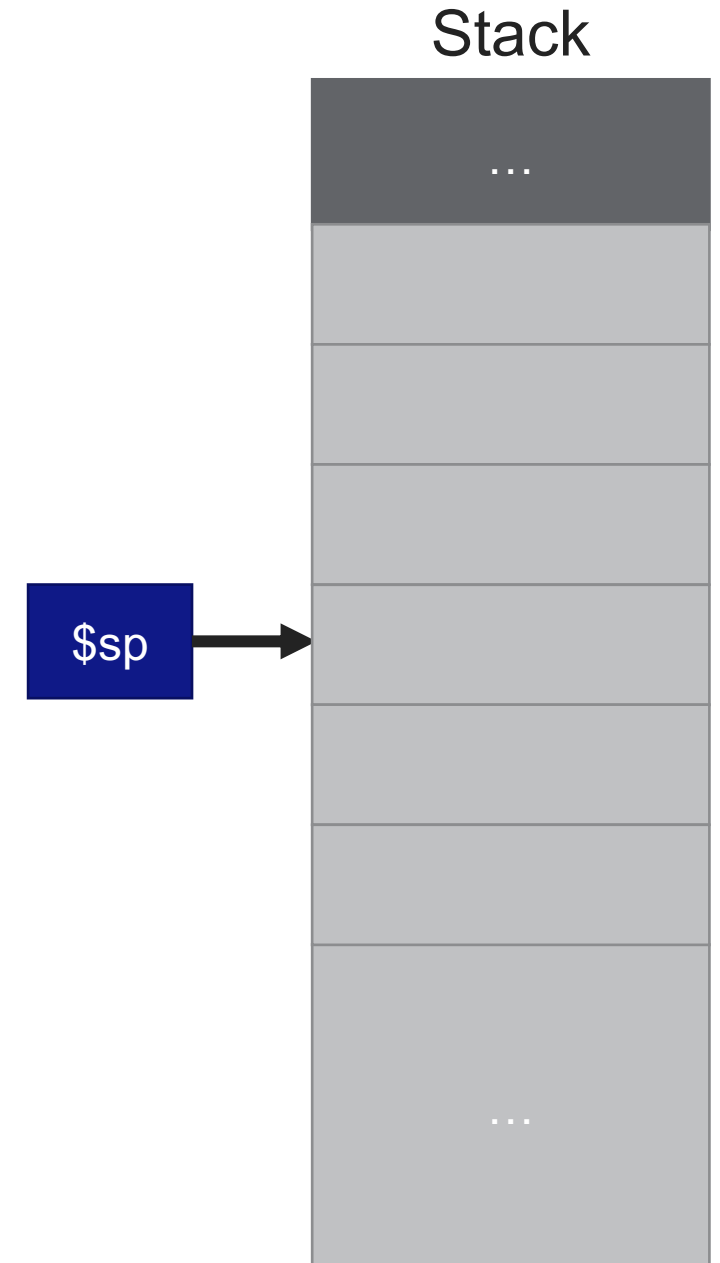
- **Stelle gesicherte Register wieder her:**
 - Vom Caller gesicherte Register wieder herstellen
 - Achtung: Evtl. über den Stack übergebene Argumente bei der Berechnung des Abstandes zum Stackpointer beachten!
- **Stelle ursprünglichen Stackpointer wieder her:**
 - Multipliziere die Zahl der Argumente und gesicherten Register mit vier und addiere sie zum Stackpointer.



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!

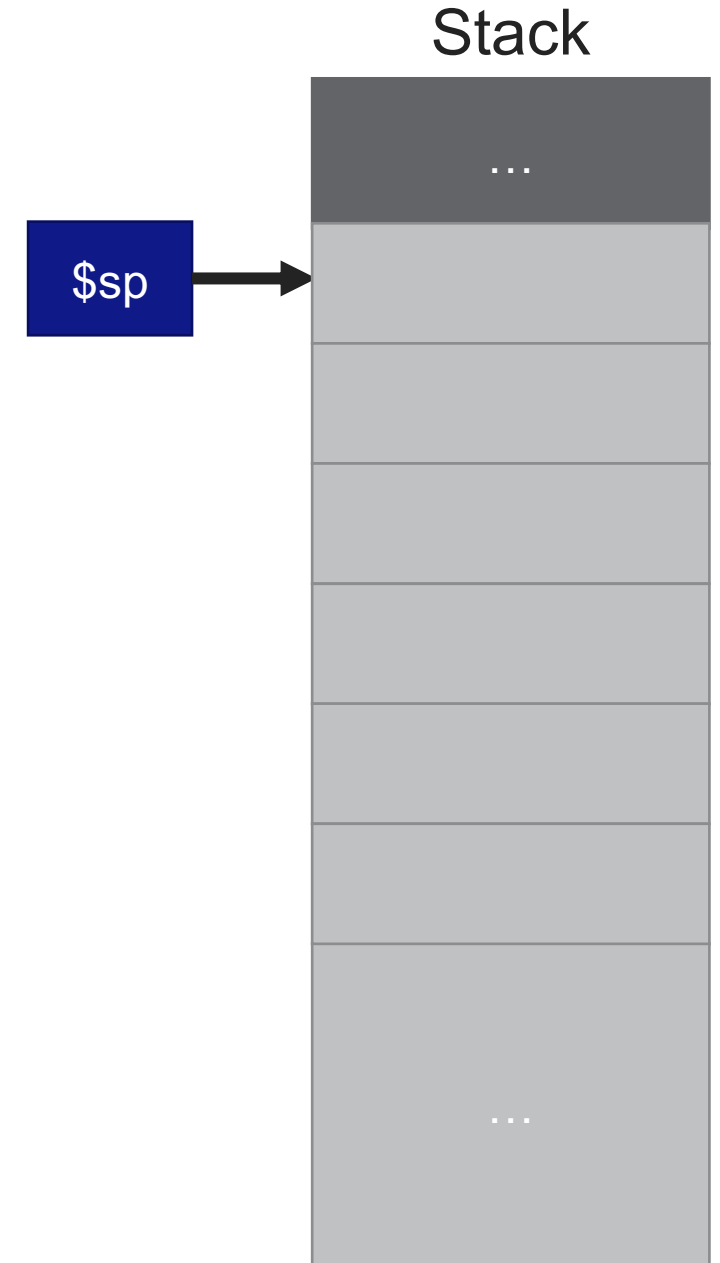
- **Stelle ursprünglichen Stackpointer wieder her:**
 - Multipliziere die Zahl der Argumente und gesicherten Register mit vier und addiere sie zum Stackpointer.



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

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 wird 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 Betriebssystems, daher `ktext`-Direktive an der Stelle 0x8000 0080 verwenden:
`.ktext 0x8000 0080`

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`)