LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Grundlagen zur Assemblerprogrammierung unter SPIM

im Sommersemester 2016

Lorenz Schauer
Mobile & Verteilte Systeme

12. Juli 2016

Agenda heute

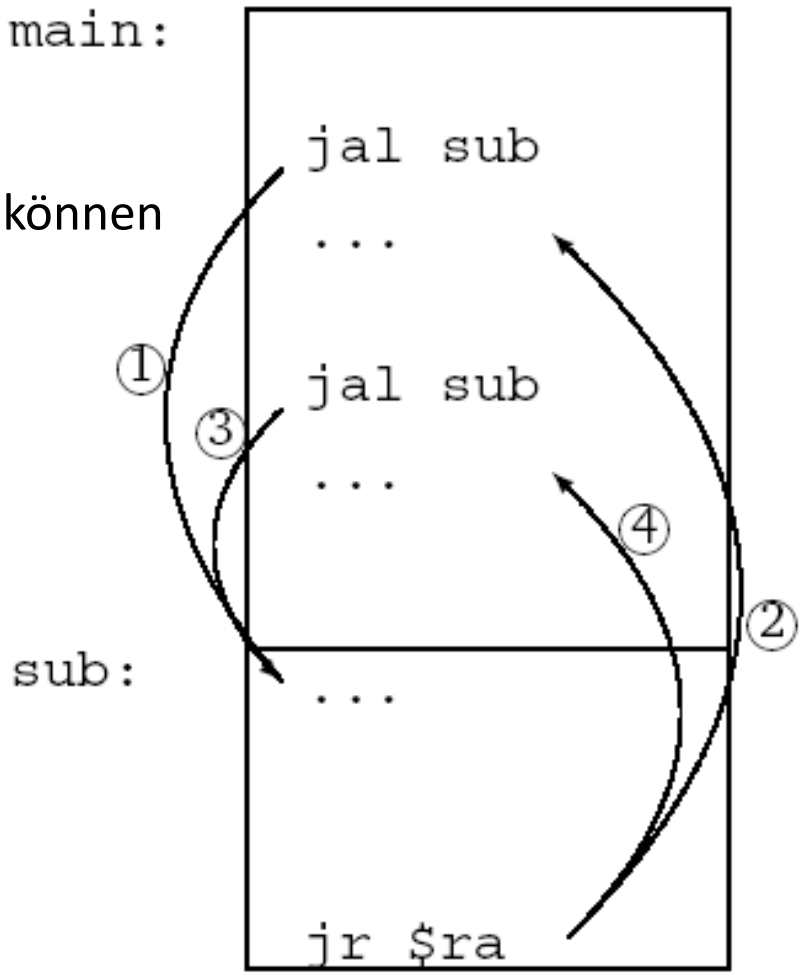
Grundlagen:

- Unterprogramme I
- Call-by-Value (CBV) vs. Call-by-Reference (CBR)
- Kontroll-Strukturen
- Stack-Speicher
- Unterprozeduren II

Unterprogramme I

Unterprogramme:

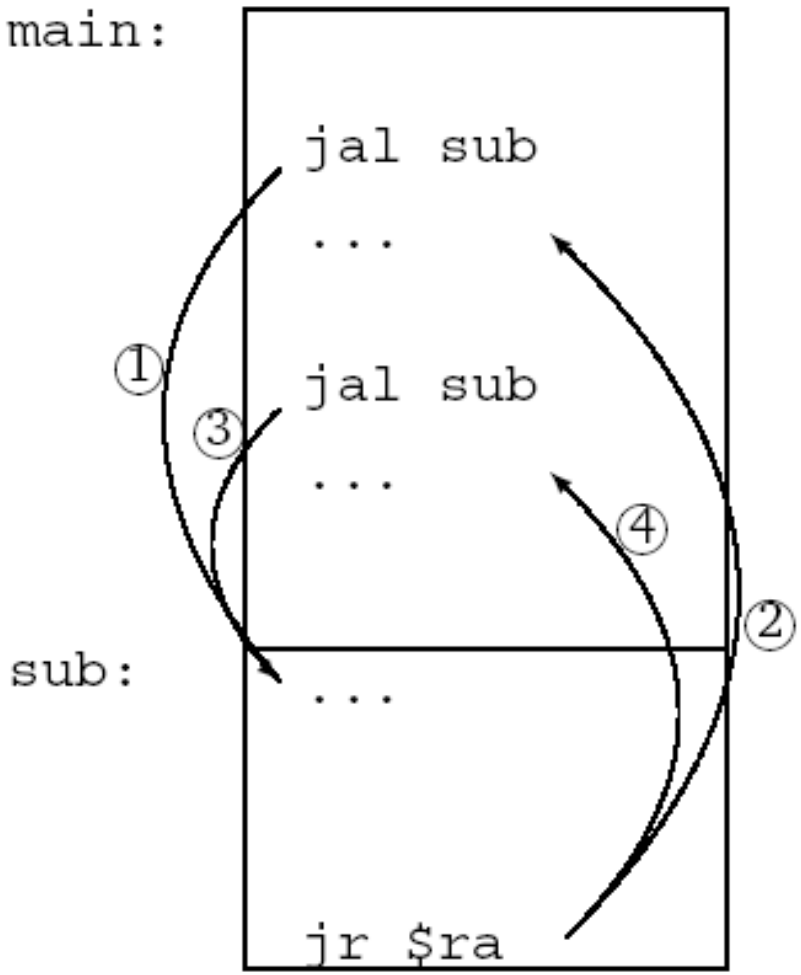
- In Hochsprachen Prozeduren, Methoden
- Programmstücke, die von unterschiedlichen Stellen im Programm angesprungen werden können
- Dienen der Auslagerung wiederkehrender Berechnungen
- Nach deren Ausführung: Rücksprung zum Aufrufer



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

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



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

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.

Aufgabe 1

Schreiben Sie ein SPIM-Programm, welches…

- … 2 Zahlen als Nutzereingaben einliest
- … die beiden Eingaben nach MIPS-Konvention dem Unterprogramm `proc` übergibt (Argumente in $a0…$a3).
- … das Unterprogramm `proc` das Produkt der beiden Zahlen berechnet.
- … das Ergebnis gemäß MIPS-Konvention im Register $v0 zurückgibt.
- … letztendlich das Ergebnis auf der Konsole ausgibt.

Call by Value / Call by Reference

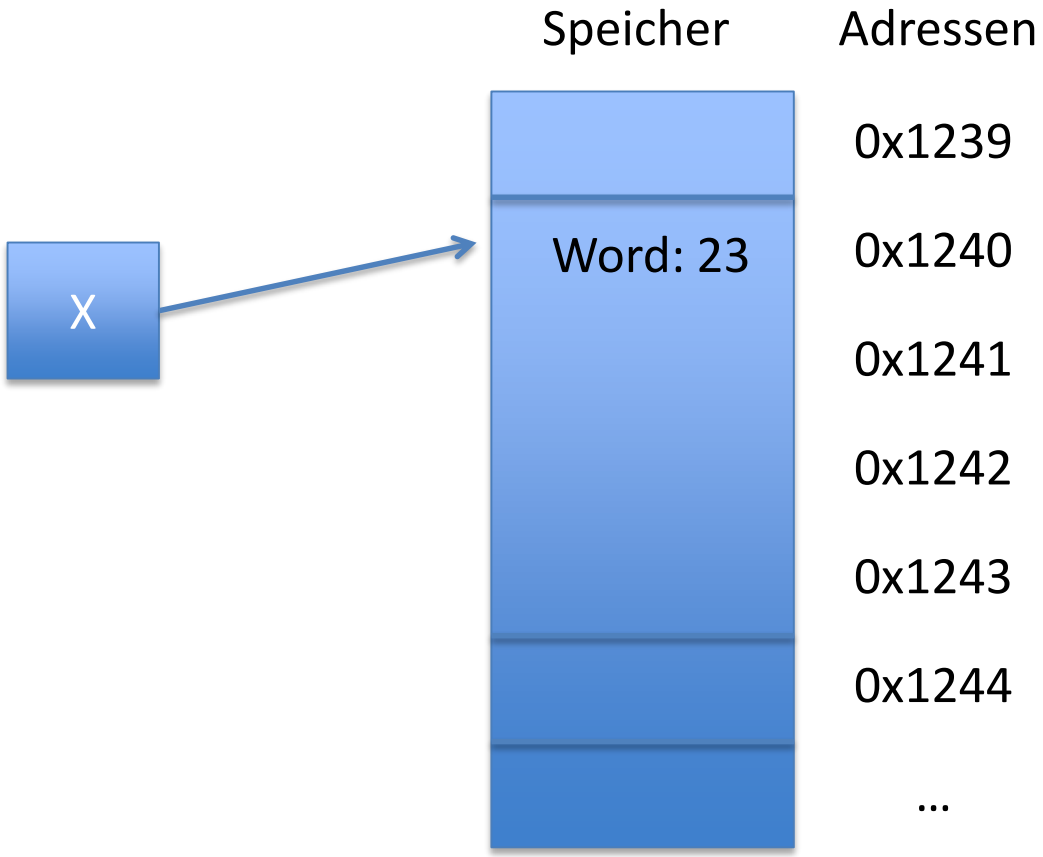
Call by Value vs. Call by Reference:

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

Call by Value / Call by Reference

Call by Value vs. Call by Reference:

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

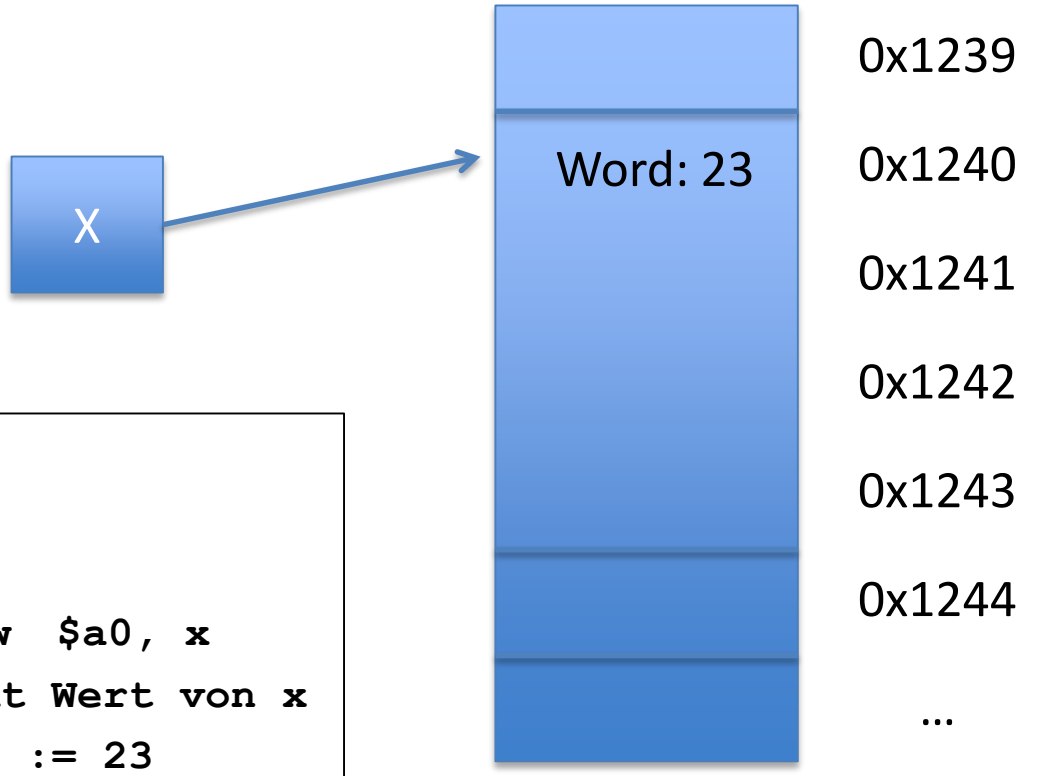


Call by Value vs. Call by Reference:

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

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

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



Call by Value / Call by Reference

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

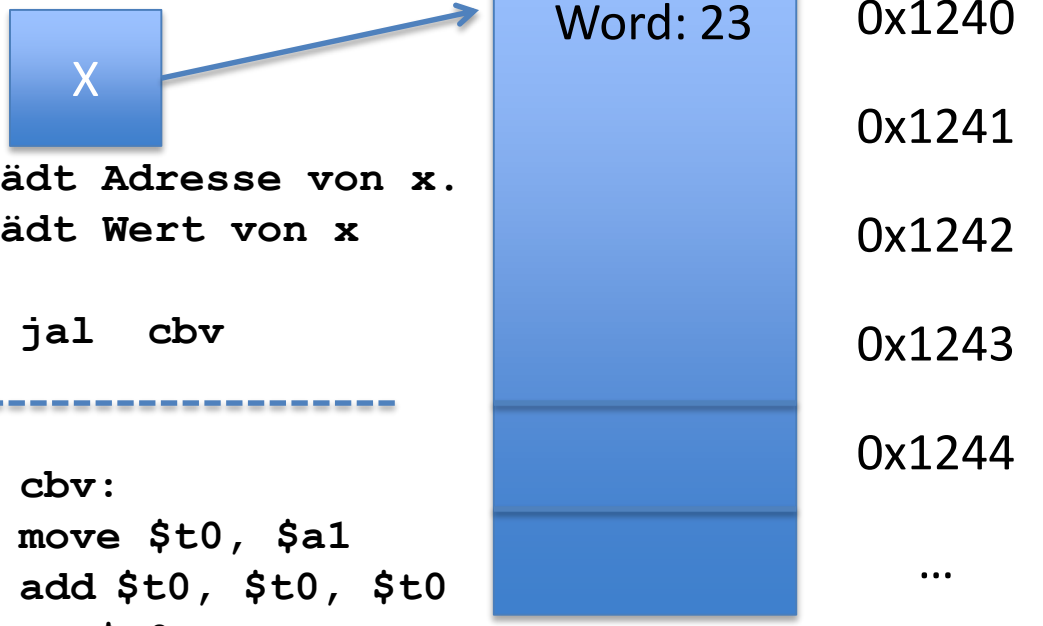
- **Daten** (call by value) oder
- die **Adressen** von Daten (call by reference)

Beispiel:

```
.data
x:  .word 23
    .text
main:
    la      $a0,x           # lädt Adresse von x.
    lw      $a1,x           # lädt Wert von x

    jal     cbr               jal  cbv
```

```
cbr:
    lw      $t0,($a0)
    add     $t0, $t0, $t0
    sw      $t0,($a0)
    jr      $ra
```

```
cbv:
move $t0, $a1
add $t0, $t0, $t0
sw $t0, x
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!).

Aufgabe 2

Schreiben Sie Ihr SPIM-Programm von Aufgabe 1 nun so um, dass die Parameter für das Unterprogramm `proc` per **Call-by-Reference** übergeben werden!

Beispiel für Sprünge


```
IF Betrag > 100
 THEN Rabatt := 3
 ELSE Rabatt := 2
END;
```

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

Assemblerprogramm ?

Beispiel für Sprünge

IF Betrag > 100
 THEN Rabatt := 3
 ELSE Rabatt := 2
END;
Annahme: Betrag ist in Register $t0, Rabatt soll ins Register $t1

Assemblerprogramm:

```
main:
  ble $t0, 100, else    # IF Betrag > 100
  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 ?

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:
```

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.

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
```

Aufgabe 3

Übersetzen Sie folgenden Java-Code in lauffähigen MIPS-Code:

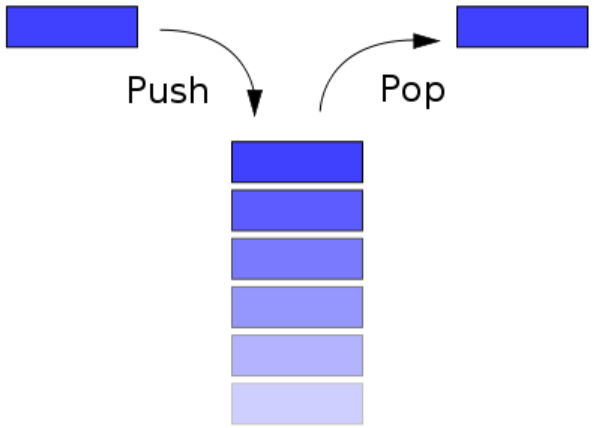
```
public class Test {

   public static void main(String[] args) {

      int[] mein_array = new int[]{3,4,5,6};

      int i = 0;

      while(i<4){
         System.out.print(mein_array[i]+" ");
         i++;
      }
   }

}
```

Ausgabe: 3 4 5 6

(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 (Bei CISC-Prozessoren)

- 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!

Arbeiten mit dem Stack

PUSH und POP existieren bei MIPS nicht.

- Nutzen der Standardbefehle!
- $sp zeigt nach Konvention auf das **erste freie Wort** auf dem Stack!

Element(e) auf den Stack ablegen:

```
### PUSH ###

addi    $sp, $sp, -4
sw      $t0, 4($sp)
```

```
### PUSH more ###

addi    $sp, $sp, -12
sw      $t0, 12($sp)
sw      $t1,  8($sp)
sw      $t2,  4($sp)
```

Element(e) holen:

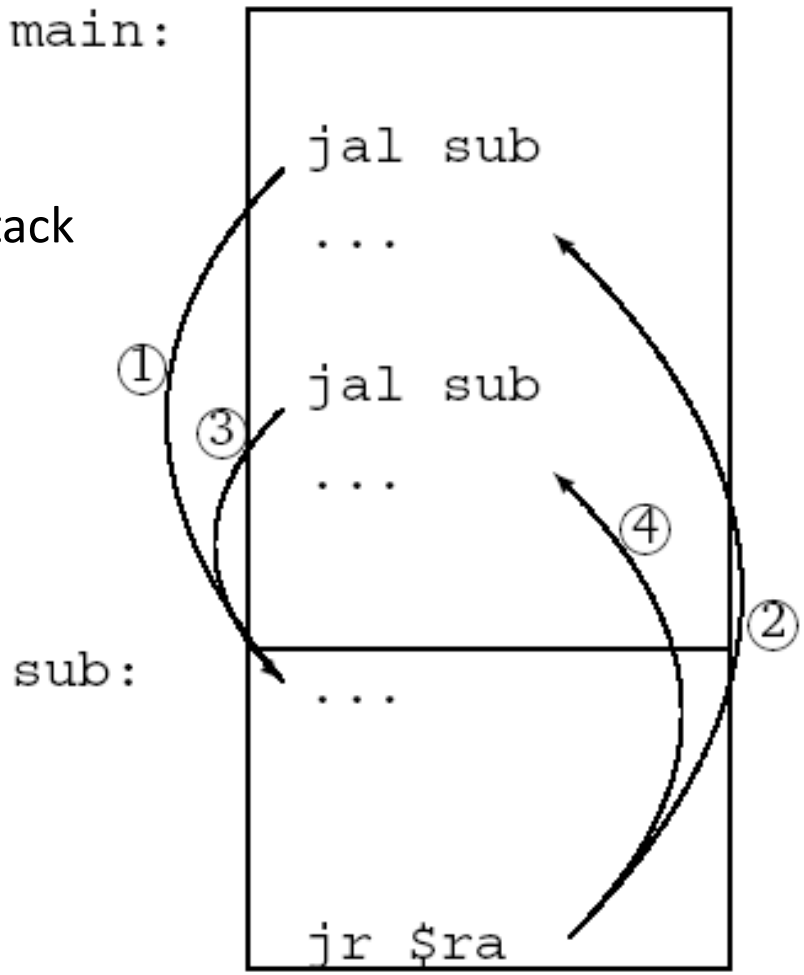
```
### POP ###

lw      $t0, 4($sp)
addi    $sp, $sp, 4
```

```
### POP more ###

lw      $t0, 12($sp)
lw      $t1,  8($sp)
lw      $t2,  4($sp)
addi    $sp, $sp, 12
```

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).



Aufgabe 4

Schreiben Sie Ihr SPIM-Programm von Aufgabe 2 nun so um, dass das Unterprogramm `proc` die benötigten Argumente auf dem **Stack** findet und das Hauptprogramm wiederum das Ergebnis auf dem Stack findet und ausgeben kann!

Problem: Überschreiben von Inhalten

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**

Prozedur-Konvention beim SPIM

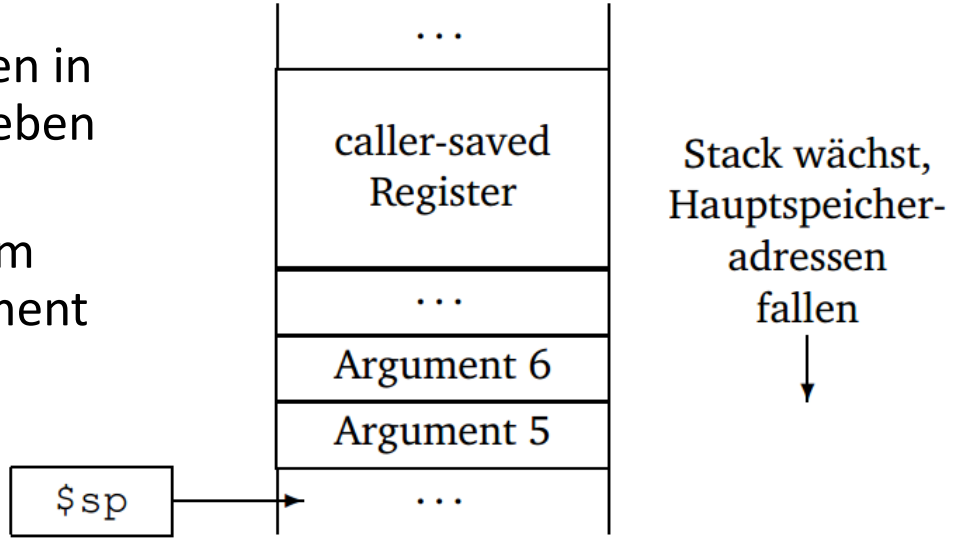
Prolog des Callers (aufrufendes Programm):

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)

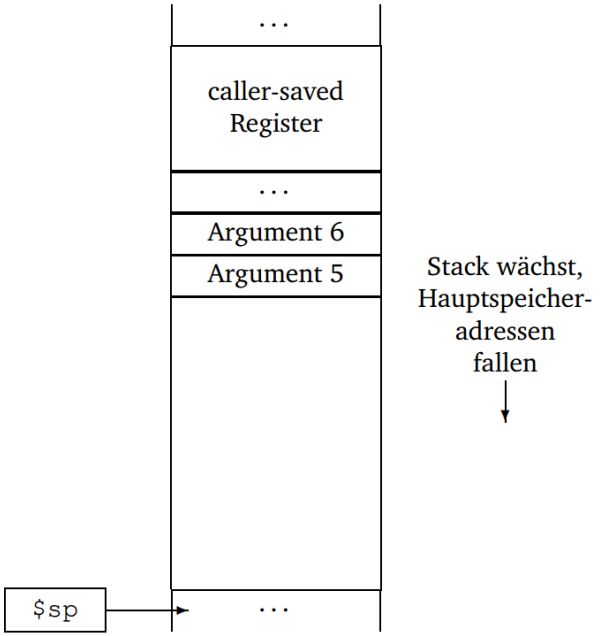


Starte die Prozedur (jal)

Prolog des Callee (I)

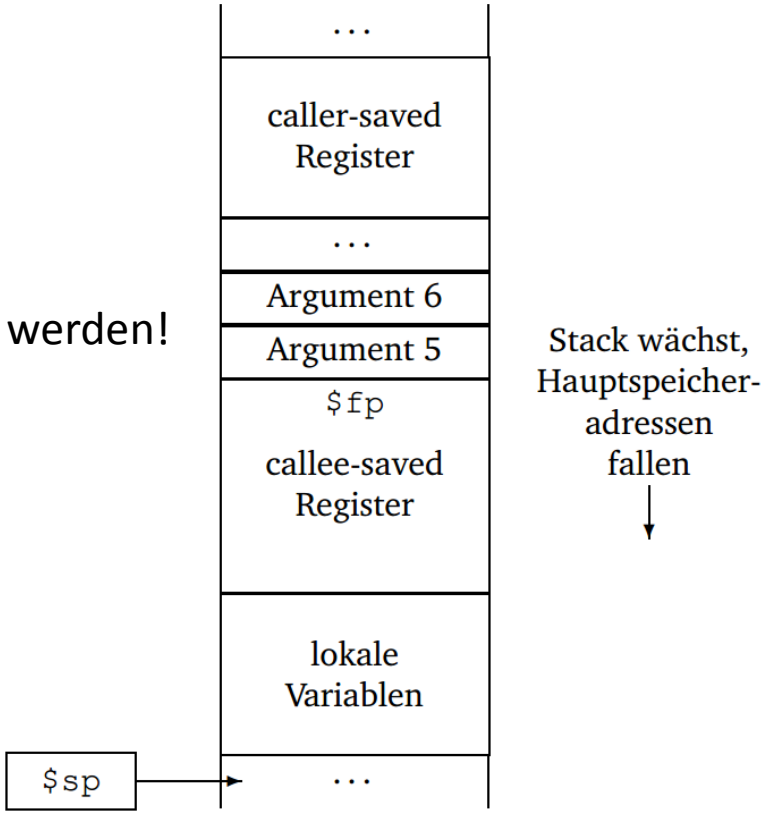
- **Schaffe Platz auf dem Stack (Stackframe)**
  - Stackframe: der Teil des Stacks, der für das Unterprogramm gebraucht wird
  - Subtrahiere die Größe des Stackframes vom Stackpointer:

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



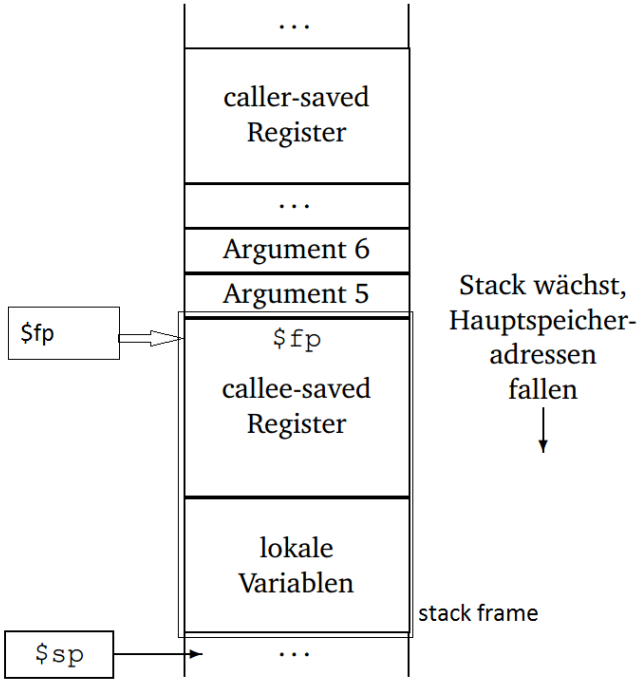
Prolog des Callee (II)

- **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)
  - $fp sollte zuerst gesichert werden
    - Entspricht der Position des ursprünglichen Stackpointers
  - **Achtung**: das Register $ra wird durch den Befehl jal geändert und muss ggf. gesichert werden!



Prolog des Callee (III)

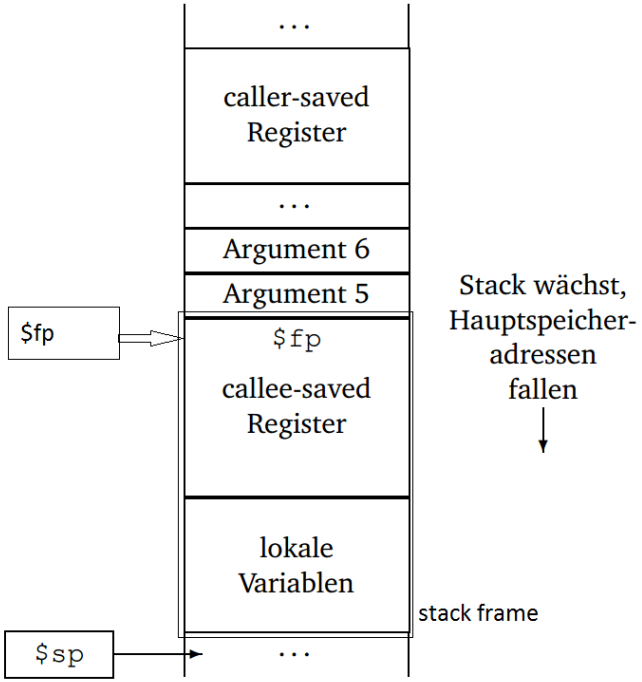
- **Erstelle den Framepointer:**
  - $fp enthält den Wert, den der Stackpointer zu Beginn der Prozedur hält
  - Addiere die Größe des Stackframe zum Stackpointer und lege das Ergebnis in $fp ab.
  - Aktueller Framepointer zeigt dann auf die Stelle, wo der vorheriger Framepointer liegt.
- Durch den Framepointer können wir auf die Argumente und lokalen Variablen der vorherigen Prozedur zugreifen.
- Effizient, aber fehleranfällig!



Der Callee

- Hat nun die Möglichkeit:
  - durch positive Indizes (z.B.: `4($fp)`) auf den Wert der Argumente zuzugreifen
  - durch negative Indizes (z.B.: `-4($fp)`) auf den Wert der lokalen Variablen zuzugreifen

- Werte der gesicherten Register dürfen dabei nicht überschrieben werden!



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.
    - Bsp.: `lw $s0, 4($sp)`
  - Achtung: den Framepointer als letztes Register wieder herstellen!
    - Bsp.: `lw $fp, 12($sp)`
- **Entferne den Stackframe:**
  - Addiere die Größe des Stackframes zum Stackpointer.
    - Bsp.: `addi $sp, $sp, 12`
- **Springe zum Caller zurück:**
  - Bsp.: `jr $ra`

Prozedur-Konvention beim SPIM

Epilog des Callers:

- **Stelle gesicherte Register wieder her:**
  - Vom Caller gesicherte Register wieder herstellen
    - Bsp.: `lw $t0, 4($sp)`
  - 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.
    - Bsp.: `addi $sp, $sp, 12`

Aufgabe 5 (Bonus)

Schreiben Sie ein SPIM Programm, das die Fakultätsfunktion **rekursiv** löst.

Analog, hier die Funktion in c:

```
long factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return(n * factorial(n-1));
}
```

Literatur

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

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