LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Vorlesung Rechnerarchitektur

Sommersemester 2017

Carsten Hahn

8. Juni 2017

Agenda

Grundlagen:

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

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
```

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
```

Trennung von Programm und Daten

Grundprinzip (Von-Neumann):

- Gemeinsamer Speicher für Daten und Programme

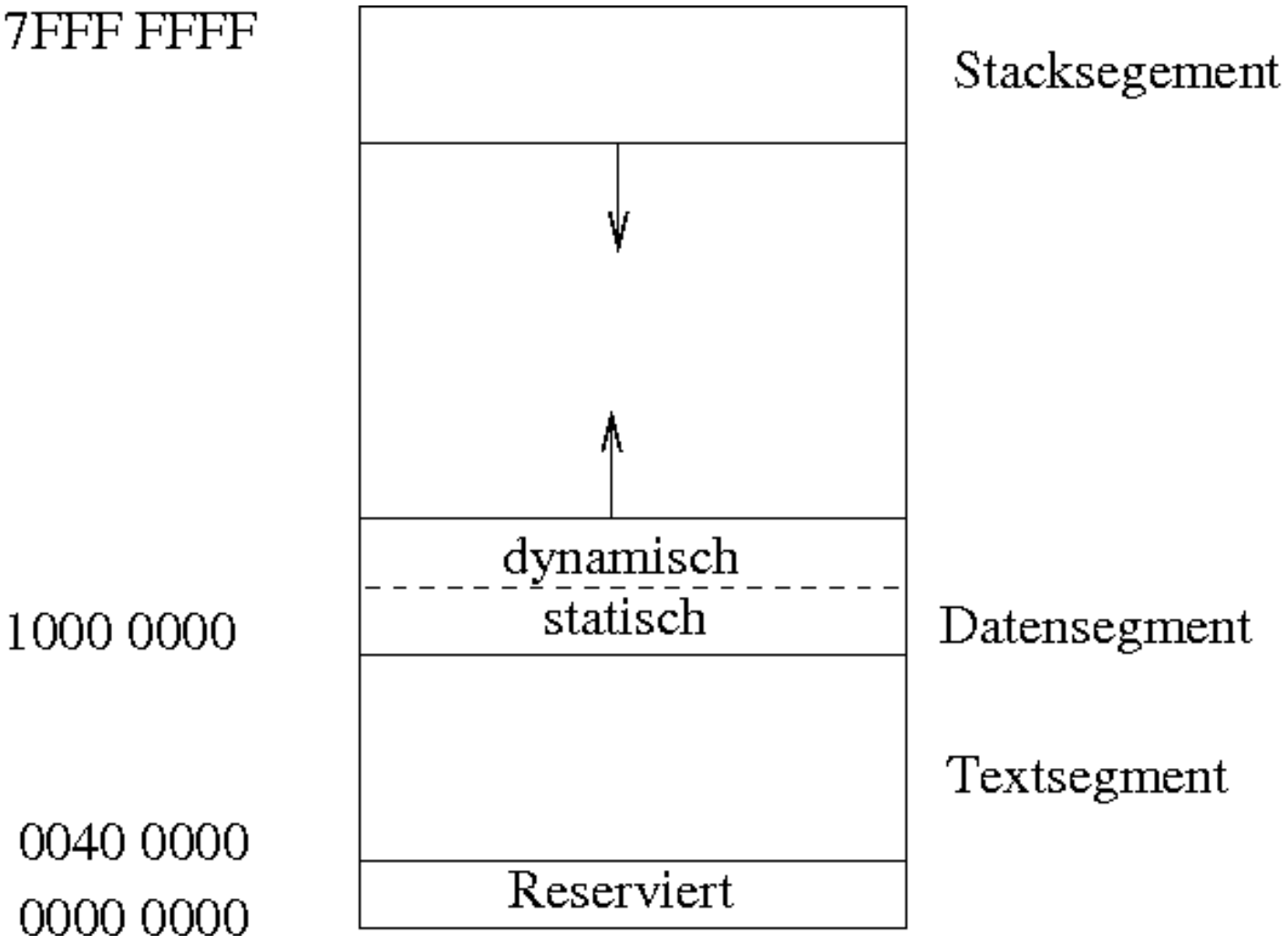
SPIM teilt den Hauptspeicher in **Segmente,** um Konflikte zu vermeiden:

- **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

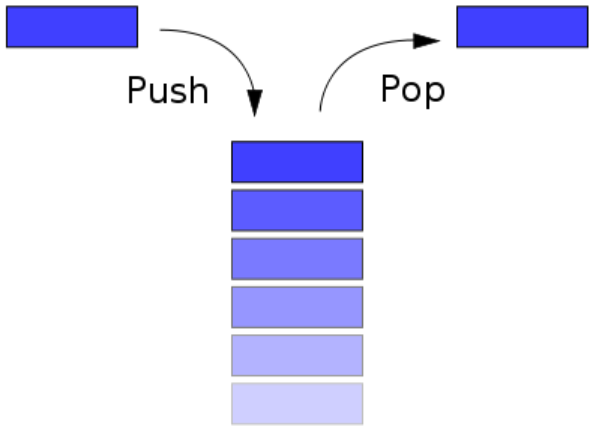
Speicherlayout



(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 (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
```

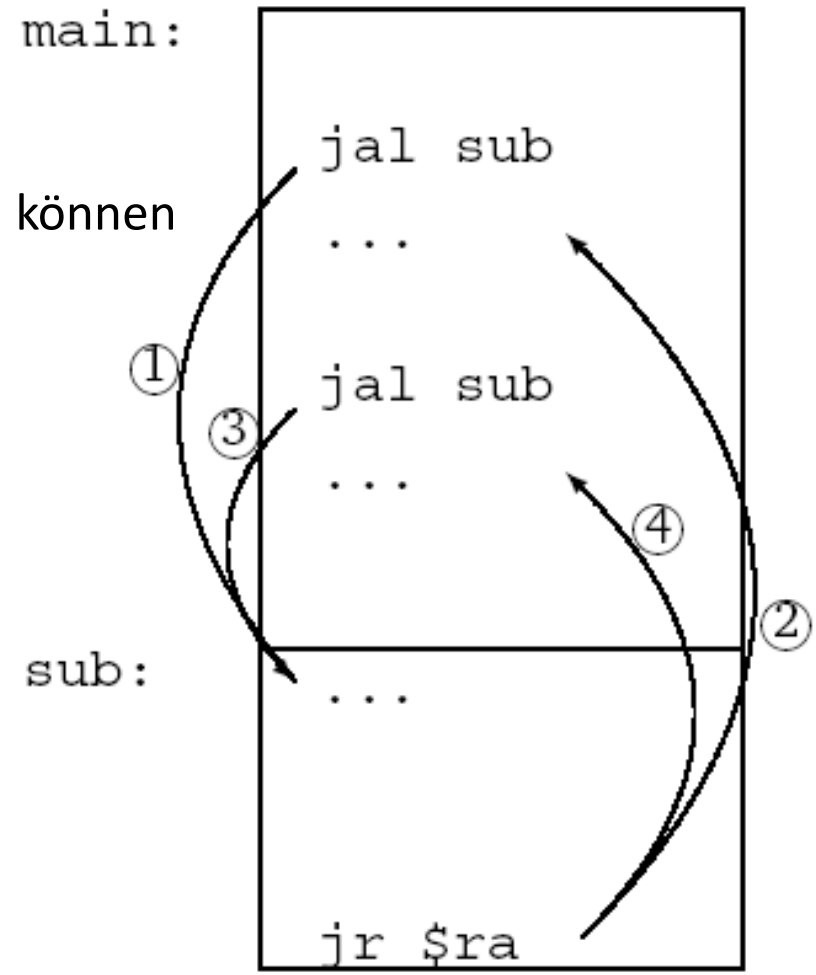
```
### POP more ###

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

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:

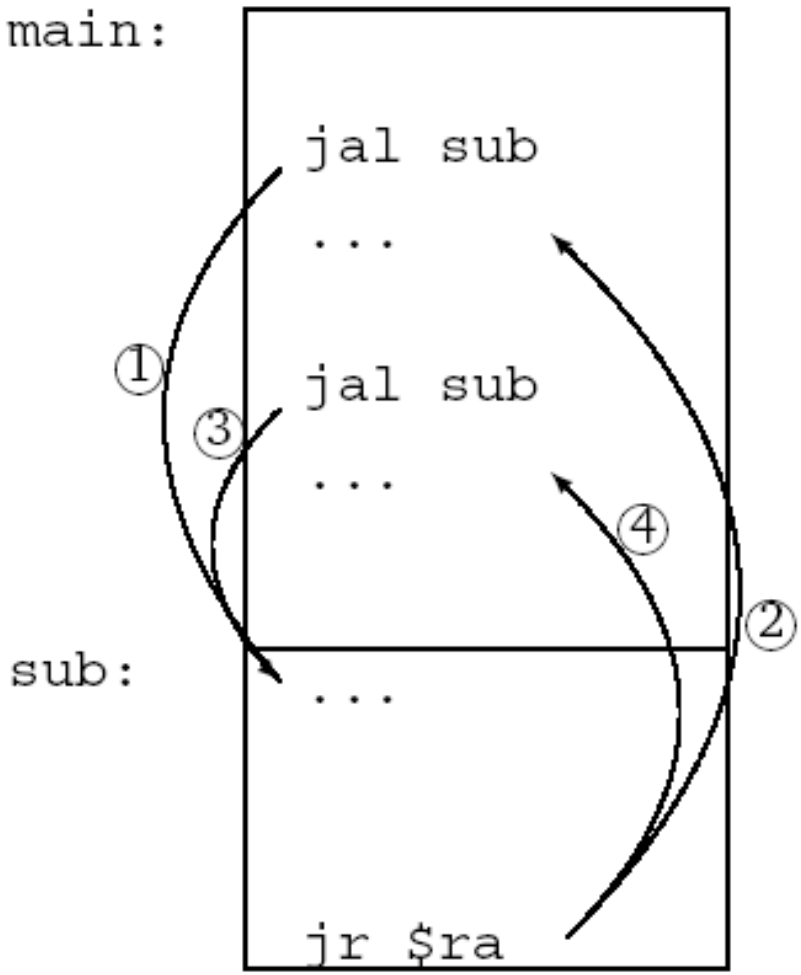
```
public String myFunction(String param){
  return "Hallo: " + param;
}
```

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 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, $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
```

Unterprogramm II

Was machen wir, wenn wir mehr Argumente übergeben wollen?

- bzw. mehr Ergebnisse bekommen wollen?

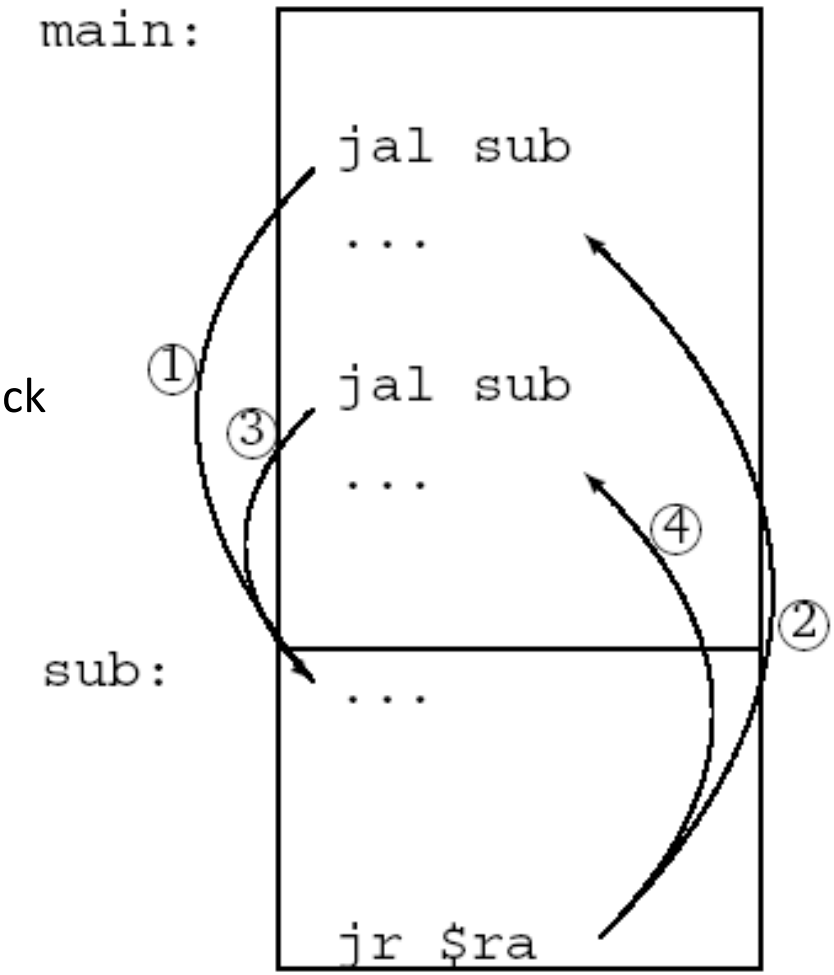
Methode 2:

- 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!**
  - 1. Prolog des Callers
  - 2. Prolog des Callees
  - 3. Epilog des Callees
  - 4. Epilog des Callers

Prozedur-Konvention beim SPIM

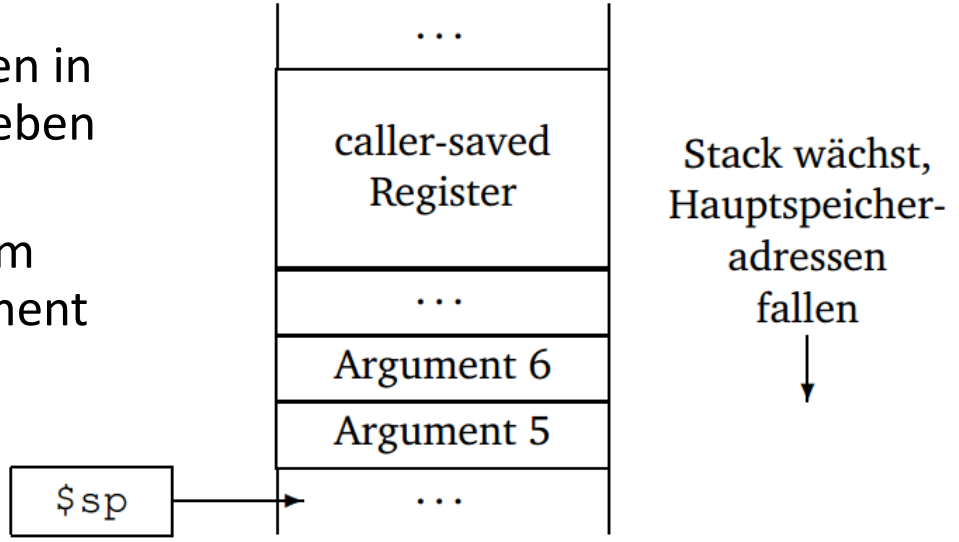
Prolog des Callers (aufrufendes Programm):

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)

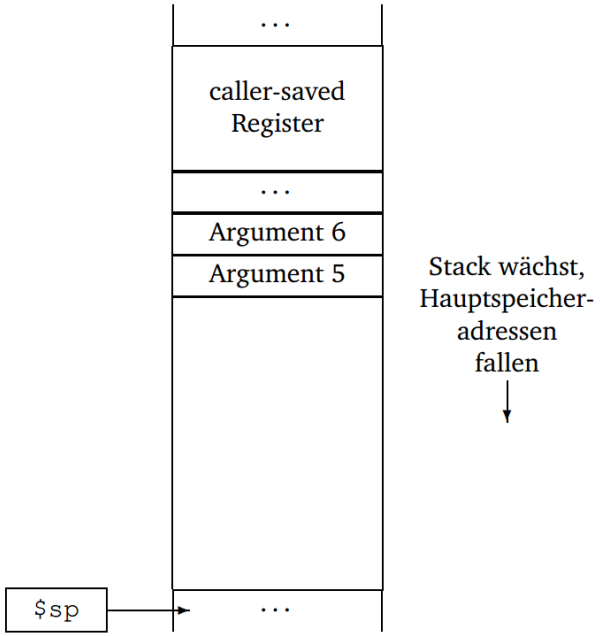


Starte die Prozedur (jal)

Prolog des Callee (I) (aufgerufenes Unterprogramm)

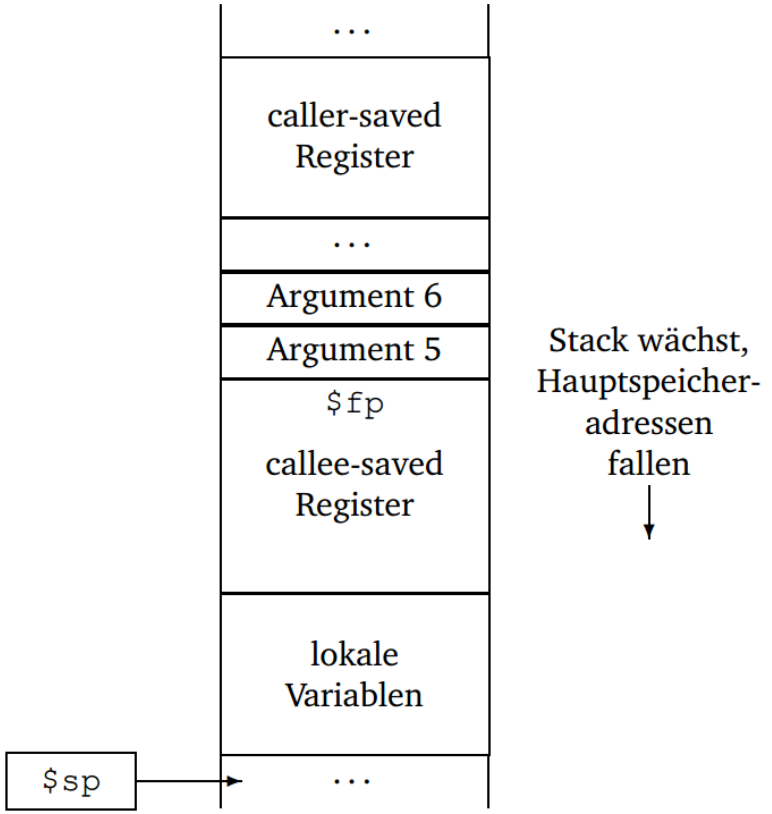
- **Schaffe Platz auf dem Stack (Stackframe)**
  - Stackframe: der Teil des Stacks, der für das Unterprogramm gebraucht wird
  - Subtrahiere die Größe des Stackframes vom Stackpointer:

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



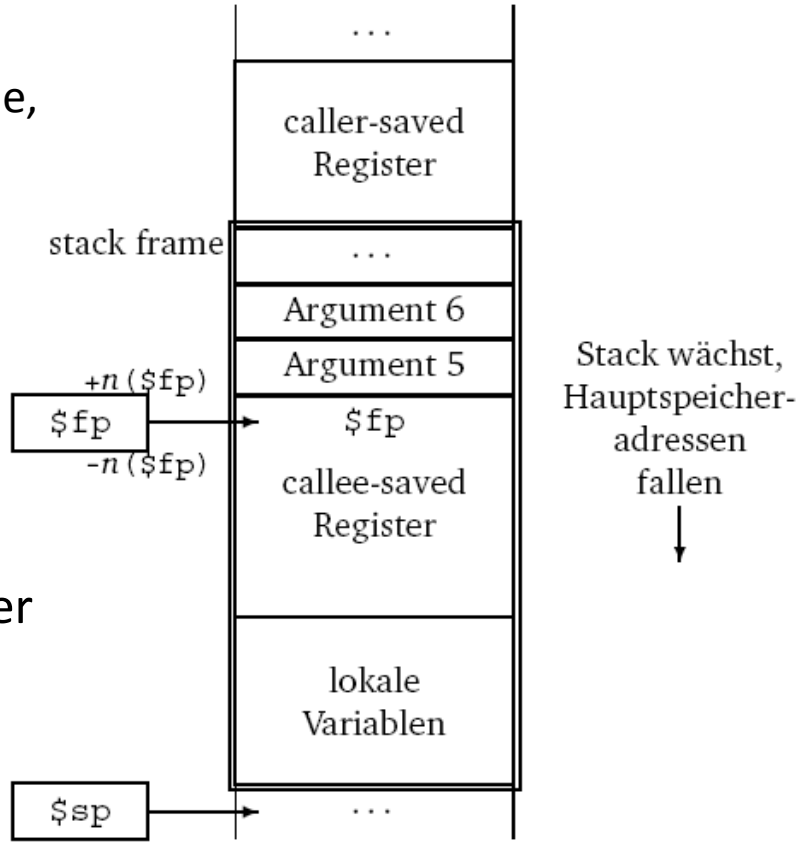
Prolog des Callee (II)

- **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)

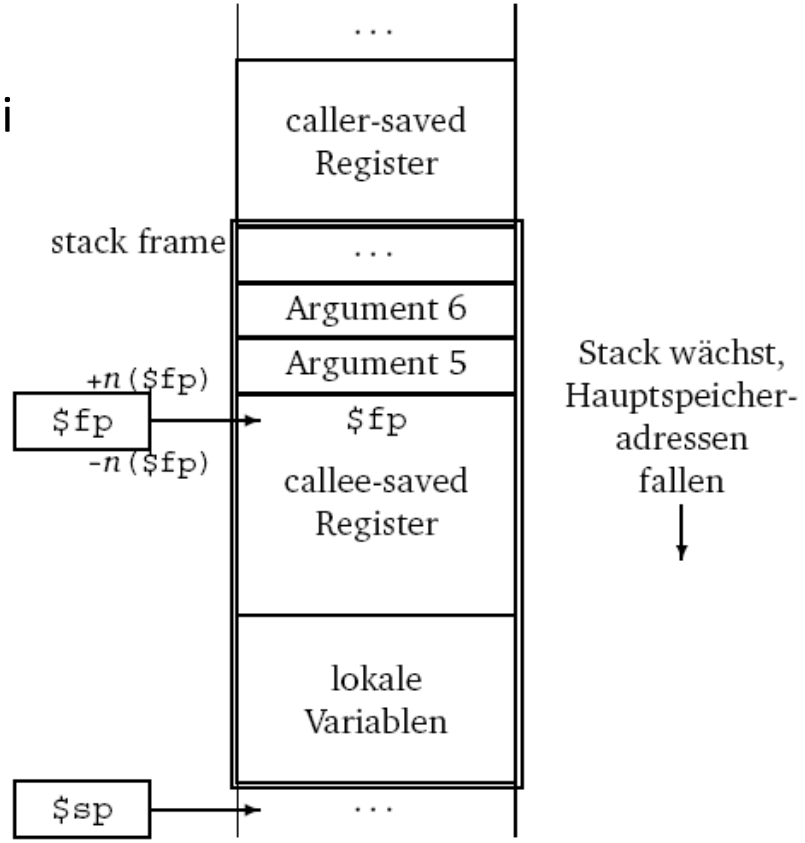
- **Erstelle den Framepointer:**
  - $fp enthält den Wert, den der Stackpointer zu Beginn der Prozedur hält
  - Addiere die Größe des Stackframe zum Stackpointer und lege das Ergebnis in $fp ab.
  - Aktueller Framepointer zeigt dann auf die Stelle, wo der vorheriger Framepointer liegt.
- Durch den Framepointer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



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

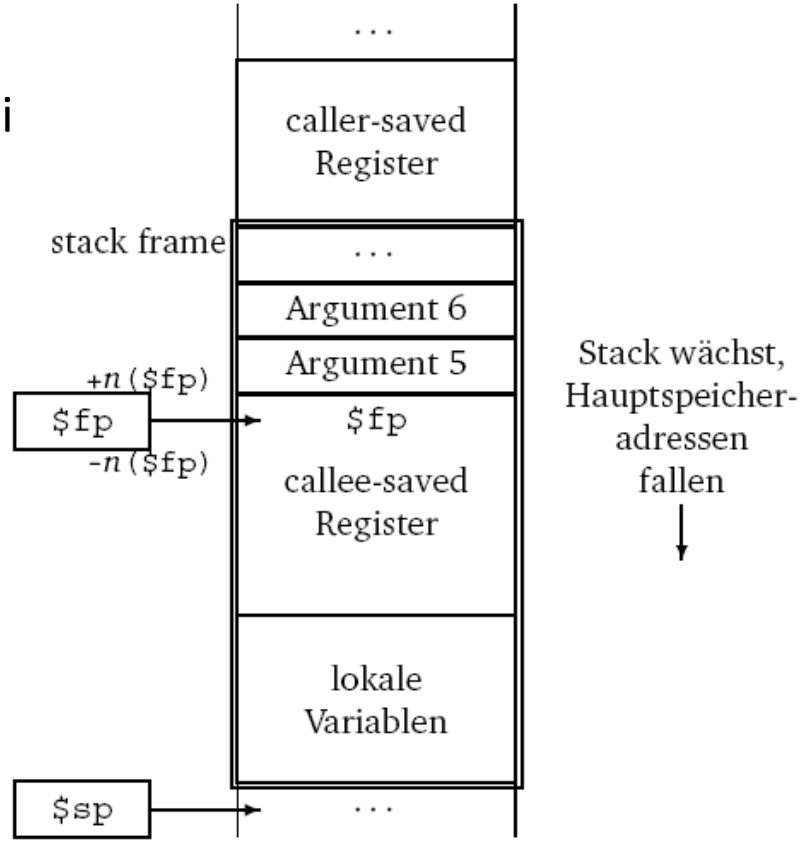
- Werte der gesicherten Register dürfen dabei nicht überschrieben werden!



Der Callee

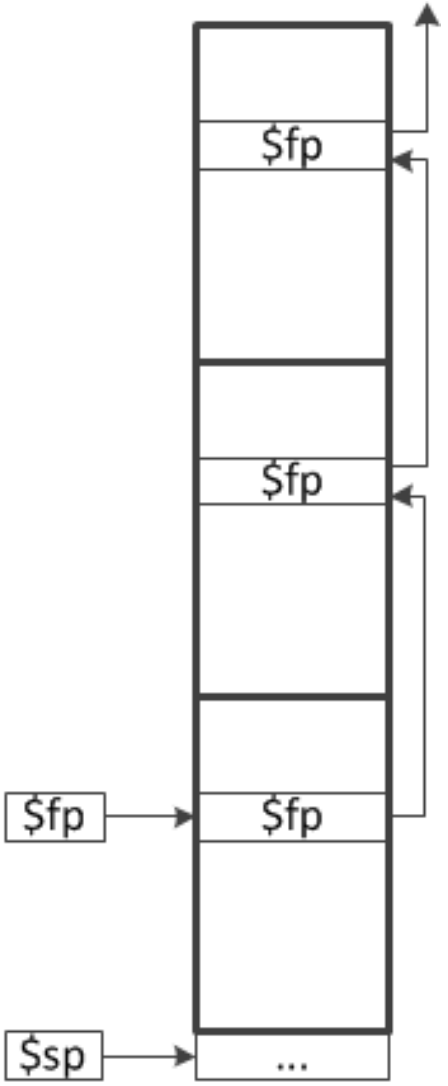
- Hat nun die Möglichkeit:
  - durch positive Indizes (z.B.: `4($fp)`) auf den Wert der Argumente zuzugreifen
  - durch negative Indizes (z.B.: `-4($fp)`) auf den Wert der lokalen Variablen zuzugreifen

- Werte der gesicherten Register dürfen dabei nicht überschrieben werden!



Geschachtelte Unterprogrammaufrufe:

- Bei jedem weiteren Unterprogrammaufruf erfolgen jeweils wieder
  - Prolog des *Callers*
  - Prolog des *Callee*
- Dadurch kommen weitere Stackframes hinzu
- Die Verkettung der Framepointer ermöglicht die Navigation durch den Aufruf-Stack (Call Stack)
  - Ermöglicht das Debuggen eines Programms
- Nach der Abarbeitung jedes Unterprogramms müssen
  - Epilog des *Callee*
  - Epilog des *Callers* erfolgen



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`

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`

Beispiel

Assembler-Programm der Fakultätsfunktion

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{k=1}^{n} k$$

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

Funktion in c:

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

Call by Value / Call by Reference

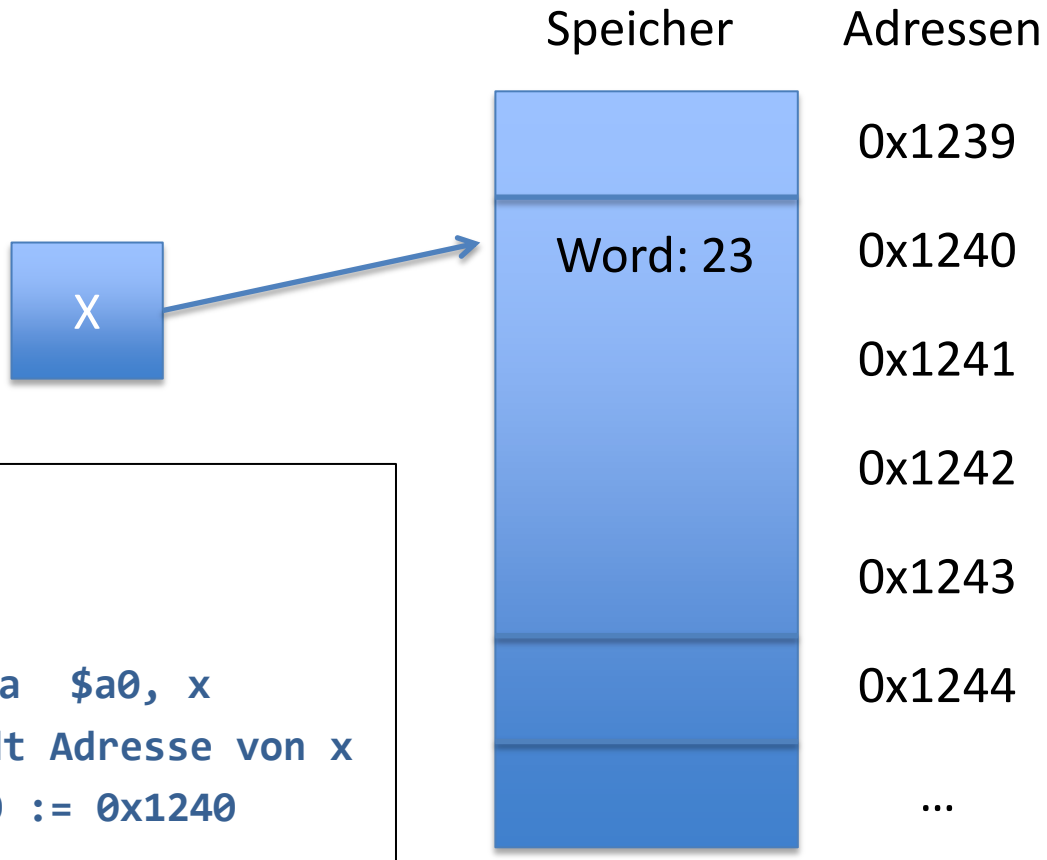
Variablen (Daten) können auf 2 Arten referenziert und an ein Unterprogramm übergeben werden:

- **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

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)

```
.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

- Sprünge, IF, SWITCH, Schleifen (`b,j,jal,beqz`,…)
- Aufbau & Speicher (Stack Segment)
- 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-sose17/