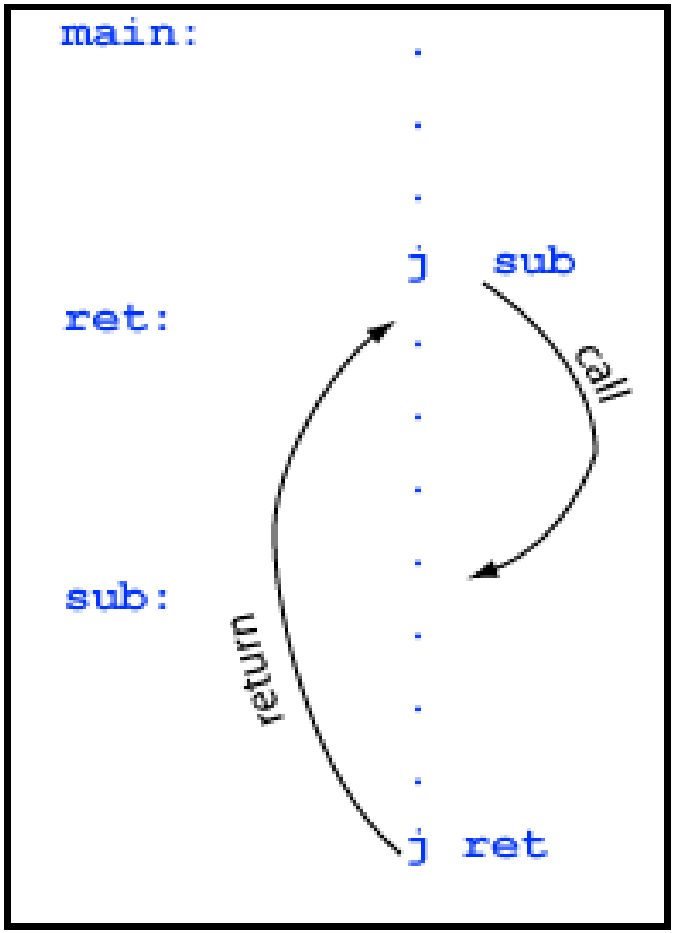
Unterprogramme

Unterprogramme

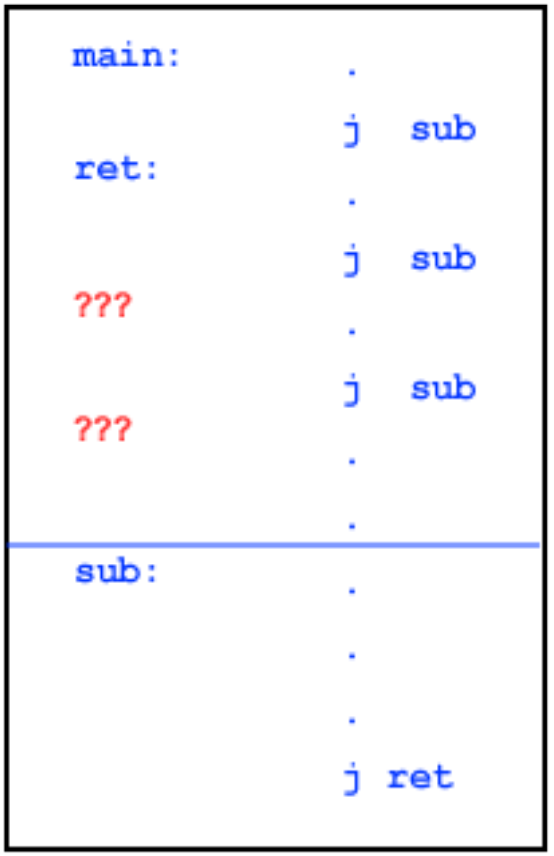
- wichtiges Hilfsmittel
- für mehrfach benötigte Programmabschnitte
- spielen in höheren Programmiersprachen eine wesentliche Rolle
- in Assembler sind bestimmte Konventionen nötig

Unterprogramme in MIPS/SPIM



Subroutine Called Once

j und *b* sind Kommandos, die zur Abarbeitung von Unterprogrammen verwendet werden:
Soll das Programm ein UP ausführen, kann mit *j* (jump) zu diesem Programmabschnitt gesprungen werden. Mit einem weiteren *j* wir an die Stelle des Aufrufs zurückgesprungen. Dazu wird die Anweisung nach dem Aufruf markiert.



Subroutine Called Three Times

Problem: mehrfacher Aufruf eines Unterprogramms – die Rücksprungadresse ist nicht eindeutig angebbar

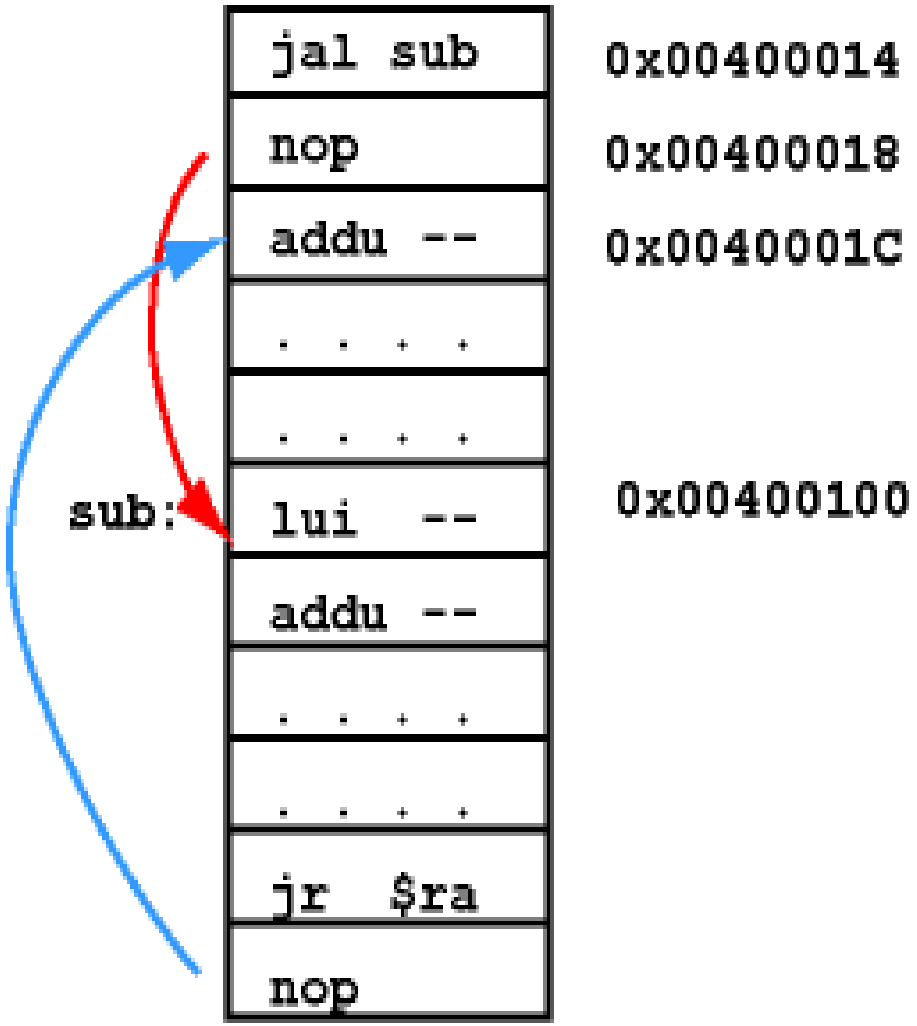
-- eine eindeutige Rücksprungadresse ist wichtig

Lösung für dieses Problem:

jal – Befehl und Register *$ra*

```
jal sub:  $ra = PC+ 4      # $ra  = adresse 8 bytes nach jal
          PC =  sub        # PC  = Adresse des Befehls mit der Marke sub
```

Beispiel für eine *jal*-Instruktion



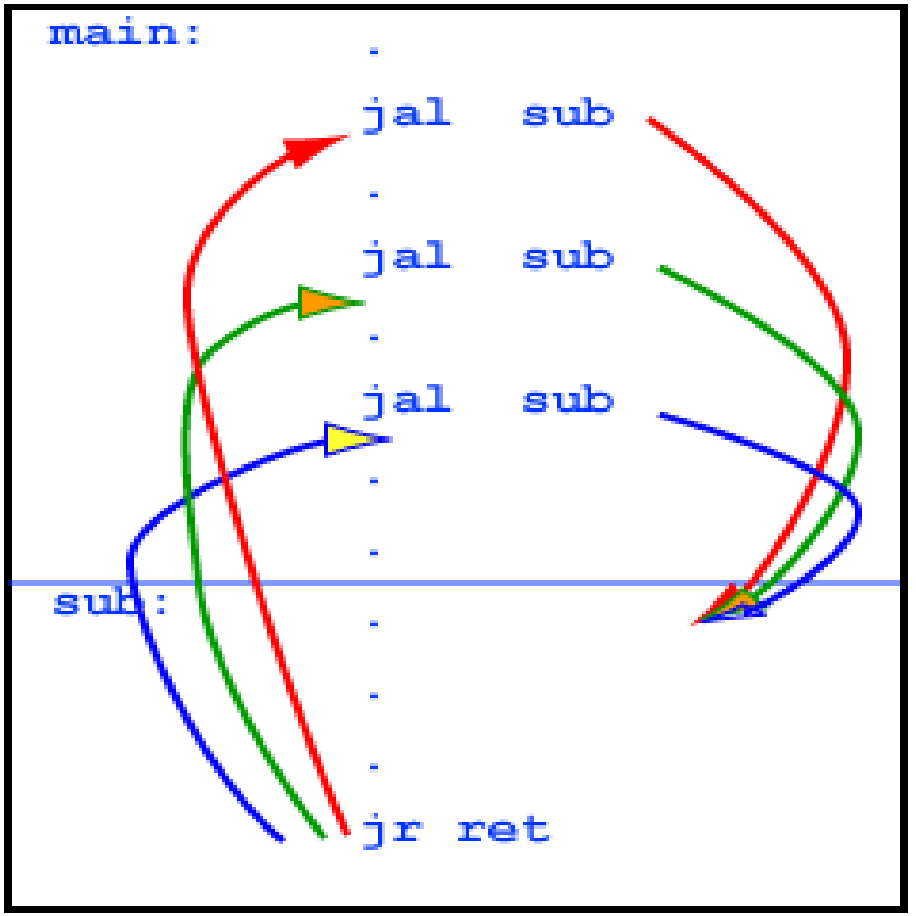
jal-adresse ist: 0x00400014

in *$ra* steht: 0x00400014 +8
0x004001C

Adresse von *sub*: 0x00400100

```
#  jal wird geladen, wenn der PC =
# 0x00400014
# der PC wird erhöht zu 0x00400018
jal sub:  $ra = PC+4
# $ra = 0x0040018+4
PC = sub
# lädt in den PC 0x00400100
Rückkehr zum aufrufenden Programm
jr  $ra      # PC = $ra
```

Korrekter Unterprogrammaufruf



Correct Subroutine Linkage

Abmachungen für den Unterprogrammaufruf

Welche Register können im UP benutzt werden?

* $t0 - $t9 - Das UP kann die Register benutzen

* $s0 - $s7 – Das UP darf diese Register nicht verändern

* $a0 - $a3 – Diese Register enthalten Argumente für das UP. Sie können verändert werden.

* $v0 - $v1 - Diese Register enthalten Resultate vom UP. (Rückkehrwerte)

Beispiel:

Ist die Verwendung der Register korrekt?

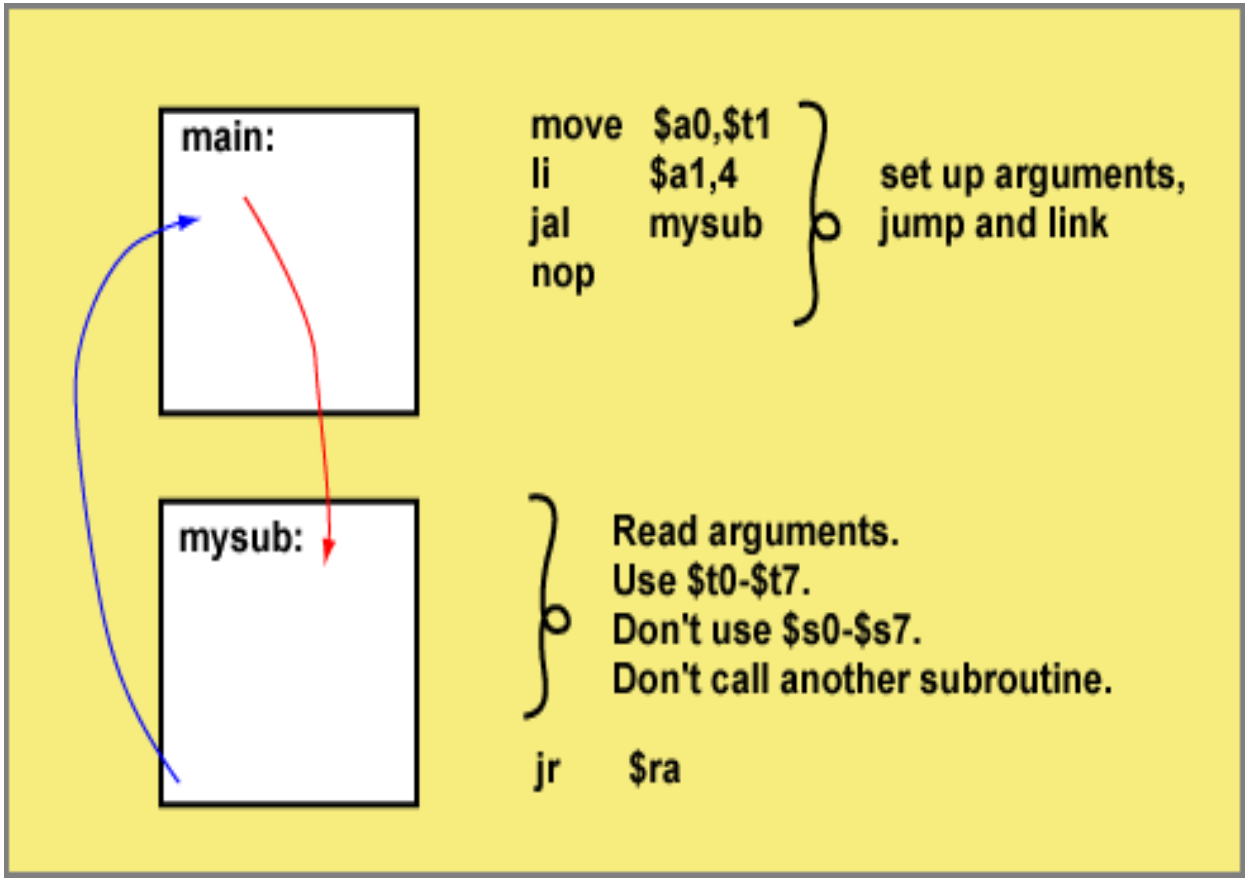
```
add     $t0,$s5,$s3     # Berechnung einer wichtigen Summe
jal     somesub         # Aufruf eines UP
mul     $s4,$t0,$v1     # Mult der Summe mit dem Resultat
```

Aufrufkonventionen:

1. Ein UP wird mit jal aufgerufen.
2. Ein UP darf kein anderes UP aufrufen.
3. Das UP benutzt zur Rückkehr *jr $ra*.
4. Benutzung der Register wie in Folie 7 beschrieben

Warum darf ein UP kein anderes UP aufrufen ?

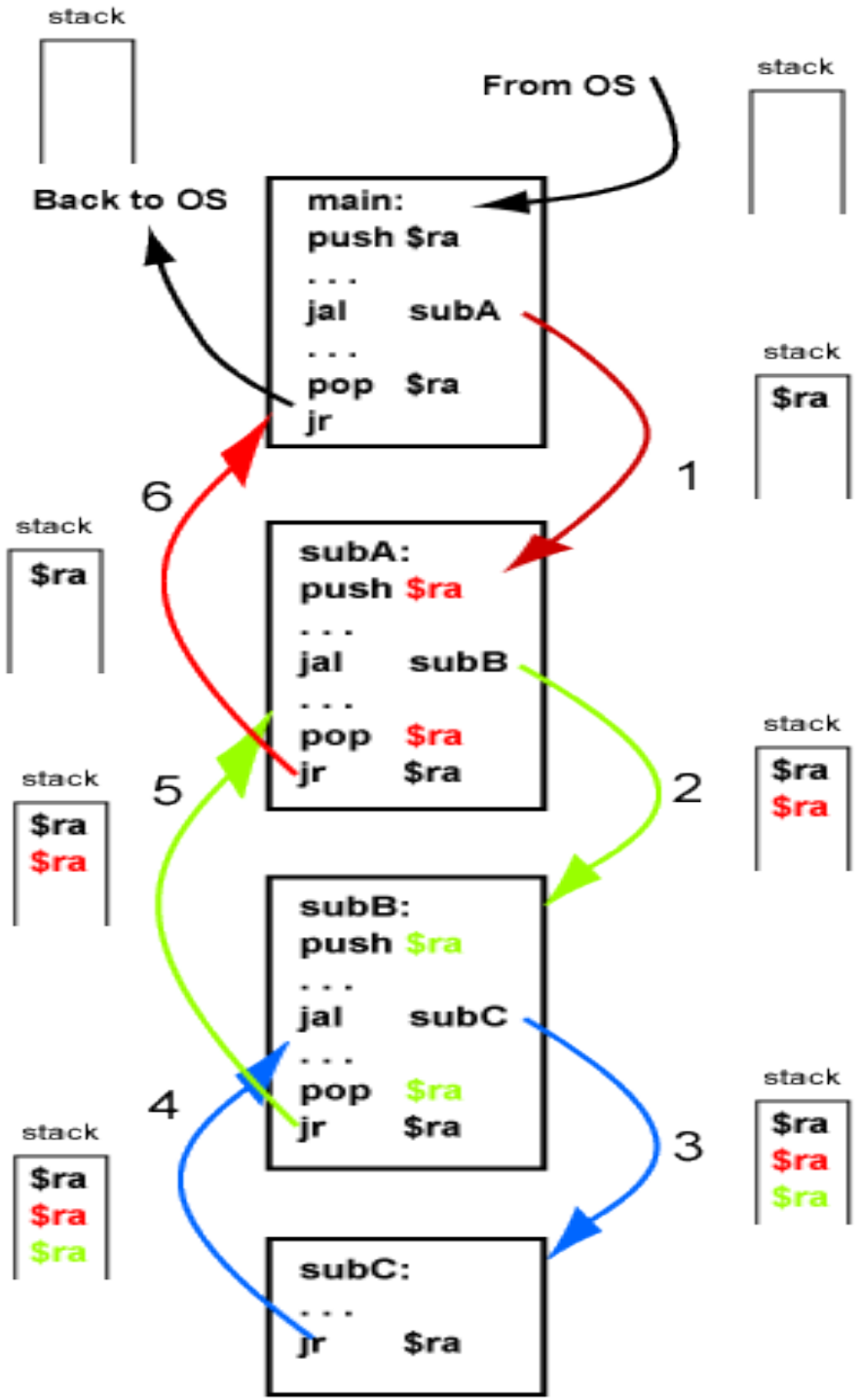
- $ra ist in Benutzung



Erweiterung der „Calling Conventions“

Abmachung, ein UP kann kein anderes UP aufrufen, ist hinderlich.

- Möglichkeit, den Stack zur Rettung des *$ra* zu benutzen



Beispiel mit erweiterten Konventionen

Abmachung , das Register $s0 - $s7 nicht benutzt werden dürfen, kann beim Aufruf von weiteren UP zu Problemen führen. UP will in $s0 - $s7 Werte retten, darf als UP diese Register aber nicht benutzen.

Lösung: Die Belegung der Register $s0 - $s7 werden im Stack gesichert.

```
subC:
        sub    $sp,$sp,4              # push $ra
        sw     $ra,($sp)
       . . . .
        jal    subD                   # call subD
        . . . .
        lw     $ra,($sp)              # pop return address
        add    $sp,$sp,4
        jr     $ra                    # return to caller

     # subD wird $s0 und $s1  benutzen
subD:
        sub    $sp,$sp,4              # push $s0
        sw     $s0,($sp)
        sub    $sp,$sp,4              # push $s1
        sw     $s1,($sp)
        . . . .                       # Anweisungen, die $s0, $s1 benutzen
        lw     $s1,($sp)              # pop s1
        add    $sp,$sp,4
        lw     $s0,($sp)              # pop s0
        add    $sp,$sp,4
        jr     $ra                    # return to subC
```

Stack-basierende Aufruf-Abmachungen (an C, C++ angelehnt)

Aufruf eines Unterprogramms (Caller-Aufgaben):

1. Die Inhalte der Registers $t0-$t9, die Werte enthalten, werden gesichert. Im UP können die Registerinhalte verändert werden.
2. Argumente werden in die Registern $a0 - $a3 kopiert.
3. Aufruf des UP mit *jal*.

Prolog des Unterprogramms (am Anfang des UP):

1. Ruft das UP andere UPs auf, muß $ra in den Stack gebracht werden.
2. Kopiere die Register $s0-$s7, die das UP ändern könnte, in den Stack.

UP-Hauptteil:

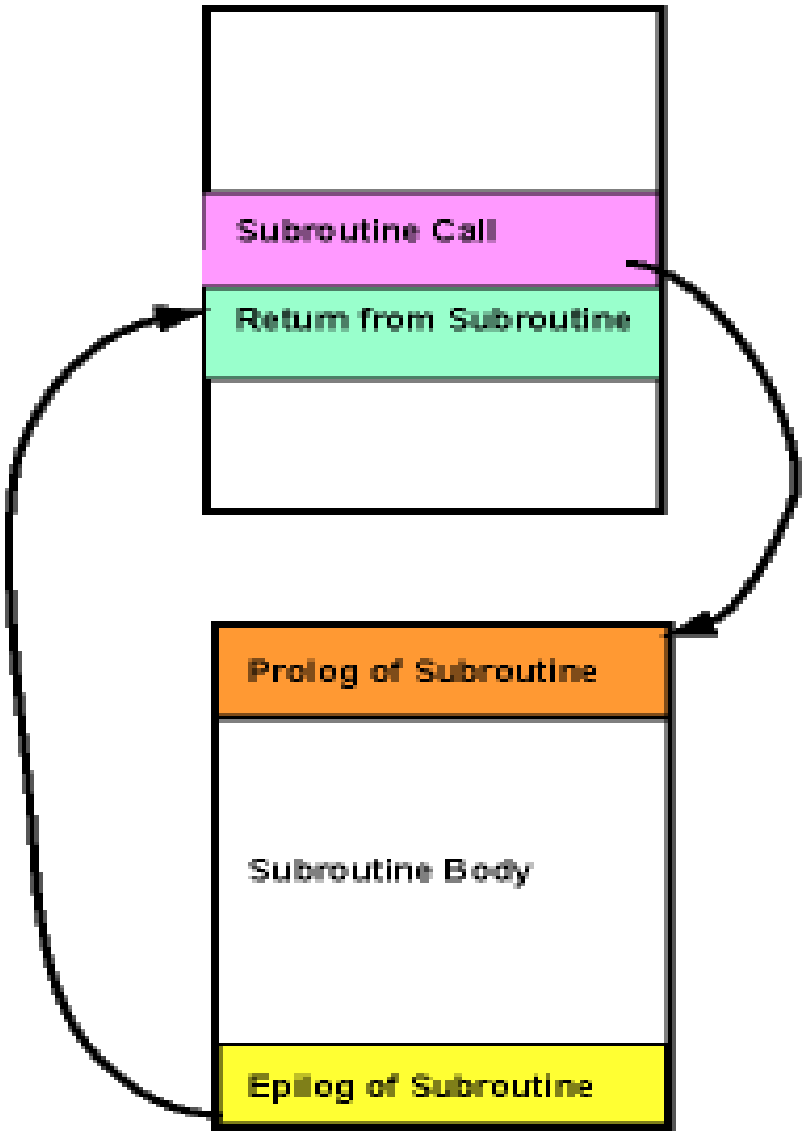
1. Das UP kann alle t und a-Register ändern und die s-Register, die vorher im Stack gesichert wurden·
2. Ruft das UP ein anderes UP auf, so müssen die entsprechenden Vorkehrungen getroffen werden.

UP-Abschluß (am Ende des UP):

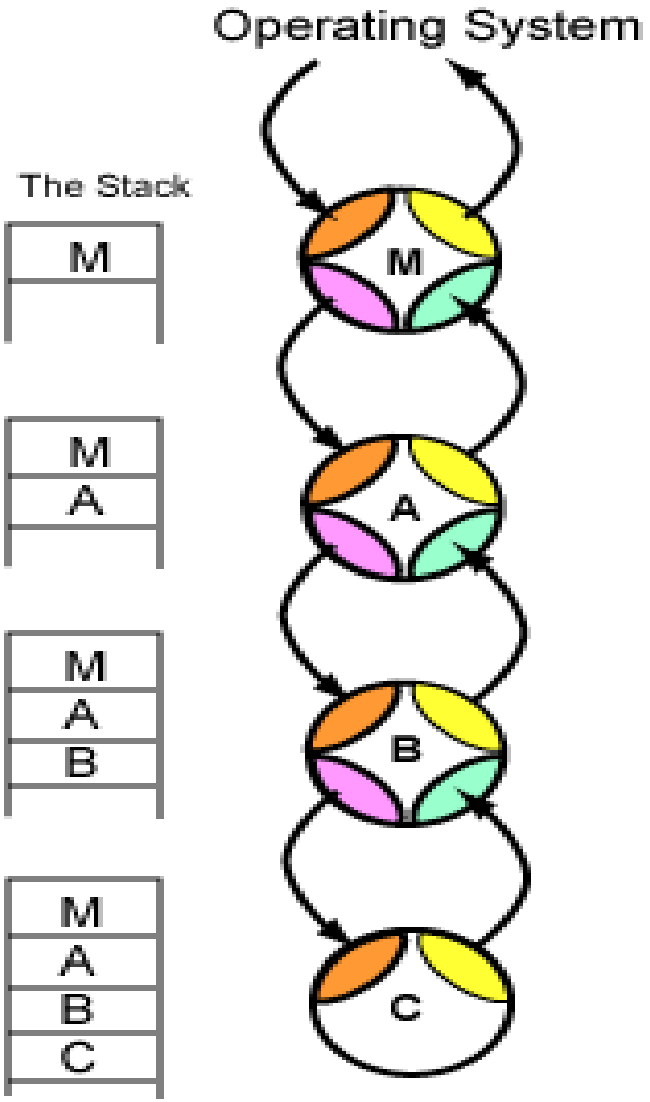
1. Rückkehrwerte in$v0-$v1
2. Kopieren der Registers $s0-$s7 in umgekehrter Reihenfolge in der sie auf den Stack gebracht wurden.
3. Wurde $ra auf den Stack gebracht – wiederherstellen von $ra.
4. Rückkehr zum aufrufenden Programm mit *jr $ra*.

Rückkehr vom UP (vom aufrufenden Programm auszuführen):

1. Die Register $t0-$t9 werden in der umgekehrten Reihenfolge aus dem Stack geholt.



Schema zum Aufbau

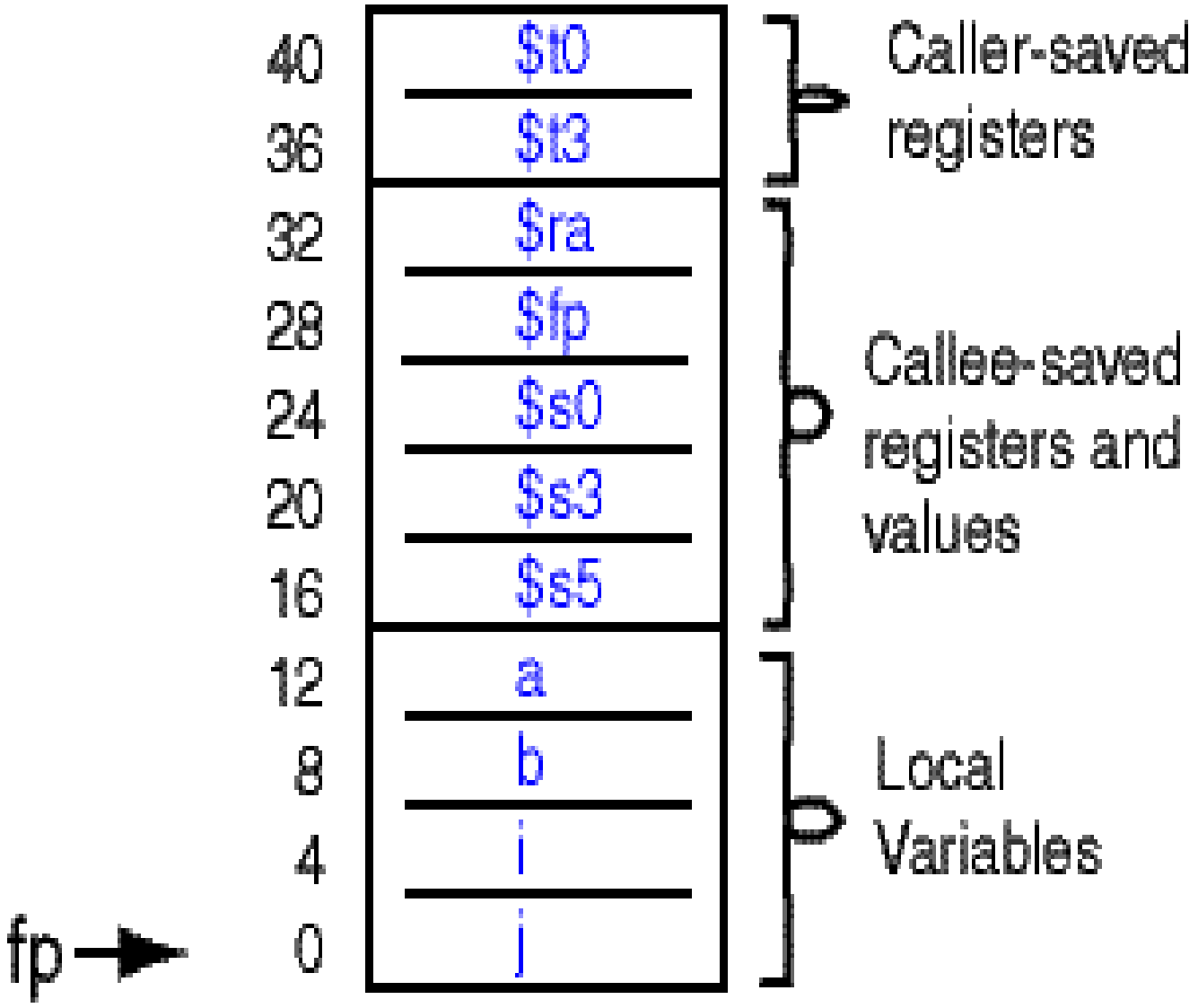


Geschachtelte UP-Aufrufe

Frame-basierende Aufruf-Konventionen

In höheren Programmiersprachen wird zwischen lokalen und globalen Variablen unterschieden.

Neben den im Stack gesicherten t-Registern des aufrufenden Programms, den im aufgerufenen Programm gesicherten s-Registern ist im Stack noch ein Bereich für lokale Variablen des Unterprogramms reserviert.


40
36
32
28
24
20
16
12
8
4
0
$t0
$t3
$ra
$fp
$s0
$s3
$s5
a
b
i
j
fp
Caller-saved registers
Callee-saved registers and values
Local Variables

Der Framepointer - $fp

Der Frampointer ($30 -$fp) enthält beim Start eines Unterprogramms den gleichen Wert wie der Stackpointer $sp.

Während der Abarbeitung der Subroutine verändert der Stackpointer seinen Wert. Es ist nützlich, ein Register zu haben, welches den Wert innerhalb der Routine nicht ändert. Dieses Register ist ist das Register $fp. In diesem Register bleibt der initiale Wert des Stackpointers erhalten.

Ruft das UP ein anderes UP auf, so wird der Framepointer im Stack mit gesichert.

Folgende Anweisung sei Teil eines Unterprogramms:

a = b + i + j;

Der Compiler könnte diese Anweisung wie folgt umsetzen

```
lw      $t0,8($fp)        # lade b
lw      $t1,4($fp)        # lade i
lw      $t2,0($fp)        # lade j
addu    $t3,$t0,$t1       # b + i
addu    $t3,$t3,$t2       # b + i + j
sw      $t3,12($fp)       # a =
```

Für die Anweisung a = a + 1 könnte die Umsetzung so sein:

```
lw      $t0,12($fp)       # get a
addiu   $t0,$t0,1         # a + 1
sw      $t0,12($fp)       # a =
```

Framebasierende Konventionen für UP-Aufruf

Aufruf einer Subroutine (aufrufendes Programm):

1. $t0-$t9 mit Werten werden im Stack gesichert.
2. Argumente in $a0-$a3.
3. Aufruf des UP mit jal.

UP-Anfang:

1. $ra sichern auf Stack.
2. $fp des aufrufenden Programms auf Stack.
3. $s0-$s7, die im UP geändert werden, auf Stack sichern.
4. Framepointer initialisieren: $fp = $sp – PlatzfürVariablen (4*Anzahl der lokalen Variablen).
5. Stack pointer initialisieren $sp = $fp.

Hauptteil des UP:

1. Bild des Stacks an dieser Stelle siehe auf Folie 19.
2. t- oder a- register oder gerettete s-Register können verändert werden
3. Das UP bezieht sich auf lokale Variablen abstand($fp)
4. Das UP wird Werte auf den Stack bringen und $sp dafür benutzen.
5. Ruft das UP ein anderes UP auf, folgt der Aufruf den Konventionen.

Beenden des UP:

1. Rückkehrwerte in $v0-$v1
2. $sp = $fp + PlatzfürVariablen.
3. Wiederherstellen der gesicherten $s0-$s7
4. Framepointer wiederherstellen $fp.
5. $ra wiederherstellen.
6. Rückkehr zum aufrufenden Programm *jr $ra*.

Rückkehr aus UP (aufrufendes Programm):

1. Wiederherstellen der Register $t0-$t9 die vor Aufruf des UP gesichert worden sind.

```
#  int mysub( int arg )
#  {
#    int b,c;                    // b: 0($fp)  // c: 4($fp)
#    b = arg*2;
#    c = b + 7;
#    return c;
#  }
                .text
                .globl  mysub
mysub:
                                                # Vorspann
        sub     $sp,$sp,4               #   1. Push return address
        sw      $ra,($sp)
        sub     $sp,$sp,4               #   2. Push caller's frame pointer
        sw      $fp,($sp)
        sub     $sp,$sp,4               #   3. Push register $s1
        sw      $s1,($sp)
        sub     $fp,$sp,8               #   4. $fp = $sp – PlatzfürVariablen
        move    $sp,$fp                 #   5. $sp = $fp
```

```
                                # Hauptteil des UP
mul     $s1,$a0,2               #     arg*2
sw      $s1,0($fp)              # b = "   "
lw      $t0,0($fp)              # get b
add     $t0,$t0,7               #     b+7
sw      $t0,4($fp)              # c = "  "
                                # Schluss
lw      $v0,4($fp)              #   1. Put return value in $v0
add     $sp,$fp,8               #   2. $sp = $fp + PlatzfürVariablen
lw      $s1,($sp)               #   3. Pop register $s1
add     $sp,$sp,4               #
lw      $fp,($sp)               #   4. Pop $fp
add     $sp,$sp,4               #

lw      $ra,($sp)               #   5. Pop $ra
add     $sp,$sp,4               #
jr      $ra                     #   6. return to caller
```

```
#  main()
#  {
#    int a;                    // a: 0($fp)
#    a = mysub( 6 );
#    print( a );
#  }
                .text
                .globl  main
main:
                                          # Vorspann
        sub     $sp,$sp,4                 #   1. Push return address
        sw      $ra,($sp)
        sub     $sp,$sp,4                 #   2. Push caller's frame pointer
        sw      $fp,($sp)                 #   3. kein S registers to push
        sub     $fp,$sp,4                 #   4. $fp = $sp – PlatzfuerVariable
        move    $sp,$fp                   #   5. $sp = $fp
```

```
                        # UP     Aufruf
                        #   1. kein T registers zu sichern
li      $a0,6           #   2. Argument in $a0
jal     mysub           #   3. Jump and link to subroutine
                        #  Rückkehr von UP
                        #   1. kein t Register wiederherzustellen
sw      $v0,0($fp)      # a = mysub( 6 )
                        # print a
lw      $a0,0($fp)      # lade a in $a0
li      $v0,1           # print integer service
syscall
                        # Nachspann
                        #   1. Kein Rueckkehrwert
add     $sp,$fp,4       #   2. $sp = $fp + PlatzfuerVariable
                        #   3. kein S registers  wiederherzustellen
lw      $fp,($sp)       #   4. Pop $fp
add     $sp,$sp,4       #
lw      $ra,($sp)       #   5. Pop $ra
add     $sp,$sp,4       #
jr      $ra             # Rueckkehr zu OS
```

Rekursive Funktionen

Funktionen, die sich selbst aufrufen
Def.: Eine Prozedur (Funktion) heißt rekursiv, wenn in ihrem Anweisungsteil ein Aufruf von ihr selbst steht (direkte Rekursion).
Damit die Rekursion terminiert, muß ein Rekursionsanfang gegeben sein.
(recurrere = zurücklaufen)

Beispiele: Fakultät, Fibonacci-Zahlen, ggT,
Türme von Hanoi:

Problemstellung:
Ein alter Mönch bekam die Aufgabe, den Scheibenturm von Position a nach b unter folgenden Bedingungen zu transportieren:

Es darf nur die oberste Scheibe eines Turmes genommen werden.
Es darf nie eine größere auf einer kleineren Scheibe liegen.