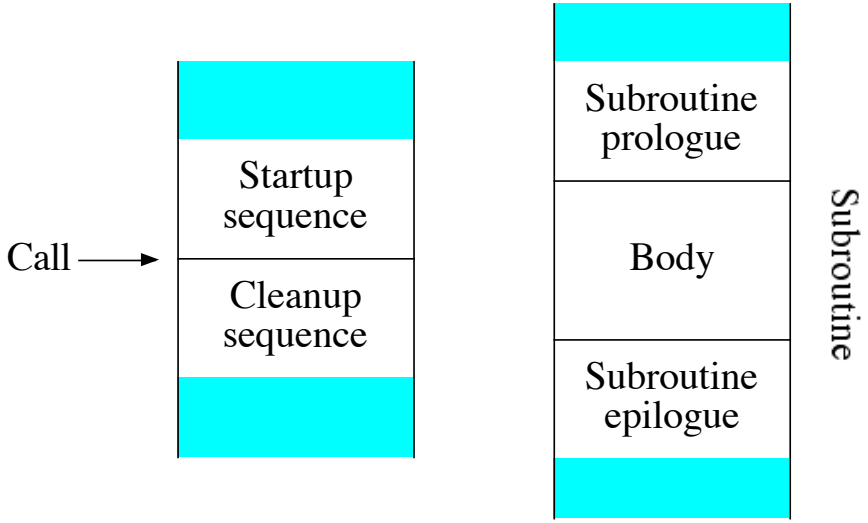
Topic 5 — Procedures in MIPS

Reading: Section 2.8, pages 112-122 (4th) or 96-106 (5th); Spim Appendix Section A.6, pages A-22 to A-33

- Overview:
  - Structure programs:
    - Make them easier to understand, and
    - Make code segments easier to re-use.
- Problems:
  - Want to call the procedure from anywhere in the code.
  - Want to pass arguments to the subroutine that may be different each time the procedure is called.
  - Want the procedure to return to the point from which it was called.
  - (May) want the procedure to return a value (technically, such a "procedure" is actually a "function").
- Issues in implementing subroutines:
  - How does the subroutine return to the caller's location?
  - Where/how is the result returned?
  - Where are the parameter(s) passed?
  - Where are the registers used (i.e., overwritten) by the subroutine saved?
  - Where does the subroutine store its local variables?

Calling Subroutines:

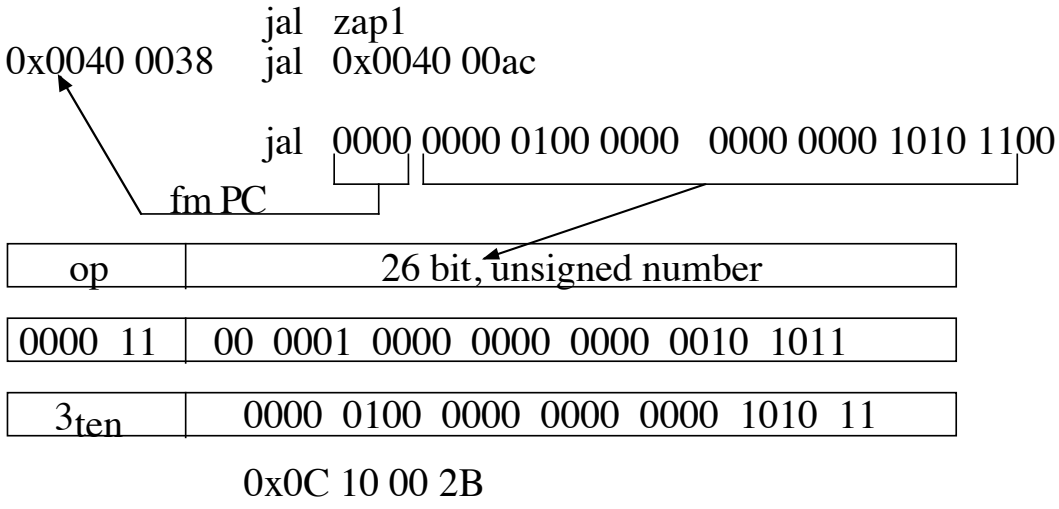
- Issues must be agreed upon by both the caller and callee in order to work.
  - Very helpful if this agreement extends across multiple high-level languages.
- Termed the *calling conventions*. Not enforced by hardware but *expected* to be followed by all programs.
- Information shared between caller and callee also termed the *subroutine linkage*.



- The caller establishes part of the subroutine linkage in the *startup sequence*.
- The callee establishes the remainder of the linkage in the *subroutine prologue*.
- The *subroutine epilogue* contains instructions that return to the caller.
- The *cleanup sequence* contains instructions to clean up the linkage.

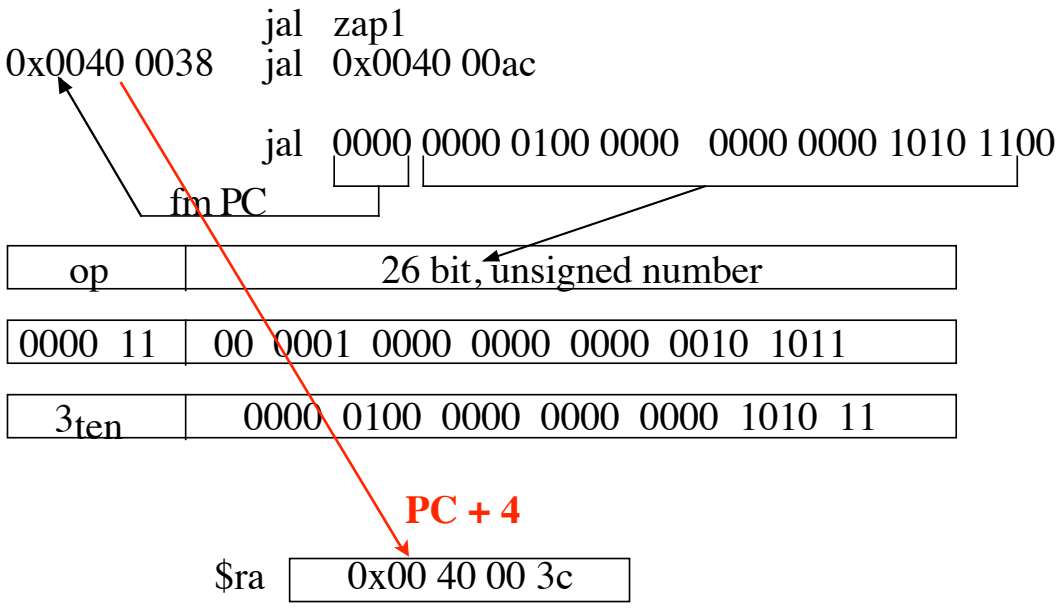
Calling and Returning:

- Two techniques have to be provided by MIPS (or *any* assembly language):
  - Calling the procedure in a way that remembers where we came from.
  - Returning to the point of the call after the procedure's work is done.
- The call is done with the jump-and-link instruction: **`jal`**.
  - Example call:
    - **`jal  zap1          # taken from funcExample1.s`**
  - Uses the J-format; the opcode is now **`3`**:



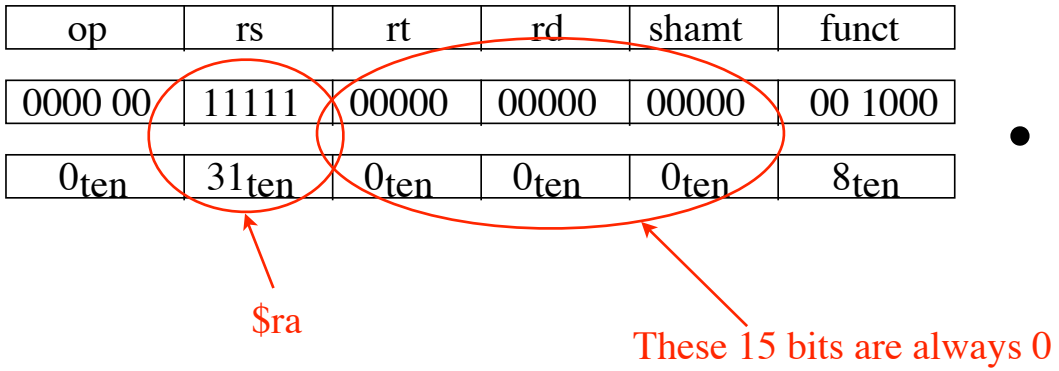
Calling and Returning (continued):

- The **`jal`** instruction does one more thing:
  - It copies the current PC into register **`$ra`** (register **`$31`**).
  - Thus, the **`jal`** from the previous slide:
    - The PC is incremented by **`4`** by the CPU while the **`jal`** instruction is being executed.
      - This increment of the PC is done by every instruction.
    - Thus, the value **`0040 0038`** + **`4`** = **`0040 003c`** is copied to **`$ra`**.



Calling and Returning (continued):

- The **`jr`**, "jump register" instruction is used to return back to the point of the call.
- When the procedure is finished, the last instruction in the procedure will be:
  - **`jr  $ra`**
- The **`jr`** instruction uses the R-Format:



- How many different **`jr`** instructions are there?

  - Any of the 32 general-purpose registers can be used with a **`jr`** command.
    - For returning from a procedure, **`$ra`** will always be the one used.
      - Because **`jal`** puts the PC+4 return address in **`$ra`**.

Registers and Parameters:

- Registers are global memory locations shared by all subroutines.
  - If a value is in a register before a **`jal`**, will it still be there after the procedure returns?
    - Sometimes "yes", sometimes "no". How does this happen?

- Two possible approaches:
  - Before a subroutine call, the caller saves to memory all the registers that it will need before the **`jal`**. OR
  - The callee saves to memory the registers that it will use, and restores all of them when done.

- MIPS compromise: Divide registers into those saved by caller (*t* registers), those saved by callee (*s* registers).
  - Done by the Caller:

    Startup sequence:

    Save the *t* registers used by the caller.

    Save the arguments sent to the subroutine.

    Store return address and jump to subroutine (**`jal`**).

    Cleanup sequence.

    Restore the *t* registers used by the caller.

  - Done by the Callee:

    Subroutine prologue:

    Save the *s* registers used in the subroutine body.

    Save the return address (**`$ra`**), if necessary.

    Subroutine epilogue:

    Restore the *s* registers saved in the prologue.

    Restore the value of **`$ra`**, if necessary.

    Return (**`jr $ra`**).

Registers and Parameters (continued):

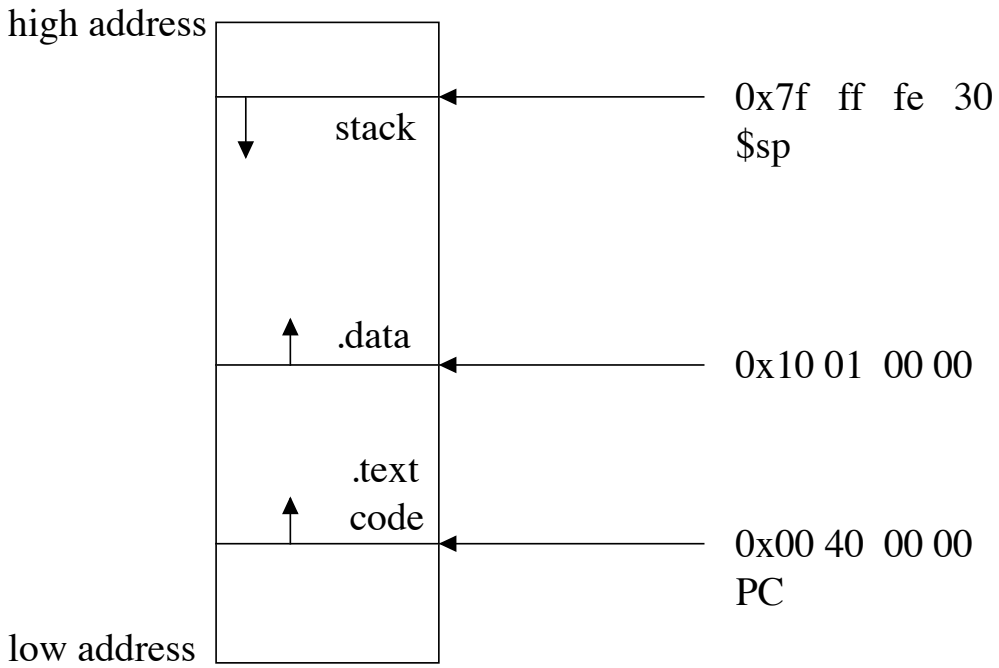
- Example: How do **`main`** and **`zap`** share **`$t3`**
  - One of them (**`main`**) must preserve the contents of **`$t3`**:

```
main:
    addi  $t3, $zero, 0
LoopBegin:
     ....
     #Startup Sequence
     save $t3 to memory
     jal    zap
     # Cleanup sequence
     load $t3 from memory
     ....
     addi $t3, $t3, 1
     j    LoopBegin
     ....
```

```
zap:
    ...
    addi $t3, $zero, -5
    ...
    jr   $ra
```

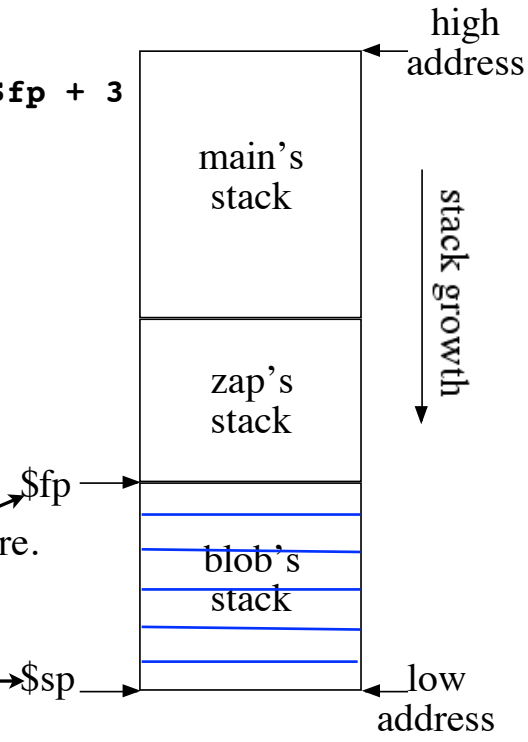
Memory Usage in SPIM:

- The memory available to a program in SPIM is divided into three regions:
  - The code section, or **`.text`** section, starts at **`0x0040 0000`** and grows to higher addresses.
  - The **`.data`** section starts at **`0x1001 0000`** and grows to higher addresses.
  - The stack section starts at **`0x7f ff fe 30`** and grows to lower addresses.
    - Current stack address stored in register **`$sp`** (register **`$29`**).



Managing the Stack:

```
address of highest byte = $fp + 3

sw  $t7, -4($sp)
sw  $a0, -8($sp)
```



- The MIPS calling conventions dictate:
  - *t* registers are saved by the caller on the caller's stack.
  - *s* registers are saved by the callee on the callee's stack.
- The stack pointer, SP, in MIPS is register **`$sp`** (**`$29`**).
- The frame pointer, FP, is **`$fp`** (**`$30`**).
  - The stack and frame pointers define limits of the stack for the current procedure.
  - The frame pointer points to the highest word in the stack frame.
  - The stack pointer points to the lowest word in the stack frame.
- Passing Parameters:
  - In general (any processor, any programming language), the parameters to a subroutine are put on the stack by the caller, and retrieved from there by the subroutine.
    - Note: if the caller has a parameter in a register, the caller must save it to the stack, then the subroutine must load it from the stack to get it back in a register.
  - MIPS optimizes this by passing the first four parameters in the registers **`$a0`** - **`$a3`**; the remainder (if more than 4) are passed on the stack.
  - Space must be reserved for *all* parameters (including those in **`$a0`** - **`$a3`**) on the stack in case the callee wants/needs to store them to memory before making calls of its own.

Managing the Stack (continued):

- Putting It All Together. Four major steps in calling a subroutine:
  - Caller executes *startup* code to set things up for the subroutine and invokes the subroutine.
  - Subroutine executes *prologue* code to manage the subroutine's stack frame.
  - Subroutine executes *epilogue* code to undo the subroutine's stack frame, then returns to the caller.
  - Caller executes *cleanup* code to clean up after the call.

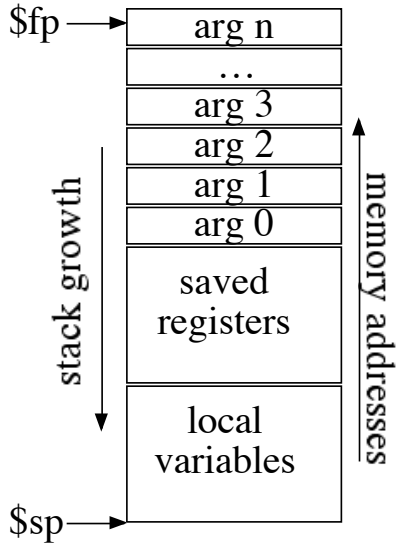
Managing the Stack (continued):

- **Startup**:
  - Save the caller-saved registers into the "saved registers" area of the current stack frame.
    - **`$t0`** - **`$t9`** registers that will be needed after the call.
    - This changes (grows) the stack of the caller.
      - Grow the stack first by subtracting from **`$sp`**.
      - Save (**`sw`**) the registers on the stack at positive offsets from **`$sp`**.
  - Pass the arguments to the subroutine:
    - The first four are placed in registers **`$a0`** - **`$a3`**.
    - The remaining arguments (if any) are put on the stack starting with the last argument first.
    - Arguments are stored by the caller at negative offsets from the stack pointer.
      - These arguments are actually being put in what will become the callee's stack.
  - Use the **`jal`** instruction to jump to the subroutine. ← The **`jal`** instruction is always the 3rd part.

```
# Startup sequence
addiu $sp, $sp, -12
sw    $t3, 8($sp)
sw    $t1, 4($sp)
sw    $t5, 0($sp)

# put arguments in $a0, $a1, ...

jal   someWhere
```

Managing the Stack (continued):

- **Prologue**:
  - Allocate a stack frame by subtracting the frame size from the stack pointer. Once set-up, a function's stack will be:
    - The stack pointer must always be word aligned, so round up the frame size to a multiple of 4.
    - The minimum frame size is **`24`** bytes (space for **`$a0`** - **`$a3`**, **`$fp`**, and **`$ra`**) and is always the minimum that must be allocated.
  - Save the callee-saved registers into the frame, including **`$fp`**.
  - Save **`$ra`** if the subroutine might call another subroutine.
  - Save any of **`$s0`** - **`$s7`** that are used by the procedure.
  - Set the frame pointer to address of the highest word (not byte) in the frame.




```
# Function prologue -- minimum amount
addiu  $sp, $sp, -24

# space for $a0, $a1, $a2, $a3
# 16 bytes

# space to save $fp, $ra
# 8 bytes

addiu  $fp, $sp, 20
```

Managing the Stack (continued):

- **Epilogue**:
  - Restore any **`$s`** registers that were saved in the prologue and **`$fp`** and **`$ra`**.
  - "Pop" the stack frame by adding the frame size to **`$sp`**.
  - Return by jumping to the address in **`$ra`**.

```
# Function prologue
addiu  $sp, $sp, -24

# space for $a0, $a1, $a2, $a3
# 16 bytes

# save $fp, $ra
# 8 bytes

addiu  $fp, $sp, 20


 # Function epilogue

 # restore $fp, $ra

 addiu  $sp, $sp, 24
 jr     $ra
```

Managing the Stack (continued):

- **Cleanup**:
  - Restore the caller-saved registers:
    - **`$t0`** - **`$t9`** registers that were saved during the Startup.
    - Shrink the caller's stack by adding to the stack pointer (**`$sp`**).
  - If the callee is returning a value in **`$v0`**, then "do something" with **`$v0`**.

```
# Startup sequence
addiu $sp, $sp, -12
sw    $t3, 8($sp)
sw    $t1, 4($sp)
sw    $t5, 0($sp)

jal   someWhere

#Cleanup sequence
lw    $t3, 8($sp)
lw    $t1, 4($sp)
lw    $t5, 0($sp)
addi  $sp, $sp, 12
# get return value from $v0
add   $t3, $t3, $v0
...

# Startup sequence
addiu $sp, $sp, -4
sw    $t2, 0($sp)

jal   bats

# Cleanup
lw    $t2, 0($sp)
addiu $sp, $sp, 4
```

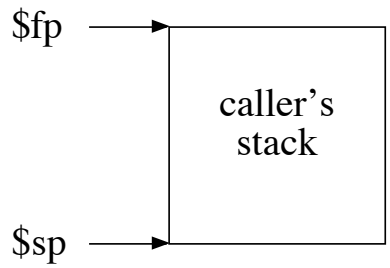
Function Call Example 1:

- Want to call a function named **`zap1`** that takes one integer as an argument, and returns an integer as its result:

```
int zap1( int x );
```

- Want to call the function **`zap1`** with **`x`** as **`15`**:

```
y = zap1( 15 );
```



- Caller's code:

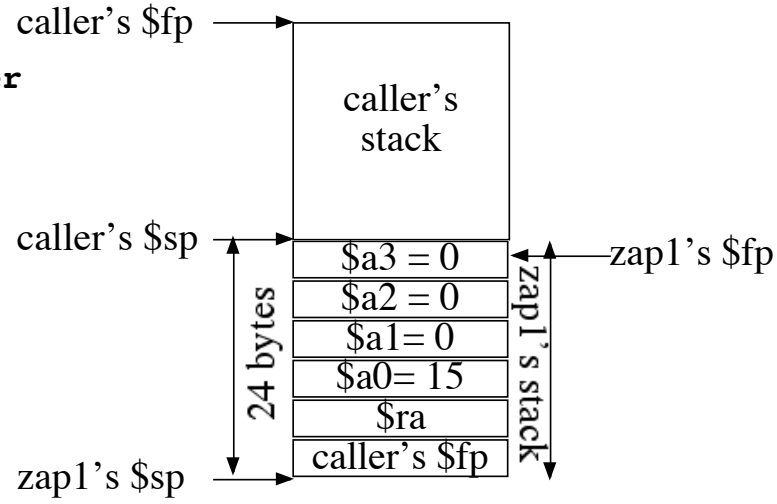
```
...

# Startup Sequence
addi   $a0, $zero, 15   # put value into $a0 to pass to zap 1
jal    zap1

# Cleanup Sequence
add    $t1, $v0, $zero  # put result of function in register $t1

...
```

- Notes:
  - The value returned by a function is put into register **`$v0`** (**`$2`**) by the procedure.
  - The code above assumes the caller does not need to save any *t* registers on the stack.
    - Simplifies both the *Startup* and *Cleanup* sequences.

Function Call Example 1 (continued):

- The Function:

```
zap1:
    # Function prologue
    addiu $sp, $sp, -24   # allocate stack space -- default of 24 here
    sw    $fp, 0($sp)     # save caller's frame pointer
    sw    $ra, 4($sp)     # save return address
    sw    $a0, 8($sp)     # save parameter value
    addiu $fp, $sp, 20    # setup zap1's frame pointer
    # $a1, $a2, $a3 not used here


    # body of zap 1 here...
    # assume:
    #    zap1 does not use s registers
    #    zap1 does not call other functions
    #    somewhere in the body, zap1 puts
    #       the return value in $v0


    # Function epilogue -- restore stack & frame pointers and return
    lw    $a0, 8($sp)     # restore original value of $a0 for caller
    lw    $ra, 4($sp)     # get return address from stack
    lw    $fp, 0($sp)     # restore the caller's frame pointer
    addiu $sp, $sp, 24    # restore the caller's stack pointer
    jr    $ra             # return to caller's code
```



Function Call Example 1 (continued):

```
.data
main1String: .asciiz  "Inside main, after call to zap1, returned value = "
zap1String:  .asciiz  "Inside function zap1, quadrupled value = "
newline:     .asciiz  "\n"

.text
main:
        # Function prologue -- even main has one
        addiu $sp, $sp, -24   # allocate stack space -- default of 24 here
        sw    $fp, 0($sp)     # save caller's frame pointer
        sw    $ra, 4($sp)     # save return address
        addiu $fp, $sp, 20    # setup main's frame pointer

        # body of main
        # call function zap1 with 15
        addi  $a0, $zero, 15
        jal   zap1

        add   $t0, $v0, $zero   # save return value in $t0

        la    $a0, main1String
        addi  $v0, $zero, 4
        syscall
        add   $a0, $t0, $zero
        addi  $v0, $zero, 1
        syscall
```

Function Call Example 1 (continued):

```
        la    $a0, newline
        addi  $v0, $zero, 4
        syscall

        # call function zap1 with 42
        addi  $a0, $zero, 42
        jal   zap1

        add   $t0, $v0, $zero   # save return value in $t0

        la    $a0, main1String
        addi  $v0, $zero, 4
        syscall
        add   $a0, $t0, $zero
        addi  $v0, $zero, 1
        syscall
        la    $a0, newline
        addi  $v0, $zero, 4
        syscall

done:   # Epilogue for main -- restore stack & frame pointers and return
        lw    $ra, 4($sp)     # get return address from stack
        lw    $fp, 0($sp)     # restore the caller's frame pointer
        addiu $sp, $sp, 24    # restore the caller's stack pointer
        jr    $ra             # return to caller's code
```

Function Call Example 1 (continued):

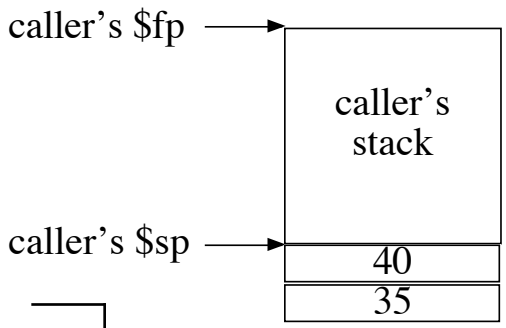
```
zap1:   # Function prologue
        addiu $sp, $sp, -24   # allocate stack space -- default of 24 here
        sw    $fp, 0($sp)     # save caller's frame pointer
        sw    $ra, 4($sp)     # save return address
        sw    $a0, 8($sp)     # save parameter value
        addiu $fp, $sp, 20    # setup zap1's frame pointer

        # something for zap to do
        add   $t0, $a0, $a0   # double the parameter
        add   $t0, $t0, $t0   # quadruple the parameter

        # print results
        la    $a0, zap1String # print the string
        addi  $v0, $zero, 4
        syscall
        add   $a0, $t0, $zero # print the quadruple'd value
        addi  $v0, $zero, 1
        syscall
        la    $a0, newline
        addi  $v0, $zero, 4
        syscall
        # put result of function in $v0
        # Note: could not do this before printing!
        add   $v0, $t0, $zero

        # Function epilogue -- restore stack & frame pointers and return
        lw    $ra, 4($sp)     # get return address from stack
        lw    $fp, 0($sp)     # restore the caller's frame pointer
        addiu $sp, $sp, 24    # restore the caller's stack pointer
        jr    $ra             # return to caller's code
```

Function Call Example 2:

- Want to call a function that takes more than four arguments:

```
int zap2( int a, int b, int c, int d, int e, int f);
```

- Need to put **a**, **b**, **c**, and **d** into **$a0** - **$a3**.
- Where to put **e** and **f**? On the stack!
  - They go on the stack of the callee, not the caller.
- Caller's code; the Caller puts **e** and **f** on the stack of the Callee's:



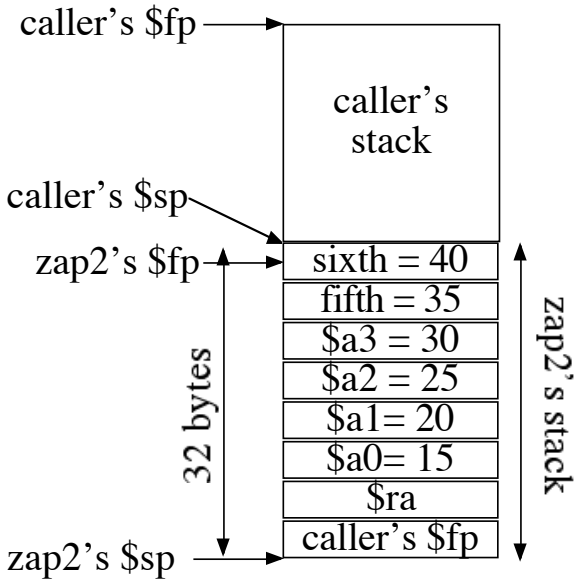

```
# add up some numbers using zap2 and print result
addi    $a0, $zero, 15   # put value into $a0 for zap2
addi    $a1, $zero, 20   # put value into $a1 for zap2
addi    $a2, $zero, 25   # put value into $a2 for zap2
addi    $a3, $zero, 30   # put value into $a3 for zap2

addi    $t0, $zero, 40
sw      $t0, -4($sp)     # put value onto stack for zap2
addi    $t1, $zero, 35
sw      $t1, -8($sp)     # put value onto stack for zap2

jal     zap2             # calling function zap2

# print result from function
add     $a0, $v0, $zero  # put result of function in register $a0
```

The order of these two sections of code can be reversed.

Function Call Example 2 (continued):

- The Function:

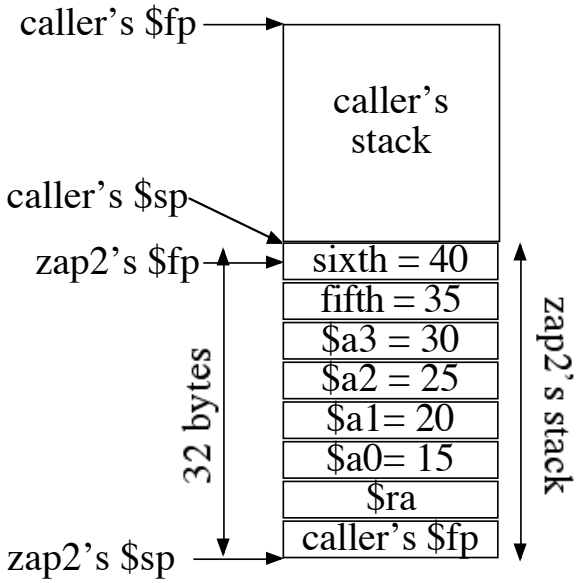
```
zap2:   # Prologue: set up stack and frame pointers for zap2
        addiu   $sp, $sp, -32     # allocate stack space -- 32 needed here
        sw      $fp, 0($sp)       # save caller's frame pointer
        sw      $ra, 4($sp)       # save return address
        # save parameter values $a0-$a3 on the stack
        sw      $a0, 8($sp)
        sw      $a1, 12($sp)
        sw      $a2, 16($sp)
        sw      $a3, 20($sp)
        addiu   $fp, $sp, 28      # setup zap2's frame pointer
        # assuming zap2 does not use any s registers or call any functions

        # add up all six values:
        add     $t0, $a0, $a1     # add $a0 + $a1
        add     $t0, $t0, $a2     # add $a2
        add     $t0, $t0, $a3     # add $a3

        lw      $t1, 24($sp)      # get 5th argument
        add     $t0, $t0, $t1     # add 5th argument
        lw      $t1, 28($sp)      # get 6th argument
        add     $t0, $t0, $t1
```



Function Call Example 2 (continued):

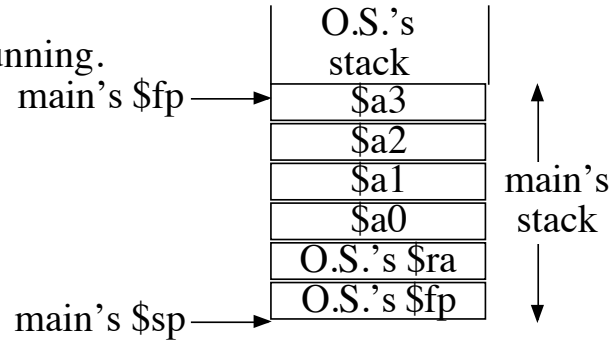
- The Function (continued):

```
# zap2 puts the return value into $v0
add     $v0, $t0, $zero

# Epilogue: restore stack and frame pointers and return
lw      $a0, 8($sp)
lw      $a1, 12($sp)
lw      $a2, 16($sp)
lw      $a3, 20($sp)
lw      $ra, 4($sp)       # get return address from stack
lw      $fp, 0($sp)       # restore the caller's frame pointer
addiu   $sp, $sp, 32      # restore the caller's stack pointer
jr      $ra               # return to caller's code
```



Function Call Example 2 (continued):

- **`main`** is a Function:
  - The "outside world" does a function call to main to start our program running.
    - "Outside world" can be the O.S., can be a command-line shell, etc.
    - Parameters can be passed to our program from the outside.
  - Have to set up **`main`**'s stack correctly.
    - First code in **`main`** will always be:



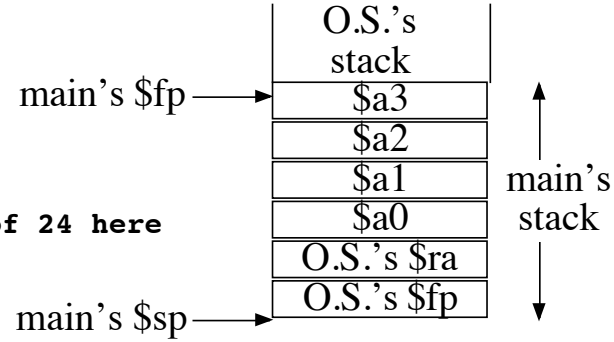

```
main:
        # Prologue: set up stack and frame pointers for main
        addiu   $sp, $sp, -24     # allocate stack space -- default of 24 here
        sw      $fp, 0($sp)       # save caller's frame pointer
        sw      $ra, 4($sp)       # save return address
        addi    $fp, $sp, 20      # setup main's frame pointer
```

    - Final code in main will always be:

```
mainDone:
        # Epilogue for main -- restore stack & frame pointers and return
        lw      $ra, 4($sp)       # get return address from stack
        lw      $fp, 0($sp)       # restore the caller's frame pointer
        addiu   $sp, $sp, 24      # restore the caller's stack pointer
        jr      $ra               # return to caller's code
```

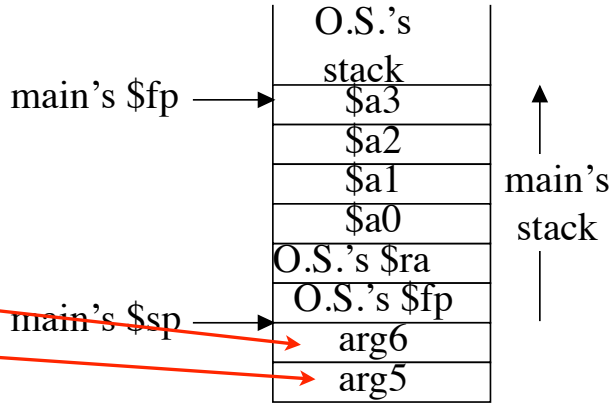
Function Call Example 2 (continued):

```
.data
str1:   .asciiz "Result of call #1 to function zap2 is "
str2:   .asciiz "Result of call #2 to function zap2 is "
nl:     .asciiz "\n\n"
.text
main:
        # Prologue: set up stack and frame pointers for main
        addiu   $sp, $sp, -24    # allocate stack space -- default of 24 here
        sw      $fp, 0($sp)      # save caller's frame pointer
        sw      $ra, 4($sp)      # save return address
        addi    $fp, $sp, 20     # setup main's frame pointer

        # add up some numbers using zap2 and print result
        addi    $a0, $zero, 15   # put value into $a0 for zap2
        addi    $a1, $zero, 20   # put value into $a1 for zap2
        addi    $a2, $zero, 25   # put value into $a2 for zap2
        addi    $a3, $zero, 30   # put value into $a3 for zap2

        addi    $t0, $zero, 40
        sw      $t0, -4($sp)     # put value onto stack for zap2
        addi    $t1, $zero, 35
        sw      $t1, -8($sp)     # put value onto stack for zap2

        jal     zap2             # calling function zap2

        # print result from function
        add     $a0, $v0, $zero  # put result of function in register $a0
        addi    $a1, $zero, 1    # indicate which result this is
        jal     printResult
```



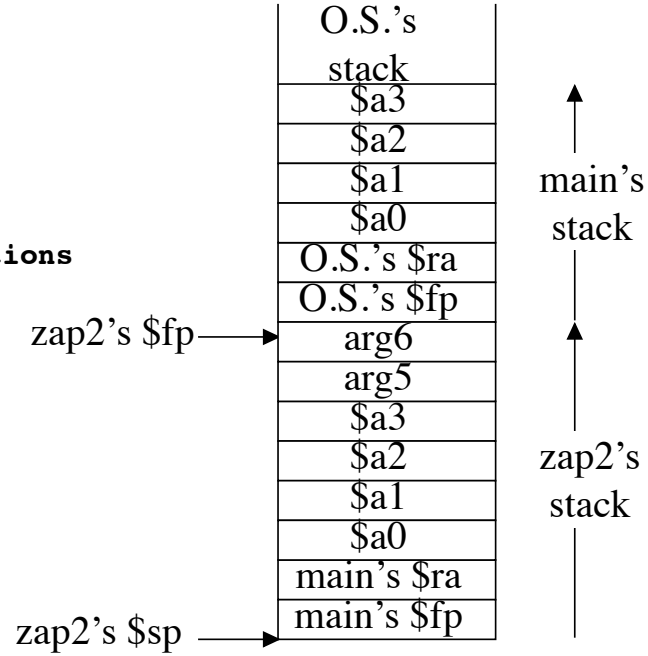
Function Call Example 2 (continued):

```
        # and, to show we can do it again...
        # add up some numbers using zap2 and print result
        addi    $a0, $zero, -15  # put value into $a0 for zap2
        addi    $a1, $zero, -20  # put value into $a1 for zap2
        addi    $a2, $zero, -25  # put value into $a2 for zap2
        addi    $a3, $zero, -30  # put value into $a3 for zap2

        addi    $t0, $zero, -40
        sw      $t0, -4($sp)     # put value onto stack for zap2
        addi    $t1, $zero, -35
        sw      $t1, -8($sp)     # put value onto stack for zap2

        jal     zap2             # calling function zap2

        # print result from function
        add     $a0, $v0, $zero  # put result of function in register $a0
        addi    $a1, $zero, 2    # indicate which result this is
        jal     printResult

mainDone:
        # Epilogue for main -- restore stack & frame pointers and return
        lw      $ra, 4($sp)      # get return address from stack
        lw      $fp, 0($sp)      # restore the caller's frame pointer
        addiu   $sp, $sp, 24     # restore the caller's stack pointer
        jr      $ra              # return to caller's code
```



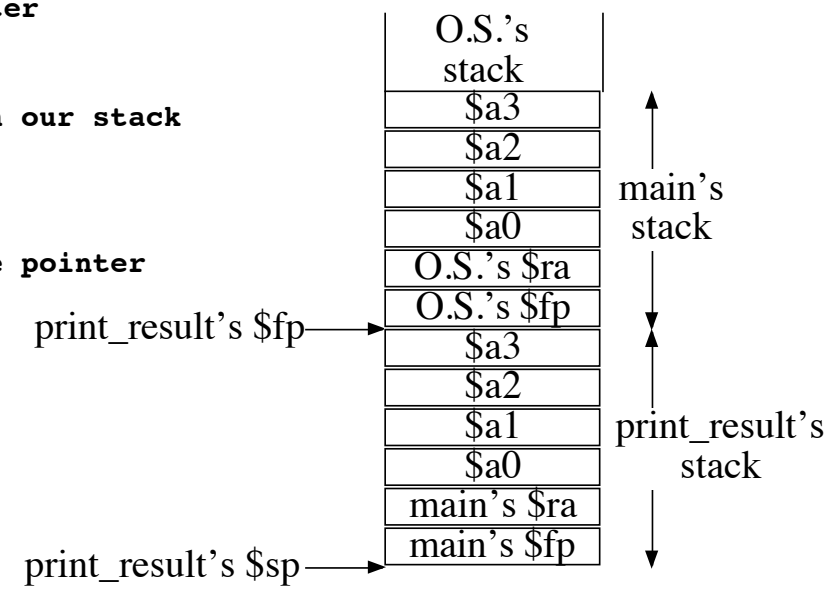
Function Call Example 2 (continued):

```
zap2:   # Prologue: set up stack and frame pointers for zap2
        addiu   $sp, $sp, -32     # allocate stack space -- 32 needed here
        sw      $fp, 0($sp)       # save caller's frame pointer
        sw      $ra, 4($sp)       # save return address
        # save parameter values $a0-$a3 on the stack
        sw      $a0, 8($sp)
        sw      $a1, 12($sp)
        sw      $a2, 16($sp)
        sw      $a3, 20($sp)
        add     $fp, $sp, 28      # setup zap2's frame pointer
        # assuming zap2 does not use any s registers or call functions

        # add up all six values:
        add     $t0, $a0, $a1     # add $a0 + $a1
        add     $t0, $t0, $a2     # add $a2
        add     $t0, $t0, $a3     # add $a3
        lw      $t1, 24($sp)      # get 5th argument
        add     $t0, $t0, $t1     # add 5th argument
        lw      $t1, 28($sp)      # get 6th argument
        add     $t0, $t0, $t1

        # zap2 puts the return value into $v0
        add     $v0, $t0, $zero

        # Epilogue: restore stack and frame pointers and return
        lw      $ra, 4($sp)       # get return address from stack
        lw      $fp, 0($sp)       # restore the caller's frame pointer
        addiu   $sp, $sp, 32      # restore the caller's stack pointer
        jr      $ra               # return to caller's code
```



Function Call Example 2 (continued):

```
printResult:
        # Function prologue
        addiu   $sp, $sp, 24     # allocate stack space -- default of 24 here
        sw      $fp, 0($sp)      # save caller's frame pointer
        sw      $ra, 4($sp)      # save return address
        # save parameter values $a0-$a1 on the stack
        # syscall's below use $a0, so must save parameter on our stack
        # can also save $a1, but not necessary...
        sw      $a0, 8($sp)
        sw      $a1, 12($sp)
        addi    $fp, $sp, 20     # setup printResult's frame pointer

        # second parameter tells us which string to print
        beq     $a1, 2, printResultSecond
        la      $a0, nl          # print some blank lines
        addi    $v0, $zero, 4
        syscall
        la      $a0, str1        # print first message
        addi    $v0, $zero, 4
        syscall
        j       printResultPrintSum


printResultSecond:
        la      $a0, str2        # print second message
        addi    $v0, $zero, 4
        syscall
```



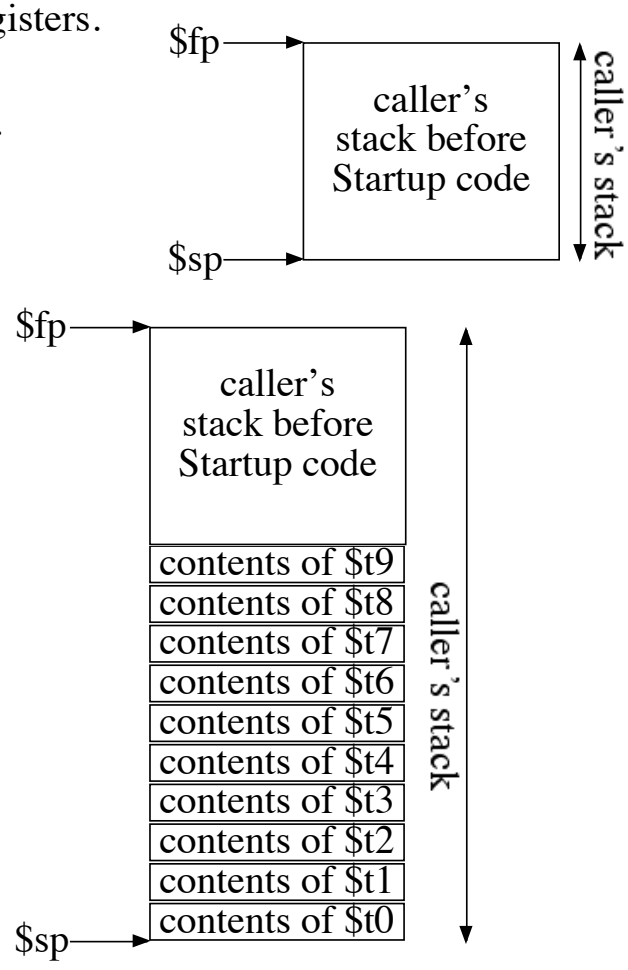
Function Call Example 2 (continued):

```
printResultPrintSum:
        lw      $a0, 8($sp)      # print the sum
        addi    $v0, $zero, 1
        syscall

        la      $a0, nl          # print the newline
        addi    $v0, $zero, 4
        syscall

        # Function epilogue -- restore stack & frame pointers and return
        lw    $a0, 8($sp)        # restore original value of $a0 for caller
        lw      $ra, 4($sp)      # get return address from stack
        lw      $fp, 0($sp)      # restore the caller's frame pointer
        addiu   $sp, $sp, 24     # restore the caller's stack pointer
        jr      $ra              # return to caller's code
```

Function Call Example 3 — Saving Registers:

- The calling function is using all of the *t* registers and needs to preserve their contents:
  - Grow the size of the caller's stack to add the contents of the *t* registers.

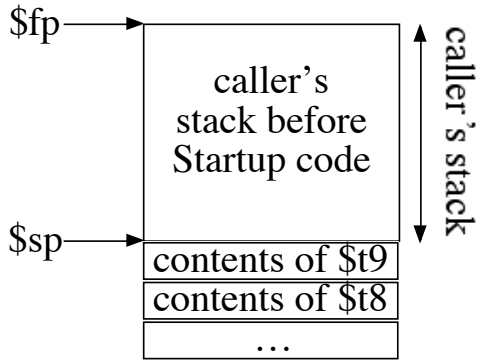
```
# Startup sequence to call function zap3
# Save t registers on stack, need 4 bytes for each
addiu $sp, $sp, -40  # make room on my stack
sw    $t9, 36($sp)
sw    $t8, 32($sp)
sw    $t7, 28($sp)
sw    $t6, 24($sp)
sw    $t5, 20($sp)
sw    $t4, 16($sp)
sw    $t3, 12($sp)
sw    $t2, 8($sp)
sw    $t1, 4($sp)
sw    $t0, 0($sp)

# Two parameters, put in $a0 and $a1
la    $s1, x
lw    $a0, 0($s1)
la    $s1, y
lw    $a1, 0($s1)

jal   zap3  # call the function
```



Function Call Example 3 — Saving Registers (continued):

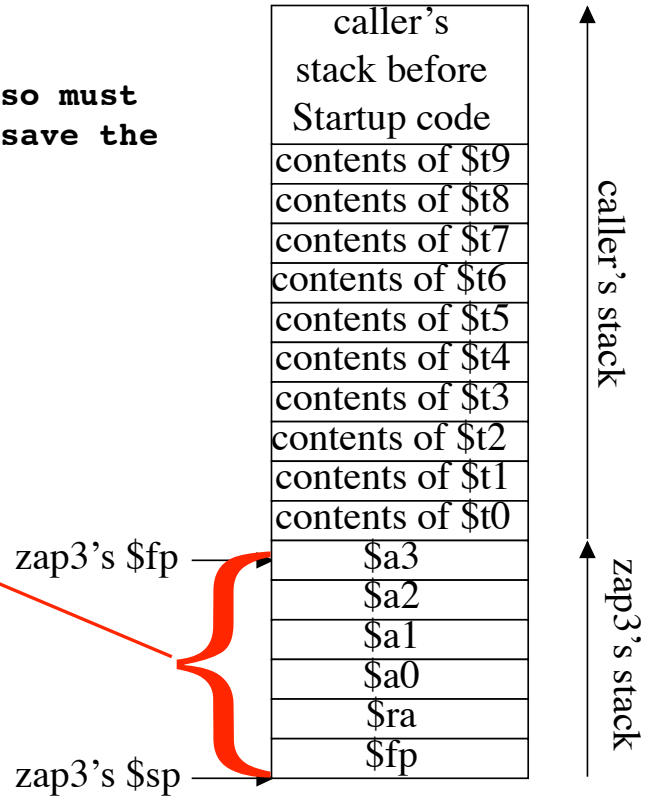
- After the function call, the Caller uses the Cleanup section to restore the contents of the *t* registers.

```
# Restore the t registers
   lw    $t9, 36($sp)
   lw    $t8, 32($sp)
   lw    $t7, 28($sp)
   lw    $t6, 24($sp)
   lw    $t5, 20($sp)
   lw    $t4, 16($sp)
   lw    $t3, 12($sp)
   lw    $t2, 8($sp)
   lw    $t1, 4($sp)
   lw    $t0, 0($sp)
   addiu $sp, $sp, 40   # Shrink stack

   #... code that follows function call
```



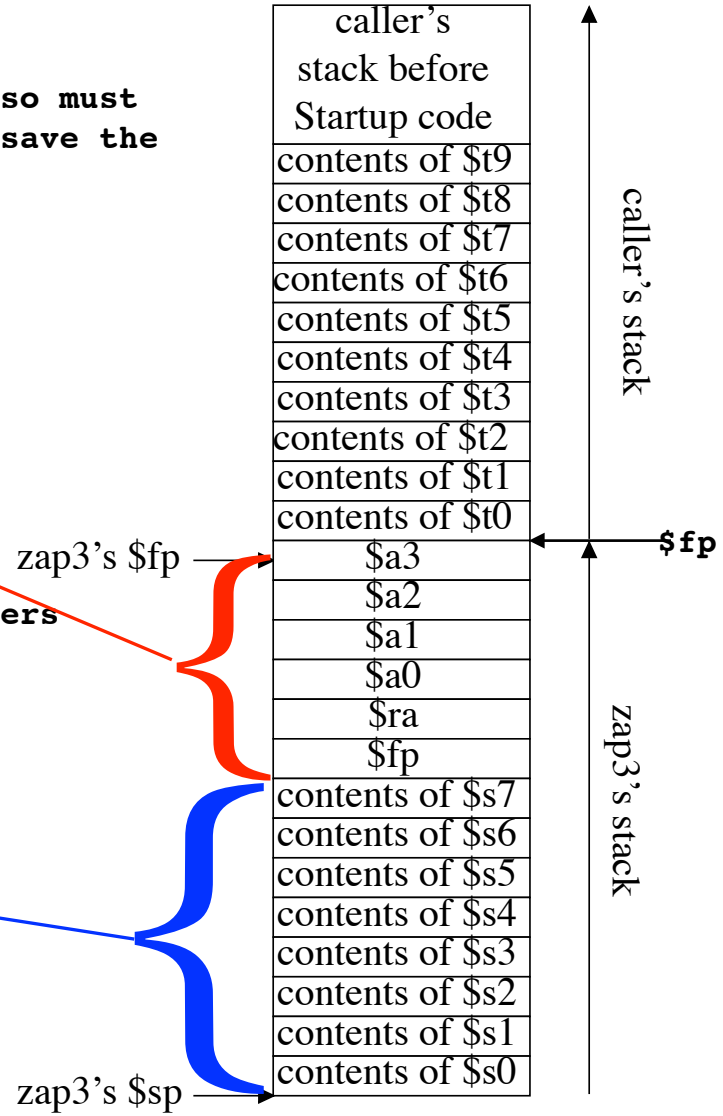
Function Call Example 3 — Saving Registers (continued):

```
zap3:    # zap3 may need to call another function, so must
         # save all the arguments on the stack and save the
         # $s registers on the stack

         # Prologue code:  Do this in two parts.
         # Start by creating the "standard" stack
         addiu $sp, $sp, -24
         sw    $a1, 12($sp)
         sw    $a0,  8($sp)
         sw    $ra,  4($sp)
         sw    $fp,  0($sp)
         addiu $fp, $sp, 20     # set zap3's $fp
```



Function Call Example 3 — Saving Registers (continued):

```
zap3:    # zap3 may need to call another function, so must
         # save all the arguments on the stack and save the
         # $s registers on the stack

         # Prologue code:  Do this in two parts.
         # Start by creating the "standard" stack
         addiu $sp, $sp, -24
         sw    $a1, 12($sp)
         sw    $a0,  8($sp)
         sw    $ra,  4($sp)
         sw    $fp,  0($sp)
         addiu $fp, $sp, 20     # set zap3's $fp

         # then, expand stack to hold the $s registers

         addiu $sp, $sp, -32
         sw    $s7, 28($sp)
         sw    $s6, 24($sp)
         sw    $s5, 20($sp)
         sw    $s4, 16($sp)
         sw    $s3, 12($sp)
         sw    $s2, 8($sp)
         sw    $s1, 4($sp)
         sw    $s0, 0($sp)
         # Note: Do NOT set $fp here; it already
         # has the correct value
```



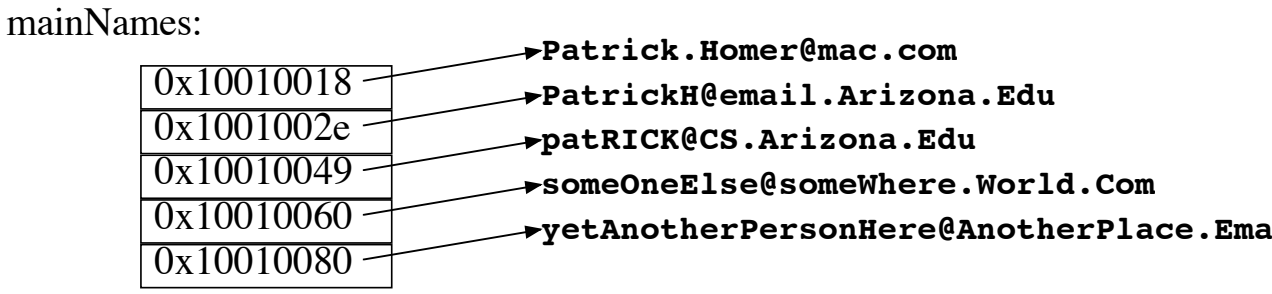
Function Call Example 3 — Saving Registers (continued):

```
# Epilogue code:
# Undo the stack in two parts. REVERSE the order of the parts.

# Restore the $s registers we wish to save

lw    $s7, 28($sp)
lw    $s6, 24($sp)
lw    $s5, 20($sp)
lw    $s4, 16($sp)
lw    $s3, 12($sp)
lw    $s2, 8($sp)
lw    $s1, 4($sp)
lw    $s0, 0($sp)
addiu $sp, $sp,  32      # shrink the lower part of the stack

# Start by creating the "standard" stack
lw    $a1, 12($sp)
lw    $a0,  8($sp)
lw    $ra,  4($sp)
lw    $fp,  0($sp)
addiu $sp, $sp, 24      # shrink the remainder of the stack
jr    $ra
```

Function Call Example 4 — Local String:

- Want to convert a string to all lower-case letters, but not modify the string used by main.
  - Pass the address of the string to the function.
  - The function copies the string to a local array of characters in the string.
- Example also shows how to use an array of strings:

```
.data
mainNumNames:
        .word   5
mainNames:
        .word   mainName1
        .word   mainName2
        .word   mainName3
        .word   mainName4
        .word   mainName5
```

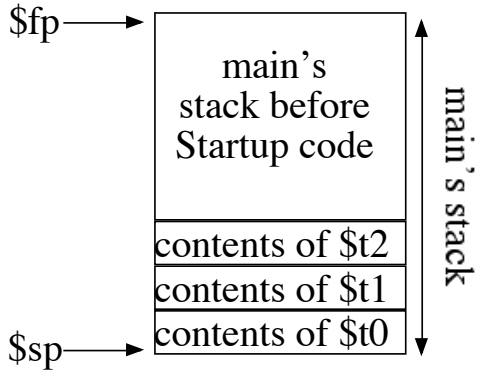



```
#                    12345678901234567890123456789012345678901234567890123456789012
mainName1: .asciiz  "Patrick.Homer@mac.com"
mainName2: .asciiz  "PatrickH@email.Arizona.Edu"
mainName3: .asciiz  "patRICK@CS.Arizona.Edu"
mainName4: .asciiz  "someOneElse@someWhere.World.Com"
mainName5: .asciiz  "yetAnotherPersonHere@AnotherPlace.Email.tv"
```

Function Call Example 4 — Local String (continued):

- Before the function call, the Caller uses the Startup section to save the contents of the *$t* registers.

```
# call convertCase
# save $t0, $t1, $t2 on the stack of main
addiu   $sp, $sp, -12    # use 12 for 3 $t registers
sw      $t2,  8($sp)
sw      $t1,  4($sp)
sw      $t0,  0($sp)

# put addr of mainNames[i] in $a0
addi    $a0, $t5, 0

jal     convertCase
```



Function Call Example 4 (continued):

```
# funcExample4.s
.data
mainNumNames:
         .word    5
mainNames:
         .word    mainName1
         .word    mainName2
         .word    mainName3
         .word    mainName4
         .word    mainName5
#                    12345678901234567890123456789012345678901234567890
mainName1: .asciiz  "Patrick.Homer@mac.com"
mainName2: .asciiz  "PatrickH@email.Arizona.Edu"
mainName3: .asciiz  "patRICK@CS.Arizona.Edu"
mainName4: .asciiz  "someOneElse@someWhere.World.Com"
mainName5: .asciiz  "yetAnotherPersonHere@AnotherPlace.Email.tv"

mainString1:
         .asciiz  "The original string:  "
mainNewLine:
         .asciiz  "\n"
.text
main:
         # Prologue: set up stack and frame pointers for main
         addiu   $sp, $sp, -24    # allocate stack space -- default of 24
         sw      $fp, 0($sp)      # save frame pointer of caller
         sw      $ra, 4($sp)      # save return address
         addi    $fp, $sp, 20     # setup frame pointer for main
```

Function Call Example 4 (continued):

```
        # for (i = 0; i < mainNumNames; i++)
        #    get contents of mainNames[i]
        #    print string that starts at address in mainNames[i]
        #    jal convertCase
        addi    $t0, $zero, 0    # $t0 = i = 0
        la      $t1, mainNumNames
        lw      $t1, 0($t1)      # $t1 = mainNumNames
        la      $t2, mainNames   # $t2 = addr of mainNames[0]
mainLoopBegin:
        slt     $t3, $t0, $t1    # $t3 = i < mainNumNames
        beq     $t3, $zero, mainLoopEnd
        la      $a0, mainString1
        addi    $v0, $zero, 4
        syscall
        # get address of mainNames[i]
        sll     $t4, $t0, 2      # $t4 = i * 4
        add     $t5, $t4, $t2    # $t5 = addr of mainNames[i]
        # get the address of the start of the string
        lw      $t5, 0($t5)
        # print the original string
        addi    $a0, $t5, 0
        addi    $v0, $zero, 4
        syscall
        la      $a0, mainNewLine
        addi    $v0, $zero, 4
        syscall
```

$t2 = 1001 0004

second iteration:
$t0 = 1

$t5 = 1001 0008

$t5 = 1001 002e

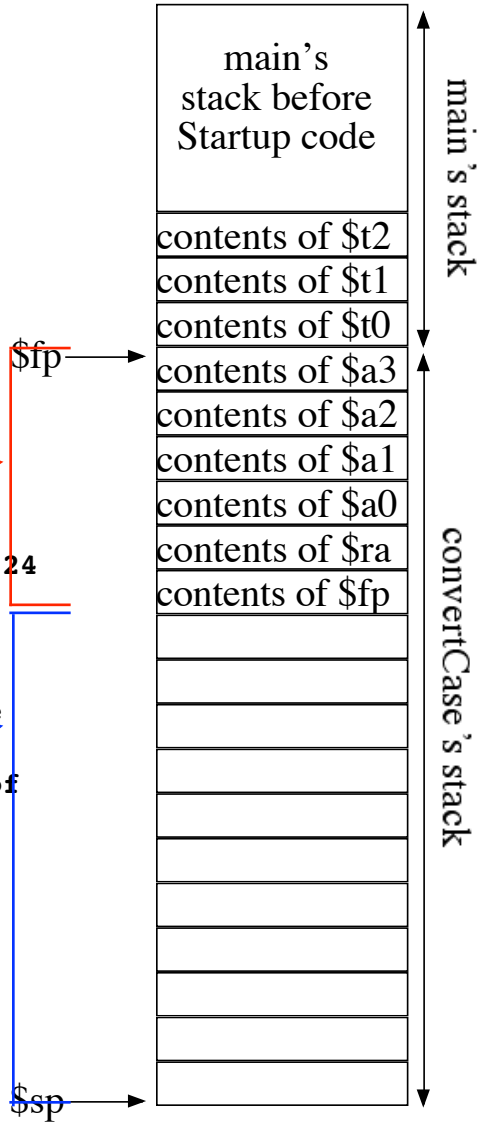
$a0 = 1001 002e

Function Call Example 4 (continued):

```
        # call convertCase
        # save $t0, $t1, $t2 on the stack of main
        addiu   $sp, $sp, -12    # use 12 for 3 $t registers
        sw      $t2,  8($sp)
        sw      $t1,  4($sp)
        sw      $t0,  0($sp)
        # put addr of mainNames[i] in $a0
        addi    $a0, $t5, 0
        jal     convertCase

        # restore $t0, $t1, $t2
        lw      $t2,  8($sp)
        lw      $t1,  4($sp)
        lw      $t0,  0($sp)
        addiu   $sp, $sp, 12

        addi    $t0, $t0, 1      # i++
        j       mainLoopBegin

mainLoopEnd:

mainDone:
        # Epilogue for main -- restore stack & frame pointers and return
        lw      $ra, 4($sp)      # get return address from stack
        lw      $fp, 0($sp)      # restore frame pointer for caller
        addiu   $sp, $sp, 24     # restore stack pointer for caller
        jr      $ra              # return to caller
```

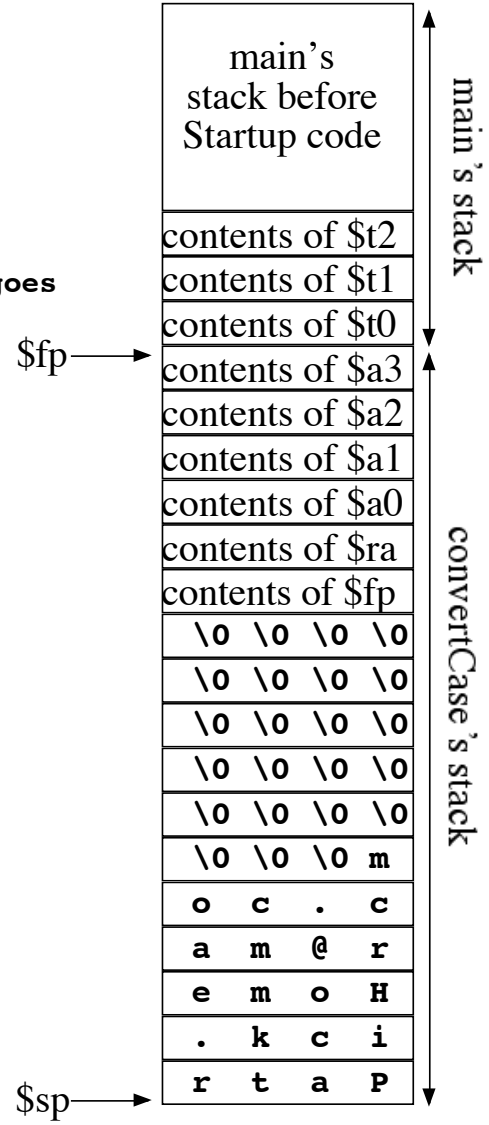
Function Call Example 4 (continued):

```
# ConvertCase procedure:
# Will copy the string to the local stack.
# Will convert upper-case letters in the string to lower-case
# Will print the converted string

.data
convertCaseString:
        .asciiz "The converted string: "
convertCaseNewLine:
        .asciiz "\n"
.text
convertCase:
        # Prologue: set up stack and frame pointers for convertCase
        addiu   $sp, $sp, -24     # allocate stack space -- default of 24
        sw      $a0, 8($sp)       # preserve $a0 in case we print
        sw      $ra, 4($sp)       # save return address
        sw      $fp, 0($sp)       # save frame pointer of caller
        addi    $fp, $sp, 20      # setup frame pointer for convertCase

        # We need additional space, 43 bytes, to hold the characters of
        # the string. Since the stack has to be word aligned, we
        # add 44 bytes to the size of the stack.
        addiu   $sp, $sp, -44
```



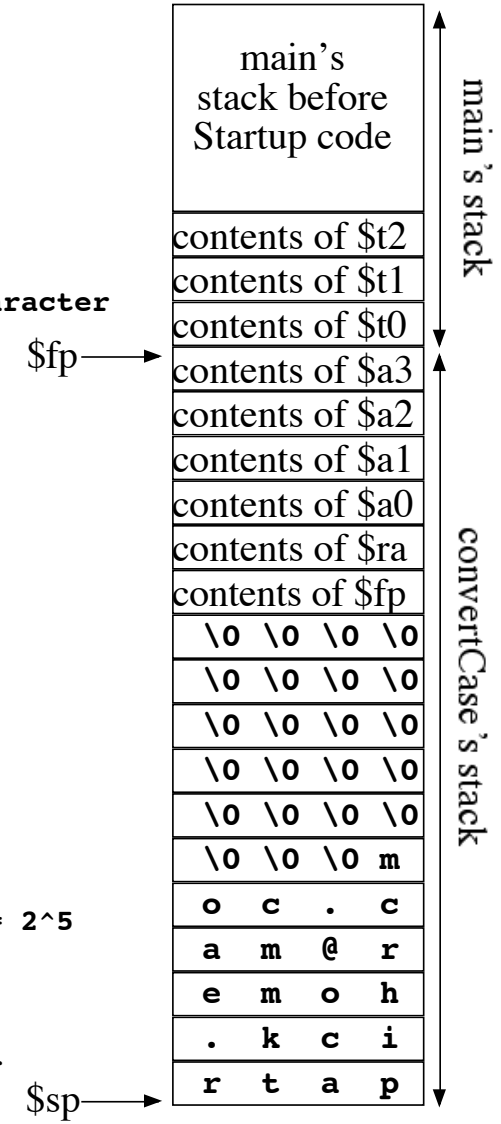
Function Call Example 4 (continued):

```
        # Copy the characters of the string onto our stack
        # while ( character != nul )
        #    copy character to our stack
        # put a nul character at the end
        addi    $t1, $sp, 0      # $t1 is where the next character goes

convertCaseCopyLoopBegin:
        lb      $t0, 0($a0)
        beq     $t0, $zero, convertCaseCopyLoopEnd
        sb      $t0, 0($t1)
        addi    $t1, $t1, 1      # $t1++
        addi    $a0, $a0, 1      # $a0++
        j       convertCaseCopyLoopBegin

convertCaseCopyLoopEnd:
        # put a nul character at the end
        sb      $zero, 0($t1)
```



Function Call Example 4 (continued):

```
        # Convert the upper-case letters to lower case
        # while ( character != nul )
        #    if ( 'A' <= character && character <= 'Z' )
        #       convert by masking with 32, which is 0x20

        addi    $t1, $sp, 0      # $t1 is where we get the next character

convertCaseConvertLoopBegin:
        lb      $t0, 0($t1)      # get the character
        beq     $t0, $zero, convertCaseConvertLoopEnd

        addi    $t2, $zero, 'A'
        addi    $t2, $t2, -1     # ascii character before 'A'
        slt     $t3, $t2, $t0    # $t3 = ('A' <= character)
        beq     $t3, $zero, convertCaseNoConvert

        addi    $t2, $zero, 'Z'  # put 'Z' into $t2
        addi    $t2, $t2, 1      # ascii character after 'Z'
        slt     $t3, $t0, $t2    # $t3 = (character <= 'Z')
        beq     $t3, $zero, convertCaseNoConvert

        # convert the case by adding a bit in position 2^5
        ori     $t0, $t0, ' '    # the char ' ' is ascii value 32 = 2^5
        sb      $t0, 0($t1)

convertCaseNoConvert:
        addi    $t1, $t1, 1      # $t1++, to get the next character
        j       convertCaseConvertLoopBegin
```



Function Call Example 4 (continued):

```
convertCaseConvertLoopEnd:
        # Print the converted string
        la      $a0, convertCaseString
        addi    $v0, $zero, 4
        syscall

        addi    $a0, $sp, 0
        addi    $v0, $zero, 4
        syscall

        la      $a0, convertCaseNewLine
        addi    $v0, $zero, 4
        syscall
        syscall                 # Why two syscall's in a row??

convertCaseDone:
        # Epilogue for convertCase -- restore stack & frame pointers

        # Remove the extra bytes used by the string
        addiu   $sp, $sp, 44

        # Now, clean up the rest of the stack
        lw      $ra, 4($sp)     # get return address from stack
        lw      $fp, 0($sp)     # restore frame pointer for caller
        addiu   $sp, $sp, 24    # restore stack pointer for caller
        jr      $ra             # return to caller
```