University of Arizona

# CSc 340

Foundations of Computer Systems

Christian Collberg

February 14, 2001

# Subroutines



## Example I

- Suppose we want to write a subroutine that converts an upper-case ASCII character to lower case. This involves adding `0x20` to the upper-case value. The subroutine will have one parameter, the upper-case character, and return the lower-case character. Here is the first attempt:

```
          .data
char:     .byte 'A'
          .text
main:     lbu $s0, char
          b tolower
finish:   li $v0, 10
          syscall           #exit
          ...
tolower:  add $s0, $s0, 0x20
          b finish
```



## Subroutine Call Problems

- Problems with this approach:
  1. The subroutine `tolower` branches to label `finish` when it is done.
  2. The subroutine tolower modifies the register `$s0`.
- Issues in implementing subroutines:
  1. How does the subroutine return to the caller?
  2. Where is the result returned?
  3. Where are parameters passed?
  4. How and where are registers used by the subroutine saved?
  5. Where does the subroutine store its local variables?



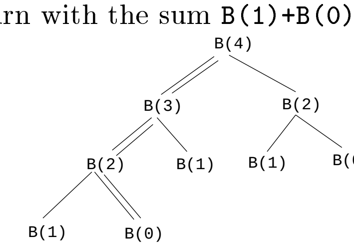
## Recursion Examples

**Example I (Factorial function):**

- $R_0$ and $R_1$ are registers that hold temporary results.

**Example II (Fibonacci function):**

- We show the status of the stack after the first call to `B(1)` has completed and the first call to `B(0)` is almost ready to return.
- The next step will be to pop `B(0)`'s AR, return to `B(2)`, and then for `B(2)` to return with the sum `B(1)+B(0)`.



## Recursion Example I

```
PROCEDURE F (
  n:INTEGER)
  :INTEGER;
  VAR L:INTEGER;
BEGIN
(1) IF n <= 1
(2) THEN L:=1;
(3) ELSE
(4)   R0:=F(n-1);
(5)   R1:=n;
(6)   L:=R0 * R1;
(7) ENDIF;
(8) RETURN L;
END F;
BEGIN
(9) C:=F(3);
(10)
END
```

| Stack | Frame |
|---|---|
| $n = 1$ | F(1) |
| $L = 1$ | F(1) |
| RetAddr=(5) | F(1) |
| RetVal=1 | F(1) |
| $n = 2$ | F(2) |
| $L = ?$ | F(2) |
| RetAddr=(5) | F(2) |
| RetVal=? | F(2) |
| $n = 3$ | F(3) |
| $L = ?$ | F(3) |
| RetAddr=(10) | F(3) |
| RetVal=? | F(3) |
| $C = ?$ | main |

**Slide 10–4**

## Recursion Example II

```
PROCEDURE B (
 n:INTEGER)
 :INTEGER;
 VAR L:INTEGER;
BEGIN
(1)  IF n <= 1
(2)  THEN L:=1;
(3)  ELSE
(4)    R0:=B(n-1);
(5)    R1:=B(n-2);
(6)    L:=R0 + R1
(7)  ENDIF;
(8)  RETURN L;
END B;
BEGIN
(9)  C:=B(4);
(10)
END
```

| Stack | Frame |
|---|---|
| $n = 1; L = 1$ | B(0) |
| RetAddr=(6) | B(0) |
| RetVal=1 | B(0) |
| n=2; L=?; $R_0$=1 | B(2) |
| RetAddr=(5) | B(2) |
| RetVal=? | B(2) |
| $n = 3; L = ?$ | B(3) |
| RetAddr=(5) | B(3) |
| RetVal=? | B(3) |
| $n = 4; L = ?$ | B(4) |
| RetAddr=(10) | B(4) |
| RetVal=? | B(4) |
| $C = ?$ | main |

**Slide 10–5**

## Calling Conventions

- These issues have to be agreed upon by both the caller and the callee (subroutine), otherwise the subroutine will not work properly. Imagine that the caller puts the first parameter in `$s0`, but the subroutine expects it in `$s1`. The routines may be written by different people, making it important to agree on calling conventions for subroutines.
- Calling conventions are not enforced by the hardware, but don't make up your own for the programming assignments. Use the following calling conventions also described in the SPIM handout.
- The alternative register names (e.g. `$s0`) are primarily to help you out with remembering the calling conventions.

**Slide 10–6**

## Returning to the caller

- The MIPS supports subroutines via the "and link" instructions. These instructions not only change the PC, they also store the PC of the next instruction in `$r31` (`$ra`). Most commonly subroutines are invoked using *jump-and-link* ⌜`jal`⌝. A jump is similar to an unconditional branch except the argument is a 26-bit word-aligned address, instead of a 20-bit offset, allowing for a bigger change in the PC.
- To return to the caller the subroutine invokes ⌜`jr $ra`⌝, which is a standard (non-linking) jump to the address stored in register `$ra`.
- The MIPS calling convention is that the result of a subroutine (if any) is returned in `$v0`.

**Slide 10–7**

Slide 10–8

## Saving registers

- The MIPS calling conventions dictate that some registers are saved by the caller, and some by the callee. In particular, `"t"` registers that are in use must be saved by the caller, and `"s"` registers that will be used must be saved by the callee.
- Where should the caller store the `"s"` registers? It could allocate a static region to hold them. Why won't this work?

```
          .data
S:        .word 0,0,0,0,0,0,0,0
foo:      sw $s0, S + 0
          sw $s1, S + 4
          ...
```

Slide 10–9

## The stack

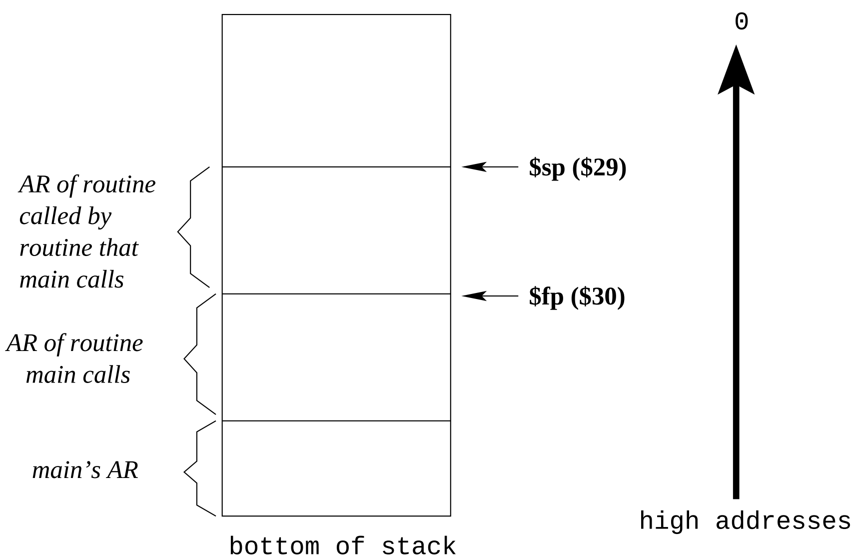
- The registers must be saved in dynamically allocated memory, but notice that if subroutine `A` calls `B`, then `A` cannot return before `B` does (in most programming languages). This means that the last memory to be allocated is the first to be freed, i.e. allocation is LIFO. The data structure to use for this is a stack.
- Think of a stack of dishes – the last one put on the stack is the first one to be used.

Slide 10–10

## The stack...

- When a subroutine is called it will allocate memory on the stack to use to store registers (and for other things described shortly), and when the subroutine returns it will put the stack back the way it found it.
- The stack grows from high addresses towards low addresses. The stack pointer contains the address of the current top of the stack. On the MIPS this is register `$r29` (`$sp`). The stack pointer on the MIPS must always be double-word aligned. A subroutine allocates memory on the stack by subtracting the amount from `$sp`, then adding it back when it returns.

Slide 10–11



Slide 10–12

## The Stack Frame

- The memory a subroutine allocates from the stack is called its stack frame.
- The stack pointer points to the first word of the frame (lowest address).
- The frame pointer (`$r30` aka `$fp`) points to the last word (highest address) of the frame.
- The frame pointer is usually used to access the contents of a frame because the subroutine may change the stack pointer while it runs.

Slide 10–13

## The Stack Frame

- What does a subroutine store in its stack frame?
  1. Callee-saved registers
  2. Previous frame pointer
  3. Return address (if the subroutine calls another subroutine)
  4. Local variables (that don't fit in registers)
  5. Caller-saved registers (if the subroutine calls another subroutine)

Slide 10–14

## Passing parameters

- In general the parameters to a subroutine are pushed on the stack by the caller, and loaded from there by the subroutine.
- Notice that if the caller has a parameter in a register it must store it to the stack, then the subroutine must load it from the stack to get it back in a register.
- The MIPS optimizes this by passing the first four parameters in the registers `$a0-$a3`. If the subroutine has more than four parameters the rest are passed on the stack.

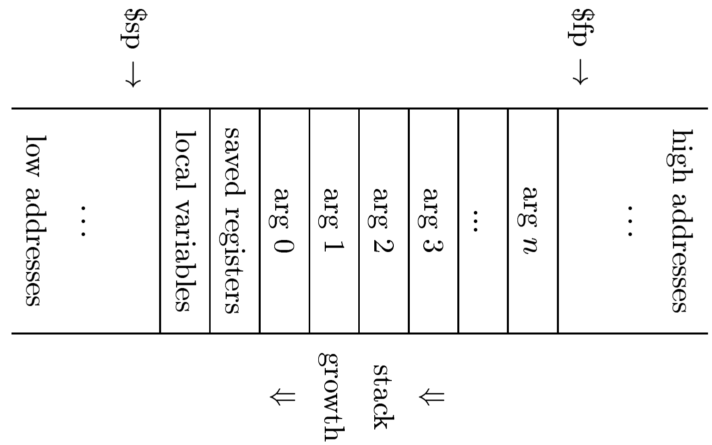
Slide 10–15

## Passing parameters

- Even though the first four parameters are passed in registers, space must be reserved for them on the stack (in case the callee wants to store them to memory).
- If the subroutine has local variables it reserves space for them in its stack frame.

Slide 10–16

## Local variables




Slide 10–17

## Procedure call steps

There are four major steps in calling a subroutine:

**Setup** the caller executes setup code to set things up for the subroutine, and invokes the subroutine,

**Prologue** the subroutine executes prologue code to manage the stack frame,

**Epilogue** before the subroutine returns it executes epilogue code to undo the stack frame, and return to the caller, and

**Cleanup** the caller executes cleanup code to clean up.

Slide 10–18

## Procedure call steps – Setup

1. Pass the arguments to the subroutine. The first four are in registers `$a0-$a3`, the rest are put on the stack starting with the last argument first. Arguments are stored at negative offsets from the stack pointer.
2. Save the caller-saved registers, including `$t0-$t9` into the "saved registers" area of the current stack frame.
3. Use the `jal` instruction to jump to the subroutine.

Slide 10–19

## Procedure call steps – Prologue

1. Allocate a stack frame by subtracting the frame size from `$sp`.
2. Remember the stack pointer must always be double-word aligned, so round the frame size up to a multiple of 8.
3. The minimum frame size is 24 bytes (space for `$a0-$a3` and $ra is always allocated).
4. Save the callee-saved registers into the frame, including `$fp`, `$ra` if the subroutine calls another subroutine, and any of `$s0-$s7` that are used.
5. Set the frame pointer to `$sp` plus the frame size.

## Procedure call steps – Epilogue

1. Restore any registers that were saved in the prologue, including `$fp`.
2. Pop the stack frame by adding the frame size to $sp.
3. Return by jumping to the address in $ra.

**Slide 10–20**

## Procedure call steps – Cleanup

1. Restore any caller-saved registers.

**Slide 10–21**

## Example I

- Compute 10! (10 factorial). The C code is:

```
main() {
   print_str("The answer is:  ");
   print_int(fact(10));
}

int fact(int n) {
   if (n < 1)
      return 1;
   else
      return (n * fact(n-1));
}
```

**Slide 10–22**

## Example I – Notes

1. main will have to save `$fp` and `$ra` because it calls a subroutine, so the stack frame needs room for 6 registers or 24 bytes (no local variables are used).
2. fact will also have to save `$fp` and `$ra` so its stack frame will also be 24 bytes.
3. The format of each stack frame will be:

| | | |
|---|---|---|
| $fp → | | |
| -4 | arg 3 | 20 |
| -8 | arg 2 | 16 |
| -12 | arg 1 | 12 |
| -16 | arg 0 | 8 |
| -20 | ra | 4 |
| -24 | fp | 0 |
| | | ← $sp |

**Slide 10–23**

Slide 10–24

## MIPS Factorial

```
         .data
msg:     .asciiz "The answer is: "
         .text
         .globl main
main:    subu    $sp,$sp, 24      # Allocate stack frame
         sw      $fp, 0($sp)      # Save $fp in frame
         addu    $fp, $sp, 24     # Set up $fp
         sw      $ra, -20($fp)    # Save $ra
         la      $a0, msg         # Arg 0 = msg
         li      $v0, 4           # print_str
         syscall
```

Slide 10–25

## MIPS Factorial. . .

```
         li      $a0, 10          # Arg 0 = 10
         jal     fact
         move    $a0, $v0         # Arg 0 = result
         li      $v0, 1           # print_int
         syscall
         lw      $ra, -20($fp)    # Restore $ra
         lw      $fp, -24($fp)    # Restore $fp
         addu    $sp, $sp, 24     # Pop stack frame
         jr      $ra
```

Slide 10–26

```
fact:     subu   $sp, $sp, 24        # Allocate stack frame
          sw     $fp, 0($sp)         # Save $fp
          addu   $fp, $sp, 24        # Set up $fp
          sw     $ra, -20($fp)       # Save return address
          sw     $a0, -16($fp)       # Save arg 0 (n)
          bgtz   $a0, recurse        # if (n == 0) then
          li     $v0, 1              #    return 1
          b      done                # else
recurse:  subu   $a0, $a0, 1         #    n = n - 1
          jal    fact                #    fact(n-1)
          lw     $t0, -16($fp)       #    load n
          mul    $v0, $v0, $t0       #    compute fact(n-1)*n
done:     lw     $ra, -20($fp)       # restore $ra
          lw     $fp, -24($fp)       # restore $fp
          addu   $sp, $sp, 24        # pop stack frame
          jr      $ra
```

Slide 10–27

## Example 2

Write a routine that computes the (integral) distance

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}.$$

between two points $(x_1, y_1, z_1)$ and $(x_2, y_2, z_2)$ in 3-space.

```
int distance(x1,y1,z1,x2,y2,z2)
   int x1,y1,z1,x2,y2,z2;{
   int xdiff, ydiff, zdiff, dist;
   xdiff = x2-x1; ydiff = y2-y1; zdiff = z2-z1;
   dist = sqrt(xdiff*xdiff + ydiff*ydiff + zdiff*zdiff);
   return dist;
}
```

## Example II

- Assume that the routine **sqrt** takes an integer parameter and returns an integer result of the square root computation.
- I could have done this all on one line, but chose not to because this format will help illustrate the MIPS code.
- First, figure out the format of the stack frame. Although all temporaries could be stored in registers, I'll show **xdiff**, **ydiff**, and **zdiff** as being stored on the stack.

Slide 10–28

## The Stack Frame

| $fp offset | | $sp offset |
|---|---|---|
| $fp → | | |
| -4 | arg 5 (z2) | 44 |
| -8 | arg 4 (y2) | 40 |
| -12 | arg 3 (x2) | 36 |
| -16 | arg 2 (z1) | 32 |
| -20 | arg 1 (y1) | 28 |
| -24 | arg 0 (x1) | 24 |
| -28 | ra | 20 |
| -32 | fp | 16 |
| -36 | zdiff | 12 |
| -40 | ydiff | 8 |
| -44 | xdiff | 4 |
| -48 | | 0 ← $sp |

Slide 10–29

- The code that calls distance looks like:

```
lw      $t0, z2
sw      $t0, -4($sp)        # push z2 on stack
lw      $t0, y2
sw      $t0, -8($sp)        # push y2 on stack
lw      $a3, x2             # pass x2 in a3
lw      $a2, z1             # pass z1 in a2
lw      $a1, y1             # pass y1 in a1
lw      $a0, x1             # pass x1 in a0
jal distance
move    $t0, $v0            # get result from $v0
```

- The code for the distance routine itself is on the next slide.

Slide 10–30

## MIPS `distance` – Create Frame

```
distance:
   subu       $sp,$sp, 48        # Push stack frame
   sw         $fp, 16($sp)       # Save $fp in frame
   addu       $fp, $sp, 48       # Set up $fp
   sw         $ra, -28($fp)      # Save $ra
```

Slide 10–31

## MIPS distance – Compute

```
sub     $t0, $a3, $a0     # xdiff = x2-x1
sw      $t0, -44($fp)     # store xdiff
lw      $t1, -8($fp)      # load y2
sub     $t0, $t1, $a1     # ydiff = y2-y1
sw      $t0, -40($fp)     # store ydiff
lw      $t1, -4($fp)      # load z2
sub     $t0, $t1,$a2      # zdiff = z2-z1
sw      $t0, -36($fp)     # store zdiff
lw      $t0, -44($fp)     # load xdiff
mul     $t1, $t0, $t0     # square it
lw      $t0, -40($fp)     # load ydiff
mul     $t0, $t0, $t0     # square it
addu    $t1, $t1, $t0     # add to sum
lw      $t0, -36($fp)     # load zdiff
mul     $t0, $t0, $t0     # square it
addu    $t1, $t1, $t0     # add to sum
move    $a0, $t1          # pass sum arg
jal     sqrt
```

**Slide 10–32**

## MIPS distance – Return

```
lw      $ra, -28($fp)     # restore $ra
lw      $fp, -32($fp)     # restore $fp
addu    $sp, $sp, 48      # pop stack frame
jr      $ra
```

**Slide 10–33**

## Readings and References

- Chapter 6 in Maccabe.

**Slide 10–34**