Implementing Procedure Calls

February 18–22, 2013

Outline

- Intro to procedure calls
  - Caller vs. callee
  - Procedure call basics
- Calling conventions
- The stack
  - Interacting with the stack
  - Structure of a stack frame
- Subroutine linkage

What is a procedure?

Procedure – a reusable chunk of code in your program

- used to do the same thing in different places (reuse)
- used to logically organize your program (decomposition)
- like a method in Java, or a procedure/function in C

Can make a distinction between:

- **procedure** – does not return a result
- **function** – does return a result

(but don't worry too much about that)

Procedures can call other procedures

- including themselves! (recursion)

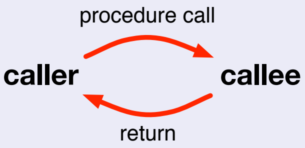
What happens when you call a procedure?

Caller vs. callee

- **caller** the code that calls the procedure
- **callee** the code that implements the procedure

Procedure call – high-level view

1. caller **calls** callee
   - caller stops executing
   - control is passed to callee
2. callee does its thing
3. callee **returns** to the caller
   - callee stops executing
   - caller resumes executing from the place of the call



Calling and returning from a procedure

To call a procedure: `jal label`

`jal` – "jump and link"

1. sets `$ra` to PC+4 (`$ra` – "return address")
   - save the address of the next instruction of the caller
2. sets PC to `label`
   - jump to the address of the first instruction of the callee

To return from a procedure: `jr $ra`

`jr` – "jump to register"

- jumps back to the next instruction of the caller

(MARS demo: ProcJoke.asm)

Arguments and return values

By convention ...

- put first four arguments to procedure in `$a0` – `$a3`
- put return value(s) in `$v0` and `$v1`

Note: this is a **very incomplete** picture!

Our view so far only works when ...

- four or fewer arguments
- every procedure is a **leaf**
    - i.e. it doesn't call any other procedures

Arguments and return values

Procedure definition

```
# Pseudocode:
#   int sumOfSquares(int a, int b) {
#     return a*a + b*b
#   }
# Registers: a => $a0, b => $a1, res => $v0
sumOfSquares:
  mult $t0, $a0, $a0  # tmp1 = a*a
  mult $t1, $a1, $a1  # tmp2 = b*b
  add  $v0, $t0, $t1  # res = tmp1 + tmp2
  jr   $ra            # return res
```

Procedure use

```
# Pseudocode:
#   c = sumOfSquares(3,5)
# Registers: c => $t2
  li   $a0, 3         # (set up arguments)
  li   $a1, 5
  jal  sumOfSquares   # (call procedure)
  move $t2, $v0       # (get result)
```

Outline

Intro to procedure calls
- Caller vs. callee
- Procedure call basics

Calling conventions

The stack
- Interacting with the stack
- Structure of a stack frame

Subroutine linkage

The need for calling conventions (pt. 1)

What's wrong with this code?

```
# Pseudocode:
#   c = sumOfSquares(x,y)
#   c = c - x
# Registers: x => $t0, y => $t1, c => $t2
  move $a0, $t0        # (set up arguments)
  move $a1, $t1
  jal  sumOfSquares    # (call procedure)
  move $t2, $v0        # (get result)
  sub  $t2, $t2, $t0   # c = c - x

# Pseudocode:
#   int sumOfSquares(int a, int b) {
#     return a*a + b*b
#   }
# Registers: a => $a0, b => $a1, res => $v0
sumOfSquares:
  mult $t0, $a0, $a0   # tmp1 = a*a
  mult $t1, $a1, $a1   # tmp2 = b*b
  add  $v0, $t0, $t1   # res = tmp1 + tmp2
  jr   $ra             # return res
```

`sumOfSquares` changed **`$t0`**!

Whose job is it to preserve it?

(Caller or callee?)

The need for calling conventions (pt. 2)

What's wrong with this code?

```
# Pseudocode:
# void question() {
#   print(quest)
#   waitForGiveUp()
#   return
# }
question:
    li $v0, 4          # print(quest)
    la $a0, quest
    syscall

    jal waitForGiveUp  # waitForGiveUp()

    jr $ra             # return

# Pseudocode:
# void waitForGiveUp() { ... }
waitForGiveUp:
    ...
    jr $ra             # return
```

`jal` changes `$ra`!

Whose job is it to preserve it?

(Caller or callee?)

Summary of issues that need to be agreed on

How do we pass data to/from procedures?

- partial solution:
  - put arguments `$a0` – `$a3`
  - put results `$v0` and `$v1`
- what about more arguments?

Registers are “global” variables

- are the values we need after the procedure call still there?
- is `$ra` correct after calling another procedure?

The data segment is also “global” memory

- what if a procedure needs its own space in memory?
  - i.e. local variables!
- can’t just declare a global space for it because of recursion

What are calling conventions?

A set of conventions that programmers follow

- to ensure their code is well-behaved
- so that it can cooperate with code written by others

Calling conventions answer the following questions:

- how do we pass data to/from procedures?
- what are the responsibilities of the caller?
- what are the responsibilities of the callee?
- where do we store variables local to a procedure?

None of this is **implemented** in MIPS!

There are multiple conventions to choose from

(we'll be using the most common)

Who is responsible for saving which registers?

Number	Name	Usage	Preserved?
`$0`	`$zero`	constant `0x00000000`	N/A
`$1`	`$at`	assembler temporary	N/A
`$2–$3`	`$v0–$v1`	function return values	✗
`$4–$7`	`$a0–$a3`	function arguments	✗
`$8–$15`	`$t0–$t7`	temporaries	✗
`$16–$23`	`$s0–$s7`	saved temporaries	✓
`$24–$25`	`$t8–$t9`	more temporaries	✗
`$26–$27`	`$k0–$k1`	reserved for OS kernel	N/A
`$28`	`$gp`	global pointer	✓
`$29`	`$sp`	stack pointer	✓
`$30`	`$fp`	frame pointer	✓
`$31`	`$ra`	return address	✓

✗ = **caller** is responsible ✓ = **callee** is responsible

Outline

Motivating the stack

When we need to save a register, where do we put it?

- a variable in the data segment?
- in another register?

What happens when we call another procedure? and another?

These places are not **extensible**

Overview of the stack

The stack

- a place in memory
- composed of **stack frames**
- each frame stores stuff specific to one procedure call
- each call can generate a new stack frame
    - stack is extensible!

Note that the stack may contain many frames for the same procedure if it is called multiple times!

Overview of a stack frame

Things we can store in a stack frame

- additional arguments to a procedure
- the values of saved registers
- the value of `$ra`
- local variables (e.g. local strings and arrays)

Gory details on stack frames later!

Calling conventions dictate:

- how to manage the stack
- how to structure a stack frame

How the stack works

LIFO – Last In, First Out

- at start of a procedure **push** a new stack frame
- at end of a procedure **pop** that stack frame

Frame of current procedure is always at the "top" of the stack

Analogy: a stack of scratch paper

- can only write on the top piece of paper
- at start of procedure, put a new piece of paper on top
- at end of procedure, throw the paper away

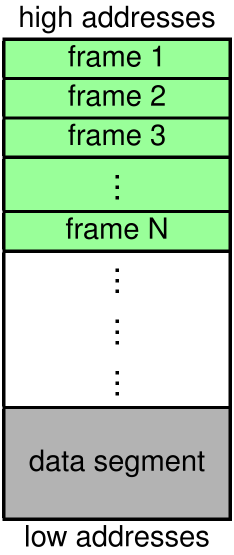
The stack in memory

"The stack" is just a region of memory

- text segment: program machine code
- data segment: constants and global vars
- the stack: supports procedure calls
  - local vars, arg passing, register backup

Memory layout

- data and stack share an address space
- stack starts at highest address
- data segment starts at lowest address
- stack grows "downward"
  - top of stack is at the "bottom"



How to use the stack in assembly

The stack pointer – register `$sp`

- contains the address of the top of the stack
- OS initializes `$sp` when your program is loaded
- after that, it is your responsibility!

Push stack frame

```
myProcedure:                # start of procedure
    addiu $sp, $sp, -24     # allocate 6 words on the stack
```

```
    ...                     # procedure body
```

Pop stack frame

```
    addiu $sp, $sp, 24      # deallocate 6 words on the stack
    jr    $ra               # return
```

How to use the stack in assembly

Reading and writing to the stack

- just like reading and writing to the data segment!
- e.g. use **sw** to write, **lw** to read

Example stack usage

```
# Registers: myVar => $t0
myProcedure:                # start of procedure
    addiu $sp, $sp, -24 # push a new stack frame (6 words)
    sw    $ra, 20($sp)  # save return address
    ...
    sw    $t0, 16($sp)  # save myVar
    jal   subProcedure  # call sub-procedure
    lw    $t0, 16($sp)  # restore myVar
    ...
    lw    $ra, 20($sp)  # restore return address
    addiu $sp, $sp, 24  # pop stack frame
    jr    $ra           # return
```

Stack frames

In the previous example, we saved:

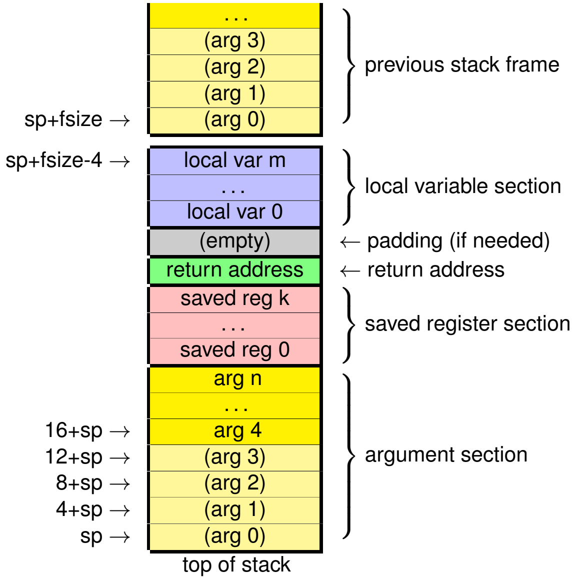
- `$t0` in `16($sp)`
- `$ra` in `20($sp)`

How did we determine these offsets?

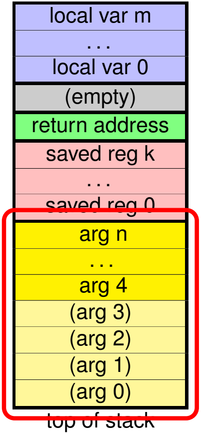
Why didn't we use offsets 0, 4, 8, or 12?

Calling conventions dictate the **structure** of a stack frame

Anatomy of a stack frame



Argument section (probably the most confusing section)



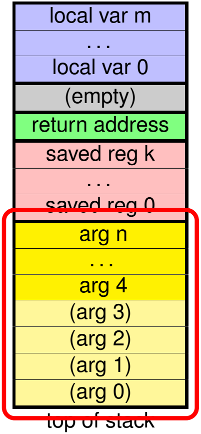
Used for passing arguments to **subroutines**

- *procedures called by this procedure*

First four words (arg 0 – arg 3)

- `0($sp), 4($sp), 8($sp), 12($sp)`
- must always be allocated!
  (even if no subroutine takes four args)
- never used by this procedure
- place for subroutines to store `$a0`–`$a3`
  (and for interfacing with other calling conventions)

Argument section (probably the most confusing section)



Used for passing arguments to **subroutines**

- *procedures called by this procedure*

Remaining words (arg 4 – arg n)

- used to pass more args to subroutines
- written to by this procedure (caller)
- read by subroutine (callee)

What about args passed to this procedure?

- at the top of *previous* stack frame
- read before pushing *this* stack frame

Using the argument section

```
# Pseudocode: myProcedure(a,b,c,d,e)
# Registers: a,b,c,d => $a0-$a3, e => $s0
myProcedure:
    lw    $s0, 16($sp)   # retrieve e from prev stack frame
    addiu $sp, $sp, -32 # push new stack frame
    ...
    ...                  # put first four args in $a0--$a3
    sw    $s1, 16($sp)   # store j for subroutine
    sw    $s2, 20($sp)   # store k for subroutine
    jal   subRoutine     # call subRoutine
    ...
```

```
# Pseudocode: subRoutine(f,g,h,i,j,k) { ... }
# Registers: f,g,h,i => $a0-$a3, j => $t0, k => $t1
subRoutine:
    lw    $t0, 16($sp)   # retrieve j from prev stack frame
    lw    $t1, 20($sp)   # retrieve k from prev stack frame
    ...
```

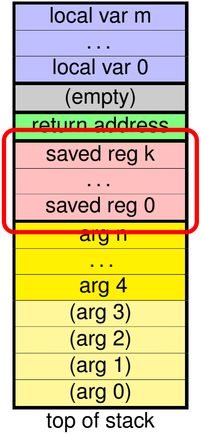
Size of argument section

How much space do we need for the argument section?

1. look at all of the subroutines this procedure calls
2. let *n* be the largest number of args to any subroutine
3. need **max(*n*,4)** words

If we call *any* subroutines, we need at least 4 words!

Saved register section



Initial values of saved registers (`$s0`–`$s7`) that are used in this procedure

- so we can restore them at the end

How to use

- at beginning of procedure, save each `$s` register used in the body
- at end of procedure, restore values

This is our responsibility as a callee!

(even if you “know” caller doesn’t use them)

Using the saved register section

```
# Registers: a => $s0, b => $s1
myProcedure:
    ...                     # maybe retrieve args
    addiu $sp, $sp, -32     # push new stack frame
    sw    $s0, 16($sp)      # save $s0
    sw    $s1, 20($sp)      # save $s1
    ...
    (body of procedure)     # (uses $s0 and $s1)
    ...
    lw    $s0, 16($sp)      # restore $s0
    lw    $s1, 20($sp)      # restore $s1
    addiu $sp, $sp, 32      # pop stack frame
    jr    $ra               # return
```

Return address

local var m
. . .
local var 0
(empty)
return address
saved reg k
. . .
saved reg 0
arg n
. . .
arg 4
(arg 3)
(arg 2)
(arg 1)
(arg 0)

top of stack

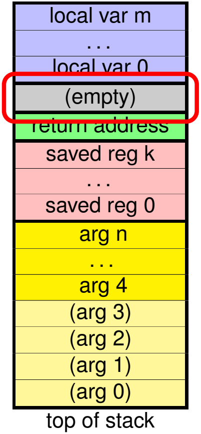
Save `$ra`, so we can restore it later

- needed if we call any subroutines

How to use

- at beginning of procedure, save `$ra`
- at end of procedure, restore `$ra`

Padding

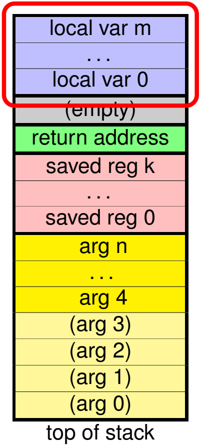


Another seemingly arbitrary rule

- `$sp` must always be a multiple of 8!
  - reason: double-length args passed in `$a0`+`$a1`, `$a2`+`$a3`

If size of stack frame is a multiple of 4, the empty word of padding goes here

Local variable section



Place to save:

- values of temp registers (`$t0`–`$t9`)
  (during subroutine calls)
- local variables in memory

How to use

- save temps before procedure call
- restore temps after procedure call
- local variables – just like data segment except **not initialized**

How much space do you need for your stack frame?

Three kinds of procedures:

Simple leaf no subroutines or local data	$\Longrightarrow$ no stack frame!
Leaf w/ data no subroutines, local data	$\Longrightarrow$ however much you need
Non-leaf calls subroutines	$\Longrightarrow$ most sections of stack frame

Minimum size: 6 words (24 bytes)

- arg 0 – arg 3 (4)
- return address (1)
- padding (1)

Calculating non-leaf stack frame size

To determine number of words, calculate:

1. size of argument section
   - look at all of the subroutines this procedure calls
   - let *n* be the largest number of args to any subroutine
   - need **max(*n*,4)** words
2. \+ size of saved register section
   - number of `$s` registers your procedure uses
3. \+ 1 for return address
4. \+ 1 for padding, if needed to make frame size multiple of 8
5. \+ size of local variable section
   - number of `$t` registers your procedure uses both before and after a subroutine
   - \+ space needed for local memory variables

. . . then multiply by 4 to get frame size

Outline

Subroutine linkage

Definition

The “boilerplate” code needed to:

- satisfy the calling conventions
- manage the stack

This is the same stuff you’ve already seen organized in a different way

Caller

1. startup sequence
2. call procedure
3. cleanup sequence

Callee

1. procedure prologue
2. procedure body
3. procedure epilogue

Caller responsibilities

Caller startup sequence

1. save `$t` registers needed after call (local var section)
2. setup args to send to procedure (`$a0`–`$a3`, arg section)

(procedure call)

Caller cleanup sequence

1. retrieve result of procedure (`$v0`–`$v1`)
2. restore `$t` registers saved in startup

Callee responsibilities

Callee procedure prologue

1. retrieve arguments from stack (prev arg section)
2. push new stack frame
3. save **`$s`** registers used in body (saved register section)
4. save **`$ra`** (return address)

(procedure body)

Callee procedure epilogue

1. restore **`$s`** registers saved in prologue
2. restore **`$ra`**
3. pop stack frame

Responsibilities of a procedure

Remember: non-leaf procedure can be both a callee and caller!

```
myProcedure:
  # (procedure prologue, as callee)
  ...
  # (caller startup)
  jal subRoutine1
  # (caller cleanup)
  ...
  # (caller startup)
  jal subRoutine2
  # (caller cleanup)
  ...
  # (procedure epilogue, as callee)
  jr  $ra
```