Supporting Procedure Call

Introduction

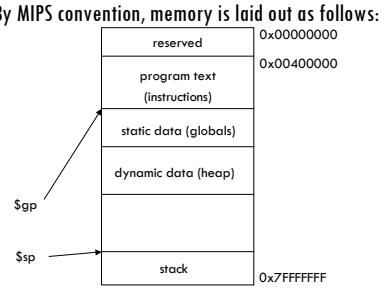
- Procedures/functions are a crucial program structuring mechanism.
- To support them, we need a *calling convention*. Why?
- Different compilers define their own calling conventions, but usually they are pretty similar.
- In RISC machines, the hardware performs only simple instructions, so the programmer/compiler has to implement the bulk of a convention...

The Program Stack

- Each process has its own stack.
- The stack is a dynamic data structure, accessed in LIFO manner.
- Memory for the stack is allocated during the load process.
- The register $sp (29) in MIPS is loaded to point to the first empty spot on the stack.
- By convention, the stack grows towards *lower* memory addresses
  - To free stack space, we add to $sp
  - To allocate stack space, we increment $sp

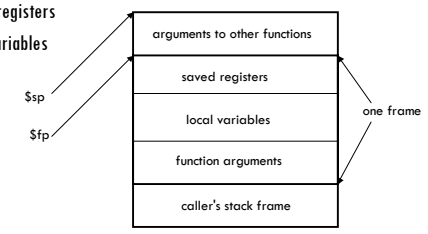
MIPS Program/Memory Layout

- By MIPS convention, memory is laid out as follows:



A Stack Frame

- A stack frame is a block of memory on the stack that is used for:
  - Passing arguments
  - Saving registers
  - Local variables



Procedure Call Sequence

- Terms: *callee* (the procedure that is called); *caller* (the procedure doing the calling)
- Here's a generic sequence of events surrounding a call:
  - Caller must pass the return address to the callee
  - Caller must pass parameters to the callee
  - Caller must save any registers that the callee might want to use
  - Jump to the 1st instruction of the callee
  - Callee must allocate space for local variables, possibly save registers
  - Callee executes...
  - Callee has to restore registers (possibly) and return to caller
  - Caller continues...

Mechanisms

- How do we save information? Pass information? Make space for locals? The MIPS convention uses registers to:
  - Pass the return address in $ra
  - Pass a small number of arguments in $a0-$a4
  - Keep track of the stack pointer $sp
  - Return values from functions (in $v0 and $v1)
- The stack is used for:
  - Saving registers the callee might use
  - Save information about the caller ($ra, why?)
  - Pass additional parameters
  - Allocate space for locals

Register Conventions

- This table should make more sense now:

Number	Name	Use	Comments
$2-$3	$v0-$v1	Function return value	
$4-$7	$a0-$a3	Function call parameters	
$8-$15	$t0-$t7	Volatile temporaries	caller saved
$16-$23	$s0-$s7	Saved temporaries	callee saved
$24-$25	$t8-$t9	Volatile temporaries	
$28	$gp	Global pointer	
$29	$sp	Stack pointer	
$30	$fp	Frame pointer	
$31	$ra	Return address	

Who Saves/Restores Registers?

- When one procedure calls another, what happens to the data in the registers it was using? Either we don't care, or we do, in which case someone has to save the values...
- Two main approaches:
  - *Caller saves*: The caller saves any registers that it wants preserved before making a call and restores them afterwards
  - *Callee saves*: The callee saves any registers that it wants to use, and restores them before it returns

MIPS approach

- MIPS takes a hybrid approach. It classifies some registers as *caller-saved* and some as *callee-saved*
- The caller must save registers $t0-$t9 before making a call and restore them afterwards. These are called *volatiles*, sometimes.
- The callee must save registers $s0-$s7 and $ra before using them and clean them up afterwards
- Why such a crazy approach?
  - Compilers are good at choosing between long-lived and short-lived values and putting them in the right registers... Which go where?

A Convention of Our Invention

- The trouble with conventions: No one agrees on them.
  - The text presents two different ones.
  - The MIPS manual suggests another
  - gcc uses another
- So we'll make up our own: it's very simple.
- Think about these 4 points of execution:
  - Entry to a procedure
  - Before calling another procedure
  - After calling another procedure
  - Exit from the procedure

Procedure Entry

- Allocate stackspace:

```
addi    $sp, $sp, -[framesize]
```

- *framesize* is calculated by determining the number of bytes for:
  - Local variables
  - Saved registers (only $ra and $fp in our scheme)
- Save registers:

```
sw      $ra, 0($sp)
sw      $fp, 4($sp)
```

- Set up frame pointer:

```
add     $fp, $sp, $0
```

Procedure Exit

- Undo the entry code
  - restore registers
  - shrink the stack

```
lw      $ra, 0($fp)
lw      $fp, 4($fp)
addi    $sp, $sp, [framesize]
```

- Return to the caller

```
jr      $ra
```


Prior to a Call

- Save in-use registers. Grow the stack the right amount, store the registers to the stack.
- Pass arguments on the stack. This means growing the stack by 4 * the number of arguments:

```
addi    $sp, $sp, -[argsize]
```

- Put the arguments on the stack:

```
sw      ..., 0($sp)     # first argument
sw      ..., 4($sp)     # second argument
sw      ..., 8($sp)     # etc.
```

- Jump to the function by executing jal

```
jal     someFunctionLabel
```

  - (This will put the return address into $ra.)

After a Call

- First shrink the stack to pop off the arguments.
- Then restore registers we had saved before the call.
- Then restore the stack pointer to its original place.

- The framepointer. Why?
  - In our scheme, the stack pointer grows and shrinks throughout a function body. But the framepointer always points to a fixed place.
  - This is important so we can easily calculate offsets to locals, arguments, and so on...

Example: Recursive Factorial

```
int factorial(int n) {
   if (n==0) {
     return 1;
   }
   else {
     return n * factorial(n-1);
   }
}
```