COMPUTER ORGANIZATION (CDA – 3103)

SPRING 2005

Lab # 9: Stacks & Procedure Calls

For reference use:

1. ***Appendix A of your text book. (This lab has been designed using this source)***
2. ***Previous SPIM Lab Handouts (available on Course Website)***

1 REGISTER-USE CONVENTIONS

Conventions governing the use of registers are necessary when procedures in a program are compiled separately. To compile a particular procedure, a compiler/assembly programmer must know which registers it may use and which registers are reserved for other procedures. Rules for using registers are called register use or procedure call conventions.

As the name implies, these rules are, for the most part, conventions followed by software rather than rules enforced by hardware. However, most compilers and programmers try very hard to follow single set of conventions because violating them may cause insidious bugs.

1.1 CALLER-SAVED REGISTER

A register saved by the routine being called.

1.2 CALLEE-SAVED REGISTER

A register saved by the routine making a procedure call.

2 PROCEDURE CALLING

Conventions governing the use of registers are necessary when procedures in a program are compiled separately. To compile a particular procedure, a compiler/assembly programmer must know which registers it may use and which registers are reserved for other procedures. Rules for using registers are called register use or procedure call conventions.

This section describes the steps that occur when one procedure (the caller) invokes another procedure (the callee). Programmers who write in a high-level language (like C or Pascal) never see the details of how one procedure calls another because the compiler takes care of this low-level bookkeeping. However, assembly language programmers must explicitly implement every procedure call and return.

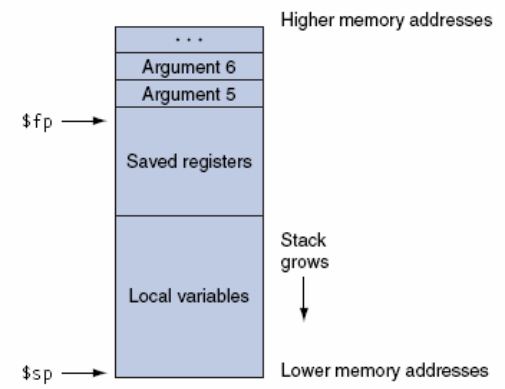
2.1 PROCEDURE CALL FRAME

Most of the bookkeeping associated with a call is centered around a block of memory called a "procedure call frame".

This memory is used for a variety of purposes:

1) To hold values passed to a procedure as arguments
2) To save registers that a procedure may modify, but which the procedure's caller does not want changed
3) To provide space for variables local to a procedure

In most programming languages, procedure calls and returns follow a strict last-in, first-out (LIFO) order, so this memory can be allocated and deallocated on a stack, which is why these blocks of memory are sometimes called stack frames.



A stack frame may be built in many different ways; however, the caller and callee must agree on the sequence of steps. The steps below describe the calling convention used on most MIPS machines. This convention comes into play at three points during a procedure call: immediately before the caller invokes the callee, just as the callee starts executing, and immediately before the callee returns to the caller.

In the first part, the caller puts the procedure call arguments in standard places and invokes the callee to do the following:

1. Pass arguments: By convention, the first four arguments are passed in registers $a0–$a3. Any remaining arguments are pushed on the stack and appear at the beginning of the called procedure's stack frame.
2. Save caller-saved registers: The called procedure can use these registers ($a0–$a3 and $t0–$t9) without first saving their value. If the caller expects to use one of these registers after a call, it must save its value before the call.
3. Execute a JAL instruction: which jumps to the callee's first instruction and saves the return address in register $ra.

Before a called routine starts running, it must take the following steps to set up its stack frame:

1. Allocate memory for the frame by subtracting the frame's size from the stack pointer.
2. Save callee-saved registers in the frame. A callee must save the values in these registers ($s0–$s7, $fp, and $ra) before altering them since the caller expects to find these registers unchanged after the call. Register $fp is saved by every procedure that allocates a new stack frame. However, register $ra only needs to be saved if the callee itself makes a call. The other callee-saved registers that are used also must be saved.
3. Establish the frame pointer by adding the stack frame's size minus 4 to $sp and storing the sum in register $fp.

Finally, the callee returns to the caller by executing the following steps:

1. If the callee is a function that returns a value, place the returned value in register $v0.
2. Restore all callee-saved registers that were saved upon procedure entry.
3. Pop the stack frame by adding the frame size to $sp.
4. Return by jumping to the address in register $ra.

3 PROCEDURE CALL EXAMPLE

3.1 EXAMPLE C CODE

```
main ()
{
        printf ("The factorial of 10 is %d\n", fact (10));
}

int fact (int n)
```

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

3.2 MIPS CODE

3.2.1 Main Functions

Upon entry, the routine main creates its stack frame and saves the two callee-saved registers it will modify: $fp and $ra. The frame is larger than required for these two registers because the calling convention requires the minimum size of a stack frame to be 24 bytes. This minimum frame can hold four argument registers ($a0–$a3) and the return address $ra, padded to a double-word boundary (24 bytes). Since main also needs to save $fp, its stack frame must be two words larger (remember: the stack pointer is kept doubleword aligned).

```
.text
        .globl main
main:
        subu $sp,$sp,32 # Stack frame is 32 bytes long
        sw $ra,20($sp) # Save return address
        sw $fp,16($sp) # Save old frame pointer
        addiu $fp,$sp,28 # Set up frame pointer
```

The routine main then calls the factorial routine and passes it the single argument 10. After fact returns, main calls the library routine printf and passes it both a format string and the result returned from fact:

```
        li $a0,10 # Put argument (10) in $a0
        jal fact # Call factorial function

        move $v1 , $v0
        li $v0, 4                       # system call code for print_str
        la $a0, $LC             # address of string to print
        syscall                 # print the string

        li $v0, 1                       # system call code for print_int
        move $a0 , $v1
        syscall                 # print int
```

Finally, after printing the factorial, main returns. But first, it must restore the registers it saved and pop its stack frame:

```
        lw $ra,20($sp) # Restore return address
        lw $fp,16($sp) # Restore frame pointer
        addiu $sp,$sp,32 # Pop stack frame
        jr $ra # Return to caller
        .rdata

        $LC:
        .ascii "The factorial of 10 is %d\n\000"
```

3.2.2 Factorial Functions

The factorial routine is similar in structure to main. First, it creates a stack frame and saves the callee-saved registers it will use. In addition to saving $ra and $fp, fact also saves its argument ($a0), which it will use for the recursive call:

```
.text

fact:
        subu $sp,$sp,32 # Stack frame is 32 bytes long
```

```
        sw $ra,20($sp) # Save return address
        sw $fp,16($sp) # Save frame pointer
        addiu $fp,$sp,28 # Set up frame pointer
        sw $a0,0($fp) # Save argument (n)
```

The heart of the fact routine performs the computation from the C program. It tests if the argument is greater than 0. If not, the routine returns the value 1. If the argument is greater than 0, the routine recursively calls itself to compute
fact(n-1) and multiplies that value times n:

```
        lw $v0,0($fp) # Load n
        bgtz $v0,$L2 # Branch if n > 0
        li $v0,1 # Return 1
        jr $L1 # Jump to code to return
$L2:
        lw $v1,0($fp) # Load n
        subu $v0,$v1,1 # Compute n - 1
        move $a0,$v0 # Move value to $a0

        jal fact # Call factorial function
        lw $v1,0($fp) # Load n
        mul $v0,$v0,$v1 # Compute fact(n-1) * n
```

Finally, the factorial routine restores the callee-saved registers and returns the value in register $v0:

```
$L1: # Result is in $v0
        lw $ra, 20($sp) # Restore $ra
        lw $fp, 16($sp) # Restore $fp
        addiu $sp, $sp, 32 # Pop stack
        jr $ra # Return to caller
```

3.3 FINAL MIPS CODE

```
.text

.globl main

main:
        subu $sp,$sp,32 # Stack frame is 32 bytes long
        sw $ra,20($sp) # Save return address
        sw $fp,16($sp) # Save old frame pointer
        addiu $fp,$sp,28 # Set up frame pointer

        li $a0,10 # Put argument (10) in $a0
        jal fact # Call factorial function

        move $v1 , $v0
        li $v0, 4                       # system call code for print_str
        la $a0, $LC             # address of string to print
        syscall                 # print the string

        li $v0, 1                       # system call code for print_int
        move $a0 , $v1
        syscall                 # print int

        #la $a0,$LC # Put format string in $a0
        #move $a1,$v0 # Move fact result to $a1
        #jal printf # Call the print function
```

```
        lw $ra,20($sp) # Restore return address
        lw $fp,16($sp) # Restore frame pointer
        addiu $sp,$sp,32 # Pop stack frame
        jr $ra # Return to caller
        .rdata
$LC:
        .ascii "The factorial of 10 is %d\n\000"

.text

fact:
        subu $sp,$sp,32 # Stack frame is 32 bytes long
        sw $ra,20($sp) # Save return address
        sw $fp,16($sp) # Save frame pointer
        addiu $fp,$sp,28 # Set up frame pointer
        sw $a0,0($fp) # Save argument (n)

        lw $v0,0($fp) # Load n
        bgtz $v0,$L2 # Branch if n > 0
        li $v0,1 # Return 1
        jr $L1 # Jump to code to return
$L2:
        lw $v1,0($fp) # Load n
        subu $v0,$v1,1 # Compute n - 1
        move $a0,$v0 # Move value to $a0

        jal fact # Call factorial function
        lw $v1,0($fp) # Load n
        mul $v0,$v0,$v1 # Compute fact(n-1) * n

$L1: # Result is in $v0
        lw $ra, 20($sp) # Restore $ra
        lw $fp, 16($sp) # Restore $fp
        addiu $sp, $sp, 32 # Pop stack
        jr $ra # Return to caller
```

3.4 TEST RUNS