Computer Systems Architecture

http://cs.nott.ac.uk/∼txa/g51csa/

Thorsten Altenkirch and Liyang Hu

School of Computer Science
University of Nottingham

Lecture 10: MIPS Procedure Calling Convention and Recursion

A procedure by any other name...

- A portion of code within larger program, typically called:
  - *procedures* or *subroutines* in imperative languages like C
  - *methods* in OO languages like Java
  - and *functions* in functional languages such as Haskell
- *Functions* usually return a value; *procedures* don't
- Procedures are necessary to
  - reduce duplication of code and enable re-use
  - decompose complex programs into manageable parts
- Procedures can call other procedures; even themselves
- What happens when we call a procedure?
  - Control hands over to the *callee*; the *caller* is suspended
  - Callee performs requested task
  - Callee returns control to the *caller*

The University of Nottingham

An Example of Procedures in C

```
int f(int x, int y) {
  return sqrt(x * x + y * y);
}

int main() {
  printf("f(5,12)=%d\n",f(5, 12));
}
```

Calling Procedures in MIPS Assembly

`jal` *label* – jump and link

- `$ra := PC + 4;  PC := [`*label*`]`
- Calls procedure at address *label*

`jr` *src* – jump register

- `PC :=` *src*
- Issuing `j $ra` is the assembly equivalent of `return`

- Register `$ra` contains the return address of the caller
- Arguments are passed in registers `$a0` to `$a3`
- Results are left in registers `$v0` and `$v1`

First Attempt

```
f:   mult $a0, $a0, $a0
     mult $a1, $a1, $a1
     add $a0, $a0, $a1
     jal sqrt
     jr $ra
main:
     li $a0, 5
     li $a1, 12
     jal f
     move $a0, $v0
     li $v0, 1
     syscall
     jr $ra
```

- What's wrong with this?
  - `$ra` modified by `jal`, so...
  - `j $ra` jumps to wrong address
- Must save required registers
  - Previous value of `$ra`
  - `f` overwrites `$a0` and `$a1`
- What if we need > 4 arguments?

The University of Nottingham

The Stack

- Not enough registers?
  - Save the contents of some registers to memory
- The stack provides *last-in, first out* (LIFO) storage
  - Register `$sp` points to the topmost word on the stack
  - By convention, the stack grows *downwards*
  - Placing words onto the stack is termed *pushing*
  - Taking words off the stack is called *popping*

Calling Convention

Caller

- Push any of `$a0-3`, `$v0-1` and `$t0-9` needed later
- Place arguments in `$a0` to `$a3`, and stack if necessary
- Make the call using `jal` *callee*; result in `$v0` and `$v1`
- Pop saved registers and/or extra arguments off stack

Callee

- Push any of `$ra`, `$s0-$s9` that may be overwritten
- Perform desired task; place result in `$v0` and `$v1`
- Pop above registers off the stack
- Return to caller with `jr $ra`

Procedure Example

```
f:
    addi $sp, $sp, -4    # allocate space on stack
    sw $ra, 0($sp)       # push $ra onto stack
    mult $a0, $a0, $a0
    mult $a1, $a1, $a1
    add $a0, $a0, $a1
    jal sqrt             # call sqrt
    lw $ra, 0($sp)       # pop $ra off stack
    addi $sp, $sp, 4     # deallocate space on stack
    jr $ra
```

Calling Convention Summary

Preserved by Callee	Not Preserved
Saved registers `$s0`-`$s7`	Temporary registers `$t0`-`$t9`
Stack pointer `$sp`	Argument registers `$a0`-`$a3`
Return address `$ra`	Return values `$v0` and `$v1`
Stack at/above `$sp`	Stack below `$sp`

- Items not preserved but needed later, caller must preserve
- Stack contents preserved by not writing at/above `$sp`
- Stack pointer 'saved' by always popping what we pushed
- Leaf procedures are those which do not make further calls
  - In such instances, we needn't explicitly save `$ra`
- The `main` label is just another procedure
  - Ought to follow the same conventions

Local Variables

- We needn't preserve the stack below the initial `$sp`
  - A convenient location for local storage
- Creating locals: subtract number of bytes from `$sp`
- Complex functions may do this many times
  - Each time this changes the `$sp`-offset of previous locals!
  - Assembly harder for humans and debuggers to read!
- Solution: save initial value of `$sp` in `$fp`
  - Hence local variables always have the same `$fp`-offset
  - Note callee must preserve previous value of `$fp`!

Example: Recursive Factorial

```
# int fact(int n): return n <= 0 ? 1 : n * fact(n-1);
fact:
    addi $sp, $sp, -8   # space for two words
    sw $ra, 4($sp)      # save return address
    sw $a0, 0($sp)      # temporary variable to hold n
    li $v0, 1
    ble $a0, $zero, fact_return
    addi $a0, $a0, -1
    jal fact
    lw $a0, 0($sp)      # retrieve original n
    mul $v0, $v0, $a0   # n * fact(n - 1)
fact_return:
    lw $ra 4($sp)       # restore $ra
    addi $sp, $sp, 8    # restore $sp
    jr $ra              # back to caller
```

Example: Recursive Fibonacci

```
# int fib(int n): return n < 2 ? n : fib(n-1) + fib(n-2)
fib:    addi $sp, $sp, -8     # room for $ra and one temporary
        sw $ra, 4($sp)        # save $ra
        move $v0, $a0         # pre-load return value as n
        blt $a0, 2, fib_rt    # if(n < 2) return n
        sw $a0, 0($sp)        # save a copy of n
        addi $a0, $a0, -1     # n - 1
        jal fib               # fib(n - 1)
        lw $a0, 0($sp)        # retrieve n
        sw $v0, 0($sp)        # save result of fib(n - 1)
        addi $a0, $a0, -2     # n - 2
        jal fib               # fib(n - 2)
        lw $v1, 0($sp)        # retrieve fib(n - 1)
        add $v0, $v0, $v1     # fib(n - 1) + fib(n - 2)
fib_rt: lw $ra, 4($sp)        # restore $ra
        addi $sp, $sp, 8      # restore $sp
        jr $ra                # back to caller
```

The University of Nottingham

Reading

- Read up on calling conventions in H&P:
  - §2.7 (pp 79–86)
  - Appendix A §6 (pp 22–33)