

# ECE260: Fundamentals of Computer Engineering

---

## Supporting Procedures in Computer Hardware

James Moscola  
Dept. of Engineering & Computer Science  
York College of Pennsylvania



# What are Procedures?

---

- Come with many different names:
  - **Subroutine** – a repeatable piece of code that you can call by name
  - **Procedure** – a subroutine that doesn't return a value
    - May have side-effects that write to files, print to the screen, modify input values, etc.
  - **Function** – a subroutine that returns one or more output values based on one or more input values
    - Given some input, produces some output (similar to a mathematical function)
  - **Method** – a procedure or function that is executed in the context of an object-oriented class
    - May have side-effects that modify the state of the object

# Why Use Procedures?

---

- Readability
  - Divide up large programs into smaller procedures
- Reusability
  - Call same procedure from many parts of code
  - Programmers can use each others' code
- Parameterizability
  - Same function can be called with different arguments/parameters at runtime to produce different output or side-effects
- Polymorphism (in OOP – C++/Java/etc.)
  - Behavior can be determined at runtime as opposed to compile time based on object type

# Why Use Procedures? (continued)

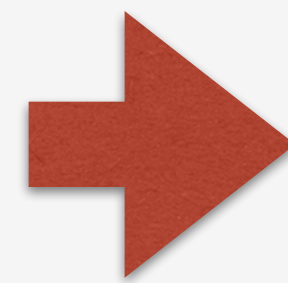
- Scoping of variable names
  - Use locally scoped variable names that make sense in the context of the procedure
  - No need to keep track of ALL variable names throughout entire program

```
int x = 5;

int foo(int x) {
    return x + 1;
}

int bar(int y) {
    int x = 0;
    return foo(x) + y;
}

main() {
    foo(bar(x));
}
```



```
int x1 = 5;

int foo(int x3) {           // arg x3 shadows x1
    return x3 + 1;
}

int bar(int y) {
    int x2 = 0;             // local var x2 shadows x1
    return foo(x2) + y;
}

main() {
    foo(bar(x1));           // no local x, use x1
}
```

# What's the Procedure?

---

- Main idea for procedures
  - Some main code routine **M** calls a procedure **P**
  - **P** does some work, then returns to **M**
    - Execution in **M** picks up where it left off prior to calling **P**
    - i.e., execution continues at the instruction in **M** after the instruction that called **P**

# Using Procedures

---

- A “calling” program (**Caller**) must:
  - Provide procedure parameters
    - Put the arguments in a place where the procedure can access them
  - Transfer control to the procedure (i.e. jump to the procedure)
- A “called” procedure (**Callee**) must:
  - Acquire the resources (i.e. registers) needed to perform the function
  - Perform the function
  - Place results in a place where the Caller can find them
  - Return control back to the Caller

# Steps for Calling Procedure on MIPS

---

1. Place parameters in registers before calling so callee can access them (use \$a0 - \$a3)
2. Transfer control to procedure using special instruction
  - Use **jal** instruction ... it saves a return address ... more info coming soon
3. Acquire storage resources needed for the procedure
  - Must backup (and later restore) contents of any \$sX registers used in procedure
  - \$tX registers may be overwritten by subroutines ... be careful
4. Perform procedure's operations
5. Place result in register where the caller can access it (use \$v0 - \$v1)
6. Return to place of call
  - Register \$ra contains return address



# MIPS Registers (now with more info!)

- MIPS architecture has a  $32 \times 32$ -bit register file (e.g. it has 32 32-bit registers)

| Register Number | Register Name | Use                     |
|-----------------|---------------|-------------------------|
| 0               | \$zero        | Constant value 0        |
| 1               | \$at          | Assembler temporary     |
| 2 - 3           | \$v0 - \$v1   | Procedure return values |
| 4 - 7           | \$a0 - \$a3   | Procedure arguments     |
| 8 - 15          | \$t0 - \$t7   | Temporary values        |
| 16 - 23         | \$s0 - \$s7   | Saved temporary values  |
| 24 - 25         | \$t8 - \$t9   | More temporary values   |
| 26 - 27         | \$k0 - \$k1   | Reserved for OS         |
| 28              | \$gp          | Global pointer          |
| 29              | \$sp          | Stack pointer           |
| 30              | \$fp          | Frame pointer           |
| 31              | \$ra          | Return Address          |

← **Not for you!**

← **Callee puts return value here**

← **Caller puts arguments here**

← **Callee may overwrite these**

← **Must be saved by callee if used**

← **Callee may overwrite these**

← **Also, not for you!**

← **Easy access to constants/globals**

← **Top of stack**

← **Points to local variables on stack**

← **Where to go when returning from procedure**



# Another Important CPU Register – Program Counter

---

- Register that contains the address of the instruction currently being executed
  - Abbreviated as **PC** for Program Counter
  - Should always point to a memory address in the TEXT segment of memory
    - Side note: common security exploit forces PC to point to point to stack ... this is **BAD**
  - Typically increments by 4 after each instruction to point to the next instruction
    - $PC = PC + 4$
    - Increments by 4 since instructions are 4 bytes each (32-bits)
- PC can be set using branch and jump instructions in lieu of typical increment behavior
  - This is how procedures are "called"

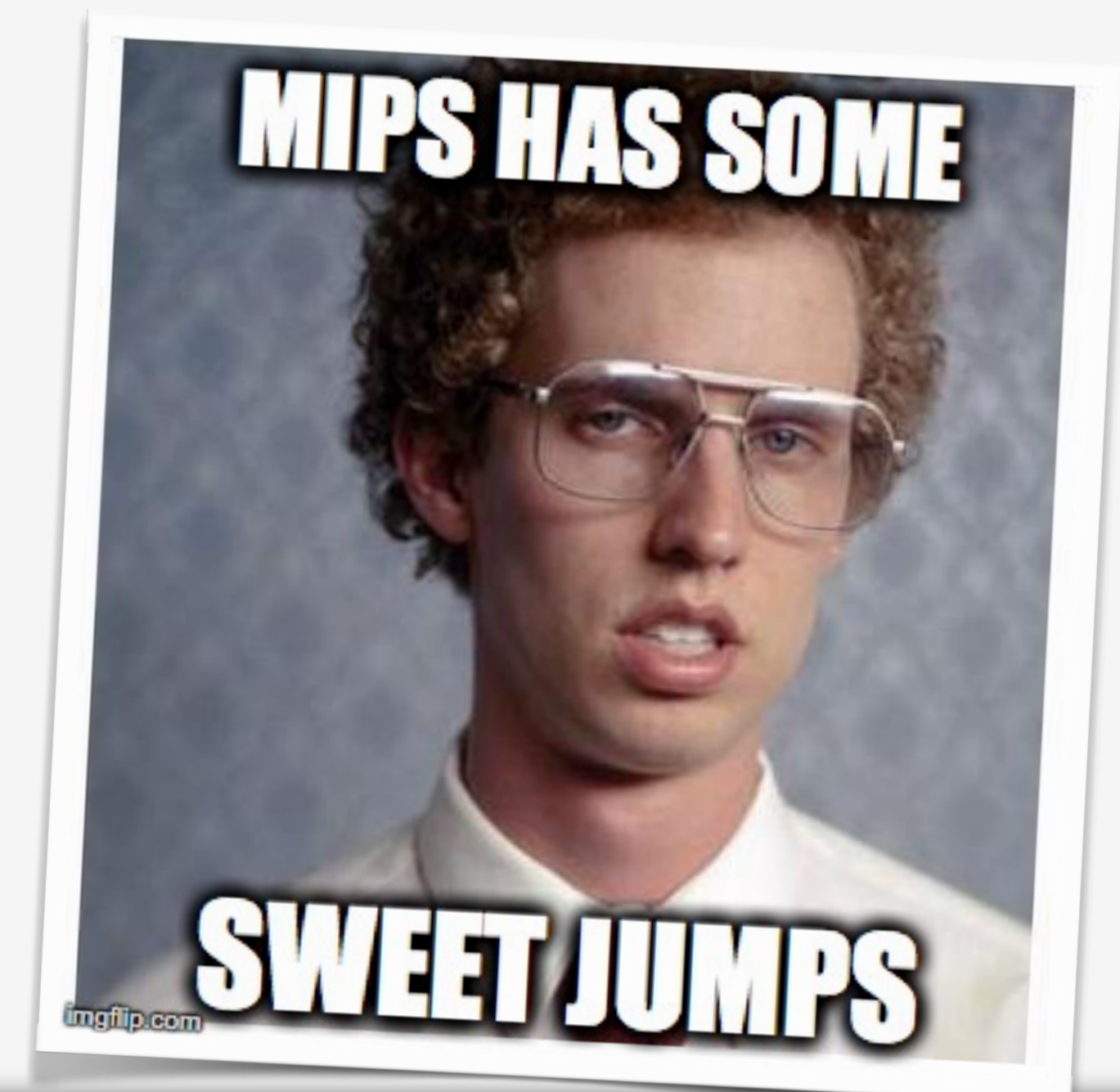
# Procedure Call Instructions

- Jump-and-Link (jal) – a procedure call instruction
  - Address of instruction immediately after jal is saved in \$ra (i.e.  $\$ra = PC + 4$ )
  - Jumps to target address (e.g. address of ProcedureLabel)

**jal ProcedureLabel**

- Jump Register (jr) – return from a procedure
  - Copies \$ra to program counter (i.e.  $PC = \$ra$ )
  - Can also be used for computed jumps (32-bit jump distance :-)
    - e.g. for case/switch statements

**jr \$ra**



# Leaf Procedure Example

---

- A leaf procedure is a procedure that does NOT call another procedure
- Example C code

```
int leaf_example (int g, h, i, j) {  
    int f;  
    f = (g + h) - (i + j);  
    return f;  
}
```

- Assume the following:
  - Parameters g, h, i, j are passed in registers \$a0, \$a1, \$a2, \$a3
  - leaf\_example code is called using a jal instruction that sets the \$ra register
  - Plan to use register \$s0 for local variable f
  - Must place return value in register \$v0 when done

# Leaf Procedure Example (continued)

```
int leaf_example (int g, h, i, j) {  
    int f;  
    f = (g + h) - (i + j);  
    return f;  
}
```

```
leaf_example:  
    addi $sp, $sp, -4      # plan to use $s0 locally, so make room on stack to save old $s0  
    sw   $s0, 0($sp)      # store callers value of $s0 on stack ... will be restored later  
  
    add  $t0, $a0, $a1     # $t0 = g + h  
    add  $t1, $a2, $a3     # $t1 = i + j  
    sub  $s0, $t0, $t1     # f = (g + h) - (i + j) ... recall f is stored in $s0  
  
    add  $v0, $s0, $zero   # put f into special return value register $v0  
  
    lw   $s0, 0($sp)      # before returning, restore callers value of $s0 from stack  
    addi $sp, $sp, 4      # shrink stack to previous size (recall increasing $sp shrinks stack)  
  
    jr   $ra              # return to caller (instruction after jal instruction that called this)
```

# Leaf Procedure Example (continued ... maybe better)

```
int leaf_example (int g, h, i, j) {  
    int f;  
    f = (g + h) - (i + j);  
    return f;  
}
```

**For this example, no real need to use \$s0, just put result directly into \$v0 and skip the stack manipulation**

**leaf\_example:**

```
addi $sp, $sp, -4      # plan to use $s0 locally, so make room on stack to save old $s0  
sw   $s0, 0($sp)      # store callers value of $s0 on stack ... will be restored later  
  
add  $t0, $a0, $a1      # $t0 = g + h  
add  $t1, $a2, $a3      # $t1 = i + j  
sub  $v0, $t0, $t1      # f = (g + h) - (i + j) ... put directly into $v0  
  
add  $v0, $s0, $zero   # put f into special return value register $v0  
  
lw   $s0, 0($sp)      # before returning, restore callers value of $s0 from stack  
addi $sp, $sp, 4      # shrink stack to previous size (recall increasing $sp shrinks stack)  
  
jr   $ra              # return to caller (instruction after jal instruction that called this)
```