Lecture 8: Procedure
+ Nested Procedure
Lab 1-3 Assigned

- Connect 4
- Lab 1: input/output
- Lab2: winner detection
- Lab3: smarter program

- Various due dates: 4/29, 5/13, 5/20
Midterm

- 5/3, 75 minutes
- Cover everything from number representation to assembly
- Will send out sample from past years
- Closed book, closed notes, no calculators
REVIEW: PROCEDURE
Instruction Support for Functions

```c
... sum(a,b);... /* a,b:$s0,$s1 */
}

int sum(int x, int y) {
    return x+y;
}
```

address

```
1000  add  $a0,$s0,$zero    # x = a
1004  add  $a1,$s1,$zero    # y = b
1012  jal  sum              # jump to sum
1016  ...
2000  sum:  add  $v0,$a0,$a1
2004  jr   $ra             # new instruction
```
Procedure calls

Main Program

...  
...  
...  
...  
...  

Sum(a,b)  

Function

...  
...  
...  
loc:  
...  
...  
...  
...  
...
Instruction Support for Functions

• Single instruction to jump and save return address:
  
  jump and link (jal)

1008 jal sum  # $ra=1012?, go to sum

• Why have a jal? Make the common case fast: function calls are very common. Also, you don’t have to know where the code is loaded into memory with jal.
Instruction Support for Functions

• Syntax for $jr$ (jump register):

\[
jr \ register
\]

• Instead of providing a label to jump to, the $jr$ instruction provides a register which contains an address to jump to

• Only useful if we know exact address to jump

• Very useful for function calls:

\[jal\] stores return address in register ($ra$)
\[jr\ $ra\] jumps back to that address
Rules for Procedures

• Called with a `jal` instruction, returns with a `jr $ra`
• Accepts up to 4 arguments in `$a0, $a1, $a2` and `$a3`
• Return value is always in `$v0` (and if necessary in `$v1`)
Volatile Register Conventions

- $ra$: Can Change. The jal call itself will change this register. Caller needs to save on stack if nested call.

- $v0$-$v1$: Can Change. These will contain the new returned values.

- $a0$-$a3$: Can change. These are volatile argument registers. Caller needs to save if they’ll need them after the call.

- $t0$-$t9$: Can change. That’s why they’re called temporary: any procedure may change them at any time. Caller needs to save if they’ll need them afterwards.
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}

NESTED PROCEDURES
Nested Procedures

int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}

• Something called sumSquare, now sumSquare is calling mult.
$ra \xrightarrow{jal \ loc} \ldots \ldots \ldots \ldots \ldots \ldots$

$ra \xrightarrow{jal \ loc2} \ldots \ldots \ldots \ldots \ldots \ldots$

$ra \xrightarrow{jr \ $ra} \ldots \ldots \ldots \ldots \ldots \ldots$

...
Problem on Nested Procedure

```
caller
...
 jal loc
...
...
...
 callee & caller
loc:
...
...
...
...
...
 jal loc2
...
...
...
...
 jr $ra
...
...
...
...
 another callee
loc2:
...
...
...
...
...
...
...
...
...
...
...
...
 jr $ra
...
...
...
...
...
```

X
Nested Procedures

```c
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}
```

• Something called `sumSquare`, now `sumSquare is calling mult`.

• So there’s a value in `$ra` that `sumSquare` wants to jump back to, but this will be `overwritten` by the call to `mult`.

• Need to `save` `sumSquare return address` before call to `mult`. 
Nested Procedures

• In general, may need to save some other info in addition to $r_a$.

• When a C program is run, there are 3 important memory areas allocated:
  – **Static**: Variables declared once per program, cease to exist only after execution completes. E.g., C globals
  – **Heap**: Variables declared dynamically
  – **Stack**: Space to be used by procedure during execution; this is where we can save register values
C Memory Allocation

- **Address $\infty$**
- **Stack**
  - Space for saved procedure information
  - Stack pointer $sp$
  - 0

- **Heap**
  - Explicitly created space, e.g., malloc(); C pointers

- **Static**
  - Variables declared once per program

- **Code**
  - Program
Using the Stack

• So we have a register $sp$ which always points to the last used space in the stack.

• To use stack, we decrement this pointer by the amount of space we need and then fill it with info.

• So, how do we compile this?

```cpp
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}
```
Using the Stack

• Hand-compile

sumSquare:

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

“push”

```
addi $sp,$sp,-8   # space on stack
sw $ra, 4($sp)   # save ret addr
sw $a1, 0($sp)   # save y

add $a1,$a0,$zero   # mult(x,x)
jal mult            # call mult

lw $a1, 0($sp)      # restore y
add $v0,$v0,$a1     # mult()+y
lw $ra, 4($sp)      # get ret addr
addi $sp,$sp,8      # restore stack
```

“pop”

```
jr $ra
```

mult: ...
Nested Procedure

...  
jal loc
...

$cra$  

...  
jal loc2
...

loc:
sub $sp, 4
sw $ra, ($sp)
...
jal loc2

loc2:
...

jr $ra

...  

another callee

...  

$ra$  

...  

jr $ra

...
Basic Structure of a Function

**Prologue**

entry_label:

```assembly
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)  #save $ra
.. save other regs if need be
```

**Body**

*(call other functions...)*

**Epilogue**

```assembly
..restore other regs if need be
lw $ra, framesize-4($sp)  #restore $ra
addi $sp,$sp, framesize
jr $ra
```
EXERCISE --
FIBONACCI NUMBERS

F(n) = F(n – 1) + F(n – 2)
F(0) = F(1) = 1