Procedures 101:
There and Back Again

Slides revised 3/25/2014 by Patrick Kelley

Procedures

- Unlike other branching structures (loops, etc.) a Procedure has to return to where it was called from.
- The instruction that invokes a procedure is a ‘caller’
- The procedure itself is often referred to as ‘callee’
  - Must remember address called from
  - Must have access to parameters, if any
  - Must be able to return values

Shared Memory

- The simple way is to simply use global memory
- This is hard to design for in advance
  - Don't forget to define returns
  - If procedure has many calls, one return may not work
- Dangerous if procedure overwrites something
- Usually end up allocating extra space to be sure
- Called routine may crush registers used by caller.
- Easy to program, though…

Function Calls

- All function calls are made using either jal or jalr:

```
# Label = Name of function
    jal Label
# Rd should be $ra, Rs holds procedure address
    jalr Rd, Rs
```

- Since we save the return address (either Label or Rd above), we can easily return using the jr instruction
- While in a procedure, any memory can be accessed
- On return, execution will continue after the call.

Example Function

```
  .data
param1:  .word 0x0h
param2:  .word 0x0h
return:  .word 0x0h
  .test
… # some code before this, then set up for the call
   sw    $t0, param1          # set the first parameter
   sw    $s2, param2          # set the second parameter
   jal   Sum                  # call the proceedure
   lw    $s1, $return         # get the result
… # more code and eventually a program exit.  After that:

Sum:
   lw    $t0, param1           # get the first parameter
   lw    $t1, param2           # get the second parameter
   add   $t1, $t1, St0         # add them
   sw    $t1, return           # save the result
   jr    $ra                   # return to caller
```

Better Procedure Structure

- Save registers before calling
  - Any register the callee may use
  - Any register passing values
- Pass values or pointers through registers
- Pass return values or pointers back through registers
- Callee can also save and restore registers
  - May be necessary anyway if nested calls
  - Use stack to simplify process

Better Sum Function

```
# Sum
# Takes two numbers passed in $t0 and $t1 and adds them
# together.  The result is passed back in $t2.  No other
# registers are affected.
Sum:
   addiu $sp, -8              # allocate 2 words on stack
   sw    $t0, 0($sp)          # put $t0 on stack
   sw    $t1, 4($sp)          # put $t1 on stack
                              # don't bother with $t2
   add   $t2, $t1, St0        # add them

   lw    $t1, 4($sp)          # restore $t1
   lw    $t0, 0($sp)          # restore $t0
   addiu $sp, 8               # deallocate 2 words on stack
   jr    $ra                  # return to caller
```

Better Sum call

```
…
# Allocate space for registers I want to save.
   addiu $sp, -12              # allocate 3 words on stack
# Save my important registers (these are just an example)
   sw    $t0, 0($sp)           # put $t0 on stack
   sw    $t1, 4($sp)           # put $t1 on stack
   sw    $t2, 8($sp)           # I know Sum is changing $t2

   jal   Sum                   # call the routine
   sw    $t2, result           # store the return

# Restore registers and stack pointer
   lw    $t2, 8($sp)           # saved so I can restore it
   lw    $t1, 4($sp)           # restore $t1
   lw    $t0, 0($sp)           # restore $t0
   addiu $sp, 12               # deallocate 3 words on stack
…
```

Better Still…

```
…
# Allocate space for registers I want to save.
   addiu $sp, -4              # allocate 1 word on stack
# Save my important registers. I read the function header
# and know that only $t2 is in danger.
   sw    $t2, 0($sp)          # I know Sum is changing $t2

   jal   Sum                  # call the routine
   sw    $t2, result          # store the return

# Restore registers and stack pointer
   lw    $t2, 0($sp)          # saved so I can restore it
   addiu $sp, 4               # deallocate 1 word on stack
…
```

About the register file

- $a_ registers are used for parameter passing by convention
  - Callee needs to save and restore if making nested calls
  - Should also normally preserve
- $t_ registers are the caller's responsibility
  - Save before calling
  - Restore after calling
- $s_ registers are the callee's responsibility to preserve
- Don't depend on other programmers to obey the rules
- Other ISAs have similar but different conventions.