Instruction Sets - Part 3
MIPS Subroutines and Programs

Ref: Chapter 3

Subroutines

```
main:
        # multiply 3 x 2
        addi $a0,$zero,3
        addi $a1,$zero,2

        # call the subroutine
        jal multiply

        # print out the result
        move $s0,$v0
        li $v0,4
        la $a0, msg
        syscall

        li $v0,1
        move $a0,$s0
        syscall

        li $v0,10
        syscall
```

```
# mult subroutine needs some registers
# so we save $t0 first
# 2 arguments $a0 and $a1 are multiplied
# using repeated addition

multiply:
        sub $sp,$sp,4   # make room for $t0
        sw  $t0,0($sp)  # put t0 on the stack

        # start with $t0 = 0
        add $t0,$zero,$zero
mult_loop:
        # loop on a1
        beq $a1,$zero,mult_eol

        # add another $a0
        add $t0,$t0,$a0
        # decrement $a1
        sub $a1,$a1,1
        j mult_loop

mult_eol:
        # put the result in $v0
        add $v0,$t0,$zero

        # restore $t0
        lw $t0,0($sp)
        add $sp,$sp,4
        # return to caller
        jr $ra
```


Subroutine Issues

- How to call a subroutine
  - how to pass parameters
  - how to get the return value
- How to write a subroutine
  - where to look for parameters
  - saving registers
  - returning a value
  - returning to the *caller*

Special Registers

`$a0`-**`$a4`**: argument registers
– this is where we put arguments before calling a subroutine.

`$v0`, **`$v1`**: return value registers
– where subroutines put return values

`$ra`: return address register
– holds the address the subroutine should jump to when it's done.

Jump and Link Instruction
`jal` ***`address`***

- Puts the address of the next instruction (**`PC+4`**) in the **`$ra`** register.
- Jumps to the specific address.
- Addressing mode is just like the **`j`** instruction (26 bit absolute address).

Returning from the Subroutine

- Assuming the subroutine doesn't clobber the **`$ra`** register:
  – when the subroutine is done, it jumps to the address in **`$ra`**

```
jr $ra
```

What if the subroutine uses a register?

Accepted convention:

`$t0, $t1, … $t7` are always OK to use.

- if you call a subroutine and you need the value of **`$t0`** to be the same after the call, you must save it in memory!
- *caller saves* **`$t0`** … **`$t7`**

`$s0-$s7` must not be changed by a subroutine.

- If you need them in your subroutine you need to save the previous value and restore them before returning.
- *callee saves* **`$s0`** … **`$s7`**

Saving registers and the Stack

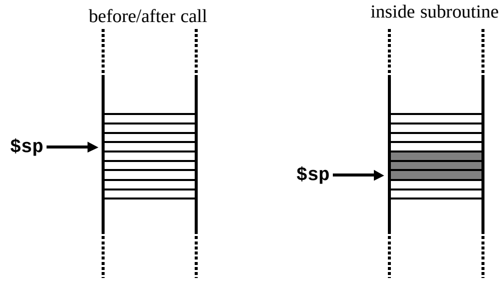
- Most of the time we use whatever registers we want inside subroutines.
  - must save and restore **`$s0-$s7`**
- This happens so often there is a special register and data structure used to support saving and restoring registers.

The Stack

The Stack

- The stack is an area of memory reserved for the purpose of saving registers.
- The **`$sp`** register (*stack pointer)* holds the address of the *top* of the stack.
- The stack grows and shrinks as registers are saved and restored.

`$sp` and Memory



Stack handling code

- Suppose your subroutine needs to use 3 registers: **`$s0`**, **`$s1`** and **`$s2`**:
  - first make room for saving three words by subtracting 12 from the stack pointer

```
sub $sp,$sp,12
```

  - now put copies of the three registers on the stack.

```
sw $s0,0($sp)
sw $s1,4($sp)
sw $s2,8($sp)
```


Stack handling code (cont.)

- Before returning, your subroutine should restore the 3 registers:

```
lw $s2,8($sp)
lw $s1,4($sp)
lw $s0,0($sp)
```

  - And put the stack pointer back to its original value:

```
add $sp,$sp,12
```

Why bother?

- We write subroutines so that they can be called from *any other code*.
  - as far as the *caller* is concerned, **`$s0`**- **`$s7`** don't change.

- The stack provides a single mechanism that will work no matter who called the subroutine.

Exercise

- Create a multiplication subroutine.
  - $a0 is multiplied by $a1 and the product is returned in $v0

- We've already looked at the multiply code, all we need to do is make this a subroutine.

multiply in 'C'

```
int multiply(int x, int y) {
   prod=0;
   while (y>0) {
     prod = prod + x;
     y--;
   }
   return(prod);
}
```

Assembly Multiply

```
int multiply(int x, int y) {
    prod=0;
    while (y>0) {
       prod = prod + x;
       y--;
    }
    return(prod);
}
```

```
multiply:
     add $t0,$zero,$zero # prod=0
m_loop:
     beq $a1,$zero,m_eol # while y>0
     add $t0,$t0,$a0     # prod = prod+x
     subi $a1,$a1,1      # y--;
     j m_loop
m_eol:
     add $v0,$t0,$zero   # return(prod)
     jr $ra
```


Not a typical example!

- `multiply` doesn't need many registers and it doesn't call any subroutines.
  - no need to save and restore registers
- Let's go back and make our assembly version of strcpy a subroutine.

strcpy in C

```
strcpy( char *str1, char *str2 ) {
  while (*str2) {
     *str1 = *str2;
     str1++;
     str2++;
  }
}
```

strcpy in Assembly

str1 is **$s1** and **str2** is **$s2**

```
Loop:     lb   $t0,0($s2)      # $to = *str2
          sb   $t0,0($s1)      # *str1 = $t0
          addi $s2,$s2,1       # str2++
          addi $s1,$s1,1       # str1++
          bne  $t0,$zero,Loop  #
```

- Uses registers **$s0, $s1** and **$t0**
- Remember our convention: callee (the subroutine) must save and restore **$s0-$s7**

strcpy subroutine

```
strcpy:
      addi $sp,$sp,-8   # make room for 2 regs
      sw $s2,4($sp)     # save $s2
      sw $s1,0($sp)     # save $s1
      add $s1,$a0,$zero #
      add $s2,$a1,$zero #
Loop:
      lb    $t0,0($s2)        # $to = *str2
      sb    $t0,0($s1)        # *str1 = $t0
      addi  $s2,$s2,1         # str2++
      addi  $s1,$s1,1         # str1++
      bne   $t0,$zero,Loop    # jump if not done

      lw $s1,0($sp)           # restore $s0
      lw $s2,4($sp)           # restore $s1
      addi $sp,$sp,8          # adjust stack
      jr $ra                  # return
```


Recursive Exercise

Write the MIPS Assembly Language code for the following C program:

```
int factorial( int x ) {
  if (x<1) return 1;
  else return x * factorial(x-1);
}
```

Recursion - Issues

- Since this subroutine calls another subroutine (in this case it calls itself!):
  - we need to save **`$ra`**
  - we need to save any temp registers (**`$t0-$t7`**) before calling a subroutine.
    - only if we need the value of a temp register to still be the same after the call!

Outline of **`factorial`** subroutine

- save registers **`$ra`**, and **`$a0`** (the argument **`x`**)
- check to see if **`x<1`**, if so just return 0
- if **`x>=1`**:
  - call **`factorial(x-1)`** and put result in **`$a1`**
  - put **`x`** in **`$a0`**
  - call multiply: result in **`$v0`**
  - restore **`$ra`** and **`$a0`**
  - return

`factorial` (part 1)

```
factorial:
     # make room for 2 registers
     addi $sp,$sp,-8

     # save $ra and $a0 on stack
     sw $a0,4($sp)
     sw $ra,0($sp)

     slti $t0,$a0,1     # is x < 1 ?
     bne $t0,$zero,L1   # yes - go to L1
```

factorial (part 2) when x>=1

```
sub $a0,$a0,1      # x--;
jal factorial      # call fact(x-1)

# Now multiply the result by x
# a0 is no longer x,
#    but we still have it on the stack

lw $a0,4($sp)
add $a1,$v0,$zero  # $v0 is fact(x-1)
jal multiply       # get the product
```


factorial (part 3)

```
# restore $a0 and $ra before returning
# multiply may have changed $a0
#    (so we must restore again)
# $v0 is already the return value

  lw $ra,0($sp)   # restore $ra
  lw $a0,4($sp)   # restore $a0
  add $sp,$sp,8   # restore the stack
  jr $ra
```


factorial (part 4) x<1

```
L1:
  # x<1 so we just return 1
  addi $v0,$zero,1
  # $a0 and $ra have not changed,
  # so there is no need to restore
  # but we need to restore the stack
  add $sp,$sp,8
  jr $ra
```

Exercise: Simulate `factorial(3)`

- Step through the code, keeping track of:
  - all the used registers
  - $sp and the contents of the stack
- Spim makes this easy!

What about saving `$t0-$t7`?

- The convention says we should expect subroutines to use `$t0-$t7`.
- If we use them and need the value to be the same after a subroutine call – we need to save them before calling the subroutine.
- We also need to restore them after calling the subroutine.

Saving registers

- The code is the same – use the stack:

```
add $sp,$sp,-4
sw $t0,0($sp)

jal whatever

lw $t0,0($sp)
add $sp,$sp,4
```

```
add $sp,$sp,-8
sw $t1,4($sp)
sw $t0,0($sp)

jal whatever

lw $t1,4($sp)
lw $t0,0($sp)
add $sp,$sp,4
```

Writing & Calling Subroutines

- When calling a subroutine:
  - you don't need to worry about **`$s0-$s7`**, they won't change.
  - you do need to worry about **`$t0-$t7`** they may change.
- When writing a subroutine:
  - you need to save/restore the callers **`$s0-$s7`** if you use them.
  - **`$t0-$t7`** are always free to use

More Writing Subroutines

- Careful with **`$ra`** – if you call a subroutine this will change **`$ra`** (and your return won't work!). Might need to save/restore **`$ra`**.
- Careful with **`$a0-$a4`** and **`$v0-$v1`**.
- ALWAYS: Make sure **`$sp`** is the same as when you were called!!!!

Pseudoinstructions

- There are many instructions you can use in MIPS assembly language that don't really exist!
- They are a *convienence* for the programmer (or compiler) – just a shorthand notation for specifying some operation(s).

MIPS **`move`** pseudoinstruction

`move` ***`destreg, sourcereg`***

There is no move instruction, but the assembler lets us pretend.

The assembler can achieve this using **`add`** and **`$zero:`**

`move $s0, $s1` is really **`add $s0,$s1,$zero`**

`blt` revisited

- *Branch if less than* is a pseudoinstruction based on slt and bne:

`blt $s0,$s1,foo` is really

```
slt $at,$s0,$s1
bne $at,foo
```

Register **`$at`** is reserved for use by the assembler (we can't use it – the assembler needs it for pseudoinstructions).

Some useful pseudoinstructions

`li`	load immediate
`la`	load address
`sgt, sle, sge`	set if greater than, …
`bge, bgt, ble, blt`	conditional branching