MIPS Procedures

Overview:

There are only two instructions necessary for creating and calling functions: `jal` and `jr`. If you follow register conventions when calling functions, you will be able to write much simpler and cleaner MIPS code.

Part I: Know your conventions:

1) How should `$sp` be used? When do we add/subtract from the stack pointer?

We subtract when we need to save new stuff. We add after restoring. (see lecture notes for a great description of it in action)

2) Which registers need to be saved before using `jal`?

Any non-s* register ($t*, $v*, $a*) if we plan to use it again. And always save $ra

3) Which registers need to be saved before using `jr`?

None. :p But we should restore any s* we modified.

4) How do we pass arguments into functions?

$a0 is first, $a1 is second, $a2 is third, $a3 is fourth.

5) What do we do if there are more than four arguments we wish to pass?

Spill it to the stack

6) How are values returned by functions?

Return value is in $v0. Another is allowed in $v1 but C doesn't do that

Part II: More Conventions:

When calling a function in MIPS, who needs to save the following variables to the stack? Answer **R** for the caller, **E** for the callee, or **N** for neither.

`$v0-$v1`	`$a0-$a3`	`$t0-$t9`	`$s0-$s7`	`$sp`	`$ra`
R	**R**	**R**	**E**	**N**	**R**

Now assume our function `foo` calls another function `bar`, which is known to call other functions. `foo` takes one argument and uses `$t0` and `$s0`. `bar` takes two arguments, returns an integer, and uses `$t0-$t1` and `$s0-$s1`.

In the boxes below, draw a possible ordering of the stack just before `bar` calls a function (you may not need all the spaces). The top of the left box is the address of `$sp` when `foo` is first called and the top of the right box follows directly after the bottom of the left box. Add "(f)" if the register is stored by `foo` and "(b)" if the register is stored by `bar`. The first one is written in for you.

$ra (f)
$s0 (f)
$v0 (f)
$a0 (f)
$t0 (f)
$ra (b)
$s0 (b)
$s1 (b)

$v0 (b)
$a0 (b)
$a1 (b)
$t0 (b)
$t1 (b)

Part III: Your very own guide to writing functions:

If you plan on calling other functions or using saved registers, you'll need to use the following function template. But wait! There are lines missing. Fill in the blanks:

Prologue:	FunctionFoo:
	`# begin by reserving space on the stack:`
	`addiu $sp, $sp, -FrameSize #blank 1`
	`# now, store needed registers:`
	`sw $ra, 0($sp) #blank 2`
	`sw $s0, 4($sp) #blank 3`
	`… save the rest of the registers …`
	`sw $sx, FrameSize – 4($sp)  #blank 4`

Body:	`… The cleanest, most elegant, and most well commented MIPS code ever, all written by you to do as the function intends :) …`

Epilogue:	`# restore registers:`
	`lw $sx, FrameSize -4($sp) #blank 5`
	`… load the rest of the registers…`
	`lw $s0,` **`4($sp)`** `#blank 6`
	`lw` **`$s0,`** `0($sp) #blank 7`
	`# release stack spaces:`
	`addiu $sp, $sp, FrameSize #blank 8`
	`# finally, return to normal execution:`
	`jr $ra #blank 9`

Part IV: CS61B meets MIPS:

1) Write an insertion sort function in MIPS that uses a swap function to accomplish the task of sorting an array of integers. The arguments to the function should be an integer array and its size. Here is the C version of the function:

```
void insertionSort(int * arr, int size) {
   int i, j;  //Use i=$t0 and j=$t1
   for(i=1;i<size;i++)
   {
      j=i;
      while(j>0 && arr[j]<arr[j-1])
      {
         swap(arr, j, j-1);
         j--;
      }
   }
}

void swap(int * arr, int i1, int i2) {
   int temp=arr[i1];  //Use temp=$t0
   arr[i1]=i2;
   arr[i2]=i1;
}
```

```
*** YOUR MIPS CODE HERE ***
Swap:
     sll $a1, $a1, 2
     sll $a2, $a2, 2
     addu $a1, $a0, $a1
     addu $a2, $a0, $a2
     lw $t0, 0($a1)
     lw $t1, 0($a2)
     sw $t0, 0($a2)
     sw $t1, 0($a1)
     jr $ra

InsertionSort:
     addiu $sp, $sp, -20
     sw $s0, 0($sp)
     sw $s1, 4($sp)
     sw $s2, 8($sp)
     sw $s3, 12($sp)
     sw $ra, 16($sp)
     move $s2, $a0
```

```
      move $s3, $a1
      addiu $s0, $0, 1

forLoop:
      slt $t0, $s0, $s3
      beq $t0, $0, forLoopEnd
      move $s1, $s0

whileLoop:
      slt $t0, $0, $s1
      beq $t0, $0, whileLoopEnd
      sll $t0, $s1, 2
      addu $t1, $s2, $t0
      lw $t0, 0($t1)
      lw $t1, -4($t1)
      slt $t0, $t0, $t1
      beq $t0, $0, whileLoopEnd

      move $a0, $s2
      move $a1, $s1
      addiu $a2, $s1, -1
      jal Swap
      addiu $s1, $s1, -1
      j whileLoop

whileLoopEnd:
      addiu $s0, $s0, 1
      j forLoop

forLoopEnd:
      lw $s0, 0($sp)
      lw $s1, 4($sp)
      lw $s2, 8($sp)
      lw $s3, 12($sp)
      lw $ra, 16($sp)
      addiu $sp, $sp, 20
      jr $ra
```

Part IV, Continued:

2) Why did we have to save registers to the stack in our code?

Because swap would blow them out – and clobber $ra

3) How did using `jal` and `jr` make life easier?

Otherwise, swap would need to know where to jump back to! We'd have to hardcode function returns.

4) Compare: branching/jumping in a loop and jumping for function calls.

One is static (labels) and one is dynamic (functions)