Model 3: Proposal #3 for translating procedure definitions and calls

Now consider a slightly different program and its translation.

High level code	With proposal #2	With proposal #3

`// definition`
`int plus_4(int x) {`
`  return x+4;`
`}`
`…`
`// call the function and use`
`// return value`
`x[1] = plus_4(x[0]);`
	`# definition`
	`plus_4:`
	`addi $s0, $a0, 4`
	`add $v0, $zero, $s0`
	`jr $ra`
	
	`# call the function and`
	`# use return value`
	`lw $a0, 0($s0)`
	`jal plus_4`
	`sw $v0, 4($s0)`
		`# definition`
		`plus_4:`
		`SAVE $s0`
		`addi $s0, $a0, 4`
		`add $v0, $zero, $s0`
		`RESTORE $s0`
		`jr $ra`
		
		`# call the function and`
		`# use return value`
		`lw $a0, 0($s0)`
		`jal plus_4`
		`sw $v0, 4($s0)`

assume register `$s0 holds the address of array x`

legend: SAVE $r – record $r's current value somewhere in memory
RESTORE $r – take the value that was previously saved and put it in $r

1. In "With proposal #2", describe what the `lw` and `sw` instructions are each doing. Be specific to this program.

the `lw` is doing:

the `sw` is doing:

2. Lets suppose the array is x = {4, 1, 1, 1, 1}. The array starts at address 0x00. What is the contents of the array after we run the program for…

the "High level code": x = { , , , , }

the "With proposal #2": x = { , , , , }

3. The difference in results indicates our translation is buggy. Find the bug and describe it here.

Read this!

You just uncovered an example of another one of the limitations of naïve translations of procedure calls to MIPS.

4. How does the program in "With proposal #3" solve the bug that you uncovered in "With proposal #2"?

Read this!

Callee refers the function *being* called. **Caller** refers the code that calls the function. By *convention* $s_ registers are treated as ***callee-saved registers***. In other words, if a function writes to an $s_ register, it must first SAVE its current value. And, before returning, the function must RESTORE the value of that register. In this way, the caller never notices that the register got overwritten temporarily. If a caller needs to keep data around after a function call, then it should put that data in an $s_ register.

5. Why is it so important that MIPS has the convention described above?

6. What registers should be saved?

Exercises

7. Consider the following procedure definition in MIPS. Say which, what registers *must* be saved before the procedure runs and restored before it returns? For each register, say YES (must be saved/restored) or NO and write why.

	YES or NO?	Why or why not?

`mystery:`
`# ... SAVE registers here ...`
 	
`addiu $t0, $a0, 0`
`andi $s0, $t0, 0xFF`
`ori $a0, $zero, 0`
`jal some_other_procedure`
`addu $s1, $s0, $v0`
`addu $v0, $s2, $s1`
 	
`# ... RESTORE registers here ...`
`jr $ra`
	`$t0`
	
	`$a0`
	
	`$zero`
	
	`$s0`
	
	`$s1`
	
	`$s2`
	
	`$ra`
	
	`$v0`	

Extension Questions

8. Can you think of an alternative convention that would achieve the same effect as the one you talked about in #22?

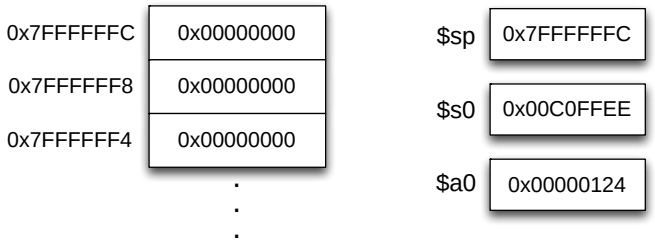
Model 4: The Stack

It turns out that the approach of Proposal #3 will be sufficient for translating any function definitions and function calls to MIPS. Now let's figure out how to implement SAVE and RESTORE.

Code

```
mysteryFunction:
addiu $sp, $sp, -4
sw $s0, 0($sp)
sll $s0, $a0, 1
addi $v0, $s0, $zero
lw $s0, 0($sp)
addiu $sp, $sp, 4
```

State of memory and registers right before we jump to mysteryFuction



9. Redraw the whole memory/registers diagram after this code runs.

```
addiu $sp, $sp, -4
sw $s0, 0($sp)
```

10. Starting from your answer in #25, redraw the whole memory/registers diagram again after this code runs.

```
sll $s0, $a0, 1
addi $v0, $s0, $zero
```

11. Starting from your answer in #26, redraw the whole memory/registers diagram again after this code runs.

```
lw $s0, 0($sp)
addiu $sp, $sp, 4
```

12. Describe in words what happened to the value of the $s0 register over the course of #25-27.

13. Describe in words how the program used the $sp register. Relate your answer to #28.

Read this!

We call the region of memory in that diagram ***the stack***, and we call $sp the ***stack pointer***.

14. Why are stack and stack pointer good names for these two things?

Exercises

15. Let's try using the stack in another example. Consider the mult_by_2 program, now with SAVE and RESTORE implemented. Assume execution starts on line 16. In the table draw the values of all registers and memory as they change over time.

	1 # definition 2 plus_4:
0x404 0x408	3 # prolog (saving register) 4 addiu $sp, $sp, -4 5 sw $s0, 0($sp)
0x40C 0x410	6 # body (doing the work) 7 addi $s0, $a0, 4 8 add $v0, $zero, $s0
	9 # epilog (restoring register)

```
0x414      10 lw $s0, 0($sp)
0x418      11 addiu $sp, $sp, 4
0x41C      12 jr $ra
           13 ...

           14 # call the function and
           15 # use return value
0x500      16 lw $a0, 0($s0)
0x504      17 jal plus_4
0x508      18 sw $v0, 4($s0)
```

(write in a new value to the right each time the value in that register or memory location changes. The '?' means we just don't know the initial value of that register or memory location.)

$s0	0x10								
$a0	?								
$v0	?								
$sp	0xFFC								
$ra	?								

0xFFFC	?								
0xFFF8	?								
0xFFF4	?								
0xFFF0	?								

Model 5: Recursion

The combination of procedure calling convention and the stack is very powerful! Let's see what it can do.

Here is a software implementation of integer multiplication using recursion.

High level code	Translation to MIPS

`// x*n for positive n`
`int multbyn(int x, int n) {`
`  if (n<=1) return 0;`
`  int result = multbyn(n-1);`
`  else return x + result;`
`}`
	`multbyn:`
	`# prolog`
	`# "push" registers onto the stack`
	`addiu $sp,$sp,-8`
	`sw $ra, 0($sp)`
	`sw $s0, 4($sp)`
	`move $s0, $a0`
	
	`# body`
	`ble $a1, 1, return0`
	`addiu $a1, $a1, -1`
	`# recursive call`
	`jal multbyn`
	`# set return value`
	`add $v0, $s0, $v0`

	`j end`
	
	`return0:`
	`# set return value`
	`addiu $v0, $zero, 0`
	
	`end:`
	`# epilog`
	`# "pop" registers off the stack`
	`lw $ra, 0($sp)`
	`lw $s0, 4($sp)`
	`addiu $sp,$sp,8`
	`j $ra`

16. **Individually.** Write on the diagram to match each line of MIPS code to a specific part of the Java code. If there are lines you cannot match, mark them with a question mark (?).

17. **Check with your team.** Discuss the matches you made and see if you can resolve the question marks. If you cannot, consult with an instructor.

Read this!

Although the code above looks complicated, translating recursive code is ***no different*** than translating other functions and code that calls functions. When you are translating a procedure: (1) translate the ***body*** first, (2) use the body to determine the registers that need to be saved/restored, (3) finally, write the ***prolog*** and ***epilog***, ensuring the epilog undoes the actions of the prolog.

Exercises

Now it is your turn to translate some recursive functions to MIPS code.

18. add 1 to each array element in-place (recursive)

a) Java implementation

```
void add1(int[] x, int size)  {
```

b) translation to MIPS

```
addiu $sp,$sp,-4
sw $ra,0($sp)

beq $a1,$0,because
addiu $a1,$a1,-1
sll $t0,$a1,2
addu $y0,$t0,$a0
lw $t1,0($t0)
addiu $t1,$t1,1
sw $t1,0($t0)
jal add1

because:
lw $ra,0($sp)
addiu $sp,$sp,4
jr $ra
```

19. return the sum of the elements in the array (recursive)

a) Java implementation

```
int sum(int[] x, int size) {
```

```
lw $ra,0($sp)
addiu $sp,$sp,4
jr $ra
```

```
def fib(n)
if n==1:return 1
if n==0; return 0
return fib(n-1)+fib(n-2)
```

b) translation to MIPS