CSE 30321 – Lecture 07-09 – In Class Example Handout

Part A: A Simple, MIPS-based Procedure:

Swap Procedure Example:

Let's write the MIPS code for the following statement (and function call):

```
if (A[i] > A [ i+1])              // $s0 = A
      swap (&A[i], &A[i+1])       // $t0 = 4*i
```

We will assume that:

- The address of A is contained in $s0 ($16)
- The index (4 x i) will be contained in $t0 ($8)

Answer:

The Caller:

```
        …
        // Calculate address of A(i)
        add     $s1, $s0, $t0        // $s1 ← address of array element i in $s1

        // Load data
        lw      $t2, 0($s1)          // load A(i) into temporary register $t2
        lw      $t3, 4($s1)          // load A(i+1) into temporary register $t3

        // Check condition
        ble     $t2, $t3, else       // is A(i) <= A(i+1)?  If so, don't swap

        // if >, fall through to here…
        addi    $a0, $t0, 0          // load address of x into argument register (i.e. A(i))
        addi    $a1, $t0, 4          // load address of y into argument register (i.e. A(i+1))

        // Call Swap function
        jal     swap                 // PC ← address of swap; $ra l $31 = PC + 4
else:
```

// Note that swap is "generic" – i.e. b/c of the way data is passed in, we do not assume the values
// to be swapped are in contiguous memory locations – so two distinct physical addresses are
// passed in

Swap:

```
        lw $t0, 0($a0)          // $t0 = mem(x) – use temporary register
        lw $t1 0($a1)           // $t1 = mem(y) – use temporary register
        sw 0($a0), $t1          // do swap
        sw 0($a1), $t0          // do swap
        jr $ra                  // $ra should have PC = PC + 4
                                // PC = PC + 4 should be the next address after jump to swap
```

Part B: Procedures with Callee Saving (old exam question):

Assume that you have written the following C code:

```
//-------------------------------------------------------------------
int   variable1   = 10;                    // global variable
int   variable2   = 20;                    // global variable
//-------------------------------------------------------------------
int main(void) {
      int i       = 1;                     // assigned to register s0
      int j       = 2;                     // assigned to register s1
      int k       = 3;                     // assigned to register a3
      int m;
      int n;

      m = addFourNumbers(i, j);

      n = i + j;                           // 1 + 2    = 3

      printf("m is %d\n", m);              // printf modifies no registers
      printf("n is %d\n", n);              // printf modifies no registers
      printf("k is %d\n", k);              // printf modifies no registers
}
//---------------------------------------------------------------------
int addFourNumbers(int x, int y) {
      int i;                               // assigned to register s0
      int j;                               // assigned to register s1
      int k;                               // assigned to register s2

      i     = x + y;                       // 1 + 2    = 3
      j     = variable1 + variable2;       // 10 + 20  = 30
      k     = i + j;                       // 3 + 30   = 33

      return k;
}
//---------------------------------------------------------------------

The output of the printf statements in main is: m is 33
                                                n is 3
                                                k is 3
```

Assume this program was compiled into MIPS assembly language with the register conventions described on Slide 12 of Lecture 07/08. Also, note that in the comments of the program, I have indicated that certain variables will be assigned to certain registers when this program is compiled and assembled. Using a callee calling convention, answer the questions below:

Q-i: Ideally, how many <u>arguments</u> to the function addFourNumbers must be saved on the stack?

0. By default, arguments should be copied into registers.

Q-ii: What (if anything) should the assembly language for main() do right before calling addFourNumbers?

Copy values of s registers into argument registers; save value of k (in $a3) onto the stack

Q-iii: What is the first thing that the assembly language for addFourNumbers should do upon entry into the function call?

Callee save the s registers

Q-iv: What is the value of register number 2 (i.e. 0010_2) after main completes (assuming there were no other function calls, no interrupts, no context switches, etc.)

33. Register 2 = v0. It should *not* have changed.
(different answer if you assume printf returns value)

Q-v: Does the return address register ($ra) need to be saved on the stack for this program? Justify your answer. (Assume main() does not return).

No – if no other procedures are called.

Part C: Procedures with Callee Saving (old exam question):

Assume you have the following C code:

```
int main(void) {
        int x = 10;                     # x maps to $s1
        int y = 20;                     # y maps to $s2
        int z = 30;                     # z maps to $s3

        int a;                          # a maps to $t0
        int b;                          # b maps to $t1
        int c;                          # c maps to $t2

        c = x + y;

        a = multiply(x, z);

        b = c + x;
}

int multiply(int a, int b) {
        int q;                          # q maps to $t0
        int z;          <-- [■]         # z maps to $t1

        q = add();
        z = a*b;        <-- [▲]

        return z;
}

int add() {
        int m = 5;                      # m maps to $t4
        int n = 4;                      # n maps to $t5
        int y;
        y = a + b;
        return y;
}
```

Assuming the MIPS calling convention, answer questions A-E. Note – no assembly code/machine instructions are required in your answers; simple explanations are sufficient.

Q-i: What, if anything must main() do before calling multiply?

- **Save \$t2 to stack, needed upon return.**
- **Also, copy \$s1 to \$a0 and copy \$s3 to \$a1**

Q-ii: Does multiply need to save anything to the stack? If so, what?

- **\$31**
- **The s registers associated with main()**
- **The argument registers passed into multiply before calling add()**

Q-iii: Assume that multiply returns its value to main() per the MIPS register convention. What machine instructions might we see at ▲ to completely facilitate the function return?

- **We have to copy the value in \$t1 to \$2.**
- **We would call a jal instruction**
- **We would adjust the stack pointer, restored saved registers.**

Q-iv: What line of code should the return address register point to at ▲?

- **b = c + x**

Other answers were considered correct based on stated assumptions.

Q-v: What line of code should the return address register point to at ■?

- **b = c + x**

Other answers were considered correct based on stated assumptions.

Part D: More Complex Example:

Let's write the MIPS code for the following:

```
for(i=1; i<5; i++) {
  A(i) = B*d(i);
  if(d(i) >= e) {
    e = function(A,i);
  }
}
```

```
int function(int, int) {
  A(i) = A(i-1);
  e = A(i);
  return e;
}
```

```
Assume:
Addr. of A = $18
Addr. of d = $19
B = $20
e = $21
```

(We pass in starting "address of A" and "i")

Question/Comment	My Solution	Comment
1st, want to initialize loop variables. **What registers should we use, how should we do it?**	addi $16, $0, 1 addi $17, $0, 5	# Initialize i to 1 # Initialize $17 to 5 (in both cases, *saved* registers are used – we want this data available post function call)

2nd, calculate address of d(i) and load. **What kind of registers should we use?**	Loop: sll $8, $16, 2
	add $8, $19, $8
	lw $9, 0($8)
		# store i*4 in $8 (temp register OK)
		# add start of d to i*4 to get address of d(i)
		# load d(i) → needs to be in register to do math

Calculate B*d(i)	mult $10, $9, $20	# store result in temp to write back to memory

Calculate address of A(i)	sll $11, $16, 2
	add $11, $11, $18
	
	CANNOT do:
	add $11, $8, $18
		# Same as above
		
		# We overwrote
		# But, would have been better to save i*4
		Why? Lower CPI

Store result into A(i)	sw $10, 0($11)	# Store result into a(i)
Now, need to check whether or not d(i) >= e. **How? Assume no ble.**	slt $1, $9, $22 bne $0, $1, start again	# Check if $9 < $22 (i.e. d(i) < e) # Still OK to use $9 → not overwritten # (temp does not mean goes away immediately) # if d(i) < e, $1 = 1 # if d(i) >= e, $1 = 0 (and we want to call function) # (if $1 != 0, do not want to call function)

Given the above setup, what comes next? (Falls through to the next function call). **Assume argument registers, what setup code is needed?**	add $4, $18, $0
	add $5, $16, $0
	x: jal function
		# load address of (A) into an argument register
		# load i into an argument register
		# call function; $31 ← x + 4 (if x = PC of jal)

Finish rest of code: **What to do?** Copy return value to $21. Update counter, check counter. **Where is "start again" at?**	add $21, $0, $2 sa: addi $16, $16, 1 bne $16, $17, loop	# returned value reassigned to $21 # update i by 1 (array index) # if i < 5, loop A better way: Could make array index multiple of 4
Function Code		

Assume you will reference A(i-1) with lw … 0($x). **What 4 instruction sequence is required?**	func: subi $5, $5, 1
	sll $8, $5, $2
	add $9, $4, $8
	lw $10, 0($9)
		# subtract 1 from i
		# multiply i by 4 → note
		# add start of address to (i-1)
		# load A(i-1)

Finish up function.	sw $10, 4($9) add $2, $10, $0	# store A(i-1) in A(i) # put A(i-1) into return register ($2)
Return	jr $31	# PC = contents of $31