CPSC 213, Winter 2010, Term 1 — Quiz 3 **Solution**

Date: November 10, 2010; Instructor: Tamara Munzner

1 (22 marks) Function Pointers / Dynamic Dispatch, Static Control Flow, and Writing Assembly Code. Consider the following C code that uses function pointers, with an array of pointers and a procedure that uses it to invoke one of them:

```
void procA () { printf ("A"); }
void procB () { printf ("B"); }
void procC () { printf ("C"); }
void (*proc[3])() = { procA, procB, procC };
int a;
void foo () {
    if (a > 100) proc[0] ();
    else if (a > 0) proc[1] ();
    else if (a == 0) proc[2] ();
}
```

Give SM213 assembly code that implements the procedure call "`foo`", assuming that the labels `$a` and `$proc` have already been created and point to appropriate memory locations for storage. Comment your code.

```
foo:    deca r5          # allocate frame
        st r6, 0x0(r5)   # save r6 on the stack
        ld  $a, r0       # r0 = &a
        ld  0x0(r0), r0     # r0 = a
        ld  $proc, r1    # r1 = &proc
        ld  $0xffffff9c, r2   # r2 = -100  (also correct: ld $-100, r2)
        add r0, r2       # r2 = a-100
        bgt r2, gthun    # jump if (a - 100) > 0
        bgt r0, gtzero   # jump if a > 0
        beq r0, eqzero   # jump if a == 0
        br  tdown        # jump to end of if logic since no cases matched
gtehun: ld  0x0(r1), r2 # r2 = proc[0]
        br  end          # jump down to end of if logic
gtzero: ld  0x4(r1), r2 # r2 = proc[1]
        br  end          # jump down to end of if logic
eqzero: ld  0x8(r1), r2 # r2 = proc[2]
end:    gpc r6           # r6 = pc
        inca r6          # r6 = return address
        j   0x0(r2) # goto m[proc[a]], indirect (dereferenced above)
tdown:  ld  0x0(r5), r6 # restore r6 from stack
        inca r5          # tear down frame
        j   0x0(r6)      # return to foo's caller
```

also correct, variant for lines starting at gthun:

```
gthun:  ld $0x0, r2 # r2 = 0
        br end      # jump down to end of if logic
gtzero: ld $0x1, r2 # r2 = 1
        br end      # jump down to end of if logic
eqzero: ld $0x2, r2 # r2 = 2
end:    gpc r6      # r6 = pc
        inca r6     # r6 = return address
        j *(r1,r2,4)# goto m[proc[a]], indirect indexed
tdown:  ld  0x0(r5), r6 # restore r6 from stack
        inca r5          # tear down frame
        j   0x0(r6)      # return to foo's caller
```

also correct, variant for same lines:

```
gthun:  gpc r6      # r6 = pc
        inca r6     # r6 = return address
        jmp *0(r1)  # goto m[proc[a]], indirect offset
        br tdown        # jump down to to teardown/return
gtzero: gpc r6      # r6 = pc
        inca r6     # r6 = return address
        jmp *4(r1)  # goto m[proc[a]], indirect offset
        br tdown    # jump down to teardown/return
eqzero: gpc r6      # r6 = pc
        inca r6     # r6 = return address
        jmp *8(r1)  # goto m[proc[a]], indirect offset
tdown:  ld  0x0(r5), r6 # restore r6 from stack
        inca r5          # tear down frame
        j   0x0(r6)      # return to foo's caller
```

Note: a lot of people are still confused about where in the code setup/teardown of the stack needs to happen. The key idea to understand is what does the caller do vs. what does the callee do: the *caller* does setup/teardown for arguments, and does setup/teardown for saving the old r6 return value to the stack, and does the setup of the new r6 return address and the actual jump. In contrast, the *callee* does setup/teardown for locals. This can all be confusing because in many cases a function is both a callee (it's called from some other function) and a caller (it calls another function). In this case, the foo procedure that you implement is a caller for the proc functions, so it needs to do setup/teardown of the former r6 return value onto the stack before it sets up the new r6 return value for the call to proc. In contrast, in the midterm 6c question you were asked to implement the callee function sum, which was passed arguments by its caller and did not call any other function, thus there was no setup/teardown.

2 **(4 marks)** **Switch Statements.** A `switch` statements can be implemented using a jump table or a sequence of `if`

statements.

2a Explain the benefit of using jump tables over if statements (just the benefit; not how they work).

The running time of the switch implement with jump tables is independent of the number of case arms, whereas if statements may need to check every alternative in the worst case.

2b Are jump tables always the best implementation choice for every switch statement? Explain carefully.

No. If case labels are sparse, the jump table becomes too large for this approach to be feasible.

3 (6 marks) **I/O.** Programmed IO (PIO) supports IO-mapped memory, where loads and stores into memory addresses beyond the end of main memory are handled by the I/O controllers. Carefully explain a) what are the limitations of programmed IO and b) how can interrupts be used to solve these problems?

a) The limitation of PIO is that the CPU would have to either 1) wait for the IO controllers to carry out the operation, which would waste millions of instructions worth of time and worse yet only one word would be transferred at a time, or 2) keep polling the IO controllers to check if they've finished the operation yet, which would have high overhead if the check is done often but is costly, or high latency if the check isn't done often enough.

b) Interrupts provide a low-overhead way for the CPU to quickly check whether the IO controller has serviced the requests, where a dedicated register makes the cost of the check is low enough that it can be done on every fetch/execute cycle. Interrupts thus allow asynchronous IO where the CPU is not busywaiting while waiting for the IO controllers, and can do other work in the mean time.

These answers are very detailed in order to be a complete guide for studying: full credit was given for just the main ideas in shorter form.