CPSC 313, 05w Term 1— Midterm 1 — Solutions

Date: October 7, 2005; Instructor: Mike Feeley

1. **(10 marks)** Short answers.

1a. What is the advantage of using two different registers (i.e., `%ebp` and `%esp`) to store virtual addresses to the runtime stack?

> The stack pointer changes during the execution of a procedure when temporary values, registers or arguments are saved and restored. Keeping a separate base pointer that does not change during the execution of a procedure allows the compiler to access local variables and formal arguments using static offsets from the base pointer register that are the same everywhere in the procedure.

1b. What does a `call` instruction do that a `jmp` instruction does not?

> It saves the return address (i.e., the virtual address of the next instruction to execute after the procedure `return` statement executes) by pushing it on the stack before jumping to the called procedure.

1c. We discussed two ways to implement a C-language `switch` statement in assembly language. What are they? Under what conditions would one be favoured over the other (both ways)?

> The two methods are nested if-then-else statements and a jump table. Jump tables are faster when there are more than a few case bodies. If-then-else is preferred if there are just a few cases or if the case labels are sparse (i.e., the numeric difference between the largest and smallest case labels is much larger than the number of cases).

1d. Why is it faster to compute the address of an element of an array of structs if the size of each struct is a power of two?

> Accessing an element of an array typically requires multiplying an index value by the element size and then adding this *displacement* to the *base* virtual address of the array (i.e., the virtual address of the first element in the array). Multiplying by a power of two can be performed by shifting left (e.g., `shll`), which is much faster than actually multiplying (e.g., `imull`). Best of all are element sizes of 1, 2, 4 or 8, because access to these arrays can be encoded using the IA32's scaled-displacement addressing mode using a single instruction (e.g., `movl d(rb,ri,s), D`).

1e. Write the two assembly-language instructions that compute "`if (a<=b) goto X`" where `a` and `b` are signed integers stored in registers `%eax` and `%ebx` respectively and `X` is a label.

```
    cmpl    %ebx, %eax       # if a ? b
    jle     X                # if a <= b goto X
```

2. **(4 marks)** Indicate whether each of the following values are determined statically or dynamically. For each, write the word **static** or **dynamic**.

2a. The virtual address of a global variable: **static**

2b. The virtual address of a local variable: **dynamic**

2c. The offset from the register `%ebp` of a local variable: **static**

2d. The virtual address to which control is transferred when a procedure is called: **static**

2e. The virtual address to which control is transferred when a procedure returns: **dynamic**

3. **(4 marks)** Write the assembly-language instructions that compute "`%eax = %eax` × 15 + 15" **most efficiently** (i.e., your could should execute faster than any alternative).

```
    leal    (%eax,%eax,4), %eax      # %eax = %eax * 5
    leal    15(%eax,%eax,2), %eax    # %eax = %eax * 3 + 15
```

4. **(8 marks)** Consider the following C-language procedure. Answer each question with the appropriate IA32 (gas) assembly language. Treat each question in isolation and assume that none of the registers other than `%esp` and `%ebp` hold useful values. **Comment your code.**

```
int A[100];     // a global variable

void foo(int b, int* c)
{
    int d;

    ...
}
```

4a. Give assembly code for `d = b;`

```
    movl    8(%ebp), %eax            # %eax = b
    movl    %eax, -4(%ebp)           # d = b
```

4b. Give assembly code for `d = A[d];`

```
    movl    -4(%ebp), %eax           # %eax = d
    movl    A(,%eax,4), %eax         # %eax = A[d]
    movl    %eax, -4(%ebp)           # d = A[d]
```

4c. Give assembly code for `*c = *c + d;`

```
    movl    -4(%ebp), %eax           # %eax = d
    movl    12(%ebp), %ebx           # %ebx = c
    addl    (%ebx), %eax             # %eax = *c + d
    movl    %eax, (%ebx)             # *c = *c + d
```

5. **(9 marks)** Consider the following assembly-language procedure.

```
        # prologue omitted
        movl    8(%ebp), %ebx        # %ebx = a1 (int a1[])
        movl    12(%ebp), %ecx       # %ecx = a2
        movl    $0, %edx             # %edx = 0
        movl    $0, %eax             # %eax = 0
        cmpl    %ecx, %edx           # if %edx ? %ecx
        jge     .L2                  # if %edx >= %ecx goto .L2
.L0:    cmpl    $0,(%ebx,%ecx,4)     # if a1[a2] ? 0
        jle     .L1                  # if a1[a2] <= 0 goto .L1
        addl    $1, %eax             # %eax = %eax + 1
.L1:    addl    $1, %edx             # %edx = %edx + 1
        cmpl    %ecx, %edx           # if %edx ? a2
        jl      .L0                  # if %edx < a2 goto .L0
.L2:    # epilogue omitted
```

5a. Comment the code above and then explain what this procedure does (pseudo-code not necessary).

This is a function `f(int a[], int i)` that returns `0` if `a[i]`$\leq$`0` and `i` otherwise.

6. **(6 marks)** Now consider this piece of code.

```
        .section .rodata
.L0:    .long   .L1             # .L0[0] = .L1
        .long   .L2             # .L0[1] = .L2
        .long   .L3             # .L0[2] = .L3
        .section .text
        # some stuff left out
        movl    $1, %eax
        jmp     *.L0(,%ebx,4)   # goto .L0[%ebx]
.L1:    sall    $1, %eax        # %eax = %eax * 2
.L2:    sall    $1, %eax        # %eax = %eax * 2
.L3:    sall    $1, %eax        # %eax = %eax * 2
```

6a. Comment the code above and then explain what this procedure does (pseudo-code not necessary).

If `0`$\leq$`%ebx`$\leq$2, the code computes %eax = %eax * $2^{\%ebx+1}$. Otherwise, it branches to an unknown address and likely generates an address fault (i.e., crashes and dumps core).

7. **(9 marks)** Consider this C-language code. Assume that one caller-save register (i.e., `%ecx`) and one callee-save register (i.e., `%ebx`) must be saved/restored for `foo()` to call `bar()`. Answer the following three questions with **commented** assembly code.

```
void foo(int i, int j) {
    bar(i,j);
}
```

7a. Give the assembly code of the procedure-call statement `bar(i,j)`.

```
    pushl   %ecx            # save caller-save reg %ecx
    pushl   12(%ebp)        # push arg2 = j onto stack
    pushl   8(%ebp)         # push arg1 = i onto stack
    call    bar
    addl    $8, %esp        # move stack ptr to reg save area
    popl    %ecx            # restore caller-save reg %ecx
```

7b. Give the assembly code of `bar`'s *prologue*.

```
    pushl   %ebp            # save frame pointer
    movl    %ebp, %esp      # create new stack frame
    pushl   %ebx            # save callee-save reg %ebx
```

7c. Give the assembly code of `bar`'s *epilogue*. Assuming nothing about the current value of the stack pointer when the epilogue starts executing.

```
    leal    -4(%ebp), %esp  # move stack ptr to reg save area
    popl    %ebx            # restore callee-save reg %ebx
    popl    %ebp            # restore old stack frame
    ret
```