CPSC 313, 06w Term 1— Midterm 1 — Solutions

Date: October 4, 2006; Instructor: Norm Hutchinson

1. **(8 marks)** Short answers.

1a. **(2 marks)** Is the address of a local variable in a C function determined *statically* or *dynamically*? Briefly explain.

> It is determined *dynamically*, as it is allocated on the stack.

1b. **(2 marks)** Is the code address to which a procedure returns when it exits determined *statically* or *dynamically*? Briefly explain.

> It is *dynamically* determined when the procedure is called during execution of the program, by pushing the return address on the stack.

1c. **(2 marks)** Does the IA32 instruction-set architecture require that `%esp` be used as the stack pointer? Briefly explain.

> It does require it. The `call` and `ret` instructions implicitly refer to the `%esp` register and there are no reasonable alternatives that compiled code can use if chooses to use a different register to point to the stack.

1d. **(2 marks)** Give assembly-language code that computes `%eax = %eax * 9 + 7` as efficiently as possible.

```
    leal    7(%eax,%eax,8), %eax        # eax = eax * 9 + 7
```

2. **(8 marks)** Consider the following C source file.

```
/* global variables */
int g, *gp, **gpp;

void foo (int *a1, int a2) {
    int l;
    /* consider each statement as if it were here */
}
```

Give an assembly-code implementation of each of the following statements of function `foo()`. Consider each statement in isolation (i.e., as if it were the only statement of foo). Do not assume that variables start out in registers. Be sure to write results to the appropriate location in memory. Assume that the local variable `l` is in memory (not in a register). A fully correct answer will use as few instructions as possible. **Comment your code.**

2a. **(2 marks)** `gp = &l;`

```
     leal   -4(%ebp), %eax  # eax = &l
     movl   %eax, gp        # gp = &l
```

2b. **(2 marks)** `l = *a1 * a2;`

```
     movl   8(%ebp), %eax   # eax = a1
     movl   (%eax), %eax    # eax = *a1
     imull  12(%ebp), %eax  # eax = *a1 * a2
     movl   %eax, -4(%ebp)  # l = *a1 * a2
```

2c. **(2 marks)** `**gpp = 3;`

```
     movl   gpp, %eax        # eax = gpp
     movl   (%eax), %eax     # eax = *gpp
     movl   $3, (%eax)       # **gpp = 3
```

2d. **(2 marks)** `if (a2 > g) l = a2 else l = g;`

```
     movl   12(%ebp), %eax   # eax = a2
     cmpl   g, %eax          # (a2 ? g)
     jg     .L0              # if (a2 > g) goto .L0 (then-part)
     movl   g, %eax # %eax is now g rather than a2
.L0: movl   %eax, -4(%ebp)   # l = either a2 or g
```

3. **(8 marks)** Consider the following assembly language code.

3a. **(4 marks)** Comment every line of this code carefully.

```
foo:
    pushl   %ebp             # prologue
    movl    %esp, %ebp
    movl    8(%ebp), %eax    # arg 0 in %eax
    movl    12(%ebp), %edx   # arg 1 in %edx
    cmpl    %edx, %eax       # if the same, return arg 0
    je  L8
L6:
    cmpl    %edx, %eax       # if arg0 > arg1
    jle L4
    subl    %edx, %eax       # arg0 -= arg1
    jmp L2
L4:
    subl    %eax, %edx       # arg1 -= arg0
L2:
    cmpl    %edx, %eax       # if arg0 != arg1
    jne L6                   # loop
L8:
    popl    %ebp             # epilogue
    ret
```

3b. **(4 marks)** Give an equivalent C-language function.

```
    int gcd (int a, int b)
    {
        while (a != b) {
            if (a > b) {
                a -= b;
            } else {
                b -= a;
            }
        }
        return a;
    }
```

4. **(10 marks)** Consider this C procedure

```
int foo(int i)
{
```

```
    switch (i) {
        case 1:
            i = i * 2;
            break;
        case -1:
        case -4:
            i = i - 7;
            break;
        default:
            i = 0;
    }
    return i;
}
```

Give the assembly code that implements this switch statement using a jump table, in the most efficient manner possible. Include both `.text` and `.rodata` definitions. **Comment your code.**

```
.text
         movl   8(%ebp), %eax        # eax = i
         movl   %eax, %ecx
         subl   $-4, %ecx            # ecx = i-(-4) (base)
         cmpl   $5, %ecx             # if out of range goto default
         ja .L2
         jmp    *.L4(,%ecx,4)        # goto .L4[i-(-4)]
    .L0: sall   $1, %eax             # case 1: i = i * 2
         jmp    .L3
    .L1: subl   $7, %eax             # case -1, -4: i = i - 7
         jmp    .L3
    .L2: movl   $0, %eax             # default:  i = 0
    .L3:
.rodata
    .L4: .long  .L1                  # case -4
         .long  .L2                  # case -3 => default
         .long  .L2                  # case -2 => default
         .long  .L1                  # case -1, like -4
         .long  .L2                  # case 0 => default
         .long  .L0                  # case 1
```

5. **(6 marks)** Consider the procedure call to `callee()` in this C code.

```
void caller (int i, int j) {
    int l;

    l = callee (&j);
}
```

Assume that `callee` uses one callee-save register (i.e., `%esi`) and that `caller` has a value in one caller-save register (i.e., `%edx`) that must not be changed by the call to `callee`.

5a. **(4 marks)** Give the assembly code of the procedure call to `callee()`, including storing the result in local variable `l` in memory.

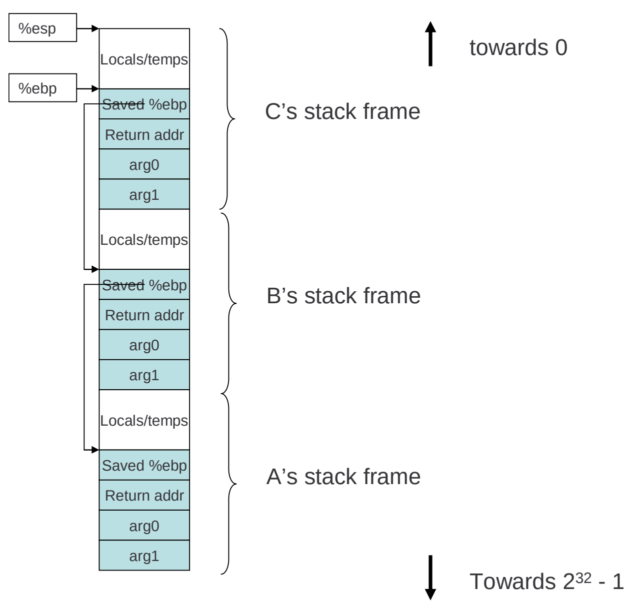
```
    pushl   %edx               # save caller-save register edx
    leal    12(%ebp), %eax     # eax = &j
    pushl   %eax               # push &j as argument
    call    callee             # callee (&j)
    movl    %eax, -4(%ebp)     # l = callee (&j)
    addl    $4, %esp           # discard argument from stack
    popl    %edx               # restore saved register edx
```

5b. **(2 marks)** Give the assembly code of `callee()`'s *prologue*.

```
    pushl   %ebp               # save old base pointer
    movl    %esp, %ebp         # create new frame
    pushl   %esi               # save callee-save register esi
```

6. **(10 marks)** If you think about what we've been doing in this course so far, you may have been getting frustrated that it is all about doing again in assembly language things that you could already do in C. However, assembly code is strictly more powerful than C because you can access information that is not exposed to the C language programmer. This question hints at some of this information.

6a. **(5 marks)** Draw a picture of the runtime stack indicating three procedures, `A` which calls `B` which calls `C`. On your picture, very clearly indicate the locations of the saved frame pointers and return addresses. You should indicate the general location of parameters and local variables, but need not show them in detail. Clearly indicate exactly where in the stack the registers %esp and %ebp point.



6b. **(5 marks)** Write in assembler a function `fetch` that can be called by a function like C and fetches the program counter of C's caller and C's caller's caller. Its prototype is:

```
void fetch(int *callerspc, int *callerscallerspc);
```

Be careful to get the return address of `C`'s caller, and not `fetch`'s caller!

```
    .text
    .p2align 4,,15
.globl fetch
    .type   fetch, @function
fetch:
                            # I don’t push the %ebp
    movl    %ebp, %ecx      # caller’s %ebp is in %ecx
    movl    4(%esp), %edx   # callerspc in %edx
    movl    4(%ecx), %eax   # caller’s caller’s return addr
    movl    %eax, (%edx)    # callerspc = caller’s caller’s return addr

                            # repeat for the caller’s caller
    movl    (%ecx), %ecx    # caller’s callers %ebp is in %ecx
    movl    8(%esp), %edx   # callerscallerspc in %edx
    movl    4(%ecx), %eax   # caller’s caller’s caller’s return addr
    movl    %eax, (%edx)    # callerscallerspc = caller’s caller’s caller’s re

    ret
```