CPSC 313, 04w Term 2 — Midterm Exam 1 — Solutions

1. (8 marks) Short answers.

1a. Assume memory address 0x1000 stores 0x0, 0x1004 stores 0x4 and so on and that `%ebx` stores the value 0x1000. What is the value of `%eax` after each of the following instructions complete?

```
movl %ebx, %eax                  %eax = 0x1000
movl (%ebx), %eax                %eax = 0x0
movl 8(%ebx), %eax               %eax = 0x8
leal 8(%ebx), %eax               %eax = 0x1008
leal 8(%ebx, %ebx, 4), %eax      %eax = 0x5008
```

1b. Consider the following assembly-language code.

```
subl    %ebx, %edx
cmpl    %edx, %ebx
jlt     .L0
```

Briefly explain, using C-language-like pseudo code, what this code does.

```
%edx -= %ebx; if (%ebx < %edx) goto .L0;
```

Is there any simple way to modify this code to make it run faster? If so, what?

Not that I know of. This was a bug in the exam. The second instruction should have been `testl %edx, %edx`, in which case the `testl` would been redundant, because the `subl` would have set the condition codes properly. Sorry for my mistake.

1c. Carefully describe what a *procedure activation* is and how it differs from a *procedure*

A procedure activation is the *execution* of a procedure at runtime. The state of an activation is stored in an *activation record*, also know as a *stack frame* on the *runtime stack*. A *procedure* is the code, a static construct.

1d. Transistors are used to implement gates. Give a simple abstract description of the key feature of a transistor that makes this possible. That is, describe *what* a transistor does, *not how it does it*.

A transistor is a switch.

2. (5 marks) Consider the following C-language data-structure declarations.

```
int A[10][2];
struct { int i; int *p; } S[10];
```

And the following pre-conditions:

1. `%ebx` stores the virtual-memory address of A[0][0]
2. `%ecx` stores the virtual-memory address of S
3. `%edx` stores the value of i

For each of the following C-language statements, give an assembly-language implementation that uses **as few instructions as possible** and leaves the value of `r` in `%eax`, without storing it in memory.

2a. `r = A[0][0]`:

```
movl (%ebx), %eax
```

2b. `r = A[i][1]`:

```
movl 4(%ebx,%edx,8), %eax
```

2c. `r = (int) &A[i][1]`:

```
leal 4(%ebx,%edx,8), %eax
```

2d. `r = *S[0].p`:

```
movl 4(%ecx), %eax
```

2e. `r = *S[i].p`:

```
movl 4(%ecx,%edx,8), %eax
```

3. (5 marks) Consider the following assembly language code.

```
foo:
        # prologue omitted
        movl    8(%ebp), %ebx
        movl    12(%ebp), %eax
        testl   %ebx, %ebx
        jeq     .L1
.L0:    incl    %eax
        decl    %ebx
        jne     .L0
.L1:    # epilogue omitted
        ret
```

Give the most concise C-language program that **could plausibly have been generated by a C compiler** into assembly language of this form (except for the comments).

```
while (i) {
    j++;
    i--;
}
```

4. (5 marks) Given the following C-language `switch` statement.

```
switch (i) {
    10: a+=1;
        break;
    12: a+=a;
    13: a+=b;
        break;
    default:
        a=0;
}
```

And pre-conditions:

1. `%ebx` stores the value of `i`
2. `%ecx` stores the value of `a`
3. `%edx` stores the value of `b`

Complete the following assembly-language implementation of this statement, using a jump table.

```
    .data
.L5:    .long   .L0             # switch=10
        .long   .L3             # switch=11 (default)
        .long   .L1             # switch=12
        .long   .L2             # switch=13
    .text
        subl    $10, %ebx       # i -= 10
        cmpl    $3, %ebx        # compare i to 3
        ja      .L3             # if (i<0 || i>3) goto default
        jmp     *.L5(,%ebx,4)   # goto .L5[i]
.L0:    incl    %ecx            # case 10:
        jmp     .L4
.L1:    addl    %ecx, %ecx      # case 12:
.L2:    addl    %edx, %ecx      # case 13:
        jmp     .L4
.L3:    xorl    %ecx, %ecx      # case default:
.L4:
```

5. (12 marks) Consider the following C program.

```
int A[3][3];
int *B;

int foo (int C[], int i, int j)
{
    int k;
    return A[i][j] + B[i] + C[j] + (int) &k;
}

int main ()
{
    B = (int *) calloc ( sizeof(int), 3*3 );
    printf ("%d\n", foo (B,1,2));
}
```

In the space below give the assembly-language declaration of all variables stored in static data section of the program's executable image (i.e., `.data`); use the "`.skip <number-of-bytes>, 0`" directive to allocate space for each variable.) Then, give a complete assembly language implementation for the procedure `foo` (and don't give anything for `main`). Be sure to include the procedure prologue and epilogue and clearly indicate, using comments, which instructions compute each term of the expression. Comment every line of the assembly-language code you write.

```
    .data
A:      .skip   3*3*4, 0                # int A[3][3]
B:      .skip   1*4, 0                  # int *B
    .text
foo:    # prologue
        pushl   %ebp                    # save old frame pointer
        movl    %esp, %ebp              # make new stack frame
        leal    -4(%esp), %esp          # int k

        # body
        movl    12(%ebp), %ecx          # %ecx =  i
        movl    16(%ebp), %edx          # %edx =  j
        leal    (%ecx,%ecx,2), %eax     # %eax =  i*3
        addl    %edx, %eax              # %eax =  i*3+j
        movl    A(,%eax,4), %eax        # %eax =  A[i][j]
        addl    B(,%ecx,4), %eax        # %eax += B[i]
        movl    8(%ebp), %ecx           # %ecx =  C
        addl    (%ecx,%edx,4), %eax     # %eax += C[j]
        leal    -4(%ebp), %ecx          # %ecx =  &k
        addl    %ecx, %eax              # %eax += &k

        # epilogue
        movl    %ebp, %esp              # (%esp) = saved %ebp
        popl    %ebp                    # restore callers frame
        ret                             # return to caller
```

6. **(5 marks)** Consider this implementation of the recursive procedure `bar`, which differs from a typical procedure implementation by not using a prologue or epilogue.

```
foo:    # prologue omitted
        movl    $2, %eax
        call bar
        # epilogue omitted
bar:    decl    %eax
        jne     .L0
        call    bar
.L0:    ret
```

6a. Is this correct implementation okay? That is, does the recursion work and does `bar` return to `foo` after the proper number of recursive steps?

Yes it is correct.

6b. If this code is okay, what restrictions must the compiler place on procedures like `bar` for it to use this approach?

The compiler must insure that the recursive procedure has no local variables or arguments stored on the stack that change for difference recursive activations. The reason is that this implementation does not create a new stack frame for each recursive call and thus all recursive calls share the same set of local variables and arguments.

7. **(5 marks)** An n-type mosfet transistor can be implemented by introducing two wells of silicon doped with

arsnic (which has five electrons in its valance band compared to Silicon's four), a layer of glass and some wires called the source, drain and gate. What value of the gate closes the transistor (i.e., connects them together so that current flows between them)? Briefly explain how this works.

An n-type transistor conducts (i.e., it is a closed switch) if and only if the gate voltage is high (i.e., gate=1). In this case the high gate voltage induces a field on a small channel in the silicon between source and drain. The silicon is normally an insulator, but the field draws electrons toward the gate, which is separated from the silicon by a small layer of insulating silicon oxide. The electrons drawn to the channel gather there and, once there is enough of them, they change the channel from an insulator to conductor, thus closing the circuit between source and drain.

8. **(5 marks)** Give the HCL description of a circuit that takes two 32-bit integer inputs, A and B, and computes a function that evaluates to:

- 0 if either A or B are 0,
- A if A is odd and B is even,
- B if B is odd and A is even,
- $A + B$ if both are even, and
- $A - B$ if both are odd.

You can use the notation, $A[i]$ to refer to the i^{th} bit of A (where the lowest-order bit is bit 0).

```
int func = [
    A==0 || B==0:           0
    A[0]==1 && B[0]==0:     A
    A[0]==0 && B[0]==1:     B
    A[0]==0:                A+B
    1:                      A-B
]
```