Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Spring 2016

Test II

You have 50 minutes to finish this quiz.

Write your name and athena username on this cover sheet.

Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

This exam is open book and open laptop. Additionally, you may access the course website, but aside from that you may NOT USE THE NETWORK.

Please do not write in the boxes below.

I (xx/16)	II (xx/32)	III (xx/24)	IV (xx/28)	Total (xx/100)

Name:

Athena username:

I Assembly Code Generation

In this problem, you are going to use the following assembly syntax. The only available instructions are described in the table below.

`enter $n, $0`	Adjust stack for n bytes of local storage (You don't need to adjust the stack pointer or the frame pointer explicitly)
`mov a, b`	Move value of a into destination b
`add a, b`	Add value of a to value in b; store result in b
`mul a, b`	Multiplies values of a and b; store result in b
`call sym`	Call function `sym`
`leave`	Undo effects of `enter`
`ret`	Return from function call

Addressing modes: `%rax` references the value in register %rax, `(%rax)` references memory at the address in %rax, and `100(%rax)` references memory at 100 bytes + the address in %rax. Only one memory dereference (e.g. `(%rax)`) is allowed per instruction.

Assume Linux ABI calling convention: The first argument is passed in register %rdi. Temporary registers include %r10 and %r11. The integer return value should be stored in register %rax.

Stack variables are stored in memory locations `-8(%rbp)`, `-16(%rbp)`, etc.

Consider the following functions.

```
int foo () {
  int p[2];
  p[0] = 123; // Store 123 in address p
  p[1] = 45;  // Store 45 in address (p+8)
  return sqsum(p); // Call sqsum with p
}
```

```
int sqsum(int p[]) {
  // Parameter p is an address
  int x = p[0] * p[0];
  int y = p[1] * p[1];
  return x + y;
}
```

The assembly code for function `foo` is given below.

```
foo:
  enter $16, $0
  mov $123, -16(%rbp)
  mov $45, -8(%rbp)
  mov $-16, %rdi
  add %rbp, %rdi
  call sqsum
  leave
  ret
```

1. **[4 points]:** Use register %rdi to perform the following instructions.

A. **[2 points]:** In function `sqsum`, how to load the value of `p[0]` into %r10?

B. **[2 points]:** In function `sqsum`, how to load the value of `p[1]` into %r11?

2. [12 points]: Fill in the assembly code for function `sqsum` in the space below. Be sure to use the assembly language given for this problem.

```
sqsum:
  enter $0, $0

  ## Your code here...
```

```
  leave
  ret
```

II Available Expression Analysis

3. [4 points]: Recall that, in lectures, we defined four sets for each basic block b – GEN[b], KILL[b], IN[b], and OUT[b] – for available expression analysis. These sets represent the sets of expressions computed in block, expressions killed in block, expressions available at start of block, and expressions available at end of block, respectively.

For this question, write the general form of the data-flow equations for available expression analysis. You may use parentheses and set operations including $\cup$, $\cap$, and $-$.

A. [2 points]: Write the general equation for IN[b] in terms of OUT[p_1], ..., OUT[p_n] where $p_1, \ldots, p_n$ are predecessors of block b in a CFG.

B. [2 points]: Write the general equation for OUT[b] in terms of IN[b], KILL[b], and GEN[b].

Now, you will perform an available expression analysis on the following piece of code using a bit-vector formalization. Each statement is labeled with a number.

```
1:  x = a + b;
2:  y = a * b;

3:  if (x < y) {
4:    b = a + b;
5:    y = a * b;
    } else {
6:    y = a + b;
    }

7:  s = a * b;
8:  r = a + b;
```

4. [4 points]: Draw the control flow graph (CFG) for this program. Label each basic block with a letter.

5. [16 points]: Perform available expression analysis for the CFG. Use the bit-vector representation, where the order of the expresssions in the bit vectors is `a+b`, `a*b`, and `x<y`.

A. [8 points]: Compute GEN[b] and KILL[b] for each basic block b. Fill in your bit vectors in the table below, each row for a block b. (If the definitions for GEN and KILL permit multiple answers, write down any such answer that ensures correct analysis for IN and OUT.)

b	GEN[b]	KILL[b]

B. [8 points]: Compute the maximal solution of the data-flow equations, specifically, IN[b] = ... and OUT[b] = ... for each basic block b. Fill in your bit vectors in the table below, each row for a block b.

b	IN[b]	OUT[b]

6. [8 points]: Assume that all variables are live after this piece of code.

A. [4 points]: Can we avoid the re-computation for `a*b` in statement 7? If yes, specify the relevant changes. If no, explain why. (If both are possible, explain both sides.)

B. [4 points]: Can we avoid the re-computation for `a+b` in statement 8? If yes, specify the relevant changes. If no, explain why. (If both are possible, explain both sides.)

III Loop Optimization

Consider the following piece of code and its control flow graph (CFG). Each statement is labeled with a number. Each basic block is labeled with a letter.

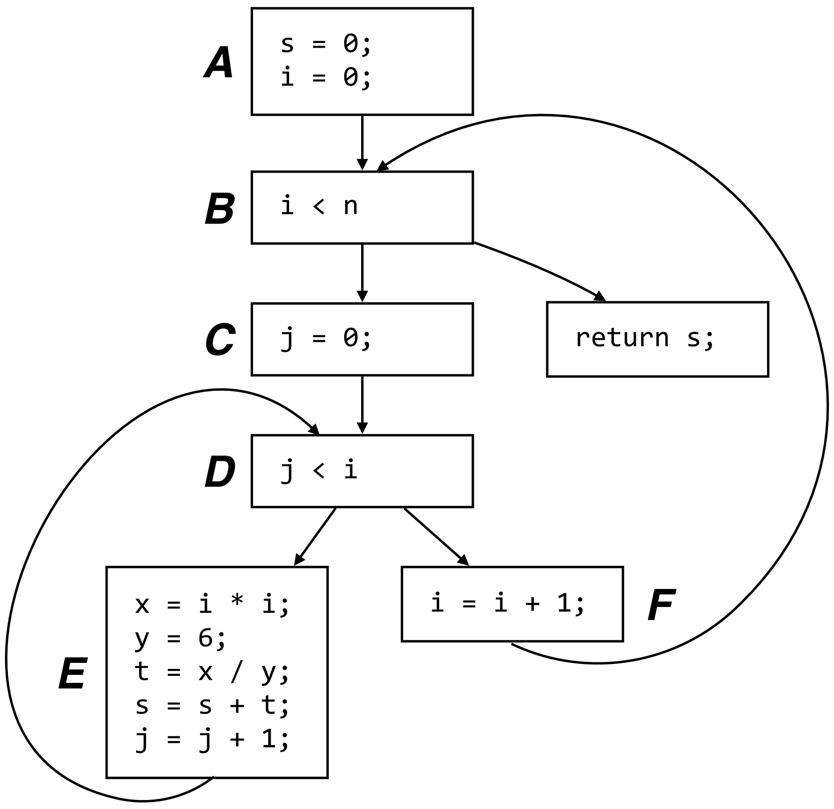
```
 1:  s = 0;
 2:  i = 0;

 3:  while (i < n) {
 4:    j = 0;

 5:    while (j < i) {
 6:      x = i * i;
 7:      y = 6;
 8:      t = x / y;
 9:      s = s + t;
10:      j = j + 1;
       }

11:    i = i + 1;
     }

12:  return s;
```



```
A: s = 0;
   i = 0;
B: i < n
C: j = 0;
   return s;
D: j < i
E: x = i * i;
   y = 6;
   t = x / y;
   s = s + t;
   j = j + 1;
F: i = i + 1;
```

7. [12 points]: Draw the dominator tree for the CFG. Each node should correspond to a basic block with the same label.

8. [12 points]: Assume that we iteratively find loop invariant code and perform invariant code motion until reaching a fixed point. Find as many loop invariant code as you can. Assume that we do not apply any other optimizations, such as dead code elimination or strength reduction.

For each loop invariant code, (a) state the label of the invariant statement, (b) state the place (inner loop, outer loop, or both loops) where it is invariant, and (c) state to which basic block it should move. An example loop invariant statement, `y = 6`, is given below. Fill in all other loop invariant code in the table below, each row for an invariant statement.

Statement	Invariant for which loop(s)	Move to which basic block
7	both	A

IV Register Allocation

In this problem, you will perform register allocation for the following code. Each instruction is labeled with a number.

```
1:  x = read_int(); // read an integer from user input
2:  y = read_int();

3:  if (x < y) {
4:    y = read_int();

5:    while (y >= x) {
6:      t = y / 2;
7:      y = t - 1;
      }

8:    print (x); // print result to the screen
    }
```

9. [4 points]: Can we store variables `x` and `t` in the same register?

10. [6 points]: Write the set of def-use chains for each variable in the program. Write each def-use chain as number pair (d, u) where d is the label of an instruction that defines the variable and u is the label of an instruction that uses that definition.

The def-use chains for variable `x` is given below. Write down the def-use chains for `y` and `t`.

`x`: (1, 3), (1, 5), (1, 8)

`y`:

`t`:

11. [6 points]: Write the set of webs for each variable in the program. Write each web as the set of instructions that belong to the web. Label each web with a name.

The web for variable `x`, w_x, is given below. Write down the sets of webs for `y` and `t`.

`x`: w_x (1, 2, 3, 4, 5, 6, 7, 8)

`y`:

`t`:

12. [8 points]: Draw the interference graph for the webs. Each node in the interference graph should represent one web. There should be an edge between two nodes if the two webs interfere. Label each node with the name of the corresponding web.

13. [4 points]: Suppose that the architecture we are targeting for compilation has two registers. Can we place all the variables in this code in registers? If yes, describe an assignment of variables to registers. If no, explain the reason using your interference graph as part of your justification.