Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Fall 2014

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/10)	II (xx/25)	III (xx/25)	IV (xx/40)	Total (xx/100)

Name:

Athena username:

I Assembly Code Generation

In this problem, you will generate relevant pieces of code for the following program:

```
void main()
{
  int x, y;
  // x = ... ;

  x = xysquare (x,y);  // Loc.1
  // ...
}
```

```
int xysquare(int x, int y)
{
  x = x + y;
  return x * x;
}
```

Write your assembly in AT&T syntax (src then dest). You should only use the instructions described in the table below. Assume Linux ABI calling convention. Remember that the first argument is passed in register `rdi` and the second in `rsi`.

Finally, remember the simple x86_64 addressing modes: `%rax` references register rax, `(%rax)` references memory at the address in rax, and `100(%rax)` references memory at 100 bytes + the address in rax. **Only one dereference** (e.g. `(%rax)`) **is allowed per instruction.**

x86_64 instructions to use:

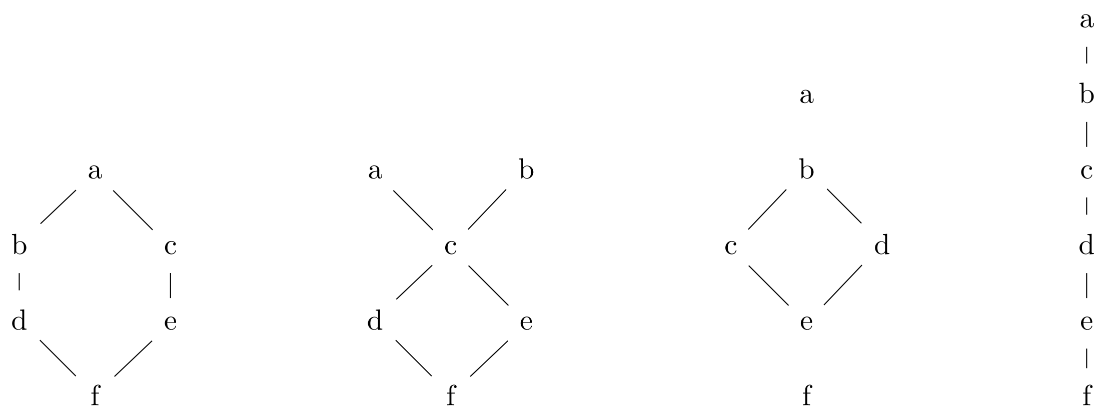
`enter $n, $0`	Adjust stack for n bytes of local storage
`mov a, b`	Move value of a into destination b
`add a, b`	Add value of a to value in b; store result in b
`mul a, b`	Multiplies values of a and b; store result in b
`call sym`	Call function `sym`
`leave`	Undo effects of `enter`
`ret`	Return from function call

1. [6 points]: Write the assembly code *only* for the function `xysquare`:

2. [4 points]: Assuming that in the function `main` the variable `x` is located at `%rbp - 8` and the variable `y` is located at `%rbp - 16`, write assembly code for the line in the main function that calls the function `xysquare` (Loc.1).

II Lattices

3. [4 points]: Let the set $P = \{a, b, c, d, e, f\}$. Circle those graphs below that are Hasse diagrams that graphically present a lattice $(P, \leq_i)$. For the graphs that you *do not* circle, give an explanation why.



For the remaining problems in this section, consider the set of natural numbers $\mathbb{N} = \{1, 2, \ldots\}$ and the partial order $b \leq a$, defined as:

$$b \leq a = \begin{cases} \text{True, if } b \text{ is a divisor of } a \\ \text{False, otherwise.} \end{cases}$$

4. [8 points]: Prove that the pair $(\mathbb{N}, \leq)$ is a lattice.

5. [3 points]: Is the lattice $(\mathbb{N}, \leq)$ complete? Explain why or why not.

6. [3 points]: What is the bottom element of the lattice $(\mathbb{N}, \leq)$?

7. [3 points]: What numbers does number 6 cover?

8. [4 points]: What are the greatest lower bound (GLB) and the least upper bound (LUB) for the set $\{2, 3, 5\}$?

III Reachability Analysis

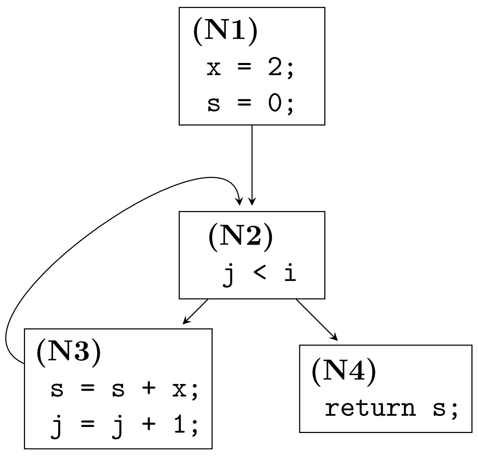
As you know, reaching definitions is the standard analysis for the constant propagation transformation. In this problem, we will consider the following modified reaching definition dataflow analysis and constant propagation transformation:

- The transfer functions are the same as for the standard reaching definition analysis.
- The set of the lattice elements is the same as in the standard bitvector reachability analysis.
- The join operator is intersection instead of union.

9. [5 points]: Given the same set of elements as in the standard bitvector reachability analysis, and the different joint operator, what should be the partial order for this analysis?

Example Program. We will first apply our modified reachability analysis on an example program. Figure below presents the CFG of the function `f(int i, int j)`. Each node is annotated with the index **N1** to **N4**.

10. [5 points]: Label all definitions in the CFG and compute the final IN and OUT sets for each node (use the space on the right of the CFG).



Transformation. We define the modified constant propagation transformation as follows. For each variable use, check the number of reaching definitions. If there is exactly one reaching definition for that use, and that definition sets the variable to a constant, then replace the variable use with a constant.

11. [3 points]: Can we apply the transformation to the example program? If so, what statements change. If not, explain why.

12. [3 points]: Does the transformed program produce the correct output? (i.e., does the transformation preserve the semantics of this program)

13. [5 points]: Is the analysis conservative for this transformation *for all programs*? In other words, does this combination of the analysis and the transformation preserve the semantics of the program? If so, explain why. If not, give a program for which the modified constant propagation transformation changes the semantics of the program.

14. [4 points]: If we change the join operator to union, is the analysis still conservative for the modified constant propagation transformation? If so, explain why. If not, give a program for which the transformation changes the semantics of the program.

IV Precise Sign Analysis

In this question we will build a more precise sign analysis. The purpose of this analysis is to enable the compiler to perform safety checks for calls to the log(x) function.

The analysis will analyze programs with one variable x. The language is defined as a sequence of the statements of this form:

$$\begin{aligned} S ::= {} & x = c \\ & \mid x = x + c \\ & \mid \texttt{if}\ (x == c)\{S_1\}\ \texttt{else}\ \{S_2\}; \end{aligned}$$

In addition, the very last statement in the program is a call to the `log(x)` function. In the previous definition, each c is a (signed) integer constant.

To keep track of the sign of variable x, we will use the lattice $(\mathcal{P}(\{-, 0, +\}), \subseteq)$. For example, if x has a non-negative value, then the analysis will represent this as a set $\{0, +\}$. If x has a positive value, the analysis will represent this as a set $\{+\}$.

15. [4 points]: Draw a Hasse diagram for the lattice.

Analysis. The safety check compiler pass looks at the sign analysis result at the function call to `log(x)` at the end of the program. For the `log(x)` function it checks if the analysis result indicates that x is positive. If so, the program passes the safety check, otherwise it fails.

We want our analysis to be conservative for this use. In other words, if the program passes the safety check, it must be the case that no execution of the program will ever pass a zero or negative value to `log(x)`. Your job is to implement a conservative sign analysis for this use.

16. [2 points]: Is your proposed analysis forward or backward?

17. [2 points]: What is the join operator for the analysis?

Note: Below, the input v of the transfer function is either the set IN or the set OUT of the node, depending on whether your analysis is forward or backward.

18. [5 points]: What is the transfer function for a statement `x = c`?

$f_{x=c}(v) =$

19. [5 points]: What is the transfer function for a statement `x = x + c`?

$f_{x=x+c}(v) =$

20. [8 points]: Are these two kinds of transfer functions distributive? Prove or provide a counterexample.

21. [4 points]: Does your analysis always terminate? If so, explain why. If not, give an example program where it does not terminate.

We can take into consideration the path information to make the analysis more precise when analyzing conditionals (since x is compared with a constant c). Conditional statements get compiled down into nodes with a true and false branch as follows:

$$\begin{tikzcd}
 & \boxed{\mathrm{x} == \mathrm{c}} \arrow[dl] \arrow[dr] & \\
\boxed{\texttt{S\_true}} & & \boxed{\texttt{S\_false}}
\end{tikzcd}$$

22. [5 points]: Give the transfer function $f_{x==c,true}(v)$ for the true branch and $f_{x==c,false}(v)$ for the false branch when analyzing the top node $x == c$:

$f_{x==c,true}(v) =$

$f_{x==c,false}(v) =$

23. [5 points]: Give a program that always passes a positive value to log(x) but fails the safety check.