6.110 Quiz 2 (Spring 2025)

Before starting the quiz, write your name on this page and read the following instructions:

- There are 7 problems on this quiz. It is 18 pages long; make sure you have the whole quiz. You will have 50 minutes in which to work on the problems. You will likely find some problems easier than others; read all problems before beginning to work, and use your time wisely.
- The quiz is worth 50 points total. The point breakdown each problem is given in the table below, and is also printed with the problem. Some of the problems have several parts, so make sure you do all of them!
- This is an open-book quiz. You may use a laptop to access anything on or directly linked to from the course website, **except for Godbolt.** You may also use any handwritten notes. You **may not** use Godbolt, any compilers, the broader internet, any search engines, large language models, or other resources.
- Do all written work on the quiz itself. If you are running low on space, write on the back of the quiz sheets and be sure to write (OVER) on the front side. It is to your advantage to show your work — we will award partial credit for incorrect solutions that are headed in the right direction. If you feel rushed, try to write a brief statement that captures key ideas relevant to the solution of the problem.

Problem	Title	Points
1	Dataflow Analysis	6
2	Register Allocation	13
3	Loop Analysis	6
4	Loop Optimization	6
5	Sign Information	10
6	Peephole Optimizations	8
7	Feedback	1
	Total	**50**

Name 6.110 Staff

MIT Email `6.110-staff@mit.edu`

1. **Dataflow Analysis** [6 pts]

   Reaching definitions analysis determines which definitions of variable x reach any particular use of variable x. More formally, for any variable x, we say a definition D of x "reaches" the use U if:

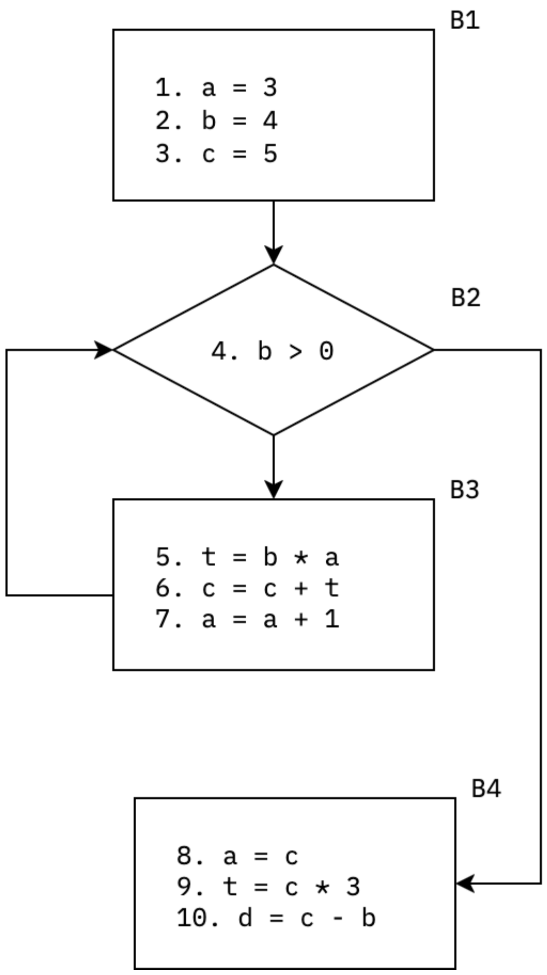
   - U reads the value x
   - There exists some path from D to U that does not redefine x.

   Reaching definitions analysis is a forwards dataflow analysis performed using the following dataflow equations for each basic block b:

$$\mathsf{IN}[b] = \bigcup_{p \in \mathsf{pred}(b)} \mathsf{OUT}[p] \tag{1}$$

$$\mathsf{OUT}[b] = (\mathsf{IN}[b] \setminus \mathsf{KILL}[b]) \cup \mathsf{GEN}[b] \tag{2}$$

   In this question, we will analyze the following program, where B1 is the entry block and B4 is the exit block.



```
B1
1. a = 3
2. b = 4
3. c = 5

B2
4. b > 0

B3
5. t = b * a
6. c = c + t
7. a = a + 1

B4
8. a = c
9. t = c * 3
10. d = c - b
```

(a) [2 pts] Fill in the following table with the GEN and KILL sets for each basic block. Write each set as a set of labels. Not all labels on the CFG have a corresponding definition. As a starting point, we have filled in the first row (B1) for you.

Block	GEN$[B]$	KILL$[B]$
B1	$\{1,2,3\}$	$\{6,7,8\}$
B2	$\{\}$	$\{\}$
B3	$\{5,6,7\}$	$\{1,3,8,9\}$
B4	$\{8,9,10\}$	$\{1,5,7\}$

(b) [3 pts] For the following question, possible IN and OUT sets for block B3 is provided. For each possibility, first determine whether or not those sets could be a *valid* fixed-point solution to the dataflow analysis for the entire control graph. If not, say which dataflow equation, either **(1)** or **(2)**, is violated.

IN$[B3]$	OUT$[B3]$	Valid/Invalid?	Violation (**(1)** or **(2)**)
$\{1,2,3,5,6,7\}$	$\{2,5,6,7\}$	Valid	N/A
$\{1,2,3,5,6,7,10\}$	$\{2,5,6,7,10\}$	Valid	N/A
$\{2,3,5,6,7\}$	$\{2,5,6,7\}$	Invalid	**(1)**
$\{1,2,3,5,6,7,9\}$	$\{2,5,6,7,9\}$	Invalid	**(2)**

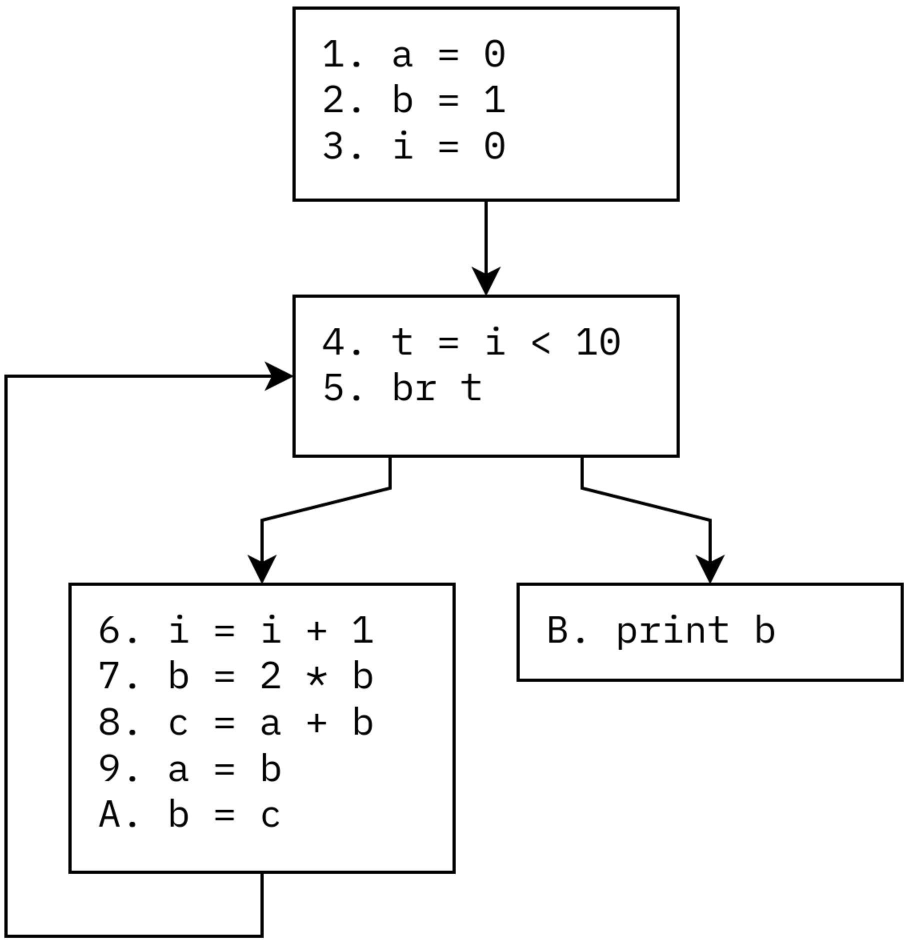
Note. The dataflow equations on page 2 of the exam are numbered (1) and (2).

(c) [1 pt] Would instruction 8 (`a = c`) be affected by copy propagation assuming no other optimizations? Explain why or why not.

Answer:
No, it would not be affected by copy propagation since the use of c in B4 may either refer to definition 3 or 6. Recall that a variable can only be copy propagated if (1) it has exactly one reaching definition and (2) that reaching definition is a copy instruction.

2. **Register Allocation** [13 pts]

   In this problem, you will perform register allocation for the following CFG:



```
1. a = 0
2. b = 1
3. i = 0
```

```
4. t = i < 10
5. br t
```

```
6. i = i + 1
7. b = 2 * b
8. c = a + b
9. a = b
A. b = c
```

```
B. print b
```

(a) [5 pts] Identify all webs in the CFG above. Write each def/use as "<label>: def/use <variable>". A web has been given to you as an example.

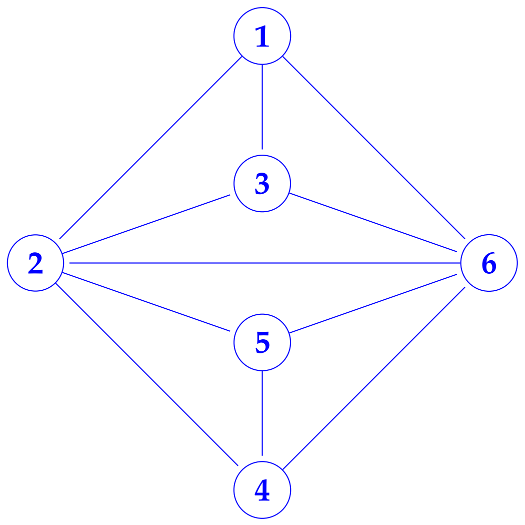
- Within each web, order the def/uses by ascending label number.
- For webs of the same variable, order by the first label in ascending order.

There may be more rows than needed.

Web #	Variable	def-uses
1	t	4: def t, 5: use t
2	a	1: def a, 8: use a, 9: def a
3	a	
4	b	7: def b, 8: use b, 9: use b
5	b	2: def b, 7: use b, A: def b, B: use b
6	c	8: def c, A: use c
7	c	
8	i	3: def i, 4: use i, 6: def i, 6: use i
9	i	

(b) [2 pts] Draw the interference graph of the webs above. Assume that within a single instruction, the def is sequenced after all uses. You may assume any order for uses in a single instruction. (**Hint**: it may be helpful to 1. draw the webs over the CFG and 2. start by considering webs that don't interfere)

Answer:



(c) [2 pts] From the interference graph, at least how many registers do we need?

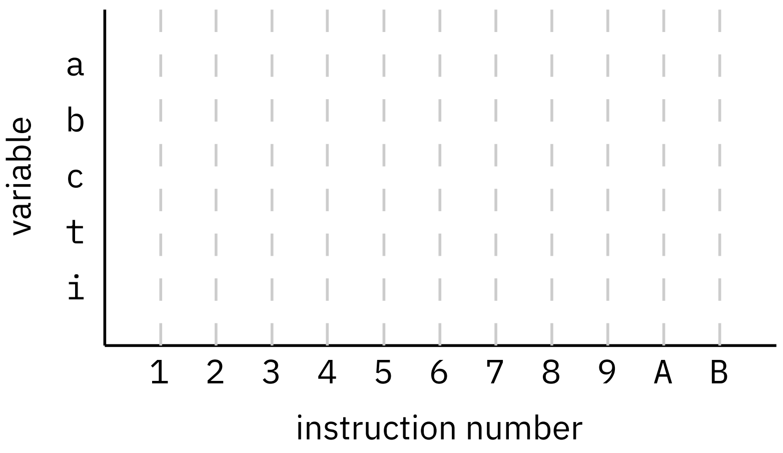
Answer:

4

(d) [3 pts] Now let's explore another way to do register allocation.

i. Assume we allocate registers to variables instead of webs;
ii. Assume instructions are ordered as follows: 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B;
iii. Assume a variable will be stored in the same register from its first use/def to the last use/def according to the order above.

For each variable, draw a line segment in the figure below from its first def/use to its last def/use.



(e) [1 pt] What is the minimum number of registers needed by this register allocator?

Answer:

4

3. **Loop Analysis** [6 pts]

   Consider the following CFG. **A** is the entry block. Left arrows represent false branches, and right arrows represent true branches.

$$\begin{tikzcd}
 & & \mathbf{A} \arrow[dl] \arrow[dr, bend left] & \\
 & \mathbf{C} \arrow[dl] \arrow[dr, bend left] & & \mathbf{B} \arrow[ul, bend left] \\
\mathbf{E} \arrow[uurr, bend left] & & \mathbf{D} \arrow[ul, bend left] \arrow[loop left] &
\end{tikzcd}$$

   (a) [1 pt] Draw the dominator tree of this CFG.

Answer:

$$\begin{tikzcd}
 & & \mathbf{A} \arrow[dl] \arrow[dr] & \\
 & \mathbf{C} \arrow[dl] \arrow[dr] & & \mathbf{B} \\
\mathbf{E} & & \mathbf{D} &
\end{tikzcd}$$

(b) [3 pts] Identify the loops of this CFG. (**Hint:** there are at least 3)

Loop Header	Loop Nodes	Loop Back-Edge(s)
A	**A, B**	**(B, A)**
C	**C, D**	**(D, C)**
D	**D**	**(D, D)**
A	**A, C, D, E**	**(E, A)**

(c) [2 pts] Is it possible to write a Decaf function that could have this CFG? (**Note:** you must provide reasoning about your answer to receive any credit)

Answer:

No, as **E** is outside of the **A-B** loop, and we do not have goto in Decaf to jump all the way back.

4. **Loop Optimizations** [6 pts]

   Consider the following Decaf snippet.

```
int foo(int p) {
    int w, x, y, z;
    x = 0;
    while (x < 10) {
        z = 4 * x + 6;
        y = p * p + 100;
        w += foo(z) + y;
        x += 2;
    }
    return w;
}
```

   (a) [2 pts] Identify any Base Induction Variables and Derived Induction Variables.

Variable	Base/Derived?	Induction Variable Triple
x	Base	(x, 1, 0)
z	Derived	(x, 4, 6)

(b) [4 pts] Rewrite the program after performing all the loop optimizations covered in lecture: **loop-invariant code motion**, **induction variable strength reduction**, and **induction variable elimination**. Eliminate as many induction variables as you can.

Answer:

```
int foo(int p) {
    int w, z, y;
    z = 6;
    y = p * p + 100;
    while (z < 46) {
        w += foo(z) + y;
        z += 8;
    }
    return w;
}
```

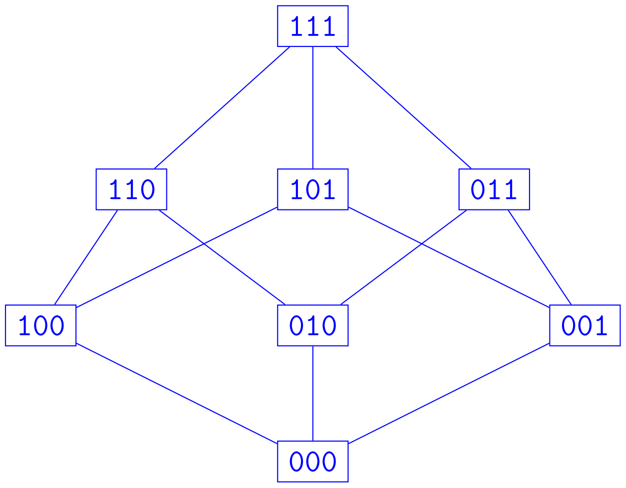
5. **Sign Information** [10 pts]

   Suppose we want to keep track of sign information for integer variables. Each variable can be either negative (**Neg**), zero (**Zero**), or positive (**Pos**). We represent the set of possible signs of a variable as a **3-bit vector**, where the most significant bit (MSB) indicates whether the variable could be negative, the middle bit indicates if the variable could be zero, and the least significant bit (LSB) indicates if the variable could be positive. For example:

   - **Pos** only $\rightarrow$ `001`
   - **Zero** only $\rightarrow$ `010`
   - **Neg** or **Zero** $\rightarrow$ `110`

   (a) [2 pts] Draw the Hasse diagram for this lattice and define the join operator as an arithmetic or boolean operation. **Hint**: it should be a very simple operator.

   Answer:



   The join operator is boolean or:

$$A \vee B = A|B$$

(b) [4 pts] We define the lattice L for program variables x, y, and z, Each element $m \in L$ is a mapping:

$$m : \{x, y, z\} \rightarrow \text{SignSets}$$

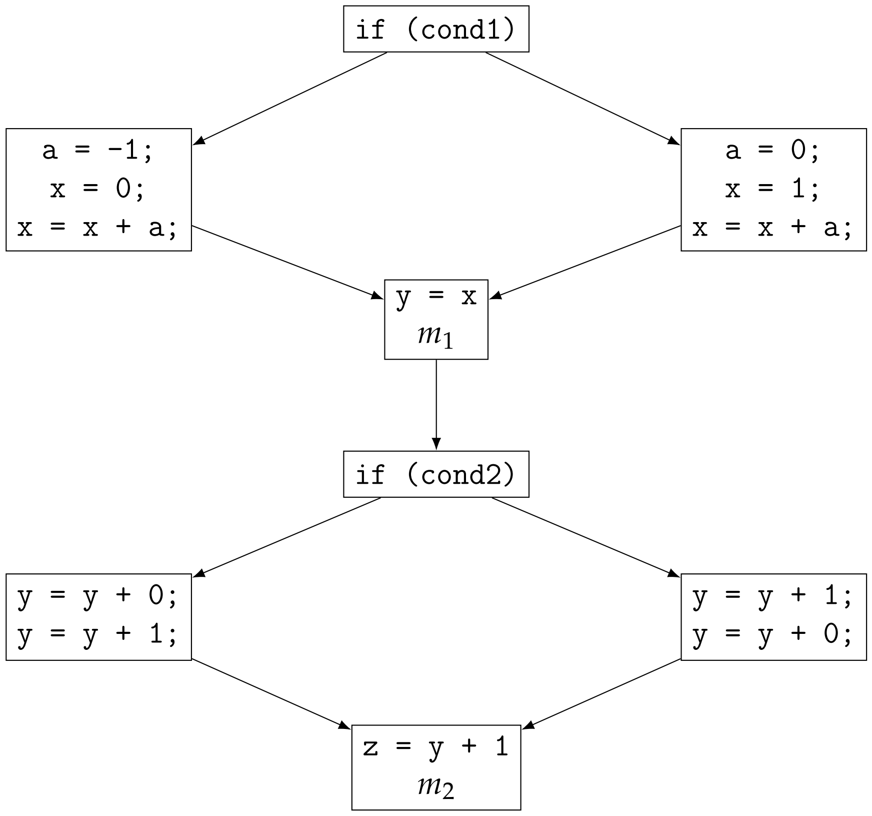
where SignSets are subsets of {**Neg**, **Zero**, **Pos**}. Complete the following transfer function information:

Operation	$m(x)$
`x = -1`	100
`x = 0`	010
`x = 1`	001
`x = y`	$m(y)$

Additionally, complete the partial table for addition: if x and y have known signs, what is $z = x + y$?

x \ y	100	010	001
100	100	100	111
010	100	010	001
001	111	001	001
110	100	110	111
111	111	111	111

Consider the following CFG where `if (cond1)` marks the entry block:



(c) [1 pt] m_1 is the mapping that the dataflow analysis computes for the program point after`y = x`. What is $m_1(y)$?

Answer:

$m(y) = 101$

(d) [1 pt] m_2 is the mapping that the dataflow analysis computes for the program point after `z = y + 1`. What is $m_2(z)$?

Answer:

$m(z) = 111$

(e) [2 pts] How could one use sign set information to optimize code?

Answer:

Can be used to optimize conditionals for > 0, < 0 jumps (many valid answers)

6. **Peephole Optimizations** [8 pts]

   Ben Bitdiddle has written a poorly optimized x86 code generator. Help him identify opportunities to make his assembly more performant. Make sure your optimized assembly is **semantically identical** to the left-hand side, and that it runs in **fewer cycles**, given the latency table below. Write your assembly code in AT&T syntax (source operand on the left), and make sure it is syntactically correct (i.e, an assembler would accept it). Assume a single-core, single-thread CPU executes one instruction at a time and in order.

   Ben Bitdiddle downloads a latency table from a popular x86 reference website for Intel Skylake-X CPUs, which is the processor family on the 6.110 Derby Server:

	Operands	**Latency**		**Operands**	**Latency**		**Operands**	**Latency**
`MOVQ`	r/i,r	1	`ADDQ SUBQ`	r/i,r	1	`SHLQ`	i,r	1
`MOVQ`	r/i,m	2	`ADDQ SUBQ`	m,r	5	`SHLQ`	i,m	2
`CMOVcc`	r,r	1	`CMPQ`	r/i,r	1	`SHRQ`	i,r	1
`XCHGQ`	r,r	2	`CMPQ`	r/i,m	1	`SHRQ`	i,m	2
`PUSHQ`	r	3	`IMULQ`	r/m	3	`SARQ`	i,r	1
`POPQ`	r	2	`IMULQ`	r,r	3	`SARQ`	i,m	2
			`IDIVQ`	r	95			
			`INCQ DECQ`	r	1			
			`INCQ DECQ`	m	3			

   Key: Operands: i (immediate), r (register), m (memory location).

Assembly	Optimized Assembly

`movq  %r10, %rcx`
`addq  %r8, %r10`
`movq  %r10, %rcx`
`addq  $1, %r10`
`movq  %r10, %rcx`
	`addq %r8, %r10`
	`addq $1, %r10`
	`movq %r10, %rcx`

`movq  $8, %r11`
`idivq %r11`
`movq  %rax, %r10`
`movq  $0, %rdx`
	`movq %r11, %rax`
	`shrq $3, %rax`
	`movq $0, %rdx`

`movq  $32, %r11`
`imulq %r11, %r10`
	`shlq $5, %r10`

`shlq  $2, %rax`
`addq  %rcx, %rax`
`movq  (%rax), %rcx`
	`movq (%rcx, %rax, 4), %rcx`

7. **Feedback** [1 pt]

   (a) [1 pt] What is one thing you studied that did not show up on the exam?

   Answer:
   Anything valid.

   (b) [0 pts] **[Optional]**: Any other feedback for the course staff? (or feelzbox)

   Answer:
   Anything valid.