6.110 Quiz 2 (Spring 2024)

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

- There are 6 problems on this quiz. It is 17 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	Program Analysis	8
2	Register Allocation	10
3	Loop Analysis	4
4	Loop Optimizations	8
5	Parallelization	8
6	Bit-width Analysis	12
	Total	**50**

Name 6.110 Staff

Kerberos `6.110-staff`

1. **Program Analysis** [8 pts] (parts a–c)

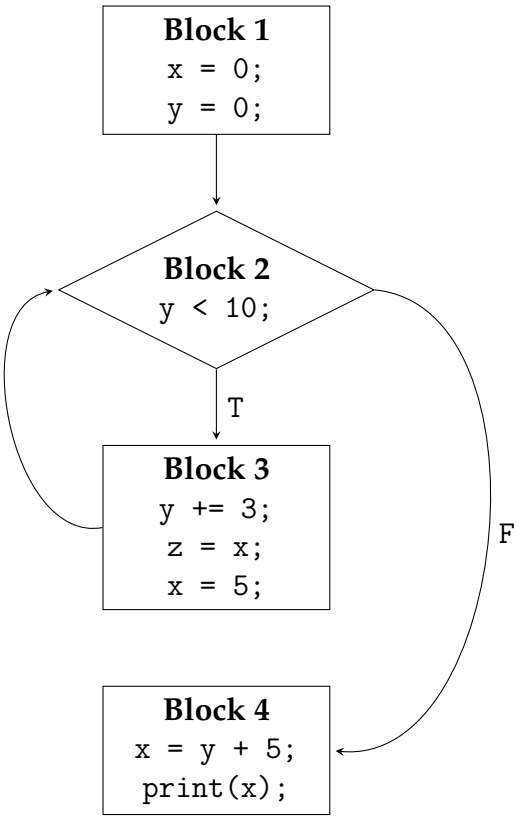
   In this problem we will perform liveness analysis and dead code elimination. Remember that a variable V is said to be live at point P if:

   - there is a use U of V along some path starting at P.
   - there is no redefinition of V along that path until U.

   As taught in lecture, liveness analysis a backwards dataflow analysis performed using the following transfer function for each basic block b:

   $$\text{IN}[b] = \text{USE}[b] \cup (\text{OUT}[b] - \text{DEF}[b]).$$

   We will analyze the program represented by the following CFG, where block 1 is the entry block and block 4 is the exit block.



(a) [4 pts] Fill in the following table with the USE and DEF sets for each basic block, as well as the *final* values of the IN and OUT sets for each basic block. Write each set as a set of variables.

As a starting point in this backwards analysis, we have filled in the row of the exit block (Block 4) for you.

Basic block b	USE[b]	DEF[b]	IN[b]	OUT[b]
Block 1	$\varnothing$	{x, y}	$\varnothing$	{x, y}
Block 2	{y}	$\varnothing$	{x, y}	{x, y}
Block 3	{x, y}	{x, y, z}	{x, y}	{x, y}
Block 4	{y}	{x}	{y}	$\varnothing$

Note that the statement `y += 3` in Block 3 counts as a USE of `y`.

(b) [2 pts] Based only on the results of your analysis in part (a), which statement(s) constitute dead code and can be removed?

Answer:

Variable `z` is never alive, so the statement `z = x` is dead code and can be removed.

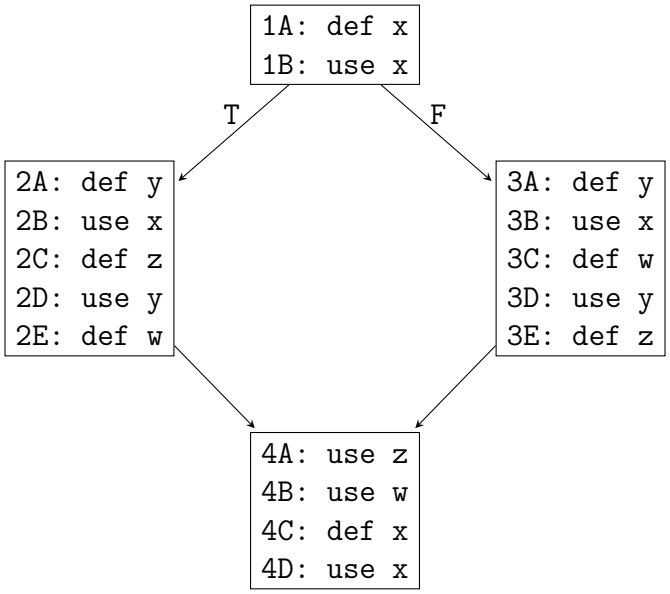
(c) [2 pts] Assume that you have removed the statement(s) identified as dead in part (b), and you perform liveness analysis again. Which additional statement(s) will be identified as dead, if any?

Answer:

After removing the statement `z = x`, `x` is not used in block 3 anymore, which means that a liveness analysis will find `x` to be dead in blocks 1-3, so the statements `x = 0` and `x = 5` will be identified as dead.

2. **Register Allocation** [10 pts] (parts a–d)

In this problem, you will perform register allocation for the following CFG with def and use statements:

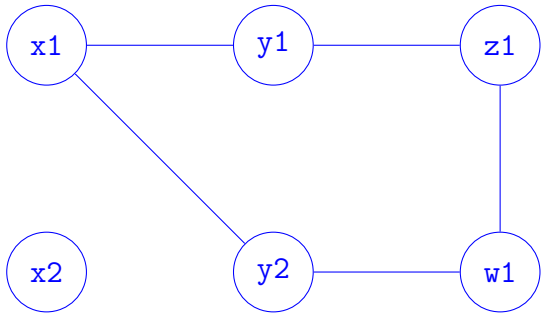


(a) [3 pts] Identify the webs in the program. Write each web as the set of `def` and `use` instructions that belong to the web, using the statement numbers given in the CFG. We have given you web names `x1`, `x2` corresponding to the variable `x` (and similarly for the other variables). Use only as many webs as you need.

Web	Instructions in web
x1	1A, 1B, 2B, 3B
x2	4C, 4D
y1	2A, 2D
y2	3A, 3D
z1	2C, 3E, 4A
z2	
w1	2E, 3C, 4B
w2	

(b) [3 pts] Draw an interference graph for the webs you identified. Each node in the intereference 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.

Answer:



(c) [2 pts] Describe an assignment of webs to registers using three general purpose registers `%r1`, `%r2`, `%r3`.

Answer:
There are many ways to 3-color the 5-cycle, and the isolated node can have any color. One possible way is: `x1, w1` gets assigned `%r1`, `y1, z1` gets assigned `%r2`, and `y2, x2` gets assigned `%r3`.

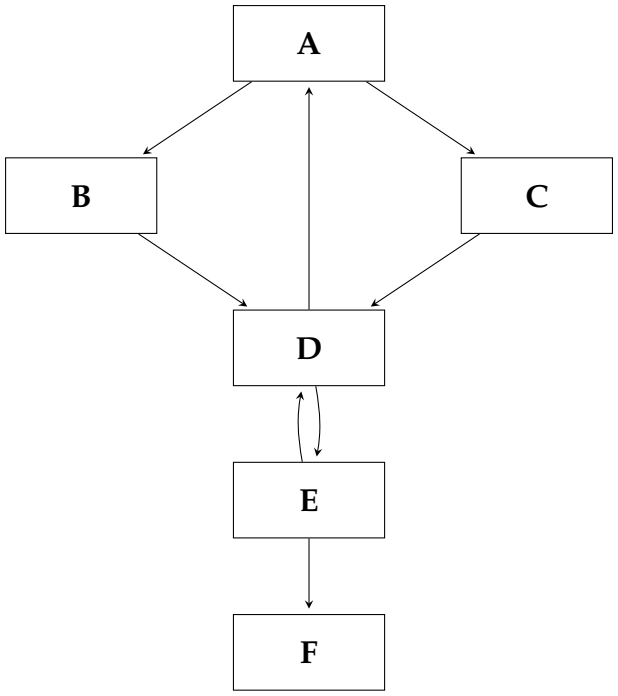
(d) [2 pts] Now suppose that we are only given two general purpose registers `%r1` and `%r2`. In order to reduce the number of registers needed, we will need to split the web of a variable into two webs by spilling (storing) the variable to memory over some portion of the program.

Suppose that you know that the true branch (from block 1 to block 2) is taken in 80% of program executions. Which variable would you spill, where will you store and load this variable from memory, and why?

Answer:
Spill `w` around statement `3D` (splitting web `w1`), or spill `y` around statement `3B` (splitting web `y2`). Pick these spill locations because block 3 is least likely to be executed.

3. **Loop Analysis** [4 pts] (parts a–b)

   In the following control flow graph, **A** is the entry node and **F** is the exit node.



(a) [2 pts] Draw the dominator tree for this control flow graph.

Answer:

Edges: AB, AC, AD, DE, EF.

(b) [2 pts] For each loop in this control flow graph, give the (i) loop header, (ii) back edge, and (iii) set of nodes within the loop by filling in the table below. Use only as many rows of the table as you need.

Loop header	Back edge	Nodes in loop
A	D $\rightarrow$ A	A, B, C, D, E
D	E $\rightarrow$ D	D, E

4. **Loop Optimizations** [8 pts] (parts a–c)

Consider the following program.

```
1  int a, b, c, d, e;
2  c = 4;
3  while (c < 20) {
4      a = 4;
5      c = c + 2;
6      e = 5 * a;
7      b = 3 * c + 1;
8      d = 2 * b - 1;
9      printf(\d: %d, e: %d\n", d, e);
10 }
```

(a) [3 pts] For each variable within the program, state in the table below whether the variable is an induction variable or not based on the criteria described in lecture. For each induction variable, also indicate whether it is a base induction variable or a derived induction variable, and give its triple (in the form $\langle k, c, d \rangle$).

Variable	Is induction variable? (T/F)	Base or derived?	Induction variable triple
a	F		
b	T	derived	$\langle$c, 3, 1$\rangle$
c	T	base	$\langle$c, 1, 0$\rangle$
d	T	derived	$\langle$c, 6, 1$\rangle$
e	F		

(b) [1 pt] Which statements in the loop are loop-invariant? List the **line numbers** of all statements within the loop that are loop-invariant.

Answer:

Lines 4 and 6.

(c) [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.

Partial credit will be given if you performed only some of the optimizations listed.

Answer:

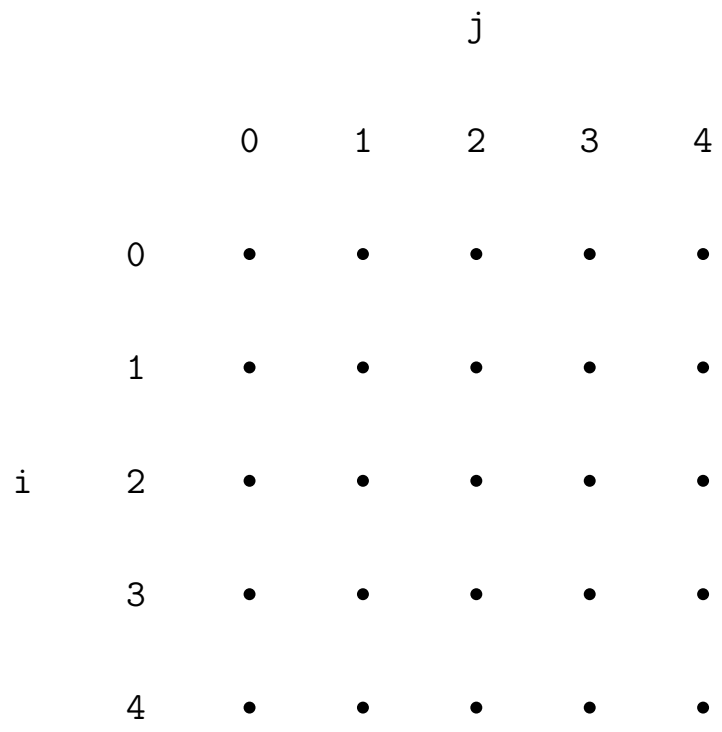
```
1       int a, d, e;
2       a = 4;
3       e = 5 * a;
4       d = 25;
5       while (d < 121) {
6           d = d + 12;
7           printf(\d: %d, e: %d\n", d, e);
8       }
```

5. **Parallelization** [8 pts] (parts a–c)

Consider the following loops, where `A[i,j]` refers to the element in the i^{th} row and j^{th} column in a two-dimensional array.

```
for (i = 1; i < n; i += 1) {
    for (j = 0; j <= i; j += 1) {
        A[i, j] = A[i - 1, j - 1] + A[i - 1, j];
    }
}
```

(a) [4 pts] Assume $n = 5$. In the grid below, circle the iteration space for the loops and draw the dependence vectors. You may ignore out-of-range cases.



(b) [2 pts] List all distance vectors for these loops.

Answer:

$$\begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$$

(c) [2 pts] For each loop, state in the table below whether the loop can be parallelized into a for-all loop or not.

Loop	Can be parallelized? (T/F)
Outer loop (`i` loop)	F
Inner loop (`j` loop)	T

6. **Bit-width Analysis** [12 pts] (parts a–g)

We would like to design an analysis to determine the "bit-width" of each **unsigned integer variable**. The bit-width of a nonnegative integer n the minimal number of bits required to store n, i.e. the smallest positive integer b such that $2^b > n$.

We define a base lattice $(W, \leq)$ to keep track of bit-width for a single unsigned integer variable as follows:

$$W = \{0, 1, 2, \cdots\},$$

and $\leq$: $\mathcal{P}(W \times W)$ is defined by

$$0 \leq 1 \leq 2 \leq \cdots.$$

The element 0 represents an undefined variable (no information). Other numbers represent the fact that the variable may contain a value up to that many bits.

(a) [1 pt] Define the join operator, $\vee : W \times W \to W$, such that it is consistent with $\leq$. You may use the function max defined for integers.

Answer:

Given $a, b \in W$, then $a \vee b = \max\{a, b\}$.

Suppose our programs consist of three variables x, y, and z. We can define the actual lattice $(L, \leq_L)$ as a function lattice, where each lattice element $m \in L$ is a mapping from program variables to elements of the base lattice W. (In other words, elements of L look like $[x \mapsto w_1, y \mapsto w_2, z \mapsto w_3]$, where $w_1, w_2, w_3 \in W$.)

(b) [1 pt] Define the order relation $\leq_L$ between elements of L in terms of the order relation $\leq$ between elements of W.

Answer:

Given $m_1 \in L$ and $m_2 \in L$,
$m_1 \leq_L m_2$ if and only if $m_1(x) \leq m_2(x)$, $m_1(y) \leq m_2(y)$, and $m_1(z) \leq m_2(z)$.

(c) [4 pts] For each basic block B given below, write the most precise and sound transfer function $f_B : L \to L$. We have provided examples in the first two rows.

Statements in B	Transfer function f_B
`x = 2 * y;`	$f_B(m) = m[x \mapsto m(y) + 1]$
`x = 6; x += 1;`	$f_B(m) = m[x \mapsto 3]$
`x = y + 1;`	$f_B(m) = m[x \mapsto m(y) + 1]$
`x = y * z;`	$f_B(m) = m[x \mapsto m(y) + m(z)]$
`x = y + z;`	$f_B(m) = m[x \mapsto \max\{m(y), m(z)\} + 1]$
`printf("%d", x);`	$f_B(m) = m$

Consider the following program.

```
if (...) {
    x = 3;
    y = 6;
} else {
    y = 3;
    x = 6;
}
z = x * y;
```

(d) [1 pt] Assuming you analyze this program using the forward data-flow analysis algorithm, what is the final lattice element that this analysis computes at the program point after the final statement `z = x * y;`?

Answer:

$$\{x \mapsto 3, y \mapsto 3, z \mapsto 6\}$$

(e) [1 pt] In the meet-over-paths solution for the above program, what is the lattice element at the program point after the final statement `z = x * y`?

Answer:

$$\{x \mapsto 3, y \mapsto 3, z \mapsto 5\}$$

(f) [2 pts] The base lattice is not complete. What issue could this possibly cause if one were to use this in a real data-flow analysis? Provide a short, example program to demonstrate this issue.

Answer:
Non-termination. Example program:

```
x = 0;
while (true) {
    x++;
}
```

(g) [2 pts] Bit-width analysis can be viewed as a weaker version of other analyses such as maximum value analysis or integer range analysis. List one advantage and one disadvantage of using bit-width analysis over other more powerful analyses.

Answer:
Pros: Faster compile time, less memory required during compilation.
Cons: Imprecise.

Fun fact: You might wonder why we would ever use bit-width analysis. Bit-width analysis allows us to select the best instructions and register types if we know we are dealing with values that are guaranteed to be small. For example, in Decaf, you could use `div` *instead of* `divq` *if you know your operands are 32-bit integers rather than 64-bit!*