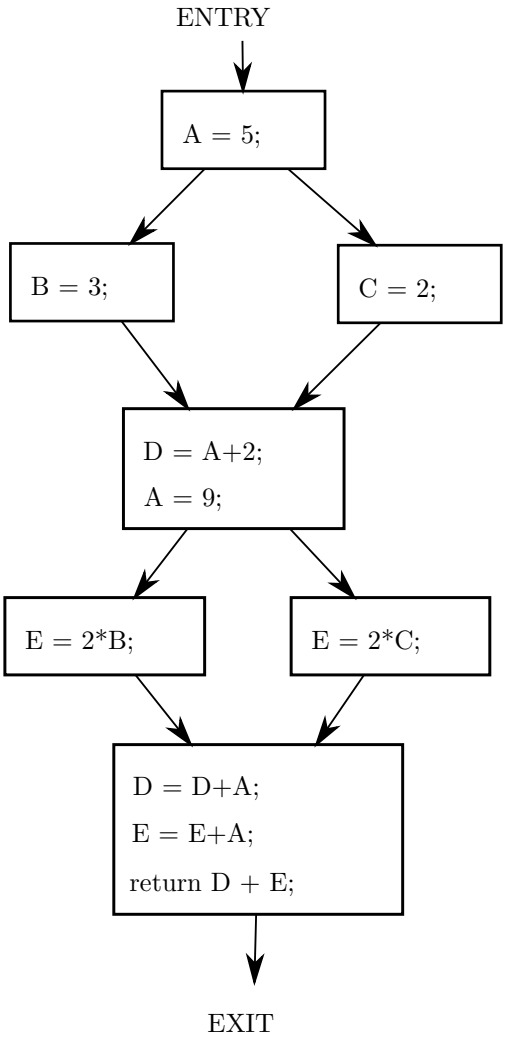
CS 243 review session, winter 2021 week 4

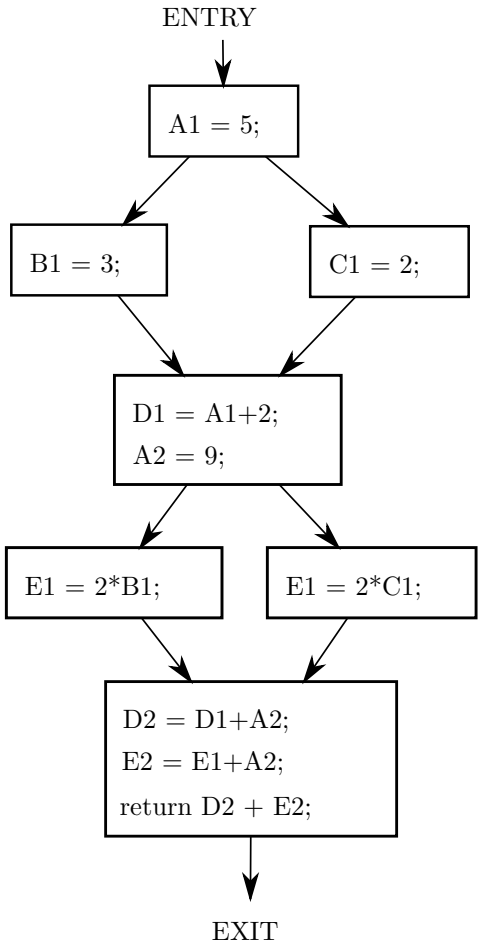
1. Register Allocation

For the following control flow graph, perform register allocation. Show the results of the following steps.

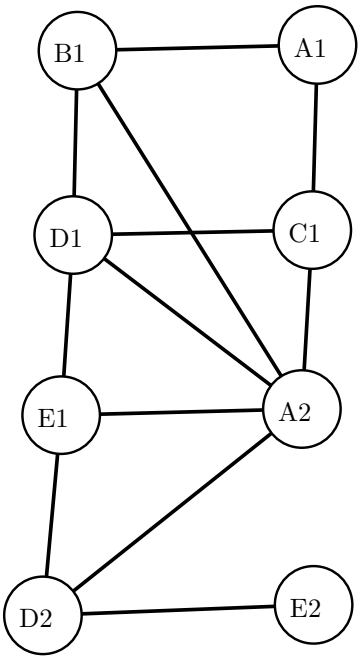
1. Assign each definition and use of a variable to a live range. For example, all instances of A must be replaced with either A_1 or A_2 to signify one of two live ranges.
2. Draw the register interference graph with lines between nodes that represent live ranges.
3. Apply the heuristic-based register allocation algorithm with for a machine with 3 registers. Show the resulting "stack" of registers and show which ones, if any, are marked as spilled.
4. Assign the live ranges to registers.



1. Assign each definition and use of a variable to a live range. For example, all instances of A must be replaced with either A_1 or A_2 to signify one of two live ranges.



2. Draw the register interference graph with lines between nodes that represent live ranges.



3. Apply the heuristic-based register allocation algorithm with for a machine with 3 registers. Show the resulting "stack" of registers and show which ones, if any, are marked as spilled.

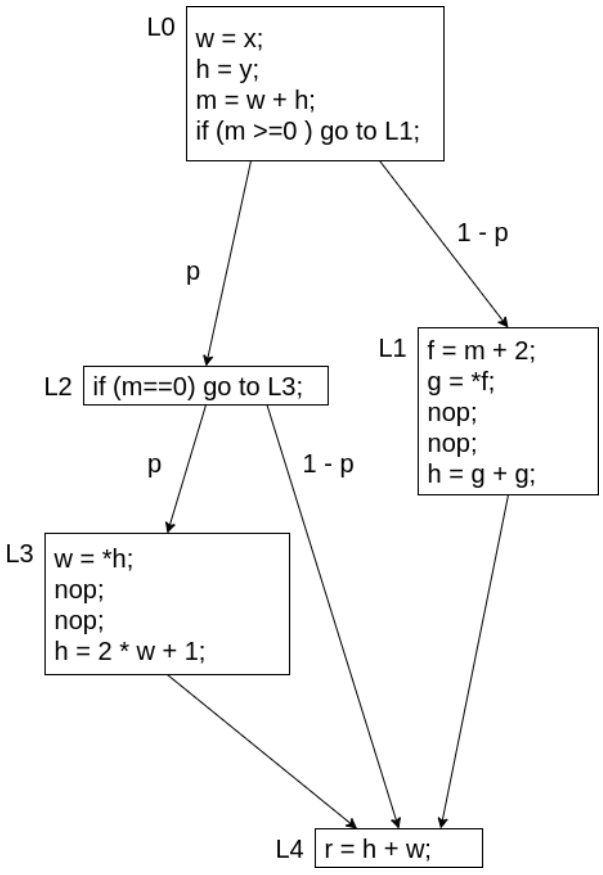
   There are multiple answers. One possible answer is as follows:
   E2 D2 E1 A1 B1 A2 C1 D1

4. Assign the live ranges to registers. There are many possible answers. One possible answer is as follows:

Live Range	Register
A1	R0
A2	R2
B1	R1
C1	R1
D1	R0
D2	R0
E1	R1
E2	R1

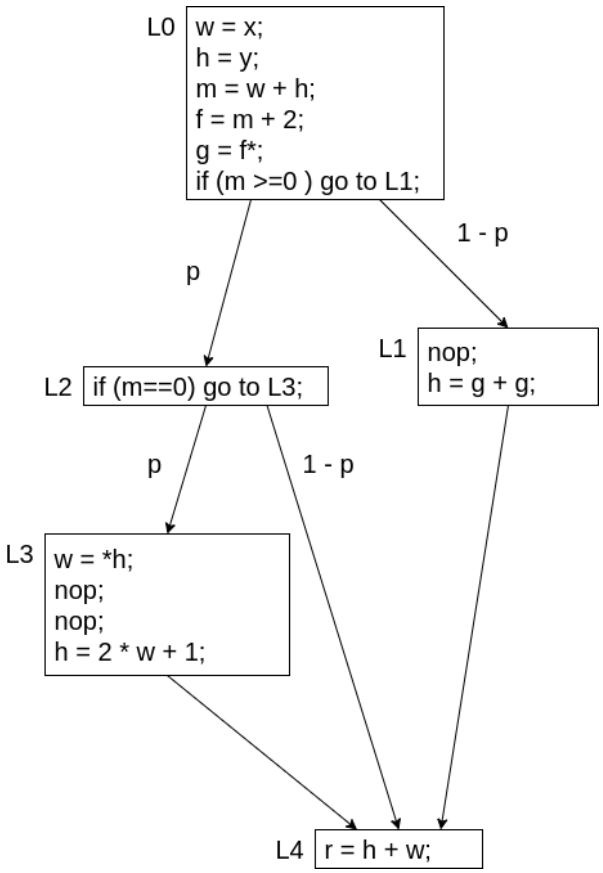
2. Global instruction scheduling

Assume you have a statically scheduled machine that can only issue one operation every clock. All operations have a latency of one clock cycle, with the exception of its memory load operation, which has a latency of three clock cycles. Consider the following locally scheduled program:



Assume that only r is live at the end of the program. Each branch in the flow graph is labeled with the probability that it is taken dynamically. To answer the following, you may apply any of the code motions discussed in class, but no other optimizations.

1. Is this the best globally scheduled code that can be generated given that p = 0.1? If not, provide the improved code along with its expected execution time.



Path1: L0,L1,L4: 9 instructions
Path2: L0,L2,L4: 8 instructions
Path3: L0:L2,L3,L4: 12 instructions.
The expected execution time is 9 * 0.9 + 8 * 0.09 + 12 * 0.01 = 8.94 clock cycles.

The original program takes 10 * 0.9 + 6 * 0.09 + 10 * 0.01 = 9.64 clock cycles. Therefore, the reduction is runtime is roughly 7.2%.

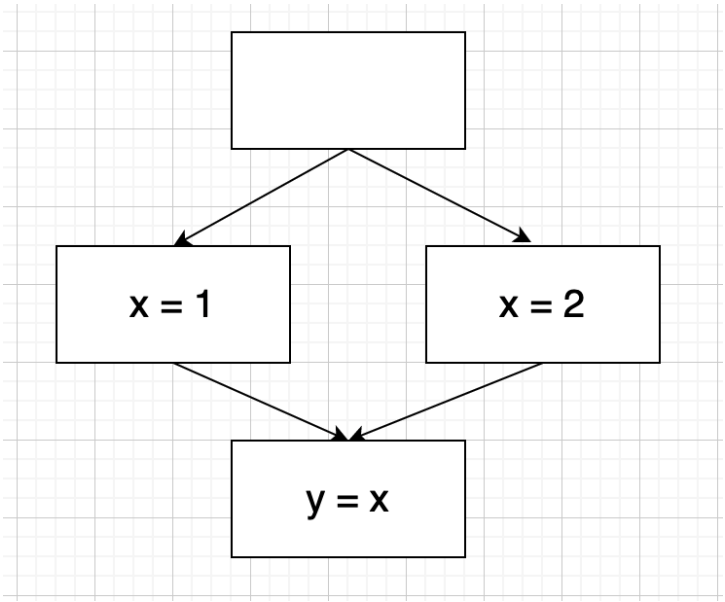
3. Enforcing data dependency in code motion

When performing global instruction scheduling on a program using code motion, we want to make sure all code motions honor existing data dependencies between instructions in the program, among other constraints.

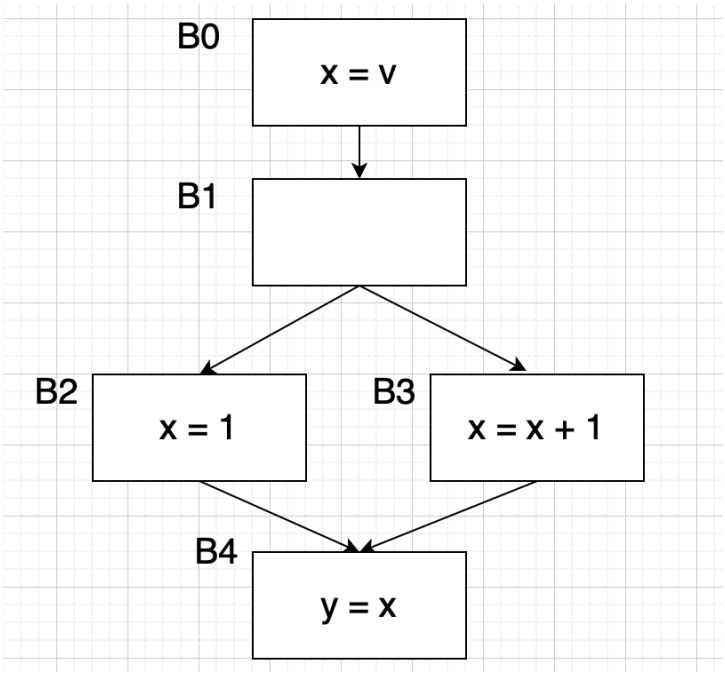
Let's look at an example. We apply the global scheduling algorithm in slide 18 of lecture 7 to the region shown in the diagram below. Instructions inside each basic block that do not involve x are omitted. We are going to observe a particular point in the execution of the algorithm, where it is analyzing the topmost basic block along with all other blocks it dominates (i.e. all four blocks in the diagram). The algorithm is now running the two innermost loops: for each clock cycle, we schedule instructions off of the candidate instruction list `CandInsts`, until all instructions from the top block are scheduled.

Let's say we already scheduled $x = 2$ to the top block. This is an upward code motion that breaks control equivalence ($x = 2$ is not anticipated in the left branch) and leads to speculative execution. However, we are fine with this type of code motion as long as all the data dependency constraints are satisfied, and there are no unwanted side effects due to speculation. Later on, let's assume $x = 1$ becomes the top priority instruction. Now, our algorithm would also try to schedule $x = 1$ in the top block, after $x = 2$. However, as shown in class, this violates data dependency between $x = 2$ and $y = x$ along the right branch. Obviously, our algorithm should honor this fact and reject this scheduling proposal for $x = 1$.

You task is as follows: You will help design this step in the algorithm where you check whether a given upward code motion of a definition d of the form $x = a$ or $x = a + b$ will preserve all the data dependencies. You may assume the rest of the algorithm works correctly, so that by the time d is scheduled, it is ready: all instructions it depends on have been scheduled. You should incorporate a data flow analysis to perform this step.



Here, your algorithm should only allow either x = 1 or x = 2 to be moved to the top block, but not both.



Here, your algorithm should allow $x = x + 1$ to be moved to B1, but not $x = 1$.

Solution: liveness analysis. If a variable is live at a program point, then we cannot move speculative definitions to the variable above that program point. (Chapter 10.4.4)

If a variable a is live at program point p, that means there exists a program path from p that uses a before it gets overwritten. That is, if we assume a has been initialized at p, then its liveness at p signifies the existence of a Write-Read dependence on a path that crosses p, between a write operation i_W and read operation i_R. During upward code motion, if we move a definition of a above p, then it will interfere with this existing WR dependence, by making itself the new source to the Read operation i_R.

Notice how it gets more tricky in the second example: According to our liveness criteria, since x is live at $B1$.out, we can't move either $x = 1$ or $x = x + 1$ past $B1$.out. However, it is actually legal for us to move $x = x + 1$ from B3 to B1. The reason our criteria is being too conservative here is as follows. At $B1$.out, x is live and the only use of it is $x = x + 1$. But since we're exactly moving $x = x + 1$, this WR dependence between $x = v$ and $x = x + 1$ will actually be preserved.

Now that we saw our liveness criteria is blocking certain legal code motions. Can we do anything to remedy this? Well, we can tighten the criteria in many ways. Next up we will show you one direction you can go. We create a new analysis based off of liveness: Instead of recording the live variables at each program point, we can record for each variable x the instructions that anticipate it, i.e. that might use x before x is reassigned. Let's call it "Anticipating Instructions" analysis (ANT_INS).

Direction of your analysis (forward/backward)	Backwards
Lattice elements and meaning	Set of all instructions that might use the variable for its value at the current program point, before the variable gets redefined. Intuitively, the set of all instructions that contribute to the liveness of the variable by using it. The product of the semi-lattices of all variables in the program will be the full semi-lattice
Meet operator or lattice diagram	$\cup$
Is there a top element? If yes, what is it?	$\emptyset$
Is there a bottom element? If yes, what is it?	U
Transfer function of a basic block	Assume every instruction is a basic block. For each variable x, apply the following: Set kill and gen set to empty first. If $d : x = ...$, kill set = U If $d : ... = a + b$, x in the RHS, gen set = { d }. IN[B][x] = (OUT[B][x] - kill) $\cup$ gen.
Boundary condition initialization	Top
Interior points initialization	Top

Then our criteria becomes:

If a variable x is live at p, then we cannot move speculative definition d to x above p, unless ANT_INS[x][p] = {d}.

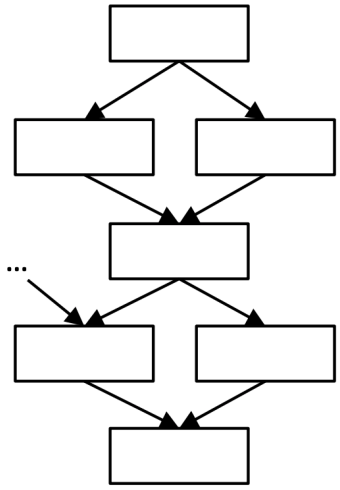
Now, ANT_INS[x][B3].in = $\{x = x + 1\}$, ANT_INS[x][B2].in = {}, ANT_INS[x][B1].out = $\{x = x + 1\}$. So our analysis will now correctly allow us to move $x = x + 1$ past these points, up in B1, while rejecting the movement of $x = 1$.

Lastly, imagine if there's another block $x = x + 2$ parallel to B2 and B3, with B1 as its only predecessor and B4 as its only successor. You can verify yourself that ANT_INS[x][B1].out = $\{x = x + 1, x = x + 2\}$, and our analysis will still behave correctly by rejecting all code motions.

(Below is the original algorithm for your reference. We are only looking at the two innermost layers of the loop when we are scheduling instructions off of a region. The algorithm will have removed the back edges for us so we are only dealing with a DAG of basic blocks. See 10.4 in the book for more details.)

Algorithm

```
Compute data dependences;
For each region from inner to outer {
   For each basic block B in prioritized topological order {
      CandBlocks = ControlEquiv{B} ∪
                   Dominated-Successors{ControlEquiv{B}};
      CandInsts = ready operations in CandBlocks;
      For (t = 0, 1, ... until all operations from B are scheduled)  {
         For (n in CandInst in priority order) {
            if (n has no resource conflicts at time t) {
               S(n) = < B, t >
               Update resource commitments
               Update data dependences
            }}
         Update CandInsts;
      }}}
```



Node *d* dominates node *n* in a graph (*d* dom *n*):

- if every path from the start node to *n* goes through *d* (a node dominates itself)

Priority functions: non-speculative before speculative