ECE 468 & 573
Problem Set 4: Local register allocation, instruction scheduling.

For the following problems, consider the following piece of three-address code:

```
1.  READ(A) {A}
2.  READ(B) {A, B}
3.  C = A + B {A, B, C}
4.  D = A + B {A, C, D}
5.  B = C * D {A, B, C}
6.  T1 = A + C {T1, A, B, C}
7.  T2 = T1 + C {T2, A, B, C}
8.  F = A + C {T2, A, B, F}
9.  C = F + B {T2, A, C}
10. G = A + C {T2, G}
11. T3 = T2 + G {T3}
12. WRITE(T3)
```

1. Perform liveness analysis on this piece of code, assuming that it is the entire code of the program. Show which variables are live after each of the statements in the program.

   Answer: The live sets are shown next to the instructions above.

2. For the next two subproblems, we will perform register allocation using bottom-up register allocation as presented in the notes. READ(X) defines the value of X (i.e., it assigns to X). WRITE(X) reads the value of X to write it out to the screen.

   Show what code would be generated for each 3AC instruction. Use LOAD X Rx to load from a variable/temporary into a register, STORE Rx X to store from a register into a variable/temporary, Rx = Ry + Rz for addition, and Rx = Ry * Rz for multiplication.

   When choosing registers to allocate, always allocate the lowest-numbered register available. When choosing registers to spill, choose the non-dirty register that will be used farthest in the future. In case all registers are dirty, choose the register that will be used farthest in the future. In case of a tie, choose the lowest-numbered register.

   If a load or store is the result of a spill (kicking a value out of a register earlier than required, or loading one of those values back into the register), indicate that.

   - Perform register allocation for a machine with 4 registers.

     Answer:
     When we start out, none of the registers are allocated.

```
R1:  R2:  R3:  R4:
```

For each 3AC instruction, we will explain the steps of register allocation and show the assembly instructions that are generated.

READ(A) There are no source operands in this instruction, so we do not need to ensure that anything is in a register. Then we find a register for A, R1:

```
READ(R1) // R1:  A* R2:  R3:  R4:
```

READ(B) We put B in R2:

```
READ(R2) // R1:  A* R2:  B* R3:  R4:
```

C = A + B We ensure that A and B are in registers (which they are), then put C in R3:

```
R3 = R1 + R2 // R1:  A* R2:  B* R3:  C* R4:
```

D = A + B We ensure that A and B are in registers. We then free R2, because B is now dead. We do not generate a store from this free, because even though B is dirty, it is not live. We then put D in R2:

```
R2 = R1 + R2 // R1:  A* R2:  D* R3:  C* R4:
```

B = C * D We ensure that C and D are in registers. We then free R2, because D is dead. We then put B in R2:

```
R2 = R3 * R2 // R1:  A* R2:  B* R3:  C* R4:
```

T1 = A + C We ensure that A and C are in registers. We then put T1 in R4:

```
R4 = R1 + R3 // R1:  A* R2:  B* R3:  C* R4:  T1*
```

T2 = T1 + C We ensure that T1 and C are in registers. We then free T1, and put T2 in R4:

```
R4 = R4 + R3 // R1:  A* R2:  B* R3:  C* R4:  T2*
```

F = A + C We ensure that A and C are in registers, then free C. We put F in R3:

```
R3 = R1 + R3 // R1:  A* R2:  B* R3:  F* R4:  T2*
```

C = F + B We ensure that F and B are in registers. We then free F and B, and put C in R2. Note that R3 is now just empty.

```
R2 = R3 + R2 // R1:  A* R2:  C* R3:  R4:  T2*
```

G = A + C We ensure that A and C are in registers, then free them, and put G in R1:

```
R1 = R1 + R2 // R1:  G* R2:  R3:  R4:  T2*
```

T3 = T2 + G We ensure that T2 and G are in registers, then free them and put T3 in R1:

```
R1 = R4 + R1 // R1:  T3* R2:  R3:  R4:
```

WRITE(T3) We write out R1!

```
WRITE(R1)
```

- Perform register allocation for a machine with 3 registers.

  Answer:
  When we start out, none of the registers are allocated.
  `R1:  R2:  R3:`
  READ(A) We put A in R1:
  `READ(R1) // R1:  A* R2:  R3:`
  READ(B) We put B in R2:
  `READ(R2) // R1:  A* R2:  B* R3:`
  C = A + B We ensure that A and B are in registers and put C in R3:
  `R3 = R1 + R2 // R1:  A* R2:  B* R3:  C*`
  D = A + B We ensure that A and B are in registers. We then free R2, because B is dead, and put D in R2.
  `R2 = R1 + R2 // R1:  A* R2:  D* R3:  C*`
  B = C * D We ensure that C and D are in registers. We then free R2, because D is dead, and put B in R2:
  `R2 = R3 * R2 // R1:  A* R2:  B* R3:  C*`
  T1 = A + C We ensure that A and C are in registers. We then must find a register for T1, but all the registers are taken. B is used farthest away, so we free it. Because B is dirty, we must generate a store. *This store is a spill.* We then put T1 in R2.
  `ST R2, B R2 = R1 + R3 // R1:  A* R2:  T1* R3:  C*`
  T2 = T1 + C We ensure that T1 and C are in registers, then free T1. We can then put T2 in R2:
  `R2 = R2 + R3 // R1:  A* R2:  T2* R3:  C*`
  F = A + C We ensure that A and C are in registers, then free C. We put F in R3:
  `R3 = R1 + R3 // R1:  A* R2:  T2* R3:  F*`
  C = F + B F is in a register, but we need to pull C into a register. We can't put it in R3, because F is used immediately. We kick T2 out of its register, because A is used sooner. Because T2 is dirty, we generate a store. *This is spill code.* We then load B into R2. *This load is spill code, since we're only loading B again because we were forced to free it.* We then free B and F, and put C in R2.
  `ST R2, T2 LD B, R2 R2 = R3 + F2 // R1:  A* R2:  C* R3:`
  G = A + C We ensure that A and C are in registers, then free them and put G in R1:
  `R1 = R1 + R2 // R1:  G*`

T3 = T2 + G We load T2 into R2. *This is spill code.* We then free T2 and G and put T3 in R1:

```
LD T2, R2 R1 = R1 + R2 // R1:  T3*
```

WRITE(T3) We write out R1!

```
WRITE(R1)
```

3. Assume you have a machine with 2 ALUs (ALU0 and ALU1) and one LD/ST unit. Your instruction set consists of 5 instructions: ADD, MUL, DIV, LOAD and STORE. An ADD instruction can be performed on either ALU, and takes one cycle. A MUL instruction can only be performed on ALU0, and takes two cycles. A DIV instruction can only be performed on ALU1, and takes three cycles. A LOAD takes up either of the ALUs for one cycle, then the LD/ST unit for two cycles. A STORE takes up either of the ALUs for one cycle, then the LD/ST unit for one cycle. All of the functional units are fully pipelined. Draw the reservation tables for the instructions.

   Answer:

   ADD1:

ALU0	ALU1	LD/ST
x		

   ADD2:

ALU0	ALU1	LD/ST
	x	

   MUL: Note that the ALU is fully pipelined. Even though the MUL has a two-cycle latency, it only occupies the ALU for one cycle.

ALU0	ALU1	LD/ST
x		

   DIV: Just like MUL, DIV only occupies its ALU for one cycle even though it's a multi-cycle instruction.

ALU0	ALU1	LD/ST
	x	

   LOAD1: Note that because the LD/ST unit is pipelined, the LOAD only occupies the LD/ST unit for one cycle:

ALU0	ALU1	LD/ST
x		
		x

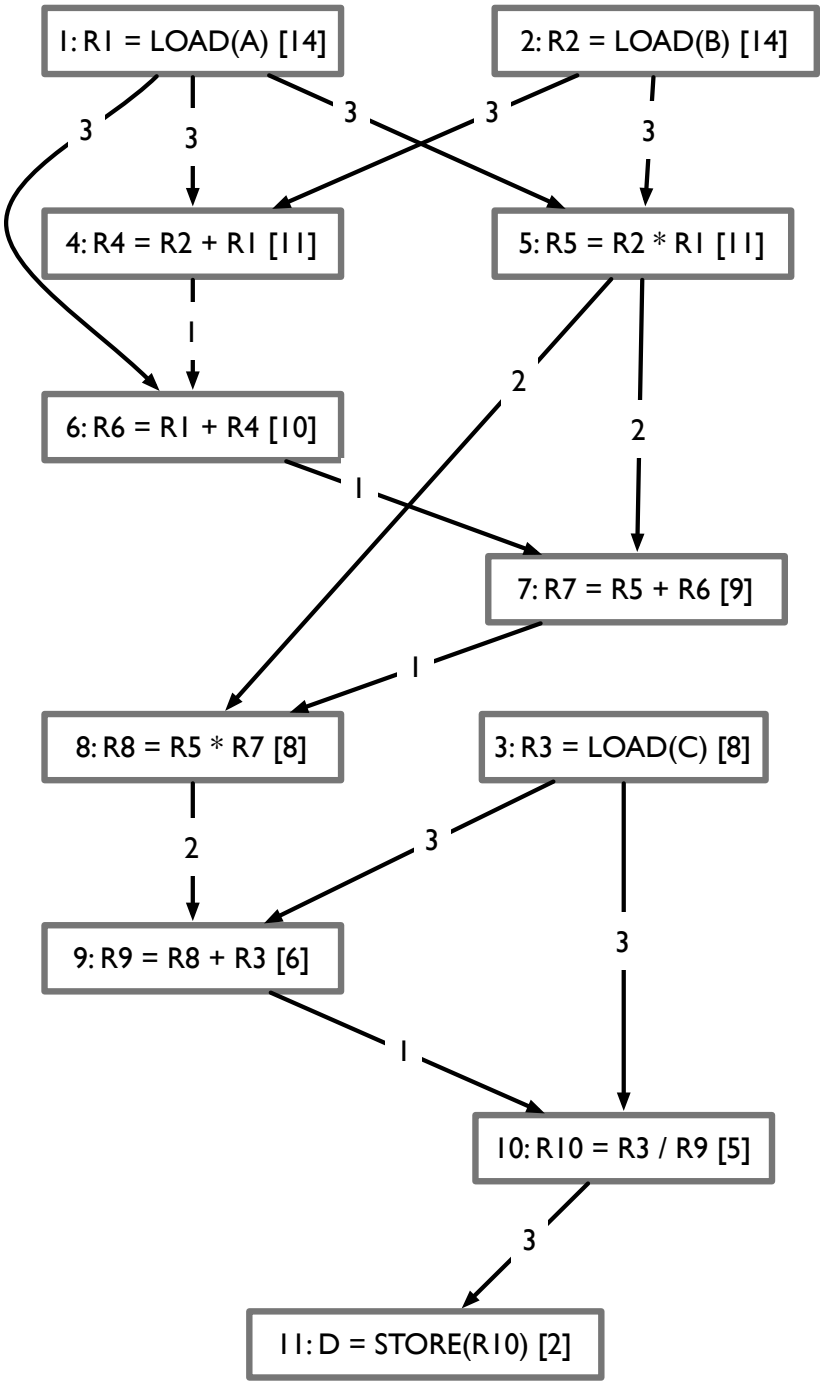
   LOAD2:

ALU0	ALU1	LD/ST
	x	
		x

Interestingly, even though STORE is a two-cycle instruction instead of a 3-cycle instruction like LOAD, its reservation tables look the same (since the LD/ST unit is pipelined).

4. Draw the data dependence graph for the following piece of code. Don't forget to label the dependence edges with their latency.

```
R1 = LOAD(A)
R2 = LOAD(B)
R3 = LOAD(C)
R4 = R2 + R1
R5 = R2 * R1
R6 = R1 + R4
R7 = R5 + R6
R8 = R5 * R7
R9 = R8 + R3
R10 = R3 / R9
D = STORE(R10)
```



5. What are the "heights" of the instructions in the DDG?

Answer: The heights are given in brackets in the DDG above.

6. Give a schedule for the program, using height-based list scheduling. If there is a tie in heights, break the tie by scheduling the instruction that is earlier in the program order.

Cycle	ALU0	ALU1	LD/ST
0	1		
1	2		1
2	3		2
3			3
4	5	4	
5	6		
6	7		
7	8		
8			
9	9		
10		10	
11			
12			
13	11		
14			11