ECE 468 & 573
Problem Set 4: Instruction scheduling & loop transformations

Instruction scheduling

1. Assume that your machine has two ALUs, which can execute ADDs in a single cycle, SUBs in two cycles, and *one* of the two ALUs can execute MULs in two cycles. There is also a single LD/ST unit. LOADs take up the ALU for 1 cycle and the LD/ST the next, while STs take up the LD/ST for one cycle. Draw the reservation tables for this machine.

   Answer:

   ADD 1:

ALU1	ALU2	LD/ST
X		

   ADD 2:

ALU1	ALU2	LD/ST
	X	

   SUB 1:

ALU1	ALU2	LD/ST
X		
X		

   SUB 2:

ALU1	ALU2	LD/ST
	X	
	X	

   MUL 1:

ALU1	ALU2	LD/ST
X		
X		

   LOAD 1:

ALU1	ALU2	LD/ST
X		
		X

   LOAD 2:

ALU1	ALU2	LD/ST
	X	
		X

STORE:

ALU1	ALU2	LD/ST
		X

2. (ECE 573 only): How many states would the scheduling automaton for this machine have?

   Answer:

   This is somewhat tricky. The states in the automaton need two cycles worth of state, because some of the instructions have a latency of two cycles. However, not every possible combination of functional unit occupancy can occur in the automaton. For example, if the LD/ST unit is occupied in the second cycle of a state, that is because a LOAD has been issued to get to that state. However, this means that one of the ALUs will be taken up with the LOAD instruction in the first cycle, so there is no way for that ALU to be occupied in the second cycle (which can only happen if a MUL or a SUB is scheduled). We can break down the possible states through the following two cases:

   Case 1: The LD/ST unit is occupied in the second cycle. The LD/ST unit can have either occupancy state in the first cycle. One of the ALUs has to have the LOAD scheduled, and the other can have three possible states: an instruction scheduled in the first cycle (e.g., an ADD), an instruction scheduled in neither cycle, or an instruction scheduled in both cycles (e.g, a SUB). This means that in case 1, there are 6 possible states.

   Case 2: The LD/ST unit is unoccupied in the second cycle. This means that a LOAD is not scheduled. The LD/ST unit can have either occupancy state in the first cycle. Each of the ALUs can be in one of three states, as above. This means that there are 18 possible states.

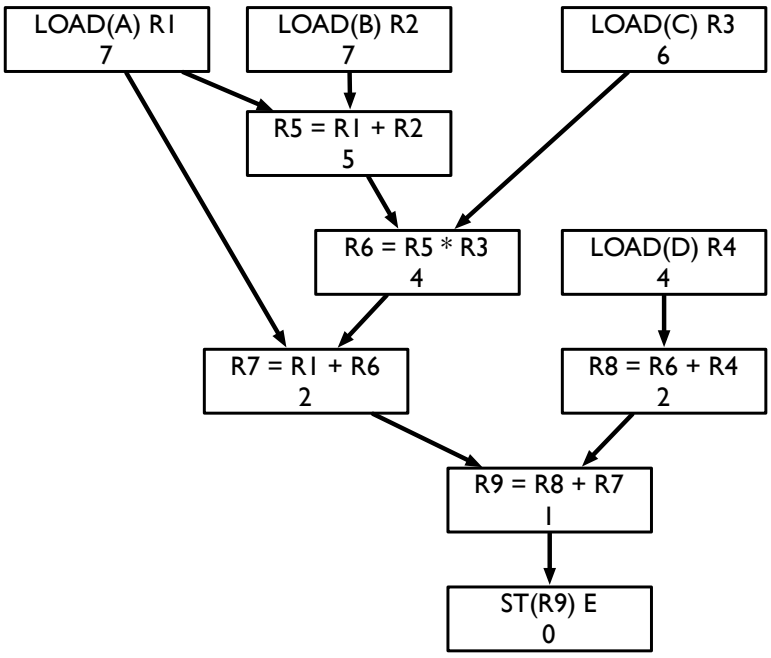
   In total, then, there are 24 possible states.

3. Draw the scheduling DAG for the following program:

```
1.  LOAD(A) R1
2.  LOAD(B) R2
3.  LOAD(C) R3
4.  LOAD(D) R4
5.  R5 = R1 + R2
6.  R6 = R5 * R3
7.  R7 = R1 + R6
8.  R8 = R6 + R4
```

```
9.  R9 = R8 + R7
10. ST(R9) E;
```

Answer:
The DDG is given below. The heights of each node are listed below the instruction.



4. Give the schedule you obtain after performing height-based list scheduling on the above code.

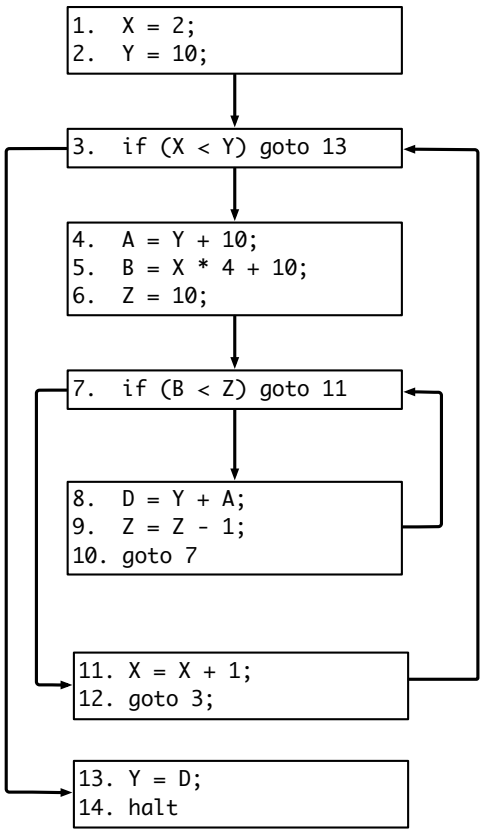
Cycle	ALU1	ALU2	LD/ST	Sched. Instrs.
1	X			1
2	X		X	2, (rest of 1)
3	X		X	3, (rest of 2)
4	X	X	X	4, 5, (rest of 3)
5	X		X	6, (rest of 4)
6	X	X		8, (rest of 6)
7	X			7
8	X			9
9			X	10

Loop transformations (I)
For the following problems, consider the code below:

```
1.   X = 2;
2.   Y = 10;
3.   if (X < Y) goto 13
4.     A = Y + 10;
5.     B = X * 4 + 10;
6.     Z = 10;
7.     if (B < Z) goto 11
8.        D = Y + A;
9.        Z = Z - 1;
10.        goto 7;
11.    X = X + 1;
12.    goto 3;
13.  Y = D;
14.  halt;
```

1. Draw the CFG for the code above. Identify the loops in the code.

   Answer:



The loop headers are instructions 3 and 7. The loops are: {3, 4, 5, 6, 7, 8, 9, 10, 11, 12} and {7, 8, 9, 10}

2. Which statements are loop invariant? Can they be moved outside their enclosing loop? Show the code that results after hoisting any loop invariant code outside the loop.

Answer:
Statement 4 is loop invariant (Y is only defined outside the loop). Statement 8 is loop invariant (Y is only defined outside the loop, and A is only defined by statement 4, which is loop invariant). Statement 4 can be moved outside the loop (A is only defined once inside the loop, A is not live coming into the loop, nor is A live at any exit of the loop). Statement 8 *cannot* be moved outside either loop: D is only defined once in the loop, but statement 4 does not dominate every loop exit, and D is live outside (both) loops.

The code after moving loop invariant code would swap statements 3 and 4:

```
1.   X = 2;
```

```
2.   Y = 10;
3h.  A = Y + 10;
3.   if (X < Y) goto 13
5.     B = X * 4 + 10;
6.     Z = 10;
7.     if (B < Z) goto 11
8.        D = Y + A;
9.        Z = Z - 1;
10.        goto 7;
11.    X = X + 1;
12.    goto 3;
13.  Y = D;
14.  halt;
```

3. Identify the induction variables in this code. Show the code that results after performing any possible strength reduction.

   Answer:
   Induction variables: X and Z. Strength reduction is possible for B, as it is a mutual induction variable of X:

```
1.   X = 2;
2.   Y = 10;
3h.  A = Y + 10;
3h'. B' = X * 4 + 10;
3.   if (X < Y) goto 13
5.     B = B'
6.     Z = 10;
7.     if (B < Z) goto 11
8.        D = Y + A;
9.        Z = Z - 1;
10.        goto 7;
11.    X = X + 1;
11'.   B' = B' + 4;
12.    goto 3;
13.  Y = D;
14.  halt;
```

4. Show the code after performing any possible linear test replacement.

   Answer:

We can now perform linear test replacement on X. Note that the complex expression in the loop test is actually loop invariant, so another round of loop invariant code motion would let us hoist it outside the loop.

```
1.   X = 2;
2.   Y = 10;
3h.  A = Y + 10;
3h'. B' = X * 4 + 10;
3.   if (B' < (Y * 4 + 10)) goto 13
5.     B = B'
6.     Z = 10;
7.     if (B < Z) goto 11
8.        D = Y + A;
9.        Z = Z - 1;
10.        goto 7;
11'.   B' = B' + 4;
12.    goto 3;
13.  Y = D;
14.  halt;
```

Loop transformations (II)
Consider a machine that *does not have a cache*. For the following loop transformations, explain whether the transformation would still be worth doing or not.

1. Loop interchange — This is not worth doing, as it does not change instruction overhead, but only locality. Locality effects are irrelevant without a cache.

2. Loop fusion — This is still worth doing. While there may not be any locality benefits from loop fusion, fusing loops can reduce instruction overhead (and, if a compiler is clever, lead to better use of registers).

3. Loop unrolling — This is still worth doing as unrolling reduces instruction overhead.