ECE 468

Problem Set 5: Local register allocation, instruction scheduling. **Solutions**

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

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

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

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.

```
1.  READ(A)       {A, D}
    READ R1;      (R1: A*)
2.  READ(B)       {A, B, D}
    READ R2;      (R1: A*, R2: B*)
3.  C = A + B     {A, B, C, D}
    R3 = R1 + R2; (R1: A*, R2: B*, R3: C*)
4.  A = A + B     {A, C, D}
    R1 = R1 + R2; (R1: A*, R3: C*)
5.  B = C * D     {A, B, C, D}
    LD D, R2;
    R4 = R3 * R2; (R1: A*, R2: D, R3: C*, R4: B*)
6.  T1 = C * D    {T1, A, B, C}
    R2 = R4 * R2; (R1: A*, R2: T1*, R3: C*, R4: B*)
7.  T2 = T1 + C   {T2, A, B}
    R2 = R2 + R3; (R1: A*, R2: T2*, R4: B*)
8.  F = A + B     {T2, F, B}
    R1 = R1 + R4; (R1: F*, R2: T2*, R4: B*)
9.  C = F + B     {T2, F, B}
    R3 = R1 * R4; (R1: F*, R2: T2*, R4: B*) -- C is dead
10. G = T2 + B    {F, B}
    R2 = R2 + R4; (R1: F*, R4: B*) -- G is dead
11. T3 = F + B    {T3}
    R1 = R1 + R4; (R1: T3)
12. WRITE(T3)     { }
    WRITE R1;     ()
```

- Perform register allocation for a machine with 3 registers.

```
1.  READ(A)       {A, D}
    READ R1;      (R1: A*)
```

```
2.  READ(B)       {A, B, D}
    READ R2;      (R1: A*, R2: B*)
3.  C = A + B     {A, B, C, D}
    R3 = R1 + R2; (R1: A*, R2: B*, R3: C*)
4.  A = A + B     {A, C, D}
    R1 = R1 + R2; (R1: A*, R3: C*)
5.  B = C * D     {A, B, C, D}
    LD D, R2;
    ST R1, A; //spill R1 to memory
    R1 = R3 * R2; (R1: B*, R2: D, R3: C*)
6.  T1 = C * D    {T1, A, B, C}
    R2 = R3 * R2; (R1: B*, R2: T1*, R3: C*)
7.  T2 = T1 + C   {T2, A, B}
    R2 = R2 + R3; (R1: B*, R2: T2*)
8.  F = A + B     {T2, F, B}
    LD A, R3;
    R3 = R3 + R1; (R1: B*, R2: T2*, R3: F*)
9.  C = F + B     {T2, F, B}
    ST R3, F; //spill R3 to memory
    R3 = R3 + R1; (R1: B*, R2: T2*) -- C is dead
10. G = T2 + B    {F, B}
    R2 = R2 + R1; (R1: B*) -- G is dead
11. T3 = F + B    {T3}
    LD F, R2;
    R1 = R2 + R1; (R1: T3)
12. WRITE(T3)     { }
    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, LOAD and STORE. An ADD instruction can be performed on either ALU, and takes one cycle. A MUL instruction can be performed on ALU0 or ALU1, and takes two 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	

MUL1: 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		

MUL2: 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	

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

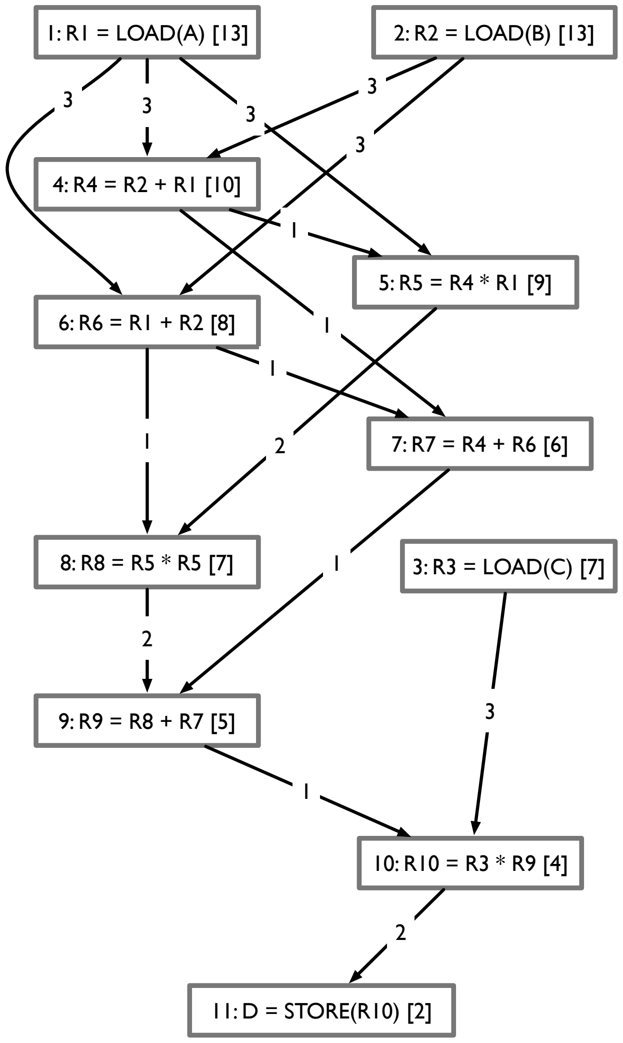
Interestingly, even though STORE is a two-cycle instruction instead of a 3-cycle instruction like LOAD, its reservation tables look the same (since the LD/ST unit is pipelined).

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 = R4 * R1
R6 = R1 + R2
R7 = R4 + R6
R8 = R5 * R6
R9 = R8 + R7
```

```
R10 = R3 * R9
D = STORE(R10)
```

Answer:



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

   Answer: The heights are in brackets 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.

   Answer:

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