ECE 468 & 573 — Midterm 2

November 5, 2012

Name: _____

Purdue email: _____

Please sign the following:
I affirm that the answers given on this test are mine and mine alone. I did not receive help from any person or material (other than those explicitly allowed).

X _____

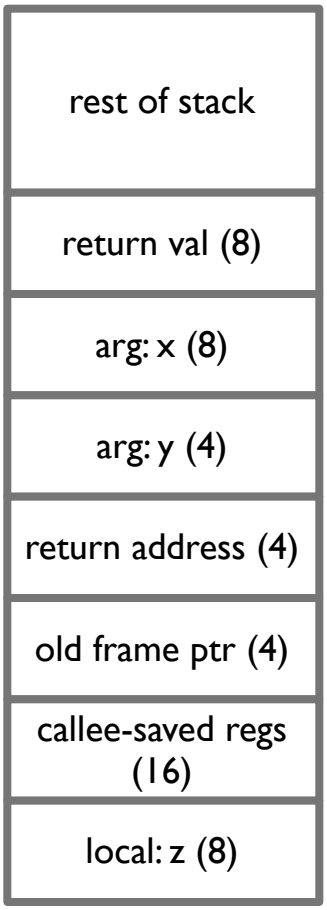
Part	Points	Score
1	10	
2	30	
3	40	
4	20	
Total	100	

Part 1: Function calls and semantic actions (10 pts)

1) For the following function, show what the activation record for calling the function would be, including both what the caller sets up and what the callee sets up. Make the following assumptions: (i) the machine has four registers, plus an FP register and an SP register, and all the registers are four bytes; (ii) the program is using callee saves; (iii) the function doesn't have any spilled registers. Show the stack growing down (as in the notes). For each entry in the activation record, show how many *bytes* that entry takes up. (7 pts)

```
double foo (double x, int y) {
     double z;
     z = x * y;
     return z;
}
```

The most common mistake here was forgetting to add space for z on the stack. That was worth 2 pts



2) Why do we distinguish between R-values and L-values when generating code? (3 pts)

R-values are data and L-values are addresses. We must distinguish because when we need to use a value in a temporary, we need to know whether to load from it, or whether we can just use it directly

Part 2: Register allocation (30 pts)

For the next 3 problems, consider the following code (assume this is the full program):

```
1:  A = 7
2:  B = A + C
3:  A = A + C
4:  C = B + D
5:  E = C + B
6:  B = E + A
7:  A = C + B
8:  B = B + A
9:  WRITE(A) //this counts as a use of A
10: WRITE(B) //this counts as a use of B
```

1) Show which variables are live *after* each instruction (assume nothing is live at the end of the code) (10 pts) (1 pt each)

1	A, C, D
2	A, B, C, D
3	A, B, D
4	A, B, C
5	A, C, E
6	C, B
7	A, B
8	A, B
9	B
10	{}

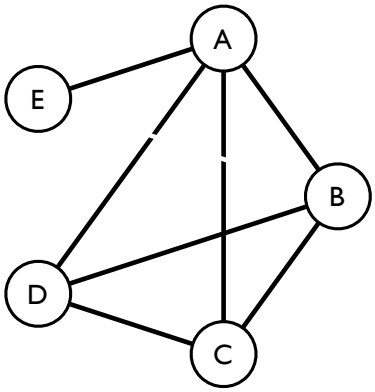
2) Can any instructions be simplified with CSE? Which ones? (4 pts)

Yes: instruction 3 can be replaced with A = B (instruction 7 cannot, because B changes in instruction 6)

3) Name (at least) two variables that could share the same register safely (1 pt)

D and E can share a variable (not live at the same time), or B and E.

4) Draw the interference graph for the code (5 pts)



Consider performing bottom up register allocation. For the following scenarios, show what code needs to be generated for the given three-address-code instruction and give the state of the registers after code generation (if a value in a register is dirty, mark it with a *) If variables need to be spilled, always spill the variable in the numerically lowest register first.

4) Before this instruction, the state of the registers is as follows:

R1	R2	R3
A	B*	C

a) What code is generated for `D = E + F` **where A, B, C and D are live after this instruction (3 points)?**

```
LOAD E, R1 //R1 can be freed immediately
ST R2, B //free R2
LOAD F, R2 //R2 can be freed immediately
R1 = R1 + R2 //reuse R1 because it was freed
```

b) What is the state of the registers *after* this code (2 points)?

R1	R2	R3
D*		C

5) Before the instruction, the state of the registers is as follows:

R1	R2	R3
D*	B*	E

a) What code is generated for `A = B + E` where A and D are live after this instruction (3 points)?

```
ST R2, B
R2 = R2 + R3 //Reuse R2, free R3
```

b) What is the state of the registers *after* this code (2 points)?

R1	R2	R3
D*	A*	

Part 3: Instruction Scheduling (40 pts)

For the following problems, assume a machine that has 2 ALUs, 1 MU and 1 L/S unit. The ALUs can execute ADDs with a single-cycle latency and SUBs with a two-cycle latency. The MU can execute MULs with a 2 cycle latency and DIVs with a three-cycle latency. LDs take two cycles, and occupy either ALU in the first cycle and the L/S unit in the second. STs occupy the L/S unit for one cycle

1) Draw the reservation tables for the following instructions: LD, ST, ADD, SUB, MUL, DIV (8 pts):

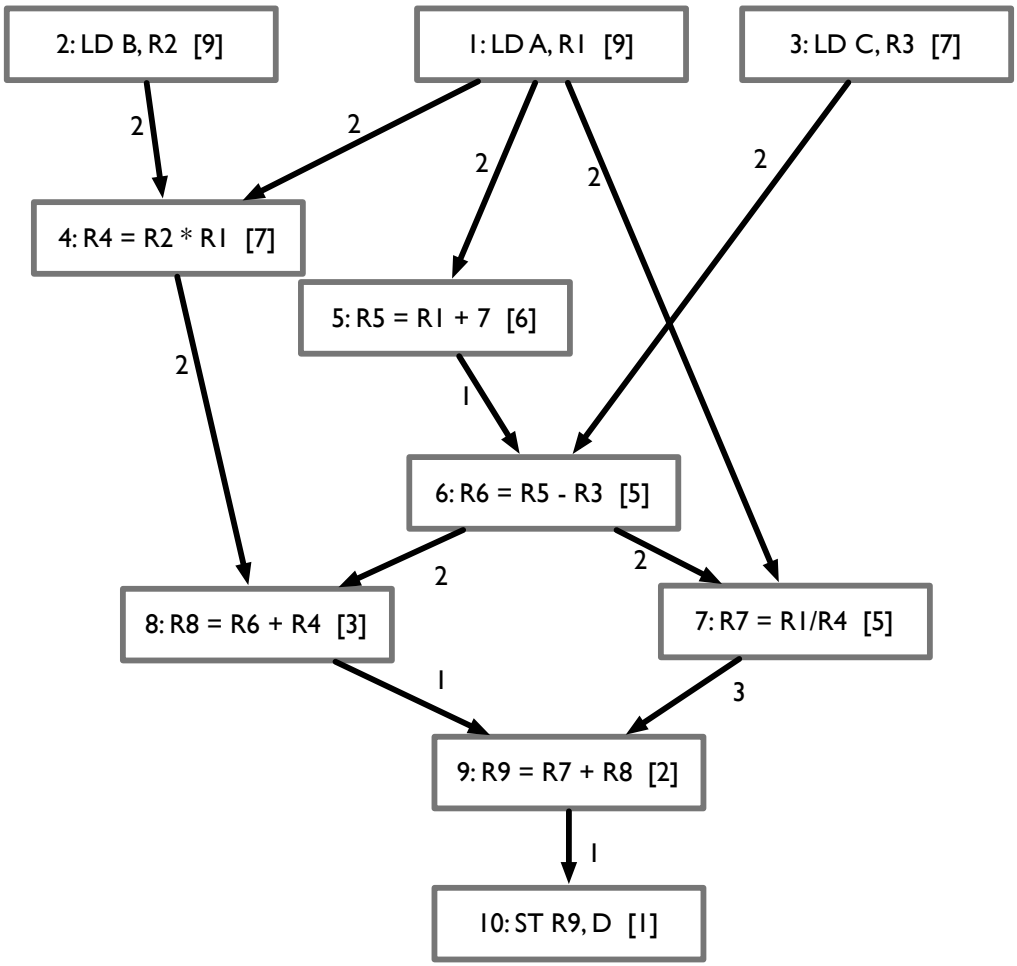
1 point per reservation table

LD: 2 reservation tables with an ALU occupied for one cycle and the LD/ST unit occupied in the next
ST: reservation table with LD/ST unit occupied for one cycle
MUL: reservation table with MU occupied for two cycles
DIV: reservation table with MU occupied for three cycles
SUB: 2 reservation tables, with an ALU occupied for two cycles
ADD: 2 reservation tables, with an ALU occupied for one cycle

2) Draw the data-dependence graph for the following piece of code, including latencies. Show the *heights* of each node in the graph (recall that the height of an instruction with no dependent instructions is its latency) (10 pts):

```
1:  LD A, R1; //Load A into R1
2:  LD B, R2;
3:  LD C, R3;
4:  R4 = R2 * R1;
5:  R5 = R1 + 7;
6:  R6 = R5 - R3;
7:  R7 = R1 / R4;
8:  R8 = R6 + R4;
9:  R9 = R7 + R8
10: ST R9, D;
```

Heights in brackets, latencies on edges.



1 pt per error, max of 2 pts for latency mistakes, 5 points for height mistakes and 3 pts for dependence mistakes.

3) For each instruction above, show in which cycle it will be executed if we use height-based list scheduling. If there is a tie in heights, give priority to the instruction earlier in program order. Show your work in the table below (16 points)

Cycle	ALU1	ALU2	MU	LD/ST	Inst(s) scheduled
1	1				1
2	2			1	1, 2
3	3	5		2	2, 3, 5
4			4	3	3, 4
5	6		4		4, 6
6	6		7		6, 7
7	8		7		7, 8
8			7		7
9	9				9
10				10	10
					2 points per instruction

4) Assume that the MU is now fully pipelined. Give a short sequence of code (you should not need more than two instructions—assume that all registers already have useful values so loads are unnecessary) where having a fully pipelined MU *does not* result in a faster schedule than the non-fully-pipelined MU. (6 pts)

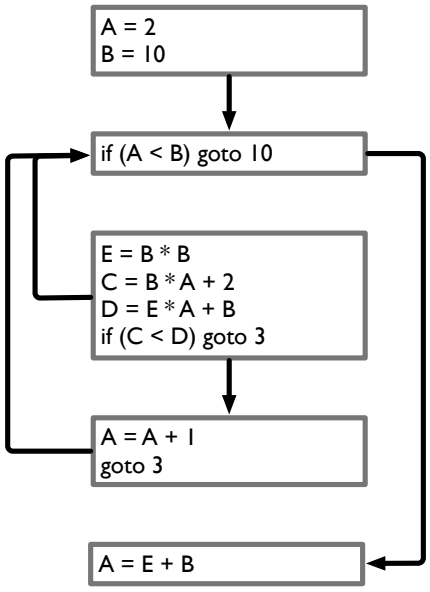
Many possible solutions. The best way is to have an instruction that uses the MU, and a second instruction that uses the *result* of the first instruction–the dependence keeps the second instruction from executing until the first finishes, whether or not the MU is pipelined.

Part 4: Loop optimizations (20 pts)

For the next 2 problems, consider the following code:

```
1:  A = 2
2:  B = 10
3:  if (A < B) goto 9
4:  E = B * B;
5:  C = B * A + 2
6:  D = E * A + B
7:  if (C < D) goto 3
8:  A = A + 1;
9:  goto 3
10: A = E + B
```

1) Draw the control flow graph (basic-block level) for this code. Each node should show all of the instructions for that basic block. (10 points)



1 point for missing edges (or an edge between the last two basic blocks
2 points for splitting basic blocks that should be combined (e.g., the third and fourth blocks)

2) Give the code that would be produced after performing *both* loop invariant code motion and strength reduction. For partial credit, identify any loop induction variable(s) and mutual induction variable(s). You do not have to do test replacement. (10 points)

Instruction 4 is loop invariant, but cannot be moved, because E is live outside the loop (if you move the instruction, then instruction 10 may do the wrong thing). I did not intend to test this, so I did not take points off if E was moved. However, if you correctly did not move E (and hence either did not perform strength reduction on D, or used some other technique to correctly do it), I gave you two bonus points.

Induction variable: A (2 points)
Mutual induction variables: C, D (1 point)

If you demonstrated that you knew which variables were induction variables in your code, I did not take off points for not explicitly stating that A was an induction variable and C & D were mutual induction variables.

```
1:  A = 2
2:  B = 10
2a: C' = B * A + 2 //1 point
2b: D' = B * B * A + B //1 point
3:  if (A < B) goto 9
4:  E = B * B;
5:  C = C' //1 point
6:  D = D' //1 point
7:  if (C < D) goto 3
8:  A = A + 1;
8a: C' = C' + B //1 point
8b: D' = D' + E //1 point
9:  goto 3
10: A = E + B
```