ECE 573 — Midterm 2

April 5, 2011

Name: ___KEY_____

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	25	
2	10	
3	18	
4	27	
5	20	
Total	100	

Part 1: Semantic actions and functions (25 pts)

1) Give an example with two functions, *foo* and *bar*, where foo calls bar, where passing all arguments to bar by *value-result* will give different behavior than passing the arguments by *reference*. You may use global variables. (8 pts):

```
int x;

foo () {
   x = 0;
   bar (x);
   print(x);
}

bar (int y) { y++; print(x); }
```

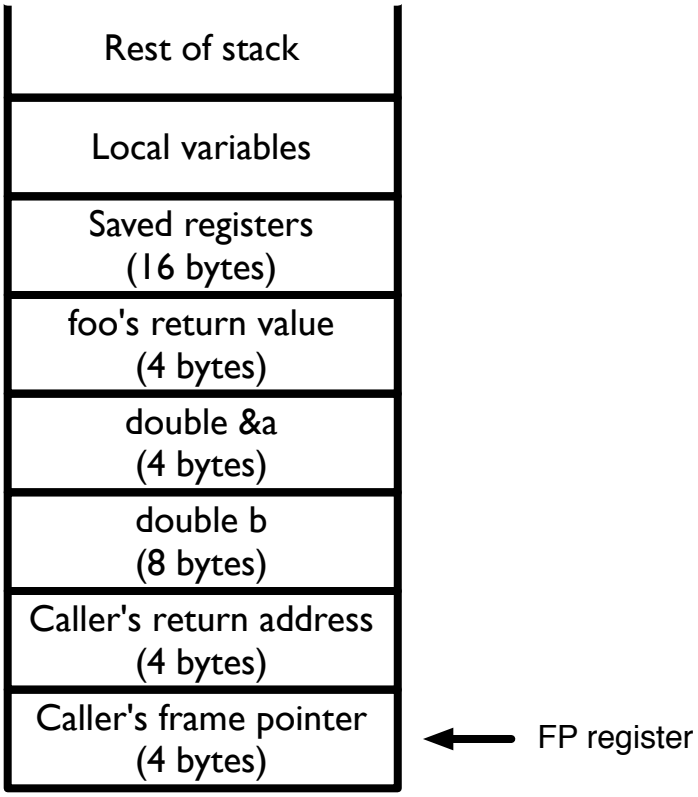
2) Give an example of a function *bar* where using the callee-saves convention will always be *at least as efficient* as using the caller-saves convention. (7 pts):

```
bar() {
}
```

Empty function means that we don't need any registers to generate code for bar, which means that callee-saves won't have to save anything.

Other alternate answers were acceptable, too (e.g., only one return site, so only one place to generate register restore code).

3) Here is a partial stack of a method being executed (the stack grows down). Show the stack after calling `int foo(double &a, double b)`. Note that the first argument is passed by reference, while the second is passed by value. Show the frame pointer, and note how much space each part of the stack occupies (32-bit ints and pointers, 64-bit doubles). Assume that we are using a *caller-saves* convention, and that the machine has 4 registers (plus FP & SP registers). Assume that there are no nested scopes or local variables in foo. (10 pts)



Part 2: Code generation (10 points)

For the next problems, consider a modified for-loop:

```
for_double (<init_stmt>; <test_exp>; <incr_stmt>) {<body>}
```

This loop executes the body of the for-loop *twice* in every iteration (meaning the body is executed twice per incr_stmt); "break" and "continue" statements operate as before, breaking out of the loop entirely, or skipping to the next incr_stmt, respectively.

1) Using a format like we did in the class slides, show what code you would generate for for_double. (10 pts):

```
     <init_stmt>
L0:  <test_exp>
     j!op B0; //if test_exp is false, jump
     <body>
     <body>
C1:  <incr_stmt>
     jmp L0;
B0:
```

I took off points for various inefficiencies. For example, using an extra register to store how many times you executed the body. Other errors actually change the semantics; executing incr_stmt too many times, or test_exp too many times (since test_exp might have side effects).

Part 3: Common subexpression elimination (18 pts)

For the next questions, consider the following piece of code:

```
1: A = B + C;
2: P = A + D;
3: B = B + C;
4: Q = A + D;
5: A = B + C;
6: R = B + C;
```

1) Assume there is no aliasing between variables. For each statement, list which expressions are "available" *after* the statement executes (6 pts)

1	B+C
2	B+C, A+D
3	A+D
4	A+D
5	B+C
6	B+C

2) What does the code look like after performing CSE (when eliminating a redundant expression, replace it with the variable that holds the calculated value of the expression) (6 pts)

```
1: A = B + C;
2: P = A + D;
3: B = A;
4: Q = P;
5: A = B + C;
6: R = A;
```

3) There are two variables which, if they were aliased, would make it so that *no* expressions are redundant. What are they? (6 pts)

A & B. The definitions of A & B in lines 1, 3 and 5 would keep A+D or B+C from ever being redundant. Note that the expressions might still become available at some point -- they just won't ever be able to be exploited.

Part 4: Register allocation (27 pts)

For the next problems, consider the following code:

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

1) Show which variables are live *after* each instruction (5 pts)

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

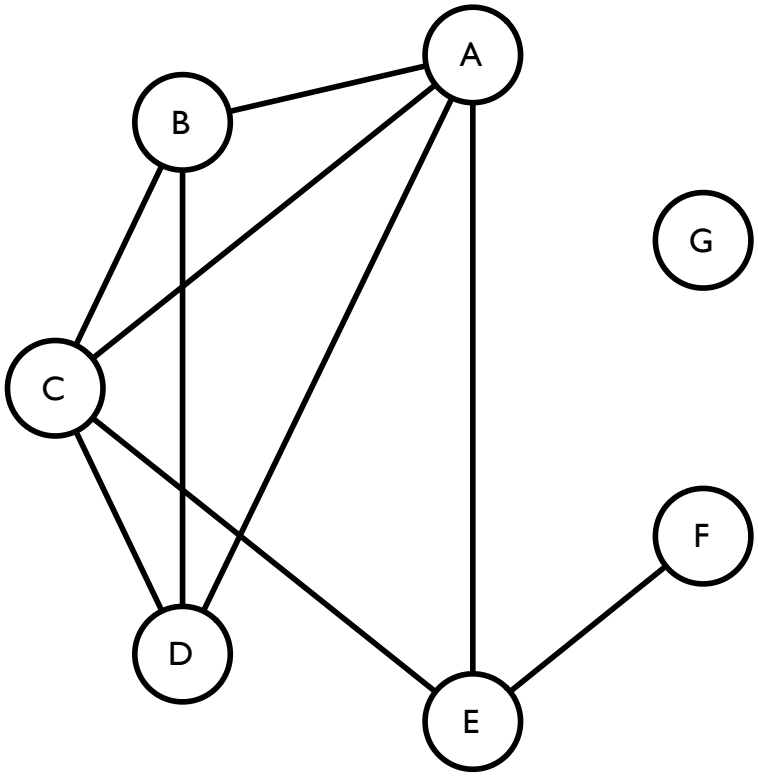
2) How many registers are needed to perform register allocation on this code without spills? (2 pts)

4 – there are 4 variables live in instruction 5.

3) Perform bottom-up register allocation on this piece of code, assuming there are 3 registers. At each instruction, show which variable is assigned to which register *after* the instruction is executed (if a register is freed, mark it as such even if it still holds a value). When a register needs to be spilled, pick the one whose value is next used the farthest away. If there is a tie, pick the lowest numbered register. If multiple registers are free when allocating registers, choose the lowest numbered one. Indicate where loads and stores due to spills happen (10 pts)

Inst	R1	R2	R3	Loads/Stores due to spills
1	A			
2	A	B		
3	A	C		B is freed as it stops being live
4	A	C	D	
5	A	B	D	Spill C. There is a tie between C & D, so we pick C. I also accepted picking D
6	E		D	Free A & B
7	E	C	A	Unspill C into R2, use C & D, free D, put A in R3
8	E	F		Free C & A
9	G			Free E & F
10				Free G

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



5) Show the order in which the graph will be simplified when performing register allocation via graph coloring. If there is a tie (multiple possible simplifications can occur in a step), choose the variable that appears earliest in alphabetical order. If you cannot simplify without spilling, *choose the variable with the fewest uses* next. Mark any variables that are potential spills with a "*". (5 pts)

F G E* D* A B C

There's a tie for uses between B and D, so an alternate order would be:

F G E* B* A C D

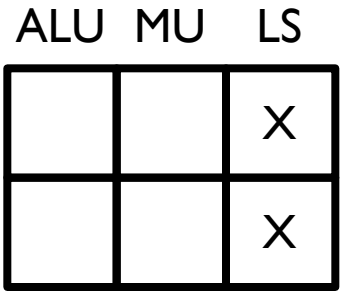
Note that many people spilled A first (rather than E). Although A interferes with the most other variables, gets used many times. So spilling A results in fewer spilled variables, but more actual spill operations.

Part 5: Instruction Scheduling (20 pts)

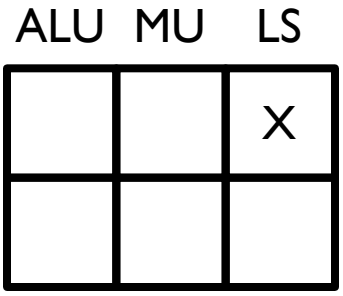
For the following problems, assume a machine that has 1 ALU that can execute adds and shifts, with a single-cycle latency, 1 *pipelined* MU that can do integer multiplication with a two-cycle latency and adds with a single cycle latency, and 1 *non-pipelined* LS unit that can execute loads with a two-cycle latency and stores with a one-cycle latency.

1) Draw the reservation tables for the following instructions: LD, ST, ADD, MUL, SHIFT (6 pts):

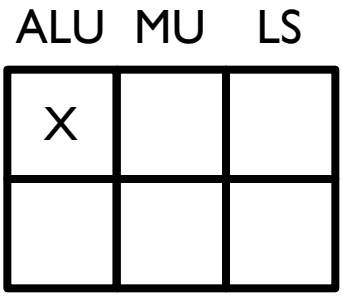
LD



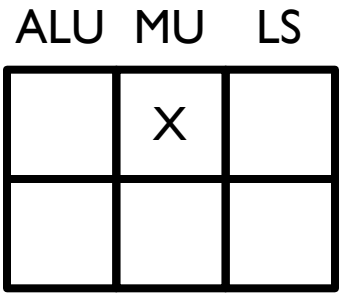
ST



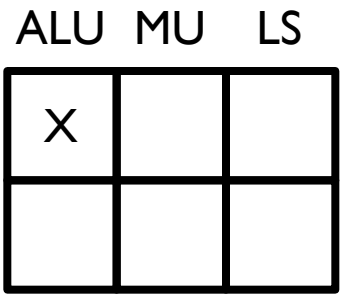
ADD



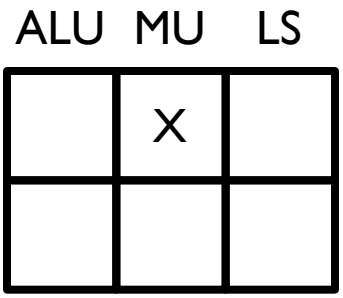
ADD



SHIFT



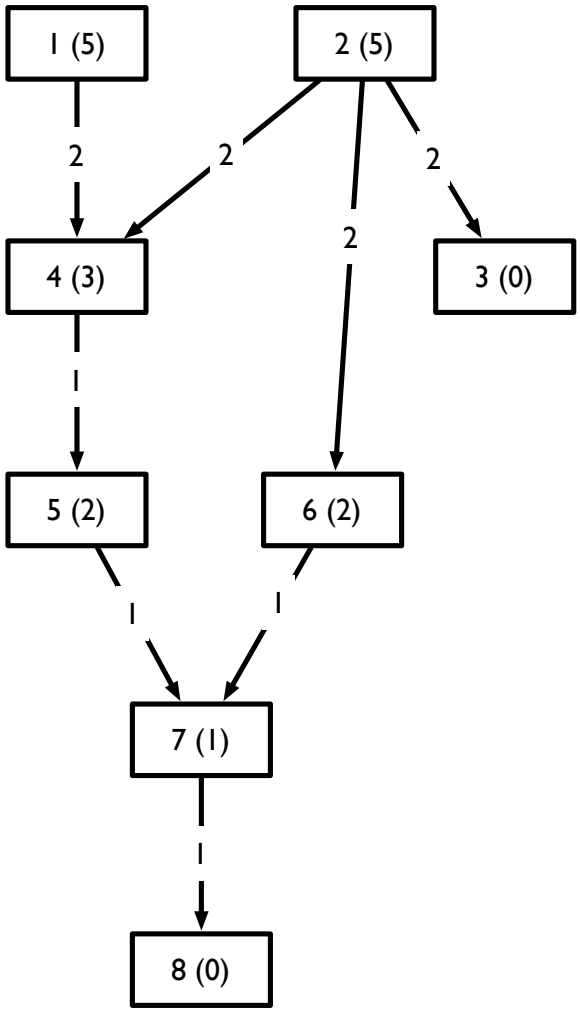
MUL



2) Draw the data-dependence graph for the following piece of code, including latencies. Show the *heights* of each node in the graph (8 pts):

```
1:  LD A, R1; //Load A into R1
2:  LD B, R2;
3:  R3 = R2 * 2;
4:  R4 = R1 + R2;
5:  R5 = R4 << 1; //Shift R4 by 1
6:  R6 = R2 + 1;
7:  R7 = R5 + R6
8:  ST R7, C; //Store R7 into C
```

DDG is below. The heights are in parentheses. Note that height is defined as the longest path from the instruction to the end of the DDG, so instr. 3 has a height of 0. I also accepted as correct people who offset heights by the latencies of instructions 8 and 3.



3) Consider the peephole optimization that converts "* 2" into "<< 1". Why might this optimization be useful on this machine *in general*? (3 pts)

On this machine, shifts have a latency of one cycle, but multiplies have a latency of two. In general, we would prefer to use the faster instruction.

4) Why might that peephole optimization be a bad idea *for this particular piece of code*? (3 pts)

This question was a little broken. The answer I was going for is that the MU isn't utilized very much, and the multiply isn't on the critical path, so it's OK to let it use the MU, whereas turning it into a shift would force it to use the oversubscribed ALU. However, in this particular piece of code, the total run time would be the same either way (because all the adds could just be moved over to the MU).

I accepted pretty much any answer that gave an argument for why it might or might not be useful.