ECE 468 — Midterm 2

November 4, 2010

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	30	
2	20	
3	15	
4	35	
Total	100	

Part 1: Semantic actions and functions (30 pts)

1) Explain when pass-by-value will produce the same results as pass-by-reference (2 pts):

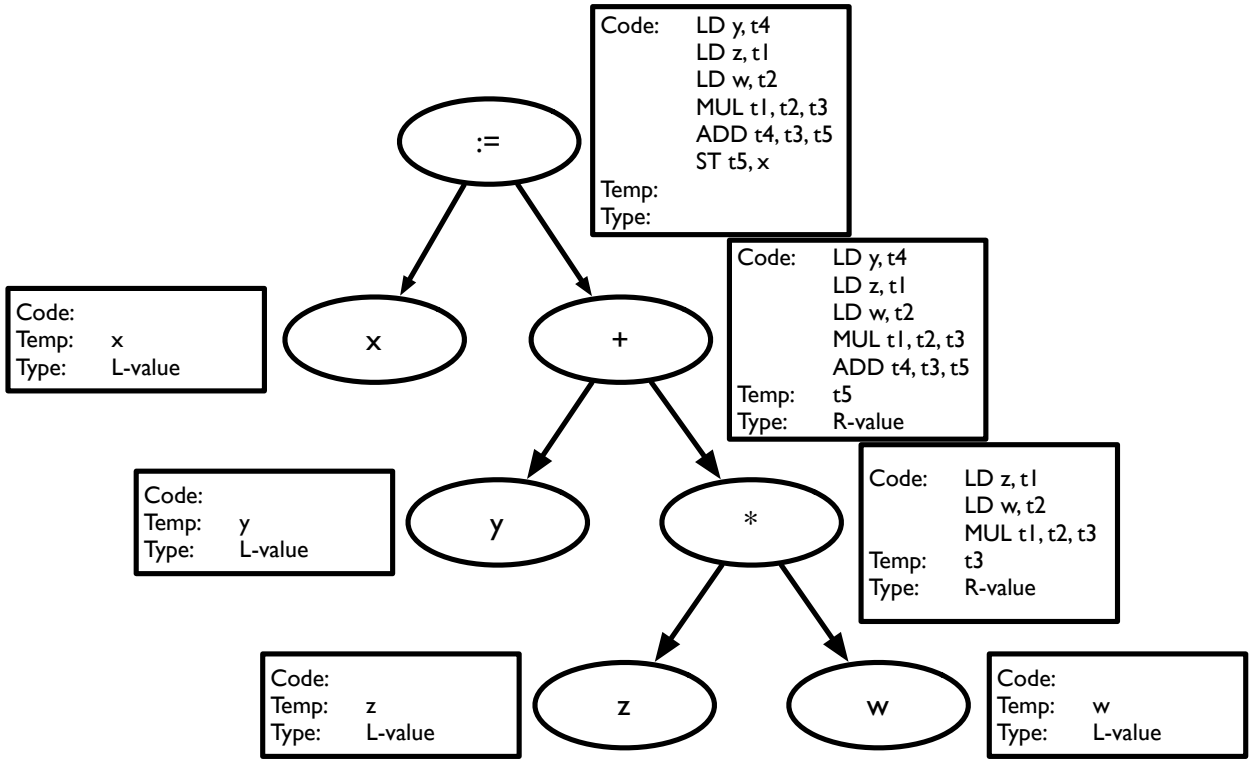
They will produce the same results if the arguments are not modified inside the function.

2) a) Give one reason that caller-saves is better than callee-saves (2 points). b) Give one reason that callee-saves is better than caller-saves (2 points).

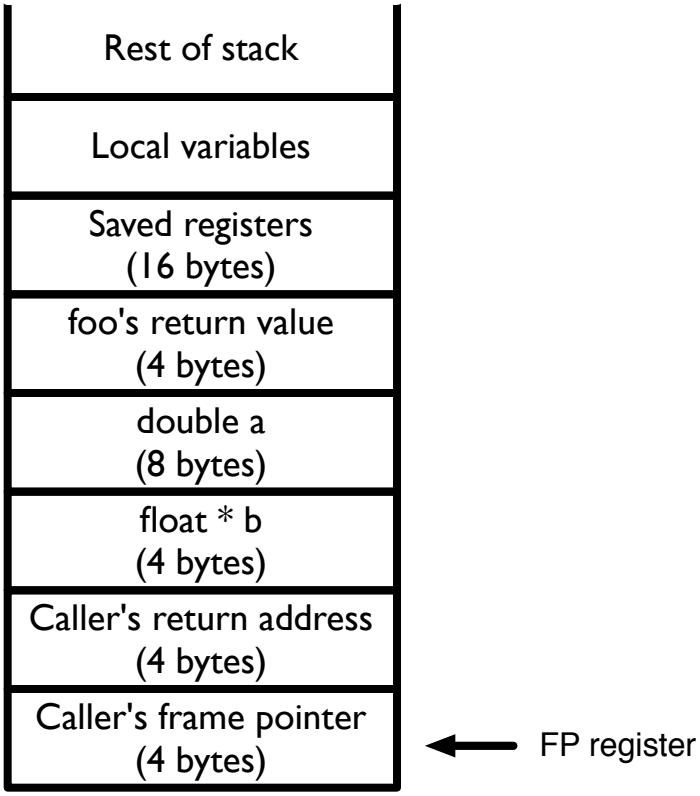
Callee-saves > Caller-saves: less code to generate to *save* the registers (only at function header)

Caller-saves > Callee-saves: easier to generate code to *restore* registers (don't have to look for all return points)

3) Draw the AST that would be generated for the expression `x := y + z * w` and for each AST node, show the data object that would be generated. For each data object, show the code that would be generated (if any), the temporary the result will be stored in, and whether the temporary is an R-value or an L-value. (14 pts)



3) Here is a partial stack of a method being executed (the stack grows down). Show the stack after calling `int foo(double a, float* b)`. 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 (20 points)

For the next problems, consider a for-loop:

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

Now consider a new statement, akin to a continue statement:

```
skip_one;
```

This statement *skips one iteration* of a for-loop. Rather than evaluating the incr_stmt once, like continue, it evaluates it *twice*. If either the skipped iteration or the next one would have broken out of the loop (because test_exp evaluates to false), then the loop should terminate. Otherwise, the loop will continue, having *skipped one iteration*. Think of skip_one as a continue statement that skips past the end of the current iteration and the next one, as well.

1) Using a format like we did in the class slides, show what code you would generate for a for-loop in order to support the skip_one statement (16 points):

```
 1:       <init_stmt>
 2:  L0:  <test_exp>
 3:       j!op F0
 4:       <body>
 5:       jmp C0
 6:  S0:  <incr_stmt>
 7:       <test_exp>
 8:       j!op F0
 9:  C0:  <incr_stmt>
10:       jmp L0
11:  F0:
```

2) Using your code for part 1 as a guide, show what code you would generate for a skip_one statement in the body of the for-loop (2 points):

```
jmp S0
```

3) Is there an optimization that could make the code generated for a for-loop more compact? If so, describe it (2 points):

Supporting “continue” doesn’t require extra instructions. Supporting “skip_one” does. If skip_one does not appear in the body of the for loop, then don’t generate the extra code (take out lines 5–8)

Part 3: Common subexpression elimination (15 pts)

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

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

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	A*Q
3	A*Q, B+C
4	B+C
5	B+C, B+D
6	B+D, A*P

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) (5 pts)

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

3) There are two variables which, if they were aliased, would make one additional expression redundant. Which are they (4 pts)?

If C and D were aliased, then we could reuse the results of B+C in instruction 5 (although note that we couldn't replace it with E = A, since A has been overwritten. We can replace it with E = P).

Also note that aliasing Q and P doesn't help. First, it kills A*Q in instruction 3, so we can't reuse it in instruction 4. Second, because instruction 4 redefines A, A*P wouldn't be available in instruction 6. A similar problem happens if we alias A & B.

Part 4: Register allocation (35 pts)

For the next problems, consider the following code:

```
1:  LD A, T1
2:  LD B, T2
3:  LD C, T3
4:  T4 = T1 * T2
5:  T5 = T1 + T3
6:  LD D, T6
7:  T7 = T2 + T5;
8:  T8 = T6 + T7;
9:  T9 = T1 + T7;
10: ST T9, A
```

1) Show which temporaries are live *after* each instruction (10 pts)

1	T1
2	T1, T2
3	T1, T2, T3
4	T1, T2, T3
5	T1, T2, T5
6	T1, T2, T5, T6
7	T1, T6, T7
8	T1, T7
9	T9
10	

2) Assume we have a machine with 3 registers available (not including spill registers). What temporaries get assigned to registers if we perform *top-down* register allocation. If there is a tie in the algorithm, choose the lowest numbered temporary. (5 pts)

T1 is used 3 times, T2 and T7 are used twice, so they will be put in registers.

3) Perform bottom-up register allocation on this piece of code. At each instruction, show which temporary 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 (20 pts)

Inst	R1	R2	R3	Loads/Stores due to spills
1	T1			
2	T1	T2		
3	T1	T2	T3	
4	T1		T3	Need to spill T2 (to find a register for T4). Since T4 is never used again, we can free R2
5	T1	T5		No spills, but T3 is never used again
6	T1	T5	T6	No spills because T2 has already been spilled
7	T7		T6	T1 spilled (to make room for loading T2). T2 and T5 never used again, so freed. T7 moves into R1
8	T7			T6 never used again. T8 put in R2, but then immediately freed (no use)
9	T9			T1 loaded into R2, then T1 and T7 are never used again, so freed.
10				Nothing used after this, so all registers freed.