Final Exam
CS6013

Maximum marks = 60, Time: 3hrs

28-Apr-2016

Read all the instructions and questions carefully. You can make any reasonable assumptions that you think are necessary; but state them clearly. It is your responsibility to write legibly. There are total five questions totaling 60 (+ 4 bonus) marks. Each twelve marks question will approximately require 35-40 minutes to answer. For questions that have sub-parts, the division for the sub-parts is mentioned in square brackets.

Leave the first page empty. Start each question on a new page. Think about the question before you start writing and write briefly. **The answer for any question (including all the sub-parts) should NOT cross more than two pages.** If the answer is spanning more than two pages, we will ignore the spill-over text. If you scratch/cross some part of the answer, you can get compensation space from the next page.

1. [12] **Runtime Management**

Differentiate between caller-save and callee-save registers [2]. Briefly (max 3-4 sentences) explain why the general purpose registers are divided into caller-save and callee-save registers [2]. If a architecture does not demarcate caller-save and callee-save registers, what has to be done at the time of function call, beginning of callee, before the return from callee, after the function call? [0.5 $\times$ 4]. For the code shown in the right hand side, show the generated code to be inserted at **C1**, **C2**, **C3**, and **C4** [1.5 $\times$ 4]. Assume the following:

- $R8-$R15 are caller save registers.
- $R16-$R23 are callee save registers.
- Return address is stored in register $R31
- All the arguments are passed on the stack.

```
void foo() {
   int a;  // assume it to be spilled
   ... // code uses $R8, $R9, $R16, $R17

   //  C1

   bar ($R8);

   //  C2

   ... // code uses $R8, $R16,
}
void bar (int x) {

   //  C3

   // ... code using $R23;
   // also uses x (assigned $R8)

   //  C4

}
```

2. [12 + 2 Bonus] **Garbage Collection + Escape Analysis** If the lifetime of a memory chunk (allocated on heap using `malloc`) does not extend the lifetime of the function in which the chunk is allocated, the chunk can be allocated on the stack of the function. Thus we can do the garbage collection of those objects for free (just by modifying the stack-pointer, at the end of the function)! Our goal is to classify each memory allocation into one of the two categories: global or local. In such a case, all the local chunks need not be heap allocated. Present an intra-procedural flow insensitive analysis to identify local memory allocation sites in C programs (note – this is a 'must' analysis). Assume that these are C programs with no-pointer arithmetic, and no arbitrary goto statements. [12].
Present a C language to C language translation to replace such heap accesses with appropriate stack accesses and apply your scheme on the following code. [Bonus 2]

```
int foo(){
    int a;
    struct X *ptr;
    ptr = (struct X*)malloc(sizeof(struct X)); // local
    bar(ptr);
    printf("%d", ptr->f1);
}
void bar(struct X *ptr){
   ptr->f1 = 2;
   return;
}
```

3. [12] **Points-to/Alias Analysis**:

Extend the subset based flow-insensitive, context-insensitive alias/points-to analysis discussed in the class to handle Java programs with exceptions. Mainly focus on the alias/points-to information of the exception objects and arguments of the catch blocks only, by concentrating mainly on the try, throw, catch, and function call statements. Given a statement of the form `throw x`, assume that $\rho(\texttt{x})$ returns the set of object that `x` may point to.

Java exceptions primer (target subset for the exam):

`throw y`	throws an object pointed-to by `y`. Control gets transferred to the appropriate `catch` block

`try {`
`  S1`
`} catch (X x) {`
`  S2`
`}`
	Say an exception object (say `O1`) is thrown in `S1` (could be from a function inside S1), and the run-time type of the exception object be `Y`. If `X` = `Y`, then the exception object `O1` will be passed to `x`, and the control will get transferred to this `catch` block; in this case, we will say that `x` points to `O1`. Else, the exception object will be again thrown by the `try` block, to be caught by another `catch` block.

`void foo(){`
`  ...`
`  throw x;`
`  ...`
`}`
	If the thrown exception object is not caught in `foo`, then the same exception object is re-thrown by the caller of `foo`. Thus an exception thrown in a function, may be caught either in the same function, or one of its parents in the call-graph.

4. [12 + 2 Bonus] **Loop optimizations** For each of the following loop optimizations, briefly explain

1. about the transformation,
2. feasibility: state the conditions under which the transformation is semantically correct,
3. profitability: state when the transformation is profitable.

Optimizations:

1. Loop-tiling [3],
2. Loop-distribution [3],
3. Strip-mining [3],
4. Loop-peeling [3].

Give an example where loop distribution is feasible, but loop-tiling is not [2 marks].

5 [12] **Register Allocation** For the following code, compute the liveness (enough to show the interference graph) [4 marks], and then use the Iterared-Register-Coalescing algorithm to do register allocation [6 marks]; show the interference graph after each step. Show the generated code, after eliminating redundant moves [2]. Use the following condition to coalesce: two nodes can be coalesced only if the coalesced node has $< K$ number of significant degree neighbors, where K is the number of available colors. State any criteria you make regarding spilling.

```
// Assume available registers: R1, R2, and R3. K = 3.
// Assume: R1 and R2 are live in.
a = R1;
b = R2;
c = a;
d = c + b;
f = d;
e = 0;
while (c < 1000){
  c = e;
  e = c + f;
}
R1 = c;
// R1 is live out.
```