Final Exam
CS6013

Maximum marks = 60, Time: 3hrs

20-Nov-2013

Read all the instructions and questions carefully. You can make any reasonably assumptions that you think are necessary; but state them clearly. There are total five questions totaling 60 marks. Each five marks will approximately take 15 minutes. For questions that have sub-parts, the division for the sub-parts are 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. [15] **Bitwidth aware variable packing**
a) Briefly describe combined packing and coalescing algorithm [7.5].
b) Intra-procedural bit sensitive analysis suffers from a serious drawback: all the arguments are assumed to use all the bits as per the declaration. Give a brief sketch to do inter-procedural bit sensitive analysis algorithm. Assume that the globals are live throughout the program [7.5].

2. [10] **Common Sub-expression Elimination (CSE)**
CSE identifies expressions that evaluate to the same value, and replace one of the expressions with the variable that holds the desired value. Common sub-expressions can be identified based on the computation of *available expressions*. For example, an expression $a + b$ is available at a program point $L1$ if:

- every path from the ENTRY to $L1$ computes $a + b$ before reaching $L1$.
- a and b are not redefined between the last computation of $a + b$ and $L1$.

If an expression e is "available" in a temporary (say t_1), at a program point $L1$ and the statement at $L1$ uses e, we can replace e by t_1.

Write an iterative algorithm to do intra-procedural CSE. Assume that the input is in three-address-code form and the control flow graph is already built. Test your algorithm on the following code.

```
d = b * c;   c1 = a + d;
x = b * c;   e = a + x; // can be replaced with x = d; e = c1;
t1 = e + f
if (a < c) {
   t2 = c + d;
   e = e + 1;
} else
   t3 = c + d;
t4 = e + f;  t5 = c + d; // c + d is available. How to replace?
```

3. [15] **Dependence analysis**
Define a canonical loop nest [1.5] and answer the below mentioned queries for the given sample code:

```
for (i = 1; i <= n; i+=2) do
  for (j = n; j >= 1; --j) do
    for (k = 3; k <= 2*n+1; ++k) do
      S1: A[i,j,k] = A[i-1,j-1,k-1] + A[i-1,j,k]
      S2: B[i,j-1,k] = A[i,j-1,k-1] * 2.0
      S3: A[i,j,k+1] = B[i,j,k] + 1.0
    endfor
  endfor
endfor
```

1. Transform the code to consist of only canonical loop nests. [1.5]
2. Draw the iteration space for the loop nest. [1.5]
3. Draw the execution order relationships between the three labeled statements. [1.5]
4. Write the dependence relations (flow, anti and output) between the three labeled statements. [1.5]
5. Restate the dependence relations in terms of distance vectors, direction vectors and dependence vectors. [4.5]
6. For the different references to A and B, use the GCD test to check if/when there exists any same iteration or different iteration dependence. [3]

4. [10] **Garbage Collection**:
Many objects allocated within a function can be freed at the end of the function automatically. One simple way to do it would be to allocate those objects on the stack (instead of heap), and when the function returns this space is automatically reclaimed. State the (sufficient) conditions under which an object can be safely allocated on the stack (instead of heap). Write an intra-procedural compiler transformation, that replaces some of the heap allocation statements to use stack. That is, replace `new A()` with `StackAllocate A()`.

```
A foo(){
 A a = new A(); // can be allocated on stack.
 a.f = 3;
 A b = new A(); // has to be allocated on heap.
 b.f = a.f;

 return b;
}
```

5. [10] Answer one of the following questions:

   a. Present a scheme to do parallel, flow sensitive context sensitive alias analysis for Java.

   b. Like Null pointer checks before a dereference, Java runtime does array-out-of-bounds checks before any array element is accessed. Present a scheme to eliminate array out of bounds exception checks in Java. For example in the following code, we need array-out-of-bounds check only at `L1`.

```
a = new int[n];
for (int i=0;i <n;++i){
  a[i] = i; // checks not required.
  j = foo(i);
}
/* L1: */ print (a[j]); // array bounds check required.
```