CS 701

Midterm Exam

Tuesday, October 30, 2001

9:00 AM— 11:00 AM

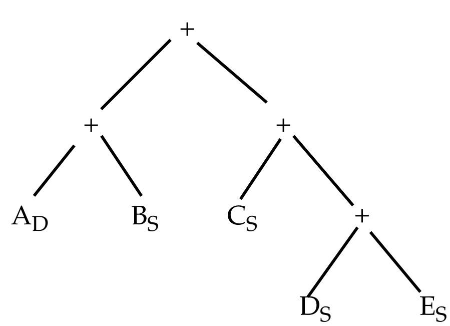
1263 CSST

Instructions

Answer question #1 and any three others. (If you answer more, only the first four will count.) Point values are as indicated. Please try to make your answers neat and coherent. Remember, if we can't read it, it's wrong. Partial credit will be given, so try to put something down for each question (a blank answer always gets 0 points!).

1. (1 point)
   Who manufactures the processor for computers labelled with "Intel inside"?

2. (33 points)
   The Sethi-Ullman algorithm, as presented in class, assumes that all operands and results are one word long. Many expressions in programs involve single and double word values. Single word values fit within a single register; double word values are stored in two paired registers. Assume that you are given an expression tree whose operands are all distinct variables. Each variable is marked as single or double length. All operators produce a single word result if their operands are both single length; otherwise a double length result is produced.
   What changes must be made to the Sethi-Ullman register needs rules and to the Sethi-Ullman code generator to accommodate this extension? (For simplicity, assume *any* pair of registers may be used to hold a double length value.)
   Illustrate your extensions using the following expression tree (a D or S subscript denotes a double or single word operand):



3. (a) (13 points)
   What changes to the Chaitin/Briggs graph-coloring register allocator were made by George & Appel to optimize register to register moves?

   (b) (5 points)
   How does the George/Appel register allocator avoid the dangers of "reckless coalescing" that Chaitin's allocator had?

   (c) (5 points)
   How does the George/Appel register allocator improve upon the "conservative coalescing" approach of Briggs?

   (d) (10 points)
   Does the George/Appel scheme ever miss a chance to give two move-related nodes the same color? If not, explain why. If so, give a simple example and suggest how their approach might be improved upon.

4. (a) (11 points)
   When a compiler must access an integer literal too large to be incorporated as an immediate operand, it must load the value into a register prior to its use. If it happens that from the program's flow of control it must have previously loaded and used the same literal value, it may be possible for multiple uses to share the same register. How can we determine if, at a use of a literal L, whether L has necessarily already been loaded and used?

   (b) (11 points)
   Assume we know that a particular use of L necessarily follows some previous use of L. How can we ensure that the current and previous uses of L share the same register? How can allocation of a register to L be integrated into the general problem of allocating registers to register candidates?

   (c) (11 points)
   It may happen that the only use of L is within a loop body. The analyses of parts (a) and (b) don't help here, since there are no earlier loads of L to reuse. An alternative is to add a load of L outside the loop, thereby avoiding repeated loads of L in the loop body. Suggest how to decide where to place the initial load of L. What are the factors that control your choice of placement?

5. Consider the following C function:

```
void f (int a[]) {
   for (i=2;i<1000;i++)
      a[i]=a[i-2]*2;
}
```

   Assume the function's loop body is translated into the following SPARC code:

```
L: ld      [%o0-8], %g2
   addcc   %g3, -1, %g3
   sll     %g2, 1, %g4
   st      %g4, [%o0]
   bpos    L
   add     %o0, 4, %o0
```

   (a) (11 points)
   Assuming there is no limit to the number of independent instructions that can be issued and executed simultaneously, schedule the loop body to take as few cycles as possible. Do not unroll or software pipeline the loop; just schedule its body as a basic block. How many cycles are needed?

(b) (11 points)

This loop body contains a loop-carried dependence, since `a[i]` depends on `a[i−2]`. In this case the *dependence distance* is two, since `a[i]` depends on the value of `a[i]` computed two iterations earlier. How does the dependence distance influence our estimate of the best possible initiation interval based on loop-carried dependences? What is the limit on the initiation interval for the above loop based on its loop carried dependence? Is a software pipelined loop based on this estimate of the initiation interval feasible? Why or why not?

(c) (11 points)

An alternative to software pipelining the above loop is to expand it, using different result registers for each copy of the loop body. The expanded body can then be scheduled using standard techniques. Assume the above loop body is expanded into two copies, using different result registers for the two loads and the two shifts. Show the loop body that results after scheduling. How many cycles per iteration are required? How close is this to the limiting initiation interval you computed in part (b)?