CS 701

Midterm Exam

Tuesday, October 31, 2000

11:00 AM— 1:00 PM

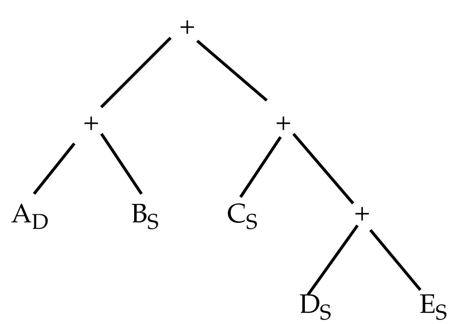
1263 CSST

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)
   In a 64 bit architecture (like the Intel IA-64), how many bits wide are addresses and general purpose registers?

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):

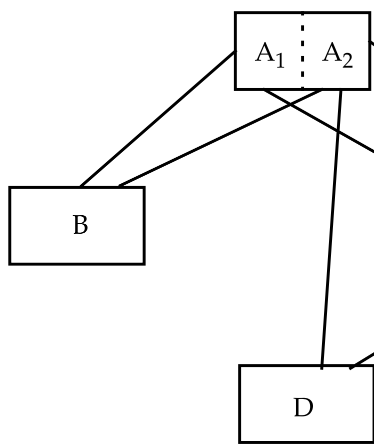


3. (33 points)

A weakness of Chaitin-style register allocation is that it takes an "all or nothing" approach to register allocation. That is, a live range either is assigned a register or it is spilled in its entirety. Other techniques, like that of Larus and Hilfinger, allow a live range to be split if necessary.

Assume we wish to extend Chaitin's allocator (as presented in class) to allow a limited form of live range splitting. Selected live ranges are marked as "splittable" into two halves. The interferences between halves is shown in the interference graph, and the spill cost of each half is known (the spill cost for a whole node is the sum of the costs for its two halves).

What changes are needed in the Chaitin-style register allocator to accommodate this limited form of live range splitting? Illustrate your modified allocator using the following interference graph, where A and C are splittable, and the spill costs are A_1=A_2=C_1=C_2=50, D=75, B=200. Assume 2 registers are available for allocation.



4. (a) (18 points)

Assume we are doing a save-free interprocedural register allocation for a program whose call graph forms a *tree*. We have *N* registers to allocate among all procedures. There are no saves and restores across calls, so a caller and callee can never use the same register. You are given *cost(i,r)*, the cost of executing all calls of procedure p_i with exactly *r* registers. This cost function is monotonically non-increasing in *r*. That is $cost(i,r) \geq cost(i,r+1)$.

Give a series of equations that allow us to solve for a least-cost save-free interprocedural register allocation.

(b) (15 points)

Assume now we have a very limited form of register windows. There are two windows of *M* registers each. Initially, at the start of the main program's execution, the first window is active. At the very beginning of a procedure's execution, a `save` may be done, which activates the other register window (a `restore` is done at the very end of the procedure's execution).

Since there are only two windows, a `save` may only be done once in any call chain. Procedures above a routine that does a `save` use one set of *M* registers, while a routine that does a `save` (and its descendents) use the other set of *M* registers.

Assume that `save` and `restore` instructions cost nothing and that there are no restrictions on how registers in a window may be used (i.e., no registers are reserved for special purposes). Extend your solution to (a) to determine which procedures in a program should do `saves` to obtain an optimal interprocedural register allocation.

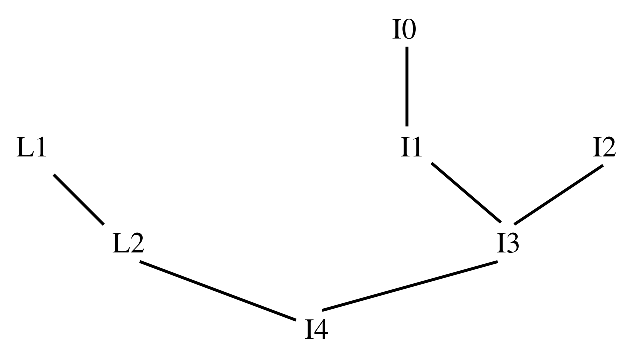
5. This question involves the DLS expression tree code generator and scheduler. Recall that DLS is the extension of the Sethi-Ullman algorithm that aims to avoid load stalls as well as to minimize register use

   (a) (7 points)
   The DLS algorithm noted that the register estimate computed by the Sethi-Ullman algorithm is insufficient to avoid loads stalls, but that using one additional register is sufficient to avoid all stalls for loads with a delay of 1 (excluding trivial expressions which always must stall).
   Explain carefully why this is the case (that is, why Sethi-Ullman's estimate is insufficient, but one extra register always suffices).

   (b) (6 points)
   The DLS algorithm works in the framework of expression tress, where each operand is a variable that is used exactly once. Assume we have an expression DAG where operands are variables that can be used more than once. Is the minimum register allocation always insufficient to avoid load stalls, or is a stall-free schedule sometimes possible?

   (c) (20 points)
   Assume you are given a code sequence that is generated from an expression DAG. Is uses exactly N registers, but alas, it has one or more load stalls. (Ignore all other kinds of pipeline stalls).
   You are given an extra register, not used within the code sequence. Propose an algorithm that transforms the code sequence into an equivalent stall-free sequence (that uses the original N registers plus the additional register). Does you algorithm always eliminate loads stalls? Why?

6. (33 points)
   The Eggers balanced scheduling algorithm is designed to handle the unpredictable latencies of non-blocking load instructions. It treats all loads equally, scheduling on the basis of available ILP. If it happens that we know that some loads are more likely to miss than others (using profiling or source level information), the algorithm is unable to utilize this information.
   For example, consider the following dependency dag. Assume that load L1 has an 80% probability of hitting in the primary cache, while load L2 has a 90% probability of hitting. Assume further that hits in the primary cache have a latency of 1 cycle, while a miss to the secondary cache has a latency of 10 cycles (deeper misses are ignored).



   The Eggers algorithm will treat L1 and L2 symmetrically, while given our additional information, we'd probably like to give L1 preference over L2 in scheduling.
   Suggest an extension to Eggers' balanced scheduling algorithm that utilizes hit/miss estimates in placing loads. Illustrate your extension on the above example.