CS 701 Midterm Exam

Monday, March 25, 1996

11:00 AM - Noon CST

Instructions

Answer question #1 and *any three* others. (If you answer all six, 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!).

Question #1 (1 point)

The Goodman-Hsu code scheduling algorithm was invented by:

(a) Mr. Ed
(b) Ty Cobb
(c) Sir Walter Raleigh
(d) James Goodman and Wei Hsu
(e) Hannibal Hamlin

Question #2 (33 points)

Define a Sethi-Ullman style code generator (ignoring spills) for expression trees for a architecture that has the following instruction forms:

Instruction		Meaning
LD	[Mem],Reg	Load contents of memory location Men into register Reg
MV	Lit,Reg	Move literal value Lit into register Reg
ST	Reg,[Mem]	Store contents of register Reg into memory location Mem
OP	R1,R2,R3	Compute (R1 OP R2) into R3
OP	R1,Lit,R2	Compute (R1 OP Lit) into R2

Assume that all values and operations are integer, all operators are non-commutative and that all literal values fit within a single instruction.

Illustrate your code generator on the following expression:

```
((A+1)*(2+B))-((C+D)*(E+F))
```

Question #3 (33 points)

Assume we have built an interference graph for the live ranges of a procedure. Define Chaitin's algorithm for coloring the interference graph.

When nodes are being popped from the stack at the end of the algorithm, the standard rule is to assign a node *any* color not already assigned to any of the node's neighbors. Could Chaitin's algorithm be improved upon if we took more care in choosing a color? That is, could it be the case that choosing one unused color leads to a better overall coloring than the choice of another unused color? If so, give a *simple* example. If all unused colors lead to equivalent coloring, explain why.

Question #4 (33 points)

Assume we have a Sparc-like architecture in which loads are (as usual) delayed by one cycle, and multiplies and divides are delayed by two cycles (that is two instructions must separate a multiply or divide from the first use of its result). Adds and subtracts are not delayed.

What changes, if any, must be made to the Gibbons-Muchnick code scheduler to allow it to handle this new architecture?

Illustrate your answer by scheduling the following instruction sequence, generated for the expression `((c+d)+(e*f))+(a+b)`:

```
ld    [c],%r1
ld    [d],%r2
add   %r1,%r2,%r1
ld    [e],%r3
ld    [f],%r4
smul  %r3,%r4,%r3
add   %r1,%r3,%r1
ld    [a],%r2
ld    [b],%r4
add   %r2,%r4,%r2
add   %r1,%r2.%r1
```

As generated, the instruction sequence has 5 wasted (stalled) cycles. How many remain after code scheduling?

Question #5 (33 points)

Explain how to label the nodes of a dag to estimate the number of registers needed to evaluate the dag. Illustrate your labeling technique on the dag corresponding to the following expression:

`(a+a)+((a+a)+a)`

Ignoring assignments, how can this labeling drive a code generator? What Sparc code would be generated for the above expression?

Question #6 (33 points)

It is sometimes useful to allocate a register to hold the value of a frequently referenced literal (constant) value, especially if the literal is too large to be included as an immediate operand of an instruction.

One obvious way to allocate a literal, `L`, to a register is to create a local variable named `C` that is initialized to `L`. All references to `L` are replaced with references to `C`.

Because there is only one definition of `C`, this approach creates one large live range that can span most of the subprogram. This may create many interferences, making it difficult to allocate `C` a register.

An alternative is to move definitions to `C` "closer" to where they will be used. We can examine each use of `L`, classifying is a potential definition of `C` or a potential use of `C`. A use of `L` is considered a definition of `C` if it is possible that this is the first time in the program `L` is used (`C` can be loaded with `L` at this point). A use of `L` is considered a use of `C` if it *must be* the case that `L` has been previously used in the program (`L`'s value must already be loaded into `C` at this point). Now we replace all uses of `L` with definitions or uses of `C` and do standard live range analysis and graph coloring. Live ranges of `C` that are assigned a register will access all uses of `C` directly from a register. Live ranges not allocated a register will load `C`'s value (`L`) whenever `C` is referenced (just as was previously done for `L` anyway).

Explain how to do a data flow analysis to determine which uses of `L` are to be considered definitions of `C` and which are to be considered uses of `C`.