

16.10 Exercises

- 16.1** In Section ©16.2 we suggested replacing the instruction $r1 := r2 / 2$ with the instruction $r1 := r2 >> 1$, and noted that the replacement may not be correct for negative numbers. Explain the problem. You will want to learn the difference between *logical* and *arithmetic* shift operations (see almost any assembly language manual). You will also want to consider the issue of rounding.
- 16.2** Prove that the division operation in the loop of the combinations subroutine (Example ©16.10) always produces a remainder of zero. Explain the need for the parentheses around the numerator.
- 16.3** Certain code improvements can sometimes be performed by the programmer, in the source-language program. Examples include introducing additional variables to hold common subexpressions (so that they need not be recomputed), moving invariant computations out of loops, and applying strength reduction to induction variables or to multiplications by powers of two. Describe several optimizations that cannot reasonably be performed by the programmer, and explain why some that could be performed by the programmer might best be left to the compiler.
- 16.4** In Section 6.5.1 (page 257) we suggested that the loop

```
// before
for (i = low; i <= high; i++) {
    // during
}
// after
```

be translated as

```
-- before
i := low
goto test
top:
    -- during
    i += 1
test:
    if i ≤ high goto top
    -- after
```

And indeed this is the translation we have used for the combinations subroutine. The following is an alternative translation:

```
-- before
i := low
if i > high goto bottom
```

```

top:
  -- during
  i += 1
  if i ≤ high goto top
bottom:
  -- after

```

Explain why this translation might be preferable to the one we used. (Hints: Consider the number of branches, the migration of loop invariants, and opportunities to fill delay slots.)

- 16.5 Beginning with the translation of the previous exercise, reapply the code improvements discussed in this chapter to the `combinations` subroutine.
- 16.6 Give an example in which the numbered heuristics listed on page ©354 do not lead to an optimal code schedule.
- 16.7 Show that forward data flow analysis can be used to verify that a variable is assigned a value on every possible control path leading to a use of that variable (this is the notion of *definite assignment*, described in Section 6.1.3).
- 16.8 In the sidebar on page 774, we noted two additional properties (other than definite assignment) that a Java Virtual Machine must verify in order to protect itself from potentially erroneous byte code. On every possible path to a given statement S , (a) every variable read in S must have the same type (which must of course be consistent with operations in S), and (b) the operand stack must have the same current depth, and must not overflow or underflow in S . Describe how data flow analysis can be used to verify these properties.
- 16.9 Show that *very busy* expressions (those that are guaranteed to be calculated on every future code path) can be detected via backward, all-paths data flow analysis. Suggest a space-saving code improvement for such expressions.
- 16.10 Explain how to gather information during local value numbering that will allow us to identify the sets of variables and registers that contributed to the value of each virtual register. (If the value of register vi depends on the value of register vj or of variable x , then during available expression analysis we say that $vi \in Kill_B$ if B contains an assignment to vj or x and does not contain a subsequent assignment to vi .)
- 16.11 Show how to strength-reduce the expression i^2 within a loop, where i is the loop index variable. You may assume that the loop step size is one.
- 16.12 Division is often much more expensive than addition and subtraction. Show how to replace expressions of the form $i \div c$ on the inside of a `for` loop with additions and/or subtractions, where i is the loop index variable and c is an integer constant. You may assume that the loop step size is one.

16.13 Consider the following high-level pseudocode.

```

read(n)
for i in 1 .. 100
    B[i] := n × i
    if n > 0
        A[i] := B[i]

```

The condition $n > 0$ is loop invariant. Can we move it out of the loop? If so, explain how. If not, explain why.

- 16.14** Should live variable analysis be performed before or after loop invariant elimination (or should it be done twice, before *and* after)? Justify your answer.
- 16.15** Starting with the naive gcd code of Figure 1.6 (page 34), show the result of local redundancy elimination (via value numbering) and instruction scheduling.
- 16.16** Continuing the previous exercise, draw the program's control flow graph and show the result of global value numbering. Next, use data flow analysis to drive any appropriate global optimizations. Then draw and color the register conflict graph in order to perform global register allocation. Finally, perform a final pass of instruction scheduling. How does your code compare to the version in Example 1.2?
- 16.17** In Section ©16.6 (page ©352) we noted that hardware register renaming can often hide anti- and output dependences. Will it help in Figure ©16.12? Explain.
- 16.18** Consider the following code:

```

v2 := *v1
v1 := v1 + 20
v3 := *v1
—
v4 := v2 + v3

```

Show how to shorten the time required for this code by moving the update of $v1$ forward into the delay slot of the second load. (Assume that $v1$ is still live at the end.) Describe the conditions that must hold for this type of transformation to be applied, and the alterations that must be made to individual instructions to maintain correctness.

16.19 Consider the following code:

```

v5 := v2 × v36
—
—
—
—
v6 := v5 + v1
v1 := v1 + 20

```

Show how to shorten the time required for this code by moving the update of `v1` backward into a delay slot of the multiply. Describe the conditions that must hold for this type of transformation to be applied, and the alterations that must be made to individual instructions to maintain correctness.

- 16.20** In the spirit of the previous two exercises, show how to shorten the main loop of the `combinations` subroutine (prior to unrolling or pipelining) by moving the updates of `v26` and `v34` backward into delay slots. What percentage impact does this change make in the performance of the loop?
- 16.21** Using the code in Figures ©16.11 and ©16.13 as a guide, unroll the loop of the `combinations` subroutine three times. Construct a dependence DAG for the new Block 2. Finally, schedule the block. How many cycles does your code consume per iteration of the original (unrolled) loop? How does it compare to the software pipelined version of the loop (Figure ©16.15)?
- 16.22** Write a version of the `combinations` subroutine whose loop is both unrolled *and* software pipelined. In other words, build the loop body from the instructions between the left-most and right-most vertical bars of Figure ©16.14, rather than from the instructions between adjacent bars. You should update the array pointers only once per iteration. How many cycles does your code consume per iteration of the original loop? How messy is the code to “prime” and “flush” the pipeline, and to check for sufficient numbers of iterations?
- 16.23** Consider the following code for matrix multiplication:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        C[i][j] = 0;
    }
}
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Describe the access patterns for matrices A, B, and C. If the matrices are large, how many times will each cache line be fetched from memory? Tile the inner two loops. Describe the effect on the number of cache misses.

- 16.24** Consider the following simple instance of Gaussian elimination:

```

for (i = 0; i < n-1; i++) {
    for (j = i+1; j < n; j++) {
        for (k = n-1; k >= i; k--) {
            A[j][k] -= A[i][k] * A[j][i] / A[i][i];
        }
    }
}

```

(Gaussian elimination serves to triangularize a matrix. It is a key step in the solution of systems of linear equations.) What are the loop invariants in this code? What are the loop-carried dependences? Discuss how to optimize the code. Be sure to consider locality-improving loop transformations.

- 16.25** Modify the tiled matrix transpose of Example ©16.32 to eliminate the min calculations in the bounds of the inner loops. Perform the same modification on your answer to Exercise ©16.23.

16.11 Explorations

- 16.26** Investigate the back-end structure of your favorite compiler. What levels of optimization are available? What techniques are employed at each level? What is the default level? Does the compiler generate assembly language or object code?

Experiment with optimization in several program fragments. Instruct the compiler to generate assembly language, or use a disassembler or debugger to examine the generated object code. Evaluate the quality of this code at various levels of optimization.

If your compiler employs a separate assembler, compare the assembler input to its disassembled output. What optimizations, if any, are performed by the assembler?

- 16.27** As a general rule, a compiler can apply a program transformation only if it preserves the correctness of the code. In some circumstances, however, the correctness of a transformation may depend on information that will not be known until run time. In this case, a compiler may generate two (or more) versions of some body of code, together with a run-time check that chooses which version to use, or customizes a general, parameterized version.

Learn about the “inspector-executor” compilation paradigm pioneered by Saltz et al. [SMC91]. How general is this technique? Under what circumstances can the performance benefits be expected to outweigh the cost of the run-time check and the potential increase in code size?

- 16.28** A recent trend is the use of static compiler analysis to check for patterns of information flow that are likely (though not certain) to constitute