Compiler Construction - Assignment 5

LIACS, Leiden University

Fall 2008

Introduction

In this assignment, we will finish the subset Pascal compiler that has been studied in the previous assignments. This involves optimizing the generated code by trying to decrease the amount of instructions that have to be executed, while preserving the original functionality of the input program. For this assignment, you will build further upon the code generator that you have written for assignment 4.

Framework

In your CVS repository, a module named "`assignment5`" is available. Before you start working on this assignment, first add the necessary files of your `assignment4` directory to the `assignment5` module. The object files are already in the `assignment5` module. You have to add all other files (i.e., the Makefile, header files and your own files of assignment 4) yourself.

Example Optimizations

In this section, some example optimizations are listed. Note that these are just examples, you do not have to implement all of them in order to get a passing grade for this assignment. However, it should be noted that your grade is partly determined by the amount of implemented optimizations. You are allowed to implement other optimizations that are not mentioned as well. Consult [1] for a more elaborate discussion of code optimizations. In principle, every optimization is allowed, as long as the resulting assembly is still correct with regard to the input program.

Constant Optimizations

As a first starting point, you can reuse some of the optimization code you have written for assignment 3. This involved constant folding, zero-product elimination and algebraic simplification using identity elements. You can extend your code to perform other optimizations and/or simplifications as well, as long as you document this properly.

Constant Propagation

Consider the following example:

```
a := 2;
x := a * 4;
y := x + 2;
```

During this code fragment, the variable a is constant. Since there is no basic block boundary inbetween, we can safely replace this variable by 2 in the assignment to x. If we now apply a pass of constant folding, x also becomes a constant (x := 2 * 4 = 8). This procedure repeated for the variable x yields the following result:

```
  a := 2;
  x := 8;
  y := 10;
```

The variables a and x might turn out to be "dead" by now, so the first two assignment statements can be removed entirely. See section "Dead/Unreachable Code Elimination" for more about this.

Register Allocation

Instead of loading and storing a variable each time you need to read respectively write it, a code generator can decide to bind the variable to a register. When enough free registers are available, variables can reside in a register for their entire lifetime, such that they do not have to be stored in memory at all. Register allocation can be applied on the level of basic blocks, subprograms or even entire programs.

Control Flow Optimization

An example of control flow optimization is to remove "goto-chains". For example, consider the following code:

```
    j     _L0
    ...

_L1:
    addi  $3, $3, 1
    ...

_L0:
    j     _L1
```

Instead of jumping to _L0, the first operation could also jump immediately to _L1.

Common Subexpression Elimination

Consider the following example:

```
  x := a * b + c * d + 1;
  y := a * b + c * d - 1;
```

The subexpression a * b + c * d yields the same result for both statements. Therefore, it is sufficient to compute it only once:

```
  t1 := a * b + c * d;
  x := t1 + 1;
  y := t1 - 1;
```

Common subexpression elimination is typically performed on the intermediate representation. This is done by maintaining a set of "available expressions", thereby keeping track of the expressions that are valid while iterating over the statements. Note that basic block boundaries like labels and branch instructions often invalidate the entire set of available expressions.

Dead/Unreachable Code Elimination

This optimization serves two goals. First, it could remove code that has become unreachable because of earlier optimizations. Second, it could remove code that is also not reachable in the original input program, as illustrated in the following example:

```
x := 0;
if (x = 1) then
  writeinteger(1)
else
  writeinteger(2)
```

Since the variable `x` will always be 0 when the if-condition is evaluated, the then-part will never be executed. Thus, the code could be rewritten as:

```
x := 0;
writeinteger(2)
```

Another aspect of dead code elimination is to remove "dead" variables, i.e., variables that are written, but never read. Optimizations like constant propagation might introduce dead variables. Removing dead variables reduces the amount of store instructions.

Redundant Code Removal

Consider the following example:

```
  sw    $3, y
  lw    $3, y
```

The `sw` instruction will not modify register 3, so it still contains the value of `y` afterwards. This means the `lw` instruction is not needed at all and hence it can be removed safely.

Code Motion

This technique tries to move operations out of a loop, resulting in faster execution of an iteration. This is allowed as long as the moved operation is loop-invariant. Consider the following example:

```
i := 100;
b := 2;
while (i > 0) do
begin
  a := 2 * b;
  i := i - a
end
```

Both operands of the right hand side of the assignment to `a` are loop-invariant (2 is, since constants are loop-invariant by definition, and `b` is, since it is defined outside the loop). Therefore, `a` itself is also loop-invariant and we can put the assignment right before the loop.

Like other loop optimizations, it should be noted that this is a rather advanced and probably difficult-to-implement optimization. You need to be really careful not to change the original functionality of the input program. You are advised to start implementing simpler optimizations first before you decide to give this one a try.

Assignment

What you have to deliver: a compiler which can parse a given code file and translate this into optimized and functionally equivalent MIPS assembly that can be simulated on the (X)SPIM simulator. A binary of the XSPIM simulator is available in your repository. The user manual of the (X)SPIM simulator is available on Blackboard, under "Course Documents", "Practicum Resources".

You should also provide clear documentation about the implemented optimizations. This documentation should be placed in your CVS repository in the form of a file called `DOCUMENTATION.TXT`. For each implemented optimization, you have to explain the concept, preferably illustrated with an example, and indicate (informally) how correctness of the resulting code can still be guaranteed.

Submission & Grading

Submit your work using CVS. Do not send anything by email. Make sure the latest version of your work has been committed to the CVS repository. Then, tag that version using the command:

```
cvs tag DEADLINE5
```

This tag command has to be issued *before* December 4th, 2008 at 23:59.

For this assignment, 0-10 points can be obtained, which account for 25% of your final grade. If you do not submit your work in time, it will not be graded. The results, once available, can be found on BlackBoard.

All submissions will be tested with a rather complex and not published input program, containing a lot of "optimizable" code, i.e., common subexpressions, redundant statements etc. The team whose compiler manages to produce the fastest code for this test (in terms of number of instructions executed), will receive 10 points for this assignment. Of course the output needs to be functionally equivalent to the input program, and documentation should be decent too. The grades of the other teams will depend on the amount and complexity of the implemented optimizations, the quality of the resulting code and documentation. Documentation should be submitted in English only.

Assistance will be given by Michiel Helvensteijn in room 306/308. A schedule for this can be found on BlackBoard. Additionally, you can go to Sven van Haastregt in room 1.22.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley, 1986.