Programming Assignment 5 - Assembly Code Generation

Collaborators

- Michael Maitland, mtm68
- Scott Bass, sb2383
- Michael Tobin, mat292

Running the Program

Main class: `mtm68.Main.java`

Run `xic-build` (this requires having Maven installed). Then `xic [options...] <source-files>` becomes available.

Summary

In this assignment we implemented tiling and register allocation. In accomplishing this, we also designed a set of abstract assembly instructions as well as a means to convert the abstract assembly to real assembly. In accomplishing the tiling, we designed a pattern matching module which makes it very easy to specify tiles we want to match against and programatically match these patterns so they do not need to be checked explicitly for each pattern. By taking this approach we able to split the project up into a few distinct parts: the tiling algorithm, the pattern matching, and register allocation. This made it easy to keep us organized, work on independent parts of the project concurrently, and test separately. We also added full integration testing support so that we can have a set of tests already written when it is time to do optimizations.

Specification

The following are choices we made regarding specification: - For the -target command line option, if any option other than "linux" is provided, we terminite with exit code 0 and report the valid target option. This means that no compilation or writing to files is done if -target doesn't equal "linux".

Design and Implementation

Architecture

The key classes and packages we created or updated for this assignment are the following...

- mtm68.Main.java
  - Our Main functions very similarly to the previous assignment. We added new options to the command line as described in the spec. We also added intermediate code generation to the source file pipeline.

- edu.cornell.cs.cs410.ir.visit.Tiler
  - This is the IRVisitor whose purpose is to tile the IR code. At a high level, this visitor converts IR code to abstract assembly. This is done by the IRNodes specifying how they can be tiled based on the root node and any children they might have. The IRNode implements the tiling algorithm described in class in order to determine the lowest cost assembly sequence, and the PatternMatcher (described below) is responsible for determining if a node and its children match the patterns each node says it can be tiled as. This class also hosts the functionality that describes how to tile IRCallStmt because that IRNode is depends on the state of the Tiler.
- mtm68.assem.* and mtm68.assem.op
  - This package contains the abstract assembly instructions that we use to tile IRNodes. This set of abstract assembly also provides functionality to assist in converting itself into real assembly. By this, we mean it provides a way to replace abstract registers with real registers.
- mtm68.assem.operand.*
  - This package contains a set of classes that represent different types of assembly operands. AbstractReg and RealReg are the most notable classes in this package. The abstract assembly instructions contain instances of these two classes and register allocation converts these AbstractRegs into RealRegs during register allocation.
- mtm68.assem.visit.TrivialRegisterAllocator
  - This class performs trivial register allocation as described in class on the CompUnitAssem that represents the entire program. Its responsibility, for each function in the program, is to represent the abstract registers as locations on the stack, and shuttle between the stack and into real registers as needed.
- mtm68.assem.visit.AssemToFileBuilder
  - This class converts a list of assembly to a text file containing the textual representation of the assembly so it can be linked against and executed.
- mtm68.assem.pattern.Pattern
  - This interface is the base from which we do our tiling. Patterns provide a declarative to specify tiles. In our tiling algorithm, we use pattern to determine if a specific tile will work on a subtree of the IR tree. Patterns can match things such as: 32-bit constants, any subtree, temps whose names match a specified regex, commutative binary operations, etc. Convenient constructors for each pattern class are provided in the Patterns factory class to be used by the TileFactory to construct tiles.
- mtm68.assem.tile.Tile
  - This abstract class is used to specify a valid tile used in tiling. A tile is composed of three pieces:
    1. A pattern that's used to determine if the tile is valid on a given

subtree.
    2. A cost which is used to determine how expensive the tile is.
    3. A method for constructing the actual assembly code given the matched sub-expressions.

  - Tiles are created in the TileFactory and are returned in a list by each IRNode to be used by the Tiler to do optimal tiling.
- mtm68.assem.tile.TileFactory
  - This factory class is used to construct tiles. It uses the Patterns factory to construct declarative patterns and then produces tiles using the patterns and assigns each tile a cost.

Code Design

- For this assignment we wre able to use the IRVisitor and IRNodes to perform the tiling. This was great because we already had the required nodes, interfaces, abstract classes, etc to visit IRNodes. Additionally, we were able to reuse the same pattern that we did in the past where each node in the tree contained a field containing the data that was the result of the visit. Since we had done this in the past, we were able to get off the ground and get to work quite fast. It's also helpful for maintainability that we reused the same design ideas that we did in the past.
- The conversion from abstract assembly to real assembly via register allocation was accomplished through the well structured CompUnitAssem. This abstract assembly made it easy to get each function and convert them individually. Each function is represented as a SeqAssem, which contains a list of assembly. This allowed us to do register allocation on a list of assembly. Since each instruction could be converted individually, all we had to do was loop through each instruction in the function and do the trivial allocation. At the end of allocation, we finished with a list of real assembly, ready to be converted to a file later on.
- With respect to the actual conversion of abstract assembly to real assembly, we first tried to take an object oriented approach to replacing the registers by having each assembly instruction report out which abstract registers they used and then this class would report back to each instruction which registers to replace. We had trouble with this approach and opted for a more functional approach where each instruction would provide the function to set its registers and the allocator would set the registers as it wished. This second approach gave us much more flexibility and resulted in a more robust and simple solution.
- For tiling, we wanted to have a robust and extendable system. To achieve this, we opted for a declarative approach to tiling. Each IRNode reports a list of Tiles that can be used to tile it. Each tile includes a pattern, cost, and method for constructing the assembly. The Tiler then performs

tiling in a general way by trying all possible tiles for a given node, and picking the one that minimizes cost using memoization on subtrees that have already been tiled. The patterns turned into a mini-DSL, making the tile factory methods very readable. We believe this approach to tiling is elegant and less error-prone than a more imperative approach involving if statements, instanceof checks, and a lot of casting.

Programming

- The greatest challenge during this assignment was coming up with a set of abstract assembly and representation of operands. The tiling and the register allocation depended on it so we wanted to agree on this up front. Our first agreement stayed mostly the same, but the original register allocation worked off the concept of whether an instruction had one, two, or three operands. We ended up getting rid of this idea which caused us to go back to the drawing board and restart on register allocation. The good part was that the register allocator was designed so the changes only affected a single method.
- Another challenge we had was doing the tiling using our custom PatternMatcher. We had to wait until that was done to make most of our progress, but we believe it was worth it. Our PatternMatcher allows for us to very quickly add new tilings into our tiling system in a cohesive, organized structure.
- The following is the team coding/responsibility breakdown for this assignment...
  - **Tobin:**
    - AssemblyBuilder
    - Command Line Options
    - Fix errors from PA4
    - Basic Tiling + tests
    - Integration tests
  - **Maitland:**
    - IRFunctionDefn and IRCompUnit tiling
    - TrivialRegisterAlloctor + tests
    - Advanced Tiling + tests
    - Integration Testing support for linux
  - **Bass:**
    - Basic Tiling + tests
    - PatternMatcher + tests
    - TrivialRegisterAllocator
- We used our previous code for lexing, parsing, and typechecking. Fortunately, we have been on top of correcting our errors after each assignment so there were very few changes that needed to be made to this code.

Testing

In this assignment, testing was a little bit harder. It was useful for us to have test cases that printed out results for us to compare with IRCode, Abstract Assem, and Real Assem and we made changes based on visual examination. We also added integration tests which did allow us to make assertions. These integration tests will also be useful when we do optimizations because we will know if the optimizations broke anything. We did have more robust tests for tiling and pattern matching which helped us be confident in those parts.

Our integration tests stepped through the entire pipeline of compilation. First, we turned the xi program into IR code and tested the output of this IR code against the expected output using the IRSimulator. Then, we generated assembly from the IR code and automatically ran it using linkxi.sh. Setting up this architecture in JUnit allows us to very easily add more programs as test cases. It simply requires writing xi programs and determining their expected output.

When we ran into bugs in our integration tests, we would write a simpler program that isolated the issue. Then, we would generate the executable for this smaller program and step through the execution using gdb. Although being very painful, this was effective in helping us isolate a number of issues (such as improper stack pointer alignment and register value clobbering).

It should also be said that we utilized the xth test suite to make sure our output was compliant with the autograder's expectations.

Work plan

Before being able to split up, we needed to devise a structure within our code to represent assembly instructions. We designed a simple base set of assem objects that could cover all functionality. Then, we were able to implement basic tiling which involved creating a default, non-efficient case for all IRNodes. At this point, we were able to split up to handle trivial register allocation as well as our pattern matching system. Our work plan did result in some friction here and there. Occasionally, while designing the more complicated aspects of this assignment, we realized that earlier design decisions should've been made differently. This required us to go back and refactor certain structures a handful of times. We accept these inefficiencies as it may have taken even longer to try and consider all of these issues upfront before designing anything rather than dealing with an occasional redesign.

We did a great job of communicating frequently, working together to clarify things, and adapting to challenges we faced.

Known Problems

We currently are not aware of any issues with our compiler.

Comments

WORKING COMPILER WOOT WOOT!!!