CS 380C: Advanced Topics in Compilers
Assignment 6: Assembly Code Generation
Due: March 31st

March 3, 2016

Late submission policy: Submission can be at the most 2 days late. There will be a 10% penalty for each day after the due date (cumulative).

The goal of this assignment is to generate assembly code for x86-64 architecture from LLVM IR for a subset of the C language; the specifics are fairly open-ended.

1 Generating x86-64 assembly code from LLVM IR

The Intermediate Representation (IR) in a compiler represents the instruction set of an abstract machine. To execute it on a machine, the compiler should convert it to the instruction set of the assembly language supported by that machine's architecture. In order to do so, it maps a sequence of IR instructions to a sequence of assembly instructions. While doing this, it should also map the operands of the IR instructions to specific hardware locations - registers or memory.

As you know, LLVM IR is in SSA form with infinite virtual registers. Architectures like x86-64 have limited physical registers and these registers as well as memory are not restricted to a single-assignment (they can be written multiple times). Moreover, SSA form is a three-address code where the destination operand is distinct from the source operand(s), whereas in x86-64 assembly code, the destination operand is typically also a source operand (e.g., for a binary operation). These differences have to be dealt with during code generation. In this assignment, you will write a low-level compiler that will generate x86-64 assembly code from LLVM IR (the interface of your compiler will be similar to LLVM's `llc`). Your goal should be to minimize the execution time of your generated code (without altering the intended behavior).

1.1 x86-64 assembly language

To generate assembly code, you would need to understand the assembly language. You can find all the necessary details for the x86-64 architecture here:

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

x86-64 is a CISC architecture. A single instruction can execute several operations (like a load from memory, an arithmetic operation, a store to memory). An instruction can support different addressing modes for its operands, i.e., the operand for the instruction can be an immediate value, a

register, or a memory location. Some instructions can have implicit operands or can require some of its operands to be in specific registers. Consequently, there are many ways (sequences of instructions) to execute the same sequence of operations.

1.2 Instruction Selection

A one-to-one mapping of instructions from IR to assembly language is not efficient and such a mapping might not even exist. So, instruction selection is typically done for an expression tree - a sequence of instructions which computes an expression. To identify the optimal sequence of instructions for that expression tree, you can use the `Olive` tool.

`Olive` takes a cost-augmented tree grammar as input and generates code to pattern match the given expression tree to the minimal cost sequence of instructions (as defined by the grammar). The grammar specification includes the actions that should be run for each pattern and the generated code will run those actions for the minimal cost pattern. You can find the tool along with its source code and paper here. You can also read the iburg paper on which `Olive` is based.

You will write a cost-augmented tree grammar along with the actions to call your code generator for each pattern. `Olive` will generate the pattern matcher code for you. In your compiler, you will build expression trees from LLVM IR and provide it as input to the pattern matcher code, which would generate the assembly code. Your goal should be to reduce the execution time of the generated code.

Some points to consider for your design:

- How can expression trees be built from SSA form? The dataflow graph is implicit in the SSA form; how can this be split into expression trees? What should be the root of an expression tree? Can two expression trees have overlapping nodes (which implies that the overlapping nodes would be recomputed)? How do these choices affect register allocation?
- How can you reduce the execution time of the generated code, i.e., sequence of instructions - by generating fewer instructions, by generating instructions with fewer memory accesses, by generating simpler instructions, or something else (are they correlated or independent of each other)? How can cost be assigned to achieve that goal? e.g., is loading both operands into registers for an addition operation and then storing the result into memory better than loading one operand into register and directly operating on memory as the source/destination operand? How does this choice affect register allocation?

1.3 Register Allocation

Once the instructions have been selected to generate code, you need to assign registers to each variable being used in those instructions. If no register is available, then you would have to spill an existing variable in a register to memory.

You can use the linear scan register allocation algorithm to assign variables to registers and determine which variables to spill. This algorithm will not be covered in class; read the paper on it (and the paper on it for SSA). You can instead implement the register allocation algorithm covered in class that uses graph coloring. However, that is more complex to implement. So, we recommend implementing linear scan register allocation. Linear scan register allocation can take much lesser time and memory to generate assembly code than register allocation by graph coloring but it could

result in more spilling.

Some physical registers may be reserved for special purposes; all the other available registers should be used by default. Your register allocator should also accept a configuration parameter that limits the number of available registers (excluding special purpose registers). Your goal should be to minimize spilling variables from registers to memory.

Some points to consider for your design:

- How can you operate on variables spilled to memory - by using instructions that support memory loads, by having a dedicated register to load it into, or by introducing a temporary variable only for that operation that cannot be spilled?
- How can you reduce the number of a times a variable is spilled to memory (loaded from memory and stored to memory)? Can you minimally split the live range of a variable so that the variable can be kept in a register for each split?

2 Supporting features of the C language

The subset of the C language features that your code generator supports is decided by you. Your goal should be to support as many features as possible. Supporting a feature means that your code generator must be robust (no crashes or undefined behavior) and must generate precise (correct output) and efficient (faster execution time) assembly code for that feature. For example, here is a list of features you can support:

- scalar integers - local variables and arguments,
- pointers - local variables and arguments,
- single-dimensional arrays of integers - local variables and arguments,
- constants and global variables (integers),
- character strings/arrays,
- terminator instructions - unconditional and conditional branches, return with and withoout a value,
- loops - with break and continue,
- function calls - standard library functions and recursion,
- structures,
- floating-point datatypes.

We advise you to support some simple features end-to-end before experimenting with more complex features (though supporting complex features may require a change in your design). It would be a good idea to target generating code for simple/toy applications like searching and sorting.

3 Getting Started with LLVM

By now, you should be familiar with the LLVM compiler infrastructure. You can find all the necessary documentation here:

http://llvm.org/docs/

You can use the following LLVM manuals to learn more about the compiler:

Downloading and building LLVM	http://llvm.org/docs/GettingStarted.html
LLVM command line tools	http://llvm.org/docs/CommandGuide/
LLVM IR Reference Manual	http://llvm.org/docs/LangRef.html
LLVM Programmers Manual	http://llvm.org/docs/ProgrammersManual.html
Writing an LLVM Pass	http://llvm.org/docs/WritingAnLLVMPass.html
LLVM testing infrastructure	http://llvm.org/docs/TestingGuide.html

Read these manuals very selectively, or you will spend far too much time on them!

(*You should have done this already:*) Download and build the source code for the latest release version 3.7.1 (do not get the source code from the SVN head); configure LLVM for a debug build and turn assertions on (Debug+Asserts build).

Stampede on TACC: You can use any machine for development but you should ensure that your code can be built and run on Stampede (since we will be testing your submission on it). Use the login node on Stampede only for development - do not run or debug any executable on it. Run and debug your applications using the job scheduler. Read the user guide to learn how to submit jobs: http://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide#running

4 Implementation Guidelines

Please follow these guidelines precisely because the grading scripts will be based on it.

1. Create a new directory `llc-olive` in the `tools/` folder of the LLVM source tree - `$SRC_ROOT`. All your source files, including your build script (`CMakeLists.txt`), should be within this directory. This is the source directory you will submit.

2. Create a `cmake` build script to build `llc-olive` in the `bin/` folder of the LLVM object tree - `$OBJ_ROOT`. You can use `$SRC_ROOT/tools/llc/CMakeLists.txt` as a template.

3. Include this line in `$SRC_ROOT/tools/CMakeLists.txt`:

   ```
   add_llvm_tool_subdirectory(llc-olive)
   ```

4. Run your compiler using this command:

   ```
   llc-olive [--num_regs=$NUM] $INPUT -o $OUTPUT 2>$LOG
   ```

   $INPUT will be a LLVM bitcode file (`.bc`). $OUTPUT should be the human-readable assembly file. The standard error output $LOG can be used for logging information.
   Your compiler should also accept an optional parameter `num_regs` to specify the number of available registers for your register allocator (the default should be the maximum available registers in the architecture).

5. Assemble the generated code and execute it:

```
gcc $OUTPUT -o $EXEC
./$EXEC
```

Try using the C++ Standard Template Library, if you haven't already - it can enable you to use effective data structures quickly (e.g., sets, maps, lists, and vectors). LLVM also offers many data structures in its library. See the LLVM Programmer's Manual for choosing an appropriate data structure. If you find you are writing code for doing something basic within LLVM, it is quite likely that the code for this exists already - use it.

You are not allowed to copy source code from anywhere, including other LLVM source files. You can include the header files and call those functions if you think it does precisely what you want. If you are not sure whether you can use something or if you think some LLVM source code is making the assignment trivial, then please let us know on Piazza (you can use a private note).

You can read research papers (or lectures) available online and try to implement those ideas; if you do, don't forget to cite it in the README file. You can brainstorm your design with the TA during office hours.

5 Testing

You are responsible for writing test cases to test your compiler. We advise you to write simple/toy applications like searching and sorting that test different aspects of language features and compiler components. Feel free to use LLVM test programs.

You can compare the behavior of your low-level compiler with that of LLVM's (replace `llc-olive` with `llc` in your command). Note that LLVM's code generator is more complex than what you are required to implement.

We can use any program for testing your compiler. Your program must be robust (no crashes or undefined behavior) and must generate precise (correct output) and efficient (faster execution time) assembly code. So, please test your code thoroughly.

6 Deliverables

Submit (to canvas) an archive (preferably, `.tar.gz`/`.tgz`) of the source directory containing your source code, build script (`CMakeLists.txt`), test cases, report (PDF), and `README` file. Please do not submit the generated/built object files or binaries.

The `README` file should mention all the materials that you have read or used for this assignment, including LLVM documentation and source files; you can only skip mentioning header files that you have explicitly included in your source code. Please also mention your project partner.

In the report (PDF), mention the status of your submission - the features supported and the optimizations implemented. Describe your design choices briefly (as well as the reasons for your choice). Describe your testing approach (for each component) and mention the tests for which your compiler worked. Please keep it concise. The report is intended for us to help grade your submission better

(the report itself does not have any points). You can also include feedback, like what was challenging and what was trivial.

You should also submit the test cases for which your pass worked (in the same or sub-directory); if they are publicly available, then just mention them in the report.

Please feel free to help each other with the build system on Piazza since that is not focus of this class.

7 Grading

We will do the following on Stampede:

1. Unzip the archive you submit into the `$SRC_ROOT/tools` folder, which will contain a `CMakeLists.txt` that adds your directory as a sub-directory.
2. Run `make` inside your sub-directory in the `$OBJ_ROOT/tools` folder.
3. Use `llc-olive` as mentioned in the implementation guidelines to test your compiler on a few input programs. Your program should not crash.
4. For each program, compare (`diff`) the output of your generated assembly code (compiled to executable) against that of `llc` and it should match.
5. For some programs, compare the performance of your generated assembly code (compiled to executable) against that of `llc` (`llc` is expected to perform better).

Note that this will be done through scripts. So, please follow the implementation guidelines precisely.

Both the features supported and the optimizations implemented will be evaluated, though each feature and each optimization may have different points. The more complicated features for which correct assembly code is generated, higher the points. The faster the generated (correct) assembly code runs, higher the points.

Acknowledgements

This is an adaptation of a project in Dr. Y N Srikant's Compiler Design course at IISc.

Any clarifications and corrections to this assignment will be posted on Piazza.