CS 4240: Compilers and Interpreters, Project 2, Spring 2020

Assigned: February 27, 2020, 15% of course grade
(100 points total, with a maximum of 15 points in extra credit options)
Due in Canvas by 11:59pm on April 22, 2020
Automatic penalty-free extension until 11:59pm on April 30, 2020

Changes since original project handout are in italics

1 Project Description

In this project, you will build a compiler back-end for Tiger-IR that generates MIPS32 assembly code. Your back-end should read in a text-formatted Tiger-IR program, perform instruction selection and register allocation, and write the generated assembly code to a file. The generated assembly code should be able to execute on the SPIM simulator that will be given to you.

1.1 Instruction Selection

The first step is to build an instruction selector that transforms Tiger-IR instructions to MIPS3 instructions that operate on an unlimited number of virtual registers. *It is fine if the instructions generated by your project include pseudoinstructions supported by SPIM. The main criterion is that the code that you generate must be executable on the SPIM simulator.* It will suffice to implement a simple instruction selector that translates one IR instruction at a time. Also, note that all the intrinsic functions in Tiger-IR can be implemented using SPIM system calls.

You can get familiar with MIPS and SPIM through the following links:

- CS 4240 Lecture 13 slides
- MIPS architecture and assembly language overview
- More information about pseudo instructions (which may simplify your work)
- SPIM instruction set (including pseudo instructions, not complete)
- A complete list of (pseudo) instructions supported by SPIM (starting from page A-51)
- MIPS floating point instructions

To generate code for function calls, we suggest that you consider using the calling convention described in the following document:

https://courses.cs.washington.edu/courses/cse410/09sp/examples/MIPSCallingConventionsSummary.pdf. However, you are also free to design your own calling convention, so long as it executes correctly on the SPIM simulator. By default, you should also generate code to ensure that all 10 “temporary” registers, designated as \$t0 – \$t9 (corresponding to physical registers \$8 – \$15 and \$24 – \$25) are saved before each function call, and restored after each function call. This code for register save and restore operations ensures that the program's results will be correct even if a callee function overwrites temporary registers that are being used by its caller. We also recommend that all your register allocator implementations only use the 10 “temporary” registers by default, so no other save/restore operations will be needed. Details on the standard convention for using MIPS registers can be found in http://students.cse.tamu.edu/pritam2309/csce350/reference/quick_ref_MIPS.html.

One of the challenges you will face is in memory management of arrays. Since each array in Tiger-IR has a static size, you can allocate space for it on the stack. Another choice is to allocate arrays in the heap using the `sbrk` system call. If you choose to use the heap, you need not worry about memory leaks or garbage-collecting the arrays in this project (though those are important real-world considerations).

As a debugging aid, we have made available an extension to the Tiger-IR interpreter that supports MIPS instructions instead of IR instructions, but (unlike real MIPS instructions) also works with symbolic registers. For simple test programs, this extension can be used to test your instruction selection implementation before you perform register allocation as discussed in the next section.

Finally, we have provided the following extra credit options, described in more detail in later subsections. The maximum extra credit that will be given is 15 points, regardless of how many options you implement. However, it is possible for you to (say) get partial credit on two options that add up to 15 points:

- *Section 1.3: For this extra credit option, you will need to add one or more optimizations to reduce the number of memory load operations by at least 15%, relative to whichever register allocator you choose as a baseline (naive, intra-block, global) on a single IR program of your choosing (which you should submit with your project).*
- *Section 1.4: Your base project need not support floating-point operations. For this extra credit option, you will need to support floating-point operations and perform register allocation for general purpose registers and floating-point registers separately. Your project report should summarize the results from running a set of test programs on the MIPS simulator that include floating point operations. Note that MIPS has 32 single precision (32-bit) floating-point registers (\$f0 – \$f31), and that floating-point instructions can only work with these registers.*

1.2 Register Allocation

After instruction selection, the main remaining step to get to fully-executing machine code is register allocation. You are required to implement two register allocators in this project: the first is a naive register allocator (Section 1.2.1) and *the second can be a local/intra-block register allocator (Section 1.2.2) or a global Chaitin-Briggs style graph coloring register allocator that spans an entire function/procedure (Section 1.3.1, which can also contribute to extra credit).* You will compare the performance of the code generated by the two register allocators. The choice of register allocator should be enabled by a command-line option when invoking your compiler back-end, and the option usage should be specified in your design document.

1.2.1 Naive Allocation

The simplest register allocation scheme is one in which there is no analysis required. Each virtual register is allocated space on the stack. Before each instruction, its operands are loaded into physical registers; the instruction then executes; and finally the result is stored back into that register's location on the stack. Thus, for each instruction in the IR stream, you will generate and insert the necessary load(s) before that instruction, and you will generate and insert the necessary store(s) after that instruction. This scheme is the slowest, but it will produce correct, working code.

1.2.2 Intra-Block Register Allocation

An improvement to the naive scheme is to identify maximal basic blocks in the IR stream, and then perform a liveness analysis at the intra-block level (i.e. only within each basic block). Notice that at the start of each block, you will need to load a set of variables that you expect to use, and similarly, at the end of each block, you will need to store all values from your registers to memory.

You may use any algorithm you choose for this register allocation, so long as it is correct and an improvement over the naive allocation in Section 1.2.1, One possibility is to use a simple greedy algorithm as follows – assign a register to the live range which has the maximum number of uses in a block, then a register to the live range with the next highest number of uses, and so on. After you run out of registers, all remaining virtual registers will need to be spilled. To spill a virtual register, you need to allocate space for it on the stack and insert load/store instructions for all its accesses. You may need to reserve some registers to temporarily hold the values for spilled registers when they are loaded from the stack.

Note that this approach degenerates to the naive approach in Section 1.2.1, if basic blocks were not maximal but instead contained only one instruction each.

1.3 Extra Credit Option 1 (15 points)

For extra credit (15 points), you will need to combine one or more additional optimizations with any of the two allocators listed above to decrease the number of memory load operations. Example optimizations include:

- Implement a global Chaitin-Briggs style graph coloring register allocator that spans an entire function/procedure (Section 1.3.1).
- Optimization of register save/restore instructions based on which registers are actually used in the function being compiled.
- Careful usage of more registers than the 10 temporary registers.
- Any of the optimizations from Project 1 performed as a pre-pass.

These optimizations should be enabled by a command-line option when invoking your compiler back-end (similar to the -O flag used by C compilers).

You will receive the full 15 points of extra credit if you can obtain at least a 15% reduction in the number of memory loads performed by enabling your optimization option, relative to whichever register allocator you choose as a baseline (naive, intra-block, global) on a single IR program of your choosing (which you should submit with your project). Thus, it is possible for you to receive this extra credit even if you don't fully implement both allocators.

You will receive partial extra credit (< 15 points) depending on the best reduction you can demonstrate in the number of memory loads, e.g., 10 points for a 10% reduction.

1.3.1 Global Chaitin-Briggs Style Graph Coloring Register Allocation

For this option, you will need to build a register allocator that is "global" in scope, i.e., its scope spans the entire function being compiled. First, you will build the control flow graph (CFG) and perform a full CFG liveness analysis across the basic blocks. Then, you will build live ranges and an interference graph. Next, perform graph coloring using Chaitin-Briggs optimistic coloring algorithm to allocate registers. If spilling is required, you can use any heuristic of your choice to select virtual registers to be spilled. As before, if a virtual register needs to be spilled, you should allocate space for it on the stack and insert load/store instructions for all its accesses; and, you may need to reserve some registers to temporarily hold the values for spilled registers when they are loaded from the stack.

1.4 Extra Credit Option 2 (15 points)

An alternative extra credit option (15 points) is to generate correct code with naive register allocation for IR functions containing floating-point computations. Test cases for floating-point numbers are also provided in `https://github.gatech.edu/cs4240sp20/cs4240/tree/master/projects/project2/test_cases`. Note that the extra credit for the entire project will be capped at 15% even if you do both extra credit options.

2 Provided Code

2.1 Helper Code

We are not providing new helper code for this project. You can use the code from Project 1 to parse Tiger-IR files.

2.2 The SPIM Simulator

SPIM is a MIPS simulator you will use to run the MIPS assembly code you generate. Please download SPIM from `https://github.com/rudyjantz/spim-keepstats`. This version outputs some statistics to your console that are useful for performance evaluation.

The installation instructions are available at: https://github.com/rudyjantz/spim-keepstats/blob/master/README_4240.

3 Grading

3.1 Correct Implementations (50 Points)

Several test cases are given to you to test the correctness of your back-end: https://github.gatech.edu/cs4240sp20/cs4240/tree/master/projects/project2/test_cases. You will get corresponding points for correct implementations of the following items:

- Instruction selection + naive register allocation (35 points)
- Additional intra-block register allocation (15 points)

3.2 Performance Evaluation (20 Points)

For this section, you will need to compare the performance of the two register allocators and show the results in your project report. The performance metric we will focus on in this project is the number of memory loads (reads). While the number of loads is only a partial contributor to performance in a real-world setting, it is the metric that is most directly influenced by register allocation which makes it an appropriate focus for this project. This metric will be printed to your console by the provided SPIM simulator.

You will need to run MIPS programs generated from the test cases using your back-end with two different register allocator options, and analyze the performance statistics. If you feel that the given test cases cannot show the benefits of a better register allocator, please design your own test cases and submit them along with your code and report.

3.3 Design (30 Points)

In your final report, in addition to the performance evaluation results, briefly describe the following:

1. High-level architecture of your back-end, including the algorithm(s) implemented, and why you chose that approach.
2. Low-level design decisions you made in instruction selection and register allocation, along with their rationale.
3. Software engineering challenges and issues that arose and how you resolved them.
4. Any known outstanding bugs or deficiencies that you were unable to resolve before the project submission.
5. Build and usage instructions for your back-end, including all test cases that you created to obtain performance results for the two register allocators and the optional extra credit features.

4 Submission

On Canvas, submit a single ZIP file that contains:

- The complete source code of your project.
- The report.pdf file described in Section 3.2 and 3.3.
- Any test cases you added.

5 Collaboration

We will award identical grades to each member of a given project team, unless members of the team directly register a formal complaint. We assume that the work submitted by each team is their work solely. Any clarification question about the project handout should be posted on the course's public Piazza message board. Any non-obvious discussion or questions about design and implementation should be either posted on the course's Piazza message boards privately for the instructors or presented in person during office hours. If the instructors determine that parts of the discussion are appropriate for the entire class, then they will forward selections. Under no condition is it acceptable to use code written by another team, or obtained from any other source. As part of the standard grading process, each submitted solution will automatically be checked for similarity with other submitted solutions and with other known implementations.