CSE 131: Compilers Project 2, Spring 2025
(100 points total)

1 Project Description

In this project, you will build a compiler backend for Tiger-IR that generates MIPS32 assembly code. Your backend 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. To limit the scope of the project, all Tiger-IR programs which will be tested against your backend won't contain any floating–point variables.

As with Project 1, you may choose any of the following implementation languages: Java, Python, C, or C++. Detailed instructions on how to submit your project and what to include are given in Section 3.

1.1 Instruction Selection

The first step is to implement an instruction selector that transforms Tiger-IR instructions to MIPS32 instructions that operate on an unlimited number of virtual registers. The code that you generate must be executable on the SPIM simulator, and it may utilize pseudo–instructions supported by SPIM. It will suffice to implement a simple instruction selector that translates one IR instruction at a time. Use SPIM system calls to implement intrinsic functions in Tiger-IR.

In cases where there are multiple ways to translate an IR instruction to a corresponding MIPS32 instruction, you may use any of them in your implementation as long as doing so does not add unnecessary memory operations or useless instructions. You do not need to care about handling integer overflow exceptions.

While we will learn about MIPS in class, you can also find some helpful resources on MIPS and SPIM at `Project-2/resources`.

- Lecture notes from UArizona: Learning MIPS & SPIM
- Documentation for MIPS & SPIM (refer to sections A.6 & A.10)
- Lecture notes from UWashington: MIPS Calling Convention

There are no strict requirements on the calling convention you implement, but make sure it executes correctly on the SPIM simulator. Make sure you correctly insert register save and restore operations for temporary registers before and after each function call. The program's results need to be correct even if a callee function overwrites temporary registers that were previously used by its caller. We also recommend that all your register allocator implementations only use the *10 temporary registers ($t0 – $t9)* by default.

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 provide the MIPS Interpreter which also works with symbolic registers (unlike real MIPS instructions). For simple test programs, this can be used to test your instruction selection implementation before you perform register allocation as discussed in the next section.

1.2 Register Allocation

After instruction selection, the main remaining step to produce executable 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 is a ***local/intra-block register allocator*** (Section 1.2.2).

You will compare the performance of the code generated by the two register allocators. The choice of register allocator should be possible with a command-line option when invoking your compiler backend, and the option usage should be specified in your design document.

1.2.1 Naive Allocation

The simplest register allocation scheme is the one which requires no analysis. Each virtual register is allocated space on the stack. Before each instruction, the operands are loaded into physical registers. The instruction then executes, and finally the result is stored back into the appropriate 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 will be slow, but it will produce correct working code.

1.2.2 Intra-Block Register Allocation

An improvement to the naive scheme is to identify 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.

Use the simple ***greedy algorithm*** as follows: assign a register to the live range which has the maximum number of uses in a block, then another 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. Simply allocating new stack memory for each spilled register is good enough in this project. You may also need to reserve some registers to temporarily hold the values for spilled registers when they are loaded from the stack.

Your work on analyzing liveness in Project 1 may be helpful in identifying live variables. For intra–block register allocation, you can build an intra–block interference graph and assume that all variables are alive at the end of a block. This allows you to reuse a physical register by storing a temporarily dead value (in the current block) to memory immediately before the physical register is reused. This will save the old register value, which may or may not be used in later blocks. This approach suffices for a correct implementation. However, you are also welcome to consider global liveness information to improve your implementation.

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 MIPS Interpreter

We provide an interpreter along with essential debugging tools for you to debug and check the correctness of the **.s** file generated by your Project 2. See the **README.md** in the mips-interpreter folder for instructions on how to compile and run it. During grading, we will run your code on SPIM.

2.3 The SPIM Simulator

SPIM is a MIPS simulator you will use to run the MIPS assembly code you generate. You can install SPIM by following the instructions below on a Linux machine, or on WSL on windows:

```
$ git clone https://github.com/portersrc/spim-keepstats.git
$ cd spim-keepstats/spim
$ sudo apt-get install flex bison
$ make
$ sudo make install
```

This version of SPIM outputs some statistics to your console that are useful for performance evaluation. Below is an example usage of SPIM.

```
#!/bin/bash
$ spim -keepstats -f helloworld.s
Loaded: /usr/share/spim/exceptions.s
Hello World
Stats -- #instructions : 13
         #reads : 2  #writes 0  #branches 2  #other 9
```

```
# Usage example of feeding input file
# spim -keepstats -f file.s < input.in
```

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. The value reported by SPIM is a dynamic count which depends on program input, not a static count that can be made only from the MIPS instructions.

3 Grading

3.1 Correct Implementation (50 Points)

You may organize your implementation in any way using one of Java, Python, C, or C++. Your submission must include two shell scripts, `build.sh` and `run.sh`. `build.sh` should build your submission from source, as in Project 1. Make sure to document in your `report.pdf` any dependencies or version numbers required for `build.sh`. `run.sh` **must take exactly one terminal argument—a path to an input IR file—and one flag specifying either the naive (`--naive`) or intra-block greedy register allocation (`--greedy`). It should output an output a file `out.s` containing the generated MIPS32 code.** For example, if a Tiger-IR file is located at `path/to/file.ir`, then it should be possible to run `run.sh` as follows.

```
$ run.sh path/to/file.ir --naive
# Produces out.s
$ run.sh path/to/file.ir --greedy
# Produces out.s
```

Several test cases are given to you to test the correctness of your backend: `public_test_cases`. Similarly to Project 1, we may test for correctness using additional test cases which are not provided. Since there are many implementation choices in instruction selection, there is no canonical output which your backend must produce. To test it, we compare the results of running the TigerIR interpreter (same as in Project 1, `src`) on the input IR program with the results of running SPIM on an output assembly program. As in Project 1, you are welcome to design and add your own test cases to check that instruction selection behaves correctly on all TigerIR instructions.

You will get points for correct implementations of the following items:

- Instruction selection and naive register allocation (35 points): This portion of the evaluation is based on the correctness of programs generated by your backend; *i.e.*, do the programs produced by your backend pass test cases.
- Additional intra–block register allocation (15 points): This portion of the evaluation is based on the correct implementation of intra–block register allocation and the number of load instructions used by the generated code. To receive full credit, your implementation must

still pass test cases for correctness, reduce the number of load instructions compared to the naive implementation, and include at least intra–block liveness analysis.

Since there are many different ways to implement register allocation and the results depend on the input to a program, it is not possible to give concrete values for the number of load instructions in each test case. Depending on the input program, if your register allocation reduces the number of memory loads by 30-60% over the naive approach, you should be in good shape. We may also inspect your implementation in evaluating the correctness of intra–block register allocation.

3.2 Performance Evaluation (20 Points)

For this section, compare the performance of the two register allocators and show the results in your project report. You will need to run MIPS programs generated from the test cases using your backend with the two different register allocator options, and analyze the performance statistics. The provided SPIM simulator will print the number of memory reads to your console. 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 backend, 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. Any test cases that you created to obtain performance results during testing. In addition, include any additional usage instructions for your backend, including any dependencies or version numbers required for `build.sh`.

4 Submission

On Canvas, submit a single ZIP file that contains:

- The complete source code of your project.
- The files `build.sh` and `run.sh` as described in Section 3.1.
- 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.