CS 8803-O08: Project Phase 2

Santosh Pande
santosh.pande@cc.gatech.edu

Georgia Institute of Technology — Due: April 26, 2021 AOE

Overview

The purpose of phase 2 is to implement the backend of a compiler for the Tiger language. You will use the 3-address IR generated previously to output MIPS assembly code. You will complete this process in three steps. First, you will allocate variables present in the IR to physical registers in the MIPS machine. Then, you will select MIPS instructions for each quadruple and generate the MIPS assembly. Finally, you will implement compiler optimizations and compare the code quality improvements they provide.

Requirement 1

Accept IR as an input file using the flag `-r <path/to/file.ir>`. When both `-r` and `-i` are present on the command line, the compiler is free to choose either the Tiger or IR file as the starting point.

1 Register Allocation

Intermediate representations generally contain large numbers of temporary variables to hold the intermediate results of evaluating an expression. The IR also contains instructions for storing variables to memory and loading data from memory into variables. Before generating MIPS instructions, you must allocate the variables (including temporaries) to registers available in MIPS.

The general idea of register allocation is straightforward. At any given point, the processor can hold a finite number of values in its registers, and the values being operated on must be in those registers. If there are not enough registers available in the processor then we must temporarily store (known as a spill) a variable's value in memory. This will require additional store and load instructions.

There is a large body of knowledge and research surrounding register allocation, and it is a critical phase of compilation for obvious reasons. For the purposes of this project, we will be implementing a naïve register allocation strategy, a basic block-based register allocator, and finally a global Briggs' style graph coloring register allocator. You will implement these three register allocation schemes starting with the simplest (naïve) and ending with the most complex (global Briggs' style).

1.1 Naïve Register Allocation

The naïve register allocation scheme requires no analysis. The operands for each instruction are loaded into registers from memory. Then, the instruction is executed and the result is stored back into memory. This scheme is very slow at runtime, but it produces correct, working code.

Requirement 2

Implement a naïve register allocator. Use this allocator when the `-n` flag is provided.

1.2 CFG Construction & Intra-block Allocation

An improvement on the naïve secheme is to allocate registers to variables on a block by block basis. Basic blocks are identified by constructing a control flow graph (CFG). Then, you will perform liveness analysis on each basic block. At the start of each basic block, you will load the set of variables that are used (live) in the block into registers. Similarly, at the end of the basic block, the values in the registers must be stored back into memory. Once you have built the live ranges, you can use a simple greedy algorithm to assign the variables to registers. The variable with the highest number of uses in a block gets the first register, then the variable with the next highest number of uses gets the next available register. If you run out of registers, the remaining live variables must be loaded from memory and stored back for every use. This aspect is the same as naïve allocation.

Requirement 3

Implement an intra-block register allocator using a CFG and liveness analysis. Use this allocator when the `-b` flag is provided.

1.3 Global Briggs' Style Graph Coloring Allocation

The last exercise in register allocation is full inter-block (whole function) register allocation. This requires you to:

1. build a control flow graph
2. perform liveness analysis across the basic blocks
3. build the live ranges and webs
4. build the interference graph
5. perform graph coloring using Briggs' optimistic coloring algorithm
6. allocate the registers

This allocation scheme is complex and involved. We recommend that you test each step individually as you go and refer to the video lessons and textbook readings for all the details.

Requirement 4

Implement a global (whole function) Brigg's style graph coloring register allocator. Use this allocator when the `-g` flag is provided.

Requirement 5

Print the corresponding data structure to standard output when the following flags are provided: `--cfg`, `--liveness`, `--webs`, `--colors`. You are free to choose the exact format of the output.

Info: A default register allocator for the compiler is not defined. If an allocator flag (`-n`, `-b`, `-g`) is not supplied, any of the three register allocators may be used.

2 Instruction Selection & Code Generation

Instruction selection is the process of converting the Tiger IR that was generated, selecting the corresponding assembly operations, and outputting assembly code in the proper format. For this project, you will generate MIPS assembly code which will be run in the SPIM simulator.

2.1 MIPS

MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed in 1985. While it is now generally used for education, historically, MIPS was used in personal computers, servers, and video game consoles such as the Nintendo 64, Sony PlayStation, PlayStation 2, and the PlayStation Portable.

MIPS has thirty-two general purpose registers each with their own intended use and conventions. We recommend that you take time to familiarize yourself with the MIPS registers and instructions. Examples of common programming structures converted to MIPS are available here.

Requirement 6

Perform instruction selection and generate valid MIPS assembly.

2.2 SPIM

SPIM is a MIPS simulation that you will use to run the MIPS assembly output by your compiler. For this project, you will use a modified version of SPIM which collects additional information about the running program. Follow the example below to setup your installation of SPIM:

Command Line

```
$ apt install --assume-yes git make g++ bison flex
$ git clone https://github.com/rudyjantz/spim-keepstats
$ cd spim-keepstats/spim
$ make
$ make install
$ spim -keepstats -f ../helloworld.s
Hello World
Stats -- #instructions : 13
    #reads : 2  #writes 0  #branches 2  #other 9
```

Warning: You must generate a `main:` label where your program's statement sequence begins. Otherwise, SPIM will not run properly. You must also generate a final instruction, `jr $ra` to return to the caller of your program.

Requirement 7

Output the generated MIPS assembly in a format readable by SPIM.

Requirement 8

The MIPS assembly output file should have the same name as the input file with the extension changed to `.s`.

3 Compiler Optimizations

A compiler optimization is an algorithm which takes a program and transforms it to produce a semantically equivalent output program that is "better" in some way. Common optimizations attempt to minimize a program's execution time, memory footprint, storage size, or power consumption. Often, these optimization problems are NP-complete or even undecidable.

The intra-block and global register allocators you implemented previously are examples of a compiler optimization. When analyzing the affects of these optimizations we are interested in two code quality metrics:

- instruction count: number of MIPS assembly instructions
- code size: static size of the MIPS assembly

Requirement 9

Document and analyze the code quality of the assembly generated for the two benchmarks provided when using the three register allocation methods. Name the report `allocator_analysis.pdf`.

3.1 Extra Credit

For extra credit you may implement Superlocal Value Numbering (SVN):

From the CFG, first detect the extended basic block or EBB (see the relevant lesson and book material related to this). Then implement the value numbering algorithm that detects and eliminates the redundancy of expressions within the EBB by starting with the root of the EBB and walking towards the leaves. SVN walks the IR top down in the above order, performs value numbering on IR statements and simultaneously performs three different types of optimizations:

1. propagation and folding of constants

2. removal of lexically identical expressions(commutativity wherever applicable included)
3. removal of value identical expressions that are not lexically identical

Please classify and report the number of each category of optimization performed. You will position this optimization pass before the Global Register Allocation pass. Use SPIM to output dynamic instruction counts and code size with the optimization enabled. Compare them with values seen when the SVN optimization is disabled. Please enable this optimization pass using a flag `--svn`.

To receive credit, please include an example Tiger program which benefits from the optimization and a report detailing the desired information from above. The optimization should only be applied when the flag listed above is provided.

4 Grading

4.1 Building and Testing

You must include a *Makefile* at the root of your source code directory. You can assume that `javac 11.0.9.1` and `g++ 7.5.0` will be available via the `PATH` variable. Your *Makefile* should compile the code, then build a single JAR or EXE called `tigerc` under a `cs8803_bin` folder.

4.2 Deliverables

- Register allocator code quality analysis (allocator_analysis.pdf)
- Design internals report (design.pdf)
- Compiler source code (C++/Java)

4.3 Rubric

• Register allocation	**40 points**
– Naïve	10 points
– Intra-block	15 points
– Global Briggs' style	15 points
• Instruction Selection & Code Generation	**40 points**
– Basic operations including integer and floating point expressions	10 points
– Control flow	10 points
– Array operations	5 points
– Functions and scope operations	5 points
– Example programs	10 points
• Reports	**20 points**
– Register allocation code quality analysis	10 points
– Design internals	10 points
• Extra Credit	*10 points*