COMS W4115
Programming Languages and Translators
Programming Assignment 4: MIPS Tiger Compiler

Prof. Stephen A. Edwards
Columbia University

Assigned April 17th 25th, 2002
Due 11:59 PM on May 6th, 2002

> For this assignment, you will be adding code to the classes for the interpreter's intermediate format that translates it into MIPS assembly code. To test your output, you will execute this MIPS assembly code on the SPIM simulator. When you complete this assignment, you will have a fully working Tiger compiler.

In the last assignment, you translated the AST you created in the first assignment and tested in the second assignment into instructions for a virtual machine whose instructions knew how to interpret themselves. In this assignment, you will write code that translates each of those instructions into MIPS assembly code that can run on the SPIM simulator.

The quickest way to learn about the MIPS architecture is to read through James Larus's Appendix A of Patterson and Hennessey's *Computer Organization and Design*, second edition. I've put a PDF version of this appendix at `~cs4115/spim.pdf`.

Your main challenge in this assignment is writing the `mips()` method for classes that represent instructions and operands that returns a string containing MIPS instructions that implement the instruction. For example, the Binop method should generate a MIPS `add` instruction when it represents an addition instruction.

We will be generating slow but correct MIPS code. A good compiler would try to use as many registers as possible and only access memory when absolutely necessary. We, by contrast, will keep most information in memory (i.e., on the stack), and only use registers as temporary scratchpads. This choice makes it much easier to generate code, but would be an unacceptable choice in a production compiler.

```
# "Hello World" for the SPIM simulator
    .data
st1:
    .asciiz "Hello, World!\n"

    .text
    .globl  main
main:
    li      $v0, 4    # Code for print_str
    la      $a0, st1  # Load address of string constant
    syscall

    li      $v0, 10   # Code for exit
    syscall           # Terminate the program
```

Figure 1: "Hello World" in MIPS assembly code for SPIM.

We will not use the MIPS calling conventions of putting the first four arguments in registers (see Appendix A of CO&D), but instead pass them on the stack as we did with the interpreter.

1 Getting Started

I've installed the SPIM simulator and put the "Hello World" program in `~cs4115/prog4/hello-world.s`. You can invoke the text-mode simulator as follows:

```
$ cp -r ~cs4115/prog4 .
$ cd prog4
$ ~cs4115/bin/spim -file hello-world.s
SPIM Version 6.4a of January 12, 2002
Copyright 1990-2002 by James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /u/4/c/cs4115/lib/trap.handler
Hello, World!
```

The `xspim` command (also in `~cs4115/bin`) is a graphical version of the same simulator that you might find easier to use. It continually displays the state of the registers as well as disassembled code near the program counter. Single-stepping through code is usually a very good way to figure out what does and does not work.

Page A-51 in Appendix A lists assembly directives. In Hello World, the `.data` directive says the string constant is to be placed in the data segment; similarly, the `.text` directive puts code in the text segment. These directives can be interleaved.

The `.globl` directive marks its argument as being a global label, one that is visible to other files. SPIM insists the `main` label—the start of the program—be global.

Labels, such as `st1` and `main` start at the beginning of the line and terminate with a colon. Note that all labels must be unique, something the interpreter didn't require (the names were "just for show"). Calling `new Interp.Label()` automatically generates a unique label.

2 Generating Code

The MIPS processor has thirty-two "general-purpose" registers, listed in Figure 2. Your code should probably restrict itself to using `$t0`–`$t9`, `$zero`, `$sp`, and `$fp`. We will not be using `$a0`–`$a3` for function arguments; these will be passed on the stack. Similarly, `$v0`–`$v1` will not be used to return a function's value.

$zero	0	Hardwired constant 0
$at	1	Reserved for assembler
$v0–$v1	2–3	Temporaries & function return value
$a0–$a3	4–7	Function arguments
$t0–$t7	8–15	Temporaries not preserved across call
$s0–$s7	16–23	Temporaries you must preserve across calls
$t8–$t9	24–25	Temporaries not preserved across call
$k0–$k1	26–27	Reserved for OS Kernel
$gp	28	Pointer to global area
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Function return address

Figure 2: MIPS registers.

8($fp)	"fp(-2)"
4($fp)	"fp(-1)"
0($fp)	Return address
-4($fp)	Static Link
-8($fp)	Saved frame pointer
-12($fp)	"fp(0)"
-16($fp)	"fp(1)"

Figure 3: Activation Record Layout

Figure 3 shows the layout of the activation record. The function return address, static link, and previous frame pointer are stored at the beginning of the activation record followed by the actual data fields. Understanding this layout is critical for generating code for `jsr` and `rts` instructions, which must save and restore this state properly. I've written the code for these two instructions for you.

The MIPS only has one "complex" addressing mode: register plus immediate offset, so the more complicated addressing modes such as `StackLinks` must be simulated through additional instructions. For example, loading an operand such as `2*fp(3)` into `$v0` becomes the code sequence

```
mov $v0, $fp
lw $v0, -4($v0)
lw $v0, -4($v0)
lw $v0, -24($v0)
```

This first loads the current frame pointer into register `$v0`, then follows the static link twice. Finally, it reads field 3, which is at offset $-12-3\times 4=-24$.

3 Putting it together

I've added a new instruction, Ent, that generates code for setting up an activation record at the beginning of a function. I've modified RecordInto.java to insert this instruction just after the label at the beginning of a function. If you modified RecordInfo.java, you will want to merge in this change.

Broadly, you need to write or complete the `mips()` methods for the pseudo-instructions such as Jmp, Bnz, etc. I've written them for Psh, Jsr, Rts, Ent, Mov, Label, and Binop, although Binop is incomplete.

Things you will have to do:

- Write the `mips()` method for the remaining statements: Neg, Jmp, Bnz, Bz, Rec, and Arr.

  Of these, Rec and Arr will be the most challenging. Use the `sbrk` system call (see the SPIM documentation) to allocate memory (you pass it the number of bytes you want, it returns the starting address of your new block). With the Arr command, you will also have to generate MIPS code that initializes the contents of the array. This should be a simple counted loop that can be done in registers.

- Write the `mipsGet()` and `mipsSet()` code for BlockRel. Use the code in FrameRel and StackLinks as a starting point.

- Complete the code for Binop. It should work for most operands, but you need to write a variant for string comparisons. Create a "Binopstr" class specifically for string comparisons and modify your translator to generate the right instruction depending on the type of the operands for `=`, etc.

- Write code for the standard library functions. I've done two of these for you already (print and printi). Note that functions that return new strings should use the `sbrk` system call supplied by SPIM to allocate space for their results.

- Test your code. You should be able to use many of the same test cases that you used for the third programming assignment. In fact, you can compare the output of the interpreter with the output of your compiler by running SPIM on the assembly you generate and comparing the results. This is how we will test your code.

4 Deliverables

Feel free to use, modify, or ignore any of the files I give you. Ultimately, you will only be graded on whether the simple test programs on which we run your interpreter produce the correct results. Make the print and printi standard library functions work correctly.

As before, use `~cs4115/bin/submit_code` to submit

- All the .java files you created or modified in the Interp package.
- All other .java files (and .class files for the parser) you use or create, including those we gave you.
- A README file describing your compiler. I want to hear how you dealt with
  - Code for Rec and Arr
  - String comparisons in Binop
  - The standard library functions
- A file called `MEMBERS` that contains a space-separated list of the uni IDs of each of the members in your group.

Make sure we can build your compiler by running ANTLR on `TigerTranslate.g` and then `javac` on `TC.java`. Your compiler should take a Tiger source file and print assembly code that can be run directly on SPIM on the standard output. We do not want a Makefile.