CS352s2014 Project 6: MIPS Code Generation

April 11, 2014

1 Objective

The objective of this project is to write a code generator for MiniJava targeting MIPS, as implemented by the SPIM MIPS simulator.

2 Code

The project's code environment is available on the course web page. We are providing Java class files for a complete compiler, including the code generator, but are not providing accompanying Java source. Through the course of these projects, it will be your job to rewrite the Java source for each component, creating a complete compiler of your own design. You are advised to keep a copy of the Java class files so that you can use them to generate correct output and test against your own.

You are free to use the Java class files provided in lieu of your own implementations of the lexer, parser, SSA compiler, type checker and register allocator, or to use your own if you prefer. If you use your own, then upon finishing this project, you will have a complete compiler in which all major components are of your own construction.

2.1 Testing

The compiler provides a number of tools for testing each stage. They are available in the `bin` directory. One is of value for testing the code generator.

- `bin/mjcompile-mips`: Compiles MiniJava code into MIPS assembler code.

To test the output of `bin/mjcompile-mips`, you will have to use the `spim` simulator. An example usage of `spim` with one of the provided examples:

```
$ ./bin/mjcompile-mips examples/Factorial.java > test.s
$ spim -file test.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
```

```
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
3628800
$
```

The behavior of programs generated by your code generator should be correct. The precise code generated is not expected to be identical to that of the version provided.

A template of `edu.purdue.cs352.minijava.backend.AsmMIPS` is provided in the code environment.

3 Code Generation

The challenges of code generation are:

- Correct pinning of registers.
- Correct usage of the register allocator.
- Assocation of *abstract registers* from the register allocator with *conrete registers* of MIPS.
- Generating static data such as vtables.
- Use of the MIPS linkage convention.
  - Generation of prologue with enough free space for spills and both caller- and callee-saved registers.
  - Generation of epilogue which undoes prologue steps.
  - Correct caller-saving of registers during Call, callee-saving of registers during prologue/epilogue.
  - Use of argument/return registers.
- Name mangling (handled automatically in the template).
- Generation of correct code (using correct registers) per each SSAStatement.

3.1 MIPS linkage convention

MIPS has 32 registers, of which 31 are modifiable (register 0 always stores the value 0). The registers free for general-purpose use are a0-a4, t0-t9 and s0-s7. sp and fp represent the stack and frame pointer, respectively, and ra the return address.

Function calls expect their arguments in a0-a4. Any argument registers which are *not* used for function arguments may be considered caller-saved registers, and used freely. It is the role of the code generator to pin the `Parameter` and `Arg` statements to the argument registers such that their values are always in the correct registers. `Parameter`, in fact, should generate no code; its only purpose is to assure that the parameter is expected in an argument register.

Function calls return in register `v0`. As an extension to the standard MIPS calling convention, your compiler is expected to use `v0` as an additional argument register for the value of `this`. i.e.:

- When a function is entered, `v0` will refer to the receiver (`this`).
- When a function call is performed, the previous value of `v0` must be stored (as a caller-saved register) and `v0` must be replaced with the callee's `this`.
- When a function call completes, the caller must put the new value of `v0` (the return value of the function) into the register that the register allocator assigned to the `Call` statement, and restore the original value of `v0`.
- When a function returns, it must put its return value in `v0`, discarding the value of `this`.

This is merely a recommended strategy, and not a part of the standard linkage convention.

Since code generation is likely to require a "scratch" register for temporary storage (within the duration of a single `SSAStatement`), you are recommended to use `v1` for that purpose. The template excludes `v1` from the list of registers it provides to the register allocator for this purpose.

Since your compiler does not link with the code generated by any other compilers (i.e., it is entirely self-contained), you could theoretically implement your own linkage convention, and are free to do so if you please. You are graded only on the correct behavior of your compiled code, and not on strict adherence to the linkage convention. For the sake of ease of implementation, however, it is recommended that you follow the linkage convention with only the extensions mentioned above.

3.2 System Calls

The `NewObj`, `NewIntArray` and `Print` statements depend on system calls. Functions are provided in the template which you may use to implement them:

- `minijavaNew`: Allocates a new object. Expects the vtable pointer in register a0 and the size, in words, in register a1. Returns an address in v0, which has the vtable pointer at offset 0 and 0 at all other locations.
- `minijavaNewArray`: Allocates a new array. Expects the size of the array, in words, in register a0. Allocates that much space plus 4 extra bytes. Returns an address in v0, which has the size at offset 0. The remainder of the space is to be used as the array.
- `minijavaPrint`: Prints an integer. Expects the value to print in a0.

3.3 Registers

The register allocator is generic, and so operates on abstract registers, simply numbered up from 0. You will need to maintain a mapping of abstract registers to real registers in order to generate correct code. The template provides a sample mapping in the final field `freeRegisters`. If you follow the template, then the register values in the SSA will become string register names like so:

```
registers[freeRegisters[ssaStatement.getRegister()]]
```

The template's `argFreeRegisters` field maps argument registers to their register-allocator numbers, which happen to be 0, 1, 2 and 3. You will need such a mapping to perform pinning.

The `Parameter` and `Arg` statements require register pinning. Because only the first four arguments are in registers, this means that the pinning and code generation for arguments 0–3 will differ from those of arguments 4 and up:

- `Parameter`
  - 0–3: Will need to be pinned to a register. This pinning actually assures that no code must be generated for `Parameter`: The linkage convention assures that the argument will be in the correct register, so pinning `Parameter` itself simply assures that that register is used for the parameter value.
  - 4–: Will *not* need to be pinned to a register. Must generate code to retrieve the argument from the stack.
- `Arg`
  - 0–3: Will need to be pinned to a register. Simply moves the argument value (the result of the `Arg`'s `left` statement) into the (pinned) argument register.
  - 4–: Will *not* need to be pinned to a register. Must generate code to store the argument on the stack.

3.4 Spills

The register allocator may cause spills, changing the code that the compiler is compiling. It is up to the code generator to reserve sufficient space for all spilled values and correctly implement `Store` and `Load`.

3.5 Object Orientation

It is the role of the code generator to generate object layouts and virtual tables for objects, and to use them properly in the `Member`, `MemberAssg` and `Call` statements. In the canonical implementation, the logic for this is implemented in `edu.purdue.cs352.minijava.backend.ClassLayout`. If you would like to implement this logic similarly, a template is provided for this class. As you are only graded on the correct behavior of your code generator, you may choose to implement this logic differently if you please.

3.6 Control Flow

Labels in MIPS may simply be duplicated directly from the `special` field of a `Label` statement. It is recommended that you precede these labels with a single dot (.), which indicates that the labels are local to the surrounding function.

`Goto`, `Branch` and `NBranch` may simply jump (under the correct condition) to the label. You are free to assume that the label is defined. The code generator does not need to be further concerned with jumping.

To separate the epilogue from code, it is recommended that you implement `Return` to jump to a special label set aside for the epilogue (as well as storing the return value into `v0`, of course). Note that in our compiler, `Return` will always be the last statement, making this jump technically unnecessary, but it does allow the compiler to generalize to extensions of MiniJava which allow `return` statements anywhere.

Most of the complication of `Call` is in `Arg`, described above. Depending on how you implement the linkage convention, you may pre-allocate stack space for caller-saved registers in the prologue (this is what the canonical implementation does), or allocate it during the `Call` itself (this is the easier solution and is no less efficient). `Call` is expected to save its caller-saved registers, get the receiver (value of `Call`'s `left`), get the method to be called out of the receiver's vtable, perform the call, save the return value, and restore the caller-saved registers.

3.7 Others

With the exception of the linkage convention (`Parameter`, `Arg` and `Call`) and control flow (`Label`, `Goto`, `Branch`, `NBranch` and `Return`), all SSA statements should correspond relatively directly to zero or more MIPS operations. If you are not familiar with MIPS, it is recommended that you study the output of the canonical implementation to learn these MIPS operations.

4 Goals

For this assignment, you must implement one Java class: `edu.purdue.cs352.minijava.backend.AsmMIPS`.
If you follow the canonical implementation precisely, you will also implement `edu.purdue.cs352.minijava.backend.ClassLa` which is used by `AsmMIPS`.

The interface to `AsmMIPS` is the `compile` method, which takes an `SSAProgram` as its argument and returns the compiled code as a string.

A template will be provided in `edu/purdue/cs352/minijava/backend/RegisterAllocator.java-template`.

A template for `ClassLayout` is also provided, in `edu/purdue/cs352/minijava/backend/ClassLayout.java-template`.
Please note that you are not required to use `ClassLayout`, but if you do, you may not use the canonical implementation, and must implement it yourself.

There are many correct ways to generate code for a given program. Your output is expected to have the correct behavior in the SPIM MIPS simulator, and not necessarily to be identical to the output of the canonical implementation.

5 Submission

You will submit your project code via the `turnin` command. Please include the entire project code directory, including all the class files provided by us, the `bin` and `examples` directories, etc.

To turn in a project directory: `turnin -c cs352 -p proj6 <project directory>`

This project is due by Friday, April 25, 2014 at 12:30PM Purdue time. i.e., before class, *not* before midnight. Late submissions will not be accepted.

6 Grading

We will test your code on machines similar to those in the Linux labs used by this course's PSOs. Your grade will be based on the behavior of code you generate in the SPIM simulator for a number of examples.

To reiterate: The only grading criterion is correct behavior of the output code. You are free to disregard linkage conventions, implement vtables in your own way, or do anything you please so long as the result is correct. You are recommended, however, to follow the standards.