Part III : Code Generation

Important: these instructions are subject to change. Please make sure to check any updates made to this page (use the "watch" feature on gitlab to be automatically notified).

The goal of part III is to write the code generator, targeting MIPS32 assembly. For this part, you will only be using virtual registers (except for special purpose registers such as `$sp, $fp, ...`). We provide you with a naive register allocator (`NaiveRegAlloc`) that assigns each virtual register to a label and uses memory to store the content of the registers as seen in class.

Tips

- Write many small passes to break down the complexity of the task ahead of you. For instance, have one pass to perform variable allocations, one to allocate string literals, one to emit code for the value of expression, ...
- Make use of the `children()` method when writing your passes to reduce the size of your code.
- When you are not sure about the correct semantics of your code, go with the C semantics (unless otherwise specified).

0. Setup and Learning

Your first task consist of setting up the MARS MIPS simulator. First download the simulator here and follow Part 1 of the tutorial to learn how to use the simulator. We also encourage you to have a look at the documentation provided by MARS which can be found here as well as the MIPS 32 Instruction Set Quick Reference. For a more detailed explanation about the MIPS architecture, please have a look at the MIPS Assembly WikiBooks. Another MIPS summary of all the instructions supported can be found here: MIPS Green Card

You can also use this online simulator for debugging your programs as the comments remain visible when stepping through instructions.

Important: The marking system will run the Mars simulator from the command line which may change slightly the behaviour of your program (especially when it comes to handling input). You should always make sure that all your tests execute correctly with the Mars simulator run from the command line with the following command: `java -jar tests/gen/Mars4_5.jar sm nc me tests/gen/custom.asm`

1. Generating a simple program and main function

Your first real task should consist of producing an empty program (e.g. just an empty main function) and see that you can produce an assembly file. A well-formed assembly program should always have a `main` function with the following signature: `void main()`. If the signature of `main` is different, or if `main` is missing, we will consider this as undefined behaviour. The `main` function will always act as the entry point to your program. You should ensure that upon starting, the simulator will execute the code of the `main` function. If you use the command line above, the `sm` option will ensure that execution starts at the instruction with global label `main`.

You should ensure that the simulator exits properly when the main function finishes (otherwise random instructions might execute). You can use the following sequence of instruction upon returning from main:

```
li $v0, 10
syscall     # Use syscall 10 to stop simulation
```

To simplify, we will assume that the main function is never called recursively.

Next, we suggest that you implement the `print_i` function using the corresponding system calls (check the lecture notes and the link above to the MARS documentation that explain how to do this). To test it, you could implement support for integer literals and have a simple call to `print_i` in `main`. To understand how instructions can be generated using the starting code we give you, take a look at the `Test` class in the `gen/` package.

2. Binary Operators

Having succeeded in compiling a very simple program, your next task should be to add support for all the binary operators, which is mostly done by implementing the `ExprCodeGen`. When you need to request a new virtual register to store the results of an operation, simply instantiate one with `Register.Virtual.create()`.

Integer operations

All operations on integer types should be performed using signed arithmetic.

Logical operators

Please note that the `||` and `&&` operators should be implemented with control flow as seen in the lecture. In the following example

```
if ((1==0) && foo() == 2)
    ...
```

the function foo is never called at runtime since the semantics of `&&` are that if the left side is false, the right side expression should not be executed. Similar logic applies for `||`.

3. Variable allocations and uses

Next, you should implement memory allocation of global and local variables.

As seen during the course, the global variables all go in the static storage area (data section of the assembly file).

The local variables (variables inside a function) go onto the stack. You should allocate them at a fix offset from the frame pointer (`$fp`) and store this offset directly in the `VarDecl` AST node as a field (alternatively, you can store them in a symbol table).

You should complete and implement the `MemAllocCodeGen` class to deal with the allocation of variables. Note that you can also separate the allocator into a global one and a local one. In this case, you can call the global allocator from the `ProgramCodeGen` and then call your local allocator just before processing the function declaration in the `FunCodeGen`.

Next you should implement the logic to read and write local or global variables. You can use the `lw` and `sw` instruction to read from or write to a variable respectively. The tricky part will be to identify the location of the variables; either a label if globally allocated, or an offset from the frame pointer if locally allocated. We encourage you to store this allocation information in the `VarDecl` node when allocating variables (or again use a symbol table if you wish to).

`sizeof` and data alignment

We will adopt the following specification for the size of the different types: `sizeof(void)==0`, `sizeof(char)==1`, `sizeof(int)==4`, `sizeof(pointer_type)==4`

Arrays should always be represented in a compact form, while struct may require padding. As seen during the lecture, you must ensure that variable declaration (and fields in structures) are aligned correctly.

4. struct/array accesses and assignments

Next you should add support for struct and array accesses. This can be implemented using the `lw` and `sw` instructions for struct and a combination of `add` instruction with the `lw` and `sw` instructions for array accesses. The idea is to get the address of an array into a register, then add to it the index, keeping in mind that addresses are expressed in byte. So, an access to `a[x]` where `a` is an int array means an offset of `x*4` from the base address where the array is stored).

As part of this step, we also suggest that you implement assignments.

5. Branching (if-then-else, loop, logical operators)

We suggest that you then implement the loop and if-then-else control structures as seen during the course using the branch instructions.

6. Function calls

You can them move on to implementing function calls, by far the most challenging part.

To keep things simple, we highly recommend that you pass all arguments and return values using the stack, rather by register. This will simplify greatly your code generation logic.

As seen during the lectures, you have to emit instructions on the caller side before the call occurs (precall) and after the call (postreturn). On the callee side you have to emit instructions in the epilogue and prologue.

To facilitate the implementation of function call, we provide you with two "fake" instructions: `pushRegisters` and `popRegisters`. These two instructions are responsible for pushing all the registers used by the function onto the stack and restore them. These two instructions are "expended" during register allocation since this is only at that stage of the compilation that we know exactly how many registers the function uses. We suggest that you follow the convention presented in the lecture when it comes to function calls.

Beware that structs are passed by value, both when used as argument to the function and when returned from a function.

Updating the stack pointer

When manipulating the stack pointer, you should use the `addiu` instruction to avoid encountering an exception with an integer overflow. This can happen if the `$sp` is initialized at 0 by the simulator.

7. Standard library functions

Finally, you should add support for all the standard library functions found in the file `minic-stdlib.h` provided in the `tests/` folder. These should all be implemented using system calls.

New Files

A new package has been added under `gen/`. This package should be used to store your code generator.

- The `gen.CodeGenerator` is the only class which `Main.java` directly interfaces with.
- The `gen.*Gen` classes are the main classes that you will use to produce code (you can add more if you wish of course).
- The class `gen.Test` shows you an example on how to emit instructions.

Another new package has been added under `gen/asm`. This package defines the components of assembly programs:

- `gen.asm.Register` represents registers and defines most MIPS32 registers in its `Arch` subclass.
- `gen.asm.AssemblyProgram` represents assembly programs that consist of several `Section` instances. `AssemblyProgram`'s `emit` methods provide a fluent, type-safe instruction generation API.
- `gen.asm.AssemblyParser` can parse textual assembly programs as `gen.AssemblyProgram` instances. `AssemblyParser` restricts itself to the subset of assembly programs that the compiler can output.
- `gen.asm.AssemblyItem` has subclasses for the items that appear in an assembly program: `Label`, `Instruction`, `Directive` and `Comment`.
- `gen.asm.Instruction` has subclasses that represent families of instructions with similar behavior.
- `gen.asm.OpCode` enumerates MIPS opcodes.

Note: Do not modify the files under `gen/asm`. For grading reasons, we may roll back these files to the original version we provided. This rollback will overwrite any and all local changes you made, likely breaking your compiler if you made changes. Open a question on ED if you need additional features that the classes in `gen/asm` do not support, such as an instruction/opcode that is essential but not currently exposed.

The new package `regalloc/` contains a naive register allocator: `NaiveRegAlloc`. You should not need to modify it.