Stage 5: Code Generation

Due Jun 5, 2021 by 11:59pm **Points** 100

Introduction

In this assignment we will transform the IR into machine executable x86-64 assembly code. Because generating executable files involves can be very tricky, we will instead generate an assembly listing. This listing can be translated by a compiler like gcc into a native executable.

Generating efficient x86 code is difficult due to a limited number of registers and many complex instructions. Hence, we will only use a subset of x86 instructions, treat all values as 64 bit and load values only into registers when necessary.

Some resources to get up and running with x86 are:

- x64 Cheat Sheet and
- X86-64 Architecture Guide

We strongly recommend looking at both of these.

You may also find looking at the assembly output of other compilers useful. You may find the website Compiler Explorer to be useful. You can also run gcc with the -S option.

Code generation

Global variables

We need to allocate space for every global variable in our program. Global variables are defined in the IR with allocate instructions. To allocate space in x86 for a global variable, we just need to emit

```
.comm VarName, Size, 8
```

The 8 at the end sets the alignment to 8, so with this directive we allocate `Size` bytes of memory.

Functions

As a rough overview, the code generator will do the following steps for every function:

1. Assign labels to all instructions that can be the target of a jump a label.
2. Rearrange all instructions of the body into a linear list and generate the corresponding x86 code.
3. While generating code for instructions, we need to update the map and size of our stack.
4. Emit the function prologue, emit the body and emit the function epilogue.

Assigning labels

An instruction is assigned a label, if it either is the target of a jump control edge or it has two predecessors (i.e., the join-nop of an if-else construct).

To implement this, we perform a depth first search on the function starting with the start instruction. We need to keep track of all instructions that we have discovered. If we discover an instruction for the first time, put it on our stack. If we discover it a second time, assign a label for it. If the control edge between the instruction we are processing and the instruction we discovered is a jump, we assign the discovered instruction a label.

Linearizing the IR

Industry-strength compilers organize the linearized code in such a way that if statements and short circuit operations only generate forward edges by jumping from the condition to the else block and from the end of the then block to the join block. In our example, we take a simpler, more naive approach. We first emit code for the false branch of jumps, ignoring the true branch. Then after the false-path is done, we emit the code if the true branches.

Consider this small crux program:

```
void main() {         // main
  if (false) {                // L1
    printInt(1);
  } else {              // main
    printInt(0);
  }
  printInt(0-1);        // L2
}
```

This code will generate the following listing (Note that this might not be 100% consistent with your output, because we might have a slightly different IR):

```
    .globl main
main:
    enter $(8 * 6), $0
    /* CopyInst: */              | |
    movq $0, %r10                | |
    movq %r10, -8(%rbp)          | |
    /* Jump: */                  | |
    movq -8(%rbp), %r10          | |                  // if (false)
    cmp $1, %r10                 | |
    je L1                        |  ----+             // jump to then
    /* CopyInst: */              |      |             // else
    movq $0, %r10                |      |
    movq %r10, -16(%rbp)         |      |
    /* CallInst */               |      |
    movq -16(%rbp), %rdi         |      |
    call printInt                |      |             // ::printInt(0)
L2:                              | <----+--+
    /* Nop */                    | |      |
    /* CopyInst: */              | |      |
    movq $0, %r10                | |      |
    movq %r10, -24(%rbp)         | |      |
    /* CopyInst: */              | |      |
    movq $1, %r10                | |      |
    movq %r10, -32(%rbp)         | |      |
    /* BinaryOperator */         | |      |
    movq -24(%rbp), %r10         | |      |
    subq -32(%rbp), %r10         | |      |
    movq %r10, -40(%rbp)         | |      |
    /* CallInst */               | |      |
    movq -40(%rbp), %rdi         | |      |
    call printInt                | |      |   // ::printInt(0-1)
    leave                        | |      |
    ret                          | |      |   // exit
L1:                              | <--+ |
    /* CopyInst: */              |    | |
    movq $1, %r10                |    | |
    movq %r10, -48(%rbp)         |    | |
    /* CallInst */               |    | |
    movq -48(%rbp), %rdi         |    | |
    call printInt                |    | |   // ::printInt(1)
    jmp L2                       |  --+-+
```

As we can see in this example, after the jump, we emit the code of the else branch and then continue with the join (L2). If the condition is fulfilled, we jump into a separate sequence, that contains the code for the true-branch and at the end we jump back to the join label.

While loops look similar: we first skip the body generating first the code after the while loop and then the body of the while loop is appended afterwards.

To implement this, we perform a depth first search on the function. When we pop an instruction from our search-stack we first visit it with the instruction visitor to emit its code. Then we need to decide the order to visit the successors of the current instruction if there is more than one. The successor that is the target of a jump is pushed first onto the stack, followed by the other successor.

Managing the stack

In our compiler we do not perform register allocation. Instead, we will allocate for every variable (including temporaries) a slot on the stack. To do this, we want to store a unique slot number for every variable. So whenever we encounter a variable we haven't seen, we increase a counter for the current function and store the counter with the variable as key in a map.

When we want to use the variable, we can retrieve its slot from the stack and multiply it with -8 (negative eight, because the stack grows downwards in our address space relative to to the current frame pointer `%rbp`). Note that all local variables are integers or booleans, but our language doesn't have any arrays within functions. To access the variable, you can now write `offset(%rbp)`, where offset is that negative, with 8 multiplied number. This notation in assembly means, that we want to address the memory located at `%rbp + offset`.

We also need to make space on the stack for arguments. We allocate one slot on the stack for each argument. The first six arguments will be passed in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`. We want to transfer these at the beginning of a function onto the stack as well. A function taking two parameters will thus have these instructions at the beginning after the prologue:

```
    movq %rdi, -8(%rbp)
    movq %rsi, -16(%rbp)
```

Finally, we need to make space on the stack, if our current function *calls* a function that has more than 6 parameters and thus needs arguments to be passed on the stack. For all call instructions in our IR, we need to find the one, that passes the most arguments on the stack. We need to allocate as many additional slots at the top of the stack.

Prologue and Epilogue

Now that we know the size of the stack, we can generate the prologue and epilogue. We can use two convenient instructions, that do all the work for us: `enter` and `leave`. The `enter` instruction is not commonly used by industrial-strength compilers because it is relatively slow. However, to simplify this project, you may use it. The `enter` instruction pushes `%rbp` to the stack, sets `%rbp = %rsp`, and adds the value of the first operand to `%rsp`. This means, that `%rbp` always points to the beginning of the current stack frame and `%rsp` to top of the stack (which again is the smallest address in our stack, because the stack grows in negative direction).

At the end of the function we have a `leave` instruction, which resets the stack to the state before using `enter` and then `ret`, which pops the return address from the stack and jumps back to the caller.

A function thus looks like this:

```
    .globl main
main:
    enter $(8 * slots), $0
    ...
    leave
    ret
```

The `.globl` directive tells the assembler, that this is a function.

Make sure that your stack is 16-byte aligned, otherwise your program will immediately crash. So if you end up with an uneven number of stack slots, you'll have to allocate one additional slot.

Returning values

To return a value, we need to make sure, that the value we want to return is in register `%rax` and then return from the function:

```
    movq -192(%rbp), %rax
    leave
    ret
```

Addressing

We already now, that we can access local variables relative to the stack. However, x86 generally allows us to specify only one operand to instructions as a memory address. The other one needs to be loaded into a register first. For example if we want to add two values, we move the first value into `%r10` and then we can add it directly with the other value from the stack like this:

```
    movq -328(%rbp), %r10
    addq -336(%rbp), %r10
    movq %r10, -344(%rbp)
```

The addition not only uses `%r10` as operand, but also stores the result in there. Hence we store the result back on to the stack with the third instruction in the listing above.

Global variables

We rarely want to address global variables directly. Specifying the address of a global in a 64-bit architecture takes 8 bytes plus the opcode and the target register. We want to avoid that and thus use program counter (PC) relative addressing. We can have the assembler do most of the work for us here, by writing our address like this: `VarName@GOTPCREL(%rip)` (`VarName` is the same name we used when declaring it in the global variables section).

To load a global variable we can thus write:

```
    movq VarName@GOTPCREL(%rip) , %r11
    movq %r11, -304(%rbp)
```

If we want to load a variable with a given offset (as specified in an `AddressAt` instruction), we can extend this:

```
    movq -360(%rbp), %r11            // Load offset
    movq $8, %r10
    imul %r10, %r11                  // Multiply offset by 8
    movq Data@GOTPCREL(%rip) , %r10   // Load array address
    addq %r10, %r11                  // Add array base address with offset
    movq %r11, -368(%rbp)
```

Running the assembly

To assemble and run the generated code, you should use a Linux machine. If you are running on Windows 10, you can also try using the Windows Subsystem for Linux. Instructions for running the compiler on openlab were given in the Project Overview.

To generate an assembly listing from an input program, run the compiler specifying the input program without any additional flags:

```
mvn exec:java -Dexec.args="path/to/your/crux/file.crx"
```

This will generate a file `a.s` in your current working directory.

To generate an object file from your assembly, use gcc:

```
gcc -c -g file.s -o file.o
```

You will also need to compile the runtime once:

```
gcc -c -g src/runtime/runtime.c -o runtime.o
```

To invoke the linker and produce an executable, use gcc:

```
gcc file.o runtime.o -o file
```

Now you should be able to run your program with:

```
./file
```

If you want to do development on a Mac, you can do that but you must add a `_` before function names. When you turn in code, it must run on openlab and thus must not add the `_`.

Running the debugger

You might find a debugger useful if the generated code does not behave as expected.

You can run the debugger with:

```
gdb ./file
```

You can run the in the debugger with:

```
run
```

You can see see the values of registers with:

```
info registers
```

If the program crashes, you can see a stack trace by using:

```
where
```

You can move up and down the call stack using the `up` and `down` commands, respectively.

You also might find the reg layout in gdb useful. You can change the layout by typing in gdb:

```
layout reg
```

Submission

Submission Details

Grade: 96% (100 pts possible)
Graded Anonymously: no

Comments:
No Comments