CSE 302: Compiler Design — Lab 2

AMD64 Assembly Target

2019-09-19

---

1 INTRODUCTION

In this lab you will modify the compiler you built in lab 1 to retarget AMD64 assembly language instead of the restricted fragment of C. The source language will continue to be the same, BX0, as in lab 1.

This lab will be assessed. It is worth 10% (i.e., 2 points) of your final grade. To get full credit, you must submit the final version of your compiler in 1 week, i.e., strictly before 2019-09-26 10:00:00 CEST. Every additional hour past this deadline incurs a $(20/24)\% = 0.83\%$ penalty.

- You are not constrained to start from the same compiler as in lab 1. Nor are you required to use the same programming language. Feel free to start from any of the sample solutions for lab 1.
- This lab is intended to be worked on individually, not in teams.

2 AMD64 ASSEMBLY LANGUAGE TARGET

The bulk of this lab will be modifying your instruction selection implementation from lab 1 to switch the target language from C to AMD64. This section describes some relevant aspects of this target.

2.1 *Output Template*

CHOICE OF SYNTAX AMD64 actually occurs in several dialects or *syntaxes*. The official documentation for AMD64 (from AMD) [1, 2] or x86-64 (from Intel) [3] uses the so called "Intel syntax", whereas the GNU toolchain (GCC, the GNU assembler, etc.) by default use the "AT&T syntax". Fortunately, the GNU toolchain also supports the Intel syntax as long as you begin your file with the invocation:

```
.intel_syntax noprefix
```

Whether you use the Intel syntax or not is up to you. The course materials will be given in AT&T syntax, since it is the standard syntax used in the GNU toolchain. In any case, whatever syntax you use for instructions, you should always use the GNU assembler ("as"). This keeps the syntax of definitions, sections, comments, etc. uniform.

SECTIONS AND LABELS This lab uses only the .text section, which is specified at the top with:

```
.section .text
```

Recall that the contents of the .text section is read-only and interpreted as a list of instructions. In this lab there is no interesting control structure, so you will write all your generated assembly code in a single sequence. This code will be indexed with the global main symbol, which the linker will use to link the C runtime and build the executable binary. The contents of this main label will be interpreted as a main() function, so the last instructions you generate should be ret to relinquish control back to the C runtime and thence to the operating system, ending the program.

Managing the Stack This `main` function will need to make use of a number of stack slots for spilled temporaries (see lecture slides). Each stack slot in this lab will be 64-bits wide, since the only data type in BX0 is 64-bit (signed) integers. To allocate this stack, we need to modify the RBP and RSP registers at the start, and then restore the RBP register right before exiting from the function.

Template To summarize, therefore, here is a template for an assembly program that might be the result of compiling a source file `template.bx`. The template begins with a `.file` directive (optional) that asserts that the assembly file was generated from a given source file by means of a compiler, instead of being hand-written by a human.

```
    .file "template.bx"
    .section .text
    .globl main
main:
    pushq %rbp              # save the old value of RBP (callee-save)
    movq %rsp, %rbp         # make RSP (stack top) a copy of RBP

    # Now we will allocate 42 stack slots, which (say) is the number
    # of temporaries that template.bx requires. This is only
    # an example -- your compiler will have to tailor this
    subq $336, %rsp         # 336 = 42 * 8 bytes

    ##########################################

    # ---your compiler output here---

    ##########################################

    movq %rbp, %rsp         # restore the old RSP (deallocate temps)
    popq %rbp               # restore the old RBP
    movq $0, %rax           # return code 0 stored in RAX
    retq                    # return to the C runtime, which exits the program
```

2.2 *Temporaries and Registers*

Registers Because BX0 has only 64 bit integers, this lab will only use the full registers. That is, you may use the following registers in your generated code.

RAX, RBX, RCX, RDX, RSI, RSP, RBP, RDI, R8, R9, R10, R11, R12, R13, R14, R15

Of these, RSP and RBP are intended to be used for addressing the stack slots, so make sure not to use them in your own code for other computations. The following six registers are *callee-save* in the C (SysV) calling convention used in Unix: RBX, RBP, R12–R15. This means that the `main` function must guarantee that on exit these registers have the same values they had on entry to the function. Failure to do this will lead to unpredictable behavior up to and including your program crashing after exiting from the `main` function.

Spilling If your compiled code makes use of n temporaries, then you should allocate $8n$ bytes in the stack by subtracting $8n$ from RSP. Then, the slot number $(k+1)$ (for $k \in \{0, 1, \ldots, n-1\}$) will be written $8k$(`%rsp`). That is to say, the stack slots are addressed as `0(%rsp)`, `8(%rsp)`, `16(%rsp)`, etc. These slots have to be allocated before any instructions that use the slots, and deallocated before exiting the function.

This is done by means of the `RBP` register: the initial value of `RSP` is stored (remembered) in `RBP`, then `RSP` is decremented by $8n$ bytes, and eventually the old value of `RBP` is moved back to `RSP`.

Loading and Storing Slots It is recommended to use the register `R11` to load/store these values from/to the stack.

- No instruction implicitly refers to this register.
- This register is not callee-save (i.e., it may be freely overwritten), nor is it used in any `C` calling convention for passing arguments to functions (this will become relevant much later, in lab 4).

If you need to place these values loaded from the stack in a different register such as `RAX`, then you can copy `R11` to `RAX` after the load. Likewise with stores.

2.3 *Important* `AMD64` *Instructions*

This section contains a number of examples of instructions. In most cases the behavior of the instruction is obvious, and you can easily adapt it to your particular scenario.

Setting and Copying Registers and Stack Slots

- `movq $42, %rax` (set `RAX` to 42)
- `movq %rdx, %rax` (copy `RDX` to `RAX`)
- `movq 8(%rsp), %rax` (load stack slot 2 into `RAX`)
- `movq $42, 8(%rsp)` (store 42 into stack slot 2)
- `movq %rdx, 8(%rsp)` (store `RDX` into stack slot 2)

Additive Arithmetic

- `addq $42, %rax` (increment `RAX` by 42)
  `subq $42, %rax` (decrement `RAX` by 42)
- `addq %rdx, %rax` (increment `RAX` by value of `RDX`)
  `subq %rdx, %rax` (decrement `RAX` by value of `RDX`)
- `addq 8(%rsp), %rax` (increment `RAX` by value of stack slot 2)
  `subq 8(%rsp), %rax` (decrement `RAX` by value of stack slot 2)
- `addq $42, 8(%rsp)` (increment stack slot 2 by 42)
  `subq $42, 8(%rsp)` (decrement stack slot 2 by 42)
- `addq %rdx, 8(%rsp)` (increment stack slot 2 by value of `RDX`)
  `subq %rdx, 8(%rsp)` (decrement stack slot 2 by value of `RDX`)
- `negq %rax` (set `RAX` to the negative of its previous value)
  `negq 8(%rsp)` (set stack slot 2 to the negative of its previous contents)

Multiplication, Division, and Modulus These instructions need to represent a 128 bit quantity, which is done as follows: the least significant 64 bits of the quantity are stored in `RAX`, while its most significant 64 bits are stored in `RDX`. In particular, the sign bit is stored in `RDX` alone. This is generally depicted as `RDX:RAX`, even though that is not valid assembly syntax.

- `imulq $42` (multiply the value of `RAX` by `42`, store in `RDX:RAX`)
  `imulq %rcx` (multiply the values of `RAX` and `RCX`, store in `RDX:RAX`)
  `imulq 8(%rsp)` (multiply the values of `RAX` and stack slot 2, store in `RDX:RAX`)
- `idivq $42` (divide the value of `RDX:RAX` by `42`, put quotient in `RAX`, remainder in `RDX`)
  `idivq %rcx` (divide the value of `RDX:RAX` by that of `RCX`, put quotient in `RAX`, remainder in `RDX`)
  `idivq 8(%rsp)`
  (divide the value of `RDX:RAX` by that of stack slot 2, put quotient in `RAX`, remainder in `RDX`)
- `cqo` (sign-extend the value of `RAX` to 128 bits, store in `RDX:RAX`)

Warning: don't use the unsigned instructions `mulq`/`divq` for signed operands!

Bitwise Operations

- `andq $42, %rax` (store the bitwise and of `42` and the value of `RAX` in `RAX`)
  `orq $42, %rax` (store the bitwise or of `42` and the value of `RAX` in `RAX`)
  `xorq $42, %rax` (store the bitwise xor of `42` and the value of `RAX` in `RAX`)
- `andq %rdx, %rax` (store the bitwise and of the values of `RDX` and `RAX` in `RAX`)
  `orq %rdx, %rax` (store the bitwise or of the values of `RDX` and `RAX` in `RAX`)
  `xorq %rdx, %rax` (store the bitwise xor of the values of `RDX` and `RAX` in `RAX`)
- `notq %rax` (set `RAX` to the complement of its previous value)
  `notq 8(%rsp)` (set stack slot 2 to the complement of its previous contents)

In each case `RAX` and `RDX` could be replaced by stack slots as well (but not both at once!).

- `sarq $7, %rax` (shift the value of `RAX` right by `7` and store in `RAX`)
  `salq $7, %rax` (shift the value of `RAX` left by `7` and store in `RAX`)
- `sarq %cl, %rax` (shift the value of `RAX` right by the value of `CL` and store in `RAX`)
  `salq %cl, %rax` (shift the value of `RAX` left by the value of `CL` and store in `RAX`)

In the second form of the shift operators, the first register operand can *only* be `CL`; no other registers or overlays are allowed.

2.4 *Handling* `print`*: the Start of the* `BX0` *Runtime*

The `BX0` Runtime The `print` instruction of `BX0` cannot be easily mapped to assembly instructions. Indeed, we need to make use of a `C` library function in order to achieve the output. Unfortunately, calling the `C` library function `printf()` directly is rather complicated, having to do with the fact that `printf()` can have a variable number of arguments. Let us simplify this by writing a different function called `bx0_print()`, in a file called `bx0rt.c` (standing for "`BX0` runtime").

```c
#include <stdio.h>
#include <stdint.h>

void bx0_print(int64_t x)
{
    printf("%ld\n", x);
}
```

Calling the Runtime Function from AMD64 From your generated `AMD64` assembly code, you will need to use the `callq` instruction to call the `bx0_print()` function. The first argument, `x`, to this function will be supplied in the `RDI` register (per convention). If the temporary that contains the value to be printed is spilled to stack slot 7, then here is how you would write this call in `AMD64`:

```
main:
    ## ... earlier code here

    movq 48(%rsp), %rdi      # load stack slot 7 as first argument, RDI
    callq bx0_print
    # bx0_print() will return to here

    ## rest of the main() function ...
```

Linking the Runtime You must now compile the `AMD64` assembly together and then link it with the `BX0` runtime to obtain an executable progarm. This is fairly easy to do in one GCC invocation:

```
  ## run the BX0 compiler to generate example.s from example.bx
$ ./bx0.exe example.bx

  ## compile and link example.s with the runtime to get example.exe
$ gcc -no-pie -o example.exe example.s bx0_rt.c
```

REFERENCES

[1] AMD. AMD64 Architecture Programmer's Manual Volume 1: Application Programming. Technical Report 24592, Advanced Micro Devices, Dec. 2017. LINK.

[2] AMD. AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions. Technical Report 24594, Advanced Micro Devices, Sept. 2019. LINK.

[3] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Technical Report 325383-060US, Volume 2, Intel Corporation, Sept. 2016. LINK.