x86-64 Assembler

Our compilers will generate assembly-language code, which we will turn into executable binaries with the gnu asssembler.

By convention we will use the .s suffix for an assembly-language file. You assemble file foobar.s into the executable file s with the command

```
gcc foobar.s -o foo
```

We will be using some C library functions for input and output; gcc will automatically link the C libraries to your program.

If you want to see what assembly code gcc generates for a particular piece of C code, put the code in file foo.c and compile it with gcc -S foo.c; the compiler will produce file foo.s

The gnu assembler uses an old AT&T (ie Bell Labs) format for instructions:

instruction source destination

The assembler is case-insensitive; it doesn't matter if you say MOVL or movl.

The instruction names generally end with a length-specifier:

Q -- quadwords (8 bytes or 64 bits)
L -- longwords (4 bytes or 32 bits)
W -- words (2 bytes or 16 bits)
B -- bytes (8 bits)

You will probably not use anything lengths but Q and L.

Addresses in this system are all 64 bits. Integers are 32 bits, though you may find it more convenient to pretend that integers are 64 bits, which means that almost every instruction you generate involves a quadword.

There are 16 64-bit registers. Most allow you to work with only the lower 32-bits, so each register has 2 names: a %r-name for the full 64 bits and an %e-name for the lower 32-bits.

Note that the %-e names were the names of the 32-bit registers in the old x86-32 architecture.

Register	Lower half	Purpose
%rax	%eax	This is general purpose, but many people treat it as an "accumulator", where the results of calculations go. The Intel programming conventions call for return values to be placed in rax before returning.
%rsp	%esp, which you should never use	This is the stack pointer. It points to the top element currently on the stack. The stack grows towards smaller addresses, so a push operation decrements rsp. You can allocate local variables by decrementing rsp yourself, and pop the stack by incrementing rsp.
%rdi	%edi	This is general purpose. The Intel conventions call for the first argument for a function call to be passed in rdi. We will do this only when calling C routines, but you should be careful about trashing this register.
%rsi	%esi	The conventions call for the second argument to be passed in rsi.
%rdx	%edx	The third argument.
%rcx	%ecx	The fourth argument.
%r8	%r8d	The fifth argument.
%r9	%r9d	The sixth argument
%r12	%r12d	This is undesignated, and not used by the C-compiler.
%rbx	%ebx	These registers are designated "callee saved" in the conventions. This means that if you are using them you should save their prior values, and restore those values when you are done with them.
%rbp	%ebp	
%r10	%r10d	
%r13	%r13d	
%r14	%r14d	
%r15	%r15d	
%r11	%r11d	This is used for linking. I would avoid using it.

Addressing modes: We will primarily use 3 simple addressing modes:

Register mode: the operand is the name of a register. For example

```
movq %rsp, %rax
```

which puts the stack pointer into register %rax.

Indirect mode: the operand is the value stored in a memory location, which is specified by an offset from the value in a register. For example

```
movl %eax, 8(%rsp)
```

This moves the 32 bits stored in %eax (the lower half of %rax) to the stack, 8 bytes below the top of the stack (the stack "grows" towards smaller addresses).

Immediate mode: The operand is a number, which is preceeded by a $:

```
movl $23, %eax
```

puts the literal number 23 into %eax.

You will occasionally use a fourth mode for labels:

Direct mode: The operand is specified by a symbol in the program. For example

```
call f
```

Here f must be a label at the start of a function.
For another example

```
jmp  .L0
```

is an unconditional branch to label .L0

Data Instructions

MOVL, MOVQ: move 32 or 64 bits from the source to the destination. For example

```
movl $23, %eax  puts 23 into the 32-bit register %eax
movq %rsp, %rax puts the stack pointer into %rax
```

LEAQ loads the "effective address" of the source into the destination. The data is an address so you want the Q-mode. For example

```
leaq 8(%rsp), %rax
```

puts the address 8 bytes below the top of the stack into %rax. You could just as easily do this as

```
movq %rsp, %rax
addq  8, %rax
```

CLRL, CLRQ loads 0 into the 32-bit or 64-bit destination

clrl %eax puts 0 into %eax

clrq 0(%rsp) changes the top of the stack to 0

PUSH decrements $rsp by 8 bytes and puts the source data at this location.

push $23 pushes the number 23 onto the stack

Note that if you want to have 3 64-bit local variables allocated on the stack, you could do this with

```
push $0
push $0
push $0
```

which initializes them to 0, or with

```
subq $24, %rsp
```

which makes room for them on the stack but doesn't initialize them.

POP puts the 8 bytes at the top of the stack into the destination and increments %rsp by 8.

pop %rax pops the stack into %rax

Before you leave a function call you will need to pop the local environment off the stack. This is usually more easily accomplished by incrementing %rsp than by issuing the right number of pop instructions.

Branches

jmp label is an unconditional jump to the label

There are 7 conditional branches: je, jne, jl, jle, jg, jge, jz. These all act on the value of the *condition codes* in the processor. Most arithmetic instructions set the condition codes, so if you are sure that the previous instruction was an arithmetic operation you might follow it immediately with a conditional branch. In most situations it is safer to do a comparison with CMP and then follow it with the conditional branch. The sequence

```
cmpl $8, %eax
jle L2
```

will branch to label L2 if the value in register %eax is less than or equal to 8. Note that this is backwards of the way most people would read this.

Calls

call F pushes the instruction pointer (the address of the *next* instruction; this is in register %RIP) and branches to label F.

return pops the stack into the instruction pointer register %rip.

Note that both call and return are minimalist instructions. Call does nothing with arguments or with local variables. We will have a set of conventions for calling that I call the "run-time environment". These will say where arguments go, who puts them there and who gets rid of them. Note also that return assumes that anything the called function has put on the stack has been popped off, so that the return address is at the top of the stack when you are ready to return.

Arithmetic

ADDL, ADDQ adds 32-bit or 64-bit data

SUBL, SUBQ subtracts 32-bit or 64-bit data.

Both of these work by adding or subtracting the source and destination values, leaving the result in the destination:

```
addl $8, %eax    increments %eax by 8
```

64-bit multiplication is weird; I at least failed to make sense out of the documentation for it.

IMUL does 32-bit multiplication in ways you expect:

```
imul 0(%rsp), %eax    multiplies the value in %eax by
```

the value at the top of the stack.

The basic division operation is set up to divide 128 bits, stored in two registers, by 64 bits stored in one. If the dividend is negative we need to “sign-extend” it to fill up the rest of its 128 bits with copies of the sign bit (0’s for positive dividends, 1’s for negative dividends). Here is a sequence of steps that makes it work:

a) Put the divisor into ebx.
b) Put the dividend (the number being divided into) into eax.
c) Do a cltq instruction (with no operands) to sign-extend it to all of rax. cltq stands for “convert long to quad” – a “longword”is 32 bits, a quadword” is 64.
d) Do a cqto instruction (with no operands) to sign-extend it to rdx. cqto stands for “convert quadword to octword.”
e) Do a idivl instruction, whose only argument is the divisor ebx. The quotient is put into eax, and the remainder (for a mod or % operation) is put into edx.

For example,

```
movl $7, %ebx
movl $23, %eax
cltq
cqto
idivl %ebx
```

the result leaves the quotient, 3 in eax and the remainder, 2 in edx

Assembler directives

.comm is used to allocate read/write memory. The format for it is

```
.comm  label, bytes-allocated, alignment
```

as in

```
.comm  X, 40, 32
```

We will use this to allocate global variables and arrays.

.section .rodata

This allocates read-only memory, where C expects to find string constants. At a minimum you will define strings in this section for working with the scanf and printf functions for I/O. Here is a section that has two strings, one named .WriteIntString and one named .Bob:

```
.section .rodata
.WriteIntString: .string "%d"
.Bob: .string "Compilers are Fun!"
```

.text

This indicates the start of the text (i.e. code) segment of the program. You typically have the data segments (.comm and .rodata) first, then the text segment.

.globl is used to indicate that a symbol in the program has global scope. We are using gcc to link our program to C-libraries. This will assume that the program has (like all C programs) a main( ) function. The start of your executable program should have a global main label:

```
.globl main
```