Introduction to x64 Assembly

```
.intel_syntax
.text
    .globl  _main
  _main:
    /* function prologue */
    push    rbp
    mov     rbp, rsp
    /* call puts("Hello, World!") */
    lea     rdi, [rip + _main.S_0]
    call    _puts
    /* return zero */
    mov     rax, 0
    mov     rsp, rbp
    pop     rbp
    ret
  _main.end:
  _main.S_0:
    .string "Hello, World!"
```

Figure 1: A simple program in x64 assembler. See also: `http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/x64-intro/hello_world.txt`

Abstract

The name x64 refers to a 64-bit instruction set for Intel and AMD processors, which are commonly found in current-generation laptop and desktop computers. This document introduces a subset of x64, including the features needed for a Compilers course at USI. We write assembler files using "Intel syntax", and we adopt the C calling conventions of Mac OS X.

1 Example: Hello, World!

Figure 1 shows an example program in x64 assembler that prints a greeting to standard output. Before we look at what this does, let's try running it. The steps are as follows:

- Put the code in a file called `main.s`. The file extension `.s` indicates an assembler file.
- Run the assembler and linker. We use `gcc` for this. When the input file to `gcc` is an assembler file, `gcc` skips its C front-end, and just assembles and links the file. We need to provide the option `-m64` to choose x64 assembler. Putting it all together, we get the following command-line:
  `gcc -m64 main.s`
- The previous step produced an executable file `a.out` with the machine code. Run it:
  `./a.out`

While we are on the topic of tooling, you can also use `gcc` to compile a C program to assembly by using the `gcc` command-line option `-S`. This option instructs `gcc` to run the front-end only. That is a useful approach for getting concrete examples of various code sequences. If you specify `-Wall` to enable warnings and `-ansi` to select the C dialect, the following command-line compiles a source program `main.c` to a target program `main.s`:

```
gcc -Wall -ansi -m64 -masm=intel -S main.c
```

2 x64 Syntax

Rather than give a formal grammar for x64, this section describes it using the example in Figure 1. There is one statement per line, which means that changing newlines would change the meaning of the program. Other than that, the syntax is insensitive to whitespace, meaning that additional spaces, tabs, or comments do not affect program behavior. Comments start with `/*` and end with `*/`.

The example program contains two kinds of statements: directives and instructions. Directives start with a period, such as `.intel_syntax`, whereas instructions consist of an operator and a list of operands, such as `mov rbp, rsp`. In addition, a statement can start with labels, which are symbols followed by colon, such as `_main.end:`.

The directive `.intel_syntax` at the start of the file selects Intel syntax. Without that directive, the default is `.att_syntax`. One major difference between the two options is the order of operands: Intel syntax shows the destination operand first, whereas AT&T syntax shows the destination operand last. While the

Dragon book does not show any x64 instructions, it does adopt the destination-first convention for assembler code, so using Intel syntax is less confusing.

The directive `.text` tells the assembler to put the following statements in the text section, which contains executable code. The directive `.string "Hello, World!"` copies the characters in the string to the binary file, and ends it with a 0-byte. The string constant may contain escapes, such as `\n` for newline.

The directive `.globl _main` makes the label `_main` visible to the linker so it can be called from outside.

To summarize, we start each file with `.intel_syntax`, followed by `.text` directive declaring the text section containing code for the functions and read-only data, such as strings. We won't need to declare any mutable data in the assembly file, as we will use dynamically allocated memory to represent Tack values. Function should have a `.globl` directive matching the function's start label for it to be called from outside (for our purposes it suffice to only make `_main` global as our simple compiler only supports single-file programs). Please note, that C compiler adds underscore prefix to function names, so all C library function names are prefixed and C run-time system is expecting your `main` to have this prefix as well. To learn more about `as` (the GNU assembler), see the user manual:
`http://sourceware.org/binutils/docs-2.21/as`

3 Addresses

As mentioned before, an instruction consists of zero or more labels, an operator, and zero or more operands. We refer to operands as "addresses", even when they are non-pointer values. We use the following kinds of addresses:

- Registers (r). There are sixteen 64-bit general-purpose registers: `rax` to `rdx`, `rsp`, `rbp`, `rsi`, `rdi`, and `r8` to `r15`. However, some of these registers play a special role, for example, `rsp` and `rbp` typically hold the stack pointer and base pointer, as their names imply.
- Immediate operands (i). These are either integer constants or labels.
  - Integer constants are written as either the digit `0`, or a digit from `1-9` followed by zero or more digits from `0-9`.
  - Label operands come in two forms. On the one hand, control transfer instructions, such as `jmp` or `call`, use code labels, which are simply the symbol name, such as `main`. On the other hand, to get the address of data stored in the text section we can use `lea` (Load Effective Address) instruction where the label serves as an offset relative to the current execution address (stored in `rip` register), like `lea dx, [rip + _main.S_0]` in Figure 1.
- Memory operands (m). These addresses add a constant to a register, and then dereference the resulting pointer. For example, `qword ptr [rax+8]` takes the value of register `rax`, adds `8`, interprets the result as a pointer, and dereferences it. The offset can also be negative, such as `qword ptr [rsp-16]`, or it can be omitted when zero, such as `qword ptr [rbp]`.

Not every instruction accepts every kind of address. The abbreviations r, i, and m serve to indicate which addressing modes are supported. To learn more about addresses, see Volume 1 of the "Intel 64 and IA-32 Architectures Software Developer's Manual": `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html/`

4 Instructions

The following reference lists instructions in alphabetical order. When an instruction has multiple addressing modes, the alternatives are separated by a vertical bar |. As a general rule of thumb, most instructions support only one memory operand (m), not two. Typically, the first operand is a destination operand, in other words, many instructions store their result in the first operand.

add → `add` r, i | `add` r, r | `add` r, m | `add` m, i | `add` m, r
Compute the sum of the two operands, and store the result in the first operand.

call → `call` *label*
Store the return address into `[rsp]`. Subtract 8 from `rsp`. Jump to the *label.*

cmp → `cmp` r, i | `cmp` r, r | `cmp` r, m | `cmp` m, i | `cmp` m, r
Compare the two operands. Encode the result in status flags in an internal register, which can then be

used for the various conditional jump instructions: `je`, `jg`, `jge`, `jl`, `jle`, and `jne`.

idiv $\rightarrow$ `idiv` r | `idiv` m

Treat `edx:eax` as a single, signed 128-bit integer value. Divide this value by the operand. Store the rounded-down quotient in `rax`, and the remainder in `%rdx`. A common idiom to prepare `edx` for this instruction is to first do `mov rdx, rax; sar rdx, 63`, which fills `rdx` entirely with the appropriate sign bit.

imul $\rightarrow$ `imul` r, r | `imul` r, m

Compute the product of the two signed integer operands, and store the result in the first operand.

jmp $\rightarrow$ `jmp` *label*

Jump unconditionally to *label*.

je $\rightarrow$ `je` *label*

Jump to *label* if the first operand of the preceding `cmp` instruction was equal to the second operand.

jg $\rightarrow$ `jg` *label*

Jump to *label* if the first operand of the preceding `cmp` instruction was $>$ the second operand.

jge $\rightarrow$ `jge` *label*

Jump to *label* if the first operand of the preceding `cmp` instruction was $\geq$ the second operand.

jl $\rightarrow$ `jl` *label*

Jump to *label* if the first operand of the preceding `cmp` instruction was $<$ the second operand.

jle $\rightarrow$ `jle` *label*

Jump to *label* if the first operand of the preceding `cmp` instruction was $\leq$ the second operand.

jne $\rightarrow$ `jne` *label*

Jump to *label* if the first operand of the preceding `cmp` instruction was $\neq$ the second operand.

mov $\rightarrow$ `mov` r, i | `mov` r, r | `mov` r, m | `mov` m, i | `mov` m, r

Copy the value of the second operand to to the first operand.

neg $\rightarrow$ `neg` r | `neg` m

Replace the operand with its two's complement negation, i.e., signed integer minus.

pop $\rightarrow$ `pop` r | `pop` m

Copy the value from `[rsp]` to the operand, then add 8 to `rsp`.

push $\rightarrow$ `push` i | `push` r | `push` m

Copy the operand value to `[rsp]`, then subtract 8 from `rsp`.

ret $\rightarrow$ `ret`

Retrieve the return address from `[rsp]`. Add 8 to `rsp`. Jump to the return address.

shl $\rightarrow$ `shl` r, i | `shl` m, i

Perform a left-shift on the first operand, with the amount given by the second operand. A left-shift fills in with zero bits.

sar $\rightarrow$ `sar` r, i | `sar` m, i

Perform an arithmetic right-shift on the first operand, with the amount given by the second operand. An arithmetic right-shift preserves the sign, by filling in with the left-most (sign) bit.

shr $\rightarrow$ `shr` r, i | `shr` m, i

Perform a logical right-shift on the first operand, with the amount given by the second operand. A logical right-shift ignores the sign, by filling in with zero bits.

sub $\rightarrow$ `sub` r, i | `sub` r, r | `sub` r, m | `sub` m, i | `sub` m, r

Subtract the second operand from the first operand, and store the result in the first operand.

lea $\rightarrow$ `lea` r, m

Load the address specified by memory operand and store it into the register.

The x64 instruction set has many more instructions than shown here. Furthermore, most of the instructions support more addressing modes than listed. The reference here should suffice for our compiler construction project, but if you want to learn more, see Volume 2 of the "Intel 64 and IA-32 Architectures Software Developer's Manual": `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html/`

5 Calling Conventions

We adopt the same calling conventions as for the C programming language, because that enables us to call external functions defined in the runtime, such as, a `print` function. In the assembly code of the caller, the calling sequence is the same, irrespective of whether the callee is written in assembly or compiled from C. In fact, Figure 1 shows a call from assembly to the C function `puts`. Conversely, C code can call assembly functions. In fact, `main` gets invoked from "outside".

The stack grows "down", which means that new slots are added at the bottom, and older slots reside at higher addresses. This is reflected in some of the instructions we saw in the previous section (`push`, `pop`, `call`, and `ret`).

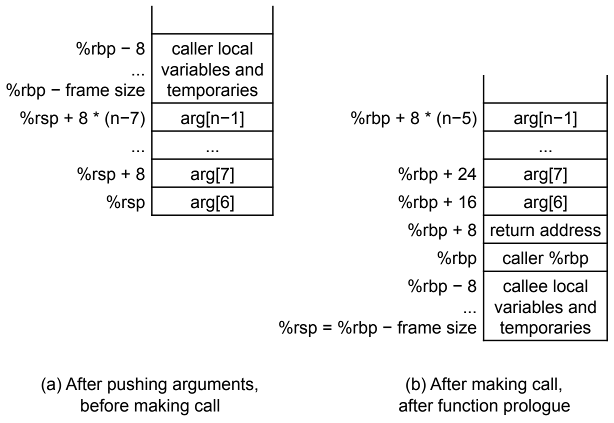

Figure 2: Stack layout for C calling conventions on x64/Mac OS X.

Figure 2 shows the stack layout, both before and after a function call. If the function has more than 6 arguments, then arguments $0 \ldots 5$ get passed in registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`, and arguments $6 \ldots n-1$ get passed on the stack. If the function has at most 6 arguments, all arguments get passed in registers. Just before the caller executes the `call` instruction, the stack layout is as shown in Figure 2(a), with register `rsp` (the stack pointer) pointing to the lowest argument on the stack.

The call instruction pushes the return address on the stack. Then, the callee is responsible for pushing the old value of `rbp` (the base pointer) on the stack, and setting the new value of `rbp` to point to the location of the old value. After that, the callee can use further stack space for its own local variables and temporaries. In that case, `rbp` remains unchanged, pointing to the base of the stack frame, whereas `rsp` points to the end of the stack frame, as shown in Figure 2(b). The following pseudo-code shows the callee's prologue sequence:

```
push rbp
move rbp, rsp
sub  rsp, /*frame size*/
```

Please note that Mac OS X dynamic linker requires stack pointer (`rsp` register) to be 16-byte aligned. To achieve this, you have to make sure that the allocated frame size is always a multiple of 16. Also, you have to take care about this alignment while allocating space for arguments passed via stack.

A non-void function returns its result through register `rax`. Other than that, the function epilogue resets `rsp` back to the start of the frame, pops the old value of `rbp`, and uses the `ret` instruction to pop the return address and jump back to the caller. The following pseudo-code shows the callee's return sequence:

```
move rax, /*return value*/
move rsp, rbp
pop  rbp
ret
```

One issue we have not yet discussed is caller-save registers and callee-save registers. As the name implies, the caller must save values of caller-save registers before it makes the call, as they may be lost when the callee overwrites them. In other words, caller-save registers "belong to" the callee. On the other hand, the callee must save values of callee-save registers in the prologue sequence and restore them in the epilogue sequence, as the caller may expect that their value after the return is the same as before the call. In other words, callee-save registers "belong to" the caller.

In the C calling conventions for x64/Mac OS X, registers `rbp`, `rbx`, and `r12` thru `r15` belong to the caller (are callee-save registers), and all remaining registers belong to the callee (are caller-save registers). However, it is often not necessary to save and restore registers, since they may not hold live values. For example, consider the caller-save register `rdx`. If the caller does not keep a value in `rdx` across a call, it does not need to save and restore `rdx`.

Calling conventions are often part of the so-called ABI, which stands for "application binary interface". The calling conventions described here are only a subset of the C ABI for x64/Mac OS X: we only discuss values of size 8 bytes that can be stored in a single register or stack slot, and we only discuss general-purpose registers, no floating point or vector registers. If you want to learn more about the full-fledged ABI, you can use the following document: `http://x86-64.org/documentation/abi.pdf`