x64 Cheat Sheet

Fall 2014

1 x64 Registers

x64 assembly code uses sixteen 64-bit registers. Additionally, the lower bytes of some of these registers may be accessed independently as 32-, 16- or 8-bit registers. The register names are as follows:

8-byte register	Bytes 5-8	Bytes 7-8	Byte 8
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

For more details of register usage, see Register Usage, below.

2 Operand Specifiers

The basic types of operand specifiers are below. In the following table,

- Imm refers to a constant value, e.g. `0x8048d8e` or `48`,
- E_x refers to a register, e.g. `%rax`,
- `R[`E_x`]` refers to the value stored in register E_x, and
- `M[`x`]` refers to the value stored at memory address x.

Type	Form	Operand value	Name
Immediate	\$$Imm$	Imm	Immediate
Register	E_a	R[E_a]	Register
Memory	Imm	M[Imm]	Absolute
Memory	(E_a)	M[R[E_b]]	Absolute
Memory	Imm(E_b, E_i, s)	M[Imm+R[E_b]+(R[E_i]$\times s$)]	Scaled indexed

More information about operand specifiers can be found on pages 169-170 of the textbook.

3 x64 Instructions

In the following tables,

- "byte" refers to a one-byte integer (suffix `b`),
- "word" refers to a two-byte integer (suffix `w`),
- "doubleword" refers to a four-byte integer (suffix `l`), and
- "quadword" refers to an eight-byte value (suffix `q`).

Most instructions, like `mov`, use a suffix to show how large the operands are going to be. For example, moving a quadword from `%rax` to `%rbx` results in the instruction `movq %rax, %rbx`. Some instructions, like `ret`, do not use suffixes because there is no need. Others, such as `movs` and `movz` will use two suffixes, as they convert operands of the type of the first suffix to that of the second. Thus, assembly to convert the byte in `%al` to a doubleword in `%ebx` with zero-extension would be `movzbl %al, %ebx`.

In the tables below, instructions have one suffix unless otherwise stated.

3.1 Data Movement

Instruction		Description	Page #
		Instructions with one suffix	
`mov`	S, D	Move source to destination	171
`push`	S	Push source onto stack	171
`pop`	D	Pop top of stack into destination	171
		Instructions with two suffixes	
`movs`	S, D	Move byte to word (sign extended)	171
`movz`	S, D	Move byte to word (zero extended)	171
		Instructions with no suffixes	
`cwtl`		Convert word in `%ax` to doubleword in `%dx:%ax`	182
`cltq`		Convert doubleword in `%eax` to quadword in `%edx:%eax`	182
`cqto`		Convert octoword in `%rax` to octoword in `%rdx:%rax`	182

3.2 Arithmetic Operations

Unless otherwise specified, all arithmetic operation instructions have one suffix.

3.2.1 Unary Operations

Instruction	Description	Page #
`inc` D	Increment by 1	178
`dec` D	Decrement by 1	178
`neg` D	Arithmetic negation	178
`not` D	Bitwise complement	178

3.2.2 Binary Operations

Instruction	Description	Page #
`leaq` S, D	Load effective address of source into destination	177
`add` S, D	Add source to destination	178
`sub` S, D	Subtract source from destination	178
`imul` S, D	Multiply destination by source	178
`xor` S, D	Bitwise XOR destination by source	178
`or` S, D	Bitwise OR destination by source	178
`and` S, D	Bitwise AND destination by source	178

3.2.3 Shift Operations

Instruction	Description	Page #
`sal` / `shl` k, D	Left shift destination by k bits	179
`sar` k, D	Arithmetic right shift destination by k bits	179
`shr` k, D	Logical right shift destination by k bits	179

3.2.4 Special Arithmetic Operations

Suffixes are already included in these instructions.

Instruction	Description	Page #
`imulq` S	Signed full multiply of `%rax` by S Result stored in `%rdx:%rax`	182
`mulq` S	Unsigned full multiply of `%rax` by S Result stored in `%rdx:%rax`	182
`idivq` S	Signed divide `%rdx:%rax` by S Quotient stored in `%rax` Remainder stored in `%rdx`	182
`divq` S	Unsigned divide `%rdx:%rax` by S Quotient stored in `%rax` Remainder stored in `%rdx`	182

3.3 Comparison and Test Instructions

Comparison instructions also have one suffix.

Instruction		Description	Page #
`cmp`	S_2, S_1	Set condition codes according to $S_1 - S_2$	185
`test`	S_2, S_1	Set condition codes according to $S_1 \& S_2$	185

3.4 Accessing Condition Codes

None of the following instructions have any suffixes.

3.4.1 Conditional Set Instructions

Instruction		Description	Condition Code	Page #
`sete` / `setz`	D	Set if equal/zero	`ZF`	187
`setne` / `setnz`	D	Set if not equal/nonzero	`~ZF`	187
`sets`	D	Set if negative	`SF`	187
`setns`	D	Set if nonnegative	`~SF`	187
`setg` / `setnle`	D	Set if greater (signed)	`~(SF^OF)&~ZF`	187
`setge` / `setnl`	D	Set if greater or equal(signed)	`~(SF^OF)`	187
`setl` / `setnge`	D	Set if less (signed)	`SF^OF`	187
`setle` / `setng`	D	Set if less or equal	`(SF^OF)|ZF`	187
`seta` / `setnbe`	D	Set if above (unsigned)	`~CF&~ZF`	187
`setae` / `setnb`	D	Set if above or equal (unsigned)	`~CF`	187
`setb` / `setnae`	D	Set if below (unsigned)	`CF`	187
`setbe` / `setna`	D	Set if below or equal (unsigned)	`CF|ZF`	187

3.4.2 Jump Instructions

Instruction		Description	Condition Code	Page #
`jmp`	*Label*	Jump to label		189
`jmp`	**Operand*	Jump to specified location		189
`je` / `jz`	*Label*	Jump if equal/zero	`ZF`	189
`jne` / `jnz`	*Label*	Jump if not equal/nonzero	`~ZF`	189
`js`	*Label*	Jump if negative	`SF`	189
`jns`	*Label*	Jump if nonnegative	`~SF`	189
`jg` / `jnle`	*Label*	Jump if greater (signed)	`~(SF^OF)&~ZF`	189
`jge` / `jnl`	*Label*	Jump if greater or equal(signed)	`~(SF^OF)`	189
`jl` / `jnge`	*Label*	Jump if less (signed)	`SF^OF`	189
`jle` / `jng`	*Label*	Jump if less or equal	`(SF^OF)|ZF`	189
`ja` / `jnbe`	*Label*	Jump if above (unsigned)	`~CF&~ZF`	189
`jae` / `jnb`	*Label*	Jump if above or equal (unsigned)	`~CF`	189
`jb` / `jnae`	*Label*	Jump if below (unsigned)	`CF`	189
`jbe` / `jna`	*Label*	Jump if below or equal (unsigned)	`CF|ZF`	189

3.4.3 Conditional Move Instructions

Conditional move instructions do not have any suffixes, but their source and destination operands must have the same size.

Instruction		Description	Condition Code	Page #
`cmove` / `cmovz`	S, D	Move if equal/zero	`ZF`	206
`cmovne` / `cmovnz`	S, D	Move if not equal/nonzero	`~ZF`	206
`cmovs`	S, D	Move if negative	`SF`	206
`cmovns`	S, D	Move if nonnegative	`~SF`	206
`cmovg` / `cmovnle`	S, D	Move if greater (signed)	`~(SF^OF)&~ZF`	206
`cmovge` / `cmovnl`	S, D	Move if greater or equal(signed)	`~(SF^OF)`	206
`cmovl` / `cmovnge`	S, D	Move if less (signed)	`SF^OF`	206
`cmovle` / `cmovng`	S, D	Move if less or equal	`(SF^OF)|ZF`	206
`cmova` / `cmovnbe`	S, D	Move if above (unsigned)	`~CF&~ZF`	206
`cmovae` / `cmovnb`	S, D	Move if above or equal (unsigned)	`~CF`	206
`cmovb` / `cmovnae`	S, D	Move if below (unsigned)	`CF`	206
`cmovbe` / `cmovna`	S, D	Move if below or equal (unsigned)	`CF|ZF`	206

3.5 Procedure Call Instructions

Procedure call instructions do not have any suffixes.

Instruction		Description	Page #
`call`	*Label*	Push return address and jump to label	221
`call`	**Operand*	Push return address and jump to specified location	221
`leave`		Set `%rsp` to `%rbp`, then pop top of stack into `%rbp`	221
`ret`		Pop return address from stack and jump there	221

4 Coding Practices

4.1 Commenting

Each function you write should have a comment at the beginning describing what the function does and any arguments it accepts. In addition, we strongly recommend putting comments alongside your assembly code stating what each set of instructions does in pseudocode or some higher-level language. Line breaks are also helpful to group statements into logical blocks for improved readability.

4.2 Arrays

Arrays are stored in memory as contiguous blocks of data. Typically an array variable acts as a pointer to the first element of the array in memory. To access a given array element, the index value is multiplied by the element size and added to the array pointer. For instance, if `arr` is an array of `int`s, the statement:

```
arr[i] = 3;
```

can be expressed in x86-64 as follows (assuming the address of `arr` is stored in `%rax` and the index `i` is stored in `%rcx`):

```
movq $3, (%rax, %rcx, 8)
```

More information about arrays can be found on pages 232-241 of the textbook.

4.3 Register Usage

There are sixteen 64-bit registers in x86-64: `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rdi`, `%rsi`, `%rbp`, `%rsp`, and `%r8-r15`. Of these, `%rax`, `%rcx`, `%rdx`, `%rdi`, `%rsi`, `%rsp`, and `%r8-r11` are considered caller-save registers, meaning that they are not necessarily saved across function calls. By convention, `%rax` is used to store a function's return value, if it exists and is no more than 64 bits long. (Larger return types like structs are returned using the stack.) Registers `%rbx`, `%rbp`, and `%r12-r15` are callee-save registers, meaning that they are saved across function calls. Register `%rsp` is used as the *stack pointer*, a pointer to the topmost element in the stack.

Additionally, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` are used to pass the first six integer or pointer parameters to called functions. Additional parameters (or large parameters such as structs passed by value) are passed on the stack.

In 32-bit x86, the *base pointer* (formerly `%ebp`, now `%rbp`) was used to keep track of the base of the current stack frame, and a called function would save the base pointer of its caller prior to updating the base pointer to its own stack frame. With the advent of the 64-bit architecture, this has been mostly eliminated, save for a few special cases when the compiler cannot determine ahead of time how much stack space needs to be allocated for a particular function (see Dynamic stack allocation).

4.4 Stack Organization and Function Calls

4.4.1 Calling a Function

To call a function, the program should place the first six integer or pointer parameters in the registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`; subsequent parameters (or parameters larger than 64 bits) should be pushed onto the stack, with the first argument topmost. The program should then execute the call instruction, which will push the return address onto the stack and jump to the start of the specified function.

Example:

```
# Call foo(1, 15)
movq   $1, %rdi   # Move 1 into %rdi
movq   $15, %rsi    # Move 15 into %rsi
call   foo         # Push return address and jump to label foo
```

If the function has a return value, it will be stored in `%rax` after the function call.

4.4.2 Writing a Function

An x64 program uses a region of memory called the *stack* to support function calls. As the name suggests, this region is organized as a stack data structure with the "top" of the stack growing towards lower memory addresses. For each function call, new space is created on the stack to store local variables and other data. This is known as a *stack frame*. To accomplish this, you will need to write some code at the beginning and end of each function to create and destroy the stack frame.

Setting Up: When a `call` instruction is executed, the address of the following instruction is pushed onto the stack as the return address and control passes to the specified function.

If the function is going to use any of the callee-save registers (`%rbx`, `%rbp`, or `%r12-r15`), the current value of each should be pushed onto the stack to be restored at the end. For example:

```
pushq     %rbx
pushq     %r12
pushq     %r13
```

Finally, additional space may be allocated on the stack for local variables. While it is possible to make space on the stack as needed in a function body, it is generally more efficient to allocate this space all at once at the beginning of the function. This can be accomplished using the call `subq $`N`, %rsp` where N is the size of the callee's stack frame. For example:

```
subq      $0x18, %rsp          # Allocate 24 bytes of space on the stack
```

This set-up is called the *function prologue*.

Using the Stack Frame: Once you have set up the stack frame, you can use it to store and access local variables:

- Arguments which cannot fit in registers (e.g. structs) will be pushed onto the stack before the call instruction, and can be accessed relative to `%rsp`. Keep in mind that you will need to take the size of the stack frame into account when referencing arguments in this manner.
- If the function has more than six integer or pointer arguments, these will be pushed onto the stack as well.
- For any stack arguments, the lower-numbered arguments will be closer to the stack pointer. That is, arguments are pushed on in right-to-left order when applicable.
- Local variables will be stored in the space allocated in the function prologue, when some amount is subtracted from `%rsp`. The organization of these is up to the programmer.

Cleaning Up: After the body of the function is finished and the return value (if any) is placed in `%rax`, the function must return control to the caller, putting the stack back in the state in which it was called with. First, the callee frees the stack space it allocated by adding the same amount to the stack pointer:

```
     addq      $0x18, %rsp         # Give back 24 bytes of stack space
```

Then, it pops off the registers it saved earlier

```
     popq      %r13                # Remember that the stack is FILO!
     popq      %r12
     popq      %rbx
```

Finally, the program should return to the call site, using the `ret` instruction:

```
     ret
```

Summary: Putting it together, the code for a function should look like this:

```
foo:
     pushq     %rbx                # Save registers, if needed
     pushq     %r12
     pushq     %r13
     subq      $0x18, %rsp         # Allocate stack space

     # Function body

     addq      $0x18, %rsp         # Deallocate stack space
     popq      %r13                # Restore registers
     popq      %r12
     popq      %rbx
     ret                           # Pop return address and return control to caller
```

4.4.3 Dynamic stack allocation

You may find that having a static amount of stack space for your function does not quite cut it. In this case, we will need to borrow a tradition from 32-bit x86 and save the base of the stack frame into the base pointer register. Since `%rbp` is a callee-save register, it needs to be saved before you change it. Therefore, the function prologue will now be prefixed with:

```
pushq    %rbp
movq     %rsp, %rbp
```

Consequently, the epilogue will contain this right before the `ret`:

```
movq     %rbp, %rsp
popq     %rbp
```

This can also be done with a single instruction, called `leave`. The epilogue makes sure that no matter what you do to the stack pointer in the function body, you will always return it to the right place when you return. Note that this means you no longer need to add to the stack pointer in the epilogue.

This is an example of a function which allocates between 8-248 bytes of random stack space during its execution:

```
pushq    %rbp               # Use base pointer
movq     %rsp, %rbp
pushq    %rbx               # Save registers
pushq    %r12
subq     $0x18, %rsp        # Allocate some stack space

...

call     rand               # Get random number
andq     $0xF8, %rax        # Make sure the value is 8-248 bytes and
                            # aligned on 8 bytes
subq     %rax, %rsp         # Allocate space

...

movq     (%rbp), %r12       # Restore registers from base of frame
movq     0x8(%rbp), %rbx
movq     %rbp, %rsp         # Reset stack pointer and restore base pointer
popq     %rbp
ret
```

This sort of behavior can be accessed from C code by calling pseudo-functions like `alloca`, which allocates stack space according to its argument.

More information about the stack frame and function calls can be found on pages 219-232 of the textbook.