Guide to x86-64

A CS107 joint staff effort (Erik, Julie, Nate)

x86-64 (also known as just x64 and/or AMD64) is the 64-bit version of the x86/IA32 instruction set. Below is our overview of its features that are relevant to CS107. There is more extensive coverage on these topics in Chapter 3 of the B&O textbook. See also our x86-64 sheet (/class/cs107/onepage_x86-64.pdf) for a compact one-page reference.

Registers

The table below lists the commonly used registers (sixteen general-purpose plus two special). Each register is 64 bits wide; the lower 32-, 16- and 8-bit portions are selectable by a pseudo-register name. Some registers are designated for a certain purpose, such as `%rsp` being used as the stack pointer or `%rax` for the return value from a function. Other registers are all-purpose, but have a conventional use depending on whether **caller**-saved or **callee**-saved. If the function `binky` calls `winky`, we refer to `binky` as the *caller* and `winky` as the *callee*. For example, the registers used for the first 6 arguments and return value are all caller-saved. The callee can freely use those registers, overwriting existing values without taking any precautions. If `%rax` holds a value the caller wants to retain, the caller must copy the value to a "safe" location before making a call. The caller-saved registers are ideal for scratch/temporary use by the callee. In contrast, if the callee intends to use a callee-saved register, it must first preserve its value and restore it before exiting the call. The callee-saved registers are used for local state of the caller that needs to preserved across further function calls.

Register	Conventional use	Low 32-bits	Low 16-bits	Low 8-bits
`%rax`	Return value, caller-saved	`%eax`	`%ax`	`%al`
`%rdi`	1st argument, caller-saved	`%edi`	`%di`	`%dil`
`%rsi`	2nd argument, caller-saved	`%esi`	`%si`	`%sil`
`%rdx`	3rd argument, caller-saved	`%edx`	`%dx`	`%dl`
`%rcx`	4th argument, caller-saved	`%ecx`	`%cx`	`%cl`
`%r8`	5th argument, caller-saved	`%r8d`	`%r8w`	`%r8b`
`%r9`	6th argument, caller-saved	`%r9d`	`%r9w`	`%r9b`
`%r10`	Scratch/temporary, caller-saved	`%r10d`	`%r10w`	`%r10b`
`%r11`	Scratch/temporary, caller-saved	`%r11d`	`%r11w`	`%r11b`
`%rsp`	Stack pointer, callee-saved	`%esp`	`%sp`	`%spl`
`%rbx`	Local variable, callee-saved	`%ebx`	`%bx`	`%bl`
`%rbp`	Local variable, callee-saved	`%ebp`	`%bp`	`%bpl`
`%r12`	Local variable, callee-saved	`%r12d`	`%r12w`	`%r12b`
`%r13`	Local variable, callee-saved	`%r13d`	`%r13w`	`%r13b`
`%r14`	Local variable, callee-saved	`%r14d`	`%r14w`	`%r14b`
`%r15`	Local variable, callee-saved	`%r15d`	`%r15w`	`%r15b`
`%rip`	Instruction pointer			
`%rflags`	Status/condition code bits			

Addressing modes

True to its CISC nature, x86-64 supports a variety of addressing modes. An *addressing mode* is an expression that calculates an address in memory to be read/written to. These expressions are used as the source or destination for a `mov` instruction and other instructions that access memory. The code below demonstrates how to write the immediate value 0 to various memory locations in an example of each of the available addressing modes:

```
movl $0, 0x604892            # direct (address is constant value)
movl $0, (%rax)              # indirect (address is in register %rax)

movl $0, -24(%rbp)           # indirect with displacement (address = base %rbp + displacement -24)

movl $0, 0xc(%rsp, %rdi, 4)  # indirect with displacement and scaled-index
                              (address = base %rsp + displacement 0xc + index %rdi * scale 4)

movl $0, (%rax, %rcx, 8)     # (special case of scaled-index, displacement assumed 0)

movl $0, 0x8(, %rdx, 4)     # (special case of scaled-index, base assumed 0)

movl $0, 0x4(%rax, %rcx)      # (special case of scaled-index, scale assumed 1)
```

Common instructions

A note about instruction suffixes: many instructions have a suffix (`b` , `w` , `l` , or `q`) which indicates the bitwidth of the operation (1, 2, 4, or 8 bytes, respectively). The suffix is often elided when the bitwidth can be determined from the operands. For example, if the destination register is `%eax` , it must be 4 bytes, if `%ax` it must be 2 bytes, and `%al` would be 1 byte. A few instructions such as `movs` and `movz` have two suffixes: the first is for the source operand, the second for the destination. For example, `movzbl` moves a 1-byte source value to a 4-byte destination.

Mov and lea

By far most frequent instruction you'll encounter is `mov` in one of its its multi-faceted variants. Mov copies a value from source to destination. The source can be an immediate value, a register, or a memory location (expressed using one of the addressing mode expressions from above). The destination is either a register or a memory location. At most one of source or destination can be memory. The `mov` suffix (b, w, l, or q) indicates how many bytes are being copied (1, 2, 4, or 8 respectively). For the `lea` (load effective address) instruction, the source operand is a memory location (using an addressing mode from above) and it copies the calculated source *address* to destination. Note that `lea` does not dereference the source address, it simply calculates its location. This means `lea` is nothing more than an arithmetic operation and commonly used to calculate the value of simple linear combinations that have nothing to do with memory locations!

```
mov src, dst                 # general form of instruction dst = src
mov $0, %eax                 # register %eax = 0
movb %al, 0x409892           # write to memory address 0x409892 = low-byte from register %eax
mov 8(%rsp), %eax            # register %eax = value read from memory address %rsp + 8

lea 0x20(%rsp), %rdi         # register %rdi = %rsp + 0x20 (address, no dereference!)
lea (%rdi,%rdi,1), %rax      # register %rax = %rdi + %rdi (looks suspiciously like lea being used for ordinary math)
```

The `mov` instruction copies the same number of bytes from one location to another. In situations where the move is copying a smaller bitwidth to a larger, the `movs` and `movz` variants are used to specify how to fill the additional bytes, either sign-extend or zero-fill.

```
movsbl %al, %edx             # copy low-byte from register %eax, sign-extend to 4 byte long in %edx
movzbl %al, %edx             # copy low-byte from register %eax, zero-extend to 4 byte long in %edx
```

A special case to note is that a **mov** to write a 32-bit value into a register also zeroes the upper 32 bits of the register by default, i.e does an implicit zero-extend to bitwidth q. This explains use of instructions such as **mov %ebx, %ebx** that look odd/redundant, but are, in fact, being used to zero-extend from 32 to 64. Given this default behavior, there is no need for an explicit **movzlq** instruction. To instead sign-extend from 32-bit to 64-bit, there is an **movslq** instruction.

The **cltq** instruction is a specialized **movs** that operates on **%rax**. This no-operand instruction does sign-extension in-place on **%rax**; source bitwidth is l, destination bitwidth is q.

```
cltq                        # operates on %rax, sign-extend 4-byte src to 8-byte dst, shorthand for movslq %eax,%rax.
```

Arithmetic and bitwise operations

The binary operations are generally expressed in a two-operand form where the second operand is both a source to the operation and the destination. The source can be an immediate constant, register, or memory location. The destination must be either register or memory. At most one of source or destination can be memory. The unary operations have one operand which is both source and destination, which can be either register or memory. Many of the arithmetic instructions are used for both signed and unsigned types, i.e. there is not a signed add and unsigned add, the same instruction is used for both. Where needed, the condition codes set by the operation can be used to detect the different kinds of overflow.

```
add src, dst                # dst += src
sub src, dst                # dst -= src
imul src, dst               # dst *= src
neg dst                     # dst = -dst (arithmetic inverse)

and src, dst                # dst &= src
or src, dst                 # dst |= src
xor src, dst                # dst ^= src
not dst                     # dst = ~dst (bitwise inverse)

shl count, dst              # dst <<= count (left shift dst by count positions), synonym sal
sar count, dst              # dst >>= count (arithmetic right shift dst by count positions)
shr count, dst              # dst >>= count (logical right shift dst by count positions)

# some instructions have special-case variants with different number of operands
imul src                    # single operand imul assumes other operand in %rax, computes 128-bit result and stores
                            # high 64-bits in %rdx, low 64-bits in %rax
shl dst                     # dst <<= 1     (no count => assume 1, same for sar, shr, sal)
```

Branching instructions

The special **%flags** register stores a set of boolean flags called the *condition codes*. Most arithmetic operations update those codes. A conditional jump reads the condition codes to determine whether to take the branch or not. The condition codes include ZF (zero flag), SF (sign flag), OF (overflow flag, signed), and CF (carry flag, unsigned). For example, if the result was zero, the ZF is set, if a operation overflowed (into sign bit), OF is set.

The general pattern for all branches is to execute a **cmp** or **test** operation to set the flags followed by a jump instruction variant that reads the flags to determine whether to take the branch or continue on. The operands to a **cmp** or **test** are immediate, register, or memory location (with at most one memory operand). There are 32 variants of conditional jump, several of which are synonyms as noted below.

```
cmpl op2, op1    # computes result = op1 - op2, discards result, sets condition codes
test op2, op1    # computes result = op1 & op2, discards result, sets condition codes

jmp target       # unconditional jump
je  target       # jump equal, synonym jz jump zero (ZF=1)
jne target       # jump not equal, synonym jnz jump non zero (ZF=0)
jl  target       # jump less than, synonym jnge jump not greater  (SF!=OF)
jle target       # jump less or equal, synonym jng jump not greater or equal (ZF=1 or SF!=OF)
jg  target       # jump greater than, synonym jnle jump not less or equal (ZF=0 and SF=OF)
jge target       # jump greater or equal, synonym jnl jump not less (SF=OF)
ja  target       # jump above, synonym jnbe jump not below or equal (CF=0 and ZF=0)
jb  target       # jump below, synonym jnae jump not above or equal (CF=1)
js  target       # jump signed (SF=1)
jns target       # jump not signed (SF=0)
```

Function call stack

The `%rsp` register is used as the "stack pointer"; `push` and `pop` are used to add/remove values from the stack. The `push` instruction takes one operand: an immediate, a register, or a memory location. Push decrements `%rsp` and copies the operand to be tompost on the stack. The `pop` instruction takes one operand, the destination register. Pop copies the topmost value to destination and increments `%rsp`. It is also valid to directly adjust `%rsp` to add/remove an entire array or a collection of variables with a single operation. Note the stack grows downward (toward lower addresses).

```
push %rbx         # push value of %rbx onto stack
pushq $0x3        # push immediate value 3 onto stack
sub $0x10, %rsp   # adjust stack pointer to set aside 16 more bytes

pop %rax          # pop topmost value from stack into register %rax
add $0x10, %rsp   # adjust stack point to remove topmost 16 bytes
```

Call/return are using to transfer control between functions. The `callq` instruction takes one operand, the address of the function being called. It pushes the return address (current value of `%rip`) onto the stack and then jumps to that address. The `retq` instruction pops the return address from the stack into the destination `%rip`, thus resuming at the saved return address.

To set up for a call, the caller puts the first six arguments into registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` (any additional arguments are pushed onto the stack) and then executes the `call` instruction.

```
mov $0x3, %rdi    # first arg is passed in %rdi
mov $0x7, %rsi    # second arg is passed in %rsi
callq Binky       # transfers control to function Binky
```

When callee finishes, it writes the return value (if any) to `%rax`, cleans up the stack, and use `retq` instruction to return control to the caller.

```
mov $0x0, %eax    # write return value to %rax
add $0x10, %rsp   # deallocate stack frame
retq              # return from currently executing function, resume caller
```

The target for a branch or call instruction is most typically an absolute address that was determined at compile-time. However there are cases where the target is not known until runtime, such as a `switch` statement compiled into a jump table or when invoking a function pointer. For these, the target address is computed and stored in a register and the branch/call variant is used `je *%rax` or `callq *%rax` to read the target address from the specified register.

Assembly and gdb

The debugger has many features that allow you to trace and debug code at the assembly level. You can print the value in a register by prefixing its name with **`$`** or use the command **`info reg`** to dump the values of all registers:

```
(gdb) p $rsp
(gdb) info reg
```

The **`disassemble`** command will print the disassembly for a function by name. The **`x`** command supports an **`i`** format which interprets the contents of a memory address as an encoded instruction.

```
(gdb) disassemble main        // disassemble and print all instructions of main
(gdb) x/8i main               // disassemble and print first 8 instructions of main
```

You can set a breakpoint a particular assembly instruction by its direct address or offset within a function

```
(gdb) b *0x08048375
(gdb) b *main+7               // break at instruction 7 bytes into main
```

You can advance by instruction (instead of source line) using the **`stepi`** and **`nexti`** commands.

```
(gdb) stepi
(gdb) nexti
```