Y86-64 Quick Reference

Dr. Chris Archibald

CS 224

1 Introduction

This document is provided as a concise overview of the Y86-64 instruction set architecture. All of this content is from the book, but a lot of it is spread out across chapter 3 (where the book first introduces assembly instructions using X86-64) and chapter 4 (where the book introduces Y86-64). Most of the Y86-64 presentation assumes that the reader is familiar with X86-64 instructions, which is not the case for the order we have decided to cover the material. So, this handout is provided so that you can look in one place to find out the details that are spread across multiple places in the book. The hope is that this will enable you to more quickly understand Y86-64 and complete HW3.

2 Y86-64 Program State

The program state of the Y86-64 architecture consists of the following components:

- **Program Counter** - contains the address of the instruction currently being executed
- **Memory** - a single array of byte-addressable memory
- **Registers** - 15 registers that each can store a single 64-bit value. The registers are named as follows, with the identifier for each register given in parentheses after the name. With the exception of `%rsp`, which is assumed by several instructions to contain the address of the top of the stack, any of the registers can be used in any manner to contain any value.
  - `%rax` (0)
  - `%rcx` (1)
  - `%rdx` (2)
  - `%rbx` (3)
  - `%rsp` (4)
  - `%rbp` (5)
  - `%rsi` (6)
  - `%rdi` (7)
  - `%r8` (8)
  - `%r9` (9)
  - `%r10` (10)
  - `%r11` (11)
  - `%r12` (12)
  - `%r13` (13)
  - `%r14` (14)

The identifier 15 (`0xF`) - refers to no register.

- **Condition Codes**. There are 3 condition codes, each consisting of a single bit. These are:
  - `SF` - Sign flag - the most recent operation resulted in a negative value
  - `ZF` - Zero flag - the most recent operation yielded zero.
  - `OF` - Overflow flag - the most recent operation resulted in overflow (either over- or under-flow)
- **Program Status**. This indicates the status of the program, and the options for its value are, (with associated values in parenthesis)
  - `AOK` (1) - Normal operation
  - `HLT` (2) - Halt instruction encountered. Program halted.
  - `ADR` (3) - Invalid address encountered
  - `INS` (4) - Invalid instruction encountered

3 Y86-64 Instruction Set

This section contains a list of all of the instructions included in the Y86-64 instruction set, and an explanation of what they do.

4 Aside: `q`

The `q` at the end of some of the instructions refers to the fact that the instruction is working with a *quad-word*, which is the terminology for a 64-bit value. Historically, the a 16-bit value was called a *word*, and referred to with the letter `w`. This was used as the base name for other, longer values. 32-bit values were called *double words* and referred to with the letter `l`, while 64-bit values are called quad-words. Single byte values are manipulated by some instructions, with the letter `b`. These other letters (`b,w,l`) are used by X86-64 instructions, but the reduced Y86-64 ISA only has instructions that manipulate 64-bit quad-word quantities.

4.1 General Instructions

- `halt` - Halt the computer. Shuts down the entire system. Can be used at the end of a program, but since the encoding for halt is `0x00` any empty memory past the last instruction will be interpreted as `halt` anyway.
- `nop` - No operation. This is a "do nothing" command. Probably not a lot of reasons to use this in your code.

4.2 Data Movement Instructions

These instructions copy/move a single 64-bit value from one location to another. After these instructions the value is still in the old location, but now is also in the new location. The letters to help remember which does what are: (`i`) - immediate value (encoded in the instruction), (`r`) - register value, (`m`) - memory value. Values cannot be copied from memory to memory. These instructions are used very frequently, as to complete almost any useful task, values have to be moved around from memory to registers and back.

- `rrmovq rA, rB` - Move the 64 bit/8 byte value currently in register A into register B
- `irmovq V, rB` - Move the 64 bit value V into register B
- `rmmovq rA, D(rB)` - Move the 64 bit value in register A into the memory location with the address computed by adding the contents of register B to the value D.
- `mrmovq D(rB), rA` - Take the 64 bit value in the memory location with the address computed by adding the contents of register B to the value D and move that value into register A.

4.3 Arithmetic and Logical Operation Instructions

These instructions are fairly commonly used, especially the `addq` and `subq` instructions. These are used to increment/decrement counters, compute addresses in memory, and many other things.

- `OPq rA, rB` - Perform the specified operation and put the result in register B. These instructions set the three condition codes `ZF, SF, and OF` (zero, sign, overflow), depending on the value that results from the instruction. Note that the same register can be used as both `rA` and `rB` in these instructions. `rB` will get overwritten with the result in each case.
  - `addq rA, rB` - add the values in the registers - (`ifun = 0`)
  - `subq rA, rB` - subtract the value in register A from the value in register B - (`ifun = 1`)
  - `andq rA, rB` - bitwise and the values in the registers - (`ifun = 2`)

    A surprisingly useful command is `andq rA, rA`, where a register is and-ed with itself. This doesn't change the value that is in the register. So why is this useful? Well, this command does update the condition codes based on the value in the register. So if the value in the register is 0 now the `ZF` flag is set, and if it is negative then the `SF` is set. This is useful for control instructions (see next section) that depend on the value in a single register.
  - `xorq rA, rB` - bitwise xor the values in the registers - (`ifun = 3`)

4.4 Control Instructions

Most of the control instructions will look at the *condition codes* to perform an operation only if the condition specified by the instruction is met by the condition codes. These conditions will be expressed as logical functions of the 1-bit condition codes. The condition codes are `SF` - the sign flag, which indicates that the most recent operation yielded a negative number, `OF` - the overflow flag, which indicates that the result of the most recent operation could not be represented exactly using the bits available, and `ZF` - the zero flag, which indicates that the most recent operation resulted in a value of zero. Unconditional instructions (`rrmovq` and `jmp`) always perform their operation, regardless of the condition codes.

One way to match the instructions to the condition code combinations that cause them to execute is by thinking of the condition codes as resulting from the use of the subtraction instruction (`subq`) to compare two values. We will refer to these hypothetical values as `valA` and `valB`, and the value that is computed is `valA - valB`. The result of this combination we will think of as causing the condition flags to be set. The conditions are named as follows (equal to, less than, etc), and can be thought of as referring to the relationship of `valA` to `valB`.

- Equal to (`e`) - So how can we tell if `valA = valB`? If this is the case, then the subtraction should have resulted in a zero. So, this condition simply checks the `ZF` to see if it is set.

  `e : ZF`

- Less than (`l`) - This means that `valA` is less than `valB`. For this to be true, the subtraction result would have to be negative. The result is negative if the sign flag is set, and there is no overflow. It can also be negative if the sign flag is not set, but there was overflow. This can be expressed succinctly as the condition `SF ^ OF` (where ^ is the XOR operator).

  `l : (SF ^ OF)`

- Less than or equal to (`le`) - This will be true if either the difference is negative (less than case) or if the difference is zero (equals case). We simply check to see if either one of those two cases is satisfied. If they are, then we know that `valA` is less than or equal to `valB`. To check if one of them is true we can OR them together, giving a final expression of.

  `le : (SF ^ OF) | ZF`.

- Not equal (`ne`) - If `valA` $\neq$ `valB` then the zero flag `ZF` will *not* be set. This is simply the negation of the condition for the equal to case above, and is true when the zero flag is not set.

  `ne` : $\neg$`ZF`

- Greater than or equal to (`ge`) - For it to be true that `valA` is greater than or equal to `valB` it simply must not be true that it is less than. We get the condition for this as the negation of the `l` condition.

Function code	Conditional Instruction Suffix	Performed if ...
0	Unconditional	Always performed
3	`e`	`ZF`
4	`ne`	`¬ZF`
2	`l`	`(SF ^ OF)`
1	`le`	`(SF ^ OF) | ZF`
6	`g`	`¬(SF ^ OF) & ¬ZF`
5	`ge`	`¬(SF ^ OF)`

Table 1: Control Conditions

`ge` : ¬(`SF ^ OF`)

- Greater than (`g`) - If `valA` is greater than `valB` then we can't have a negative value, and we also can't have a zero flag. Both of these conditions have to be true. So we have the negation of the `l` condition AND the negation of the `e` condition. (This is also equal to the negation of the `le` condition).

  `g` : ¬(`SF ^ OF`) & ¬ `ZF`

Those are the different conditions that can trigger the conditional instructions, and the combinations of the flags that each looks at. Note that there isn't really a `valA` and `valB` that are being compared, rather those just help us remember which combinations of the flags go with which letters and cases. When a conditional instruction is executed it will simply look at the corresponding combination of the flags, and if it evaluates to true, then it will execute the instruction, otherwise it won't.

4.5 The Control Instructions

- `jXX Dest` - Jump to the destination address `Dest` encoded in the instruction. This is done by setting the program counter (PC) to the specified address. Each jump occurs when the associated conditions are satisfied, which look at the condition codes as specified. Jump commands are frequently used as the main mechanism for having code that does different things depending on conditions in the code. Any sort of a `while` loop, or a `for` loop, or an `if` statement, can all be implemented with jump commands. You will definitely use some of these in HW3.

  - `jmp Dest` - Unconditional jump. This jump is always taken.
  - `jle Dest` - Jump less than or equal. Jump taken if the `le` condition (Table 1) is true.
  - `jl Dest` - Jump less than. Jump taken if the `l` condition (Table 1) is true.
  - `je Dest` - Jump equal. Jump taken if the `e` condition (Table 1) is true.
  - `jne Dest` - Jump not equal. Jump taken if the `ne` condition (Table 1) is true.
  - `jge Dest` - Jump greater than or equal. Jump taken if the `ge` condition (Table 1) is true.
  - `jg Dest` - Jump greater than. Jump taken if the `g` condition (Table 1) is true.

- `cmovXX rA, rB` - These instructions are conditional move instructions, and the value in register A is placed into register B if the specified condition code is met. These instructions are not frequently used, as we can often use a conditional jump and a normal move command instead. You can definitely complete HW3 without these.
  - `cmovle rA, rB` - Move if less than or equal. This move is performed if the `le` condition is true (Table 1) .
  - `cmovl rA, rB` - Move if less than. This move is performed if the `l` condition is true (Table 1) .
  - `cmove rA, rB` - Move if equal. This move is performed if the `e` condition is true (Table 1) .
  - `cmovne rA, rB` - Move if not equal. This move is performed if the `ne` condition is true (Table 1) .
  - `cmovge rA, rB` - Move if greater than or equal. This move is performed if the `ge` condition is true (Table 1) .
  - `cmovg rA, rB` - Move if greater than. This move is performed if the `g` condition is true (Table 1) .

4.6 Procedure/Function Instructions

You shouldn't need these for HW3, but we will talk about them more as we get into the X86-64 instructions set and looking at real code.

- `call Dest` - Function call. This transfers control to the specified destination address in code. It first stores the next instruction address (PC) onto the stack and then changes the PC to be `Dest`.
- `ret` - Return from function. This transfers control back after a function is done. It will pop the address off of the stack and sets the PC to be this new address.

- `pushq, rA` - Push the value in register A onto the stack, and update the stack pointer
- `popq, rA` - Pop the value off the top of the stack into register A, and update the stack pointer.