Y86-64: Instruction Set Architecture

CMPU 224 – Computer Organization
Jason Waterman
Instruction Set Architecture

- **Assembly Language View**
  - Processor state
    - Registers, memory, ...
  - Instructions
    - `addq`, `pushq`, `ret`, ...
    - How instructions are encoded as bytes

- **Layer of Abstraction**
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously
Y86-64 Processor State

- Program Registers
  - 15 registers (omit %r15)
  - Each 64-bits long

- Condition Codes
  - Single-bit flags set by arithmetic and logical instructions
    - ZF: Zero
    - SF: Negative
    - OF: Overflow

- Program Counter
  - Indicates address of next instruction

- Program Status
  - Indicates either normal operation or some error condition

- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order
Y86-64 Instructions

• Largely a subset of x86-64 instructions
• Only 8 byte integer operations
• Format
  • 1–10 bytes of information read from memory
  • Can determine instruction length from first byte
  • Not as many instruction types, and simpler encoding than with x86-64
## Y86-64 Instruction Set

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Y86-64 Instruction Set

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **rrmovq**: 2 0
- **cmovle**: 2 1
- **cmovl**: 2 2
- **cmove**: 2 3
- **cmovne**: 2 4
- **cmovge**: 2 5
- **cmovg**: 2 6
- **jmp**: 7 0
- **jle**: 7 1
- **jl**: 7 2
- **je**: 7 3
- **jne**: 7 4
- **jge**: 7 5
- **jg**: 7 6
- **addq**: 6 0
- **subq**: 6 1
- **andq**: 6 2
- **xorq**: 6 3
Encoding Registers

- Each register has 4-bit ID
- Same encoding as in x86-64
- Register ID 15 (0xF) indicates “no register”
- Will use this in our hardware design in multiple places

<table>
<thead>
<tr>
<th>Register</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0</td>
</tr>
<tr>
<td>%rcx</td>
<td>1</td>
</tr>
<tr>
<td>%rdx</td>
<td>2</td>
</tr>
<tr>
<td>%rbx</td>
<td>3</td>
</tr>
<tr>
<td>%rsp</td>
<td>4</td>
</tr>
<tr>
<td>%rbp</td>
<td>5</td>
</tr>
<tr>
<td>%rsi</td>
<td>6</td>
</tr>
<tr>
<td>%rdi</td>
<td>7</td>
</tr>
<tr>
<td>%r8</td>
<td>8</td>
</tr>
<tr>
<td>%r9</td>
<td>9</td>
</tr>
<tr>
<td>%r10</td>
<td>A</td>
</tr>
<tr>
<td>%r11</td>
<td>B</td>
</tr>
<tr>
<td>%r12</td>
<td>C</td>
</tr>
<tr>
<td>%r13</td>
<td>D</td>
</tr>
<tr>
<td>%r14</td>
<td>E</td>
</tr>
<tr>
<td>No Register</td>
<td>F</td>
</tr>
</tbody>
</table>

- Byte

<table>
<thead>
<tr>
<th>Instruction</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Example

• Addition Instruction

  • Add value in register rA to that in register rB
  • Store result in register rB
  • Note that Y86-64 only allows addition to be applied to register data

• Set condition codes based on result
  • e.g., addq %rax,%rsi Encoding: 60 06

• Two-byte encoding
  • First indicates instruction type
  • Second gives source and destination registers
Arithmetic and Logical Operations

- Refer to generically as "OPq"
- Encodings differ only by "function code"
  - Low-order 4 bits in first instruction word
- Set condition codes as side effect

<table>
<thead>
<tr>
<th>Instruction Code</th>
<th>Function Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td></td>
</tr>
<tr>
<td>addq rA, rB</td>
<td>6 0 rA rB</td>
</tr>
<tr>
<td>Subtract (rA from rB)</td>
<td></td>
</tr>
<tr>
<td>subq rA, rB</td>
<td>6 1 rA rB</td>
</tr>
<tr>
<td>And</td>
<td></td>
</tr>
<tr>
<td>andq rA, rB</td>
<td>6 2 rA rB</td>
</tr>
<tr>
<td>Exclusive-Or</td>
<td></td>
</tr>
<tr>
<td>xorq rA, rB</td>
<td>6 3 rA rB</td>
</tr>
</tbody>
</table>
Move Operations

• Like the x86-64 movq instruction
• Simpler format for memory addresses
• Give different names to keep them distinct

rrmovq rA, rB  2 0 rA rB  
Immediate ➔ Register

irmovq V, rB  3 0 F rB  

rmmovq rA, D(rB)  4 0 rA rB  
Register ➔ Memory

mrmovq D(rB), rA  5 0 rA rB  
Memory ➔ Register
Move Instruction Examples

X86-64

movq $0xabcd, %rdx

movq %rsp, %rbx

movq -12(%rbp),%rcx

movq %rsi,0x41c(%rsp)

Y86-64

irmovq $0xabcd, %rdx

rrmovq %rsp, %rbx

rmrovq -12(%rbp),%rcx

rmmovq %rsi,0x41c(%rsp)

Encoding:

30 F2 cd ab 00 00 00 00 00 00

20 43

50 15 f4 ff ff ff ff ff ff ff

40 64 1c 04 00 00 00 00 00 00

Little-endian

Two’s complement
### Conditional Move Instructions

**Move Unconditionally**

- **cmovq rA, rB**

**Move When Less or Equal**

- **cmovle rA, rB**

**Move When Less**

- **cmovl rA, rB**

**Move When Equal**

- **cmove rA, rB**

**Move When Not Equal**

- **cmovne rA, rB**

**Move When Greater or Equal**

- **cmovge rA, rB**

**Move When Greater**

- **cmovg rA, rB**

---

**Key Points**

- Refer to generically as “cmovXX”
- Encodings differ only by “function code”
- Based on values of condition codes
- Variants of `rrmovq` instruction
  - (Conditionally) copy value from source to destination register
Jump Instructions

• Refer to generically as “jXX”
• Encodings differ only by “function code” fn
• Based on values of condition codes
• Same as x86-64 counterparts
• Encode full destination address
  • Unlike PC-relative addressing seen in x86-64
# Jump Instructions

Jump Unconditionally

| `jmp` `Dest` | 7 | 0 | Dest |

Jump When Less or Equal

| `jle` `Dest` | 7 | 1 | Dest |

Jump When Less

| `jl` `Dest` | 7 | 2 | Dest |

Jump When Equal

| `je` `Dest` | 7 | 3 | Dest |

Jump When Not Equal

| `jne` `Dest` | 7 | 4 | Dest |

Jump When Greater or Equal

| `jge` `Dest` | 7 | 5 | Dest |

Jump When Greater

| `jg` `Dest` | 7 | 6 | Dest |
Y86-64 Program Stack

- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by $\%\text{rsp}$
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer
Stack Operations

- **pushq rA**
  - Decrement `%rsp` by 8
  - Store word from rA to memory at `%rsp`
  - Like x86-64

- **popq rA**
  - Read word from memory at `%rsp`
  - Save in rA
  - Increment `%rsp` by 8
  - Like x86-64
Subroutine Call and Return

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64

```
call Dest
```

- Pop value from stack
- Use as address for next instruction
- Like x86-64

```
ret
```
Miscellaneous Instructions

- **nop**
  - 1 0
  - Don’t do anything

- **halt**
  - 0 0
  - Stop executing instructions
  - x86-64 has comparable instruction, but can’t execute it in user mode
  - We will use it to stop the simulator
  - Encoding ensures that program hitting memory initialized to zero will halt
Status Conditions

• Normal operation
• Halt instruction encountered
• Bad address (either instruction or data) encountered
• Invalid instruction encountered

• Desired Behavior
  • If AOK, keep going
  • Otherwise, stop program execution

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOK</td>
<td>1</td>
</tr>
<tr>
<td>HLT</td>
<td>2</td>
</tr>
<tr>
<td>ADR</td>
<td>3</td>
</tr>
<tr>
<td>INS</td>
<td>4</td>
</tr>
</tbody>
</table>
Writing Y86-64 Code

• Can try to Use C Compiler
  • Write code in C
  • Compile for x86-64 with `gcc -Og -S`
  • Transliterate into Y86-64
  • *Modern compilers make this more difficult*

• Coding Example
  • Find number of elements in null-terminated list
    
    ```c
    int len1(int a[]);
    ```

<table>
<thead>
<tr>
<th>a</th>
<th>5043</th>
<th>6125</th>
<th>7395</th>
<th>0</th>
<th>3</th>
</tr>
</thead>
</table>

4/1/2019
CMPU 224 -- Computer Organization
Y86-64 Code Generation Example

• First Try
  • Write typical array code

```c
/* Find number of elements in null-terminated list */
long len(long a[])
{
    long len;
    for (len = 0; a[len]; len++);
    return len;
}
```

• Compile with `gcc -Og -S`

• Problem
  • Hard to do array indexing on Y86-64
    • Since don’t have scaled addressing modes

```assembly
len:
    movl $0, %eax
.L3:
    cmpq $0, (%rdi,%rax,8)
    je .L2
    addq $1, %rax
    jmp .L3
.L2:
    ret
```
Y86-64 Code Generation Example #2

- Second Try
  - Write C code that mimics expected Y86-64 code

```c
long len(long a[]) {
    long val = *a;
    long len = 0;
    while (val) {
        a++;
        len++;
        val = *a;
    }
    return len;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>a</td>
</tr>
<tr>
<td>%rax</td>
<td>len</td>
</tr>
<tr>
<td>%rdx</td>
<td>val</td>
</tr>
</tbody>
</table>

```
len:
    movq (%rdi), %rdx     # val = *a
    movl $0, %eax         # len = 0
.L3:
    testq %rdx, %rdx      # while(val)
    je .L2                # while(val)
    addq $8, %rsi         # a++
    addq $1, %rax         # len++
    movq (%rdi), %rdx     # val = *a
    jmp .L3               # jump to while test
.L2:
    ret                   # return len
```
len:
    movq (%rdi), %rdx
    movl $0, %eax
.L3:
    testq %rdx, %rdx
    je .L2
    addq $8, %rdi
    addq $1, %rax
    movq (%rdi), %rdx
    jmp .L3
.L2:
    rep ret

len:
    irmovq $1, %r8      # Constant 1
    irmovq $8, %r9      # Constant 8
    mrmovq (%rdi), %rdx # val = *a
    irmovq $0, %rax      # len = 0

test:
    andq %rdx, %rdx      # Test val
    je done              # If zero, goto Done
    addq %r9, %rdi       # a++
    addq %r8, %rax       # len++
    mrmovq (%rdi), %rdx  # val = *a
    jmp test             # Jump to test

done:
    ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>a</td>
</tr>
<tr>
<td>%rax</td>
<td>len</td>
</tr>
<tr>
<td>%rdx</td>
<td>val</td>
</tr>
<tr>
<td>%r8</td>
<td>1</td>
</tr>
<tr>
<td>%r9</td>
<td>8</td>
</tr>
</tbody>
</table>
Y86-64 Sample Program Structure #1

- Program starts at address 0
- Must set up stack
  - Where located
  - Make sure don’t overwrite code!
- Must initialize data
  - See next slide

```assembly
# Initialization
  .pos 0     # Execution begins at address 0
  irmovq stack, %rsp # Set up stack pointer

  call main    # Execute main program
  halt

  .align 8      # Program data
array:
  . . .

main:               # Main function
  . . .
  call len
  . . .

len:                 # Length function
  . . .

  .pos 0x200       # Placement of stack
stack:
```
Y86-64 Program Structure #2

- Must initialize data
  - Can use symbolic names
- Set up call to `len`
  - Follow x86-64 procedure conventions
- Push array address as argument

```assembly
# Initialization
.pos 0  # Execution begins at address 0

irmovq stack, %rsp  # Set up stack pointer

call main  # Execute main program
halt

# Array of 4 elements + terminating 0
.align 8
array:
  .quad 0x000d000d000d000d000d
  .quad 0x00c000c000c000c000c0
  .quad 0x0b000b000b000b000b0
  .quad 0x0a000a000a000a000a
  .quad 0

main:
  ...

.pos 0x200  # Placement of stack

Main:
  irmovq array, %rdi
  # call len(array)
  call len
  ret
```

---

4/1/2019 CMPU 224 -- Computer Organization 25
Assembling Y86-64 Programs (yas)

• Generates “object code” file len.yo
  • Actually looks like disassembler output

Linux> yas len.ys

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x054:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x054:</td>
<td>30f8010000000000000000000</td>
<td>irmovq $1, %r8 # Constant 1</td>
</tr>
<tr>
<td>0x05e:</td>
<td>30f9080000000000000000000</td>
<td>irmovq $8, %r9 # Constant 8</td>
</tr>
<tr>
<td>0x068:</td>
<td>5027000000000000000000000</td>
<td>mrmovq (%rdi), %rdx # val = *a</td>
</tr>
<tr>
<td>0x072:</td>
<td>30f0000000000000000000000</td>
<td>irmovq $0, %rax # len = 0</td>
</tr>
<tr>
<td>0x07c:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x07c:</td>
<td>6222</td>
<td>andq %rdx, %rdx # Test val</td>
</tr>
<tr>
<td>0x07e:</td>
<td>739e000000000000000000000</td>
<td>je done # If zero, goto Done</td>
</tr>
<tr>
<td>0x087:</td>
<td>6097</td>
<td>addq %r9, %rdi # a++</td>
</tr>
<tr>
<td>0x089:</td>
<td>6080</td>
<td>addq %r8, %rax # len++</td>
</tr>
<tr>
<td>0x08b:</td>
<td>5027000000000000000000000</td>
<td>mrmovq (%rdi), %rdx # val = *a</td>
</tr>
<tr>
<td>0x095:</td>
<td>707c000000000000000000000</td>
<td>jmp test # Jump to test</td>
</tr>
<tr>
<td>0x09e:</td>
<td></td>
<td>done:</td>
</tr>
<tr>
<td>0x09e:</td>
<td>90</td>
<td>ret</td>
</tr>
</tbody>
</table>
Simulating Y86-64 Programs (yis)

- Instruction set simulator
  - Computes effect of each instruction on processor state
  - Prints changes in state from original

Linux> yis len.yo

Stopped in 37 steps at PC = 0x13. Status 'HLT', CC Z=1
S=0 O=0
Changes to registers:
%rax: 0x0000000000000000 0x0000000000000004
%rsp: 0x0000000000000000 0x0000000000000200
%rdi: 0x0000000000000000 0x000000000000038
%r8: 0x0000000000000000 0x0000000000000001
%r9: 0x0000000000000000 0x0000000000000008

Changes to memory:
0x01f0: 0x0000000000000000 0x0000000000000053
0x01f8: 0x0000000000000000 0x0000000000000013
# Y86-64 Instruction Set

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Register States

- %rax: 0
- %rcx: 1
- %rdx: 2
- %rbx: 3
- %rsp: 4
- %rbp: 5
- %rsi: 6
- %rdi: 7
- %r8: 8
- %r9: 9
- %r10: A
- %r11: B
- %r12: C
- %r13: D
- %r14: E
- No Register: F