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>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>
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>

- pushq rA | A | 0 | rA | F |   |   |   |   |   |   |
- popq rA | B | 0 | rA | F |   |   |   |   |   |   |
- 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 |   |   |   |   |   |   |   |   |

10/30/2018

CMPU 224 -- Computer Organization
## Encoding Registers

- Each register has 4-bit ID
  
<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>

- Same encoding as in x86-64

- Register ID 15 (0xF) indicates “no register”

- Will use this in our hardware design in multiple places

```plaintext
<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>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>
```
• 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

<table>
<thead>
<tr>
<th>Instruction Code</th>
<th>Function Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td></td>
</tr>
<tr>
<td>\texttt{addq rA, rB}</td>
<td>\begin{tabular}{c</td>
</tr>
<tr>
<td>Subtract (rA from rB)</td>
<td></td>
</tr>
<tr>
<td>\texttt{subq rA, rB}</td>
<td>\begin{tabular}{c</td>
</tr>
<tr>
<td>And</td>
<td></td>
</tr>
<tr>
<td>\texttt{andq rA, rB}</td>
<td>\begin{tabular}{c</td>
</tr>
<tr>
<td>Exclusive-Or</td>
<td></td>
</tr>
<tr>
<td>\texttt{xorq rA, rB}</td>
<td>\begin{tabular}{c</td>
</tr>
</tbody>
</table>

- 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
Move Operations

- Like the x86-64 `movq` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct
### Move Instruction Examples

<table>
<thead>
<tr>
<th>X86-64</th>
<th>Y86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $0xabcd, %rdx</td>
<td>irmovq $0xabcd, %rdx</td>
</tr>
<tr>
<td>Encoding: 30 F2 cd ab 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>movq %rsp, %rbx</td>
<td>rrmovq %rsp, %rbx</td>
</tr>
<tr>
<td>Encoding: 20 43</td>
<td></td>
</tr>
<tr>
<td>movq -12(%rbp),%rcx</td>
<td>mrmovq -12(%rbp),%rcx</td>
</tr>
<tr>
<td>Encoding: 50 51 f4 ff ff ff ff ff ff ff</td>
<td></td>
</tr>
<tr>
<td>movq %rsi,0x41c(%rsp)</td>
<td>rmmovq %rsi,0x41c(%rsp)</td>
</tr>
<tr>
<td>Encoding: 40 64 1c 04 00 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>

- **Little-endian**
- **Two’s complement**
Conditional Move Instructions

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Instruction</th>
<th>Encodings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move Unconditionally</td>
<td>\texttt{rrmovq \textit{rA, rB}}</td>
<td>2 0 \textit{rA rB}</td>
</tr>
<tr>
<td>Move When Less or Equal</td>
<td>\texttt{cmovle \textit{rA, rB}}</td>
<td>2 1 \textit{rA rB}</td>
</tr>
<tr>
<td>Move When Less</td>
<td>\texttt{cmovl \textit{rA, rB}}</td>
<td>2 2 \textit{rA rB}</td>
</tr>
<tr>
<td>Move When Equal</td>
<td>\texttt{cmove \textit{rA, rB}}</td>
<td>2 3 \textit{rA rB}</td>
</tr>
<tr>
<td>Move When Not Equal</td>
<td>\texttt{cmovne \textit{rA, rB}}</td>
<td>2 4 \textit{rA rB}</td>
</tr>
<tr>
<td>Move When Greater or Equal</td>
<td>\texttt{cmovge \textit{rA, rB}}</td>
<td>2 5 \textit{rA rB}</td>
</tr>
<tr>
<td>Move When Greater</td>
<td>\texttt{cmovg \textit{rA, rB}}</td>
<td>2 6 \textit{rA rB}</td>
</tr>
</tbody>
</table>

10/30/2018 CMPU 224 -- Computer Organization 12
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 (Conditionally)

| jXX Dest | 7 | fn | Dest |
## 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 $\%rsp$
  - 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
Miscellaneous Instructions

- Don’t do anything

- 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[]);
    ```

    | a  |  5043 |
    |----|-------|
    |    |  6125 |
    |    |  7395 |
    |    |  0    |

    $\Rightarrow 3$
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;
}
```

```assembly
len:
    movq (%rdi), %rdx # val = *a
    movl $0, %eax    # len = 0
.L3:
    testq %rdx, %rdx # while(val)
    je .L2           # while(val)
    addq $8, %rdi    # a++
    addq $1, %rax    # len++
    movq (%rdi), %rdx # val = *a
    jmp .L3           # jump to while test
.L2:
    ret               # 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>
Y86-64 Code Generation Example #3

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

<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>

done:
  ret

test:
  andq %rdx, %rdx
  je done
  addq %r9, %rdi
  addq %r8, %rax
  mrmovq (%rdi), %rdx
  jmp test

len:
  irmovq $1, %r8     # Constant 1
  irmovq $8, %r9     # Constant 8
  mrmovq (%rdi), %rdx # val = *a
  irmovq $0, %rax    # len = 0
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
Y86-64 Program Structure #2

- Must initialize data
  - Can use symbolic names
- Set up call to \texttt{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 0x000d000d000d000d
.quad 0x00c000c000c000c
.quad 0x0b000b000b000b
.quad 0xa000a000a000a
.quad 0
main:
.call len(array)
call len
ret
```

Main:

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

Stack:

```assembly
.pos 0x200 # Placement of stack
```

10/30/2018
CMPU 224 -- Computer Organization
Assembling Y86-64 Programs (yas)

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

```
0x054: | len:
0x054: 30f8010000000000000000000 | irmovq $1, %r8  # Constant 1
0x05e: 30f9080000000000000000000 | irmovq $8, %r9  # Constant 8
0x068: 5027000000000000000000000 | mrmovq (%rdi), %rdx # val = *a
0x072: 30f0000000000000000000000 | irmovq $0, %rax # len = 0
0x07c: | test:
0x07c: 6222 | andq %rdx, %rdx # Test val
0x07e: 739e00000000000000 | je done # If zero, goto Done
0x087: 6097 | addq %r9, %rdi # a++
0x089: 6080 | addq %r8, %rax # len++
0x08b: 5027000000000000000000000 | mrmovq (%rdi), %rdx # val = *a
0x095: 707c00000000000000 | jmp test # Jump to test
0x09e: | done:
0x09e: 90 | ret
```
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 0x0000000000000038
%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>V</td>
<td></td>
<td>F</td>
<td>rB</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></td>
<td>D</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></td>
<td>D</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>
</tbody>
</table>

### Register Set

- %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