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

<table>
<thead>
<tr>
<th>Application Program</th>
<th>Compiler</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>CPU</td>
<td>Design</td>
</tr>
<tr>
<td></td>
<td>Circuit</td>
<td>Design</td>
</tr>
<tr>
<td></td>
<td>Chip</td>
<td>Layout</td>
</tr>
</tbody>
</table>

4/12/2021

CMPU 224 -- Computer Organization
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>Instruction</th>
<th>Byte</th>
<th>Function</th>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2 fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3 0 F</td>
<td>rB</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4 0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>mrmmovq D(rB), rA</td>
<td>5 0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6 fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7 fn</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A 0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B 0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Y86-64 Instruction Set

<table>
<thead>
<tr>
<th>Byte</th>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>halt</td>
<td>00</td>
<td>Halt the program.</td>
</tr>
<tr>
<td>1</td>
<td>nop</td>
<td>10</td>
<td>No operation.</td>
</tr>
<tr>
<td>2</td>
<td>mmovq rA, rB</td>
<td>2 fn rA rB</td>
<td>Move byte from register B to register A.</td>
</tr>
<tr>
<td></td>
<td>cmovXX rA, rB</td>
<td></td>
<td>Conditional move based on X (less than or equal to, larger than, etc.).</td>
</tr>
<tr>
<td>3</td>
<td>irmovq V, rB</td>
<td>3 0 F rB V</td>
<td>Immediate move.</td>
</tr>
<tr>
<td>4</td>
<td>rrmovq rA, D(rB)</td>
<td>4 0 rA rB D</td>
<td>Register move with offset D.</td>
</tr>
<tr>
<td>5</td>
<td>rmovq D(rB), rA</td>
<td>5 0 rA rB D</td>
<td>Move register contents of D to register A.</td>
</tr>
<tr>
<td>6</td>
<td>OPq rA, rB</td>
<td>6 fn rA rB</td>
<td>Operation between registers A and B.</td>
</tr>
<tr>
<td>7</td>
<td>jXX Dest</td>
<td>7 fn Dest</td>
<td>Jump to destination address.</td>
</tr>
<tr>
<td>8</td>
<td>call Dest</td>
<td>8 0 Dest</td>
<td>Call function.</td>
</tr>
<tr>
<td>9</td>
<td>ret</td>
<td>9 0</td>
<td>Return from function.</td>
</tr>
<tr>
<td>A</td>
<td>pushq rA</td>
<td>A 0 rA F</td>
<td>Push register A onto the stack.</td>
</tr>
<tr>
<td>B</td>
<td>popq rA</td>
<td>B 0 rA F</td>
<td>Pop register A from the stack.</td>
</tr>
</tbody>
</table>

Instruction formats:
- **mmovq**: Move byte from register B to register A.
- **cmovXX**: Conditional move based on X (less than or equal to, larger than, etc.).
- **irmovq**: Immediate move.
- **rrmovq**: Register move with offset D.
- **rmovq**: Move register contents of D to register A.
- **OPq**: Operation between registers A and B.
- **jXX**: Jump to destination address.
- **call**: Call function.
- **ret**: Return from function.
- **pushq**: Push register A onto the stack.
- **popq**: Pop register A from the stack.
## 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><strong>halt</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>nop</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>cmovXX rA, rB</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>irmovq V, rB</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rmmovq rA, D(rB)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>mrmovq D(rB), rA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OPq rA, rB</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>jXX Dest</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>call Dest</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ret</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pushq rA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>popq rA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example Instructions

- `rrmovq rA` (Instruction 2 0)
- `cmovle rA` (Instruction 2 1)
- `cmovl rA` (Instruction 2 2)
- `cmove rA` (Instruction 2 3)
- `cmovne rA` (Instruction 2 4)
- `cmovge rA` (Instruction 2 5)
- `cmovg rA` (Instruction 2 6)
- `jmp rA` (Instruction 7 0)
- `jle rA` (Instruction 7 1)
- `jl rA` (Instruction 7 2)
- `je rA` (Instruction 7 3)
- `jne rA` (Instruction 7 4)
- `jge rA` (Instruction 7 5)
- `jg rA` (Instruction 7 6)

### Examples of Operations

- `addq rA` (Instruction 6 0)
- `subq rA` (Instruction 6 1)
- `andq rA` (Instruction 6 2)
- `xorq rA` (Instruction 6 3)
- `nop` (Instruction 1 0)
- `halt` (Instruction 0 0)
Encoding Registers

- Each register has 4-bit ID
  - %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

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

- **Add**
  - **Instruction Code**: addq rA, rB
  - **Function Code**: 6 0 rA rB

- **Subtract (rA from rB)**
  - **Instruction Code**: subq rA, rB
  - **Function Code**: 6 1 rA rB

- **And**
  - **Instruction Code**: andq rA, rB
  - **Function Code**: 6 2 rA rB

- **Exclusive-Or**
  - **Instruction Code**: xorq rA, rB
  - **Function Code**: 6 3 rA rB

- **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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rrmovq rA, rB</code></td>
<td>2 0 rA rB</td>
<td>Register ➔ Register</td>
</tr>
<tr>
<td><code>irmovq V, rB</code></td>
<td>3 0 F rB</td>
<td>Immediate ➔ Register</td>
</tr>
<tr>
<td><code>rmmovq rA, D(rB)</code></td>
<td>4 0 rA rB</td>
<td>Register ➔ Memory</td>
</tr>
<tr>
<td><code>mrmovq D(rB), rA</code></td>
<td>5 0 rA rB</td>
<td>Memory ➔ Register</td>
</tr>
</tbody>
</table>

- 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><strong>Encoding:</strong> 30 F2 cd ab 00 00 00 00 00 00</td>
<td><strong>Little-endian</strong></td>
</tr>
<tr>
<td>movq %rsp, %rbx</td>
<td>rrmovq %rsp, %rbx</td>
</tr>
<tr>
<td><strong>Encoding:</strong> 20 43</td>
<td><strong>Two’s complement</strong></td>
</tr>
<tr>
<td>movq -12(%rbp),%rcx</td>
<td>mrmovq -12(%rbp),%rcx</td>
</tr>
<tr>
<td><strong>Encoding:</strong> 50 15 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><strong>Encoding:</strong> 40 64 1c 04 00 00 00 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>
Conditional Move Instructions

- 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

**Move Unconditionally**

- `rrmovq rA, rB` (Encoding: 2 0 rA rB)

**Move When Less or Equal**

- `cmovle rA, rB` (Encoding: 2 1 rA rB)

**Move When Less**

- `cmovl rA, rB` (Encoding: 2 2 rA rB)

**Move When Equal**

- `cmove rA, rB` (Encoding: 2 3 rA rB)

**Move When Not Equal**

- `cmovne rA, rB` (Encoding: 2 4 rA rB)

**Move When Greater or Equal**

- `cmovge rA, rB` (Encoding: 2 5 rA rB)

**Move When Greater**

- `cmovg rA, rB` (Encoding: 2 6 rA rB)
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump Unconditionally</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jmp Dest</code></td>
<td>7 0</td>
<td>Dest</td>
</tr>
<tr>
<td>Jump When Less or Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jle Dest</code></td>
<td>7 1</td>
<td>Dest</td>
</tr>
<tr>
<td>Jump When Less</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jl Dest</code></td>
<td>7 2</td>
<td>Dest</td>
</tr>
<tr>
<td>Jump When Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>je Dest</code></td>
<td>7 3</td>
<td>Dest</td>
</tr>
<tr>
<td>Jump When Not Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jne Dest</code></td>
<td>7 4</td>
<td>Dest</td>
</tr>
<tr>
<td>Jump When Greater or Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jge Dest</code></td>
<td>7 5</td>
<td>Dest</td>
</tr>
<tr>
<td>Jump When Greater</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jg Dest</code></td>
<td>7 6</td>
<td>Dest</td>
</tr>
</tbody>
</table>
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 $\%r{sp}$
  - 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

- 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 the number of elements in null-terminated list
    ```c
    int len1(int a[]);
    ```

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

⇒ 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

```asm
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, %rdi    # a++
    addq $1, %rax    # len++
    movq (%rdi), %rdx # val = *a
    jmp .L3          # jump to while test
.L2:
    ret             # return len
```
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

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 .L2                   # If zero, goto Done
   addq $8, %rdi
   addq $1, %rax
   movq (%rdi), %rdx
jmp .L3
.L2:
   rep ret

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 \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 0xa000a000a000a00
    .quad 0
main:
    irmovq array, %rdi
    call len
    ret
...
    .pos 0x200    # Placement of stack
stack:
```

4/12/2021
CMPU 224 -- Computer Organization
Assembling Y86-64 Programs (yas)

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

```
Linux> yas len.ys
```

```
0x054: | len:          
0x054: 30f8010000000000000 | irmovq $1, %r8     # Constant 1
0x05e: 30f9080000000000000 | irmovq $8, %r9     # Constant 8
0x068: 5027000000000000000 | mrmovq (%rdi), %rdx # val = *a
0x072: 30f0000000000000000 | irmovq $0, %rax     # len = 0
0x07c: | test:         
0x07c: 6222 | andq %rdx, %rdx   # Test val
0x07e: 739e000000000000000 | je done           # If zero, goto Done
0x087: 6097 | addq %r9, %rdi   # a++
0x089: 6080 | addq %r8, %rax   # len++
0x08b: 5027000000000000000 | mrmovq (%rdi), %rdx # val = *a
0x095: 707c000000000000000 | 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

```bash
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 0x0000000000000380
%rdx:  0x0000000000000000 0x0000000000000001
%rsi:  0x0000000000000000 0x0000000000000008

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

Byte | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
---|---|---|---|---|---|---|---|---|---|---|
halt | 0 | 0 |  |  |  |  |  |  |  |  |
nop  | 1 | 0 |  |  |  |  |  |  |  |  |
cmovXX rA, rB | 2 | fn | rA | rB | | | | | | |
irmovq V, rB | 3 | 0 | F | rB | V | | | | | |
rmmovq rA, D(rB) | 4 | 0 | rA | rB | D | | | | | |
rmmovq D(rB), rA | 5 | 0 | rA | rB | D | | | | | |
OPq rA, rB | 6 | fn | rA | rB | | | | | | |
jXX Dest | 7 | fn | Dest | | | | | | | |
call Dest | 8 | 0 | Dest | | | | | | | |
ret | 9 | 0 | | | | | | | | |
pushq rA | A | 0 | rA | F | | | | | | |
popq rA | B | 0 | rA | F | | | | | | |

<table>
<thead>
<tr>
<th>Instruction</th>
<th>2</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>
</tr>
</thead>
<tbody>
<tr>
<td>cmovvle</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovvl</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovve</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovvne</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovvge</td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovvg</td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jle</td>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jl</td>
<td>7</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>je</td>
<td>7</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jne</td>
<td>7</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jge</td>
<td>7</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jg</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Yes Register