Computer Architecture

Instruction Set Architecture – Y86

CSCI 2021: Machine Architecture and Organization

Pen-Chung Yew
Department Computer Science and Engineering
University of Minnesota

With Slides from Bryant, O’Hallaron and Zhai

ISA

Compiler

OS

CPU Design

Circuit Design

Chip Layout

Assembly Programmer’s View

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

BF: Program registers

CC: Condition codes

Stat: Program status

RF

PC

DMEM: Memory

• Program Registers
  • 15 registers (omit %r15). Each 64 bits
• Condition Codes
  • Single-bit flags set by arithmetic or 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 Instruction Set #1

<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>hlt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX,rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>leaq/v, rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovq/rA, D64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovq D64/rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPh-rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Y86-64 Instructions

• 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
  • Each accesses and modifies some part(s) of the program state
### Y86-64 Instruction Set #2

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Y86-64 Instruc(on</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
</tr>
<tr>
<td>Set</td>
<td>#4</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
</tr>
</tbody>
</table>

### Y86-64 Instruction Set #3

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Y86-64 Instruc(on</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
</tr>
<tr>
<td>Set</td>
<td>#2</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
<td>&amp;ncol;</td>
</tr>
</tbody>
</table>

### Instruction Example

- **Addition Instruction**

  ![Generic Form](image)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function Code (fn)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add qA, qB</code></td>
<td><code>0 0 [hex]</code></td>
</tr>
</tbody>
</table>

  - Add value in register `qA` to that in register `qB`
  - Store result in register `qB` (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**
  - `add qA, qB` 0 0 [hex]

- **Subtract (qA from qB)**
  - `subq qA, qB` 0 1 [hex]

- **And**
  - `andq qA, qB` 0 2 [hex]

- **Exclusive-Or**
  - `xorq qA, qB` 0 3 [hex]

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

### Encoding Registers

- Each register has a 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
Move Operations

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

Move Instruction Examples

- X86-64
  - movq $0xabcd, %rdx
  - Encoding: 30 82 cd ab 00 00 00 00 00 00

- Y86-64
  - movq %rsp, %rbx
  - Encoding: 50 15 64 ff ff ff ff ff ff ff

Conditional Move Instructions

- Refer to generically as "cmovXX"
- Encodings differ only by "function code"
- Based on values of condition codes
- Variants of xmovq instruction
  - (Conditionally) copy value from source to destination register

Jump Instructions

- Refer to generically as "jXX"
- Encodings differ only by "function code" %n
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in x86-64

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 %esp
  - 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

- `call Dest` RA 0
  - Push address of next instruction onto stack
  - Start executing instructions at Dest
  - Like x86-64

- `set` 0
  - Pop value from stack
  - Use as address for next instruction
  - Like x86-64

Miscellaneous Instructions

- `nop` 0
  - Don’t do anything

- `halt` 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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK</td>
<td>1</td>
</tr>
<tr>
<td>HLT</td>
<td>2</td>
</tr>
<tr>
<td>AOK</td>
<td>3</td>
</tr>
<tr>
<td>INS</td>
<td>4</td>
</tr>
</tbody>
</table>

- 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

Writing Y86-64 Code

- Try to Use C Compiler as Much as Possible
  - 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
  ```
  long len(int a[])
  {
    long len = 0;
    for (len = 0; a[len]; len++);
    return len;
  }
  ```
  ```
  L3:
  addq $1, %rax
  cmpq $0, (%rdi, %rax, 8)
  jne L3
  ```

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

- Compile with `gcc -Og -S`
Y86-64 Code Generation Example #2

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

```c
long len2(long *a) {
    long ip = (long) a;
    long val = *(long *) ip;
    long len = 0;
    while (val) {
        ip += sizeof(long);
        val = *(long *) ip;
    }
    return len;
}
```

• Result
  • Compiler generates exact same code as before!
  • Compiler converts both versions into same intermediate form

Y86-64 Sample Program Structure #1

```c
Init:  # Initialization
    ...
    call Main
    halt
.align 8  # Program data
Array:
    ...
    Main:  # Main function
        ...
    call len ...
Len:  # Length function
    ...
    .pos 0x100  # Placement of stack
Stack:
    ...
```

• Program starts at address 0
• Must set up stack
• Where located
• Pointer values
• Make sure don't overwrite code!
• Must initialize data

Y86-64 Program Structure #2

```c
init:
    # Set up stack pointer
    irmovq Stack, %rsp
    # Execute main program
    call Main
    # Terminate
    halt
```

• Program starts at address 0
• Must set up stack
• Must initialize data
• Can use symbolic names

Y86-64 Program Structure #3

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

• Set up call to len
• Follow x86-64 procedure conventions
• Push array address as argument

Y86-64 Code Generation Example #3

```c
len:
    irmovq $1, %r8          # Constant 1
    irmovq $8, %r9          # Constant 8
    irmovq $0, %rax         # len = 0
    mrmovq (%rdi), %rdx    # val = *a
    andq %rdx, %rdx         # Test val
    je Done                 # If zero, goto Done
    Loop:
        addq %r8, %rax     # len++
        addq %r9, %rdi     # a++
        mrmovq (%rdi), %rdx # val = *a
        andq %rdx, %rdx    # Test val
        jne Loop           # If !0, goto Loop
    Done:
        ret
```

Assembling Y86-64 Program

```bash
unix> yas
len.ya
```

• Generates "object code" file len.ya
• Actually looks like disassembler output
### Simulating Y86-64 Program

```
unix> yis
len: 10
```

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

```bash
Stopped in 33 steps at PC = 0xa13. Status 'HLT'. CC E=1 D=0 O=0
```

Changes to registers:
- `$rax`: 0x0000000000000004
- `$ksp`: 0x0000000000000010
- `$esi`: 0x0000000000000038
- `$edi`: 0x0000000000000001
- `%r9`: 0x0000000000000008

Changes to memory:
- `0x0000000000000000`
- `0x0000000000000013`

### MIPS Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>Constant 0</td>
<td></td>
</tr>
<tr>
<td>$1</td>
<td>Reserved Temp.</td>
<td></td>
</tr>
<tr>
<td>$2</td>
<td>Return Values</td>
<td></td>
</tr>
<tr>
<td>$3</td>
<td>Procedure arguments</td>
<td></td>
</tr>
<tr>
<td>$4</td>
<td>Caller Save</td>
<td></td>
</tr>
<tr>
<td>$5</td>
<td>Temporaries: May be overwritten by called procedures</td>
<td></td>
</tr>
<tr>
<td>$6</td>
<td>Global Pointer</td>
<td></td>
</tr>
<tr>
<td>$7</td>
<td>Stack Pointer</td>
<td></td>
</tr>
<tr>
<td>$8</td>
<td>Caller Save Temp</td>
<td></td>
</tr>
<tr>
<td>$9</td>
<td>Reserved for Operating Sys</td>
<td></td>
</tr>
<tr>
<td>$10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### MIPS Instruction Examples

**R-N**

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Rd</th>
<th>00000</th>
<th>Fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>$3,$2,$1</td>
<td># Register add: $3 = $2+$1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**R-I**

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Rd</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>$3,$2, 3145</td>
<td># Immediate add: $3 = $2+3145</td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>$3,$2, 2</td>
<td># Shift left: $3 = $2 &lt;&lt; 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Branch**

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgez</td>
<td>$3,$2,dstart</td>
<td># Branch when $3 = $2</td>
<td></td>
</tr>
</tbody>
</table>

**Load/Store**

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>$3,16($2)</td>
<td># Load Word: $3 = M[$2+16]</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>$3,16($2)</td>
<td># Store Word: M[$2+16] = $3</td>
<td></td>
</tr>
</tbody>
</table>

#### CISC vs. RISC

- **Original Debate**
  - Strong opinions!
  - CISC proponents—easy for compiler, fewer code bytes
  - RISC proponents—better for optimizing compilers, can make run fast with simple chip design
- **Current Status**
  - For desktop processors, choice of ISA not a technical issue
    - With enough hardware, can make anything run fast
    - Code compatibility more important
  - x86-64 adopted many RISC features
    - More registers; use them for argument passing
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power
    - Most cell phones use ARM processor
Summary

- Y86-64 Instruction Set Architecture
  - Similar state and instructions as x86-64
  - Simpler encodings
  - Somewhere between CISC and RISC
- How Important is ISA Design?
  - Less now than before
    - With enough hardware, can make almost anything go fast