Computer Architecture: Instruction Set Architecture

CSci 2021: Machine Architecture and Organization
October 24th-26th, 2018

Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron

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

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 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 #1

Byte: 0 1 2 3 4 5 6

<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>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXXX a, b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lmrnq V, d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmrnq a, D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmrnq D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmrnq D (rB), a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmrnq D(rB), D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
</tr>
<tr>
<td>popq rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Y86-64 Instruction Set #2

Byte: 0 1 2 3 4 5 6

<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>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXXX a, b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lmrnq V, d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmrnq a, D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmrnq D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmrnq D (rB), a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmrnq D(rB), D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmrnq D(rB), D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
</tr>
<tr>
<td>popq rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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

### Instruction Example

**Addition Instruction**

**Generic Form**

- 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

**Encoded Representation**

```
addq rA, rB 60 06
```

### Arithmetic and Logical Operations

- Refer to generically as “OOpq”
- 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>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq rA, rB</code></td>
<td>60 06</td>
<td></td>
</tr>
<tr>
<td><code>subq rA, rB</code></td>
<td>61 06</td>
<td></td>
</tr>
<tr>
<td><code>andq rA, rB</code></td>
<td>62 06</td>
<td></td>
</tr>
<tr>
<td><code>xorq rA, rB</code></td>
<td>63 06</td>
<td></td>
</tr>
</tbody>
</table>

### Move Operations

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

<table>
<thead>
<tr>
<th>Instruction Code</th>
<th>Register → Register</th>
<th>Immediate → Register</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>movq rA, rB</code></td>
<td>rA</td>
<td>rB</td>
</tr>
<tr>
<td><code>movq rA, D[rB]</code></td>
<td>rA</td>
<td>D[rB]</td>
</tr>
<tr>
<td><code>movq rA, D[rB]</code></td>
<td>D[rB]</td>
<td>rA</td>
</tr>
<tr>
<td><code>movq rA, [rB]</code></td>
<td>D[rB]</td>
<td>rA</td>
</tr>
<tr>
<td><code>movq rA, [rB]</code></td>
<td>D[rB]</td>
<td>rA</td>
</tr>
</tbody>
</table>
Move Instruction Examples

X86-64     Y86-64
mov $0xabcd, rdx    imovq $0xabcd, rdx
Encoding: 30 82 cd ab 00 00 00 00 00 00 00
movq rdx, rdx    imovq rdx, rdx
Encoding: 20 43
movq -12(rsp), rdx    imovq -12(rsp), rdx
Encoding: 50 15 e4 ff ff ff ff ff ff ff
movq [rdx], rdx    fmovq [rdx], rdx
Encoding: 40 64 1c 04 00 00 00 00 00 00

Conditional Move Instructions

Move Unconditionally

rrmovq rA, rB
movq rA, rB
Encoding: 20 43

Jump Instructions

Jump (Conditionally)

jXX Dest    jXX Dest

- 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

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

Jump Instructions

Jump Unconditionally

jmp Dest
jle Dest
jl Dest
je Dest
jne Dest
jge Dest
jg Dest
Encoding: 7 0

Stack Operations

pushq rA
Encoding: 31 90

- Decrement %esp by 8
- Store word from rA to memory at %esp
- Like x86-64

popq rA
Encoding: 31 90

- Read word from memory at %esp
- Save in rA
- Increment %esp 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
```

Ret

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

```
ret
```

Miscellaneous Instructions

- Don’t do anything

```
nop
```

- 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

```
halt
```

Status Conditions

<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>

- 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, alas

Coding Example

- Find number of elements in null-terminated list

```
int len1(int a[])
{
    long len;
    for (len = 0; a[len]; len++)
        return len;
}
```

```
L3:
addq $1,%rax
cmpq $0, (%rdi,%rax,8)
je L3
```

Result

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

Y86-64 Code Generation Example #2

Second Try

- Write C code that mimics expected Y86-64 code

```
long len2(long *a)
{
    long ip = (long) a;
    long val = *(long *) ip;
    long len = 0;
    while (val)
    {
        ip += sizeof(long);
        len++;
        val = *(long *) ip;
    }
    return len;
}
```
Y86-64 Code Generation Example #3

```
len:
  irmovq $1, %r8          # Constant 1
  irmovq $8, %r9          # Constant 8
  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
```

Y86-64 Sample Program Structure #1

```
it:          # Initialization
  .align 8    # Program data
  .section .data
  array:      # Array of 4 elements + terminating 0
    .align 8
    .quad 0x000d000d000d000d
    .quad 0x00c000c000c000c0
    .quad 0x0b000b000b000b00
    .quad 0xa000a000a000a000
Main:        # Main function
  .align 8    # Program data
  .section .data
  array:      # Array of 4 elements + terminating 0
    .align 8
    .quad 0x000d000d000d000d
    .quad 0x00c000c000c000c0
    .quad 0x0b000b000b000b00
    .quad 0xa000a000a000a000
array:
```

Y86-64 Program Structure #2

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

Y86-64 Program Structure #3

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

Assembling Y86-64 Program

```
unix> yas len.ys

set up call to len
Follow x86-64 procedure conventions
Push array address as argument
```

Simulating Y86-64 Program

```
unix> yis len.yo

Stopped in 33 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000004
%rdi:   0x0000000000000000      0x0000000000000038
%r8:    0x0000000000000000      0x0000000000000001
%r9:    0x0000000000000000      0x0000000000000008
Changes to memory:
0x00f0: 0x0000000000000000      0x0000000000000053
0x00f8: 0x0000000000000000      0x0000000000000010
```
ChimeIn break: missing in Y86-64

The following x86-64 instructions don’t exist in Y86-64.
Which one would be hardest to replace with a sequence of Y86-64 instructions?
- nolq
- neq
- testq
- jae
- shlq
- shrq
- leaq  
  [https://chimein.cla.umn.edu/course/view/2021](https://chimein.cla.umn.edu/course/view/2021)
- jmp *$rax

RISC Instruction Sets

- Reduced Instruction Set Computer
- Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)

Fewer, simpler instructions
- Might take more to get a given task done
- Can execute them with small and fast hardware

Register-oriented instruction set
- Many more (typically 32) registers
- Use for arguments, return pointer, temporaries

Only load and store instructions can access memory
- Similar to Y86-64 `mmovq` and `mmovq`

No Condition codes
- Test instructions return 0/1 in register

MIPS Instruction Examples

<table>
<thead>
<tr>
<th>R-R</th>
<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>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ra</td>
<td>Rb</td>
<td>Rd</td>
<td>00000</td>
<td>00</td>
</tr>
<tr>
<td>addu</td>
<td>$3,$2,3145</td>
<td># Immediate add: $3 = $2+3145</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ra</td>
<td>Rb</td>
<td>Rd</td>
<td>00000</td>
<td>01</td>
</tr>
<tr>
<td>sll</td>
<td>$3,$2,2</td>
<td># Shift left: $3 = $2 &lt;&lt; 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00000</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>$3,$2,dest</td>
<td># Branch when $3 = $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00000</td>
<td>01</td>
</tr>
<tr>
<td>lw</td>
<td>$3,16($2)</td>
<td># Load Word: $3 = M[$2+16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ra</td>
<td>Rb</td>
<td>Rd</td>
<td>00000</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>$3,16($2)</td>
<td># Store Word: M[$2+16] = $3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ra</td>
<td>Rb</td>
<td>Rd</td>
<td>00000</td>
<td>00</td>
</tr>
</tbody>
</table>

CISC Instruction Sets

- Complex Instruction Set Computer
- I32 is example

Stack-oriented instruction set
- Use stack to pass arguments, save program counter
- Explicit push and pop instructions

Arithmetic instructions can access memory
- `addq %rax, 12 (%rbx,%rcx,8)`  
  - requires memory read and write
- Complex address calculation

Condition codes
- Set as side effect of arithmetic and logical instructions

Philosophy
- Add instructions to perform “typical” programming tasks

MIPS Registers

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>045</td>
<td>077</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
</tr>
</tbody>
</table>

MIPS 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 processors
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