Computer Architecture: Instruction Set Architecture

CSci 2021: Machine Architecture and Organization
Lecture #16, February 24th, 2016

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 Instruction Set #1

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 Instructions

<table>
<thead>
<tr>
<th>Byte</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></td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovq %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lmovq V, %rB</td>
<td>%rB</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovq %rA, D(%rB)</td>
<td>%rA</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovq %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovz %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovle %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovl %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmove %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovne %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovge %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovg %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Y86-64 Instruction Set #2

<table>
<thead>
<tr>
<th>Byte</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></td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovq %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lmovq V, %rB</td>
<td>%rB</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovq %rA, D(%rB)</td>
<td>%rA</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovq %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovz %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovle %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovl %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmove %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovne %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovge %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovg %rA, %rB</td>
<td>%rA</td>
<td>%rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></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>
<td></td>
</tr>
<tr>
<td>movq X, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpq X, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq X, D(rB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq rA, rB, SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq rA, rB, OF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpq rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq rA, rB, SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq rA, rB, OF</td>
<td></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>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Y86-64 Instruction Set #4

<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>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></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>
<td></td>
</tr>
<tr>
<td>cmpq X, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq X, D(rB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq rA, rB, SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq rA, rB, OF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpq rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq rA, rB, SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq rA, rB, OF</td>
<td></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>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td></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**

- 60 06

### Arithmetic and Logical Operations

**Instruction Code**

- Add
  - `addq rA, rB`
- Subtract (rA from rB)
  - `subq rA, rB`
- And
  - `andq rA, rB`
- Exclusive-Or
  - `xorq rA, rB`

**Function Code**

- 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

### Move Operations

**Register ➔ Register**

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

**Immediate ➔ Register**

**Register ➔ Memory**

**Memory ➔ Register**
Move Instruction Examples

X86-64

`mov $0xabcd, rdx`

Encoding: 30 82 cd ab 00 00 00 00 00 00

`mov rdx, rdx`

Encoding: 20 43

Y86-64

`movq $0xabcd, %rdx`

Encoding: 50 15 e4 00 00 00 00 00 00 00 00

`movq %rdx, %rdx`

Encoding: 40 64 2c 04 00 00 00 00 00 00

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`

Jump Instructions

Jump (Conditionally)

`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

Jump Instructions

Jump Unconditionally

`jmp Dest`

Jump When Less or Equal

`jle Dest`

Jump When Less

`jl Dest`

Jump When Equal

`je Dest`

Jump When Not Equal

`jne Dest`

Jump When Greater or Equal

`jge Dest`

Jump When Greater

`jg 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
```

```
ret
```

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

Miscellaneous Instructions

- Don't do anything

```
rop
```

- 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:
```

```c
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

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

Y86-64 Sample Program Structure #1

```assembly
init:                 # Initialisation
  ...                # Program starts at address 0
  call Main          # Must set up stack
  halt               # Where located
  .align 8           # Pointer values
  array:             # Make sure don't overwrite code!
  ...               # Must initialize data
Main:                 # Main function
  ...                # Program data
  call len ...      # Placement of stack
len:                  # Length function
  ...                # Program data
  .pos 0x100         # Stack:
```

Y86-64 Program Structure #2

```assembly
init:                 # Program starts at address 0
  # Set up stack pointer
  irmovq,%r8           # Execute main program
  call Main
  # Call main program
  halt                 # Array: 0
  # Array of 4 elements + terminating 0
  .align 8             # Program data
Array:                 # Program data
  .quad 0x0000000000000000
  .quad 0x0000000000000001
  .quad 0x0000000000000002
  .quad 0x0000000000000003
  .quad 0
```

Y86-64 Program Structure #3

```assembly
init:                 # Program starts at address 0
  # Set up stack pointer
  irmovq,%r8           # Execute main program
  call array
  # Call main program
  halt                 # Array:
Main:                 # Main function
  ...                # Program data
  array %r8
  # Call len(array)
  call len
  ...                # Placement of stack
set up call to len:   # Stack:
  ...                # Program data
  ...                # Program data
  ...                # Program data
```

Assembling Y86-64 Program

```bash
unix> yas len.yo
```

Simulating Y86-64 Program

```bash
unix> yis len.yo
```

- Generates "object code" file len.o
- Actually looks like disassembler output

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

stopped in 33 steps at PC = 0x13. Status "HLT", CC Z=1 S=0 O=0
Changes to registers:
  %rax: 0x0000000000000000 0x0000000000000001 0x0000000000000002 0x0000000000000003
  %rdx: 0x0000000000000000 0x0000000000000001 0x0000000000000002 0x0000000000000003
  %rdi: 0x0000000000000000 0x0000000000000001 0x0000000000000002 0x0000000000000003
Changes to memory:
  0x0000: 0x0000000000000000 0x0000000000000001 0x0000000000000002 0x0000000000000003
  0x0004: 0x0000000000000000 0x0000000000000001 0x0000000000000002 0x0000000000000003
  0x0008: 0x0000000000000000 0x0000000000000001 0x0000000000000002 0x0000000000000003
```
CISC Instruction Sets
- Complex Instruction Set Computer
- IA32 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

Ra
Add instructions to perform “typical” programming tasks

Rb
Use for arguments, return pointer, temporaries

With enough hardware, can make almost anything go

MIPS Registers

<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>
<tr>
<td>addu</td>
<td>$3,$2,3145</td>
<td>Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sll</td>
<td>$3,$2,3145</td>
<td>Immediate add: $3 = $2&lt;&lt;1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>$3,$2,dest</td>
<td># Branch when $3 = $2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load/Store

MIPS Instruction Examples

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 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