**Instruction Set Architecture**

**Assembly Language View**
- Processor state
  - Registers, memory, ...
- Instructions
  - add, 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
- **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>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lmovq V, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovq rA, D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovc rA, rB</td>
<td></td>
<td></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>
<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></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></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></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>
<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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**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>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
</tr>
<tr>
<td>lmovq V, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovq rA, D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovc rA, rB</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>
<tr>
<td>jXX Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
</tr>
<tr>
<td>ret</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

<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>r15</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

### Arithmetic and Logical Operations

<table>
<thead>
<tr>
<th>Instruction Code</th>
<th>Function Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq rA, rB</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>subq rA, rB</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>andq rA, rB</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>xorq rA, rB</td>
<td>0 1 2 3 4 5 6 7</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

<table>
<thead>
<tr>
<th>Instruction Code</th>
<th>Function Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq rA, rB</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>movq V, rB</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>movq D(rB), rA</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>movq D(rB), rA</td>
<td>0 1 2 3 4 5 6 7</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

- X86-64
  - mov $0xabcd, rdx
  - Encoding: 30 82 cd ab 00 00 00 00 00
- Y86-64
  - movq $0xabcd, %rdx
  - Encoding: 40 64 1c 04 00 00 00 00 00

Conditional Move Instructions

- Move Unconditionally
  - cmpmovq rA, rB
  - Encoding: 22 1a 0b 0b
- Move When Less or Equal
  - cmovle rA, rB
  - Encoding: 22 1a 0b 03
- Move When Less
  - cmovl rA, rB
  - Encoding: 22 1a 0b 01
- Move When Equal
  - cmove rA, rB
  - Encoding: 22 1a 0b 02
- Move When Not Equal
  - cmove rA, rB
  - Encoding: 22 1a 0b 05
- Move When Greater or Equal
  - cmovge rA, rB
  - Encoding: 22 1a 0b 06
- Move When Greater
  - cmovg rA, rB
  - Encoding: 22 1a 0b 04

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

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

ret

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

call Dest

Dest

Miscellaneous Instructions

- Don't do anything

halt

- 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

nop

- Halt instruction encountered

HLT

- Bad address (either instruction or data) encountered

ADR

- Invalid instruction encountered

INS

Status Conditions

Mnemonic | Code
---|---
ADR | 3
HLT | 2
ADR | 3
INS | 4

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

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

Y86-64 Code Generation Example

First Try
- Write typical array code
- Try to use C compiler as much as possible

```c
/* Find number of elements in null-terminated list */
long len2(long a[])
{
    long len;
    for (len = 0; a[len]; len++)
        ;
    return len;
}
```

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

```c
addq $1,%rax
```
```c
cmpq $0, (%rdi,%rax,8)
```
```c
jne L3
```

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);
        len++;
        val = *(long*) ip;
    }
    return len;
}
```

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

```c
addq $1,%rax
```
```c
cmpq $0, (%rdi,%rax,8)
```
```c
jne L3
```
Y86-64 Code Generation Example #3

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

Y86-64 Sample Program Structure #1

```
init:       # Initialization
  call Main
  halt

.align 8   # Program data
array:     
  .quad 0
Main:       # Main function
  .call len ...
  .call len ...
len:        # Length function
  .pos 0x100   # Placement of stack
Stack:
```

Y86-64 Program Structure #2

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

`Array:
  .quad 0
  .quad 0
  .quad 0
  .quad 0
```

Y86-64 Program Structure #3

```
Main:
  irmovq array, %rdi
  # call len(array)
  # call len ...
  # call len ...
  .pos 0x100   # Placement of stack
Stack:
```

Assembling Y86-64 Program

```
unix> yas len.ys
```

Simulating Y86-64 Program

```
unix> yis len.ys
```
Think & chat: 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?

- notq
- negq
- testq
- jae
- shlq
- shrq
- leaq
- jmp %rax

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?

- notq → XOR with -1
- negq → subtract from 0
- testq → AND to scratch register
- jae → subtract TMin from both sides, then cmpjge
- shlq → add to itself = left shift by one
- shrq → via rotate-left, or by-byte table lookup
- leaq → combination of shl (above) and addition
- jmp %rax → push and then return

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

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 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 mrmovq and rmmovq

No Condition codes

- Test instructions return 0/1 in register

MIPS Registers

- Constant 0
- Reserved Temp.
- Return Values
- Procedure arguments
- Caller Save
- Temporaries: May be overwritten by called procedures
- Call Save Temp
- Reserved for Operating Sys
- Global Pointer
- Stack Pointer
- Call Save Temp
- Return Address

MIPS Instruction Examples

- Add $3, $2, $1 # Register add: $3 = $2+$1
- Add $3, $2, 3145 # Immediate add: $3 = $2+3145
- Add $3, $2 # Shift left: $3 = $2 << 2
- Beq $3, $2, dest # Branch when $3 = $2
- Lw $3,16($2) # Load Word: $3 = M[$2+16]
- Sw $3,16($2) # Store Word: M[$2+16] = $3
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