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 #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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<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>

- halt
- nop
- cmovXX rA, rB
- limovq V, rB
- zmovq rA, D[rB]
- zmovq D[rA], rB
- cmovq rA, rB
- irmovq V, rB
- rmmovq rA, D[rB]
- mrmovq D[rB], rA
- OPq rA, rB
- ret
- pushq rA
- popq rA

- Be

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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<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>

- halt
- nop
- cmovXX rA, rB
- limovq V, rB
- zmovq rA, D[rB]
- zmovq D[rA], rB
- cmovq rA, rB
- irmovq V, rB
- rmmovq rA, D[rB]
- mrmovq D[rB], rA
- OPq rA, rB
- ret
- pushq rA
- popq rA

- Be

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

- 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

Arithmetic and Logical Operations

Instruction Code | Function Code | Add

- addq rA, rB
- subq rA, rB
- andq rA, rB
- xorq rA, rB

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** | **Y86-64**  
---|---  
mov $0xabcd, %rdx | lemov $0xabcd, %rdx  
Encoding: 30 82 cd ab 00 00 00 00 00 00 00 00  
mov $0xabcd, %rdx | lemov $0xabcd, %rdx  
Encoding: 30 82 3e 00 00 00 00 00 00 00 00  
movq $0xabcd, %rdx | lemovq $0xabcd, %rdx  
Encoding: 30 82 cd ab 00 00 00 00 00 00 00 00  
movq $0xabcd, %rdx | lemovq $0xabcd, %rdx  
Encoding: 30 82 3e 00 00 00 00 00 00 00 00  
movl $0xabcd, %rdx | lemovl $0xabcd, %rdx  
Encoding: 30 89 cd ab 00 00 00 00 00 00 00 00  
movl $0xabcd, %rdx | lemovl $0xabcd, %rdx  
Encoding: 30 89 3e 00 00 00 00 00 00 00 00  
movl $0xabcd, %rdx | lemovl $0xabcd, %rdx  
Encoding: 30 89 cd ab 00 00 00 00 00 00 00 00  
movl $0xabcd, %rdx | lemovl $0xabcd, %rdx  
Encoding: 30 89 3e 00 00 00 00 00 00 00 00

### Conditional Move Instructions

**Move Unconditionally**  
- rrmovq %rsp, %rbx  
- movq %rsp, %rbx  
- rmmovq -12(%rbp), %rcx  
- movq -12(%rbp), %rcx  
- rmmovq %rsi, 0x41c(%rsp)  
- movq %rsi, 0x41c(%rsp)  

- 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

### Jump Instructions

**Jump (Conditionally)**  
- jmp Dest  
- jle Dest  
- jl Dest  
- je Dest  
- jne Dest  
- jge Dest  
- jg 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

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

- 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

```plaintext
call Dest 8 0
```

Ret

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

```plaintext
ret 9 0
```

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

```plaintext
nop 1 0
halt 0 0
```

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

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
- Desired Behavior: If AOK, keep going; Otherwise, stop program execution

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

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

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 #2

Second Try
- Write C code that mimics expected Y86-64 code
- Compiler generates exact same code as before!
- Compiler converts both versions into same intermediate form
**Y86-64 Code Generation Example #3**

```
len:
  irmovq $1, %r8          # Constant 1
  irmovq $8, %r9          # Constant 8
  irmovq (%rdi), %rax    # If zero, goto Done
  jne Loop               # If != 0, goto Loop
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**

```
init:  # Initialisation
  .align 8  # Program data
array:  # Main function
  .quad 0x000d000d000d000d  # Array of 4 elements + terminating 0
  .quad 0x00c000c000c000c0
  .quad 0x0b000b000b000b00
  .quad 0xa000a000a000a000
  .quad 0
  call Main
  ret
```

**Y86-64 Program Structure #2**

```
init:  # Set up stack pointer
  irmovq Stack, %rsp     # Execute main program
  call Main
  .align 8  # Program data
Array:  # Main function
  .quad 0x000d000d000d000d  # Array of 4 elements + terminating 0
  .quad 0x00c000c000c000c0
  .quad 0x0b000b000b000b00
  .quad 0xa000a000a000a000
  .quad 0
Main:
  irmovq array, %rdi     # call len(array)
  call len
  ret
```

**Assembling Y86-64 Program**

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

**Simulating Y86-64 Program**

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

---

**Y86-64 Program Structure #3**

```
Main:  # Array of 4 elements + terminating 0
  irmovq array, %rdi     # call len(array)
  call len
  ret
```

**Program Structure #1**

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

**Program Structure #2**

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

**Program Structure #3**

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

**Assembling Y86-64 Program**

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

**Simulating Y86-64 Program**

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

---

**Assembling Y86-64 Program**

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

**Simulating Y86-64 Program**

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

---

**Assembling Y86-64 Program**

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

**Simulating Y86-64 Program**

```
unix> yis len.yo
```
Chimeln 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
- neqg
- testq
- jae
- shlg
- shrq
- leaq
- jmp *%rax

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

<table>
<thead>
<tr>
<th>R</th>
<th>Ra</th>
<th>Rb</th>
<th>Rd</th>
<th>Constant 0</th>
<th>Reserved Temp.</th>
<th>Return Values</th>
<th>Procedure arguments</th>
<th>Caller Save Temp</th>
<th>Stack Pointer</th>
<th>Global Pointer</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0a</td>
<td>0b</td>
<td>0c</td>
<td>0d</td>
<td>0e</td>
<td>0f</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>1a</td>
<td>1b</td>
<td>1c</td>
<td>1d</td>
<td>1e</td>
<td>1f</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>2a</td>
<td>2b</td>
<td>2c</td>
<td>2d</td>
<td>2e</td>
<td>2f</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>3a</td>
<td>3b</td>
<td>3c</td>
<td>3d</td>
<td>3e</td>
<td>3f</td>
<td>40</td>
<td>41</td>
</tr>
<tr>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>4a</td>
<td>4b</td>
<td>4c</td>
<td>4d</td>
<td>4e</td>
<td>4f</td>
<td>50</td>
<td>51</td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>5a</td>
<td>5b</td>
<td>5c</td>
<td>5d</td>
<td>5e</td>
<td>5f</td>
<td>60</td>
<td>61</td>
</tr>
<tr>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>6a</td>
<td>6b</td>
<td>6c</td>
<td>6d</td>
<td>6e</td>
<td>6f</td>
<td>70</td>
<td>71</td>
</tr>
<tr>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>7a</td>
<td>7b</td>
<td>7c</td>
<td>7d</td>
<td>7e</td>
<td>7f</td>
<td>80</td>
<td>81</td>
</tr>
<tr>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>8a</td>
<td>8b</td>
<td>8c</td>
<td>8d</td>
<td>8e</td>
<td>8f</td>
<td>90</td>
<td>91</td>
</tr>
<tr>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>9a</td>
<td>9b</td>
<td>9c</td>
<td>9d</td>
<td>9e</td>
<td>9f</td>
<td>a0</td>
<td>a1</td>
</tr>
<tr>
<td>a6</td>
<td>a7</td>
<td>a8</td>
<td>a9</td>
<td>a10</td>
<td>a11</td>
<td>a12</td>
<td>a13</td>
<td>a14</td>
<td>a15</td>
<td>a16</td>
<td>a17</td>
</tr>
<tr>
<td>a22</td>
<td>a23</td>
<td>a24</td>
<td>a25</td>
<td>a26</td>
<td>a27</td>
<td>a28</td>
<td>a29</td>
<td>a30</td>
<td>a31</td>
<td>b0</td>
<td>b1</td>
</tr>
<tr>
<td>b6</td>
<td>b7</td>
<td>b8</td>
<td>b9</td>
<td>b10</td>
<td>b11</td>
<td>b12</td>
<td>b13</td>
<td>b14</td>
<td>b15</td>
<td>c0</td>
<td>c1</td>
</tr>
<tr>
<td>c6</td>
<td>c7</td>
<td>c8</td>
<td>c9</td>
<td>c10</td>
<td>c11</td>
<td>c12</td>
<td>c13</td>
<td>c14</td>
<td>c15</td>
<td>d0</td>
<td>d1</td>
</tr>
<tr>
<td>d6</td>
<td>d7</td>
<td>d8</td>
<td>d9</td>
<td>d10</td>
<td>d11</td>
<td>d12</td>
<td>d13</td>
<td>d14</td>
<td>d15</td>
<td>e0</td>
<td>e1</td>
</tr>
<tr>
<td>e6</td>
<td>e7</td>
<td>e8</td>
<td>e9</td>
<td>e10</td>
<td>e11</td>
<td>e12</td>
<td>e13</td>
<td>e14</td>
<td>e15</td>
<td>f0</td>
<td>f1</td>
</tr>
</tbody>
</table>

MIPS Instruction Examples

<table>
<thead>
<tr>
<th>R</th>
<th>Ra</th>
<th>Rb</th>
<th>Rd</th>
<th>00000</th>
<th>Fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu $3,$2,$1</td>
<td># Register add: $3 = $2+$1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addu $3,$2,3145</td>
<td># Immediate add: $3 = $2+3145</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sll $3,$2</td>
<td># Shift left: $3 = $2 &lt;&lt; 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $3,16($2)</td>
<td># Load Word: $3 = M[$2+16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $3,16($2)</td>
<td># Store Word: M[$2+16] = $3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chimeln 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
- neqg → subtract from 0
- testq → AND to scratch register
- jae → subtract TMin from both sides, then cmp/jge
- shlg → 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 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

Administrative Notes
- Looks like the midterm may have been harder than we had intended
  - I’ll have more to say about results versus our expectations after it is graded, probably Monday
- Attack lab out today
  - Formerly known as the Buffer lab, now with ROP