

# Computer Systems Organization (CS2.201)

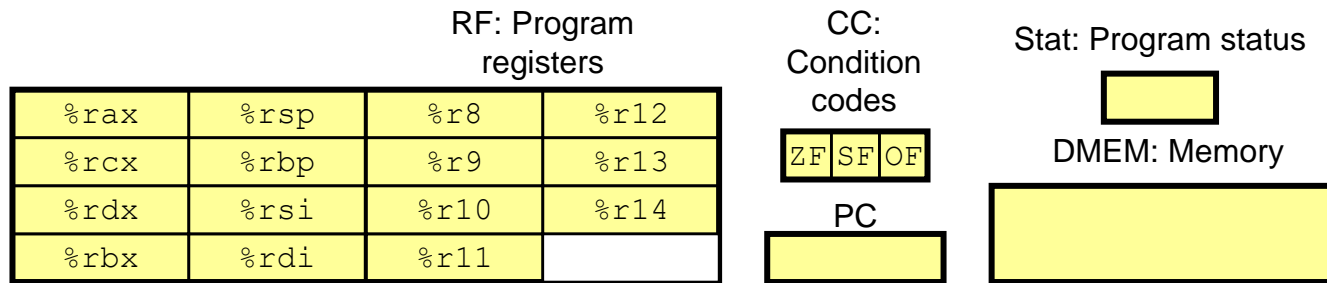
Y86-64: INSTRUCTION SET ARCHITECTURE

---

Deepak Gangadharan  
Computer Systems Group (CSG), IIIT Hyderabad

Slide Contents: Based on materials from text books and other public sources

# 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

---

- Subset of x86-64 instruction set
  - includes only 8-byte integer operations
  - fewer addressing modes (second index register and scaling not supported)
  - No transfer of immediate data to memory
  - smaller set of operations

# Y86-64 Instruction Set #1

| Byte             | 0 | 1  | 2    | 3  | 4 | 5 | 6 | 7 | 8 | 9 |
|------------------|---|----|------|----|---|---|---|---|---|---|
| halt             | 0 | 0  |      |    |   |   |   |   |   |   |
| nop              | 1 | 0  |      |    |   |   |   |   |   |   |
| cmovXX rA, rB    | 2 | fn | rA   | rB |   |   |   |   |   |   |
| irmovq V, rB     | 3 | 0  | F    | rB | V |   |   |   |   |   |
| rmmovq rA, D(rB) | 4 | 0  | rA   | rB | D |   |   |   |   |   |
| mrmovq D(rB), rA | 5 | 0  | rA   | rB | D |   |   |   |   |   |
| OPq rA, rB       | 6 | fn | rA   | rB |   |   |   |   |   |   |
| jXX Dest         | 7 | fn | Dest |    |   |   |   |   |   |   |
| call Dest        | 8 | 0  | Dest |    |   |   |   |   |   |   |
| ret              | 9 | 0  |      |    |   |   |   |   |   |   |
| pushq rA         | A | 0  | rA   | F  |   |   |   |   |   |   |
| popq rA          | B | 0  | rA   | F  |   |   |   |   |   |   |

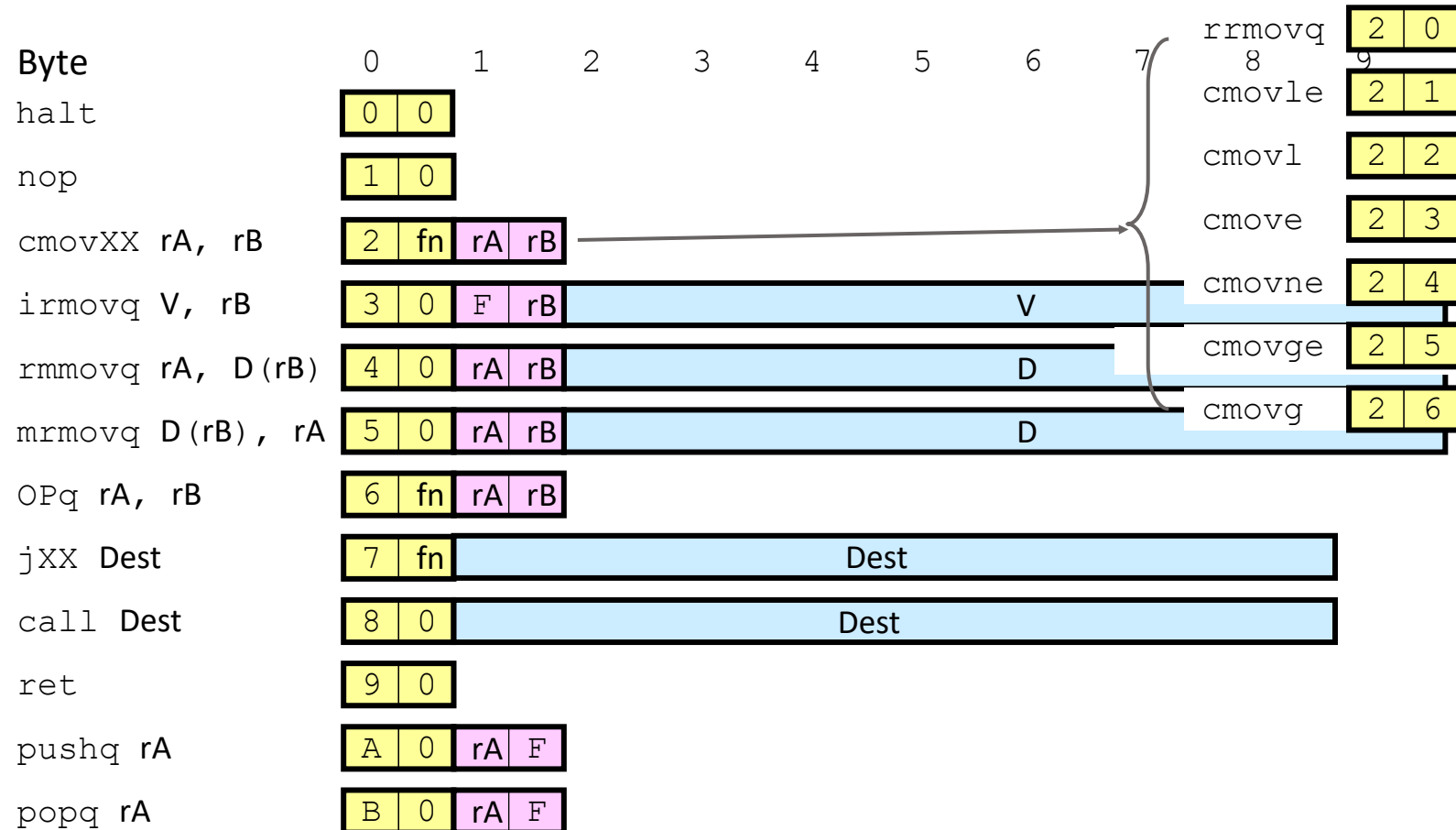
# Y86-64 Instructions

---

## 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



# Y86-64 Instruction Set #3

| Byte             | 0 | 1  | 2    | 3  | 4 | 5 | 6 | 7 | 8    | 9 |   |
|------------------|---|----|------|----|---|---|---|---|------|---|---|
| halt             | 0 | 0  |      |    |   |   |   |   |      |   |   |
| nop              | 1 | 0  |      |    |   |   |   |   |      |   |   |
| cmovXX rA, rB    | 2 | fn | rA   | rB |   |   |   |   |      |   |   |
| irmovq V, rB     | 3 | 0  | F    | rB | V |   |   |   |      |   |   |
| rmmovq rA, D(rB) | 4 | 0  | rA   | rB | D |   |   |   |      |   |   |
| mrmovq D(rB), rA | 5 | 0  | rA   | rB | D |   |   |   |      |   |   |
| OPq rA, rB       | 6 | fn | rA   | rB |   |   |   |   | addq | 6 | 0 |
|                  |   |    |      |    |   |   |   |   | subq | 6 | 1 |
|                  |   |    |      |    |   |   |   |   | andq | 6 | 2 |
| jXX Dest         | 7 | fn | Dest |    |   |   |   |   |      |   |   |
| call Dest        | 8 | 0  | Dest |    |   |   |   |   |      |   |   |
| ret              | 9 | 0  |      |    |   |   |   |   |      |   |   |
| pushq rA         | A | 0  | rA   | F  |   |   |   |   |      |   |   |
| popq rA          | B | 0  | rA   | F  |   |   |   |   |      |   |   |

# Y86-64 Instruction Set #4

| Byte             | 0 | 1  | 2    | 3  | 4 | 5 | 6 | 7 |         |
|------------------|---|----|------|----|---|---|---|---|---------|
| halt             | 0 | 0  |      |    |   |   |   |   | jmp 7 0 |
| nop              | 1 | 0  |      |    |   |   |   |   | jle 7 1 |
| cmovXX rA, rB    | 2 | fn | rA   | rB |   |   |   |   | j1 7 2  |
| irmovq V, rB     | 3 | 0  | F    | rB | V |   |   |   | je 7 3  |
| rmmovq rA, D(rB) | 4 | 0  | rA   | rB | D |   |   |   | jne 7 4 |
| mrmovq D(rB), rA | 5 | 0  | rA   | rB | D |   |   |   | jge 7 5 |
| OPq rA, rB       | 6 | fn | rA   | rB |   |   |   |   | jg 7 6  |
| jXX Dest         | 7 | fn | Dest |    |   |   |   |   |         |
| call Dest        | 8 | 0  | Dest |    |   |   |   |   |         |
| ret              | 9 | 0  |      |    |   |   |   |   |         |
| pushq rA         | A | 0  | rA   | F  |   |   |   |   |         |
| popq rA          | B | 0  | rA   | F  |   |   |   |   |         |



# Encoding Registers

Each register has 4-bit ID

|      |   |             |   |
|------|---|-------------|---|
| %rax | 0 | %r8         | 8 |
| %rcx | 1 | %r9         | 9 |
| %rdx | 2 | %r10        | A |
| %rbx | 3 | %r11        | B |
| %rsp | 4 | %r12        | C |
| %rbp | 5 | %r13        | D |
| %rsi | 6 | %r14        | E |
| %rdi | 7 | No Register | F |

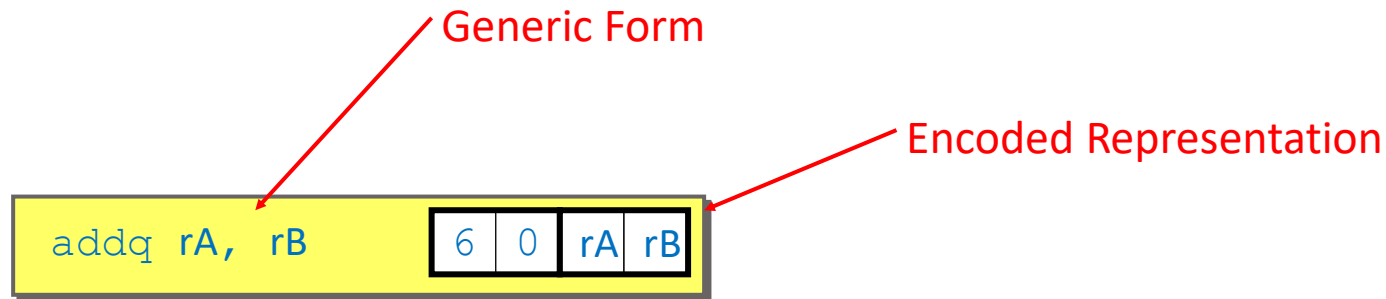
- Same encoding as in x86-64

Register ID 15 (0xF) indicates “no register”

- Will use this in our hardware design in multiple places

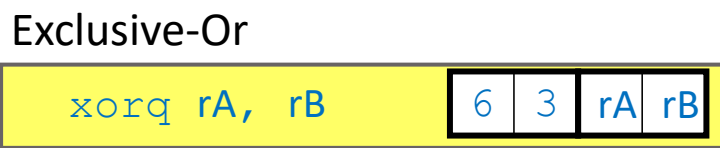
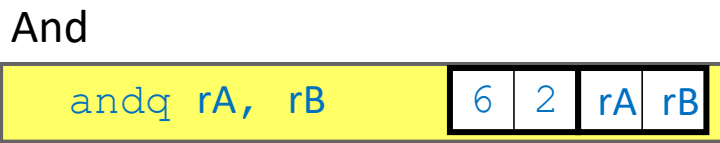
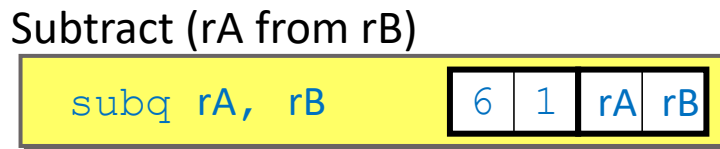
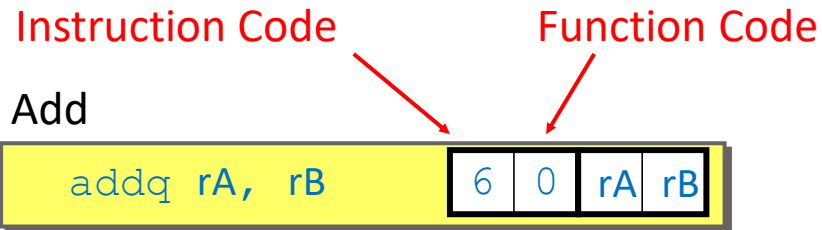
# Instruction Example

## Addition Instruction



- 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

# Arithmetic and Logical Operations



- Refer to generically as “OP<sub>q</sub>”
- Encodings differ only by “function code”
  - Low-order 4 bits in first instruction word
- Set condition codes as side effect

# Move Operations

|                               |   |   |    |    |   |                      |
|-------------------------------|---|---|----|----|---|----------------------|
| <code>rrmovq rA, rB</code>    | 2 | 0 | rA | rB |   | Register → Register  |
| <code>irmovq V, rB</code>     | 3 | 0 | F  | rB | V | Immediate → Register |
| <code>rmmovq rA, D(rB)</code> | 4 | 0 | rA | rB | D | Register → Memory    |
| <code>mrmovq D(rB), rA</code> | 5 | 0 | rA | rB | D | Memory → Register    |

- 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:

```
movq %rsp, %rbx
```

Encoding:

```
movq -12(%rbp), %rcx
```

Encoding:

```
movq %rsi, 0x41c(%rsp)
```

Encoding:

Y86-64

```
irmovq $0xabcd, %rdx
```

30 F2 cd ab 00 00 00 00 00 00

```
rrmovq %rsp, %rbx
```

20 43

```
mrmovq -12(%rbp), %rcx
```

50 15 f4 ff ff ff ff ff ff ff

```
rmmovq %rsi, 0x41c(%rsp)
```

40 64 1c 04 00 00 00 00 00 00

# Conditional Move Instructions

Move Unconditionally



Move When Less or Equal



Move When Less



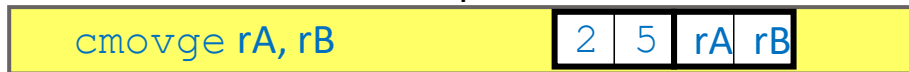
Move When Equal



Move When Not Equal



Move When Greater or Equal



Move When Greater



- 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)



- 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



Jump When Less or Equal



Jump When Less



Jump When Equal



Jump When Not Equal



Jump When Greater or Equal

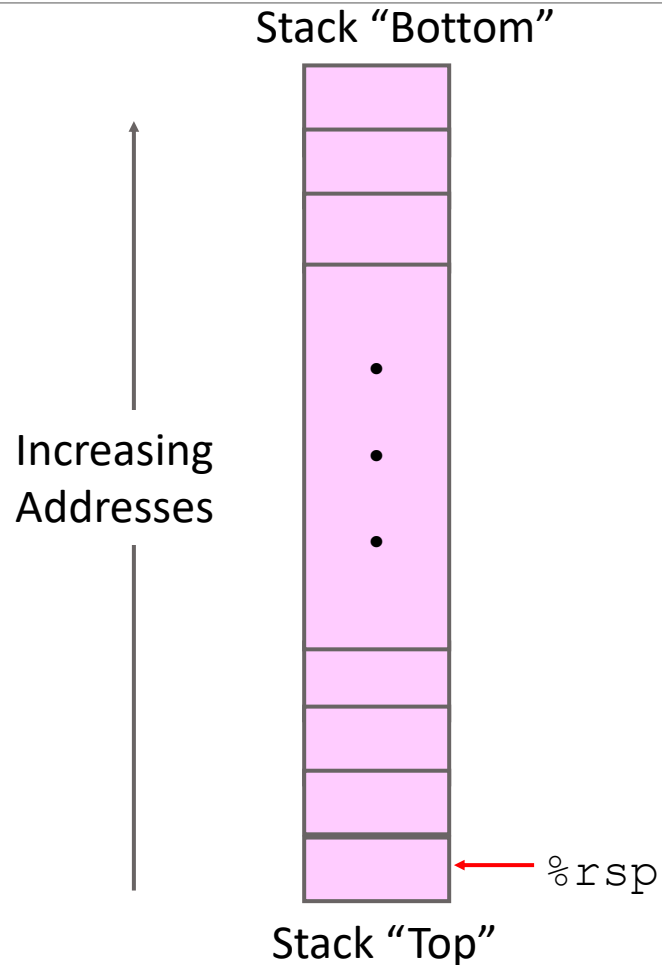


Jump When Greater





# 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`

|   |   |    |   |
|---|---|----|---|
| A | 0 | rA | F |
|---|---|----|---|

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

`popq rA`

|   |   |    |   |
|---|---|----|---|
| B | 0 | rA | F |
|---|---|----|---|

- Read word from memory at `%rsp`
- Save in `rA`
- Increment `%rsp` by 8
- Like x86-64

# Subroutine Call and Return

`call Dest`

8 0

Dest

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

`ret`

9 0

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

# 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

# Status Conditions

---

| Mnemonic | Code |
|----------|------|
| AOK      | 1    |

- Normal operation

| Mnemonic | Code |
|----------|------|
| HLT      | 2    |

- Halt instruction encountered

| Mnemonic | Code |
|----------|------|
| ADR      | 3    |

- Bad address (either instruction or data) encountered

| Mnemonic | Code |
|----------|------|
| INS      | 4    |

- Invalid instruction encountered

## Desired Behavior

- If AOK, keep going
- Otherwise, stop program execution

# Y86-64 program

---

## x86-64 code

```
1  long sum(long *start, long count)
2  {
3      long sum = 0;
4      while (count) {
5          sum += *start;
6          start++;
7          count--;
8      }
9      return sum;
10 }
```

```
1  long sum(long *start, long count)
2  start in %rdi, count in %rsi
3  sum:
4      movl    $0, %eax          sum = 0
5      jmp     .L2              Goto test
6  .L3:
7      addq    (%rdi), %rax      Add *start to sum
8      addq    $8, %rdi         start++
9      subq    $1, %rsi         count--
10     .L2:
11     testq    %rsi, %rsi       Test sum
12     jne     .L3              If !=0, goto loop
13     rep; ret                 Return
```

## Y86-64 code

```
1  long sum(long *start, long count)
2  start in %rdi, count in %rsi
3  sum:
4      irmovq  $8,%r8           Constant 8
5      irmovq  $1,%r9           Constant 1
6      xorq    %rax,%rax        sum = 0
7      andq    %rsi,%rsi        Set CC
8      jmp     test             Goto test
9  loop:
10     mrmovq   (%rdi),%r10      Get *start
11     addq     %r10,%rax        Add to sum
12     addq     %r8,%rdi         start++
13     subq     %r9,%rsi         count--. Set CC
14     test:
15     jne     loop             Stop when 0
16     ret                     Return
```

# 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

---

Thank You!