# ISAs and Y86-64

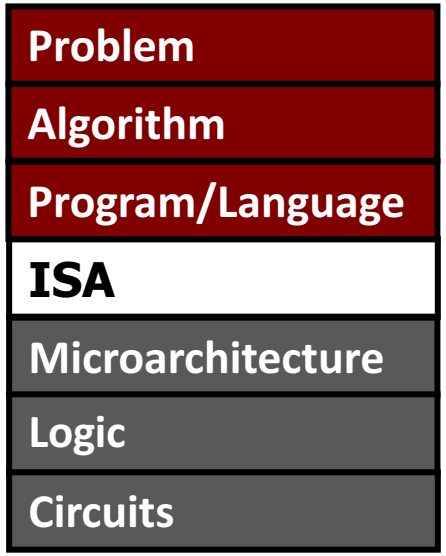
Samira Khan

# Agenda

- ISA vs Microarchitecture
- ISA Tradeoffs
- Y86-64 ISA
- Y86-64 Format
- Y86-64 Encoding/Decoding

# LEVELS OF TRANSFORMATION

- ISA
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write system/user programs
- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - **ISA, uarch**, circuits
  - "Architecture" = ISA + microarchitecture



# ISA VS. MICROARCHITECTURE

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for “acceleration”
  - Internals of the engine: implements “acceleration”
  - Add instruction vs. Adder implementation

- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Bit serial, ripple carry, carry lookahead adders
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, …

- Uarch usually changes faster than ISA
  - Few ISAs (x86, SPARC, MIPS, Alpha) but many uarchs
  - *Why?*

# ISA

- Instructions
  - Opcodes, Addressing Modes
  - Instruction Types and Format
  - Registers, Condition Codes


- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management
- Call, Interrupt/Exception Handling
- Access Control, Priority/Privilege
- I/O
- Task Management
- Power and Thermal Management
- Multi-threading support, Multiprocessor support

# Example ISAs

- x86 — dominant in desktops, servers
- ARM — dominant in mobile devices
- POWER — Wii U, IBM supercomputers and some servers
- MIPS — common in consumer wifi access points
- SPARC — some Oracle servers, Fujitsu supercomputers
- z/Architecture — IBM mainframes
- Z80 — TI calculators
- SHARC — some digital signal processors
- Itanium — some HP servers (being retired)
- RISC V — some embedded
- …

# Agenda

- ISA vs Microarchitecture
- ISA Tradeoffs
- Y86-64 ISA
- Y86-64 Format
- Y86-64 encoding/decoding

# ISA: INSTRUCTION LENGTH

- Fixed length: Length of all instructions the same
  - + Easier to decode single instruction in hardware
  - + Easier to decode multiple instructions concurrently
  - -- Wasted bits in instructions (Why is this bad?)
  - -- Harder-to-extend ISA (how to add new instructions?)

- Variable length: Length of instructions different (determined by opcode and sub-opcode)
  - + Compact encoding (Why is this good?)
    - Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. How?
  - -- More logic to decode a single instruction
  - -- Harder to decode multiple instructions concurrently

# ISA: ADDRESSING MODES

- Addressing mode specifies how to obtain an operand of an instruction
  - Register
  - Immediate
  - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, …)

- x86-64: `10(%r11,%r12,4)`
- ARM: `%r11 << 3` (shift register value by constant)
- VAX: `((%r11))` (register value is pointer to pointer)

# ISA: Condition Codes

```
cmpq %r11, %r12
je somewhere
```

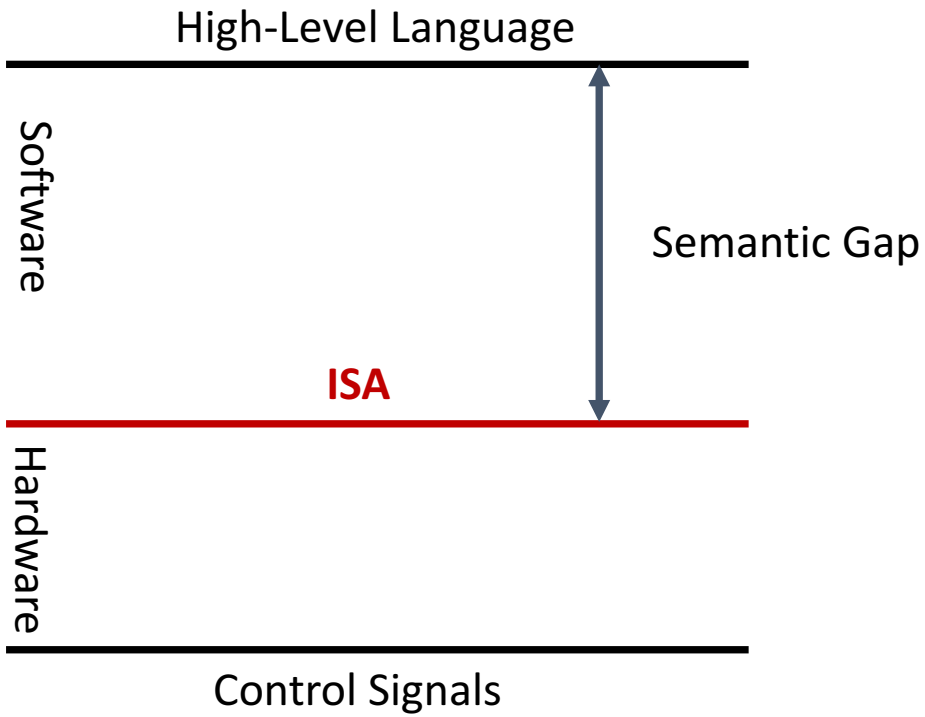
- could do:

```
/* _Branch if _EQual */
beq %r11, %r12, somewhere
```

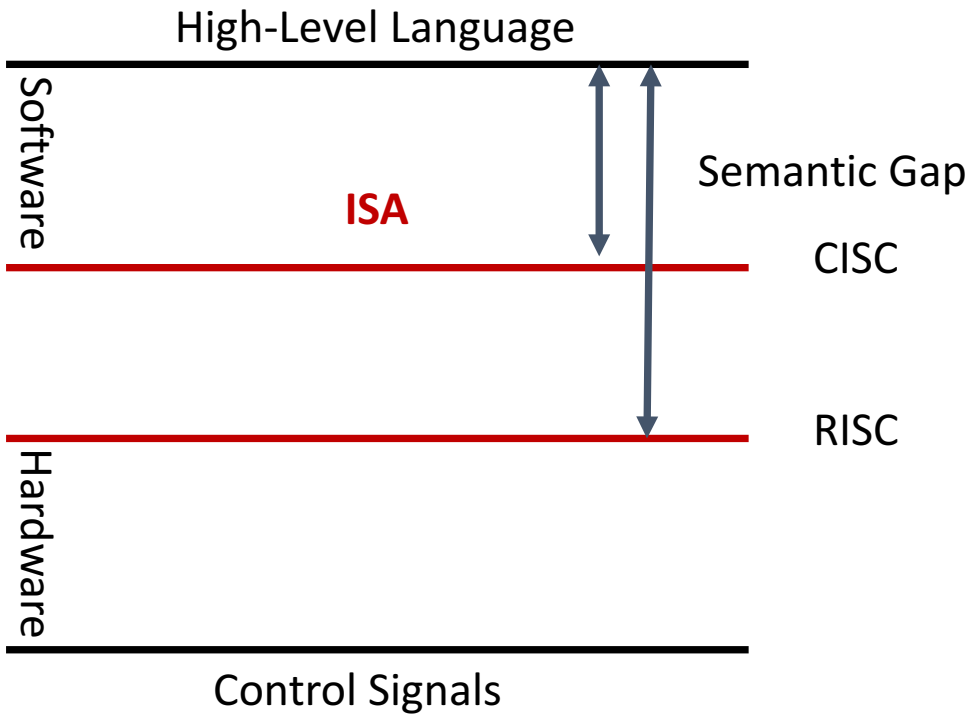
# ISA-LEVEL TRADEOFFS: SEMANTIC GAP

- Where to place the ISA? Semantic gap
  - Closer to high-level language (HLL) or closer to hardware control signals? → Complex vs. simple instructions
  - RISC vs. CISC vs. HLL machines
    - FFT, QUICKSORT, POLY, FP instructions?
    - VAX INDEX instruction (array access with bounds checking)
      - e.g., A[i][j][k] one instruction with bound check

# SEMANTIC GAP



# SEMANTIC GAP



# ISA-LEVEL TRADEOFFS: SEMANTIC GAP

- Where to place the ISA? Semantic gap
  - Closer to high-level language (HLL) or closer to hardware control signals? → Complex vs. simple instructions
  - RISC vs. CISC vs. HLL machines
    - FFT, QUICKSORT, POLY, FP instructions?
    - VAX INDEX instruction (array access with bounds checking)
  - Tradeoffs:
    - Simple compiler, complex hardware vs. complex compiler, simple hardware
    - Burden of backward compatibility
    - Performance?
      - Optimization opportunity: Example of VAX INDEX instruction: who (compiler vs. hardware) puts more effort into optimization?
      - Instruction size, code size

# SMALL SEMANTIC GAP EXAMPLES IN VAX

- FIND FIRST
  - Find the first set bit in a bit field
  - Helps OS resource allocation operations
- SAVE CONTEXT, LOAD CONTEXT
  - Special context switching instructions
- INSQUEUE, REMQUEUE
  - Operations on doubly linked list
- INDEX
  - Array access with bounds checking
- STRING Operations
  - Compare strings, find substrings, …
- Cyclic Redundancy Check Instruction
- EDITPC
  - Implements editing functions to display fixed format output

- Digital Equipment Corp., "VAX11 780 Architecture Handbook," 1977-78.

# CISC vs. RISC

REPMOVS

*x86: REP MOVS DEST SRC*

```
X:
MOV
ADD
COMP
MOV
ADD
JMP X
```

**Which one is easy to optimize?**

# SMALL VERSUS LARGE SEMANTIC GAP

- CISC vs. RISC
  - Complex instruction set computer → complex instructions
    - Initially motivated by "not good enough" code generation
  - Reduced instruction set computer → simple instructions
    - John Cocke, mid 1970s, IBM 801
      - Goal: enable better compiler control and optimization

- RISC motivated by
  - Memory stalls (no work done in a complex instruction when there is a memory stall?)
    - When is this correct?
  - Simplifying the hardware → lower cost, higher frequency
  - Enabling the compiler to optimize the code better
    - Find fine-grained parallelism to reduce stalls

# Typical RISC ISA properties

- fewer, simpler instructions
- separate instructions to access memory
- fixed-length instructions
- more registers
- no instructions with two memory operands
- few addressing modes

# Agenda

- ISA vs Microarchitecture
- ISA Tradeoffs
- Y86-64 ISA
- Y86-64 Format
- Y86-64 encoding/decoding

# Y86-64 instruction set

- based on x86
- omits most of the 1000+ instructions

**addq jmp pushq**

**subq jCC popq**

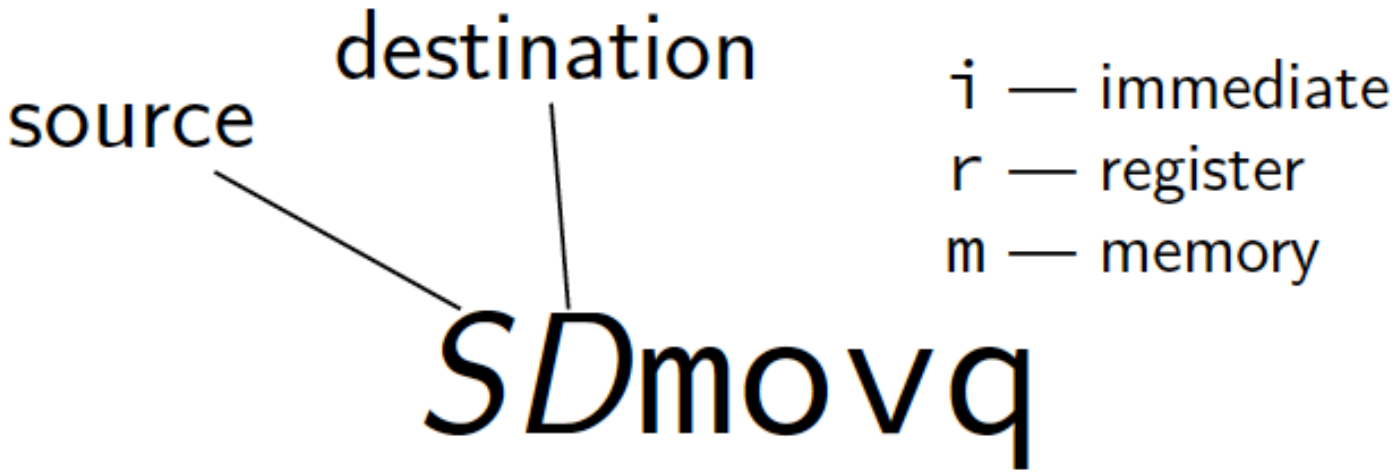
**andq cmovCC movq (renamed)**

**xorq call hlt (renamed)**

**nop ret**

- much, much simpler encoding

# Y86-64: movq




- ***irmovq*** *immovq iimovq*
- ***rrmovq rmmovq*** *mrmovq*
- ***mrmovq*** *mmmovq mimovq*

# Y86-64: cmovCC

- conditional move
- ***(Conditionally) copy value from source to destination register***
- Y86-64: register-to-register only
- instead of:

```
jle skip_move
rrmovq %rax, %rbx
skip_move:
```

- // …
- can do:

```
cmovg %rax, %rbx
```

# Y86-64: halt

- (x86-64 instruction called hlt)
- Y86-64 instruction halt
- stops the processor
- otherwise — something's in memory "after" program!
- real processors: reserved for OS

# Y86-64: specifying addresses

- ***rmmovq %r11, 10(%r12)***
- memory[10 + r12] ← r11

- r12 ← memory[10 + r11] + r12

```
mrmovq 10(%r11), %r11
/* overwrites %r11 */
addq %r11, %r12
```

# Y86-64: accessing memory

- r12 ← memory[10 + 8 * r11] + r12

```
/* replace %r11 with 8*%r11 */
addq %r11, %r11
addq %r11, %r11
addq %r11, %r11
mrmovq 10(%r11), %r11
addq %r11, %r12
```

# Y86-64 constants

- ***irmovq $100, %r11***
  - only instruction with non-address constant operand

- ***r12 ← r12 + 1***
- Invalid: ***addq $1, %r12***
- Instead, need an extra register:

```
irmovq $1, %r11
addq %r11, %r12
```

# Y86-64: condition codes

- ZF — value was zero?
- SF — sign bit was set? i.e. value was negative?
- this course: no OF, CF (to simplify assignments)
- set by ***`addq`, `subq`, `andq`, `xorq`***
- not set by anything else

# Y86-64: using condition codes

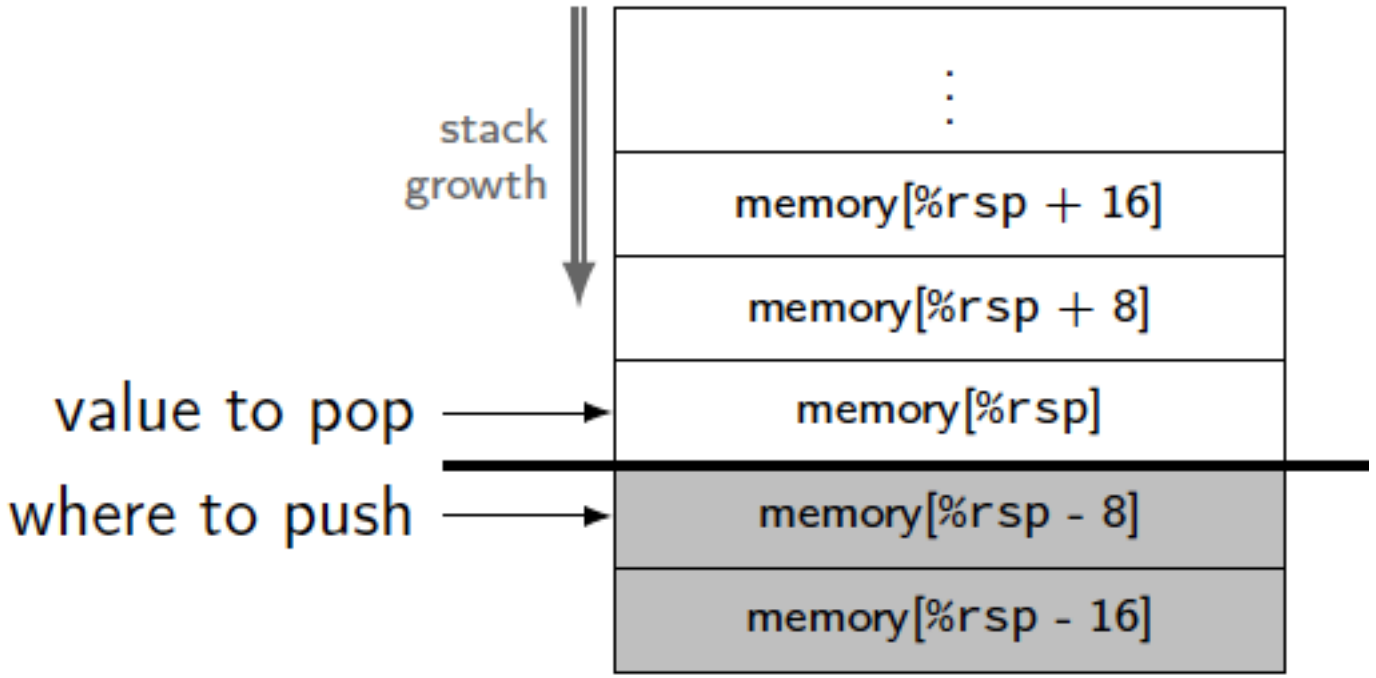
subq SECOND, FIRST (value = FIRST - SECOND)

| j__ or cmov__ | condition code bit test | value test |
|---|---|---|
| le | SF = 1 or ZF = 1 | value <= 0 |
| l | SF = 1 | value < 0 |
| e | ZF = 1 | value = 0 |
| ne | ZF = 0 | value != 0 |
| ge | SF = 0 | value >= 0 |
| g | SF = 0 and ZF = 0 | value > 0 |

# push/pop

```
pushq %rbx
%rsp  ← %rsp - 8
memory[%rsp] ←  %rbx
```

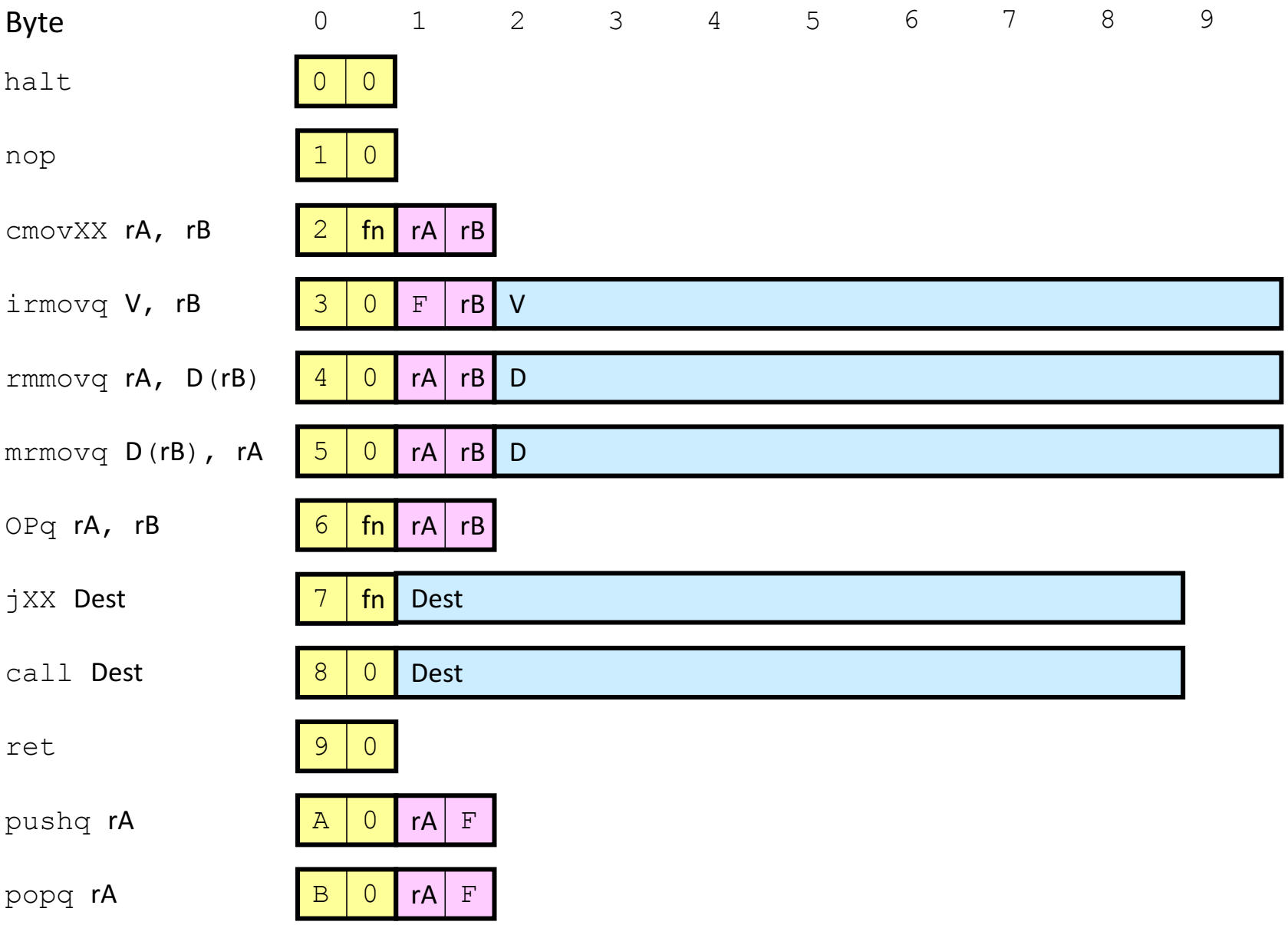
```
popq %rbx
%rbx ←  memory[%rsp]
%rsp  ← %rsp + 8
```



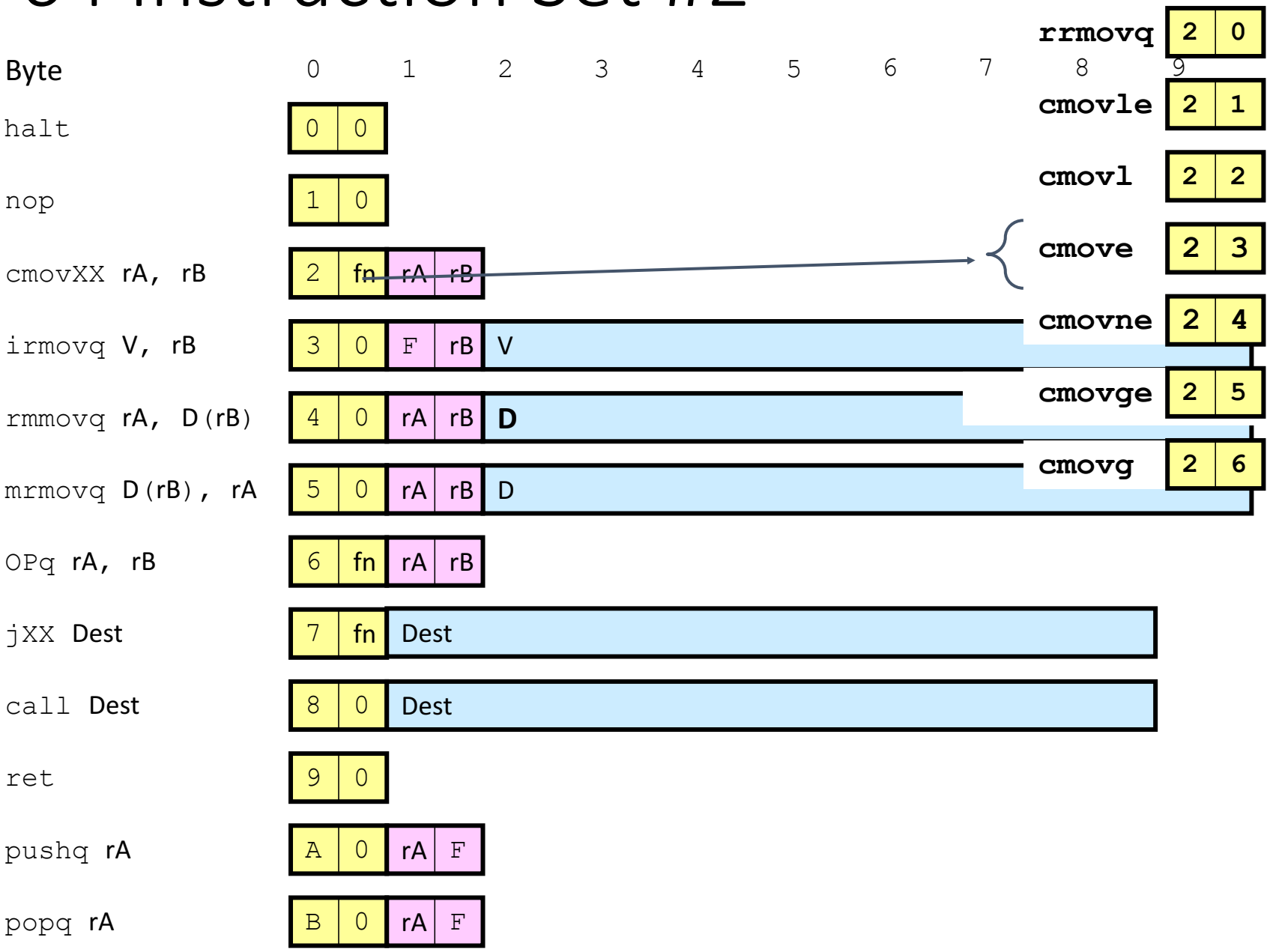
# Agenda

- ISA vs Microarchitecture
- ISA Tradeoffs
- Y86-64 ISA
- Y86-64 Format
- Y86-64 encoding/decoding

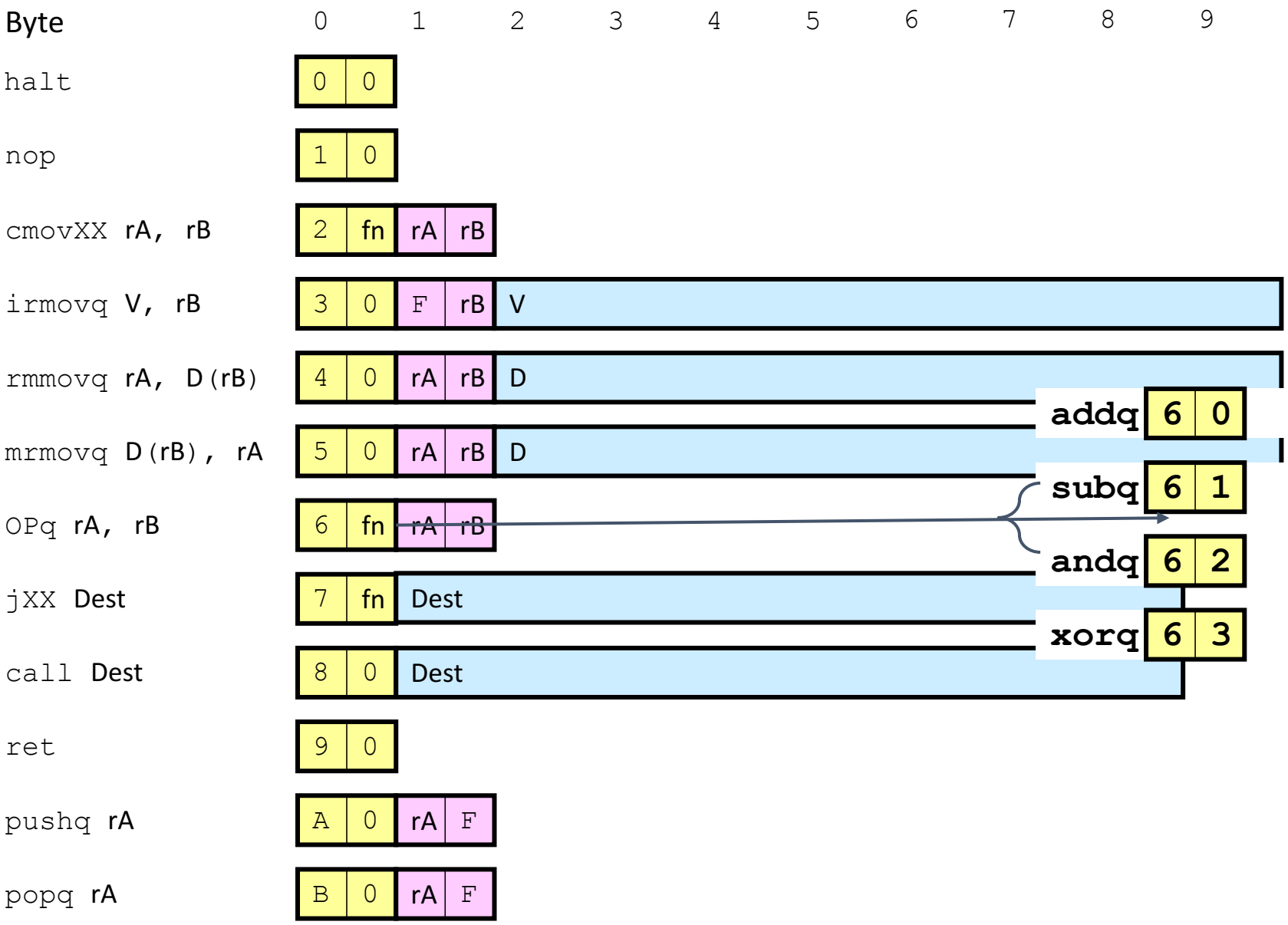
# Y86-64 Instruction Set #1



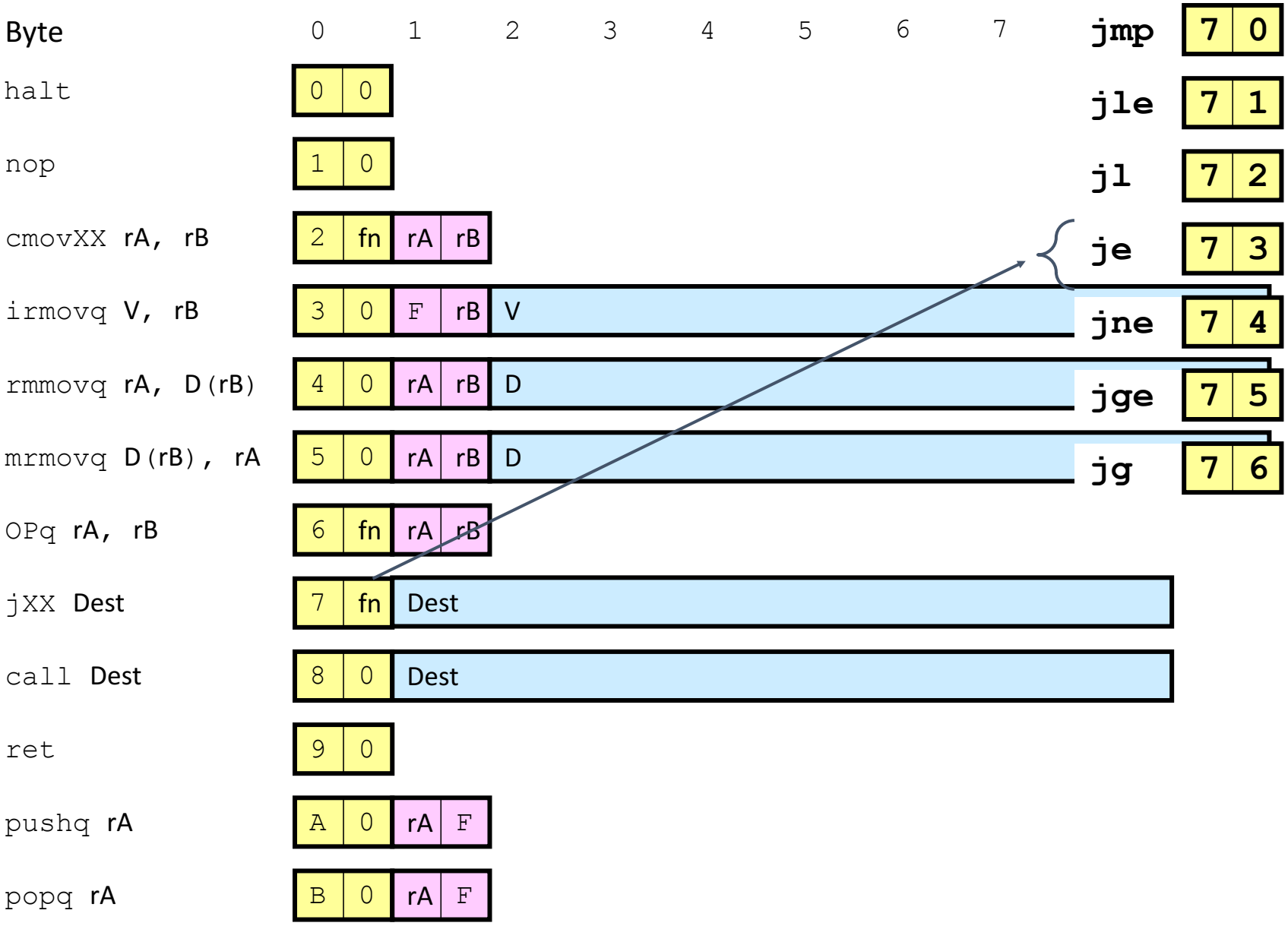
# Y86-64 Instruction Set #2



# Y86-64 Instruction Set #3



# Y86-64 Instruction Set #4



# Encoding Registers

- Each register has 4-bit ID

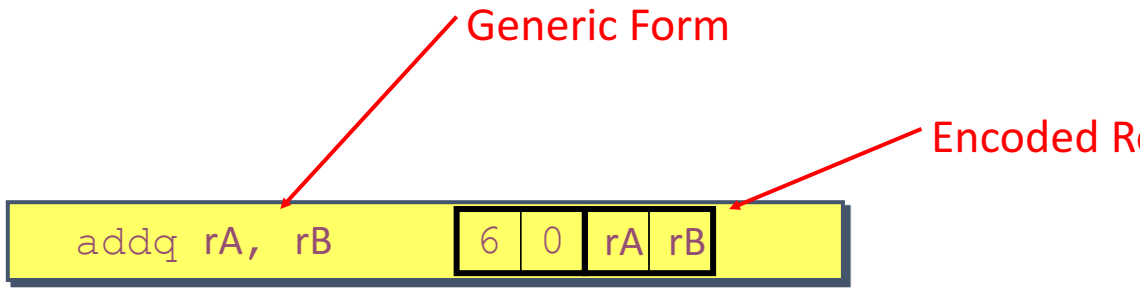
| | |
|---|---|
| %rax | 0 |
| %rcx | 1 |
| %rdx | 2 |
| %rbx | 3 |
| %rsp | 4 |
| %rbp | 5 |
| %rsi | 6 |
| %rdi | 7 |

| | |
|---|---|
| %r8 | 8 |
| %r9 | 9 |
| %r10 | A |
| %r11 | B |
| %r12 | C |
| %r13 | D |
| %r14 | E |
| 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

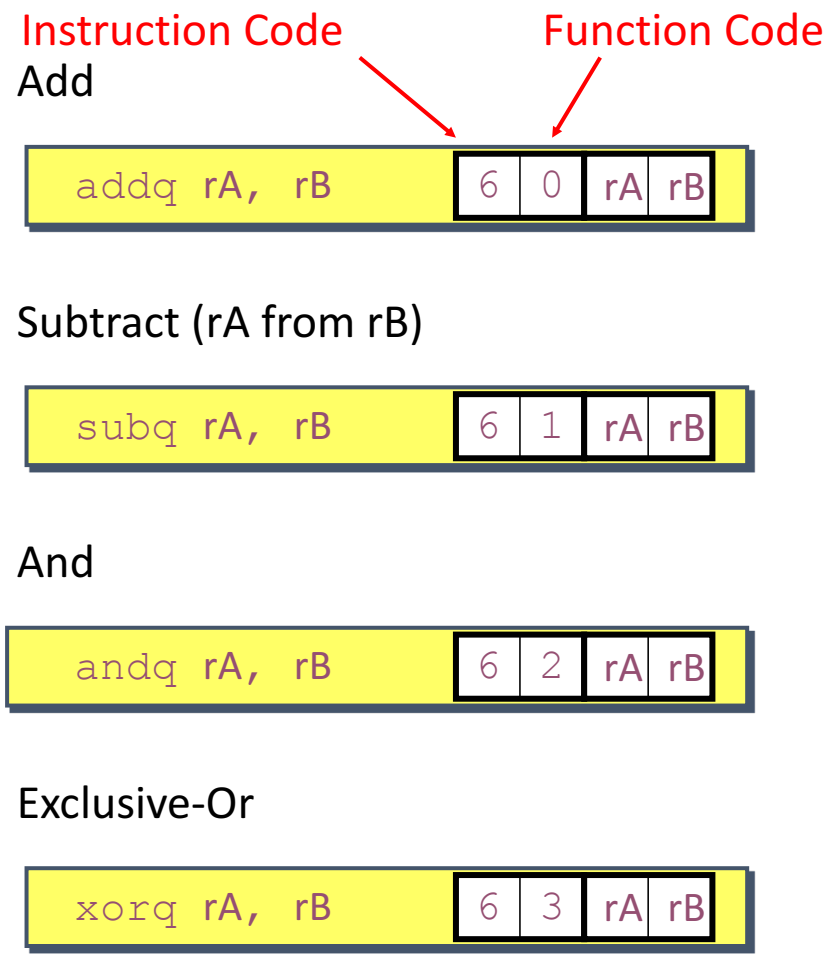
# Instruction Example

- 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 "`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


Immediate ➔ Register


Register ➔ Memory


Memory ➔ Register


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

# 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

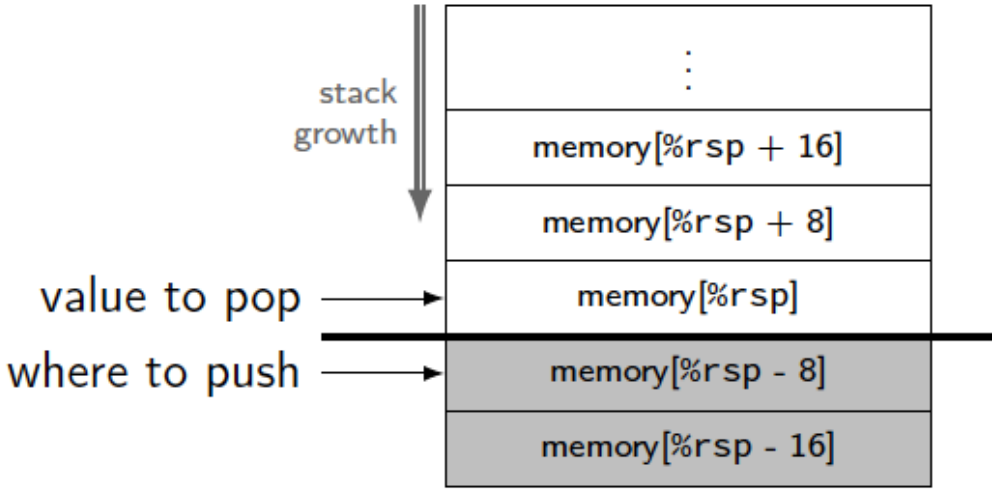

Jump When Greater or Equal


Jump When Greater

# 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


- 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

# Agenda

- ISA vs Microarchitecture
- ISA Tradeoffs
- Y86-64 ISA
- Y86-64 Format
- Y86-64 Encoding/Decoding

# Y86-64 encoding

```
long addOne(long x) {
return x + 1;
}
```

- x86-64:

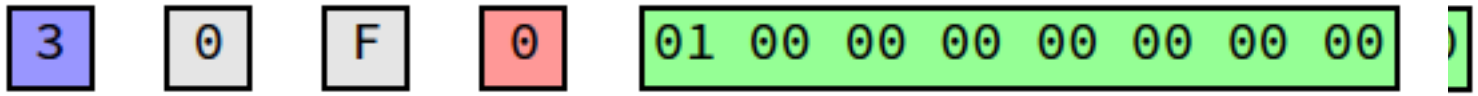
```
movq %rdi, %rax
addq $1, %rax
ret
```

- Y86-64:

```
irmovq $1, %rax
addq %rdi, %rax
ret
```

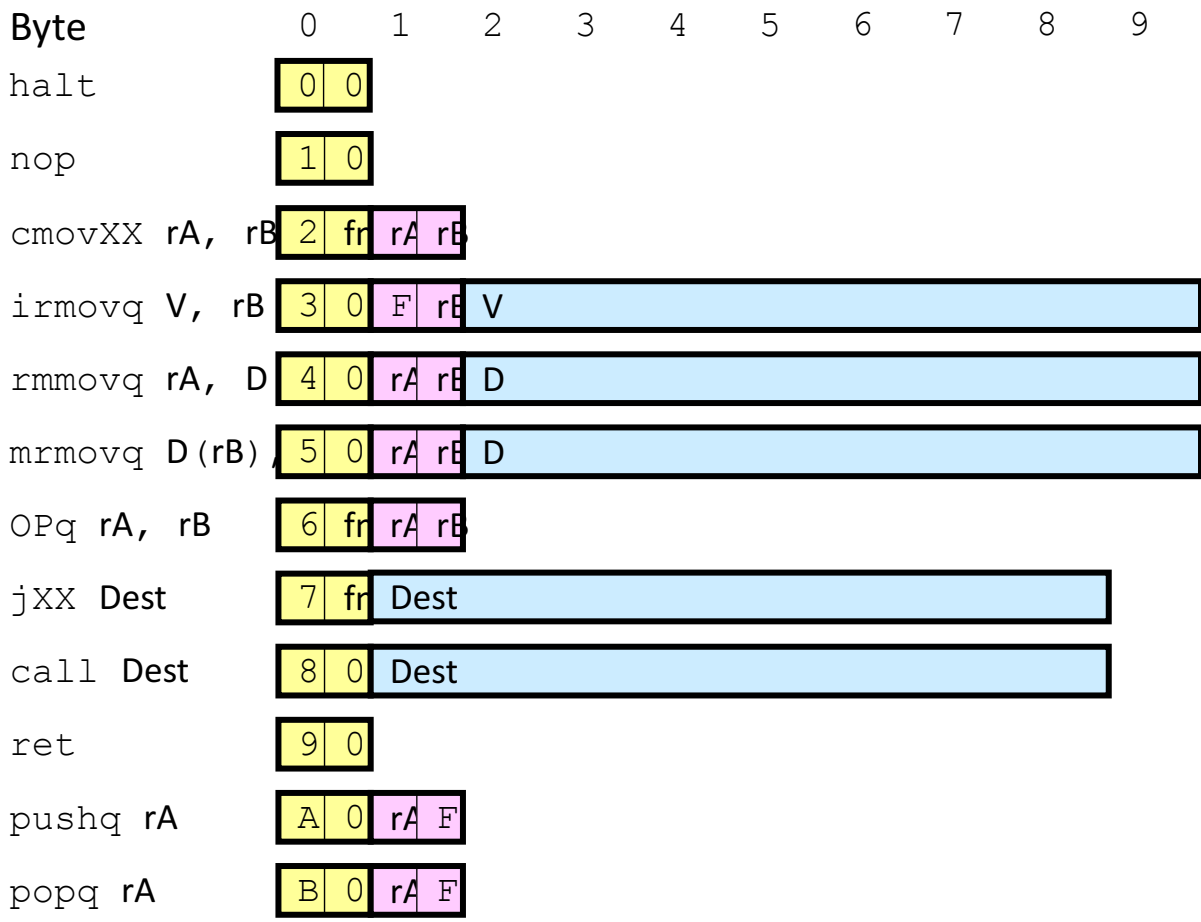
# Y86-64 encoding

```
addOne:
irmovq $1, %rax
addq %rdi, %rax
ret
```



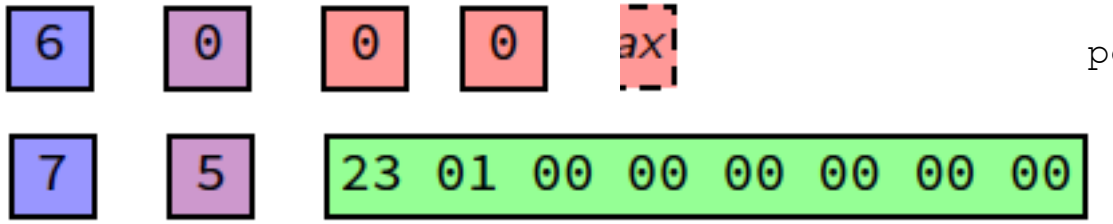


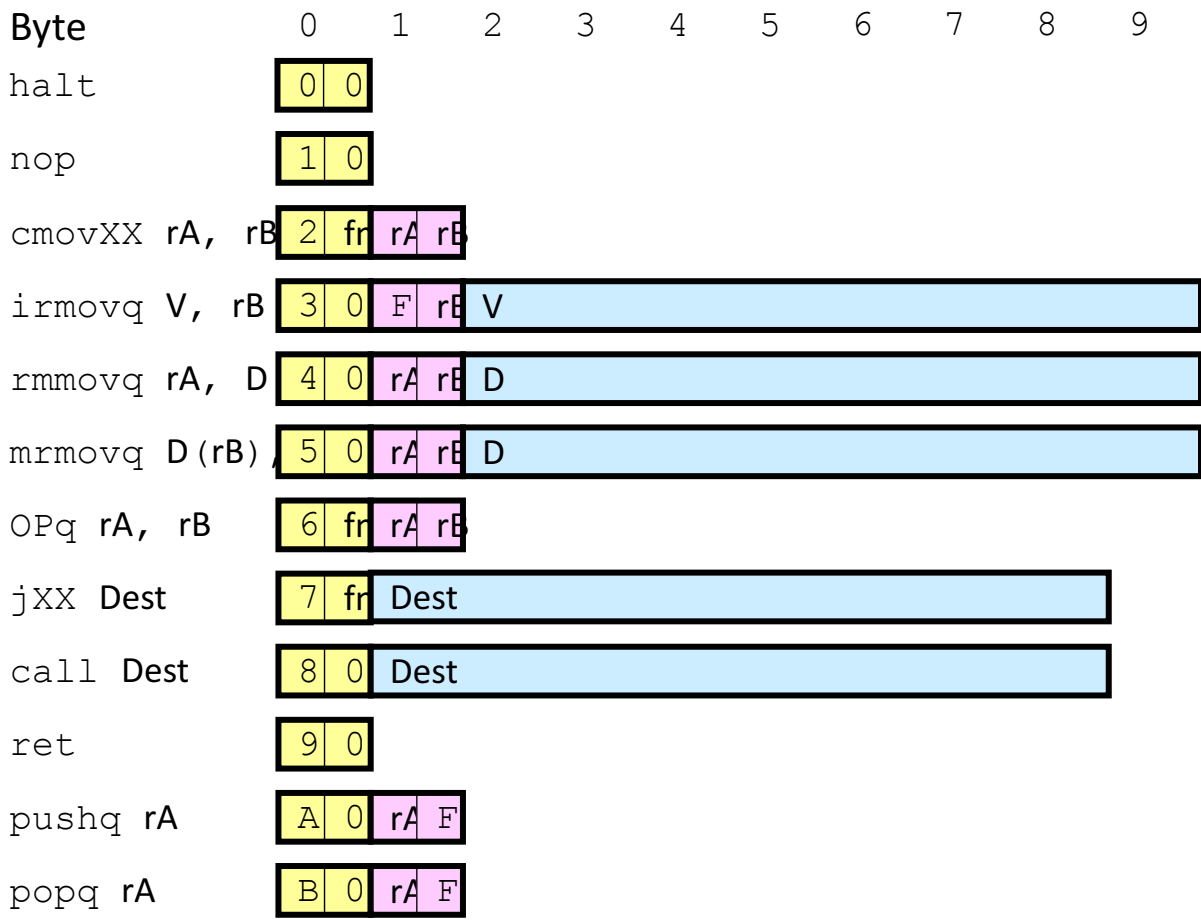

30 F0 01 00 00 00 00 00 00 00 60 70 90



# Y86-64 encoding

```
doubleTillNegative:
/* suppose at address 0x123 */
addq %rax, %rax
jge doubleTillNegative
```



# Y86-64 decoding

**20 10 60 20 61 37 72 84 00 00 00 00 00 00 00 20 12 20 01 70 68 00 00 00 00 00 00 00 00**

rrmovq %rcx, %rax

- 0 as cc: always
- 1 as reg: %rcx
- 0 as reg: %rax

addq %rdx, %rax

subq %rbx, %rdi

- 0 as fn: add
- 1 as fn: sub

jl 0x84

- 2 as cc: l (less than)
- hex 84 00… as little endian Dest: 0x84

rrmovq %rcx, %rdx

rrmovq %rax, %rcx

jmp 0x68