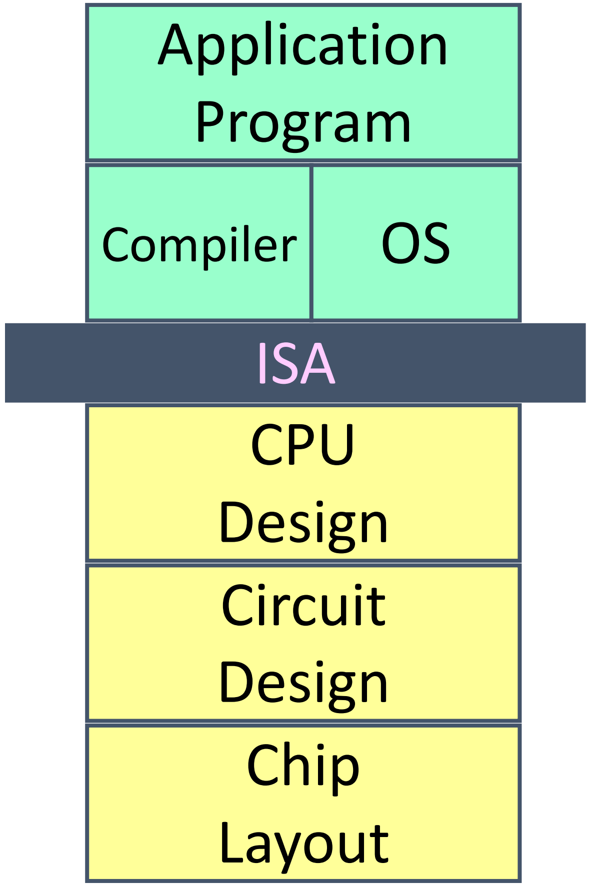
Y86-64: Instruction Set Architecture

CMPU 224 – Computer Organization
Jason Waterman

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



Application Program

Compiler

OS

ISA

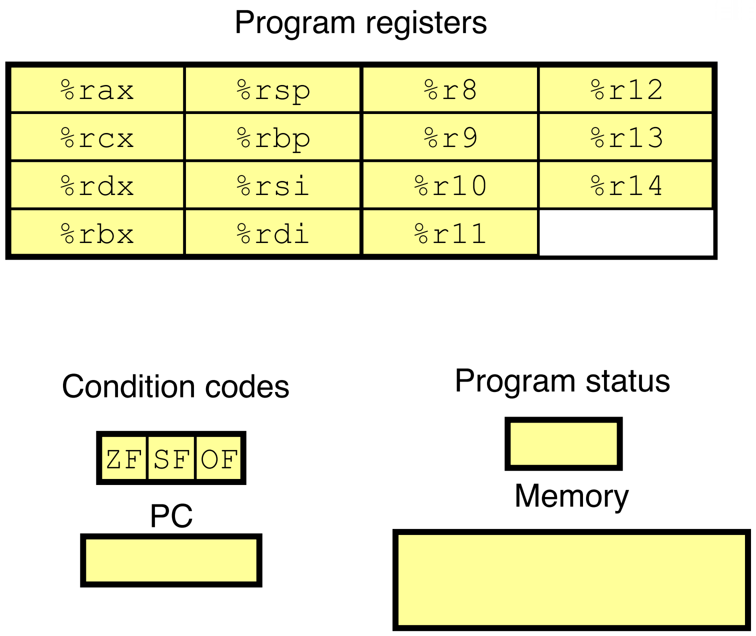
CPU Design

Circuit Design

Chip Layout

Y86-64 Processor State

- Program Registers
  - 15 registers (omit `%r15`)
  - Each 64-bits long
- Condition Codes
  - Single-bit flags set by arithmetic and 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



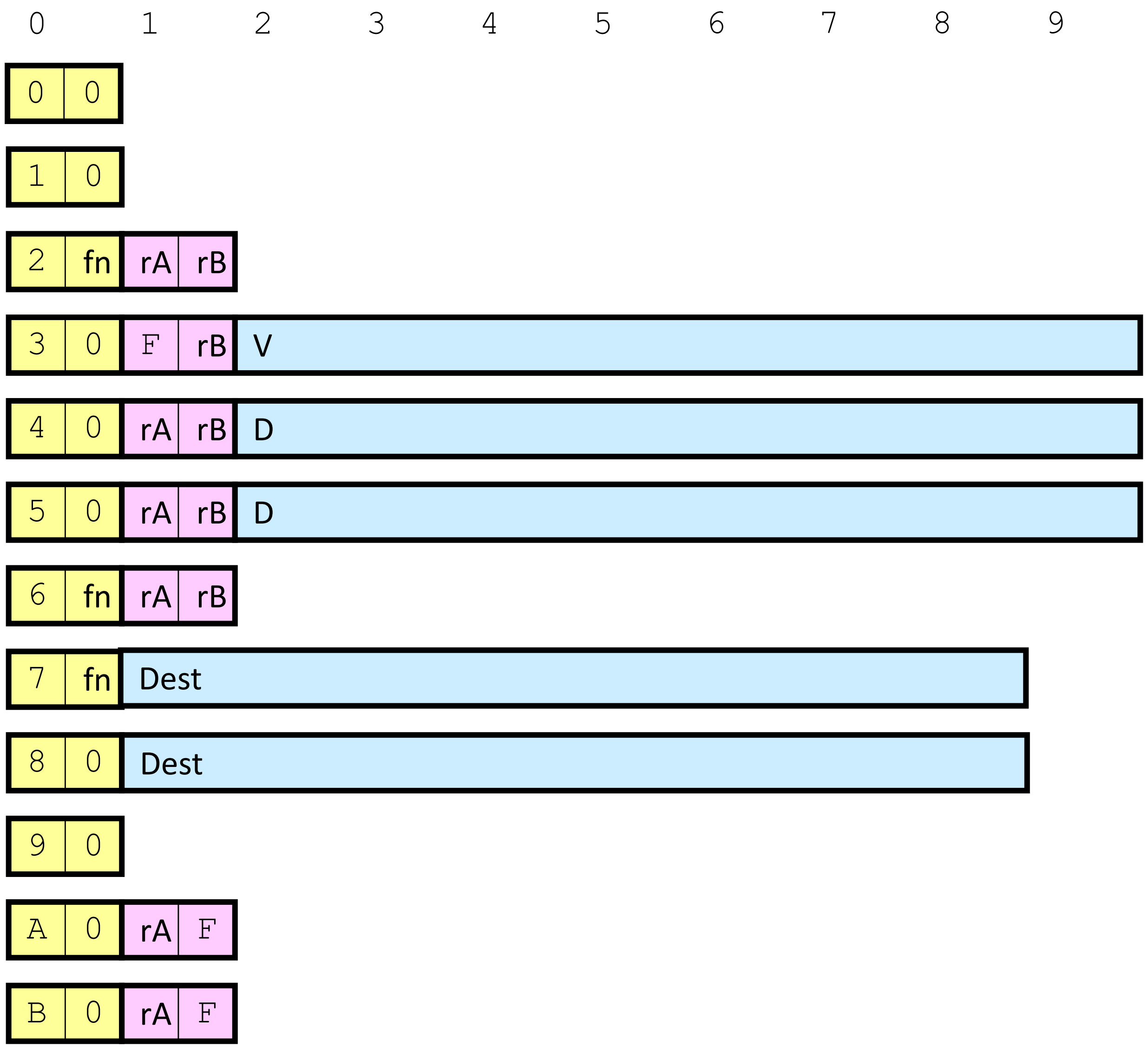
Program registers

`%rax`	`%rsp`	`%r8`	`%r12`
`%rcx`	`%rbp`	`%r9`	`%r13`
`%rdx`	`%rsi`	`%r10`	`%r14`
`%rbx`	`%rdi`	`%r11`	

Y86-64 Instructions

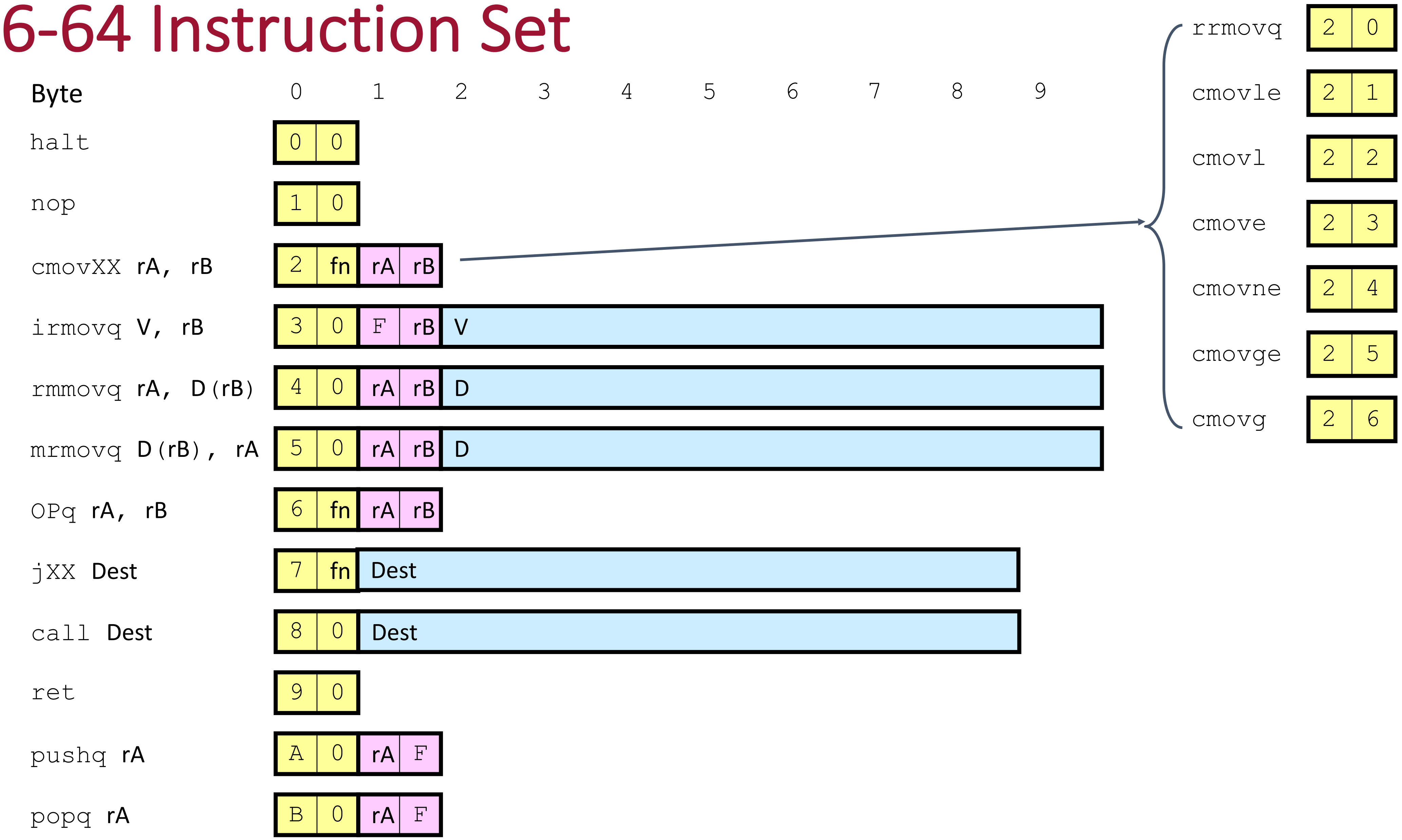
- Largely a subset of x86-64 instructions
- Only 8-byte integer operations
- Format
  - 1–10 bytes of information read from memory
  - Can determine instruction length from first byte
  - Not as many instruction types, and simpler encoding than with x86-64

Y86-64 Instruction Set



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								

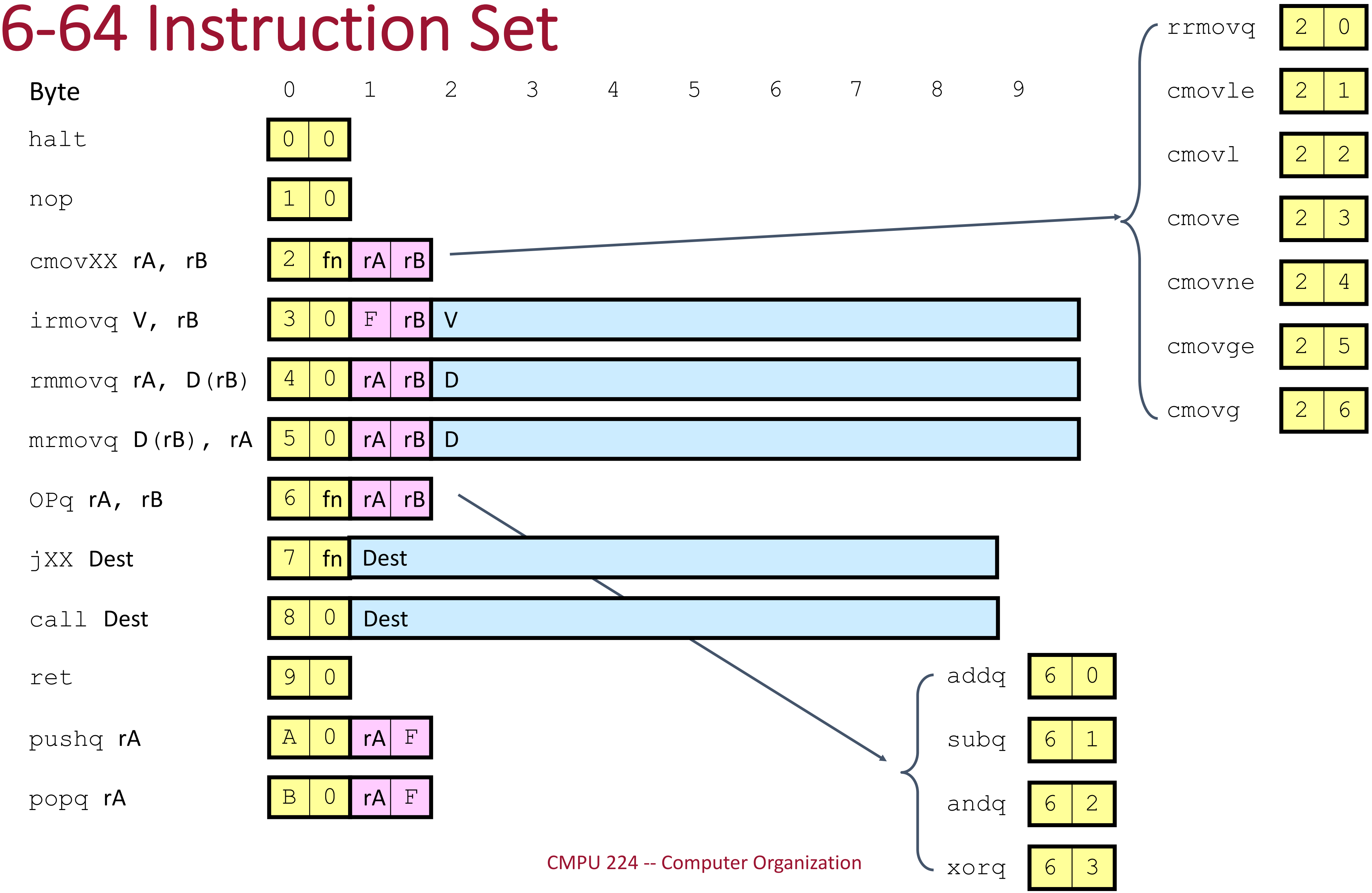
Y86-64 Instruction Set



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								

`rrmovq`	2 0
`cmovle`	2 1
`cmovl`	2 2
`cmove`	2 3
`cmovne`	2 4
`cmovge`	2 5
`cmovg`	2 6

Y86-64 Instruction Set

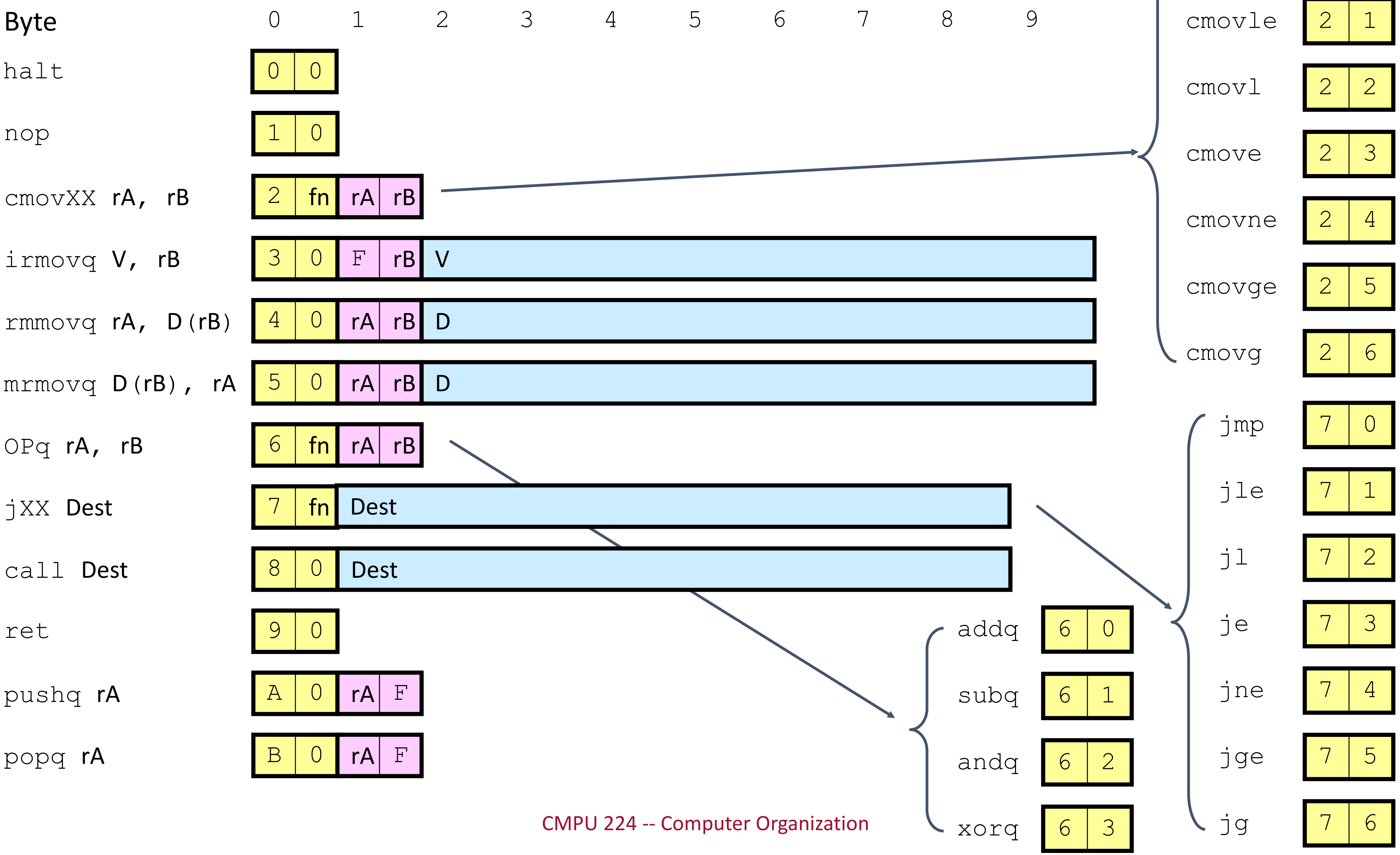


Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0 0									
`nop`	1 0									
`cmovXX` rA, rB	2 fn	rA rB								
`irmovq` V, rB	3 0	F rB	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`OPq` rA, rB	6 fn	rA rB								
`jXX` Dest	7 fn	Dest								
`call` Dest	8 0	Dest								
`ret`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

`rrmovq`	2	0
`cmovle`	2	1
`cmovl`	2	2
`cmove`	2	3
`cmovne`	2	4
`cmovge`	2	5
`cmovg`	2	6

`addq`	6	0
`subq`	6	1
`andq`	6	2
`xorq`	6	3

Y86-64 Instruction Set



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								

`rrmovq`	2 0
`cmovle`	2 1
`cmovl`	2 2
`cmove`	2 3
`cmovne`	2 4
`cmovge`	2 5
`cmovg`	2 6

`addq`	6 0
`subq`	6 1
`andq`	6 2
`xorq`	6 3

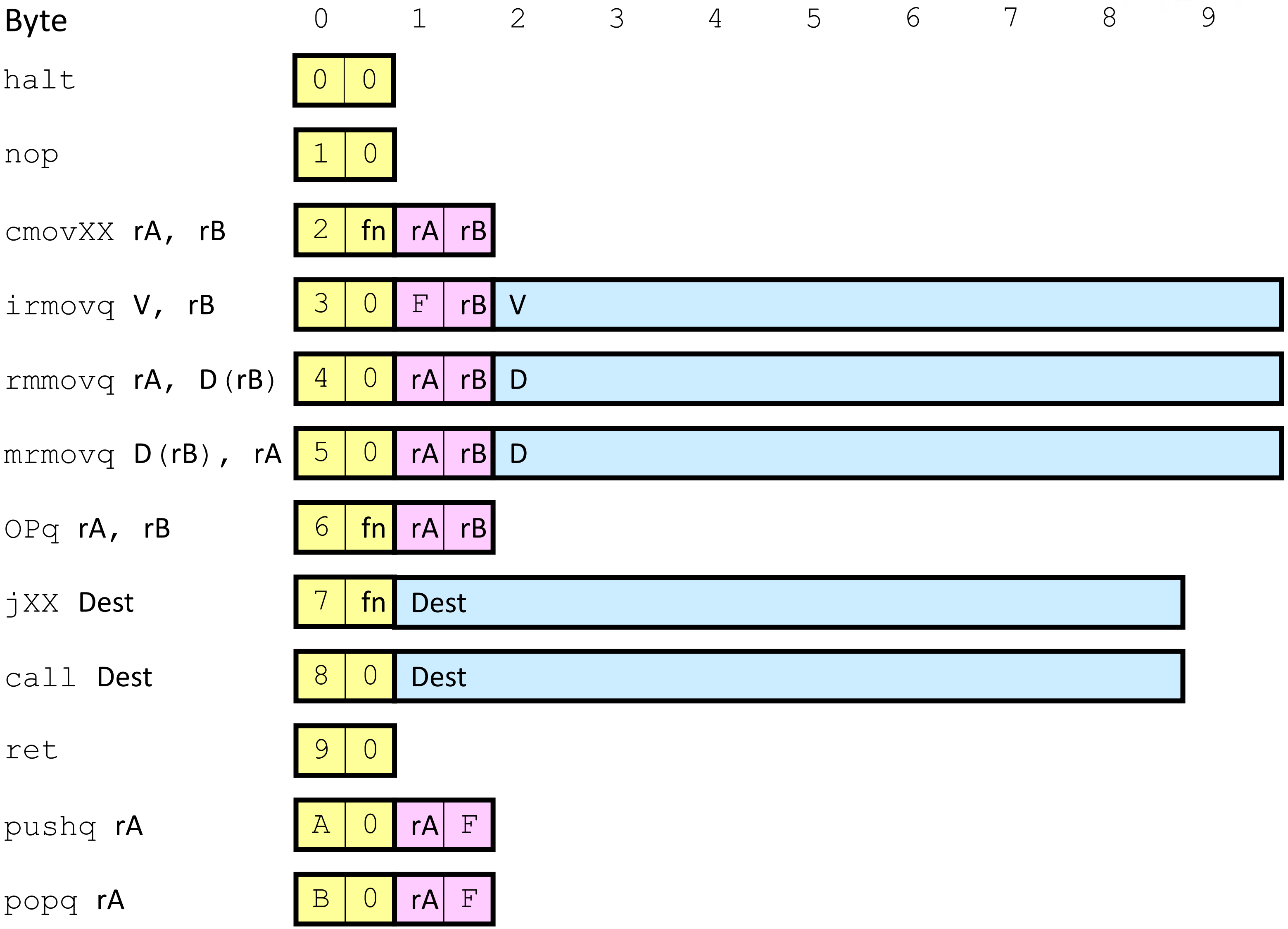
`jmp`	7 0
`jle`	7 1
`jl`	7 2
`je`	7 3
`jne`	7 4
`jge`	7 5
`jg`	7 6

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

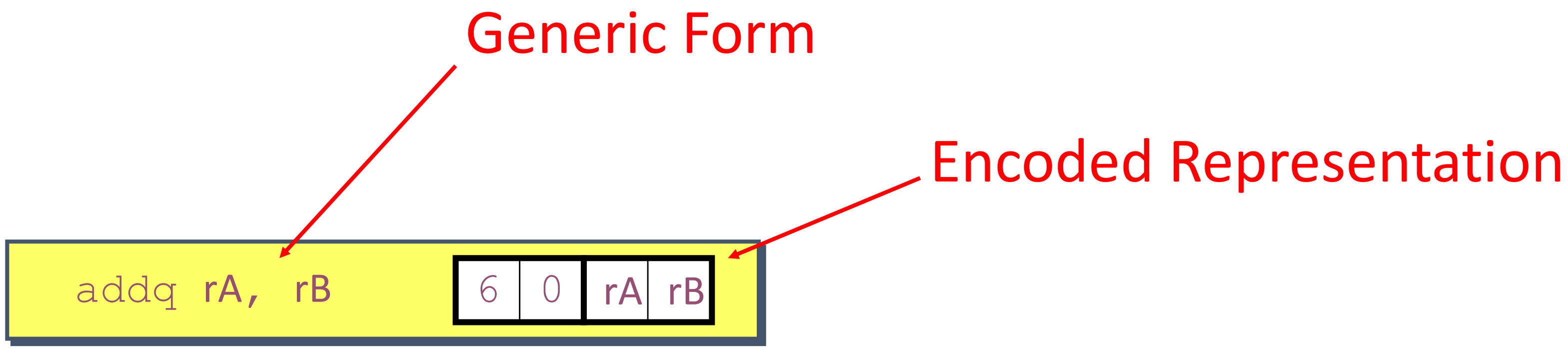
  - Same encoding as in x86-64
- Register ID 15 (0xF) indicates “no register”
  - Will use this in our hardware design in multiple places



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								

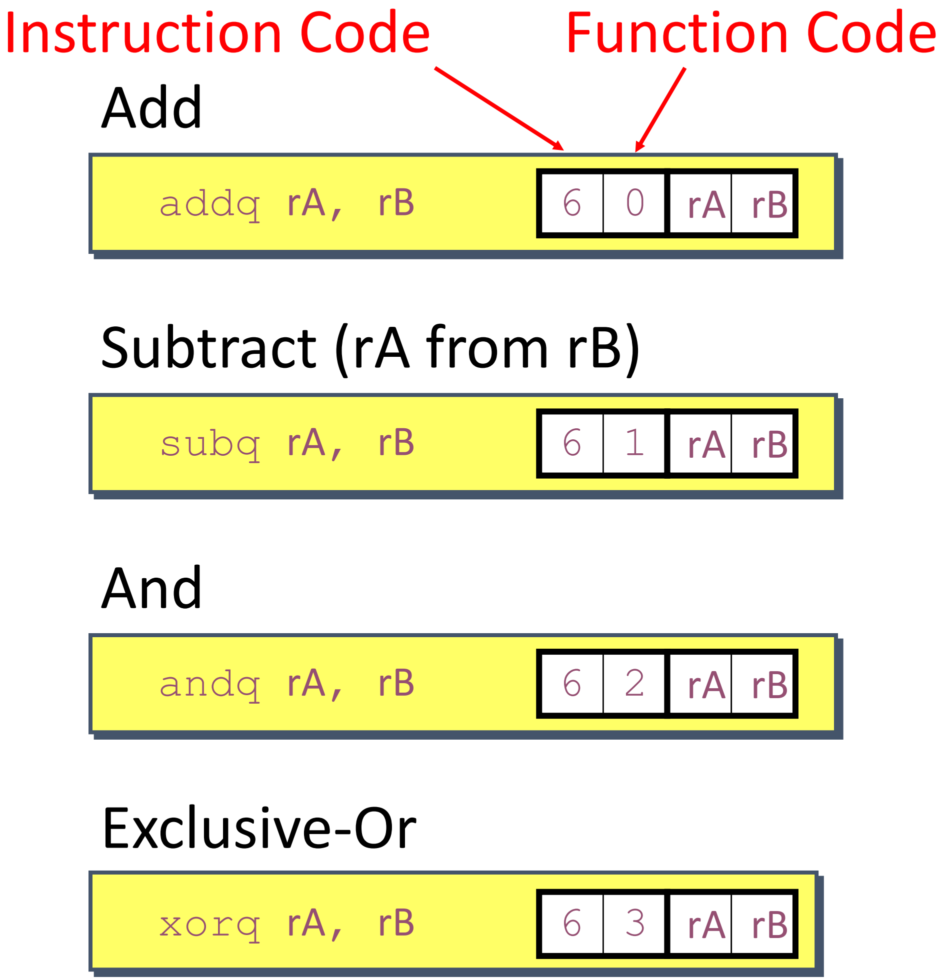
Instruction Example

- 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



Instruction Code

Function Code

Add

`addq rA, rB` | 6 | 0 | rA | rB |

Subtract (rA from rB)

`subq rA, rB` | 6 | 1 | rA | rB |

And

`andq rA, rB` | 6 | 2 | rA | rB |

Exclusive-Or

`xorq rA, rB` | 6 | 3 | rA | rB |

- 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

Register → Register						
`rrmovq` rA, rB	2	0	rA	rB		
Immediate → Register						
`irmovq` V, rB	3	0	F	rB	V	
Register → Memory						
`rmmovq` rA, D (rB)	4	0	rA	rB	D	
Memory → Register						
`mrmovq` D (rB), rA	5	0	rA	rB	D	

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

Little-endian

Encoding: `30 F2 cd ab 00 00 00 00 00 00`

X86-64	Y86-64
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`

Encoding: `20 43`

X86-64	Y86-64
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`

Two’s complement

Encoding: `50 15 f4 ff ff ff ff ff ff ff`

X86-64	Y86-64
`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)`

Encoding: `40 64 1c 04 00 00 00 00 00 00`

Conditional Move Instructions

Move Unconditionally	`rrmovq` rA, rB	2	0	rA	rB
Move When Less or Equal	`cmovle` rA, rB	2	1	rA	rB
Move When Less	`cmovl` rA, rB	2	2	rA	rB
Move When Equal	`cmove` rA, rB	2	3	rA	rB
Move When Not Equal	`cmovne` rA, rB	2	4	rA	rB
Move When Greater or Equal	`cmovge` rA, rB	2	5	rA	rB
Move When Greater	`cmovg` rA, rB	2	6	rA	rB

- 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

- 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 (Conditionally)

`jXX` Dest	7	fn	Dest

Jump Instructions

Jump Unconditionally

`jmp` **Dest**	7	0	Dest

Jump When Less or Equal

`jle` **Dest**	7	1	Dest

Jump When Less

`jl` **Dest**	7	2	Dest

Jump When Equal

`je` **Dest**	7	3	Dest

Jump When Not Equal

`jne` **Dest**	7	4	Dest

Jump When Greater or Equal

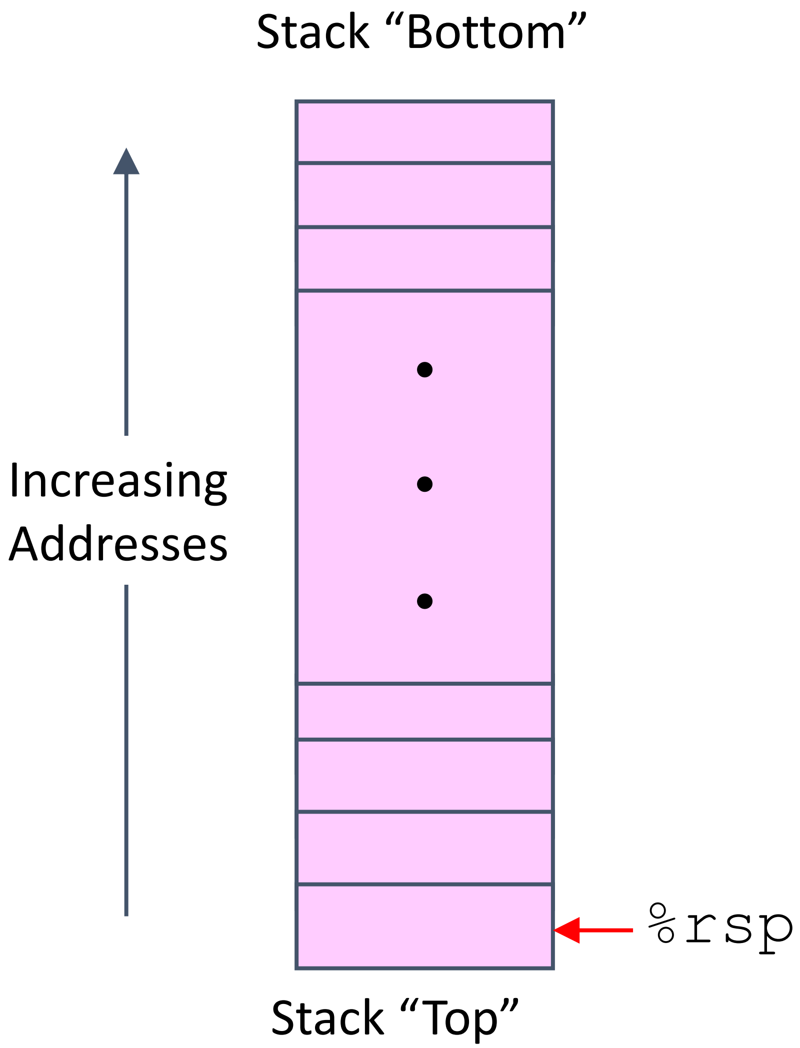
`jge` **Dest**	7	5	Dest

Jump When Greater

`jg` **Dest**	7	6	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

`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 it can’t be executed in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

Status Conditions

- Normal operation

Mnemonic	Code
AOK	1

- Halt instruction encountered

Mnemonic	Code
HLT	2

- Bad address (either instruction or data) encountered

Mnemonic	Code
ADR	3

- Invalid instruction encountered

Mnemonic	Code
INS	4

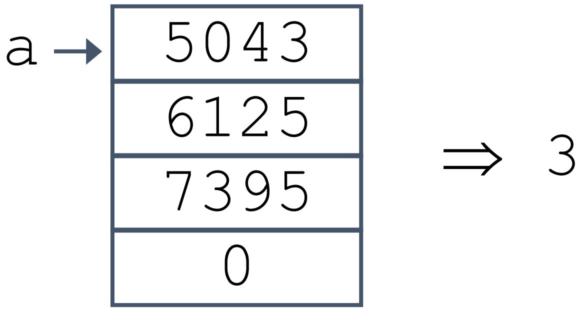
- Desired Behavior
  - If AOK, keep going
  - Otherwise, stop program execution

Writing Y86-64 Code

- Can try to Use C Compiler
  - Write code in C
  - Compile for x86-64 with `gcc -Og -S`
  - Transliterate into Y86-64
  - *Modern compilers make this more difficult*

- Coding Example
  - Find the number of elements in null-terminated list

```
int len1(int a[]);
```



Y86-64 Code Generation Example

- First Try
  - Write typical array code

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

- Compile with `gcc -Og -S`

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

```
len:
  movl $0, %eax
.L3:
  cmpq $0, (%rdi,%rax,8)
  je .L2
  addq $1, %rax
  jmp .L3
.L2:
  ret
```

Y86-64 Code Generation Example #2

- Second Try
  - Write C code that mimics expected Y86-64 code

Register	Use
%rdi	**a**
%rax	**len**
%rdx	**val**

```
long len(long a[])
{
  long val = *a;
  long len = 0;
  while (val) {
    a++;
    len++;
    val = *a;
  }
  return len;
}
```

```
len:
  movq (%rdi), %rdx   # val = *a
  movl $0, %eax       # len = 0
.L3:
  testq %rdx, %rdx    # while(val)
  je .L2              # while(val)
  addq $8, %rdi       # a++
  addq $1, %rax       # len++
  movq (%rdi), %rdx   # val = *a
  jmp .L3             # jump to while test
.L2:
  ret                 # return len
```

Y86-64 Code Generation Example #3

```
len:
  movq (%rdi), %rdx
  movl $0, %eax
.L3:
  testq %rdx, %rdx
  je .L2
  addq $8, %rdi
  addq $1, %rax
  movq (%rdi), %rdx
  jmp .L3
.L2:
  rep ret
```

```
len:
    irmovq $1, %r8          # Constant 1
    irmovq $8, %r9          # Constant 8
    mrmovq (%rdi), %rdx     # val = *a
    irmovq $0, %rax         # len = 0
test:
    andq %rdx, %rdx         # Test val
    je done                 # If zero, goto Done
    addq %r9, %rdi          # a++
    addq %r8, %rax          # len++
    mrmovq (%rdi), %rdx     # val = *a
    jmp test                # Jump to test
done:
    ret
```

Register	Use
`%rdi`	`a`
`%rax`	`len`
`%rdx`	`val`
`%r8`	`1`
`%r9`	`8`

Y86-64 Sample Program Structure #1

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

```
# Initialization
    .pos 0     # Execution begins at address 0
    irmovq stack, %rsp # Set up stack pointer

    call main          # Execute main program
    halt

    .align 8           # Program data
array:
    . . .

main:                  # Main function
    . . .
    call len
    . . .

len:                   # Length function
   . . .

   .pos 0x200          # Placement of stack
stack:
```

Y86-64 Program Structure #2

- Must initialize data
  - Can use symbolic names
- Set up call to len
  - Follow x86-64 procedure conventions
  - Push array address as argument

```
# Initialization
    .pos 0     # Execution begins at address 0
    irmovq stack, %rsp # Set up stack pointer

    call main          # Execute main program
    halt

# Array of 4 elements + terminating 0
    .align 8
array:
    .quad 0x000d000d000d000d
    .quad 0x00c000c000c000c0
    .quad 0x0b000b000b000b00
    .quad 0xa000a000a000a000
    .quad 0
main:
    irmovq array, %rdi
    call len
    ret
...
    .pos 0x200          # Placement of stack
stack:
```

Assembling Y86-64 Programs (`yas`)

- Generates “object code” file len.yo
  - Looks like disassembler output

```
    Linux> yas len.ys
```

```
0x054:                      | len:
0x054: 30f80100000000000000 |     irmovq $1, %r8       # Constant 1
0x05e: 30f90800000000000000 |     irmovq $8, %r9       # Constant 8
0x068: 50270000000000000000 |     mrmovq (%rdi), %rdx  # val = *a
0x072: 30f00000000000000000 |     irmovq $0, %rax      # len = 0
0x07c:                      | test:
0x07c: 6222                 |     andq %rdx, %rdx      # Test val
0x07e: 739e00000000000000   |     je done              # If zero, goto Done
0x087: 6097                 |     addq %r9, %rdi       # a++
0x089: 6080                 |     addq %r8, %rax       # len++
0x08b: 50270000000000000000 |     mrmovq (%rdi), %rdx  # val = *a
0x095: 707c00000000000000   |     jmp test             # Jump to test
0x09e:                      | done:
0x09e: 90                   |     ret
```

Simulating Y86-64 Programs (`yis`)

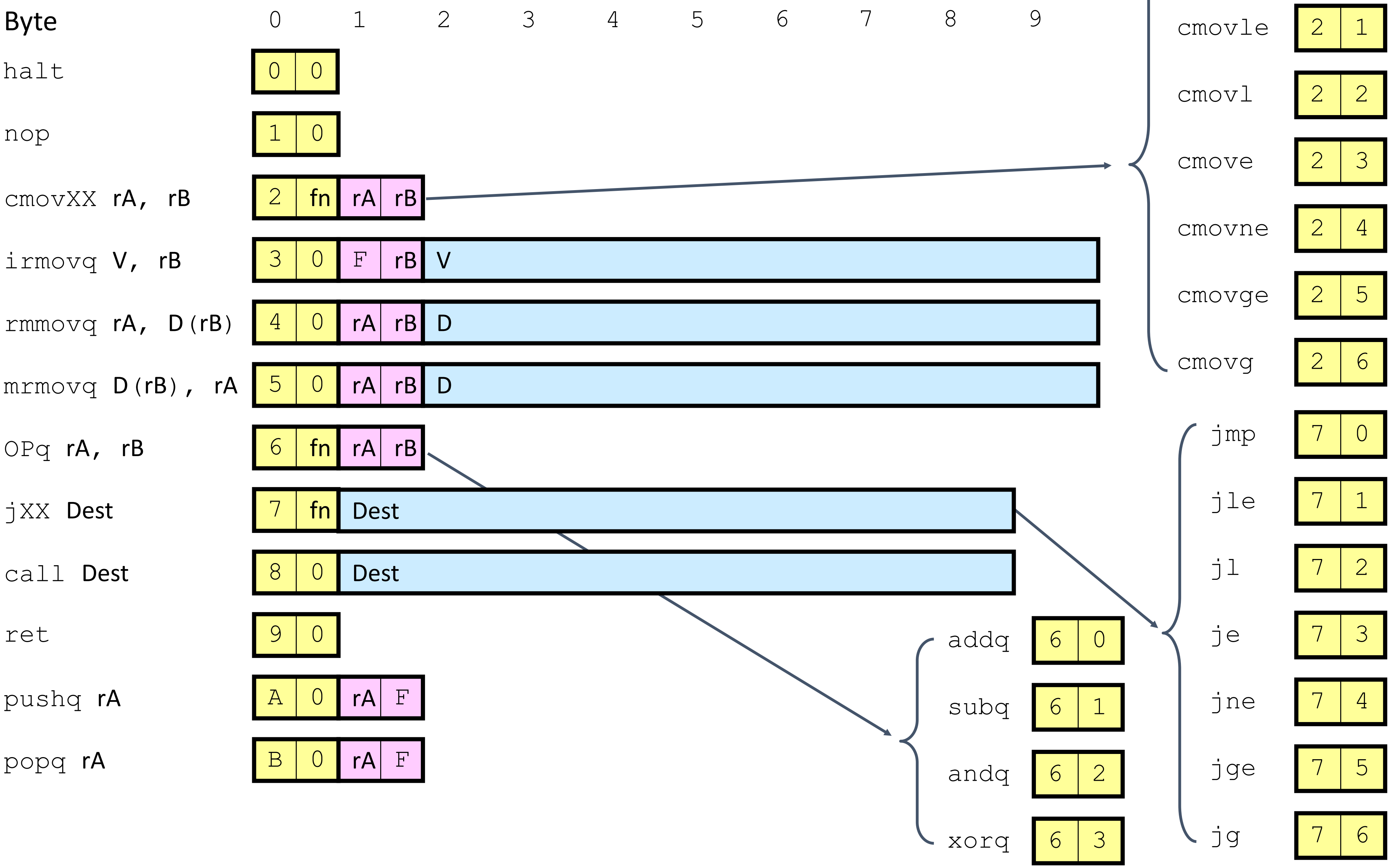
- Instruction set simulator
  - Computes effect of each instruction on processor state
  - Prints changes in state from original

```
Linux> yis len.yo
```

```
Stopped in 37 steps at PC = 0x13.  Status 'HLT', CC Z=1
S=0 O=0
Changes to registers:
%rax:   0x0000000000000000 0x0000000000000004
%rsp:   0x0000000000000000 0x0000000000000200
%rdi:   0x0000000000000000 0x0000000000000038
%r8:    0x0000000000000000 0x0000000000000001
%r9:    0x0000000000000000 0x0000000000000008


Changes to memory:
0x01f0: 0x0000000000000000 0x0000000000000053
0x01f8: 0x0000000000000000 0x0000000000000013
```

Y86-64 Instruction Set



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								

`addq`	6 0
`subq`	6 1
`andq`	6 2
`xorq`	6 3

`rrmovq`	2 0
`cmovle`	2 1
`cmovl`	2 2
`cmove`	2 3
`cmovne`	2 4
`cmovge`	2 5
`cmovg`	2 6

`jmp`	7 0
`jle`	7 1
`jl`	7 2
`je`	7 3
`jne`	7 4
`jge`	7 5
`jg`	7 6

`%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