Y86-64 ISA

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

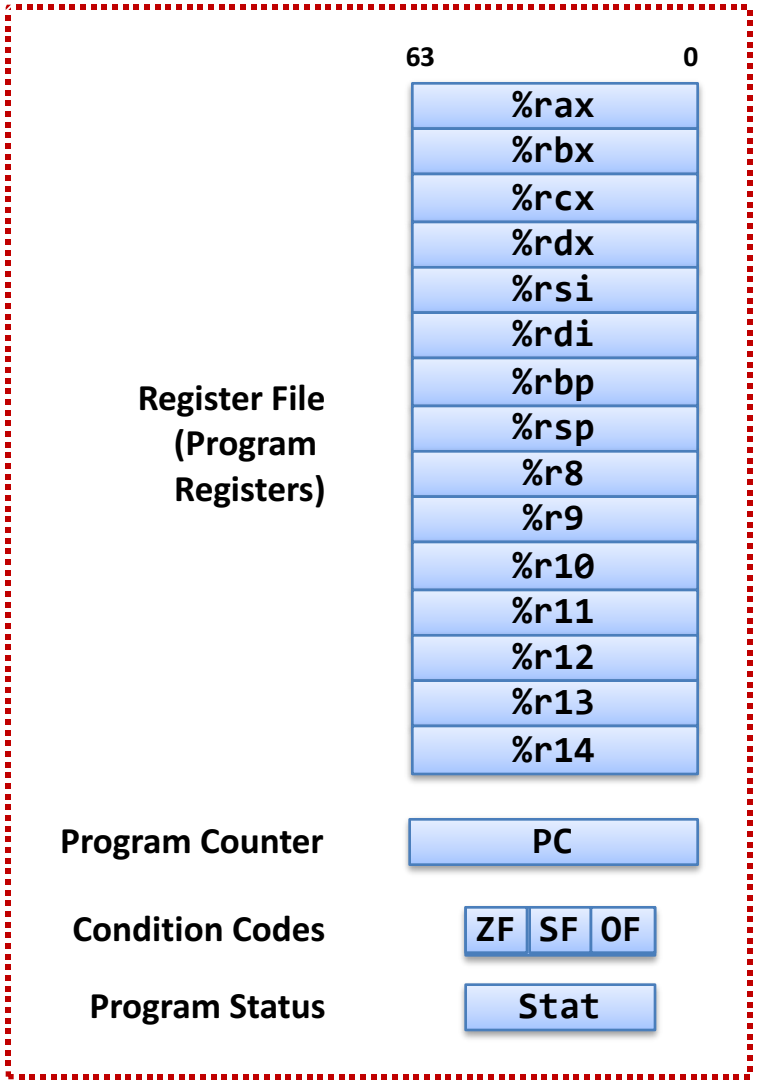
Spring 2019

Y86-64

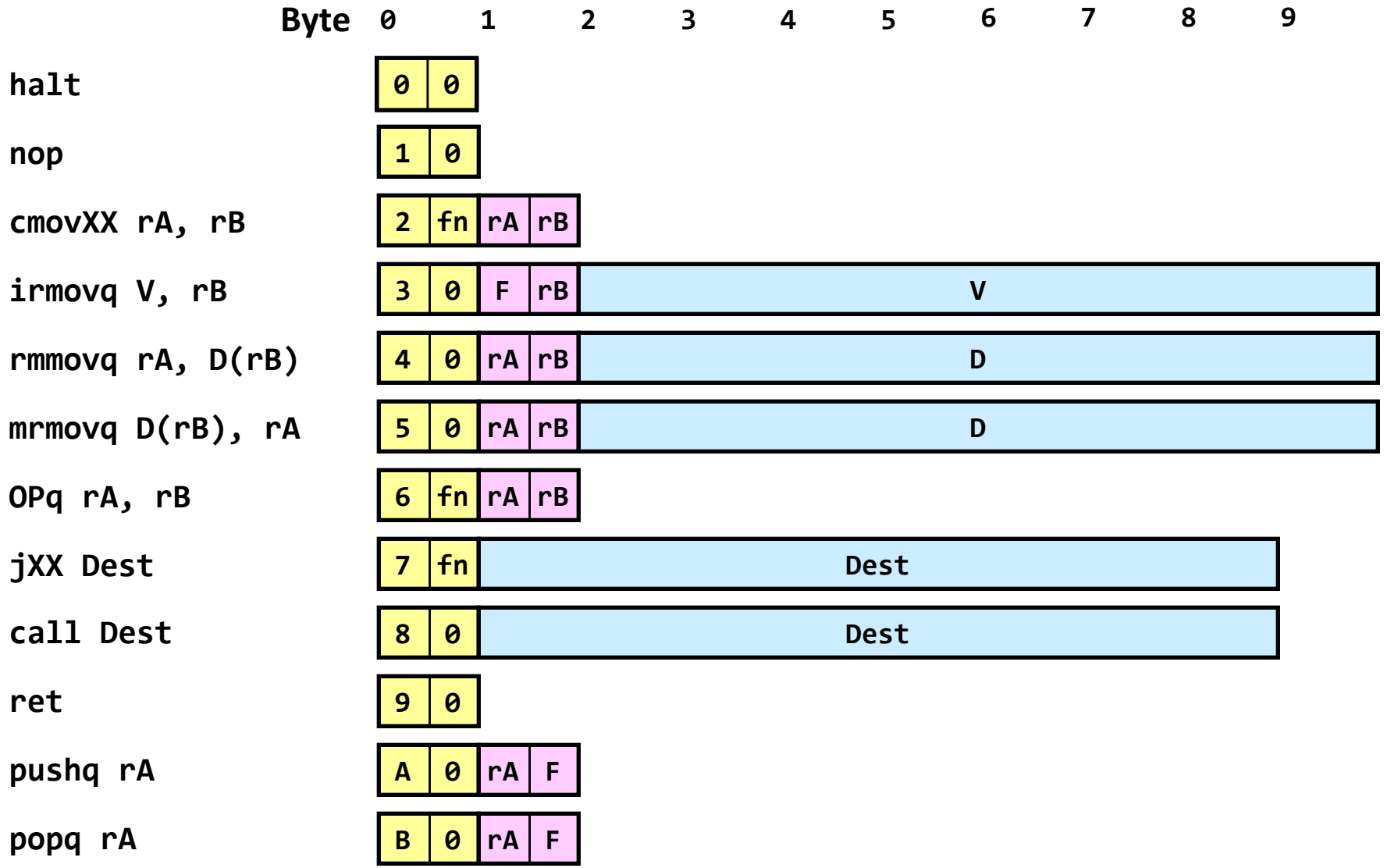
- A strip-down version of x86-64 created by textbook authors for educational purposes
  - Similar state and instructions
  - Simpler encodings
  - Somewhere between CISC and RISC
- We will work through a CPU design example with Y86-64 in Chap. 4
- Y86-64 toolset available at http://csapp.cs.cmu.edu/3e/sim.tar
  - yas (assembler), yis (ISA simulator)
  - hcl2c (HCL to C translator), hcl2v (HCL to Verilog translator)
  - ssim, ssim+, psim (hardware simulator)

Y86-64 Processor State

- Program Registers
  - 15 registers (no %r15), each 64 bits
- Condition codes
  - Single-bit flags set by arithmetic or logical instructions
  - ZF: Zero, SF: Negative, OF: Overflow
- Program Counter: address of next instruction
- Program Status: normal vs. error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order



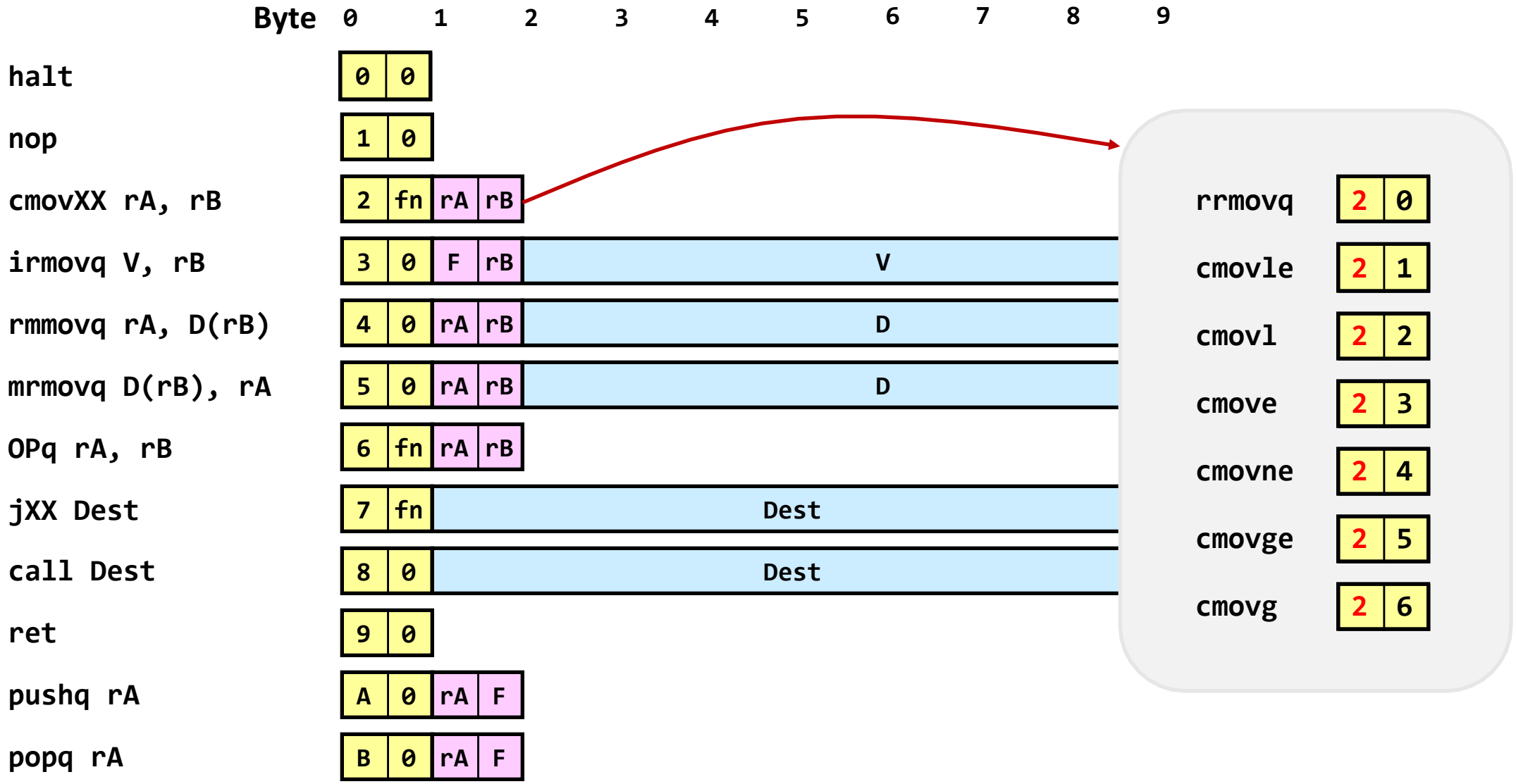
Y86-64 Instruction Set



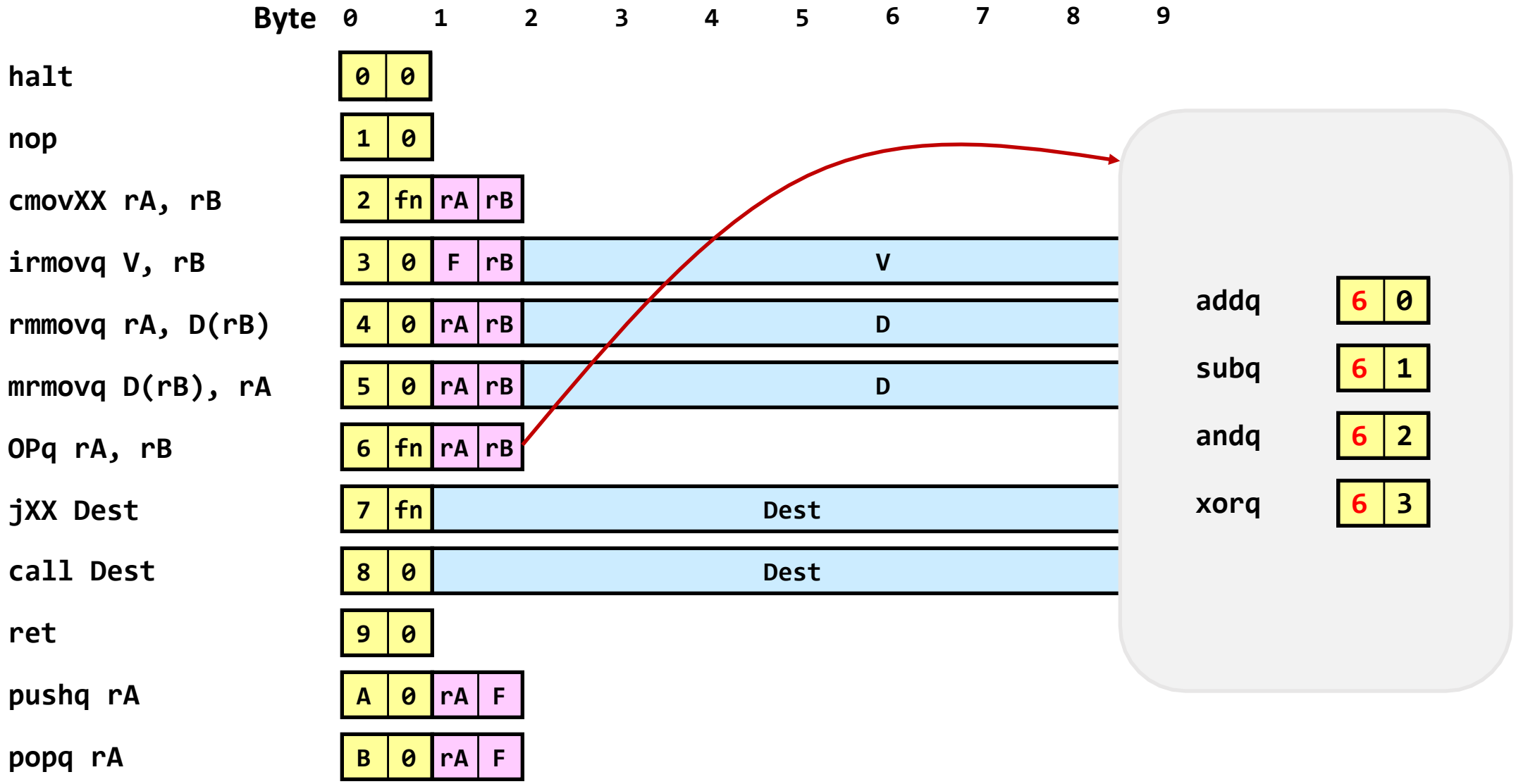
Y86-64 Instructions

- 1 – 10 bytes of information read from memory
  - Can determine instruction length from first byte
- Only supports 64-bit operations
- RISC style
  - Not as many instruction types, and simpler encoding than with x86-64
  - Simple addressing mode: `D(rA)`
  - ALU instructions operate on registers (not memory)
  - Registers are specified in the fixed location, if any
- Each accesses and modifies some part(s) of the program state

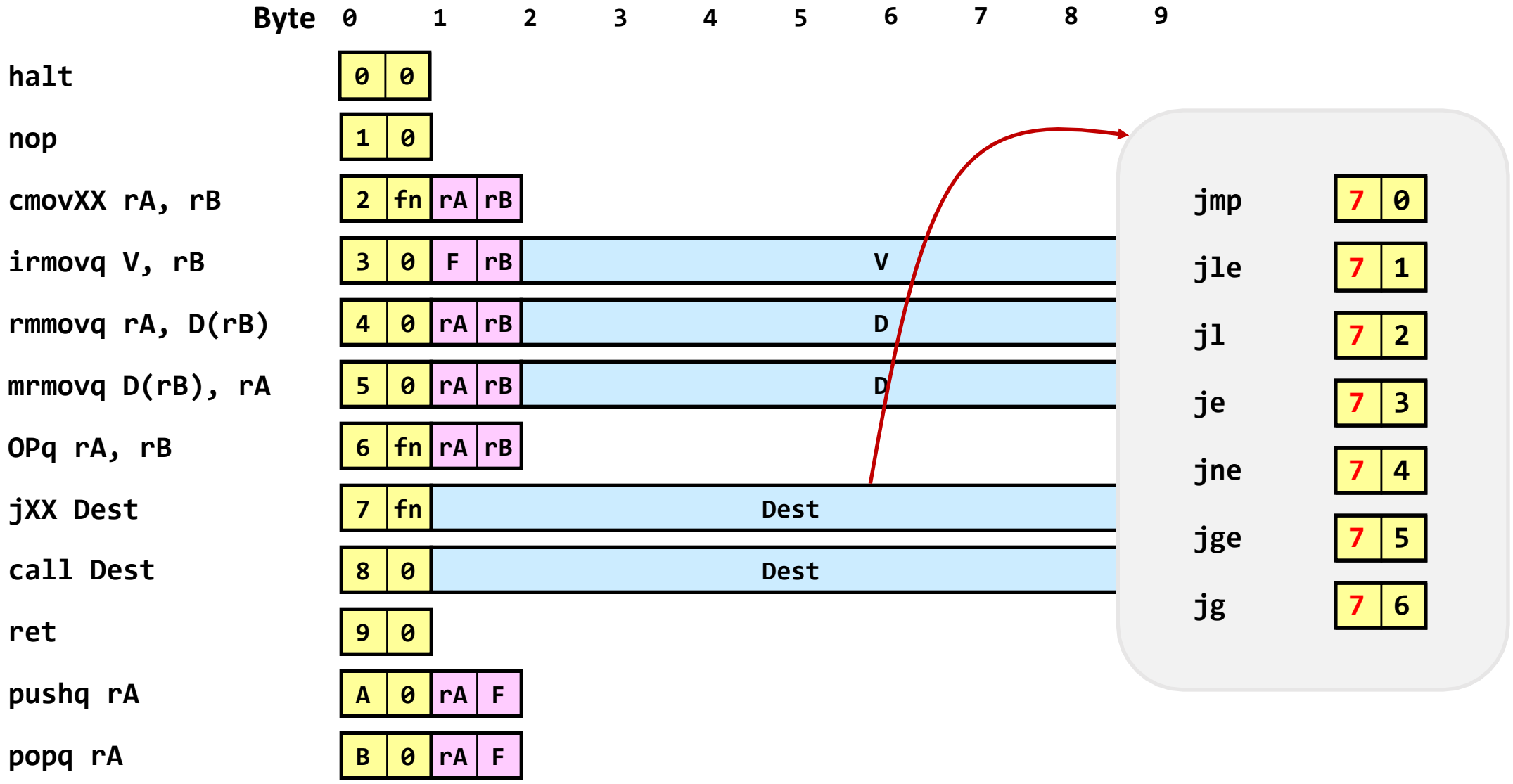
Y86-64 Conditional Move Instructions



Y86-64 ALU Instructions



Y86-64 Conditional Branch Instructions



Encoding Registers

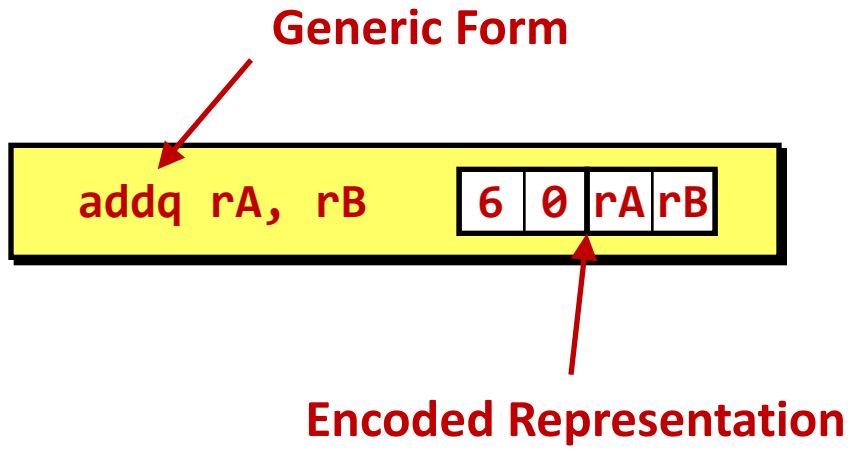
- Each register has 4-bit ID
  - Same encoding as in x86-64

- Register ID 15 (0xf) indicates "no register"
  - Will use this in our hardware design in multiple places

%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

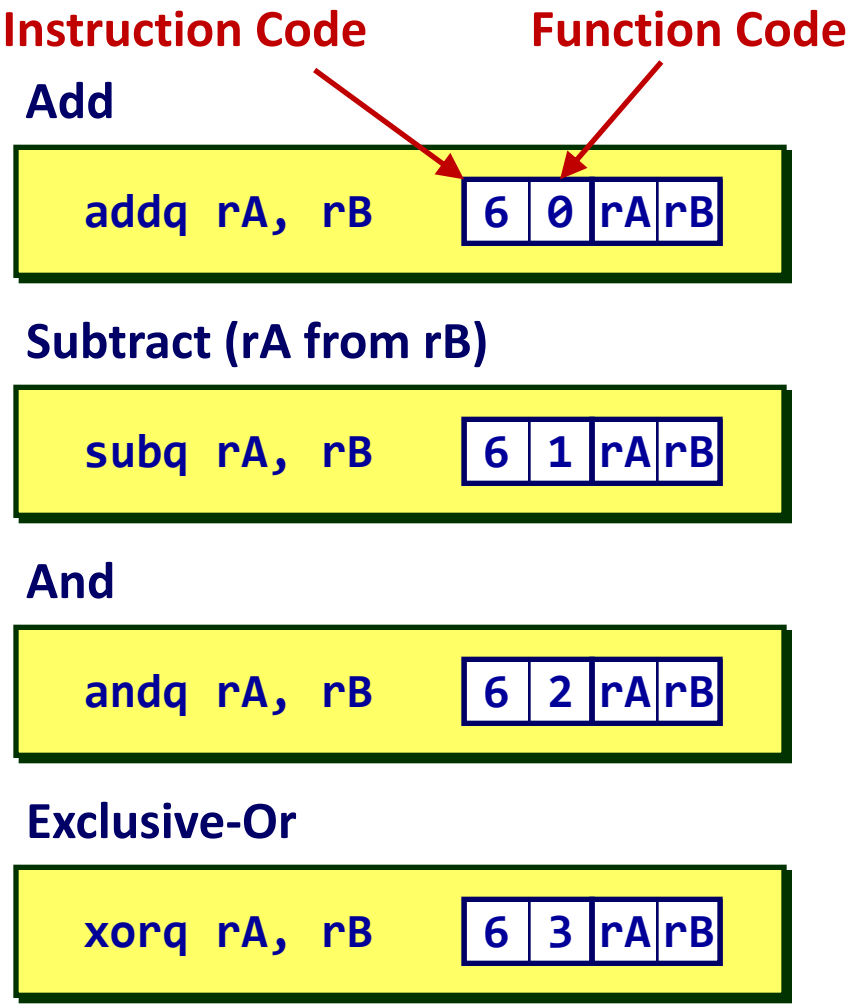
Instruction Example: addq

- Add value in register rA to that in register rB
  - Store result in register rB
  - Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers
  - e.g. `addq %rax, %rsi` → Encoding: `60 06`



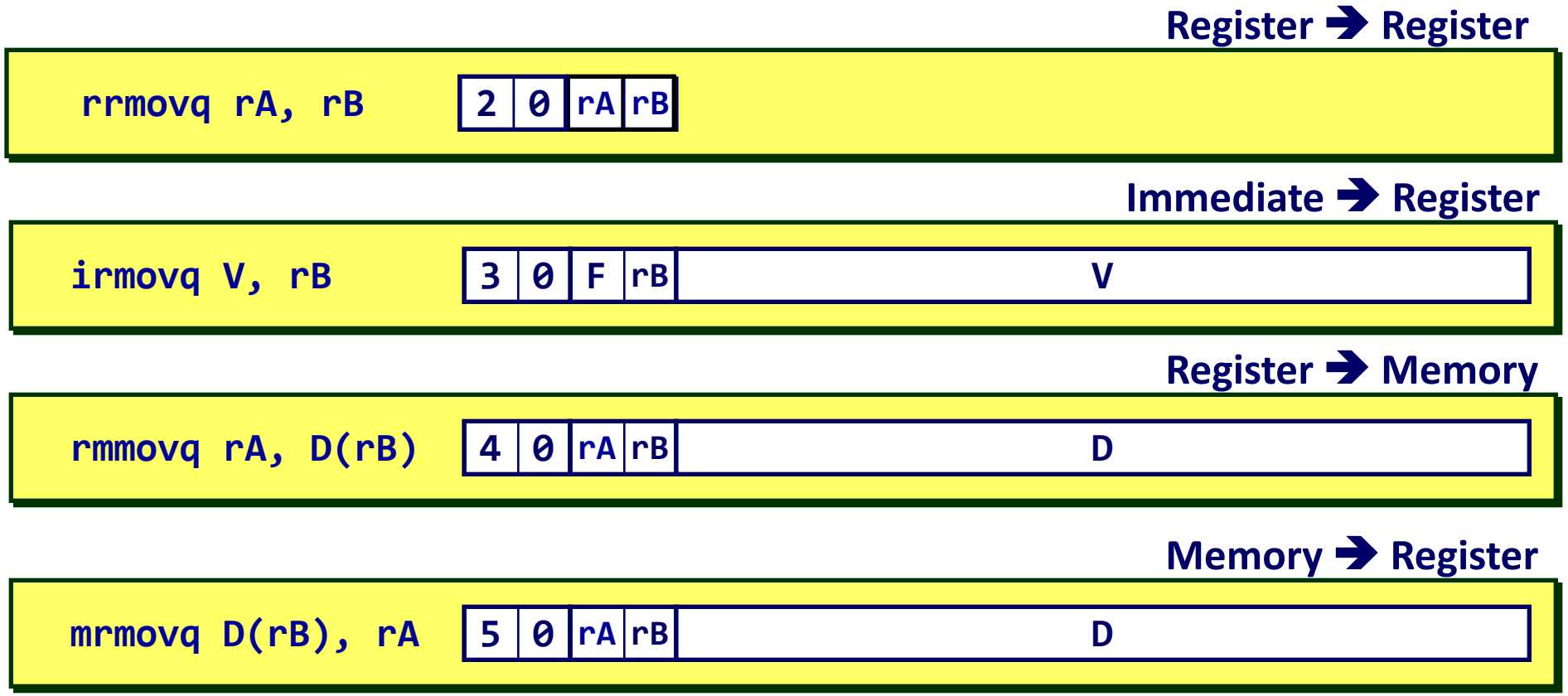
ALU Operations

- Refer to generically as "OPq"
  - rB ← rB *OPq* rA
- Encodings differ only by "function code"
  - Low-order 4 bytes in first instruction word
- Set condition codes as side effect



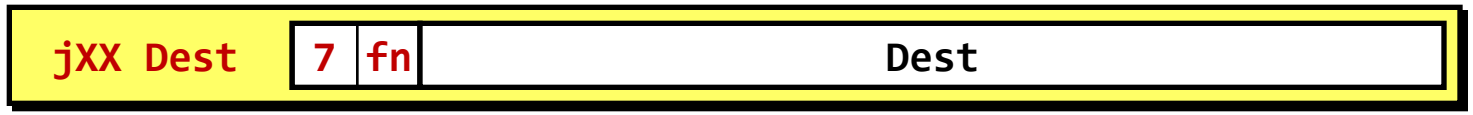
Move Operations

- Like the x86-64 movq instruction
- Simpler format for memory addresses



Jump Instructions

- Jump (conditionally)



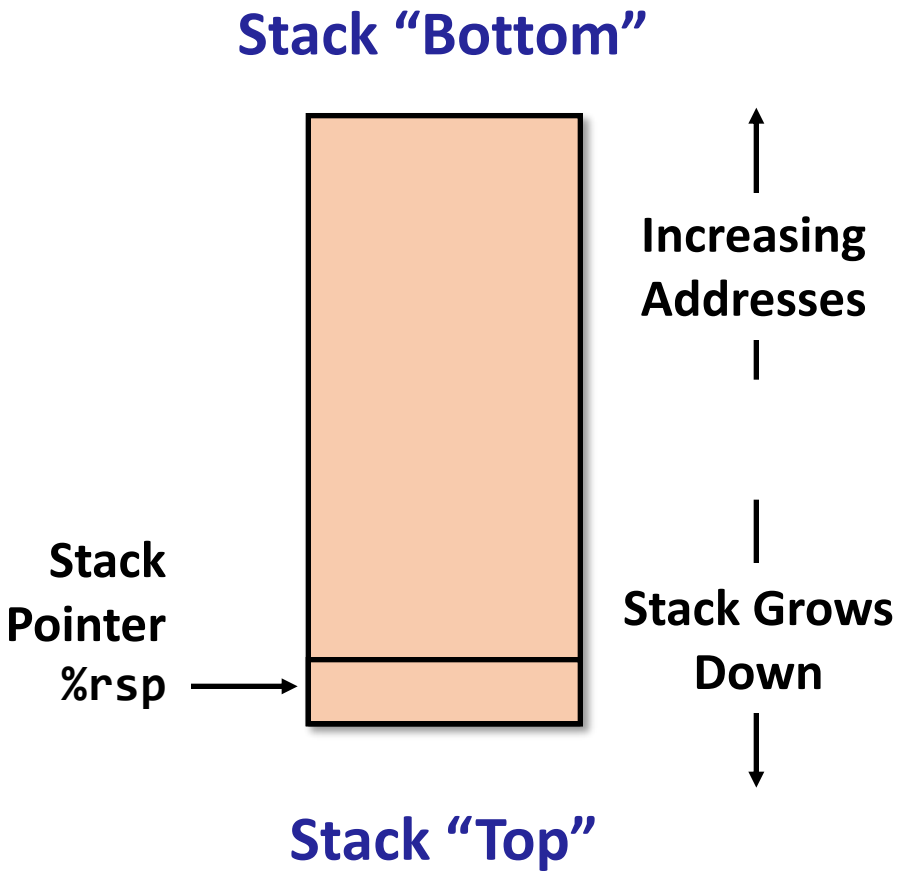
- Refer to generically as "jXX"
- Encodings differ only by "function code" fn
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in x86-64

Conditional Branch 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 Stack

- Same as in x86-64
  - Region of memory holding program data
  - Used for supporting procedure calls
- Stack top indicated by `%rsp`
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at lowest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer



Stack Operations

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

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

Subroutine Call and Return

- **`call`**

  - 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

Miscellaneous Instructions

- **`nop`**
  - 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

Status Conditions

- AOK
  - Normal operation
- HLT
  - Halt instruction encountered
- ADR
  - Bad address (either instruction or data) encountered
- INS
  - Invalid instruction encountered

Mnemonic	Code
AOK	1

Mnemonic	Code
HLT	2

Mnemonic	Code
ADR	3

Mnemonic	Code
INS	4

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

Example: max.ys

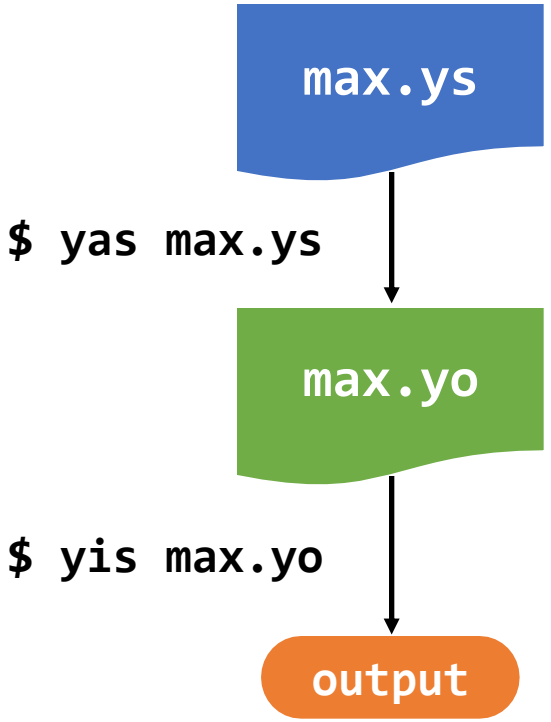
```
# Execution begins at address 0
    .pos 0
    irmovq stack, %rsp        # Set up stack pointer
    call main                 # Call main
    halt                      # Terminate program

# Global data
    .align 8
array:
    .quad 0x0000000000000003
    .quad 0x0000000000000004
    .quad 0x00000b000b000b00

main:
    irmovq array, %rbx        # %rbx <- array
    mrmovq (%rbx), %rdi       # %rdi <- array[0]
    mrmovq 8(%rbx), %rsi      # %rsi <- array[1]
    call max                  # Call max
    rmmovq %rax, 16(%rbx)     # array[2] <- %rax
    ret

# long max(long x, y)
max:
    rrmovq %rdi, %rax         # %rax <- x
    subq %rsi, %rdi           # %rdi <- x - y; set flag
    cmovl %rsi, %rax          # if (x < y) %rax <- y
    ret

# The stack starts here and grows to lower addresses
    .pos 0x200
stack:
```

Example: max.yo

- `yas max.ys`

```
                            | # Execution begins at address 0
0x000:                      |     .pos 0
0x000: 30f4000200000000000  |     irmovq stack, %rsp      # Set up stack pointer
0x00a: 8030000000000000     |     call main               # Call main
0x013: 00                   |     halt                    # Terminate program
                            |
                            | # Global data
0x018:                      |     .align 8
0x018:                      | array:
0x018: 0300000000000000     |     .quad 0x0000000000000003
0x020: 0400000000000000     |     .quad 0x0000000000000004
0x028: 000b000b000b0000     |     .quad 0x00000b000b000b00
                            |
0x030:                      | main:
0x030: 30f3180000000000000  |     irmovq array, %rbx      # %rbx <- array
0x03a: 5073000000000000000  |     mrmovq (%rbx), %rdi     # %rdi <- array[0]
0x044: 5063080000000000000  |     mrmovq 8(%rbx), %rsi    # %rsi <- array[1]
0x04e: 8062000000000000     |     call max                # Call max
0x057: 4003100000000000000  |     rmmovq %rax, 16(%rbx)   # array[2] <- %rax
0x061: 90                   |     ret
                            |
                            | # long max(long x, y)
0x062:                      | max:
0x062: 2070                 |     rrmovq %rdi, %rax       # %rax <- x
0x064: 6167                 |     subq %rsi, %rdi         # %rdi <- x - y; set flag
0x066: 2260                 |     cmovl %rsi, %rax        # if (x < y) %rax <- y
0x068: 90                   |     ret
                            |
                            | # The stack starts here and grows to lower addresses
0x200:                      |   .pos 0x200
0x200:                      | stack:
```

Example: Running max.yo on yis




```
$ yis max.yo
Stopped in 13 steps at PC = 0x13.  Status 'HLT', CC Z=0 S=1 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000004
%rbx:   0x0000000000000000      0x0000000000000018
%rsp:   0x0000000000000000      0x0000000000000200
%rsi:   0x0000000000000000      0x0000000000000004
%rdi:   0x0000000000000000      0xffffffffffffffff

Changes to memory:
0x0028: 0x00000b000b000b00      0x0000000000000004
0x01f0: 0x0000000000000000      0x0000000000000057
0x01f8: 0x0000000000000000      0x0000000000000013
```

CISC (Complex Instruction Set Computer)

- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- Arithmetic instructions can access memory
  - Requires memory read and write: e.g. `addq %rax, 12(%rbx,%rcx,8)`
  - Complex address calculation
- Condition codes
  - Set as side effect of arithmetic and logical instructions
- Philosophy
  - Add instructions to perform "typical" programming tasks
  - DEC PDP-11 & VAX, IBM System/360, Motorola 68000, IA32, x86-64, …

RISC (Reduced Instruction Set Computer)

- Fewer, simpler instructions
  - Might take more to get given task done
  - Can execute them with small and fast hardware
  - Stanford MIPS, UCB RISC, Sun SPARC, IBM Power/PowerPC, ARM, SuperH, …
- 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

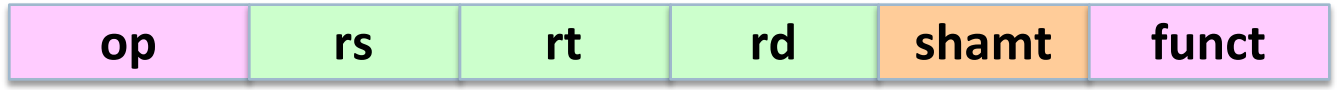
#	Name	Usage
0	$zero	The constant value 0
1	$at	Assembler temporary
2	$v0	Values for results and expression evaluation
3	$v1	
4	$a0	Arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	Temporaries (Caller-save registers)
9	$t1	
10	$t2	
11	$t3	
12	$t4	
13	$t5	
14	$t6	
15	$t7	

#	Name	Usage
16	$s0	Saved temporaries (Callee-save registers)
17	$s1	
18	$s2	
19	$s3	
20	$s4	
21	$s5	
22	$s6	
23	$s7	
24	$t8	More temporaries (Caller-save registers)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

MIPS Instruction Formats

```
add   $3, $2, $1        ; R[3] = R[2] + R[1]
```

R-type




```
addu  $3, $2, 1234      ; R[3] = R[2] + 1234
lw    $3, $2, 12        ; R[3] = Mem[R[2] + 12]
beq   $3, $2, dest      ; if (R[3] == R[2])
                            PC = PC + 4 + dest
```

I-type


```
j     dest              ; PC = dest
jal   dest              ; R[31] = PC + 4, PC = dest
```

J-type

Summary: CISC vs. RISC

- Original debate
  - 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 (e.g. most cell phones use ARM processor)