The Y86-64 Instruction Set Architecture

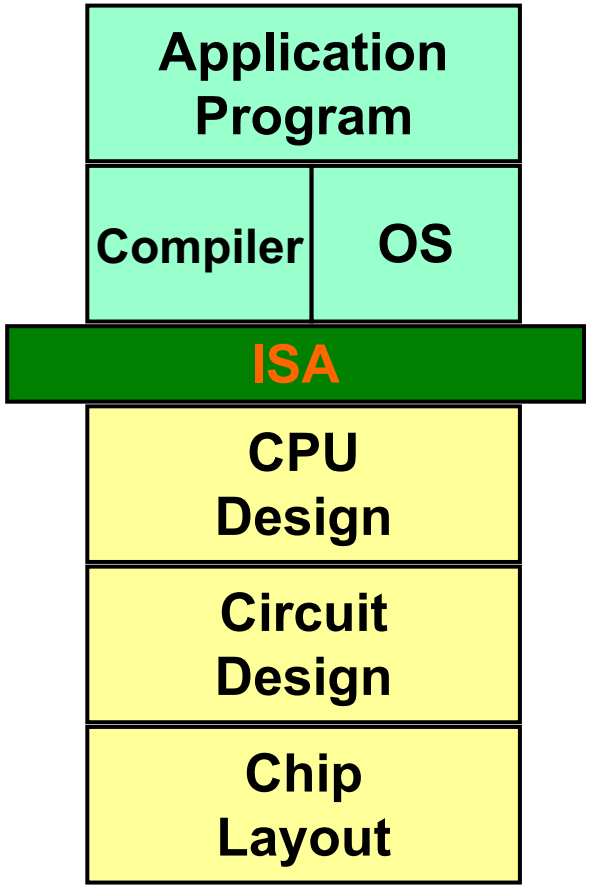
Lecture 8
October 18th, 2018

Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: [CS:APP3e] slides from CMU; [COD5e] slides from Elsevier Inc.*

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

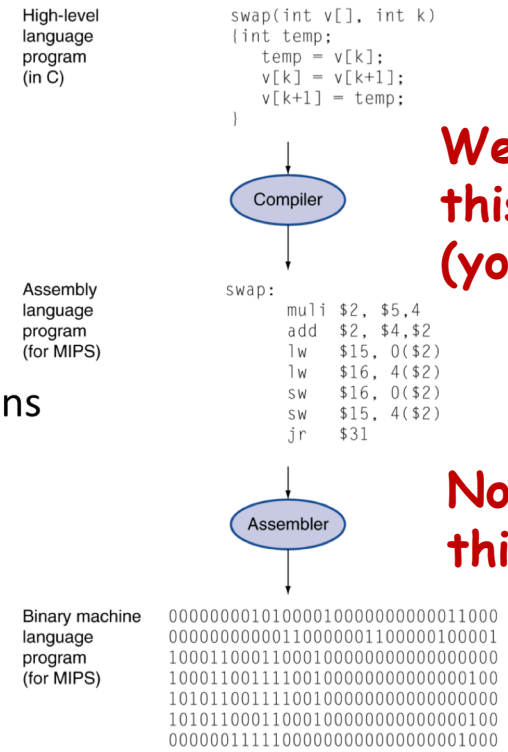


Do you remember…?

- From Lecture 1

Understanding performance: Levels of program code

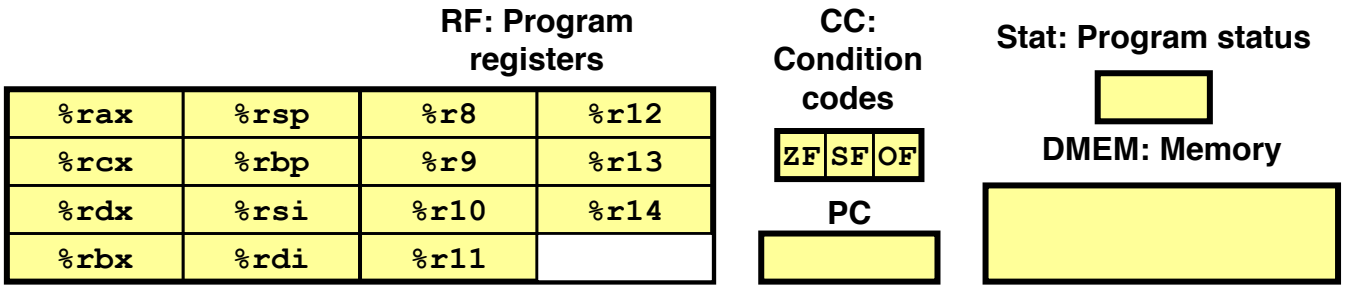
- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data



We already looked at this part.
(you = human compiler)

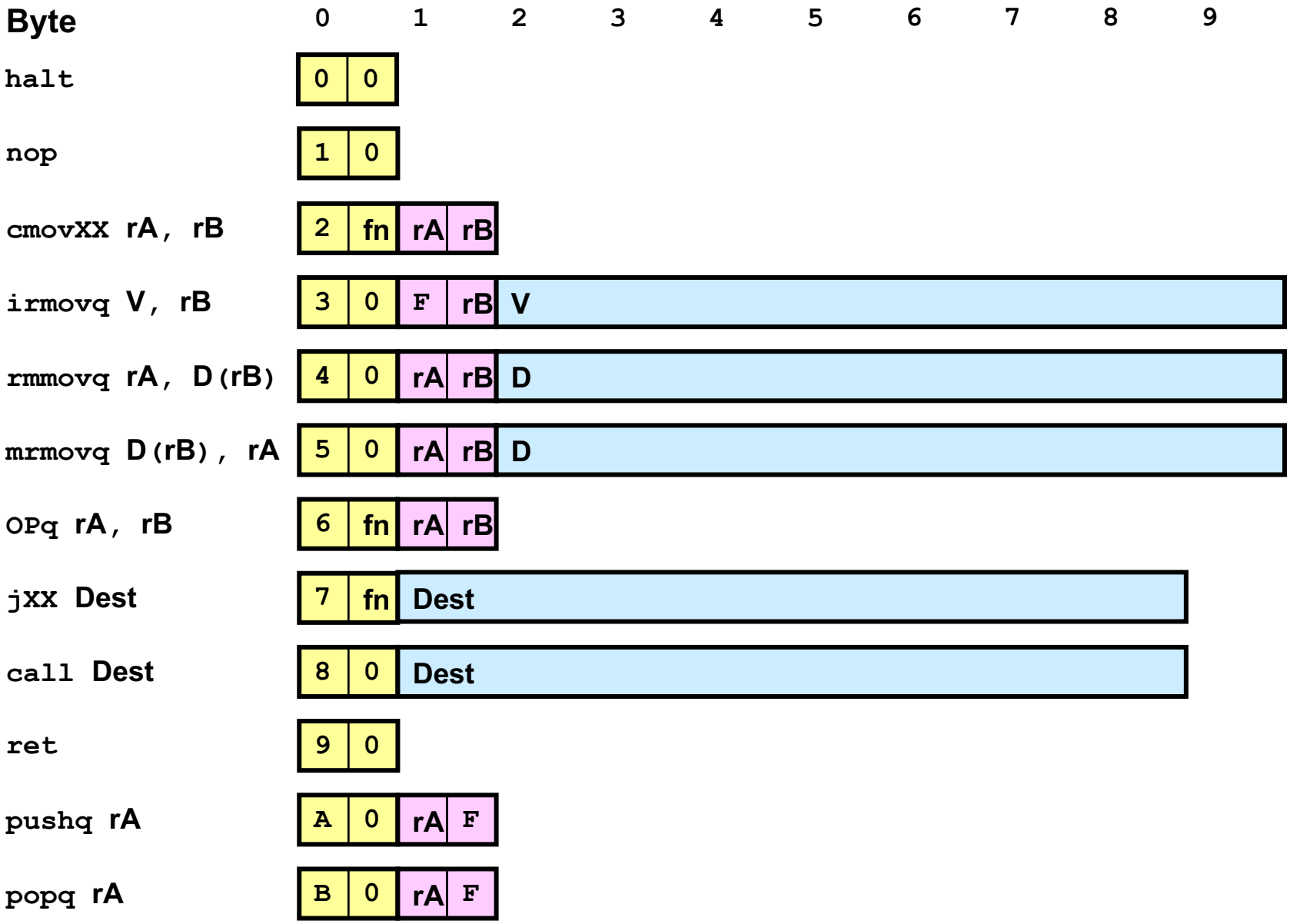
Now we will look at this part!

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

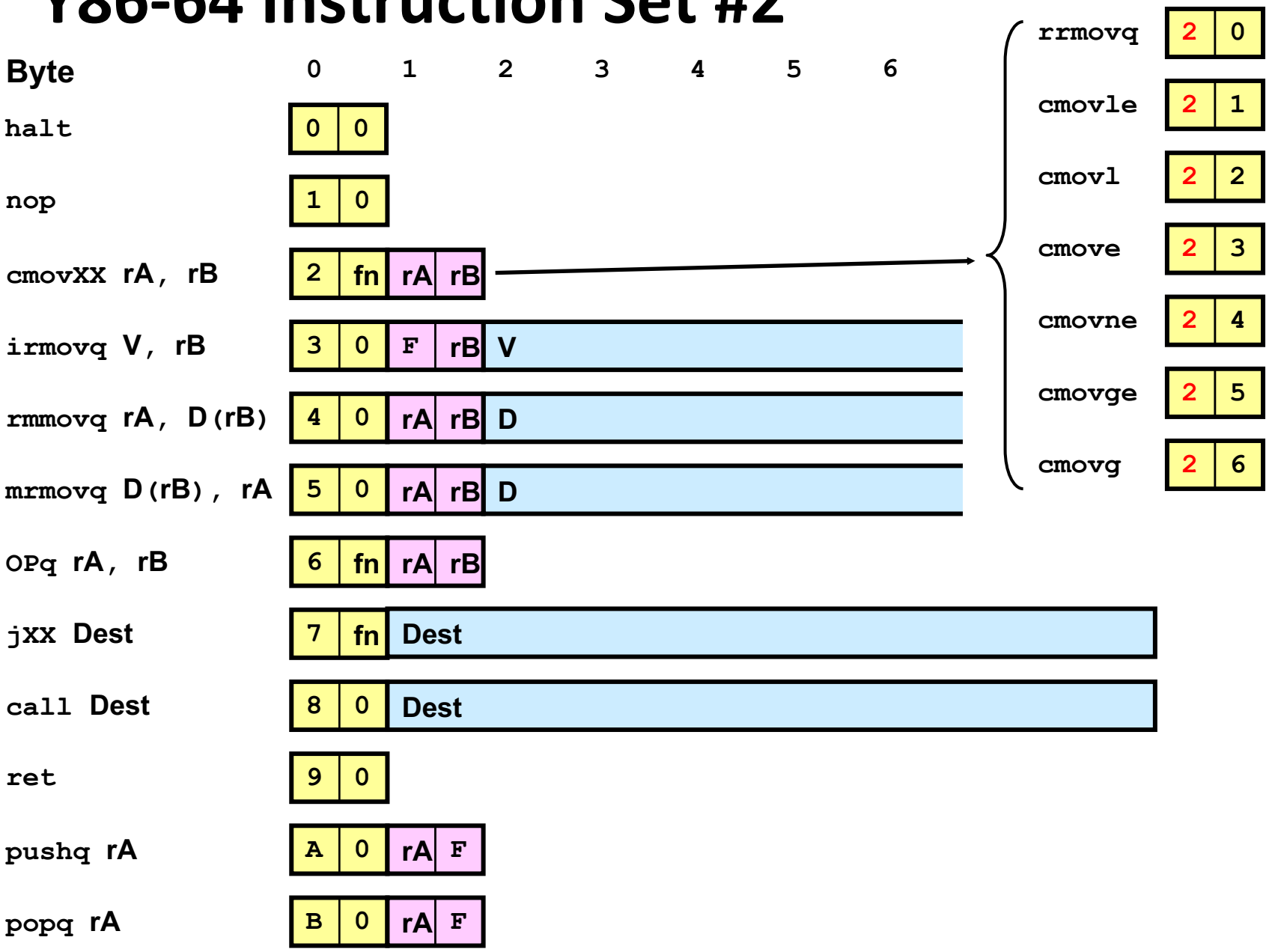
Y86-64 Instruction Set #1



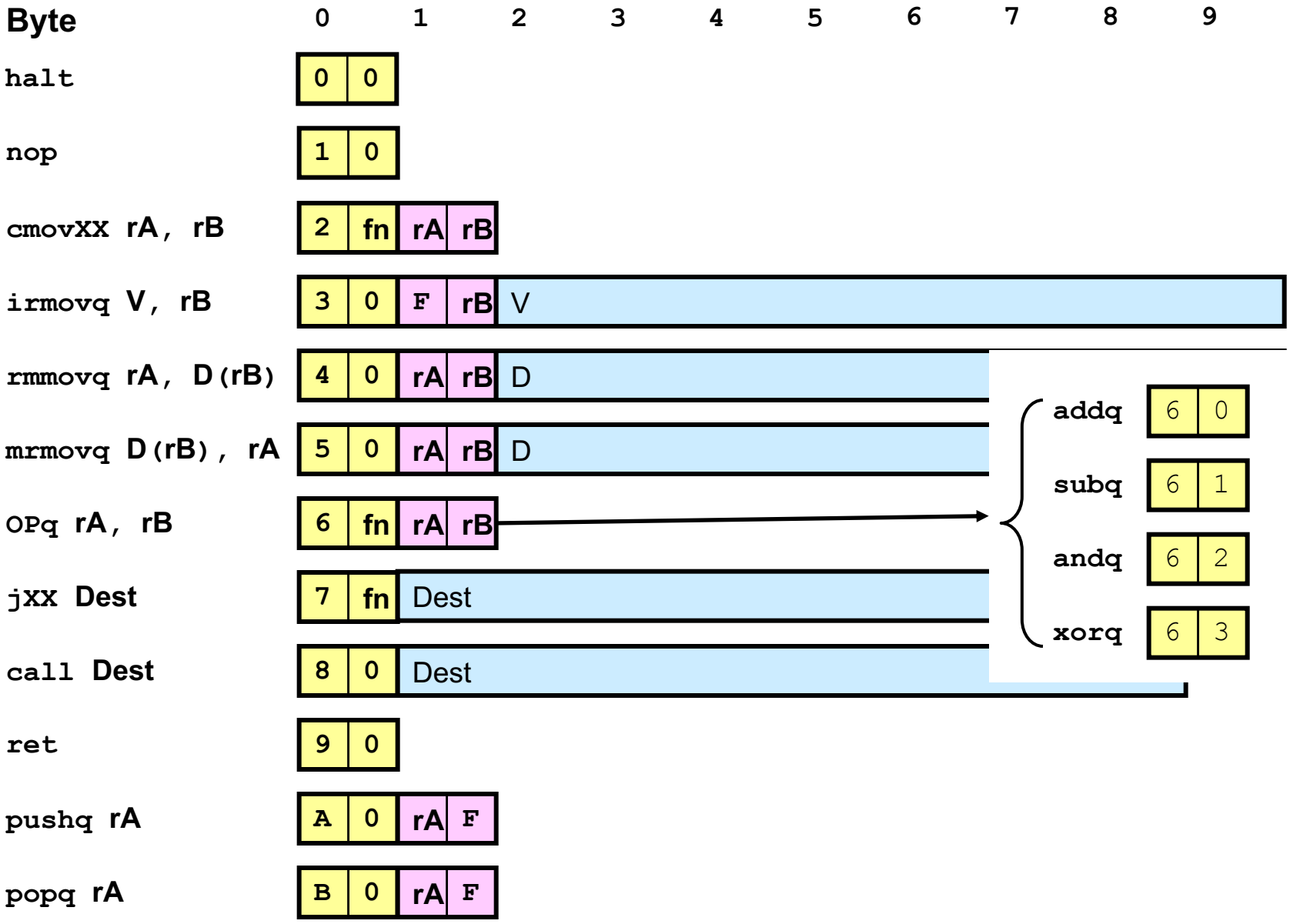
Y86-64 Instructions

- **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

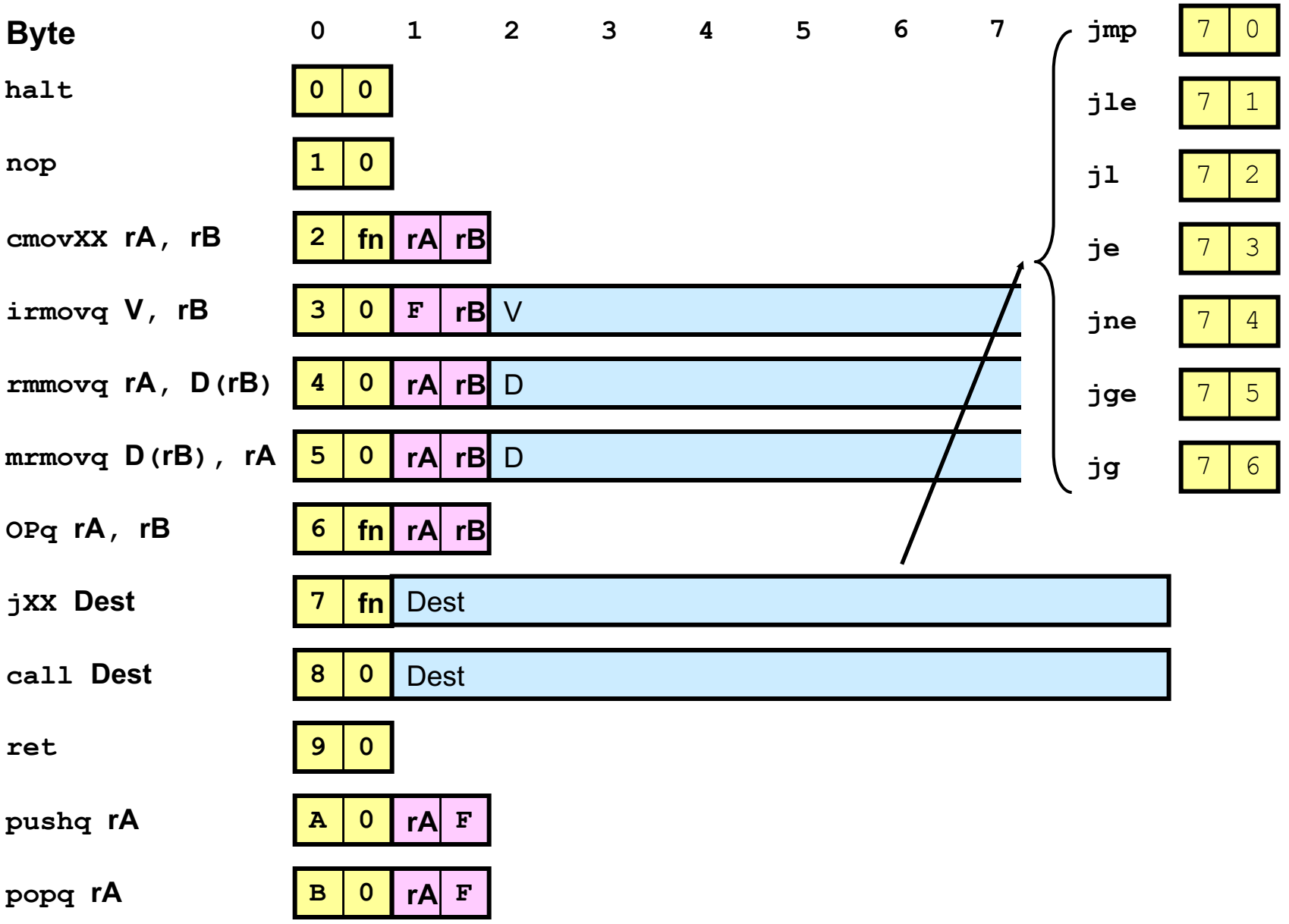
Y86-64 Instruction Set #2



Y86-64 Instruction Set #3



Y86-64 Instruction Set #4



Encoding Registers

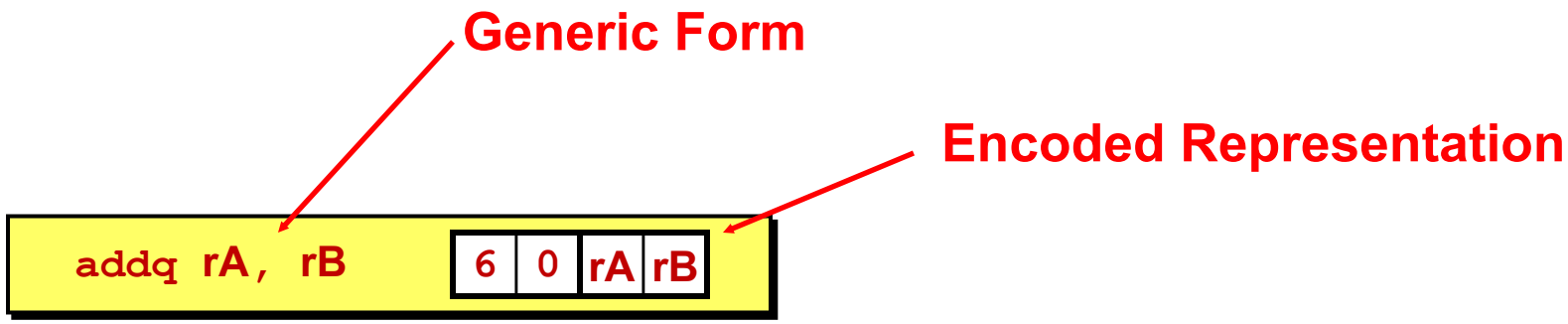
- **Each register has 4-bit ID**

%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

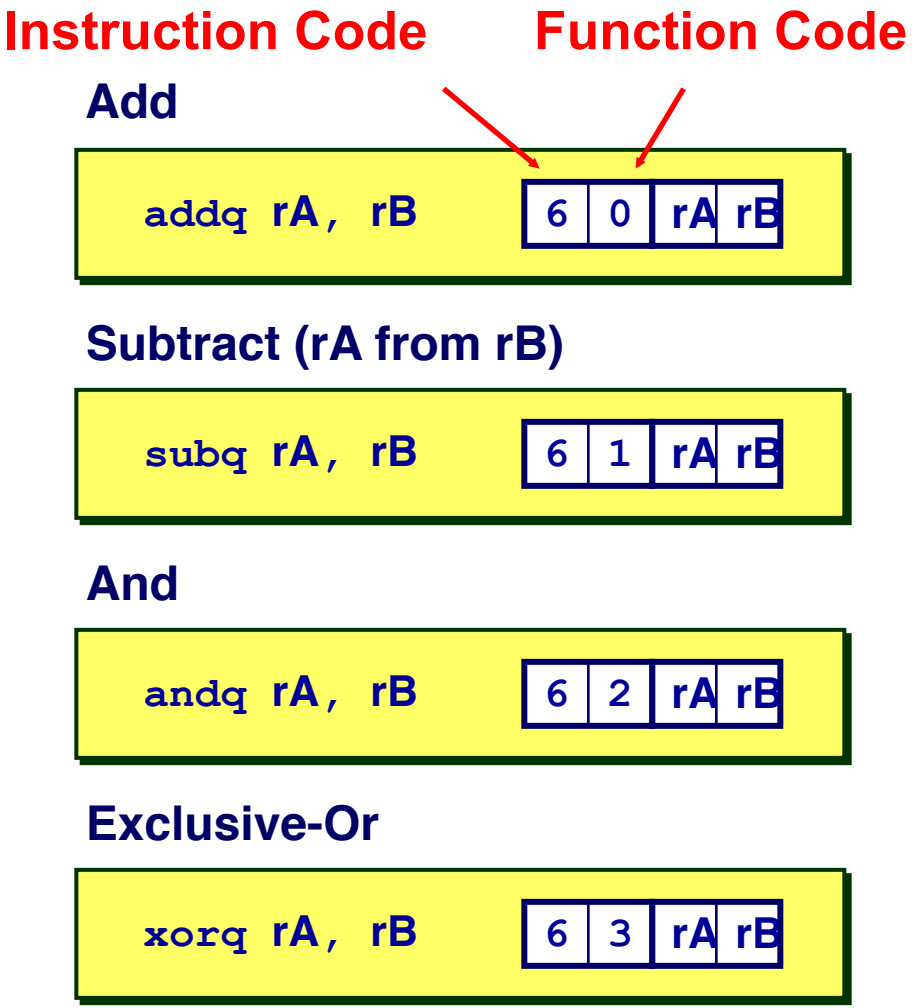
Instruction Example

- **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 "`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

Move Instruction Examples

X86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`
Encoding:	`30 82 cd ab 00 00 00 00 00 00`
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`
Encoding:	`20 43`
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`
Encoding:	`50 15 f4 ff ff ff ff ff ff ff`
`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

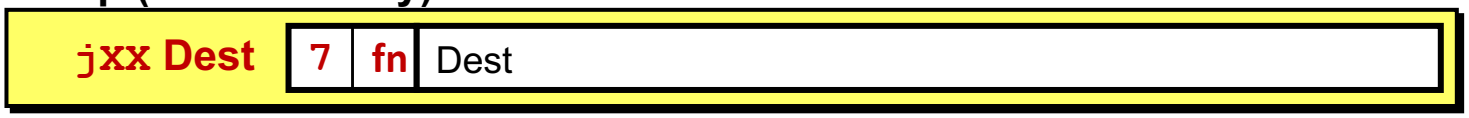
Move When Greater

`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

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

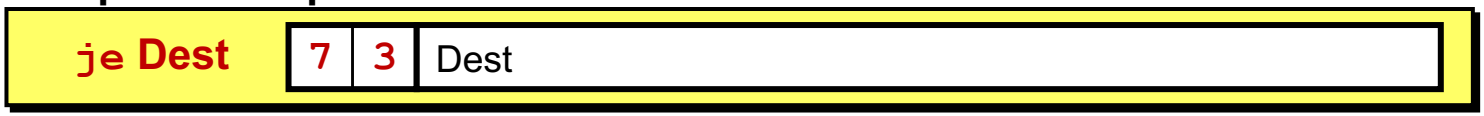
Jump Instructions

Jump Unconditionally

Jump When Less or Equal

Jump When Less

Jump When Equal

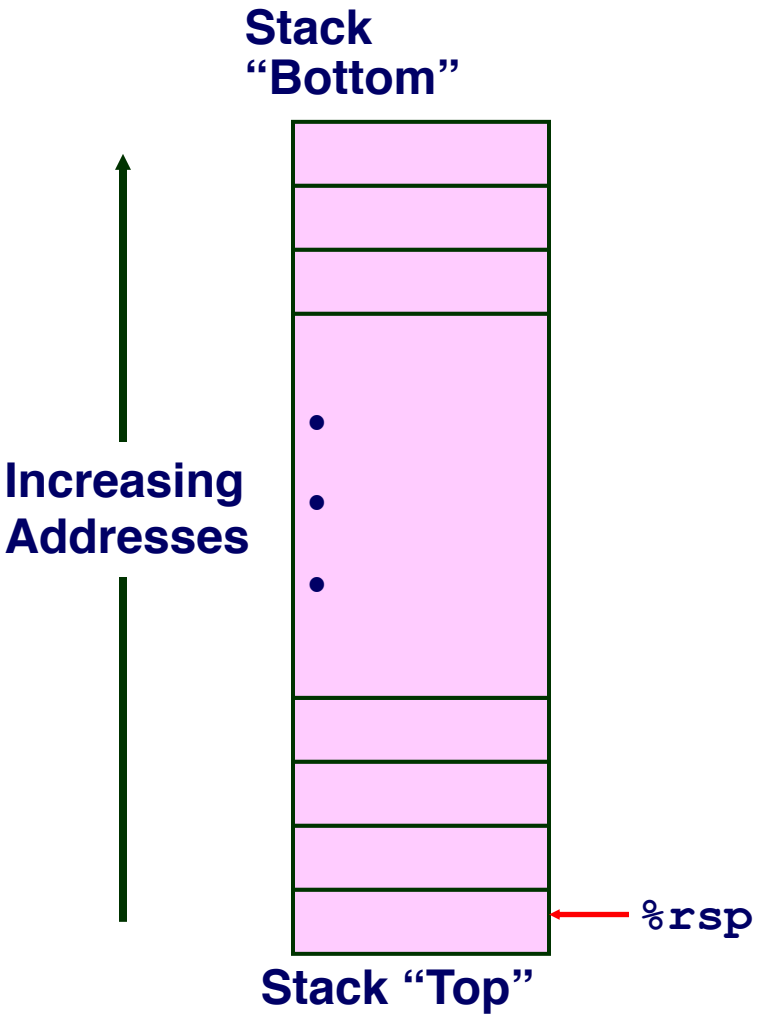


Jump When Not Equal

Jump When Greater or Equal

Jump When Greater

Y86-64 Program Stack



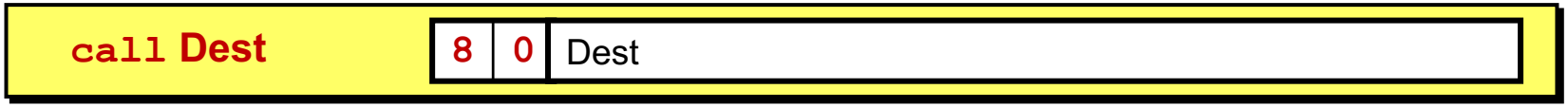
- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by `%rsp`
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at ~~highest~~ lowest address in the stack
  - When pushing, must first decrement 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

- 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

CISC Instruction Sets

- Complex Instruction Set Computer
- IA32 (and x86-64) is example

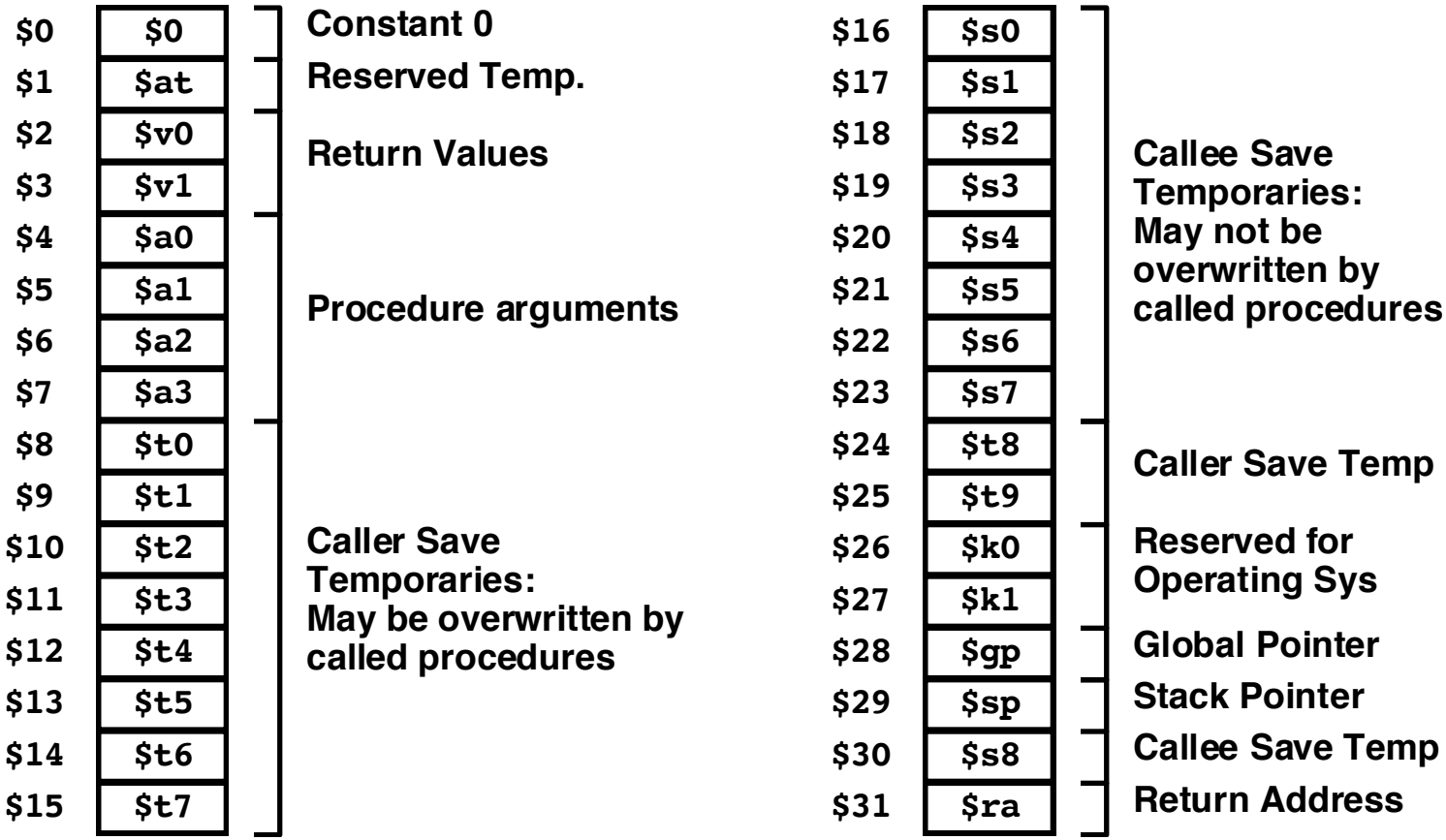
- **Stack-oriented instruction set**
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- **Arithmetic instructions can access memory**
  - `addq %rax, 12(%rbx,%rcx,8)`
    - requires memory read and write
    - Complex address calculation
- **Condition codes**
  - 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)

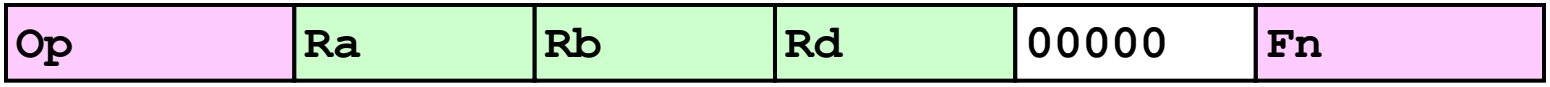
- **Fewer, simpler instructions**
  - Might take more to get given task done
  - Can execute them with small and fast hardware
- **Register-oriented instruction set**
  - Many more (typically 32) registers
  - Use for arguments, return pointer, temporaries
- **Only load and store instructions can access memory**
  - Similar to Y86-64 **`mrmovq`** and **`rmmovq`**
- **No Condition codes**
  - Test instructions return 0/1 in register

MIPS Registers



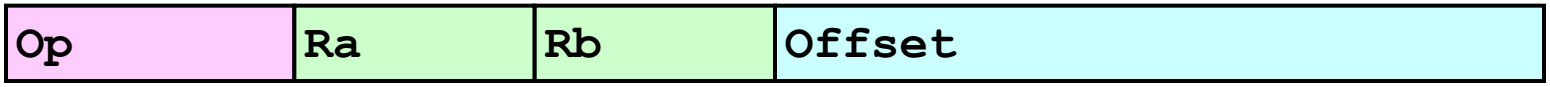
MIPS Instruction Examples

R-R



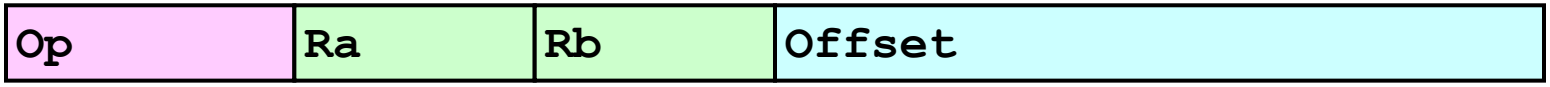

```
addu $3,$2,$1          # Register add: $3 = $2+$1
```

Load/Store




```
lw $3,16($2)           # Load Word: $3 = M[$2+16]
sw $3,16($2)           # Store Word: M[$2+16] = $3
```

Branch




```
beq $3,$2,dest         # Branch when $3 = $2
```

Jump


```
jmp  Dest              # Jump to dest
```

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 processor

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