CS:APP Chapter 4
Computer Architecture
Overview

Yuan Tang

Adapted from CMU course 15-213

Class Staff

Instructor: 唐渊

Phone: 13917047105

Email: yuantang@fudan.edu.cn

Office hour: by appointment

TA: 陈拓

Phone: 15221927371

Email: 13302010034@fudan.edu.cn

Office hour: by appointment

Grading

Exams(60%)

- Mid term (30%)
- Final (30%)
- All exams are open books/open notes.

Labs (35%)

- 4 labs(35%), (4-12% each)

Home work(5%)

Scribing (15%)

Normalize, normalize, & normalize

Course Outline

Background

- Instruction sets
- Logic design

Sequential Implementation

- A simple, but not very fast processor design

Pipelining

- Get more things running simultaneously

Pipelined Implementation

- Make it work

Advanced Topics

- Performance analysis
- High performance processor design

Coverage

Our Approach

- **Work through designs for particular instruction set**
  - **Y86---a simplified version of the Intel IA32 (a.k.a. x86).**
  - **If you know one, you more-or-less know them all**
- **Work at "microarchitectural" level**
  - **Assemble basic hardware blocks into overall processor structure**
    - » Memories, functional units, etc.
  - **Surround by control logic to make sure each instruction flows through properly**
- **Use simple hardware description language to describe control logic**
  - **Can extend and modify**
  - **Test via simulation**
  - **Route to design using Verilog Hardware Description Language**
    - » See Web aside ARCH:VLOG

Schedule

- Instruction set architecture
- Logic design

Assignment: Write & test assembly code programs

- Sequential implementation
- Pipelining and initial pipelined implementation

Assignment: Add new instructions to sequential implementation

- Making the pipeline work
- Modern processor design

Assignment: Optimize program+pipeline for maximum performance

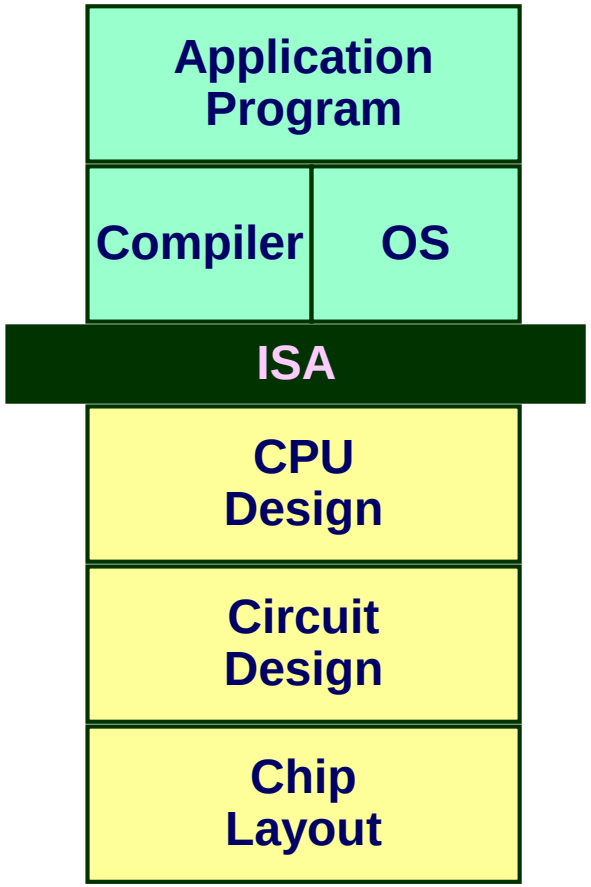
Instruction Set Architecture

Assembly Language View

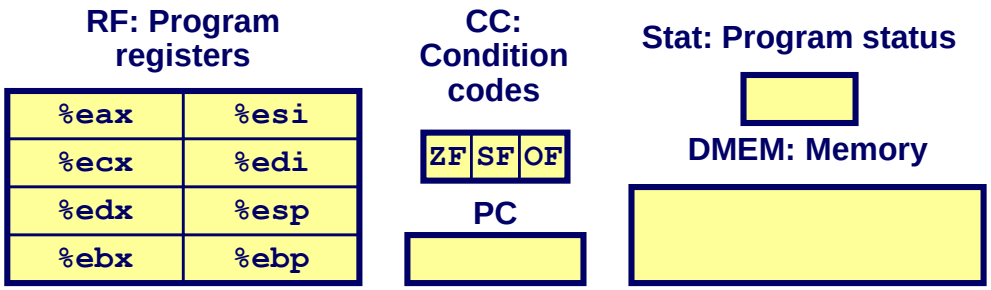
- Processor state
  - Registers, memory, …
- Instructions
  - `addl`, `pushl`, `ret`, …
  - How instructions are encoded as bytes

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

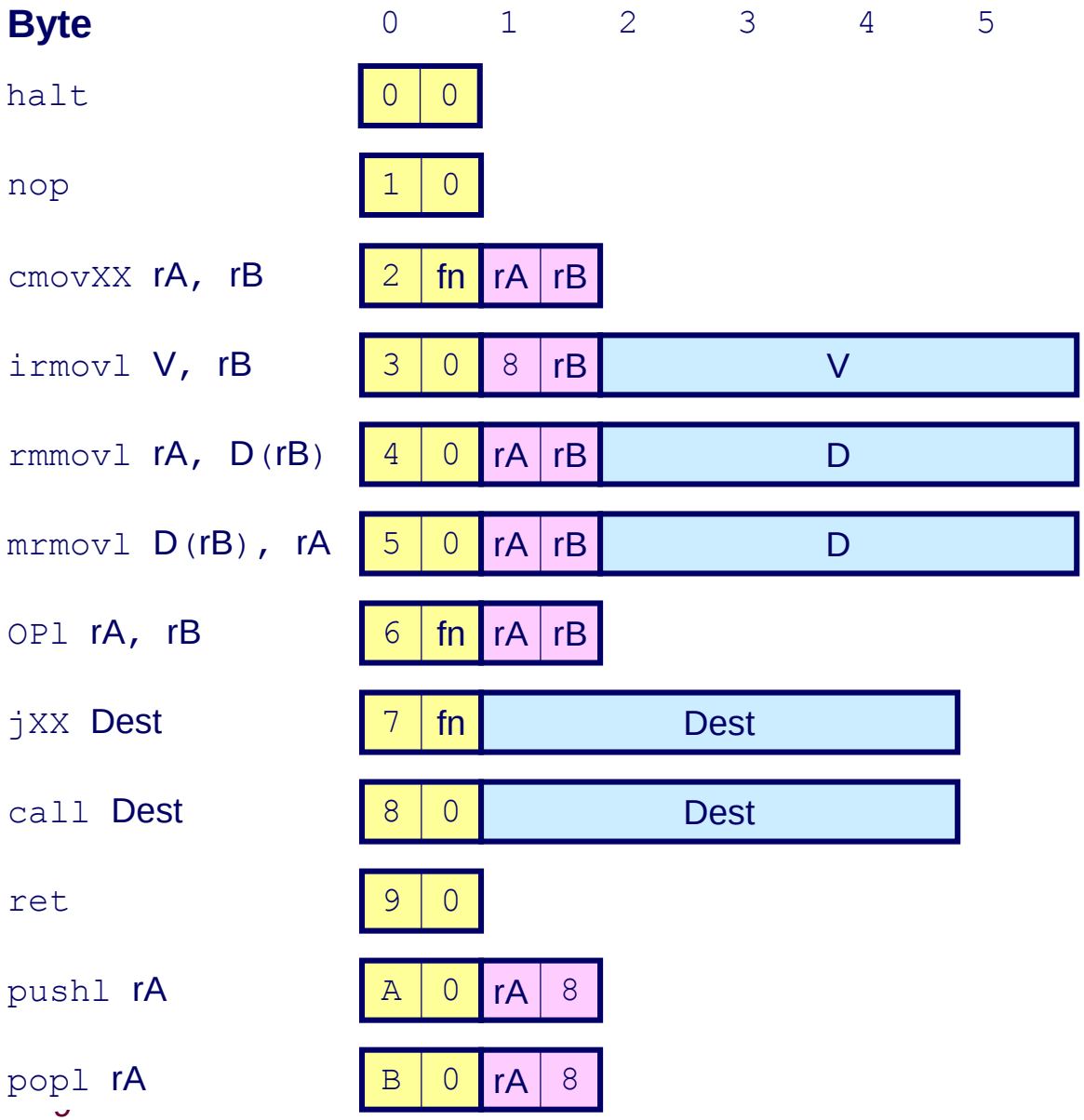


Y86 Processor State



- **Program Registers**
  - **Same 8 as with IA32. Each 32 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 Instruction Set #1

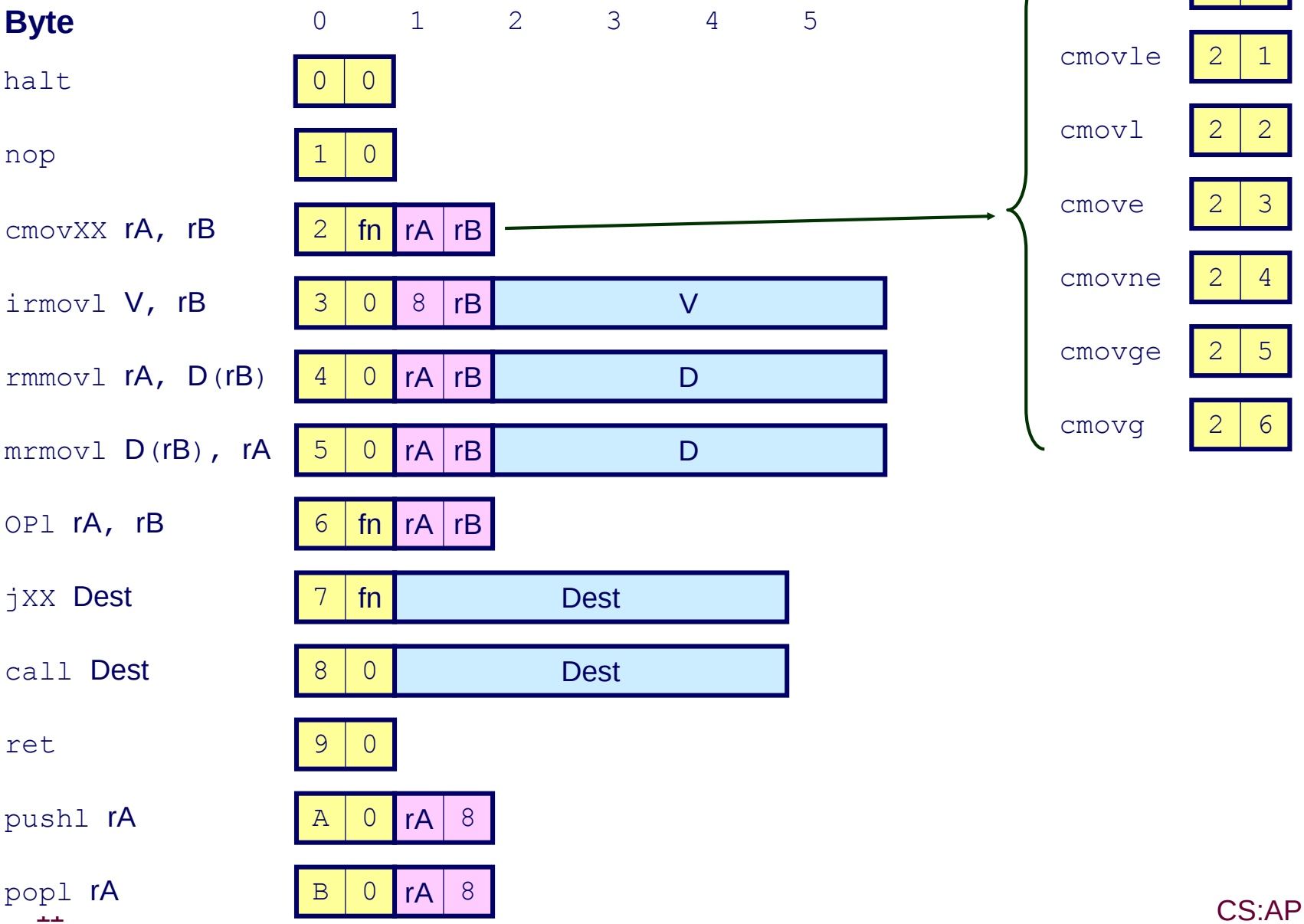


Y86 Instructions

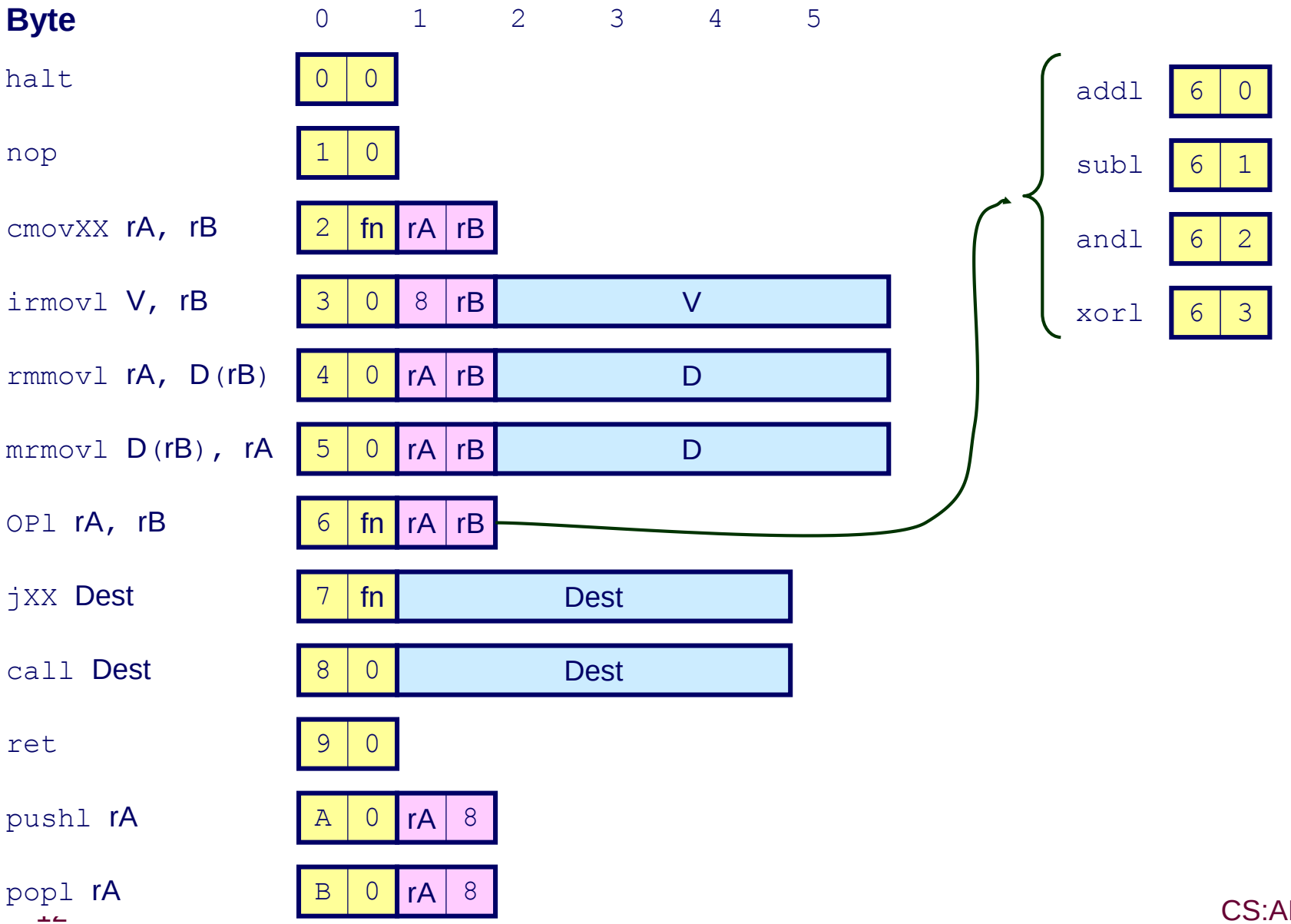
Format

- 1–6 bytes of information read from memory
  - Can determine instruction length from first byte
  - Not as many instruction types, and simpler encoding than with IA32
- Each accesses and modifies some part(s) of the program state

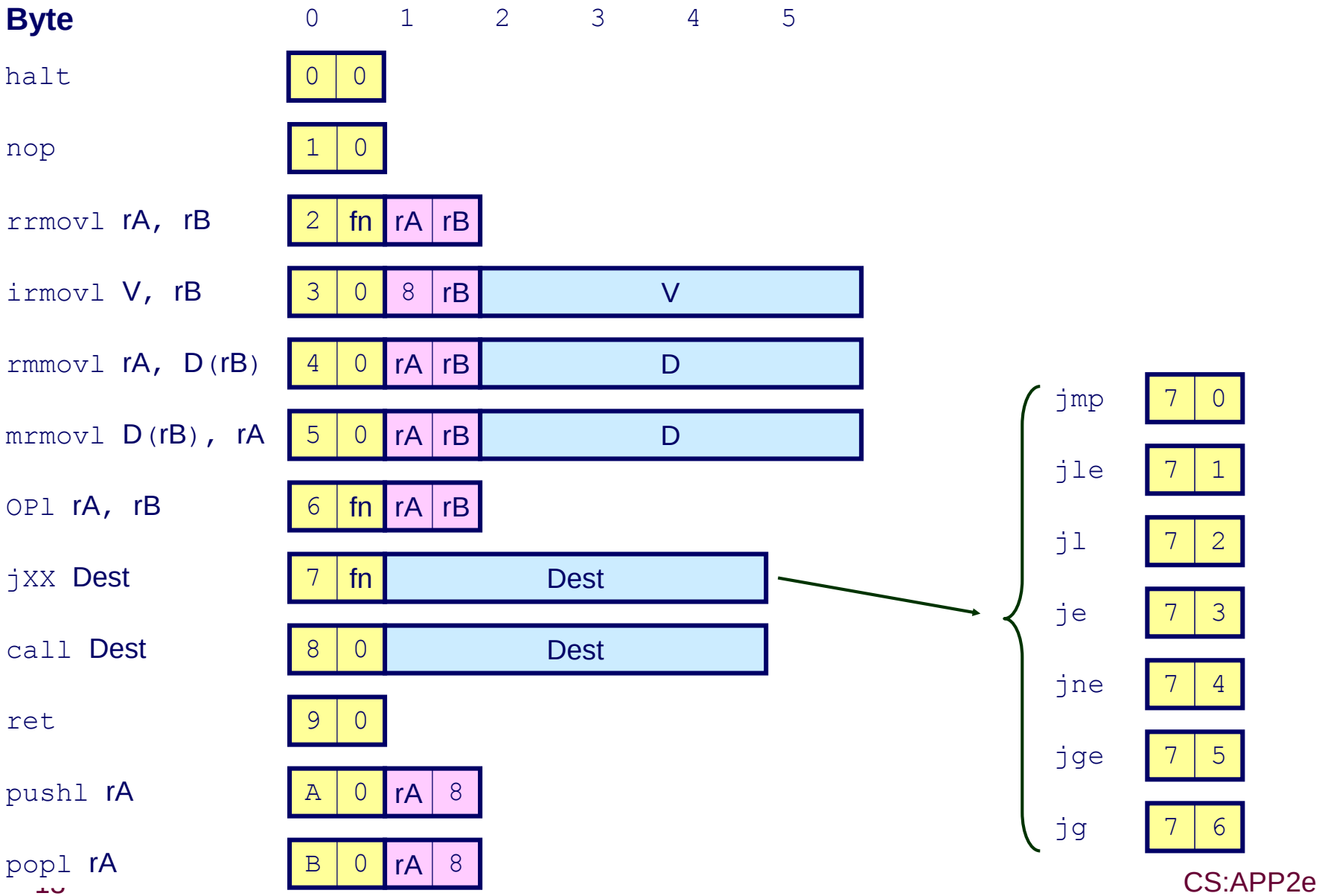
Y86 Instruction Set #2



Y86 Instruction Set #3



Y86 Instruction Set #4



Encoding Registers

Each register has 4-bit ID

%eax	0
%ecx	1
%edx	2
%ebx	3

%esi	6
%edi	7
%esp	4
%ebp	5

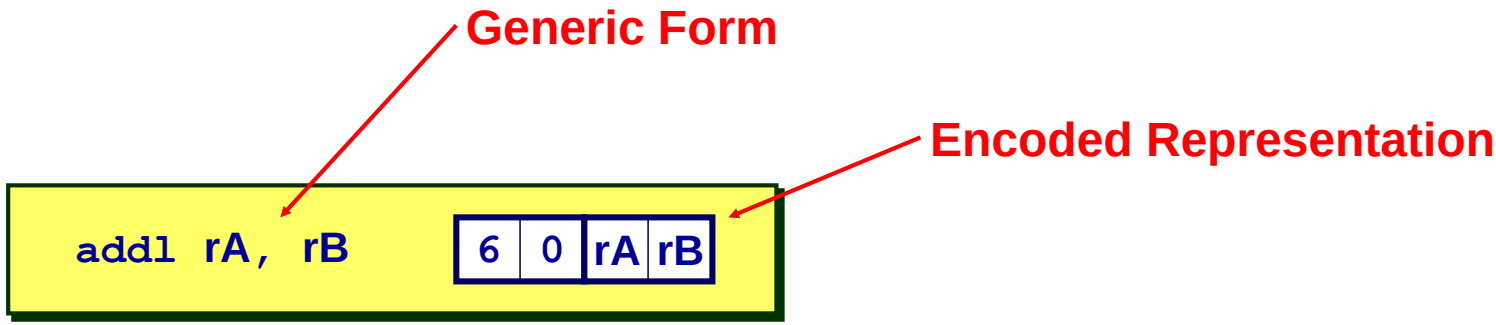
- Same encoding as in IA32

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

- Will use this in our hardware design in multiple places

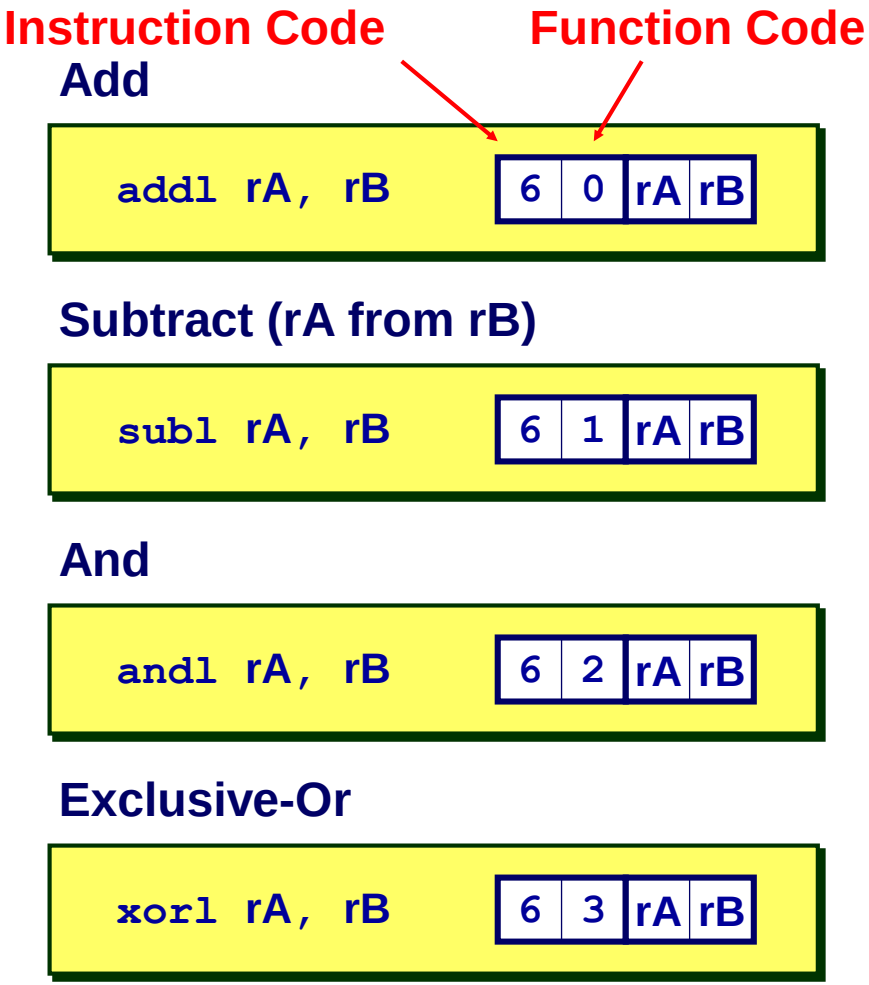
Instruction Example

Addition Instruction



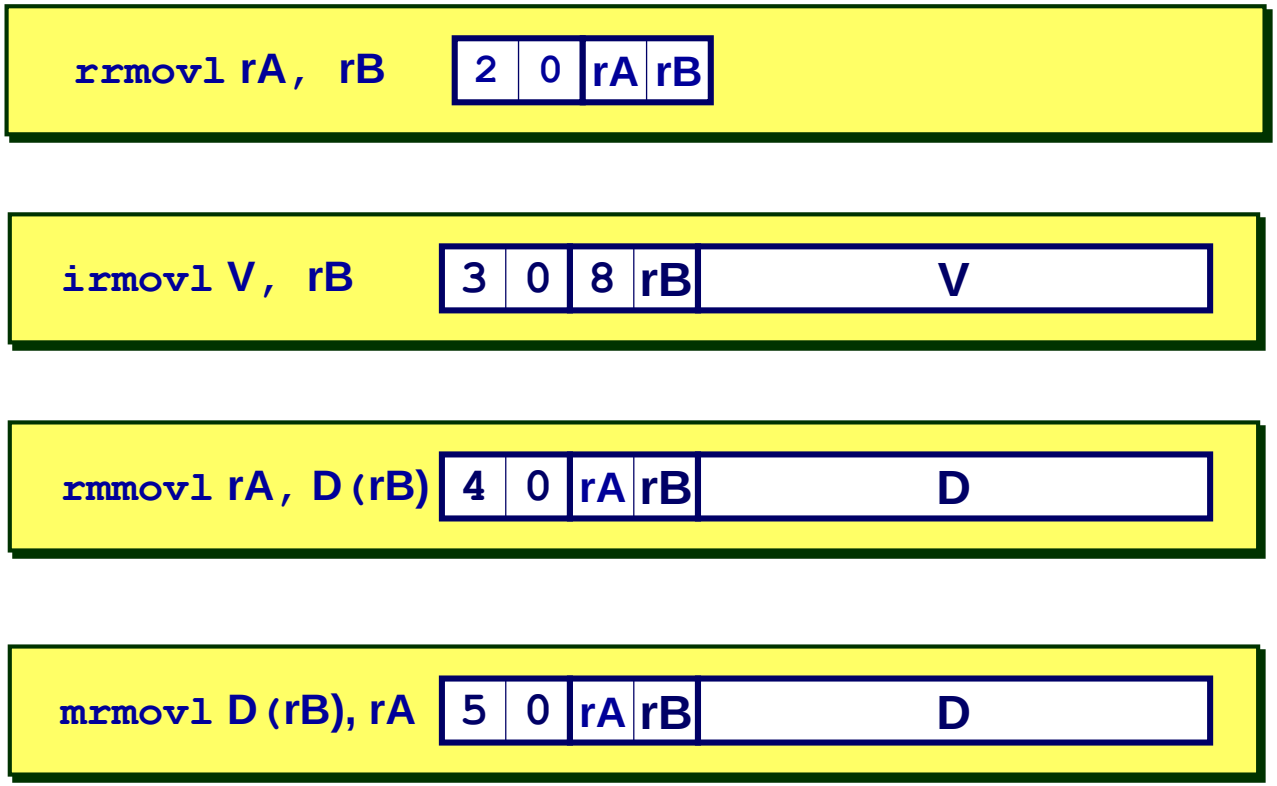
- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86 only allows addition to be applied to register data
- Set condition codes based on result
- e.g., `addl %eax,%esi` Encoding: `60 06`
- Two-byte encoding
  - First indicates instruction type
  - Second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

Arithmetic and Logical Operations



- Refer to generically as "`OPl`"
- Encodings differ only by "function code"
  - Low-order 4 bits in first instruction byte
- Set condition codes as side effect

Move Operations



- Like the IA32 `movl` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Move Instruction Examples

IA32	Y86	Encoding
`movl $0xabcd, %edx`	`irmovl $0xabcd, %edx`	`30 82 cd ab 00 00`
`movl %esp, %ebx`	`rrmovl %esp, %ebx`	`20 43`
`movl -12(%ebp),%ecx`	`mrmovl -12(%ebp),%ecx`	`50 15 f4 ff ff ff`
`movl %esi,0x41c(%esp)`	`rmmovl %esi,0x41c(%esp)`	`40 64 1c 04 00 00`

`movl $0xabcd, (%eax)`	—
`movl %eax, 12(%eax,%edx)`	—
`movl (%ebp,%eax,4),%ecx`	—

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 `rrmovl` instruction
  - (Conditionally) copy value from source to destination register

Jump Instructions

Jump Unconditionally

Jump When Less or Equal

Jump When Less

Jump When Equal

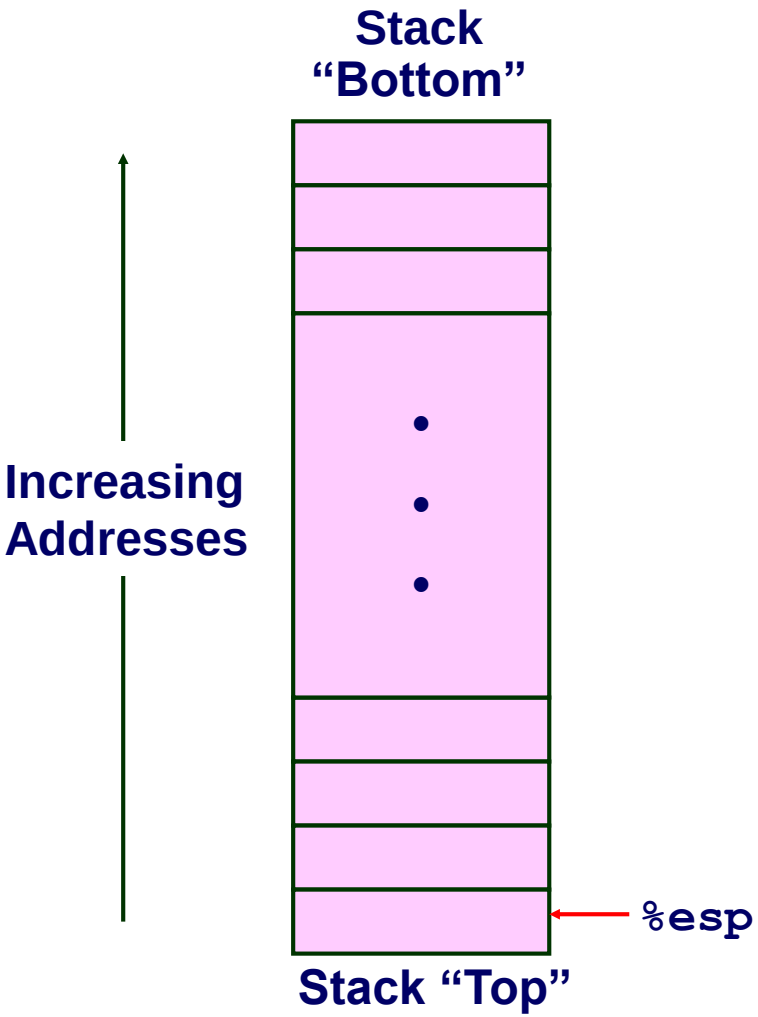
Jump When Not Equal

Jump When Greater or Equal

Jump When Greater

- Refer to generically as "`jxx`"
- Encodings differ only by "function code"
- Based on values of condition codes
- Same as IA32 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in IA32

Y86 Program Stack



- Region of memory holding program data
- Used in Y86 (and IA32) for supporting procedure calls
- Stack top indicated by `%esp`
  - 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

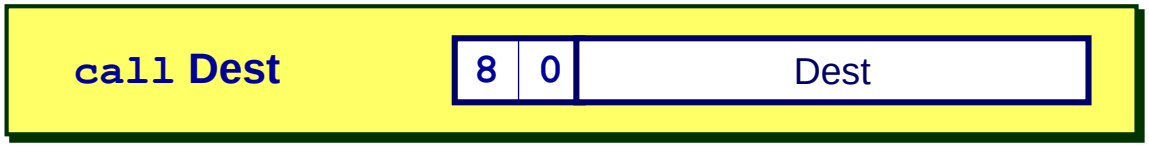
`pushl rA`

- Decrement `%esp` by 4
- Store word from rA to memory at `%esp`
- Like IA32

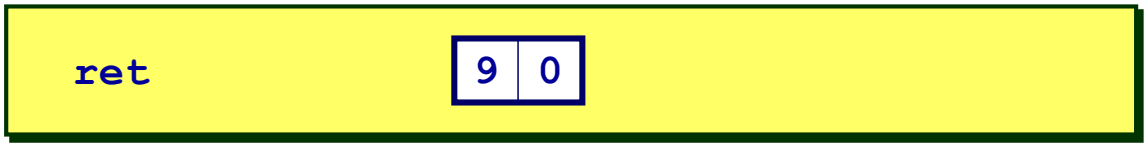
`popl rA`

- Read word from memory at `%esp`
- Save in rA
- Increment `%esp` by 4
- Like IA32

Subroutine Call and Return



- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like IA32



- Pop value from stack
- Use as address for next instruction
- Like IA32

Miscellaneous Instructions

- Don't do anything

- Stop executing instructions
- IA32 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
- Dominant style through mid-80's

Stack-oriented instruction set

- Use stack to pass arguments, save program counter
- Explicit push and pop instructions

Arithmetic instructions can access memory

- `addl %eax, 12(%ebx,%ecx,4)`
  - 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

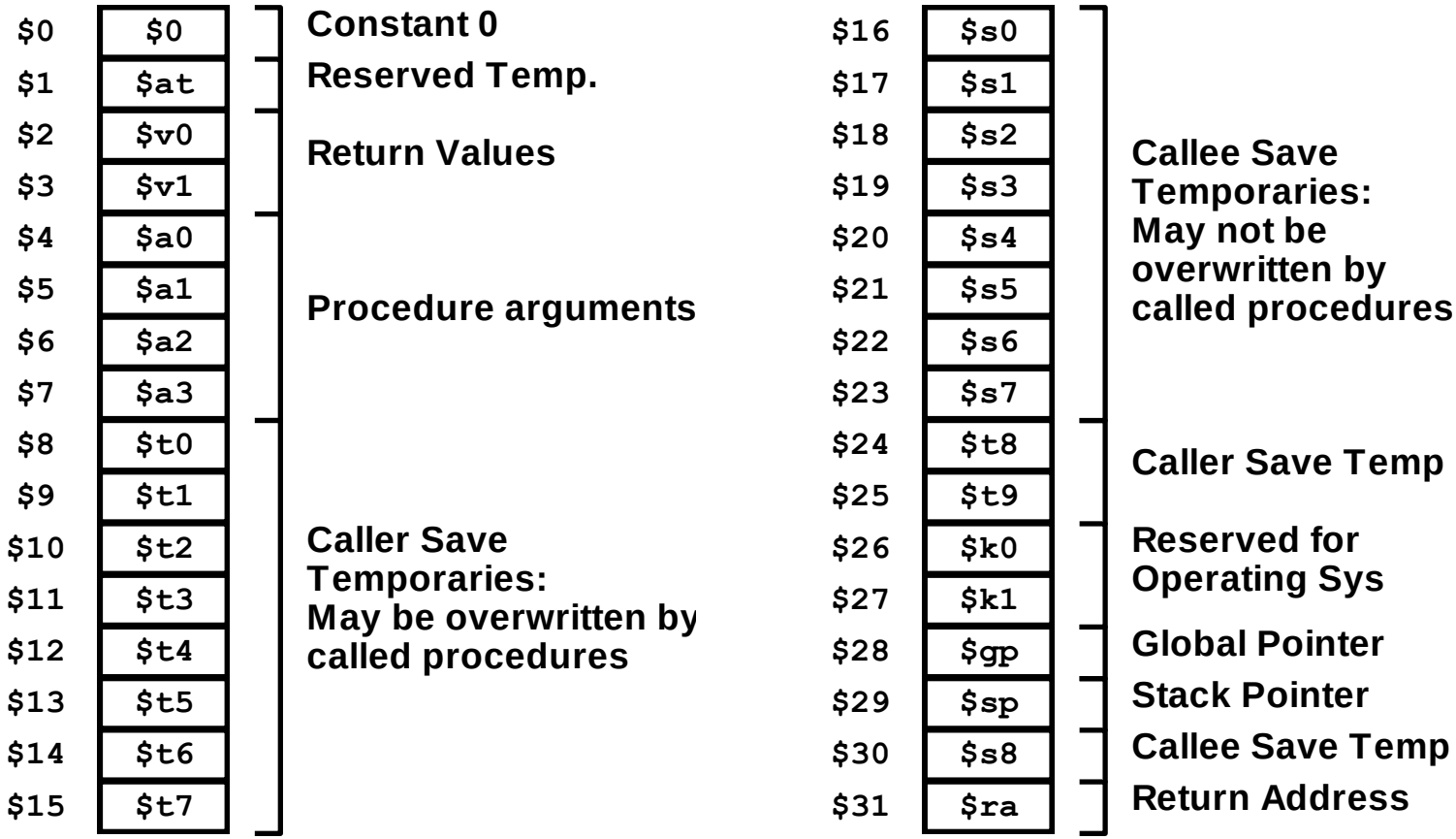
Only load and store instructions can access memory

- Similar to Y86 `mrmovl` and `rmmovl`

No Condition codes

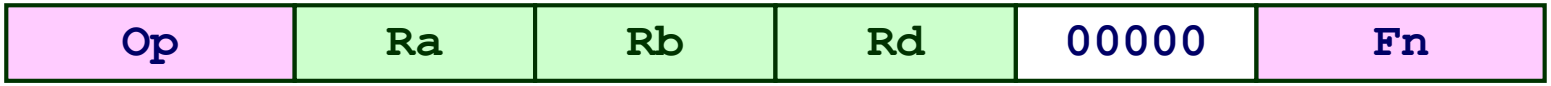
- Test instructions return 0/1 in register

MIPS Registers



MIPS Instruction Examples

R-R

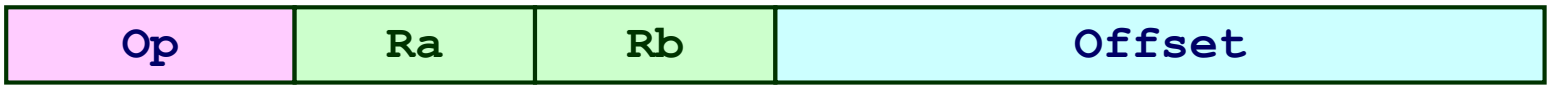



```
addu $3,$2,$1          # Register add: $3 = $2+$1
```

R-I

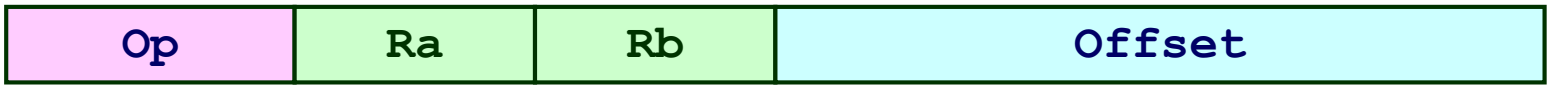

```
addu $3,$2, 3145       # Immediate add: $3 = $2+3145
sll $3,$2,2            # Shift left: $3 = $2 << 2
```

Branch




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

Load/Store




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

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
- For embedded processors, RISC makes sense
  - Smaller, cheaper, less power
  - Most cell phones use ARM processor

Summary

Y86 Instruction Set Architecture

- Similar state and instructions as IA32
- 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
- Intel has evolved from IA32 to x86-64
  - Uses 64-bit words (including addresses)
  - Adopted some features found in RISC
    - » More registers (16)
    - » Less reliance on stack