

# 198:211

## Computer Architecture

- Topics:
  - Processor Design

# Where are we now?

- C-Programming
  - A “real” computer language...
- Data Representation
  - Everything goes down to bits and bytes
- Machine representation Language
  - Very limited programming model
- Digital Logic
  - Transistors → Gates → Circuits
  - Circuits → { Memory, Registers, and Components }
- What are we going to do now....

# Processor Design

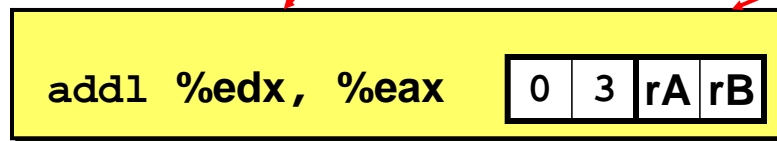
- Figure out how a small subset of the instruction set works in hardware
- Use Hardware blocks to describe data path
- Use control logic to ensure steps of the instruction flows smoothly

# Add instruction

- Type of add depends on where operands are located
  - add reg1, reg2
    - Add contents of reg1 to reg2 and store result in reg2
  - add \$constant, reg2
    - Add contents of memory immediately following the instruction to reg 2 and store result in reg2
  - add (effective-address), reg2
    - Add contents of effective address to reg2 and store result in reg2
    - Effective address could be relative; relative + offset; relative + offset, index; relative + offset, index\*scaled

# Instruction Example

- Addition Instruction **Generic Form**



- Add value in register %edx to that in register %eax
  - Store result in register %eax
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers

# ADD-Add : note src, dst are reversed in the intel manual

## ADD—Add

| Opcode          | Instruction                     | Description                                   |
|-----------------|---------------------------------|---|
| 04 <i>ib</i>    | ADD AL, <i>imm8</i>             | Add <i>imm8</i> to AL                         |
| 05 <i>iw</i>    | ADD AX, <i>imm16</i>            | Add <i>imm16</i> to AX                        |
| 05 <i>id</i>    | ADD EAX, <i>imm32</i>           | Add <i>imm32</i> to EAX                       |
| 80 /0 <i>ib</i> | ADD <i>r/m8</i> , <i>imm8</i>   | Add <i>imm8</i> to <i>r/m8</i>                |
| 81 /0 <i>iw</i> | ADD <i>r/m16</i> , <i>imm16</i> | Add <i>imm16</i> to <i>r/m16</i>              |
| 81 /0 <i>id</i> | ADD <i>r/m32</i> , <i>imm32</i> | Add <i>imm32</i> to <i>r/m32</i>              |
| 83 /0 <i>ib</i> | ADD <i>r/m16</i> , <i>imm8</i>  | Add sign-extended <i>imm8</i> to <i>r/m16</i> |
| 83 /0 <i>ib</i> | ADD <i>r/m32</i> , <i>imm8</i>  | Add sign-extended <i>imm8</i> to <i>r/m32</i> |
| 00 <i>lr</i>    | ADD <i>r/m8</i> , <i>r8</i>     | Add <i>r8</i> to <i>r/m8</i>                  |
| 01 <i>lr</i>    | ADD <i>r/m16</i> , <i>r16</i>   | Add <i>r16</i> to <i>r/m16</i>                |
| 01 <i>lr</i>    | ADD <i>r/m32</i> , <i>r32</i>   | Add <i>r32</i> to <i>r/m32</i>                |
| 02 <i>lr</i>    | ADD <i>r8</i> , <i>r/m8</i>     | Add <i>r/m8</i> to <i>r8</i>                  |
| 03 <i>lr</i>    | ADD <i>r16</i> , <i>r/m16</i>   | Add <i>r/m16</i> to <i>r16</i>                |
| 03 <i>lr</i>    | ADD <i>r32</i> , <i>r/m32</i>   | Add <i>r/m32</i> to <i>r32</i>                |

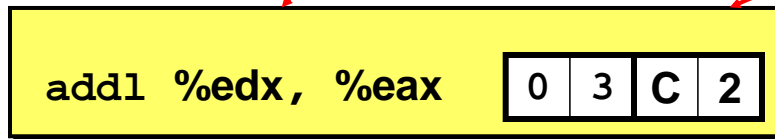
In assembly language add src, dst means src + dst → dst

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

| r8(/r)<br>r16(/r)<br>r32(/r)<br>mm(/r)<br>xmm(/r)<br>/digit (Opcode)<br>REG =  |     |  | AL<br>EAX<br>MM0<br>XMM0<br>0<br>000         | CL<br>ECX<br>MM1<br>XMM1<br>1<br>001         | DL<br>EDX<br>MM2<br>XMM2<br>2<br>010         | BL<br>EBX<br>MM3<br>XMM3<br>3<br>011         | AH<br>SP<br>ESP<br>MM4<br>XMM4<br>4<br>100   | CH<br>BP<br>EBP<br>MM5<br>XMM5<br>5<br>101   | DH<br>SI<br>ESI<br>MM6<br>XMM6<br>6<br>110   | BH<br>DI<br>EDI<br>MM7<br>XMM7<br>7<br>111   |
|--|-----|--|--|--|--|--|--|--|--|--|
| Effective Address  | Mod | R/M  | Value of ModR/M Byte (in Hexadecimal)        |  |  |  |  |  |  |  |
| [EAX]<br>[ECX]<br>[EDX]<br>[EBX]<br>[--][--] <sup>1</sup><br>disp32 <sup>2</sup><br>[ESI]<br>[EDI]   | 00  | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 00<br>01<br>02<br>03<br>04<br>05<br>06<br>07 | 08<br>09<br>0A<br>0B<br>0C<br>0D<br>0E<br>0F | 10<br>11<br>12<br>13<br>14<br>15<br>16<br>17 | 18<br>19<br>1A<br>1B<br>1C<br>1D<br>1E<br>1F | 20<br>21<br>22<br>23<br>24<br>25<br>26<br>27 | 28<br>29<br>2A<br>2B<br>2C<br>2D<br>2E<br>2F | 30<br>31<br>32<br>33<br>34<br>35<br>36<br>37 | 38<br>39<br>3A<br>3B<br>3C<br>3D<br>3E<br>3F |
| disp8[EAX] <sup>3</sup><br>disp8[ECX]<br>disp8[EDX]<br>disp8[EBX];<br>disp8[--][--]<br>disp8[EBP]<br>disp8[ESI]<br>disp8[EDI]  | 01  | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 40<br>41<br>42<br>43<br>44<br>45<br>46<br>47 | 48<br>49<br>4A<br>4B<br>4C<br>4D<br>4E<br>4F | 50<br>51<br>52<br>53<br>54<br>55<br>56<br>57 | 58<br>59<br>5A<br>5B<br>5C<br>5D<br>5E<br>5F | 60<br>61<br>62<br>63<br>64<br>65<br>66<br>67 | 68<br>69<br>6A<br>6B<br>6C<br>6D<br>6E<br>6F | 70<br>71<br>72<br>73<br>74<br>75<br>76<br>77 | 78<br>79<br>7A<br>7B<br>7C<br>7D<br>7E<br>7F |
| disp32[EAX]<br>disp32[ECX]<br>disp32[EDX]<br>disp32[EBX]<br>disp32[--][--]<br>disp32[EBP]<br>disp32[ESI]<br>disp32[EDI]  | 10  | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 80<br>81<br>82<br>83<br>84<br>85<br>86<br>87 | 88<br>89<br>8A<br>8B<br>8C<br>8D<br>8E<br>8F | 90<br>91<br>92<br>93<br>94<br>95<br>96<br>97 | 98<br>99<br>9A<br>9B<br>9C<br>9D<br>9E<br>9F | A0<br>A1<br>A2<br>A3<br>A4<br>A5<br>A6<br>A7 | A8<br>A9<br>AA<br>AB<br>AC<br>AD<br>AE<br>AF | B0<br>B1<br>B2<br>B3<br>B4<br>B5<br>B6<br>B7 | B8<br>B9<br>BA<br>BB<br>BC<br>BD<br>BE<br>BF |
| EAX/AX/AL/MM0/XMM0<br>ECX/CX/CL/MM1/XMM1<br>EDX/DX/DL/MM2/XMM2<br>EBX/BX/BL/MM3/XMM3<br>ESP/SP/AH/MM4/XMM4<br>EBP/BP/CH/MM5/XMM5<br>ESI/SI/DH/MM6/XMM6<br>EDI/DI/BH/MM7/XMM7 | 11  | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | C0<br>C1<br>C2<br>C3<br>C4<br>C5<br>C6<br>C7 | C8<br>C9<br>CA<br>CB<br>CC<br>CD<br>CE<br>CF | D0<br>D1<br>D2<br>D3<br>D4<br>D5<br>D6<br>D7 | D8<br>D9<br>DA<br>DB<br>DC<br>DD<br>DE<br>DF | E0<br>E1<br>E2<br>E3<br>E4<br>E5<br>E6<br>E7 | E8<br>E9<br>EA<br>EB<br>EC<br>ED<br>EE<br>EF | F0<br>F1<br>F2<br>F3<br>F4<br>F5<br>F6<br>F7 | F8<br>F9<br>FA<br>FB<br>FC<br>FD<br>FE<br>FF |

# Instruction Example in x86

- Addition Instruction **Generic Form**



**Encoded Representation**

- Opcode is 03 and src, dst are %edx, %eax
  - from previous table (Table 2.2) code is C2

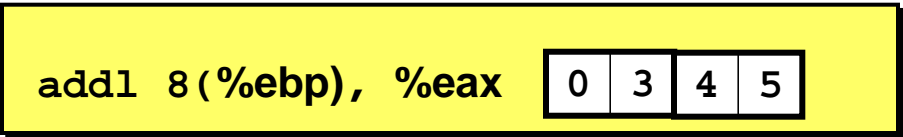
- So, 03 C2 means add %edx, %

- 03 45

- means add 8(%ebp), %eax

- 05

- add \$cons, %eax

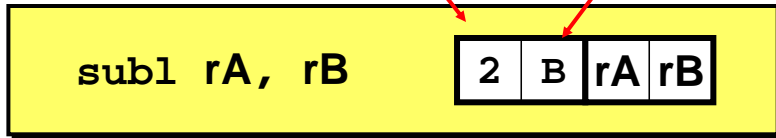


# Arithmetic and Logical Operations

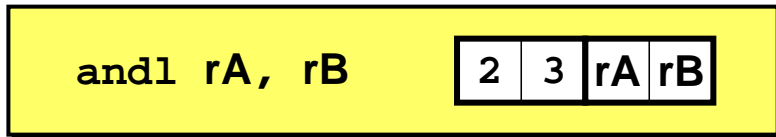
## Instruction Code

## Function Code

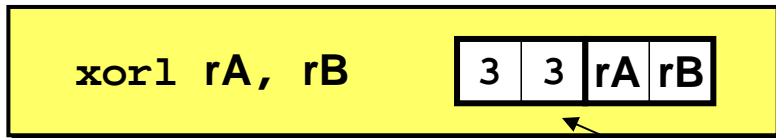
Subtract (rA from rB)



And



Exclusive-Or



- The second byte will vary based on type of operands
- Set condition codes as side effect

Lower 4 bits can vary depending upon the type Of operand  
20 means 8 byte register transfer,

Lower 4 bits can vary  
30 means 8 byte register transfer,

# Subtract operation- Instruction Set Reference

## SUB—Subtract

| Opcode          | Instruction                     | Description  |
|-----------------|---------------------------------|--|
| 2C <i>ib</i>    | SUB AL, <i>imm8</i>             | Subtract <i>imm8</i> from AL                         |
| 2D <i>iw</i>    | SUB AX, <i>imm16</i>            | Subtract <i>imm16</i> from AX                        |
| 2D <i>id</i>    | SUB EAX, <i>imm32</i>           | Subtract <i>imm32</i> from EAX                       |
| 80 <i>ib ib</i> | SUB <i>r/m8</i> , <i>imm8</i>   | Subtract <i>imm8</i> from <i>r/m8</i>                |
| 81 <i>ib iw</i> | SUB <i>r/m16</i> , <i>imm16</i> | Subtract <i>imm16</i> from <i>r/m16</i>              |
| 81 <i>ib id</i> | SUB <i>r/m32</i> , <i>imm32</i> | Subtract <i>imm32</i> from <i>r/m32</i>              |
| 83 <i>ib ib</i> | SUB <i>r/m16</i> , <i>imm8</i>  | Subtract sign-extended <i>imm8</i> from <i>r/m16</i> |
| 83 <i>ib id</i> | SUB <i>r/m32</i> , <i>imm8</i>  | Subtract sign-extended <i>imm8</i> from <i>r/m32</i> |
| 28 <i>rb</i>    | SUB <i>r/m8</i> , <i>r8</i>     | Subtract <i>r8</i> from <i>r/m8</i>                  |
| 29 <i>rb</i>    | SUB <i>r/m16</i> , <i>r16</i>   | Subtract <i>r16</i> from <i>r/m16</i>                |
| 29 <i>rb</i>    | SUB <i>r/m32</i> , <i>r32</i>   | Subtract <i>r32</i> from <i>r/m32</i>                |
| 2A <i>rb</i>    | SUB <i>r8</i> , <i>r/m8</i>     | Subtract <i>r/m8</i> from <i>r8</i>                  |
| 2B <i>rb</i>    | SUB <i>r16</i> , <i>r/m16</i>   | Subtract <i>r/m16</i> from <i>r16</i>                |
| 2B <i>rb</i>    | SUB <i>r32</i> , <i>r/m32</i>   | Subtract <i>r/m32</i> from <i>r32</i>                |

## AND—Logical AND

| Opcode          | Instruction                     | Description                                  |
|-----------------|---------------------------------|--|
| 24 <i>ib</i>    | AND AL, <i>imm8</i>             | AL AND <i>imm8</i>                           |
| 25 <i>iw</i>    | AND AX, <i>imm16</i>            | AX AND <i>imm16</i>                          |
| 25 <i>id</i>    | AND EAX, <i>imm32</i>           | EAX AND <i>imm32</i>                         |
| 80 /4 <i>ib</i> | AND <i>r/m8</i> , <i>imm8</i>   | <i>r/m8</i> AND <i>imm8</i>                  |
| 81 /4 <i>iw</i> | AND <i>r/m16</i> , <i>imm16</i> | <i>r/m16</i> AND <i>imm16</i>                |
| 81 /4 <i>id</i> | AND <i>r/m32</i> , <i>imm32</i> | <i>r/m32</i> AND <i>imm32</i>                |
| 83 /4 <i>ib</i> | AND <i>r/m16</i> , <i>imm8</i>  | <i>r/m16</i> AND <i>imm8</i> (sign-extended) |
| 83 /4 <i>ib</i> | AND <i>r/m32</i> , <i>imm8</i>  | <i>r/m32</i> AND <i>imm8</i> (sign-extended) |
| 20 /r           | AND <i>r/m8</i> , <i>r8</i>     | <i>r/m8</i> AND <i>r8</i>                    |
| 21 /r           | AND <i>r/m16</i> , <i>r16</i>   | <i>r/m16</i> AND <i>r16</i>                  |
| 21 /r           | AND <i>r/m32</i> , <i>r32</i>   | <i>r/m32</i> AND <i>r32</i>                  |
| 22 /r           | AND <i>r8</i> , <i>r/m8</i>     | <i>r8</i> AND <i>r/m8</i>                    |
| 23 /r           | AND <i>r16</i> , <i>r/m16</i>   | <i>r16</i> AND <i>r/m16</i>                  |
| 23 /r           | AND <i>r32</i> , <i>r/m32</i>   | <i>r32</i> AND <i>r/m32</i>                  |

### Description

This instruction performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. Two memory operands cannot, however, be used in one instruction. Each bit of the instruction result is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0.

### Operation

DEST ← DEST AND SRC;

## XOR—Logical Exclusive OR

| Opcode          | Instruction                     | Description   |
|-----------------|---------------------------------|---|
| 34 <i>ib</i>    | XOR AL, <i>imm8</i>             | AL XOR <i>imm8</i>                                    |
| 35 <i>iw</i>    | XOR AX, <i>imm16</i>            | AX XOR <i>imm16</i>                                   |
| 35 <i>id</i>    | XOR EAX, <i>imm32</i>           | EAX XOR <i>imm32</i>                                  |
| 80 /6 <i>ib</i> | XOR <i>r/m8</i> , <i>imm8</i>   | <i>r/m8</i> XOR <i>imm8</i>                           |
| 81 /6 <i>iw</i> | XOR <i>r/m16</i> , <i>imm16</i> | <i>r/m16</i> XOR <i>imm16</i>                         |
| 81 /6 <i>id</i> | XOR <i>r/m32</i> , <i>imm32</i> | <i>r/m32</i> XOR <i>imm32</i>                         |
| 83 /6 <i>ib</i> | XOR <i>r/m16</i> , <i>imm8</i>  | <i>r/m16</i> XOR <i>imm8</i> ( <i>sign-extended</i> ) |
| 83 /6 <i>ib</i> | XOR <i>r/m32</i> , <i>imm8</i>  | <i>r/m32</i> XOR <i>imm8</i> ( <i>sign-extended</i> ) |
| 30 <i>lr</i>    | XOR <i>r/m8</i> , <i>r8</i>     | <i>r/m8</i> XOR <i>r8</i>                             |
| 31 <i>lr</i>    | XOR <i>r/m16</i> , <i>r16</i>   | <i>r/m16</i> XOR <i>r16</i>                           |
| 31 <i>lr</i>    | XOR <i>r/m32</i> , <i>r32</i>   | <i>r/m32</i> XOR <i>r32</i>                           |
| 32 <i>lr</i>    | XOR <i>r8</i> , <i>r/m8</i>     | <i>r8</i> XOR <i>r/m8</i>                             |
| 33 <i>lr</i>    | XOR <i>r16</i> , <i>r/m16</i>   | <i>r8</i> XOR <i>r/m8</i>                             |
| 33 <i>lr</i>    | XOR <i>r32</i> , <i>r/m32</i>   | <i>r8</i> XOR <i>r/m8</i>                             |

### Description

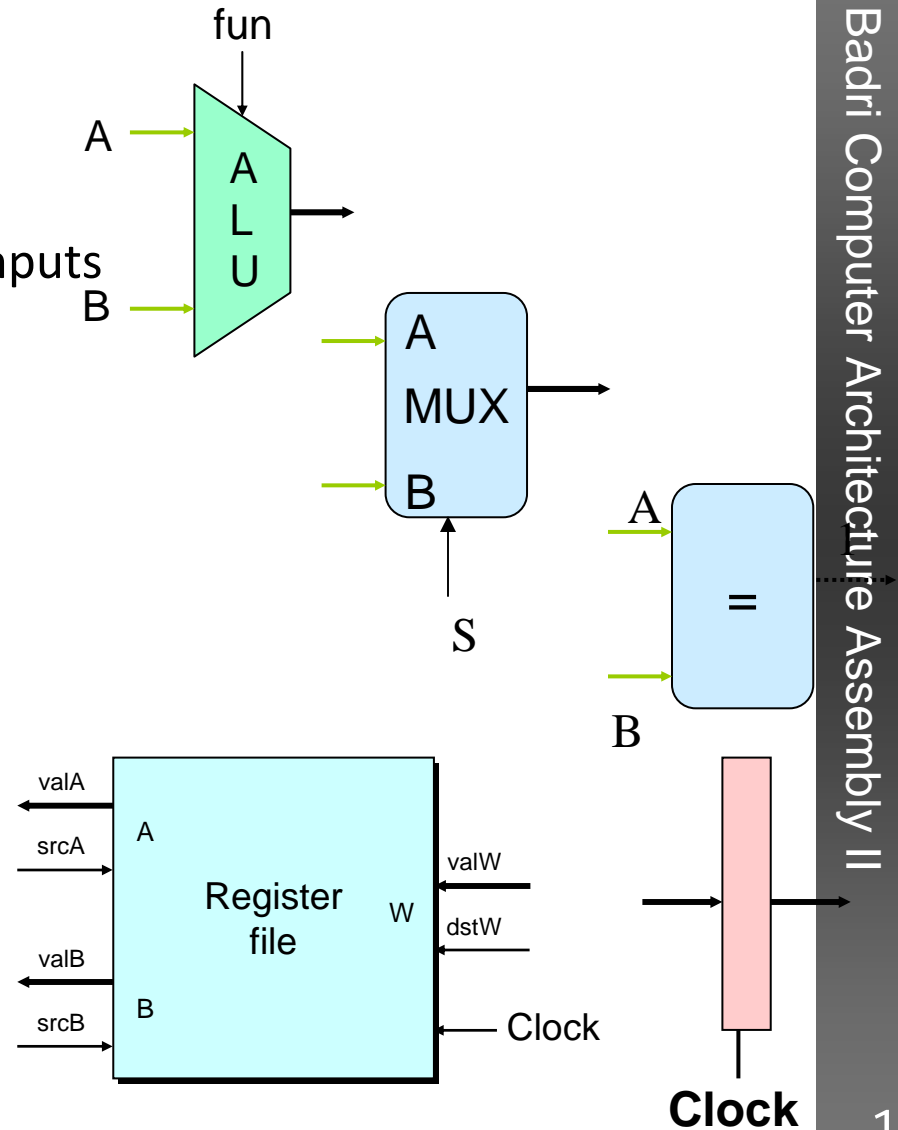
This instruction performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

# Single cycle stages

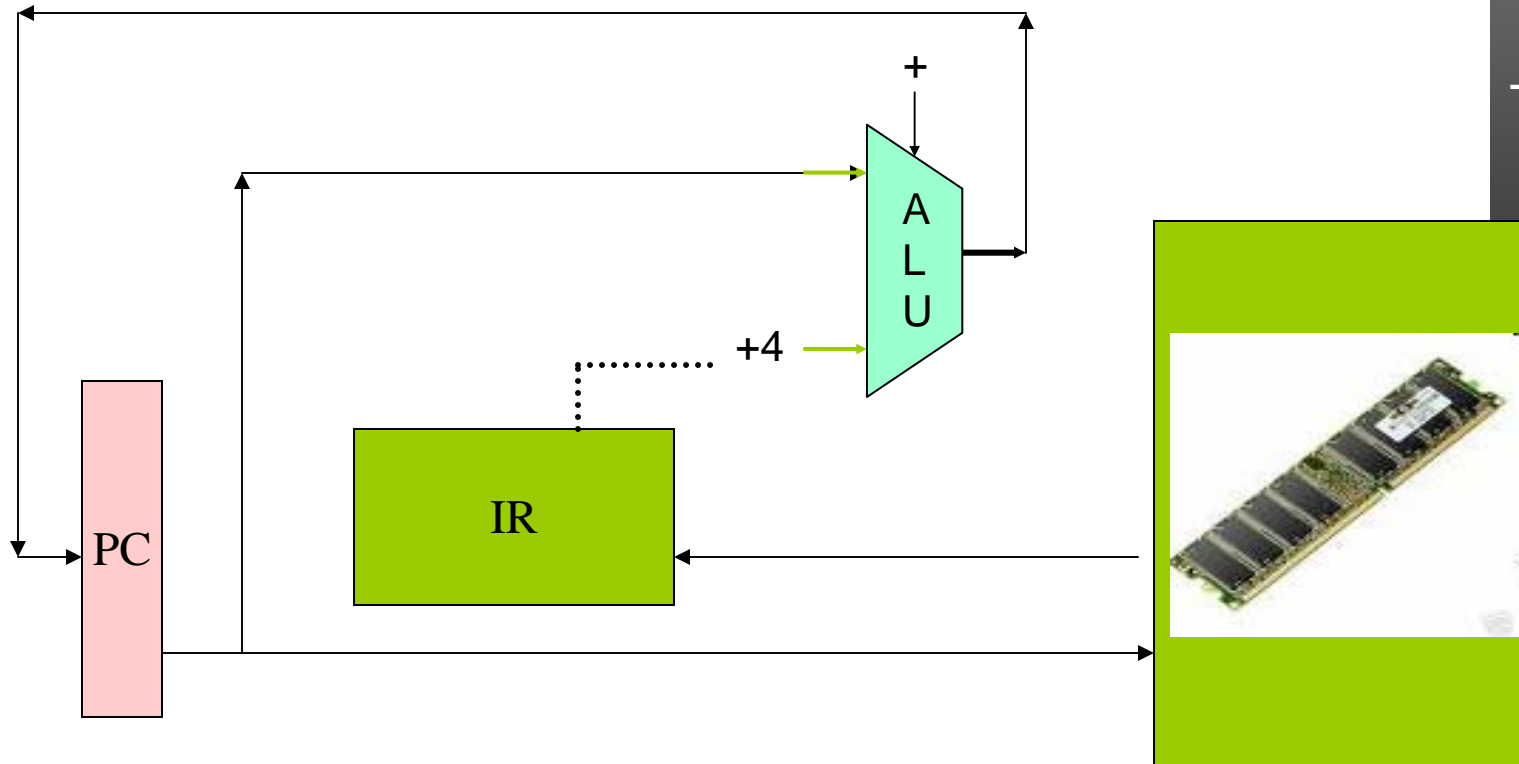
- Fetch Instruction (IF)
- Decode Instruction (ID)
- Execute Instruction (EX)
- Memory access (MEM)
- Write back (WB)
- Update Program Counter (PC)

# Building Blocks

- **Combinational Logic**
  - Compute Boolean functions of inputs
  - Continuously respond to input changes
  - Operate on data and implement control
  
- **Storage Elements**
  - Store bits
  - Addressable memories
  - Non-addressable registers
  - Loaded only as clock rises

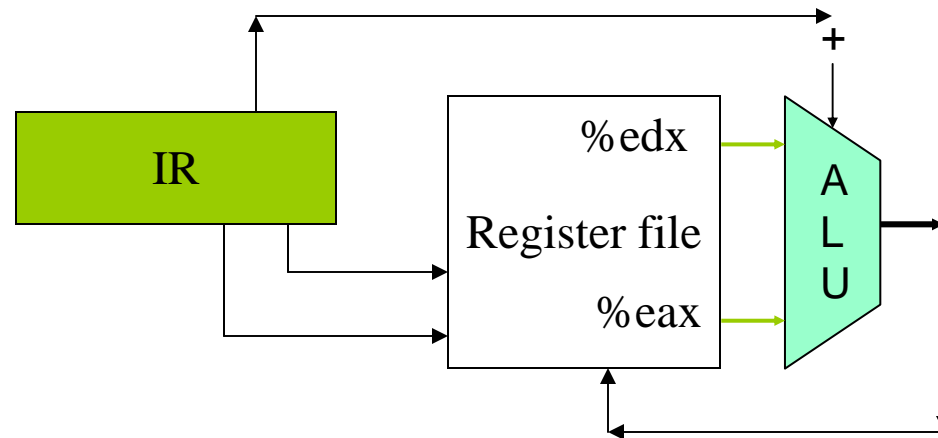


# Fetch Instruction



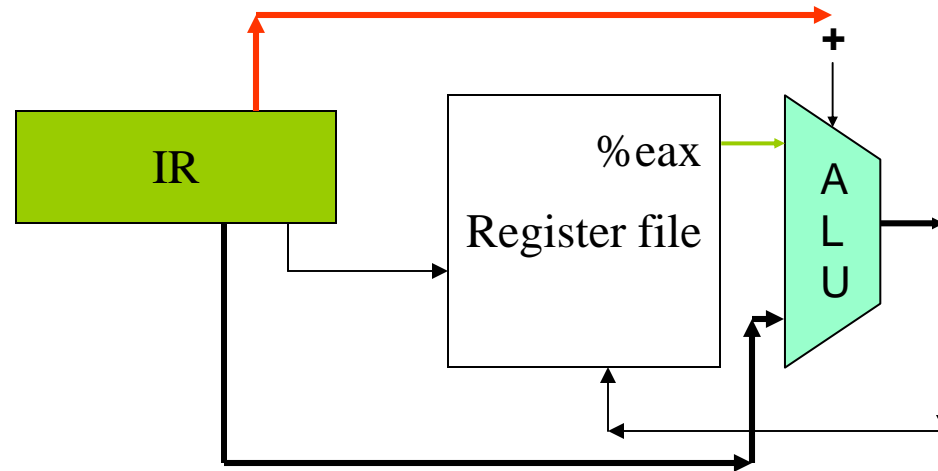
Fetch Instruction and increment Program counter

# Add instruction (add %edx, %eax)



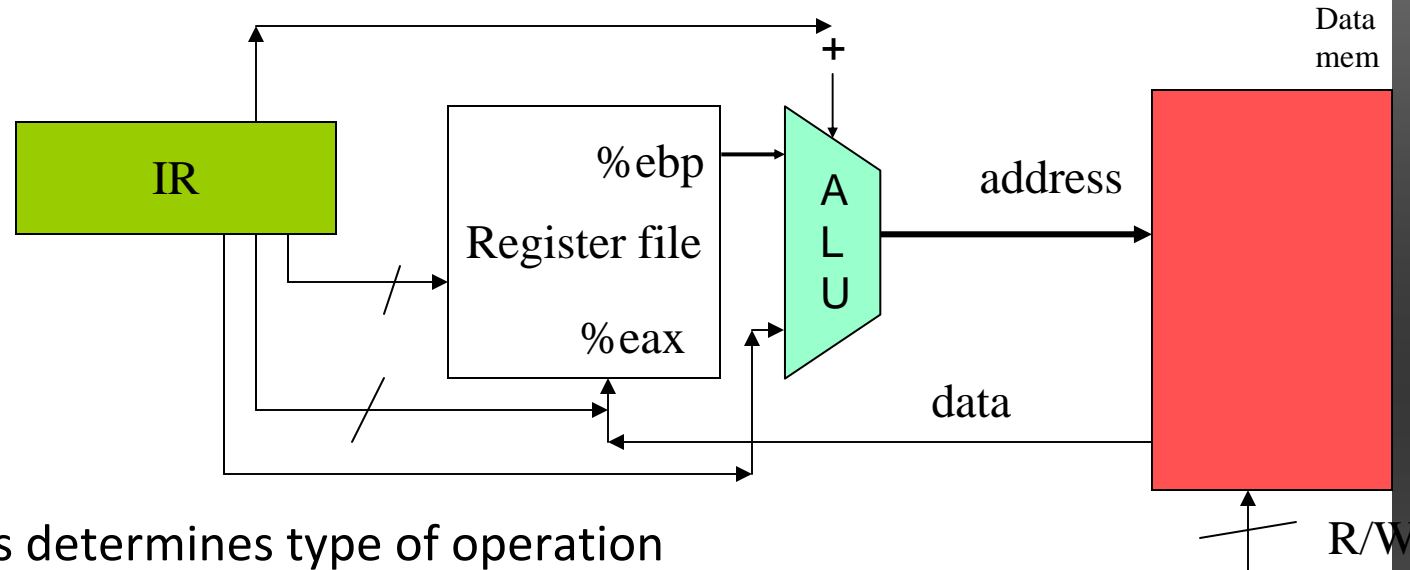
- Op code determines type of operation
  - 03 C2
- Operands are in registers
- They are fed to the adder (ALU)
- Result stored back in register
- Based On Instruction the signal to ALU can be +, -, \* or /

# Add instruction (add \$4, %eax)



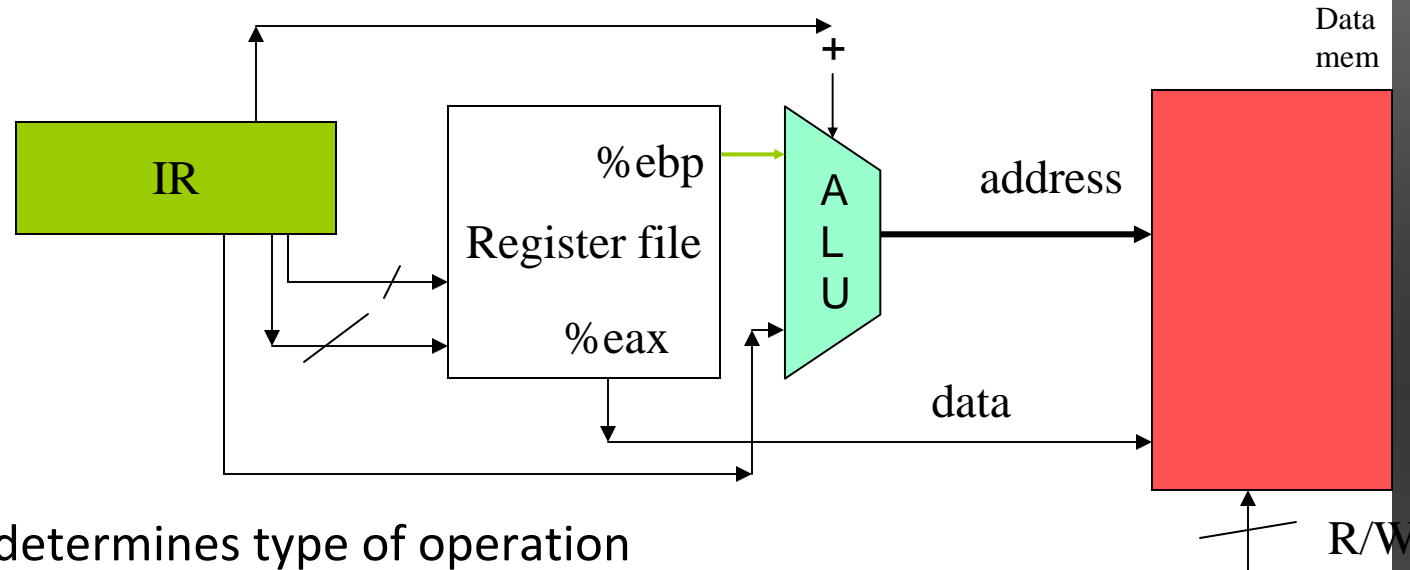
- Op code is determines type of operation
- 03 05
- One Operand is a register, the other is part of the instruction
- %eax, immediate value
- They are fed to the adder (ALU)
- Result stored back in register

# mov instruction (mov 8(%ebp), %eax)



- Op code is determines type of operation
- 8b 45
- Load word from memory onto register %eax
- Address is  $8 + (\%ebp)$

# mov instruction (mov %eax,8(%ebp))



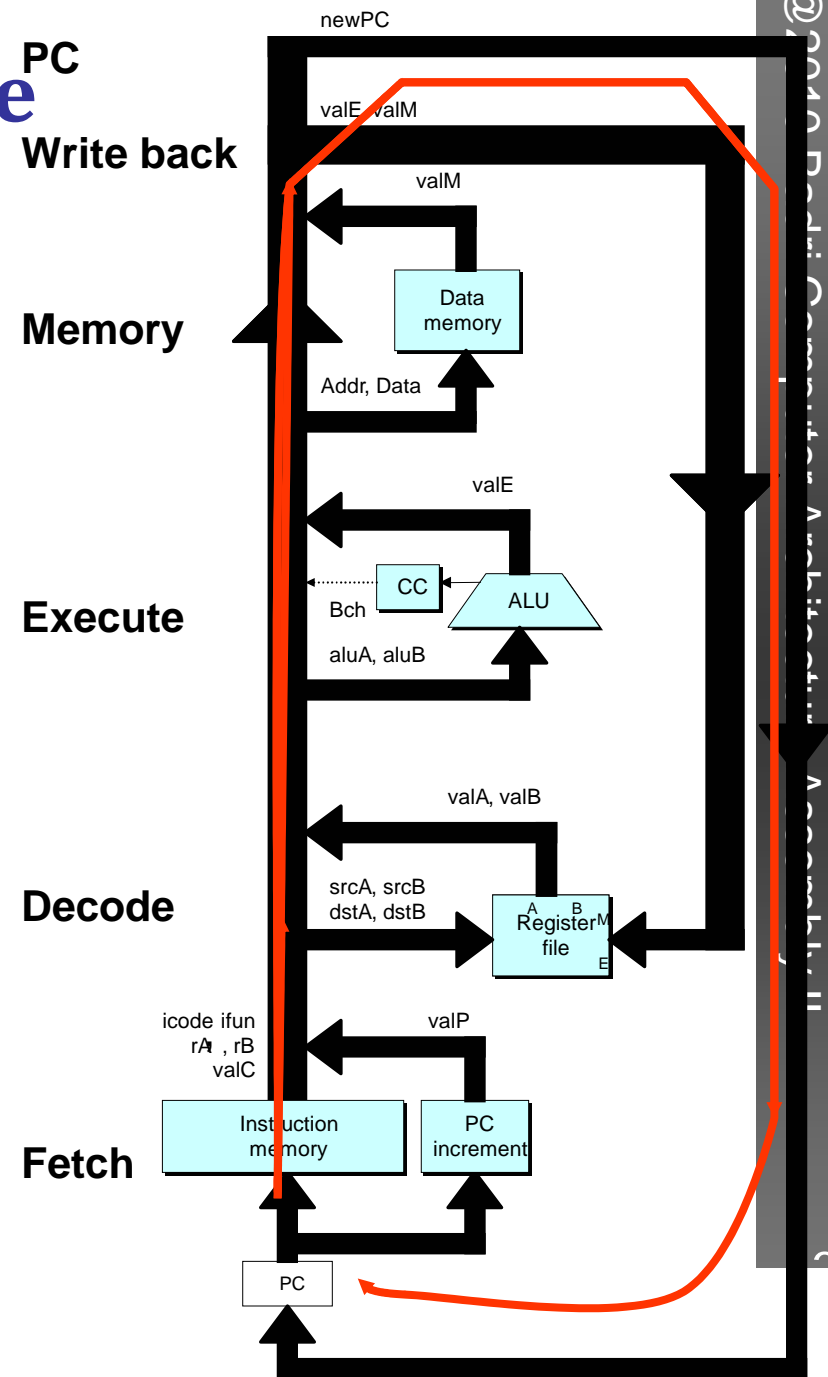
- Op code determines type of operation
- 89 45
- Store word from register `%eax` onto memory
- Address is  $8 + (\%ebp)$

## MOV—Move

| Opcode        | Instruction              | Description                                     |
|---------------|--------------------------|---|
| 88 <i>lr</i>  | MOV <i>r/m8,r8</i>       | Move <i>r8</i> to <i>r/m8</i>                   |
| 89 <i>lr</i>  | MOV <i>r/m16,r16</i>     | Move <i>r16</i> to <i>r/m16</i>                 |
| 89 <i>lr</i>  | MOV <i>r/m32,r32</i>     | Move <i>r32</i> to <i>r/m32</i>                 |
| 8A <i>lr</i>  | MOV <i>r8,r/m8</i>       | Move <i>r/m8</i> to <i>r8</i>                   |
| 8B <i>lr</i>  | MOV <i>r16,r/m16</i>     | Move <i>r/m16</i> to <i>r16</i>                 |
| 8B <i>lr</i>  | MOV <i>r32,r/m32</i>     | Move <i>r/m32</i> to <i>r32</i>                 |
| 8C <i>lr</i>  | MOV <i>r/m16,Sreg**</i>  | Move segment register to <i>r/m16</i>           |
| 8E <i>lr</i>  | MOV <i>Sreg,r/m16**</i>  | Move <i>r/m16</i> to segment register           |
| A0            | MOV AL, <i>moffs8*</i>   | Move byte at ( <i>seg:offset</i> ) to AL        |
| A1            | MOV AX, <i>moffs16*</i>  | Move word at ( <i>seg:offset</i> ) to AX        |
| A1            | MOV EAX, <i>moffs32*</i> | Move doubleword at ( <i>seg:offset</i> ) to EAX |
| A2            | MOV <i>moffs8*</i> ,AL   | Move AL to ( <i>seg:offset</i> )                |
| A3            | MOV <i>moffs16*</i> ,AX  | Move AX to ( <i>seg:offset</i> )                |
| A3            | MOV <i>moffs32*</i> ,EAX | Move EAX to ( <i>seg:offset</i> )               |
| B0+ <i>rb</i> | MOV <i>r8,imm8</i>       | Move <i>imm8</i> to <i>r8</i>                   |
| B8+ <i>rw</i> | MOV <i>r16,imm16</i>     | Move <i>imm16</i> to <i>r16</i>                 |
| B8+ <i>rd</i> | MOV <i>r32,imm32</i>     | Move <i>imm32</i> to <i>r32</i>                 |
| C6 <i>l0</i>  | MOV <i>r/m8,imm8</i>     | Move <i>imm8</i> to <i>r/m8</i>                 |
| C7 <i>l0</i>  | MOV <i>r/m16,imm16</i>   | Move <i>imm16</i> to <i>r/m16</i>               |
| C7 <i>l0</i>  | MOV <i>r/m32,imm32</i>   | Move <i>imm32</i> to <i>r/m32</i>               |

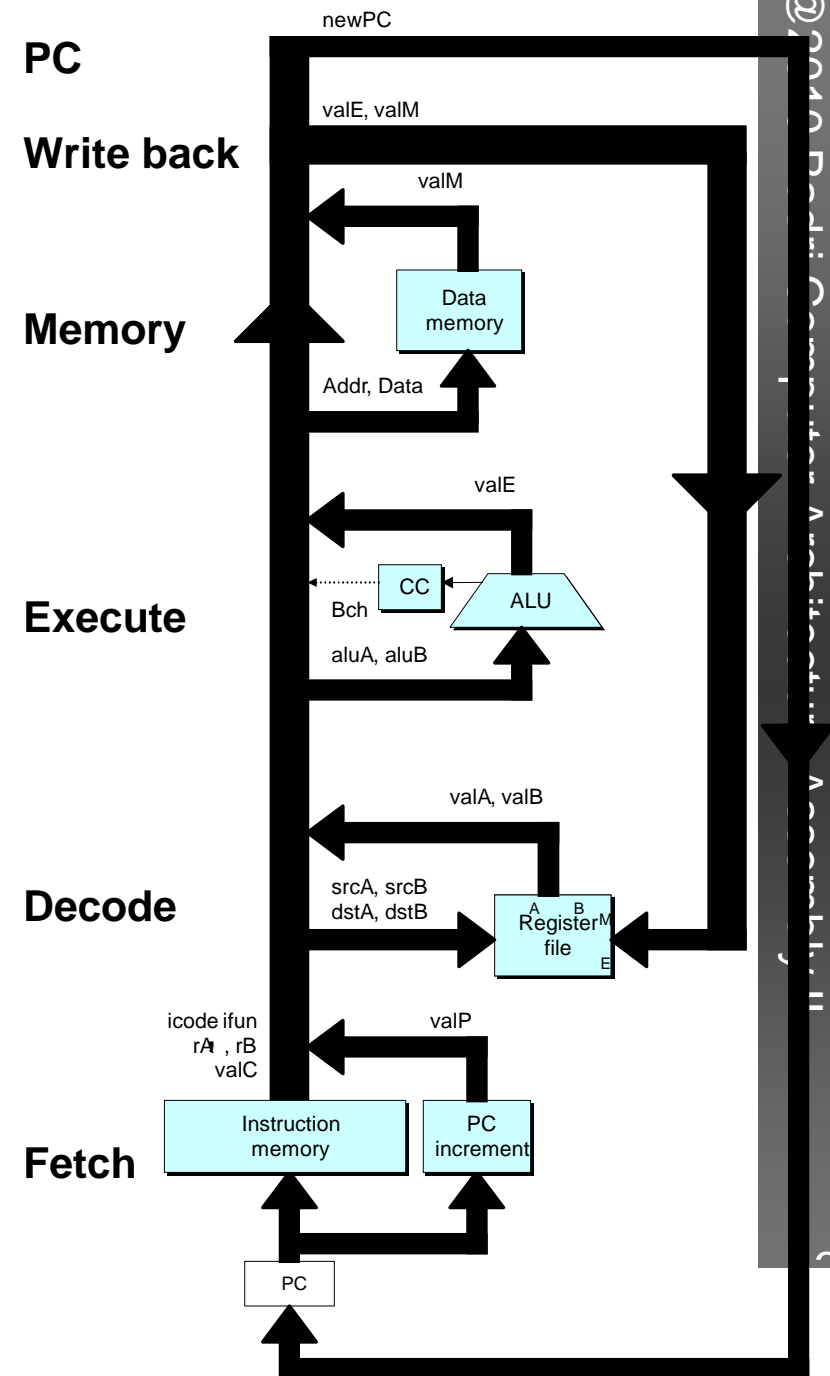
# SEQ Hardware Structure

- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter

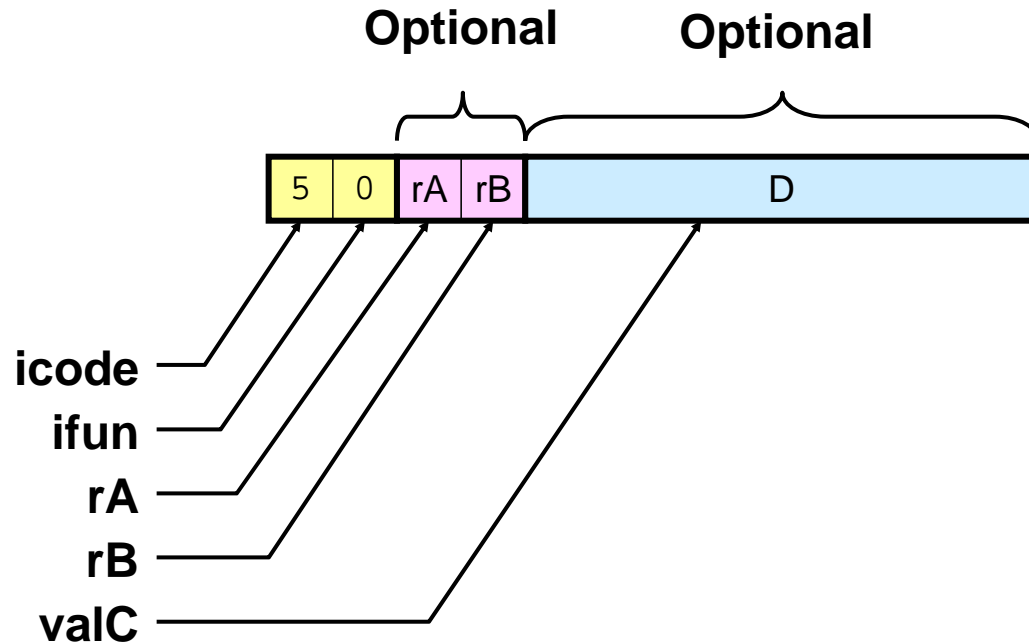


# SEQ Stages

- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter



# Instruction Decoding



- Instruction Format
  - Instruction byte                    icode:ifun
  - Optional register byte            rA:rB
  - Optional constant word            valC

# Executing Arith./Logical Operation



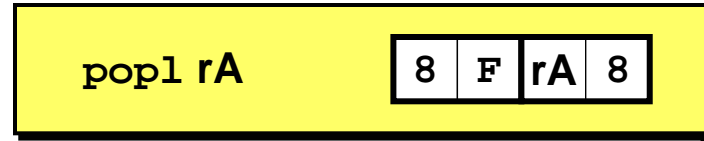
- Fetch
  - Read 2 bytes
  - Do nothing
- Decode
  - Read operand registers
  - Write back
  - Update register
- Execute
  - Perform operation
  - PC Update
  - Increment PC by 2
  - Set condition codes

# Stage Computation: Arith/Log. Ops

|                   | OPI rA, rB  |  |
|-------------------|---|--|
| <b>Fetch</b>      | $icode:ifun \leftarrow M_1[PC]$<br>$rA:rB \leftarrow M_1[PC+1]$<br>$valP \leftarrow PC+2$ | Read instruction byte<br>Read register byte<br><br>Compute next PC |
| <b>Decode</b>     | $valA \leftarrow R[rA]$<br>$valB \leftarrow R[rB]$  | Read operand A<br>Read operand B                                   |
| <b>Execute</b>    | $valE \leftarrow valB \text{ OP } valA$<br>Set CC   | Perform ALU operation<br>Set condition code register               |
| <b>Memory</b>     |   |  |
| <b>Write back</b> | $R[rB] \leftarrow valE$   | Write back result  |
| <b>PC update</b>  | $PC \leftarrow valP$  | Update PC  |

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

# Executing popl



- Fetch
  - Read 2 bytes
  - Read from old stack pointer
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Write back
  - Update stack pointer
  - Write result to register
- PC Update
  - Increment PC by 2

# POP Instruction

## POP—Pop a Value from the Stack

| Opcode        | Instruction    | Description  |
|---------------|----------------|--|
| 8F /0         | POP <i>m16</i> | Pop top of stack into <i>m16</i> ; increment stack pointer |
| 8F /0         | POP <i>m32</i> | Pop top of stack into <i>m32</i> ; increment stack pointer |
| 58+ <i>rw</i> | POP <i>r16</i> | Pop top of stack into <i>r16</i> ; increment stack pointer |
| 58+ <i>rd</i> | POP <i>r32</i> | Pop top of stack into <i>r32</i> ; increment stack pointer |
| 1F            | POP DS         | Pop top of stack into DS; increment stack pointer          |
| 07            | POP ES         | Pop top of stack into ES; increment stack pointer          |
| 17            | POP SS         | Pop top of stack into SS; increment stack pointer          |
| 0F A1         | POP FS         | Pop top of stack into FS; increment stack pointer          |
| 0F A9         | POP GS         | Pop top of stack into GS; increment stack pointer          |

# Stage Computation: popl

| popl rA    |   |   |
|------------|---|---|
| Fetch      | icode:ifun $\leftarrow M_1[PC]$<br>rA:rB $\leftarrow M_1[PC+1]$ | Read instruction byte<br>Read register byte |
|            | valP $\leftarrow PC+2$  | Compute next PC                             |
| Decode     | valA $\leftarrow \%esp$   | Read stack pointer                          |
|            | valB $\leftarrow \%esp$   | Read stack pointer                          |
| Execute    | valE $\leftarrow valB + 4$                                      | Increment stack pointer                     |
| Memory     | valM $\leftarrow M_4[valA]$                                     | Read from stack                             |
| Write back | R[%esp] $\leftarrow valE$                                       | Update stack pointer                        |
|            | R[rA] $\leftarrow valM$   | Write back result                           |
| PC update  | PC $\leftarrow valP$  | Update PC                                   |

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer

# Executing Jumps



- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

# Stage Computation: Jumps

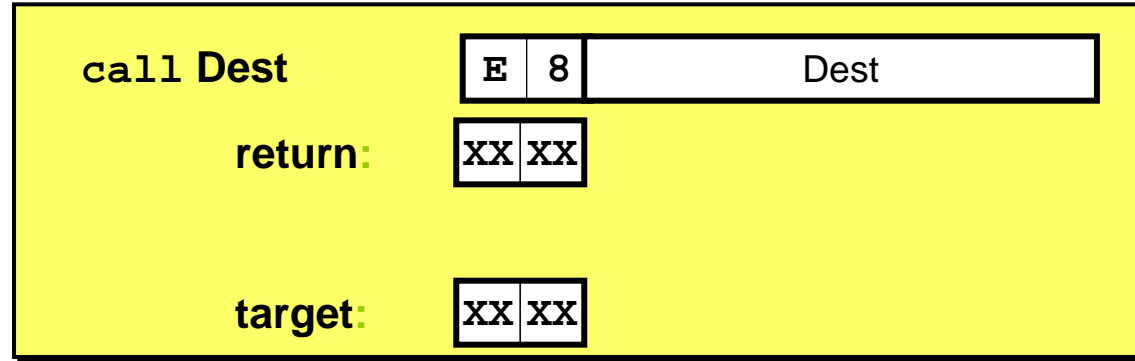
|            | jXX Dest                          |                          |
|------------|-----------------------------------|--------------------------|
| Fetch      | $icode:ifun \leftarrow M_1[PC]$   | Read instruction byte    |
|            | $valC \leftarrow M_4[PC+1]$       | Read destination address |
|            | $valP \leftarrow PC+5$            | Fall through address     |
| Decode     |                                   |                          |
| Execute    | $Bch \leftarrow Cond(CC,ifun)$    | Take branch?             |
| Memory     |                                   |                          |
| Write back |                                   |                          |
| PC update  | $PC \leftarrow Bch ? valC : valP$ | Update PC                |

- Compute both addresses
- Choose based on setting of condition codes and branch condition

## Jcc—Jump if Condition Is Met

| Opcode             | Instruction         | Description                                      |
|--------------------|---------------------|--|
| 77 <i>cb</i>       | JA <i>rel8</i>      | Jump short if above (CF=0 and ZF=0)              |
| 73 <i>cb</i>       | JAE <i>rel8</i>     | Jump short if above or equal (CF=0)              |
| 72 <i>cb</i>       | JB <i>rel8</i>      | Jump short if below (CF=1)                       |
| 76 <i>cb</i>       | JBE <i>rel8</i>     | Jump short if below or equal (CF=1 or ZF=1)      |
| 72 <i>cb</i>       | JC <i>rel8</i>      | Jump short if carry (CF=1)                       |
| E3 <i>cb</i>       | JCXZ <i>rel8</i>    | Jump short if CX register is 0                   |
| E3 <i>cb</i>       | JECXZ <i>rel8</i>   | Jump short if ECX register is 0                  |
| 74 <i>cb</i>       | JE <i>rel8</i>      | Jump short if equal (ZF=1)                       |
| 7F <i>cb</i>       | JG <i>rel8</i>      | Jump short if greater (ZF=0 and SF=OF)           |
| 7D <i>cb</i>       | JGE <i>rel8</i>     | Jump short if greater or equal (SF=OF)           |
| 7C <i>cb</i>       | JL <i>rel8</i>      | Jump short if less (SF<>OF)                      |
| 7E <i>cb</i>       | JLE <i>rel8</i>     | Jump short if less or equal (ZF=1 or SF<>OF)     |
| 76 <i>cb</i>       | JNA <i>rel8</i>     | Jump short if not above (CF=1 or ZF=1)           |
| 72 <i>cb</i>       | JNAE <i>rel8</i>    | Jump short if not above or equal (CF=1)          |
| 73 <i>cb</i>       | JNB <i>rel8</i>     | Jump short if not below (CF=0)                   |
| 77 <i>cb</i>       | JNBE <i>rel8</i>    | Jump short if not below or equal (CF=0 and ZF=0) |
| 73 <i>cb</i>       | JNC <i>rel8</i>     | Jump short if not carry (CF=0)                   |
| 75 <i>cb</i>       | JNE <i>rel8</i>     | Jump short if not equal (ZF=0)                   |
| 7E <i>cb</i>       | JNG <i>rel8</i>     | Jump short if not greater (ZF=1 or SF<>OF)       |
| 7C <i>cb</i>       | JNGE <i>rel8</i>    | Jump short if not greater or equal (SF<>OF)      |
| 7D <i>cb</i>       | JNL <i>rel8</i>     | Jump short if not less (SF=OF)                   |
| 7F <i>cb</i>       | JNLE <i>rel8</i>    | Jump short if not less or equal (ZF=0 and SF=OF) |
| 71 <i>cb</i>       | JNO <i>rel8</i>     | Jump short if not overflow (OF=0)                |
| 7B <i>cb</i>       | JNP <i>rel8</i>     | Jump short if not parity (PF=0)                  |
| 79 <i>cb</i>       | JNS <i>rel8</i>     | Jump short if not sign (SF=0)                    |
| 75 <i>cb</i>       | JNZ <i>rel8</i>     | Jump short if not zero (ZF=0)                    |
| 70 <i>cb</i>       | JO <i>rel8</i>      | Jump short if overflow (OF=1)                    |
| 7A <i>cb</i>       | JP <i>rel8</i>      | Jump short if parity (PF=1)                      |
| 7A <i>cb</i>       | JPE <i>rel8</i>     | Jump short if parity even (PF=1)                 |
| 7B <i>cb</i>       | JPO <i>rel8</i>     | Jump short if parity odd (PF=0)                  |
| 78 <i>cb</i>       | JS <i>rel8</i>      | Jump short if sign (SF=1)                        |
| 74 <i>cb</i>       | JZ <i>rel8</i>      | Jump short if zero (ZF = 1)                      |
| 0F 87 <i>cw/cd</i> | JA <i>rel16/32</i>  | Jump near if above (CF=0 and ZF=0)               |
| 0F 83 <i>cw/cd</i> | JAE <i>rel16/32</i> | Jump near if above or equal (CF=0)               |
| 0F 82 <i>cw/cd</i> | JB <i>rel16/32</i>  | Jump near if below (CF=1)                        |
| 0F 86 <i>cw/cd</i> | JBE <i>rel16/32</i> | Jump near if below or equal (CF=1 or ZF=1)       |
| 0F 82 <i>cw/cd</i> | JC <i>rel16/32</i>  | Jump near if carry (CF=1)                        |

# Executing call



- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 4
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

# Stage Computation: call

|            | call Dest                       |                             |
|------------|---------------------------------|-----------------------------|
| Fetch      | $icode:ifun \leftarrow M_1[PC]$ | Read instruction byte       |
|            | $valC \leftarrow M_4[PC+1]$     | Read destination address    |
|            | $valP \leftarrow PC+5$          | Compute return point        |
| Decode     | $valB \leftarrow R[\%esp]$      | Read stack pointer          |
| Execute    | $valE \leftarrow valB + -4$     | Decrement stack pointer     |
| Memory     | $M_4[valE] \leftarrow valP$     | Write return value on stack |
| Write back | $R[\%esp] \leftarrow valE$      | Update stack pointer        |
| PC update  | $PC \leftarrow valC$            | Set PC to destination       |

- Use ALU to decrement stack pointer
- Store incremented PC

# Call Opcodes in X86

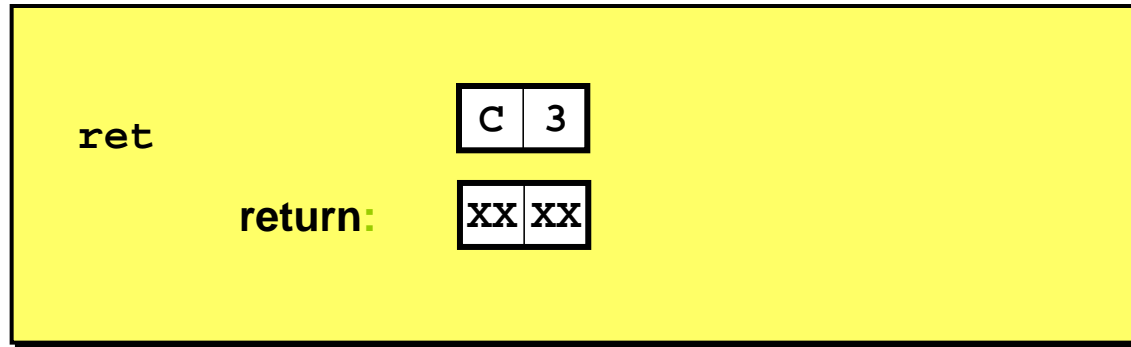
## CALL—Call Procedure

| Opcode       | Instruction          | Description  |
|--------------|----------------------|--|
| E8 <i>cw</i> | CALL <i>rel16</i>    | Call near, relative, displacement relative to next instruction |
| E8 <i>cd</i> | CALL <i>rel32</i>    | Call near, relative, displacement relative to next instruction |
| FF /2        | CALL <i>r/m16</i>    | Call near, absolute indirect, address given in <i>r/m16</i>    |
| FF /2        | CALL <i>r/m32</i>    | Call near, absolute indirect, address given in <i>r/m32</i>    |
| 9A <i>cd</i> | CALL <i>ptr16:16</i> | Call far, absolute, address given in operand                   |
| 9A <i>cp</i> | CALL <i>ptr16:32</i> | Call far, absolute, address given in operand                   |
| FF /3        | CALL <i>m16:16</i>   | Call far, absolute indirect, address given in <i>m16:16</i>    |
| FF /3        | CALL <i>m16:32</i>   | Call far, absolute indirect, address given in <i>m16:32</i>    |

### Description

This instruction saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general-purpose register, or a memory location.

# Executing ret



- Fetch
  - Read 1 byte
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read return address from old stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to return address

# Stage Computation: ret

| ret        |  |  |
|------------|--|--|
| Fetch      | $icode:ifun \leftarrow M_1[PC]$                          | Read instruction byte                                    |
| Decode     | $valA \leftarrow R[\%esp]$<br>$valB \leftarrow R[\%esp]$ | Read operand stack pointer<br>Read operand stack pointer |
| Execute    | $valE \leftarrow valB + 4$                               | Increment stack pointer                                  |
| Memory     | $valM \leftarrow M_4[valA]$                              | Read return address                                      |
| Write back | $R[\%esp] \leftarrow valE$                               | Update stack pointer                                     |
| PC update  | $PC \leftarrow valM$                                     | Set PC to return address                                 |

- Use ALU to increment stack pointer
- Read return address from memory

## RET—Return from Procedure

| Opcode       | Instruction      | Description  |
|--------------|------------------|--|
| C3           | RET              | Near return to calling procedure                                       |
| CB           | RET              | Far return to calling procedure  |
| C2 <i>iw</i> | RET <i>imm16</i> | Near return to calling procedure and pop <i>imm16</i> bytes from stack |
| CA <i>iw</i> | RET <i>imm16</i> | Far return to calling procedure and pop <i>imm16</i> bytes from stack  |

### Description

This instruction transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.