# Computer Architecture: Instruction Set Principles

Berk Sunar Thomas Eisenbarth

ECE 505


WPI

# Outline

- Measuring Performance
- **Review of Instruction Set Architecture**
  1. **Classification of ISA**
  2. **How to analyze Instruction Set architecture**
  3. **Example: MIPS ISA**

# Classification of ISAs

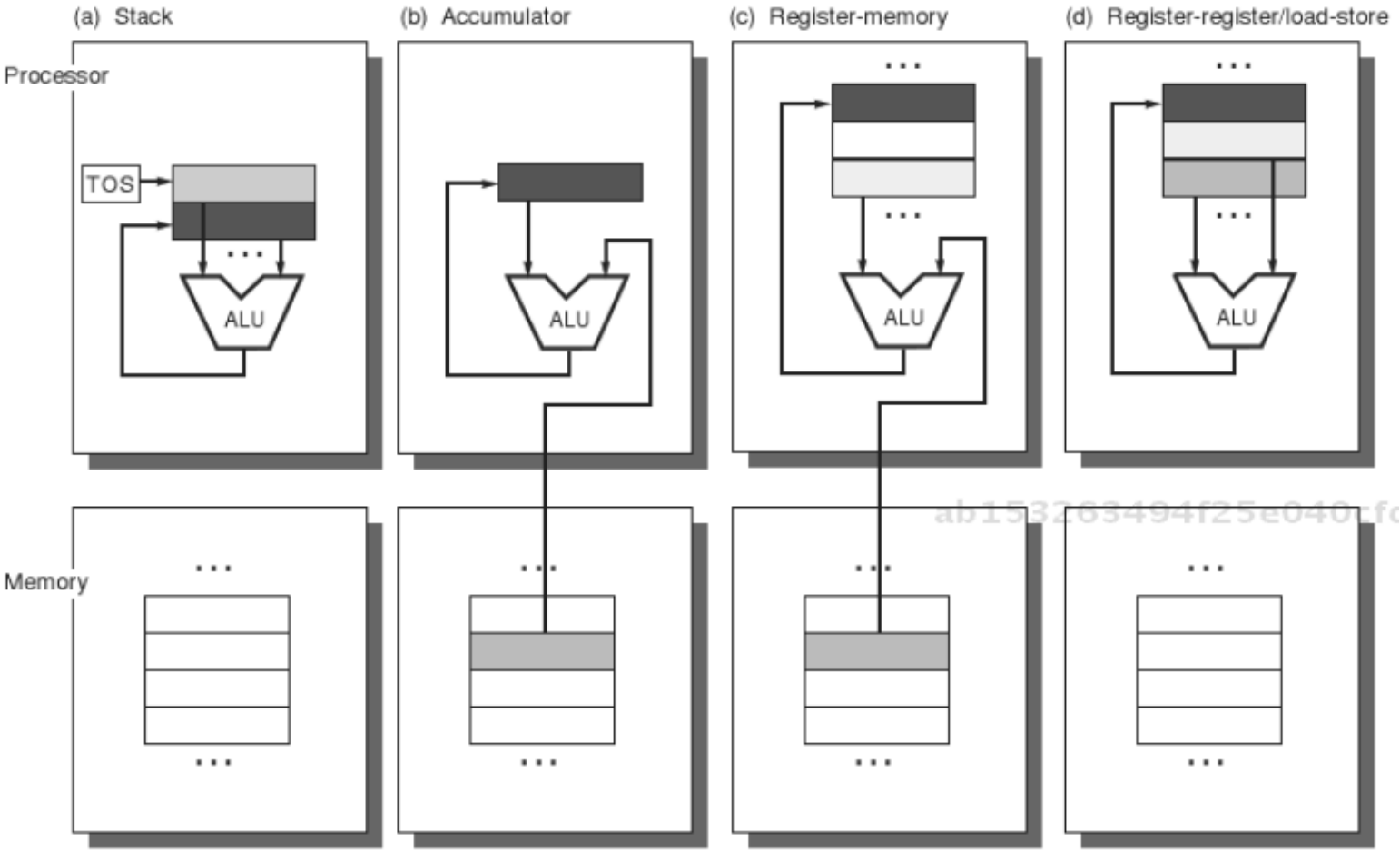
… based on operand specification:

- Stack architecture: operators on top of stack
- Accumulator architecture: one operand is accu
- General Purpose Register (GPR) Architecture
  - Most popular: registers are fast and efficient for compiler

Two types of GPR Architecture:

- Register-memory architecture: subset of operands directly from memory
- Load-store architecture: operates on registers only; operands must be moved to registers first
  - Almost all newer designs are load-store

# Examples of ISA classes




Source: H+P book

# Code Example: C=A+B

Stack architecture:

```
Push A
Push B
Add
Pop C
```

Accumulator architecture:

```
Load A
Add B
Pop C
```

Register-memory:

```
Load R1, A
Add  R3, R1, B
Stor R3, C
```

Register (load-store):

```
Load R1, A
Load R2, B
Add  R3, R1, R2
Stor R3, C
```

# GPR Computers Comparison

**Register-Register** (e.g. ARM, PowerPC, SPARC)

+ Simple fixed length instruction encoding
+ Instructions take similar clock cycles to execute
+ Simple code generation
- Higher instruction count than below architectures
- More instructions + lower density➔ larger programs

**Register-Memory** (e.g. Intel 80x86, Motorola 68000)

+ Data accessible without extra load instruction
+ Instruction format easy to encode + good density
- CPI may vary by operand location
- Source operand may be overwritten in binary operations

**Memory-Memory** (VAX)

+ Most Compact
- Large variation in instruction size (especially if 3-operand instructions possible)
- Large variation in work per instruction
- Memory accesses create bottleneck → no longer used

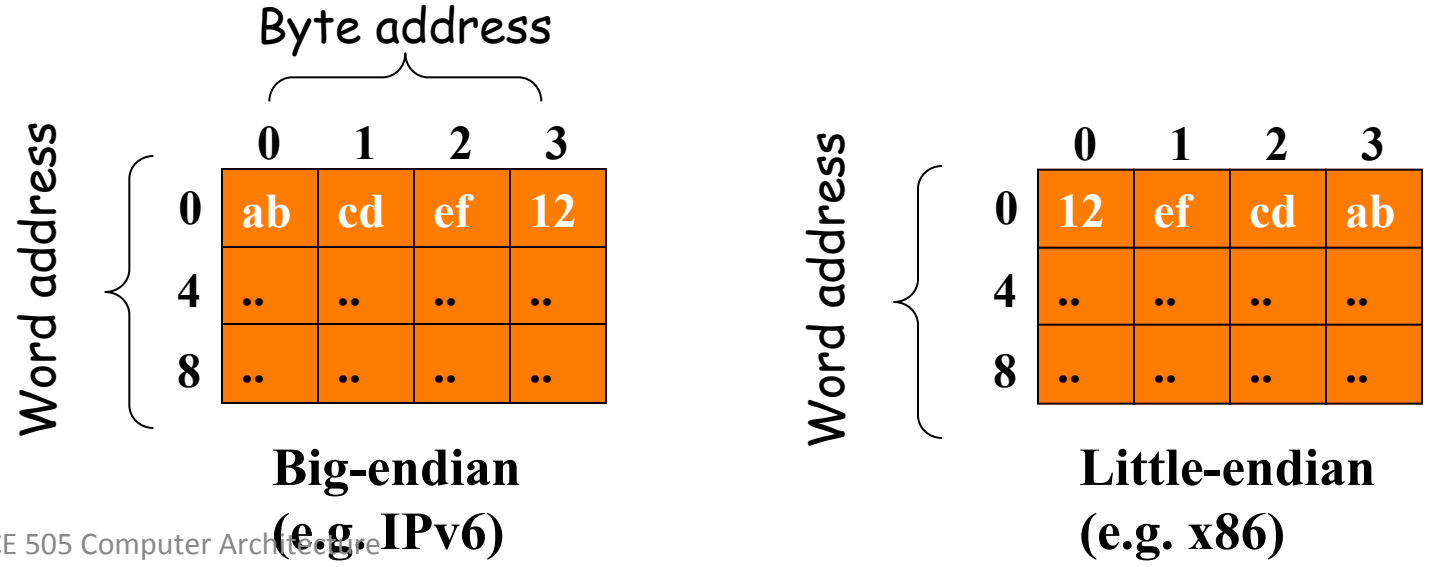
# Memory Addressing

Data available in chunks of

- Bytes (8 bit);
- Half words (16 bit)
- Words (32 bit);
- Double Words (64 bits)

"Alignment restriction" ➔ faster data transfer

Endianness:
how is `(0xabcdef12)` represented in memory?



# Addressing Modes (Selection)

- Register addressing: `Add R4, R3`
  - operands are in registers
- Immediate addressing: `Add R4, #3`
  - operand is a constant within the instruction
- Base or displacement addressing: `Add R4, 100(R1)`
  - operand is at the memory location whose address is the sum of a register(R1) and a constant (100) in the instruction
- Register indirect addressing: `Add R4,(R1)`
  - operand is at the memory location whose address is stored in register (R1)
- Direct Addressing: `Add R4,(1001)`
  - Operand is at the memory location whose address is directly given in the instruction

# Operations Classes

- **Arithmetic and logical**
  - Integer arithmetic and logical: add, subtract, and, or, multiply
- **Data transfer**
  - Load and store commands
- **Control**
  - Branch, jump, call, return

Optional:

- System: OS call, memory management instructions, VMM support etc.
- Floating Point: Floating point arithmetic: add, multiply …
- String: string move, compare, search
- Graphics: Pixel and vertex operations

# Example: Top 10 80x86 Instructions

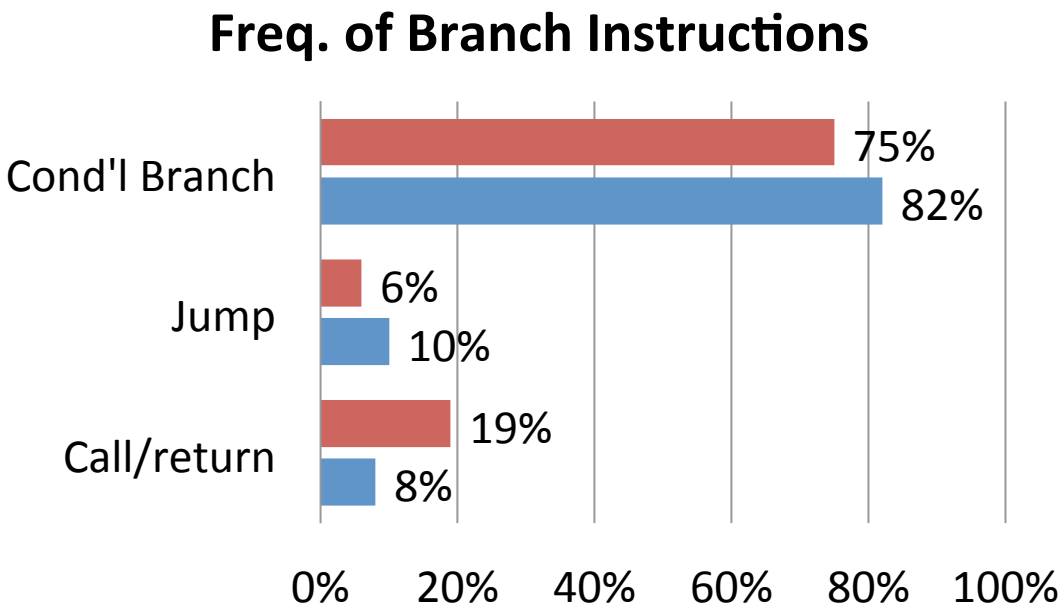
| Rank | 80x86 Instruction | % of total executed (SPECint92) |
|---|---|---|
| 1 | Load | 22% |
| 2 | Conditional branch | 20% |
| 3 | Compare | 16% |
| 4 | Store | 12% |
| 5 | Add | 8% |
| 6 | And | 6% |
| 7 | Sub | 5% |
| 8 | Move reg-reg | 4% |
| 9 | Call/return | 1%/1% |
| **Total** | | **96%** |

- Small number of instructions makes majority of executed instructions→make common case fast
- Most are data transfer and control instructions

# Instructions for Control Flow

## Types of Instructions:

- Conditional Branches
- Jumps
- Procedure Calls
- Procedure Returns




## Addressing Modes:

Destination address is specified as:

- PC-relative: displacement added to program counter
  - Position independent: helpful when code is dynamically linked
  - Q: How many bits for displacement value?
- Register Indirect: go to address provided in register

# Encoding an Instruction Set

How are instructions stored in memory (programs)?

- Operation specified as opcode
- How to represent addressing modes?
- Variable or fixed size instructions?

**Design Goals:**

- Flexibility: Address as many registers and address modes as possible
- Keep code size small: Impact of above on average instruction size, hence program size
- Speedy execution: Instructions should align (no odd sizes) and consumable by pipeline (later).

# Variations in Instruction Encoding

**Fixed Format:** e.g. ARM, MIPS, PowerPC


**Variable Format:** e.g. Intel 80x86


**Mixed Format:** e.g. Thumb, MIPS16

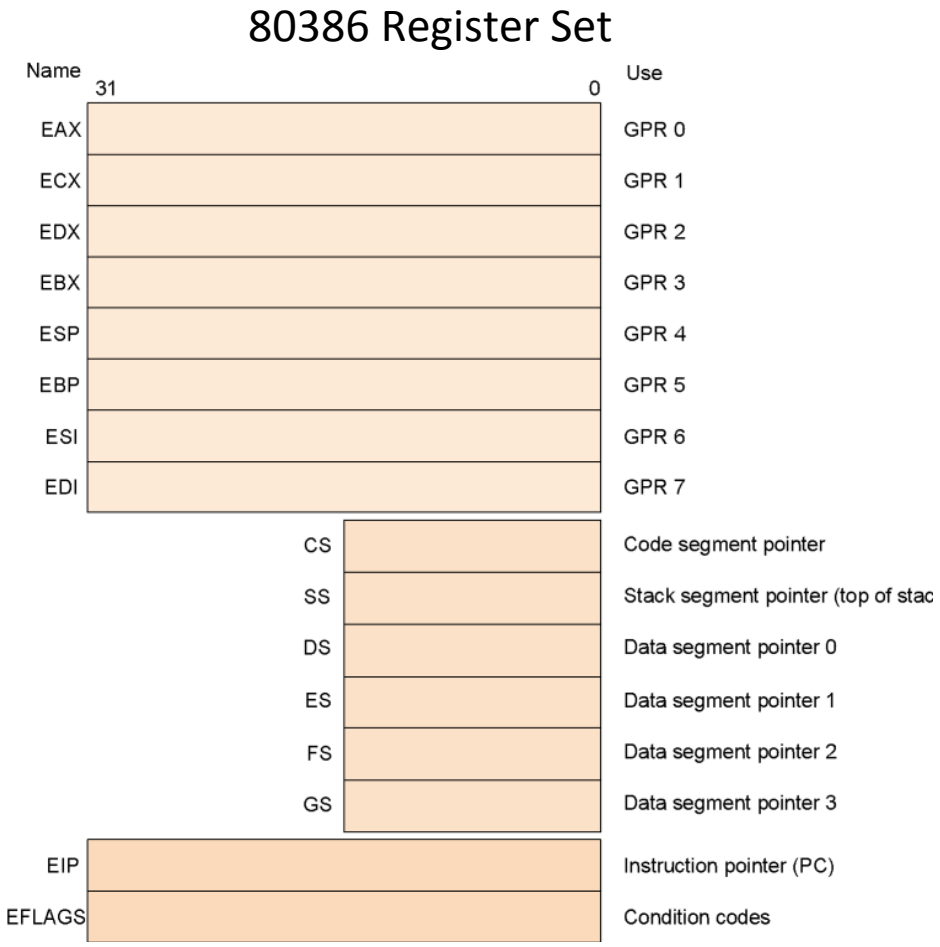
- Always same number of operands, with addressing modes specified as part of opcode
- Fields do not vary in location, but by how content is used/interpreted

# Example Instructions for Intel 80386

## Instruction encoding:

- Variable format
- Length: 1 - 17 bytes
- High flexibility

➔80x86 usually has smaller programs than RISC Architectures



# Typical 80386 Instruction Formats

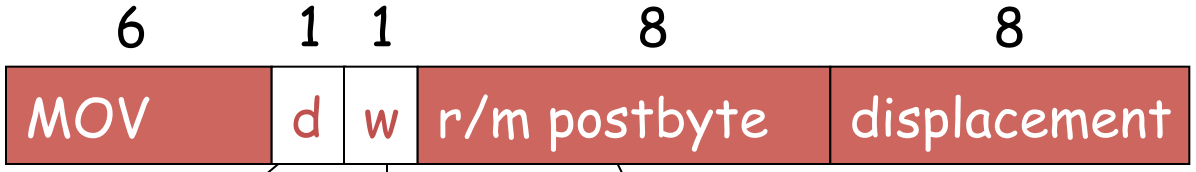

JS name

if negative (condition code)
go to [EIP + displacement]


CALL name

SP = SP - 4
M[SP] = EIP + 5
EIP = Offset

PC




MOV EBX, [EDI+45]

direction

byte or word?

specify target register, which registers are used in address calculation, how?

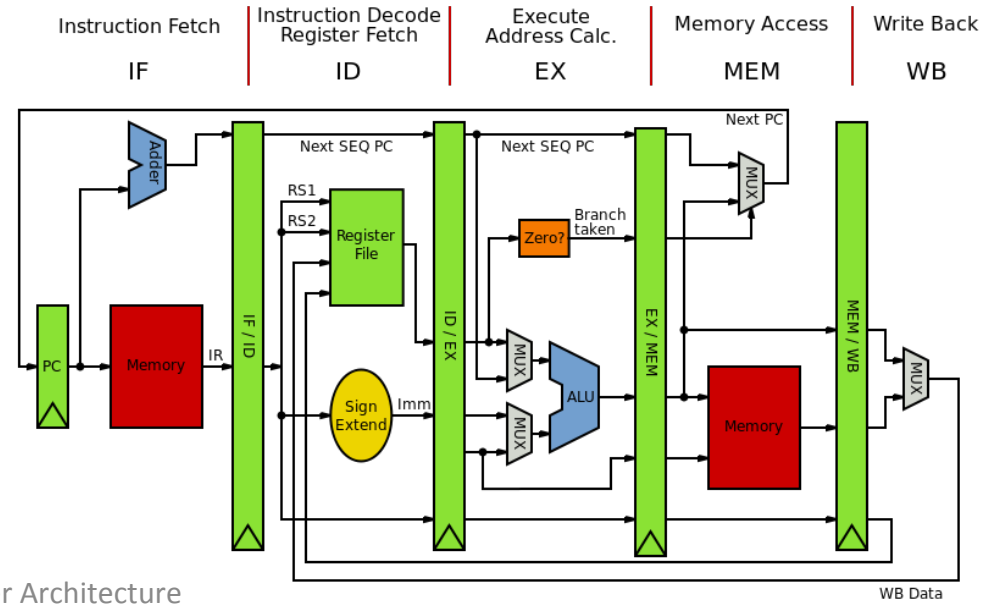
# An Example: the MIPS Architecture

A RISC processor, with design choices:

- General purpose register architecture, Load-store version
- Addressing modes:
  - Displacement (with offset of approx. 16 bits)
  - Immediate (16 bits)
  - Register indirect and Absolute are derived
- Supports:
  - common data sizes: 8, 16, 32, 64 bit integer (+float)
  - Simple instructions: load, store add, sub, move reg-reg, shift
  - Branches: Compare equal, compare not equal, compare less, branch, jump, call, return
- Fixed (performance) or variable (code size) instruction encoding; MIPS is fixed

# The MIPS Architecture

- Simple load-store instruction set
- Designed for pipelining efficiency (fixed instruction set encoding)
- Efficiency as compiler target



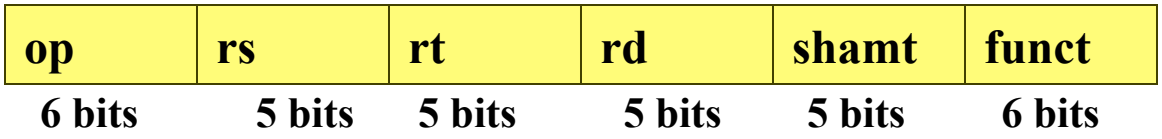

Source: Wikipedia

# The MIPS 64

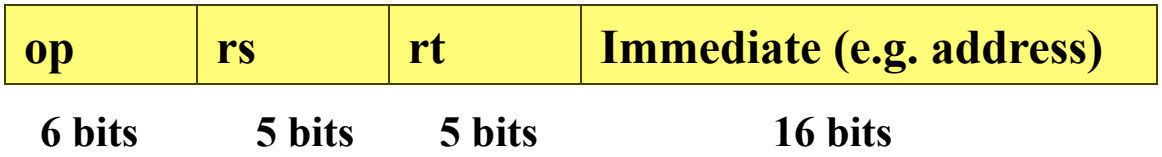
- 32x 64-bit General Purpose Registers (R0 – R31)
- 32x 64-bit (or 32bit) floating point registers (Fi)
- R0 = 0 always
- Byte addressable memory with 64-bit address
- Endianness is choosable, aligned memory access
- 32-bit instructions
  - 6-bit primary opcode
  - Up to 3 registers (5-bit each)
  - Three different types

# Three types of MIPS Instructions

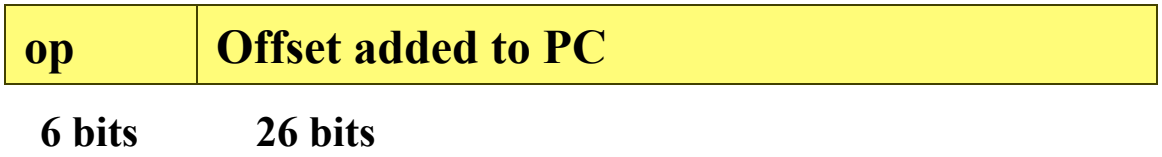
**R-type instruction:** Register-register ALU operations and moves



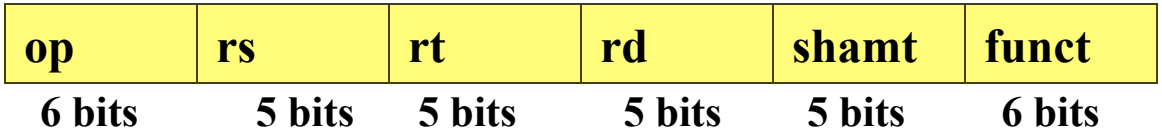

**I-type instruction:** Immediate, including load/store, conditional branch




**J-type instruction:** unconditional branches: jump/trap/return



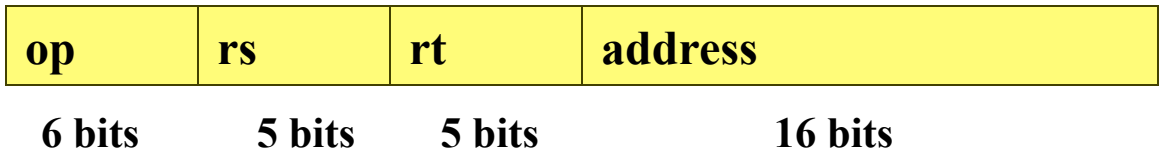
# R-Type Instruction format



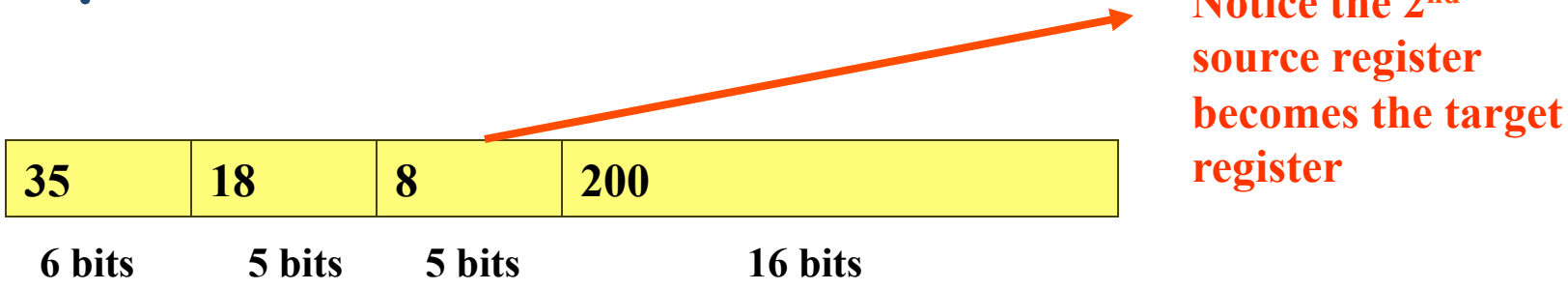

- **op:** *opcode* (type of the operation)
- **funct:** selects the specific variant of the operation in the op field. *Function* but sometimes called *function field.*
- **rs:** The 1st register source operand
- **rt:** The 2nd register source operand
- **rd:** The 3rd register destination operand
- **shamt:** shift amount

# I-Type Instruction Format

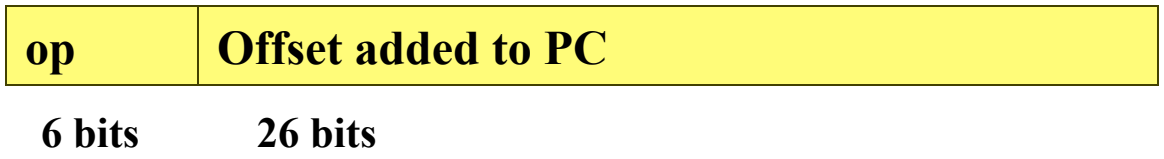
- MIPS designers elect to introduce a new type of instruction format by keeping the length same:
  - I-type for Immediate




- **Example:** `LW R8, 200(R18)`



# J-Type Instruction Format




J-type instructions offer extra long immediate as offset to PC

Example:

Jump: `J name` ##*PC↓36..63 ←name*

Jump+Link: `JAL name` ##Regs*[R31]*←*PC+8*
##*PC↓36..63 ←name*

Branch delay slot

# Example Instructions

| Example Instruction | Instruction Name | Meaning |
|---|---|---|
| DADDU R1, R2, R3 | Add unsigned | `Regs[R1]←Regs[R2]+Regs[R3]` |
| | | |
| | | |

# Example Instructions

| Example Instruction | Instruction Name | Meaning |
|---|---|---|
| DADDU R1, R2, R3 | Add unsigned | `Regs[R1]←Regs[R2]+Regs[R3]` |
| MOVZ R1, R2, R3 | Conditional move if zero | `If(Regs[R3]=0)`<br>`  Regs[R1]←Regs[R2]` |
| | | |
| | | |

# Example Instructions

| Example Instruction | Instruction Name | Meaning |
|---|---|---|
| DADDU R1, R2, R3 | Add unsigned | `Regs[R1]←Regs[R2]+Regs[R3]` |
| MOVZ R1, R2, R3 | Conditional move if zero | `If(Regs[R3]=0)`<br>`  Regs[R1]←Regs[R2]` |
| LD R1, 30(R2) | Load Double Word | `Regs[R1]←Mem[30+Regs[R2]]` |
| | | |

# Example Instructions

| Example Instruction | Instruction Name | Meaning |
|---|---|---|
| DADDU R1, R2, R3 | Add unsigned | `Regs[R1]←Regs[R2]+Regs[R3]` |
| MOVZ R1, R2, R3 | Conditional move if zero | `If(Regs[R3]=0)`<br>`  Regs[R1]←Regs[R2]` |
| LD R1, 30(R2) | Load Double Word | `Regs[R1]←Mem[30+Regs[R2]]` |
| LD R1, 1000(R0) | Load Double Word | `Regs[R1]←Mem[1000+0]` |
| | | |

# Example Instructions

| Example Instruction | Instruction Name | Meaning |
|---|---|---|
| DADDU R1, R2, R3 | Add unsigned | `Regs[R1]←Regs[R2]+Regs[R3]` |
| MOVZ R1, R2, R3 | Conditional move if zero | `If(Regs[R3]=0)`<br>`  Regs[R1]←Regs[R2]` |
| LD R1, 30(R2) | Load Double Word | `Regs[R1]←Mem[30+Regs[R2]]` |
| LD R1, 1000(R0) | Load Double Word | `Regs[R1]←Mem[1000+0]` |
| BNE R3, R4, name | Branch Not Equal | `If(Regs[R3]!=Regs[R4])`<br>`  PC←name;`<br>`((PC+4)-2^17)<=name<=((PC+4)+2^17)` |

# Comparison: Top 10 MIPS Instructions

| Rank | Instruction | % of total executed (SPECint2000) |
|---|---|---|
| 1 | `Load` | 26% |
| 2 | `Conditional branch` | 12% (20% on x86) |
| 3 | `Compare` | 5% (16% on x86) |
| 4 | `Store` | 10% |
| 5 | `Add` | 19% (8% on x86) |
| 6 | `And` | 4% |
| 7 | `Sub` | 3% |
| 8 | `OR (includes Move reg-reg)` | 9% (1% on X86) |
| 9 | `Call/return` | 1%/1% |
| **Total** | | **90%** |

- *Still* Most are data transfer and control instructions
- Less branching is good for pipeline

# Outlook

Next Time:

Pipelining