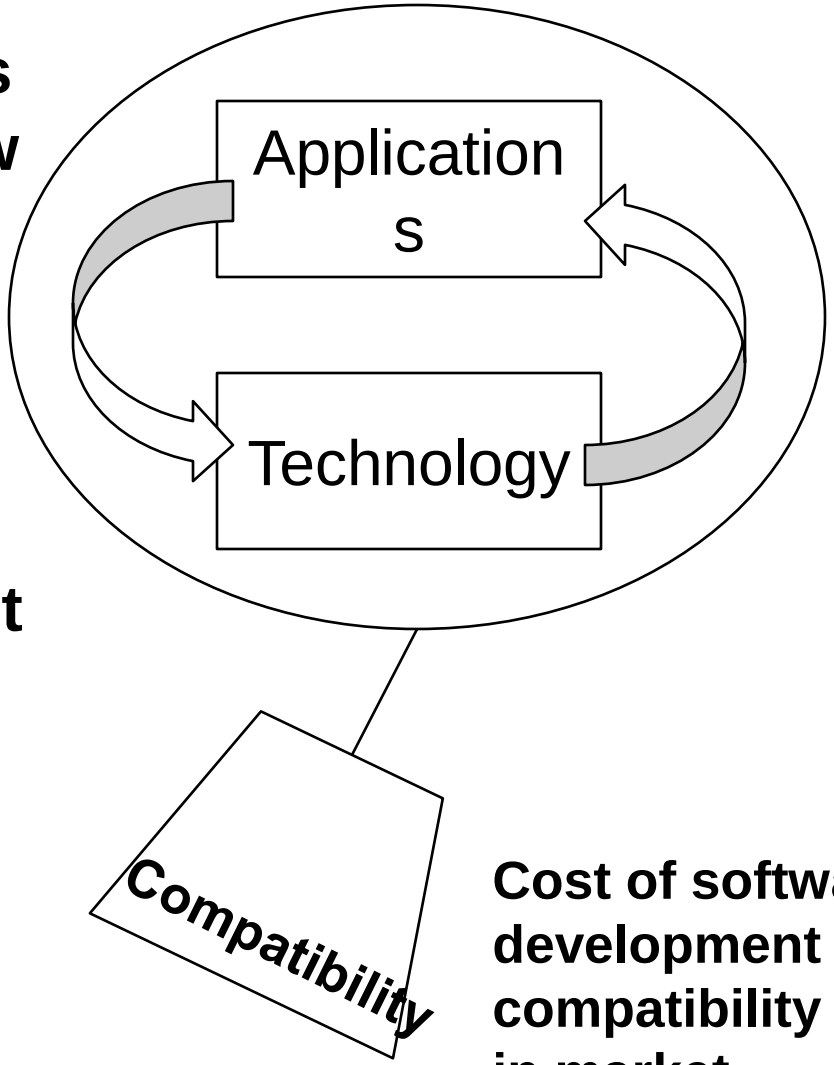
Instruction Set Principles and Architecture

Outline

- Introduction and Motivation
- Instruction Set Architecture Design
- CISC ver. RISC
- Overview of the MIPS Processor

Architecture continually changing

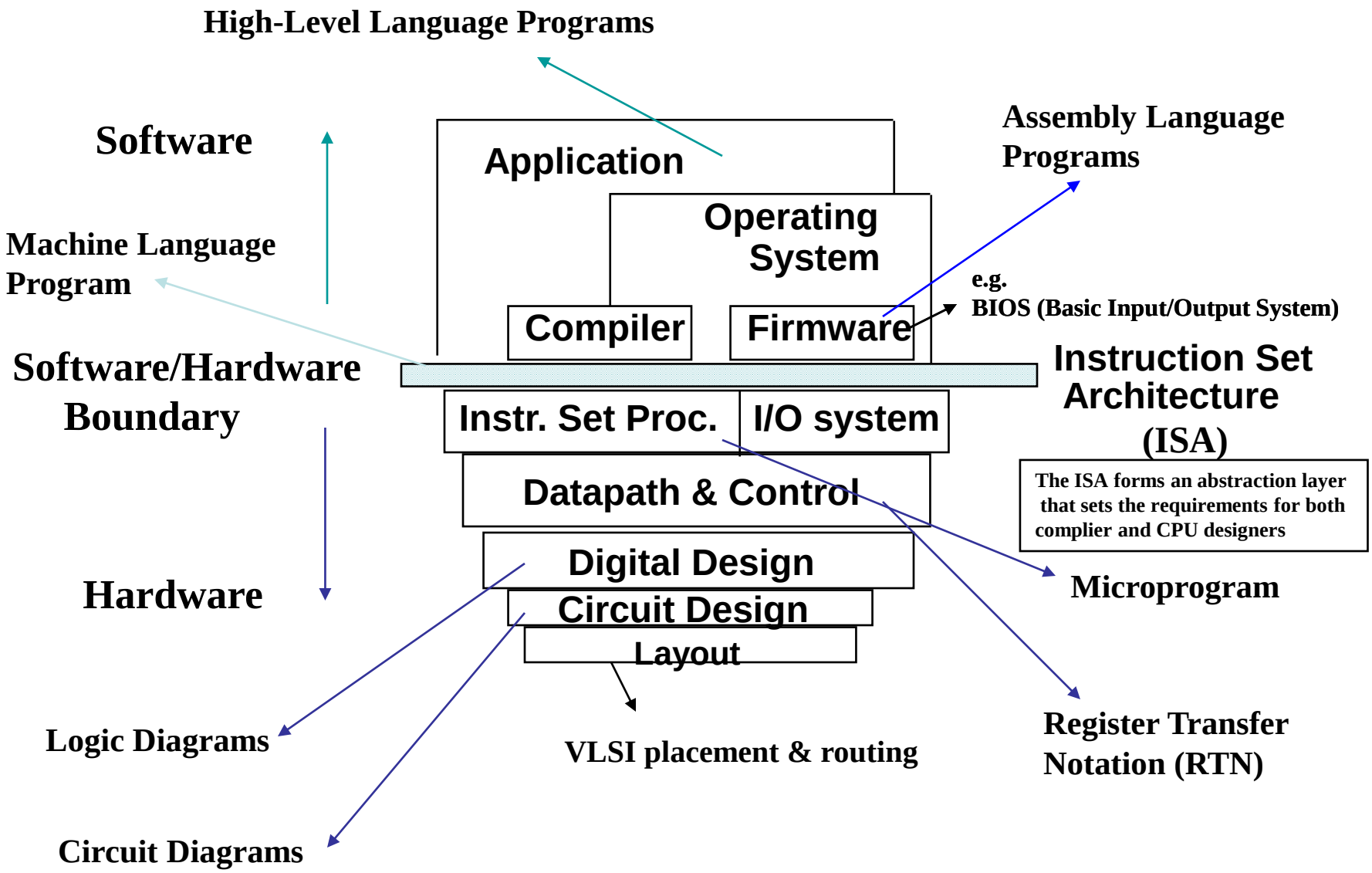
Applications suggest how to improve technology, provide revenue to fund development



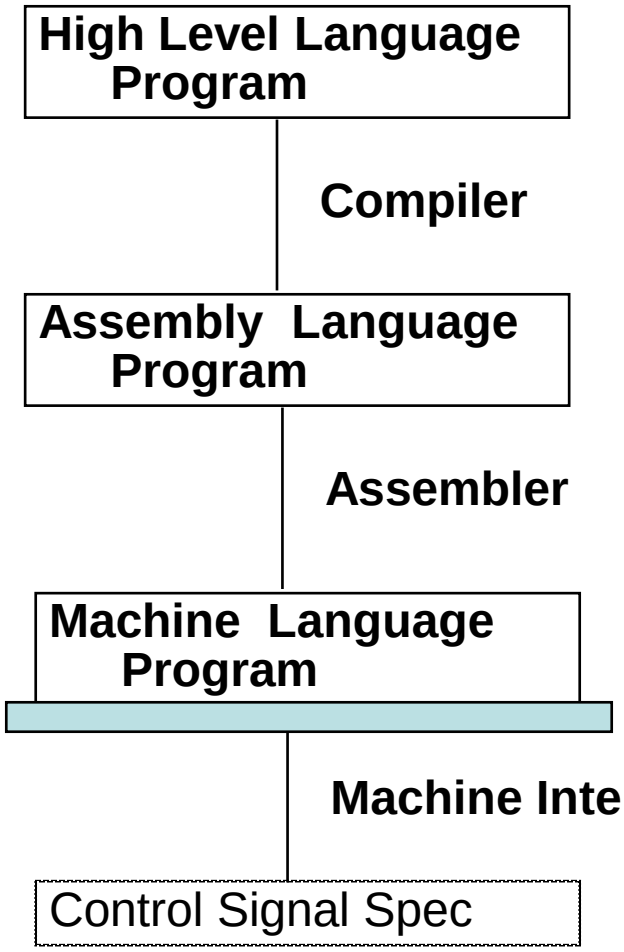
Improved technologies make new applications possible

Cost of software development makes compatibility a major force in market

Hierarchy of Computer Architecture



How to Speak to Computer




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $15,   0($2)
lw  $16,   4($2)
sw $16,   0($2)
sw $15,   4($2)
```

```
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
```

ALUOP[0:3] <= InstReg[9:11] & MASK

Need translation from application to physics

What is "Computer Architecture" ?

- Computer Architecture =

  Instruction Set Architecture +

  Computer Organization

- Instruction Set Architecture (ISA)
  - WHAT the computer does (logical view)
- Computer Organization
  - HOW the ISA is implemented (physical view)
- We will study both in this course

Instruction Set Architecture (ISA)

- Complete set of instructions used by a machine
- Abstract interface between the HW and lowest-level SW.
- An ISA includes the following …
  - Instructions and Instruction Formats
  - Data Types, Encodings, and Representations
  - Programmable Storage: Registers and Memory
  - Addressing Modes: to address Instructions and Data
  - Handling Exceptional Conditions (like division by zero)
- Advantages:
  - Different implementations of the same architecture
  - Easier to change than HW
  - Standardizes instructions, machine language bit patterns, etc.
- Disadvantage:
  - Sometimes prevents using new innovations

Classifying Instruction Sets

- Early Instruction Set Architectures
  - Accumulator-based or Stack-based
  - Replaced with General-Purpose Register (GPR) architectures
- Three classes or general-purpose register architectures
  1. Register-Register (or Load-Store) Architecture (RISC)
     - Can access memory only via load and store instructions
  2. Register-Memory Architecture (CISC)
     - Can access a memory location as part of any instruction
  3. Memory-Memory Architecture (not used today)
     - Can access two or three memory locations per instruction
     - Large variation in instruction size and work per instruction (CPI)

General Purpose Registers Dominate

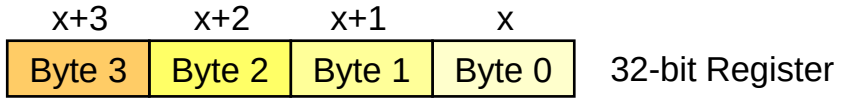
* Since 1975 all machines use general purpose registers
* Advantages of registers
  * registers are faster than memory
  * registers are easier for a compiler to use

    e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order
    Stack is the most restrictive one
  * registers can hold variables
    - memory traffic is reduced, so program is sped up (registers are also faster than memory)
    - code density improves (since register named with fewer bits than memory location)
* Stack machines: Burroughs B55/5700 mainframe, HP3000, Transputer, HP pocket calculators. With new Java machines, stack machines may make a comeback.

Memory Addressing

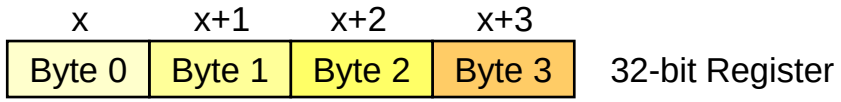
- Most architectures define memory as byte addressable
- A memory address can provide access to …
  - A byte (8 bits), 2 bytes, 4 bytes, 8 bytes, or more bytes
- The word size is defined differently by architectures
  - The word size = 2 bytes (Intel x86), 4 bytes (MIPS), or larger
- Two conventions for ordering bytes within a larger object

1. Little Endian byte ordering



  - Memory address X = address of least-significant byte (Intel x86)

2. Big Endian byte ordering



  - Memory address X = address of most-significant byte (SPARC)

Addressing Modes (Commonly Used)

- How instructions specify the addresses of their operands
- Operands can be in registers, constants, or in memory

Mode	Example	Meaning	When used
Register	ADD R1, R2, R3	R1 ← R2 + R3	Values in registers
Immediate	ADD R1, R2, 100	R1 ← R2 + 100	For constants
Register Indirect	LD R1, [R2]	R1 ← Mem[R2]	R2 contains address
Displacement	LD R1, [R2, 8]	R1 ← Mem[R2 + 8]	Address local variables
Absolute	LD R1, [1000]	R1 ← Mem[1000]	Address static data
Indexed	LD R1, [R2, R3]	R1 ← Mem[R2 + R3]	R2=base, R3=index
Scaled Index	LD R1, [R2, R3, s]	R1 ← Mem[R2 + R3 << s]	s = scale factor
Pre-update	LD R1, [R2, 8] !	R2 ← R2 + 8 R1 ← Mem[R2]	Address is pre-updated Using pointer to traverse array
Post-update	LD R1, [R2], 8	R1 ← Mem[R2] R2 ← R2 + 8	Address is post-updated Using pointer to traverse array

Types and Size of Operands

- Common operand types:
  - ASCII character = 1 byte (64-bit register can store 8 characters)
  - Unicode character or Short integer = 2 bytes = 16 bits (half word)
  - Integer = 4 bytes = 32 bits (word size on many RISC processors)
  - Single-precision float = 4 bytes = 32 bits (word size)
  - Long integer = 8 bytes = 64 bits (double word)
  - Double-precision float = 8 bytes = 64 bits (double word)
  - Extended-precision float = 10 bytes = 80 bits (Intel architecture)
  - Quad-precision float = 16 bytes = 128 bits (quad word)
- 32-bit versus 64-bit architectures
  - 64-bit architectures support 64-bit operands & memory addresses
  - Older architectures were 32-bit (can address 4 GB of memory)

Typical Operations

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	Data Types: (signed & unsigned) Integer (binary + decimal) (signed & unsigned) Floating Point Numbers Operations: Add, Subtract, Multiply, Divide
Logical	Not, and, or, set, clear
Shift	Arithmetic (& Logical) shift (left/right), rotate (left/right)
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronisation	test & set (atomic r-m-w)
String	search, compare, translate

Encoding an Instruction Set

- Variable Encoding
  - Instruction length is a variable number of bytes
  - Allows all addressing modes to be used with all operations
  - Examples: Intel x86 and VAX
- Fixed Encoding
  - All instructions have a single fixed size, typically 32 bits
  - Combines the addressing mode with the opcode
  - Examples: MIPS, ARM, Power, SPARC, etc.
- Hybrid Encoding
  - Few instruction lengths ➔ reduces the variability in length
  - Compressed encoding of some frequently used instructions
  - Examples: micro MIPS and ARM Thumb

Instruction encoding impacts the code size and ease of decoding inside the processor

CISC and RISC

- CISC is an acronym for Complex Instruction Set Computer and are chips that are easy to program and which make efficient use of memory.
- **RISC** Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

CISC	**RISC**
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

Example CISC ISA: Intel 80386

12 addressing modes:

- Register.
- Immediate.
- Direct.
- Base.
- Base + Displacement.
- Index + Displacement.
- Scaled Index + Displacement.
- Based Index.
- Based Scaled Index.
- Based Index + Displacement.
- Based Scaled Index + Displacement.
- Relative.

Operand sizes:

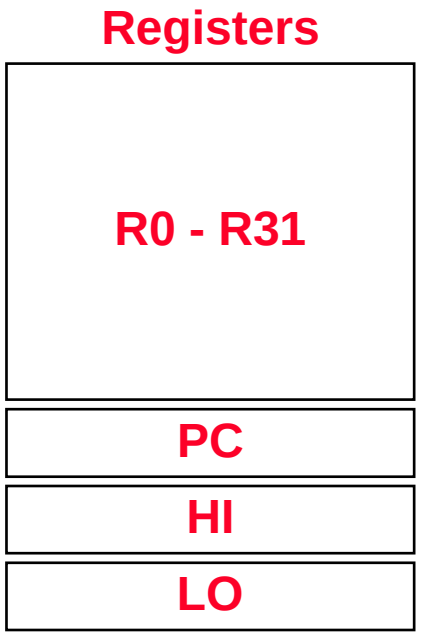
- Can be 8, 16, 32, 48, 64, or 80 bits long.
- Also supports string operations.

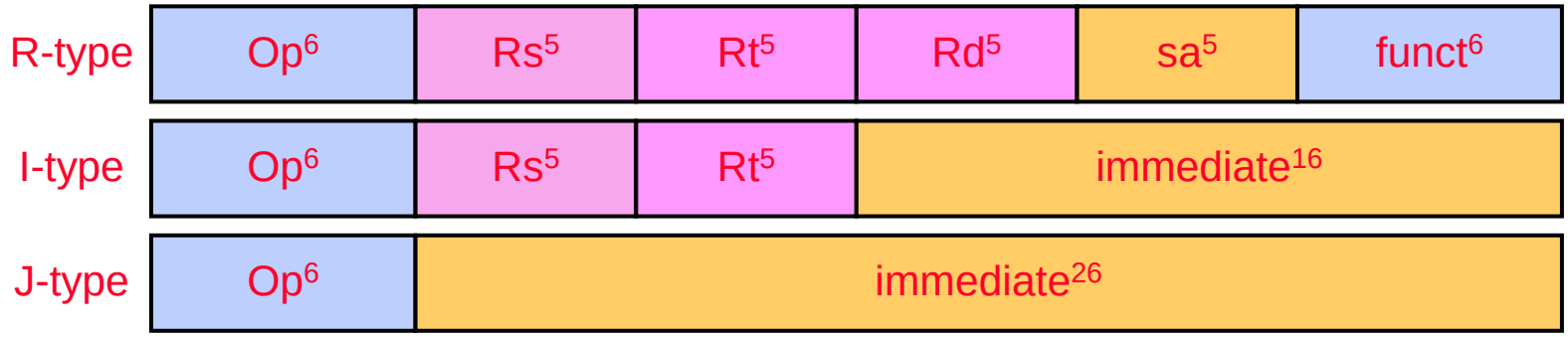
Instruction Encoding:

- The smallest instruction is one byte.
- The longest instruction is 12 bytes long.
- The first bytes generally contain the opcode, mode specifiers, and register fields.
- The remainder bytes are for address displacement and immediate data.

RISC Example: MIPS32 Architecture

- All instructions are 32-bit wide
- Instruction Categories
  - Load/Store
  - Integer Arithmetic
  - Jump and Branch
  - Floating Point
  - Memory Management
- Three Instruction Formats
- Five Addressing Modes



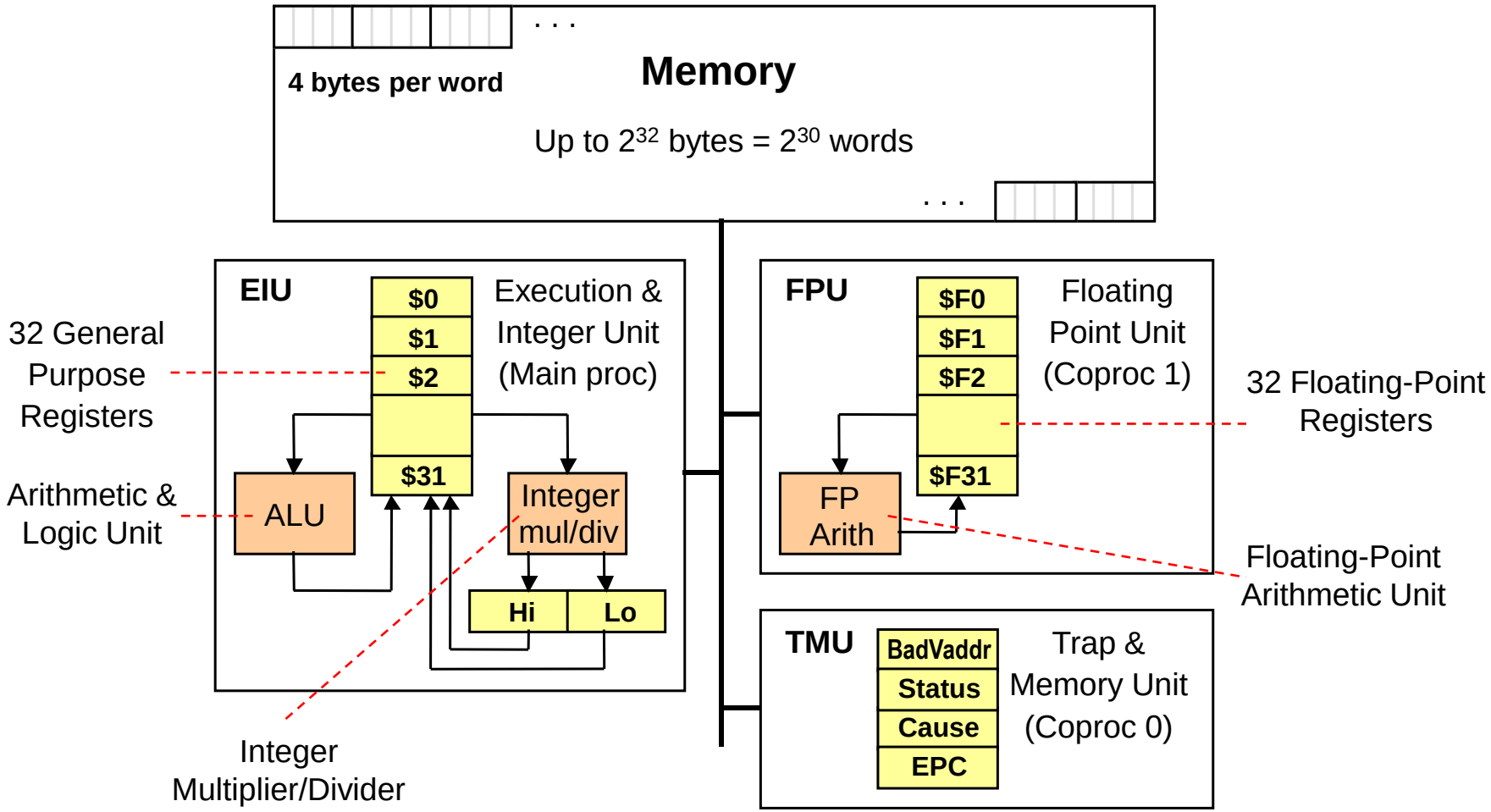


Things to Remember …

- Major reasons for GPR architectures
  - Registers are faster than memory and reduce memory traffic
  - General-Purpose Registers are easier for a compiler to use
  - Register-Register architectures are simpler than Register-Memory
- Programs with aligned memory references run faster
  - Misalignment requires multiple aligned memory references
- Addressing modes specify …
  - Registers, constants, and memory locations
  - Simple addressing modes are frequently used
  - 32 bits can address at most 4GB, 64 bits can address 16 Exabytes
- Most frequently used instructions are the simplest ones
- Instruction encoding impacts size and ease of decoding

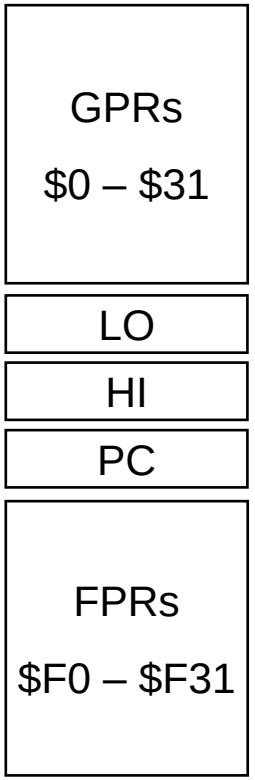
Overview of the MIPS32 Architecture (Microprocessor without Interlocked Pipelined Stages) Processor

Logical View of the MIPS32 Processor



Overview of the MIPS Registers

- 32 General Purpose Registers (GPRs)
  - 32-bit registers are used in MIPS32
  - Register 0 is always zero
  - Any value written to R0 is discarded
- Special-purpose registers LO and HI
  - Hold results of integer multiply and divide
- Special-purpose program counter PC
- 32 Floating Point Registers (FPRs)
  - Floating Point registers can be either 32-bit or 64-bit
  - A pair of registers is used for double-precision floating-point



MIPS General-Purpose Registers

- 32 General Purpose Registers (GPRs)
  - Assembler uses the dollar notation to name registers
    - $0 is register 0, $1 is register 1, …, and $31 is register 31
  - All registers are 32-bit wide in MIPS32
  - Register $0 is always zero
    - Any value written to $0 is discarded
- Software conventions
  - Software defines names to all registers
    - To standardize their use in programs
  - $8 - $15 are called $t0 - $t7
    - Used for temporary values
  - $16 - $23 are called $s0 - $s7

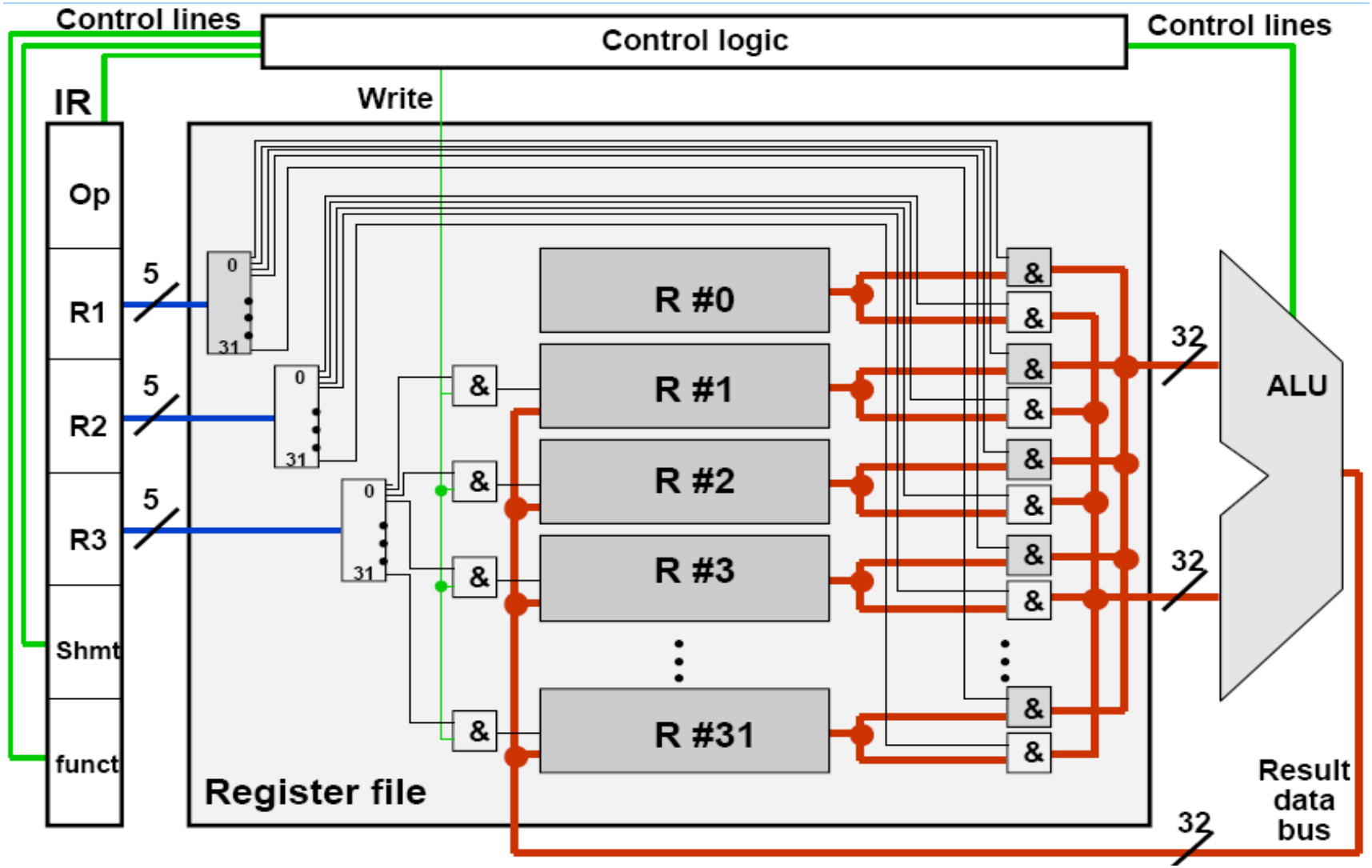
$0 = $zero	$16 = $s0
$1 = $at	$17 = $s1
$2 = $v0	$18 = $s2
$3 = $v1	$19 = $s3
$4 = $a0	$20 = $s4
$5 = $a1	$21 = $s5
$6 = $a2	$22 = $s6
$7 = $a3	$23 = $s7
$8 = $t0	$24 = $t8
$9 = $t1	$25 = $t9
$10 = $t2	$26 = $k0
$11 = $t3	$27 = $k1
$12 = $t4	$28 = $gp
$13 = $t5	$29 = $sp
$14 = $t6	$30 = $fp
$15 = $t7	$31 = $ra

MIPS Register Conventions

- Assembler can refer to registers by name or by number
  - It is easier for you to remember registers by name
  - Assembler converts register name to its corresponding number

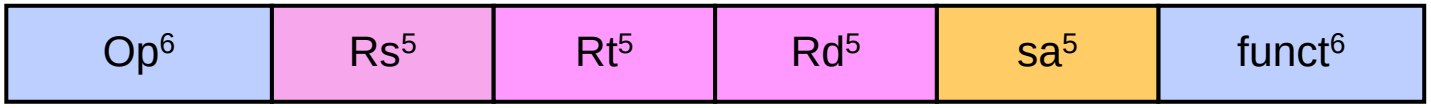
Name	Register	Usage
$zero	$0	Always 0 (forced by hardware)
$at	$1	Reserved for assembler use
$v0 – $v1	$2 – $3	Result values of a function
$a0 – $a3	$4 – $7	Arguments of a function
$t0 – $t7	$8 – $15	Temporary Values
$s0 – $s7	$16 – $23	Saved registers (preserved across call)
$t8 – $t9	$24 – $25	More temporaries
$k0 – $k1	$26 – $27	Reserved for OS kernel
$gp	$28	Global pointer (points to global data)
$sp	$29	Stack pointer (points to top of stack)
$fp	$30	Frame pointer (points to stack frame)
$ra	$31	Return address (used by jal for function call)

MIPS Register File

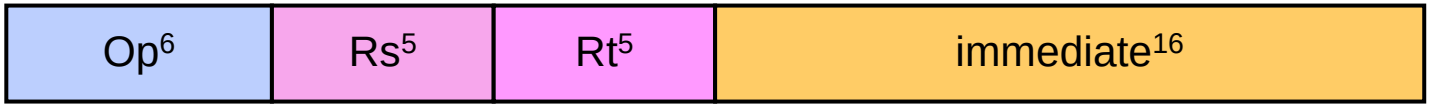


Instruction Formats

- All instructions are 32-bit wide, Three instruction formats:
- Register (R-Type)
  - Register-to-register instructions
  - Op: operation code specifies the format of the instruction



- Immediate (I-Type)
  - 16-bit immediate constant is part in the instruction



- Jump (J-Type)
  - Used by jump instructions

MIPS Five Addressing Modes

1 Register Addressing:

Where the operand is a register (R-Type)

2 Immediate Addressing:

Where the operand is a constant in the instruction (I-Type, ALU)

3 Base or Displacement Addressing:

Where the operand is at the memory location whose address is the sum of a register and a constant in the instruction (I-Type, load/store)

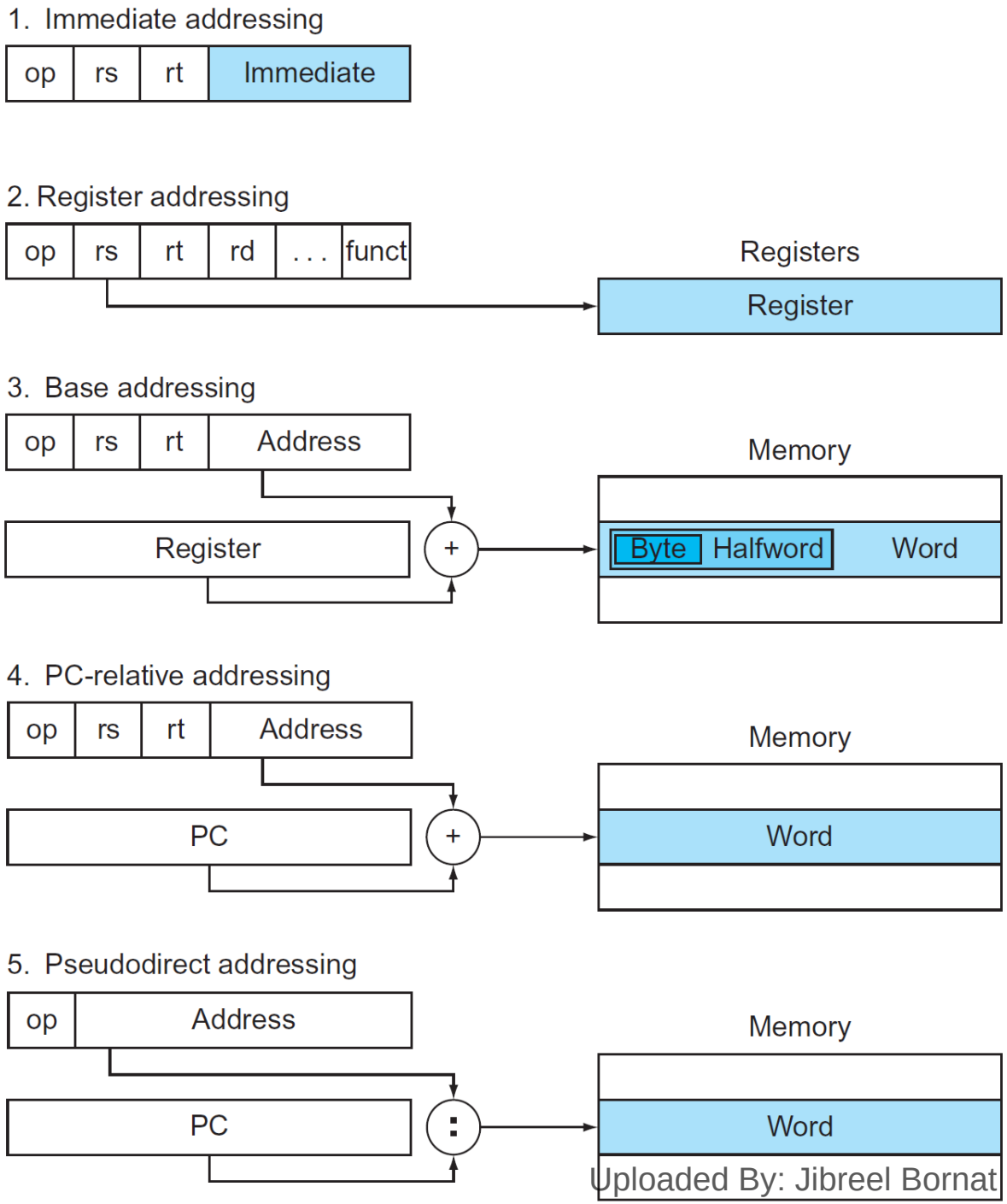
4 PC-Relative Addressing:

Where the address is the sum of the PC and the 16-address field in the instruction shifted left 2 bits. (I-Type, branches)

5 Pseudodirect Addressing:

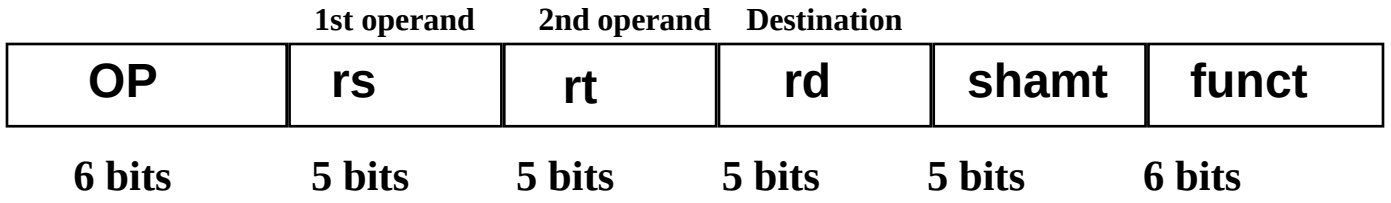
Where the jump address is the 26-bit jump target from the instruction shifted left 2 bits concatenated with the 4 upper bits of the PC (J-Type)

MIPS Addressing Modes/Instruction Formats



MIPS R-Type (ALU) Instruction Fields

R-Type: All ALU instructions that use three registers



- op: Opcode, basic operation of the instruction.
  - For R-Type op = 0
- rs: The first register source operand.
- rt: The second register source operand.
- rd: The register destination operand.
- shamt: Shift amount used in constant shift operations.
- funct: Function, selects the specific variant of operation in the op field.

Rs, rt , rd
are register specifier fields

Operand register in rs

Destination register in rd

Operand register in rt

Examples:

add $1,$2,$3

sub $1,$2,$3

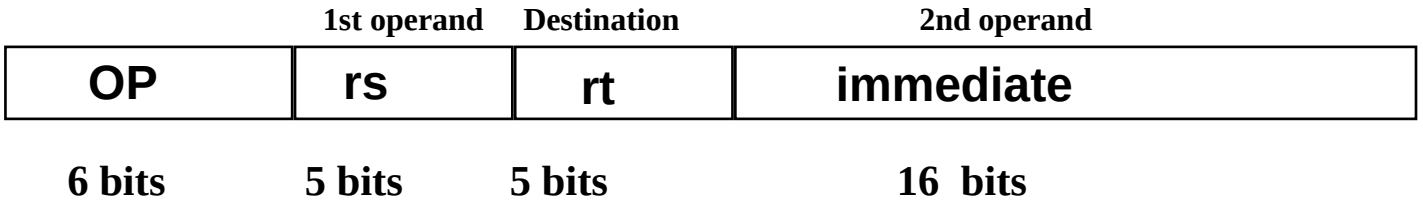
and $1,$2,$3

or $1,$2,$3

R-Type = Register Type
Register Addressing used (Mode 1)

MIPS ALU I-Type Instruction Fields

I-Type ALU instructions that use two registers and an immediate value.



- op: Opcode, operation of the instruction.
- rs: The register source operand.
- rt: The result destination register.
- immediate: Constant second operand for ALU instruction.

Examples:

add immediate:	**addi $1,$2,100**	
and immediate	**andi $1,$2,10**	

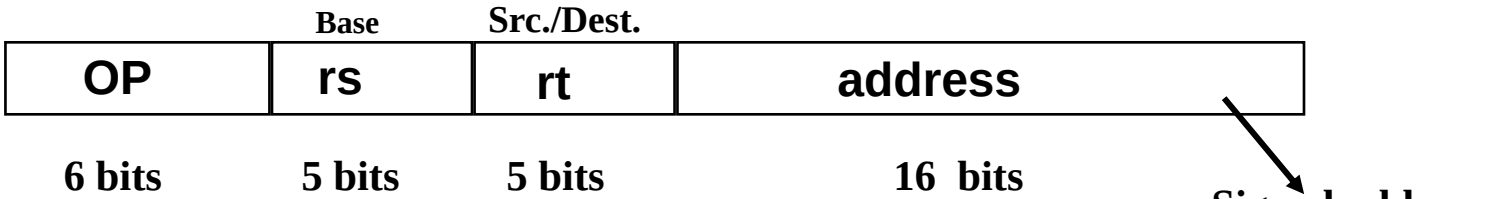
Result register in rt

Source operand register in rs

Constant operand in immediate

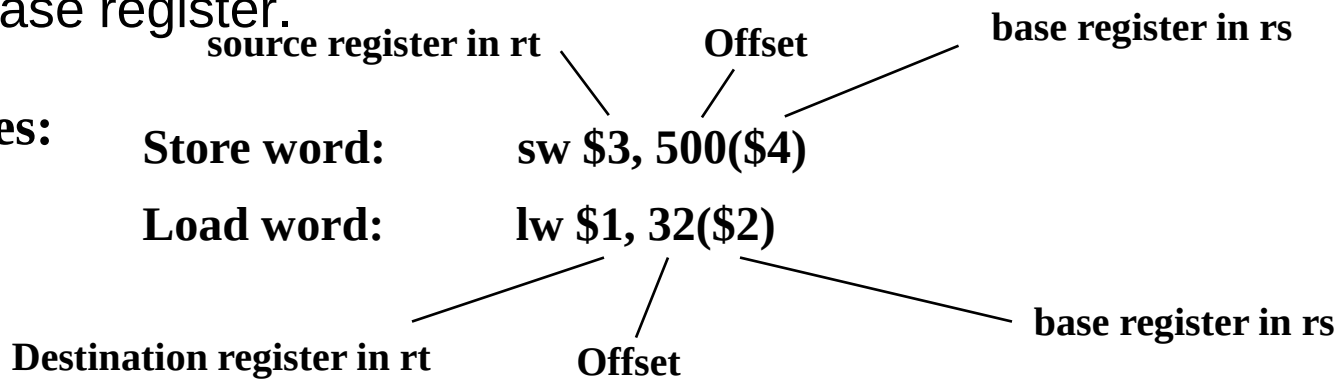
I-Type = Immediate Type
Immediate Addressing used (Mode 2)

MIPS Load/Store I-Type Instruction Fields



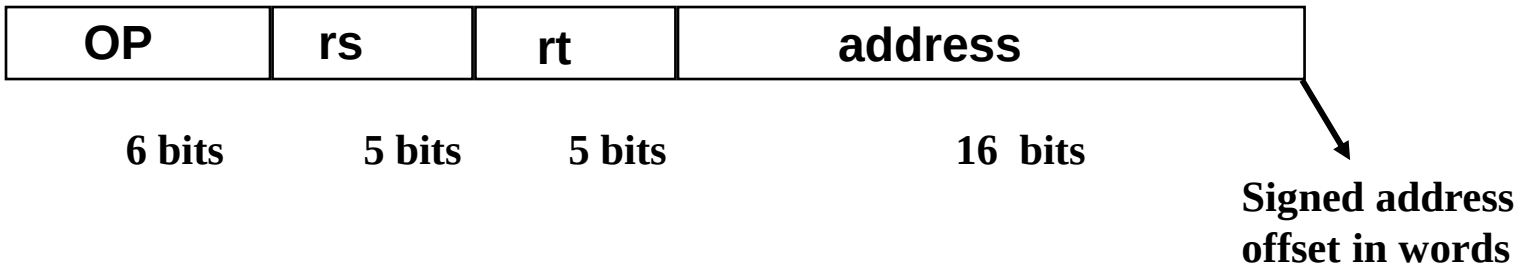
- op: Opcode, operation of the instruction.
  - For load op = 35, for store op = 43.
- rs: The register containing memory base address.
- rt: For loads, the destination register. For stores, the source register of value to be stored.
- address: 16-bit memory address offset in bytes added to base register.

Examples:



Base or Displacement Addressing used (Mode 3)

MIPS Branch I-Type Instruction Fields



- op: Opcode, operation of the instruction.
- rs: The first register being compared
- rt: The second register being compared.
- address: 16-bit memory address branch target offset in words added to PC to form branch address.

Examples:

Branch on equal	**beq $1,$2,100**	**Register in rs** / **Register in rt** / **offset in bytes equal to instruction address field x 4**
Branch on not equal	**bne $1,$2,100**	

Added to PC to form branch target

PC-Relative Addressing used (Mode 4)

MIPS J-Type Instruction Fields

J-Type: Include jump j, jump and link jal

6 bits **26 bits**

Jump target in words

- op: Opcode, operation of the instruction.
  - Jump j op = 2
  - Jump and link jal op = 3
- jump target: jump memory address in words.

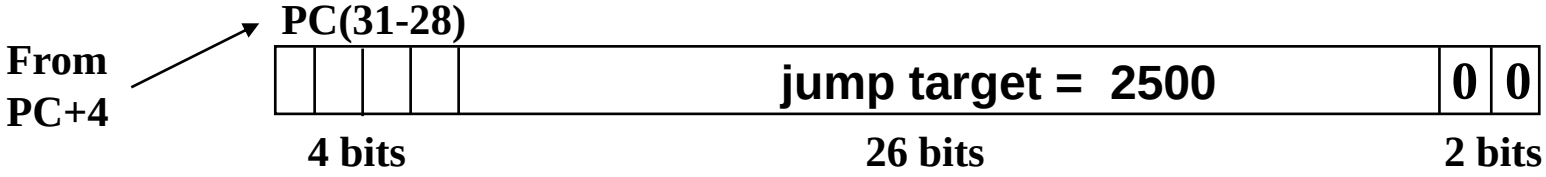
Jump memory address in bytes equal to instruction field jump target x 4

Examples:

Jump **j 10000**

Jump and link **jal 10000**

Effective 32-bit jump address: **PC(31-28),jump_target,00**



J-Type = Jump Type

Pseudodirect Addressing used (Mode 5)