

INSTRUCTIONS: LANGUAGE OF THE COMPUTER

Prof. Sebastian Eslava M.Sc. Ph.D.

iseslavag@unal.edu.co

Universidad Nacional de Colombia

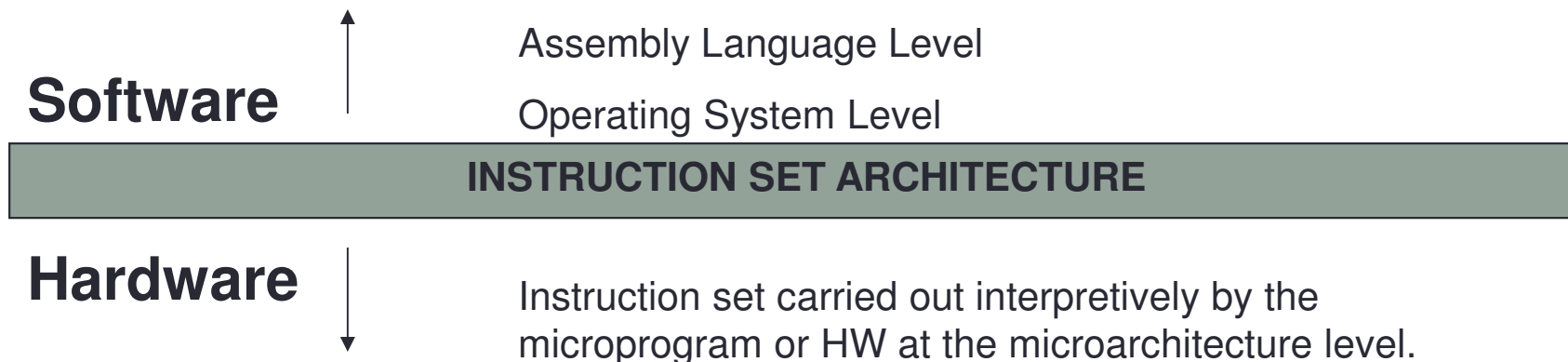
Facultad de Ingeniería

Departamento de Ingeniería Eléctrica y Electrónica

Instruction Set Architecture is ...

“The interface between the software and the hardware”:

View of computer as seen by assembly language programmer or compiler.



- Specifies instructions that the computer can perform and the format for each instruction → **Instruction Set**
 - Examples: MIPS, Intel IA32 (x86), Sun SPARC, PowerPC, IBM 390, Intel IA64

Instruction Set Architecture is ...

Also

- **Data types and data structures:**
 - encodings and representations, type and size of operands
- **Organization of programmable storage**
 - Operand storage in the CPU (Stack structure? Registers?)
- **Modes of addressing & accessing data items and instructions**
 - Effective address and operand location
- **Exceptional conditions**
- An ISA can be implemented in different ways
 - 8086, 386, 486, Pentium, Pentium II, Pentium4 implement same ISA
IA32

Design Decisions for Instruction Sets

- Why is an ISA important?
 - A “good” interface will last through many implementations (portability, compatibility)
 - “Binary compatibility “allows compiled programs to work on different computers → Standardized ISAs
 - x86 in use since the 70’s
 - IBM 390 since the 60’s (started as IBM 360)
 - By 2007, mainstream microprocessor architectures are: IA-32, 64-bit AMD/Intel, Itanium, SPARC and Power/PowerPC

Classifying ISA

- **Based on CPU internal storage options AND Operands.**

Operand Storage in CPU	<i>Where are they other than memory ?</i>
Number of explicit operands named per instruction	<i>How many?</i>
Addressing Modes	<i>How is the effective address for an operand calculated? Can all operands use any mode?</i>
Operations	<i>What are the options for the opcode</i>
Type and Size of operands	<i>How is “typing” done? How is the size specified?</i>

Designing an Instruction Set

- Design Principles
 1. Simplicity favors regularity
 2. Smaller is faster
 3. Make the common case fast
 4. Good design demands good compromises
- MIPS Instruction Set used as example.

RISC Design Principles

- All instructions directly executed by hardware
 - Not interpreted by microinstructions
- Maximize rate at which instructions are issued
 - Minimize execution time
 - Exploit parallelism for better performance
- Instructions should be easy to decode
- Only loads, stores should reference memory
- Provide plenty of registers

Machine Level

- Stored Program Concept
 - Instructions are represented as numbers
 - Programs can be stored in memory just like numbers
- Each assembly instruction is translated into a number (by the assembler). The machine really only understands the “number.”

Instruction characteristics

- Usually a simple operation
 - Identified by the **op-code** field
- But operations require operands – 0, 1, 2
 - To identify where they are, they must be addressed at some piece of storage
 - Typically main memory, registers, or a stack
 - Each has its own particular organization
- 2 options: **implicit** or **explicit** addressing
 - Implicit: the op-code implies the address of the operands
 - Explicit: the address is specified in some field of the instruction

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs
 - See MIPS Reference Data tear-out card, and Appendixes B and E

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination

add a, b, c # a gets b + c
- All arithmetic operations have this form
- *Design Principle 1*: Simplicity favours regularity
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

Register Operand Example

- C code:

`f = (g + h) - (i + j);`

- `f, ..., j` in `$s0, ..., $s4`

- Compiled MIPS code:

`add $t0, $s1, $s2`

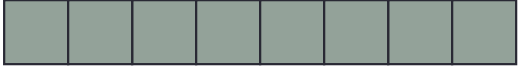
`add $t1, $s3, $s4`

`sub $s0, $t0, $t1`

Types of Operands

- Registers
- Memory
- Constant or immediate

Registers

- The CPU has several registers 
- These registers can hold bytes, half words, words, or double words:



byte (8)



half word (16)



Word (32)



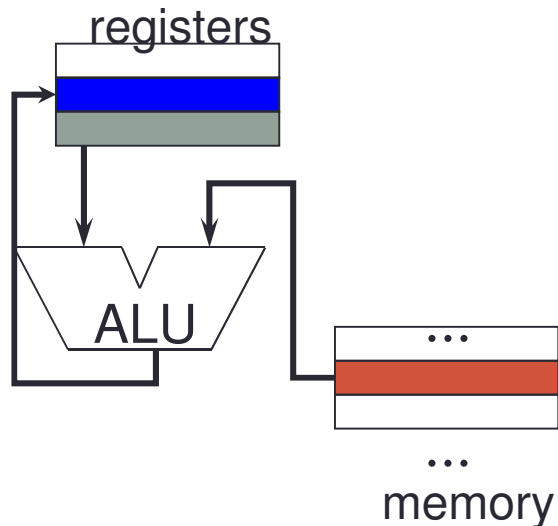
double word (64)

Register ISAs

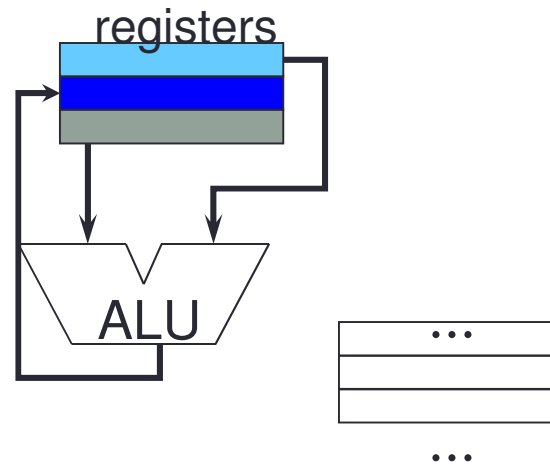
- In register ISAs, operations have only explicit operands from
 - a set of “registers”, i.e., memory cells, are located on the CPU and can hold data
 - or, Memory cells in RAM (or cache), away from the CPU
- Options #1: register-memory ISAs
 - Instructions can have some operands in memory (x86)
 - Output is stored in a register
- Option #2: register-register ISAs
 - Instructions can **only have operands in registers**
 - Output is stored in a register
 - Called **load-store architectures**

Register ISAs

Register-Memory



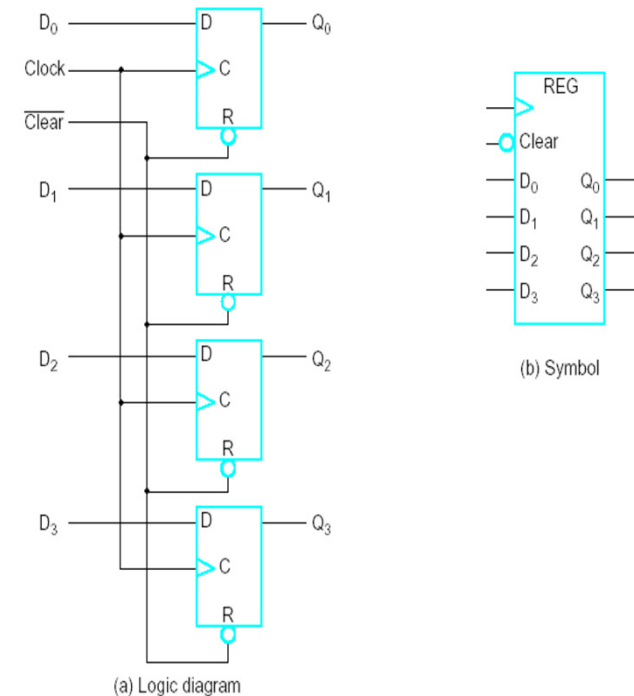
Register-Register



- Both ISAs provide ways to load data from memory into registers and store data from register into memory
- Both are called General Purpose Register (GPR) ISAs
- Memory-memory ISAs are no longer found in computers today
 - the point is that the use of registers is fast

Register ISAs

- Since 1980s, no computer has been designed with a stack or accumulator ISA.
- Reason #1: Registers are faster than memory and have become cheap(er) to build
 - Sequential logic device (implemented using D flip-flops)



A 4-bit register

Register ISAs

- Reason #2: Stack and or accumulator are inefficient because they make the compiler's job difficult
 - Example: $(A*B) - (B*C) - (A*D)$
 - With a stack or accumulator the values of A and B must be loaded from memory multiple time, which reduces performance
 - With a stack or accumulator, there is only one operand evaluation order possible, which may prevent optimizations.
 - Registers can be used to store frequently used variables and thereby reduce memory traffic
 - Registers can be named with fewer bits than a memory cell, which improves density
- Therefore, we are left with only register ISAs

Registers

In load-store architectures:

- Operands must be registers (register-to-register)
- 32-bit words -- each register holds 1 word
- There are 32 (*integer*) registers
- Design Principle 2: *smaller is faster. Why?*
- Compiler must make good use of registers.
- Issues:
 - Number of registers?
 - Storing arrays?

MIPS register

- Register names:

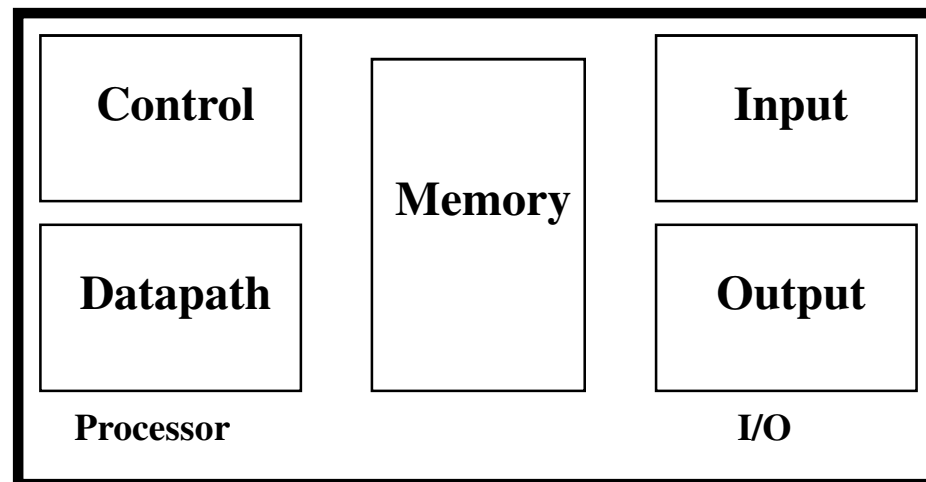
- ◆ \$(letter)(number)
- ◆ \$(2 letter designation)
- ◆ \$zero

Name	Register #	Usage
\$zero	0	constant value 0
\$v0-\$v1	2-3	for results and expr evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Register 1 (\$at) is reserved for the assembler, and registers 26-27 (\$k0-\$k1) are reserved for the operating system.

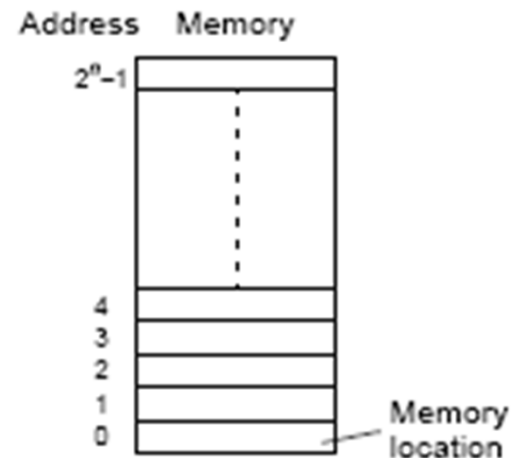
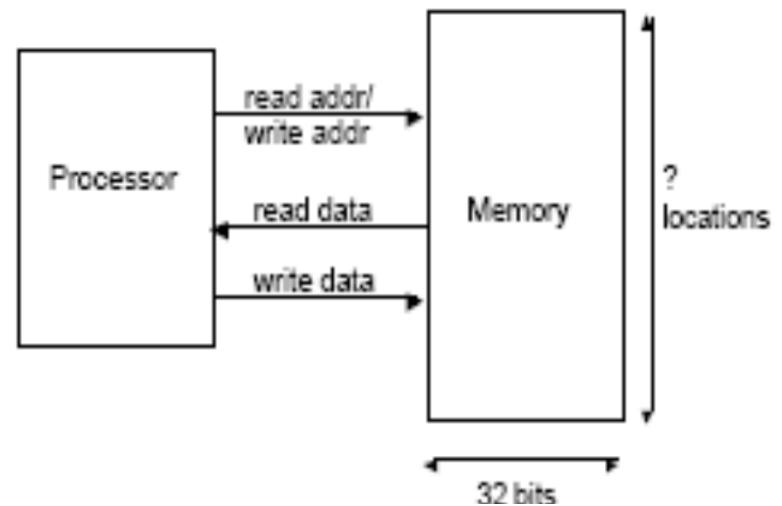
Registers vs. Memory

- Finite number of registers:
 - only 32 registers provided
 - Compiler associates variables with registers
- What about programs with lots of variables?
 - Spill registers to memory when all registers in use
- How about arrays?



Processor-Memory Interconnections

- Memory is viewed as a large, single-dimension array with an address:
 - collection of bits organized in cells
 - Cells are grouped into words
- A memory address is an index into the array:
 - an n -bit address means 2^n addresses (or cells).



Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte
- Words are aligned in memory
 - Address must be a multiple of 4
- MIPS is Big Endian
 - Most-significant byte at least address of a word
 - *c.f.* Little Endian: least-significant byte at least address

Memory Operand Example 1

- C code:

`g = h + A[8];`

- g in \$s1, h in \$s2, base address of A in \$s3

- Compiled MIPS code:

- Index 8 requires offset of 32
 - 4 bytes per word

```
lw    $t0, 32($s3)    # load word
add   $s1, $s2, $t0
```

offset

base register

Memory Operand Example 2

- C code:

`A[12] = h + A[8];`

- `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

- Index 8 requires offset of 32

```
lw    $t0, 32($s3)    # load word
add   $t0, $s2, $t0
sw    $t0, 48($s3)    # store word
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction

- Just use a negative constant

```
addi $s2, $s1, -1
```

- *Design Principle 3: Make the common case fast*
 - Small constants are common
 - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
 - Cannot be overwritten
- Useful for common operations
 - E.g., move between registers
`add $t2, $s1, $zero`

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$
- Example
 - 0000 0000 0000 0000 0000 0000 0000 1011₂
= $0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
= $0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
 - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$

- Example

- $$\begin{aligned} &1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 \\ &= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= -2,147,483,648 + 2,147,483,644 = -4_{10} \end{aligned}$$

- Using 32 bits

- $-2,147,483,648$ to $+2,147,483,647$

2s-Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^n - 1)$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111 \dots 111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
 - $+2 = 0000 \ 0000 \ \dots \ 0010_2$
 - $-2 = 1111 \ 1111 \ \dots \ 1101_2 + 1$
 $= 1111 \ 1111 \ \dots \ 1110_2$

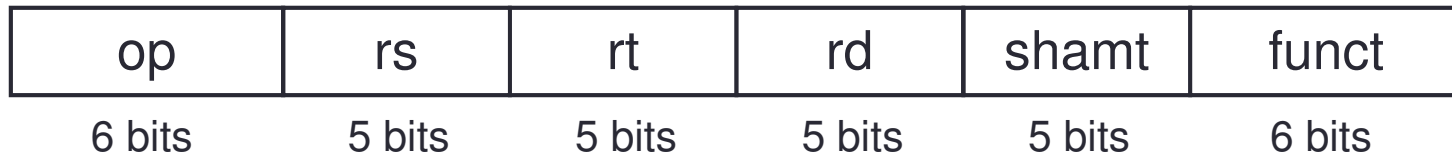
Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- In MIPS instruction set
 - `addi`: extend immediate value
 - `lb`, `lh`: extend loaded byte/halfword
 - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110

Representing Instructions (R, I, J)

- Instructions are encoded in binary
 - Called machine code
- MIPS instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!
- Register numbers
 - \$t0 – \$t7 are reg's 8 – 15
 - \$t8 – \$t9 are reg's 24 – 25
 - \$s0 – \$s7 are reg's 16 – 23

MIPS R-format Instructions



- Instruction fields
 - op: operation code (opcode)
 - rs: first source register number
 - rt: second source register number
 - rd: destination register number
 - shamt: shift amount (00000 for now)
 - funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add \$t0, \$s1, \$s2

special	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

Hexadecimal

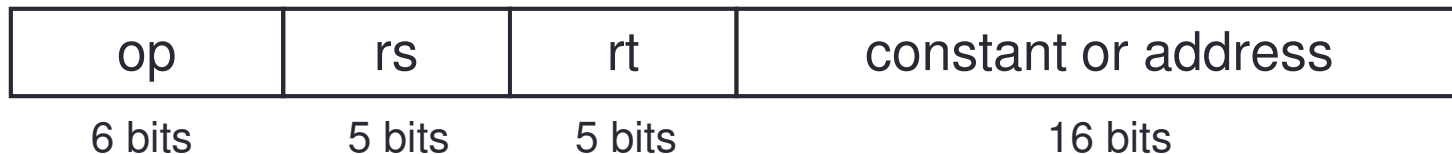
- Base 16
 - Compact representation of bit strings
 - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

■ Example: eca8 6420

■ 1110 1100 1010 1000 0110 0100 0010 0000

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
 - rt: destination or source register number
 - Constant: -2^{15} to $+2^{15} - 1$
 - Address: offset added to base address in rs
- *Design Principle 4: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible