RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

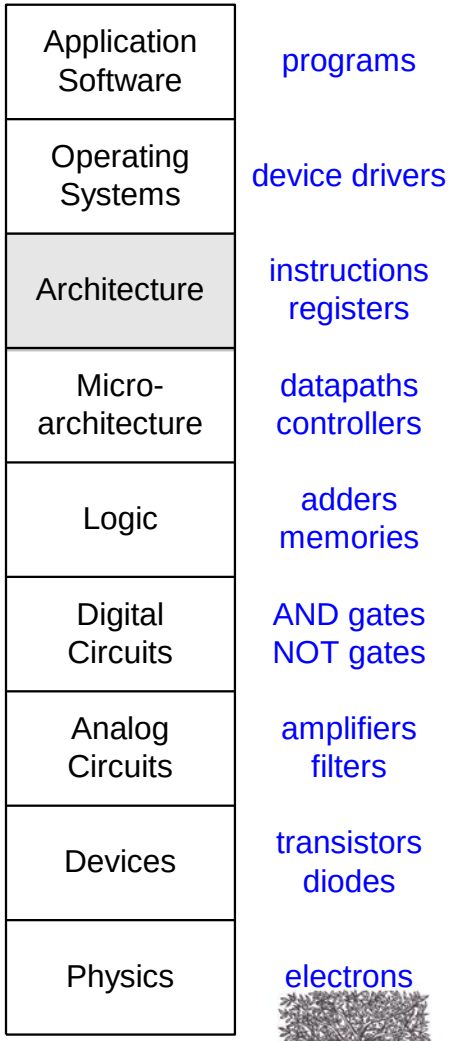
ECE-332:437
DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 8a
Architecture (RISC - MIPS) - Recap

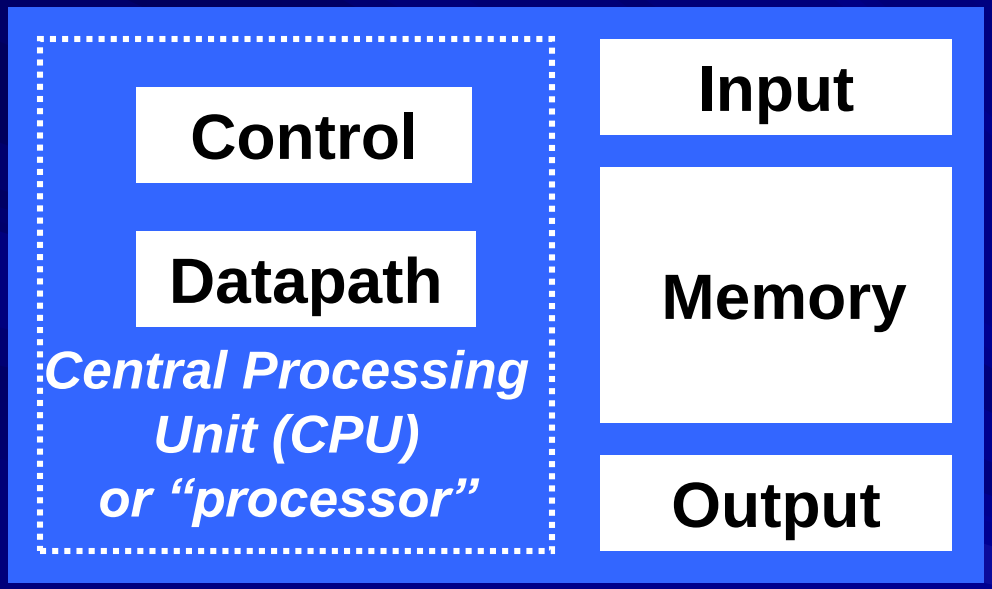
Nagi Naganathan
November 3, 2016

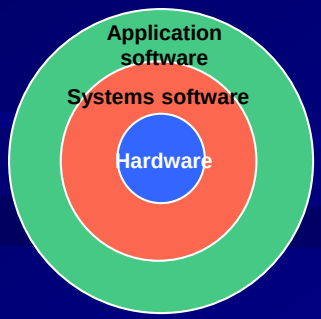
Introduction

- Jumping up a few levels of abstraction
- **Architecture:** programmer's view of computer
  - Defined by instructions & operand locations
- **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)



The Hardware of a Computer





FIVE EASY PIECES

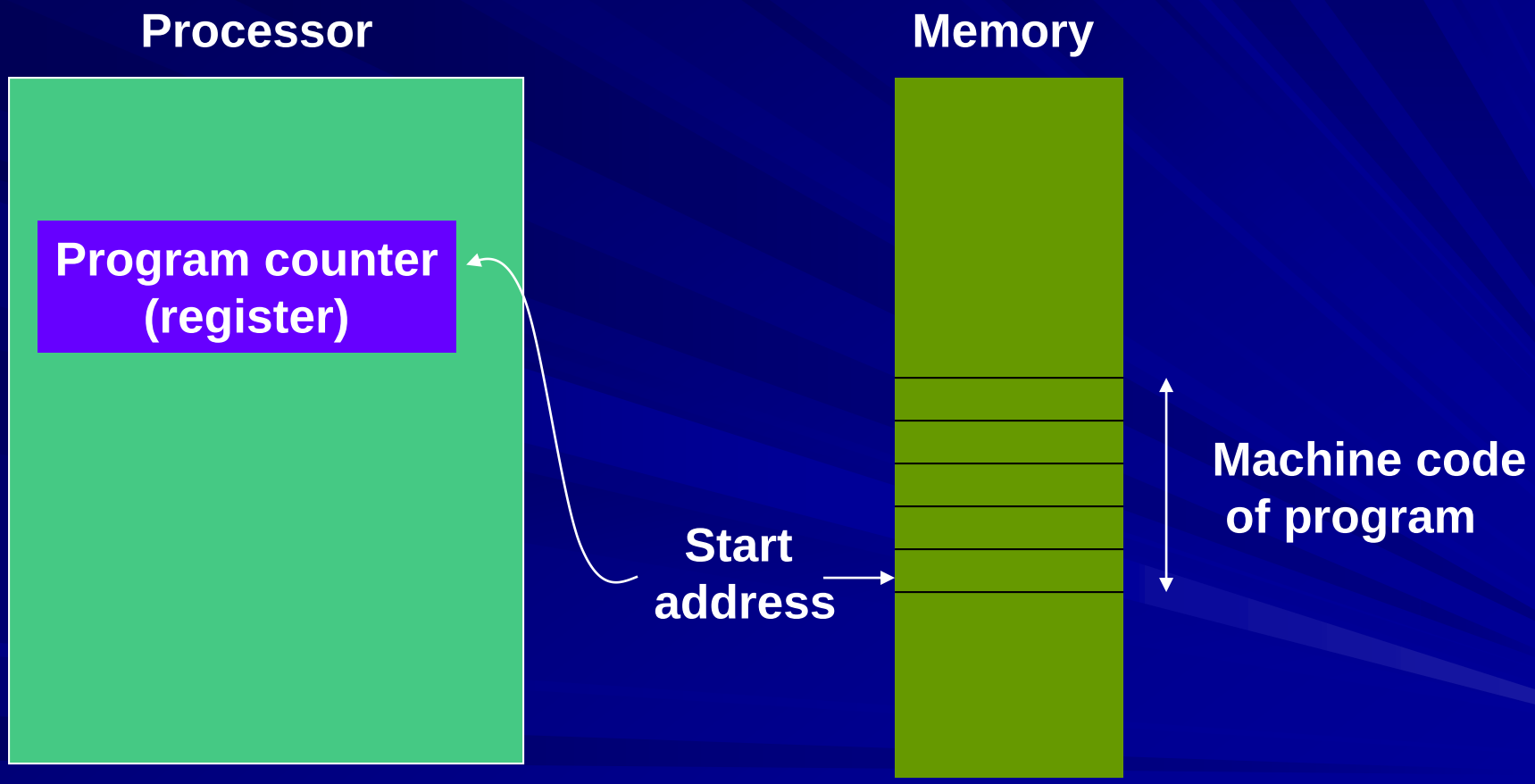
Computer *Architecture*

- Architecture: System attributes that have a direct impact on the logical execution of a program
- Architecture is visible to a programmer:
  - Instruction set
  - Data representation
  - I/O mechanisms
  - Memory addressing

Where Does It All Begin?

- In a register called *program counter (PC)*.
- PC contains the memory address of the next instruction to be executed.
- In the beginning, PC contains the address of the memory location where the program begins.

Where is the Program?



Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- **MIPS** architecture:
  - Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
  - Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco

Once you've learned one architecture, it's easy to learn others

Architecture Design Principles

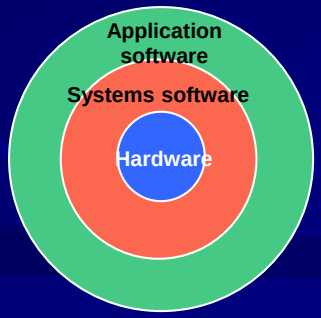
Underlying design principles, as articulated by Hennessy and Patterson:

1. **Simplicity favors regularity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

- Instruction Set Architecture
  - the machine behavior as observable and controllable by the programmer
- Instruction Set
  - the set of commands understood by the computer
- Machine Code
  - a collection of instructions encoded in binary format
  - directly consumable by the hardware
- Assembly Code
  - a collection of instructions expressed in "textual" format e.g. Add r1, r2, r3
  - converted to machine code by an assembler
  - one-to-one correspondence with machine code (mostly true: compound instructions, address labels ....)

Instruction Set Architecture (ISA)

- A set of assembly language instructions (ISA) provides a link between software and hardware.
- Given an instruction set, software programmers and hardware engineers work more or less independently.
- ISA is designed to extract the most performance out of the available hardware technology.



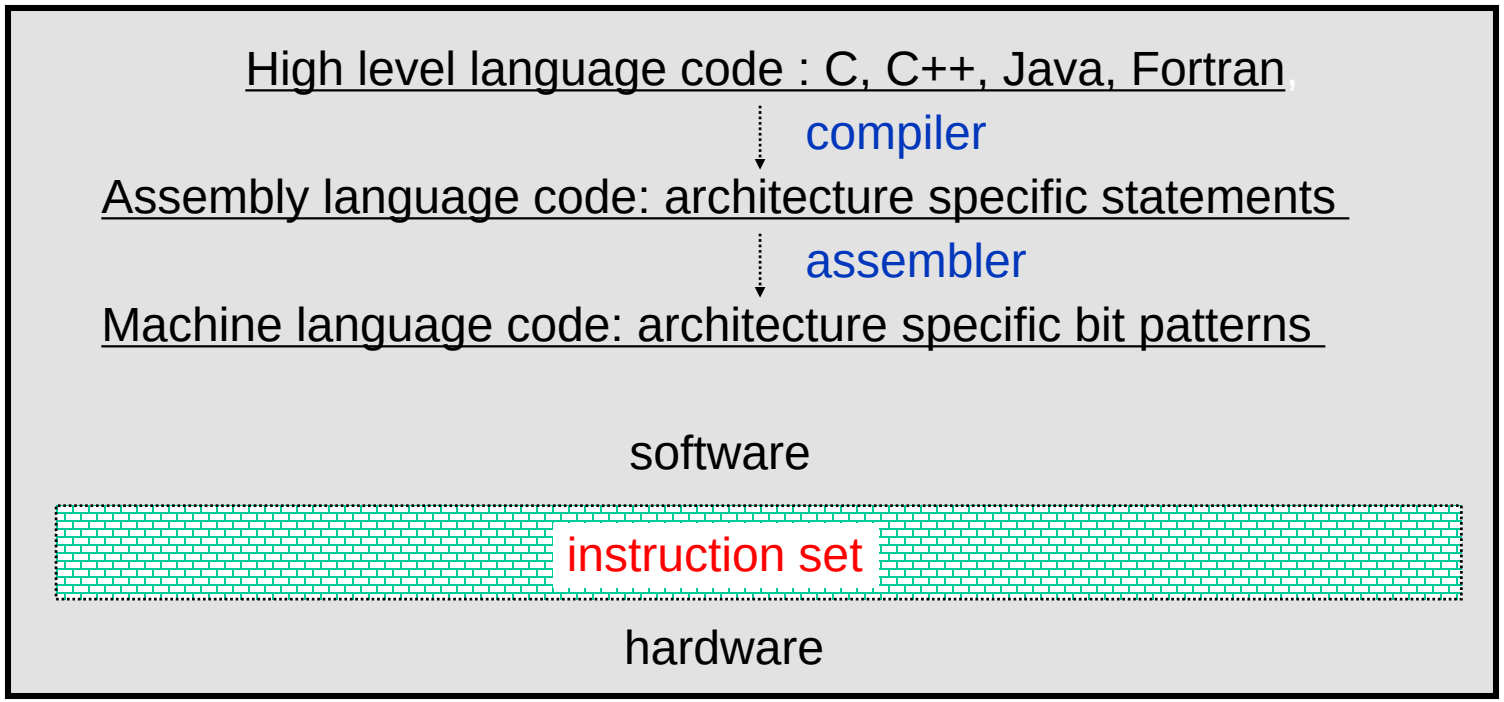
Software

Instruction set

Hardware

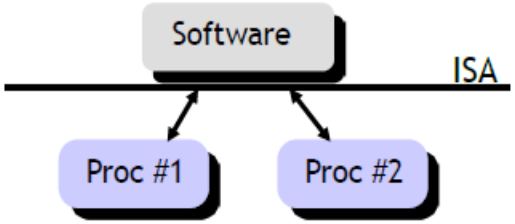
Instruction Set Architecture (ISA)

- Serves as an interface between software and hardware.
- Provides a mechanism by which the software tells the hardware what should be done.



What's an ISA?

- The ISA is the interface between hardware and software.
- The ISA serves as an abstraction layer between the HW and SW
  - Software doesn't need to know how the processor is implemented
  - Any processor that implements the ISA appears equivalent



- An ISA enables processor innovation without changing software
  - This is how Intel has made billions of dollars.
- Before ISAs, software was re-written for each new machine.

ISA

- Defines registers
- Defines data transfer modes between registers, memory and I/O
- Types of ISA: RISC, CISC, VLIW, Superscalar
- Examples:
  - IBM370/X86/Pentium/K6 (CISC)
  - PowerPC (Superscalar)
  - Alpha (Superscalar)
  - MIPS (RISC and Superscalar)
  - Sparc (RISC), UltraSparc (Superscalar)

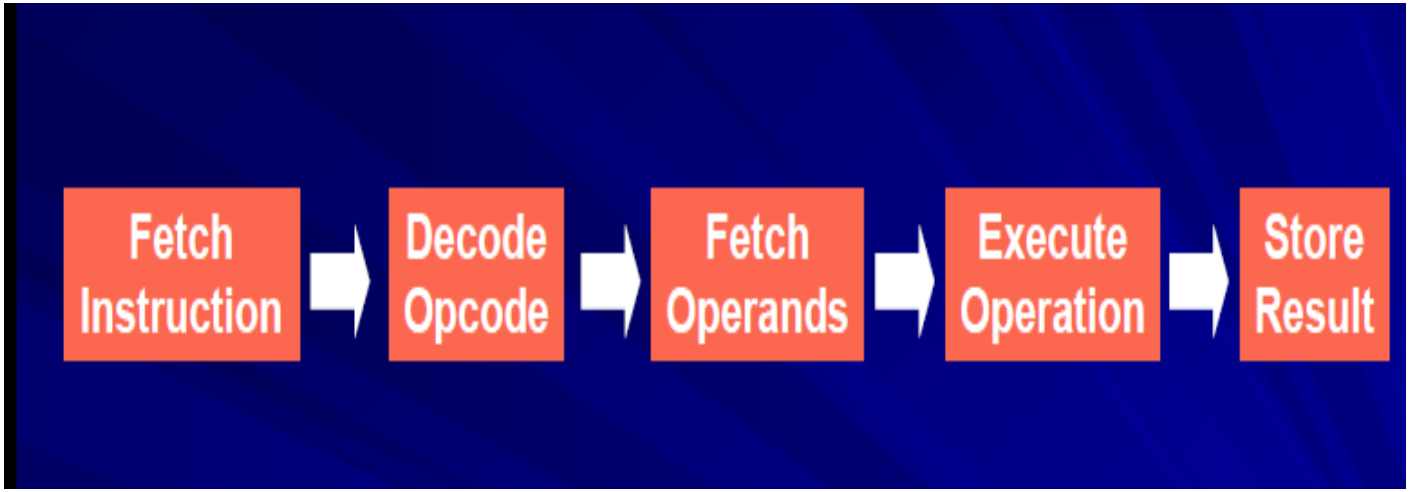
ISA

- An ISA consists of:
  - a set of operations (instructions) ← arithmetic, logical, conditional, branch, etc.
  - data units (sized, addressing modes, etc.) ← 32-bit data word
  - processor state (registers) ← 32, 32-bit registers
  - input and output control (memory operations) ← load and store
  - execution model (program counter) ← 32-bit program counter

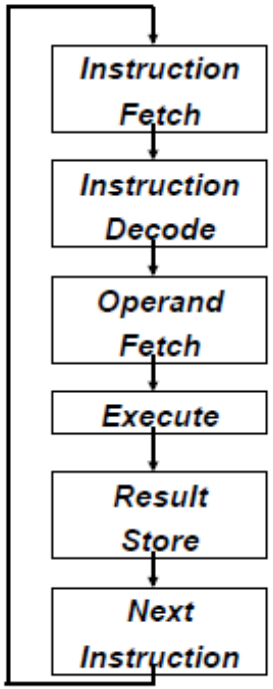
(for MIPS)

MIPS Instruction Set (RISC)

- Instructions execute simple functions.
- Maintain regularity of format – each instruction is one word, contains *opcode* and *arguments*.
- Minimize memory accesses – whenever possible use registers as arguments.
- Three types of instructions:
  - Register (R)-type – only registers as arguments.
  - Immediate (I)-type – arguments are registers and numbers (constants or memory addresses).
  - Jump (J)-type – argument is an address.


Fetch Instruction
Decode Opcode
Fetch Operands
Execute Operation
Store Result

How are Instructions Executed?



- Instruction Fetch:
  Read instruction bits from memory
- Decode:
  Figure out what those bits mean
- Operand Fetch:
  Read registers (+ mem to get sources)
- Execute:
  Do the actual operation (e.g., add the #s)
- Result Store:
  Write result to register or memory
- Next Instruction:
  Figure out mem addr of next insn, repeat

MIPS Arithmetic Instructions

- All instructions have 3 operands
- Operand order is fixed (destination first)

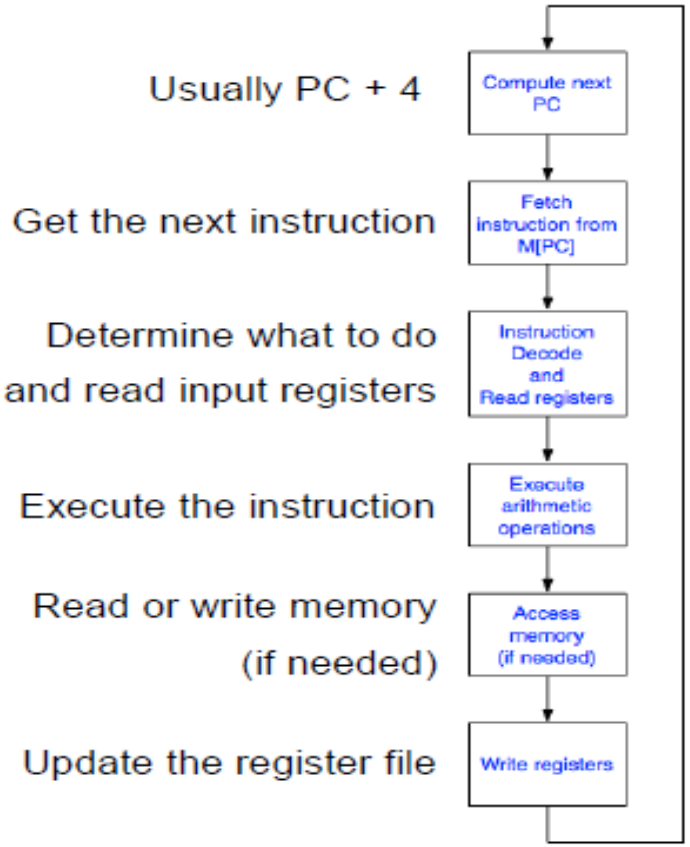
Example:

C code: `a = b + c;`

MIPS 'code': `add a, b, c`

"The natural number of operands for an operation like addition is three… requiring every instruction to have exactly three operands conforms to the philosophy of keeping the hardware simple"

Executing a MIPS program



- All instructions have
  - <= 1 arithmetic op
  - <= 1 memory access
  - <= 2 register reads
  - <= 1 register write
  - <= 1 branch
- All instructions go through all the steps
- As a result
  - Implementing MIPS is (sort of) easy!
  - The resulting HW is (relatively) simple!

MIPS Register Set

Name	Register Number	Usage
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	Function return values
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved variables
`$t8-$t9`	24-25	more temporaries
`$k0-$k1`	26-27	OS temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	Function return address

To summarize:

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = `$s1`	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = `$s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

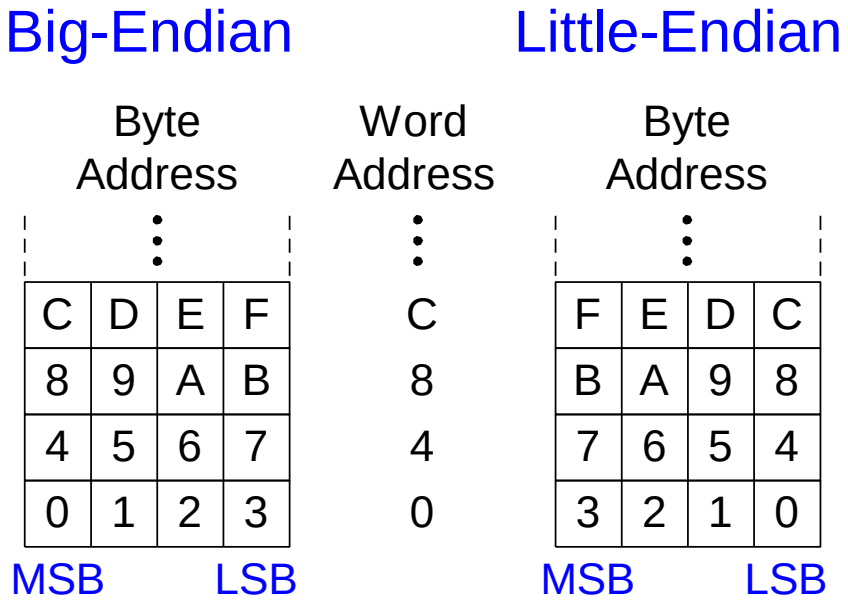
Instruction Format

- **R-Type** : instructions use opcode **000000**.
  - This group Contains all instructions that do not require an immediate value, target offset, memory address displacement, or memory address to specify an operand.
  - Includes arithmetic and logic with all **operands in registers**, shift instructions, and register direct jump instructions (jalr and jr).
- **I-Type** : Opcodes **except 000000, 00001x, and 0100xx**
  - This group includes instructions with **an immediate operand**, branch instructions, and load and store instructions. In the MIPS architecture, all memory accesses are handled by the main processor, so coprocessor load and store instructions are included in this group.
- **J-Type** : instructions use opcode **00001x**.
  - This group consists of the two **direct jump instructions** (j and jal).
  - These instructions require a memory address to specify their operand.

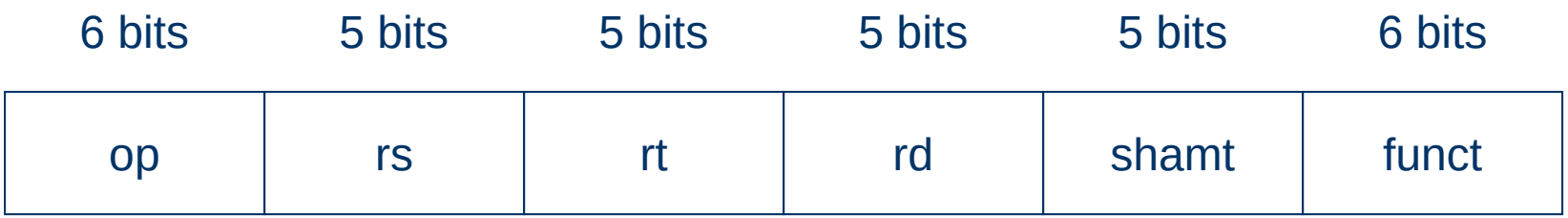
12

Big-Endian & Little-Endian Memory

- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- **Word address** is the **same** for big- or little-endian



R-type instruction format

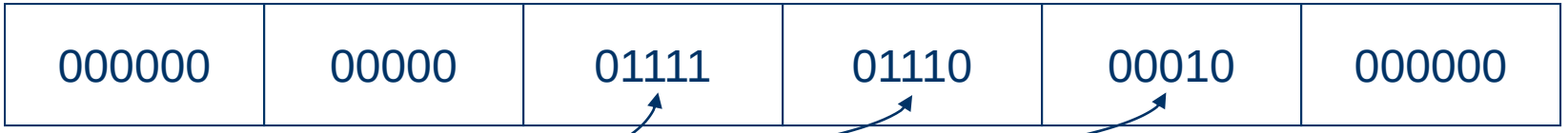


- Op + funct: instruction ID
- rs, rt, rd: register operands
  - 5 bits: long enough for register ID number
  - rs, rt: source registers
  - rd: destination register
- Shamt: shift amount (only used for shifts)
  - 5 bits: shift amount always in range 0..31

R-type examples

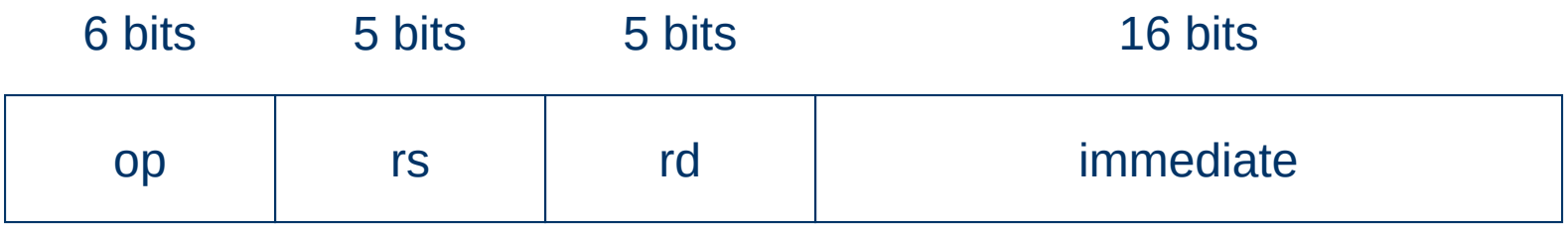
```
add $5, $6, $7
```

op (6) rs (5) rt (5) rd (5) shamt (5) funct (6)




```
sll $t6, $t7, 2    #($t6=$14 $t7=$15)
```

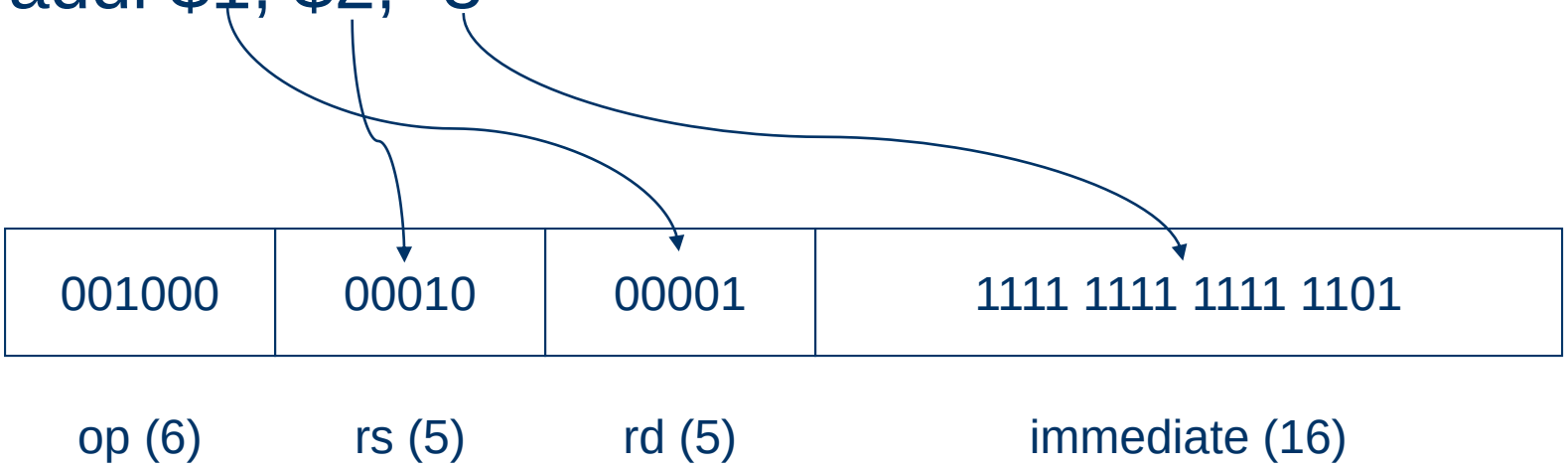
I-type instruction format



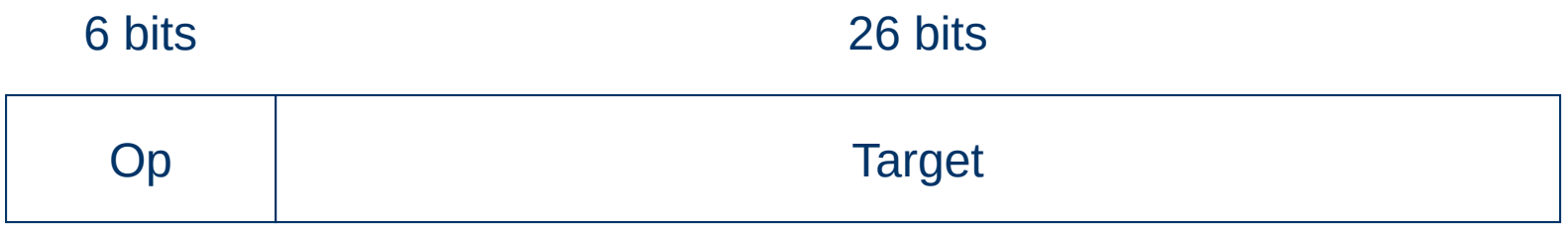
- Op: instruction ID
- rs, rt: register operands
- Immediate: branch target/immediate operand

I-type example

addi $1, $2, -3



J-type instruction format



- Op: instruction ID
- Target: (most of) jump address

J-type examples

```
j bottom        # bottom = 0x00400058
```

0000 0000 0100 0000 0000 0000 0101 1000

Op (6) Target (26)


```
jal f           # f = 0x00400058
```

Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:      ....
```

Control

- MIPS unconditional branch instructions:

```
j  label
```

- Example:

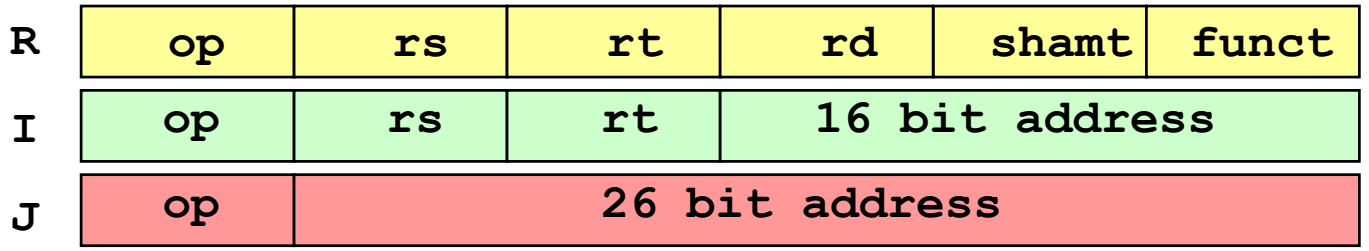
```
if (i!=j)                beq $s4, $s5, Lab1
    h=i+j;               add $s3, $s4, $s5
else                     j Lab2
    h=i-j;          Lab1:sub $s3, $s4, $s5
                    Lab2:...
```

So far (including J-type instr):

- Instruction Meaning

```
add $s1,$s2,$s3   $s1 = $s2 + $s3
sub $s1,$s2,$s3   $s1 = $s2 - $s3
lw $s1,100($s2)   $s1 = Memory[$s2+100]
sw $s1,100($s2)   Memory[$s2+100] = $s1
beq $s4,$s5,L     Next instr. is at Label if $s4 = $s5
j Label           Next instr. is at Label
```

- Formats:



- Write the instruction that loads the word at address 0x00400060 into register $8. Assume that register $10 contains 0x00400000
- lw $8, ?? ,??
- lw $8, 0x60($10)

- Copy 0xFF00AA11 into Big Endian and Little Endian