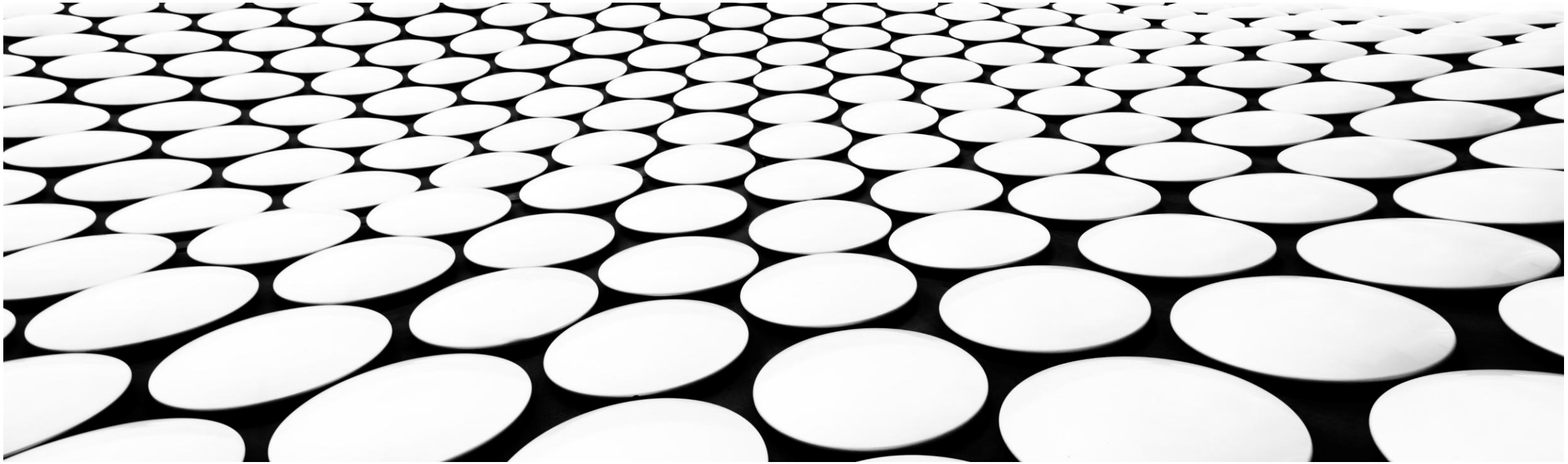
CS5102: FOUNDATIONS OF COMPUTER SYSTEMS

TOPIC-5: **LANGUAGES**

DR. ARIJIT ROY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY PATNA

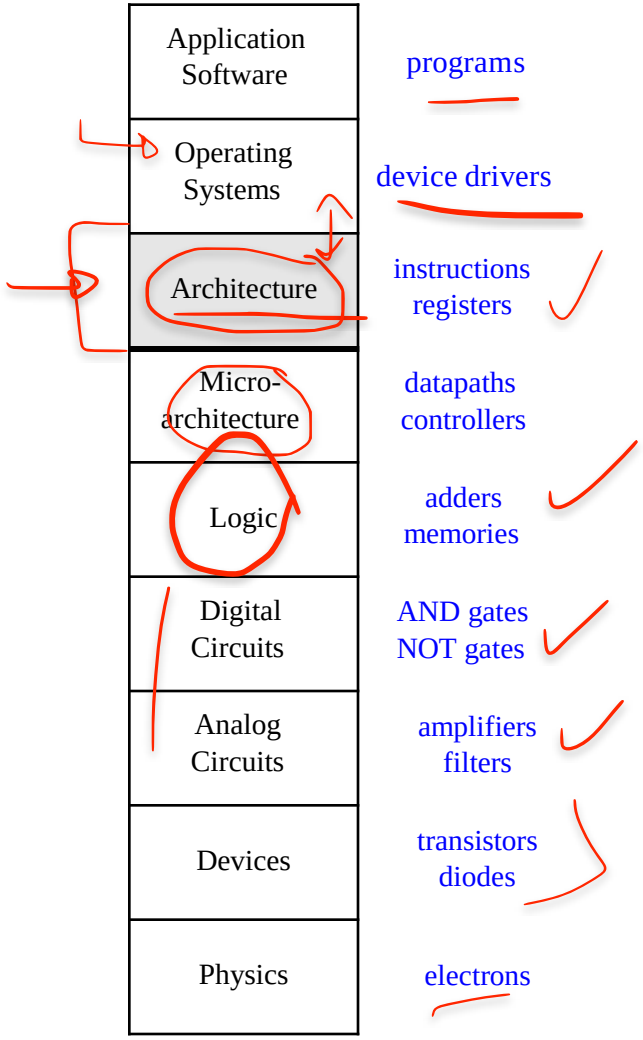


These slides are based on the book: David Money Harris and Sarah L. Harris, Computer Organization and Design

INTRODUCTION

- Jumping up a few levels of abstraction.
- **Architecture:** the programmer's view of the computer

  - Defined by instructions (operations and operand locations
- **Microarchitecture:** how to implement an architecture in hardware



ASSEMBLY LANGUAGE

- To command a computer, you must understand its language.
  - **Instructions:** words in a computer's language
  - **Instruction set:** the vocabulary of a computer's language

- Instructions indicate the operation to perform and the
- operands to use.
  - **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.

*MIPS (Microprocessor without Interlocked Pipelined Stages) is a family of reduced instruction set computer (**RISC**) instruction set architectures (ISA), developed by MIPS Computer Systems, now MIPS Technologies, based in the United States. -- Wiki*

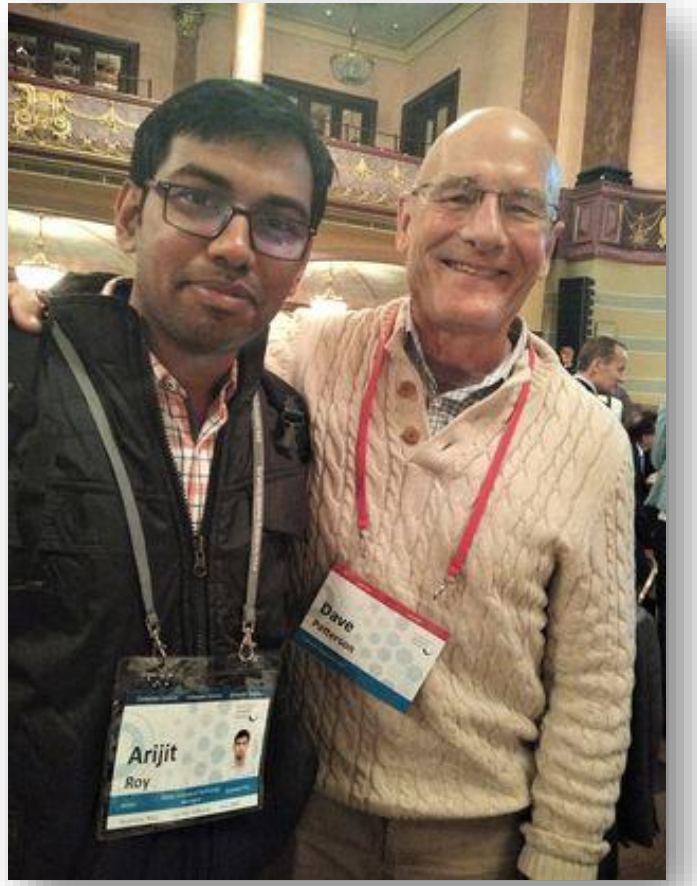
JOHN LEROY HENNESSY

- The 10th President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Coinvented the Reduced Instruction Set Computer (RISC)
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold

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



INSTRUCTIONS: ADDITION

High-level code

```
a = b + c;
```

MIPS assembly code

```
add a, b, c
```

- add: mnemonic indicates what operation to perform
- b, c: source operands on which the operation is performed
- a: destination operand to which the result is written

INSTRUCTIONS: SUBTRACTION

- Subtraction is similar to addition. Only the mnemonic changes.

High-level code

```
a = b - c;
```

MIPS assembly code

```
sub a, b, c
```

- sub: mnemonic indicates what operation to perform
- b, c: source operands on which the operation is performed
- a: destination operand to which the result is written

DESIGN PRINCIPLE 1

Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
  - easier to encode and handle in hardware

INSTRUCTIONS: MORE COMPLEX CODE

- More complex code is handled by multiple MIPS instructions.

High-level code

```
a = b + c - d;
// single line comment
/* multiple line comment */
```

MIPS assembly code

```
add t, b, c   # t = b + c
sub a, t, d   # a = t - d
```

DESIGN PRINCIPLE 2

Make the common case fast

- MIPS includes only simple, commonly used instructions.
- Hardware to decode and execute the instruction can be simple, small, and fast.
- More complex instructions (that are less common) can be performed using multiple simple instructions.
- MIPS is a ***reduced instruction set computer* (RISC)**, with a small number of simple instructions.
- Other architectures, such as Intel's IA-32 found in many PC's, are ***complex instruction set computers* (CISC)**. They include complex instructions that are rarely used, such as the "string move" instruction that copies a string (a series of characters) from one part of memory to another.

OPERANDS

- A computer needs a physical location from which to retrieve binary operands
- A computer retrieves operands from:
  - Registers
  - Memory
  - Constants (also called *immediates*)

OPERANDS: REGISTERS

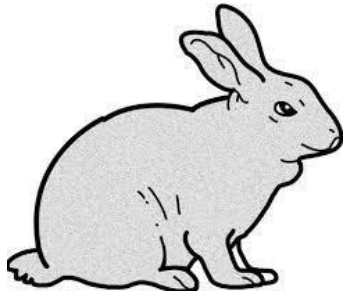
- Memory is slow.
- Most architectures have a small set of (fast) registers.
- MIPS has thirty-two 32-bit registers.
- MIPS is called a 32-bit architecture because it operates on 32-bit data.
  (A 64-bit version of MIPS also exists, but we will consider only the 32-bit version.)

DESIGN PRINCIPLE 3

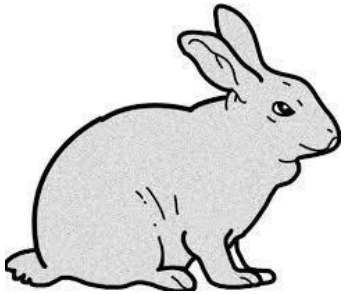
Smaller is Faster

- MIPS includes only a small number of registers
- Just as retrieving data from a few books on your table is faster than sorting through 1000 books, retrieving data from 32 registers is faster than retrieving it from 1000 registers or a large memory.

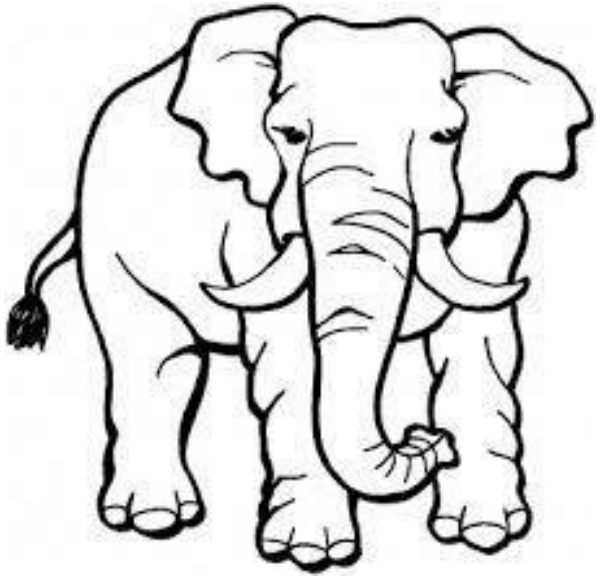
Register:
- *Typically, smallest and fastest memory*
- *Located in the CPU*
- *Stores frequently used data by the CPU*
- *For any processing, a register needs to play its role*



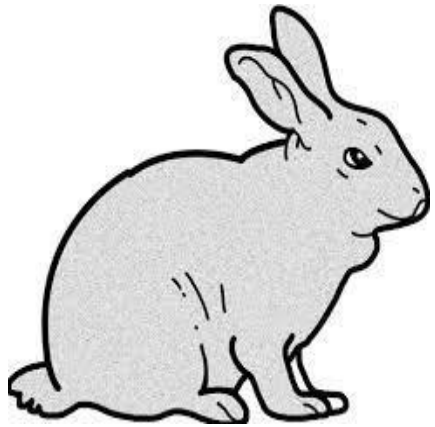
ACTUAL MEMORY SYSTEMS



+



Type1: Fast and Expensive (Small)



Can we achieve?:

Fast, Large

Type2: Slow, Cheap (Large)

(Cache Principle)

THE MIPS REGISTER SET

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	procedure return values
$a0-$a3	4-7	procedure 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	procedure return address

Explanation from Wiki

Always have 0, build-in hardware

Temporary values within pseudo commands

Returning values from functions

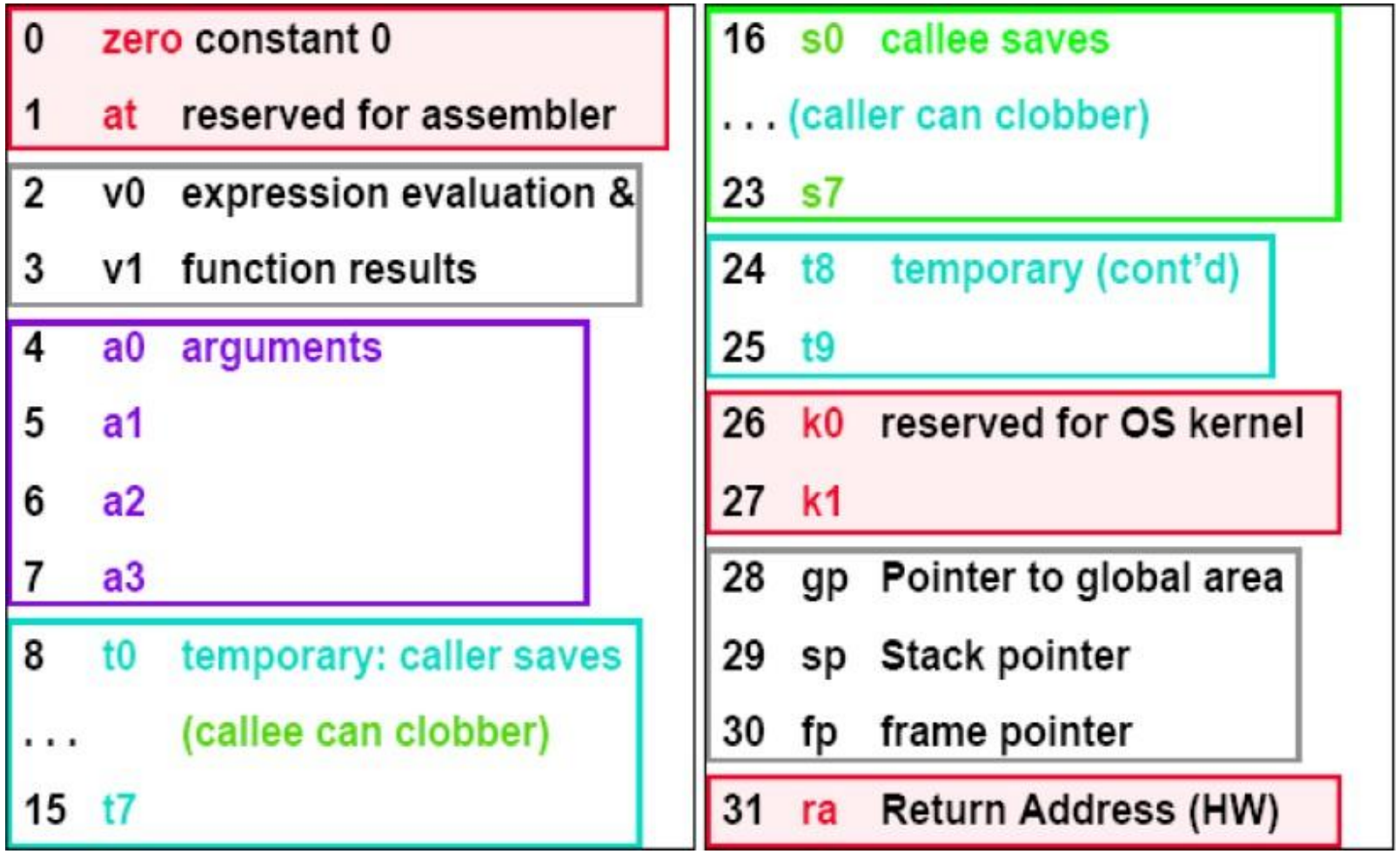
Passing arguments to functions

Store intermediate values

Store longer lasting values

Store intermediate values

MIPS SOFTWARE CONVENTIONS FOR REGISTERS



Callee saved registers: whose values the called method must save and restore (if it uses them).
Caller saved registers whose value the calling method must save and restore (if it depends on them after the call).

OPERANDS: REGISTERS

- Registers:
  - Written with a dollar sign ($) before their name
  - For example, register 0 is written “$0”, pronounced “register zero” or “dollar zero”.
- Certain registers used for specific purposes:
  - For example, $0 always holds the constant value 0.
- the saved registers, $s0-$s7, are used to hold variables
- the temporary registers, $t0 - $t9, are used to hold intermediate values during a larger computation.
- For now, we only use the temporary registers ($t0 - $t9) and the saved registers ($s0 - $s7).
- We will use the other registers in later slides.

INSTRUCTIONS WITH REGISTERS

- Revisit add instruction

High-level code

```
a = b + c
```

MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
```

OPERANDS: MEMORY

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, so it can hold a lot of data
- But it’s also slow
- Commonly used variables kept in registers
- Using a combination of registers and memory, a program can access a large amount of data fairly quickly

DESIGN PRINCIPLE 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `add`, `sub`: use 3 register operands
  - `lw`, `sw`: use 2 register operands and a constant
- Number of instruction formats kept small
- to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

OPERANDS: CONSTANTS/IMMEDIATES

- `lw` and `sw` illustrate the use of constants or *immediates*
- Called immediates because they are ***immediately*** available from the instruction
- Immediates don't require a register or memory access.
- The add immediate (`addi`) instruction adds an immediate to a variable (held in a register).
- An immediate is a 16-bit two's complement number.
- Is subtract immediate (`subi`) necessary?

High-level code

```
a = a + 4;
b = a - 12;
```

MIPS assembly code

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```

MACHINE LANGUAGE

- Computers only understand 1's and 0's
- Machine language: binary representation of instructions
- 32-bit instructions
  - Again, simplicity favors regularity: 32-bit data and instructions
- Three instruction formats:
  - R-Type: register operands
  - I-Type: immediate operand
  - J-Type: for jumping

R-TYPE

- *Register-type*
- 3 register operands:
  - `rs`, `rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode* (0 for R-type instructions)
  - `funct`: the *function*
- together, the opcode and function tell the computer what operation to perform
  - `shamt`: the *shift amount* for shift instructions, otherwise it's 0

- *The operand for R-type instruction is 000000*
- *If there is no shift value, shamt is 00000*

R-Type

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

R-TYPE EXAMPLES

Assembly Code

```
add $s0, $s1, $s2

sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

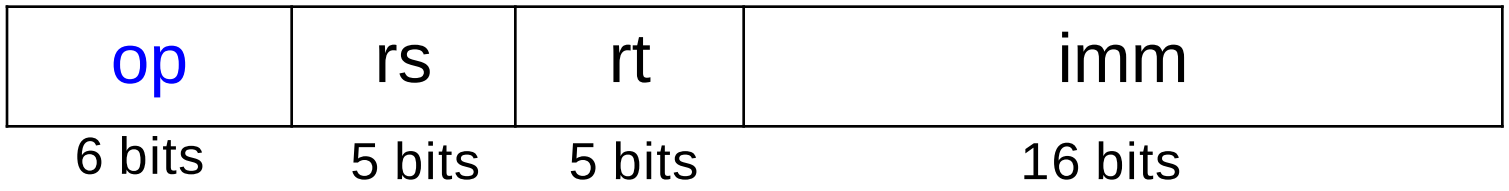
Note the order of registers in the assembly code:

```
add rd, rs, rt
```

I-TYPE

- *Immediate-type*
- 3 operands:
  - `rs`, `rt`: register operands
  - `imm`: 16-bit two's complement immediate
- Other fields:
  - `op`: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by the opcode

I-Type



I-TYPE EXAMPLES

Data inside the memory address 32($0) into register $t2

Assembly Code	op	rs	rt	imm
`addi $s0, $s1, 5`	8	17	16	5
`addi $t0, $s3, -12`	8	19	8	-12
`lw $t2, 32($0)`	35	0	10	32
`sw $s1, 4($t1)`	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

Field Values

Note the differing order of registers in the assembly and machine codes:

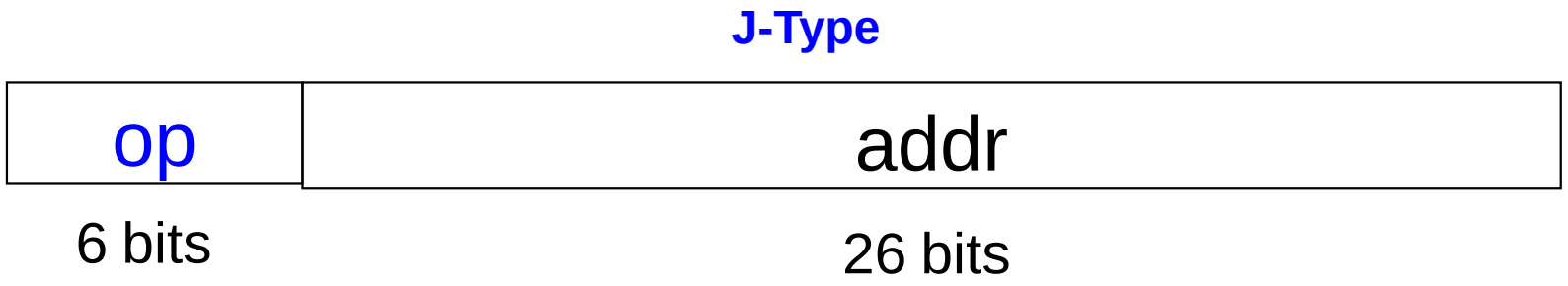
```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

MACHINE LANGUAGE: J-TYPE

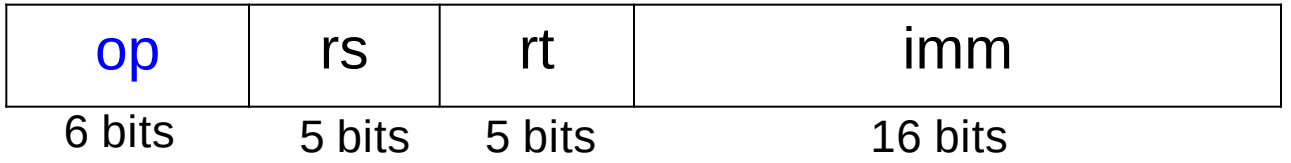
- *Jump-type*
- 26-bit address operand (`addr`)
- Used for jump instructions (`j`)



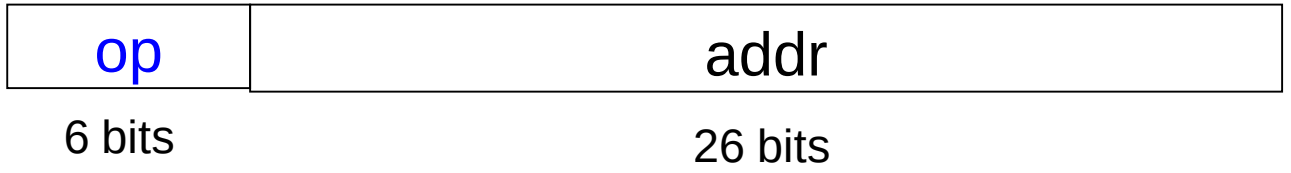
REVIEW: INSTRUCTION FORMATS

R-Type

I-Type

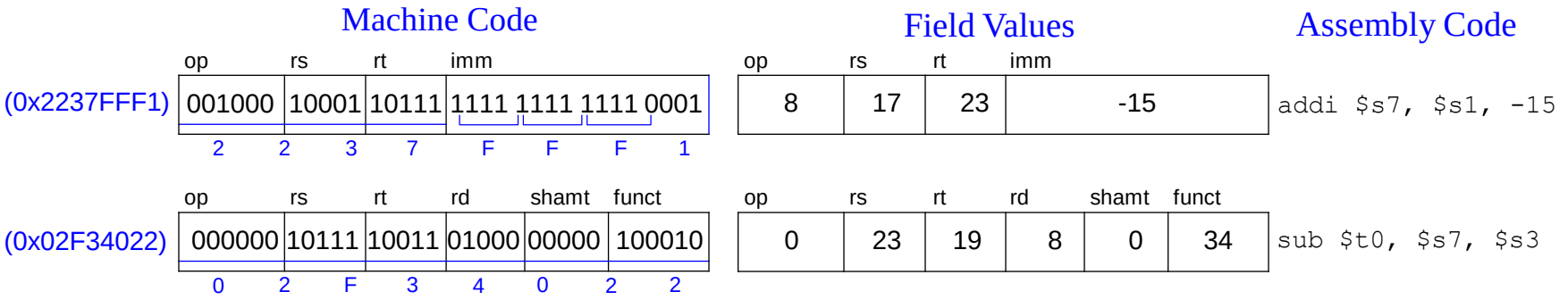


J-Type



INTERPRETING MACHINE LANGUAGE CODE

- Start with opcode
- Opcode tells how to parse the remaining bits
- If opcode is all 0's
  - R-type instruction
  - Function bits tell what instruction it is
- Otherwise
  - opcode tells what instruction it is



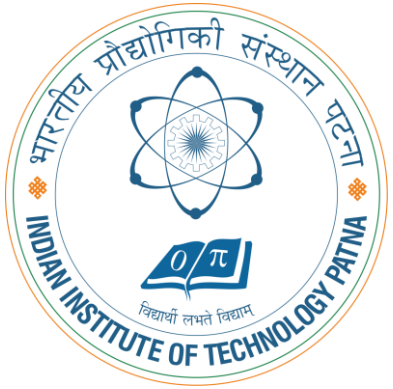
MIPS ASSEMBLY

C code: `A[8] = h + A[8];`

MIPS code:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)
```

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`


भारतीय प्रौद्योगिकी संस्थान पटना
INDIAN INSTITUTE OF TECHNOLOGY PATNA
विद्यार्थी लभते विद्याम्

THANK YOU!