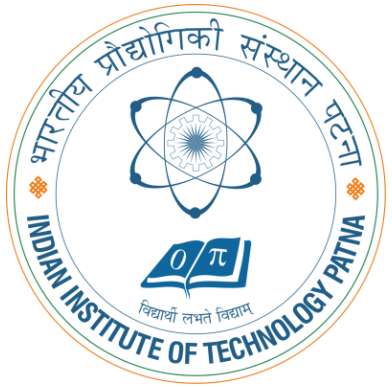
CS5102: FOUNDATIONS OF COMPUTER SYSTEMS



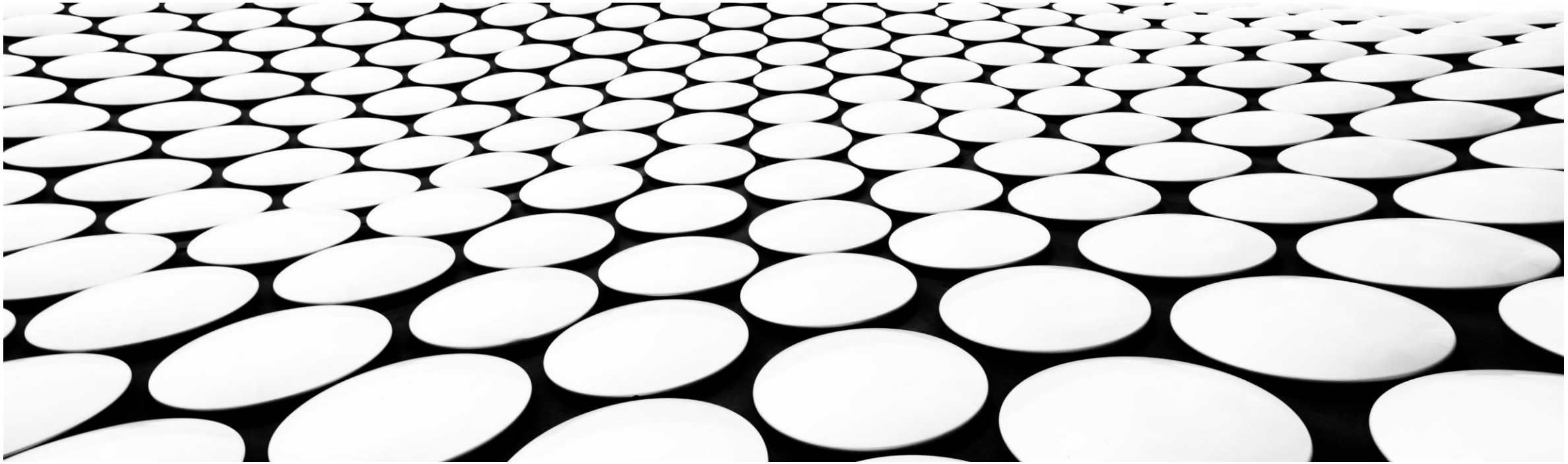
TOPIC-6: **PROGRAMMING-I**

a = b + c;
add $s0, $s1 $s2

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

RECAP

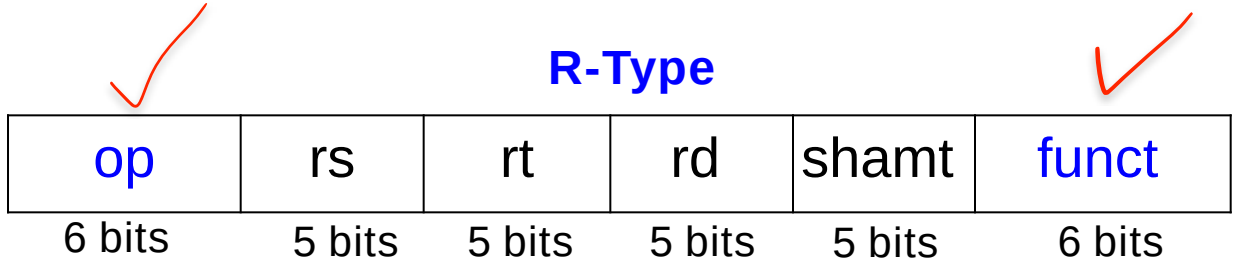
MACHINE LANGUAGE

- 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 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	

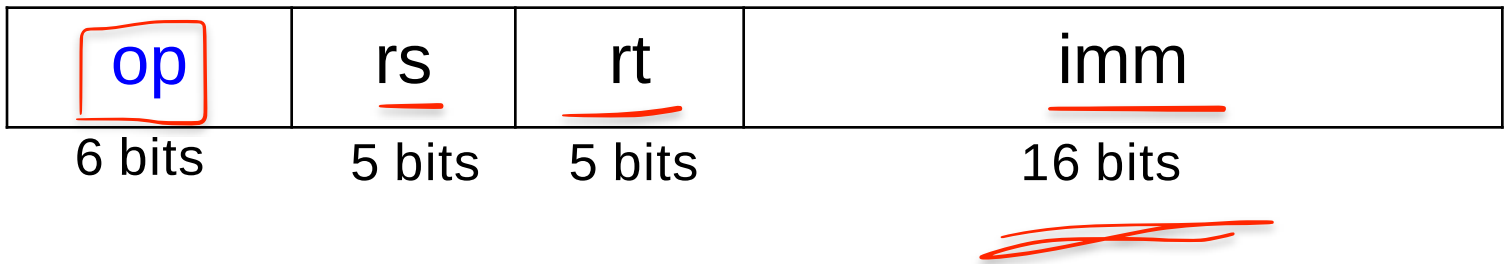
Note the order of registers in the assembly code:

```
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

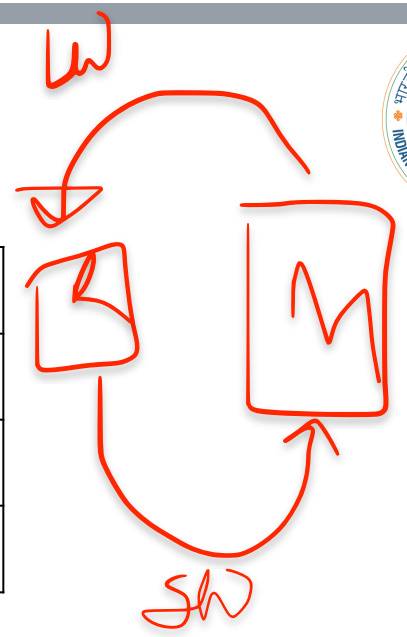
Data inside the memory address 32($0) into register $t2

Assembly Code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1, 4($t1)
```

Field Values

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits



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

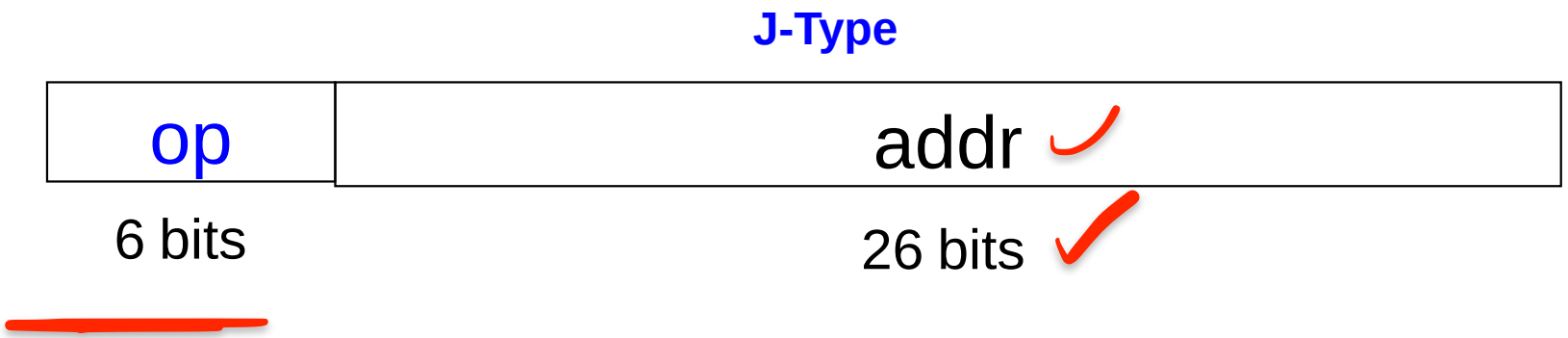
```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

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

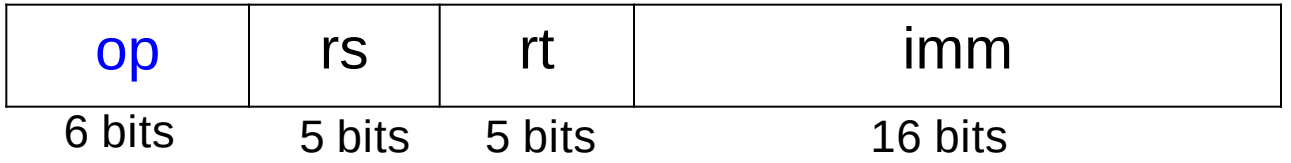
- *Jump-type*
- 26-bit address operand (`addr`)
- Used for jump instructions (`j`)



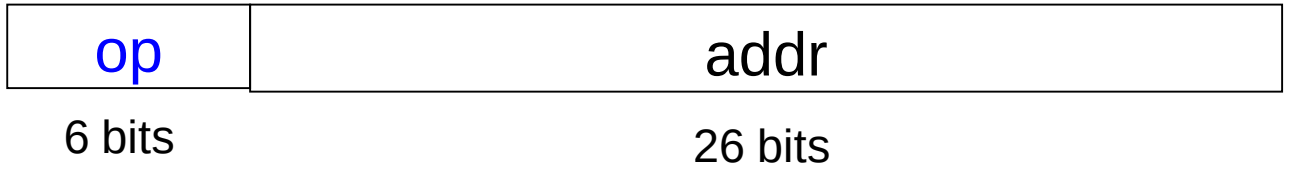
REVIEW: INSTRUCTION FORMATS

R-Type

I-Type

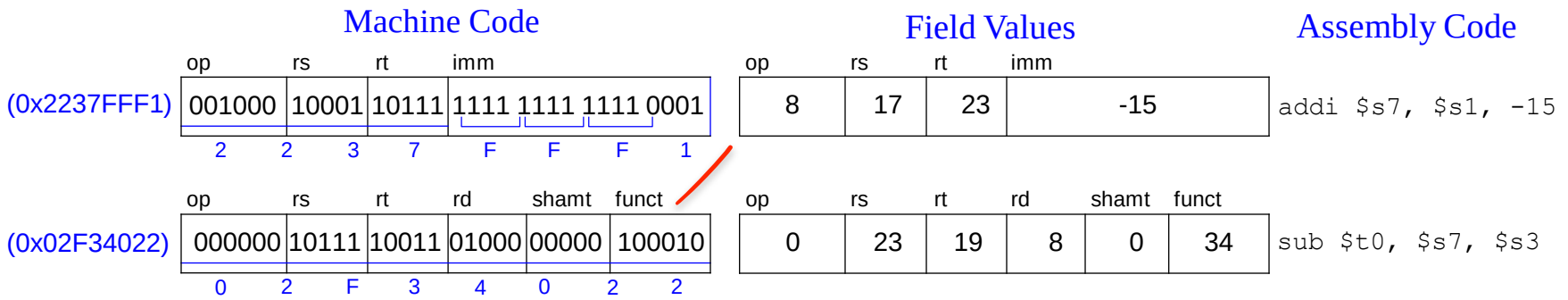


J-Type

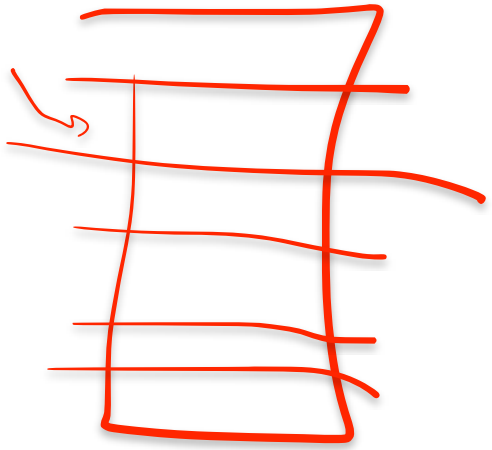


INTERPRETING MACHINE LANGUAGE CODE

- 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`

PROGRAMMING

PROGRAMMING

- High-level languages:
  - e.g., C, Java, Python
  - Written at more abstract level
- Common high-level software constructs:
  - if/else statements
  - for loops
  - while loops
  - array accesses
  - procedure calls
- Other useful instructions:
  - Arithmetic/logical instructions
  - Branching

LOGICAL INSTRUCTIONS

- `and`, `or`, `xor`, `nor`

- `and`: useful for *masking* bits
  - Masking all but the least significant byte of a value: 0xF234012F AND 0x000000FF = 0x0000002F
- `or`: useful for combining bit fields
  - Combine 0xF2340000 with 0x000012BC: 0xF2340000 OR 0x000012BC = 0xF23412BC
- `nor`: useful for inverting bits: A NOR $0 = NOT A
- `andi`, `ori`, `xori`
  - 16-bit immediate is zero-extended (*not* sign-extended)
  - `nori` not needed

LOGICAL INSTRUCTION EXAMPLES

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

Result

`and $s3, $s1, $s2`	$s3								
`or $s4, $s1, $s2`	$s4								
`xor $s5, $s1, $s2`	$s5								
`nor $s6, $s1, $s2`	$s6								

LOGICAL INSTRUCTION EXAMPLES

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2
or  $s4,  $s1,  $s2
xor  $s5,  $s1,  $s2
nor $s6, $s1, $s2
```

Result

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

LOGICAL INSTRUCTION EXAMPLES

Source Values

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

zero-extended

Assembly Code

```
andi $s2, $s1, 0xFA34
ori $s3, $s1, 0xFA34
xori $s4, $s1, 0xFA34
```

Result

$s2								
$s3								
$s4								

LOGICAL INSTRUCTION EXAMPLES

Source Values

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

zero-extended

Assembly Code

```
andi $s2, $s1, 0xFA34
ori $s3, $s1, 0xFA34
xori $s4, $s1, 0xFA34
```

Result

$s2	0000	0000	0000	0000	0000	0000	0011	0100
$s3	0000	0000	0000	0000	1111	1010	1111	1111
$s4	0000	0000	0000	0000	1111	1010	1100	1011

SHIFT INSTRUCTIONS

- `sll`: shift left logical
  - **Example:** `sll $t0, $t1, 5 #$t0 <=$t1 << 5`
- `srl`: shift right logical
  - **Example:** `srl $t0, $t1, 5 #$t0 <=$t1 >> 5`

- `sra`: shift right arithmetic
  - **Example:** `sra $t0, $t1, 5 #$t0 <=$t1 >>> 5`

SHIFT INSTRUCTIONS

Assembly Code

```
sll $t0, $s1, 2
srl $s2, $s1, 2
sra $s3, $s1, 2
```

Field Values

op	rs	rt	rd	shamt	funct
0	0	17	8	2	0
0	0	17	18	2	2
0	0	17	19	2	3
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

GENERATING CONSTANTS

- 16-bit constants using `addi`:

High-level code

```
// int is a 32-bit signed word
   int a = 0x4f3c;
```

MIPS assembly code

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- 32-bit constants using load upper immediate (`lui`) and `ori`:

  (`lui` loads the 16-bit immediate into the upper half of the register and sets the lower half to 0.)

```
lui $s0, 10101010 10101010 11110000 11110000

s0 = 10101010 10101010 00000000 00000000

ori $s0, 11110000 11110000

s0 = 10101010 10101010 11110000 11110000
```

High-level code

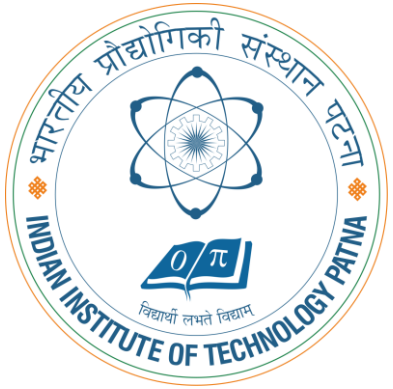
```
int a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

MULTIPLICATION, DIVISION

- Special registers: `lo`, `hi`
- 32 × 32 multiplication, 64 bit result
  - `mult $s0, $s1`
  - Result in {`hi`, `lo`}
- 32-bit division, 32-bit quotient, 32-bit remainder
  - `div $s0, $s1`
  - Quotient in `lo`
  - Remainder in `hi`
- Moves from lo/hi special registers
  - `mflo $s2`
  - `mfhi $s3`


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

THANK YOU!