
# RISC V Architecture

**Mahesh Awati**
Department of Electronics and Communication Engg.

# RISC V ARCHITECTURE

## UNIT 2 – Instructions: The Language of Computer

**Mahesh Awati**
Department of Electronics and Communication Engineering

# Instructions – Language of Computer

## Supporting Procedures in Computer Hardware

RISC-V features for handling Procedures

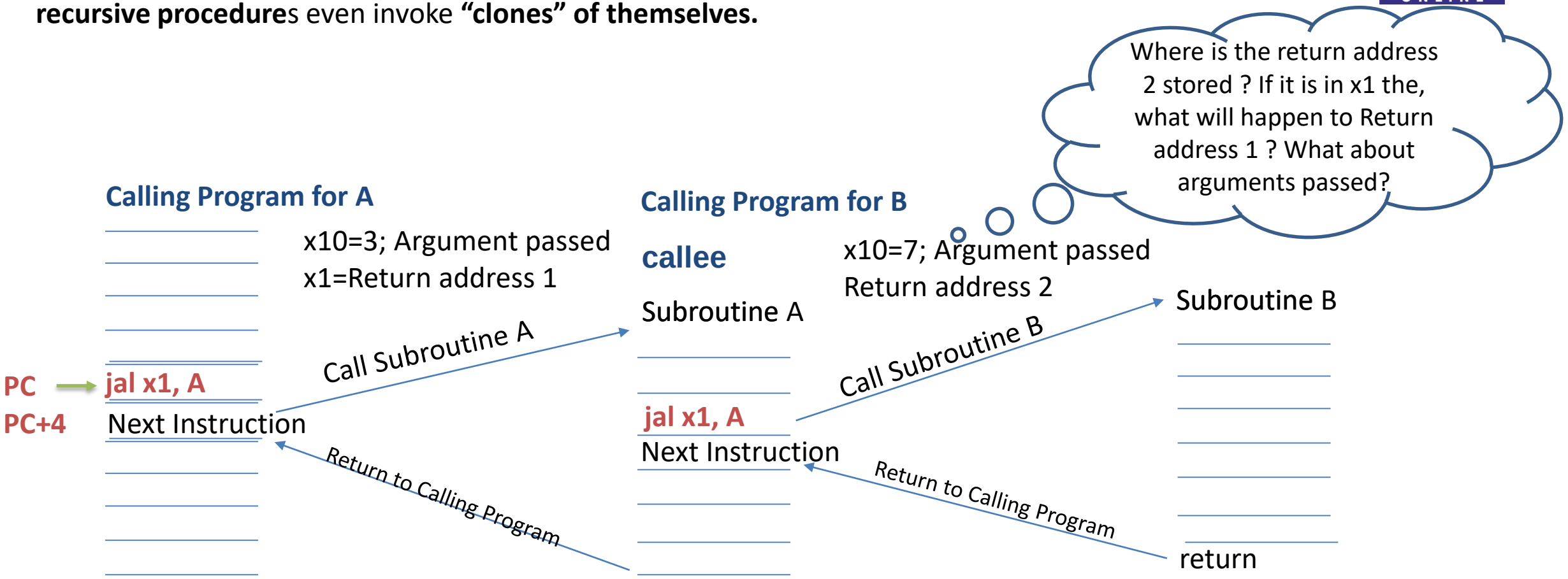
**Case b) Nested Procedures**: Procedures that invoke other procedures. Moreover, **recursive procedure**s even invoke **“clones” of themselves.**

# Instructions – Language of Computer

## Supporting Procedures in Computer Hardware

RISC-V features for handling Procedures

**Case b) Nested Procedures**: Procedures that invoke other procedures. Moreover, **recursive procedure**s even invoke **"clones" of themselves.**



# Instructions – Language of Computer

## Supporting Procedures in Computer Hardware

RISC-V features for handling Procedures

**Case b) Nested Procedures**: Procedures that invoke other procedures. Moreover, **recursive procedure**s even invoke **"clones" of themselves.**

### Steps to prevent the problem

- ✓ The caller pushes any argument registers (x10–x17) or temporary registers (x5-x7 and x28-x31) that are needed after the call.
- ✓ The callee pushes the return address register x1 and any saved registers (x8- x9 and x18-x27) used by the callee.
- ✓ The stack pointer sp is adjusted to account for the number of registers placed on the stack. Upon the return, the registers are restored from memory, and the stack pointer is readjusted.

## Nested Procedures

Compiling a Recursive C Procedure, Showing Nested Procedure Linking
Ex: Let's tackle a recursive procedure that calculates factorial:

```
int fact (int n)
{
if (n < 1) return (1);
else return (n * fact(n − 1));
}
```

What is the RISC-V assembly code?

# Instructions – Language of Computer

## Supporting Procedures in Computer Hardware

**What is and what is not preserved across a procedure call**

| Preserved | Not preserved |
|---|---|
| Saved registers: x8-x9, x18-x27 | Temporary registers: x5-x7, x28-x31 |
| Stack pointer register: x2(sp) | Argument/result registers: x10-x17 |
| Frame pointer: x8(fp) | |
| Return address: x1(ra) | |
| Stack above the stack pointer | Stack below the stack pointer |

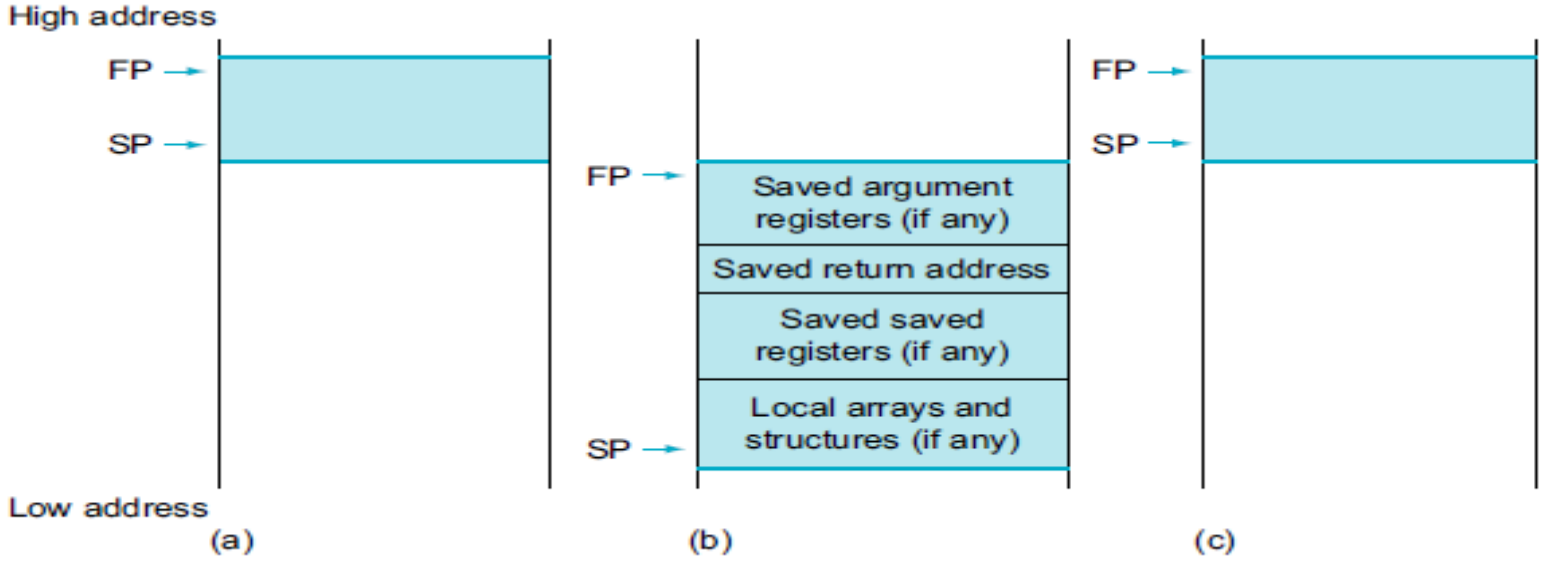
# Instructions – Language of Computer

## Supporting Procedures in Computer Hardware

### Allocating Space for New Data on the Stack

- Stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures.
- **Procedure frame** or **Activation record : It is** the segment of the stack containing a procedure's saved registers and local variables

Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call.




**frame pointer** A value denoting the location of the saved registers and local variables for a given procedure.

# Instructions – Language of Computer

## Supporting Procedures in Computer Hardware

**RISC-V register conventions for assembly language.**

| Name | Register number | Usage | Preserved on call? |
|---|---|---|---|
| x0 | 0 | The constant value 0 | n.a. |
| x1 (ra) | 1 | Return address (link register) | yes |
| x2 (sp) | 2 | Stack pointer | yes |
| x3 (gp) | 3 | Global pointer | yes |
| x4 (tp) | 4 | Thread pointer | yes |
| x5-x7 | 5–7 | Temporaries | no |
| x8-x9 | 8–9 | Saved | yes |
| x10-x17 | 10–17 | Arguments/results | no |
| x18-x27 | 18–27 | Saved | yes |
| x28-x31 | 28–31 | Temporaries | no |

# Instructions – Language of Computer

## RISC-V Addressing for wide immediate and addresses (Unit3)

### How do we Initialize 32-bit or larger constants or addresses in RISC-V ?

- RISC-V I-type Instruction support 12 bit immediate values.
- Although constants are frequently short and fit into the 12-bit fields, sometimes they are bigger.
- The RISC-V instruction set includes the instruction ***Load upper immediate(lui)***

| | | | |
|---|---|---|---|
| **Syntax** | | lui rd, $imm_{20}$ | |
| **Operation:** | | load a 20-bit constant into bits 12 through 31 of a destination register. Rd[31:12] = $imm_{20}$ | |
| **Example:** | | lui x19, 3D0H; | |
| **Before Execution** | | | |
| **x19=** | = | 0000 0000 0000 0000 0000 | 0000 0000 0000b |
| **After Execution** | | | |
| **x19=** | = | **0000 0000 0011 1101 0000** | 0000 0000 0000b (0x003D0 000) |

# Instructions – Language of Computer

## RISC-V Addressing for wide immediate and addresses

**Upper Format for Upper Immediate**


- ✓ Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- ✓ One destination register, rd
- ✓ This format is used for lui and **auipc**

# Instructions – Language of Computer

## RISC-V Addressing for wide immediate and addresses

### How do we Initialize 32-bit or larger constants or addresses in RISC-V ?

Write a assembly code to initialize 0x003D0500 in register X19

lui X19,0x003D0
addi X19,X19,0X500

| | | | |
|---|---|---|---|
| x19 | 0x0000 0000 | 0000 0000 0000 0000 0000 | 0000 0000 0000 |

Before execution of lui X19,0x003D0

| | | | |
|---|---|---|---|
| x19 | 0x003D0000 | 0000 0000 0011 1101 0000 | 0000 0000 0000 |

After execution of lui X19,0x003D0

| | | | |
|---|---|---|---|
| imm12 | 0x500 | 0000 0000 0000 0000 0000 | **0**101 0000 0000 |

| | | | |
|---|---|---|---|
| x19 | 0x003D0000 | 0000 0000 0011 1101 0000 | 0101 0000 0000 |

After execution of addi X19,X19,0x500

**Note: addi**12-bit immediate is always **sign-extended**

# Instructions – Language of Computer

## RISC-V Addressing for wide immediate and addresses

### How do we Initialize 32-bit or larger constants or addresses in RISC-V ?

Write a assembly code to initialize **0xDEADBEEF** in register X19

lui X19,0xDEADB
addi X19,X19,0XEEF

| | | | |
|---|---|---|---|
| x19 | 0x0000 0000 | 0000 0000 0000 0000 0000 | 0000 0000 0000 |

Before execution of lui X19,0x003D0

| | | | |
|---|---|---|---|
| x19 | 0xDEADB000 | 1101 1110 1010 1101 1011 | 0000 0000 0000 |

After execution of lui X19,0x003D0

| | | | |
|---|---|---|---|
| imm12 | 0x500 | 1111 1111 1111 1111 1111 | 1110 1110 1111 |

| | | | |
|---|---|---|---|
| x19 | 0xDEADAEEF | 1101 1110 1010 1101 1010 | 1110 1110 1111 |

After execution of addi X19,X19,0x500

| | |
|---|---|
| x19 | 0xDEADAEEF - It is Not a expected Initialization. What has to be done to initialize x19 with the correct result? |

## Addressing in Branches

- The RISC-V branch instructions use an RISC-V instruction format with a 12-bit immediate. This format can represent branch addresses from −4096 to 4094, in multiples of 2 as it is only possible to branch to even addresses.

**bne x10, x11, 2000  rs1=x10, rs2=x11 and imm= 0x7D0 =  0111 11  01000  0**

The above instruction can be assembled into the S format

| 0011111 | 01011 | 01010 | 001 | 01000 | 1100111 |
|---|---|---|---|---|---|
| imm[12:6] | rs2 | rs1 | funct3 | imm[5:1] | opcode |

where the **opcode** for conditional branches is **1100111**two and **bne's funct3** code is $\mathbf{001}_{\mathbf{two}}$.

| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | beq |
|---|---|---|---|---|---|---|

# Instructions – Language of Computer

## RISC-V Addressing for wide immediate and addresses

### Addressing in Branches

| | | | | | | |
|---|---|---|---|---|---|---|
| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | beq |
| imm[12\|10:5] | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | bne |
| imm[12\|10:5] | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | blt |
| imm[12\|10:5] | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | bge |
| imm[12\|10:5] | rs2 | rs1 | 110 | imm[4:1\|11] | 1100011 | bltu |
| imm[12\|10:5] | rs2 | rs1 | 111 | imm[4:1\|11] | 1100011 | bgeu |

# Instructions – Language of Computer

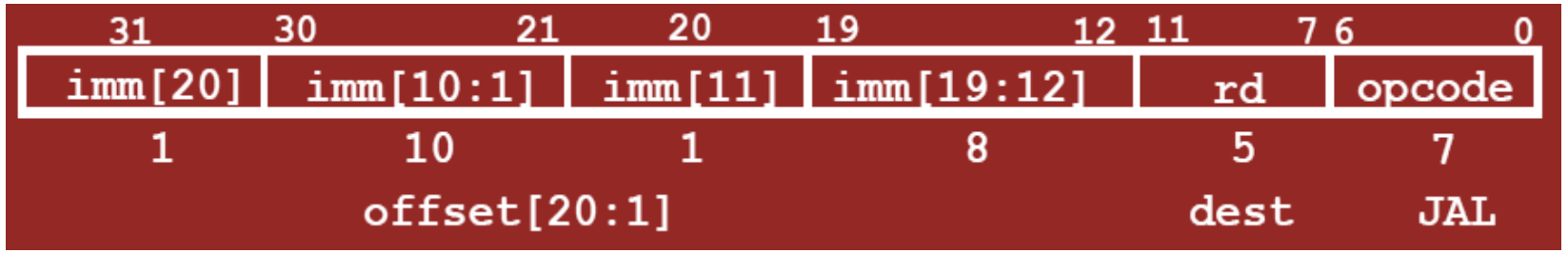
## RISC-V Addressing for wide immediate and addresses

### Addressing in Branches

**What do we do if destination is > $2^{10}$ instructions away from branch?**

**$2^{10}$ =1024 words = 1024*4=4092**

### UJ Format for Jump Instructions




The UJ-type format's address operand uses an unusual immediate encoding, and it cannot encode odd addresses.

jal x0, 2000 // go to location $2000_{ten}$ = 0111 1101 0000

| 00000000001111101000 | 00000 | 1101111 |
|---|---|---|
| imm[20:1] | rd | opcode |

## Addressing in Branches

### PC- Relative Addressing

- If addresses of the program had to fit in this 20-bit field, it would mean that no program could be bigger than $2^{20}$, which is far too small to be a realistic option today.

  Program counter  Register Branch offset

- This sum allows the program to be as large as 232 and still be able to use **conditional branches**, solving the branch address size problem. Then the question is, which register?

  **jalr x0, 0(x1)**

- **lui** writes bits 12 through 31 of the address to a temporary register, and
- **jalr** adds the lower 12 bits of the address to the temporary register and jumps to the sum.
- Refer to the number of w*ords* between the branch and the target instruction, rather than the number of bytes.

## Addressing in Branches

Showing Branch Offset in Machine Language
The *while* loop on page 100 was compiled into this RISC-V assembler code:

```
Loop:
slli x10, x22, 2       // Temp reg x10 = i * 4
add x10, x10, x25 // x10 = address of save[i]
lw x9, 0(x10)          // Temp reg x9 = save[i]
bne x9, x24, Exit   // go to Exit if save[i] != k
addi x22, x22, 1   // i = i + 1
beq x0, x0, Loop   // go to Loop
Exit:
```

| Address | Instruction | | | | | |
|---|---|---|---|---|---|---|
| 80000 | 0000000 | 00010 | 10110 | 001 | 01010 | 0010011 |
| 80004 | 0000000 | 11001 | 01010 | 000 | 01010 | 0110011 |
| 80008 | 0000000 | 00000 | 01010 | 011 | 01001 | 0000011 |
| 80012 | 0000000 | 11000 | 01001 | 001 | 01100 | 1100011 |
| 80016 | 0000000 | 00001 | 10110 | 000 | 10110 | 0010011 |
| 80020 | 1111111 | 00000 | 00000 | 000 | 01101 | 1100011 |

If we assume we place the loop starting at location 80000 in memory, what is the RISC-V machine code for this loop?

# Instructions – Language of Computer

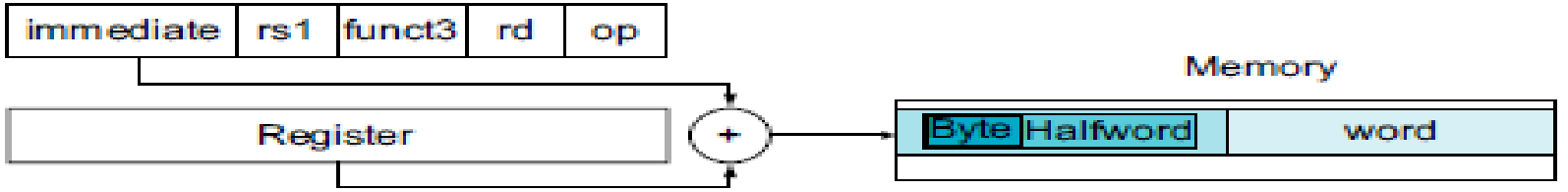
## Illustration of four RISC-V addressing modes

### Addressing in Branches

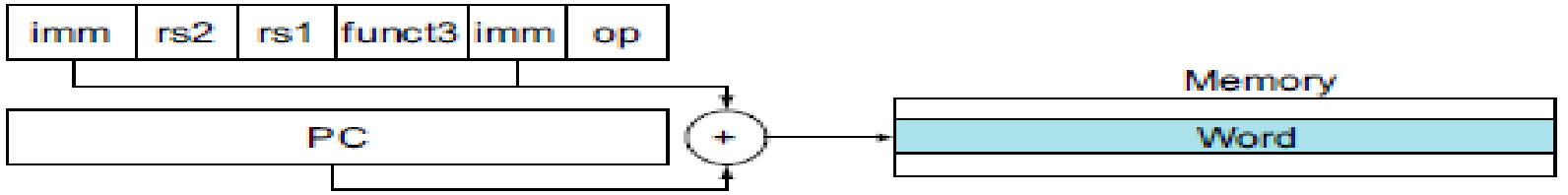
1. Immediate addressing


2. Register addressing


3. Base addressing




4. PC-relative addressing



# Instructions – Language of Computer

## RISC-V instruction encoding

| Format | Instruction | Opcode | Funct3 | Funct6/7 |
|---|---|---|---|---|
| R-type | add | 0110011 | 000 | 0000000 |
| | sub | 0110011 | 000 | 0100000 |
| | sll | 0110011 | 001 | 0000000 |
| | xor | 0110011 | 100 | 0000000 |
| | srl | 0110011 | 101 | 0000000 |
| | sra | 0110011 | 101 | 0000000 |
| | or | 0110011 | 110 | 0000000 |
| | and | 0110011 | 111 | 0000000 |
| | lr.d | 0110011 | 011 | 0001000 |
| | sc.d | 0110011 | 011 | 0001100 |
| I-type | lb | 0000011 | 000 | n.a. |
| | lh | 0000011 | 001 | n.a. |
| | lw | 0000011 | 010 | n.a. |
| | lbu | 0000011 | 100 | n.a. |
| | lhu | 0000011 | 101 | n.a. |
| | addi | 0010011 | 000 | n.a. |
| | slli | 0010011 | 001 | 000000 |
| | xori | 0010011 | 100 | n.a. |
| | srli | 0010011 | 101 | 000000 |
| | srai | 0010011 | 101 | 010000 |
| | ori | 0010011 | 110 | n.a. |
| | andi | 0010011 | 111 | n.a. |
| | jalr | 1100111 | 000 | n.a. |
| S-type | sb | 0100011 | 000 | n.a. |
| | sh | 0100011 | 001 | n.a. |
| | sw | 0100011 | 010 | n.a. |
| SB-type | beq | 1100111 | 000 | n.a. |
| | bne | 1100111 | 001 | n.a. |
| | blt | 1100111 | 100 | n.a. |
| | bge | 1100111 | 101 | n.a. |
| | bltu | 1100111 | 110 | n.a. |
| | bgeu | 1100111 | 111 | n.a. |
| U-type | lui | 0110111 | n.a. | n.a. |
| UJ-type | jal | 1101111 | n.a. | n.a. |

### Decoding Machine Code

What is the assembly language statement corresponding to this machine instruction?

**00578833hex**

**0000 0000 0101 0111 1000 1000 0011 0011b**

| funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|
| 0000 000 | 00101 | 01111 | 000 | 10000 | 0110011 b |
| | x5 | x15 | | x16 | |


# THANK YOU

**Mahesh Awati**
Department of Electronics and Communication

**mahesha@pes.edu**

+91 9741172822