DESIGNING A SINGLE CYCLE RISCV PROCESSOR USING VERILOG HDL

By Anish Rooj

WHAT HAPPENS WHEN YOU RUN A C PROGRAM

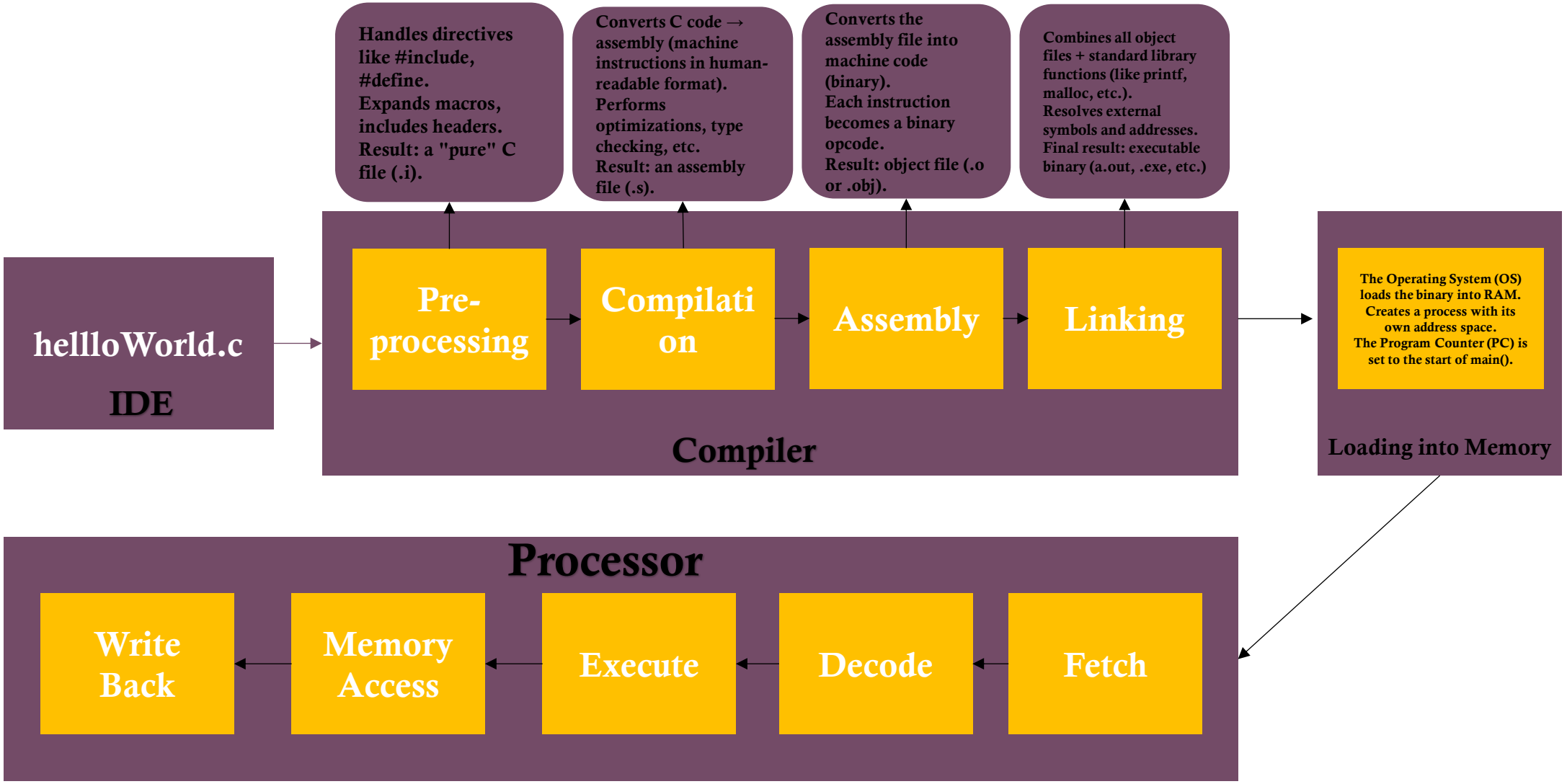
Suppose, a user writes a C program and runs it. What exactly happens?

1. The **Compiler** processes the **.c** file through multiple stages (preprocessing, compilation, assembly, and linking), ultimately converting it into machine-level code understood by the processor.

2. This machine code consists of low-level instructions (typically 32-bit wide for **RISC** architectures), which a single-cycle processor can execute step by step—one instruction per clock cycle.

3. These **Instructions** are loaded into memory, and the **Program Counter** (PC) is initialized (usually to address 0). From there, the processor begins executing instructions sequentially using the **Fetch-Decode-Execute** cycle.

WHAT HAPPENS WHEN YOU RUN A C PROGRAM



THE PROCESSOR

A **Processor**, also known as a **CPU** (Central Processing Unit), is the brain of a computer or digital system. It executes instructions from a program and coordinates the operations of the system. The processor operates through a well-defined cycle:

- **Fetch** – Retrieve the next instruction from memory (address given by the **Program Counter**).
- **Decode** – Interpret the instruction to determine the required operation and operands.
- **Memory Access** – If the instruction requires data (e.g., lw, sw), read from or write to memory at a computed address.
- **Execute** – Perform the operation using the ALU (e.g., arithmetic, logic, branching).
- **Write Back** – Store the result into a register or memory, depending on the instruction type.

DIFFERENT TYPES OF INSTRUCTION SET COMPUTERS

FEATURE	REDUCED INSTRUCTION SET COMPUTER	COMPLEX INSTRUCTION SET COMPUTER
Instruction Complexity	Simple, fixed-length instructions	Complex, variable-length instructions
Execution Time	Usually 1 cycle per instruction	Multiple cycles per complex instruction
Instruction Length	Fixed (e.g., 32 bits)	Variable (1–15 bytes in x86)
Memory Access	Load/store architecture (register-only ops)	Instructions can directly access memory
Decoding Simplicity	Easy to decode (uniform format)	Harder to decode (varied formats)
Pipelining & Parallelism	Easier to pipeline	More difficult due to complex instructions
Compiler Role	Heavily relies on compiler optimization	Less compiler effort needed
Examples	RISC-V, ARM, MIPS, SPARC	x86, VAX, Intel 8086, Motorola 68k

OUR GOAL

In this presentation, our goal is to design a Simple **Single-Cycle RISC-V Processor** that supports a wide subset of the **RV32I** base instruction set, including arithmetic, logical, memory, and control flow instructions.

We will:

i) Implement the processor using **Verilog**, a Hardware Description Language used for designing digital circuits.

ii) Test the processor using various **RISC-V** assembly programs to verify functional correctness and instruction compatibility.

iii) Focus on building a clean, modular design that follows standard **Datapath** and control principles.

iv) Incorporate key components like the **Program Counter** (PC), **Instruction Memory**, **Register File**, **ALU**, **Data Memory**, and a **Control Unit** to realize the full execution cycle.

v) Emphasize the fetch-decode-execute-writeback cycle, with each instruction completed in a single clock cycle.

THE RISC-V BASE INSTRUCTIONS FORMAT

The RV32I base instruction set defines six fundamental instruction formats, each tailored to specific types of operations:

- **R-type:** Used for register-to-register arithmetic and logical operations (e.g., **add**, **sub**, **and**).
- **I-type:** Used for immediate operations, load instructions, and system instructions like **jalr** and **ecall**.
- **S-type:** Used for store instructions, where data from a register is stored to memory (e.g., **sw**, **sb**, **sh**).
- **B-type:** Used for conditional branch instructions that alter control flow based on comparisons (e.g., **beq**, **bne**, **blt**).
- **U-type:** Used for instructions requiring large immediate values placed in the upper bits, such as **lui** and **auipc**.
- **J-type:** Used for unconditional jump instructions like **jal**, enabling long-range jumps in code execution.

RV32I INSTRUCTION FORMATS: A CLASSROOM ANALOGY

The RV32I instruction set contains many instructions, but they all fit into six basic formats. You can think of these instructions like students in different classes.

To identify a specific instruction, we follow a 3-step process — similar to identifying a student in a school:

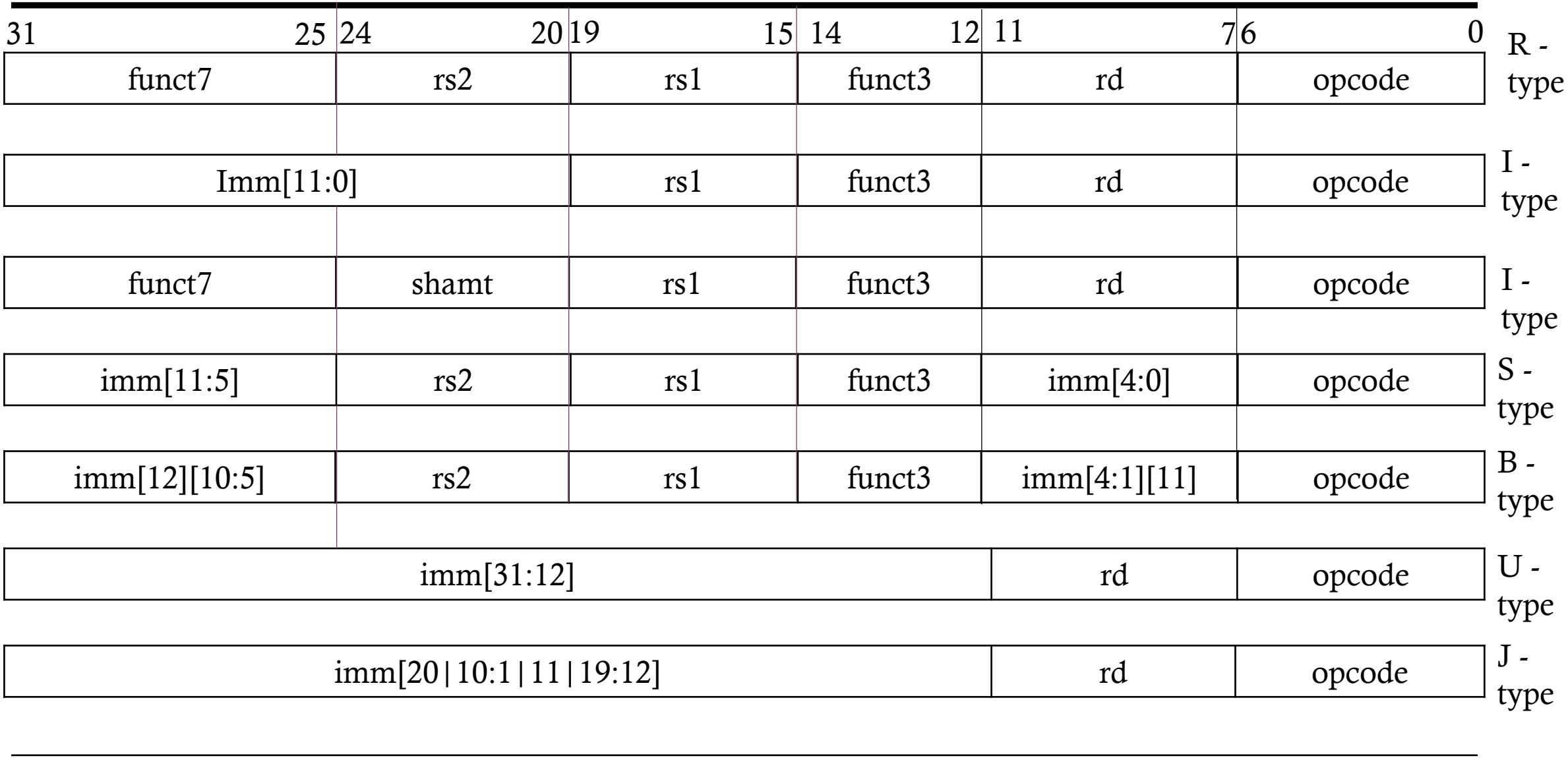
- **Class (Opcode)** – This tells us which group the instruction belongs to (e.g., R-type, I-type, etc.).
- **Section (funct7)** – Within a class, students may be split into different sections. This helps further narrow down the instruction's category.
- **Roll Number (funct3)** – Finally, each instruction (student) has a unique identifier within that section.

Also, there are three different fields :
i) **rs1 and rs2** – This tells us about the addresses of the source registers. Source Registers are generally operands. There are two Sources i.e. **rs1** and **rs2** in the Instruction Formats. **rs1** and **rs2** are used in **R**, **S**, and **B**-type instructions

ii) **rd** – This tells us about the address of the destination register --- register at which the result will be stored. Rd is used in **R**-type, **I**-type, **J**-type, **U**-type instructions.

iii) **Imm** – Finally, **imm** or **immediate** refers to a Constant value with which many operations can be done.

RISC-V BASE INSTRUCTION FORMATS



R-TYPE INSTRUCTIONS

The R-type instructions is basically operation between two operands stored in two different (or same) registers. The format of the R-type Instruction is very meaningful. The rs1 and rs2 indicates two operands, the rd is the destination register at which the result will be stored. The opcode, funct3 and funct7 determines which operation is to be performed between rs1 and rs2. The opcode is constant for R-type instructions since the Class is same (Remember the class analogy?) There are several operations that are supported :
i) add rd, rs1, rs2 : As its name suggests, this instruction indicates addition operation between rs1 and rs2 i.e. (rs1 + rs2). The result is stored in the destination register rd.

ii) sub rd, rs1, rs2 : Similarly this is subtraction between rs1 and rs2. Again, the result is stored in the destination register rd

To avoid repetitiveness the following instructions are just written without any description since it is obvious from their names and the instructions which needs some clarifications are written with description:
and, **or**, **xor** (Bitwise operation between rs1 and rs2)

iii) sll rd, rs1, rs2 : This is the Shift Left Logical operation on rs1. Shifts rs1 left by amount in lower 5 bits of rs2.

iv) srl rd, rs1, rs2 : This is the Shift Right Logical operation on rs1. Shifts rs1 right (fill with zeros) by the lower 5 bits of rs2.

v) slt rd, rs1, rs2 : Signed comparison between rs1 and rs2 ---> if rs1 < rs2, then rd = 1, else rd = 0.

vi) sltu rd, rs1, rs2 : This is same as **slt** but unsigned comparison is done.

vii) sra rd, rs1, rs2 : This is Shift Right Arithmetic operation done on rs1. rs1 is shifted to the right by rs2 amount preserving the Sign Bit i.e. rs[4].

THE SUMMARY OF R-TYPE INSTRUCTIONS

Instruction	funct7	funct3	Operation
add	**0000000**	**000**	**Add (signed)**
sub	**0100000**	**000**	**Subtract (signed)**
sll	**0000000**	**001**	**Shift Left Logical**
slt	**0000000**	**010**	**Set Less Than (signed)**
sltu	**0000000**	**011**	**Set Less Than (unsigned)**
xor	**0000000**	**100**	**Bitwise XOR**
srl	**0000000**	**101**	**Shift Right Logical**
sra	**0100000**	**101**	**Shift Right Arithmetic**
or	**0000000**	**110**	**Bitwise OR**
and	**0000000**	**111**	**Bitwise AND**

Opcode is same for all : **0110011**

I-TYPE INSTRUCTIONS

The I-type instructions is also operation between two operands but one of them is a constant provided by the user and the other is a value stored in some register. This constant value provided by the user is called immediate and it is of 12 bits, thus, it is Sign Extended to 32 bits by the Sign Extender Block. ***Supported I-type Instrucitons :***

i) addi rd, rs1, imm : This is signed addition between rs1 and immediate i.e. rs1+imm. The result is stored at the destination register rd.

Similarly **sub rd, rs1, imm**

ii) xorI rd, rs1, imm : This is Bitwise XOR operation between rs1 and immediate i.e. rs1^imm. The result is stored at the destination register rd.

Similarly, **andi**, **ori**

iii) slti rd, rs1, imm : Signed Comparison between **rs1** and **imm** : Set rd = 1 if rs1 < imm (signed)

iv) sltiu rd, rs1, imm : Same as slti but Unsigned comparison is done

v) slli rd, rs1, shamt : Logical left shift by **shamt** bits

vi) srli rd, rs1, shamt : Logical right shift by **shamt** bits

vii) srai rd, rs1, shamt : Arithmetic right shift (preserves sign)

viii) lw rd, imm(rs1) : Load 32-bit word from memory at rs1(acts as base address) + imm (acts as offset) and store it in rd.

ix) lh rd, imm(rs1) : Load 16-bit halfword (sign-extended)

x) lhu rd, imm(rs1) : Load 16-bit halfword (unsigned)

xi) lb rd, imm(rs1) : Load 8-bit byte (sign-extended)

xii) lbu rd, imm(rs1) : Load 8-bit byte (unsigned)

xiii) jalr : Used for function returns or indirect jumps. Sets **rd** = **PC** + 4 and **PC** = (**rs1** + **imm**) & **~1** → clears LSB to ensure **word alignment**

THE SUMMARY OF I-TYPE INSTRUCTIONS

Instruction	opcode	funct7	funct3	Operation
addi	0010011	-	000	Add Immediate (signed)
slli	0010011	0000000	001	Shift Left Logical (Immediate)
slti	0010011	-	010	Set Less Than (signed)
sltiu	0010011	-	011	Set Less Than (unsigned)
xori	0010011	-	100	Bitwise XOR with Immediate
srli	0010011	0000000	101	Shift Right Logical (Immediate)
srai	0010011	0100000	101	Shift Right Arithmetic (Immediate)
ori	0010011	-	110	Bitwise OR with Immediate
andi	0010011	-	111	Bitwise AND with Immediate
jalr	1100111	-	000	Jump and Link Register
lw	0000011	-	010	Load Word from Memory

S-TYPE INSTRUCTIONS

S-type or Store type instructions are used to Store data from a register (rs2) into the Data Memory. These instructions do not have a destination register rd, since the value is to be stored in the Data Memory no in the Register File. The rs1 acts as the Base Address and the Sign Extended Immediate acts as the Offset. The immediate value is split between two parts:

> imm[11:5] → bits [31–25]

> imm[4:0] → bits [11–7]

These two parts are concatenated and sign-extended to form a full 12-bit offset.

So, the data is effectively stored at memory[rs1+imm]. These stored data can be accessed using the Load instructions which are I type instructions. The opcode = 0100011 is fixed for all S-type instructions.

Supported S-Type Instructions :

i) sb rs2, imm(rs1) : Store 8-bit byte from rs2 to memory[rs1+imm]

ii) sh rs2, imm(rs1) : Store 16-bit halfword from rs2 to memory[rs1+imm]

ii) sw rs2, imm(rs1) : Store 32-bit word from rs2 to memory[rs1+imm]

Instruction	funct3	Description
sw	**010**	**Store Word (32 bits)**
sb	**000**	**Store Byte**
sh	**001**	**Store Halfword**

B-TYPE INSTRUCTIONS

B-type (Branch-type) instructions are used for conditional program control. These instructions compare two register values (rs1 and rs2) and change the PC to a new address (PC-relative) if the condition is true. The immediate (imm) used for the branch offset is scattered across multiple fields and must be reconstructed and sign-extended to form a 13-bit PC-relative offset :

- imm = {imm[12], imm[10:5], imm[4:1], imm[11], 0}
- The 0 at the end means the offset is always even (instructions are word-aligned).
- The PC is Loaded with Branch Target = PC + imm which represents a Label.
- Opcode = 1100011 is same for all B-type instructions

Supported Branch Type Instructions :

- **beq rs1, rs2, imm** : If rs1 == rs2, then go to Label i.e. Branch Target = PC + imm
- **bne rs1, rs2, imm** : If rs1 != rs2, then go to Label i.e. Branch Target = PC + imm
- **bge rs1, rs2, imm** : If rs1 > rs2 (signed), then go to Label i.e. Branch Target = PC + imm
- **beq rs1, rs2, imm** : If rs1 < rs2 (signed), then go to Label i.e. Branch Target = PC + imm
- **bltu rs1, rs2, imm** : If rs1 < rs2 (unsigned), then go to Label i.e. Branch Target = PC + imm
- **bgtu rs1, rs2, imm** : If rs1 > rs2 (unsigned), then go to Label i.e. Branch Target = PC + imm

SUMMARY OF B-TYPE INSTRUCTIONS

Instruction	funct3	Operation
beq	**000**	**Branch if Equal**
bne	**001**	**Branch if Not Equal**
bge	**010**	**Branch if Greater or Equal (signed)**
blt	**011**	**Branch if Less Than (signed)**
bgtu	**100**	**Branch if Greater/Equal (unsigned)**
bltu	**101**	**Branch if Less Than (unsigned)**

- funct7 is not used for B-type instructions
- PC + imm (13 bits) determines the Label's address (the Address at which the PC will point)
- If the condition is true, the PC is updated with:
  - PC ← PC + imm
- If the condition is false, the PC is incremented normally:
  - PC ← PC + 4

J-TYPE INSTRUCTIONS

The J-type format is used for unconditional jumps, specifically for the jal (Jump and Link) instruction. It allows jumping to a PC-relative address, and simultaneously saving the return address (PC + 4) into a register.

To compute the **jump offset**, the immediate field must be reconstructed as:

- imm = {imm[20], imm[10:1], imm[11], imm[19:12], 0}
- Total: 21 bits after appending the 0 (LSB = 0 for alignment)
- Then sign-extended to 32 bits to get the full jump offset
- Final address = **PC** + **imm**
- The return address is stored in the destination register rd
- Opcode = **1101111** for all J-type instructions

Instruction	Description
jal	Jump and Link (unconditional jump + save return address)

Supported J-Type Instruction :

> **jal rd, imm :** Jump to **PC** + **imm**, store **PC** + **4** in **rd**

U-TYPE INSTRUCTIONS

U-type (Upper Immediate) instructions are used for operations that require a **large immediate value**, typically to set or manipulate the upper 20 bits of a register. These instructions do not perform ALU operations or memory access directly — they **load or compute constants**.

- imm[31:12]: A 20-bit immediate constant
- The lower 12 bits are automatically set to 0 (effectively a left shift by 12 bits)
- The result is stored at the Destination Register rd.

Supported U-type Instructions :

> **lui rd, imm** : Load Upper Immediate: rd = imm << 12

- **auipc rd, imm** : Add Upper Immediate to PC: rd = PC + (imm << 12)

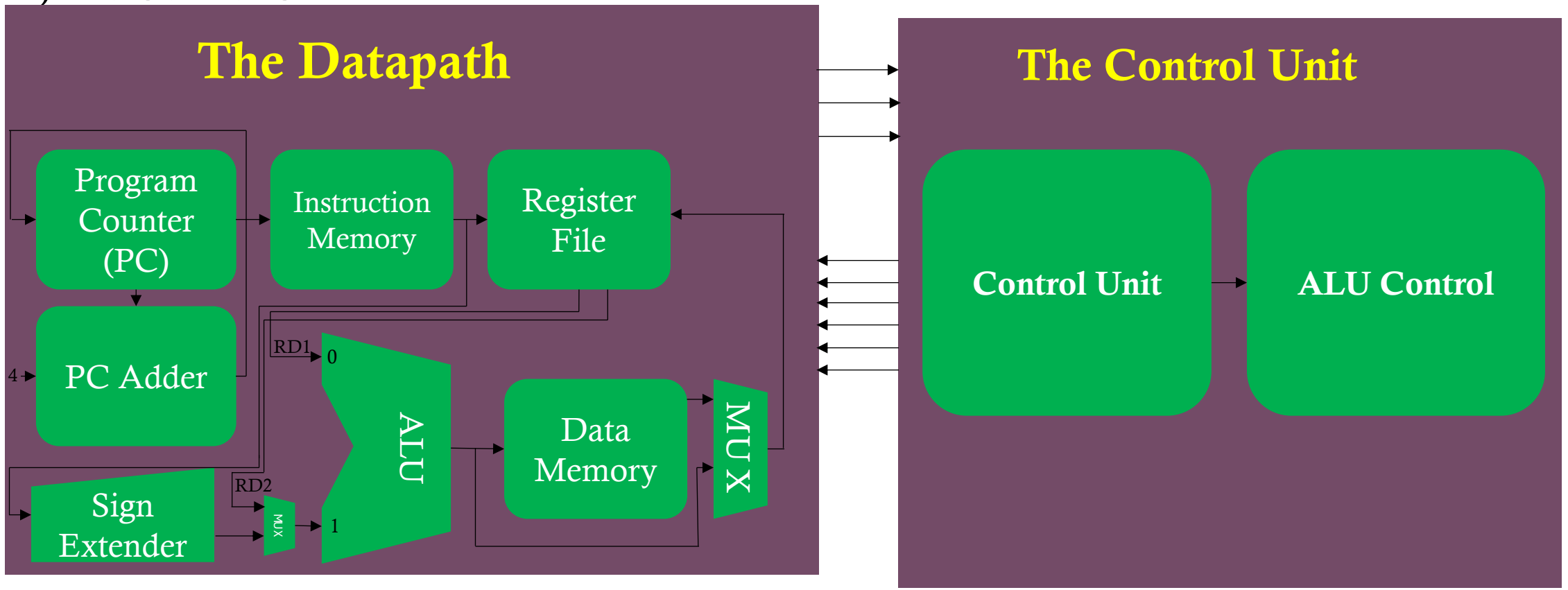
Instruction	opcode	Description
lui	**0110111**	**Load upper immediate into register**
auipc	**0010111**	**PC-relative address calculation**

HIGH LEVEL VIEW OF THE SYSTEM

A Single Cycle RISCV Processor executes instructions by coordinating two core subsystems :

i) **The Datapath**

ii) **The Control Unit**



THE DATAPATH

The **Datapath** is the part of the processor that **performs operations on data** — such as fetching, computing, and storing — under the control of the **Control Unit**. The Datapath consists of :

i) The Program Counter : The **Program Counter** is a register that holds the **address of the next instruction** to execute. On every clock cycle the PC :

a) Sends its value to **Instruction Memory** to fetch the current instruction.

b) Gets updated to point to the next instruction (normally PC + 4 or to a **branch/jump target**).

ii) The Instruction Memory : A **Read-Only Memory (ROM)** block that stores the program's instructions (in binary format). The PC acts as an address input to this memory. Outputs a 32-bit instruction based on the PC's value. Operates synchronously with the clock.

ii) The Register File : A set of 32 registers (x0 to x31), each 32 bits wide. These registers are also called General Purpose Registers or GPR. Reads values from source registers rs1 and rs2. Writes the result to rd if RegWrite signal is active.

iv) Sign Extender : Extracts the immediate (constant) field from the instruction based on its format (I-type, S-type, etc.). Performs sign-extension (e.g., a 12-bit **imm** is extended to 32 bits with sign bit replicated). The generated immediate is used as a second operand in ALU operations or address offset in memory and branch instructions.

v) PC Adder : A Simple Full Adder which adds a constant 4 (in decimal) to PC. It accepts PC's output and generates PC+4 which is given to the Program Counter as an Input to point to the next Instruction.

vi) ALU : Performs core **Arithmetic** and **Logical** operations like add, or, xor, sub, compare etc. Its Inputs are rs1(fixed) and rs2 or Immediate. Its outputs are Zero and ALUResult.

vii) MUX : A simple 2:1 MUX which selects between the Data coming from the Data Memory (0) and the ALU Result (1). The purpose of this MUX is to differentiate between Store and other instructions.

Also, there is another MUX before ALU input 1 to select imm or RD2. The selector signal is ALUSrc which comes from the Control Unit.

THE CONTROL UNIT

The **Control Unit** is responsible for **decoding the instruction** and generating **control signals** that drive the datapath components (registers, ALU, memory, multiplexers, etc.). It consists of two main parts:

- **Main Control Unit** – This unit takes the **7-bit opcode** from the instruction and produces control signals that affect the entire **Datapath**. Based on the opcode, this block produces various control signals to Datapath.
- **ALU Control :** The ALU Control Unit is responsible for selecting the exact operation that the ALU must perform, based on:
  - Instruction type (R-type, I-type, Branch, Load/Store)
  - funct3 and funct7 fields from the instruction
  - A higher-level control signal: ALUOp (2 bits), generated by the Main Control Unit

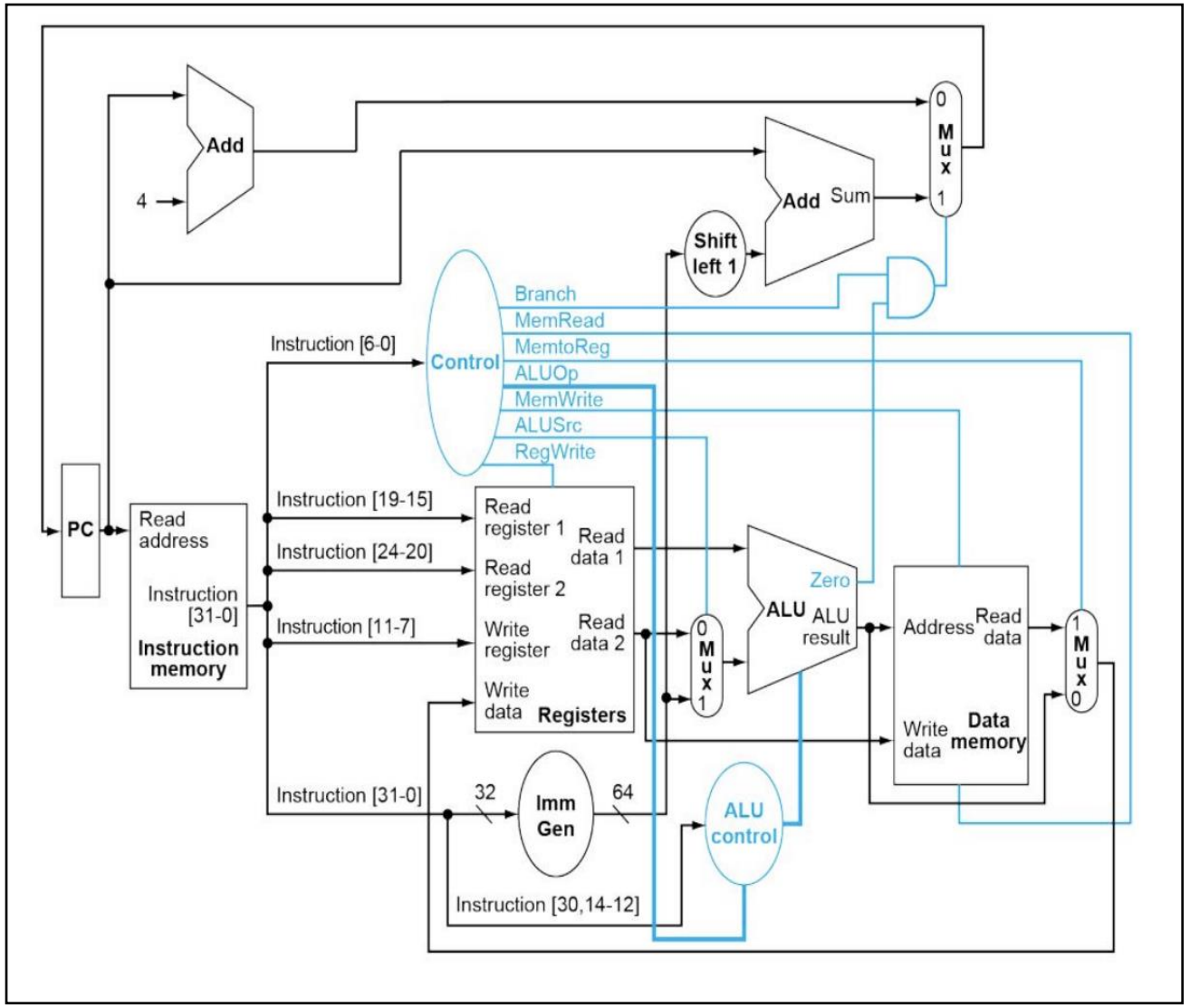
These two modules will be discussed in great details in the Designing section.

THE OVERALL SYSTEM

This diagram illustrates the **complete integration** of the **Datapath** and the **Control Unit**, forming the core of the **single-cycle RISC-V processor**. Each connection shown is critical for executing instructions correctly in one clock cycle.

Notice that the PC is loaded with PC+4 or PC + some offset depending upon the Branch (from the Control Unit) and the Zero (from the ALU). This is due to the fact that the Branch instructions can be used to jump to different locations.

This picture is taken from *Computer Organization and Design* by David A. Patterson and John L. Hennessy



DESIGNING THE CONTROL UNIT

Our first step will be to design the Control Unit which will send Control Signals to the Datapath. As, discussed previously, the CU consists of a Main Control Unit and an ALU Control Unit. We will start designing the Main Control Unit first since it sends the ALUOp signal to the ALU Control Unit.

The Main Control Unit (controlUnit.v)

This block accepts the **opcode (7 bits)** as its input and produces necessary control signals. Its outputs are :

Signal	Purpose
RegWrite	Enables writing to the register file (for rd).
ALUSrc	Selects between rs2 (0) or an immediate (1) as ALU input.
MemtoReg	Selects whether ALU result (0) or memory read data (1) is written back to rd.

Signal	Purpose
MemRead	Enables reading from data memory (for lw).
MemWrite	Enables writing to data memory (for sw).
Branch	Indicates a branch instruction (beq, bne).

Signal	Purpose
ALUOp (2 bits)	Guides the ALU Control on which operation category to perform: - 00 = ADD (loads/stores) - 01 = SUB/SLT (branches) - 10 = Decode funct3/funct7 (R/I-type).
Jal	Indicates a jump-and-link (JAL) instruction. Updates PC and writes PC+4 to rd.
Jalr	Indicates a jump-and-link-register (JALR) instruction. Uses rs1 + imm for jump target.

Signal	Purpose
Lui	Indicates load upper immediate (LUI). Writes imm << 12 to rd.
Auipc	Indicates add upper immediate to PC (AUIPC). Writes PC + (imm << 12) to rd.

DESIGNING THE CONTROL UNIT

The main purpose of this block is to determine the Class based on the opcode. There are 9 distinct opcodes representing R (opcode = 0110011), I (0010011), Load (0000011), Store (0100011), Branch (1100011), Jal (1101111), Jalr (1100111), LUI (0110111) and AUIPC (0010111). The control signals will be decided based on these opcodes. For example, if opcode = 0110011 i.e. R type Instruction :

- **ALUSrc** = 0 → Indicates that there is no Imm in the instruction, so, choose the ALU input 1 (Remember there is a MUX before ALU input 1) to RD2 (coming from the Register File).
- **RegWrite** = 1 → Tells the Register File to Write the Available data into the destination register rd.
- **ALUOp** = 2'b10 → Tells the ALU Control Block that this might be R or I type instruction, so further check funct3 and funct7.

We will naturally use case statement in Verilog under an always combinational block. The control signals will have a default value of zero.

> A Natural Question may arise that how are we deciding the ALU Op ? Is it random or is it our choice ?
> The answer is, yes, it entirely depends on the Processor Designer. If you, assign different ALU Op, then accordingly you have to design the ALU Control.
> The ALU Op basically helps us to distinguish between Memory Access (Load and Store), Branch and R-I type instructions.
> In many books, it is decided to use the following convention for the ALU Op :

00 : Load and Store 10 : R-I type instr
01 : Branch 11 : Invalid

THE VERILOG CODE FOR THE CONTROL UNIT

```
`timescale 1ns/1ps
module controlUnit(
    input [6:0] opcode,
    output reg Branch, MemRead, MemtoReg,
    output reg [1:0] ALUOp,
    output reg MemWrite, ALUSrc, RegWrite,
    output reg Jal, Jalr,
    output reg Lui, Auipc
);
    always @(*) begin
        // Default values
        Branch = 0; MemRead = 0; MemtoReg = 0;
        MemWrite = 0; ALUSrc = 0; RegWrite = 0;
        ALUOp = 2'b00;
        Jal = 0; Jalr = 0;
        Lui = 0; Auipc = 0;
        case (opcode)
            7'b0110011: begin // R-type
                ALUSrc = 0;
                RegWrite = 1;
                ALUOp = 2'b10;
            end
            7'b0010011: begin // I-type ALU (ADDI, SLTI, ANDI, SLLI, etc.)
                ALUSrc = 1;
                RegWrite = 1;
                ALUOp = 2'b10;
            end
            7'b0000011: begin // Load (LB, LH, LW, LBU, LHU)
                ALUSrc = 1;
                RegWrite = 1;
                MemRead = 1;
                MemtoReg = 1;
            end
            7'b0100011: begin // Store (SB, SH, SW)
                ALUSrc = 1;
                MemWrite = 1;
            end
            7'b1100011: begin // Branch (BEQ, BNE, etc.)
                ALUSrc = 0;
                Branch = 1;
                ALUOp = 2'b01;
            end
            7'b1101111: begin // JAL
                RegWrite = 1;
                Jal = 1;
            end
            7'b1100111: begin // JALR
                RegWrite = 1;
                Jalr = 1;
                ALUSrc = 1;
            end
            7'b0110111: begin // LUI
                RegWrite = 1;
                Lui = 1;
            end
            7'b0010111: begin // AUIPC
                RegWrite = 1;
                Auipc = 1;
            end
            default: begin
                // Already covered by default values above
            end
        endcase
    end
endmodule
```

DESIGNING THE ALU CONTROL BLOCK

The ALU Control Block will accept the ALU Op (2 bits) from the Control Unit and produces an ALU Control Signal (4 Bits) which will be further accepted by the ALU to decide which operation is to be performed. So, the ALU Control Block has to decide which operation is to be performed. To design this block, we will use some observations :

- For Memory Access instructions (Load and Store) we only add (signed) the rs2 with imm. So, if ALU Op = 00, the ALU should perform Addition operation. The ALU Control for Addition is 0010.
- For Branch type instructions :
  - For beq, bne (funct3 = 000, 001 respectively) we subtract rs2 from rs2, so the ALU should do Subtraction operation. The ALU Control for Subtraction is 0010.
  - For blt, bge (funct3 = 100, 101 respectively) we do signed comparison between rs1 and rs2, so the ALU should do the same. The ALU Control for Signed Comparison is 0011.
  - For bltu, bgeu (funct3 = 110, 111 respectively) we do unsigned comparison between rs1 and rs2, so the ALU should do the same. The ALU Control for Unsigned Comparison is 1000.
- For R and I type instructions :
  - The fucnt7 is 0000000 always except for two subtraction and shift right arithmetic operations. So, when funct7 == 000000, according to funct3 we will decide the ALU Control. Please refer to the tables shown in the previous slides.
  - When funct7 is not 0000000, we will again decide the operation (whether subtraction of shift right arithmetic) based on the funct3. Accordingly we will set the ALU Control.

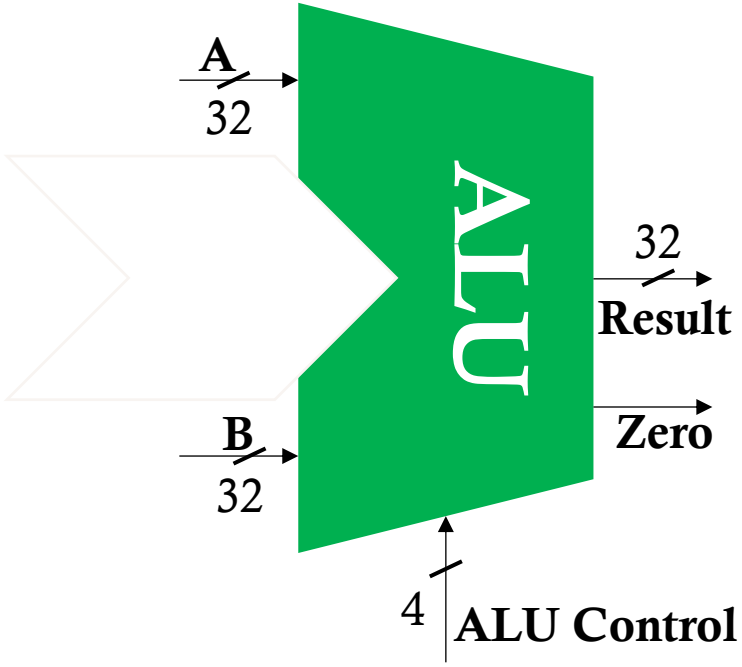
```verilog
`timescale 1ns / 1ps
module aluControl(
    input  [1:0] ALUOp,
    input  [2:0] funct3,
    input  [6:0] funct7,
    output reg [3:0] ALUControl
);
    always @(*) begin
        case (ALUOp)
            2'b00: // Load/Store
                ALUControl = 4'b0010; // ADD
            2'b01: begin
              case (funct3)
                3'b000: ALUControl = 4'b0110; // BEQ → SUB
                3'b001: ALUControl = 4'b0110; // BNE → SUB
                3'b100: ALUControl = 4'b0111; // BLT → SLT
                3'b101: ALUControl = 4'b0111; // BGE → SLT
                3'b110: ALUControl = 4'b1000; // BLTU → SLTU
                3'b111: ALUControl = 4'b1000; // BGEU → SLTU
                default: ALUControl = 4'b1111;
              endcase
            end
            2'b10: begin
          // Decode both R-type and I-type
            if (funct7 == 7'b0000000) begin
              case (funct3)
                3'b000: ALUControl = 4'b0010; // ADD (or ADDI)
                3'b001: ALUControl = 4'b0100; // SLL (or SLLI)
                3'b010: ALUControl = 4'b0111; // SLT (or SLTI)
                3'b011: ALUControl = 4'b1000; // SLTU (or SLTIU)
                3'b100: ALUControl = 4'b0011; // XOR (or XORI)
                3'b101: ALUControl = 4'b0101; // SRL (or SRLI)
                3'b110: ALUControl = 4'b0001; // OR (or ORI)
                3'b111: ALUControl = 4'b0000; // AND (or ANDI)
                default: ALUControl = 4'b1111;
              endcase
            end else begin
              case ({funct7, funct3})
                10'b0100000000: ALUControl = 4'b0110; // SUB
                10'b0100000101: ALUControl = 4'b1101; // SRA
                default: ALUControl = 4'b0010;        // Default to ADD
              endcase
            end
          end
          default: ALUControl = 4'b1111;
        endcase
    end
endmodule
```

THE VERILOG CODE FOR THE ALU CONTROL BLOCK

DESIGNING THE ARITHMETIC LOGIC UNIT (ALU)

ALU Control	Operation
0000	Bitwise AND
0001	Bitwise OR
0010	Addition
0011	Bitwise XOR
0100	Shift Logical Left
0101	Shift Logical Right
0110	Subtraction
0111	Signed Comparison
1000	Unsigned Comparison
1101	Shift Right Arithmetic

- The ALU accepts the ALU Control signal generated by the ALU Control Block and decides which operation to perform
- It accepts two operands : rd1 (fixed) and rd2 or imm depending on the ALU Src
- The output of the Zero is HIGH if the Result == 32'b0



```
1  `timescale 1ns/1ps
2  module alu(
3    input [31:0] A, B,
4    input [3:0] ALUControl,
5    output reg [31:0] Result,
6    output reg Zero
7  );
8    always @(*) begin
9      case (ALUControl)
10       4'b0000: Result = A & B;
11       4'b0001: Result = A | B;
12       4'b0010: Result = A + B;
13       4'b0110: Result = A - B;
14       4'b0111: Result = ($signed(A) < $signed(B)) ? 32'b1 : 32'b0;
15       4'b1000: Result = (A < B) ? 32'd1 : 32'd0;
16       4'b0011: Result = A ^ B;
17       4'b0100: Result = A << B[4:0];
18       4'b0101: Result = A >> B[4:0];
19       4'b1101: Result = $signed(A) >>> A[4:0];
20       default: Result = 32'h00000000;
21     endcase
22     Zero = (Result == 32'b0);
23     //$display("ALU: A=%0d, B=%0d, ctrl=%b, out=%0d", A, B, ALUControl, Result);
24   end
25 endmodule
26
```

THE VERILOG CODE FOR THE ALU

DESIGNING THE SIGN EXTENDER

The Sign Extender Block extends the sign of the immediate to make it 32 bits signed number, since the ALU does many Signed operation with Immediate. This block accepts opcode (7 bits) and the instruction (32 bits). Depending upon the opcode, the sign of the immediate is extended. To design this block we need to carefully observe the Base Formats shown in the previous slide :
> For I-type instructions, the lower 12 bits of the immediate is equal to the upper 12 bits of the instruction. So, we need to sign extend the remaining 20 bits. Hence, the upper 20 bits of the immediate will be same as the MSB of the instruction i.e. instruction[31].

- For S-type instructions, the lower 5 bits of the imm is equal to the inst[11:7] and the imm[11:5] instr[31:25]
- The rest of the sign extension logic is similar : See the base format and assign imm according to the diagram.
- This block generates sign extended Immediate which is the input to the ALU MUX (the mux before the ALU whose selector signal is ALU Src : Selects between RD2 and Imm depending upon the instruction)
- The default value of the immediate produced by this block is 32'b0

THE VERILOG CODE FOR THE SIGN EXTENDER

```
1  `timescale 1ns/1ps
2  module signExtend(
3    input [31:0] instr,
4    input [6:0] opcode,
5    output reg [31:0] imm
6  );
7  always @(*) begin
8    case (opcode)
9      7'b0000011, // I-type: lw
10     7'b0010011, // I-type: addi, andi, ori, etc.
11     7'b1100111: // I-type: jalr
12       imm = {{20{instr[31]}}, instr[31:20]};
13     7'b0100011: // S-type: sw
14       imm = {{20{instr[31]}}, instr[31:25], instr[11:7]};
15     7'b1100011: // B-type: beq, bne, blt, etc.
16       imm = {{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0};
17     7'b1101111: // J-type: jal
18       imm = {{11{instr[31]}}, instr[31], instr[19:12], instr[20], instr[30:21], 1'b0};
19     7'b0110111, // U-type: lui
20     7'b0010111: // U-type: auipc
21       imm = {instr[31:12], 12'b0};
22     default:
23       imm = 32'd0;
24   endcase
25 end
26 endmodule
27
```

DESIGNING THE PROGRAM COUNTER

The Program Counter is nothing but a simple Positive Edge Triggered D Flip Flop with Asynchronous Active High Reset. It accepts the Next address of the instruction and outputs the same in the Positive Edge.

VERILOG CODE FOR THE PROGRAM COUNTER

```
1  `timescale 1ns/1ps
2  module programCounter(
3    input clk, rst,
4    input [31:0] pc_in,
5    output reg [31:0] pc_out
6  );
7    always @(posedge clk or posedge rst) begin
8      if (rst)
9        pc_out <= 0;
10     else
11       pc_out <= pc_in;
12     //$display("PC updated to 0x%08h at time %0t", rst ? 0 :
       pc_in, $time);
13   end
14 endmodule
15
```

DESIGNING THE INSTRUCTION MEMORY

The **Instruction Memory** in a RISC-V single-cycle processor behaves as a **Read-Only Memory (ROM)** that holds machine instructions encoded in 32-bit format. It plays a critical role in the **Fetch** stage of the instruction cycle.

- **Word Alignment** : **Word alignment** means that instructions are stored at memory addresses that are **multiples of 4 bytes** (i.e., 0x00000000, 0x00000004, 0x00000008, …).
- **Therefore, the lower 2 bits of the PC are always 00 in binary.**
- **Thus, we usually ignore the lower two bits.** In this design, we assume a fixed-size **Instruction Memory** capable of storing **256 instructions**.
- Total memory range: 0 to 255 words = 256 × 4 = 1024 bytes = 1 KB
- The PC can address locations from 0x00000000 to 0x000003FC (in 4-byte steps).
- Internally, instruction memory is indexed using:
  - instr_mem[PC[9:2]] // 8-bit word index

THE VERILOG CODE FOR THE INSTRUCTION MEMORY

```
1  `timescale 1ns/1ps
2  module instructionMemory(
3    input [31:0] addr,
4    output [31:0] instruction
5  );
6    reg [31:0] memory [0:255];
7    integer i;
8    assign instruction = memory[addr[9:2]];
9    initial begin
10     for (i = 0; i < 256; i = i + 1)
11       memory[i] = 32'h00000013;
12     $readmemh("test_program.hex", memory);
13   end
14 endmodule
15
```

- $readmemh statement reads the .hex file in which Instructions are stored in hex format (eg FE5098E3)
- Notice that the instructions are fetched Asynchronously. This enables zero-latency access.

DESIGNING THE REGISTER FILE

The Register File is kind of a collection of registers. It has 32 (0-31) registers inside it, these are also called General Purpose Registers (**GPR**). This Block has two Read Ports (**rs1**, **rs2**), one Write Port (rd). It accepts the Write Data (32 Bits) produced by the Data Memory, the **RegWrite** control Signal coming from the Control Unit and a clock.

Input	Width	Description
clk	**-**	**Clock signal. Writing to a register occurs on the rising edge.**
rs1	**5**	**Address of source register 1.**
rs2	**5**	**Address of source register 2.**
rd	**5**	**Address of destination register (write-back target).**
writeData	**32**	**Data to be written into register rd when RegWrite is high.**
regWrite	**1**	**Write enable signal. When high, data is written into register rd.**

Output	Width	Description
readData1 (RD1)	**32**	**Data read from register rs1**
readData2 (RD2)	**32**	**Data read from register rs2**

VERILOG CODE FOR THE REGISTER FILE

```
1  `timescale 1ns/1ps
2  module registerFile(
3    input clk,
4    input RegWrite,
5    input [4:0] rs1, rs2, rd,
6    input [31:0] writeData,
7    output [31:0] readData1, readData2
8  );
9    reg [31:0] registers[0:31];
10   integer i;
11   assign readData1 = (rs1 != 5'd0) ? registers[rs1] : 32'b0;
12   assign readData2 = (rs2 != 5'd0) ? registers[rs2] : 32'b0;
13   always @(posedge clk) begin
14     if (RegWrite && rd != 5'd0) begin
15       registers[rd] <= writeData;
16       //$display("RegWrite: x%0d <= %0d at time %0t", rd, writeData, $time);
17     end
18   end
19   initial begin
20     for (i = 0; i < 32; i = i + 1) begin
21       registers[i] = 32'b0;
22     end
23   end
24 endmodule
25
```

- The data is read asynchronously from rs1 and rs2 but only when rs != 0. But why? The reason is that the x0 (GPR0) is Hardwired to 32'b0 and any read from x0 always results in 32'b0
- The writeData is stored at the destination register only if rd != 5'd0 to protect register x0 to be written.
- At the start of the simulation, all the GPR are initialized to 32'b0 using initial block (non synthesizable).

DESIGNING THE DATA MEMORY

The Data Memory is a **RAM** with Asynchronous Read. In this design will choose a 1KB RAM.

It accepts MemWrite, MemRead from the Control Unit, writeData from the Register File (readData2 or RD2). The address comes from the ALU (Result).

Input	Width	Description
clk	**-**	**Clock input, used for synchronous writes**
addr	**32**	**Memory address (byte address)**
writeData	**32**	**Data to be written to memory (for store) when MemWrite is HIGH**
MemWrite	**1**	**When high, enables memory write**
MemRead	**1**	**When high, enables memory read**

Output	Width	Description
readData	**32**	**Data read from memory (for load)**

THE VERILOG CODE OF THE DATA MEMORY

```
1  `timescale 1ns/1ps
2  module dataMemory(
3    input clk,
4    input MemWrite, MemRead,
5    input [31:0] addr, writeData,
6    output [31:0] readData
7  );
8    reg [31:0] memory [0:255];
9    wire [7:0] mem_index = addr[9:2];
10   integer i;
11   always @(posedge clk) begin
12     if (MemWrite) begin
13       if (mem_index < 256) begin
14         memory[mem_index] <= writeData;
15       end else begin
16         $display("Write out-of-bounds: addr=0x%0h", addr);
17       end
18     end
19   end
20   assign readData = (MemRead && mem_index < 256) ? memory[mem_index] : 32'b0;
21   initial begin
22     for (i = 0; i < 256; i = i + 1) begin
23       memory[i] = 32'd0;
24     end
25   end
26   always @(*) begin
27     if (MemRead && mem_index < 256)
28       $display("MemRead: Mem[0x%0h] => %0d", addr, memory[mem_index]);
29   end
30 endmodule
31
```

- Again, the Word Align is used
- The Data is Read Asynchronously
- Some Debugging and Error Prevention statements have been added
- The Data Memory is initiliazed to zero to prevent showing xxx in the starting of the Simulation

DESIGNING THE PC ADDER AND THE MUX

The PC Adder is nothing but a Full Adder which adds a constant 4 (in decimal) to point at the next instruction.

VERILOG CODE FOR THE PC ADDER

```
1  `timescale 1ns/1ps
2  module pcAdder(
3      input [31:0] pc,
4      output [31:0] pc_next
5  );
6      assign pc_next = pc + 4;
7  endmodule
8
```

The MUX is a simply 2:1 MUX which selects one of its input depending upon the Selector signal.

VERILOG CODE FOR THE PC ADDER

```
1  `timescale 1ns/1ps
2  module mux2to1(
3      input [31:0] a, b,
4      input sel,
5      output [31:0] out
6  );
7      assign out = sel ? b : a;
8  endmodule
9
```

PUTTING ALL TOGETHER : DESIGNING THE TOP MODULE

The Top Module or the overall System (The Processor) will be designed according to the Block Diagram of the Overall Architecture (taken from the ***Computer Organization and Design*** **by David A. Patterson and John L. Hennessy**).

Just some minor changes will be done to add the jal, jalr, lui and auipc instructions. Also, we are not using the shift + 1 block since it will be handled in the Top Module.

The inputs of the Top Module i.e the Processor is clk and rst.

i) First, we will configure the Program Counter.

```
wire [31:0] PC, PC_next;
programCounter PC_reg(clk, rst, PC_next, PC);
```

ii) Next, we will connect the output of the PC to the PC Adder.

```
wire [31:0] PC_plus4;
pcAdder PCA(PC, PC_plus4);
```

iii) Next, the the instruction memory is to be configured.

```
wire [31:0] Instr;
instructionMemory IM(PC, Instr);
```

iv) Now, we will extract the rd, rs1, rs2, funct3, funct7 and opcode from the instruction.

```
wire [6:0] opcode = Instr[6:0];
wire [2:0] funct3 = Instr[14:12];
wire [6:0] funct7 = Instr[31:25];
wire [4:0] rs1 = Instr[19:15], rs2 = Instr[24:20], rd = Instr[11:7];
```

v) Next comes the Register File.

```
wire [31:0] RD1, RD2, WriteData;
wire RegWrite;
registerFile RF(clk, RegWrite, rs1, rs2, rd, WriteData, RD1, RD2);
```

THE TOP MODULE

vi) Now, we will set up the Sign Extender block.

```
wire [31:0] Imm;
signExtend SE(Instr, opcode, Imm);
```

vii) Next, the Control Unit.

```
wire Branch, MemRead, MemtoReg, MemWrite, ALUSrc;
wire Jal, Jalr, Lui, Auipc;
wire [1:0] ALUOp;
controlUnit CU(opcode, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite, Jal, Jalr, Lui, Auipc);
```

viii) Next, the ALU Control.

```
wire [3:0] ALUControl;
aluControl ALUCTRL(ALUOp, funct3, funct7, ALUControl);
```

ix) Next, the ALU MUX.

```
wire [31:0] ALU_in2;
mux2to1 MUX_ALU(RD2, Imm, ALUSrc, ALU_in2);
```

x) Next, the ALU.

```
wire [31:0] ALU_out;
wire zero;
alu ALU(RD1, ALU_in2, ALUControl, ALU_out, Zero);
```

xi) Now, the Data Memory.

```
wire [31:0] ReadData;
dataMemory DM(clk, MemWrite, MemRead, ALU_out, RD2, ReadData);
```

xii) Next, the Memory Mux.

```
wire [31:0] WriteData_from_mem;
mux2to1 MUX_MEM(ALU_out, ReadData, MemtoReg, WriteData_from_mem);
```

xiii) Now, we will tackle the jal, jalr, lui, auipc instructions. Recall that, for both jal and jalr instructions, we store PC + 4 (the return address) in the destination register rd. Also, we store PC and PC + imm in rd for lui and auipc respectively. So, for jal or jalr (i.e. jal | jalr) we store PC + 4, for lui and auipc we will store PC and PC + imm respectively.

THE TOP MODULE

```
wire [31:0] WriteData_from_jal = PC_plus4;
wire [31:0] WriteData_from_lui = Imm;
wire [31:0] WriteData_from_auipc = PC + Imm;
wire write_jal = Jal | Jalr;
wire write_lui = Lui;
wire write_auipc = Auipc;
assign WriteData = write_lui    ? WriteData_from_lui    :
                   write_auipc  ? WriteData_from_auipc  :
                   write_jal    ? WriteData_from_jal    :
                   MemtoReg     ? ReadData              :
                                  ALU_out;
```

For none of the lui, auipc, jal, jalr we will pass the ALU Result.

xiii) Finally, we will configure the Branching and Jumping logic.

For this, we have to recall **Branch** and **Jump** type instructions.
The **Branch** type instructions and **jal** loads the Program counter to point to some label (an address) which is basically doing the current address plus some immediate.

Again, for **jalr** we load the **PC** to point to the **RD1 + imm** (i.e. to the address stored in **rs1 + imm**)

For none of Branch, **jal** or **jalr** we simply loads the **PC** to **PC + 4** to point to the next instruction.

We will take branch only if Branch signal from the CU is high i.e. it is indeed a Branch Instruction and any of the **beq**, **bne**, **blt**, **bge**, **bltu** and **bgeu** is **TRUE**. Otherwise we will not take branch i.e. the condition is FALSE.

- **beq** is **TRUE** if **ALU Result** is **Zero** (since A – B = 0) and **funct3 = 000**
- **bne** is **TRUE** if **ALU Result** is not **Zero** (since A != B, A – B != 0) and **funct3 = 001**
- **blt** is **TRUE** if the **LSB** of the **ALU Result** = 1 (for comparison the whole ALU Result is 1 if A < B and the whole ALU Result is 0 if A > B, so, we are just choosing the **LSB**, you could choose the whole **ALU Result** if you want) and **funct3 = 100**
- **bge** is **TRUE** if the **LSB** of the **ALU Result** != 1 (i.e. A > B) and **funct3 = 101**
- **bltu** and **bgeu** is **TRUE** just like **blt** and **bge** respectively and **funct3 = 110** and **111** respectively

THE TOP MODULE

So, from the above discussion, we will load the PC accordingly.

- For jalr, we have ANDed 32'hfffffffe to ensure word alignment i.e. to make sure that the two LSB bits of the jalr_target is indeed 0.

```
wire [31:0] PC_branch, PC_jalr_target, PC_jal_target;
assign PC_branch = PC + Imm;
assign PC_jal_target = PC + Imm;
assign PC_jalr_target = (RD1 + Imm) & 32'hfffffffe;//To ensure Word Alignment i.e. the two
LSBs must be zero
wire slt_result = ALU_out[0];
wire takeBranch = (Branch && (
                    (funct3 == 3'b000 &&  Zero) ||     // BEQ
                    (funct3 == 3'b001 && !Zero) ||     // BNE
                    (funct3 == 3'b100 &&  slt_result) || // BLT
                    (funct3 == 3'b101 && !slt_result) || // BGE
                    (funct3 == 3'b110 &&  slt_result) || // BLTU (using ALU[0] for comparison
                    result)
                    (funct3 == 3'b111 && !slt_result)    // BGEU
                  ));
assign PC_next = Jalr        ? PC_jalr_target :
                 Jal         ? PC_jal_target  :
                 takeBranch  ? PC_branch      :
                               PC_plus4;
```

We have designed the Top Module following the classic Diagram of the RISCV Architecture, step by step, module by module. In many cases, we have used redundant wires for education purpose only.

TESTING AND SIMULATION

We have written all the Verilog Codes in the EDA Playground for sharing purpose. The Processor has been tested with different programs like Fibonacci Series, GCD, Bubble Sort, Sum of N natural number etc. Steps to run a program using this Processor :

i) Write the program in Assembly Language.

ii) Use an Assembler (RISCV toolchain) to get the .hex file.

iii) Make a .hex file and change the name of the file in the instructionMemory.v or just paste the contents of your .hex file in the test_program.hex

iv) Just Save and Run the Simulation!

v) A test bench is already provided

Link to EDA Playground : RISCV Single Cycle Processor - EDA Playground

Links to my GitHub :

i) Verilog Based Implementation : https://github.com/Anish-Rooj-cpu/Single-Cycle-RISCV-Processor-using-Verilog-HDL

ii) Digital Software Based Implementation (way cooler and you can easily see what really is going on inside the Processor!) : https://github.com/Anish-Rooj-cpu/Single-Cycle-RISCV-Processor-using-Digital-Software

Finally, let's connect on LinkedIn : www.linkedin.com/in/anish-rooj-6b6746275

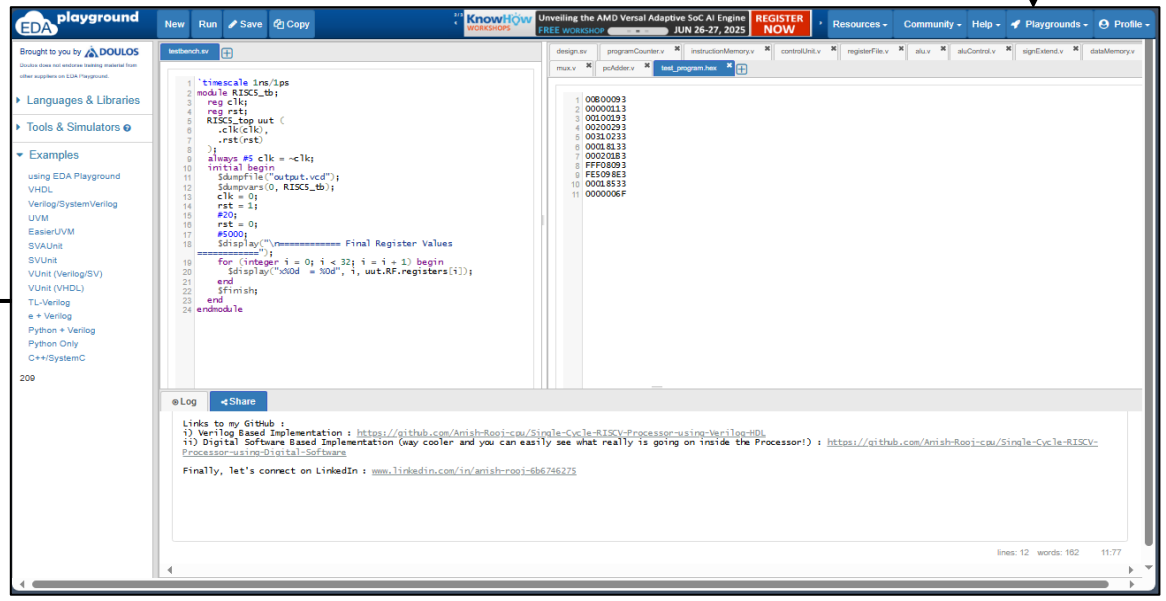
A TEST PROGRAM

```
# Fibonacci Program - F(10) stored in x10
    addi x1, x0, 11     # x1 = 11 (loop counter: we compute F(10), so 10 iterations after 0,1)
    addi x2, x0, 0      # x2 = 0  (F(0))
    addi x3, x0, 1      # x3 = 1  (F(1))
    addi x5, x0, 2      # x5 = 2  (loop exit condition: when x1 == 2)

loop:
    add x4, x2, x3      # x4 = x2 + x3 (next Fibonacci number)
    add x2, x3, x0      # x2 = previous x3
    add x3, x4, x0      # x3 = previous x4 (new Fib)
    addi x1, x1, -1     # x1 = x1 - 1
    bne x1, x5, loop    # if x1 != 2, continue loop

# x3 now holds F(10) = 55
# store result and halt
    add x10, x3, x0     # x10 = F(10)
done:
    jal x0, done        # infinite loop to end program
```

fibo.hex

```
00B00093
00000113
00100193
00200293
00310233
00018133
000201B3
FFF08093
FE5098E3
00018533
0000006F
```

```
============ Final Register Values =============
x0   = 0
x1   = 2
x2   = 34
x3   = 55
x4   = 55
x5   = 2
x6   = 0
x7   = 0
x8   = 0
x9   = 0
x10  = 55
x11  = 0
x12  = 0
x13  = 0
x14  = 0
x15  = 0
x16  = 0
x17  = 0
x18  = 0
x19  = 0
x20  = 0
x21  = 0
x22  = 0
x23  = 0
x24  = 0
x25  = 0
x26  = 0
x27  = 0
x28  = 0
x29  = 0
x30  = 0
x31  = 0
testbench.sv:22: $finish called at 5020000 (1ps)
```



REFERENCES

1. David A. Patterson, John L. Hennessy

Computer Organization and Design: The Hardware/Software Interface (5th Edition, RISC-V Edition)

2. RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.2

https://riscv.org/specifications/

Official specification for RV32I base integer instruction set architecture. Critical for instruction formats and behavior.

3. Wikipedia Contributors RISC-V – Wikipedia Entry : https://en.wikipedia.org/wiki/RISC-V

Provides a quick, readable summary of the RISC-V architecture, licensing, and ecosystem.

4. Fraser Innovations Blog RISC-V Instruction Set Explanation

https://fraserinnovations.com/risc-v/risc-v-instruction-set-explanation/

5. Verilog HDL by Samir Palnitkar