IPA Project Report

Team : 19

- Gangineni Yaswanth
  - 2023102007
- Nethavath Praveen
  - 2023102013
- Kotha Phanindra Manoj Kumar
  - 2023102008

Sequential Implementation

Introduction

Processor architecture studies the Design and Implementation of Processors to execute instruc tions efficiently. Among Modern architectures, RISC-V is an open-source, Reduced Instruction Set Computing (RISC) Architecture that has gained popularity for its simplicity, modularity, and extensibility

RISC-V Instruction Set Architecture

RISC-V is a Load-Store architecture with a fixed instruction length of 32 bits in its base form. It supports several Instruction Formats including

- R-Type
- I-Type
- S-Type
- B-Type
- U-Type
- J-Type
-

- Control signals derived from instruction

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

The RISC-V Instruction Set Architecture consists of the following key instruction categories:

- Arithmetic and Logical Instructions: Addition, subtraction, Bit-wise operations.
- Load and Store Instructions: Memory access operations.
- Control Transfer Instructions: Conditional and unconditional jumps.
- Floating Point Instructions: Operations on floating-point registers (optional extension).

- System Instructions: Environment and privilege level management.

Processor Design

Processor architecture can be classified into two main types: Sequential and Pipelined.
Sequential Processors execute one instruction at a time, while Pipelined Processors divide the in struction execution into multiple stages and overlap the execution of different instructions. Pipelined Processors can achieve higher performance and efficiency than sequential processors. Still, they also introduce challenges such as pipeline hazards, which occur when the execution of one instruction depends on the outcome of another instruction still in the pipeline.

Arithmetic Logic Unit :

We designed/constructed a Arithmetic Logic Unit that implements following operations

1. Addition : 64 bit-wise
2. Subtraction : 64 bit-wise
3. AND : 64 bit-wise
4. OR : 64 bit-wise

Arithmetic Logic Unit receives control signal ALUControl(4-bit value) and produce Result and Zero bit . Here Zero bit is set to 1'b1 when the result of Arithmetic Logic Unit is 64'd0

ALU Control :

ALU Control	Operation
0000	AND
0001	OR
0010	Add
0110	Sub

Subtraction is implemented using 2's-complement logic . Which implies if Operation is A- B then A +(2's Complement(B)). 2's Complement of a 64-bit number can be achieved by adding 1'b1 to inverted number. So we simply Invert B and use addition logic with carry-in as 1'b1 .

Sequential Part

The sequential processor implementation is based on a single-cycle architecture where each in struction is fetched, decoded, executed, and its result written back before the next instruction is processed. Although this design is less efficient compared to pipelined designs, the sequential design serves as a baseline to ensure correct execution of instructions and a clear understanding of the architecture.

Key Components of the Sequential RISC-V Processor

1. **Program Counter (PC):** Holds the address of the next instruction to fetch. Increments by the instruction size (typically 4 bytes) and updates for branch instructions.
2. **Instruction Memory:** Stores the program instructions. Uses the PC to output the corre sponding instruction during the fetch phase.
3. **Register File:** Contains 32 registers (x0 to x31), with x0 hardwired to zero. Provides two read ports and one write port for operands and results.
4. **Arithmetic Logic Unit (ALU):** Performs arithmetic (e.g., addition, subtraction) and log ical (e.g., AND, OR) operations. Also computes effective addresses for load and store opera

tions and generates a zero flag for branch decisions.

5. **Data Memory:** Used for load (ld) and store (sd) instructions. The effective address calcu lated by the ALU is used for accessing the memory.
6. **Control Unit:** Decodes instructions and generates the necessary control signals. It directs data flow, selects ALU operations, manages memory read/write, and handles branch decision logic.

Instruction Execution Flow

The sequential execution flow for each instruction in the processor consists of the following steps :

1. **Instruction Fetch:**

   The instruction at the address specified by the Program Counter (PC) is fetched from the Instruction Memory.

2. **Instruction Decode and Operand Fetch:**

   The fetched instruction is decoded by the Control Unit to determine the operation type (R-type, I-type, S-type, or B-type) and to extract operand registers and immediate values. The required operands are read from the Register File. For branch instructions (e.g., beq), the operands are fetched for comparison.

3. **Execution:**

   The Arithmetic Logic Unit (ALU) performs the necessary arithmetic or logical operation:

   – For arithmetic/logic instructions (e.g., add, sub, and, or), the ALU computes the result.

   – For load/store instructions (ld, sd), the ALU calculates the effective memory ad dress by adding the base register to the immediate offset.

   – For the branch instruction (beq), the ALU checks if the operands are equal (by subtracting and verifying if the result is zero).

4. **Memory Access:**

   Depending on the instruction type:

   – Load (ld): The computed effective address is used to read data from Data Memory.

   – Store (sd): The computed effective address is used to write data from the Register File to Data Memory.

   – For other instructions, no memory access is performed.

5. **Write-Back:**

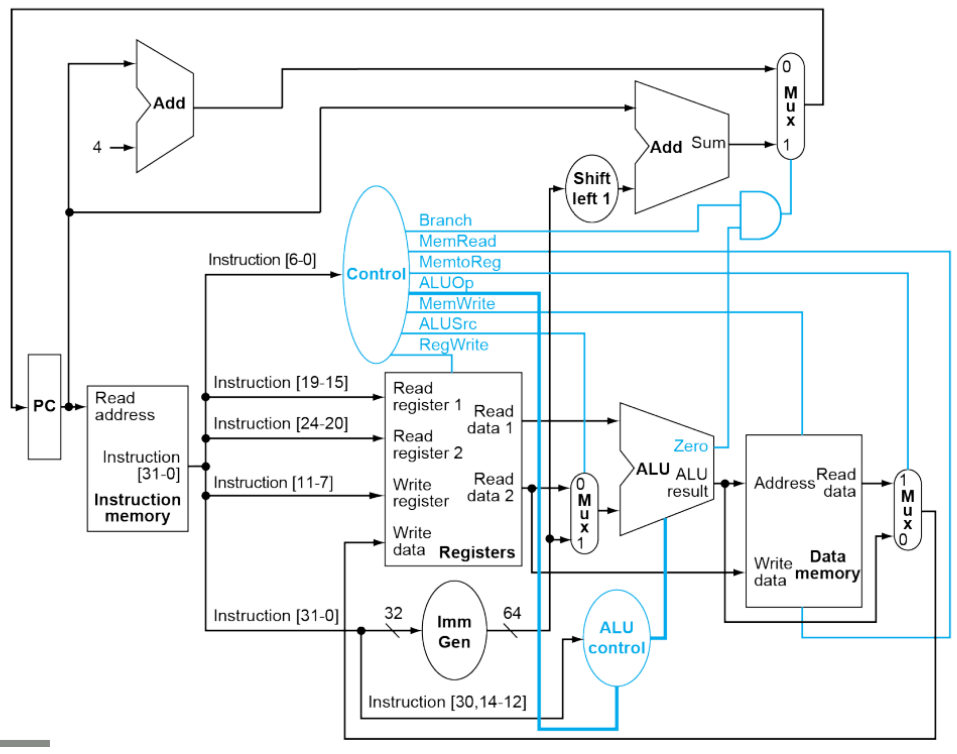
   The result from the ALU or data fetched from memory (in the case of ld) is written back to the destination register in the Register File.

6. **PC Update:**

   The Program Counter (PC) is updated to the next instruction:

   – Normally, the PC increments by 4.

   – For branch instructions (e.g., beq), if the branch condition is met, the PC is updated with the computed branch target address.



the above is the datapath for sequentional implementation

Explanation of each Module implemented in Verilog

Instruction Fetch (Stage 1)

1. Instruction_memory.v

Input: 64-bit pc value which represents the address of the memory

In the instruction memory module we created, the memory size is 1024 rows of 8-bit elements. Since each row represents 8 bits and an instruction is 32 bits, we need to fetch 4 elements from the array to retrieve a complete 32-bit instruction.

```
//this the main block block in the code to fetch the 32 bit instruction
always @(*) begin
      instruction = {memory[A3], memory[A2], memory[A1], memory[PC]};
  end
```

Instruction Decode(Stage-2)

1. Control Module:

It extracts the **opcode** from the instruction and determines the instruction type (R-type, I-type, S-type, B-type, etc.).

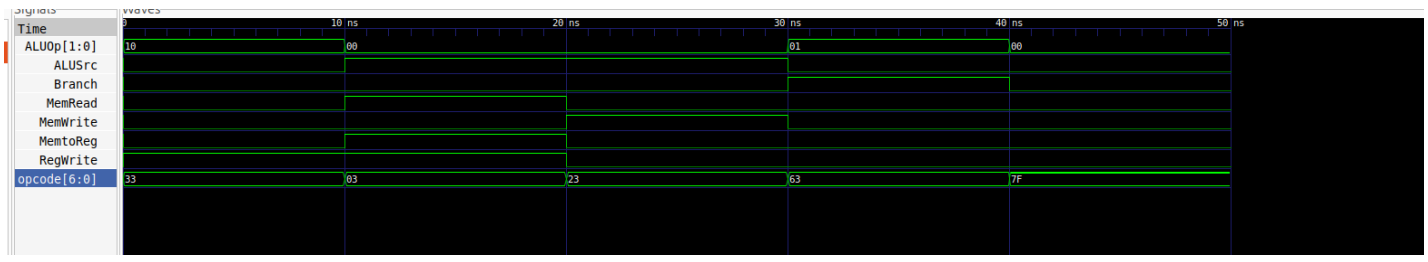
control.v

Inputs: Instruction (32 bits): The fetched instruction word whose opcode bits are used to generate control signals.

Outputs:

- **Register Write Enable (`RegWrite`)** – Determines whether the register file should be updated.
- **ALU Operation (`ALUOp`)** – Defines the operation the ALU should perform.
- **ALU Source (`ALUSrc`)** – Selects whether the second ALU operand comes from a register or immediate value.
- **Memory Read (`MemRead`)** – Indicates if data should be read from memory (for `ld` instructions).
- **Memory Write (`MemWrite`)** – Indicates if data should be written to memory (for `sd` instructions).
- **Memory to Register (`MemtoReg`)** – Selects whether the ALU result or memory data is written to a register.
- **Branch (`Branch`)** – Activates branching logic for conditional branches (`beq`).

gtkwave for sample testcases



register.v

Inputs:

- **Registers Declaration:**
  - `registers` is a 2D array that stores **32 registers** (`registers[31:0]`), each **64 bits wide**.
- **Register Initialization:**
  - `registers[0]` is always **0** (as per RISC-V convention).
  - Other registers have assigned values for testing.
- **Reading Registers:**
  - Two registers can be read simultaneously via **read_reg1** and **read_reg2**.
  - The values are **combinationally assigned** to `read_data1` and `read_data2`.
- **Writing to Registers:**
  - Writing occurs **on the positive clock edge**.
  - If `RegWrite` is **enabled** and the destination register is **not zero**, the value is updated.

ImmediateGenerator Module (immed.v)

The **Immediate Extractor Module** extracts and sign-extends the immediate values from **RISC-V instructions**. These immediate values are required for certain instruction types, such as **Load (LD), Store (SD), and Branch (BEQ)**.

-Functionality:

- The module takes a **32-bit instruction** as input.
- It extracts the **immediate value** based on the **opcode**.
- The extracted immediate is then **sign-extended to 64 bits** to match the processor's data width.

Inputs & Outputs:

Signal	Direction	Width	Description
`instruction`	Input	32 bits	The RISC-V instruction.
`imm64`	Output	64 bits	The extracted immediate value (sign-extended).

-Immediate Extraction Based on Instruction Type:

Instruction Type	Opcode	Immediate Extraction	Sign-Extension
Store (SD)	`7'b0100011`	`{instruction[31:25], instruction[11:7]}`	**From bit 31**
Load (LD)	`7'b0000011`	`{instruction[31:20]}`	**From bit 31**
Branch (BEQ)	`7'b1100011`	`{instruction[31:25], instruction[11:7]}`	**From bit 31**

- If the opcode does not match the supported types, the immediate value is set to **zero (64'd0)**.
- The extracted immediate values are used in **memory addressing (LD/SD)** and **branch offset calculations (BEQ)**.

Execution Stage(Stage-3):

-Alu Control

The **ALU Control module** determines the specific ALU operation based on the **ALUOp** signal and the **funct** field from the instruction. This module is a crucial part of the processor, as it ensures that the ALU performs the correct operation based on the instruction type.

Inputs and Outputs

Signal	Direction	Width	Description
`ALUOp`	Input	2 bits	Control signal from the main control unit that categorizes the instruction type.
`funct`	Input	4 bits	Function field from the instruction, which helps determine the exact ALU operation.
`ALUCtrl`	Output	4 bits	Control signal sent to the ALU to select the required operation.

1. The **ALUOp** signal is received from the **main control unit**.
2. The ALU Control module **decides the ALU operation** based on `ALUOp`:
   - If **ALUOp = 2'b00**, set `ALUCtrl = 4'b0010` (for load/store operations).
   - If **ALUOp = 2'b01**, set `ALUCtrl = 4'b0110` (for branch comparison).
   - If **ALUOp = 2'b10**, use `funct` to decide the exact operation:
     - `funct = 4'b1000` → Subtraction
     - `funct = 4'b0000` → Addition
     - `funct = 4'b0111` → AND
     - `funct = 4'b0110` → OR
3. The selected ALU operation is then sent as `ALUCtrl` to the **ALU module**.

It acts as a bridge between the **Control Unit** and the **ALU**.

-MUX:A **multiplexer (MUX)** is a combinational circuit that selects one of the multiple input signals and forwards it to the output based on a control signal (`sel`). In this case, the module is a **2-to-1 multiplexer** that selects between two **64-bit inputs (A and B)** based on the `sel` signal.

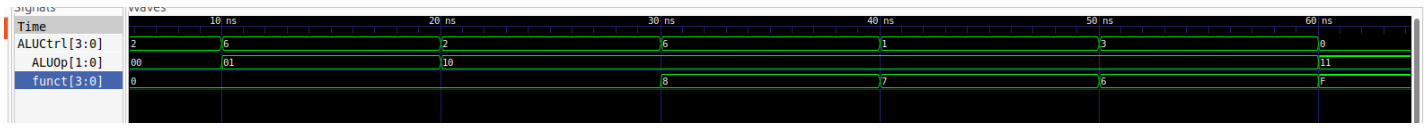
- If `sel = 0`, output `Y = A`
- If `sel = 1`, output `Y = B`
- The output `Y` is assigned using a **ternary operator (`?:`)**, which acts as a conditional selector.

Here we are using mux for selecting rs2 in some instruction types the rs2 will be selected from immediate and here the select line is AluSrc which selects second Alu operand comes from register.

-Execute(ALU Stage):

This ALU module performs arithmetic and logical operations on **64-bit** inputs based on a **4-bit control signal (`Alu_control`)**. It supports addition, subtraction, AND, OR operations, and outputs the result along with status flags (`zero` and `overflow`).

sample test cases



---

1. Inputs & Outputs

Signal	Direction	Width	Description
`a`	Input	64 bits	First operand
`b`	Input	64 bits	Second operand
`Alu_control`	Input	4 bits	Selects the ALU operation
`result`	Output	64 bits	Output of the ALU operation

`zero`	Output	1 bit	Flag set to 1 if `result` is zero
`overflow`	Output	1 bit	Flag set if overflow occurs in signed arithmetic

Functionalities

The ALU performs different operations based on the **Alu_control** input. Below are the supported operations:

A. Arithmetic Operations

1. **Addition (`4'b0010`)**
   - Adds `a` and `b` using `adder_64bit` module.
   - Overflow flag is set if there's an overflow.
   - Example:
     - `a = 5, b = 3` → `result = 8`
2. **Subtraction (`4'b0110`)**
   - Computes `a - b` using `Subtractor_64bit` module.
   - Sets `zero = 1` if the result is **0**.
   - Overflow flag is set if signed overflow occurs.
   - Example:
     - `a = 7, b = 7` → `result = 0, zero = 1`

B. Logical Operations

1. **Bitwise AND (`4'b0000`)**
   - Computes `result = a & b`.
   - Example:
     - `a = 1101, b = 1010` → `result = 1000`
2. **Bitwise OR (`4'b0001`)**
   - Computes `result = a | b`.
   - Example:
     - `a = 1101, b = 1010` → `result = 1111`

C. Default Operation

1. **Default Case (Unknown Operation)**
   - If an unsupported `Alu_control` value is provided, ALU sets `result = 0`, `overflow = 0`, and `zero = 0`.

3. Breakdown of Each Module in ALU

(A) `adder_64bit` Module

- **Performs 64-bit addition** using `FullAdder` instances.
- **Detects overflow** by comparing the carry-in and carry-out of the last bit.
- **Outputs:** `sum_result`, `sum_cout`, `sum_overflow`.

(B) `Subtractor_64bit` Module

- **Performs 64-bit subtraction** by adding `a` and `~b` (two's complement method).
- Uses **OverflowDetector** to check for overflow.
- **Outputs:** `diff_result`, `diff_cout`, `diff_overflow`.

(C) `And_64bit` Module

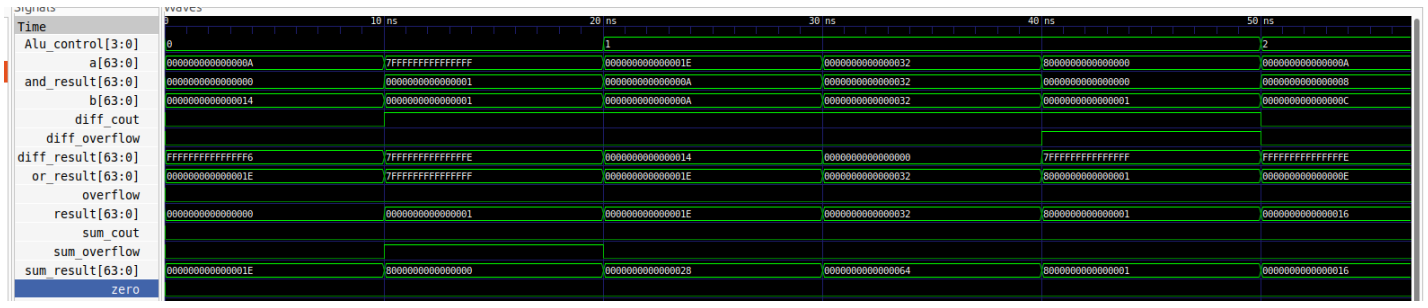
- **Performs bitwise AND** between `a` and `b`.

- Uses a **generate block** to apply AND operation to all 64 bits.

(D) `Or_64bit` Module

- **Performs bitwise OR** between `a` and `b`.
- Uses a **generate block** to apply OR operation to all 64 bits.

gtkwave for alu for sample testcases



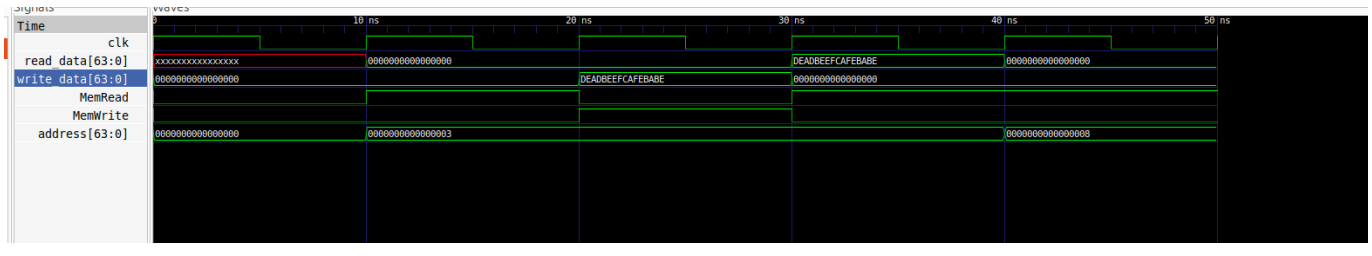
Data Memory .v(stage-4)

Data memory in a processor is used to store and retrieve data during program execution. It is a critical component of a processor's memory hierarchy and interacts with the control unit and arithmetic logic unit (ALU). In a RISC-V processor, data memory is used to implement **load (`ld`) and store (`sd`)** instructions, which allow the processor to read from and write to memory.

Functionality of the Data Memory Module

1. **Memory Storage**
   - The module defines a **256-entry memory array**, each holding a **64-bit word**. This represents a simple, small-scale memory for a processor.
2. **Inputs**
   - `MemRead`: A control signal that enables reading data from memory.
   - `MemWrite`: A control signal that enables writing data into memory.
   - `address [63:0]`: The memory address where the data is read from or written to. The lower 10 bits (`address[9:0]`) are used for indexing into the memory array (since 2^{10} = 1024, but only 256 locations are defined).
   - `write_data [63:0]`: The 64-bit data to be stored in memory when `MemWrite` is active.
3. **Outputs**
   - `read_data [63:0]`: The 64-bit data retrieved from memory when `MemRead` is active.
4. **Operations**
   - **Write Operation** (`MemWrite == 1`):
     - If `MemWrite` is asserted, the value in `write_data` is stored at the specified memory `address[9:0]`.
   - **Read Operation** (`MemRead == 1`):
     - If `MemRead` is asserted, the memory retrieves the value at `address[9:0]` and assigns it to `read_data`.
5. **Memory Initialization**
   - The `initial` block initializes memory locations to zero. In real hardware, memory is either preloaded with a program or initialized dynamically

gtk wave for sample test cases



Next PC

The **PC Next Module** (`new_pc`) is responsible for calculating the next program counter (`PC`) value in a **RISC-V processor**. It determines whether the processor should execute the next sequential instruction or jump to a branch target based on branch conditions.

Functionality of the `new_pc` Module

Inputs:

1. `Branch` (1-bit):
   - Indicates whether the instruction is a branch instruction.
2. `Zero` (1-bit):
   - Comes from the ALU, indicating whether the comparison resulted in zero (e.g., `beq` checks if two registers are equal).
3. `imm` (64-bit signed):
   - The immediate offset used for branch target computation. It represents how far to jump from the current `PC`.
4. `PC` (64-bit):
   - The current program counter value (address of the current instruction).

Outputs:

- `new_PC` (64-bit):
  - The next PC value, either the next sequential instruction (`PC + 4`) or a branch target address.
- **Calculate Sequential Next PC (`pc_plus_4`)**
  - Since RISC-V uses a **4-byte instruction width**, the next instruction is at `PC + 4`.

```
verilog
CopyEdit
assign pc_plus_4 = PC + 64'd4;
```

- **Compute Branch Target Address (`branch_target`)**
  - The branch target is calculated as `PC + (imm << 1)`.
  - RISC-V branch immediates are **signed values** in instruction encoding, and they are **word-aligned**, meaning they need to be left-shifted by 1 bit.

```
verilog
CopyEdit
assign branch_target = PC + (imm << 1);
```

- **Determine if a Branch is Taken (`branch_taken`)**
  - A branch is only taken if the instruction is a **branch** (`Branch` is 1) **AND** the condition (`Zero`) is met.

```
verilog
CopyEdit
assign branch_taken = Branch & Zero;
```

- **Select the Next PC (`new_PC`)**
  - If `branch_taken` is true, the new PC will be the branch target.
  - Otherwise, it moves to the next sequential instruction (`PC + 4`).

```
verilog
CopyEdit
always @(*) begin
    new_PC = branch_taken ? branch_target : pc_plus_4;
end
```

WB-(write Back Stage) (Stage-5)

Selecting the Write-Back Data (`final_write_data`)

```
wire signed [63:0] final_write_data;
mux write_data_mux(
    .A(ALU_result),
    .B(write_data1),
    .sel(MemtoReg),
    .Y(final_write_data)
);
```

- This **multiplexer (`mux`)** chooses between two sources for writing back to the register file:
  - `ALU_result` → The output of the ALU (for instructions like `add`, `sub`, `and`, `or`).
  - `write_data1` → Data from memory (for load instructions like `ld`).
  - `MemtoReg` (selector signal) decides:
    - `MemtoReg = 0`: Write ALU result.
    - `MemtoReg = 1`: Write memory load data.

2. Writing Data to Register File

```
verilog
CopyEdit
always @(posedge clk) begin
    if (RegWrite1 && current_instruction[11:7] != 5'b00000) begin
        register.registers[current_instruction[11:7]] <= final_write_data;
    end
end
```

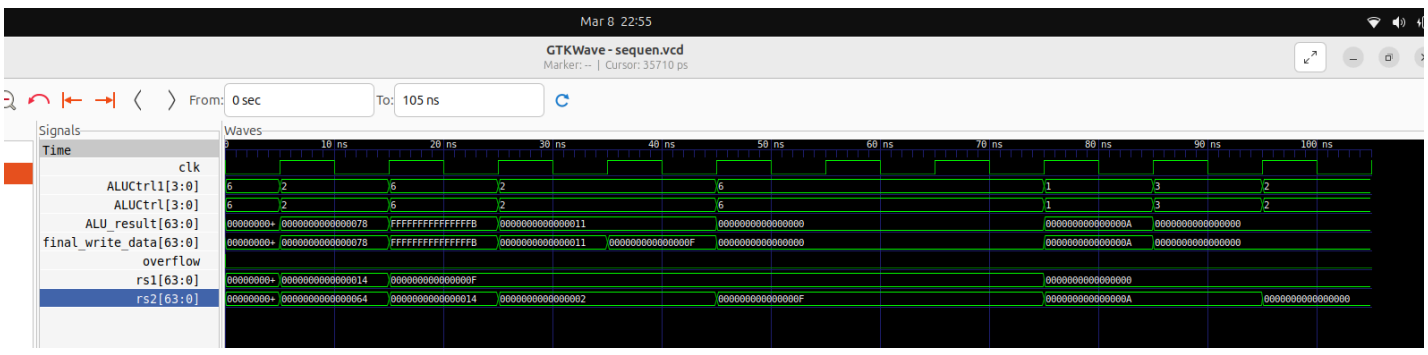
- `RegWrite1` **(control signal)**
  - When `RegWrite1` is **1**, it means we need to write data back to a register.
  - Used for **R-type (add, sub, and, or)** and **load (ld)** instructions.
  - Not used for `sd` (store) or `beq` (branch), as these don't write back.
- **Checking** `current_instruction[11:7] != 5'b00000`
  - Ensures that **register x0 is never written to** (RISC-V convention: `x0` is always 0).
- **Data is written to** `register.registers[current_instruction[11:7]]`
  - `current_instruction[11:7]` represents the **destination register (`rd`)** field.
  - The `final_write_data` (ALU result or memory data) is stored into this register.

txt:

```
        memory[0] = 8'h63;
        memory[1] = 8'h84;
        memory[2] = 8'h62;
```

```
        memory[3] = 8'h02;
        memory[4]=8'h7F;
        memory[5]=8'h00;
        memory[6]=8'h00;
        memory[7]=8'h00;
        memory[80] = 8'hB3;
        memory[81] = 8'h03;
        memory[82] = 8'h94;
        memory[83] = 8'h00;
        memory[84] = 8'h63;
        memory[85] = 8'h07;
        memory[86] = 8'h83;
        memory[87] = 8'hFE;
        memory[88] = 8'h23;
        memory[89] = 8'hB1;
        memory[90] = 8'h62;
        memory[91] = 8'h00;
        memory[92] = 8'h03;
        memory[93] = 8'hB4;
        memory[94] = 8'h22;
        memory[95] = 8'h00;
        memory[96] = 8'h63;
        memory[97] = 8'h0D;
        memory[98] = 8'h83;
        memory[99] = 8'hFE;
        memory[48] = 8'hB3;
        memory[49] = 8'h04;
        memory[50] = 8'h83;
        memory[51] = 8'h40;
        memory[52] = 8'h33;
        memory[53] = 8'hF5;
        memory[54] = 8'h44;
        memory[55] = 8'h00;
        memory[56] = 8'hB3;
        memory[57] = 8'hE1;
        memory[58] = 8'hA4;
        memory[59] = 8'h00;
        memory[60]=8'h7F;
        memory[61]=8'h00;
        memory[62]=8'h00;
        memory[63]=8'h00;

    registers[0]  = 64'h0000000000000000;
    registers[1]  = 64'h0000000000000001;
    registers[2]  = 64'd5;
    registers[3]  = 64'h0000000000000003;
    registers[4]  = 64'd10;
    registers[5]  = 64'd15;
    registers[6]  = 64'd15;
    registers[7]  = 64'h0000000000000000;
    registers[8]  = 64'd20;
    registers[9]  = 64'd100;
    registers[10] = 64'd0;
    registers[11] = 64'd0;
    registers[12] = 64'd16;
    registers[13] = 64'd19;
    registers[14] = 64'h0000000000000000;
    registers[15] = 64'h0000000000000000;
    registers[16] = 64'h0000000000000000;
    registers[17] = 64'h0000000000000000;
```

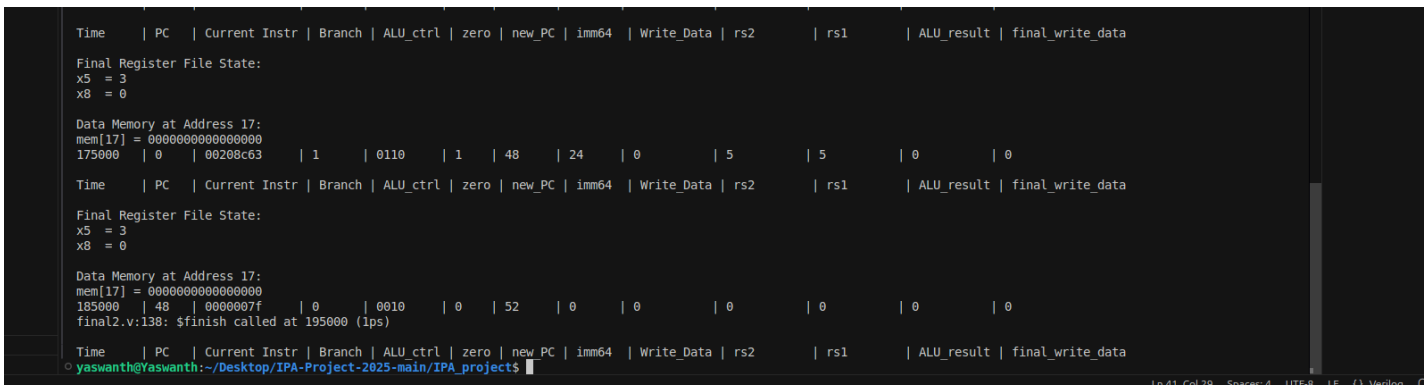
```
        registers[18] = 64'h0000000000000000;
        registers[19] = 64'h0000000000000000;
        registers[20] = 64'h0000000000000000;
        registers[21] = 64'h0000000000000000;
        registers[22] = 64'h0000000000000000;
        registers[23] = 64'h0000000000000000;
        registers[24] = 64'h0000000000000000;
        registers[25] = 64'h0000000000000000;
        registers[26] = 64'h000000000000001A;
        registers[27] = 64'h000000000000001B;
        registers[28] = 64'h000000000000001C;
        registers[29] = 64'h000000000000001D;
        registers[30] = 64'h000000000000001E;
        registers[31] = 64'h000000000000001F;

        //beq x6,x5,40 //00
        // exit // 4
        //sub x9 ×6 ×8 //48
        // or x10 ×4 ×9 //52
        // and x3 ×10 ×9 //56
        // exit //60
        //add x7 ×8 ×9 //80
        //beq x8 ×6 -18 //84
        //sd x6 2(x5) // 88
        // ld x8 2(x5)
        // beq x6 ×8 -4 //92
```

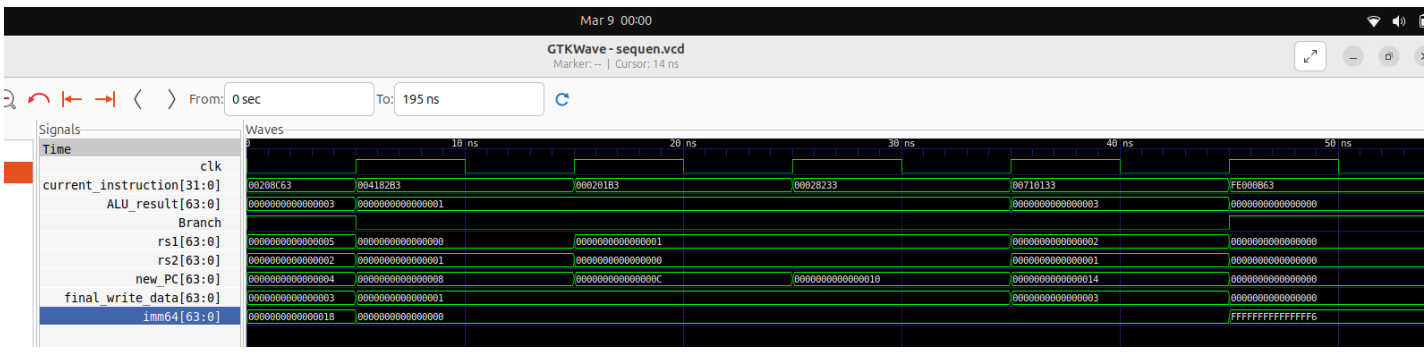


fibanocii txt

```
registers[0]  = 64'd0;
    registers[1]  = 64'd5;//n=5
     registers[2]  = 64'd2; //i=2
     registers[3]  = 64'd0;// fib(1)
     registers[4]  = 64'd1;//fib(2)
     registers[5]  = 64'd0;//result
   //fib(5)=0,1,1,2,3
   //so output x5=3
```

output:




Mar 9 00:00
GTKWave - sequen.vcd
Marker: -- | Cursor: 14 ns
From: 0 sec
To: 195 ns
Signals
Waves
Time
clk
current_instruction[31:0]
ALU_result[63:0]
Branch
rs1[63:0]
rs2[63:0]
new_PC[63:0]
final_write_data[63:0]
imm64[63:0]
10 ns
20 ns
30 ns
40 ns
50 ns
00208C63
004182B3
000201B3
00028233
00710133
FE000B63
0000000000000003
0000000000000001
0000000000000003
0000000000000000
0000000000000005
0000000000000000
0000000000000001
0000000000000002
0000000000000000
0000000000000002
0000000000000001
0000000000000000
0000000000000001
0000000000000000
0000000000000004
0000000000000008
000000000000000C
0000000000000010
0000000000000014
0000000000000000
0000000000000003
0000000000000001
0000000000000003
0000000000000000
0000000000000018
0000000000000000
FFFFFFFFFFFFFFF6