Vietnam National University Ho Chi Minh City

Ho Chi Minh City University of Technology

Department of Electronics

EE3043: Computer Architecture

Milestone 2 Report

Student
Pham Quang Anh — 2051034
Phan Quang Minh — 2051052
Vo Viet Hung — 2051076

Supervisor
Dr. Linh Tran Hoang

11th October 2023

Table of Contents

Milestone 2

Design of a Single-Cycle Processor

1.Introduction

In Milestone2, the purpose of this project is to create the (RISC-V) single-cycle RV32I processor that can perform some sort of operations, functions in order to understand how RV32I works and some of its basic applications via the DE2 kit.

In this report, our team confirmed that we have checked most of the criteria that are required throughout the project, especially the construction of each block and waveform's performance. This single-cycle processor can functionally perform most of the instructions in RV32I ISA (RV32I Instruction Set Architecture) and have not completely run programs inputted in the Instruction memory with some unexpected unknown.

2. Design Strategy

2.1. ALU (ARITHMETIC LOGIC UNIT)

Description:

To perform operation in the processor. According to the fetched instruction, the ALU component will perform different functions.

Specification:

Module name: alu_component

Signal	Type	Size	Description
operand_a	Input	32	The first operand – rs1.
operand_b	Input	32	The second operand – rs2.
alu_op	Input	4	An operation that the ALU has to perform.
alu_data	Output	32	The result of the operation.

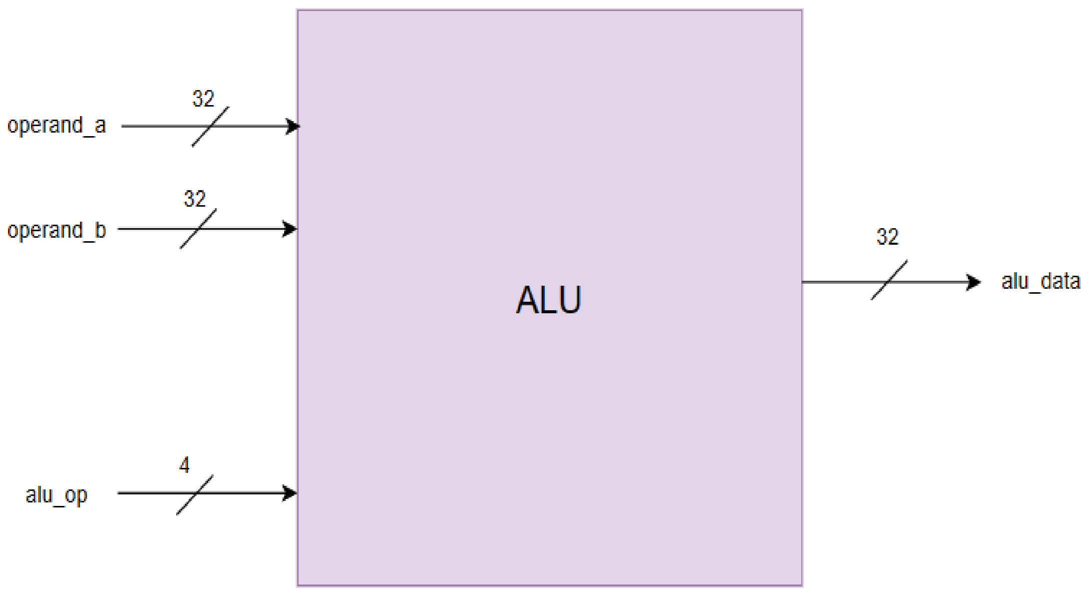
Strategy:

For each operation in the ALU component, we build the module that will perform the specific function and use the MUX to select the desired output for processor operation.

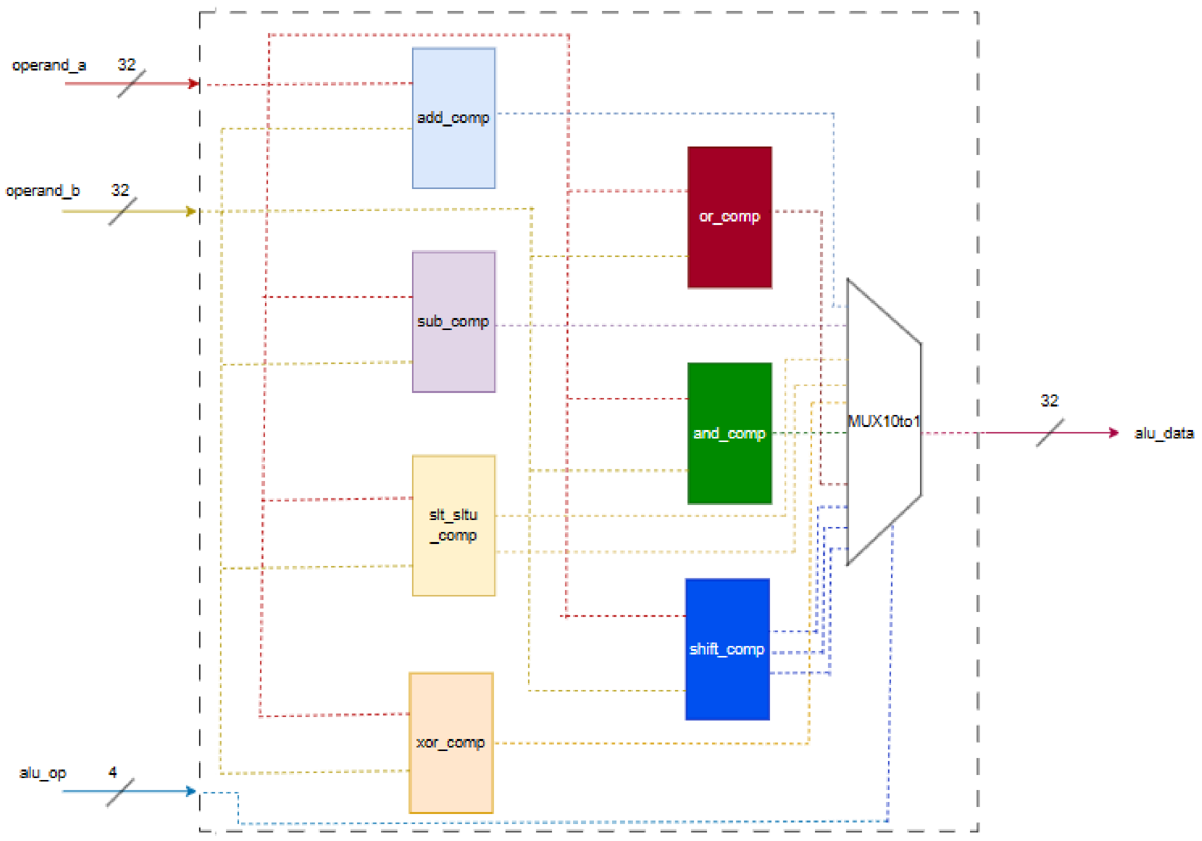
ALU Option:

Select signal	alu_op	Description (R-type)	Description (I-type)
0000	**ADD**	rd = rs1 + rs2	rd = rs1 + imm
0001	**SUB**	rd = rs1 - rs2	rd = rs1 - imm
0010	**SLT**	rd = (rs1 < rs2)? 1: 0	rd = (rs1 > imm)? 1: 0
0011	**SLTU**	rd = (rs1 < rs2)? 1: 0	rd = (rs1 > imm)? 1: 0
0100	**XOR**	rd = rs1 ⊕ rs2	rd = rs1 ⊕ imm
0101	**OR**	rd = rs1 ⋁ rs2	rd = rs1 ⋁ imm
0110	**AND**	rd = rs1 ⋀ rs2	rd = rs1 ⋀ imm
0111	**SLL**	rd = rs1 << rs2[4 : 0]	rd = rs1 << imm[4 : 0]
1000	**SRL**	rd = rs1 >> rs2[4 : 0]	rd = rs1 >> imm[4 : 0]
1001	**SRA**	rd = rs1 >>> rs2[4 : 0]	rd = rs1 >>> imm[4 : 0]

Block Diagram:



General block diagram of the ALU



Detailed Block Diagram of the ALU.

Components	Input	Output	Description
add_com	operand_a operand_b	add	add = operand_a + operand_b
sub_com	operand_a operand_b	sub	add = operand_a - operand_b

slt_sltu_com	operand_a
	
	operand_b
		slt
		
		sltu
			slt = (operand_a < operand_b)? 1: 0
			
			sltu = (operand_a < operand_b)? 1: 0 (unsigned)

xor_com	operand_a operand_b	xor_	xor_ = operand_a $\oplus$ operand_b
or_com	operand_a operand_b	or_	or_ = operand_a $\vee$ operand_b
and_com	operand_a operand_b	and_	and_ = operand_a $\wedge$ operand_b

shift_com	operand_a
	
	operand_b
		sll
		
		srl
		
		sra
			sll = operand_a << operand_b
			
			srl = operand_a >> operand_b
			
			sra = operand_a >>> operand_b

Mux10to1	add sub slt sltu xor_ or_ and_	alu_data	Select the result based on the select signal alu_op.

	sll srl sra alu_op		

2.2. BRANCH COMPARATOR

Description:

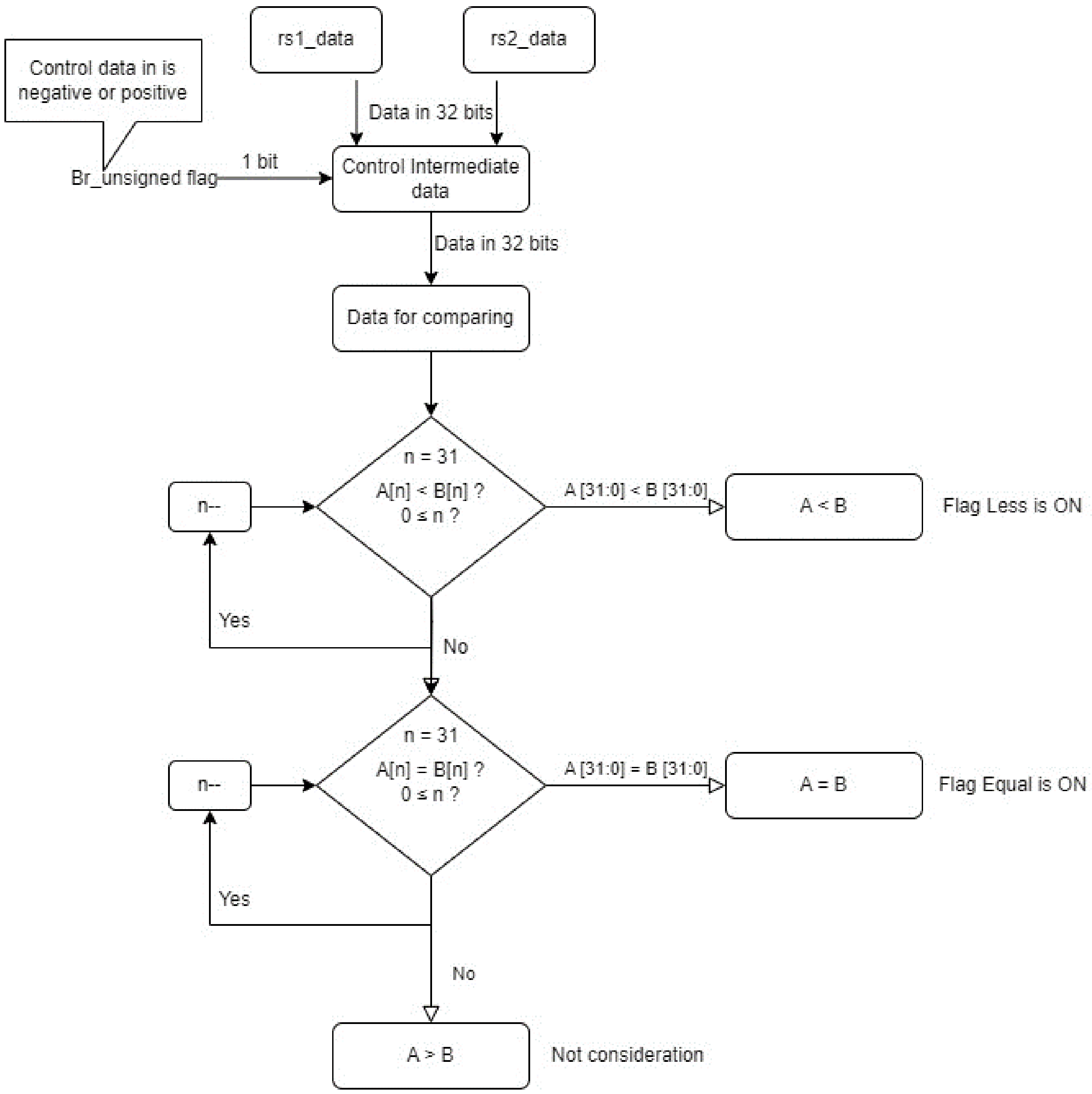
Block is used to perform **comparison** between data input (both *negative* (-) and *positive* (+) values via control signal) in the processor.

Specification:

Module name: branch_comparator

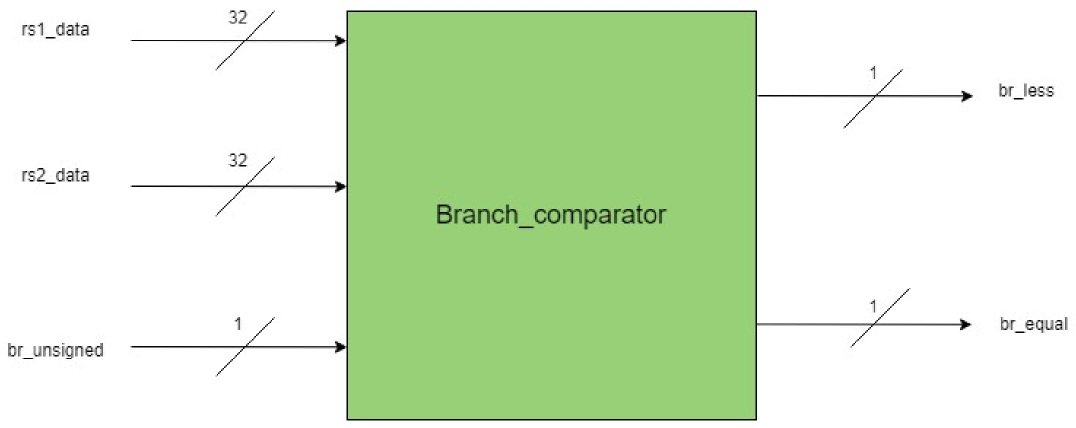
Signal	Type	Size	Description
rs1_data	Input	32	The first operand – rs1_data.
rs2_data	Input	32	The second operand – rs2_data.
br_unsigned	Input	1	The control signal to choose data value's positive or negative.
br_less	Output	1	The flag is active high if **rs1_data < rs2_data**
br_equal	Output	1	The flag is active high if **rs1_data = rs2_data**

<u>Strategy:</u> flow chart



Detailed explanation of flow design of branch comparator block

Block Diagram:



General block diagram of the Branch_comparator

2.3. LOAD STORE UNIT (LSU)

Description:

Component acts like the memory that can store the data performed in the processor (by using store instructions with a given memory address) in 2 different regions of memory (Data memory, output peripheral). We can load the data in 3 different regions of memory (Data memory, output peripheral and input peripheral) into the Register File of the processor for further processing.

This component is also used as a communication between the processor and peripherals of the kit DE2 by memory mapping technique to layout the structure of the memory into different regions for driving the state into the peripherals (LEDR, LEDG, HEX, …).

Specification:

Module name: dmem

Signal	Type	Size	Description

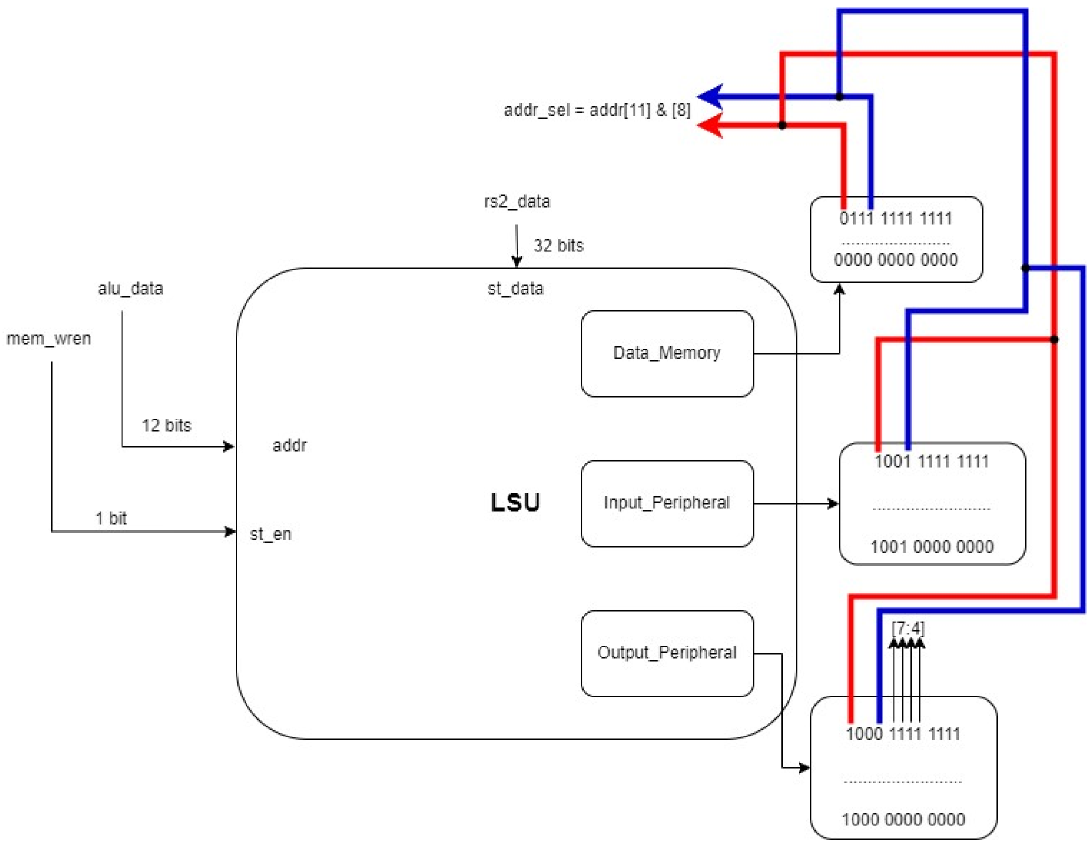
clk_i	Input	1	The input clock signal.
rst_ni	Input	1	The input signal to reset memory.
addr	Input	32	The address signal for store and load data in memory
st_data	Input	32	The input data for storing.
st_en	Input	1	The enable signal to let data to store in.
io_sw	Input	32	The input data for interfacing between program and user.
sw_en	Input	1	Enable signal for storing 32-bit input data
sh_en	Input	1	Enable signal for storing 16-bit input data
sb_en	Input	1	Enable signal for storing 8-bit input data
lw_en	Input	1	Enable signal for load 32-bit data from the memory
lh_en	Input	1	Enable signal for load 16-bit (signed extent) data from the memory
lhu_en	Input	1	Enable signal for load 16-bit data from the memory
lb_en	Input	1	Enable signal for load 8-bit (signed extent) data from the memory
lbu_en	Input	1	Enable signal for load 8-bit data from the memory
ld_data	Output	32	The output data.
io_lcd	Output	32	The data for lcd.
io_ledg	Output	32	The data for ledg.
io_ledr	Output	32	The data for ledr.
io_hex0	Output	32	The data for hex0.
io_hex1	Output	32	The data for hex1.
io_hex2	Output	32	The data for hex2.
io_hex3	Output	32	The data for hex3.
io_hex4	Output	32	The data for hex4.
io_hex5	Output	32	The data for hex5.
io_hex6	Output	32	The data for hex6.
io_hex7	Output	32	The data for hex7.

Strategy:

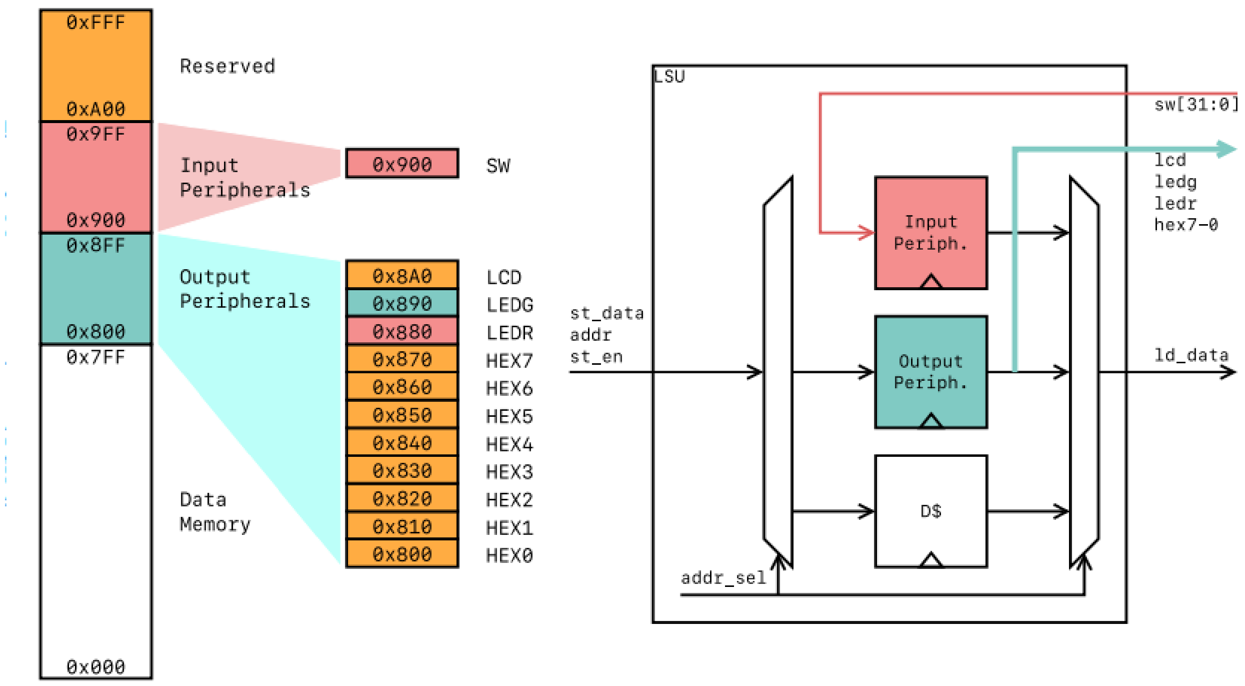
Using memory mapping for layouting the structure of the memory, we declare that…

No.	**Memory-mapping**	**Address**	**Bit Detailed**

1	Input Peripheral	0x9FF ……. 0x900	1001 1111 1111 ………………. 1001 0000 0000
2	Output Peripheral	0x8FF …….. 0x800	1000 1111 1111 ………………. 1000 0000 0000
3	Data Memory	0x7FF …….. 0x000	0111 1111 1111 ………………. 0000 0000 0000



Detailed explanation of flow design LSU block to select which register to store data.

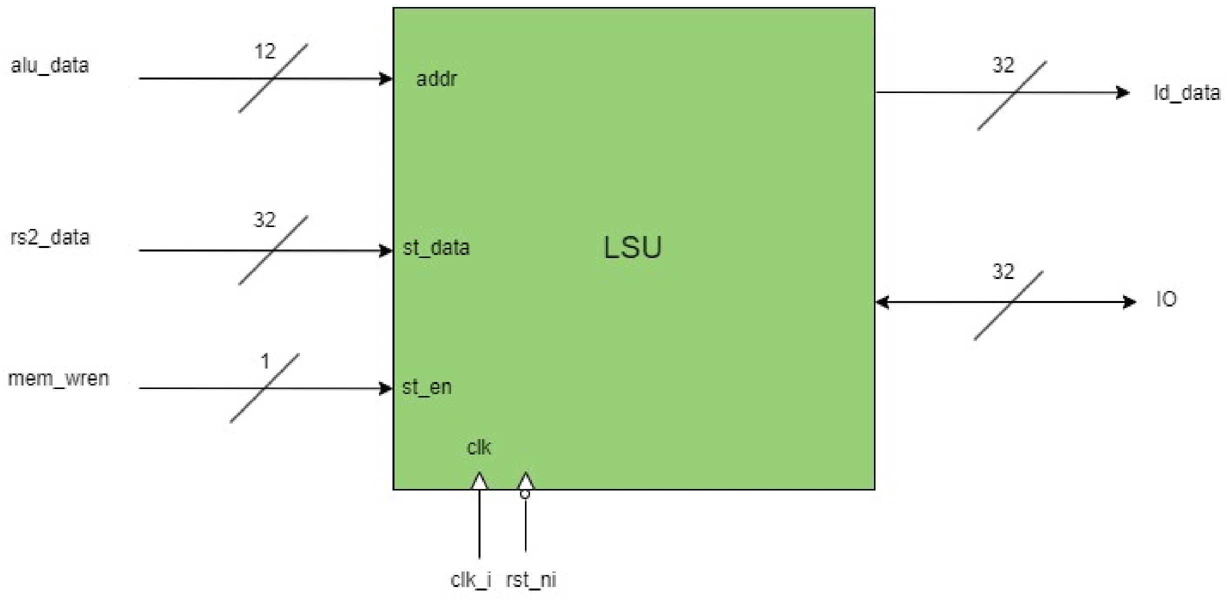


Output Peripheral: We can easily divide Output Peripheral (register) into 11 separately places for 11 functions and can be distinguished by bit # [7 : 4] of addr input port.

No.	Bit # [7:4] of addr	Representation	Detailed bits of addr[11:0]	Function
0	0000	0x800	1000 **0000** 0000	LCD
1	0001	0x810	1000 **0001** 0000	LEDG
2	0010	0x820	1000 **0010** 0000	LEDR
3	0011	0x830	1000 **0011** 0000	HEX0
4	0100	0x840	1000 **0100** 0000	HEX1
5	0101	0x850	1000 **0101** 0000	HEX2
6	0110	0x860	1000 **0110** 0000	HEX3

7	0111	0x870	1000 **0111** 0000	HEX4
8	1000	0x880	1000 **1000** 0000	HEX5
9	1001	0x890	1000 **1001** 0000	HEX6
A	1010	0x8A0	1000 **1010** 0000	HEX7

Block Diagram:



General block diagram of the LSU

2.4. INSTRUCTION MEMORY (IMEM)

Description:

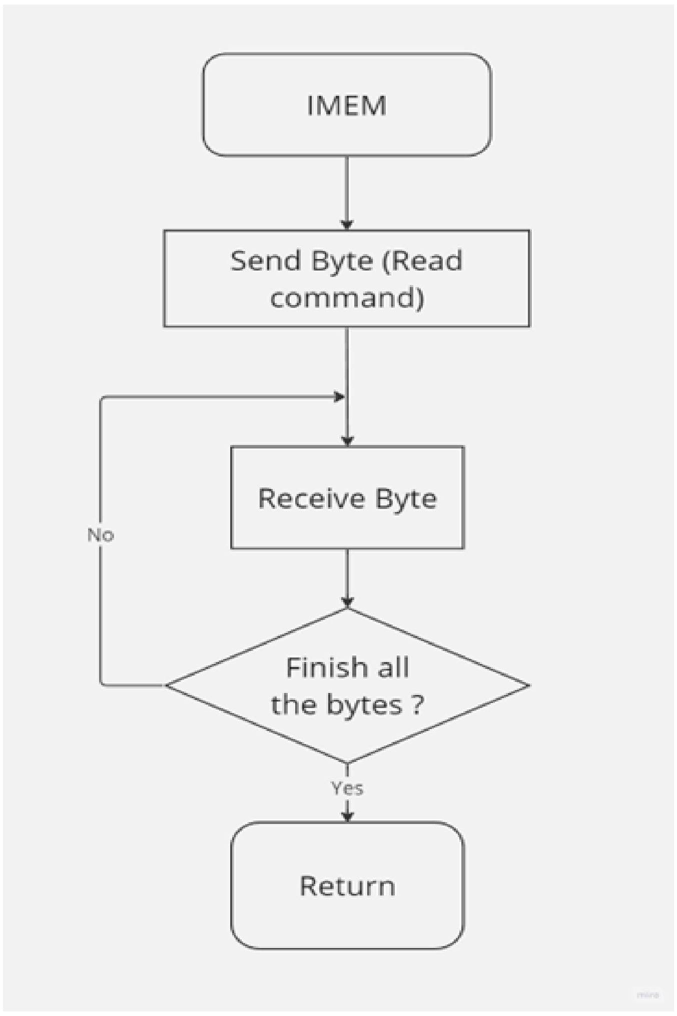
Responsible for storing the instructions that the processor will execute. It holds a sequence of instructions, and during each clock cycle, the processor fetches the next instruction, decodes it, executes it, and writes back the results within a single clock cycle. The instructions are read sequentially from memory and are executed in pipeline-like fashion, allowing for rapid and efficient processing of instructions.

Specification:

Module name: Instruction Memory

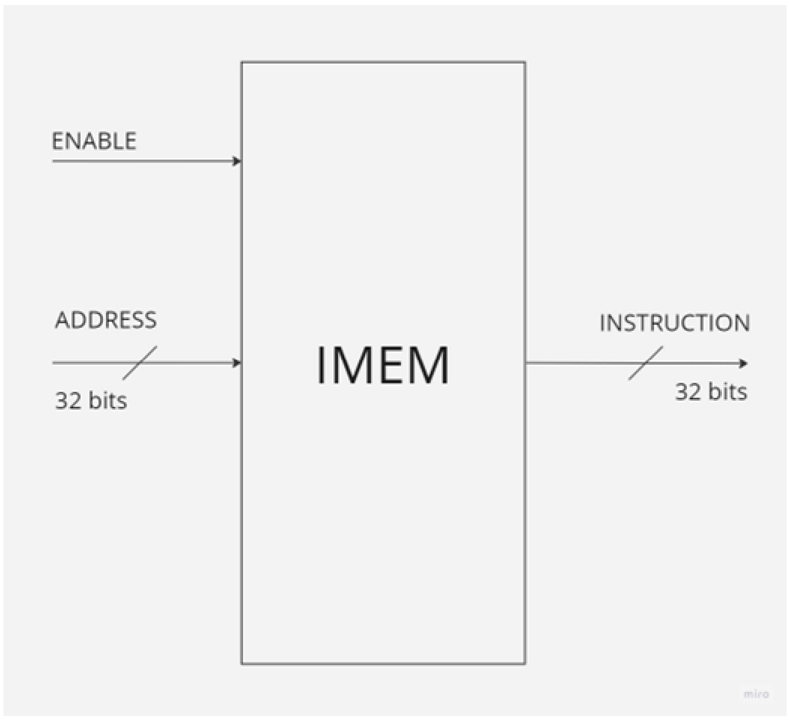
Capacity	32 bits
Data Width	32 bits
Structure	ROM
Read-Write Capabilities	2047 bits

Strategy:



IMEM data flowchart

Block diagram:



IMEM block diagram

2.5. CONTROL UNIT

Description:

This component produces the greatest number of signals in the R32V processor. These signals are used to control the functions of other components (Regfile, ALU, PC_Counter, LSU, …) with relative instruction generated by the instruction memory.

Specification:

Module name: ctr_unit

Signal	Type	Size	Description
inst	Input	32	The 32-bit instruction is fetched into the processor.
br_eq	Input	1	Data from Branch Comparison, 1 if A < B.
br_lt	Input	1	Data from Branch Comparison, 1 if A = B.
pc_sel	Output	1	select PC source: 0 if P C + 4, 1 if computed in ALU.

imm_sel	Output	5	Select the immediate for R, I, S, B, J
br_unsigned	Output	1	1 if the two operands are unsigned.
op_a_sel	Output	2	Select operand A source: 00 if rs1, 01 if PC, 10|11 is 0 (For auipc instruction)
op_b_sel	Output	1	Select operand B source: 0 if rs2, 1 if immediate value.
alu_op	Output	4	Option for alu component.
mem_wren	Output	1	Write data into the LSU, 1: write, 0: read.
rd_wren	Output	1	Enable signal for writing data into the regfile. (0: Read, 1: Write)
wb_sel	Output	2	Mux select signal to write into the regfile. 00: load_data, 01: alu_data, 10: pc+4
sw_en	Output	1	Signal for LSU store operation
sh_en	Output	1	Signal for LSU store operation
sb_en	Output	1	Signal for LSU store operation
lw_en	Output	1	Signal for LSU load operation
lh_en	Output	1	Signal for LSU load operation
lhu_en	Output	1	Signal for LSU load operation
lb_en	Output	1	Signal for LSU load operation
lbu_en	Output	1	Signal for LSU load operation

Strategy:

To control the data path of the processor according to the fetched instruction, we have to design the control unit that can distinguish between each type of the instruction and each function in it.

	Opcode	**funct3**	**funct7**	**Type**
add	0110011	000	0000000	R – Type
sub	0110011	000	0100000	
sll	0110011	001	0000000	
slt	0110011	010	0000000	
sltu	0110011	011	0000000	
xor	0110011	100	0000000	

srl	0110011	101	0000000	
sra	0110011	101	0100000	
or	0110011	110	0000000	
and	0110011	111	0000000	
addi	0010011	000	NA	
slti	0010011	010	NA	
sltiu	0010011	011	NA	
xori	0010011	100	NA	
ori	0010011	110	NA	
andi	0010011	111	NA	
slli	0010011	001	0000000	
srli	0010011	101	0000000	
srai	0010011	101	0100000	I – Type
lb	0000011	000	NA	
lh	0000011	001	NA	
lw	0000011	010	NA	
lbu	0000011	100	NA	
lhu	0000011	101	NA	L – Type
sb	0100011	000	NA	
sh	0100011	001	NA	
sw	0100011	010	NA	S – Type
beq	1100011	000	NA	
bne	1100011	001	NA	
blt	1100011	100	NA	
bge	1100011	101	NA	
bltu	1100011	110	NA	
bgeu	1100011	111	NA	B – Type
jal	1101111	NA	NA	
jalr	1100111	NA	NA	J – Type
auipc	0010111	NA	NA	U – Type

lui	0110111	NA	NA	

Instruction table of the processor

Each type of instruction has **their own Opcode located from bit 2nd to bit 6th of the instruction**, and each type of instruction, the different instruction will have different **funct3 located from the instruction from bit 12th to 14th.** For the same function but different value indicates (signed value in add – sub, srl – sra, srli - srai), **the bit 30th in the instruction will be used to distinguish (funct7).**

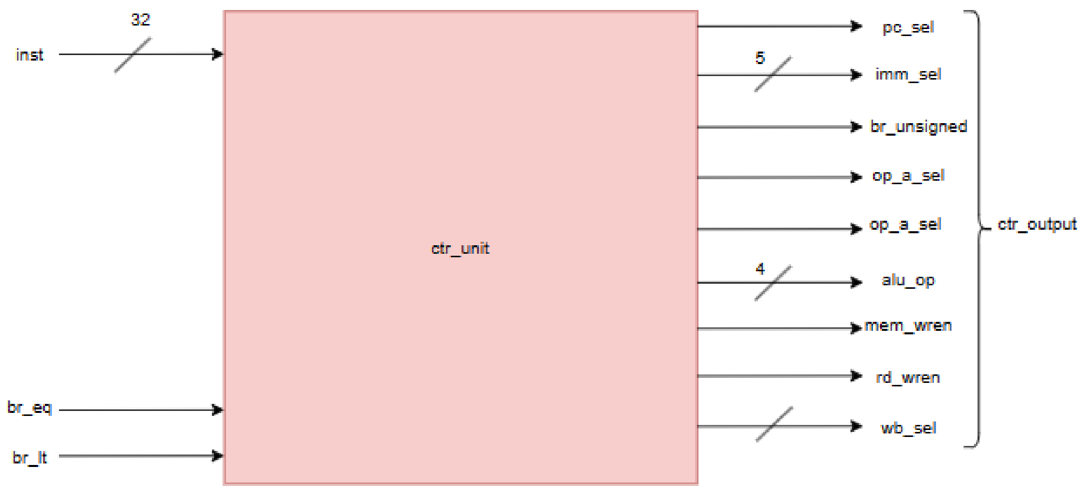
For each instruction, the control unit will decode it and send suitable signals to control the components in the data path for executing the instruction. Here is the table that describes the output signals of the control unit for each instruction.

	br_eq	br_lt	pc_sel	ImmSel	br_unsigned	op_a_sel	op_b_sel	alu_op	mem_wren	rd_wren	wb_sel
add	*	*	+4	*	*	reg	reg	add	read	1	alu
sub	*	*	+4	*	*	reg	reg	sub	read	1	alu
sll	*	*	+4	*	*	reg	reg	sll	read	1	alu
slt	*	1	+4	*	*	reg	reg	slt	read	1	alu
sltu	*	1	+4	*	*	reg	reg	sltu	read	1	alu
xor	*	*	+4	*	*	reg	reg	xor	read	1	alu
srl	*	*	+4	*	*	reg	reg	srl	read	1	alu
sra	*	*	+4	*	*	reg	reg	sra	read	1	alu
or	*	*	+4	*	*	reg	reg	or	read	1	alu
and	*	*	+4	*	*	reg	reg	and	read	1	alu
addi	*	*	+4	I	*	reg	imm	add	read	1	alu
slti	*	1	+4	I	*	reg	imm	slt	read	1	alu
sltiu	*	1	+4	I	*	reg	imm	sltu	read	1	alu
xori	*	*	+4	I	*	reg	imm	xor	read	1	alu
ori	*	*	+4	I	*	reg	imm	or	read	1	alu
andi	*	*	+4	I	*	reg	imm	and	read	1	alu
slli	*	*	+4	I	*	reg	imm	sll	read	1	alu
srli	*	*	+4	I	*	reg	imm	srl	read	1	alu
srai	*	*	+4	I	*	reg	imm	sra	read	1	alu

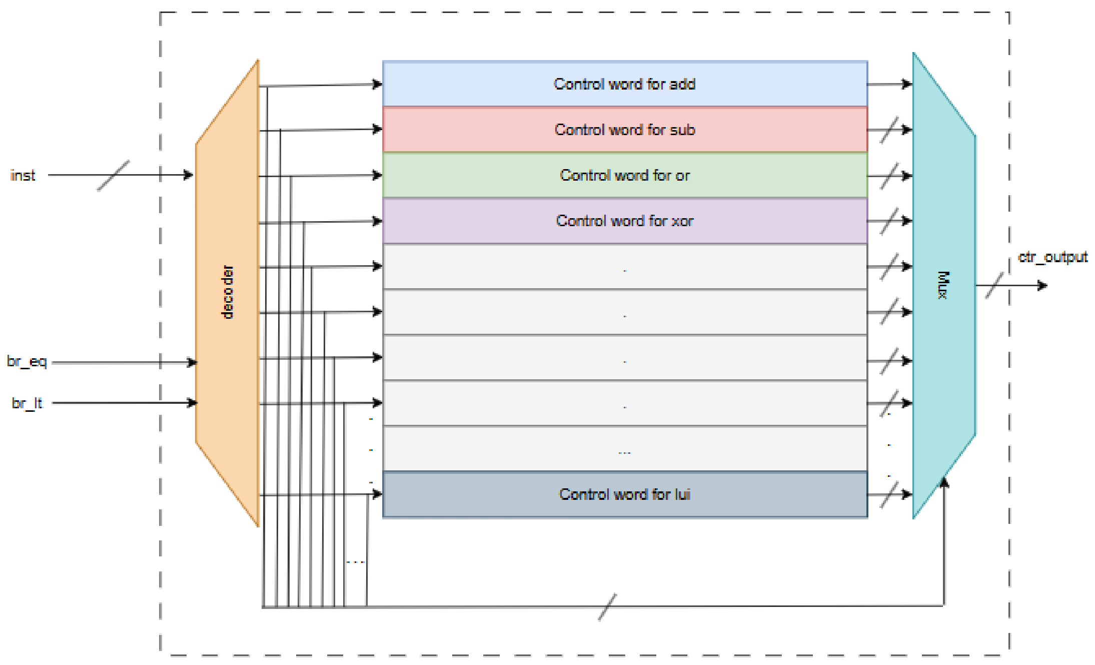
lb	*	*	+4	I	*	reg	imm	add	read	1	mem
lh	*	*	+4	I	*	reg	imm	add	read	1	mem
lw	*	*	+4	I	*	reg	imm	add	read	1	mem
lbu	*	*	+4	I	*	reg	imm	add	read	1	mem
lhu	*	*	+4	I	*	reg	imm	add	read	1	mem
sb	*	*	+4	S	*	reg	imm	add	write	0	*
sh	*	*	+4	S	*	reg	imm	add	write	0	*
sw	*	*	+4	S	*	reg	imm	add	write	0	*
beq	1	*	ALU	B	*	PC	imm	add	read	0	*
beq	0	*	+4	B	*	PC	imm	add	read	0	*
bne	1	*	+4	B	*	PC	imm	add	read	0	*
bne	0	*	ALU	B	*	PC	imm	add	read	0	*
blt	*	1	ALU	B	0	PC	imm	add	read	0	*
bge	1 | 0	1 | 0	ALU	B	0	PC	imm	add	read	0	*
bltu	*	1	ALU	B	1	PC	imm	add	read	0	*
bgeu	1 | 0	1 | 0	ALU	B	1	PC	imm	add	read	0	*
jal	*	*	ALU	J	*	PC	imm	add	read	1	PC + 4
jalr	*	*	ALU	I	*	reg	imm	add	read	1	PC + 4
auipc	*	*	+4	U	*	PC	imm	add	read	1	alu
lui	*	*	+4	U	*	reg (r0)	imm	add	read	1	alu

Output control signal of the control unit

<u>Block diagram:</u>



General block diagram of the processor control unit



Detail block diagram of the processor control unit

2.6. REGISTER FILE (REGFILE)

Description:

Register file is a part of the processor's data path, and it stores a set of registers that can be accessed and manipulated during the execution of instructions. In a single clock cycle architecture, the register file often operates within one clock cycle, meaning that register read, write, and other operations related to the registers occur in a single clock cycle.

Specifications:

Module: regfile

Size	32 bits
Data Width	32 bits
Data Memory	2048 bits
Read Ports	2
Write Ports	1
Latency	Single clock cycle access time
Initialization	All registers set to 0

Strategy:

Register Structure:

Registers: Create 32 registers (R0 to R31) each of 32 bits.

Data Storage: Use an array to store register data.

Concurrent Read Operations:

Read Ports: Implement at least two read ports for simultaneous access to multiple registers within a clock cycle.

Parallel Read: Allow concurrent reading from registers R1 and R2 in a single clock cycle, enhancing parallelism in execution.

Write Operations:

Write Port: Design a single write port to allow writing to a single register within a clock cycle.

Write Enable Signal: Incorporate a write-enable signal to control write operations.

Addressing and Control Signals:

Address Inputs: Utilize 5-bit address inputs (read_addr_1, read_addr_2, and write_addr) to select registers for read and write operations.

Write Data: Provide a 32-bit data input (write_data) to update the selected register.

Control Signals: Implement clock signal for synchronization and control and a reset signal for initialization.

Access Time Optimization:

Single Clock Cycle Access: Ensure that read and write operations can be completed within a single clock cycle to maximize efficiency.

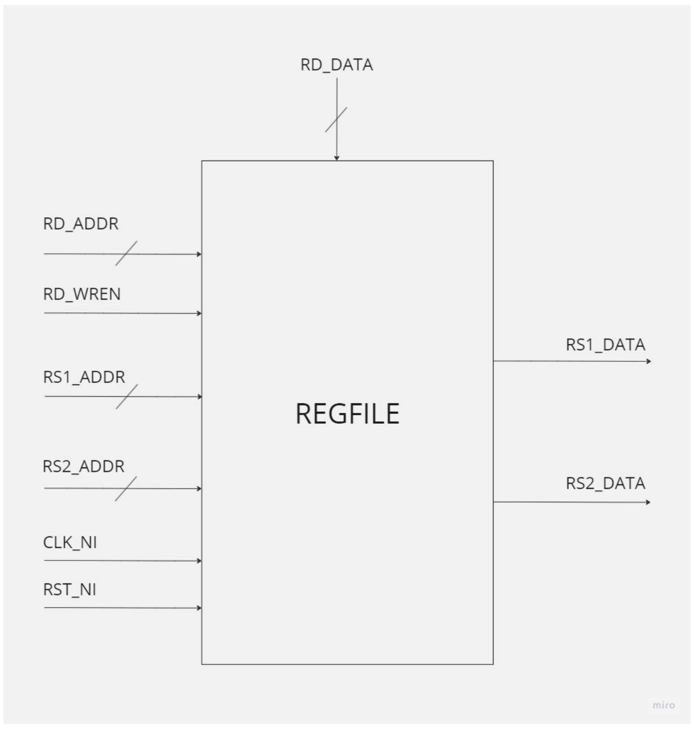
Zero Register Handling:

Zero Register (R0): Define the zero register (R0) to always return zero and restrict write operations to this register to maintain consistency.

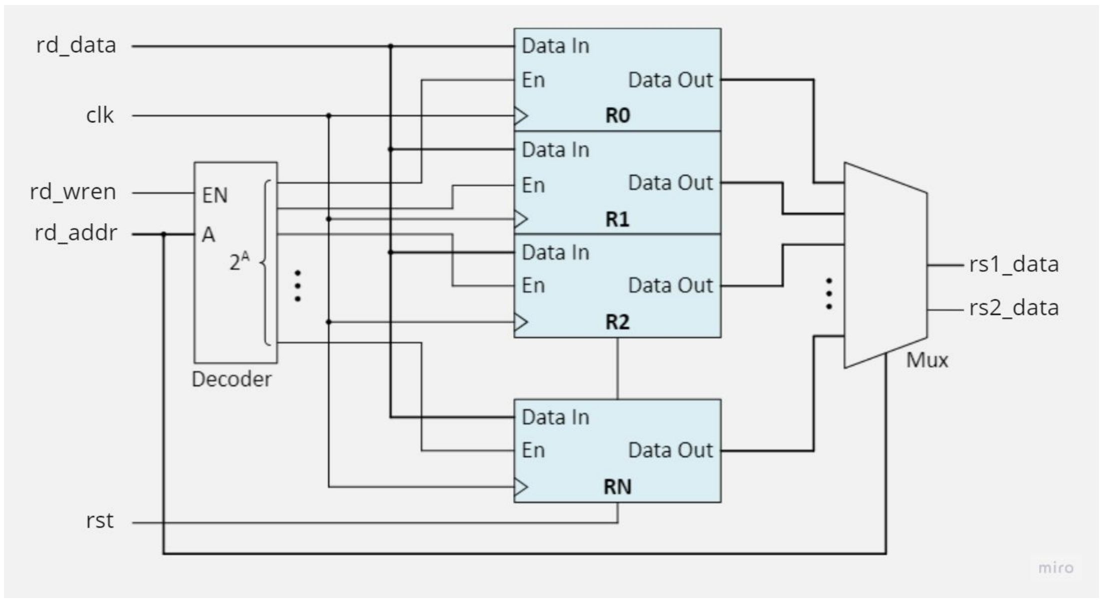
Initialization:

Reset Behavior: On reset, initialize all registers to zero for a known state.

Block diagram:



Register file block diagram



Register File detail block diagram

2.7. IMMEDIATE GENERATOR (IMM_GEN)

Description:

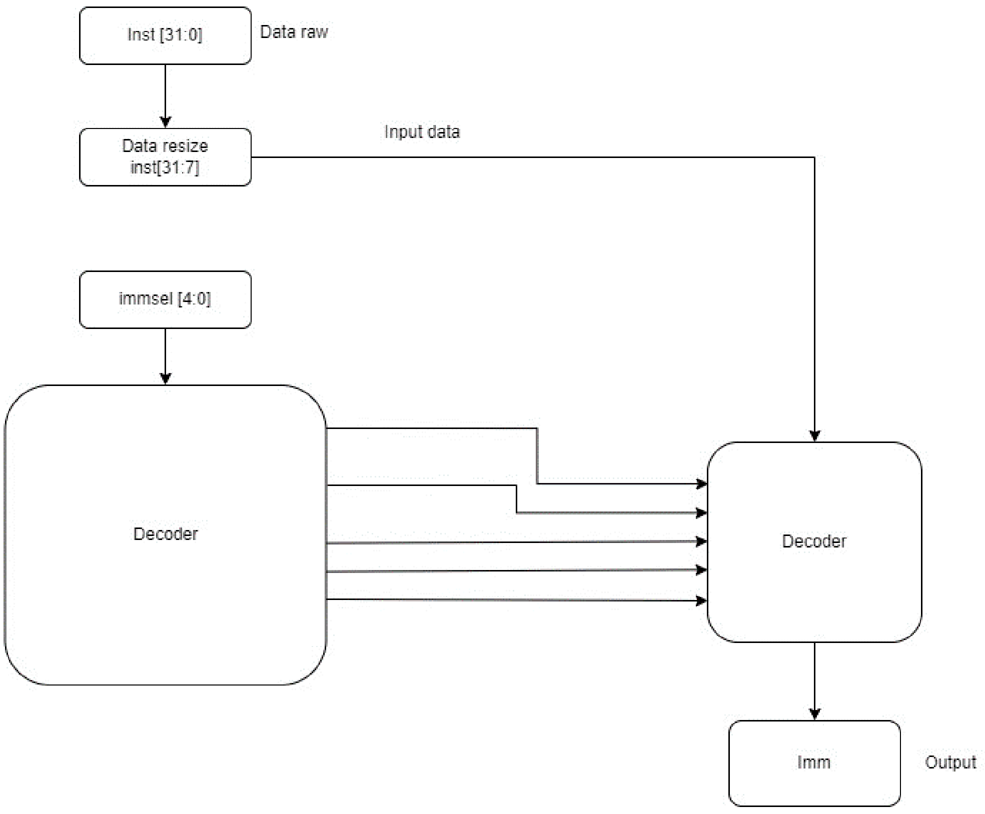
Block is used to generate immediate (based on instruction input) corresponding to input signal in the processor.

Specification:

Module name: imm_gen

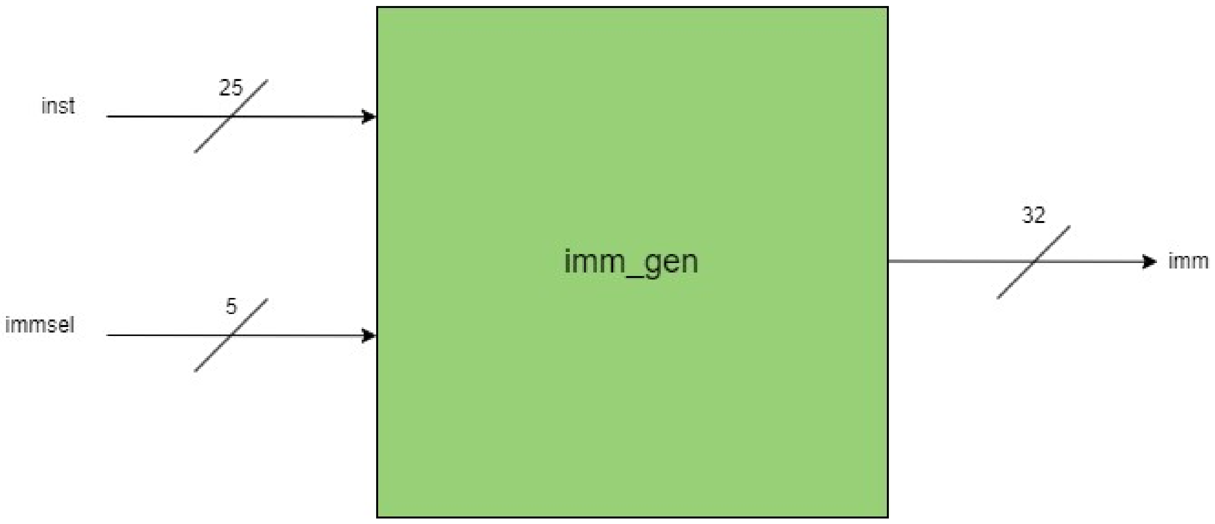
Signal	Type	Size	Description
inst	Input	25	The input instruction used for regenerating.
immsel	Input	5	The control signal to select output data type.
imm	Output	32	The output data after regenerating.

Strategy:



Detailed explanation of flow design of immediate generator block

Block Diagram:



General block diagram of the imm_gen

2.8. SINGLE-CYCLE PROCESSOR

Description:

With the combination of all components designed in the previous path, we can build a single-cycle processor that can carry out one instruction in a single clock cycle (One clock period).

Specification:

Module: singlecycle

Signal	Type	Size	Description
io_sw_i	Input	32	The 32-bits input switch.
clk_i	Input	1	Positive Clock .
rst_ni	Input	1	Low negative reset
pc_debug_o	Output	32	PC address for debugging.
io_lcd_o	Output	32	32-bit data to drive LCD
io_ledg_o	Output	32	32-bit data to drive green LEDs
io_ledr_o	Output	32	32-bit data to drive red LEDs.
io_hex0_o	Output	32	32-bit data to drive 7-segment LEDs HEX0

io_hex1_o	Output	32	32-bit data to drive 7-segment LEDs HEX1
io_hex2_o	Output	32	32-bit data to drive 7-segment LEDs HEX2
io_hex3_o	Output	32	32-bit data to drive 7-segment LEDs HEX3
io_hex4_o	Output	32	32-bit data to drive 7-segment LEDs HEX4
io_hex5_o	Output	32	32-bit data to drive 7-segment LEDs HEX5
io_hex6_o	Output	32	32-bit data to drive 7-segment LEDs HEX6
io_hex7_o	Output	32	32-bit data to drive 7-segment LEDs HEX7

3. Verification Strategy

3.1. Testbench - Test program

In this path, we write the program to test all the situations that can happen while using the processor. We consider the test program into 3 cases:

Case 1: For No Load and Store Operation in Memory

Description: Check the functions of the processor that do not store and load the data from LSU, just check the calculation of the processor. The program will be interrupted if the result does not satisfy the setup values.

Program:

```
#Case 1: For No load and store Operation in memory
        addi x1 x0 3                # x1 = 3
        addi x2 x0 5                # x2 = 5
        addi x3 x0 -9               # x3 = -9
   addi x9 x0 12                    # x9 = 12
        # Use the register executed in the previous instructions
        sub x4 x2 x3                # x4 = 14
   sub x8 x4 x2                     # x8 = 9
   add x2 x8 x3                     # x2 = 0
   bne x2 x0 END
```

```
  or x2 x1 x1                # Or itseft
  or x3 x1 x2                # x3 = x2 = x1
  bne x2 x3 END              # Could not jump to end as x2 = x3 = -9
  bne x1 x3 END              # Could not jump to end as x1 = x3 = -9
  ori x3 x2 0
  blt x1 x3 END              # Could not jump to end
  bltu x3 x1 END             #

  xor x3 x2 x1               # x3 = 0 As x2 = x1
  bne x3 x0 END              # Could not jump to end
  xori x3 x2 0xF             # 1 xor (0 | 1) = 1 => x3 > x2
  blt x3 x1 END              # Could not jump to end

  add x2 x0 x1               # x2 = x1
  and x3 x2 x1               # x2 = x1 = x3
  and x4 x3 x0               # x4 = 0
  bne x4 x0 END              # Could not jump to end as x4 = 0
  bne x3 x2 END              # Could not jump to end as x3 = x2
  beq x3 x0 END              # Could not jump to end
  andi x3 x2 0               # x3 = 0
 bne x3 x0 END               # Could not jump to end


  # Check shift
  auipc x7 0                 # Load PC value into x7 = prev_PC + 4 = 0x18
  lui x6 0x6C                # shift 0x18 12 bit left
  slli x11 x7 12             # shift x7 = 0x14 12 bit left
  sll x12 x7 x9              # shift x7 = 0x14 with x9 = 12 bit left and store
into x12
```

```
   beq x6 x7 END                  # Could not jump to end
   bne x6 x11 END                 # Could not jump to end
   bne x6 x12 END                 # Could not jump to end

# Check the sign and unsigned condition
   addi x1 x0 3
   addi x2 x0 5
   addi x3 x0 -9

   slt x5 x2 x3                   # x2 > x3 => x5 = 0
   sltu x6 x2 x3                  # x2 < x3 => x6 = 1
   beq x5 x6 END                  # Could not jump to end
   bltu x3 x2 END                 # Could not jump to end
   blt x2 x3 END                  # Could not jump to end
   bge x3 x2 END                  # Could not jump to end
   bgeu x2 x3 END                 # Could not jump to end
```

Case 2: For Load and Store Operation in Memory

Description: Check the functions of the load and store instructions and the data loaded into the Regfile that can be used correctly in the next instruction.

Program:

```
# Case 2: For load and store Operation in memory
   addi x1 x0 3
   addi x2 x0 5
   addi x3 x0 -9
# Signed value
        # Store Word
   auipc x7 0                     # Load PC value into x7 = prev_PC + 4
   sw x3 0(x7)                           # Store at address [x7 + 0]
```

```
    auipc x7 0                      # Load PC value into x7 = prev_PC + 4
    lw x4 -8(x7)                    # Load at address [x7 - 8]
    lh x5 -8(x7)                    # Load at address [x7 - 8]
    lb x6 -8(x7)                    # Load at address [x7 - 8]
    bne x4 x3 END                   # Could not jump to end
    bne x5 x3 END                   # Could not jump to end
    bne x6 x3 END                   # Could not jump to end

        # Store haftword
    auipc x7 0                      # Load PC value into x7 = prev_PC + 4
    sh x3 0(x7)                             # Store at address [x7 + 0]
    auipc x7 0                      # Load PC value into x7 = prev_PC + 4
    lw x4 -8(x7)                    # Load at address [x7 - 8]
    lh x5 -8(x7)                    # Load at address [x7 - 8]
    lb x6 -8(x7)                    # Load at address [x7 - 8]
    bge x3 x4 END                   # Could not jump to end although x4 can be
greater than x3 if the 2 values in the addresses after sh store is different from 0
    bne x5 x3 END                   # Could not jump to end
    bne x6 x3 END                   # Could not jump to end

        # Store byte
    auipc x7 0                      # Load PC value into x7 = prev_PC + 4
    sb x3 0(x7)                             # Store at address [x7 + 0]
    auipc x7 0                      # Load PC value into x7 = prev_PC + 4
    lw x4 -8(x7)                    # Load at address [x7 - 8]
    lh x5 -8(x7)                    # Load at address [x7 - 8]
    lb x6 -8(x7)                    # Load at address [x7 - 8]
    bge x3 x4 END                   # Could not jump to end although x4 can be
greater than x3 if the 2 values in the addresses after sh store is different from 0
```

```
   bge  x3 x5 END                  # Could not jump to end although x4 can be
greater than x3 if the 3 values in the addresses after sh store is different from 0
   bne  x6 x3 END                  # Could not jump to end

# Unsigned value
        # Store Word
   auipc x7 0                      # Load PC value into x7 = prev_PC + 4
   sw x3 0(x7)                             # Store at address [x7 + 0]
   auipc x7 0                      # Load PC value into x7 = prev_PC + 4
   lw x4 -8(x7)                    # Load at address [x7 - 8]
   lhu x5 -8(x7)                   # Load at address [x7 - 8]
   lbu x6 -8(x7)                   # Load at address [x7 - 8]
   bne  x4 x3 END                  # Could not jump to end
   blt   x5 x3 END                 # Could not jump to end
   blt   x6 x3 END                 # Could not jump to end

   # Store haftword
   auipc x7 0                      # Load PC value into x7 = prev_PC + 4
   sh x3 0(x7)                             # Store at address [x7 + 0]
   auipc x7 0                      # Load PC value into x7 = prev_PC + 4
   lw x4 -8(x7)                    # Load at address [x7 - 8]
   lhu x5 -8(x7)                   # Load at address [x7 - 8]
   lbu x6 -8(x7)                   # Load at address [x7 - 8]
   blt   x4 x3 END                 # Could not jump to end
   blt   x5 x3 END                 # Could not jump to end
   blt   x6 x3 END                 # Could not jump to end

   # Store byte
        auipc x7 0                         # Load PC value into x7 = prev_PC + 4
        sb x3 0(x7)                        # Store at address [x7 + 0]
```

```
auipc x7 0               # Load PC value into x7 = prev_PC + 4
lw x4 -8(x7)             # Load at address [x7 - 8]
lhu x5 -8(x7)            # Load at address [x7 - 8]
lbu x6 -8(x7)            # Load at address [x7 - 8]
blt   x4 x3 END          # Could not jump to end
blt   x5 x3 END          # Could not jump to end
blt   x6 x3 END          # Could not jump to end
```

Case 3: For Loop Structure

Description: Check the functions for jal and jalr instructions which are used to create a loop structure, which is used in the assembly program.

Program:

```
#Case 3: For Loop Structure
   addi x1 x0 5
   addi x2 x0 8
   addi x3 x0 -1
   addi x4 x0 1
   addi x6 x0 0
LOOP1:      bge x3 x2 LOOP2
   addi x3 x3 1
   addi x6 x6 1              # Count loop
   jal x5 LOOP1

LOOP2:
   addi x8 x0 9
   bne x8 x6 END             # 16 Loop
```

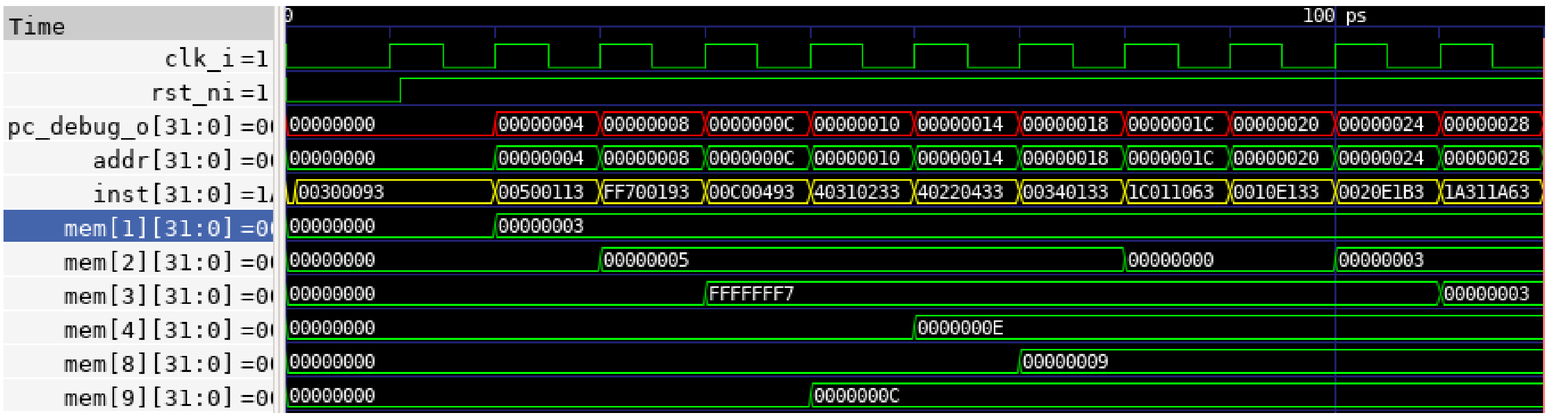
```
    addi x3 x0 8
    addi x1 x0 1
    auipc x7 0
    blt x3 x1 ENDLOOP
    sub x3 x3 x1
    jalr x5 x7 4                 # Jump to the blt instruction

ENDLOOP: auipc x7 0         # Show the end pc address
END:
```

Strategy:

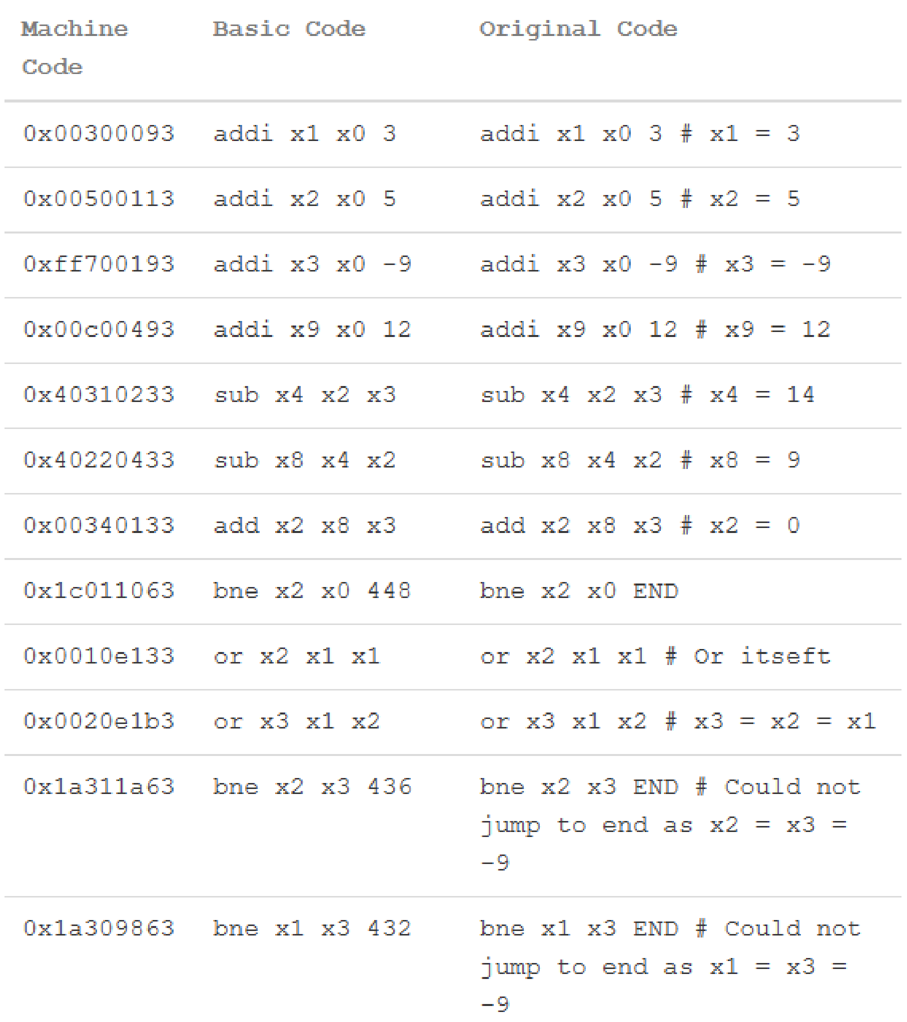
The test program is the combination of 3 cases programs above and in each check function path have the check instruction (using branch instruction), when the checking result of the program is wrong, the program will jump to the END program

3.2. Waveform Observation

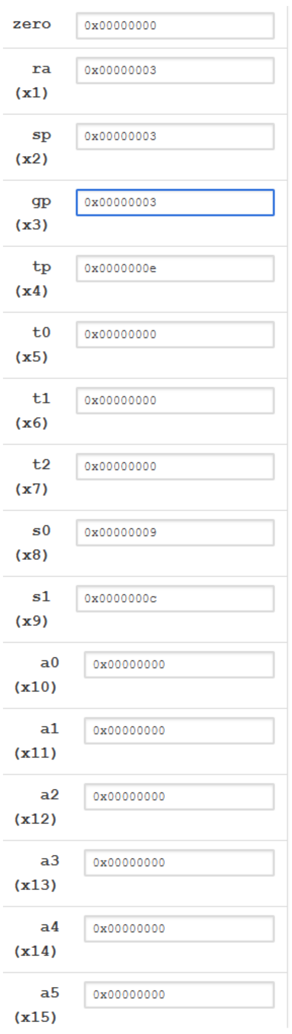
In Case 1:



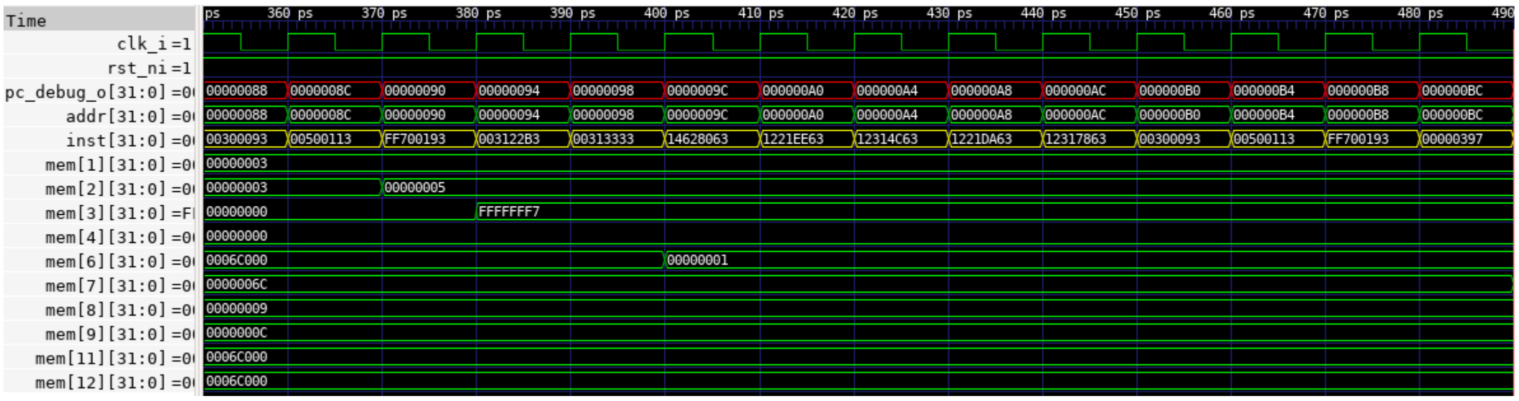
```
Time                 0                                                                                                          100 ps
clk_i=1
rst_ni=1
pc_debug_o[31:0]=0   00000000  00000004  00000008  0000000C  00000010  00000014  00000018  0000001C  00000020  00000024  00000028
addr[31:0]=0         00000000  00000004  00000008  0000000C  00000010  00000014  00000018  0000001C  00000020  00000024  00000028
inst[31:0]=1         00300093  00500113  FF700193  00C00493  40310233  40220433  00340133  1C011063  0010E133  0020E1B3  1A311A63
mem[1][31:0]=0       00000000  00000003
mem[2][31:0]=0       00000000            00000005                                                  00000000            00000003
mem[3][31:0]=0       00000000                      FFFFFFF7                                                                      00000003
mem[4][31:0]=0       00000000                                          0000000E
mem[8][31:0]=0       00000000                                                    00000009
mem[9][31:0]=0       00000000                                0000000C
```



Machine Code	Basic Code	Original Code
0x00300093	addi x1 x0 3	addi x1 x0 3 # x1 = 3
0x00500113	addi x2 x0 5	addi x2 x0 5 # x2 = 5
0xff700193	addi x3 x0 -9	addi x3 x0 -9 # x3 = -9
0x00c00493	addi x9 x0 12	addi x9 x0 12 # x9 = 12
0x40310233	sub x4 x2 x3	sub x4 x2 x3 # x4 = 14
0x40220433	sub x8 x4 x2	sub x8 x4 x2 # x8 = 9
0x00340133	add x2 x8 x3	add x2 x8 x3 # x2 = 0
0x1c011063	bne x2 x0 448	bne x2 x0 END
0x0010e133	or x2 x1 x1	or x2 x1 x1 # Or itseft
0x0020e1b3	or x3 x1 x2	or x3 x1 x2 # x3 = x2 = x1
0x1a311a63	bne x2 x3 436	bne x2 x3 END # Could not jump to end as x2 = x3 = -9
0x1a309863	bne x1 x3 432	bne x1 x3 END # Could not jump to end as x1 = x3 = -9

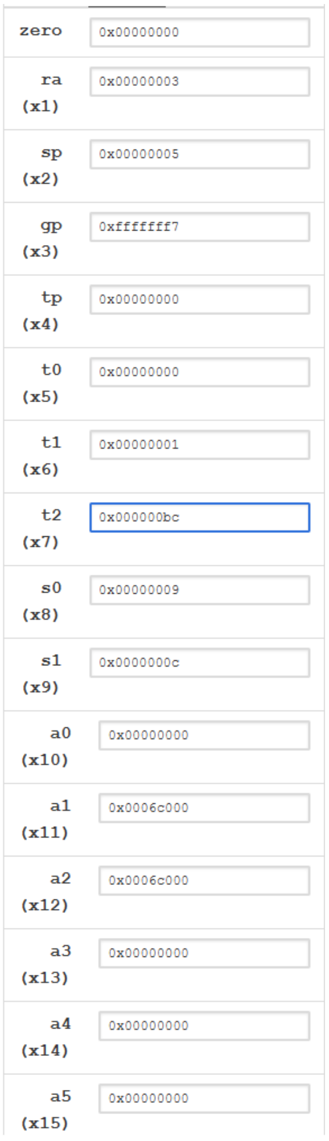


zero	0x00000000
ra (x1)	0x00000003
sp (x2)	0x00000003
gp (x3)	0x00000003
tp (x4)	0x0000000e
t0 (x5)	0x00000000
t1 (x6)	0x00000000
t2 (x7)	0x00000000
s0 (x8)	0x00000009
s1 (x9)	0x0000000c
a0 (x10)	0x00000000
a1 (x11)	0x00000000
a2 (x12)	0x00000000
a3 (x13)	0x00000000
a4 (x14)	0x00000000
a5 (x15)	0x00000000



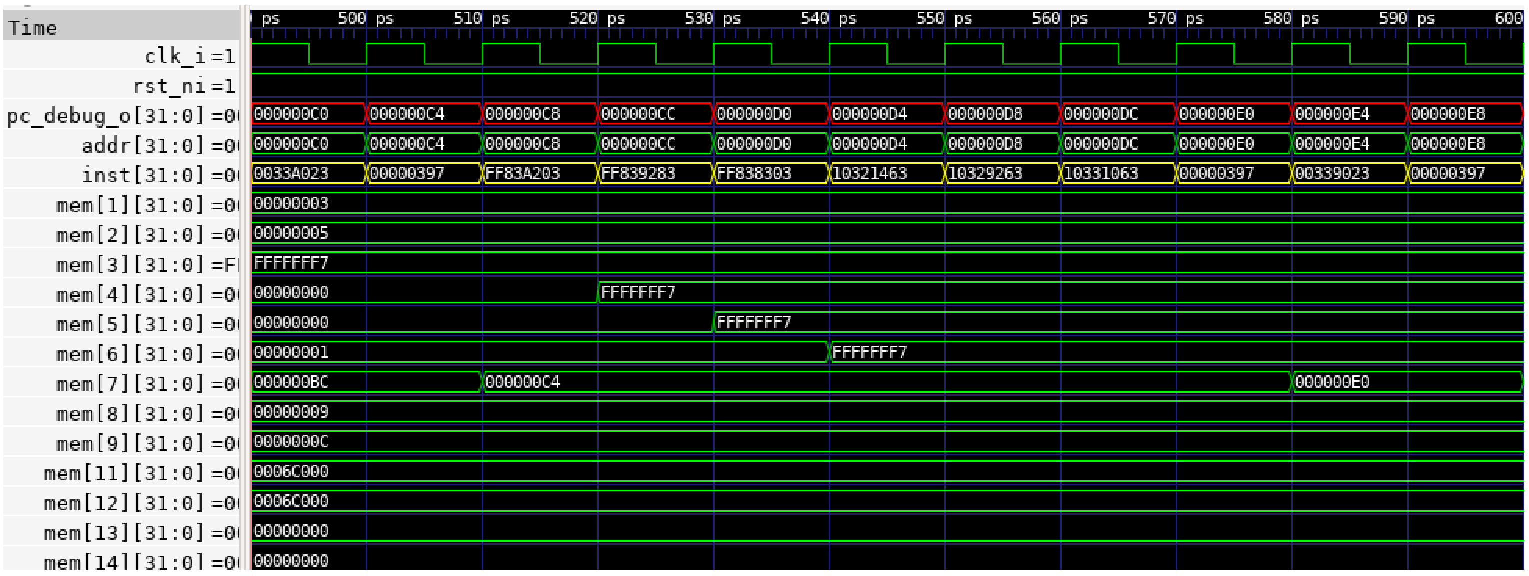
```
Time                 ps        360 ps    370 ps    380 ps    390 ps    400 ps    410 ps    420 ps    430 ps    440 ps    450 ps    460 ps    470 ps    480 ps    490
clk_i=1
rst_ni=1
pc_debug_o[31:0]=0   00000088  0000008C  00000090  00000094  00000098  0000009C  000000A0  000000A4  000000A8  000000AC  000000B0  000000B4  000000B8  000000BC
addr[31:0]=0         00000088  0000008C  00000090  00000094  00000098  0000009C  000000A0  000000A4  000000A8  000000AC  000000B0  000000B4  000000B8  000000BC
inst[31:0]=0         00300093  00500113  FF700193  003122B3  00313333  14628063  1221EE63  12314C63  1221DA63  12317863  00300093  00500113  FF700193  00000397
mem[1][31:0]=0       00000003
mem[2][31:0]=0       00000003            00000005
mem[3][31:0]=F       00000000                      FFFFFFF7
mem[4][31:0]=0       00000000
mem[6][31:0]=0       0006C000                                          00000001
mem[7][31:0]=0       0000006C
mem[8][31:0]=0       00000009
mem[9][31:0]=0       0000000C
mem[11][31:0]=0      0006C000
mem[12][31:0]=0      0006C000
```

0x00300093	addi x1 x0 3	addi x1 x0 3
0x00500113	addi x2 x0 5	addi x2 x0 5
0xff700193	addi x3 x0 -9	addi x3 x0 -9
0x003122b3	slt x5 x2 x3	slt x5 x2 x3 # x2 > x3 => x5 = 0
0x00313333	sltu x6 x2 x3	sltu x6 x2 x3 # x2 < x3 => x6 = 1
0x14628063	beq x5 x6 320	beq x5 x6 END # Could not jump to end
0x1221ee63	bltu x3 x2 316	bltu x3 x2 END # Could not jump to end
0x12314c63	blt x2 x3 312	blt x2 x3 END # Could not jump to end
0x1221da63	bge x3 x2 308	bge x3 x2 END # Could not jump to end
0x12317863	bgeu x2 x3 304	bgeu x2 x3 END # Could not jump to end
0x00300093	addi x1 x0 3	addi x1 x0 3
0x00500113	addi x2 x0 5	addi x2 x0 5
0xff700193	addi x3 x0 -9	addi x3 x0 -9
0x00000397	auipc x7 0	auipc x7 0 # Load PC value into x7 = prev_PC + 4

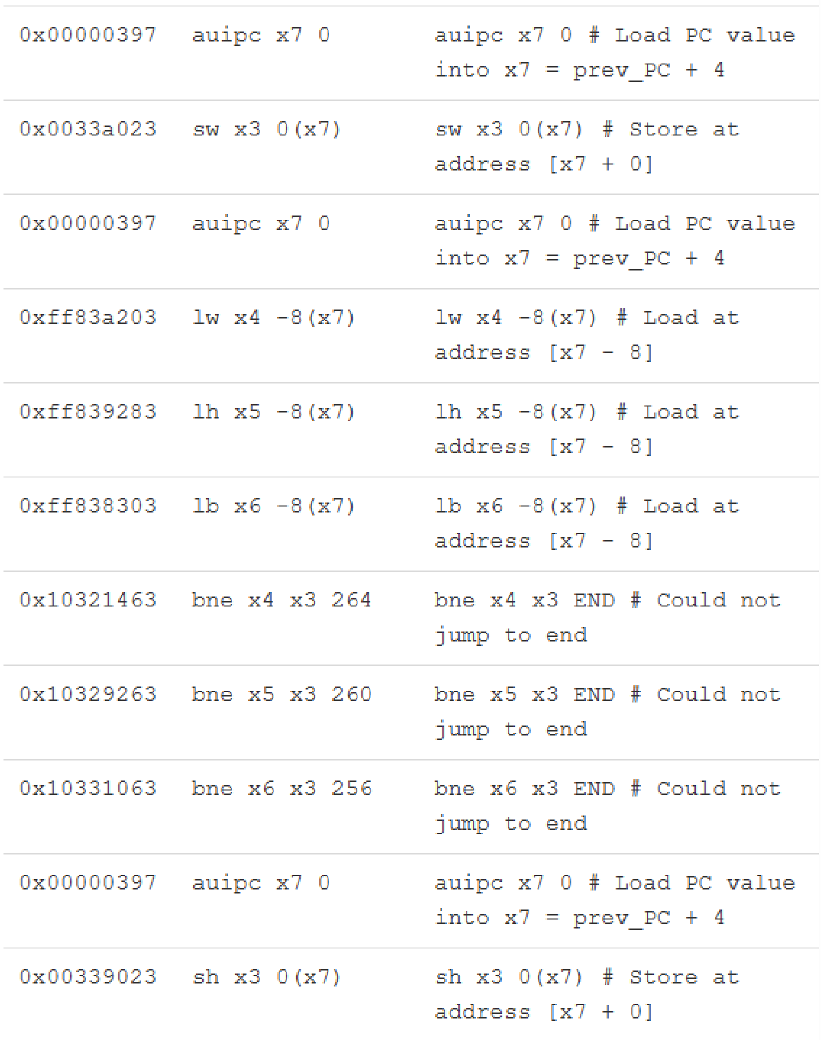


zero	0x00000000
ra (x1)	0x00000003
sp (x2)	0x00000005
gp (x3)	0xfffffff7
tp (x4)	0x00000000
t0 (x5)	0x00000000
t1 (x6)	0x00000001
t2 (x7)	0x000000bc
s0 (x8)	0x00000009
s1 (x9)	0x0000000c
a0 (x10)	0x00000000
a1 (x11)	0x0006c000
a2 (x12)	0x0006c000
a3 (x13)	0x00000000
a4 (x14)	0x00000000
a5 (x15)	0x00000000

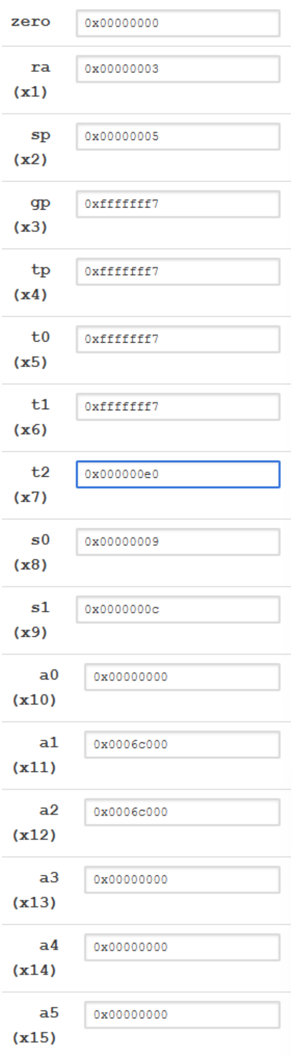
In Case 2:



```
Time              ps      500 ps    510 ps    520 ps    530 ps    540 ps    550 ps    560 ps    570 ps    580 ps    590 ps    600
clk_i=1
rst_ni=1
pc_debug_o[31:0]=0  000000C0  000000C4  000000C8  000000CC  000000D0  000000D4  000000D8  000000DC  000000E0  000000E4  000000E8
addr[31:0]=0        000000C0  000000C4  000000C8  000000CC  000000D0  000000D4  000000D8  000000DC  000000E0  000000E4  000000E8
inst[31:0]=0        0033A023  00000397  FF83A203  FF839283  FF838303  10321463  10329263  10331063  00000397  00339023  00000397
mem[1][31:0]=0      00000003
mem[2][31:0]=0      00000005
mem[3][31:0]=FI     FFFFFFF7
mem[4][31:0]=0      00000000                      FFFFFFF7
mem[5][31:0]=0      00000000                                FFFFFFF7
mem[6][31:0]=0      00000001                                          FFFFFFF7
mem[7][31:0]=0      000000BC            000000C4                                                              000000E0
mem[8][31:0]=0      00000009
mem[9][31:0]=0      0000000C
mem[11][31:0]=0     0006C000
mem[12][31:0]=0     0006C000
mem[13][31:0]=0     00000000
mem[14][31:0]=0     00000000
```

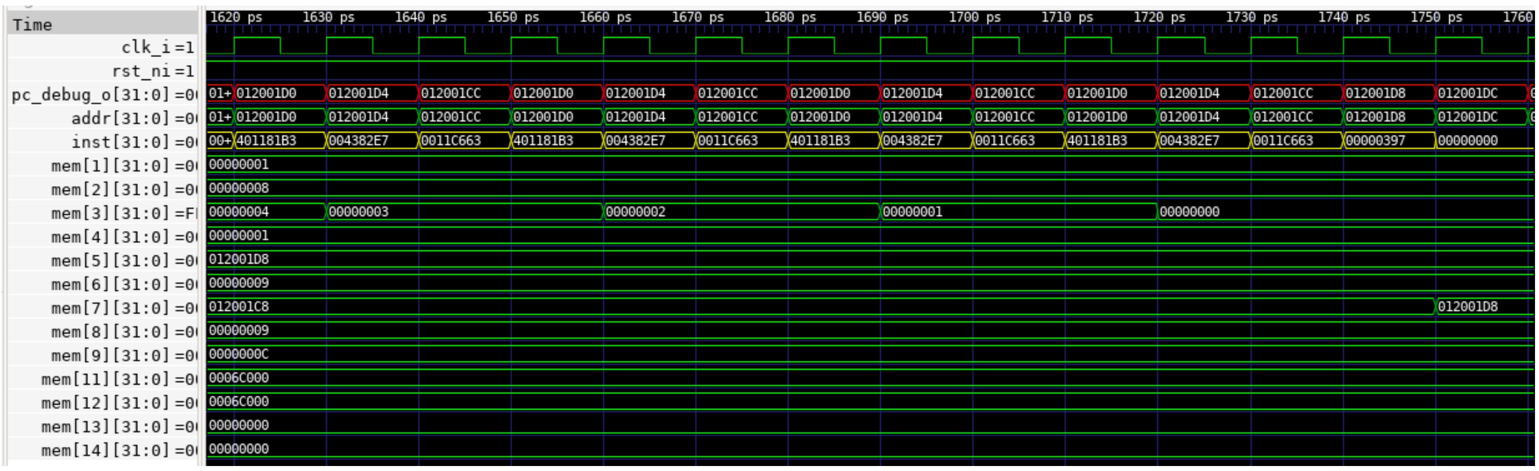


0x00000397	auipc x7 0	auipc x7 0 # Load PC value into x7 = prev_PC + 4
0x0033a023	sw x3 0(x7)	sw x3 0(x7) # Store at address [x7 + 0]
0x00000397	auipc x7 0	auipc x7 0 # Load PC value into x7 = prev_PC + 4
0xff83a203	lw x4 -8(x7)	lw x4 -8(x7) # Load at address [x7 - 8]
0xff839283	lh x5 -8(x7)	lh x5 -8(x7) # Load at address [x7 - 8]
0xff838303	lb x6 -8(x7)	lb x6 -8(x7) # Load at address [x7 - 8]
0x10321463	bne x4 x3 264	bne x4 x3 END # Could not jump to end
0x10329263	bne x5 x3 260	bne x5 x3 END # Could not jump to end
0x10331063	bne x6 x3 256	bne x6 x3 END # Could not jump to end
0x00000397	auipc x7 0	auipc x7 0 # Load PC value into x7 = prev_PC + 4
0x00339023	sh x3 0(x7)	sh x3 0(x7) # Store at address [x7 + 0]

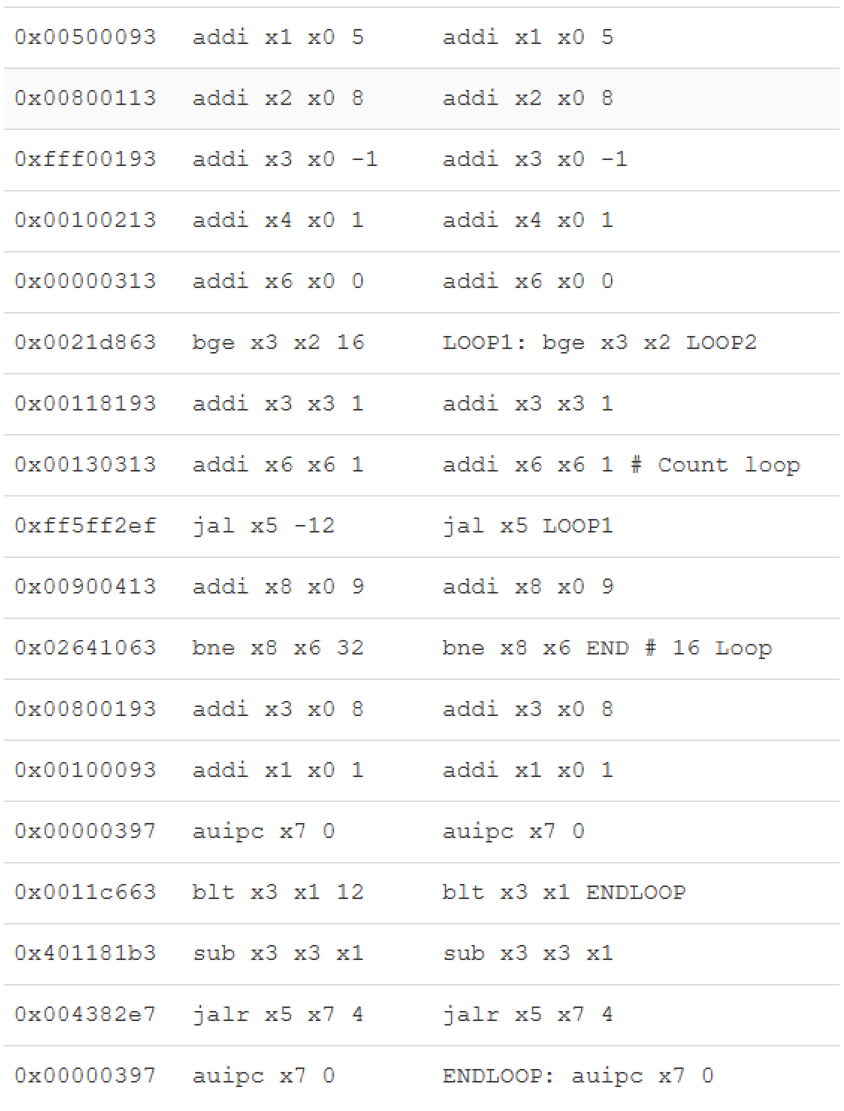


zero	0x00000000
ra (x1)	0x00000003
sp (x2)	0x00000005
gp (x3)	0xfffffff7
tp (x4)	0xfffffff7
t0 (x5)	0xfffffff7
t1 (x6)	0xfffffff7
t2 (x7)	0x000000e0
s0 (x8)	0x00000009
s1 (x9)	0x0000000c
a0 (x10)	0x00000000
a1 (x11)	0x0006c000
a2 (x12)	0x0006c000
a3 (x13)	0x00000000
a4 (x14)	0x00000000
a5 (x15)	0x00000000

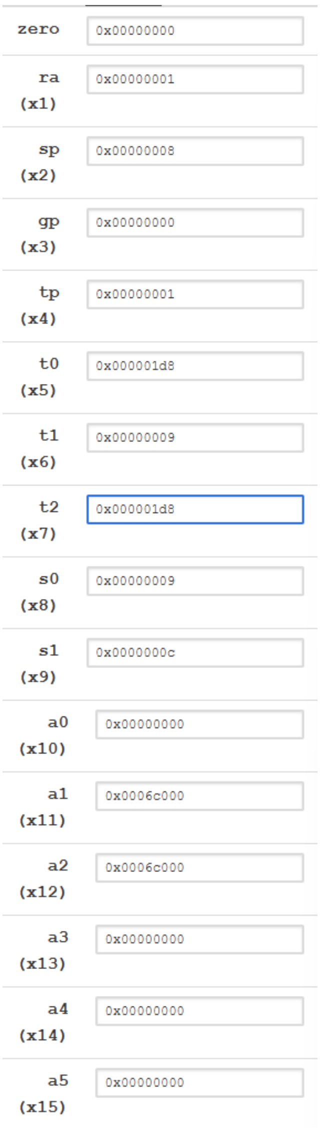
In Case 3:



```
Time               1620 ps   1630 ps   1640 ps   1650 ps   1660 ps   1670 ps   1680 ps   1690 ps   1700 ps   1710 ps   1720 ps   1730 ps   1740 ps   1750 ps   1760
clk_i=1
rst_ni=1
pc_debug_o[31:0]=0 01+ 012001D0  012001D4  012001CC  012001D0  012001D4  012001CC  012001D0  012001D4  012001CC  012001D0  012001D4  012001CC  012001D8  012001DC
addr[31:0]=0       01+ 012001D0  012001D4  012001CC  012001D0  012001D4  012001CC  012001D0  012001D4  012001CC  012001D0  012001D4  012001CC  012001D8  012001DC
inst[31:0]=0       00+ 401181B3  004382E7  0011C663  401181B3  004382E7  0011C663  401181B3  004382E7  0011C663  401181B3  004382E7  0011C663  00000397  00000000
mem[1][31:0]=0     00000001
mem[2][31:0]=0     00000008
mem[3][31:0]=F     00000004  00000003                      00000002                      00000001                      00000000
mem[4][31:0]=0     00000001
mem[5][31:0]=0     012001D8
mem[6][31:0]=0     00000009
mem[7][31:0]=0     012001C8                                                                                                                012001D8
mem[8][31:0]=0     00000009
mem[9][31:0]=0     0000000C
mem[11][31:0]=0    0006C000
mem[12][31:0]=0    0006C000
mem[13][31:0]=0    00000000
mem[14][31:0]=0    00000000
```



0x00500093	addi x1 x0 5	addi x1 x0 5
0x00800113	addi x2 x0 8	addi x2 x0 8
0xfff00193	addi x3 x0 -1	addi x3 x0 -1
0x00100213	addi x4 x0 1	addi x4 x0 1
0x00000313	addi x6 x0 0	addi x6 x0 0
0x0021d863	bge x3 x2 16	LOOP1: bge x3 x2 LOOP2
0x00118193	addi x3 x3 1	addi x3 x3 1
0x00130313	addi x6 x6 1	addi x6 x6 1 # Count loop
0xff5ff2ef	jal x5 -12	jal x5 LOOP1
0x00900413	addi x8 x0 9	addi x8 x0 9
0x02641063	bne x8 x6 32	bne x8 x6 END # 16 Loop
0x00800193	addi x3 x0 8	addi x3 x0 8
0x00100093	addi x1 x0 1	addi x1 x0 1
0x00000397	auipc x7 0	auipc x7 0
0x0011c663	blt x3 x1 12	blt x3 x1 ENDLOOP
0x401181b3	sub x3 x3 x1	sub x3 x3 x1
0x004382e7	jalr x5 x7 4	jalr x5 x7 4
0x00000397	auipc x7 0	ENDLOOP: auipc x7 0



zero	0x00000000
ra (x1)	0x00000001
sp (x2)	0x00000008
gp (x3)	0x00000000
tp (x4)	0x00000001
t0 (x5)	0x000001d8
t1 (x6)	0x00000009
t2 (x7)	0x000001d8
s0 (x8)	0x00000009
s1 (x9)	0x0000000c
a0 (x10)	0x00000000
a1 (x11)	0x0006c000
a2 (x12)	0x0006c000
a3 (x13)	0x00000000
a4 (x14)	0x00000000
a5 (x15)	0x00000000

4. EVALUATION

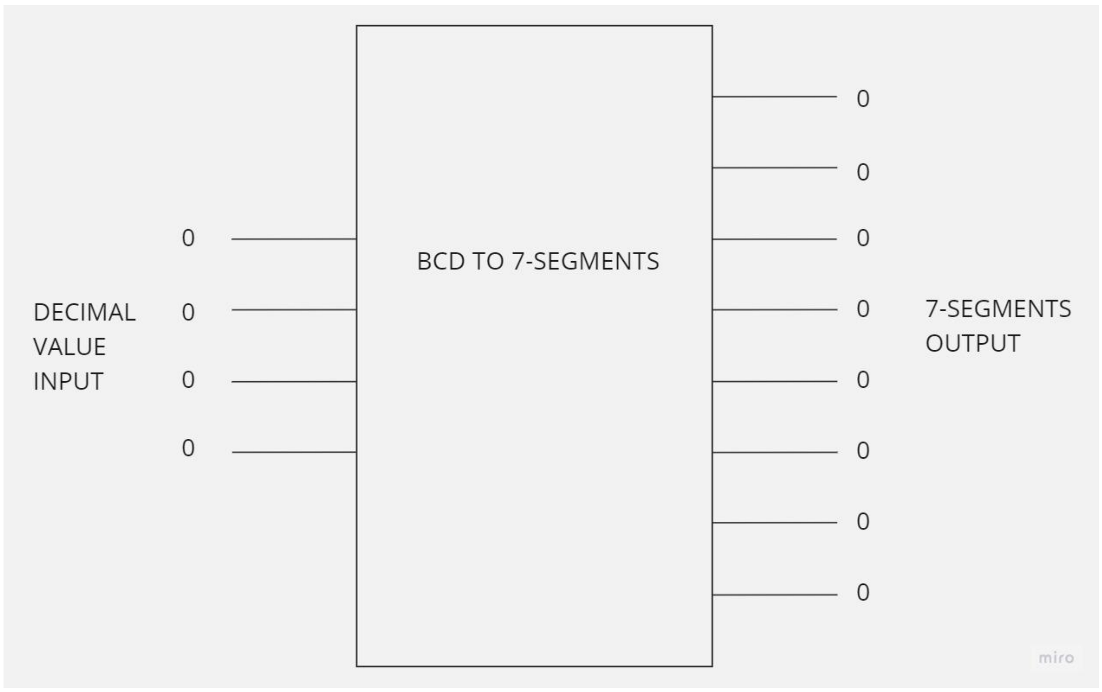
4.1. Waveform and testbench

As the program is too long for capturing pictures, we just check some important path of the waveform and compare the result to the RISC V result on the website.

4.2. Application

In this part, we want to use the Single Cycle Processor to convert a hexadecimal number to a decimal number and display on seven-segment LEDs of the DE II kit.

After completing building the Single Cycle Processor, we continue to write the application instructions in RV32I ISA assembly and put the code to the IMEM. Moreover, we add a BCD which converts the decimal value to 7-segments LEDs at the Output peripheral of the LSU for the display of the decimal number.



BCD to 7-Segments LEDs block diagram

About the conversion from hexadecimal to decimal:

```
loop:
 addi x1 x0 0x7FF    # Add the value want to display on seg7 is 2047 (max)
 addi x2 x0 10       # parameter HEX1
 addi x3 x0 100      # parameter HEX2
 addi x4 x0 1000     # parameter HEX3

 addi x14 x0 0                 # Clear register x14   (HEX3)
 addi x13 x0 0                 # Clear register x13   (HEX2)
 addi x12 x0 0                 # Clear register x12   (HEX1)
 addi x11 x0 0                 # Clear register x11   (HEX0)



HEX3_LOOP:
blt x1 x4 HEX2_LOOP
sub x1 x1 x4
addi x14 x14 1             # Increase the thousands value HEX3 to 1 after each comparision
jal x7 HEX3_LOOP

HEX2_LOOP:
blt x1 x3 HEX1_LOOP
sub x1 x1 x3
addi x13 x13 1             # Increase the hunreds value HEX3 to 1 after each comparision
jal x7 HEX2_LOOP

HEX1_LOOP:
blt x1 x2 HEX0_LOOP
sub x1 x1 x2
addi x12 x12 1             # Increase the tens value HEX3 to 1 after each comparision
jal x7 HEX1_LOOP

HEX0_LOOP:
```

```
add x11 x0 x1                    # Increase the units value HEX3 to 1 after each comparision

# Set the display value for HEX0
addi x7, zero, -2048
sw x11, 0(x7)

# Set the display value for HEX1
addi x7, zero, -2032
sw x12, 0(x7)

# Set the display value for HEX2
addi x7, zero, -2016
sw x13, 0(x7)

# Set the display value for HEX3
addi x7, zero, -2000
sw x14, 0(x7)

jal x7, loop
```

1. Initialize the necessary values: **x1** is initialized to 2047 (**0x7FF**), and **x2**, **x3**, **x4** are initialized to **10**, **100**, **1000** respectively for the algorithm.

2. Initialize registers **x11**, **x12**, **x13**, **x14** to 0 to store the value corresponding to each row of the 7-segment synchronous clock (units, tens, hundreds, thousands) which address is HEX0, HEX1, HEX2 respectively.

3. Start **HEX3_LOOP** loop: Check if the value of **x1** is less than **x4**. If true, exit the loop and continue to the next loop, otherwise subtract **x4** from **x1**, increase the value of **x14** by 1 and repeat the loop.

4. Continuing with **HEX2_LOOP** and **HEX1_LOOP**, do the same as **HEX3_LOOP** for the values of **x3** and **x2**, increasing the values of **x13** and **x12** respectively.

5. **HEX0_LOOP**: Add the current value of **x1** to **x11**, matching the row value unit.

6. The final code of this section is to set the value of the registers (**x11**, **x12**, **x13, x14**) to the required position to display **HEX0**, **HEX1**, **HEX2** on the 7-segment LED.

7. Finally, execute **jal x7, loop** to start the loop again.

4.3. Evaluation

Structure:

About the structure, we have added a new BCD to 7 segments for the application at the output so that the display can be doing well.

Instruction execution time:

The Single Cycle Processor assumes that each instruction is executed in one clock cycle. However, the assembly code includes many statements and loops, it can make execution time slow and affect the performance.

Command dependency:

All commands are carefully tested and checked to run effectively before being added to the Single Cycle Processor to ensure that it can display the decimal value on the 7-segments.

Performance:

The Single Cycle Processor is typically designed to execute basic instructions in one clock cycle and does not respond well to complex or multi-branch code but in this case, we see that the processor is still doing well to solve this problem efficiently.

Memory management:

In this case, the data memory of this processor is from 0 to 2047 bits so it is very convenient for storing and doing this application.

5. CONCLUSION

In conclusion, the development of a single-cycle processor based on the RV32 Instruction Set Architecture (ISA) has been a comprehensive and enlightening project. The primary goal of this endeavor was to design a processor capable of executing RV32 instructions in a single clock cycle, emphasizing simplicity and straightforward execution.

The successful completion of this project underscores the importance of a systematic approach to processor design, encompassing architecture, instruction set implementation, and rigorous testing methodologies. The project not only enhanced understanding of the RV32 ISA but also provided valuable insights into the challenges and considerations involved in designing processors for real-world applications.