VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRONICS

EE3043 – Computer Architecture

Semaster 241

Group : **23**

Student : **Hoang Minh Chien** **(2112932)**

Doan Dinh Nam **(2110368)**

Giang Anh Đức **(2011103)**

Teacher : **Dr. Tran Hoang Linh**

TA : **Msc. Hai Cao Xuan**

Ho Chi Minh City, November 17, 2024

PREFACE

This project centers around the design and implementation of a **RISC-V Single-Cycle 32-bit Processor** adhering to the Base Integer (RV32I) instruction set architecture. The project aims to explore the practical aspects of computer architecture by combining theoretical knowledge with real-world applications. The processor was implemented using Verilog HDL, rigorously tested through testbenches, synthesized, and successfully deployed on the **DE2-Standard FPGA development board**.

The design incorporates fundamental components essential for a fully functional RISC-V processor:

- **Arithmetic Logic Unit (ALU):** Handles core computational tasks required by the RV32I instruction set.
- **Register File:** Fully compliant with RISC-V specifications, providing 32 registers (32-bit wide) with Register 0 reserved as zero.
- **Branch Comparison Unit:** Enables efficient execution of branching instructions.
- **Load-Store Unit:** Facilitates data transfers between the processor and memory using a defined memory map.
- **Control Unit:** Orchestrates the execution of instructions and ensures the proper synchronization of signals across components.

To validate the processor's functionality, assembly programs were developed, assembled into machine code (hex format), and loaded into the **Instruction Memory (IMEM)**. These programs were executed on the FPGA, demonstrating the processor's capability to handle real-world scenarios. The project culminated in three key application demonstrations:

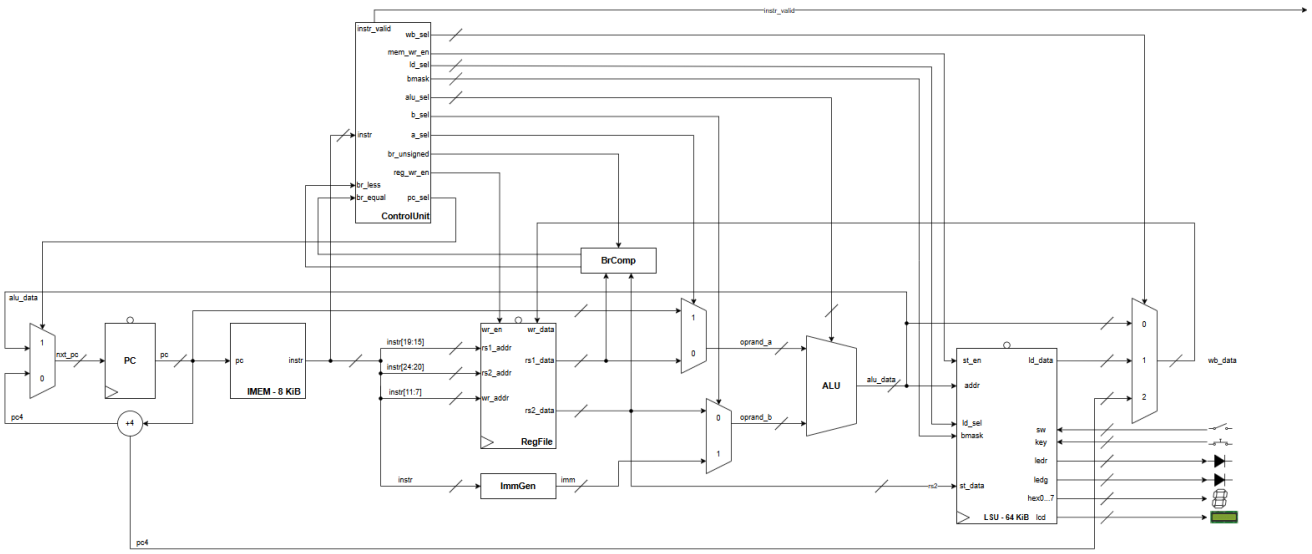
1. **Switch to Segment Mapping:** Translating input from switches to output on 7-segment displays.
2. **Stopwatch Application:** Implementing a simple timer to showcase the processor's ability to handle time-sensitive operations.
3. **"Hello World" on LCD:** Interfacing with the LCD to display text, demonstrating peripheral interaction.

Through this project, we gained invaluable insights into the process of designing, testing, and deploying a custom processor. The hands-on implementation reinforced our understanding of the RISC-V architecture while bridging the gap between simulation and hardware realization. This document provides a comprehensive overview of the processor's design, the challenges encountered, and the solutions implemented, paving the way for further exploration and development in processor design.

CONTENTS

1. DESIGN STRATEGY

1.1. Overall Design

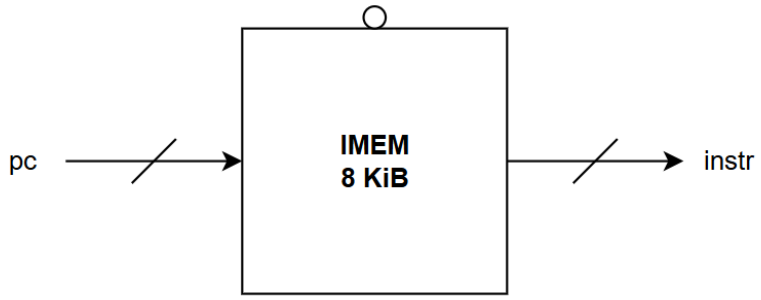


This diagram illustrates the architecture of a **RISC-V Single Cycle 32-bit Processor** with the Base Integer Instruction Set (RV32I). Key components and their functions are summarized below:

1. **IMEM (Instruction Memory):** Stores the program's instructions. Fetches the instruction at the address from the PC.
2. **RegFile (Register File):** Holds 32 registers (32-bit each). Supports two reads and one write per cycle. Register x0 is always 0.
3. **ImmGen (Immediate Generator):** Extracts and decodes the immediate values from instructions.
4. **ControlUnit:** Decodes instructions and generates control signals for other components (e.g., ALU, branch logic, memory).
5. **ALU (Arithmetic Logic Unit):** Performs arithmetic, logical operations, and bypass oprand_b feature (for LUI). Takes inputs from registers or immediate values.
6. **BrComp (Branch Comparator):** Compares register values to determine if a branch should be taken.
7. **LSU (Load-Store Unit):** Handles data memory access for load and store instructions. Interfaces with peripherals such as switches, LEDs, and LCD.

1.2. Instruction Memory (IMEM)

Block Diagram



I/O Description

Name	I/O	Width	Description
pc	Input	13	13-bit program counter address to fetch an instruction.
instr	Output	32	32-bit instruction output corresponding to the given address.

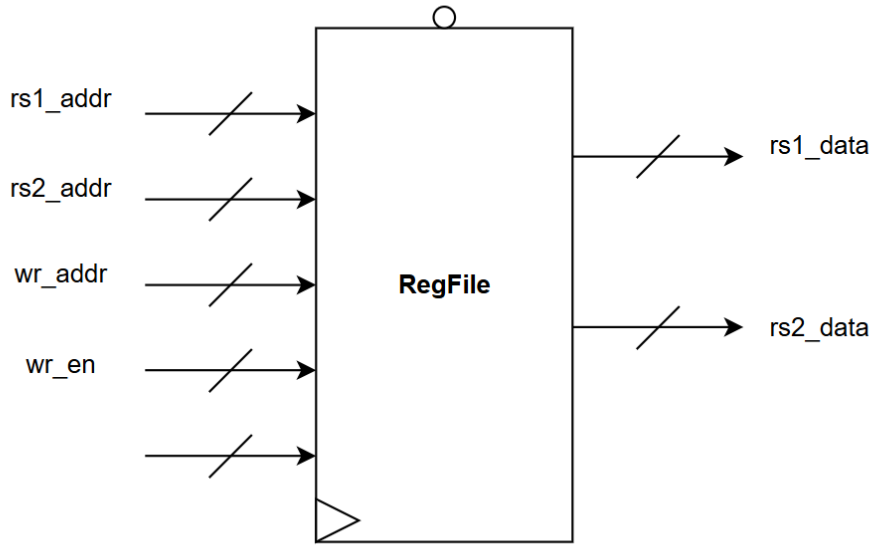
Detailed Descripton

An IMEM, which is NOT a sequential circuit since it's have no clock input, when the address appears at the input, the instruction will immediately program to output without waiting for posedge clk.

The same methodolody (read without clock) will be applied to the Data Memory (DMEM) inside a Load-Store Unit.

1.3. Register File

Block Diagram



I/O Description

Name	I/O	Width	Description
clk	Input	1	Global clock
rst_n	Input	1	Global active low reset
rs1_addr	Input	5	Address of the first source register
rs2_addr	Input	5	Address of the second source register
wr_addr	Input	5	Address of the destination register
wr_en	Input	1	Write enable for the destination register
rs1_data	Output	32	Data from the first source register
rs2_data	Output	32	Data from the second source register

Detailed Descripton

The Registers File consists of 32 32-bit registers, with register 0 always reading out as 0.

1.4. Immediate Generator

Block Diagram

I/O Description

Name	I/O	Width	Description
instr	Input	32	The 32-bit instruction input fetched from Instruction Memory.
imm	Output	32	The 32-bit immediate value extracted from the instruction.

Detailed Descripton

Immediate Format	Bit Extraction
I-type	{signxt[31:11], instr[30:20]}
S-type	{Stype[31:5], instr[11:7]}
B-type	{Btype[31:12], instr[7], Btype[10:1], 1'b0}
U-type	{instr[31:12], 12'd0}
J-type	{Jtype[31:20], instr[19:12], instr[20], Jtype[10:1], 1'b0}

The **Immediate Generator** is a key component in the design of a RISC-V CPU. Its main role is to extract and arrange immediate values from instructions in various formats (I, B, S, J, and U types) as defined by the RISC-V specification. These immediate values are crucial for forming the correct data for operations like memory addressing or arithmetic.

In detail:

- **I-Type Immediate:** Serves as a starting point with minimal sign extension, providing a reference for other formats.
- **S-Type Immediate:** Derived from the I-Type format using a multiplexer (MUX) to rearrange bits accordingly.
- **B-Type Immediate:** Similar to the S-Type but includes masking operations for conditional branches.
- **J-Type Immediate:** Derived using specific masking patterns to suit jump instructions.
- **U-Type Immediate:** Unique and distinct from other formats, used directly for larger immediate values.

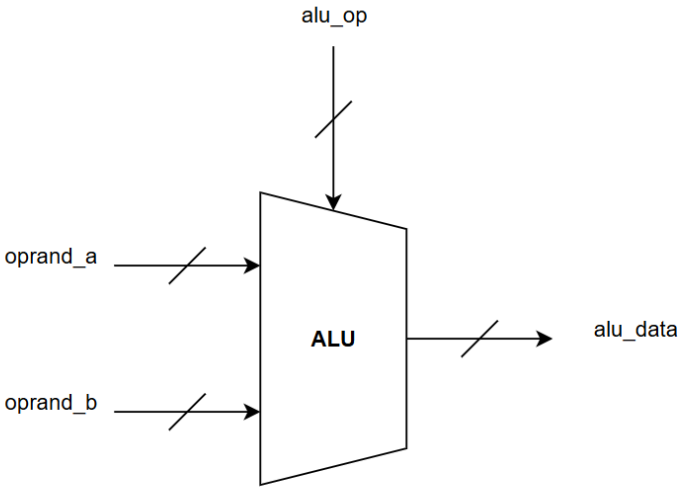
The selection of the immediate type is managed by the imm_sel signal, which is a one-hot encoded control signal with 5 bits. This ensures precise routing of the instruction bits to generate the correct immediate value.

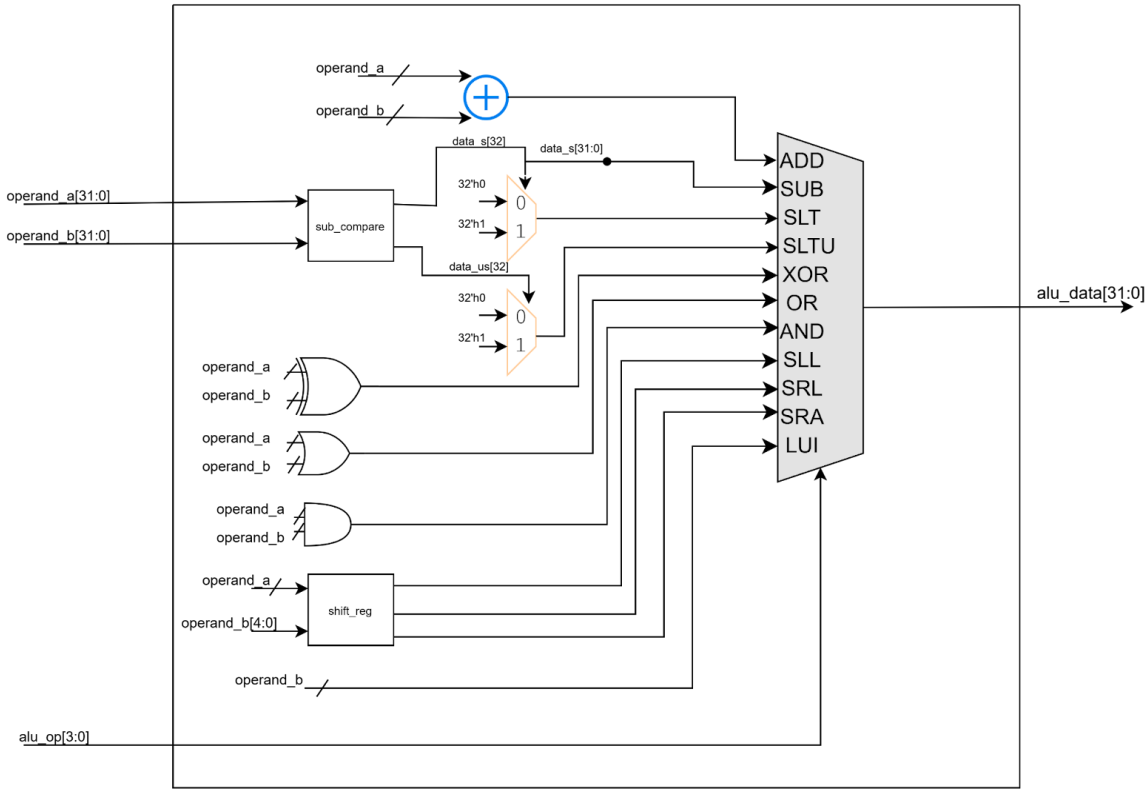
The 32-bit instruction in a Single-Cycle RISC-V CPU consists of multiple fields, such as the operation code (opcode), operands, and immediate values. One of these fields indicates whether the instruction requires an immediate value or a register value. The Immediate Generator uses these fields to output the correct immediate value based on the instruction format.

In summary, the Immediate Generator efficiently decodes and arranges the immediate bits for different instruction types, ensuring seamless execution in the processor pipeline.

1.5. ALU

Block Diagram





I/O Description

Name	I/O	Width	Description
operand_a	Input	32	First operand for ALU operations
operand_b	Input	32	Second operand for ALU operations
alu_op	Input	4	The operation to be performed
alu_data	Output	32	Result of the ALU operation

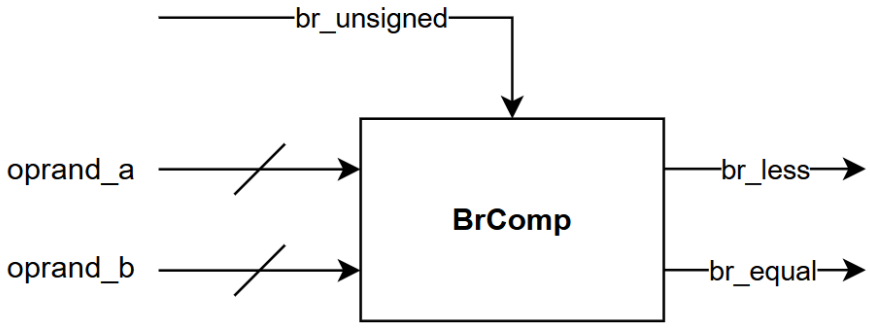
Detailed Descripton

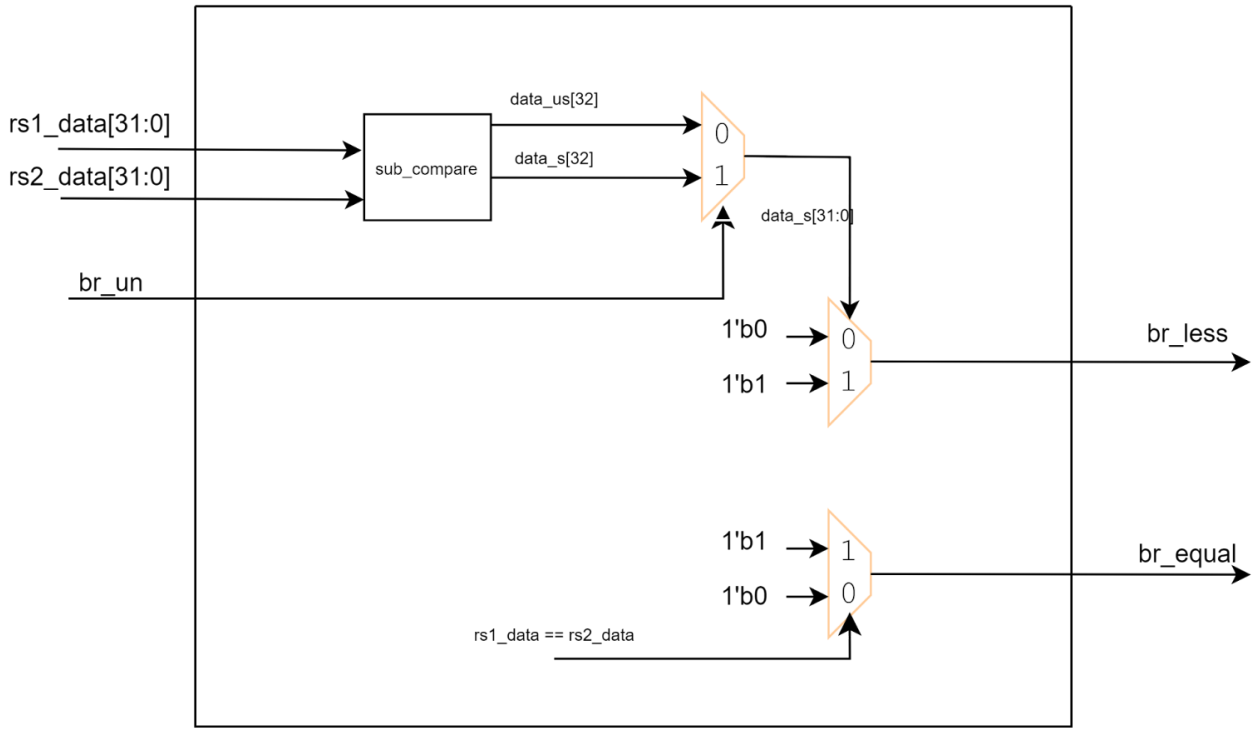
The *sub_compare* module performs the subtraction operation and simultaneously compares the two operands from that subtraction. The subtraction is executed by adding *operand_a* to the two's complement of *operand_b*. To avoid overflow during the calculation, an additional 32nd bit is added to both inputs, and this overflow bit also serves as the bit that detects the *less* signal.

The *shift_reg* module will perform the functions of logical and arithmetic left and right shifts by generating all possible shift cases for the 5-bit *operand_b* (corresponding to 32 cases).

1.6. Branch Compare

Block Diagram





I/O Description

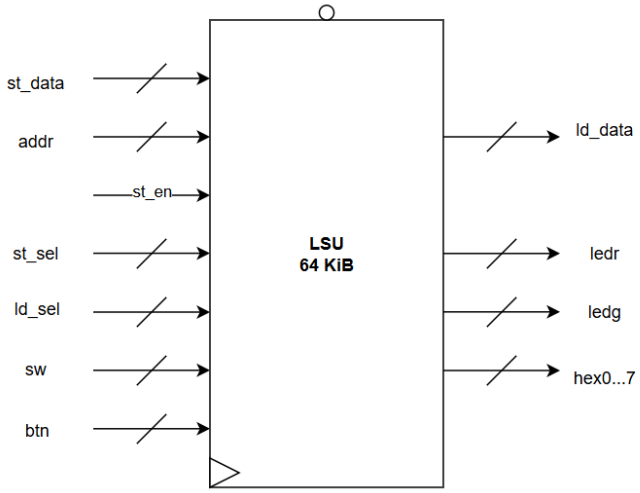
Name	I/O	Width	Description
rs1_data	Input	32	Data from the first register
rs2_data	Input	32	Data from the second register
br_un	Input	1	Comparison mode (1 if signed, 0 if unsigned)
br_equal	Output	1	Output is 1 if rs1 = rs2
br_less	Output	1	Output is 1 if rs1 < rs2

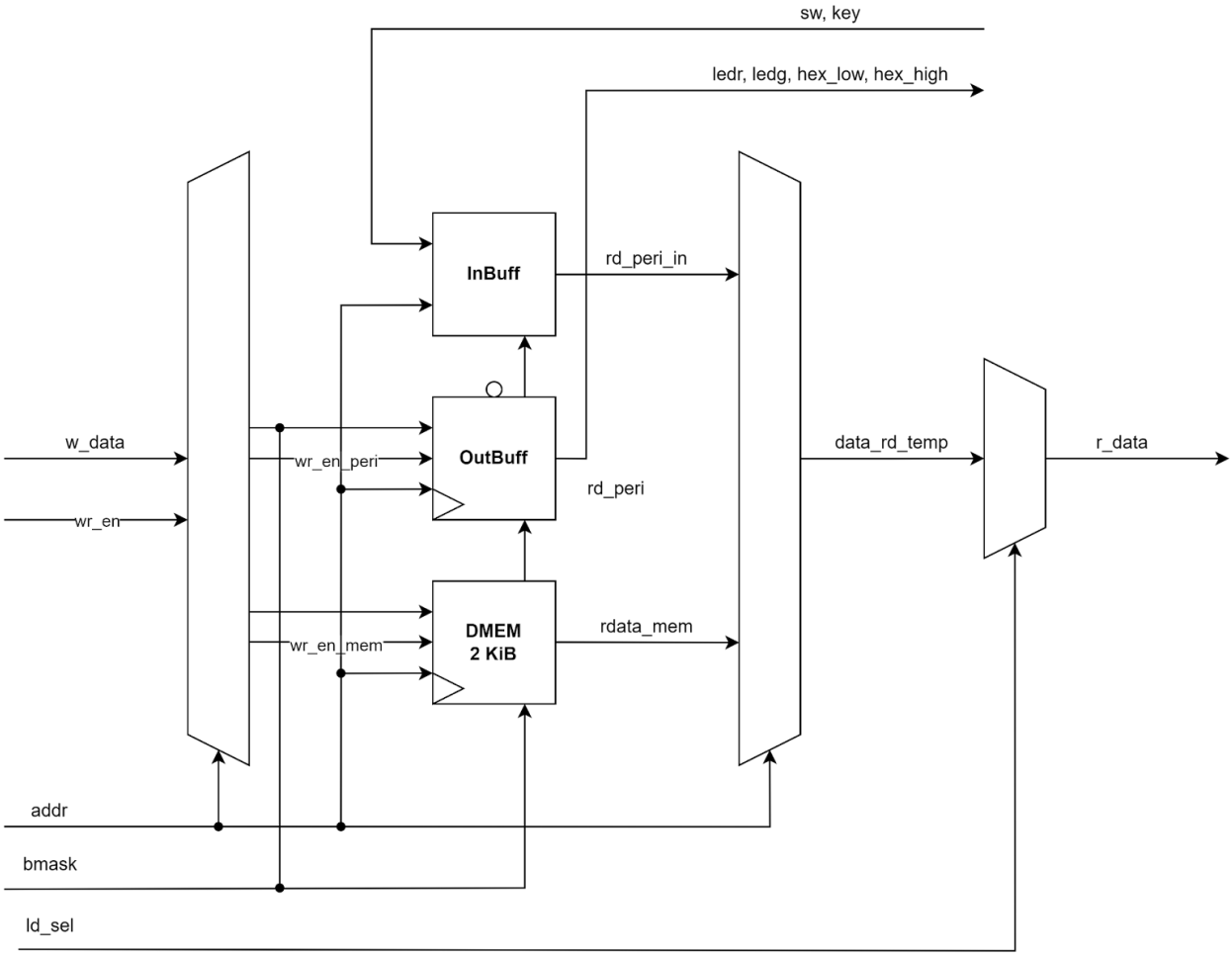
Detailed Descripton

As explained, the 32nd overflow bit after subtracting the two operands will determine which number is smaller. If the number is unsigned, the extended bit will be 0; if it is signed, the extended bit will be the MSB of the operand.

1.7. Load-Store Unit

Block Diagram





I/O Description

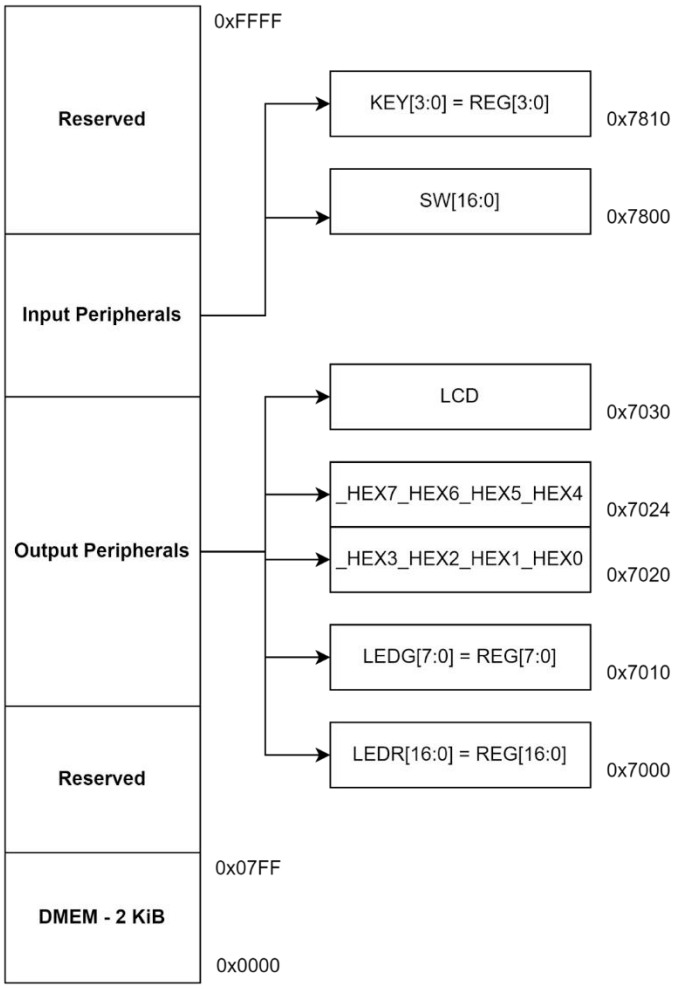
Name	I/O	Width	Description
clk	Input	1	Global clock
rst_n	Input	1	Global active low reset
addr	Input	16	Address for data read/write
w_data	Input	32	Data to be stored
wr_en	Input	1	Write enable signal (1 if writing)
bmask	Input	4	Byte mask, with each bit corresponding to one byte in the 32-bit word
ld_sel	Input	3	Select byte to be loaded
r_data	Output	32	Data read from memory
io_ledr	Output	32	Output for red LEDs
io_ledg	Output	32	Output for green LEDs
io_hex0…7	Output	7	Output for 7-segment displays
io_lcd	Output	32	Output for the LCD register
io_sw	Input	32	Input for switches
io_btn	Input	4	Input for buttons

Detailed Descripton

The LSU can load or store data in the same data memory. However, in real-world applications, a processor interfaces with peripheral devices to transmit or receive data through the implementation of an I/O system. Common peripherals include LEDs, LCDs, 7-segment displays, switches, buttons, and more. Furthermore, the Load-Store Unit (LSU) has been specifically designed to accommodate a range of data transfer operations such as Load Halfword (LH), Load Byte (LB), Load Halfword Unsigned (LHU), Load Byte Unsigned (LBU), Store Halfword (SH), and Store Byte (SB). By using a signal called *bmask*

to select which byte is stored and a signal *ld_sel* to select which byte is loaded, these operations are enabled.

I/O Memory Mapping



I/O System Conventions

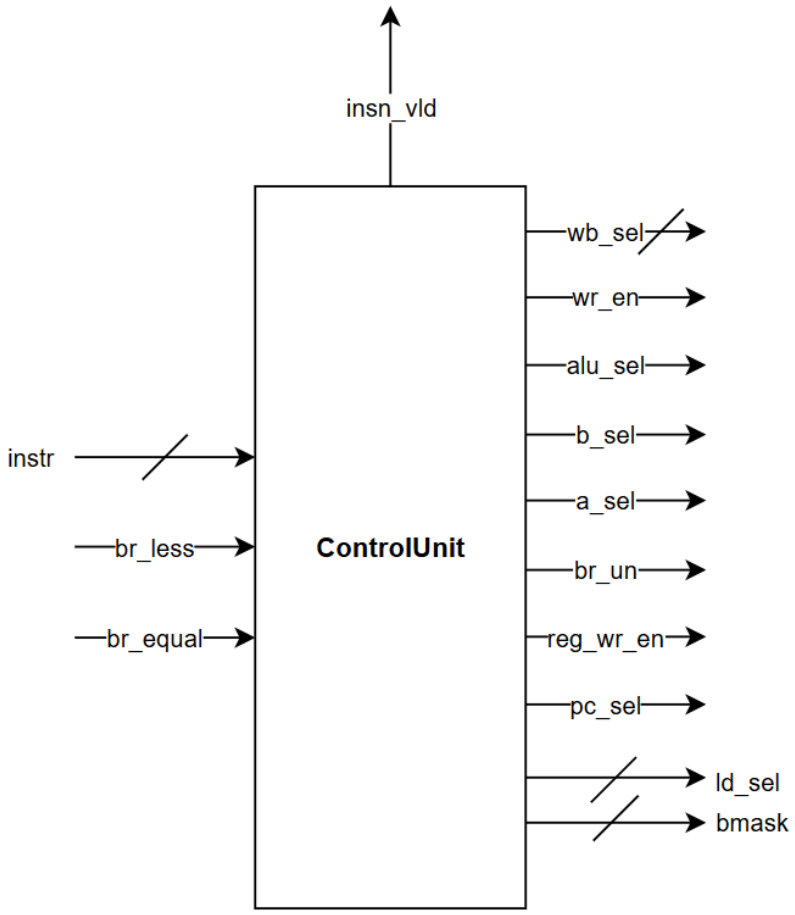
For consistent operation and testing, adhere to the following conventions for setting up with DE2 boards and interacting with the I/O system

Address	Name	Bits	Usage
0x7000	io_ledr	31 - 18	Reserved
		17 - 0	18-bit data connected to the array of 17 red LEDs in order
0x7010	io_ledg	31 - 8	Reserved
		7 - 0	8-bit data connected to the array of 8 green LEDs in order
0x7020	io_hex0…3	31	Reserved
		30 - 24	7-bit data to HEX3
		23	Reserved
		22 - 16	7-bit data to HEX2
		15	Reserved
		14 - 8	7-bit data to HEX1
		7	Reserved
		6 - 0	7-bit data to HEX0
0x7024	io_hex7…4	31	Reserved
		30 - 24	7-bit data to HEX7
		23	Reserved
		22 - 16	7-bit data to HEX6
		15	Reserved
		14 - 8	7-bit data to HEX5
		7	Reserved
		6 - 0	7-bit data to HEX4
0x7030	io_lcd	31 - 11	Reserved
		10	EN
		9	RS
		8	RW

		7 - 0	8-bit data
0x7080	io_sw	31 - 18	Reserved
		17	Global Reset
		16 - 0	17-bit data from SW16 to SW0 respectively.

1.8. Control Unit

Block Diagram



I/O Description

Name	I/O	Width	Description
instr	Input	32	The instruction of the statement
br_less	Input	1	Data from Branch Comparison, 1 if A < B.
br_equal	Input	1	Data from Branch Comparison, 1 if A = B.
wb_sel	Output	2	Select data to write into Reggile: 0 if alu_data, 1 if ld_data, 2 if pc_four
wr_en	Output	1	1 if the instruction writes data into LSU
alu_sel	Output	4	Choose the opcode to perform operations for the ALU

a_sel	Output	1	Select operand A source: 0 if *rs1*, 1 if *PC*
b_sel	Output	1	Select operand B source: 0 if *rs2*, 1 if *imm*.
br_un	Output	1	1 if the two operands are signed, 0 if unsigned
reg_wr_en	Output	1	1 if the instruction writes data into Regfile
pc_sel	Output	1	Select PC source: 0 if *PC* + 4, 1 if computed in ALU.
ld_sel	Output	3	Select byte to be loaded from LSU
bmask	Output	4	Select byte to be stored into LSU
insn_vld	Output	1	0 if illegal instruction, 0 if legal instruction

Detailed Descripton

The Control Unit module will function as the “brain”, controlling all the necessary signals for the design's components. The input is a 32-bit instruction that determines which instruction is being executed. Based on this, the Control Unit will send control signals to the other blocks corresponding to that instruction.

For instructions that need to handle bytes, I created two additional signals: *bmask* for Store operations and *ld_sel* for Load operations. When the Control Unit receives a Load or Store instruction, these signals will be activated based on the type of instruction.

The signals driven for each instruction are listed in the table below.

Control Unit table:

Instr	FMT	Opcode	funct3	funct7	Description	br_less	br_equal	pc_sel	rd_wren	br_unsigned	op_a_sel	op_b_sel	alu_op	mem_wren	wb_sel	bmask	ld_sel
LUI rd, imm	U	0110111			rd = imm << 12			0	1			1	1011	0	00		
AUIPC rd, imm	U	0010111			rd = PC + (imm << 12)			0	1		1	1	0000	0	00		
JAL rd, imm JAL ra, LABEL	J	1101111			rd=PC+4; PC+=imm			1	1		1	1	0	0	10		
JALR rd, imm(rs1) JALR x0, ra, 0	I	1100111	0x0		rd=PC+4; PC=rs1+imm			1	1		0	1	0	0	10		
BEQ rs1, rs2, imm	B	1100011	0x0		if(rs1 == rs2) PC += imm		1	1	0	0	1	1	0000	0			
							0	0	0	0			0000	0			
BNE rs1, rs2, imm	B	1100011	0x1		if(rs1 != rs2) PC += imm		0	1	0	0	1	1	0000	0			
							1	0	0	0			0000	0			
BLT rs1, rs2, imm	B	1100011	0x4		if(rs1 < rs2) PC += imm	1		1	0	0	1	1	0000	0			
						0		0	0	0			0000	0			
BGE rs1, rs2, imm	B	1100011	0x5		if(rs1 >= rs2) PC += imm	0	1	1	0	0	1	1	0000	0			
						0	0	1	0	0	1	1	0000	0			
						1		0	0	0			0000	0			
BLTU rs1, rs2, imm	B	1100011	0x6		if(rs1 < rs2) PC += imm	1		1	0	1	1	1	0000	0			
						0		0	0	1			0000	0			
BGEU rs1, rs2, imm	B	1100011	0x7		if(rs1 >= rs2) PC += imm	0	1	1	0	1	1	1	0000	0			
						0	0	1	0	1	1	1	0000	0			
						1	0	0	0	1			0000	0			
LB rd, imm(rs1)	I	0000011	0x0		rd = M[rs1+imm][0:7]			0	1		0	1	0000	0	1		000
LW rd, imm(rs1)	I	0000011	0x1		rd = M[rs1+imm][0:15]			0	1		0	1	0000	0	1		001
LBU rd, imm(rs1)	I	0000011	0x4		rd = M[rs1+imm][0:7]			0	1		0	1	0000	0	1		011
LHU rd, imm(rs1)	I	0000011	0x5		rd = M[rs1+imm][0:15]			0	1		0	1	0000	0	1		100
LW rd, imm(rs1)	I	0000011	0x2		rd = M[rs1+imm][0:31]			0	1		0	1	0000	0	1		010
SB rs2, imm(rs1)	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]			0	0		0	1	0000	1		0001	
SH rs2, imm(rs1)	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]			0	0		0	1	0000	1		0011	
SW rs2, imm(rs1)	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]			0	0		0	1	0000	1		1111	
ADDI rd, rs1, imm	I	0010011	0x0		rd = rs1 + imm			0	1		0	1	0000	0	0		
SLTI rd, rs1, imm	I	0010011	0x2		rd = (rs1 < imm)?1:0			0	1		0	1	0010	0	0		
SLTIU rd, rs1, imm	I	0010011	0x3		rd = (rs1 < imm)?1:0			0	1		0	1	0011	0	0		
XORI rd, rs1, imm	I	0010011	0x4		rd = rs1 ^ imm			0	1		0	1	0100	0	0		
ORI rd, rs1, imm	I	0010011	0x6		rd = rs1 | imm			0	1		0	1	**0110**	0	0		
ANDI rd, rs1, imm	I	0010011	0x7		rd = rs1 & imm			0	1		0	1	**0111**	0	0		
SLLI rd, rs1, imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]			0	1		0	1	**0001**	0	0		
SRLI rd, rs1, imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]			0	1		0	1	**0101**	0	0		
SRAI rd, rs1, imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]			0	1		0	1	**1101**	0	0		
ADD rd, rs1, rs2	R	0110011	0x0	0x00	rd = rs1 + rs2			0	1		0	0	0000	0	0		
SUB rd, rs1, rs2	R	0110011	0x0	0x20	rd = rs1 - rs2			0	1		0	0	1000	0	0		
SLT rd, rs1, rs2	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0			0	1		0	0	0010	0	0		
SLTU rd, rs1, rs2	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0			0	1		0	0	0011	0	0		
XOR rd, rs1, rs2	R	0110011	0x4	0x00	rd = rs1 ^ rs2			0	1		0	0	0100	0	0		
OR rd, rs1, rs2	R	0110011	0x6	0x00	rd = rs1 | rs2			0	1		0	0	0110	0	0		
AND rd, rs1, rs2	R	0110011	0x7	0x00	rd = rs1 & rs2			0	1		0	0	0111	0	0		
SLL rd, rs1, rs2	R	0110011	0x1	0x00	rd = rs1 << rs2			0	1		0	0	0001	0	0		
SRL rd, rs1, rs2	R	0110011	0x5	0x00	rd = rs1 >> rs2			0	1		0	0	0101	0	0		
SRA rd, rs1, rs2	R	0110011	0x5	0x20	rd = rs1 >> rs2			0	1		0	0	1101	0	0		

2. VERIFICATION STRATEGY

2.1. Instruction Memory (IMEM)

We just simply create a testbench that read out all meaning data in cells.

If it's the same with hex file (for $readmemh task), it's correct.

```
VSIM 10> run -all
# IMEM[0000] = 00007437
# IMEM[0004] = 00040413
# IMEM[0008] = 00040493
# IMEM[000c] = 01040913
# IMEM[0010] = 02040993
# IMEM[0014] = 02440a13
# IMEM[0018] = 03040a93
# IMEM[001c] = 40040b13
# IMEM[0020] = 400b0b13
# IMEM[0024] = 010b0b93
# IMEM[0028] = 10000393
```

mem > 1_hex2seg.hex

```
1   00007437
2   00040413
3   00040493
4   01040913
5   02040993
6   02440A13
7   03040A93
8   40040B13
9   400B0B13
10  010B0B93
```

2.2. Register File

The testbench for the 32-bit Register File ensures its functionality through systematic testing of reset, write, and read operations. The reset sequence verifies proper initialization of all registers. Randomized write operations populate registers with data, while subsequent read operations confirm data integrity. Multiple iterations are performed using randomized inputs to cover various scenarios and edge cases. The goal is to ensure data consistency, accurate simultaneous reads, and robust behavior under all tested conditions.

```
VSIM 12> run -all
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write FAILED at adress = 00000, data = 00f3e301, data in reg = 00000000
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write PASSED
# Write FAILED at adress = 00000, data = d7563eae, data in reg = 00000000
```

```
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
# Read Rs2 PASSED
# Read Rs1 PASSED
```

2.3. Immediate Generator

The testbench validates the Immediate Generator (ImmGen) module by testing 27 RISC-V instructions, including lui, auipc, jal, branch, load, store, and immediate arithmetic operations. For each instruction, the testbench compares the generated immediate value (imm) with the expected result and outputs PASSED or FAILED. It ensures correct immediate extraction, proper sign extension, and accurate handling of various instruction formats. The simulation results and a waveform dump are generated for debugging and analysis, confirming the module's functionality under diverse scenarios.

```
VSIM 14> run -all
# 1. lui x5, 4            : T =  5, instr = 000042b7, imm = 00004000,   |   PASSED
# 2. auipc x6, 11         : T = 10, instr = 0000b317, imm = 0000b000,   |   PASSED
# 3. jal x10, 8           : T = 15, instr = 0080056f, imm = 00000008,   |   PASSED
# 4. jalr x3, 12(x6)      : T = 20, instr = 00c301e7, imm = 0000000c,   |   PASSED
# 5. beq x20, x19, 12     : T = 25, instr = 013a0663, imm = 0000000c,   |   PASSED
# 6. bne x20, x19, 10     : T = 30, instr = 013a1563, imm = 0000000a,   |   PASSED
# 7. blt x20, x19, 4      : T = 35, instr = 013a4263, imm = 00000004,   |   PASSED
# 8. bge x15, x19, 6      : T = 40, instr = 0137d363, imm = 00000006,   |   PASSED
# 9. bltu x3, x5, 8       : T = 45, instr = 0051e463, imm = 00000008,   |   PASSED
# 10. bgeu x15, x16, 16   : T = 50, instr = 0107f863, imm = 00000010,   |   PASSED
# 11. lb x5, 3(x2)        : T = 55, instr = 00310283, imm = 00000003,   |   PASSED
# 12. lbu x6, 1(x15)      : T = 60, instr = 0017c303, imm = 00000001,   |   PASSED
# 13. lhu x6, 8(x15)      : T = 65, instr = 0087d303, imm = 00000008,   |   PASSED
# 14. sb x6, 36(x15)      : T = 70, instr = 02678223, imm = 00000024,   |   PASSED
# 15. sh x6, 20(x15)      : T = 75, instr = 00679a23, imm = 00000014,   |   PASSED
# 16. lw x20, 0(x6)       : T = 80, instr = 00032a03, imm = 00000000,   |   PASSED
# 17. lh x6, 2(x15)       : T = 85, instr = 00279303, imm = 00000002,   |   PASSED
# 18. sw x9, 4(x2)        : T = 90, instr = 00912223, imm = 00000004,   |   PASSED
# 19. addi x26, x17, -5   : T = 95, instr = ffb88d13, imm = fffffffb,   |   PASSED
# 20. slti x26, x17, -5   : T = 100, instr = ffb8ad13, imm = fffffffb,   |   PASSED
# 21. sltiu x26, x17, 5   : T = 105, instr = 0058bd13, imm = 00000005,   |   PASSED
# 22. xori x5, x18, 9     : T = 110, instr = 00994293, imm = 00000009,   |   PASSED
# 23. ori x5, x12, 11     : T = 115, instr = 00b66293, imm = 0000000b,   |   PASSED
# 24. andi x5, x9, -3     : T = 120, instr = ffd4f293, imm = fffffffd,   |   PASSED
# 25. slli x5, x9, 6      : T = 125, instr = 00649293, imm = 00000006,   |   PASSED
# 26. srai x11, x10, 12   : T = 130, instr = 40c55593, imm = 0000000c,   |   PASSED
# 27. add x11, x10, x5    : T = 135, instr = 005505b3, imm = 00000000,   |   PASSED
```

2.4. ALU

The testbench verifies the ALU module by testing its ability to execute various operations, including arithmetic (ADD, SUB), comparison (SLT, SLTU), logical (XOR, OR, AND), shift (SLL, SRL, SRA), and LUI. For each operation, 5 (or 1000 if you want) randomized test cases are applied, with the output compared against expected results. The testbench generates a PASSED or FAILED message for each case and saves waveforms for debugging, ensuring the ALU functions accurately and robustly under diverse inputs.

```
VSIM 18> run -all
# PASSED add: alu_op = 0, operand_a = 12153524, operand_b = c0895e81, alu_data_expect = d29e93a5, data_test = d29e93a5
# PASSED add: alu_op = 0, operand_a = 8484d609, operand_b = b1f05663, alu_data_expect = 36752c6c, data_test = 36752c6c
# PASSED add: alu_op = 0, operand_a = 06b97b0d, operand_b = 46df998d, alu_data_expect = 4d99149a, data_test = 4d99149a
# PASSED add: alu_op = 0, operand_a = b2c28465, operand_b = 89375212, alu_data_expect = 3bf9d677, data_test = 3bf9d677
# PASSED add: alu_op = 0, operand_a = 00f3e301, operand_b = 06d7cd0d, alu_data_expect = 07cbb00e, data_test = 07cbb00e
# PASSED sub: alu_op = 8, operand_a = 3b23f176, operand_b = 1e8dcd3d, alu_data_expect = 1c962439, data_test = 1c962439
# PASSED sub: alu_op = 8, operand_a = 76d457ed, operand_b = 462df78c, alu_data_expect = 30a66061, data_test = 30a66061
# PASSED sub: alu_op = 8, operand_a = 7cfde9f9, operand_b = e33724c6, alu_data_expect = 99c6c533, data_test = 99c6c533
# PASSED sub: alu_op = 8, operand_a = e2f784c5, operand_b = d513d2aa, alu_data_expect = 0de3b21b, data_test = 0de3b21b
# PASSED sub: alu_op = 8, operand_a = 72aff7e5, operand_b = bbd27277, alu_data_expect = b6dd856e, data_test = b6dd856e
# PASSED slt: alu_op = 2, operand_a = 8932d612, operand_b = 47ecdb8f, alu_data_expect = 00000001, data_test = 00000001
# PASSED slt: alu_op = 2, operand_a = 793069f2, operand_b = e77696ce, alu_data_expect = 00000000, data_test = 00000000
# PASSED slt: alu_op = 2, operand_a = f4007ae8, operand_b = e2ca4ec5, alu_data_expect = 00000000, data_test = 00000000
# PASSED slt: alu_op = 2, operand_a = 2e58495c, operand_b = de8e28bd, alu_data_expect = 00000000, data_test = 00000000
# PASSED slt: alu_op = 2, operand_a = 96ab582d, operand_b = b2a72665, alu_data_expect = 00000001, data_test = 00000001
# PASSED sltu: alu_op = 3, operand_a = b1ef6263, operand_b = 0573870a, alu_data_expect = 00000000, data_test = 00000000
# PASSED sltu: alu_op = 3, operand_a = c03b2280, operand_b = 10642120, alu_data_expect = 00000000, data_test = 00000000
# PASSED sltu: alu_op = 3, operand_a = 557845aa, operand_b = cecccc9d, alu_data_expect = 00000001, data_test = 00000001
# PASSED sltu: alu_op = 3, operand_a = cb203e96, operand_b = 8983b813, alu_data_expect = 00000000, data_test = 00000000
# PASSED sltu: alu_op = 3, operand_a = 86bc380d, operand_b = a9a7d653, alu_data_expect = 00000001, data_test = 00000001
# PASSED xor: alu_op = 4, operand_a = 359fdd6b, operand_b = eaa62ad5, alu_data_expect = df39f7be, data_test = df39f7be
# PASSED xor: alu_op = 4, operand_a = 81174a02, operand_b = d7563eae, alu_data_expect = 564174ac, data_test = 564174ac
# PASSED xor: alu_op = 4, operand_a = 0effe91d, operand_b = e7c572cf, alu_data_expect = e93a9bd2, data_test = e93a9bd2
# PASSED xor: alu_op = 4, operand_a = 11844923, operand_b = 0509650a, alu_data_expect = 148d2c29, data_test = 148d2c29
# PASSED xor: alu_op = 4, operand_a = e5730aca, operand_b = 9e314c3c, alu_data_expect = 7b4246f6, data_test = 7b4246f6
# PASSED or: alu_op = 6, operand_a = 7968bdf2, operand_b = 452e618a, alu_data_expect = 7d6efdfa, data_test = 7d6efdfa
# PASSED or: alu_op = 6, operand_a = 20c4b341, operand_b = ec4b34d8, alu_data_expect = eccfb7d9, data_test = eccfb7d9
# PASSED or: alu_op = 6, operand_a = 3c20f378, operand_b = c48a1289, alu_data_expect = fcaaf3f9, data_test = fcaaf3f9
# PASSED or: alu_op = 6, operand_a = 75c50deb, operand_b = 5b0265b6, alu_data_expect = 7fc76dff, data_test = 7fc76dff
# PASSED or: alu_op = 6, operand_a = 634bf9c6, operand_b = 571513ae, alu_data_expect = 775ffbee, data_test = 775ffbee
# PASSED and: alu_op = 7, operand_a = de7502bc, operand_b = 150fdd2a, alu_data_expect = 14050028, data_test = 14050028
# PASSED and: alu_op = 7, operand_a = 85d79a0b, operand_b = b897be71, alu_data_expect = 80979a01, data_test = 80979a01
# PASSED and: alu_op = 7, operand_a = 42f24185, operand_b = 27f2554f, alu_data_expect = 02f24105, data_test = 02f24105
# PASSED and: alu_op = 7, operand_a = 9dcc603b, operand_b = 1d06333a, alu_data_expect = 1d04203a, data_test = 1d04203a
# PASSED and: alu_op = 7, operand_a = bf23327e, operand_b = 0aaa4b15, alu_data_expect = 0a220214, data_test = 0a220214
# PASSED sll: alu_op = 1, operand_a = 78d99bf1, operand_b = 6c9c4bd9, alu_data_expect = e2000000, data_test = e2000000
# PASSED sll: alu_op = 1, operand_a = 31230762, operand_b = 2635fb4c, alu_data_expect = 30762000, data_test = 30762000
# PASSED sll: alu_op = 1, operand_a = 4fa1559f, operand_b = 47b9a18f, alu_data_expect = aacf8000, data_test = aacf8000
# PASSED sll: alu_op = 1, operand_a = 7c6da9f8, operand_b = dbcd60b7, alu_data_expect = fc000000, data_test = fc000000
# PASSED sll: alu_op = 1, operand_a = cfc4569f, operand_b = ae7d945c, alu_data_expect = f0000000, data_test = f0000000
```

```
# PASSED srl: alu_op = 5, operand_a = adcbc05b, operand_b = 44de3789, alu_data_expect = 0056e5e0, data_test = 0056e5e0
# PASSED srl: alu_op = 5, operand_a = a4ae3249, operand_b = e8233ed0, alu_data_expect = 0000a4ae, data_test = 0000a4ae
# PASSED srl: alu_op = 5, operand_a = ebfec0d7, operand_b = a8c7fc51, alu_data_expect = 000075ff, data_test = 000075ff
# PASSED srl: alu_op = 5, operand_a = 4b212f96, operand_b = 061d7f0c, alu_data_expect = 0004b212, data_test = 0004b212
# PASSED srl: alu_op = 5, operand_a = e12ccec2, operand_b = 6457edc8, alu_data_expect = 00e12cce, data_test = 00e12cce
# PASSED sra: alu_op = d, operand_a = bb825a77, operand_b = 1ef2ed3d, alu_data_expect = fffffffd, data_test = fffffffd
# PASSED sra: alu_op = d, operand_a = 090cdb12, operand_b = bf05007e, alu_data_expect = 00000000, data_test = 00000000
# PASSED sra: alu_op = d, operand_a = 36e5816d, operand_b = 1cd9e739, alu_data_expect = 0000001b, data_test = 0000001b
# PASSED sra: alu_op = d, operand_a = 0fd28f1f, operand_b = e9ebf6d3, alu_data_expect = 000001fa, data_test = 000001fa
# PASSED sra: alu_op = d, operand_a = 42d92f85, operand_b = bc148878, alu_data_expect = 00000042, data_test = 00000042
# PASSED lui: alu_op = b, operand_a = 2dda595b, operand_b = 248b4b49, alu_data_expect = 248b4b49, data_test = 248b4b49
# PASSED lui: alu_op = b, operand_a = 9ff2ae3f, operand_b = 150caf2a, alu_data_expect = 150caf2a, data_test = 150caf2a
# PASSED lui: alu_op = b, operand_a = 2c156358, operand_b = c33f3886, alu_data_expect = c33f3886, data_test = c33f3886
# PASSED lui: alu_op = b, operand_a = c71a0c8e, operand_b = ce2ff29c, alu_data_expect = ce2ff29c, data_test = ce2ff29c
# PASSED lui: alu_op = b, operand_a = 7d3599fa, operand_b = 937dbc26, alu_data_expect = 937dbc26, data_test = 937dbc26
```

2.5. Branch Compare

The testbench module of the BRC will check the comparison operation and output the results to two output signals. Here, I created a task to test the outputs. To increase accuracy, the input signals will be generated randomly, including the *br_un* signal. 500 test cases of the input signals will be performed.

```
# PASSED equal: a = b897be71, b = 42f24185, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = 9dcc603b, b = 1d06333a, br_un = 0, less_expected = 0, less_test = 0
# PASSED equal: a = 9dcc603b, b = 1d06333a, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = 0aaa4b15, b = 78d99bf1, br_un = 1, less_expected = 1, less_test = 1
# PASSED equal: a = 0aaa4b15, b = 78d99bf1, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = 31230762, b = 2635fb4c, br_un = 1, less_expected = 0, less_test = 0
# PASSED equal: a = 31230762, b = 2635fb4c, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = 47b9a18f, b = 7c6da9f8, br_un = 1, less_expected = 1, less_test = 1
# PASSED equal: a = 47b9a18f, b = 7c6da9f8, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = cfc4569f, b = ae7d945c, br_un = 1, less_expected = 0, less_test = 0
# PASSED equal: a = cfc4569f, b = ae7d945c, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = 44de3789, b = a4ae3249, br_un = 0, less_expected = 1, less_test = 1
# PASSED equal: a = 44de3789, b = a4ae3249, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = ebfec0d7, b = a8c7fc51, br_un = 0, less_expected = 0, less_test = 0
# PASSED equal: a = ebfec0d7, b = a8c7fc51, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = 061d7f0c, b = e12ccec2, br_un = 0, less_expected = 1, less_test = 1
# PASSED equal: a = 061d7f0c, b = e12ccec2, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = bb825a77, b = 1ef2ed3d, br_un = 0, less_expected = 0, less_test = 0
# PASSED equal: a = bb825a77, b = 1ef2ed3d, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = bf05007e, b = 36e5816d, br_un = 1, less_expected = 1, less_test = 1
# PASSED equal: a = bf05007e, b = 36e5816d, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = 0fd28f1f, b = e9ebf6d3, br_un = 1, less_expected = 0, less_test = 0
# PASSED equal: a = 0fd28f1f, b = e9ebf6d3, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = bc148878, b = 2dda595b, br_un = 1, less_expected = 1, less_test = 1
# PASSED equal: a = bc148878, b = 2dda595b, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = 9ff2ae3f, b = 150caf2a, br_un = 0, less_expected = 0, less_test = 0
# PASSED equal: a = 9ff2ae3f, b = 150caf2a, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = c33f3886, b = c71a0c8e, br_un = 0, less_expected = 1, less_test = 1
# PASSED equal: a = c33f3886, b = c71a0c8e, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = 7d3599fa, b = 937dbc26, br_un = 1, less_expected = 0, less_test = 0
# PASSED equal: a = 7d3599fa, b = 937dbc26, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = d18bb4a3, b = 9799a82f, br_un = 1, less_expected = 0, less_test = 0
# PASSED equal: a = d18bb4a3, b = 9799a82f, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = afd8565f, b = 22290d44, br_un = 1, less_expected = 1, less_test = 1
# PASSED equal: a = afd8565f, b = 22290d44, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = e59b36cb, b = f3091ae6, br_un = 0, less_expected = 1, less_test = 1
# PASSED equal: a = e59b36cb, b = f3091ae6, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = 14cfc129, b = f682e2ed, br_un = 0, less_expected = 1, less_test = 1
# PASSED equal: a = 14cfc129, b = f682e2ed, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = b29fb665, b = da8ae2b5, br_un = 1, less_expected = 1, less_test = 1
# PASSED equal: a = b29fb665, b = da8ae2b5, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = 3cf11979, b = 2231ff44, br_un = 0, less_expected = 0, less_test = 0
# PASSED equal: a = 3cf11979, b = 2231ff44, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = 15090b2a, b = 55f6adab, br_un = 0, less_expected = 1, less_test = 1
# PASSED equal: a = 15090b2a, b = 55f6adab, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = 6e5daddc, b = cd5ebc9a, br_un = 1, less_expected = 0, less_test = 0
# PASSED equal: a = 6e5daddc, b = cd5ebc9a, br_un = 1, equal_expected = 0, equal_test = 0
# PASSED less: a = e1f102c3, b = 2b0eed56, br_un = 0, less_expected = 0, less_test = 0
# PASSED equal: a = e1f102c3, b = 2b0eed56, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = b3d97667, b = 8531340a, br_un = 0, less_expected = 0, less_test = 0
# PASSED equal: a = b3d97667, b = 8531340a, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = 9c0e8a38, b = 3cd18779, br_un = 0, less_expected = 0, less_test = 0
# PASSED equal: a = 9c0e8a38, b = 3cd18779, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = 4a74bf94, b = 49c65d93, br_un = 0, less_expected = 0, less_test = 0
# PASSED equal: a = 4a74bf94, b = 49c65d93, br_un = 0, equal_expected = 0, equal_test = 0
# PASSED less: a = acb7ca59, b = 6dcb69db, br_un = 1, less_expected = 1, less_test = 1
# PASSED equal: a = acb7ca59, b = 6dcb69db, br_un = 1, equal_expected = 0, equal_test = 0
# Test BRC sucessful
```

2.6. Load-Store Unit (LSU)

The testbench validates the Load-Store Unit (LSU) by simulating read and write operations to various memory-mapped peripherals and data memory (DMEM). It performs multiple iterations of writing randomized data to specific addresses, including those mapped to HEX displays, LEDs, LCDs, and general DMEM. For each write operation, the testbench verifies the correctness of the data by comparing the read-back values with the written values. Peripheral-specific checks ensure proper data alignment and functionality, with results logged as PASSED or FAILED for debugging. The simulation ensures the LSU operates correctly under various scenarios and addresses.

```
# 7020: Store c0095e01
# 7020: Read data = 40095e01 --> PASSED
# HEX0,1,2,3 = 40095e01
# 7020: Store 8484d609
# 7020: Read data = 04045609 --> PASSED
# HEX0,1,2,3 = 04045609
# 7020: Store b1f05663
# 7020: Read data = 31705663 --> PASSED
# HEX0,1,2,3 = 31705663
# 7020: Store 06b97b0d
# 7020: Read data = 06397b0d --> PASSED
# HEX0,1,2,3 = 06397b0d
# 7020: Store 46df998d
# 7020: Read data = 465f190d --> PASSED
# HEX0,1,2,3 = 465f190d
# 7020: Store b2c28465
# 7020: Read data = 32420465 --> PASSED
# HEX0,1,2,3 = 32420465
# 7020: Store 89375212
# 7020: Read data = 09375212 --> PASSED
# HEX0,1,2,3 = 09375212
# 7020: Store 00f3e301
# 7020: Read data = 00736301 --> PASSED
# HEX0,1,2,3 = 00736301
# 7020: Store 06d7cd0d
# 7020: Read data = 06574d0d --> PASSED
# HEX0,1,2,3 = 06574d0d
# 7020: Store 3b23f176
# 7020: Read data = 3b237176 --> PASSED
# HEX0,1,2,3 = 3b237176
# 7020: Store 1e8dcd3d
# 7020: Read data = 1e0d4d3d --> PASSED
```

```
# HEX7,6,5,4 = 79683d72
# 7024: Store 452e618a
# 7024: Read data = 452e610a --> PASSED
# HEX7,6,5,4 = 452e610a
# 7024: Store 20c4b341
# 7024: Read data = 20443341 --> PASSED
# HEX7,6,5,4 = 20443341
# 7024: Store ec4b34d8
# 7024: Read data = 6c4b3458 --> PASSED
# HEX7,6,5,4 = 6c4b3458
# 7024: Store 3c20f378
# 7024: Read data = 3c207378 --> PASSED
# HEX7,6,5,4 = 3c207378
# 7024: Store c48a1289
# 7024: Read data = 440a1209 --> PASSED
# HEX7,6,5,4 = 440a1209
# 7024: Store 75c50deb
# 7024: Read data = 75450d6b --> PASSED
# HEX7,6,5,4 = 75450d6b
# 7024: Store 5b0265b6
# 7024: Read data = 5b026536 --> PASSED
# HEX7,6,5,4 = 5b026536
# 7024: Store 634bf9c6
# 7024: Read data = 634b7946 --> PASSED
# HEX7,6,5,4 = 634b7946
# 7024: Store 571513ae
# 7024: Read data = 5715132e --> PASSED
# HEX7,6,5,4 = 5715132e
# 7024: Store de7502bc
# 7024: Read data = 5e75023c --> PASSED
# HEX7,6,5,4 = 5e75023c
# 7024: Store 150fdd2a
```

```
# LEDG = ce2ff29c
# 7010: Store 7d3599fa
# 7010: Read data = 7d3599fa --> PASSED
# LEDG = 7d3599fa
# 7010: Store 937dbc26
# 7010: Read data = 937dbc26 --> PASSED
# LEDG = 937dbc26
# 7010: Store 39961773
# 7010: Read data = 39961773 --> PASSED
# LEDG = 39961773
# 7010: Store d18bb4a3
# 7010: Read data = d18bb4a3 --> PASSED
# LEDG = d18bb4a3
# 7010: Store 9799a82f
# 7010: Read data = 9799a82f --> PASSED
# LEDG = 9799a82f
# 7010: Store d9d292b3
# 7010: Read data = d9d292b3 --> PASSED
# LEDG = d9d292b3
# 7010: Store afd8565f
# 7010: Read data = afd8565f --> PASSED
# LEDG = afd8565f
# 7010: Store 22290d44
# 7010: Read data = 22290d44 --> PASSED
# LEDG = 22290d44
# 7010: Store 7bf8fdf7
# 7010: Read data = 7bf8fdf7 --> PASSED
# LEDG = 7bf8fdf7
# 7010: Store e59b36cb
# 7010: Read data = e59b36cb --> PASSED
# LEDG = e59b36cb
# 7010: Store f3091ae6
# 7010: Read data = f3091ae6 --> PASSED
# LEDG = f3091ae6
# 7010: Store 2d28db5a
# 7010: Read data = 2d28db5a --> PASSED
# LEDG = 2d28db5a
# 7010: Store 14cfc129
```

```
# 7000: Read data = 43356786 --> PASSED
# LEDR = 43356786
# 7000: Store ed3408da
# 7000: Read data = ed3408da --> PASSED
# LEDR = ed3408da
# 7000: Store 9eb7c63d
# 7000: Read data = 9eb7c63d --> PASSED
# LEDR = 9eb7c63d
# 7000: Store 334ea766
# 7000: Read data = 334ea766 --> PASSED
# LEDR = 334ea766
# 7000: Store b855c470
# 7000: Read data = b855c470 --> PASSED
# LEDR = b855c470
# 7000: Store b9f50473
# 7000: Read data = b9f50473 --> PASSED
# LEDR = b9f50473
# 7000: Store 5d7199ba
# 7000: Read data = 5d7199ba --> PASSED
# LEDR = 5d7199ba
# 7000: Store 2f3ab35e
# 7000: Read data = 2f3ab35e --> PASSED
# LEDR = 2f3ab35e
# 7000: Store 7d4779fa
# 7000: Read data = 7d4779fa --> PASSED
# LEDR = 7d4779fa
# 7000: Store 6a8e05d5
# 7000: Read data = 6a8e05d5 --> PASSED
# LEDR = 6a8e05d5
```

```
# 7030: Read data = 25b27b4b --> PASSED
# LCD = 25b27b4b
# 7030: Store b98c4273
# 7030: Read data = b98c4273 --> PASSED
# LCD = b98c4273
# 7030: Store f622e6ec
# 7030: Read data = f622e6ec --> PASSED
# LCD = f622e6ec
# 7030: Store c550168a
# 7030: Read data = c550168a --> PASSED
# LCD = c550168a
# 7030: Store 2758d14e
# 7030: Read data = 2758d14e --> PASSED
# LCD = 2758d14e
# 7030: Store d44b80a8
# 7030: Read data = d44b80a8 --> PASSED
# LCD = d44b80a8
# 7030: Store 549efda9
# 7030: Read data = 549efda9 --> PASSED
# LCD = 549efda9
# 7030: Store d0ca8ca1
# 7030: Read data = d0ca8ca1 --> PASSED
# LCD = d0ca8ca1
# 7030: Store 070bb90e
# 7030: Read data = 070bb90e --> PASSED
# LCD = 070bb90e
# 7030: Store f33466e6
# 7030: Read data = f33466e6 --> PASSED
# LCD = f33466e6
# 7030: Store cfd6c09f
# 7030: Read data = cfd6c09f --> PASSED
# LCD = cfd6c09f
# 7030: Store 152fb52a
# 7030: Read data = 152fb52a --> PASSED
# LCD = 152fb52a
# 7030: Store 155a1d2a
# 7030: Read data = 155a1d2a --> PASSED
# LCD = 155a1d2a
# 7030: Store c6b5f48d
# 7030: Read data = c6b5f48d --> PASSED
# LCD = c6b5f48d
# 7030: Store 4f75ff9e
# 7030: Read data = 4f75ff9e --> PASSED
# LCD = 4f75ff9e
# 7030: Store 9c6de638
# 7030: Read data = 9c6de638 --> PASSED
# LCD = 9c6de638
```

```
# 0064: Read data = adac225b --> PASSED
# DMEM[00000064] = adac225b
# 0068: Store f166fae2
# 0068: Read data = f166fae2 --> PASSED
# DMEM[00000068] = f166fae2
# 006c: Store 8273e204
# 006c: Read data = 8273e204 --> PASSED
# DMEM[0000006c] = 8273e204
# 0070: Store 39ac0373
# 0070: Read data = 39ac0373 --> PASSED
# DMEM[00000070] = 39ac0373
# 0074: Store ec50b4d8
# 0074: Read data = ec50b4d8 --> PASSED
# DMEM[00000074] = ec50b4d8
# 0078: Store 093e4d12
# 0078: Read data = 093e4d12 --> PASSED
# DMEM[00000078] = 093e4d12
# 007c: Store dc0344b8
# 007c: Read data = dc0344b8 --> PASSED
# DMEM[0000007c] = dc0344b8
# 0080: Store 9c811239
# 0080: Read data = 9c811239 --> PASSED
# DMEM[00000080] = 9c811239
# 0084: Store f287b6e5
# 0084: Read data = f287b6e5 --> PASSED
# DMEM[00000084] = f287b6e5
# 0088: Store d0c5dca1
# 0088: Read data = d0c5dca1 --> PASSED
# DMEM[00000088] = d0c5dca1
# 008c: Store 15890f2b
# 008c: Read data = 15890f2b --> PASSED
# DMEM[0000008c] = 15890f2b
# 0090: Store 40905d81
# 0090: Read data = 40905d81 --> PASSED
# DMEM[00000090] = 40905d81
# 0094: Store 641b85c8
# 0094: Read data = 641b85c8 --> PASSED
# DMEM[00000094] = 641b85c8
```

```
# 07c8: Read data = 892fc012 --> PASSED
# DMEM[000007c8] = 892fc012
# 07cc: Store 0650df0c
# 07cc: Read data = 0650df0c --> PASSED
# DMEM[000007cc] = 0650df0c
# 07d0: Store 06db6b0d
# 07d0: Read data = 06db6b0d --> PASSED
# DMEM[000007d0] = 06db6b0d
# 07d4: Store 0acd1315
# 07d4: Read data = 0acd1315 --> PASSED
# DMEM[000007d4] = 0acd1315
# 07d8: Store 20310740
# 07d8: Read data = 20310740 --> PASSED
# DMEM[000007d8] = 20310740
# 07dc: Store 4a638d94
# 07dc: Read data = 4a638d94 --> PASSED
# DMEM[000007dc] = 4a638d94
# 07e0: Store 2a1ba354
# 07e0: Read data = 2a1ba354 --> PASSED
# DMEM[000007e0] = 2a1ba354
# 07e4: Store c0620280
# 07e4: Read data = c0620280 --> PASSED
# DMEM[000007e4] = c0620280
# 07e8: Store 098d1513
# 07e8: Read data = 098d1513 --> PASSED
# DMEM[000007e8] = 098d1513
# 07ec: Store e1e386c3
# 07ec: Read data = e1e386c3 --> PASSED
# DMEM[000007ec] = e1e386c3
# 07f0: Store f695deed
# 07f0: Read data = f695deed --> PASSED
# DMEM[000007f0] = f695deed
# 07f4: Store ee4ee6dc
# 07f4: Read data = ee4ee6dc --> PASSED
# DMEM[000007f4] = ee4ee6dc
# 07f8: Store bc781078
# 07f8: Read data = bc781078 --> PASSED
# DMEM[000007f8] = bc781078
# 07fc: Store 13d40d27
# 07fc: Read data = 13d40d27 --> PASSED
# DMEM[000007fc] = 13d40d27
```

2.7. Control Unit

This testbench tests the Control Unit of a RISC-V processor by simulating various instructions and checking the generated control signals. It validates instructions like LUI, JAL, BEQ, LW, SW, and arithmetic operations (ADD, SUB, etc.). The testbench applies each instruction and verifies outputs such as branch selection, ALU operation, write-back control, and memory access.

The test ensures the Control Unit correctly decodes instructions and sets control signals like br_sel, rd_wren, alu_op, and mem_wren. Each test case compares the actual outputs with expected values, logging results as PASSED or FAILED. This confirms the Control Unit's ability to manage instruction flow and processor operations accurately.

```
VSIM 24> run -all
# LUI x10, 7
# PASSED
# AUIPC x10, 7
# PASSED
# JAL x5, 10
# PASSED
# JALR x5, 0(x2)
# PASSED
# BEQ x11, x10, 8
# PASSED
# PASSED
# BNE x11, x10, 12
# PASSED
# PASSED
# BLT x15, x16, 8
# PASSED
# PASSED
# BGE x7, x12, 4
# PASSED
# PASSED
# PASSED
# BLTU x3, x5, 8
# PASSED
# PASSED
# BGEU x15, x16, 16
# PASSED
# PASSED
# PASSED
# LW x20, 0(x6)
# PASSED
# SW x9, 4(x2)
# PASSED
# ADDI x26, x17, -5
# PASSED
# SLTI x26, x17, -5
# PASSED
# SLTIU x26, x17, -5
# PASSED
# XORI x5, x18, 9
# PASSED
# ORI x5, x12, 11
# PASSED
# ANDI x5, x9, -3
# PASSED
# SLLI x5, x9, 6
# PASSED
# SRLI x5, x10, 3
# PASSED
# SRAI x11, x10, 12
# PASSED
# ADD x11, x10, x5
# PASSED
# SUB x11, x10, x5
# PASSED
# SLT x5, x6, x7
# PASSED
# SLTU x5, x6, x7
# PASSED
# XOR x5, x6, x7
# PASSED
# OR x5, x6, x7
# PASSED
# AND x5, x6, x7
# PASSED
# SLL x5, x6, x7
# PASSED
# SRL x5, x6, x7
# PASSED
# SRA x5, x6, x7
# PASSED
```

2.8. Grand Test

This test environement is provided by Mr. Cao Xuan Hai.

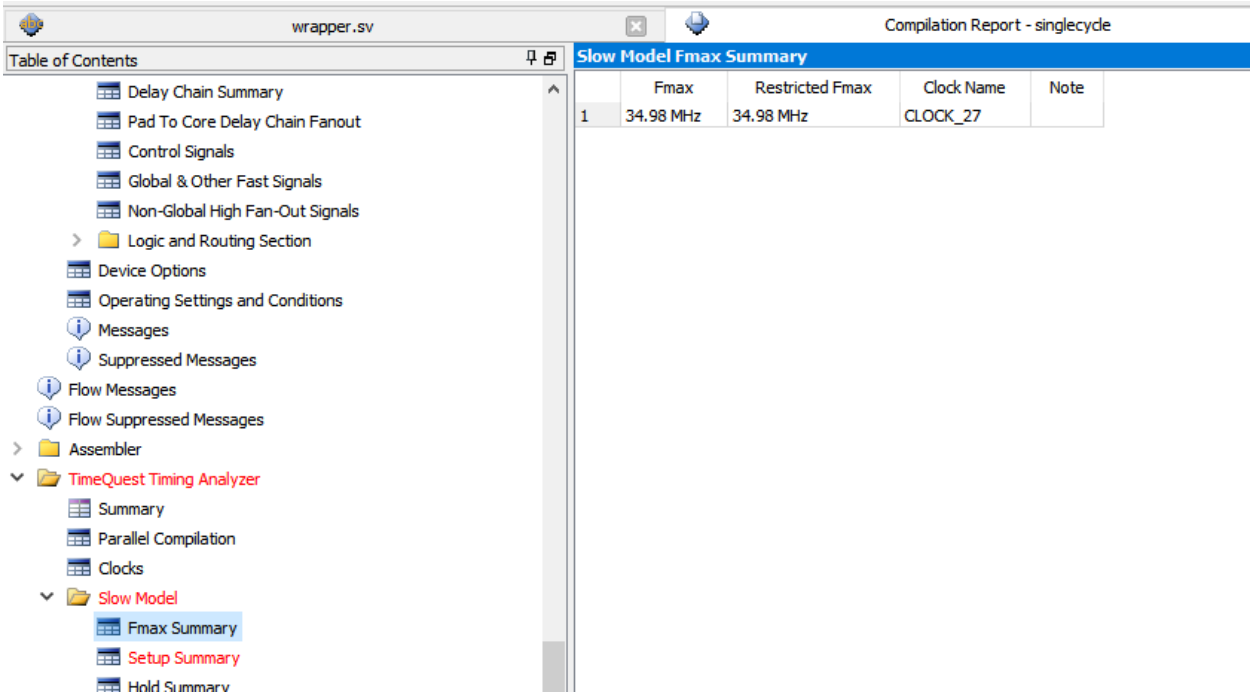
```
TEST for SINGLECYCLE
[     1800]:: 1::ADD..............PASSED
[    12600]:: 2::SUB..............PASSED
[    24200]:: 3::XOR..............PASSED
[    34800]:: 4::OR...............PASSED
[    45600]:: 5::AND..............PASSED
[    56400]:: 6::SLL..............PASSED
[    65600]:: 7::SRL..............PASSED
[    76200]:: 8::SRA..............PASSED
[    86400]:: 9::SLT..............PASSED
[    95800]::10::SLTU.............PASSED
[   105200]::11::ADDI.............PASSED
[   111600]::12::XORI.............PASSED
[   118000]::13::ORI..............PASSED
[   124200]::14::ANDI.............PASSED
[   130000]::15::SLLI.............PASSED
[   136400]::16::SRLI.............PASSED
[   143600]::17::SRAI.............PASSED
[   151000]::18::SLTI.............PASSED
[   156200]::19::SLTIU............PASSED
[   161400]::20::LUI..............PASSED
[   166000]::21::AUIPC............PASSED
[   169000]::22::LW...............PASSED
[   177200]::23::LH...............PASSED
[   182800]::24::LB...............PASSED
[   187600]::25::LHU..............PASSED
[   193400]::26::LBU..............PASSED
[   198200]::27::SW...............PASSED
[   202600]::28::SH...............PASSED
[   209000]::29::SB...............PASSED
[   215000]::30::misaligned........FAILED
[   218400]::31::BEQ..............PASSED
[   220600]::32::BNE..............PASSED
[   222800]::33::BLT..............PASSED
[   226800]::34::BGE..............PASSED
[   230800]::35::BLTU.............PASSED
[   234800]::36::BGEU.............PASSED
[   238800]::37::JAL..............PASSED
[   241800]::38::JALR.............PASSED
[   244800]::39::illegal_insn......FAILED

Timeout...

OUT is considered       P A S S E D
```

3. SYNTHESIS (QUARTUS)

Flow Status	Successful - Mon Nov 18 00:40:49 2024
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	singlecycle
Top-level Entity Name	wrapper
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	26,607 / 33,216 (80 %)
Total combinational functions	15,646 / 33,216 (47 %)
Dedicated logic registers	17,560 / 33,216 (53 %)
Total registers	17560
Total pins	117 / 475 (25 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

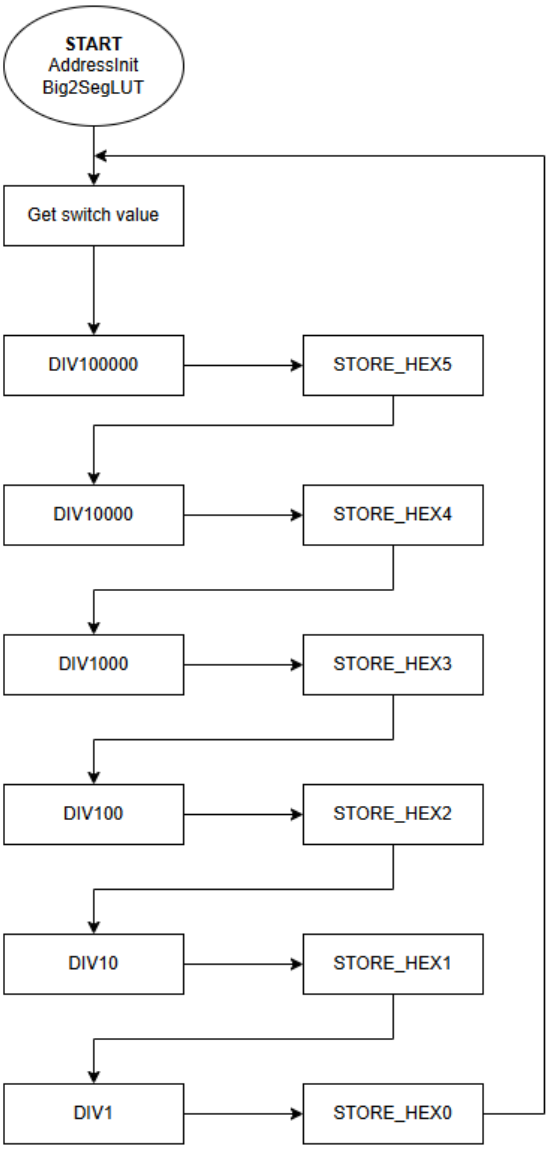


4. FPGA IMPLEMENTATION

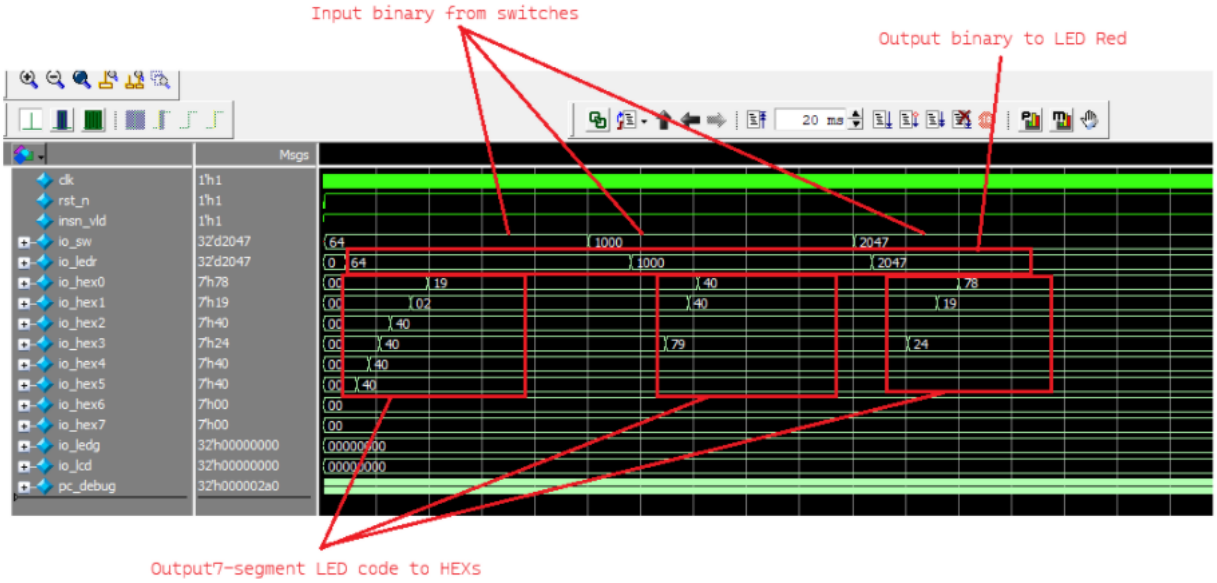
We first develop assembly risc-v programs to run on venus, we dubug this program both on venus/cornell website/ testbench before load it to the FPGA.

4.1. Switch to 7-segments

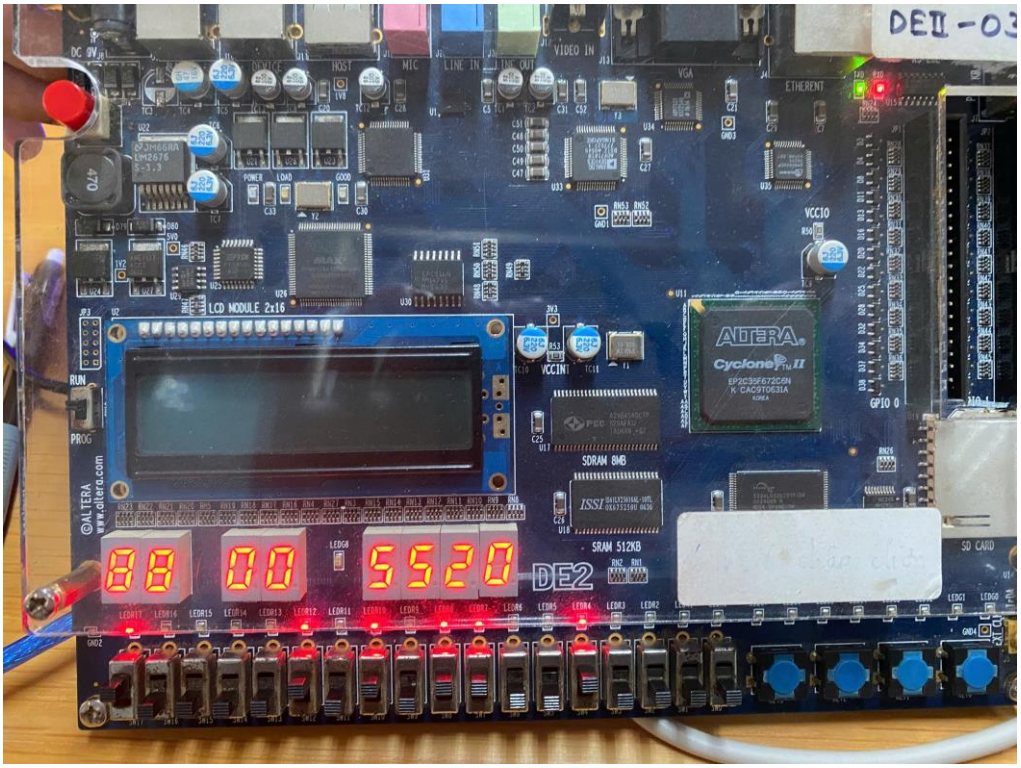
The application converts the binary value from switches (SW) into decimal format and displays the result on the 7-segment LEDs (HEX0 to HEX7). The binary input is also shown on the red LEDs (LEDR). The program uses a lookup table (LUT) to translate binary numbers into 7-segment display format and performs division operations to extract each digit for display.

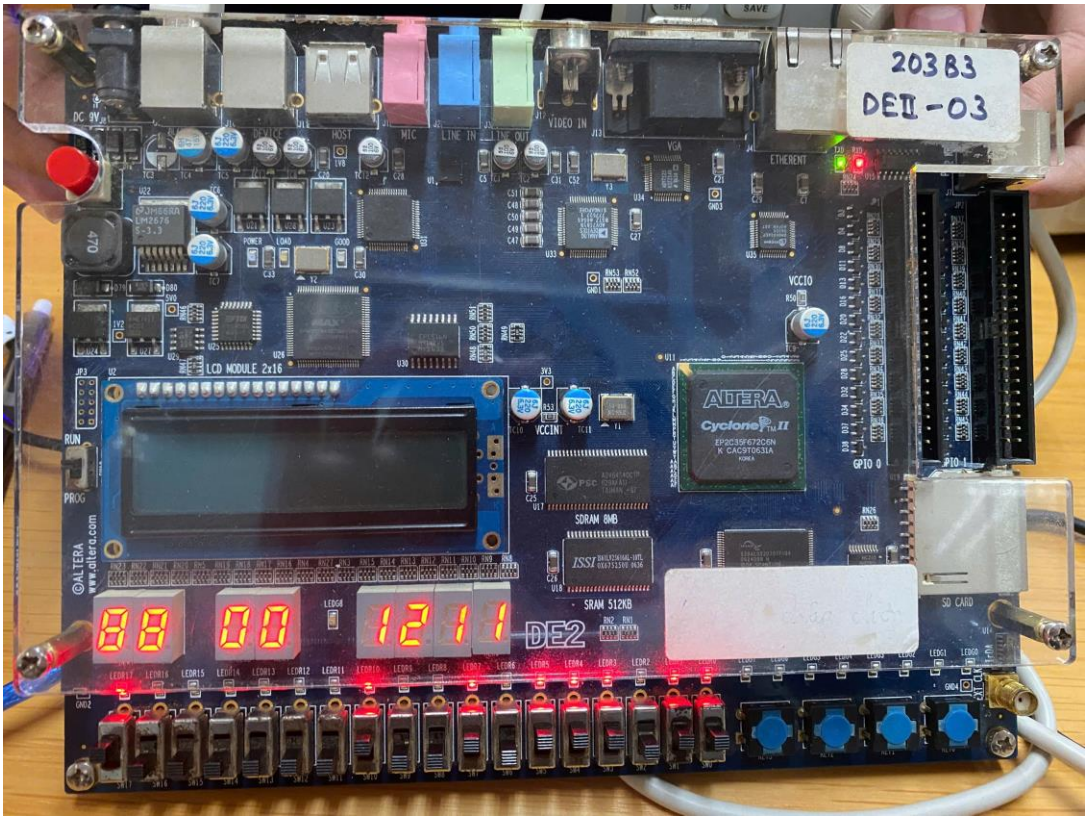


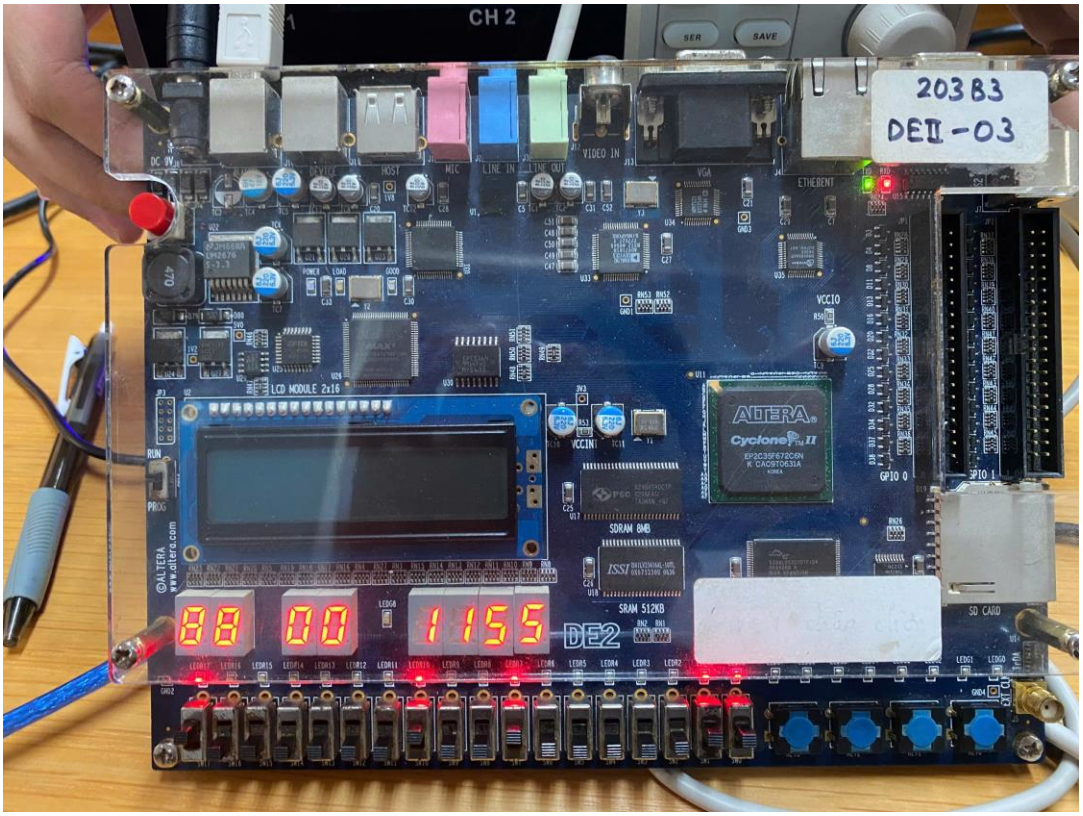
Simulation on Modelsim



FPGA Implementation



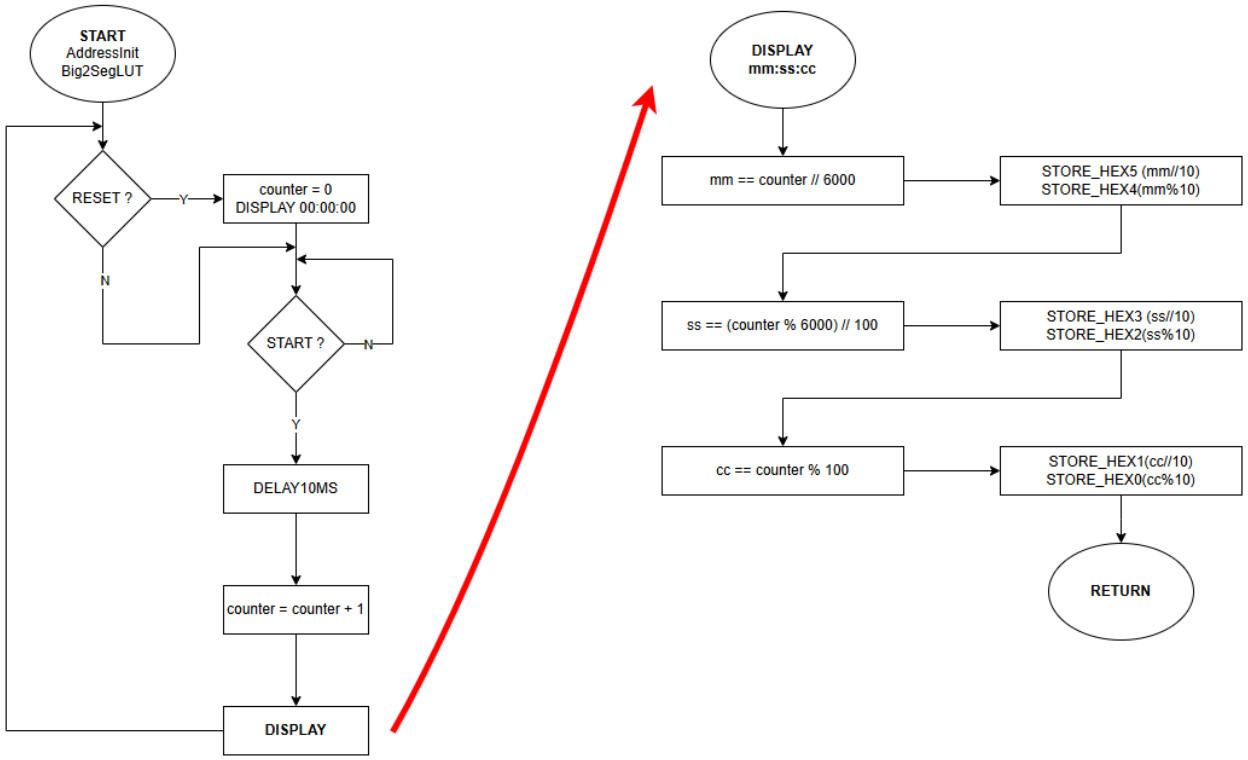




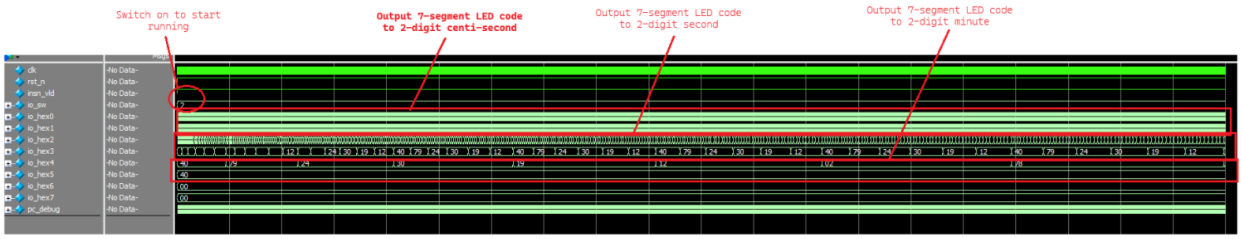
⇨ Don’t care the last two segment HEX7HEX6. I forgot to convert the 00 to 00 segment code so it’s dispay 88 (since the code is in 00 decimal)

4.2. Stopwatch

This application implements a stopwatch that displays the elapsed time in the format "MM:SS " (minutes, seconds, and centiseconds) on a 7-segment display. The input consists of control signals from switches (SW), including RESET and START. The output is the real-time update of the 7-segment display showing the elapsed time. The application calculates time divisions (minutes, seconds, centiseconds) using a counter and modular arithmetic, stores the converted values into respective registers, and updates the display. It also includes a delay mechanism for centisecond timing and resets to "00:00:00" upon receiving the RESET signal.

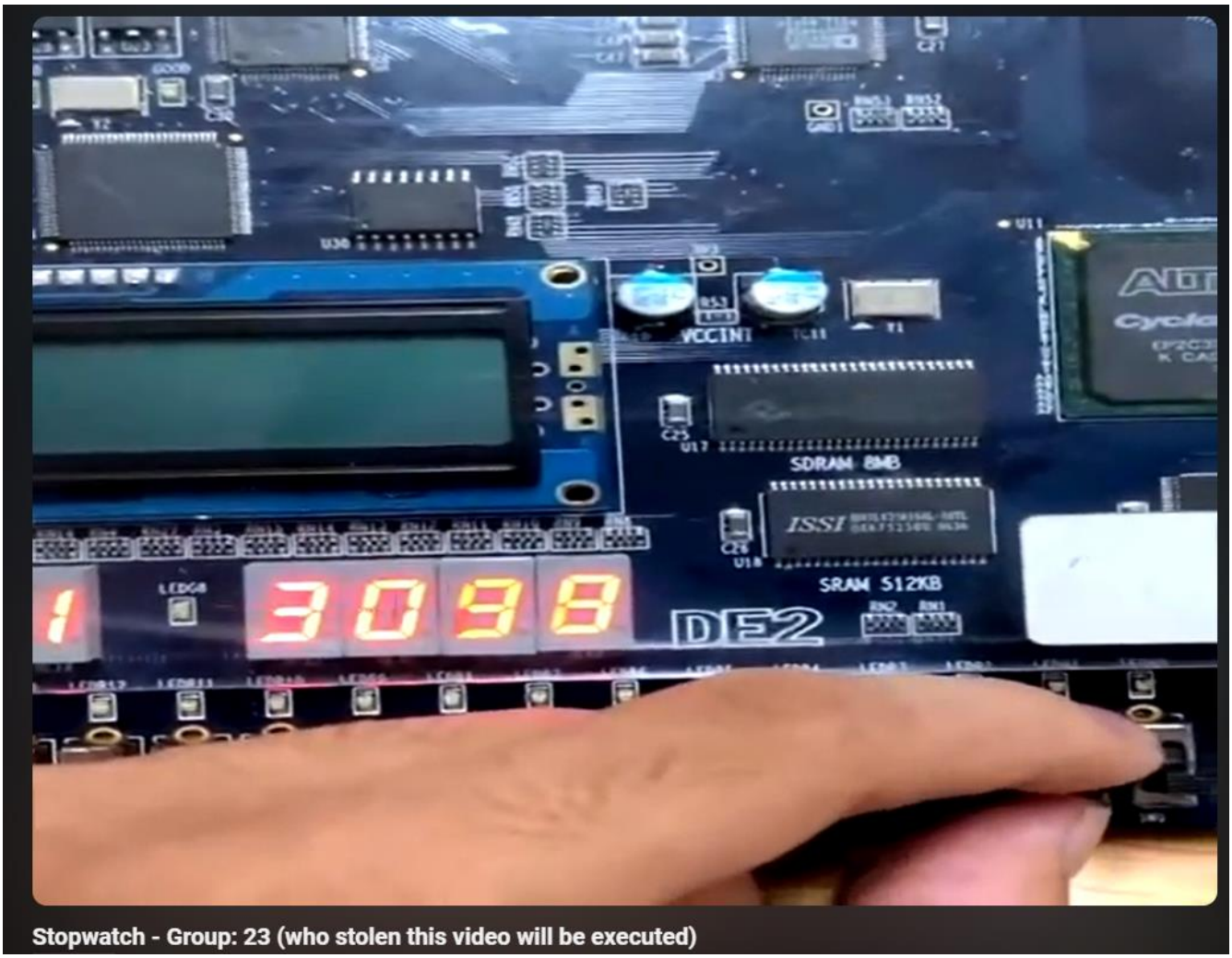


Simulation on Modelsim



FPGA Implementation

[Stopwatch - Group: 23 (who stolen this video will be executed)]

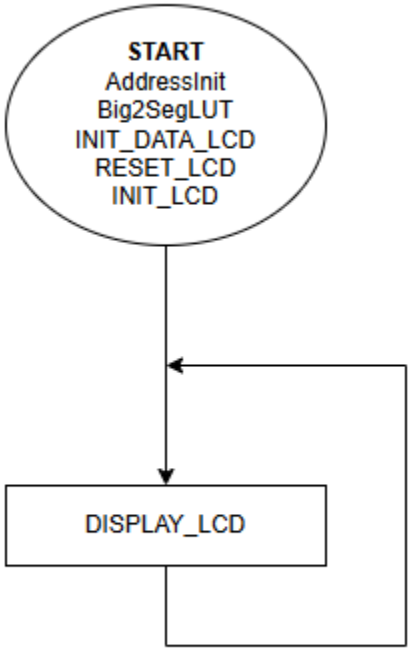


Stopwatch - Group: 23 (who stolen this video will be executed)

4.3. Display “Hello World – namd” on LCD

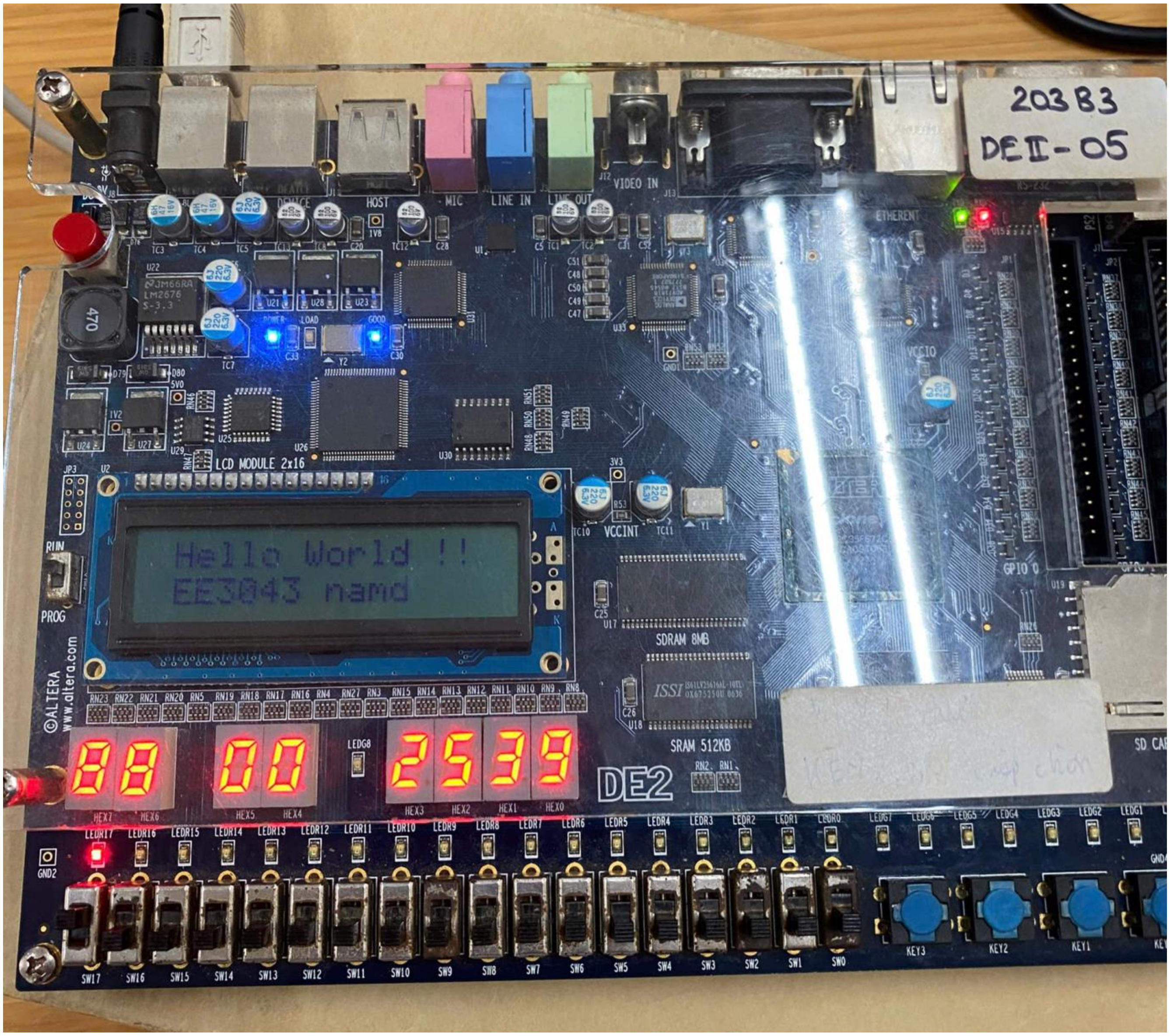
This application is a program for interfacing with an LCD to display predefined text. The input consists of initialization commands and predefined data stored in memory to be written to the LCD. The program initializes the LCD, loads character data from a predefined memory location, and displays it line-by-line on the LCD in two 16-character rows.

It includes functions for sending commands (CMDWRITE) and data (DATAWRITE) to the LCD and initializes the data with a custom message. The output is the display of the predefined text on the LCD.



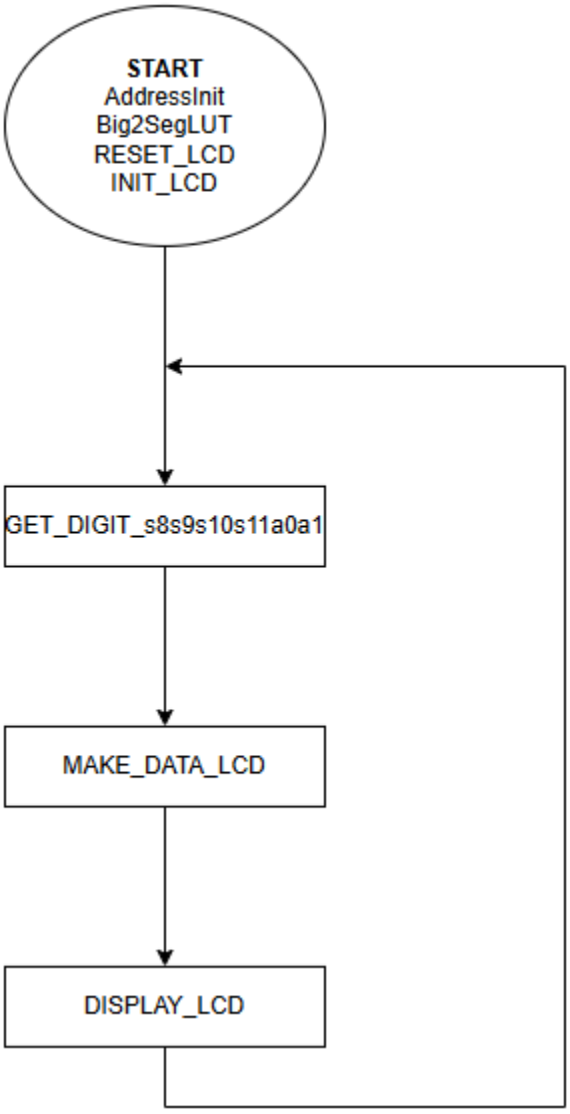
200	204	208	20C	210	214	218	21C	220	224	228	22C	230	234	238	23C
	H	e	l	l	o		W	o	r	l	d		!	!	
240	**244**	**248**	**24C**	**250**	**254**	**258**	**25C**	**260**	**264**	**268**	**26C**	**270**	**274**	**278**	**27C**
	E	E	3	0	4	3		n	a	m	d				

FPGA Implementation


Hello World !!
EE3043 namd
88 00 2539
DE2
LCD MODULE 2x16
SDRAM 8MB
SRAM 512KB
203B3
DEII-05

4.4. Display the number from Switch to LCD

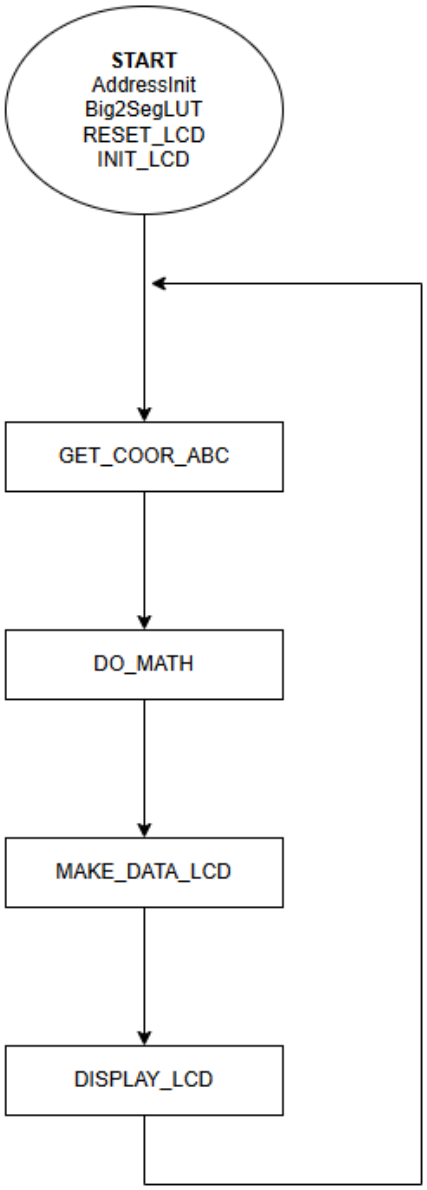
This application reads a number from switch inputs, processes the data, and displays it on both a seven-segment display and an LCD screen. The input is the binary number from switches, which is first converted into decimal digits using division algorithms. These digits are displayed on the seven-segment displays. The program then prepares the data for the LCD in a formatted "Decimal" label followed by the number and a "Binary" label for additional data. The final output shows the processed number on both the seven-segment displays and the LCD.



200	204	208	20C	210	214	218	21C	220	224	228	22C	230	234	238	23C
D	e	c	i	m	a	l	:	HEX5	HEX4	HEX3	HEX2	HEX1	HEX0		
240	244	248	24C	250	254	258	25C	260	264	268	26C	270	274	278	27C
B	i	n	a	r	y	:		<	1	7					

4.5. A simple mathematic program to calculate A or B is closer to C

This program calculates whether point A or B is closer to point C in a 2D coordinate space. The input consists of the coordinates of points A, B, and C, which are read from switches and displayed on LEDs. The program calculates the squared distances from A and B to C using mathematical operations. Based on the results, it determines which point is closer and outputs the result as 'A' or 'B'. The coordinates and the closest point are then displayed on an LCD screen, providing both the numerical and visual representation of the results.



200	204	208	20C	210	214	218	21C	220	224	228	22C	230	234	238	23C
A	(	s8	,	s9	)	B	(	s10	,	s11	)	C	(	a0	,
240	**244**	**248**	**24C**	**250**	**254**	**258**	**25C**	**260**	**264**	**268**	**26C**	**270**	**274**	**278**	**27C**
a1	)		>		a2		i	s		c	l	o	s	e	r

5. EVALUATION

Tasks		Evaluation
RTL	Register File	✓
	Immediate Generator	✓
	Branch Comparator	✓
	Arithmetic Logic Unit	✓
	IMEM, DMEM	✓
	Load Store Unit	✓
	Control Unit	✓
	Top Single Cycle	✓
Verification	Immediate Generator	✓
	Branch Comparator	✓
	Arithmetic Logic Unit	✓
	IMEM, DMEM	✓
	Load Store Unit	✓
	Control Unit	✓
	Top Single Cycle	✓
Application	Switch to Segments	✓
	Stop Watch	✓
	Hello World on LCD	✓
	Calculate number from switch and display to LCD	Develop assmebly: FINISHED Demo on FPGA: FAILED
	Coordinate LCD	