EE3043 Computer Architecture

Semester 241

Design of a Single Cycle RISC-V Processor

T.A: CAO XUAN HẢI

Student: Nguyễn Thanh Tùng – 2213874

Lê Tấn Dũng – 2210575

Lưu Lê Anh Khôi – 2211683

1. Introduction

This project focuses on designing a single-cycle RV32I RISC-V processor. The design includes essential components like the Arithmetic Logic Unit (ALU) and Load-Store Unit (LSU), with integrated memory-mapped I/O. Following RV32I specifications, the processor will be verified through testbenches to ensure accurate functionality and performance.

2. Design Strategy

2.1. Overview

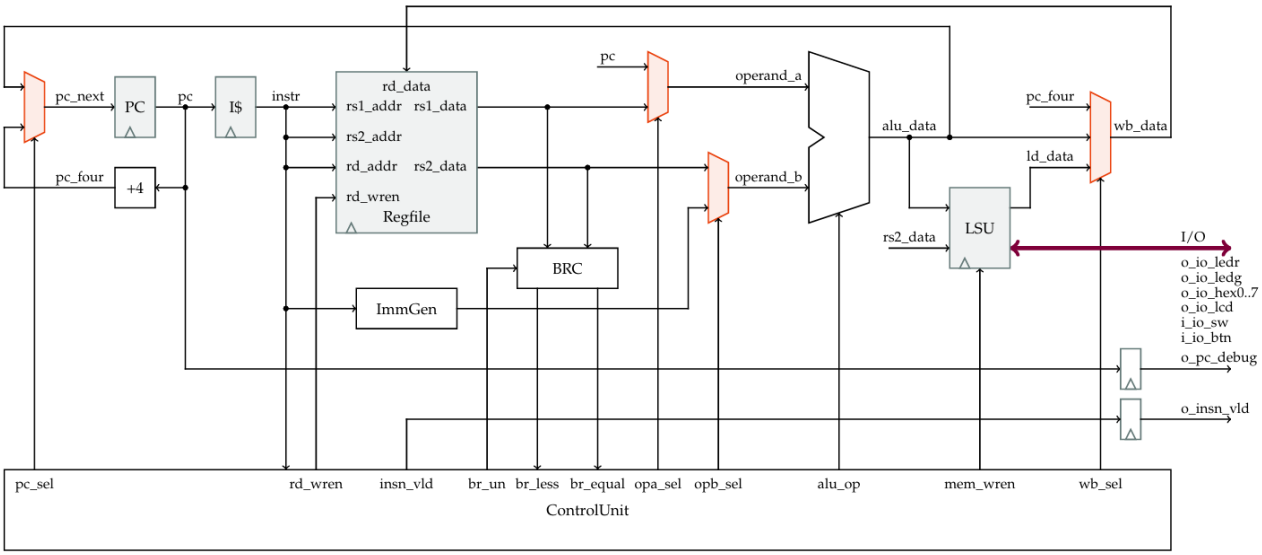

Figure 1: Single cycle overview

2.2. ALU

The ALU (arithmatic logic unit) is responsible for executing various opera tions such as addition, subtraction, bitwise XOR, bitwise AND, bitwise OR, set less than, set less than unsigned, shift left logical, shift right logical, and shift right arithmetic.

In this implementation, the ALU is typically 32 bits data width. It takes input value from processor's registers (from register file, immediate generation or program counter). The output value is selected base on alu operation, then store back to a register in register file. The ALU executes instructions of different types, including R-type and I-type instructions:

R-type: the input value always take from register file (rs1 and rs2) then store back the output value into rd.

I-type: the input value is take from register file, immediate generation or program counter and also store back the output value into rd.

To implement lui operation, the ALU need to add a new alu operation OPB that ALU output data equal to operand b data.

The ALU design requirement is “do not use subtraction (−), comparison (<, >), orshifting(≪, ≫, and ≫).”.

To perform subtraction (-) operation, calculate a - b by adding a to the two’s complement of b (a + ~b + 1).

To compare two numbers:

If operands a and b are unsigned, examine the overflow of the result from a - b: if overflow[a - b] = 1, then a < b; otherwise, a >= b.

If operands a and b are signed, check the MSB of a - b as above when a and b have the same sign. If a and b have opposite signs, then simply examine the sign bits of the operands.

To perform shift (<<, >>, and >>>), we create 5 stage mux which each stage result is a shift operation of operand_a with each low bit of operand_b. There is a example of SLL operation present in Figure 2. The operation SRL and SRA is design similar to the operation SLL.

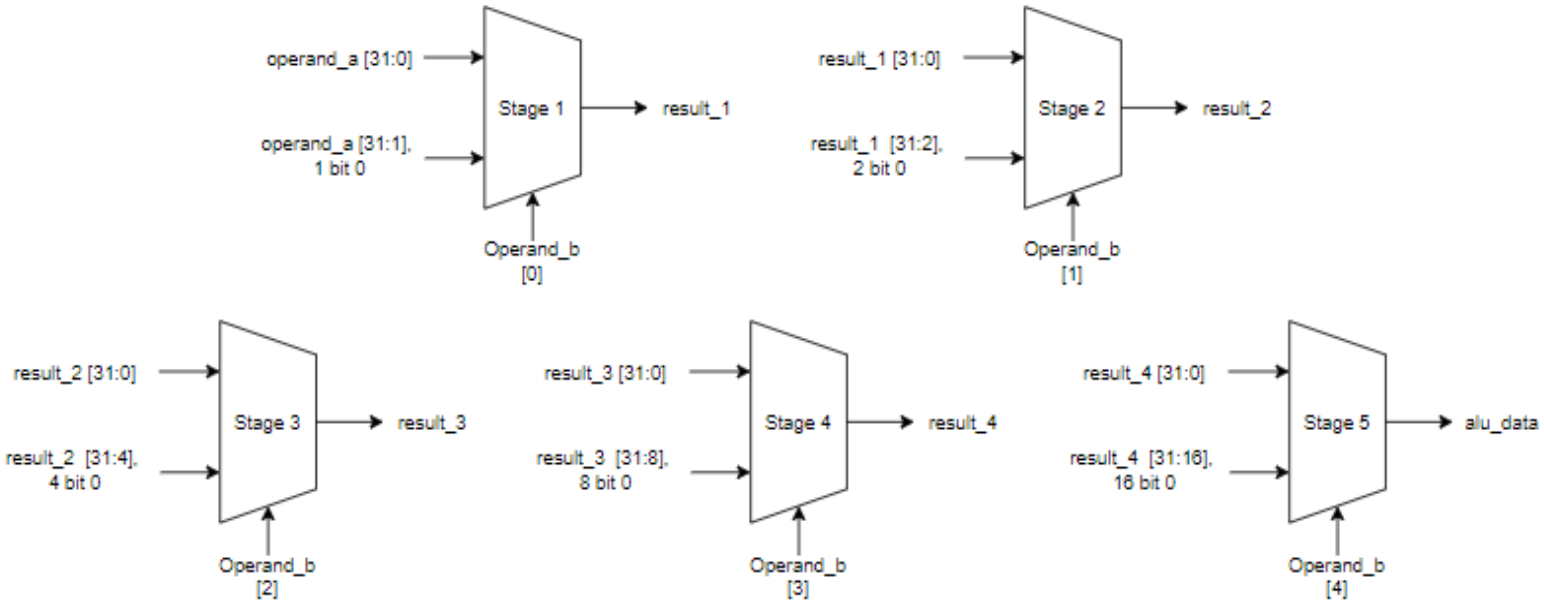

Figure 2: SLL operation

Table 1: The ALU Operation

alu_op	**Opcode**	**Description (R type)**	**Description (I type)**
ADD (0)	0000	rd = rs1 + rs2	rd = rs1 + imm
SUB (1)	0001	rd = rs1 − rs2	n/a
SLT (2)	0010	rd = (rs1 < rs2) ? 1 : 0	rd = (rs1 < imm) ? 1 : 0
SLTU (3)	0011	rd = (rs1 < rs2) ? 1 : 0	rd = (rs1 < imm) ? 1 : 0
XOR (4)	0100	rd = rs1 ^ rs2	rd = rs1 ^ imm
OR (5)	0101	rd = rs1 ∣ rs2	rd = rs1 ∣ imm
AND (6)	0110	rd = rs1 & rs2	rd = rs1 & imm
SLL (7)	0111	rd = rs1 << rs2[4:0]	rd = rs1 << imm[4:0]
SRL (8)	1000	rd = rs1 >> rs2[4:0]	rd = rs1 >> imm[4:0]
SRA (9)	1001	rd = rs1 >>> rs2[4:0]	rd = rs1 >>> imm[4:0]
OPB (10)	1010	rd = rs2	rd = rs2

2.2.1. Specification

Signal	Width	Direction	Description
i_operand_a	32	Input	First operand for ALU operations.
i_operand_b	32	Input	Second operand for ALU operations.
i_alu_op	4	Input	The operation to be performed.
o_alu_data	32	Output	Result of ALU operation

2.3. BRU

The BRU (branch comparison unit) compare two registers rs1 and rs2 with two function: the bit compara tor and the sign converter.

The input of this unit include of rs1 data, rs2 data from regfile and br_un from control unit. The function is selected base on i_br_un signal (sign went active).

The BRU design requirement is “do not use subtraction (−), comparison (<, >).

If the operands a and b are unsigned integers, compare their MSB after performing the operation rs1_data + ~rs2_data + 1. If the overflow bit of operation is 1, then a < b (br_less active); otherwise, a >= b (br_less negative).

If the operands rs1_data and rs2_data are signed integers:

If both and b are either both positive or both negative, just use the MSB of operation rs1_data + ~rs2_data + 1 as result for br_less.

If rs1_data and rs2_data have opposite signs, just use rs1_data’s MSB as result of br_less.

The br_equal active when the result and overflow bit of a + (~b) + 1 equal to zero.

2.3.1. Specification

Signal	Width	Direction	Description
i_rs1_data	32	Input	Data from the first register
i_rs2_data	32	Input	Data from the second register
i_br_un	1	Input	Comparison mode (1 if signed, 0 if unsigned).
o_br_less	1	Output	Output is 1 if *rs*1 < *rs*2.
o_br_equal	1	Output	Output is 1 if *rs*1 = *rs*2.

2.4. Regfile

The register file is a critical component in a RISC-V processor, designed to store temporary values that are needed for the execution of instructions. Here's an analysis of its operation, features, and considerations based on the provided specifications:

a. Purpose and Functionality:

The register file serves as a bank of 32 registers, each 32 bits wide, that stores operands and results of instructions.

It supports three simultaneous access operations: two read operations and one write operation. This means two registers can be read while data is written to a third register within the same clock cycle.

Register 0 is hardwired to always read as zero, which is typical in RISC architectures and useful for various operations like clearing registers or conditionally zeroing out data.

b. Key Signals:

Clock (i_clk): Synchronizes all register operations, ensuring that reads and writes occur at specific clock edges.

Reset (i_rst): Initializes all registers, typically setting them to zero during power-on or reset events.

Read Addresses (i_rs1_addr, i_rs2_addr): Specify which registers are to be read. These addresses are 5 bits wide, allowing access to 32 different registers.

Write Address (i_rd_addr): Specifies which register to write data to. This also uses a 5-bit address width.

Write Enable (i_rd_wren): Controls whether data is written to the register specified by i_rd_addr. If this signal is inactive, no data will be written, regardless of the address or data values.

c. Operation and Control Logic:

Read Operations: The register file outputs the values stored in the registers specified by i_rs1_addr and i_rs2_addr to o_rs1_data and o_rs2_data, respectively. These outputs are generated based on the provided addresses and are independent of the write operations, meaning reads can happen concurrently with writes.

2.4.1. Specification

Signal	Width	Direction	Description
i_clk	1	Input	Global clock.
i_rst	1	Input	Global active reset.
i_rs1_addr	5	Input	Address of the first source register.
i_rs2_addr	5	Input	Address of the second source register.
o_rs1_data	32	Output	Data from the first source register.
o_rs2_data	32	Output	Data from the second source register.
i_rd_addr	5	Input	Address of the destination register
i_rd_data	32	Input	Data to write to the destination register.
i_rd_wren	1	Input	Write enable for the destination register

2.5. I/O System and Memory:

In this part, we implement a Load-Store Unit (LSU) which functions as an I/O System and Memory. According to Havard structure, we separate I/O system and Memory two part: Instruction Memory and Load Store Unit.

Boundary address	Mapping
0x7820 -- 0xFFFF	(Reserved)
0x7810 -- 0x781F	Buttons
0x7800 -- 0x780F	Switches *(required)*
0x7040 -- 0x70FF	(Reserved)
0x7030 -- 0x703F	LCD Control Registers
0x7020 -- 0x7027	Seven-segment LEDs
0x7010 -- 0x701F	Green LEDs *(required)*
0x7000 -- 0x700F	Red LEDs *(required)*
0x4000 -- 0x6FFF	(Reserved)
0x2000 -- 0x3FFF	Data Memory (8KiB using SDRAM) *(required)*
0x0000 -- 0x1FFF	Instruction Memory (8KiB) *(required)*

Figure 3: LSU Mapping

Instruction Memory is implemented as 2048 x 8 bit registers (8 KiB). An instruction has 32 bit in lengh, so we need four registers for an instruction. That means the distance between two instructions is four registers (or four addresss).

The LSU includes 5 main components: Decoder, Input_Buffer, Output_Buffer and Data_Memory that present in Figure 4.

To prevent misaligned, LSU's address need to clear two LSB for SW, LW operation or clear LSB for SH, SHU, LH.

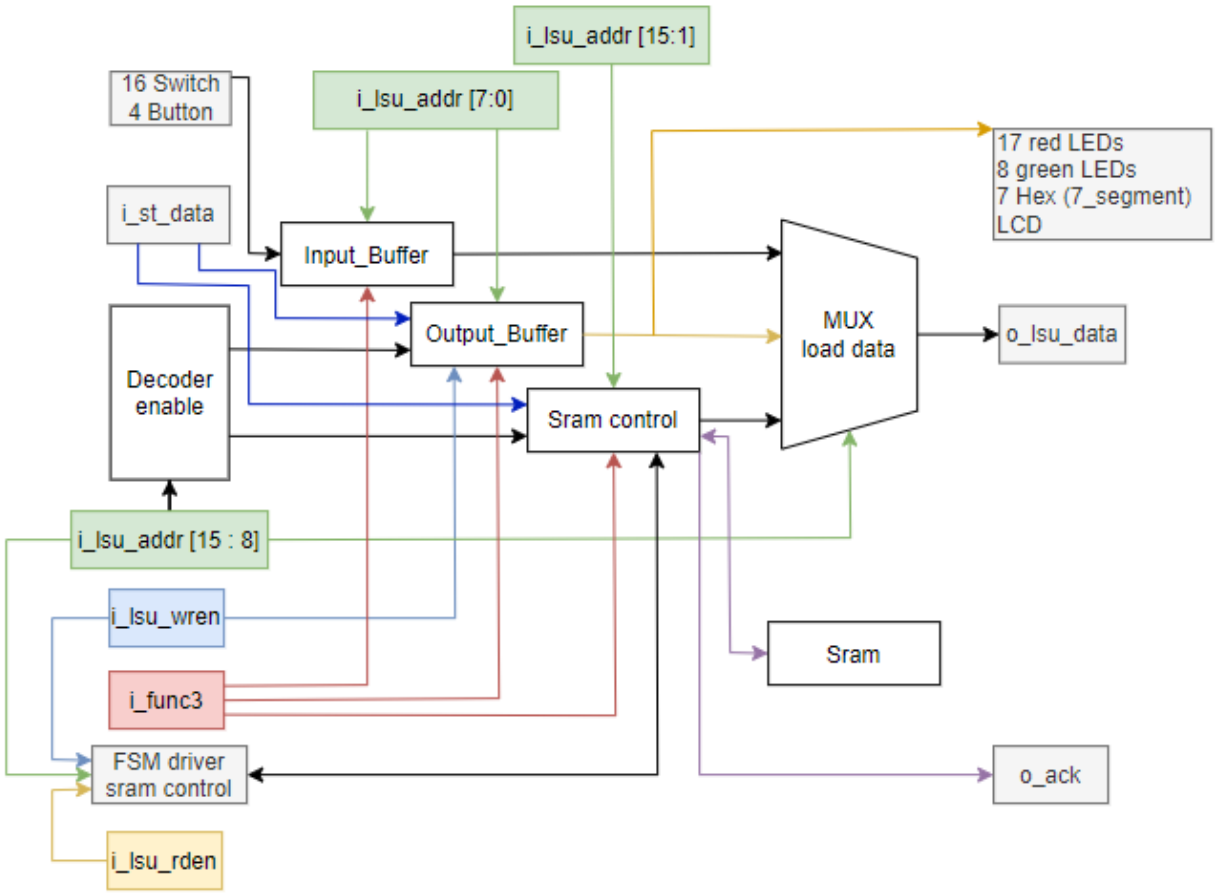

Figure 4: LSU Diagram.

LSU's address is range 0x2000 to 0x781F, which represented in 16 bits, but i_lsu_data represented in 32 bits (4 byte). We only use 2 byte low of LSU address and don't care the other.

DECODER: base on i_lsu_adder [15 : 8] (second byte) to select the store region (Output_buffer and Data_Memory).

MUX: base on i_lsu_adder [15 : 8] (second byte) to select the load region (Input_buffer, Output_buffer and Data_Memory).

*I/O System

The proccessor interacts with environment through Input_Buffer and Output_Buffer.

Input_Buffer and Output_Buffer is implemented as 8-bit register array (similar to Instruction Memory).

Input_Buffer: contains value of Switch and Button Signal. When we interact with Switch and Button that reflected in the region of Input_Buffer (the address of register). It's address is in range 0x7800 to 0x781F. The Input_Buffer is implemented as 32 x 8 bit-registers (32 byte). Because we use the second byte to select region of LSU, we just use the first byte as address for interact with Input_Buffer. We can't store data in Input_Buffer, only load the data from Input_Buffer.

Output_Buffer: is connected to the red LEDs, green LEDs, 7-segment displays and LCD1602. It's address is in range 0x7000 to 0x703F. The Output_Buffer is implemented as 64 x 8 bit-registers (64 byte). Similar to Input_Buffer, we just use the first byte as address for interact with Out_Buffer. We can load or store data in Output_Buffer.

*Data Memory

The Data Memory is used to store data and load it out when needed. It's address is in range 0x2000 to 0x3FFF. In this implementation, we use sram IS61WV25616 replace for Data_Memory.

We use the sram_control module which is provided (2 cycle for write and 5 cycle for read).

According to the datasheet, this sram is 2 bytes data width corresponding to an address. We need to adjust the LSU's address that fit to sram's address. Solution is shift right LSU's address one bit, we have sram's address. Because one LSU's address can be stored one byte data, but one sram's address can be stored two byte data. So the LSU's address is "double" than the sram's address. An example is presented in Figure 5.

LSU's address		Sram's address
0x0001	0x0000	0x0000
0x0003	0x0002	0x0001
0x0005	0x0004	0x0002
0x0007	0x0006	0x0003

Figure 5: Example of relation between LSU's address and Sram address

Sram has 18 bits address width, we just use 15 bit low, so 3 bit high set to zero.

Because the sram cause 2 or 5 cycle for store or load and the design using i_lsu_wren for control both store and load. In some sistuation, the result of ALU in range of x2000 to 0x3FFF (Sram's address region) and this cause something can't control because i_lsu_wren negative is read operation. So the Sram need one more signal (i_lsu_rden) for more stable control. When both i_lsu_wren and i_lsu_rden negative, the sram do nothing.

To driver this module, we design a combination logic for driver BMASK and store data by i_lsu_wren, i_lsu_rden and func_3 (sw, sb, sh, lw, lb, lbu, lh, lhu) that present in Figure 6 and a ASM for driver WREN and RDEN that is present in Figure 7.

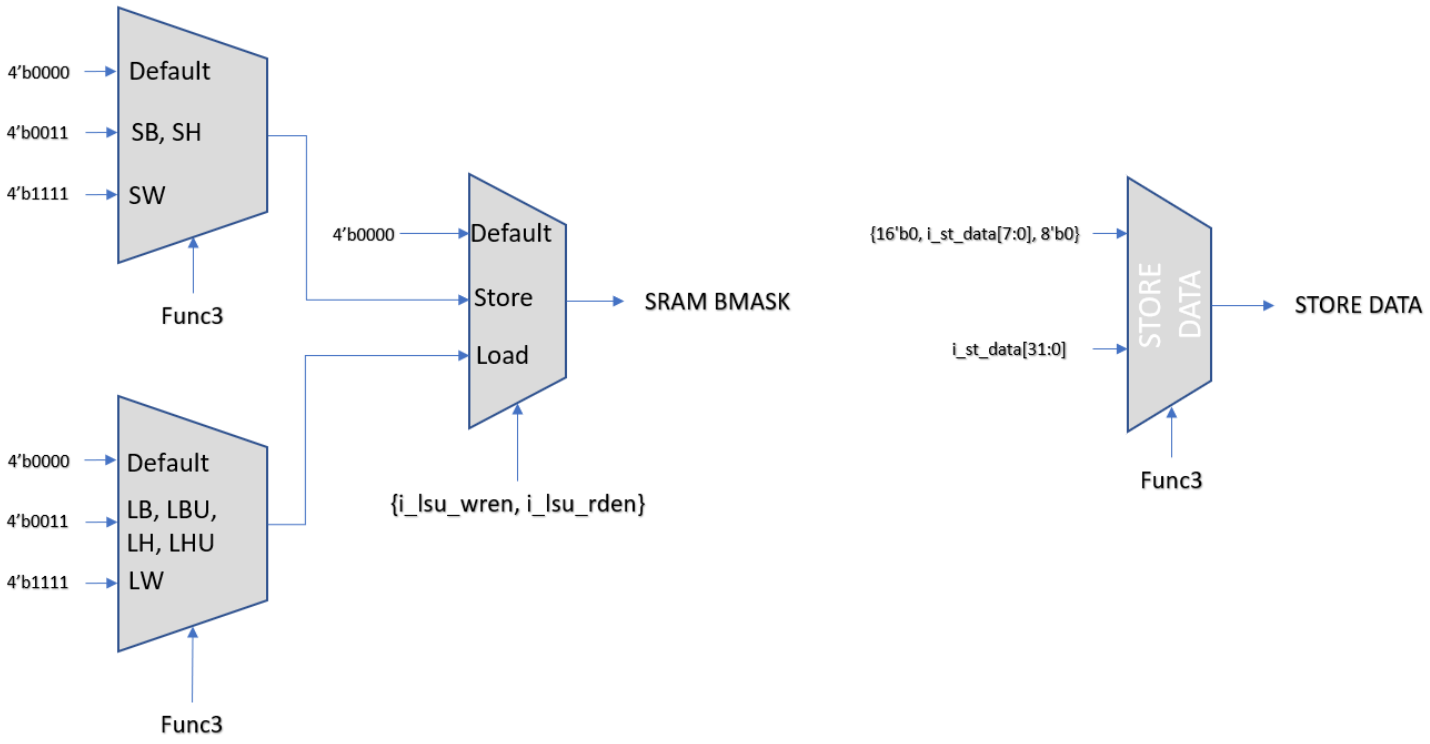

Figure 6: Combinational logic of bit mask and store data control

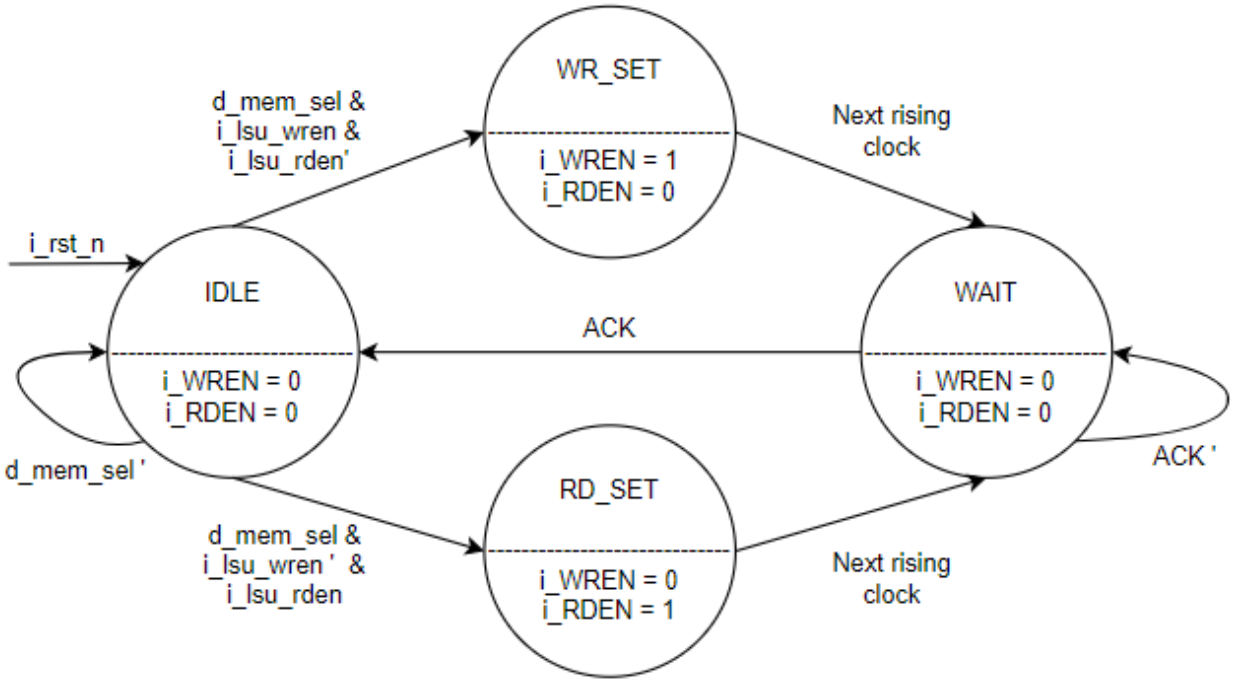

Figure 7: FSM Driver Sram Control.

Because the operation delay 2 cycle for write and 5 cycle for read, we need to hold the PC value, wait for the operation completely. Solution is design a combinational logic that can stop PC update next value after each cycle which present in Figure 8.

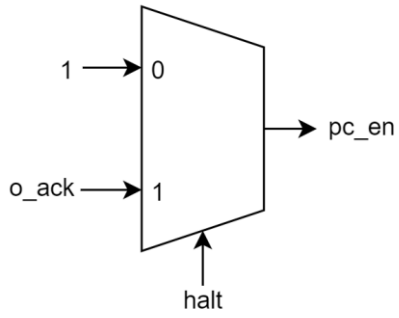

Figure 8: Combinational logic of control program counter

The halt signal is active when meet these condition: Opcode is of R-format layout (Store) or I-format load and i_lsu_adder in Data_Memory's address range (0x2000 to 0x3FFF).

The ack signal is feelback signal of sram control module. When it's active means the operation completely and allow PC update next value.

2.5.1 Specification

Signal	Width	Direction	Description
i_clk	1	input	Globa clock, active on the rising edge.
i_rst_n	1	input	Globa low active reset.
i_lsu_addr	32	input	Address for data read or write.
i_st_data	32	input	Data to be store.
i_lsu_wren	1	input	Write enable signal (1 if writting).
i_func_3	3	input	Select function sw, sb, sh, lw, lb, lbu, lh, lhu.
o_ld_data	32	output	Data read from memory.
o_io_ledr	32	output	Output for red LEDs.
o_io_ledg	32	output	Output for green LEDs.
o_io_hex0…7	7	output	Output for 7-segment displays.
o_io_lcd	32	output	Output for the LCD register.
i_io_sw	32	input	Input for switch.
i_io_btn	4	input	Input for button.
o_ack	1	output	Signal to pc_halt_by_sram

2.6. Control Unit:

Based on the RISCV ISA instruction set table, the team will design the control unit, based on opcode (inst[6:2]), funtion 3 (inst[14:12]), function 7 (inst[30]), to decide the control signals according to the table following statistics:

INSTRUCTION	TYPE	OPCODE [6:2]	funct7 [30]	funct3 [14:12]	pc_sel	rd_wren	br_un	br_less	br_equal	opa_sel	opb_sel	alu_op	mem_rd	mem_wren	wb_sel	data_out_addj
ADD	R	01100	0	000	PC + 4	allow	x	x	x	rs1_data	rs2_data	ADD	unallow	unallow	alu_data	x
SUB	R	01100	1	000	PC + 4	allow	x	x	x	rs1_data	rs2_data	SUB	unallow	unallow	alu_data	x
SLL	R	01100	0	001	PC + 4	allow	x	x	x	rs1_data	rs2_data	SLL	unallow	unallow	alu_data	x
SLT	R	01100	0	010	PC + 4	allow	x	x	x	rs1_data	rs2_data	SLT	unallow	unallow	alu_data	x
SLTU	R	01100	0	011	PC + 4	allow	x	x	x	rs1_data	rs2_data	SLTU	unallow	unallow	alu_data	x
XOR	R	01100	0	100	PC + 4	allow	x	x	x	rs1_data	rs2_data	XOR	unallow	unallow	alu_data	x
SRL	R	01100	0	101	PC + 4	allow	x	x	x	rs1_data	rs2_data	SRL	unallow	unallow	alu_data	x
SRA	R	01100	1	101	PC + 4	allow	x	x	x	rs1_data	rs2_data	SRA	unallow	unallow	alu_data	x
OR	R	01100	0	110	PC +4	allow	x	x	x	rs1_data	rs2_data	OR	unallow	unallow	alu_data	x
AND	R	01100	0	111	PC +4	allow	x	x	x	rs1_data	rs2_data	AND	unallow	unallow	alu_data	x
ADDI	I	00100	x	000	PC + 4	allow	x	x	x	rs1_data	rs2_data	ADD	unallow	unallow	alu_data	x

SLLI	I	00100	x	001	PC + 4	allow	x	x	x	rs1_data	rs2_data	SLL	unallow	unallow	alu_data	x
SLTI	I	00100	x	010	PC + 4	allow	x	x	x	rs1_data	rs2_data	SLT	unallow	unallow	alu_data	x
SLTIU	I	00100	x	011	PC + 4	allow	x	x	x	rs1_data	rs2_data	SLTU	unallow	unallow	alu_data	x
XORI	I	00100	x	100	PC + 4	allow	x	x	x	rs1_data	rs2_data	XOR	unallow	unallow	alu_data	x
SRLI	I	00100	x	101	PC + 4	allow	x	x	x	rs1_data	rs2_data	SRL	unallow	unallow	alu_data	x
SRAI	I	00100	1	101	PC + 4	allow	x	x	x	rs1_data	rs2_data	SRA	unallow	unallow	alu_data	x
ORI	I	00100	x	110	PC + 4	allow	x	x	x	rs1_data	rs2_data	OR	unallow	unallow	alu_data	x
ANDI	I	00100	x	111	PC +4	allow	x	x	x	rs1_data	rs2_data	AND	unallow	unallow	alu_data	x
LB	I	00000	x	000	PC + 4	allow	x	x	x	rs1_data	imm_data	ADD	allow	unallow	LSU_data	{24{data[7]},data[7:0]}
LH	I	00000	x	001	PC + 4	allow	x	x	x	rs1_data	imm_data	ADD	allow	unallow	LSU_data	{16{data[15]},data[15:0]}
LW	I	00000	x	010	PC + 4	allow	x	x	x	rs1_data	imm_data	ADD	allow	unallow	LSU_data	data[31:0]
LBU	I	00000	x	100	PC + 4	allow	x	x	x	rs1_data	imm_data	ADD	allow	unallow	LSU_data	{24'b0,data[7:0]}
LHU	I	00000	x	101	PC + 4	allow	x	x	x	rs1_data	imm_data	ADD	allow	unallow	LSU_data	{16'b0,data[15:0]}

SB	S	01000	x	000	PC + 4	unallow	x	x	x	rs1_data	imm_data	ADD	unallow	allow	x	x
SH	S	01000	x	001	PC + 4	unallow	x	x	x	rs1_data	imm_data	ADD	unallow	allow	x	x
SW	S	01000	x	010	PC + 4	unallow	x	x	x	rs1_data	imm_data	ADD	unallow	allow	x	x
BEQ	B	11000	x	000	br_equal	unallow	sign	x	1	PC	imm_data	ADD	unallow	unallow	x	x
	B	11000	x	000	br_equal	unallow	sign	x	0	PC	imm_data	ADD	unallow	unallow	x	x
BNE	B	11000	x	001	!br_equal	unallow	sign	x	0	PC	imm_data	ADD	unallow	unallow	x	x
	B	11000	x	001	!br_equal	unallow	sign	x	1	PC	imm_data	ADD	unallow	unallow	x	x
BLT	B	11000	x	100	br_less	unallow	sign	1	0	PC	imm_data	ADD	unallow	unallow	x	x
	B	11000	x	100	br_less	unallow	sign	0	x	PC	imm_data	ADD	unallow	unallow	x	x
BGE	B	11000	x	101	!br_less	unallow	sign	0	x	PC	imm_data	ADD	unallow	unallow	x	x
	B	11000	x	101	!br_less	unallow	sign	1	0	PC	imm_data	ADD	unallow	unallow	x	x
BLTU	B	11000	x	110	br_less	unallow	unsign	1	0	PC	imm_data	ADD	unallow	unallow	x	x
	B	11000	x	110	br_less	unallow	unsign	0	x	PC	imm_data	ADD	unallow	unallow	x	x
BGEU	B	11000	x	111	!br_less	unallow	unsign	0	x	PC	imm_data	ADD	unallow	unallow	x	x
	B	11000	x	111	!br_less	unallow	unsign	1	0	PC	imm_data	ADD	unallow	unallow	x	x

JAL	J-U	11011	x	x	PC + imm (alu)	allow	x	x	x	PC	imm_data	ADD	unallow	unallow	PC + 4	x
JALR	J-I	11001	x	x	PC + imm (alu)	allow	x	x	x	rs1_data	imm_data	ADD	unallow	unallow	PC + 4	x
AUIPC	U	00101	x	x	PC + 4	allow	x	x	x	rs1_data	imm_data	ADD	unallow	unallow	alu_data	x
LUI	U	01101	x	x	PC + 4	allow	x	x	x	x	imm_data	operand b	unallow	unallow	alu_data	x

2.6.1. Specification

Signal	Width	Direction	Description
i_instruction	32	Input	Data from Instruction Memory.
i_br_less	1	Input	Result of equal comparison from BRC
i_br_equal	1	Input	Result of less comparison from BRC
o_pc_sel	1	Output	Signal control program counter
o_rd_wren	1	Output	Signal control write enable for regfile
o_insn_vld	1	Output	Signal anounce instruction valid or not
o_br_un	1	Output	Signal control comparison sign or unsign for BRC
o_opa_sel	1	Output	Signal select input data for the ALU (rs1 or pc)
o_opb_sel	1	Output	Signal select input data for the ALU (rs2 or imm)
o_mem_wren	1	Output	Enable writing for data memory
o_mem_rd_en	1	Output	Enable read only for sram
o_alu_op	4	Output	Select the operation for ALU
o_wb_sel	2	Output	Select the write back date for regfile

2.7. Immediate Generator

The implementation of the immediate generator is a critical component in the design of a RISC-V CPU, responsible for arranging the bits or bytes of instructions to form the appropriate immediate values. In the RISC-V format, there are various types ofimmediates, including I, B, S, J, and U formats, which require the use of multiplexers to direct the wiring order of the bus. The figure illustrates a design that guides the bus of immediates for these different formats.

The I-format serves as the starting point, requiring minimal sign extension and acting as a reference for subsequent stages. The S-stage, identified by a MUX, transforms the bus into an S-type immediate, distinct from the I-type. Similar approaches are applied to B-type masking over S-type and J-type masking over J-type, with special note taken for the commonality of the last bits in J and B types, which are handled separately to reduce the need for additional MUXes.

The U-type is entirely different from the others, and the chosen mechanism is applied to determine whether it is a U-type or part of the multi-stage ISBJ formats. The selection of the immediate bus path is governed by the imm sel port, which is protocol decoded in a one-hot format with 5 bits, ensuring precise and effective control of immediate generation.

The instruction 32 bit in Single Cycle RISCV CPU is composed of different fields that encode the operation, the operands and the destination of the result. One of these fields is the bit that, which indicates whether the instruction is an immediate or a register instruction. The table below shows some summary of how the bit that affects the generation of immediate values from the instruction 32 bit in Single Cycle RISCV CPU.

Instruction Type	Bit Extraction
I-Type	{{21{i_inst[31]}}, i_inst[30:20]}
S-Type	{{21{i_inst[31]}}, i_inst[30:25], i_inst[11:7]}
B-Type	{{20{i_inst[31]}}, i_inst[7], i_inst[30:25], i_inst[11:8],1'b0}
U-Type	{{12{i_inst[31]}}, i_inst[19:12], i_inst[20], i_inst[30:21],1'b0}
J-Type	{i_inst[31:12], 12'b0}

2.7.1. Specification

Signal	Width	Direction	Description
i_instruction	32	Input	Data from Instruction Memory.
o_imm	32	Output	Output data of ImmGen

2.8. Program counter

Program counter degisn as a sequential system. At the rising of clock, PC will update new PC value (PC + 4 or PC + immediate). Because the sram cause 2 or 5 cycle delay for store or load, we add a enable signal for halt the PC. When the enable signal negactive, PC hold it’s value and can’t update new PC value until active.

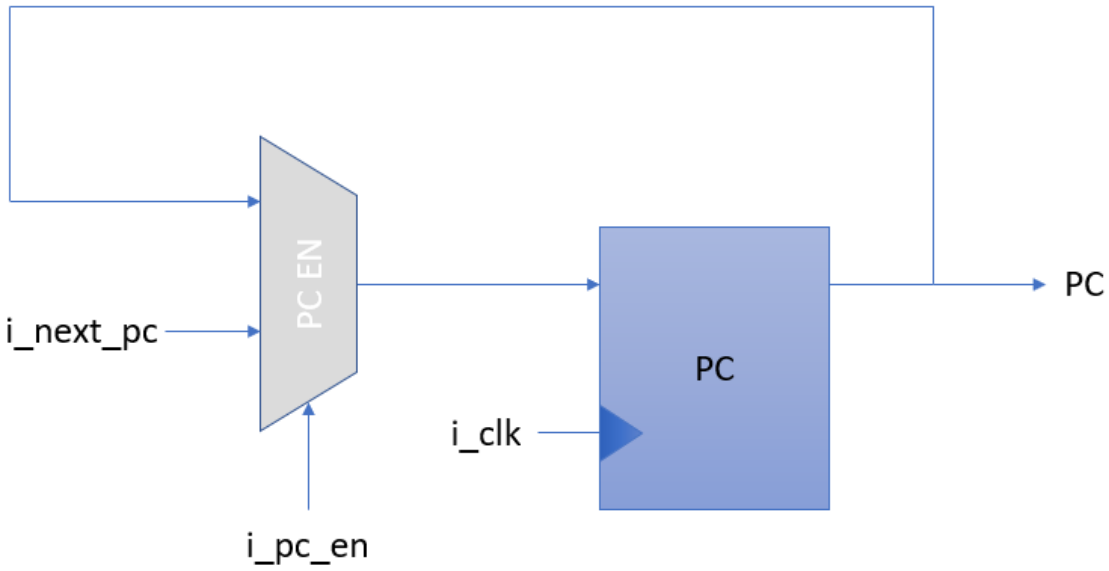

Figure 8: Program Counter

2.8.1. Specification

Signal	Width	Direction	Description
i_next_pc	32	Input	New PC value
i_pc_en	1	Input	Signal hold PC value
o_pc	32	Output	Output data of ImmGen

3. Verification Strategy

3.1. ALU

To verify the function of ALU, randomly the value of operand a and operand b, then compare the expected result with the output value of ALU corresponding to each operands case. Total sample around 500 testcase.

```
=== Starting Random Tests ===

OP_SLT 2: 37 < 24 ? 0 (ALU output: 0)- PASS

OP_SUB 1: 17- 4294967291 = 22 (ALU output: 22)- PASS

OP_AND 6: fffffff8 & 13 = 10 (ALU output: 10)- PASS

OP_AND 6: fffffff6 & ffffffe1 = ffffffe0 (ALU output: ffffffe0)- PASS

 OP_SLL 7: fffffc5 << 23 = e2800000 (ALU output: e2800000) PASS

OP_ADD 0: 4294967294 + 6 = 4 (ALU output: 4)- PASS

OP_OUTPUT_B 10: Operand B = 55 (ALU output: 55)- PASS

OP_AND 6: c & 20 = 0 (ALU output: 0)- PASS

…..

OP_SRA 9: 6 >>> 16 = 0 (ALU output: 0)- PASS

OP_AND 6: ffffffe6 & 41 = 40 (ALU output: 40)- PASS

OP_OUTPUT_B 10: Operand B = 4294967239 (ALU output: 4294967239)- PASS

OP_SLTU 3: 25 < 12 ? 0 (ALU output: 0)- PASS

Test Summary:
```

```
Total test cases: 500

 Passed: 500

Failed: 0

Pass rate: 100.00%
```

3.2. BRC

To verify the function of BRC, design two test bench for sign and unsign. Randomly the value of rs1 data and rs2 data, then compare the expected result with the the output value of BRC. Total sample around 500 testcase.

```
=== Starting Random Tests ===

 Random Signed Test PASSED - a: 303379748, b: -1064739199, unsigned: 0

 Random Signed Test PASSED - a: -2071669239, b: -1309649309, unsigned: 0

 Random Signed Test PASSED - a: 112818957, b: 1189058957, unsigned: 0

 Random Signed Test PASSED - a: -1295874971, b: -1992863214, unsigned: 0

 Random Signed Test PASSED - a: 777537884, b: -561108803, unsigned: 0

….

Random Unsigned Test PASSED - a: -359791147, b: 220526106, unsigned: 1

Random Unsigned Test PASSED - a: 1559232697, b: -1685626569, unsigned: 1

Random Unsigned Test PASSED - a: -1084580994, b: 1837990107, unsigned: 1

Random Unsigned Test PASSED - a: 1740879567, b: -1125307527, unsigned: 1
```

```
Random Unsigned Test PASSED - a: 2099832058, b: 817688161, unsigned: 1

Random Unsigned Test PASSED - a: -1953232617, b: 1358264481, unsigned: 1



=== Test Summary ===

Total Tests: 500

Passed: 500

Failed: 0

Pass rate: 100.00%
```

3.3. Regfile

To verify the function of Register File, design a test bench that can be check value of all register equal to zero after reset, check the value of register 0 always equal to zero and check all register after receive a value feedback (depend on rd_en).

```
=== Starting Tests ===

PASS: All registers reset to 0

PASS: Register[1] = a5a5a5a5 Expected = a5a5a5a5

PASS: Register[2] = 5a5a5a5a Expected = 5a5a5a5a

PASS: Register[3] = 12345678 Expected =  12345678

……

PASS: Register[31] = faceb00c Expected = faceb00c
```

```
PASS: Register[0] = 00000000 Expected = 00000000

PASS: Register x0 = 0 (unchanged)

=== Test Summary ===

Total Tests: 50

Passed: 50

Failed: 0

Pass rate: 100.00%
```

3.4. I/O system and memory

To verify the operation of the Load-Store Unit (LSU), a driver will be created to generate random input stimuli as follows:

Randomized Address Selection: Randomly assign values to dut -> i_lsu_addr within the LSU's addressable ranges (0x2000–0x781F). This will include generating addresses for the Input_Buffer (0x7800–0x781F), Output_Buffer (0x7000–0x703F), and Data_Memory (0x2000–0x3FFF) (replace for sram because we can't verification it). For each address range, 32-bit random data (dut -> i_st_data) will be generated to verify both read and write operations across LSU regions.

Randomized Write Enable and Reset Signals: Randomly set the dut -> i_lsu_wren signal to 1 or 0 to toggle between read and write operations with an equal probability (50%)

for each. Toggle the i_rst_n signal periodically to ensure that the LSU can properly reset and resume normal operation, as described in the test setup.

Output Assertions:

Use assertions to verify that the LSU outputs match expected behaviors based on address decoding and data operations. For example: Check that data written to Output_Buffer or Data_Memory can be read back correctly. Assert that o_ld_data (for memory read operations) and o_io_* signals (for I/O operations) are consistent with the expected results, ensuring correct LSU functionality.For Input_Buffer, assert that only reads occur and data is not modified by any write operations.

This verification setup ensures comprehensive testing of the LSU's operation, covering a wide range of normal and edge-case scenarios to validate functionality under various input conditions.

```
=== Starting Tests ===
PASS: Write Test - Addr: 00007000, Data: 12345678, LEDR: 12345678
PASS: Write Test - Addr: 00007002, Data: 0000abcd, LEDR: abcd5678
PASS: Write Test - Addr: 00007001, Data: 000000ef, LEDR: abcdef78
PASS: Read Test - Addr: 00007800, Control: 010, Data: 89abcdef
PASS: Read Test - Addr: 00007802, Control: 001, Data: ffff89ab
PASS: Read Test - Addr: 00007803, Control: 000, Data: ffffff89
….
```

```
PASS: Read Test - Addr: 00007803, Control: 100, Data: 00000089
PASS: Read Test - Addr: 00007802, Control: 101, Data: 000089ab
PASS: Write Test - Addr: 00007010, Data: faceb00c, LEDR: abcdef78
PASS: Write Test - Addr: 00007020, Data: 0000005a, LEDR: abcdef78
PASS: Read Test - Addr: 00007020, Control: 000, Data: 0000005a
=== Test Summary ===
Total Tests: 500
Passed: 500
Failed: 0
Pass rate: 100.00%
```

3.5. Control Unit

To verify the functionality of the Control Unit, we will use a driver to generate random input stimuli as described below:

Simulation Count: Run MAX_SIM = 2000 simulations to thoroughly test the Control Unit across a broad spectrum of cases, including both typical and edge-case scenarios.

Random Operation Mode Selection:

Randomly set the unsigned/signed mode to verify that the Control Unit can correctly handle both signed and unsigned operations. This is essential for testing the

Control Unit's ability to perform comparisons and branching accurately, regardless of operand sign.

Random Operand Generation:

Generate random 32-bit values for the two operands, ensuring diverse input values. This includes testing with both small and large numbers to cover a full range of potential inputs. This approach allows us to examine how the Control Unit handles various operand values in different operations.

Output Assertions:

Use assertions to check that the output signals (such as the comparison results for less-than or equal) behave as expected based on the operation mode and operand values. This helps verify that the Control Unit is performing correct branching and comparison operations:

For unsigned operations, confirm that the output signals accurately reflect the comparison between the operands.

For signed operations, ensure that the outputs align with expected behavior for signed values.

Assertions will cover conditions where the first operand is less than, greater than, or equal to the second operand, confirming that the Control Unit sets the output signals correctly

ADD : PASS	SUB : PASS	SLL : PASS	SLL: PASS
SLTU : PASS	XOR : PASS	SRL : PASS	SRA : PASS
OR : PASS	AND : PASS	SW : PASS	LB : PASS
LH : PASS	LW : PASS	LBU : PASS	LHU : PASS
BEQ : PASS	BNE : PASS	BLT : PASS	BGE : PASS
ADDI : PASS	SLTI : PASS	SLTIU : PASS	XORI : PASS
ORI : PASS	ANDI : PASS	SLLI : PASS	SRAI : PASS
SH : PASS	BLTU : PASS	BGEU : PASS	LUI : PASS
AUIPC : PASS	JAL : PASS	JALR : PASS	SRLI : PASS
SB : PASS			

3.6. Immediate Generator

To verify the functionality of the Immediate Generator, a driver will be created to generate various types of input stimuli, covering both typical and edge-case scenarios:

Simulation Count: Set the number of simulations to MAX_SIM = 2000 to ensure thorough testing across multiple cases and immediate types.

Randomized Immediate Type Selection:

Randomly select the type of immediate (e.g., I-type, S-type, B-type, U-type, and J-type) to verify the Immediate Generator's handling of each type. This tests its ability to correctly interpret and generate immediate values based on the instruction format.

Randomized Instruction Input Generation:

Generate random 32-bit values for the instruction input to simulate various possible instructions. The randomized instructions will contain different bit patterns, allowing comprehensive testing of how the Immediate Generator extracts and extends immediate fields according to each instruction type's format.

Output Assertions:

Use assertions to validate that the generated immediate values match the expected results based on the selected instruction type. Each type has a unique format for immediate extraction and sign extension, so the assertions will verify:

For I-type, that the 12-bit immediate is correctly sign-extended to 32 bits.

For S-type, that the immediate is accurately constructed by combining bits from two different fields, and then sign-extended.

For B-type, that the immediate is assembled with correct bit shifting and sign extension for branching.

For U-type, that the upper 20 bits are loaded as-is, with the lower bits set to zero.

For J-type, that the immediate is correctly assembled and extended for jump instructions.

```
Starting tests for imm_gen module...

<Test: ADD >0: Instruction 002081b3, Calculated Imm = 00000000

<Test: SUB>1: Instruction 402081b3, Calculated Imm = 00000000

<Test: SLL>2: Instruction 002091b3, Calculated Imm = 00000000

<Test: SLT>3: Instruction 0020a1b3, Calculated Imm = 00000000

<Test: SLTU >4: Instruction 0020b1b3, Calculated Imm = 00000000

<Test: XOR>5: Instruction 0020c1b3, Calculated Imm = 00000000

<Test: SRL>6: Instruction 0020d1b3, Calculated Imm = 00000000

…..

<Test: SRA>7: Instruction 4020d1b3, Calculated Imm = 00000000

<Test: OR >8: Instruction 0020e1b3, Calculated Imm = 00000000

<Test: AND>9: Instruction 0020f1b3, Calculated Imm = 00000000

<Test: ADDI >10: Instruction = 12608193, Calculated Imm = 0000012b

<Test: SLTI > 0000012b11: Instruction = 12b0a193, Calculated Imm = 0000012b

<Test: XORI > 0000012b12: Instruction. 12b0c193, Calculated Imm = 0000012b

=== Test Summary ===

Total Tests: 500

Passed: 500

Failed: 0

Pass rate: 100.00%
```

4. Alternative Design

Watchdog Timer (WDT) is a circuit or component in embedded systems and computers designed to monitor system activity and detect faults, such as freezes or unresponsiveness. If the system fails to operate correctly within a specified period, the WDT takes action to restore functionality, typically by restarting the system.

The Watchdog Timer is configured with a specific timeout period. The system must periodically send a signal (often called a "kick" or "feed") to the WDT before the timeout expires. This indicates the system is functioning normally. If the WDT doesn't receive the signal within the timeout period, it assumes the system has malfunctioned or frozen. When the timeout occurs, the WDT typically triggers action that reset the system and active a signal error (LED, buzzer,…).

In this implementation, WDT is designed as sequential system with four input: i_en, i_kick, i_clk, i_rst_n and one output: o_wdt_rst_n. At the rising of clock, i_kick active when present program counter not equal to next program counter.

If i_en active, WDT moniter the i_kick. If the i_kick active, WDT increase the counter variable by one. When the counter equal to the set value (ex: 5, 10, ….), WDT send a signal to reset the processor and repeat the loop. We provide more detail of WDT in the report of Milestone 3.

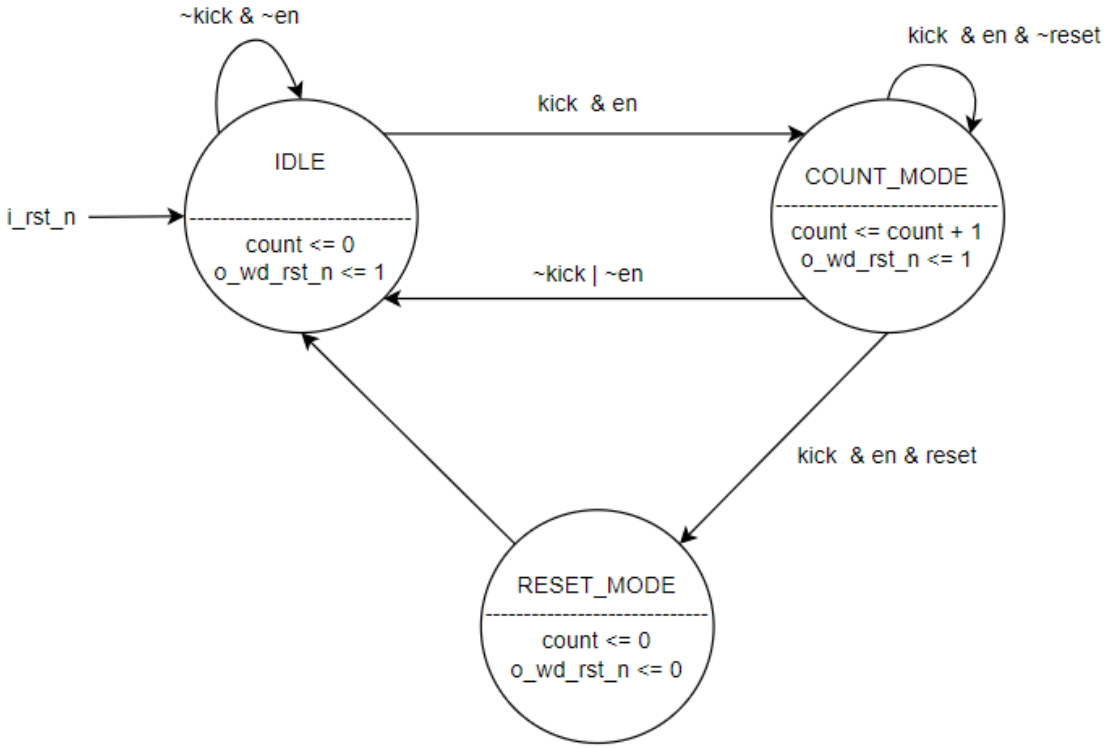

Figure 9: FSM of WDT design

5. FPGA Implement:

5.1. Stopwatch using seven-segment LEDs as the display

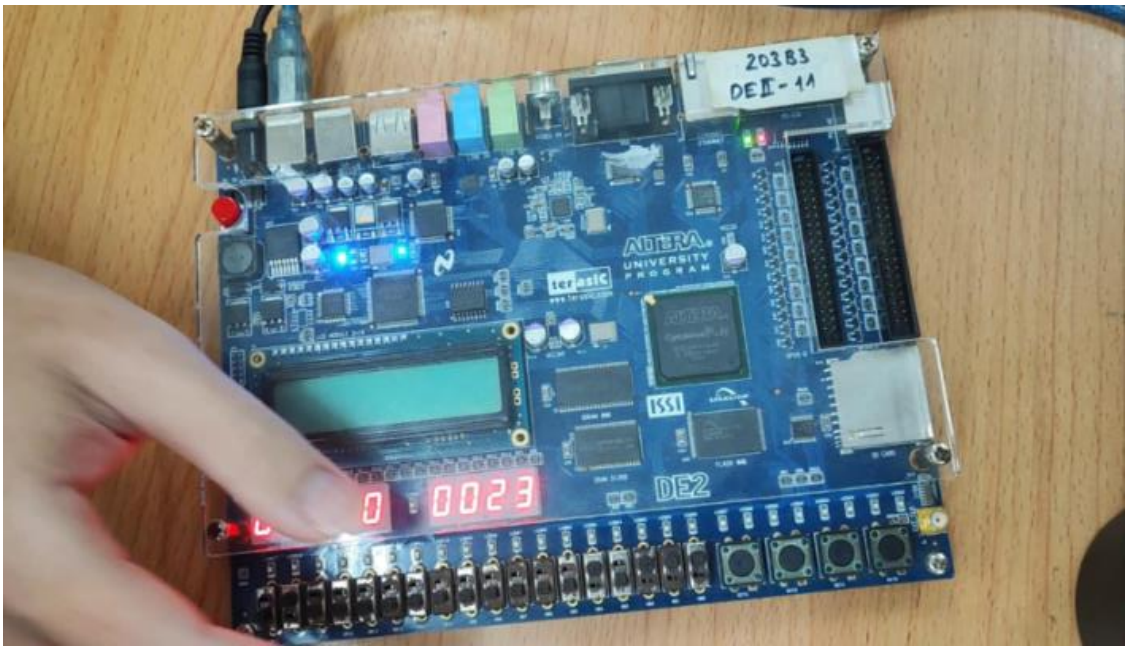

Figure 9: Stopwatch using seven-segment LEDs as the display

5.2. Display multiline text on the LCD

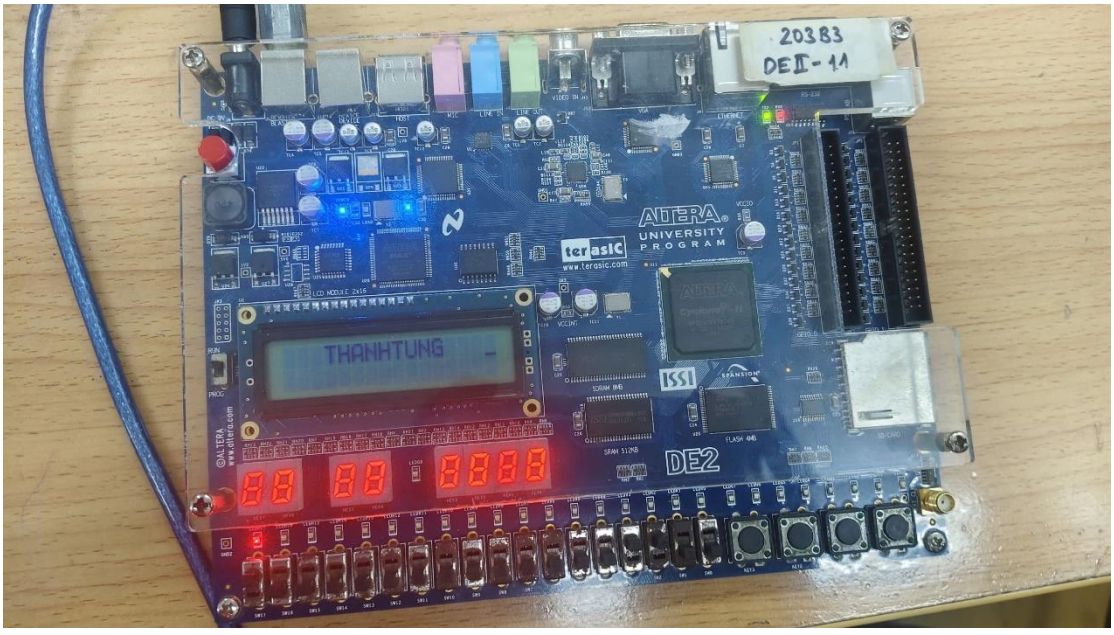

Figure 10: Text on the LCD

6. Evaluation

TASK		EVALUATION
Baseline Functionality	Arithmatic Logic Unit	Done
	Branch Comparison Unit	Done
	Register File	Done
	Instruction Memory (8KiB)	Done
	Input Buffer	Done
	Output Buffer	Done
	Sram (8KiB)	Done
	Control Unit	Done
	Immediate Generator	Done
Alternative Design	Watch Dog Timer	
Verification	Arithmatic Logic Unit	Done
	Branch Comparison Unit	Done
	Register File	Done
	Instruction Memory (8KiB)	Done
	Input Buffer	Done
	Output Buffer	Done
	Sram (using Data Memory 8KiB as an alternative)	Done
	Control Unit	Done
	Immediate Generator	Done
Hardware Implementation Program	Stopwatch using seven-segment LEDs as the display.	Done
	Display multiline text on the LCD	Done

7. Result

7.1. Grand test result:

```
Loading snapshot worklib.tbench:sv ..................... Done
xcelium> source /tools/cadence/XCELIUM2309/tools/xcelium/files/xmsimrc
xcelium> run
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.......PASSED
[   218600]::31::BEQ..............PASSED
[   220800]::32::BNE..............PASSED
[   223000]::33::BLT..............PASSED
[   227000]::34::BGE..............PASSED
[   231000]::35::BLTU.............PASSED
[   235000]::36::BGEU.............PASSED
[   239000]::37::JAL..............PASSED
[   242000]::38::JALR.............PASSED
[   245000]::39::illegal_insn.....PASSED

Timeout...

DUT is considered       P A S S E D

Simulation complete via $finish(1) at time 50 US + 0
../01_bench/tlib.svh:22             $finish;
```

Figure 11: xrun log file

7.2. Quartus

Logic Utilization/Total logic elements	5037/33.216	15.16%

References.

Patterson, L. .COMPUTER ORGANIZATION AND DESGIN THE

HARDWARE/SOFTWARE INTERFACE RISC-V EDITON, Page 243.