ĐẠI HỌC QUỐC GIA TP. HCM
TRƯỜNG ĐẠI HỌC BÁCH KHOA

ADVANCED COMPUTER ARCHITECTURE (EE5207)

MILESTONE 2: A SINGLE-CYCLE PROCESSOR

HK251

Instructor:

Trần Hoàng Linh

Students:

Trương Đức Duy - 2570181

Phạm Hồng Hiệp - 2570193

CONTENTS

LIST OF FIGURE

LIST OF TABLE

1. INTRODUCTION

This project presents the design and implementation of a complete Single-Cycle RISC-V RV32I Processor, developed at the Register-Transfer Level (RTL) using Verilog. The processor follows a straightforward single-cycle architecture in which every instruction—whether arithmetic, memory, or control-flow—completes its execution within a single clock cycle. This design choice simplifies the coordination between control logic and datapath modules while providing clear visibility into how each instruction propagates through the processor.

The top-level module integrates all major functional components required by the RV32I instruction set. The Program Counter (PC), together with a dedicated incrementer module, is responsible for instruction sequencing. In this design, the PC is enhanced beyond a simple sequential updater and incorporates a stall mechanism to support synchronous memory behavior within the Load-Store Unit (LSU). This allows the processor to correctly handle memory operations that require an additional cycle to produce valid read data.

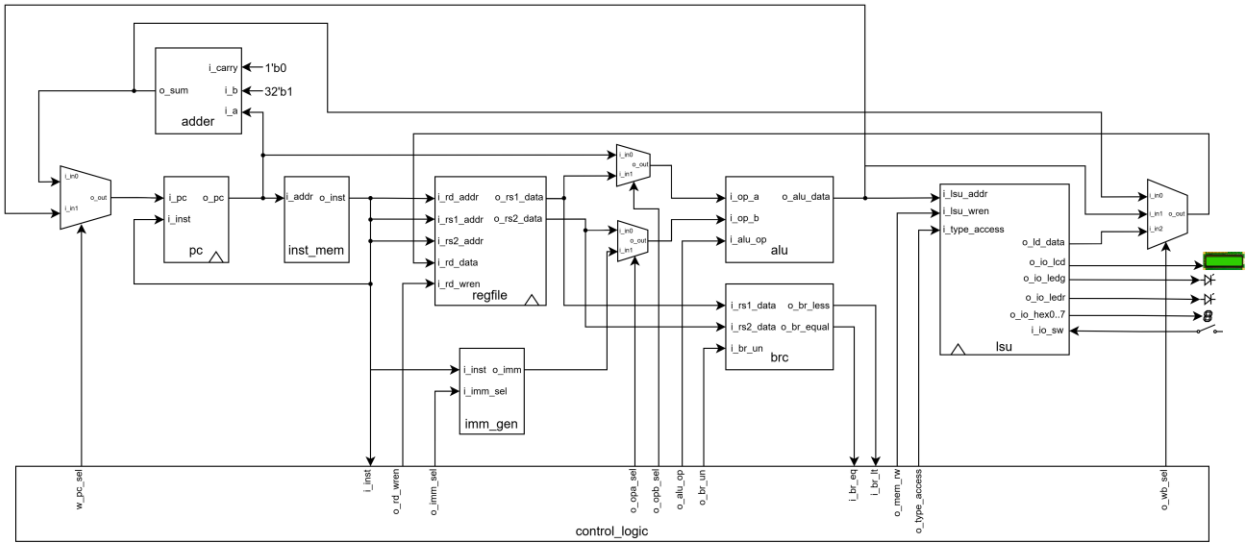
Central to the datapath is the Register File, consisting of thirty-two 32-bit general-purpose registers, with x0 permanently fixed to zero in accordance with the RISC-V specification. Immediate values for different instruction formats are generated by the Immediate Generator, while the correct operands are selected through multiplexers that feed the Arithmetic Logic Unit (ALU).

Branch decision-making is handled by the Branch Comparison Unit, which performs signed and unsigned comparisons to determine whether the PC should advance sequentially or jump to a branch target. Data memory operations are carried out by the Load-Store Unit (LSU), which also manages communication with several external peripherals, including switches, LEDs, an LCD module, and eight seven-segment displays.

Instruction decoding and overall signal coordination are performed by the Control Logic Unit, which generates the required control signals such as register write-enable, ALU

operation codes, branch control, memory access modes, and write-back selection. A multi-input write-back multiplexer chooses between ALU results, loaded data, or the PC+4 value to update the destination register.

Together, these modules form a fully functional RV32I single-cycle processor capable of executing machine instructions, handling memory and I/O operations, and producing real-time output through its peripheral interfaces. This design provides a complete and modular foundation for understanding processor architecture, datapath organization, synchronous memory integration, and embedded system development.



2. DESIGN STRATEGY

2.1. ALU

The Arithmetic Logic Unit (ALU) is a central component of the processor datapath, responsible for performing all arithmetic and logical operations required by the RV32I instruction set. The design strategy for the ALU focuses on modularity, clarity, and hardware efficiency, ensuring that each supported operation is implemented using well-defined submodules. This modular approach simplifies testing, promotes reuse, and improves overall maintainability of the processor design.

The ALU accepts two 32-bit operands (i_op_a and i_op_b) and a 4-bit control signal (i_alu_op) that specifies the operation to be performed. To support the full range of RV32I ALU instructions, the design decomposes each operation into a dedicated hardware block. For arithmetic operations, the ALU employs two separate 32-bit ripple-carry adders: one for addition and another configured for subtraction using two's complement negation of the input operand. This approach ensures consistent performance and eliminates the need for a single multifunction adder with added control complexity.

Comparison operations (SLT and SLTU) are implemented using specialized comparator modules that generate a 1-bit output indicating whether the comparison condition is met. These results are then zero-extended to 32 bits to match the architectural width.

Logical operations such as XOR, OR, and AND are expressed using corresponding gate-level modules that process the operands bit-by-bit. Shift operations (SLL, SRL, and SRA) are performed using dedicated 32-bit shifters that use the lower 5 bits of the second operand as the shift amount, aligning with RISC-V encoding rules.

At the core of the selection mechanism is a 16-to-1 multiplexer, which chooses the correct operation result based on the received ALU operation code. The mapping between i_alu_op and the selected output corresponds directly to the func7 and func3 fields defined in the RV32I instruction format. This clean separation of compute modules and operation

selection ensures that the ALU remains easy to extend and debug, while providing deterministic output for all supported functions.

2.1.1. Full adder 1 bits

The 1-bit full adder is the fundamental building block for multi-bit arithmetic operations in the processor. Its design follows the classical Boolean logic implementation, using minimal combinational logic to compute both the sum bit and the carry-out.

The module takes three single-bit inputs—two operand bits and a carry-in—and produces a sum bit and a carry-out. The sum output is generated using a three-input XOR expression, ensuring correct handling of binary addition across all input combinations. The carry-out logic is derived from the standard majority function of three inputs, implemented using AND, OR and XOR gates. This expression ensures that a carry is produced whenever at least two of the inputs are high.

Table 1. 1 bit full adder truth table

INPUT			**OUTPUT**	
i_a	**i_b**	**i_carry**	**o_sum**	**o_carry**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\begin{cases} o_{sum} = i_a \oplus i_b \oplus i_{carry} \\ o_{carry} = i_a i_b + i_{carry}\,(i_a \oplus i_b) \end{cases}$$

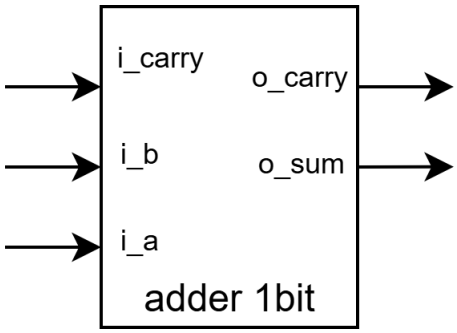


Figure 1: Adder 1 bit module

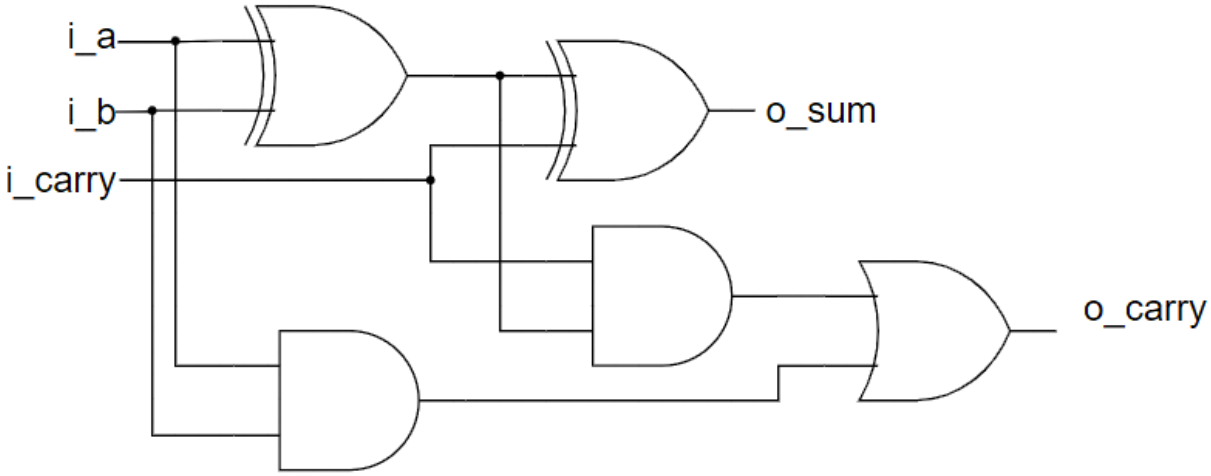


Figure 2. Full Adder Logic Circuit

2.1.2. Full adder 32 bits

The 32-bit full adder serves as a fundamental arithmetic building block within the processor, providing support for both addition and subtraction operations used by the ALU.

The full adder is constructed using a ripple-carry architecture, which chains together 32 1-bit full-adder stages. This approach allows each stage to compute its sum bit while passing a carry signal to the next stage.

The design uses a carry chain wire array to hold intermediate carry values between bit positions. The least significant bit uses the external carry-in (i_carry), enabling the module to support multiword addition or two's complement subtraction when paired with inverted input and an initial carry of 1. The most significant bit outputs the final carry-out,

which can be used by higher-level modules to detect overflow or propagate carries into extended arithmetic units.

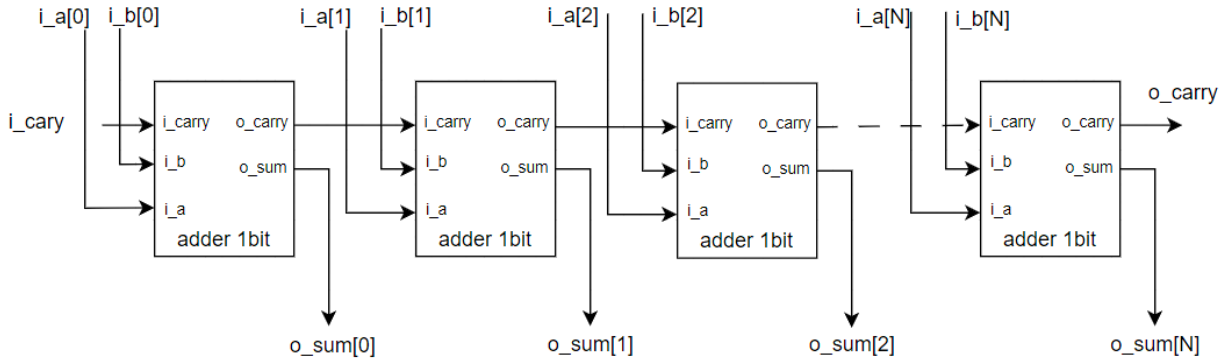


Figure 3. 32-bit ripple carry adder

2.1.3. Signed less-than

The signed less-than (SLT) comparator is designed to evaluate whether one 32-bit operand is smaller than another when interpreted as signed two's complement numbers. The strategy focuses on achieving correctness across all cases—including positive/negative combinations—while reusing existing arithmetic components to maintain hardware efficiency and architectural consistency.

The SLT operation is implemented using a subtract-based comparison method, where the circuit computes A - B and then inspects the sign of the result. This allows the module to leverage the existing 32-bit full adder used elsewhere in the datapath, eliminating the need for a separate comparator structure and ensuring uniform timing behavior.

However, because subtraction alone does not correctly handle all signed cases (particularly when operands have different signs), the design incorporates explicit sign analysis. First, the module checks whether the most significant bits of i_a and i_b differ, indicating opposite signs. When the signs differ, the comparison result is determined solely by the sign of operand A, since any negative value is always less than any positive value in signed interpretation.

When both operands share the same sign, the result of the subtraction (A - B) correctly reflects the comparison, and the module simply returns the sign bit of the subtraction output. This dual-path logic ensures full correctness for all signed combinations while maintaining a compact and efficient implementation.

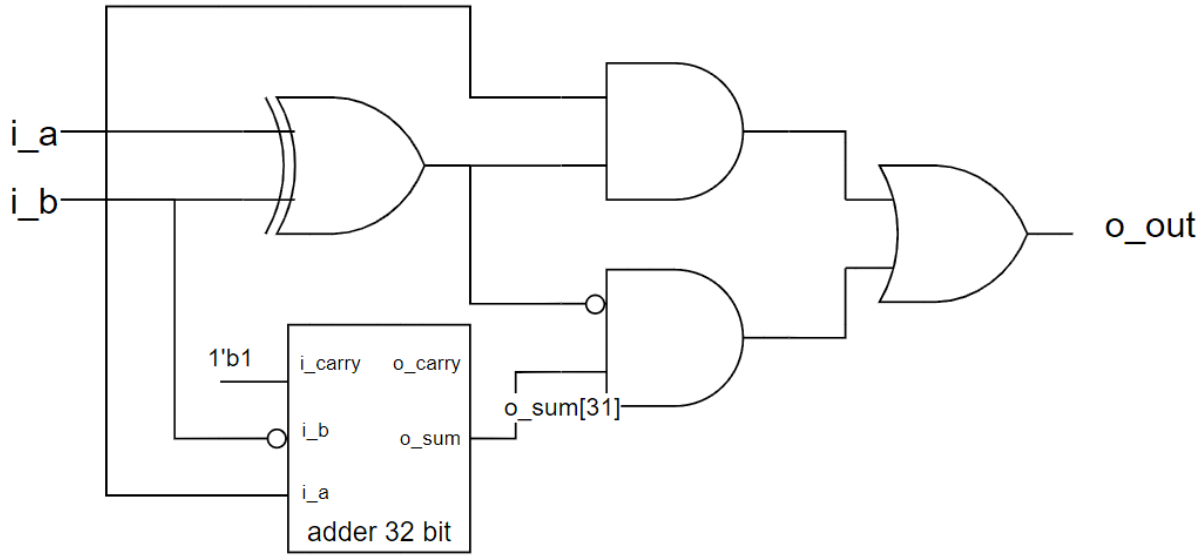


Figure 4. Less than logic circuit

2.1.4. Unsigned Less-Than

The unsigned less-than (SLTU) comparator is responsible for evaluating whether one 32-bit operand is smaller than another when treated as unsigned integers. The design leverages a subtraction-based approach, which provides a simple and hardware-efficient way to detect unsigned ordering by reusing the existing 32-bit full adder module.

In unsigned arithmetic, the key indicator of a less-than relationship is the presence of a borrow during subtraction. When computing A - B using two's complement addition (A + ~B + 1), the carry-out bit generated by the adder reflects whether a borrow occurred:

If the carry-out is 1, no borrow occurred → $A \geq B$.

If the carry-out is 0, a borrow occurred → $A < B$.

This makes the carry-out signal an ideal and compact mechanism for detecting unsigned comparisons. The design therefore performs a full 32-bit subtraction but does not use the resulting difference; instead, it only inspects the adder's carry-out bit. The final comparison result is simply the logical negation of the carry-out.

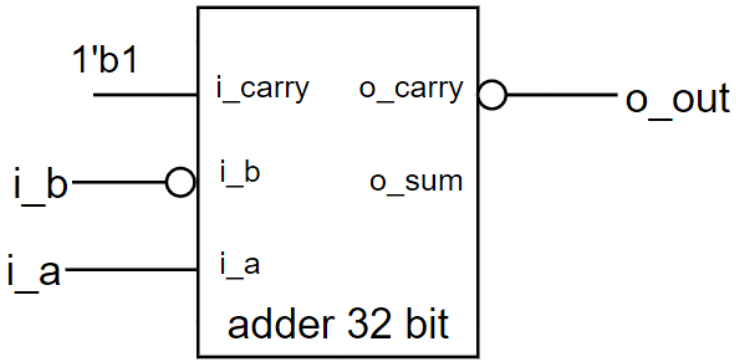


Figure 5. Less than unsigned circuit

2.1.5. 32-Bit Equality Comparator

The equality comparator is designed to determine whether two 32-bit values are identical. To achieve both high efficiency and minimal hardware complexity, the design utilizes a bitwise XOR followed by a reduction NOR operation. This approach avoids multi-level comparison logic and instead relies on the fundamental property that two bits are equal if and only if their XOR result is zero.

The module first computes the bitwise XOR of the two input operands. For each bit position, the XOR output becomes 1 when the bits differ and 0 when they match. By performing a NOR reduction on the resulting 32-bit vector, the module produces a single-bit output that indicates whether all bits are equal. If every XOR output is zero, the reduction NOR evaluates to 1, signaling equality. If any bit differs, the NOR output becomes 0.

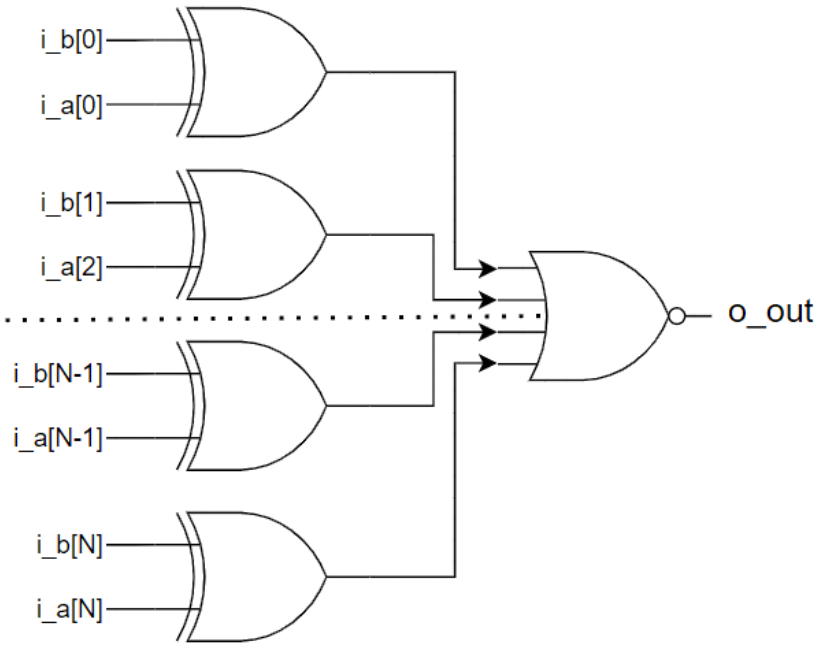


Figure 6. 32-Bit Equality Comparator circuit

2.1.6. Shift Left, Right and Arithmetic Logical

The shift unit integrates three types of 32-bit variable shift operations—Shift Left Logical (SLL), Shift Right Logical (SRL), and Shift Right Arithmetic (SRA)—using a unified barrel shifter architecture. This design enables any shift amount from 0 to 31 bits to be performed efficiently in a single combinational step.

The architecture consists of five cascaded stages of 2-to-1 multiplexers. Each stage corresponds to one bit of the 5-bit shift amount input (i_b), with each bit representing a power-of-two shift:

- i_b[0] selects a shift by 1 bit
- i_b[1] selects a shift by 2 bits
- i_b[2] selects a shift by 4 bits
- i_b[3] selects a shift by 8 bits
- i_b[4] selects a shift by 16 bits

At every stage, the multiplexer chooses between the intermediate value from the previous stage (no shift) or the value shifted by that stage's corresponding amount. By combining these conditional shifts, the barrel shifter can generate any required shift amount as the sum of these powers of two. This structure avoids the need for sequential shifting and ensures minimal propagation delay.

Although all three shift operations share the same staged multiplexer architecture, they differ in how they treat the bits introduced during the shift:

- Shift Left Logical (SLL): The operand is shifted left, and zeros are inserted into the least significant bits. This is appropriate for logical and unsigned operations.
- Shift Right Logical (SRL): The operand is shifted right, and zeros are inserted into the most significant bits, maintaining unsigned behavior.
- Shift Right Arithmetic (SRA): The operand is shifted right while preserving the sign bit. The most significant bit (MSB) is replicated into the vacated upper bits, ensuring correct behavior for signed, two's-complement values.

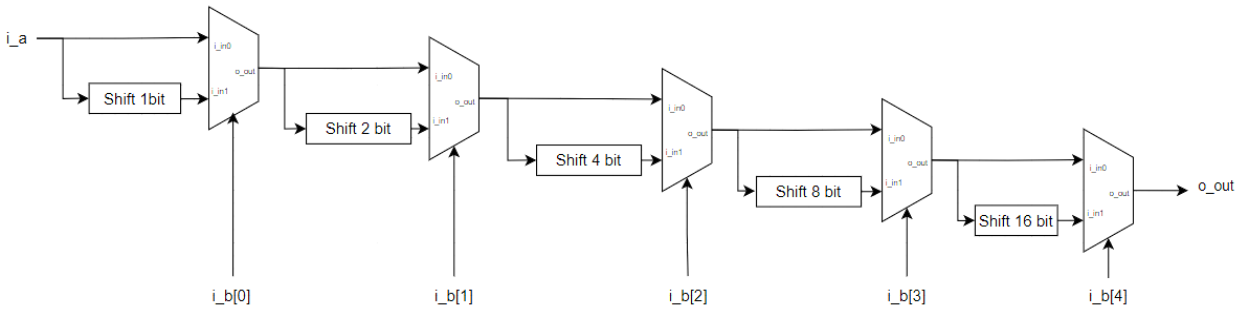


Figure 7. Shift logical Circuit

2.2. Branch Comparision

The branch comparator (BRC) is designed to provide the comparison results required for RISC-V branch instructions by evaluating equality and less-than conditions between the two source operands, rs1 and rs2.

To support the full RV32I branch set, the module implements both signed and unsigned comparisons using dedicated slt_32bit and sltu_32bit submodules,

A 2-to-1 multiplexer selects between these two results based on the control signal i_br_un, allowing the Execute stage to choose signed behavior for BLT/BGE or unsigned behavior for BLTU/BGEU without modifying the datapath. Equality checking is performed separately through the equal_32bit module, producing the o_br_equal signal used by BEQ and BNE.

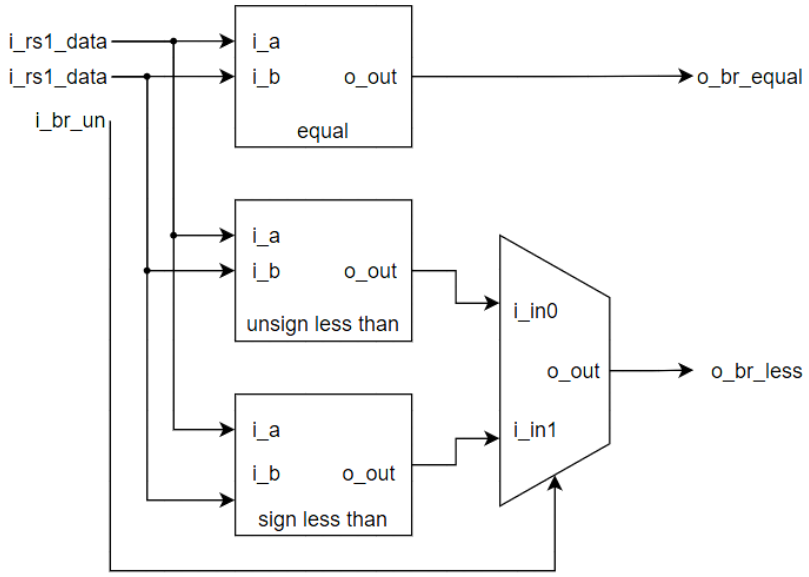
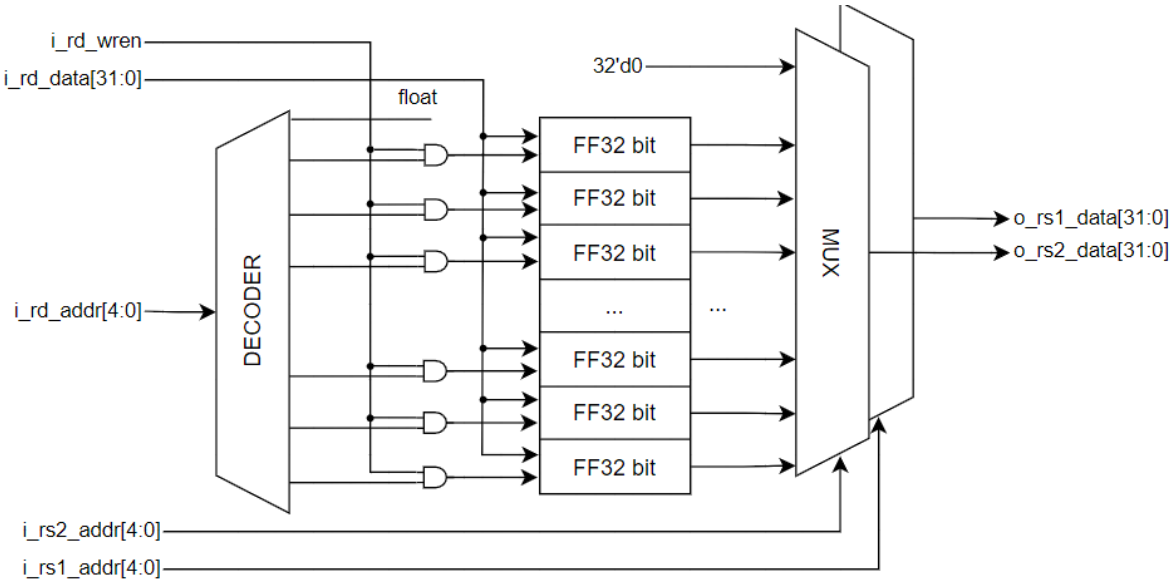


Figure 8. Branch comparator circuit

2.3. Register file

The register file needed to support two independent, asynchronous read ports and a single synchronous write port, while maintaining strict functional requirements such as preserving the fixed-zero behavior of register x0. The overall strategy focused on dividing the design into clear hierarchical components, ensuring easy verification, and preserving scalability for future modifications or expansions.



The register file must read from two registers and write to one register in the same cycle. Because this is a common operation in processor datapaths, the read ports were designed as purely combinational to allow immediate availability of operands without waiting for a clock edge.

Instead of individually comparing the write address to all register indices, a binary-to-one-hot converter produces a 32-bit one-hot vector based on the 5-bit write address. This simplifies the decoder logic and synthesizes efficiently in hardware. The implementation uses a left-shift operation where a single “1” is shifted by the value of the address, resulting in the correct one-hot pattern. This approach not only reduces logic complexity but also ensures deterministic behavior and straightforward timing. The one-hot signal is then ANDed with the global write-enable.

Each of the 32 registers is constructed using parameterized 32-bit D-type flip-flops. The design uses a generate loop to instantiate 31 of these registers, explicitly excluding register 0. On reset, all writable registers are cleared to zero, maintaining a predictable initial system state.

For the read ports, a hierarchical multiplexer tree was designed to select among the 32 registers. Instead of writing a massive 32-to-1 multiplexer directly, the design uses a

structured approach based on cascading 2-to-1 multiplexers. The selection process occurs in stages: first 32 inputs are reduced to 16, then 16 to 8, then 8 to 4, then 4 to 2, and finally 2 to 1.

2.4. Immediate Generator

The immediate generator is a vital component in the design of a RISC-V CPU, responsible for extracting and arranging specific instruction bits to produce the correct immediate values. In the RISC-V ISA, several immediate formats exist—namely I, S, B, J, and U types—each requiring distinct wiring patterns. Multiplexers are therefore used to route the instruction bits into the appropriate immediate structure. The accompanying figure illustrates the design approach used to guide the immediate bus across these formats.

2.5. Control Unit

The control logic module was designed with the primary goal of translating a 32-bit RISC-V instruction into the full set of internal control signals required to operate the processor datapath.

Table 2. Control unit truth table

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
(R-R Op)	*	*	+4	*	*	Reg	Reg	*(Op)*	Read	1	ALU
addi	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
beq	0	*	+4	B	*	PC	Imm	Add	Read	0	*
beq	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	1	*	+4	B	*	PC	Imm	Add	Read	0	*
blt	*	1	ALU	B	0	PC	Imm	Add	Read	0	*
bltu	*	1	ALU	B	1	PC	Imm	Add	Read	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4
auipc	*	*	+4	U	*	PC	Imm	Add	Read	1	ALU

The first step in the design was to extract the fields of interest from the instruction word—namely the opcode, funct3, and funct7 bits. These fields determine the instruction category, the source of immediate data, and the specific ALU operation required. To simplify later logic, the opcode bits are condensed into a 5-bit value, and explicit Boolean expressions are used to identify each major RISC-V instruction type: R-type, I-type, load, store, branch, JAL, JALR, LUI, and AUIPC.

Instruction Decoder

The control unit extracts funct3, funct7, and the opcode from the instruction. Using the opcode, it identifies the instruction type: R-type, I-type, Load, Store, Branch, JAL, JALR, LUI, and AUIPC. The InsnVld signal is asserted when the instruction belongs to any supported instruction class.

PCSel

PCSel selects the next program counter. It is set to 0 for normal sequential execution (PC + 4) and set to 1 for control-flow instructions. PCSel becomes 1 for JAL, JALR, and for Branch instructions whose conditions are satisfied, determined using BrEq and BrLT according to funct3.

ImmSel

ImmSel is a five-bit one-hot signal that selects the correct immediate format. Bit 4 corresponds to U-type (LUI, AUIPC), bit 3 to J-type (JAL), bit 2 to B-type (Branch), bit 1 to S-type (Store), and bit 0 to I-type (Load, I-type arithmetic, and JALR).

BrUn

BrUn determines whether branch comparison uses signed or unsigned values. It is set to 1 only for BLTU and BGEU, based on the funct3 field.

OPASel and OPBSel

OPASel selects the ALU's first operand. It chooses the PC for Branch, Jump, and AUIPC instructions and a register value for others. OPBSel selects the second ALU

operand. It is set to Immediate for all instructions except R-type, which use two register operands.

ALUOp

The ALU operation is determined from funct7 and funct3 for R-type instructions, or from funct3 for I-type. Shift-right immediate instructions reuse the R-type ALU code to support both SRLI and SRAI. LUI forces the ALU control to a special constant (1111), while all other instructions default to an ADD operation.

RdWren

RdWren enables register writes. It is asserted for all instructions except Store and Branch instructions, since these do not have a destination register.

MemRW

MemRW selects memory read or write. It is set to 1 only during Store instructions and 0 for all others.

WBSel

WBSel selects the value written back to the register file. The encoding is:

00 for memory data (Load instructions),

01 for ALU results (R-type, I-type, LUI, AUIPC),

10 for PC + 4 (JAL, JALR).

TypeAccess

TypeAccess outputs funct3 to specify the memory access type (word, half word, byte) for Load and Store instructions. For all other instructions, it outputs 000.

2.6. Memory Modules

Unlike the Von Neumann architecture used in x86 processors—where a single memory space stores both program instructions and data—RISC and ARM processors typically adopt the Harvard architecture. In this model, memory is divided into two distinct units: IMEM (instruction memory) for storing program instructions and DMEM (data memory) for storing data.

2.6.1. Instruction memory

IMEM operates similarly to a ROM, providing combinational, read-only access to instruction data.

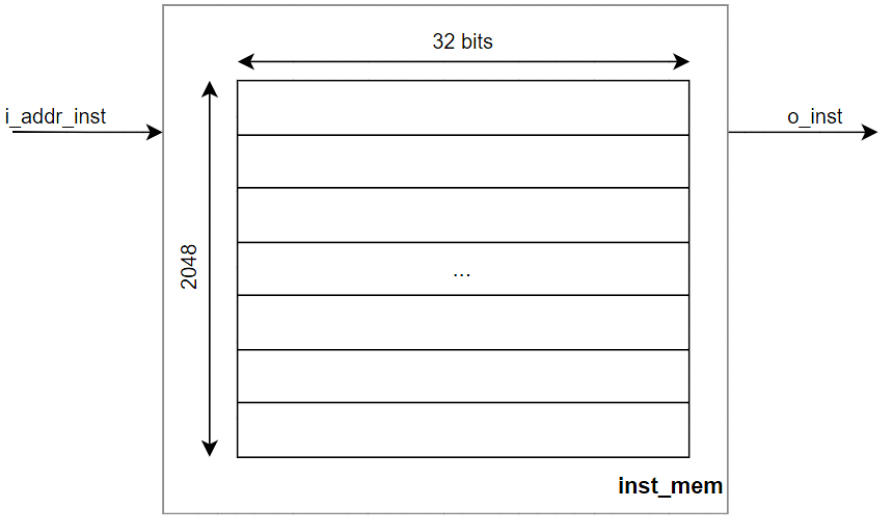


Figure 9, Instruction memory diagram

2.6.2. Data memory using BRAM infer supporting handling misaligned

The data memory is implemented using four Block RAM (BRAM) modules, with each BRAM instance storing one byte per addressable location. Implementing a memory of this size using distributed registers or LUT-based arrays would require thousands of flip-flops, significantly increasing logic utilization. This not only wastes general-purpose FPGA resources but also slows down synthesis and place-and-route. By contrast, BRAM

provides dedicated on-chip storage optimized for large memories, enabling a more compact and efficient implementation while reducing compilation time.

The BRAM blocks operate with synchronous read and write ports, meaning that read data becomes valid only on the next clock edge. Consequently, each load instruction introduces a one-cycle delay to retrieve the correct data. This is managed by stalling the pipeline and adjusting the Program Counter appropriately to ensure correct instruction sequencing.

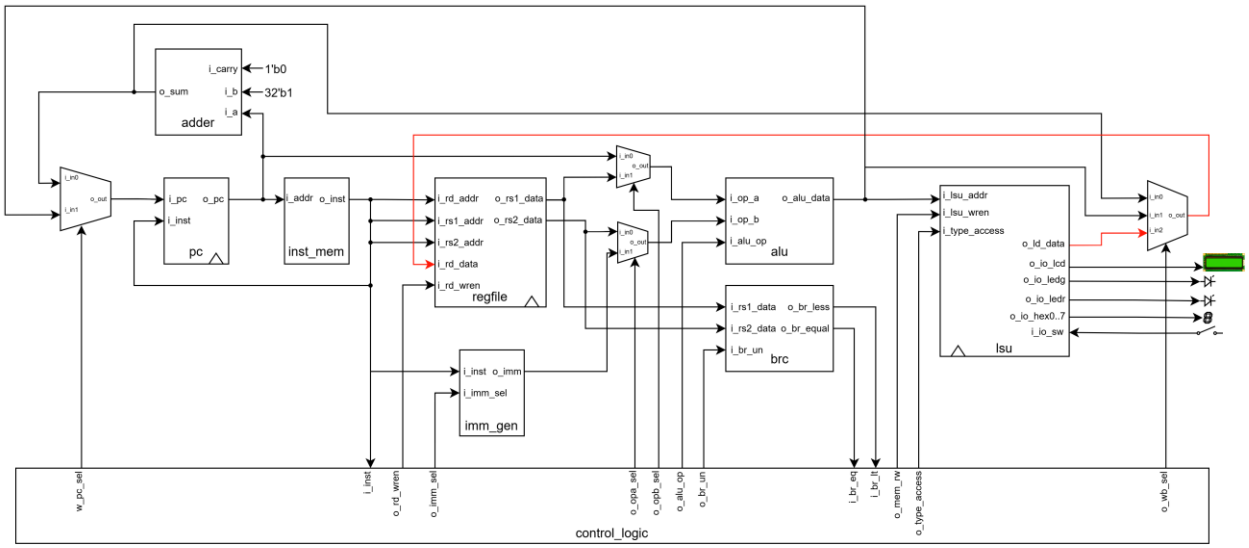


Figure 10. Due to sychronous read LSU, we need 2 clock to load data into regfile

Although the RISC-V processor uses a 32-bit data interface, the memory is byte-addressable and must support operations on bytes, half-words, and full words—including misaligned accesses. To meet these requirements, the 32-bit memory word is decomposed into four independent 8-bit RAM columns. Each column holds one byte of every word, allowing fine-grained control over which bytes are written or read.

Write operations use a rotation-based mechanism. The 32-bit input data passes through a 4-to-1 multiplexer that rotates the bytes according to the lower address bits (i_addr[1:0]). This aligns the target byte(s) into the least significant positions before being distributed to the four RAM columns. The byte-mask input (i_bmask) is rotated using the same alignment logic, ensuring that only the intended bytes are written. Each RAM column

receives its own write-enable signal derived from the rotated mask, preventing unintended modification of adjacent bytes.

This architecture naturally supports SB, SH, and SW store instructions, preserves unmodified bytes within a word, and provides correct behavior for misaligned memory accesses—all while maintaining efficient BRAM utilization and synthesizable performance on FPGA hardware.

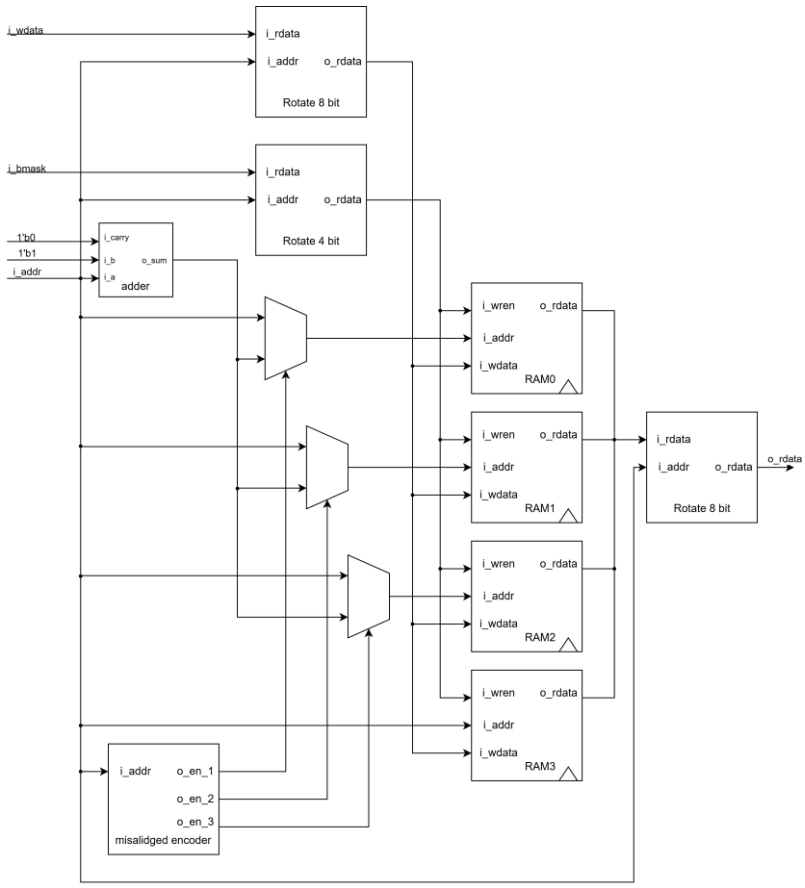


Figure 11. Data memory circuit

2.7. Program Counter with stalling whenver load instruction

The pc module implements the program counter for a RISC-V processor. It keeps track of the current instruction address and updates it every clock cycle. The PC is synchronous with the clock and can be reset asynchronously using i_reset.

To handle memory stalls caused by synchronous BRAM reads, the module detects load instructions by checking the opcode. When a load instruction is executed, the data from memory is not immediately available, so the PC must hold its value for one extra clock cycle. This prevents the processor from fetching the next instruction too early.

The stall mechanism uses two multiplexers: one to select whether the PC should advance or hold, and one to update an internal stall flag. The PC only advances when no stall is needed, ensuring that the instruction sequence remains correct.

This simple mechanism allows the processor to handle one-cycle memory stalls caused by synchronous BRAM reads, keeping the design correct and easy to implement.

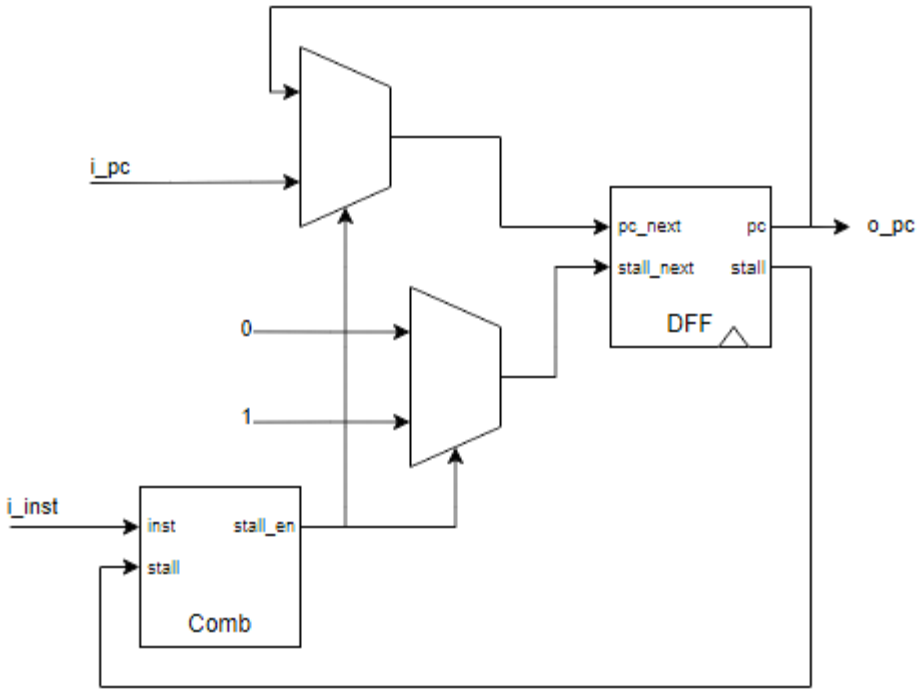


Figure 12. PC with holding one more clock whenever load instruction

2.8. Load-Store Unit

A Load-Store Unit (LSU) is an essential component of a processor's architecture, particularly within the execution pipeline. It plays a central role in systems built on the Load-Store model, such as processors implementing Reduced Instruction Set Computer

(RISC) architectures. The LSU is responsible for managing memory-related operations, including loading data from memory, storing data back into memory, and performing basic arithmetic or logical processing on data retrieved from memory.

The Load-Store Unit (LSU) manages all memory and peripheral accesses for the CPU. It takes a 32-bit address from the processor and decides whether the access targets internal RAM or one of the connected peripherals, such as LEDs, switches, LCDs, or seven-segment displays. This decision is handled by the memory_encoder module, which generates enable signals for the correct device and calculates the internal memory or peripheral address.

For store operations, the LSU uses the store_control module to generate a byte mask according to the type of access—byte, half-word, or word. Data is then written to the selected memory or peripheral device. Peripheral outputs are stored in registers to ensure stable signals, so LEDs, displays, and the LCD reflect the latest written values.

For load operations, the LSU collects data from the selected source—internal memory or a peripheral—and formats it according to the load type. The load_control module handles signed and unsigned extensions for byte and half-word loads, ensuring the CPU receives correctly sized data. Multiplexers are used to select the appropriate output among all possible memory and peripheral sources.

Overall, the LSU provides a simple, modular interface between the CPU, memory, and peripherals. It supports all standard RISC-V memory instructions, maintains peripheral synchronization, and guarantees correct data routing and formatting for both loads and stores.

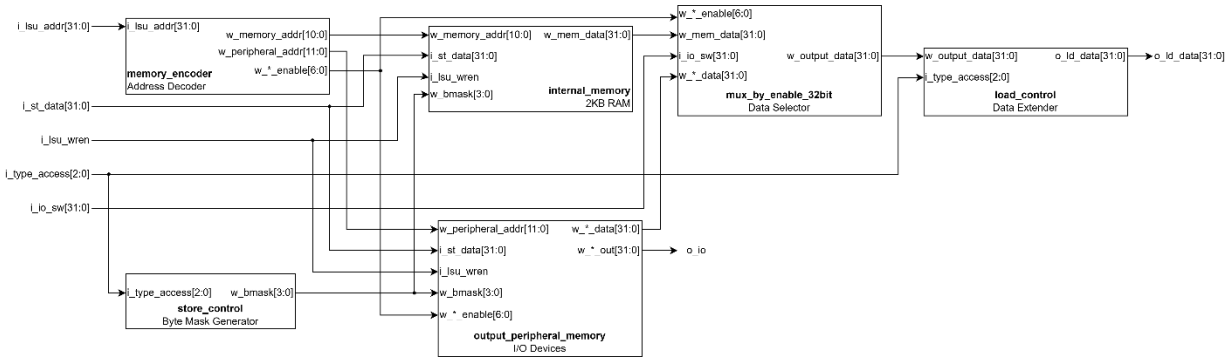


Figure 13. LSU diagram

3. VERIFICATION STRATEGY

3.1. Sub block verification

3.1.1. ALU

The ALU testbench serves the purpose of validating the functionality of the ALU module by applying a wide range of predefined input combinations. Instead of using random values, the testbench systematically assigns specific operand values and ALU operation codes to verify every supported operation, including arithmetic, logical, comparison, and shift instructions.

For each test case, the expected result is computed directly within the testbench and displayed alongside the ALU's actual output, allowing easy comparison. This step-by-step approach ensures that edge cases—such as overflows, negative numbers, large shift amounts, and signed versus unsigned comparisons—are thoroughly evaluated. Based on the displayed outputs, the ALU's behavior can be confirmed for correctness, ensuring reliable operation across all tested instructions.

```
# ADD:          0 +          0 =          0 (Result=         0)
# ADD: 80000000 + 80000000 = 00000000 (Result=00000000)
# SUB:         10 -          5 =          5 (Result=         5)
# SUB: ffffffff - 00000001 = fffffffe (Result=fffffffe)
# SLT: 4294967291 <          5 ? Result=         1 (Expected=1)
# SLTU: aaaaaaaa < 55555555 ? Result=         0 (Expected=0)
# SLL: 00000001 <<  1 = 00000002 (Result=00000002)
# SRL: 00000000 >>  5 = 00000000 (Result=00000000)
# SRA: 4294967280 >>>  2 =          -4 (Result=         -4)
```

3.1.2. BRC

The BRC testbench is responsible for validating the branch comparison module by applying a structured set of test inputs for both unsigned and signed comparisons. It systematically assigns different operand values to the inputs and toggles the control signal to switch between unsigned and signed evaluation. For every test case, the expected results

for “less than” and “equal” conditions are calculated directly within the testbench and compared against the module’s actual outputs. This ensures that boundary values, extreme signed numbers, and typical comparison scenarios are all thoroughly exercised.

The displayed results allow clear verification of correctness, confirming that the branch comparison module behaves as intended across all tested conditions.

```
# Unsigned: 0 < 0 ? less=actual:0 expected:0, equal=actual:1 expected:1
# Unsigned: 1 < 0 ? less=actual:0 expected:0, equal=actual:0 expected:0
# Unsigned: 0 < 1 ? less=actual:1 expected:1, equal=actual:0 expected:0
# Unsigned: 4294967295 < 0 ? less=actual:1 expected:0, equal=actual:0 expected:0
# Unsigned: 0 < 4294967295 ? less=actual:0 expected:1, equal=actual:0 expected:0
# Unsigned: 2147483647 < 2147483647 ? less=actual:0 expected:0, equal=actual:1 expected:1
# Unsigned: 2147483647 < 2147483648 ? less=actual:0 expected:1, equal=actual:0 expected:0
# Unsigned: 4294967295 < 4294967295 ? less=actual:0 expected:0, equal=actual:1 expected:1
# Signed: 0 < 0 ? less=actual:0 expected:0, equal=actual:1 expected:1
# Signed: -1 < 0 ? less=actual:0 expected:1, equal=actual:0 expected:0
# Signed: 0 < -2147483648 ? less=actual:1 expected:0, equal=actual:0 expected:0
# Signed: 123456789 < -123456789 ? less=actual:1 expected:0, equal=actual:0 expected:0
# Signed: -123456789 < 123456789 ? less=actual:0 expected:1, equal=actual:0 expected:0
# Signed: -2147483648 < -1 ? less=actual:1 expected:1, equal=actual:0 expected:0
# Signed: -1 < -2147483648 ? less=actual:0 expected:0, equal=actual:0 expected:0
# Signed: -1000 < -1000 ? less=actual:0 expected:0, equal=actual:1 expected:1
```

3.1.3. Regfile

The register file testbench verifies the functionality of the register file module by applying a sequence of controlled read and write operations. It performs reset initialization, writes values to various registers, and then reads them back to compare the module’s outputs with the expected results. Special behavior such as the constant-zero register (x0), overwriting existing values, and reading without write enable are also tested to ensure correct operation. Each test case checks whether the stored and retrieved data match the

intended values, confirming that the register file correctly handles addressing, write enable control, and data integrity across all tested registers.

```
# Test1: reg0 = 00000000 (Expected=00000000)
# Test2: reg5=deadbeef, reg0=00000000 (Expected=DEADBEEF, 00000000)
# Test3: reg10=12345678, reg5=deadbeef (Expected=12345678, DEADBEEF)
# Test4: reg0=00000000 (Expected=00000000)
# Test5: reg1=ffffffff, reg0=00000000 (Expected=FFFFFFFF, 00000000)
# Test6: reg5=0a0a0a0a, reg10=12345678 (Expected=0A0A0A0A, 12345678)
# Test7: reg31=cafebabe, reg1=ffffffff (Expected=CAFEBABE, FFFFFFFF)
# Test8: reg5=0a0a0a0a, reg31=cafebabe (Expected=0A0A0A0A, CAFEBABE)
# Test9: reg10=00000000, reg0=00000000 (Expected=00000000, 00000000)
# Test10: reg15=55aa55aa, reg5=0a0a0a0a (Expected=55AA55AA, 0A0A0A0A)
```

3.2. Test by program

Before testing our program, we developed a comprehensive testbench to facilitate easy monitoring and verification.

This testbench provides extensive debugging capabilities for the RISC-V processor implementation through several key features.It offers complete visibility into the processor's internal state by displaying all 32 registers in an organized 8x4 format, enabling straightforward monitoring of data manipulation and register file behavior. The memory subsystem is thoroughly tracked by monitoring the first 32 memory locations with byte-level granularity, allowing detailed observation of data storage and memory access patterns.

For system integration testing, the testbench includes comprehensive I/O peripheral monitoring that tracks LED outputs, LCD displays, and all eight seven-segment displays. It incorporates a dedicated decoder function that converts raw seven-segment control signals into human-readable hexadecimal characters. The testbench automatically captures the processor state on every program counter change, providing real-time execution

tracing, and includes robust error handling that immediately terminates simulation upon detecting invalid instructions, ensuring reliable verification outcomes.

Together, these features allow us to easily verify the program execution line by line, instruction by instruction. Below are the results from ModelSim showing the processor state after each instruction execution.

```
# ================================
# ================================
# PC = 00000054, Instruction = 00379a93
# R0 = 00000000 | R1 = 0000000a | R2 = 00000005 | R3 = 00000003
# R4 = ffffffff | R5 = 000000f5 | R6 = fffffffc | R7 = 80000000
# R8 = 00000001 | R9 = 00000000 | R10 = 000000f0 | R11 = 00000007
# R12 = 00000007 | R13 = 000000f5 | R14 = 00000005 | R15 = 00000004
# R16 = 00000001 | R17 = 00000001 | R18 = 0000000b | R19 = 000000f4
# R20 = 00000004 | R21 = 00000000 | R22 = 00000000 | R23 = 00000000
# R24 = 00000000 | R25 = 00000000 | R26 = 00000000 | R27 = 00000000
# R28 = 00000000 | R29 = 00000000 | R30 = 00000000 | R31 = 00000000
# M0 = 00 00 00 00 | M1 = 00 00 00 00 | M2 = 00 00 00 00 | M3 = 00 00 00 00
# M4 = 00 00 00 00 | M5 = 00 00 00 00 | M6 = 00 00 00 00 | M7 = 00 00 00 00
# M8 = 00 00 00 00 | M9 = 00 00 00 00 | M10 = 00 00 00 00 | M11 = 00 00 00 00
# M12 = 00 00 00 00 | M13 = 00 00 00 00 | M14 = 00 00 00 00 | M15 = 00 00 00 00
# M16 = 00 00 00 00 | M17 = 00 00 00 00 | M18 = 00 00 00 00 | M19 = 00 00 00 00
# M20 = 00 00 00 00 | M21 = 00 00 00 00 | M22 = 00 00 00 00 | M23 = 00 00 00 00
# M24 = 00 00 00 00 | M25 = 00 00 00 00 | M26 = 00 00 00 00 | M27 = 00 00 00 00
# M28 = 00 00 00 00 | M29 = 00 00 00 00 | M30 = 00 00 00 00 | M31 = 00 00 00 00
# M32 = 00 00 00 00 | M33 = 00 00 00 00 | M34 = 00 00 00 00 | M35 = 00 00 00 00
# LEDG = 00000000, LEDR = 00000000, LCD = 00000000
# HEX7 = 00, HEX6 = 00, HEX5 = 00, HEX4 = 00, HEX3 = 00, HEX2 = 00, HEX1 = 00, HEX0 = 00
# Hex Digits: 88888888
# ================================
# ================================
# PC = 00000058, Instruction = 0027db13
# R0 = 00000000 | R1 = 0000000a | R2 = 00000005 | R3 = 00000003
# R4 = ffffffff | R5 = 000000f5 | R6 = fffffffc | R7 = 80000000
# R8 = 00000001 | R9 = 00000000 | R10 = 000000f0 | R11 = 00000007
# R12 = 00000007 | R13 = 000000f5 | R14 = 00000005 | R15 = 00000004
```

Figure 14. Result of single cycle test bench by using modelsim

3.2.1. Alu and Regfile instruction test

We validate the ALU and register file by running a series of instructions and tracking the data flow. The testbench provides cycle-accurate visibility into the PC and register states, allowing us to confirm that each ALU result is stored correctly. To guarantee functional correctness, we compare our processor's output against a reference RISC-V model online for verification.

Table 3. List of ALU and Regfile instruction test

PC	Machine Code	Basic Code	Original Code
0x0	0x00A00093	addi x1 x0 10	addi x1, x0, 10
0x4	0x00500113	addi x2 x0 5	addi x2, x0, 5
0x8	0x00300193	addi x3 x0 3	addi x3, x0, 3
0xc	0xFFF00213	addi x4 x0 -1	addi x4, x0, -1
0x10	0x0F000293	addi x5 x0 240	addi x5, x0, 240
0x14	0x002282B3	add x5 x5 x2	add x5, x5, x2
0x18	0x40320333	sub x6 x4 x3	sub x6, x4, x3
0x1c	0x004193B3	sll x7 x3 x4	sll x7, x3, x4
0x20	0x0051A433	slt x8 x3 x5	slt x8, x3, x5
0x24	0x0022B4B3	sltu x9 x5 x2	sltu x9, x5, x2
0x28	0x0022C533	xor x10 x5 x2	xor x10, x5, x2
0x2c	0x0022D5B3	srl x11 x5 x2	srl x11, x5, x2
0x30	0x4022D633	sra x12 x5 x2	sra x12, x5, x2
0x34	0x0022E6B3	or x13 x5 x2	or x13, x5, x2
0x38	0x0022F733	and x14 x5 x2	and x14, x5, x2
0x3c	0x00520793	addi x15 x4 5	addi x15, x4, 5
0x40	0x00A7A813	slti x16 x15 10	slti x16, x15, 10
0x44	0x00A7B893	sltiu x17 x15 10	sltiu x17, x15, 10
0x48	0x00F7C913	xori x18 x15 15	xori x18, x15, 15
0x4c	0x0F07E993	ori x19 x15 240	ori x19, x15, 240
0x50	0x0077FA13	andi x20 x15 7	andi x20, x15, 7
0x54	0x00379A93	slli x21 x15 3	slli x21, x15, 3
0x58	0x0027DB13	srli x22 x15 2	srli x22, x15, 2
0x5c	0x4017DB93	srai x23 x15 1	srai x23, x15, 1

```
# ==============================
# PC = 00000080, Instruction = 00000000
# R0 = 00000000 | R1 = 0000000a | R2 = 00000005 | R3 = 00000003
# R4 = ffffffff | R5 = 000000f5 | R6 = fffffffc | R7 = 80000000
# R8 = 00000001 | R9 = 00000000 | R10 = 000000f0 | R11 = 00000007
# R12 = 00000007 | R13 = 000000f5 | R14 = 00000005 | R15 = 00000004
# R16 = 00000001 | R17 = 00000001 | R18 = 0000000b | R19 = 000000f4
# R20 = 00000004 | R21 = 00000020 | R22 = 00000001 | R23 = 00000002
# R24 = 00000000 | R25 = 00000000 | R26 = 00000000 | R27 = 00000000
# R28 = 00000000 | R29 = 00000000 | R30 = 00000000 | R31 = 00000000
```

Figure 15. Result of the the end of alu and regfile instruction test

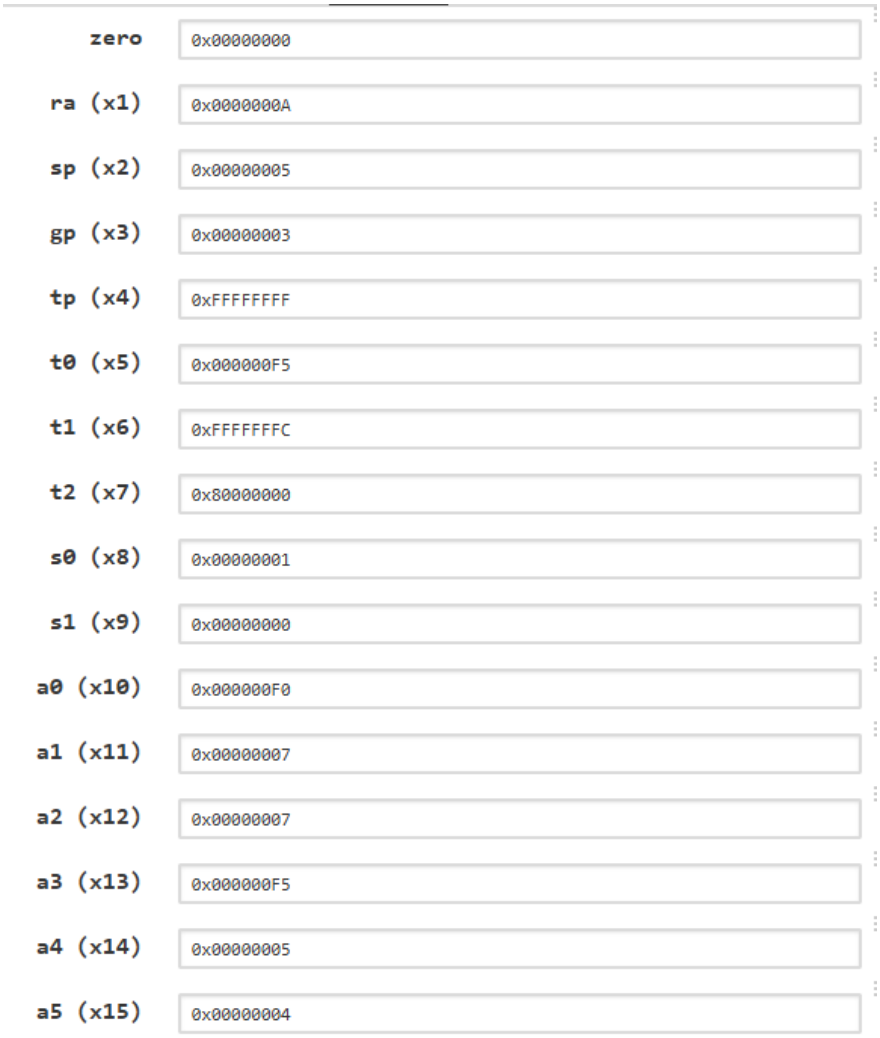


Figure 16. Result of venus after running alu and regfile instruction

3.2.2. Branch and jump instruction test

This test sequence comprehensively validates the processor's branch instruction unit by testing all six branch types with both positive and negative cases.

The test begins by verifying the BEQ (Branch if Equal) instruction, BNE (Branch if Not Equal), signed comparison branches BLT (Branch if Less Than) and BGE (Branch if Greater than or Equal), the unsigned variants BLTU and BGEU

For each test, successful branch execution sets a result register to 0x100 (pass), while failure sets it to 1, providing clear pass/fail indicators for each branch type through the testbench's cycle-accurate monitoring of program flow and register state changes.

Table 4. Branch and jump instruction

PC	Machine Code	Basic Code	Original Code
0x0	0x02A00093	addi x1 x0 42	addi x1, x0, 42
0x4	0x02A00113	addi x2 x0 42	addi x2, x0, 42
0x8	0x00208663	beq x1 x2 12	beq x1, x2, beq_taken
0xc	0x00100693	addi x13 x0 1	addi x13, x0, 1 # Fail
0x10	0x0080006F	jal x0 8	j beq_end
0x14	0x10000693	addi x13 x0 256	addi x13, x0, 0x100 # Pass
0x18	0x03700193	addi x3 x0 55	addi x3, x0, 55
0x1c	0x06300213	addi x4 x0 99	addi x4, x0, 99
0x20	0x00419663	bne x3 x4 12	bne x3, x4, bne_taken
0x24	0x00100713	addi x14 x0 1	addi x14, x0, 1 # Fail
0x28	0x0080006F	jal x0 8	j bne_end
0x2c	0x10000713	addi x14 x0 256	addi x14, x0, 0x100 # Pass
0x30	0xFFF00293	addi x5 x0 -1	addi x5, x0, -1
0x34	0x00100313	addi x6 x0 1	addi x6, x0, 1
0x38	0x0062C663	blt x5 x6 12	blt x5, x6, blt_taken
0x3c	0x00100793	addi x15 x0 1	addi x15, x0, 1 # Fail
0x40	0x0080006F	jal x0 8	j blt_end
0x44	0x10000793	addi x15 x0 256	addi x15, x0, 0x100 # Pass
0x48	0x00A00393	addi x7 x0 10	addi x7, x0, 10
0x4c	0xFFB00413	addi x8 x0 -5	addi x8, x0, -5
0x50	0x0083D663	bge x7 x8 12	bge x7, x8, bge_taken
0x54	0x00100813	addi x16 x0 1	addi x16, x0, 1 # Fail
0x58	0x0080006F	jal x0 8	j bge_end
0x5c	0x10000813	addi x16 x0 256	addi x16, x0, 0x100 # Pass
0x60	0x00100493	addi x9 x0 1	addi x9, x0, 1
0x64	0xFFF00513	addi x10 x0 -1	addi x10, x0, -1
0x68	0x00A4E663	bltu x9 x10 12	bltu x9, x10, bltu_taken
0x6c	0x00100893	addi x17 x0 1	addi x17, x0, 1 # Fail
0x70	0x0080006F	jal x0 8	j bltu_end
0x74	0x10000893	addi x17 x0 256	addi x17, x0, 0x100 # Pass
0x78	0xFFE00593	addi x11 x0 -2	addi x11, x0, -2
0x7c	0x00200613	addi x12 x0 2	addi x12, x0, 2
0x80	0x00C5F663	bgeu x11 x12 12	bgeu x11, x12, bgeu_taken
0x84	0x00100913	addi x18 x0 1	addi x18, x0, 1 # Fail
0x88	0x0080006F	jal x0 8	j bgeu_end
0x8c	0x10000913	addi x18 x0 256	addi x18, x0, 0x100 # Pass

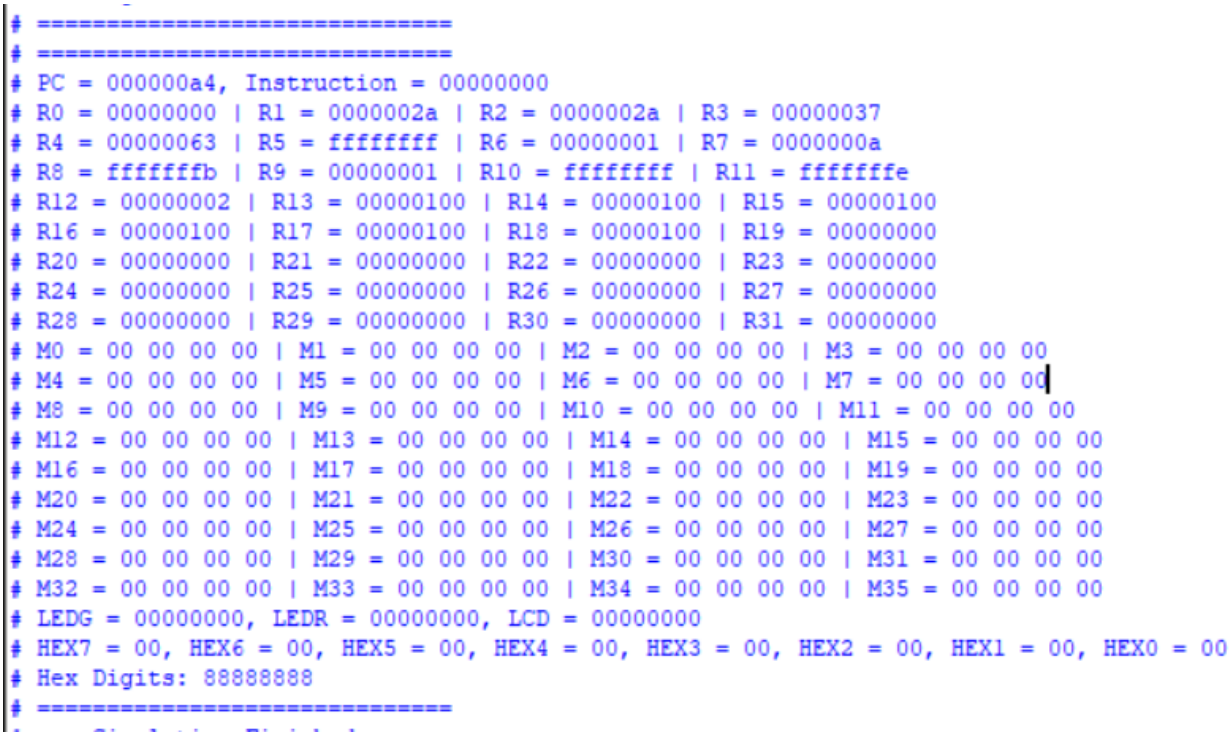


Figure 17. Result of the the end of branch and jump instruction test

3.2.3. Load and store memory instruction test

This test sequence validates the processor's load and store unit by executing a comprehensive series of memory operations. We begin by loading registers x1 and x2 with known values through LUI and ADDI instructions, then perform store operations of varying data widths: a 32-bit word (SW), a 16-bit halfword (SH), and an 8-bit byte (SB) to memory locations relative to a base address in register x4.

The subsequent load operations verify correct data retrieval and extension behavior, including signed and unsigned halfword loads (LH/LHU) and signed and unsigned byte loads (LB/LBU). Through cycle-accurate monitoring of the program counter and memory states, we confirm that each ALU-generated address and data value is properly written to and read from memory, validating the processor's ability to handle different data widths, memory alignment, and sign extension scenarios with complete functional correctness.

Table 5. Load and store memory instruction

PC	Machine Code	Basic Code	Original Code
0x0	0x123450B7	lui x1 74565	lui x1, 0x12345
0x4	0x67808093	addi x1 x1 1656	addi x1,x1, 0x678
0x8	0x00009137	lui x2 9	lui x2, 0x9
0xc	0x7FF10113	addi x2 x2 2047	addi x2,x2,2047
0x10	0x2BD10113	addi x2 x2 701	addi x2,x2,701
0x14	0x0DE00193	addi x3 x0 222	addi x3, x0, 222
0x18	0x00300213	addi x4 x0 3	addi x4, x0, 3
0x1c	0x00122023	sw x1 0(x4)	sw x1, 0(x4)
0x20	0x00221223	sh x2 4(x4)	sh x2, 4(x4)
0x24	0x00320323	sb x3 6(x4)	sb x3, 6(x4)
0x28	0x00022283	lw x5 0(x4)	lw x5, 0(x4)
0x2c	0x00421303	lh x6 4(x4)	lh x6, 4(x4)
0x30	0x00425383	lhu x7 4(x4)	lhu x7, 4(x4)
0x34	0x00620403	lb x8 6(x4)	lb x8, 6(x4)
0x38	0x00624483	lbu x9 6(x4)	lbu x9, 6(x4)

```
# PC = 0000004c, Instruction = 00000000
# R0 = 00000000 | R1 = 12345678 | R2 = 00009abc | R3 = 000000de
# R4 = 00000003 | R5 = 12345678 | R6 = ffff9abc | R7 = 00009abc
# R8 = ffffffde | R9 = 000000de | R10 = 00000000 | R11 = 00000000
# R12 = 00000000 | R13 = 00000000 | R14 = 00000000 | R15 = 00000000
# R16 = 00000000 | R17 = 00000000 | R18 = 00000000 | R19 = 00000000
# R20 = 00000000 | R21 = 00000000 | R22 = 00000000 | R23 = 00000000
# R24 = 00000000 | R25 = 00000000 | R26 = 00000000 | R27 = 00000000
# R28 = 00000000 | R29 = 00000000 | R30 = 00000000 | R31 = 00000000
# M0 = 78 00 00 00 | M1 = bc 12 34 56 | M2 = 00 00 de 9a | M3 = 00 00 00 00
# M4 = 00 00 00 00 | M5 = 00 00 00 00 | M6 = 00 00 00 00 | M7 = 00 00 00 00
# M8 = 00 00 00 00 | M9 = 00 00 00 00 | M10 = 00 00 00 00 | M11 = 00 00 00 00
# M12 = 00 00 00 00 | M13 = 00 00 00 00 | M14 = 00 00 00 00 | M15 = 00 00 00 00
# M16 = 00 00 00 00 | M17 = 00 00 00 00 | M18 = 00 00 00 00 | M19 = 00 00 00 00
# M20 = 00 00 00 00 | M21 = 00 00 00 00 | M22 = 00 00 00 00 | M23 = 00 00 00 00
# M24 = 00 00 00 00 | M25 = 00 00 00 00 | M26 = 00 00 00 00 | M27 = 00 00 00 00
# M28 = 00 00 00 00 | M29 = 00 00 00 00 | M30 = 00 00 00 00 | M31 = 00 00 00 00
# M32 = 00 00 00 00 | M33 = 00 00 00 00 | M34 = 00 00 00 00 | M35 = 00 00 00 00
# LEDG = 00000000, LEDR = 00000000, LCD = 00000000
# HEX7 = 00, HEX6 = 00, HEX5 = 00, HEX4 = 00, HEX3 = 00, HEX2 = 00, HEX1 = 00, HEX0 = 00
# Hex Digits: 88888888
# ============================
```

Figure 18. Result of the the end of load and store memory instruction test

4. FPGA IMPLEMENTATION

4.1. Compile Design process

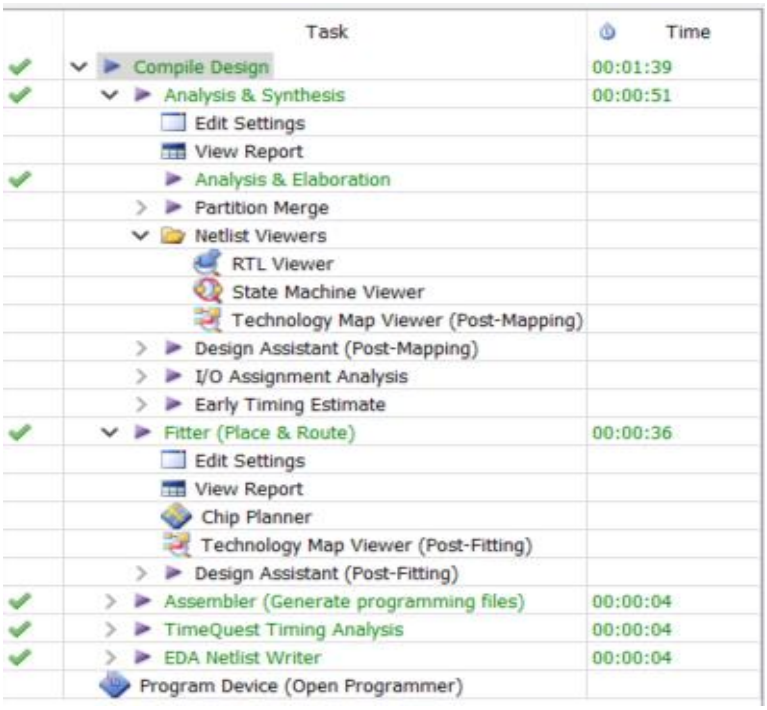


Figure 19. Total runtime for compilation

Flow Status	Successful - Sat Nov 22 15:55:18 2025
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Editio
Revision Name	singlecycle
Top-level Entity Name	single_cycle
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	2,752 / 33,216 (8 %)
Total combinational functions	2,434 / 33,216 (7 %)
Dedicated logic registers	703 / 33,216 (2 %)
Total registers	703
Total pins	219 / 475 (46 %)
Total virtual pins	0
Total memory bits	180,224 / 483,840 (37 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 20. Summary of compile result

4.2. Stopwatch

4.2.1. Flow chart

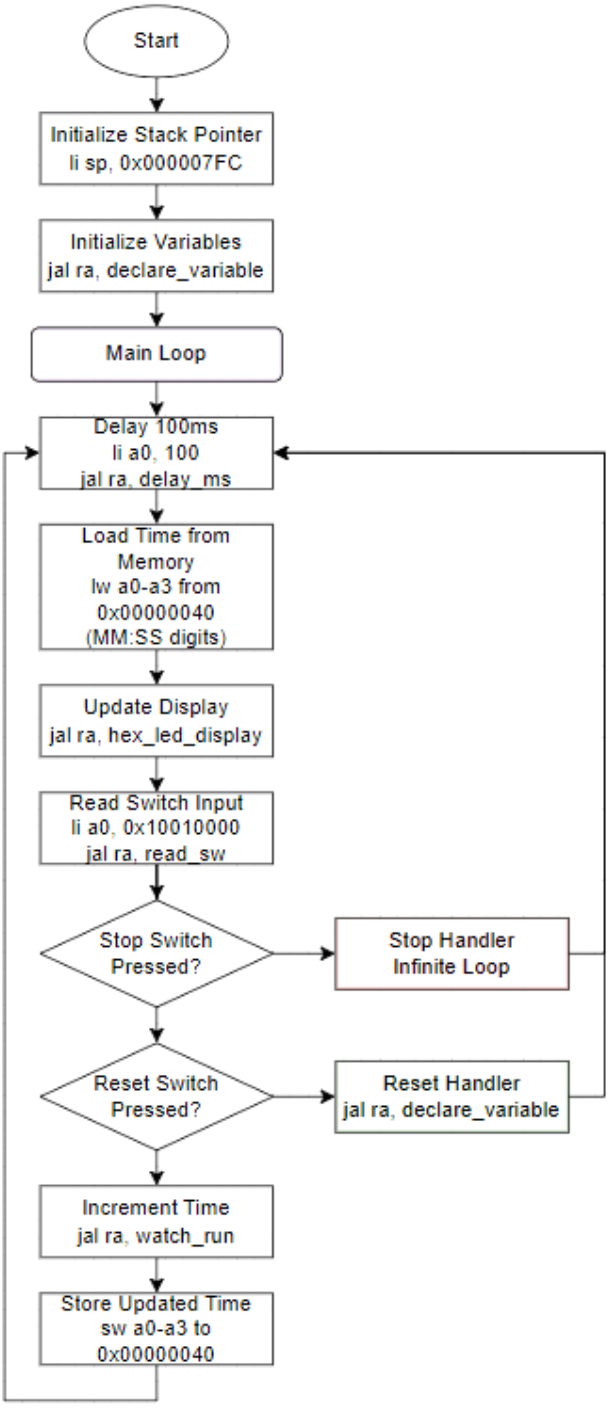


Figure 21. Stopwatch flow chart

4.2.2. FPGA application

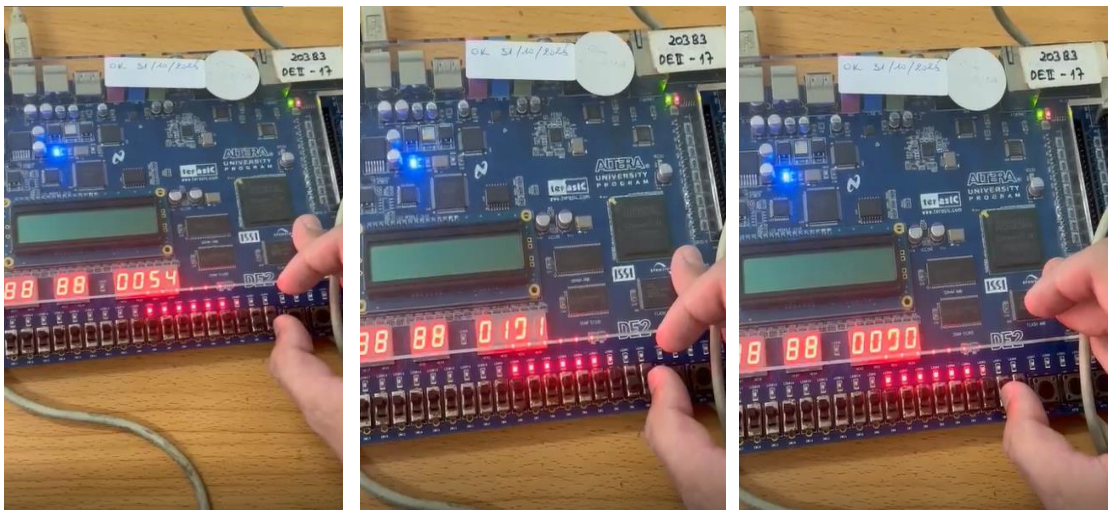


Figure 22. Stopwatch application by FPGA (run, stop-sw0 and reset-sw1)

4.3. Display name on the LCD

4.3.1. Flow chart

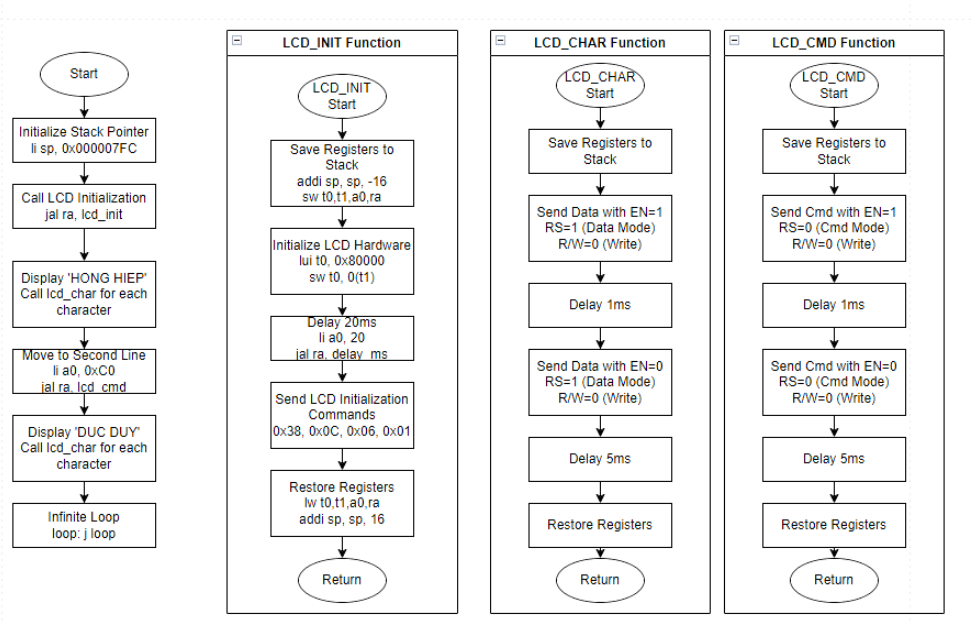


Figure 23. Flowchart of LCD display

4.3.2. FPGA application

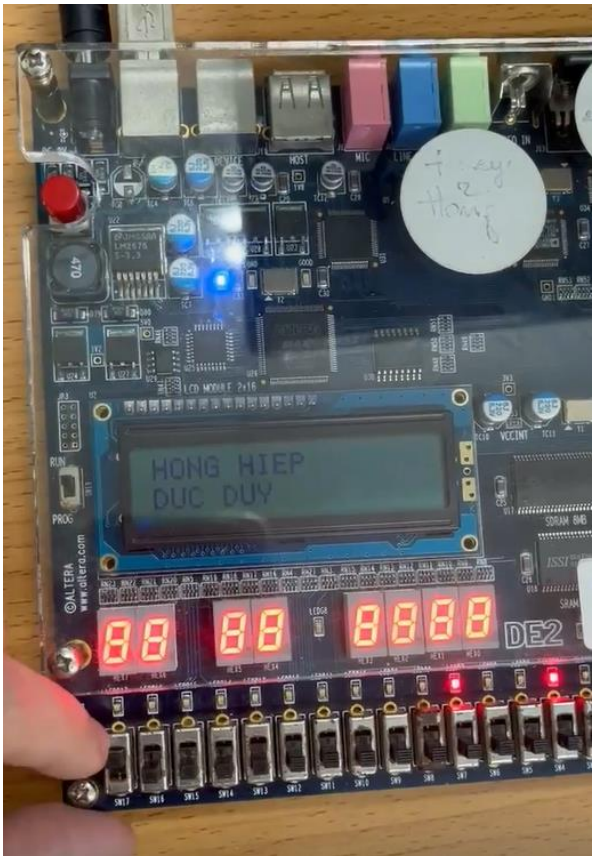


Figure 24. LCD display by FPGA

5. CONCLUSION

This project has successfully achieved its main goal: the complete design and implementation of a fully functional single-cycle RV32I processor. There are detailed the design strategy for every critical component, from the basic building blocks of the ALU, like the 1-bit full adder, to more complex modules such as the control unit, register file, and immediate generator. A key feature of our design is the careful handling of important processor functions, including branch comparison, program counter stalling for load instructions, and a load-store unit that supports misaligned memory accesses using Block RAM.

To ensure the processor worked correctly, a thorough verification strategy was used. Each major module, including the ALU, branch comparator, and register file, was tested individually with a wide range of inputs. This step-by-step testing was crucial for finding and fixing errors early, before integrating all components into the final processor design. This method ensured the overall system's reliability and correctness.

Finally, the practical value of the processor was demonstrated through its implementation on an FPGA. We created real-world applications, such as a stopwatch and a program to display a name on an LCD screen. These applications prove that our processor is not just a theoretical model but a practical and functional digital system capable of executing complex tasks.

In summary, we have designed, built, tested, and demonstrated a working processor that meets all the initial requirements, solidifying our understanding of computer architecture and digital design.