Yusuf BARAN / 2574747
Emre ANT / 2518561

MIDDLE EAST TECHNICAL UNIVERSITY

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

EE446 LABORATORY PROJECT

SINGLE CYCLE RISC-V PROCESSOR

1. INTRODUCTION

In this project, we designed and implemented a single-cycle 32-bit RISC-V processor based on the RV32I base integer instruction set architecture. Our goal was to develop a fully functional CPU capable of executing a wide range of instructions—including arithmetic, logic, memory, branching, and jump operations—on an FPGA platform, while also supporting UART communication through memory-mapped I/O. The design process involved developing the datapath, controller, and a UART peripheral module, with a strong emphasis on modularity, clean Verilog structure, and reliable synchronization across clock domains.

Throughout the project, we carefully followed the RISC-V specification to ensure that each instruction type was accurately decoded and executed. Special attention was given to handling immediate fields—particularly for B-type and J-type instructions, where bit extraction and sign extension are non-trivial. We also developed a cycle-accurate testbench in Python using the Cocotb framework. This testbench compared the hardware's behavior against a software performance model for every instruction and provided full visibility into internal signals during simulation. Lastly, we integrated a working UART module that could send and receive bytes via standard RISC-V store and load instructions, verified through both hardware and testbench simulations.

2. DATAPATH

2.1. Instructions

2.1.1. Arithmetic Instructions: ADD, ADDI, SUB

- **R-type (ADD/SUB):** The values from two registers (rs1 and rs2) are read through the Register File. The ALU performs addition or subtraction based on the control signal (ALUControl). The ALU result is written back into the destination register (rd), using the write-back data path via the ALU result multiplexer.

- **I-type (ADDI):** A similar procedure occurs, except the second operand is an immediate value. This immediate is first sign-extended by the Immediate Extender and then fed into the ALU via the ALUSrc multiplexer. The result from the ALU is then written back to the register file.

2.1.2. Logic Instructions: AND, ANDI, OR, ORI, XOR, XORI

- **R-type (AND/OR/XOR):** The source operands (rs1 and rs2) are directly read from the Register File and fed into the ALU. The ALU performs the logical operation dictated by the control signals. The computed result returns through the ALU multiplexer and is written into the register file (rd).
- **I-type (ANDI/ORI/XORI):** Here, an immediate value, extended by the Immediate Extender, is chosen as the second operand instead of a register value via the ALUSrc multiplexer. The operation proceeds identically to register-based logic instructions.

2.1.3. Shift Instructions: SLL, SLLI, SRL, SRLI, SRA, SRAI

Shift instructions (register and immediate forms): The ALU receives operands similar to arithmetic/logical instructions. However, the immediate or second register value determines the shift amount. The shift direction (left logical, right logical, or right arithmetic) is controlled via the ALUControl signal. Results are written back into the destination register.

2.1.4. Set if Less Than Instructions: SLT, SLTI, SLTU, SLTIU

These instructions use the ALU to compare two values: SLT and SLTU compare two register operands. SLTI and SLTIU compare a register operand and an immediate operand. The comparison output (1 or 0) is provided by the ALU's LessThan output signal, forming the least significant bit of the ALUResult. This output is written back into the destination register through the write-back multiplexer.

2.1.5. Conditional Branch Instructions: BEQ, BNE, BLT, BLTU, BGE, BGEU

The ALU computes the comparison, and the result is indicated via the Zero and ALU_LessThan flags. Based on these flags and branch logic (PCSrc signal), the next program counter value (PC_next) is selected from either PC+4 (no branch) or PC+Immediate (branch target), calculated via the branch adder. The instruction does not require write-back to the register file.

2.1.6. Unconditional Jump Instructions: JAL, JALR

- **JAL (Jump and Link):** The PC is updated directly with PC+Immediate, computed by the branch adder, via the PCSrc multiplexer. Additionally, the return address (PC+4) is written into register rd through the write-back multiplexer (RegWSel control).
- **JALR (Jump and Link Register):** The ALU calculates the jump target address (rs1 + Immediate). The Jalr multiplexer selects this computed target (ensuring the least significant bit is zero) and updates the PC. Similar to JAL, the return address (PC+4) is stored into rd.

2.1.7. Load Instructions: LW, LH, LHU, LB, LBU

Load instructions use the ALU to compute the effective address (base + immediate). The Data Memory is accessed at this address, retrieving the data. Retrieved data undergoes sign or zero extension based on SE2Control. Finally, the extended data is written back into the register file (rd) via the Result multiplexer controlled by ResultSrc.

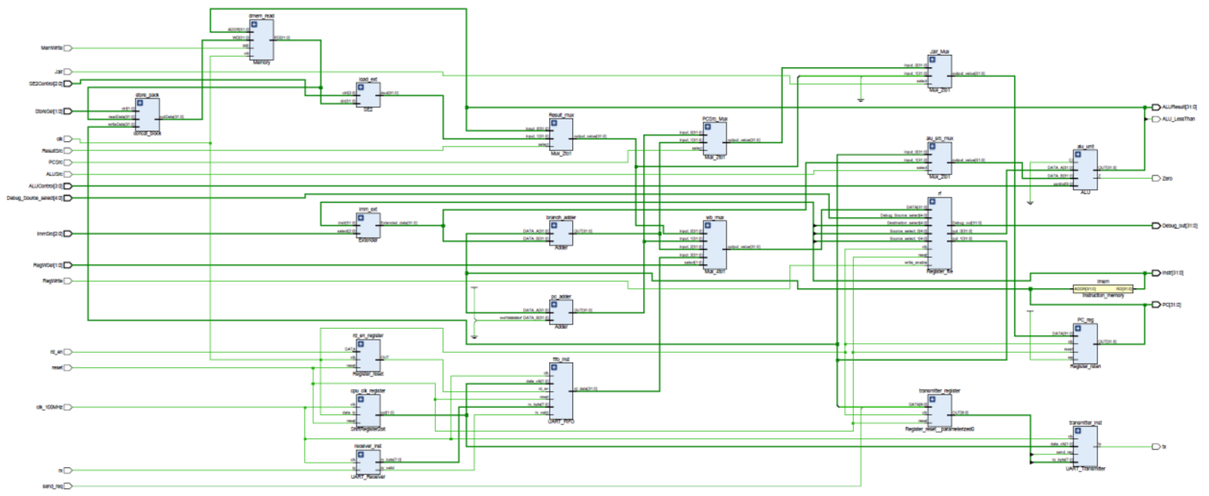
2.1.8. Store Instructions: SW, SH, SB

Store instructions calculate the effective address similarly to load instructions (base register + immediate through ALU). The register (rs2) value is packed appropriately (byte/half/word) using the concat_block based on StoreSel control signals. The packed data is written into the Data Memory at the computed address. No data is written back to the register file.

2.1.9. Other Instructions: LUI, AUIPC

- **LUI (Load Upper Immediate):** The immediate is shifted left by 12 bits via the Immediate Extender and directly written to the destination register (rd).
- **AUIPC (Add Upper Immediate to PC):** Immediate (shifted left by 12 bits) is added to the current PC using the ALU, and the result is stored in the destination register (rd).

2.2. Datapath RTL View



3. CONTROLLER

3.1. Instructions

3.1.1. Arithmetic Instructions: ADD, ADDI, SUB

- **R-type (ADD, SUB):**
  - Opcode (0110011) identifies an arithmetic instruction.
  - Register operands are selected (ALUSrc = 0).
  - The result from the ALU is routed to the register file (RegWrite = 1, RegWSel = 0).
  - The ALUControl is set by checking funct3 and funct7_5:
    - 000: ADD (funct7_5=0), SUB (funct7_5=1).
- **I-type (ADDI):**
  - Opcode (0010011) is decoded similarly, but immediate operand usage is indicated (ALUSrc = 1).
  - Immediate-type encoding selected (ImmSrc=0).
  - Results written back via ALU operation directly to the destination register (RegWrite=1, RegWSel=0).
  - ALUControl is ADD for ADDI.

3.1.2. Logic Instructions: AND, ANDI, OR, ORI, XOR, XORI

- **R-type (AND/OR/XOR):**
  - Opcode (0110011) instructs ALU to perform logical operations based on funct3:
    - 111: AND, 110: OR, 100: XOR.
  - Results are directly written back to the register file.
- **I-type (ANDI/ORI/XORI):**
  - Opcode (0010011) with immediate values.
  - Immediate operand selected (ALUSrc=1).
  - ALUControl set similarly based on funct3.

3.1.3. Shift Instructions: SLL, SLLI, SRL, SRLI, SRA, SRAI

- Decoded under R-type (0110011) and I-type (0010011) instructions.
- ALU operation set by funct3 and funct7_5:

    - 001: SLL/SLLI, 101: SRL/SRLI (if funct7_5=0), SRA/SRAI (if funct7_5=1).
- Register (R-type) or immediate (I-type) shift amounts are chosen through ALUSrc.

3.1.4. Set if Less Than Instructions: SLT, SLTI, SLTU, SLTIU

- Controlled similarly to arithmetic instructions:
    - ALU comparison operations chosen based on funct3:
        - 010: SLT/SLTI, 011: SLTU/SLTIU.
- ALU results are written back as a binary indicator (1 or 0).

3.1.5. Conditional Branch Instructions: BEQ, BNE, BLT, BLTU, BGE, BGEU

- Opcode (1100011) specifies branch instructions.
- ALU compares operands; the PC update (PCSrc) depends on ALU flags (Zero, ALU_LessThan):
    - BEQ (000): Branch if equal (PCSrc=Zero).
    - BNE (001): Branch if not equal (PCSrc=~Zero).
    - BLT (100) / BLTU (110): Branch if less (PCSrc=ALU_LessThan).
    - BGE (101) / BGEU (111): Branch if greater or equal (PCSrc=~ALU_LessThan).

3.1.6. Unconditional Jump Instructions: JAL, JALR

- **JAL (1101111):**
    - Immediate jump (PCSrc=1) to PC+Immediate.
    - Writes return address (PC+4) to register file (RegWrite=1, RegWSel=1).
- **JALR (1100111):**
    - Target computed by ALU (rs1 + immediate), controlled by Jalr=1.
    - Writes return address (PC+4) similarly (RegWrite=1, RegWSel=1).

3.1.7. Load Instructions: LW, LH, LHU, LB, LBU

- Opcode (0000011) sets data memory operations.
- ALU computes memory address (ALUSrc=1).
- Memory read data is selected as the result (ResultSrc=1).

- Loaded data is appropriately sign or zero-extended (SE2Control) based on instruction:
  - LW (010): no extension needed.
  - LH (001): 16-bit sign-extension.
  - LHU (101): 16-bit zero-extension.
  - LB (000): 8-bit sign-extension.
  - LBU (100): 8-bit zero-extension.
- Special handling at memory-mapped UART address (0x00000404) triggers read enable (rd_en=1) and a special write-back (RegWSel=3).

3.1.8. Store Instructions: SW, SH, SB

- Opcode (0100011) activates memory write (MemWrite=1).
- Memory write size controlled via StoreSel:
  - SW (010): Word store (StoreSel=0).
  - SH (001): Half-word store (StoreSel=1).
  - SB (000): Byte store (StoreSel=2).
- UART transmission (0x00000400) handled uniquely: sets send_req=1 and cancels memory write (MemWrite=0).

3.1.9. Other Instructions: LUI, AUIPC

- **LUI (0110111):**
  - Immediate extended and passed directly to register (RegWrite=1, RegWSel=0).
  - ALU set to special control (ALUControl=1101) for upper immediate.
- **AUIPC (0010111):**
  - Immediate added to current PC, writing result into register (RegWrite=1, RegWSel=2).
  - No special ALU operation needed beyond normal addition.

3.1.10. UART Peripheral Control Signals

In this project, the UART (Universal Asynchronous Receiver/Transmitter) peripheral is interfaced through memory-mapped I/O. The controller uses two specific memory addresses to interact with the UART module:

- 0x00000400: Writing a byte here initiates UART transmission (SB instruction).
- 0x00000404: Reading from here fetches a received byte from the UART receiver (LW instruction).To handle this interaction properly, the controller uses two custom control signals: send_req and rd_en. Here's how they are used:

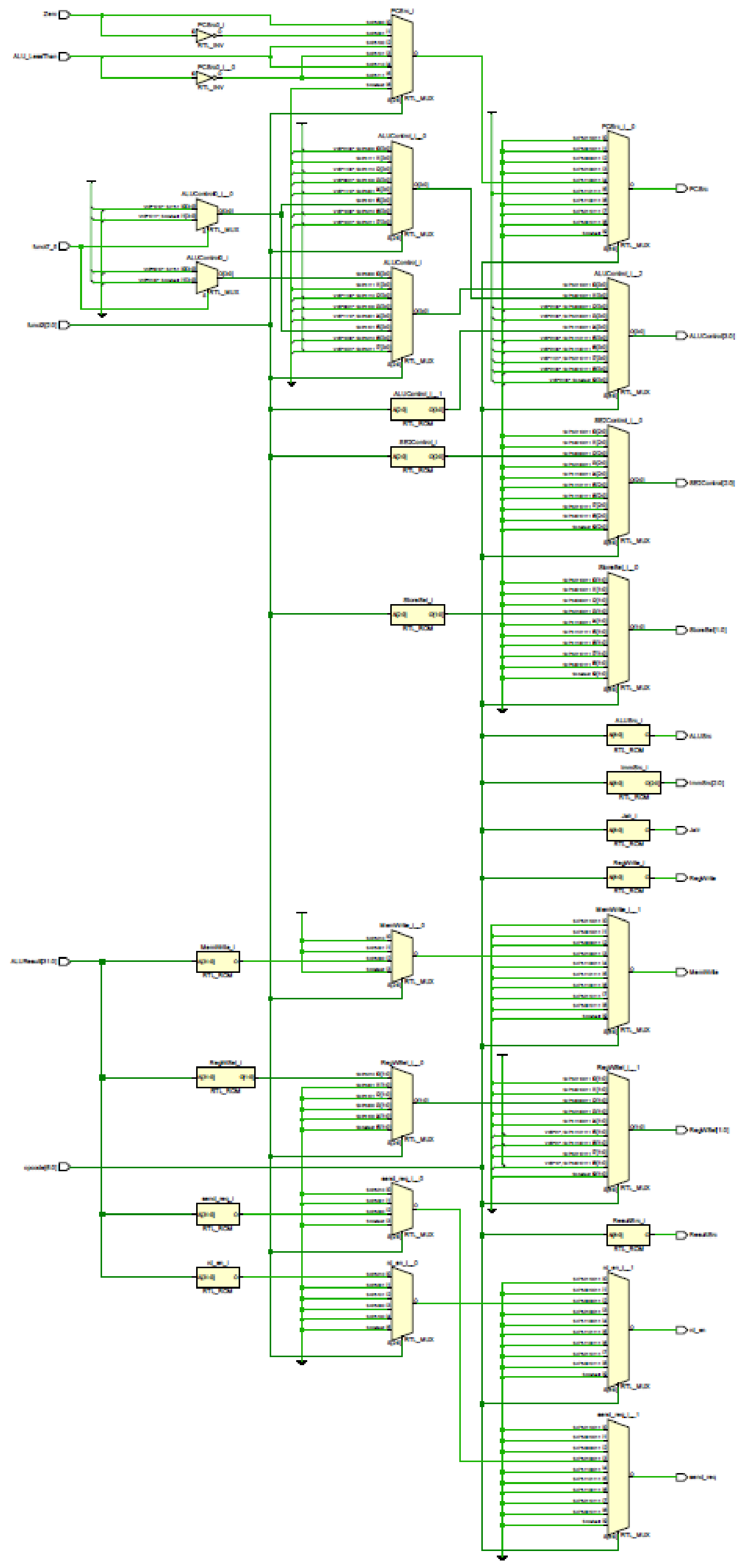
3.1.10.1. send_req: Initiates UART Transmission

- Purpose: Signals the UART transmitter to send a byte.
- When it activates:
  - During store byte (SB) instruction (opcode = 0100011) with funct3 = 000.
  - If the computed memory address (ALUResult) equals 0x00000400.
- Why it's needed: Writing to 0x00000400 should not write to memory but instead trigger the UART hardware. Setting send_req = 1 does this.
- What else happens: MemWrite is explicitly set to 0 in this case to avoid accidentally modifying real memory.

3.1.10.2. rd_en: Enables UART Reception (FIFO Read)

- Purpose: Triggers a read from the FIFO buffer that stores received UART bytes.
- When it activates:
  - During load word (LW) instruction (opcode = 0000011) with funct3 = 010.
  - If the computed memory address (ALUResult) equals 0x00000404.
- Why it's needed: Reading from this address means fetching a byte from the UART FIFO, not regular data memory. rd_en = 1 initiates this.
- What else happens: RegWSel = 3 is used to select rd_data from the FIFO as the value to write back to the register file.

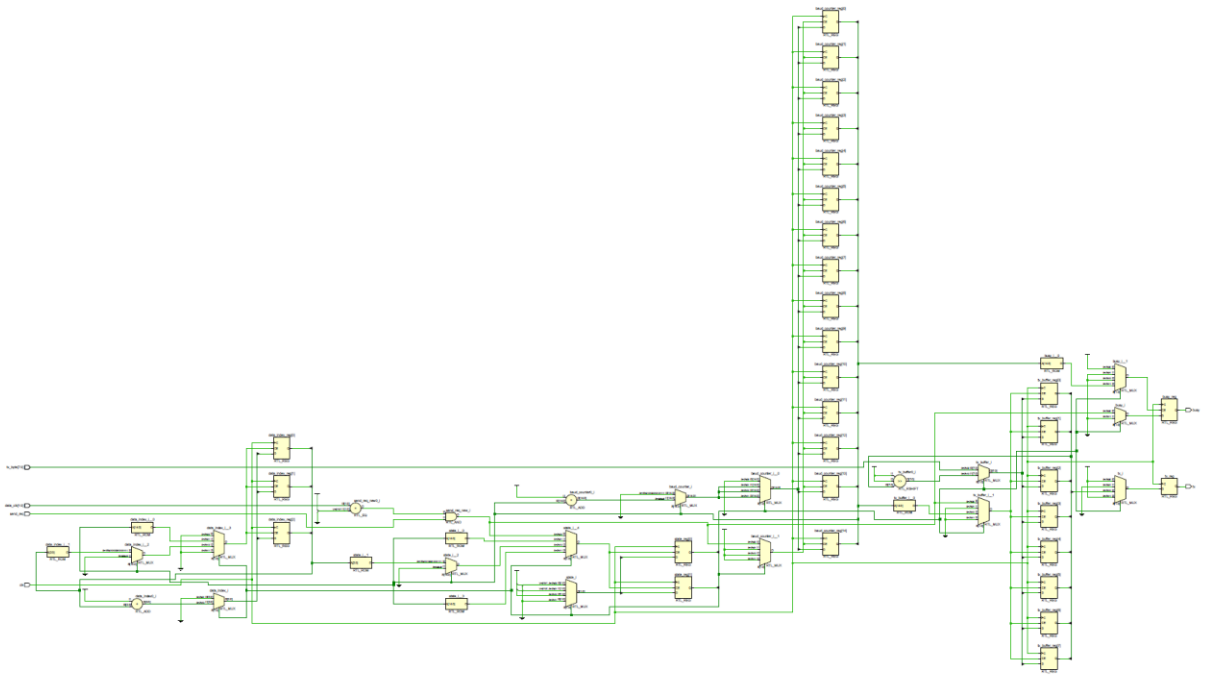
3.2. Controller RTL View



4. UART PERIPHERAL

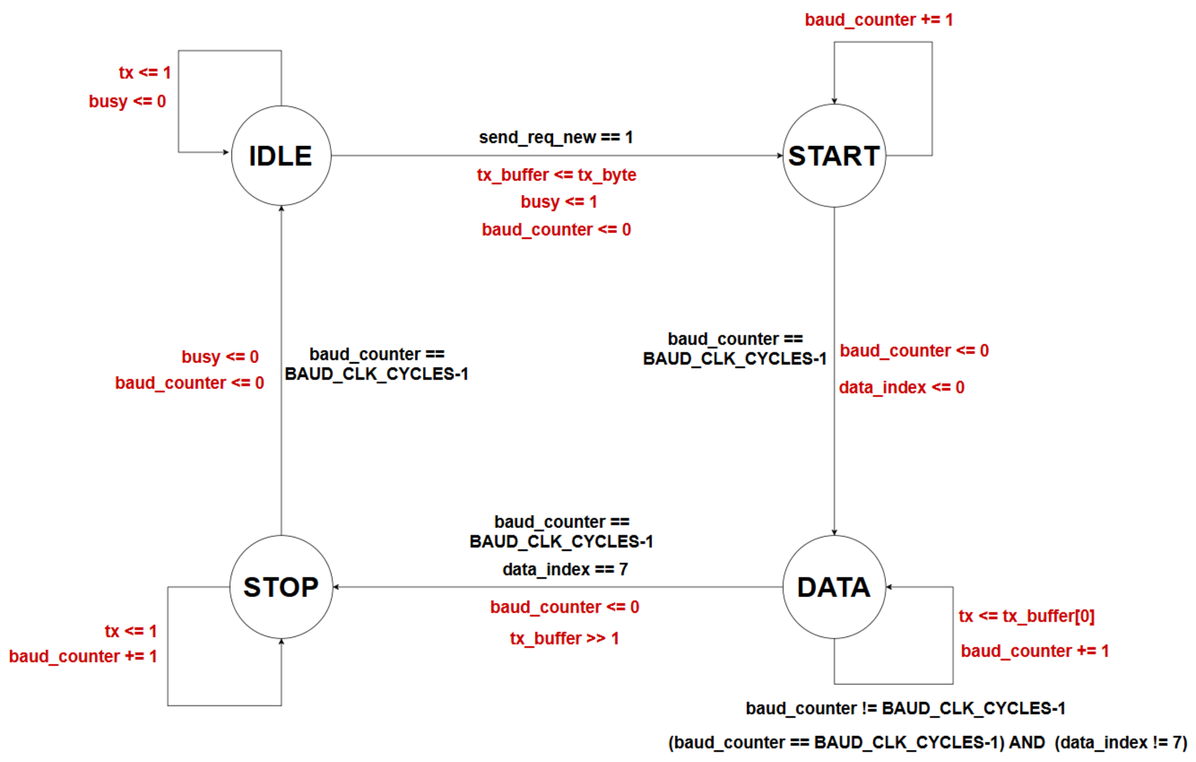
4.1. UART Transmitter

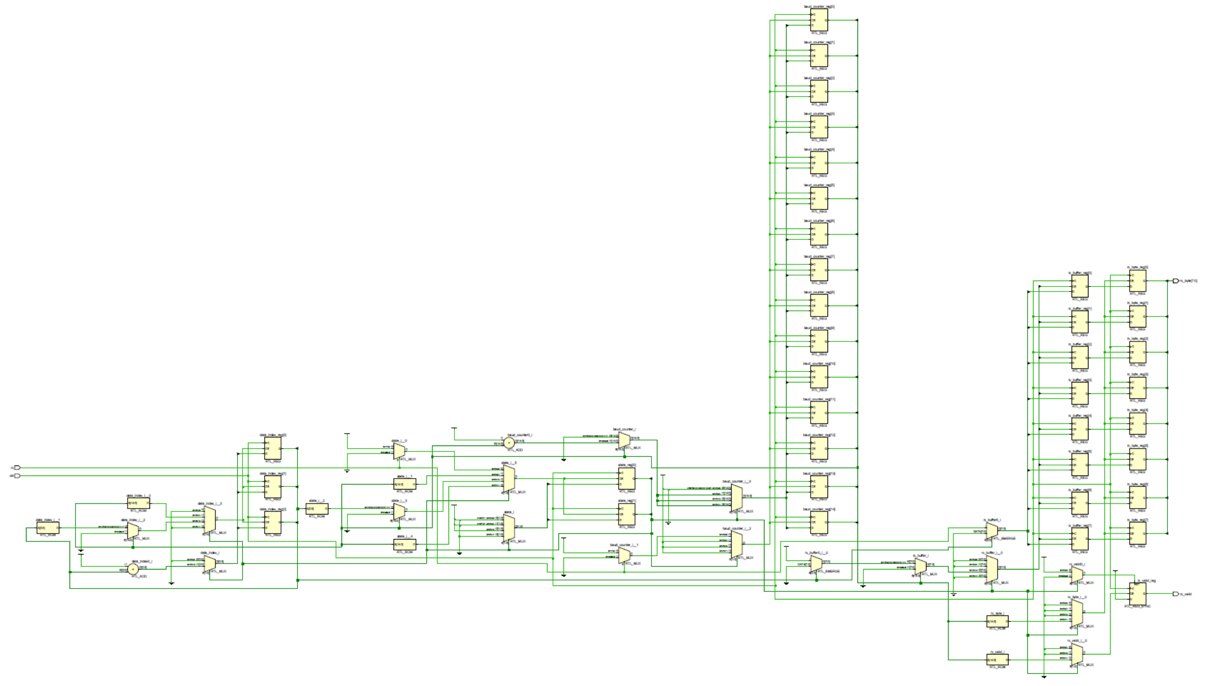
The UART_Transmitter handles converting parallel data from the CPU into serial format to be transmitted via the tx line. It receives the byte to send through tx_byte and starts transmission when send_req is asserted. However, to ensure correct timing and stable data, the signal is registered and delayed by one clock cycle before reaching the transmitter module. Once the transmission starts, the transmitter enters a state machine that outputs the start bit (logic low), all 8 data bits (least significant bit first), and the stop bit (logic high), all paced according to the UART baud rate. This module also uses the 100 MHz clock and an internal baud-rate counter to meet UART protocol timing requirements.



4.2. UART Receiver

The UART_Receiver module is responsible for monitoring the rx line and decoding incoming serial data into 8-bit parallel bytes. It runs entirely on the 100 MHz clock to achieve accurate baud-rate timing (9600 bps), and its internal state machine follows a standard 8-N-1 protocol. When the rx line falls to logic low, the receiver recognizes this as a start bit and begins sampling in the middle of each bit period using a baud counter. It then sequentially captures 8 data bits and checks for the stop bit. After all bits are correctly received, the byte is moved into an internal buffer (rx_buffer), and the rx_valid signal is asserted high for one cycle. This signal notifies the rest of the system—particularly the FIFO—that a new byte is ready to be read.

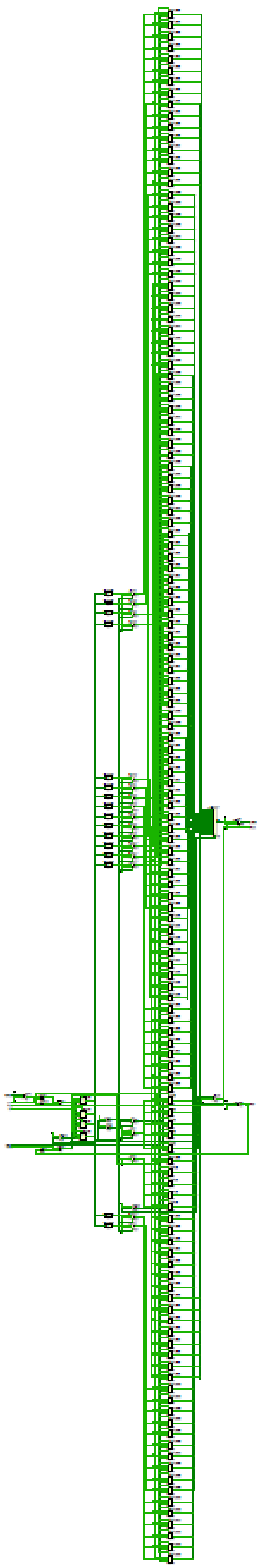




4.3. FIFO Receive Buffer

The UART_FIFO module serves as a bridge between the UART domain and the processor. Since the processor and UART operate on different clocks, and the processor may not read the incoming data immediately, the FIFO ensures no data is lost by buffering up to 16 received bytes. It maintains internal read_index, write_index, and a count register to manage the circular buffer and track how many entries are filled. On each clock cycle, if rx_valid is high and the FIFO is not full, the byte from the receiver is stored at the current write_index. On the other side, if the CPU requests a read (rd_en = 1) and the FIFO is not empty, the module returns the byte located at read_index via the rd_data output. If no data is available, rd_data defaults to 0xFFFFFFFF as specified in the project description. The read operation is synchronized to the falling edge of the CPU clock by monitoring the data_clk signal (specifically data_clk == 2'b10), ensuring proper timing between the CPU and FIFO.

Yusuf BARAN / 2574747
Emre ANT / 2518561



4.3.1. Clocking Scheme and Synchronization

One of the most critical implementation decisions in this project was choosing the right clock domains and synchronizing them correctly. While the main CPU and datapath operate on a custom CPU clock (button), the UART system runs on a high-frequency 100 MHz clock (clk_100MHz). This separation is intentional: UART requires precise baud-rate timing (9600 baud). To bridge these two domains, we implemented a 2-bit shift register (ShiftRegister2bit) that samples the slower CPU clock using the faster 100 MHz clock. This allows us to detect clock transitions—particularly the falling edge of the CPU clock—by observing when data_clk transitions from 2'b11 to 2'b10.

The falling edge is significant because UART components are required to act when we push the button. So that if we connected the faster clock to the modules, when we reach a uart instruction it will be immediately executed since our control logic is combinational. By triggering UART interactions during this edge, we ensure proper timing alignment between CPU write/read instructions and UART data transfers. Moreover, all state machines in the UART modules operate on clk_100MHz, isolating them from glitches or skew caused by the slower CPU domain.

4.3.2. Signal Timing: Delaying send_req for Stability

To ensure the UART transmitter operates reliably, it was necessary to delay the send_req signal by exactly one clock cycle before routing it to the UART_Transmitter. This decision stems from the fact that we want to send the data packet when we push the button. When we push the button at the UART instruction our program counter will jump to the next address. During this transition the packet should be sent. So that we must carry the control signals to the next clock cycle of the CPU. One other important point is we are triggering the real transmission with the falling edge. However, since the signals immediately reach to the output of the registers on the rising edge , all operation is consistent.

To handle this, we placed both the data byte and the send_req signal into a Register_reset module clocked by cpu_clk, ensuring that the values are captured and held stable across one clock cycle. This registered output ({tx_byte_TX, send_req_TX}) is then forwarded to the UART transmitter module. This guarantees that the transmitter always receives the correct byte and that send_req is only asserted once the data is stable and fully settled. This is especially crucial given the asynchronous nature of UART and the possibility of metastability or incomplete transmission if timing is misaligned. Together, this mechanism ensures safe and consistent operation across clock domains.

5. TESTBENCH

To verify the correctness of our single-cycle RISC-V processor, we developed a comprehensive testbench using the Cocotb framework in Python. The goal of the testbench was to simulate realistic execution of instruction sequences and compare the behavior of the hardware (DUT – Device Under Test) with a software-based performance model written in Python. The testbench mimics the behavior of a real CPU, including immediate calculations, register updates, PC logic, memory handling, and UART-specific corner cases.

5.1. Instruction Input and Parsing

We begin by reading the instruction stream from a hex file (Instructions.hex) using a helper function read_file_to_list(). The RISC-V instructions are stored in memory in little-endian format. To correctly interpret this in Python, we implemented a helper function called reverse_hex_string_endiannes() to reverse byte order before parsing.

Each instruction is parsed using the Instruction class, which extracts opcode, register indices, funct3, funct7, and all relevant immediate values (imm_I, imm_S, imm_B, imm_U, and imm_J). Immediate decoding was handled with extreme care, especially for B-type and J-type instructions which have disjoint and shifted bit fields. For example, the J-type immediate calculation reconstructs the full 21-bit signed value from scattered instruction bits (bit[31], bit[19:12], bit[20], and bit[30:21]) and properly applies sign-extension.

5.2. Software Performance Model

Our TB (testbench) class acts as a cycle-accurate performance model, executing one instruction per cycle just like the real processor. It maintains a shadow register file (32 registers), a program counter (PC), and a byte-addressable memory model. On each cycle, it fetches the current instruction, parses the fields, and simulates execution based on the instruction type.

The testbench correctly simulates the behavior of:

- R-type and I-type arithmetic/logic instructions (with correct handling of signed vs. unsigned operations),
- Branches and jumps (with accurate PC updates and condition checks),
- Load and store instructions with full memory access emulation,
- Special behavior for memory-mapped UART I/O at 0x00000400 (transmit) and 0x00000404 (receive).

Shift operations (SLL[I], SRL[I], SRA[I]) are implemented via the shift_helper() utility, which distinguishes between logical shift left/right and arithmetic right shift, and respects the 5-bit shamt mask (& 0x1F).

5.3. Clocking and Synchronization

The simulation clock for the DUT is generated using Cocotb's Clock object with a 10 μs period. On each iteration of the main loop, we perform the following steps:

1. Execute the performance model logic.
2. Log the datapath and controller outputs via Log_Datapath() and Log_Controller().
3. Advance simulation through a RisingEdge and then a FallingEdge of the clock.

4. After the edge, the hardware should have completed execution of the instruction.
5. Compare the Python model state (PC, register file) against the DUT.
6. This process repeats for a fixed number of cycles (120 in our case) or until execution halts.

5.4. Memory and UART Handling

The testbench includes a custom ByteAddressableMemory class that supports aligned 32-bit reads and writes, along with partial byte-level access needed for SB/SH instructions. Special logic is added to handle UART behavior:

- For SB to 0x00000400, we simulate UART transmission by doing nothing (since UART output is not verified).
- For LW from 0x00000404, the expected result is always 0xFFFFFFFF to match the hardware's empty FIFO default behavior.

These behaviors are crucial to passing UART-related tests without requiring full UART simulation.

5.5. Result Comparison

After each instruction is executed, we verify that the hardware PC and all 32 registers match the expected values from our software model. This assertion ensures bit-accurate correctness. Logs provide detailed side-by-side comparisons, making debugging easy in case of mismatches.

6. ASSEMBLY PROGRAM TO TEST THE DESIGN

To rigorously validate our RISC-V processor implementation, we developed a custom instruction sequence that spans the entire RV32I instruction set, including all instruction types—R-type, I-type, S-type, B-type, U-type, and J-type—as well as memory-mapped I/O operations. The instruction program is written in assembly, compiled to machine code, and executed via the testbench to ensure functional correctness and architectural compliance. Below is a breakdown of how the program provides thorough coverage of each category.

6.1. Arithmetic, Logic, and Shift Instructions (R-type & I-type)

The instruction sequence begins with a series of arithmetic and logical operations:

- **ADDI, ADD, SUB:** These test both immediate and register-based addition and subtraction.
- **ANDI, ORI, XORI, AND, OR, XOR:** Logical operations are verified for both I-type and R-type encodings.
- **SLL, SLLI, SRL, SRLI, SRA, SRAI:** All three shift operations (logical left, logical right, arithmetic right) are exercised using both register and immediate

forms. Special attention was given to high shift amounts (e.g., shift by 31) to test edge behavior and ensure correct sign extension in sra.

6.2. Set-Less-Than Instructions

Both signed and unsigned comparisons are tested:

- **SLT, SLTU:** Tests register comparisons.
- **SLTI, SLTIU:** Covers immediate variants.

Special cases include equal comparisons (e.g., slt x22, x1, x1) and inputs with signed/unsigned boundary values to verify correct signedness behavior in the ALU.

6.3. Branch Instructions (B-type)

Branch instructions are exhaustively tested under varying conditions:

- BEQ, BNE, BLT, BGE, BLTU, BGEU: All conditional branches are executed with both taken and not-taken paths.
- The program reinitializes registers (addi x2, x0, #0x27) before critical branches to ensure clean test separation.
- Edge cases include self-branching (beq x0, x0, #-108) to test infinite loop prevention, and symmetric condition checks (e.g., blt x2, x2) to verify correct branching when conditions are false.

6.4. Unconditional Jump Instructions (J-type & JALR)

- JAL and JALR are explicitly tested with return addresses to validate correct PC-relative jumps and register return values.
- JALR includes an offset from a base register (x13) to confirm proper calculation of the jump target (rs1 + imm_I) with LSB zero padding.

6.5. Upper Immediate Instructions (U-type)

lui and auipc are tested with large upper immediate values:

- lui x6, #0xABCDE verifies proper placement of the upper 20 bits.
- auipc x5, #0xA7C3 ensures PC-relative arithmetic is functioning.

6.6. Store and Load Instructions (S-type & I-type Memory)

Memory interaction is thoroughly tested:

- Store instructions: sw, sh, sb write values into a memory block pointed to by x8.
- Load instructions: lw, lh, lhu, lb, lbu retrieve values from the same memory and store them into different registers to verify each variant.
- Boundary cases are included by storing a full word and attempting to retrieve it as half-word and byte to validate correct sign/zero extension.

6.7. UART Peripheral (Memory-Mapped I/O)

To verify UART transmit functionality:

- Characters are loaded into x1 to x10 and sent via multiple sb instructions targeting address 0x00000400 (held in x11).
- These simulate a real-world UART send buffer writing out the string "emre yusuf".

To test UART receive logic:

- Multiple lw x12, #4(x11) instructions are included, reading from 0x00000404, which corresponds to the UART FIFO receive buffer. Since the FIFO is empty by default, each instruction correctly returns 0xFFFFFFFF, mimicking the actual hardware behavior.

6.8. Loop and Execution Duation Control

To allow the testbench to reach a safe stopping point, the final instruction beq x0, x0, #-108 creates an intentional infinite loop. This serves as a simple and effective halting mechanism within the simulation scope, ensuring the performance model and DUT don't proceed into uninitialized memory.

7. CONCLUSION

This project allowed us to explore and understand the core concepts of computer architecture by building a RISC-V CPU from the ground up. From datapath construction and control signal generation to memory interfacing and UART communication, every subsystem was designed to work harmoniously in a single-cycle architecture. Our modular Verilog design ensured clarity and reusability, while the detailed Cocotb-based testbench provided a powerful debugging and validation environment.

We successfully implemented all required instructions and validated edge cases, such as large and negative immediate values, shift operations with high shift amounts, and both signed and unsigned comparisons. Additionally, the UART peripheral integration showcased our ability to handle asynchronous communication using synchronized control signals and clock-domain bridging. The final system ran on FPGA hardware and correctly executed a comprehensive test program that touched on every instruction type and special scenario. This hands-on experience deepened our understanding of both digital design and processor microarchitecture, equipping us with valuable skills for future hardware design projects.

Yusuf BARAN / 2574747
Emre ANT / 2518561