Middle East Technical University

Department of Electrical and Electronics Engineering

EE446 – Computer Architecture II

Single Cycle RISC-V Processor

Ahmet Uğur Akdemir, 2515435

Mustafa Mert Mıhçı, 2516565

Monday 29th September, 2025

This project involves designing and implementing a single-cycle 32-bit RISC-V processor based on the RV32I instruction set. A complete datapath, control unit, and a memory-mapped UART peripheral are developed to support instruction execution and serial communication. The design is synthesized on a Nexys A7 FPGA and verified using a Python *cocotb* testbench.

Datapath Design

This Datapath module is a fundamental part of a processor design. It orchestrates various operations in the processor by directing data between different components, performing arithmetic and logic operations, accessing memory, and interacting with peripherals such as UART (Universal Asynchronous Receiver-Transmitter). Let's break it down step-by-step to understand how it works. The RTL schematics of Datapath can be seen in Figure 1.

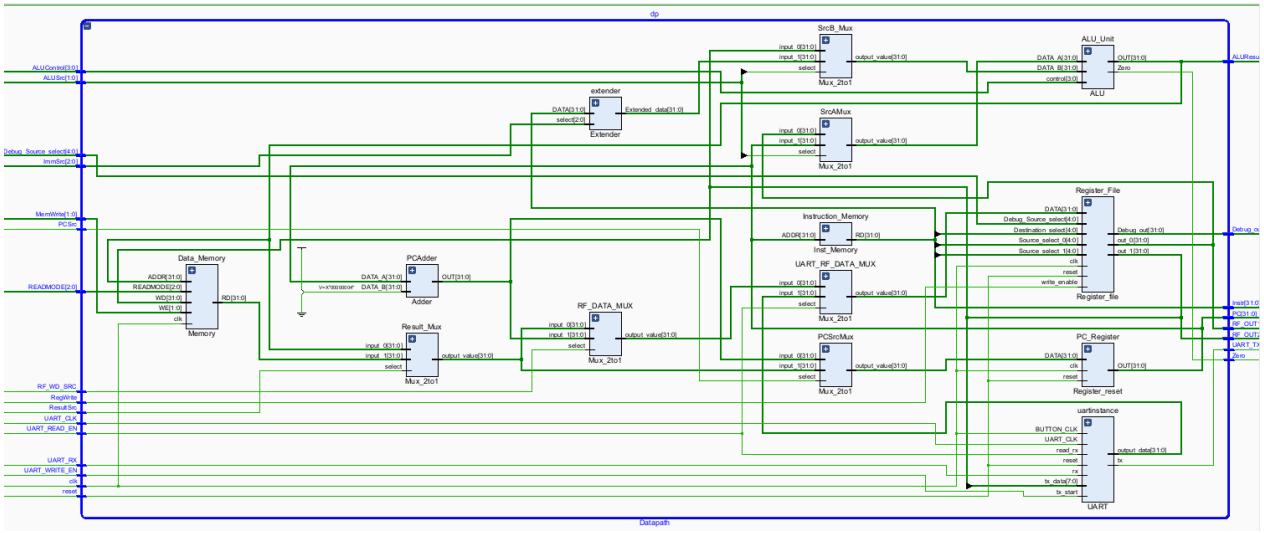

Figure 1: RTL Schematic of Datapath

Inputs and Outputs:

1. **Inputs:**

   - **clk, reset:** Control signals for synchronization and resetting the datapath.
   - **Control Signals:**
     - `PCSrc`: Selects the source for the next Program Counter (PC) value.

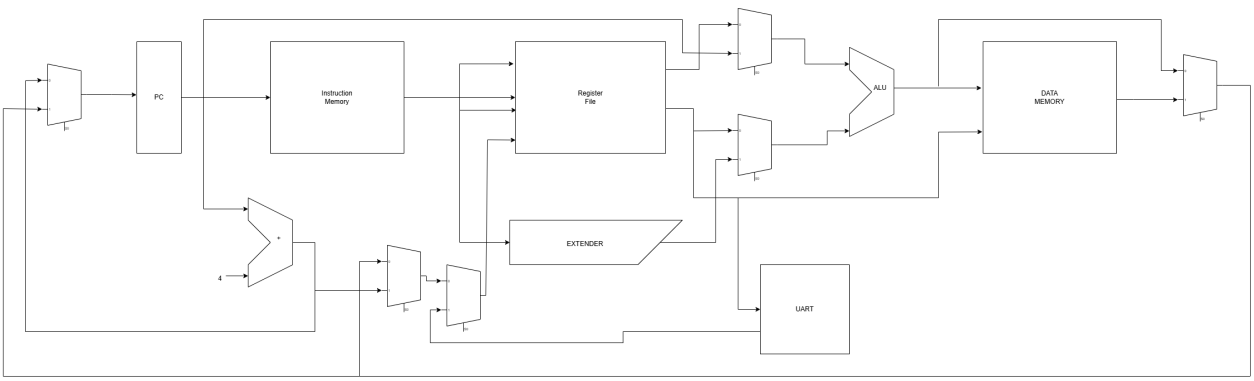

Figure 2: Datapath Design

- `RegWrite`: Enables or disables writing to the registers.
- `ResultSrc`: Chooses whether the result comes from the ALU or memory.
- `RF_WD_SRC`: Decides the source for writing to the register file (from ALU or memory).
- `MemWrite`: Controls writing to memory.
- `ALUSrc`: Selects the operands for the ALU (either from registers or immediate value).
- `ImmSrc`: Selects the source for extending the immediate value.
- `READMODE`: Specifies how to read from memory (byte, half word and word).
- `ALUControl`: Specifies ALU operation.
- `Debug_Source_select`: Used to select a debug source.
- `UART_CLK, UART_RX, UART_READ_EN, UART_WRITE_EN`: UART signals for serial communication. Notice that `UART_READ_EN` also controls data input of Register File. In case of reading UART, this signals ensures destination register loaded from UART FIFO.

2. **Outputs:**

- `Zero`: The Zero flag output from the ALU (set when ALU result is zero).
- `PC, Instr`: The Program Counter and Instruction register outputs.
- `Debug_out, RF_OUT1, RF_OUT2`: Debugging and register outputs.

- `ALUResult:` The result from the ALU and sent controller to check if UART is used or not.
- `UART_TX:` UART output data for transmission.

Datapath Components and Operation

1. **Program Counter (PC) and PC Logic:**
   - The PC Register stores the current program counter value.
   - The PCAdder adds 4 to the current PC to calculate the next instruction address (PC + 4).
   - The Mux_2to1 (PCSrcMux) chooses the next value for the PC:
     - If PCSrc is 0, the PC continues to the next instruction (PC + 4).
     - If PCSrc is 1, the PC is set to the result of the ALU. Note that this design does not have "PCTargetAdder". Address calculations are made in ALU. Conditions are checked in the Controller Unit.
2. **Instruction Memory:**
   - The Instruction Memory (Instruction_Memory) stores the instructions. It is indexed using the current PC value, and it outputs the instruction to Instr.
3. **Register File:**
   - The Register File stores values for registers.
   - It is addressed by the source registers (rs1 and rs2 from Instr[19:15] and Instr[24:20]) and the destination register (rd from Instr[11:7]).
   - Data to be written back is selected from the ALU result, the PC plus 4 or UART FIFO (RF_WD_SRC, UART_READ_EN).
4. **ALU and ALU Logic:**
   - The ALU performs arithmetic and logic operations between two operands, namely SrcA and SrcB.

  - SrcA is selected from the register file.
  - SrcB is either from the register file or an extended immediate value.
- The result of the ALU operation (ALUResult) is output along with the Zero flag (set to 1 if the result is zero).

5. **Memory:**

- The Data Memory (Data_Memory) is used for reading and writing data from/to memory.
  - MemWrite controls writing to memory.
  - READMODE determines how memory is read (byte, word, etc.).
  - Memory address is provided by the ALU result (ALUResult), and the data to be written comes from RF_OUT2.

6. **Result Selection:**

- The Result Mux (Result_Mux) selects the final result:
  - If ResultSrc is 0, the result comes from the ALU.
  - If ResultSrc is 1, the result comes from memory (ReadData).

7. **Immediate Extension:**

   The Extender module is responsible for generating a 32-bit immediate value from a 32-bit instruction based on the instruction format. Different instruction types encode their immediate fields differently, and this module decodes and extends them appropriately, depending on the select input.

   Inputs:

   - DATA [31:0]: The full 32-bit instruction from which the immediate value is extracted.
   - select [2:0]: A control signal that determines how to interpret and extend the immediate value.

   Output:

- Extended_data [31:0]: The 32-bit output that represents the extended immediate value.

Supported Immediate Types:

(a) SEX12 (select = 000): Sign-extended 12-bit immediate, used for I-type instructions like ADDI and LW. The immediate is taken from bits [31:20] of the instruction and extended to 32 bits by repeating the sign bit (bit 31).

(b) UEX12 (select = 001): Zero-extended 12-bit immediate, potentially used for custom operations. The bits [31:20] are extended to 32 bits by padding with zeros on the upper 20 bits.

(c) B_IMM (select = 010): Used in branch instructions such as BEQ or BNE. The immediate is constructed by concatenating bits [31], [7], [30:25], and [11:8], placing a zero as the least significant bit to ensure word alignment. This results in a signed 13-bit value that is extended to 32 bits.

(d) JALEX (select = 011): Used for JAL (Jump and Link) instructions. The immediate is built from bits [31], [19:12], [20], and [30:21], and includes a zero at the least significant bit. This forms a 21-bit signed offset that is sign-extended to 32 bits.

(e) U_IMM (select = 100): Used in U-type instructions like LUI and AUIPC. The upper 20 bits [31:12] of the instruction are taken and the lower 12 bits are filled with zeros. This effectively performs a logical left shift of 12 bits.

(f) S_IMM (select = 101): Used in S-type instructions like SW. The immediate is formed by combining bits [31:25] and [11:7], which are split in the instruction format. The result is a 12-bit signed value that is sign-extended to 32 bits.

Default Case:

If the select input does not match any of the defined cases, the output is set to 0.

In summary, the Extender module centralizes the logic required to interpret and expand immediate fields across various instruction types, making it a key component in the datapath for supporting arithmetic, memory access, and control flow operations.

8. **UART Interface:**

   - UART Muxes select data between the register file or UART, depending on whether a UART read or write is triggered.
   - The UART communication is handled by the UART Module (uartinstance). This module controls sending and receiving serial data over UART:
     - UART_RX: Input serial data.
     - UART_TX: Output serial data.
     - UART_WRITE_EN: Trigger to start writing data to UART.
     - UART_READ_EN: Trigger to read data from UART.

Data Flow:

1. **PC Calculation:**

   - The PC value is incremented by 4 (the size of instructions). If a branch or jump is taken (PCSrc), the next PC value comes from the Result (ALU or memory).

2. **Instruction Fetch:**

   - The instruction at the current PC is fetched from Instruction_Memory.

3. **Register File Read:**

   - The values for rs1 and rs2 are read from the register file based on the instruction.

4. **ALU Operation:**

   - The ALU performs an operation on SrcA and SrcB. The result is computed and output as ALUResult.

5. **Memory Access:**

   - If the instruction requires reading or writing from memory, the Data_Memory module handles this.

6. **Write Back:**

   - The final result is written back to the register file (RF_WD).

7. **UART Communication:**

   - If UART read or write is enabled (UART_READ_EN or UART_WRITE_EN), data is transferred over the UART interface.

Controller Design

The RTL schematics of Controller can be seen in Figure 3.

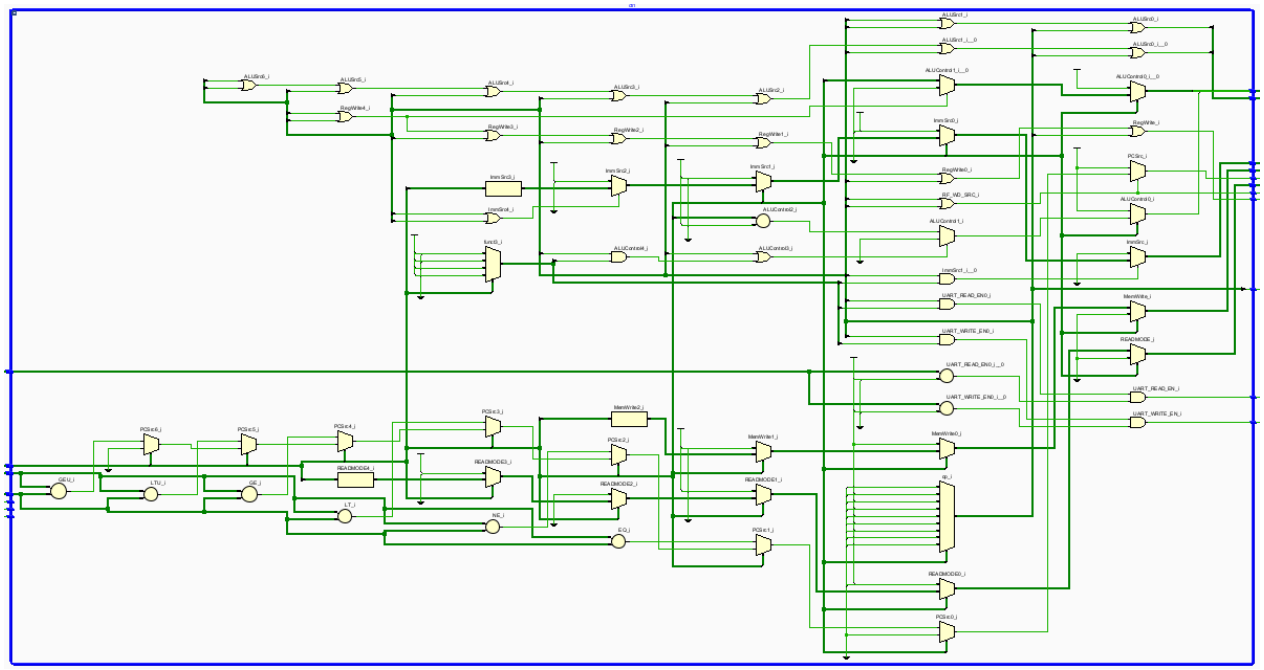

Figure 3: RTL Schematic of Controller

Instruction Execution in the Controller

The Controller module interprets the 32-bit instruction and generates control signals to guide the datapath for correct execution. It does this by decoding the opcode and relevant fields like funct3, funct7, and register addresses. Based on the instruction type, it drives the execution through various control lines:

- **R-type (REG_REG_INSTR):** These instructions use two source registers and perform ALU operations (e.g., ADD, SUB, AND). The ALU operates on RF_OUT1 and RF_OUT2. RegWrite is enabled, ALUSrc is 2'b00 (using both register operands), and ResultSrc is 0 (result comes from ALU). ALUControl is determined by funct3 and funct7. Note that ALU Mapping is changed for ease the ALUControl logic.

- **I-type (REG_IMM_INSTR):** These instructions perform ALU operations between a register and an immediate (e.g., ADDI). ALUSrc indicates the second operand comes from the immediate. The immediate is extended based on ImmSrc, and ALUControl is set using funct3 and funct7. RegWrite is enabled.

- **Load (MEM_LOAD_INSTR):** The ALU computes the memory address using RF_OUT1 and the extended immediate. MemWrite is disabled, ResultSrc is 1 (load result from memory), READMODE is set according to funct3, and RegWrite is enabled.

- **Store (MEM_STORE_INSTR):** These use register and immediate to compute an address. Data in RF_OUT2 is written to memory. MemWrite is enabled according to the byte/half/word specified by funct3. RegWrite is disabled.

- **Branch (BRANCH_INSTR):** The Controller compares RF_OUT1 and RF_OUT2. PCSrc is asserted conditionally based on the result of the comparison and the branch type (funct3). Note that ALU is responsible for calculating the branch target address, not the comparison.

- **JAL:** PCSrc is asserted unconditionally. The return address (PC + 4) is written to rd. ALUSrc is set to use immediate, RF_WD_SRC is asserted to select PC+4, and RegWrite is enabled.

- **JALR:** Similar to JAL but computes jump target as RF_OUT1 + immediate. Control is transferred indirectly. PCSrc and RF_WD_SRC are asserted, and RegWrite is enabled.

- **LUI / AUIPC:** These are upper-immediate instructions. ImmSrc selects a U-type format. ALUControl is configured to pass or add the immediate. RegWrite is enabled.

UART Peripheral Control Signals:

- **UART_READ_EN:** This signal is asserted when a load word (LW) instruction targets memory-mapped address `0x00000404`. It allows reading a byte or word from the UART receive buffer.
- **UART_WRITE_EN:** This signal is asserted when a store byte (SB) instruction targets address `0x00000400`. It enables writing a character to the UART transmit buffer.

These signals act as memory-mapped triggers for communication with a UART device, integrating peripheral I/O into the processor via standard load/store instructions.

UART Peripheral

The Universal Asynchronous Receiver/Transmitter (UART) peripheral was developed to enable serial communication between the RISC-V processor and external systems, such as a host computer. This peripheral adheres to the commonly used 8-N-1 format, which denotes 8 data bits, no parity bit, and 1 stop bit. It operates at a baud rate of 9600 bps, consistent with standard serial communication protocols. The RTL schematics of UART peripheral and FIFO buffer can be seen in Figure 4 and 5 respectively.

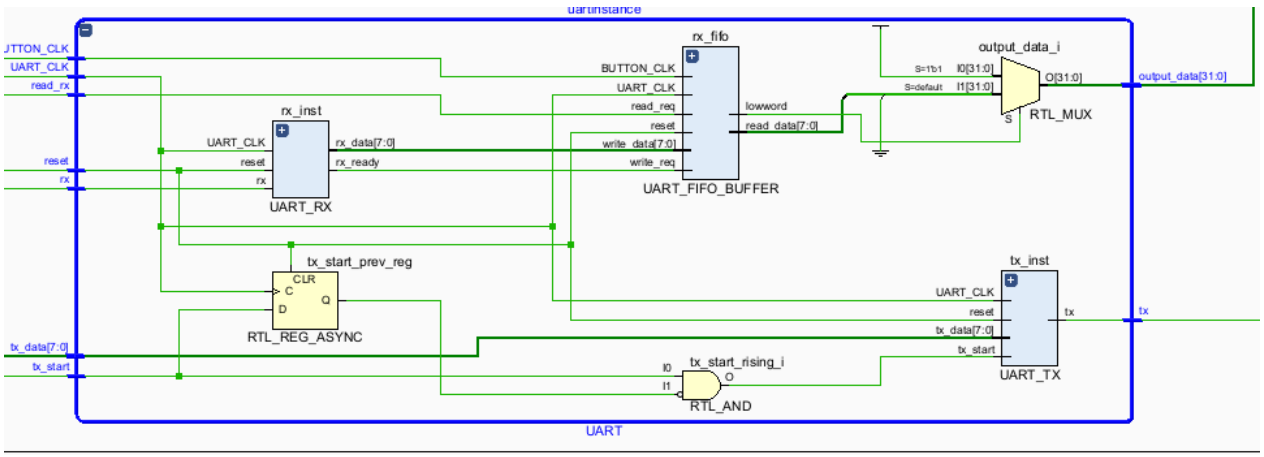

Figure 4: RTL Schematic of UART Peripheral

Architecture Overview

The UART module consists of three main components:

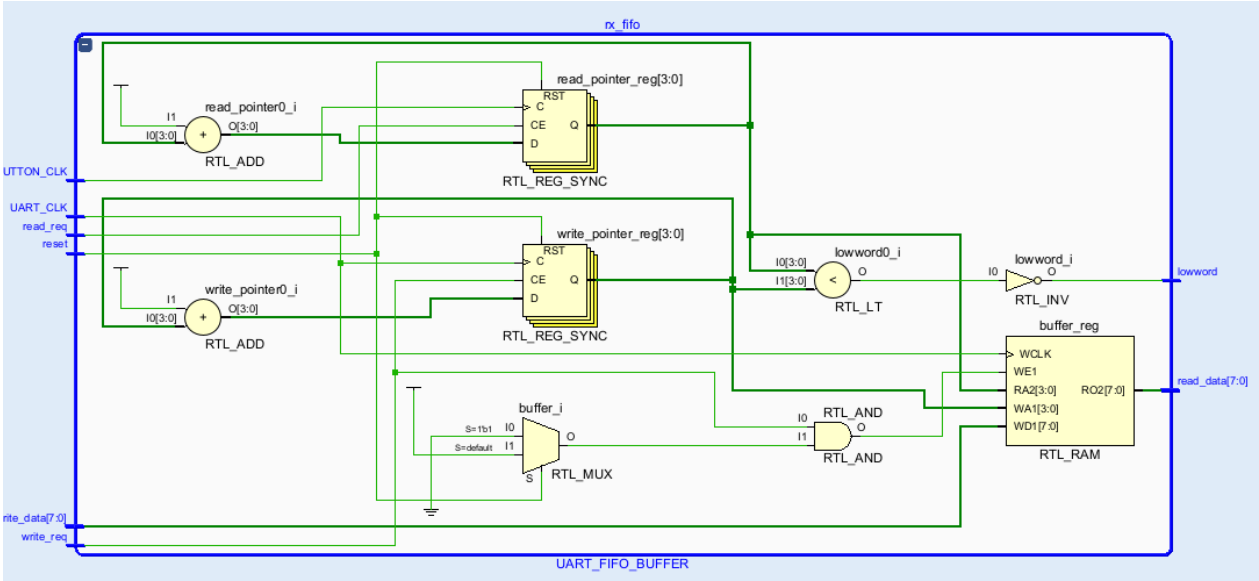

Figure 5: RTL Schematic of UART FIFO Buffer

- **Transmitter** (`UART_TX.v`): Responsible for sending bytes serially over the Tx line. It receives a byte from the processor and transmits it bit-by-bit, starting with a start bit, followed by 8 data bits (LSB first), and ending with a stop bit.
- **Receiver** (`UART_RX.v`): Continuously monitors the Rx line for incoming data. Upon detecting a valid start bit, it samples incoming bits at appropriate intervals and reconstructs the received byte.
- **FIFO Buffer** (`UART_FIFO_BUFFER.v`): Implements a 16-byte First-In-First-Out (FIFO) buffer to store incoming bytes temporarily. This prevents data loss in scenarios where the processor does not immediately read the received data.

These components are integrated into a top-level module (`UART.v`), which manages the transmit and receive processes and interfaces with the processor through memory-mapped I/O.

UART Receiver Finite State Machine

The UART receiver module is governed by a finite state machine (FSM). It converts the serial UART frame into an 8-bit parallel format, ensuring correct timing and bit alignment

based on the 8-N-1 protocol. The RTL schematics of UART RX can be seen in Figure 7.

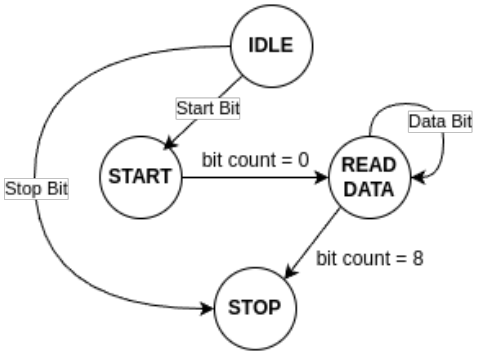

Figure 6: UART Receiver State Diagram

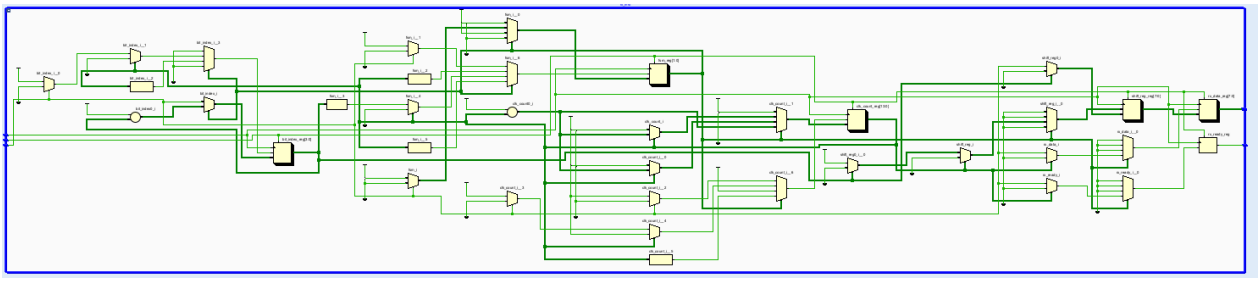

Figure 7: RTL Schematic of UART RX

State Descriptions:

- **IDLE:** This is the default state where the receiver line (`Rx`) is monitored for a low logic level, indicating the beginning of a start bit. The receiver remains in this state as long as the line stays high (logic 1), representing the idle bus state.
- **START:** Upon detecting a falling edge on the `Rx` line, the FSM enters the START state. It then waits for half a baud period before sampling the line to verify that the low level persists, confirming a valid start bit.
- **READ DATA:** If the start bit is valid, the FSM transitions to the READ DATA state. Here, the FSM samples the next 8 bits at regular baud intervals and shifts them into a register. These bits are collected least significant bit (LSB) first to reconstruct the transmitted byte.

- **STOP:** After all 8 data bits are received, the FSM samples the line one more time to ensure the presence of the stop bit (logic 1). If the stop bit is valid, the full byte is pushed to the FIFO buffer, and the FSM returns to the IDLE state.

Timing is critical in the UART receiver. The internal baud rate clock (derived from the system clock) ensures that sampling occurs in the center of each bit period, which is the most stable point for asynchronous data. After successful reception and validation of a UART frame, the 8-bit data is made available to the processor through the FIFO buffer. If the buffer is full, newly received bytes is discarded. This design uses a 16-byte FIFO buffer, as defined in `UART_FIFO_BUFFER.v`, to handle data bursts and prevent loss during processor delays.

UART Transmitter Finite State Machine

The UART transmitter operates as a simple finite state machine (FSM) that serializes an 8-bit data byte into a UART frame consisting of a start bit, 8 data bits, and a stop bit. The transmission follows the 8-N-1 protocol and is controlled by a baud rate timer. The RTL schematics of UART TX can be seen in Figure **??**.

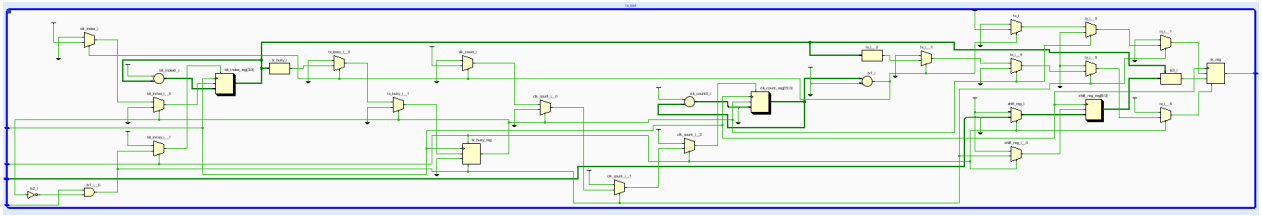

Figure 8: RTL Schematic of UART TX

State Overview:

- **IDLE:** Default state where the `tx` line is held high. When `tx_start` is asserted and the transmitter is idle, the FSM loads the transmission frame and transitions to the START state.
- **START:** Sends the start bit (logic 0) and waits one baud interval.
- **SEND DATA:** Transmits the 8 data bits serially, LSB first, one per baud cycle.

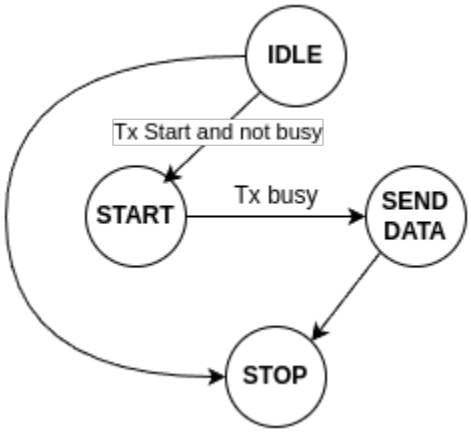

Figure 9: UART Transmitter State Diagram

- **STOP:** Sends the stop bit (logic 1), then returns to IDLE.

Each bit is held for a fixed number of clock cycles defined by `clock_baudrate_fix`, ensuring proper timing. The entire frame is preloaded into a 10-bit shift register, and a counter tracks the bit index. Once all bits are sent, the transmitter resets and waits for the next start signal.

Memory-Mapped Interface

To enable interaction between the processor and the UART peripheral, specific memory addresses are reserved:

- **Transmit Register** (`0x00000400`): Writing a byte to this address initiates transmission. The transmitter must be idle before a new byte is written; otherwise, behavior is undefined.
- **Receive Register** (`0x00000404`): Reading from this address returns the oldest byte in the FIFO buffer. If the buffer is empty, the value `0xFFFFFFFF` is returned, indicating no new data has been received.

The processor uses standard store and load instructions (`SB` and `LW`) to interact with these registers, making UART communication simple from a software perspective.

Operation and Timing

The UART operates asynchronously, meaning there is no shared clock between transmitter and receiver. Instead, both sides agree on the baud rate in advance. Internally, the transmitter and receiver use clock dividers to generate timing that matches the desired baud rate.

For transmission, the processor writes a byte to the transmit address. The transmitter appends start and stop bits, then shifts the data out serially. For reception, the receiver samples the Rx line at the midpoint of each bit interval to correctly reconstruct the incoming byte. The reconstructed byte is pushed into the FIFO buffer if space is available.

Testbench

To ensure the correctness of the RISC-V processor, a testbench was developed using the `cocotb` framework. It simulates instruction execution and compares results from a Python-based reference model with the Verilog hardware design.

Testbench Components

- **RISCV-32I_Test.py** – Main cocotb test file that manages simulation flow, clock, reset, and result comparison.
- **Helper_lib.py** – Provides instruction parsing, memory model, and helper functions.
- **Helper_Student.py** – Logs datapath and control signals; decodes instructions for readability.
- **Instructions.hex / Instructions.txt** – Encoded and human-readable instruction sets used in simulation.

Simulation Process

After initializing the DUT and applying reset, the testbench:

1. Loads instructions from `Instructions.hex`.

2. Simulates execution in both hardware and a Python model.

3. Compares register file and PC at every clock cycle.

Assertions are used to detect mismatches between hardware and the reference model.

Instruction Set Coverage

Tested instructions include:

- **Arithmetic/Logic:** `add`, `sub`, `xor`, `or`, `and`, `sll`, `srl`, `sra`
- **Immediate:** `addi`, `andi`, `ori`, `xori`, `slti`, `sltiu`, `slli`, `srli`, `srai`
- **Comparison:** `slt`, `sltu`
- **Load/Store:** `lw`, `lh`, `lhu`, `lb`, `lbu`, `sw`, `sh`, `sb`
- **Control Flow:** `beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`, `jal`, `jalr`, `auipc`, `lui`

Simulation Output

The testbench effectively validates functional correctness and catches implementation errors by comparing expected and actual outputs per instruction.

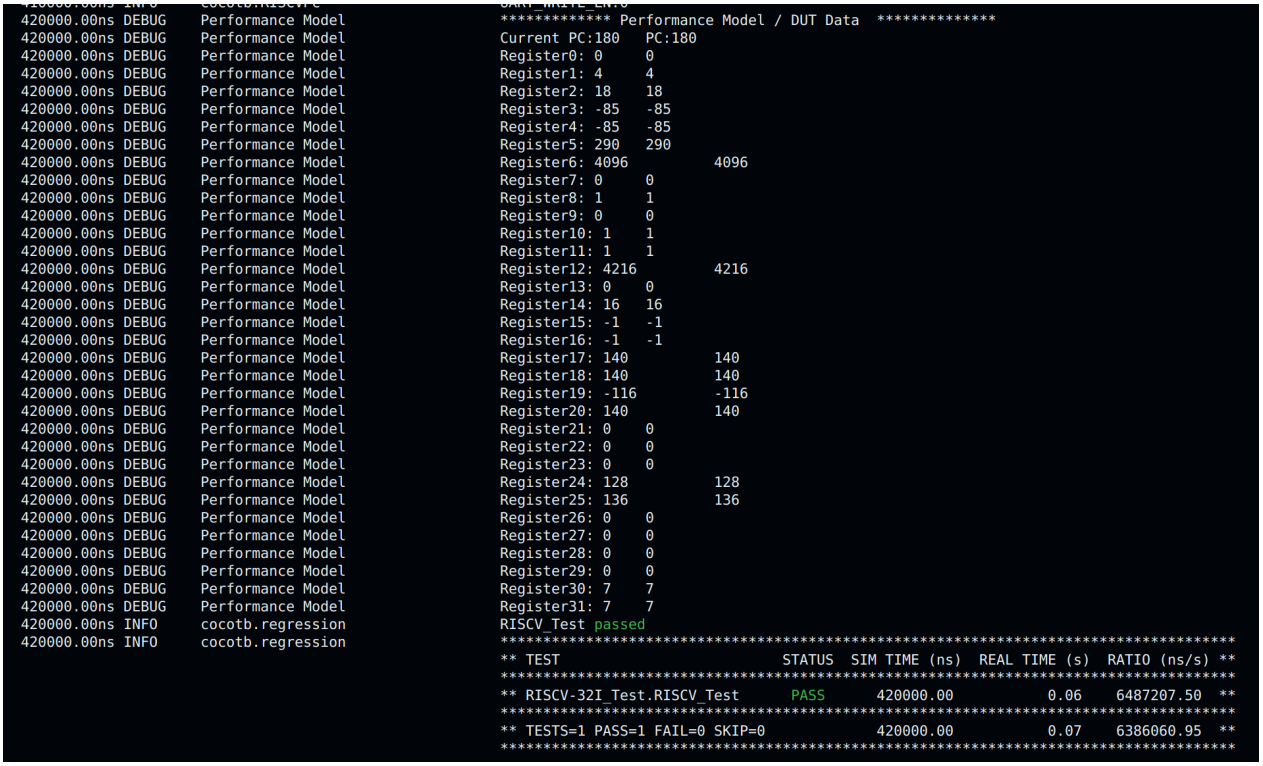

Figure 10: Testbench Result