# Middle East Technical University
# EE446 Computer Architecture II
# Term Project Report

Zülal Uludoğan, 2444057

Ahmet Taha Çelik, 2515831

25 May 2025

# Contents

# Chapter 1

# Introduction

## 1.1 Project Description

This project involves the implementation of a single-cycle RISC-V processor using Verilog HDL, supporting core RV32I instructions such as arithmetic, logic, load/store, and control flow operations. The processor is built in a modular way, with separate units for the datapath, controller, register file, memory, ALU, and UART. The list of supported instructions is shown in Table 1.1. See [1] and [3] more details.

| | |
|---|---|
| Arithmetic instructions: | ADD[I], SUB |
| Logic instructions: | AND[I], OR[I], XOR[I] |
| Shift instructions: | SLL[I], SRL[I], SRA[I] |
| Set if less than: | SLT[I][U] |
| Conditional branch: | BEQ, BNE, BLT[U], BGE[U] |
| Unconditional jump: | JAL, JALR (Return-address stack push/pop functionality will not be implemented) |
| Load: | LW, LH[U], LB[U] |
| Store: | SW, SH, SB |
| Others: | LUI, AUIPC |

Table 1.1: List of Instructions

The design focuses on correct instruction decoding, datapath control, and memory-mapped I/O integration. The UART module enables serial communication by assigning specific load/store instructions to dedicated addresses, and handles synchronization between different clock domains using FIFO buffers.

For verification, a Python-based performance model of the processor is developed and simulated using Cocotb. This model checks instruction behavior cycle-by-cycle and compares the internal states of the Verilog DUT with the expected results, allowing precise debugging and validation of processor functionality. The project demonstrates key concepts in RISC-V architecture, digital system design, and hardware verification through software-driven simulation.

# Chapter 2

# Datapath Module

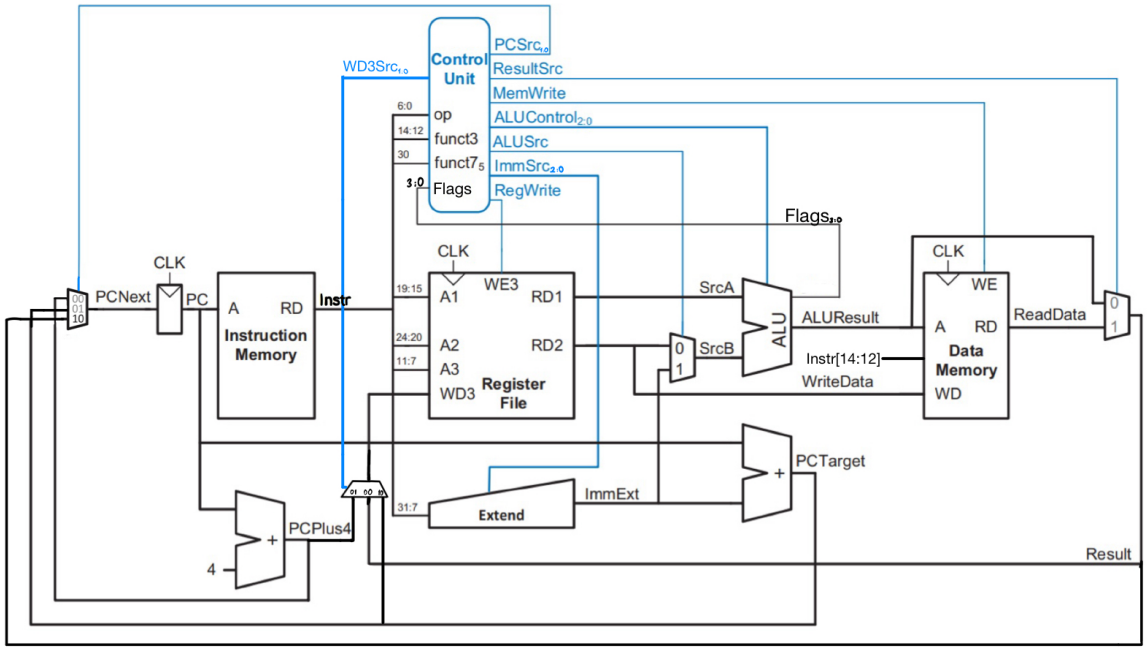


Figure 2.1: Modified RISC-V Single Cycle Computer

The datapath module is responsible for executing instructions defined in the RISC-V ISA, supporting the instruction types listed in Table 1.1. Most submodules were adapted from laboratory components, but modifications were made to support specific instruction semantics and to align with the RISC-V architecture. A synchronous reset (`RESET`) is integrated into the module to ensure all registers and memory components are properly initialized.

Unlike ARM, where 16 general-purpose registers are used, the RISC-V register file consists of 32 registers. Register `x0` is hardwired to zero, as per the RISC-V specification. A 5-to-32 decoder and a 32-to-1 multiplexer are used to handle register selection and writing. The `Register_file` module supports debug output and dual read ports, along with a single write port controlled by the `RegWrite` signal.

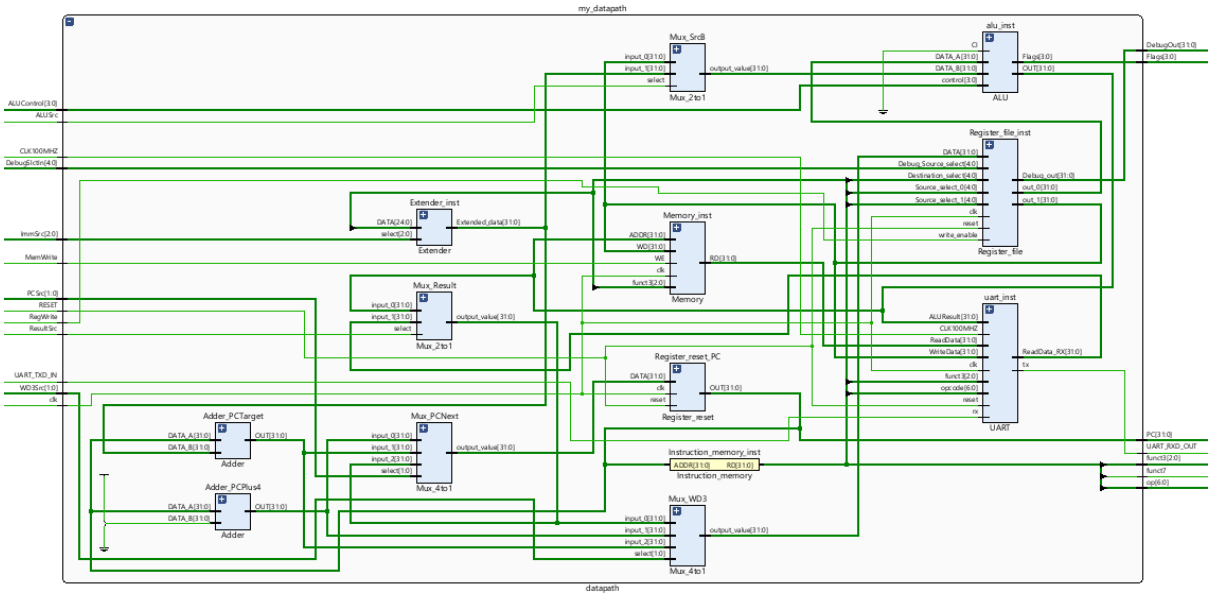


Figure 2.2: Synthesized RTL view of the datapath module

The datapath is designed to handle all six RISC-V instruction types—R-type, I-type, S-type, B-type, U-type, and J-type—by routing operands and control signals through shared hardware blocks and dedicated control logic. Each instruction type makes use of a subset of datapath components to perform its operation.

**R-type Instructions** R-type instructions perform register-to-register arithmetic and logical operations. The instruction provides two source registers (`rs1` and `rs2`) and a destination register (`rd`). The values from `rs1` and `rs2` are read from the register file and passed to the ALU. The ALU performs the operation specified by the `funct3` and `funct7` fields (e.g., `ADD`, `SUB`, `SLL`), and the result is written back to `rd` through the `WD3` multiplexer.

**I-type Instructions** I-type instructions include immediate arithmetic/logical operations (e.g., `ADDI`, `ANDI`), loads (e.g., `LW`, `LB`), and `JALR`. These instructions use one source register and an immediate. The immediate is sign-extended by the `Extender` module, then used as an operand in the ALU or as an offset for memory access. For arithmetic instructions, the ALU result is written back to `rd`. For load instructions, the data is read from memory and passed through the `Result` multiplexer to be written to the register file.

**S-type Instructions** S-type instructions are store instructions such as `SW`, `SH`, and `SB`. These use two source registers: `rs1` provides the base address, and `rs2` provides the data to be stored. The immediate is generated by concatenating two fields and sign-extended. The memory address is computed by adding the immediate to `rs1`, and the result is used by the memory module to perform the store.

**B-type Instructions** B-type instructions are conditional branches such as `BEQ`, `BNE`, and `BLT`. The two source registers are compared using the ALU, which performs subtraction and sets condition flags. If the branch condition is met, the program counter

is updated to `PC + immediate`; otherwise, it proceeds sequentially. The `PCSrc` signal controls which address is selected via the `PCNext` multiplexer.

**U-type Instructions** U-type instructions such as `LUI` and `AUIPC` use a 20-bit immediate value shifted left by 12 bits. For `LUI`, the immediate is written directly into the destination register. For `AUIPC`, the immediate is added to the current PC and the result is written back to `rd`. The extended immediate is generated by the `Extender` and routed through the `Mux_WD3` path.

**J-type Instructions** J-type instructions, specifically `JAL`, perform an unconditional jump to a target address calculated from the PC and a 21-bit immediate. Before jumping, the return address (`PC + 4`) is saved into `rd`. The PC update logic and the `Mux_WD3` handle this behavior, and the control signals ensure the correct values are selected and stored.

Each instruction type is supported through coordinated control of multiplexers, the ALU, the register file, the extender, and memory modules, all synchronized by the controller to ensure correct execution paths across the datapath.

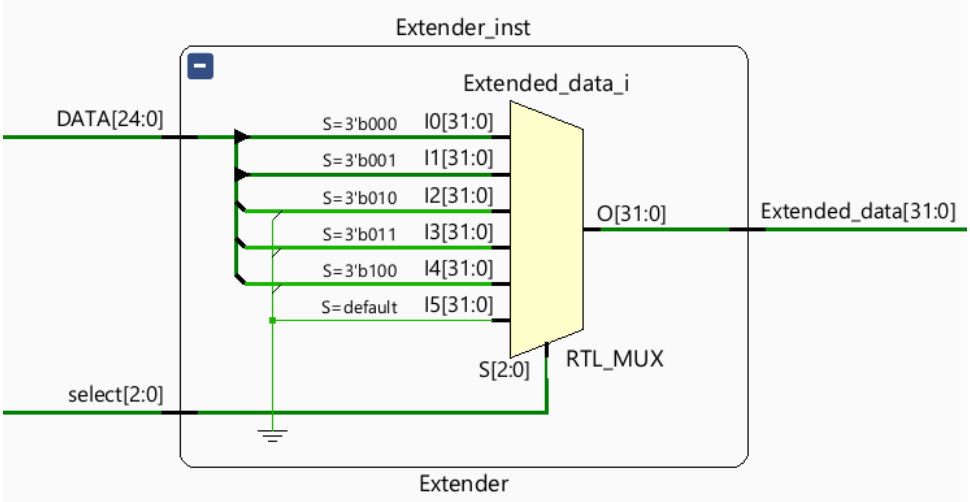


Figure 2.3: Synthesized RTL view of the extender module

All arithmetic and logic operations, including shift operations, are implemented inside the `ALU` module. Shift instructions use immediate values or register contents as shift amounts, and the ALU supports logical (`SRL`) and arithmetic (`SRA`) shifts accordingly. For branch instructions such as `BEQ` and `BNE`, the ALU performs a subtraction to set condition flags, which are later evaluated in the control path to determine branch decisions.

The `Memory` module is adapted to support byte (`LB`/`SB`), halfword (`LH`/`SH`), and word (`LW`/`SW`) memory accesses. It uses the `funct3` field of the instruction to determine the access width and whether sign-extension or zero-extension should be applied to loaded data. For store operations, it selectively writes the relevant portion (byte/halfword/word) of the register data into memory without modifying adjacent memory locations.

# Chapter 3

# Controller Module

The controller module is responsible for generating the control signals necessary to guide the datapath during the execution of RISC-V instructions. As shown in Figure 3.1, the controller design follows a modular approach and includes three functional units: the `MainDecoder`, the `ALUDecoder`, and the `PCLogic` unit.

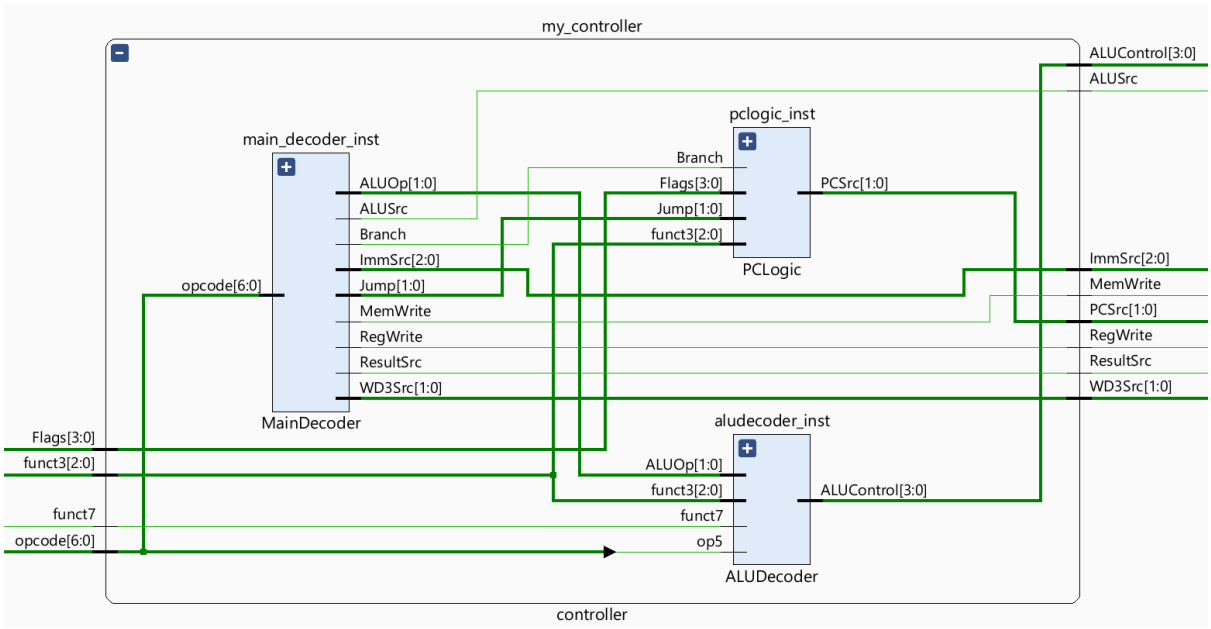


Figure 3.1: Synthesized RTL view of the controller module

The `MainDecoder` interprets the 7-bit opcode from the instruction and produces the primary control signals required by the datapath, such as register file write enable, memory write enable, ALU source selection, and immediate format selection. These signals are then distributed to the corresponding components to orchestrate the processor's behavior.

The `ALUDecoder` further refines control signal generation by examining the `funct3`, `funct7`, and other opcode-specific bits to determine the specific ALU operation to perform. It outputs a 4-bit `ALUControl` signal, ensuring the correct arithmetic or logical operation is selected based on the instruction type.

The PCLogic unit handles program counter selection. It considers both the branching conditions and the jump instructions to compute the correct value of the PCSrc signal. This logic is critical for implementing both conditional and unconditional control-flow changes in the processor.

## 3.1 Instruction Execution in the Controller

The controller governs instruction execution by generating the appropriate control signals that drive datapath components. Its architecture is modular and consists of three submodules: MainDecoder, ALUDecoder, and PCLogic. Each module has a specific responsibility in decoding the instruction and enabling the correct system behavior.

The MainDecoder identifies the instruction type based on its 7-bit opcode. Local parameters are defined to represent each category of instruction — including R-type, I-type (with subtypes for ALU immediates, loads, and JALR), S-type, B-type, U-type (LUI, AUIPC), and J-type (JAL). The decoder assigns values to high-level control signals such as RegWrite, MemWrite, ALUSrc, ResultSrc, and ImmSrc. For example, R-type instructions enable register writing and set the ALU to use register operands. Load instructions assert both ALUSrc = 1 and RegWrite = 1, and select the memory read result as the register write-back source. Store and branch instructions disable RegWrite but assert either MemWrite or Branch accordingly.

Once the instruction type is determined, the ALUDecoder refines the ALU operation by interpreting the funct3, funct7, and ALUOp values. The ALUOp signal is set by the MainDecoder to broadly classify operations (e.g., arithmetic, logical, or branching), while the ALUDecoder translates this into a precise 4-bit ALUControl signal that selects operations like addition, subtraction, shift left/right, set-less-than, and logical bitwise operations. For instance, funct3 = 000 and funct7 = 0100000 are used together to distinguish SUB from ADD. Similarly, shift instructions are detected by decoding funct3 patterns and checking the shift amount interpretation via funct7.

Control flow instructions such as branches and jumps are managed by the PCLogic module. This unit uses the Branch and Jump signals from the MainDecoder, along with the instruction type and ALU flags, to determine whether the program counter should be updated with a new address. For conditional branches, PCSrc is asserted only if the branching condition is met, based on comparisons like equality, less-than, or greater-than. For unconditional jumps (e.g., JAL, JALR), PCSrc is directly activated and the appropriate jump target is selected. The PCLogic ensures that changes to control flow are performed only when the instruction semantics and execution conditions are fully satisfied.

Together, these three submodules decompose the control problem into manageable layers: high-level instruction decoding, detailed ALU operation decoding, and control flow decision-making. This layered design supports modularity, simplifies debugging, and ensures correct control signal generation for all instruction types in the RV32I instruction set.

# Chapter 4

# UART Module

The UART (Universal Asynchronous Receiver-Transmitter) module is responsible for facilitating serial communication between the RISC-V processor and external systems. Its implementation involved significantly more complexity than the controller module due to the nature of asynchronous communication, timing constraints, and the need to bridge two separate clock domains. Figure 4.1 shows the synthesized RTL view of the UART subsystem.

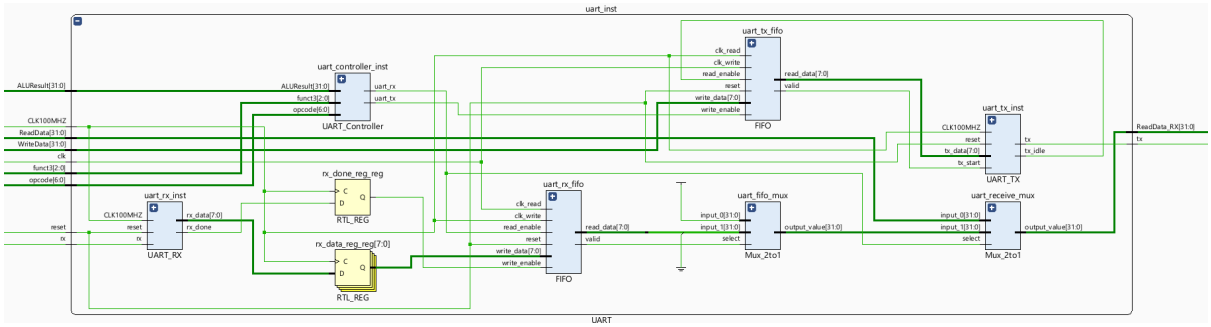


Figure 4.1: Synthesized RTL view of the UART module

## 4.1 Design Overview

The UART module integrates both transmitting and receiving functionalities while adhering to the 9600 baud 8-N-1 protocol. Memory-mapped I/O is used to send and receive bytes via store and load instructions to addresses `0x400` and `0x404`, respectively. When a store is made to `0x400`, a byte is written to the transmitter FIFO and scheduled for transmission. When a load is performed from `0x404`, the system reads the most recent byte from the receive FIFO; if no byte is available, the value `0xFFFFFFFF` is returned.

The UART controller examines the current instruction's opcode, `funct3`, and ALU result to determine whether a UART send or receive should be triggered. This controller outputs `uart_tx` or `uart_rx` flags accordingly.

## 4.2 Clock Domain Challenges and FIFO Synchronization

One of the key challenges in designing the UART module was handling the presence of two independent clocks: a 100 MHz clock (`CLK100MHZ`) used for UART timing, and a slower user-controlled system clock (`clk`) used by the processor core and instruction sequencing. These clocks operate asynchronously, making it unsafe to directly transfer data across domains without proper synchronization.

To address this, the design uses dual-clock FIFO buffers for UART transmission and reception, each operating with distinct read and write clocks. Specifically, the `rx_fifo` writes incoming bytes using `CLK100MHZ`, since it is fed by the UART receiver module, which samples serial input at high speed. Its read operations are driven by the processor's `clk`, as the register file reads received data at the system clock rate. Conversely, the `tx_fifo` receives data from the processor via `clk` during store operations to address `0x400`, and its contents are transmitted by the UART transmitter at `CLK100MHZ`.

This deliberate separation ensures safe and reliable data movement between asynchronous domains. Without this dual-clock setup, writing data from one domain and reading it from another could cause metastability or timing hazards, potentially corrupting the data. By isolating read and write logic through clock-specific ports in each FIFO, the design maintains correctness and stability throughout UART communication.

## 4.3 UART Transmitter and Receiver

The transmitter module initiates data transmission when a byte is enqueued into the FIFO and the line is idle. It serializes the data byte according to UART protocol and asserts the `tx_idle` signal once transmission is complete. If the FIFO holds multiple bytes, they are transmitted sequentially without processor intervention.

The receiver module continuously samples the input `rx` line using `CLK100MHZ`. When a valid start bit is detected, it begins capturing bits until the full byte is received. A `done` flag is raised once reception is complete. The received byte is then pushed into the 16-byte FIFO buffer, ensuring that no bytes are lost if the processor is not immediately ready to read.

## 4.4 UART Transmitter and Receiver State Machines

Both the UART transmitter and receiver modules are implemented as finite state machines (FSMs) that operate on the 100 MHz clock and comply with the 9600 baud 8-N-1 UART protocol. Their behaviors are structured as sequential state transitions, and the FSM logic ensures the correct serialization and deserialization of data.

### 4.4.1 Transmitter State Machine

The transmitter FSM begins in the `IDLE` state, where it waits for the `start` signal to be asserted, indicating that a new byte is ready for transmission. Upon activation, it transitions to the `START` state and drives the transmission line low to indicate the start bit. After holding the line low for one baud period, the FSM enters the `DATA` state, during which each of the 8 data bits is shifted out sequentially, least significant bit first.

Once all data bits are transmitted, the FSM transitions to the `STOP` state. It then drives the line high for one baud period to signal the stop bit. After this, the FSM returns to the `IDLE` state, sets the `idle` flag to indicate readiness, and awaits the next start signal.

This structure ensures that every byte is framed with a start and stop bit as per UART standards, and the `idle` output allows external logic to determine when the transmitter is available for the next byte.

### 4.4.2 Receiver State Machine

The receiver FSM also begins in an `IDLE` state, continuously monitoring the input `rx` line. Upon detecting a falling edge (start bit), it transitions to the `START` state and waits half a baud period to sample the start bit in the middle for noise immunity.

After verifying the start bit, it transitions to the `DATA` state, where it samples 8 data bits at one baud interval each. These bits are shifted into a register, with the least significant bit arriving first.

Following the data bits, the FSM enters the `STOP` state to sample the stop bit. If the stop bit is valid (logic high), the FSM sets the `done` flag and updates the `data_out` register with the fully received byte. It then returns to the `IDLE` state to prepare for the next frame.

This approach ensures that bytes are received reliably and with proper framing. The `done` signal is used to indicate reception completion, allowing external logic to push the received byte into a FIFO buffer or register.

## 4.5 Peripheral Interface and Integration Strategy

The UART module is designed to integrate seamlessly with the existing processor datapath, requiring minimal changes to the overall system architecture. It relies entirely on memory-mapped communication, interpreting store and load instructions to specific addresses as UART transmit and receive operations, respectively. This eliminates the need for additional global control signals or structural changes outside the UART module itself.

A key feature of the integration is the `ReadData_RX` signal, which is an output of the UART module. This signal is dynamically assigned based on whether the currently executed instruction is a UART receive. If the instruction is a regular memory load, `ReadData_RX` simply propagates the value from the data memory, making it identical to `ReadData`. However, if the instruction is a UART receive, the module substitutes the memory output with a value retrieved from the UART receive FIFO, either the latest received byte or a default fallback value if the buffer is empty.

This design allows the UART to be embedded into the system with minimal modification to the datapath. Specifically, the only change required is at the result selection stage, where the processor chooses between the ALU result and the output from memory (or UART). Since this behavior is already governed by an existing multiplexer, no additional control logic is needed. This elegant interface enables UART communication to function transparently within the single-cycle RISC-V system while preserving the simplicity and modularity of the original datapath.

# Chapter 5

# Cocotb Testbench

The Cocotb testbenches used in the laboratories of this course, originally designed for ARM architectures, were modified to support the RISC-V instruction set by adapting both the instruction decoder and the performance model components of the test framework. These modifications enabled cycle-accurate simulation of a single-cycle RISC-V processor and allowed for automated verification against a Verilog-based DUT (Device Under Test).

In the RISC-V architecture, each instruction is composed of standard fields such as `opcode`, `rd`, `funct3`, `rs1`, `rs2`, and `funct7`, which differ from ARM's encoding structure. To handle this, a Python class was implemented to parse each 32-bit hexadecimal instruction. The instruction is first converted to a binary string, from which the fields are extracted based on bit positions defined by the RISC-V specification. Additionally, depending on the instruction type (I-type, S-type, B-type, U-type, or J-type), the immediate values are sign-extended accordingly. Instruction decoding is performed using conditional logic that checks combinations of `opcode`, `funct3`, and `funct7` to identify the exact operation (e.g., `ADD`, `SUB`, `LW`, `BEQ`). The instruction memory content used for verification is partially shown in Figure 5.1. To obtain the hexadecimal-encoded instructions for the testbench, the RISC-V instruction converter tool was used [2].

```
0x00:   93 00 10 34             addi x1, x0, 0x341   r1: 0x341
0x04:   13 f1 c0 cc             andi x2, x1, 0xccc   r2: 0x040
0x08:   b3 81 20 40             sub x3, x1, x2   r3: 0x301
0x0C:   33 62 31 00             or x4, x2, x3    r4: 0x341
0x10:   93 82 90 01             add x5, x1, 25   r5: 0x35a
0x14:   13 13 21 00             slli x6, x2, 2   r6: 0x100
0x18:   23 00 10 40             sb x1, 0x400(x0) send A
0x1C:   23 00 50 40             sb x5, 0x400(x0) send Z
0x20:   83 23 40 40             lw x7, 0x404(x0) receive byte1
0x24:   03 24 40 40             lw x8, 0x404(x0) receive byte2
```

Figure 5.1: Some part of instructions used for the test

To simulate processor behavior, a performance model was developed in Python. This model maintains a register file and a program counter and mimics the functional behavior of the single-cycle processor. A memory abstraction is used to replicate byte-addressable memory and support load and store instructions. The simulation begins by reading instructions from the input file `Instructions.hex`, reversing their endianness to match the hardware layout, and decoding them using the instruction class. Each instruction is

then executed by the performance model, and its effects on the register file and program counter are tracked.

At each cycle, the testbench fetches the next instruction, logs its decoding, performs the corresponding operation in the software model, and compares the result with the output of the Verilog design. This includes checking all 32 registers and the PC value. For memory operations, data is either written to or read from the memory abstraction, depending on whether the instruction is a load or store. For control flow instructions such as `BEQ`, `BNE`, `JAL`, and `JALR`, the program counter is conditionally updated in accordance with the instruction semantics.

In the testbench, UART read operations are simulated by performing a load instruction from the memory-mapped address `0x00000404`, which corresponds to the UART receive buffer. During normal operation, this address returns the most recently received byte. However, if no data has been received, the UART module is designed to return the default value `0xFFFFFFFF`. Therefore, within the testbench logic, it is essential to check whether the value read from `0x00000404` equals `0xFFFFFFFF`, as this indicates that the receive FIFO is empty and no valid byte is available.

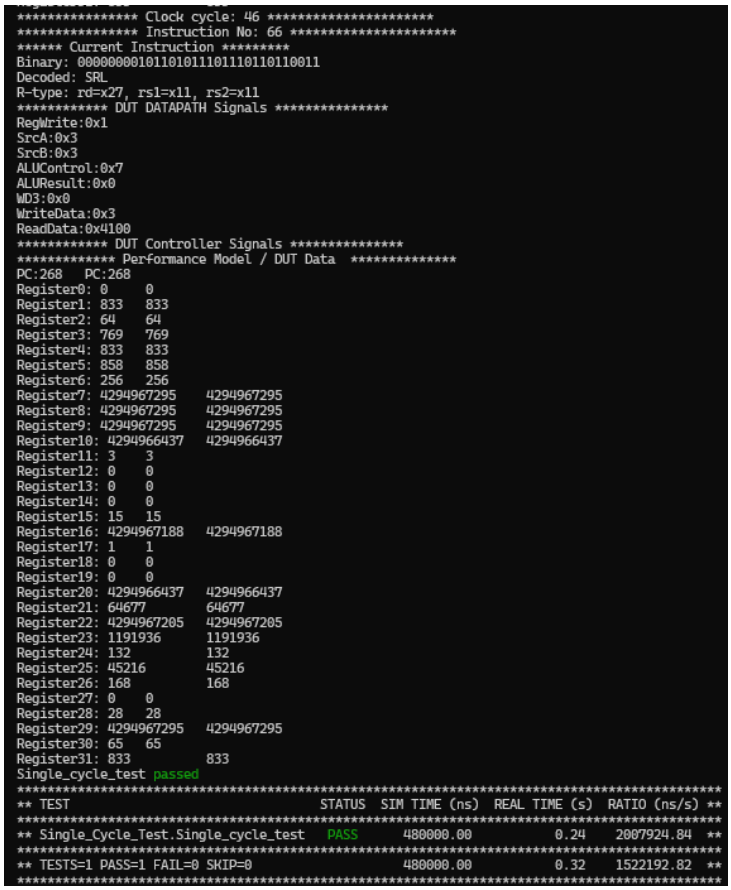


Figure 5.2: An example of testbench result

# Chapter 6

# Conclusion

In this project, a single-cycle RISC-V processor was successfully designed, implemented, and verified using Verilog HDL and Cocotb. The architecture supports a representative subset of the RV32I instruction set, including arithmetic, logic, memory, and control flow operations. All components—including the datapath, controller, ALU, memory, and UART module—were developed with a modular design philosophy to ensure flexibility and scalability.

A key achievement of this work is the integration of a memory-mapped UART module capable of transmitting and receiving data via serial communication. This involved careful management of asynchronous clock domains using dual-clock FIFOs and ensured robust data transfer across the processor and UART interfaces.

Verification was accomplished using a custom Python-based performance model and Cocotb testbenches. This approach enabled detailed cycle-by-cycle comparisons between the behavioral model and the Verilog design, helping to identify and resolve functional mismatches during development. The use of Cocotb also facilitated modular testing and improved debugging efficiency.

Overall, this project provided hands-on experience with the fundamentals of processor architecture, digital system design, and verification methodologies. Through this work, we gained deeper insights into the RISC-V ISA, HDL-based implementation techniques, and modern verification practices using software-hardware co-simulation.

# Bibliography

[1] Sarah L. Harris and David Money Harris. *Digital Design and Computer Architecture: RISC-V Edition*. Morgan Kaufmann, 2021.

[2] *RISC-V Instruction Decoder - rvcodecjs*. `https://luplab.gitlab.io/rvcodecjs/`. Accessed May 2025.

[3] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. `https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view`. Version 20240411. 2024.