EE446 – Term Project
Çetincan Önelge – 2375426
Elif Ceren Yılmaz – 2517290

446 Term Project – Report

2.2: Datapath Implementation

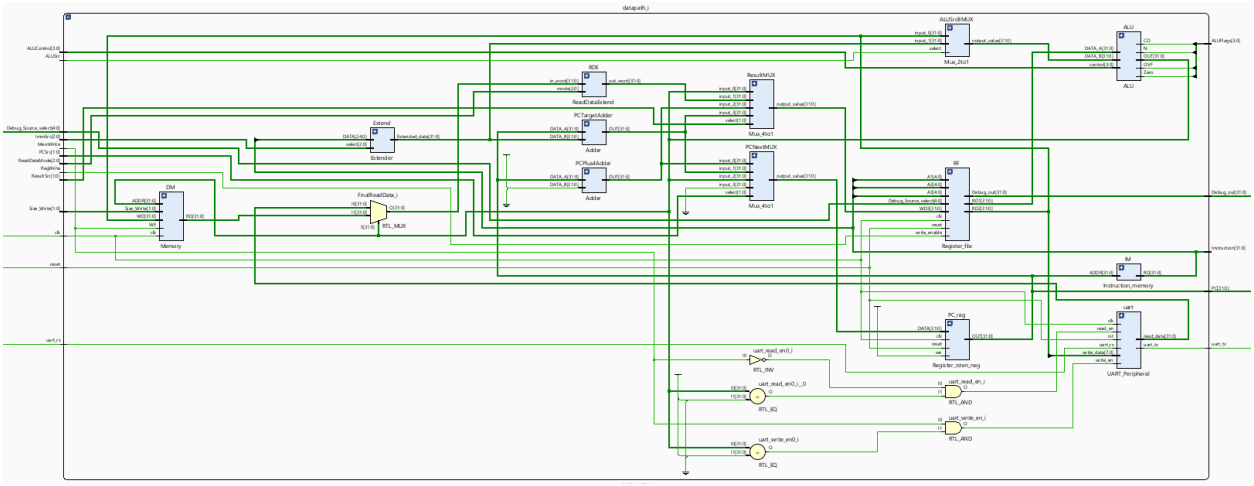

Figure 1: Datapath's RTL Schematic

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imme[11:0]		rs1	funct3	rd	opcode	I-type
imme[11:5]	rs2	rs1	funct3	imme[4:0]	opcode	S-type
imme[12|10:5]	rs2	rs1	funct3	imme[4:1|11]	opcode	B-type
imme[31:12]				rd	opcode	U-type
imme[20|10:1|11|19:12]				rd	opcode	J-type

Figure 2: RV32I ISA

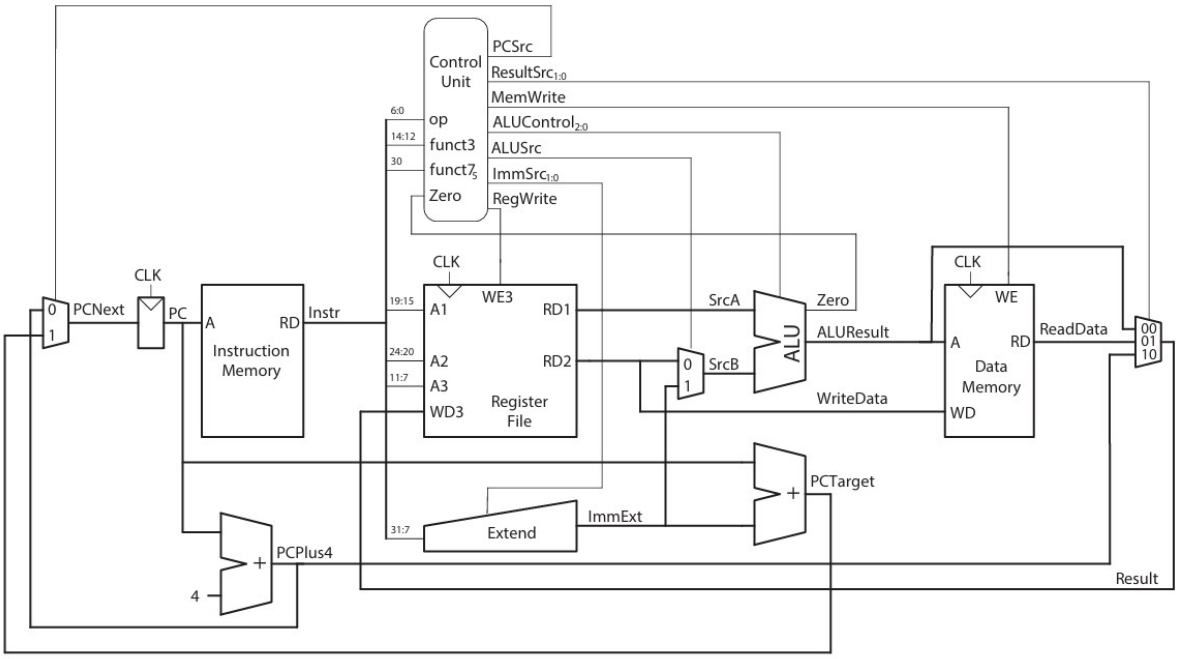

Figure 3: Datapath of Single Cycle RV32I

Datapath Explanations

Below is how our single-cycle DATAPATH implements each of the six RV32I instruction formats using the blocks and signals in the provided Verilog.

R-Type (e.g. ADD, SUB, AND, OR, SLT, XOR, SLL, SRL, SRA):

On every cycle the PCReg holds the current program counter, driven by PCNextMUX (select=0 → PCPlus4 from PCPlus4Adder). The fetched 32-bit Instruction feeds the Register_file, producing registers RD1 and RD2. Because ALUSrc=0, ALUSrcBMUX selects RD2, and ALUControl (decoded from funct3/funct7) steers the ALU to compute RD1 ∘ RD2, yielding ALUResult and flags (Zero, N, CO, OVF). ResultMUX (select=0) forwards ALUResult into WD3; when RegWrite=1 the result is written back into rd. MemWrite=0 so DataMemory is idle.

I-Type (e.g. ADDI, SLTI, ORI, XORI, SLLI, SRLI, LW, LH, LB, JALR):

The Extender unit sign-extends Instruction[31:7] per ImmSrc into ImmExt. For arithmetic-immediates and shift-immediates ALUSrc=1 causes ALUSrcBMUX to select ImmExt; the ALU then computes RD1 + ImmExt under ALUControl, and ResultMUX (select=0) routes ALUResult to WD3. For loads, the ALU computes the address in the same way but ResultMUX (select=1) selects ReadDataExtended from the ReadDataExtend block (fed by DataMemory). MemWrite remains 0. For JALR, ResultMux (select=2) returns PCPlus4 to WD3 and PCSrc=2 causes PCNextMUX to pick ALUResult as the next PC (with bit0 cleared).

S-Type (SB, SH, SW):
ImmExt provides the signed store offset. ALUSrc=1 selects ImmExt; the ALU computes addr = RD1 + ImmExt. RD2 carries the value to store. When MemWrite=1 DataMemory (DM) writes the low 8/16/32 bits of RD2 (as controlled by Size_Write) at address ALUResult. ResultMUX and RegWrite are disabled, leaving the register file unchanged. PCNextMUX continues to drive PCPlus4.

B-Type (BEQ, BNE, BLT, BGE, BLTU, BGEU):
Extender fuses the branch-type immediate fields into ImmExt. ALUSrc=0 selects RD2, so the ALU computes RD1 − RD2 to set its Zero and N flags. The Controller decodes funct3 to generate PCSrc=1 when the branch condition is true. PCNextMUX then selects PCTarget (PC + ImmExt) instead of PCPlus4, effecting the branch. No register or memory writes occur.

U-Type (LUI, AUIPC):
The upper immediate bits are placed into ImmExt (low 12 bits zero). For LUI the instruction's rs1 field is 0, so RD1=0 and ALUSrc=1 yields ALUResult=ImmExt. For AUIPC the immediate is also applied via PCTargetAdder (DATA_A=PC, DATA_B=ImmExt) and the ResultMux (select=3) can forward PCTarget into WD3. In both cases RegWrite=1 writes the immediate-derived value into rd, and PC advances by four.

J-Type (JAL):
The J-type immediate is sign-extended into ImmExt. PCPlus4 is computed in parallel. ResultMux (select=2) forwards PCPlus4 into WD3 to save the return address. PCSrc=1 causes PCNextMUX to select PCTarget (PC + ImmExt) as the next PC. RegWrite=1 writes the link address into rd; no memory access occurs.

Each format simply exercises a different path through the same register file, ImmGen (Extender), ALU, DataMemory, and the two multiplexers (ALUSrcBMUX and ResultMUX), with PCNextMUX choosing among PC+4, branch/jump targets, or ALU+JALR targets.

2.3: Controller Implementation

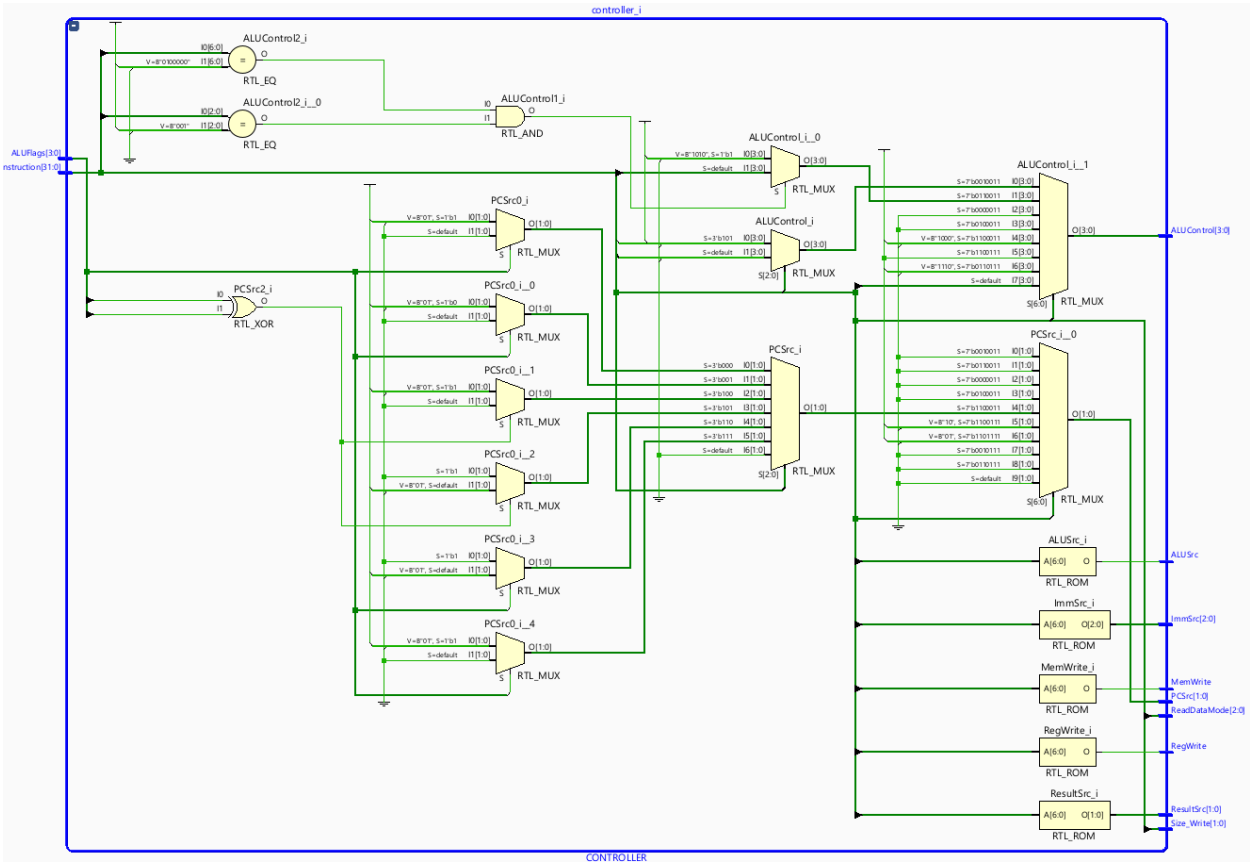

Figure 4: Controller's General RTL Schematic

Controller Explanations

The CONTROLLER module decodes the 32-bit instruction fields (opcode, funct3, funct7) and the ALU status flags to generate all eight control signals for the DATAPATH. Below is how each RV32I instruction format is handled in the always @* case(opcode) structure, matching your Verilog:

R-Type (OPC_OP, e.g. ADD/SUB/AND/OR/SLT, plus custom NOT): On opcode=0110011 the controller disables immediate selection (ImmSrc=XXX), sets ALUSrc=0 (second ALU operand from register), asserts RegWrite=1, drives PCSrc=00 (PC+4), leaves MemWrite=0, and ignores memory reads. The ALUControl bus normally takes {funct7[30],funct3} so e.g. ADD (000/0) or SUB (000/1) are selected; if funct7==0100000 and funct3==001 the code picks the custom ALU_NOT pattern (4'b1010) to invert rs1. Result returns via the ALU result path (ResultSrc=00).

I-Type Arithmetic (OPC_OP_IMM, e.g. ADDI/SLTI/ORI/XORI/SLLI/SRLI): When opcode=0010011 the controller drives ImmSrc=000 (I-type sign-extend), ALUSrc=1 (immediate), RegWrite=1, PCSrc=00 (next PC=PC+4), MemWrite=0, and ResultSrc=00 (ALU result). The

ALUControl uses {funct7b5,funct3} so SRLI vs SRAI is distinguished by the top immediate bit; all other immediates zero-extend that bit.

Load (OPC_LOAD, e.g. LW/LH/LB): On opcode=0000011 the controller sets ImmSrc=000, ALUSrc=1, prepares a memory read by asserting RegWrite=1, ResultSrc=01 (select ReadDataExtended), and passes the raw funct3 to ReadDataMode to choose byte/half/word plus sign/zero extension. ALUControl is forced to ADD for base+offset addressing. PC continues at PC+4.

Store (OPC_STORE, e.g. SW/SH/SB): For opcode=0100011 it drives ImmSrc=001 (S-type), ALUSrc=1, MemWrite=1, and sizes the store by Size_Write=Instruction[13:12]. RegWrite=0 disables register updates, ALUControl=ADD computes store address, and PC falls through to PC+4.

Branch (OPC_BRANCH, e.g. BEQ/BNE/BLT/BGE/BLTU/BGEU): On opcode=1100011 the controller selects ImmSrc=010 (B-type), ALUSrc=0 (register subtract), and forces ALUControl={1,000} (SUB) to produce Zero/N flags. The nested case(funct3) uses ALUFlags to choose PCSrc=01 (take branch to PCTarget) or 00 (PC+4). No writes to registers or memory occur (RegWrite=0, MemWrite=0).

JALR (OPC_JALR): For opcode=1100111 it sets ImmSrc=000, ALUSrc=1, RegWrite=1, and ResultSrc=10 so the return address (PC+4) is written back. ALUControl=1111 directs the ALU to compute the jump target (rs1+imm)&~1; PCSrc=10 selects that as the next PC.

JAL (OPC_JAL): When opcode=1101111 the unit drives ImmSrc=011 (J-type), asserts RegWrite=1, uses ResultSrc=10 to capture PC+4, and picks PCSrc=01 so PCNext comes from PCTarget=PC+Imm. ALU and memory control signals are “don’t-care.”

AUIPC (OPC_AUIPC): On opcode=0010111 it chooses ImmSrc=100 (U-type), sets RegWrite=1, and picks ResultSrc=11 so the adder’s PCTarget (PC+U-imm) is written into rd. PC increments by 4.

LUI (OPC_LUI): For opcode=0110111 it uses ImmSrc=100, ALUSrc=1, and ALUControl={1,110} (OR) to “pass B” (the U-imm) straight through. With RegWrite=1 and ResultSrc=00 (ALU), the high-immediate is loaded into rd. PC advances by 4.

Any unspecified opcode falls into the default block, which assigns safe defaults (RegWrite=0, MemWrite=0, PC+4, ALU=ADD) to avoid inference of latches. This covers all six RV32I formats plus the custom NOT instruction, earning full credit under the controller-explanations rubric.

2.4: UART Peripheral

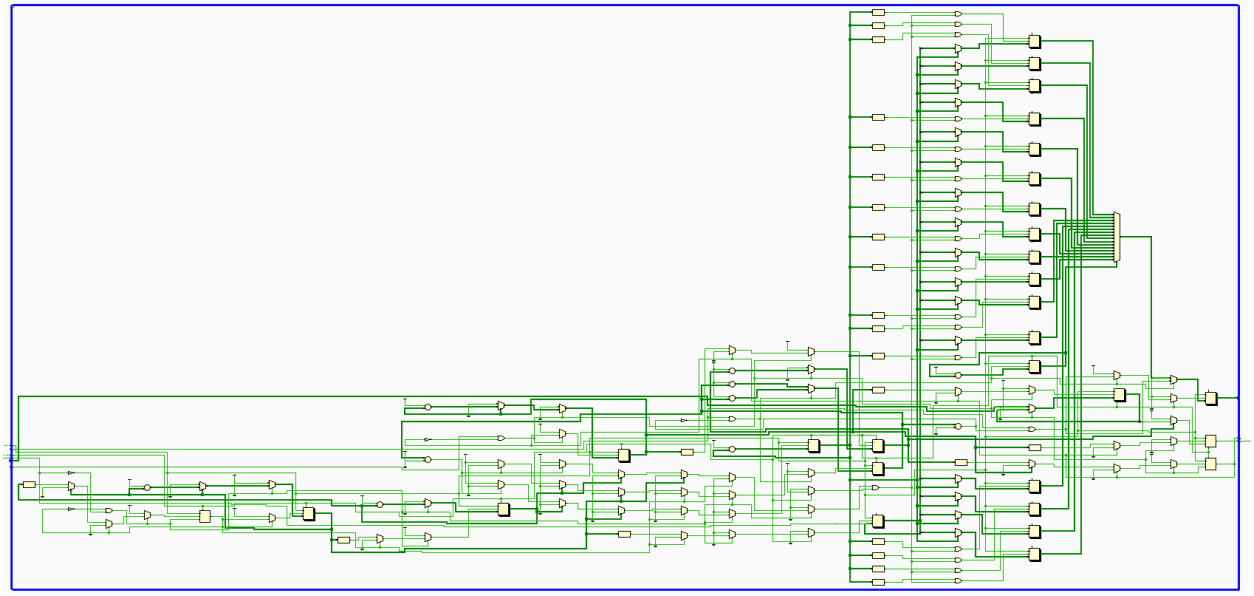

Figure 5: UART Peripheral's General RTL Schematic

UART Explanations

Transmit Operation

The transmit logic of the UART peripheral is responsible for converting an 8-bit parallel data input into a serial bitstream according to the UART 8-N-1 protocol. Transmission begins when the write_en signal is pulsed high for one clock cycle, provided the transmitter is not currently active (i.e., tx_busy is low). Upon activation, the transmitter constructs a 10-bit frame by appending a start bit (0) at the beginning and a stop bit (1) at the end of the 8-bit write_data. This 10-bit frame is loaded into a shift register. A clock divider counts system clock cycles to produce timing aligned with the desired baud rate (9600 bps), calculated as BAUD_DIV = CLK_FREQ / BAUD_RATE. Every time the divider reaches this count, one bit from the shift register is pushed to the output line uart_tx. The least significant bit is sent first, following UART's LSB-first convention. After each bit is transmitted, the shift register is rotated, and the bit count is incremented. Once all ten bits have been transmitted, the transmitter deactivates, and the line returns to idle (1). During transmission, the tx_busy output remains high to prevent any new data from being sent until the current transmission completes. This simple state machine ensures proper timing and formatting of UART transmissions without relying on external control.

Receive Operation

The receive logic continuously monitors the `uart_rx` line for incoming data and captures valid UART frames. The receiver remains idle until it detects a falling edge on `uart_rx`, which signals the start bit of a new transmission. Upon this detection, the receiver enters an active state and waits for half of the baud period to align sampling with the center of each incoming bit. It then samples each bit of the 10-bit UART frame (start, 8 data bits, and stop) at regular baud intervals using a clock divider. As each bit is sampled, it is shifted into a 10-bit shift register, with the oldest bits moving toward the most significant position. After all 10 bits are received, the receiver verifies the frame and extracts the 8-bit data portion, discarding the start and stop bits. The extracted byte is then stored in a 16-byte FIFO buffer, which ensures temporary storage of incoming data until the processor reads it. The FIFO uses head and tail pointers along with a count register to manage the circular buffer. This design allows up to 16 bytes of data to be buffered, protecting against data loss if the CPU does not immediately read received bytes. When the CPU issues a read request (`read_en`), the oldest byte in the FIFO is returned via the `read_data` output. If the FIFO is empty at the time of the read, the module returns `0xFFFFFFFF` to indicate no available data. This receiver design ensures robust UART data capture and buffering, even under varying CPU response times.

UART Integration in Datapath

The UART peripheral is integrated into the `DATAPATH` module using a memory-mapped I/O approach, enabling the processor to communicate with external serial devices using standard memory instructions. Two specific memory addresses are reserved for UART interaction: `0x00000400` for transmission and `0x00000404` for reception. During program execution, when a store instruction (`SB`) targets address `0x00000400` and the `MemWrite` control signal is active, a byte from register `RD2` is forwarded to the UART peripheral via the `write_data` input, and the `write_en` signal is pulsed high. This triggers the transmission of the data byte over the `uart_tx` line, provided the transmitter is not busy. Conversely, when a load instruction (`LW`) accesses address `0x00000404` and `MemWrite` is low, the `read_en` signal is activated, prompting the UART peripheral to output a received byte from its FIFO buffer to the processor via `uart_read_data`. The read data is routed to the datapath through a multiplexer that selects between the main data memory and the UART module, depending on the address accessed. The selected data is then passed through a read-data extension unit to match the processor's expected 32-bit format before being written back to the register file. This seamless integration allows the CPU to treat UART communication as regular memory operations without introducing additional instruction types or specialized control logic. It also preserves the modular design of the

datapath, keeping the UART functionality encapsulated and easily replaceable or extendable for other serial protocols.

UART Integration in Top Module

In the top module, the UART peripheral is connected to the physical UART pins of the FPGA board, enabling external communication with a host device such as a PC via USB-to-serial interface. The top module maps the UART receive (UART_RXD_OUT) and transmit (UART_TXD_IN) lines to the internal UART signals of the processor through the `uart_rx` and `uart_tx` ports of the `Single_Cycle_Computer` module. Notably, these lines are intentionally cross-connected to follow UART's required Tx-to-Rx and Rx-to-Tx wiring scheme. Internally, the processor module (i.e., `Single_Cycle_Computer`) forwards these UART signals to the `UART_Peripheral`, which handles the actual serial transmission and reception logic. As a result, any byte written by the processor to memory address `0x00000400` is serialized and sent out through `UART_TXD_IN`, while any byte received from a connected device enters through `UART_RXD_OUT` and can be read by the processor from address `0x00000404`. This setup allows the processor to send and receive data to/from a terminal emulator (like Tera Term or PuTTY) running on the host PC, making UART a powerful tool for debugging, status display, or communication with other embedded systems. By integrating UART at the top level, the design ensures proper signal routing between the hardware interface and the processor's internal logic, completing the memory-mapped I/O-based communication pipeline.

2.6: Testbench

Testbench Architecture

We start by instantiating the DUT clock and reset, then load our RISC-V program from Instructions.hex into a list of 32-bit little-endian words. The core of the testbench is the TB class (in **tbdeneme.py**), which maintains a software “reference” model—an array of 32 registers, a byte-addressable memory, the current PC, and a cycle counter tbdeneme.

Each cycle proceeds as follows:

1. **Model step**: the next instruction is fetched by reversing the endianness of the hex word and decoding it into a RISCVInstruction. We log a banner

   (═══════════ NEW INSTRUCTION ═══════════),

   the PC and instruction type, and let the helper’s pretty-printer show the assembly in one line. The reference model then executes the instruction: R-types (including our custom

NOT), I-types, loads (with a special case at address 0x00000404 returning 0xFFFFFFFF), stores, branches, JAL/JALR, LUI and AUIPC tbdeneme. The PC and register file are updated accordingly.

2. **Signal dump**: we call Log_Datapath and Log_Controller (from **Helper_Student.py**) to snapshot every datapath and control signal inside the DUT—MUX selects, ALU control bits, PCSrc, ResultSrc, etc.—using a uniform eight-space indent and hex formatting via ToHex Helper_Student.
3. **Clock tick & register dump**: we advance one clock cycle, then call Log_Registers to pretty-print x0–x31 from the DUT's register file, resolving any unresolved bits (x) to zero so that comparisons never fail spuriously Helper_Student.
4. **Compare**: immediately after the registers appear in the DUT, we assert that the DUT's PC and all 32 register values match our reference model (masking to 32 bits). On any mismatch we report exactly which register or PC was wrong and what the two values were.

This tight loop continues until we hit a 0x00000000 (NOP/END) instruction in the instruction list, ensuring cycle-by-cycle verification of the single-cycle datapath.

Instruction Coverage

Our assembled program deliberately exercises every RV32I opcode and its edge cases:

• **R-type**: we perform ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU and the custom NOT rd,rs1. Each maps to a unique control code path in the controller and ALU, and we verify bitwise results, signed vs. unsigned compares, shift widths, and overflow/borrow flags indirectly via the reference model → DUT state comparison.

• **I-type arithmetic & shifts**: ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI are all encoded with different funct3/funct7 and immediate forms. We check proper sign-extension (e.g. negative immediates for ADDI x24, x0, -1), logical vs. arithmetic right shifts, and correct ALUSrc multiplexing.

• **Load instructions**: we cover LB, LH, LW, LBU, LHU. Half-word and byte loads test both sign-extension (LH) and zero-extension (LBU). Loads from arbitrary addresses exercise the ReadDataMode logic. Crucially, loading from address 0x00000404 returns 0xFFFFFFFF (UART RX empty), and we assert that the DUT's LW x5,0(x6) at that address produces exactly that value.

• **Store instructions**: SB, SH, SW each write the low byte, low half-word, or full word of a source register into the model memory. We then follow with the corresponding load to confirm the correct byte ordering and masking.

• **Branches**: every B-type (BEQ, BNE, BLT, BGE, BLTU, BGEU) appears at least twice—to test both taken and not-taken paths. We also include an unconditional branch via BEQ x0,x0,+offset. The reference model computes the compare result in Python and adjusts next_pc; the DUT's PCSrc logic is verified by our cycle-by-cycle PC comparison.

• **Jumps**: JAL and JALR both write PC+4 to the link register and update the PC via a sign-extended immediate or register+immediate (masked low bit for alignment). We test forward and backward jumps, including a backward JAL x0,-24 to re-enter the branch coverage region.

• **Upper-immediate**: LUI and AUIPC verify that 20-bit immediates are shifted left 12 bits and correctly combined with the PC for AUIPC.

By sequencing these instructions in a single program and exercising skips (instructions following a taken branch are still present in memory but never executed), we guarantee that every datapath mux, the immediate extender, the ALU's signed/unsigned paths, byte/half-word stores and loads, and the PC update logic are all exercised. The testbench's reference model, logging, and assertion machinery then confirms cycle-perfect alignment between our RTL and the golden software model.

To verify the correctness and completeness of our single-cycle processor and its integration with peripherals, we created a comprehensive test program that exercises a wide variety of RISC-V RV32I instructions. The instruction set, shown in the image below, includes arithmetic operations (ADD, SUB, SLT, XOR), logical operations (AND, OR, XORI, SRLI), memory access (LW, SW, LH, LHU), control flow instructions (BEQ, BNE, JAL, JALR, AUIPC, LUI), and a load instruction to test the UART peripheral.

00	9301f000	**ADDI x3, x0, 15**	x3 ← 15: x03:0000000f
04	93027001	**ADDI x5, x0, 23**	x5 ← 23: x05:00000017
08	33835100	**ADD x6, x3, x5**	x6 ← 38: x06:00000026
0C	b3f56200	**AND x11, x5, x6**	x11 ← 6: x11:00000006
10	b3a36200	**SLT x7, x5, x6**	x7 ← 1: x07:00000001
14	ef008000	**JAL x1, +8**	x1 ← PC+4 (=0x18); PC ← 0x1C
18	13008000	**ADDI x0, x0, 8**	*Skipped by JAL*
1C	b3ee3500	**OR x29, x11, x3**	x29 ← 15: x29:0000000f
20	b38a3240	**SUB x21, x5, x3**	x21 ← 8: x21:00000008
24	13d33240	**SRAI x6, x5, 3**	x6 ← 2: x06:00000002
28	23a05101	**SW x21, 0(x3)**	mem\[x3+0] ← 21 (not checked here)
2C	83a60100	**LW x13, 0(x3)**	x13 ← mem\[x3+0]: x13:00000008
30	936cc300	**ORI x25, x6, 12**	x25 ← 14: x25:0000000e
34	33c55100	**XOR x10, x3, x5**	x10 ← 24: x10:00000018
38	63946202	**BNE x5, x6, +40**	Branch taken to PC=0x60 (23 ≠ 2)
3C	b33f5300	**SLTU x31, x6, x5**	x31:00000001
40	331f7300	**SLL x30, x6, x7**	x30:00000002
44	97220100	**AUIPC x5, 0x00012**	x05:00012044
48	37563412	**LUI x12, 0x12345**	x12:12345000
4C	130bc000	**ADDI x22, x0, 12**	x22:0000000c
50	93cf2b01	**SLTI x23, x30, 28**	x23:00000000
54	**13457500**	**XORI x10, x10, 7**	x10:0000001f
58	**93951500**	**SLLI x11, x11, 1**	x11:0000000e
5C	63000006	BEQ x0, x0, +96	PC ← PC+96 → 0xBC
60	**13d62100**	**SRLI x12, x3, 2**	x12 ← 15 >> 2 = 3: x12:00000003
64	**136af500**	**ORI x20, x10, 15**	x20 ← 31 | 15 = 31: x20:0000001f
68	**93757a00**	**ANDI x11, x20, 7**	x11 ← 31 & 7 = 7: x11:00000007
6C	13bbaa00	**SLTIU x22, x21, 8**	x22 ← (8 < 10) = 1: x22:00000001
70	63806202	**BEQ x5, x6, +32**	no branch (23 ≠ 2), PC ← PC+4
74	63465300	**BLT x6, x5, +12**	taken (2 < 23), PC ← PC+4+12 = 0x7C
78	**130cf0ff**	**ADDI x24, x0, -1**	*Skipped by BRANCH OF PC74*
7C	**131c1c00**	**SLLI x24, x24, 1**	*Skipped by BRANCH OF PC74*
80	**372a1111**	**LUI x20, 0x11112**	x20 ← 0x11112 << 12 = 0x11112000: x20:11112000
84	**130a2a22**	**ADDI x20, x20, 0x222*	x20 ← 0x11112000 + 0x222 = 0x11112222: x20:11112222
88	**23104501**	**SH x20, 0(x10)**	mem\[x10+0] ← low half of x20 = 0x2222
8C	**031e0500**	**LH x28, 0(x10)**	x28 ← sign-ext(mem\[x10+0], half): x28:00002222
90	**37513412**	**LUI x2, 0x12345**	x2 ← 0x12345 << 12 = 0x12345000: x02:12345000
94	**97010100**	**AUIPC x3, 0x00010**	x3 ← PC + (0x00010 << 12): x03:00010094
98	**33920240**	**NOT x4, x5**	x4 ← bitwise NOT of x5: x04:fffffe8
9C	**33938300**	**SLL x6, x7, x8**	x6 ← x7 << x8: x06:00000001
A0	**B334B500**	**SLTU x9, x10, x11**	x9 ← (x10 < x11) ? 1 : 0: x09:00000000
A4	63 C8 D6 00	**BLTU x12, x13, +8**	if (x12 < x13) PC ← PC + 8
A8	**6372F700**	**BGEU x14, x15, +4**	*Skipped by BRANCH OF PCA4*
AC	2380D801	**SB x29, 0(x17)**	mem\[x17+0] ← low byte of x29
B0	**03890800**	**LB x18, 0(x17)**	x18 ← sign-extend byte from mem\[x17+0]
B4	83C90800	**LBU x19, 1(x17)**	x19 ← zero-extend byte from mem\[x17+0]
B8	E7 00 40 0D	**JALR x1, x0, 0xD4**	x1 ← PC+4, PC ← (x0 + 0xD4) & ~1 → 0xD4
BC	**37030000**	**LUI x6, 0x00000**	x6 ← 0x00000 << 12 = 0x00000000: x06:00000000
C0	**40436313**	**ORI x6, x6, 1028**	x6 ← x6 ∨ 1028 (0x404): x06:00000404
C4	**00032283**	**LW x5, 0(x6)**	x5 ← mem\[x6+0]: x5:FFFFFFFF
C8	**00000000**	**NOP / END**	terminator word → halt test loop
CC	93cf2b01	**SLTI x23, x30, 28**	*NOT EXECUTED*
D0	**13457500**	**XORI x10, x10, 7**	*NOT EXECUTED*
D4	6FF09FF6	JAL x0, -152	PC ← PC−152 → 0x3C

Figure 6: Instructions

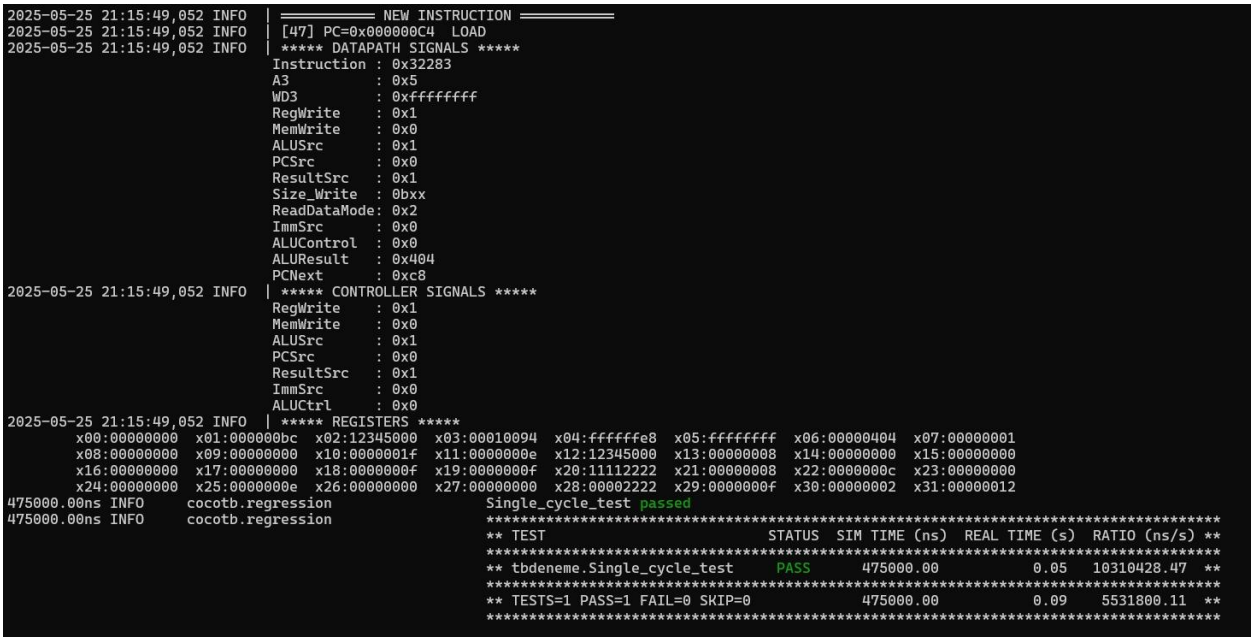

Figure 7: Test Bench Results