Department of Electronics and
Electrical Communications Engineering
Faculty of Engineering-Cairo University

Advanced Processor Architecture

Design and Implementation of an 8-Bit RISC-like processor

Submitted by:

Name	ID	BN	Sec
عبدالرحمن محمود عبدالحميد محمد	9230547	46	2
طارق سيد محمد سيد	9230492	34	2
شهاب ابو زيد عبدالعال ابو زيد	9230477	29	2
عبدالرحمن وائل محمود العوضي	9230515	48	2
زياد عبدالحفيظ عبدالحفيظ محمد	9230399	16	2
بسمة خالد عبدالحفيظ عبدالحليم	9230280	44	1

Under the supervision of:

Eng / Hassan El-Menier

Contents

Figures:

1.Introduction:

Microprocessor design is fundamental in computer and electronics engineering, forming the core of modern digital systems ranging from simple embedded controllers to complex computing platforms. This project focuses on the design and implementation of a simplified yet fully functional **8-bit RISC-like microprocessor**, intended to demonstrate core architectural concepts and pipeline efficiency.

Unlike traditional implementations that often utilize Von Neumann architecture, this proposed processor utilizes **Harvard Architecture**. By employing separate physical memories for instructions and data, the design allows for simultaneous instruction fetching and data access. This architectural choice significantly improves pipeline efficiency by eliminating the structural hazards common in unified memory systems.

The processor is structured around a robust five-stage pipeline: Fetch (IF), Decode (ID), Execute (EX), Memory (MEM), and Write-Back (WB). This pipelined approach ensures high instruction throughput while maintaining logical clarity. To support a wide range of functions—including arithmetic, logical operations, control flow, stack management, and interrupts—key components such as the Register File (R0-R3), ALU, Control Unit, and dedicated Forwarding and Hazard Units have been integrated into the datapath.

The entire design was implemented using Verilog HDL. The development process involved verifying individual blocks before integrating them into the complete system, followed by extensive simulation to ensure functional correctness and stability under various operating conditions.

2. Design procedure

2.1 Architecture Overview

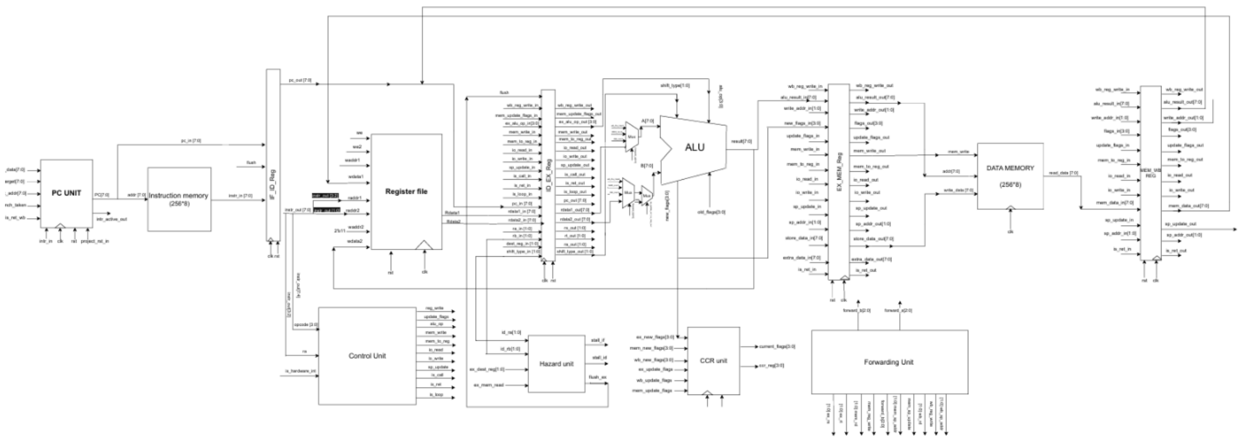


Figure 1: Architecture

The processor features an 8-bit data path and an 8-bit address space. The control unit directs the flow of data through the pipeline based on the opcode fetched from the **Instruction Memory**.

Instruction Memory (InstructionMemory.v): A dedicated read-only memory stores the program code. It is accessed during the Fetch stage using the Program Counter (PC).

Data Memory (DataMemory.v): A separate read/write memory is used for data storage (variables, stack). It is accessed during the Memory stage.

2.2 Fetch Cycle Logic

The Fetch Stage is responsible for retrieving the current instruction from memory and determining the address of the next instruction. This stage relies on the interaction between the Program Counter (PC) Unit and the dedicated Instruction Memory.

1.Instruction Memory This processor utilizes a dedicated InstructionMemory block. It is implemented as a read-only memory array of 256 bytes

Input: The 8-bit address from the Program Counter (PC).

Output: The 8-bit machine code instruction (instr) corresponding to that address.

Initialization: The memory is pre-loaded with the program code (machine code) from an external file (program.mem) during simulation start-up.

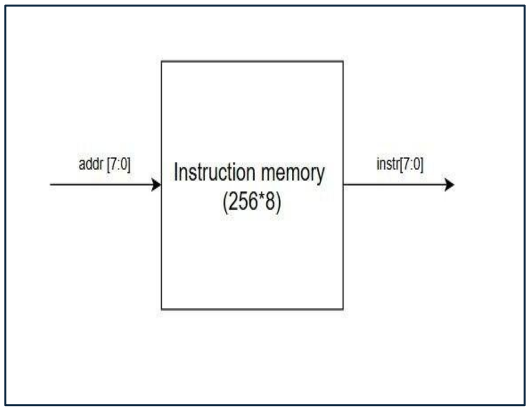


Figure 2: instruction memory block

2. PC Update Logic The PC_Unit determines the value of the Program Counter for the next clock cycle. The next PC value is selected based on a strict priority scheme implemented in the Verilog design:

Project Reset: If the reset_signal is asserted, the PC is forced to address 0x00 (the Reset Vector).

Hardware Interrupt: If the intr_signal is asserted, the PC is forced to address 0x01 (the Interrupt Vector).

Return Operation: If a RET or RTI instruction completes in the Write-Back stage, the PC is updated with the return address popped from the stack.

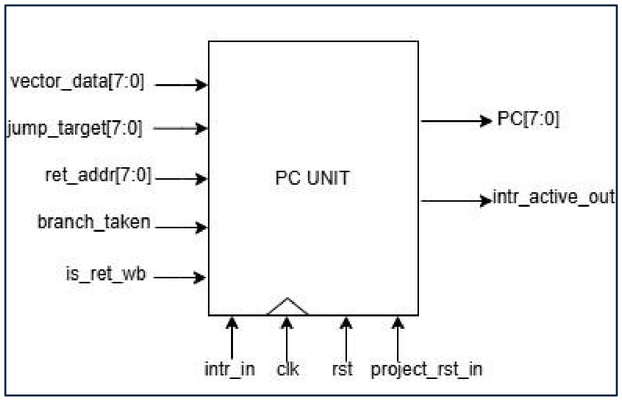


Figure 3: PC unit block

Branch/Jump: If a branch condition is met or a jump is executed, the PC loads the target address.

Sequential Execution: In all other cases, the PC increments by 1.

3. IF/ID Pipeline Register To maintain pipeline stability, the fetched instruction and the current PC value are latched into the IF_ID_Reg on the rising edge of the clock.

Flushing: The register includes a flush input. If a control hazard occurs (such as a taken branch, jump, or interrupt), the flush signal clears the register, effectively inserting a NOP (No Operation) into the pipeline to discard the incorrectly fetched instruction

2.3 Decode Cycle Logic

In the Decode Stage, the control unit interprets the fetched instructions to generate appropriate control signals, while the Register File provides the necessary operands. This stage also houses the Hazard Unit responsible for maintaining pipeline integrity.

1-Control Unit the Control Unit is the brain of the processor. It takes the 4-bit opcode and the 2-bit RA field from the instruction to determine the required operation.

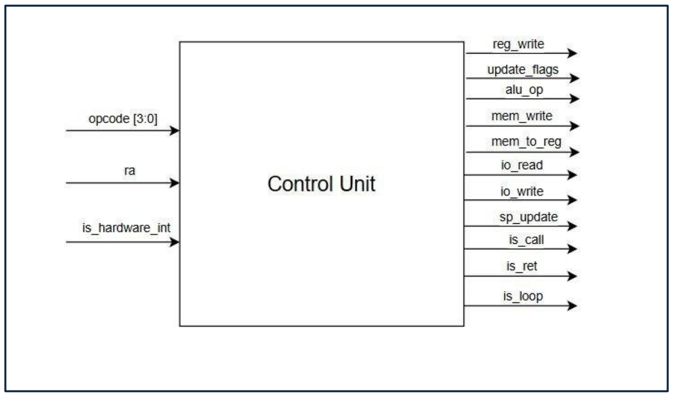


Figure 4:Control unit block

Extended Decoding: For instructions like STACK_IO (Opcode 7) and JUMP (Opcode 11), the unit uses the 2-bit ra input as a secondary function code to distinguish between operations (e.g., distinguishing PUSH vs POP, or CALL vs RET).

Interrupt Handling: The Control Unit features specific logic for hardware interrupts. When the is_hardware_int signal is active, it overrides the standard instruction decoding. It forces the control signals to mimic a CALL operation (is_call=1, mem_write=1, alu_op=SUB), ensuring the current Program Counter is saved to the stack before the Interrupt Service Routine begins.

2-Register File Access The processor utilizes a Register File module containing four 8-bit general-purpose registers (R0–R3)

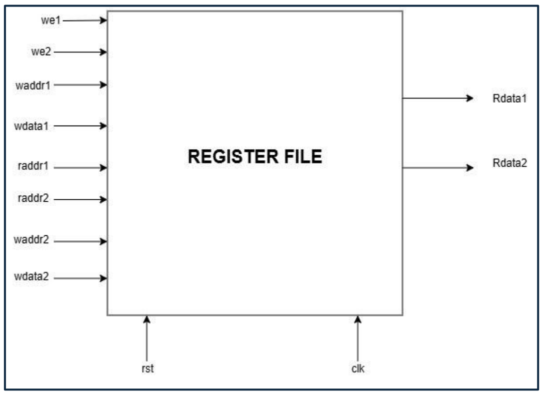


Figure 5:Register file block

Operand Reading: During the decode cycle, the source register addresses (ra and rb) select the registers to be read asynchronously via ports raddr1 and raddr2, making the data available immediately for the Execute stage.

Stack Pointer (SP): Register R3 is designated as the Stack Pointer and is initialized to 0xFF (255) upon reset.

Dual Write Ports: Uniquely, the register file includes a second write port (we2, waddr2) specifically to allow the Stack Pointer to be updated simultaneously with other register operations, preventing resource conflicts during complex stack instructions.

3-Hazard Unit This unit is responsible for preserving pipeline integrity by managing Load-Use Hazards. A Load-Use hazard occurs when an instruction in the Decode stage tries to read a register that the previous instruction (currently in Execute) is loading from memory. Since the data is not yet available, forwarding is impossible.

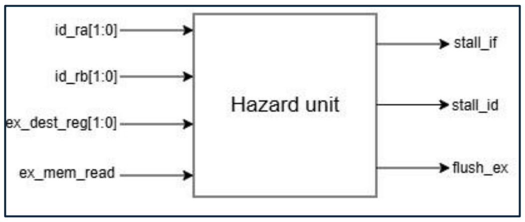


Figure 6:Hazard unit block

Detection: The unit monitors the pipeline and flags a hazard if the instruction in the Execute stage is a Load (ex_mem_read is high) and its destination register matches either source register (id_ra or id_rb) of the current instruction.

Mitigation: Upon detection, the unit forces a one-cycle stall. It asserts stall_if and stall_id to freeze the Program Counter and Fetch/Decode registers and asserts flush_ex to inject a NOP (bubble) into the Execute stage. This delay allows the memory access to complete so the correct data can be forwarded in the next cycle.

2.4 Execute Cycle Logic

The Execute stage is where the processor performs the actual heavy lifting calculating results, manipulating addresses, and ensuring the correct data is used even when the pipeline is full.

1.The ALU (Arithmetic Logic Unit) ALU is the computational engine of the processor. We designed it to handle not just standard math and logic but also to assist with system operations

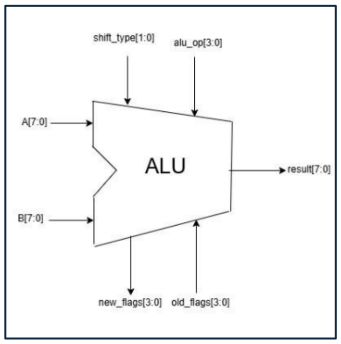


Figure 7:ALU block

Stack Management**:** A key feature of our ALU is its ability to handle stack pointer updates directly. For instructions like `PUSH` and `POP`, the Control Unit sends specific operation codes that tell the ALU to increment or decrement the stack pointer. This ensures the memory address for the stack is calculated correctly in hardware before the Memory stage tries to use it.

2. Handling Data Hazards (Forwarding) In a pipelined processor, a major challenge is

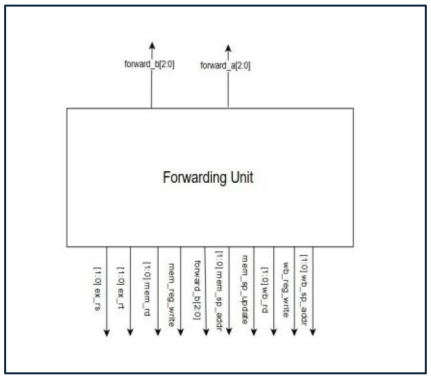


Figure 8:Forwarding unit block

ensuring that instructions use the most up-to-date data. Because results are technically not written back to the Register File until the final stage, a subsequent instruction might try to read an old, outdated value. To fix this without slowing down the processor, we implemented a **Forwarding Unit**. How it works**:** This unit acts like a traffic controller. It constantly compares the source registers of the current instructions against the destination registers of instructions further down the pipeline (in the Memory or Write-Back stages).

3. Condition Code Register (CCR) Finally, the **CCR Unit** tracks the status of the processor (Zero, Negative, Carry, and Overflow flags). We designed this unit to be smart about where it gets its information. While it usually updates flags based on the immediate result from the ALU, it can also accept flag updates from memory (for example, when returning from an interrupt using `RTI`). This ensures that the processor's status is accurately restored after handling an interruption.

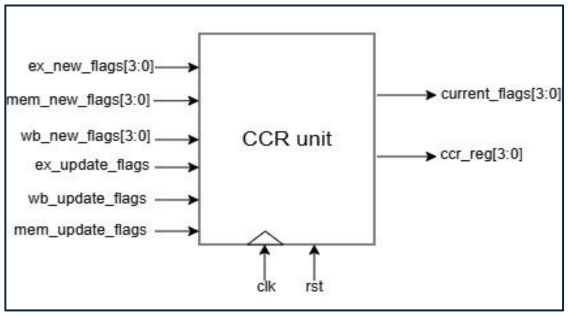


Figure 9:CCR unit block

2.5 Memory Cycle Logic

The Memory Stage is responsible for interacting with the Data Memory. While the Fetch stage handles code, this stage handles variables and the system stack.

1. Data Memory Unit We implemented the Data Memory as a distinct 256-byte storage block. Unlike the Instruction Memory, this unit supports both reading and writing.

Reading: This is an asynchronous operation. As soon as an address is presented to the memory, the data at that location is outputted immediately. This allows the value to be ready for the Write-Back stage in the same clock cycle.

Writing: This is a synchronous operation. Data is written into the memory array only on the rising edge of the clock when the write enable signal is active.

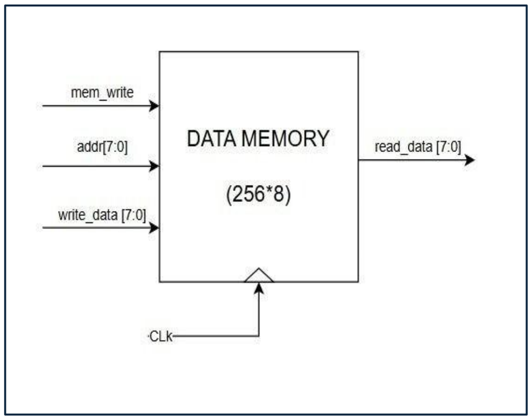


Figure 10:DATA Memory block

2. Address Selection Logic A key design decision we made in the top-level module was how to handle memory addressing. The memory address can come from two different sources depending on the operation:

General Load/Store: For standard instructions, the address is calculated by the ALU and passed down the pipeline.

Stack Operations: For stack instructions (like PUSH or CALL), the address is derived from the Stack Pointer. We implemented a selection logic that automatically switches the memory address input to the Stack Pointer value whenever a stack update is occurring.

3. EX/MEM Pipeline Register The EX/MEM register acts as the bridge to this stage. It holds the calculated results from the ALU and the data to be stored (if any) so that they are stable during the memory access window.

2.6 Write-Back Cycle Logic

The final stage of the pipeline is the Write-Back (WB) stage. Its sole purpose is to commit the results computed in previous stages into the architectural state (the Register File).

1-Data Selection Logic Since the processor supports different types of instructions, some that compute values and some that load values we need a way to choose the correct data to write. We implemented multiplexer logic in the top-level design that selects the final write data based on the mem_to_reg control signal:

ALU Result: For arithmetic and logic instructions, the data comes directly from the ALU (passed through the EX/MEM register).

Data memory: For load and pop instructions, the data comes from the Data Memory output.

2-Register File Update Once the correct data is selected, it is presented to the Write Port 1 of the Register File.

Write Enable: The write operation only occurs if the wb_reg_write signal is active. This ensures that instructions like CMP (Compare) or STORE, which should not modify general registers, do not corrupt the register file.

Timing: The write occurs on the rising edge of the clock, effectively completing the instruction's execution.

3-Feedback to Forwarding Unit Although this is the end of the pipeline, the data currently sitting in this stage is critical for performance. The Forwarding Unit constantly monitors the Write-Back destination register and the data. If a newer instruction in the Decode or Execute stage needs this value immediately, it is forwarded backward to resolve hazards without stalling.

3.Testing

Note: before any code we ran, we started with Reset of processor

Note: Wherever the transcript mentions a '?', it implies a NOP

3.1 Testing MOV ADD SUB

Code:

```
IN R0 <---- 17       (R0 = 17)
IN R1 <---- 12       (R1 = 12)
MOV R2, R1           ( R2 = 12 (Copy R1))
ADD R2, R1           (R2 = 24 (12 + 12))
SUB R0, R1           (R0 = 5 (17 - 12))
```

Waveform:

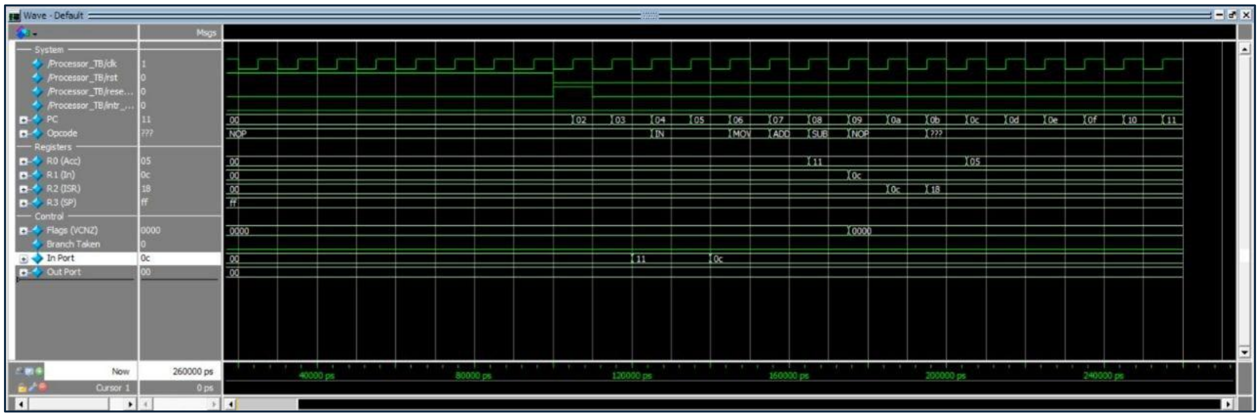
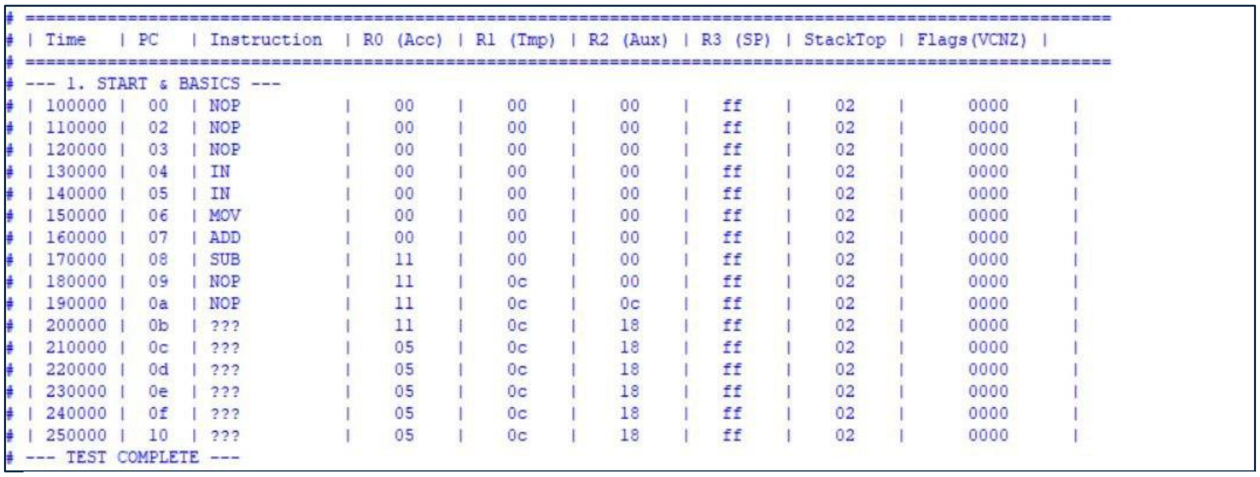


Figure 11: MOV ADD SUB waveform

Transcript:



```
# ==========================================================================================================
# | Time    | PC   | Instruction   | R0 (Acc) | R1 (Tmp) | R2 (Aux) | R3 (SP) | StackTop | Flags(VCNZ) |
# ==========================================================================================================
# --- 1. START & BASICS ---
# | 100000 |  00  | NOP          |    00    |    00    |    00    |   ff    |    02    |     0000    |
# | 110000 |  02  | NOP          |    00    |    00    |    00    |   ff    |    02    |     0000    |
# | 120000 |  03  | NOP          |    00    |    00    |    00    |   ff    |    02    |     0000    |
# | 130000 |  04  | IN           |    00    |    00    |    00    |   ff    |    02    |     0000    |
# | 140000 |  05  | IN           |    00    |    00    |    00    |   ff    |    02    |     0000    |
# | 150000 |  06  | MOV          |    00    |    00    |    00    |   ff    |    02    |     0000    |
# | 160000 |  07  | ADD          |    00    |    00    |    00    |   ff    |    02    |     0000    |
# | 170000 |  08  | SUB          |    11    |    00    |    00    |   ff    |    02    |     0000    |
# | 180000 |  09  | NOP          |    11    |    0c    |    00    |   ff    |    02    |     0000    |
# | 190000 |  0a  | NOP          |    11    |    0c    |    0c    |   ff    |    02    |     0000    |
# | 200000 |  0b  | ???          |    11    |    0c    |    18    |   ff    |    02    |     0000    |
# | 210000 |  0c  | ???          |    05    |    0c    |    18    |   ff    |    02    |     0000    |
# | 220000 |  0d  | ???          |    05    |    0c    |    18    |   ff    |    02    |     0000    |
# | 230000 |  0e  | ???          |    05    |    0c    |    18    |   ff    |    02    |     0000    |
# | 240000 |  0f  | ???          |    05    |    0c    |    18    |   ff    |    02    |     0000    |
# | 250000 |  10  | ???          |    05    |    0c    |    18    |   ff    |    02    |     0000    |
# --- TEST COMPLETE ---
```

Figure 12:MOV ADD SUB transcript

3.2 Testing AND OR NOT NEG OUT

Code:

```
IN R0 13       ; Load 13 into R0
IN R1 05       ; Load 5 into R1
IN R2 02       ; Load 2 into R2
AND R2, R1     ; R2 = 2 AND 5 (Result: 0)
OR R0, R1      ; R0 = 13 OR 5 (Result: 13)
NOT R1         ; Invert bits of R1 (Result: 250 or -6)
OUT R1         ; Output value of R1
NEG R2         ; Negate R2 (2's Complement)
OUT R2         ; Output value of R2
```

Waveform:

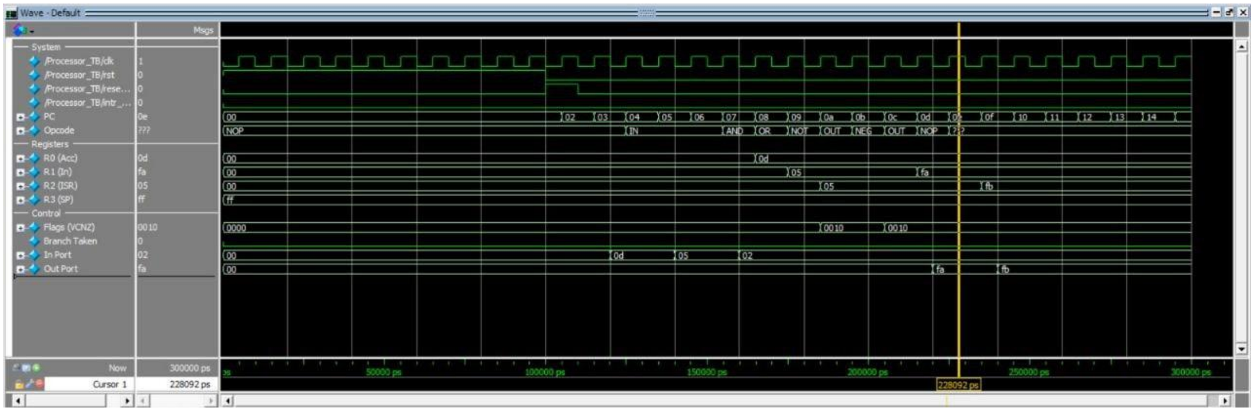
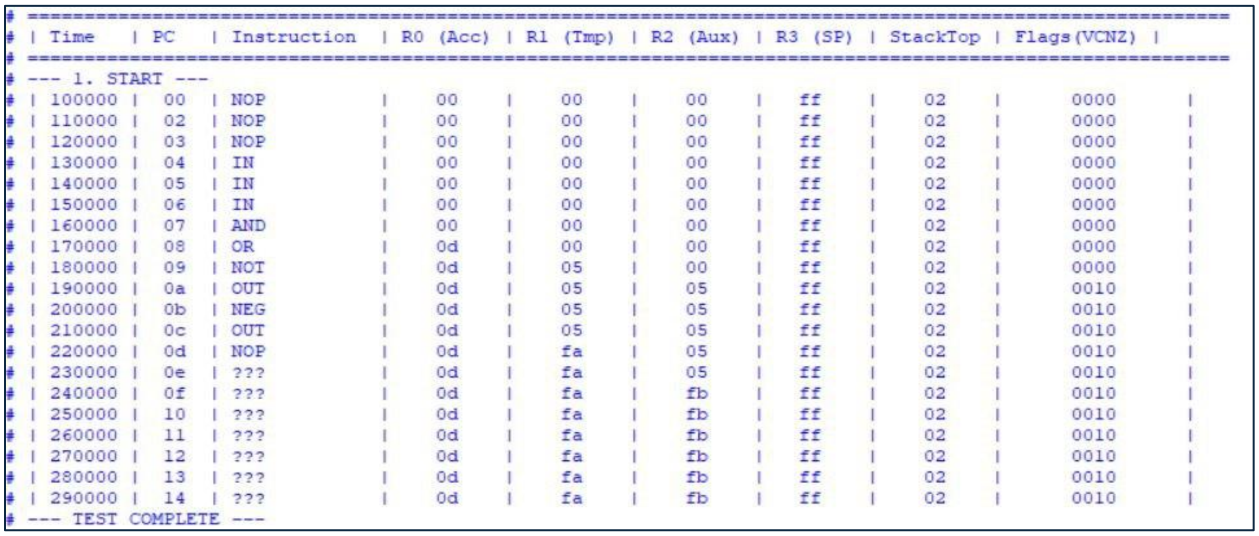


Figure 13: AND OR NOT NEG OUT waveform

Transcript:



```
# ==========================================================================================================
# | Time     | PC    | Instruction    | R0 (Acc) | R1 (Tmp) | R2 (Aux) | R3 (SP)  | StackTop | Flags(VCNZ) |
# ==========================================================================================================
# --- 1. START ---
# | 100000 |  00   | NOP            |    00    |    00    |    00    |    ff    |    02    |    0000     |
# | 110000 |  02   | NOP            |    00    |    00    |    00    |    ff    |    02    |    0000     |
# | 120000 |  03   | NOP            |    00    |    00    |    00    |    ff    |    02    |    0000     |
# | 130000 |  04   | IN             |    00    |    00    |    00    |    ff    |    02    |    0000     |
# | 140000 |  05   | IN             |    00    |    00    |    00    |    ff    |    02    |    0000     |
# | 150000 |  06   | IN             |    00    |    00    |    00    |    ff    |    02    |    0000     |
# | 160000 |  07   | AND            |    00    |    00    |    00    |    ff    |    02    |    0000     |
# | 170000 |  08   | OR             |    0d    |    00    |    00    |    ff    |    02    |    0000     |
# | 180000 |  09   | NOT            |    0d    |    05    |    00    |    ff    |    02    |    0000     |
# | 190000 |  0a   | OUT            |    0d    |    05    |    05    |    ff    |    02    |    0010     |
# | 200000 |  0b   | NEG            |    0d    |    05    |    05    |    ff    |    02    |    0010     |
# | 210000 |  0c   | OUT            |    0d    |    05    |    05    |    ff    |    02    |    0010     |
# | 220000 |  0d   | NOP            |    0d    |    fa    |    05    |    ff    |    02    |    0010     |
# | 230000 |  0e   | ???            |    0d    |    fa    |    05    |    ff    |    02    |    0010     |
# | 240000 |  0f   | ???            |    0d    |    fa    |    fb    |    ff    |    02    |    0010     |
# | 250000 |  10   | ???            |    0d    |    fa    |    fb    |    ff    |    02    |    0010     |
# | 260000 |  11   | ???            |    0d    |    fa    |    fb    |    ff    |    02    |    0010     |
# | 270000 |  12   | ???            |    0d    |    fa    |    fb    |    ff    |    02    |    0010     |
# | 280000 |  13   | ???            |    0d    |    fa    |    fb    |    ff    |    02    |    0010     |
# | 290000 |  14   | ???            |    0d    |    fa    |    fb    |    ff    |    02    |    0010     |
# --- TEST COMPLETE ---
```

Figure 14: AND OR NOT NEG OUT transcript

3.3 Testing RLC RRC DEC INC PUSH POP

Code:

```
IN R0 10      ; Load 10 into R0
IN R1 07      ; Load 7 into R1
IN R2 03      ; Load 3 into R2
RLC R2        ; Rotate R2 Left (3 becomes 6)
RRC R1        ; Rotate R1 Right (7 becomes 3)
DEC R0        ; Decrement R0 (10 becomes 9)
INC R0        ; Increment R0 (9 becomes 10)
PUSH R0       ; Push R0 (10) onto Stack
POP R0        ; Pop Stack back into R0
```

Waveform:

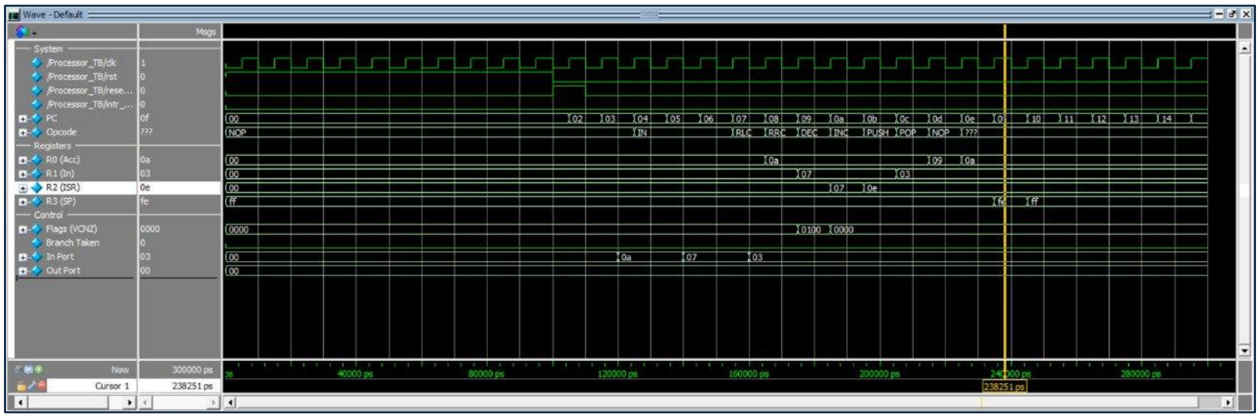
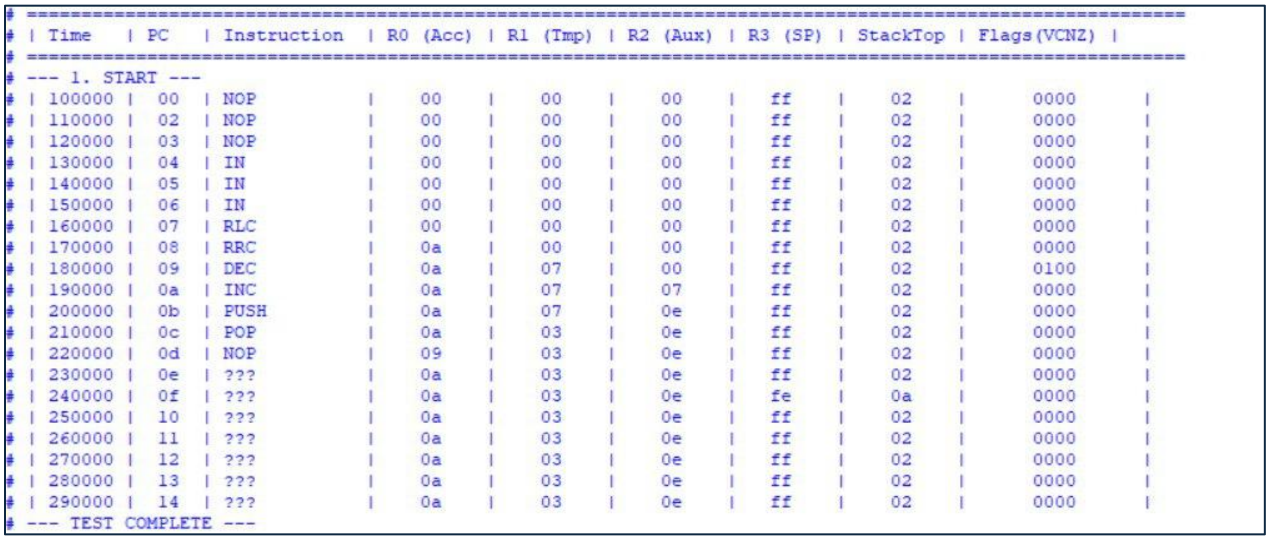


Figure 15:RLC RRC DEC INC PUSH POP waveform

Transcript:



```
# ==========================================================================================
# | Time    | PC   | Instruction   | R0 (Acc) | R1 (Tmp) | R2 (Aux) | R3 (SP) | StackTop | Flags(VCNZ) |
# ==========================================================================================
# --- 1. START ---
# | 100000 |  00  | NOP           |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 110000 |  02  | NOP           |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 120000 |  03  | NOP           |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 130000 |  04  | IN            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 140000 |  05  | IN            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 150000 |  06  | IN            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 160000 |  07  | RLC           |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 170000 |  08  | RRC           |    0a    |    00    |    00    |   ff    |    02    |    0000     |
# | 180000 |  09  | DEC           |    0a    |    07    |    00    |   ff    |    02    |    0100     |
# | 190000 |  0a  | INC           |    0a    |    07    |    07    |   ff    |    02    |    0000     |
# | 200000 |  0b  | PUSH          |    0a    |    07    |    0e    |   ff    |    02    |    0000     |
# | 210000 |  0c  | POP           |    0a    |    03    |    0e    |   ff    |    02    |    0000     |
# | 220000 |  0d  | NOP           |    09    |    03    |    0e    |   ff    |    02    |    0000     |
# | 230000 |  0e  | ???           |    0a    |    03    |    0e    |   ff    |    02    |    0000     |
# | 240000 |  0f  | ???           |    0a    |    03    |    0e    |   fe    |    0a    |    0000     |
# | 250000 |  10  | ???           |    0a    |    03    |    0e    |   ff    |    02    |    0000     |
# | 260000 |  11  | ???           |    0a    |    03    |    0e    |   ff    |    02    |    0000     |
# | 270000 |  12  | ???           |    0a    |    03    |    0e    |   ff    |    02    |    0000     |
# | 280000 |  13  | ???           |    0a    |    03    |    0e    |   ff    |    02    |    0000     |
# | 290000 |  14  | ???           |    0a    |    03    |    0e    |   ff    |    02    |    0000     |
# --- TEST COMPLETE ---
```

Figure 16: RLC RRC DEC INC PUSH POP transcript

3.4 Testing SETC CLRC

Code:

```
SETC        ; Set Carry Flag (C = 1)
CLRC        ; Clear Carry Flag (C = 0)
```

Waveform:

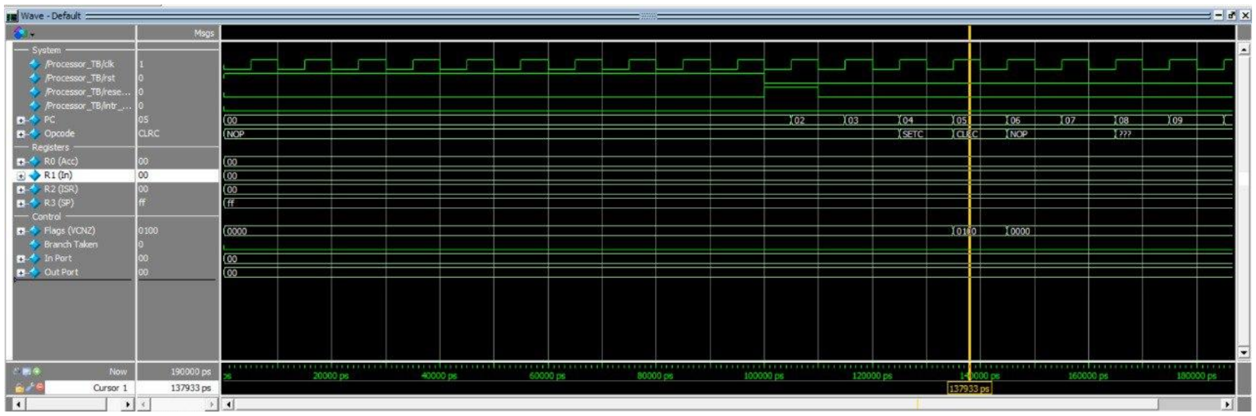
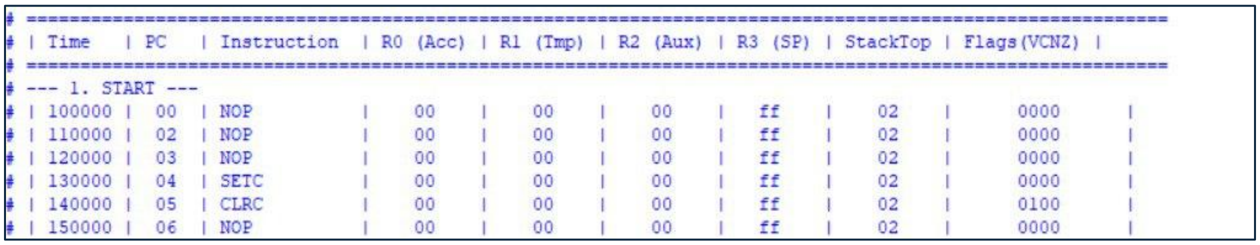


Figure 17: SETC CLRC waveform

Transcript:



```
# ==================================================================================================
# | Time     | PC    | Instruction    | R0 (Acc)  | R1 (Tmp)  | R2 (Aux)  | R3 (SP)  | StackTop | Flags(VCNZ) |
# ==================================================================================================
# --- 1. START ---
# | 100000 |   00   | NOP            |    00     |    00     |    00     |    ff    |    02    |     0000     |
# | 110000 |   02   | NOP            |    00     |    00     |    00     |    ff    |    02    |     0000     |
# | 120000 |   03   | NOP            |    00     |    00     |    00     |    ff    |    02    |     0000     |
# | 130000 |   04   | SETC           |    00     |    00     |    00     |    ff    |    02    |     0000     |
# | 140000 |   05   | CLRC           |    00     |    00     |    00     |    ff    |    02    |     0100     |
# | 150000 |   06   | NOP            |    00     |    00     |    00     |    ff    |    02    |     0000     |
```

Figure 18: SETC and CLRC transcript

3.5 Testing JZ

Code:

```
IN R1          ; Load jump target address into R1
NOP            ; Delay
NOP            ; Delay
NOP            ; Delay
SUB R0, R0     ; R0 = 0, Sets Zero Flag (Z=1)
JZ R1          ; Jump to address in R1 (Taken because Z=1)
SETC (skipped) ; Skipped (Instruction flushed)
SETC (skipped) ; Skipped
MOV R2, R1     ; Execution resumes here (Jump Target)
```

Waveform:

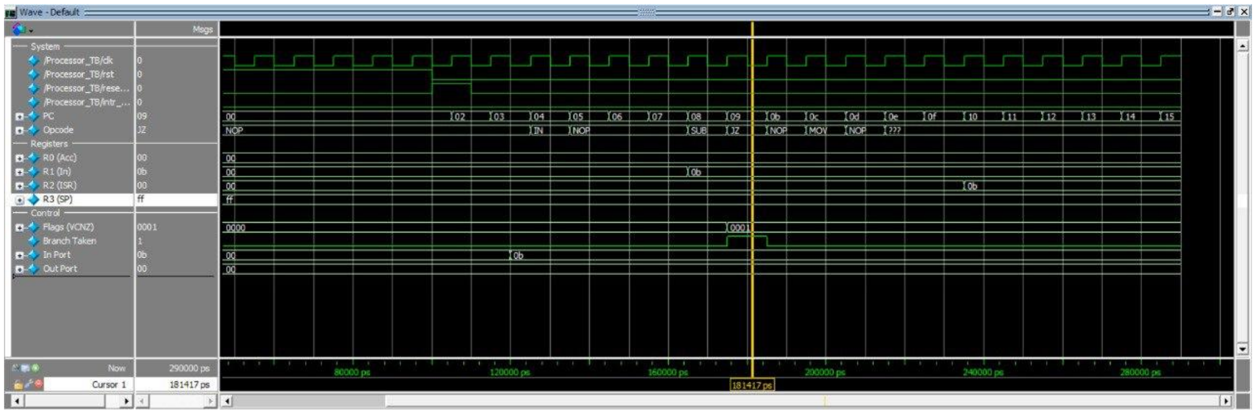
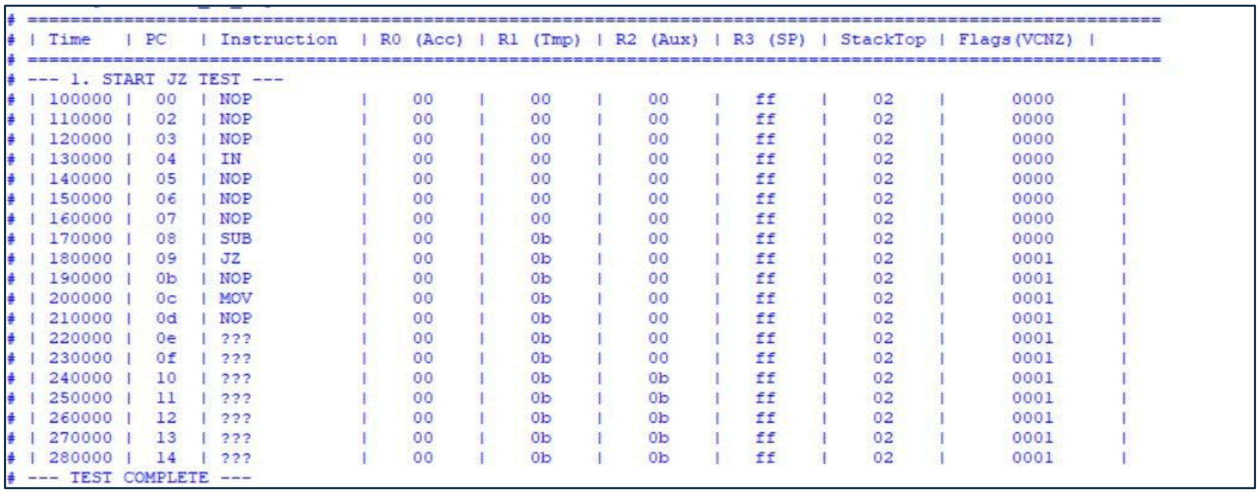


Figure 20: JZ waveform

Transcript:



```
# ==============================================================================================================
# | Time     | PC    | Instruction    | R0 (Acc)  | R1 (Tmp)  | R2 (Aux)  | R3 (SP)  | StackTop  | Flags(VCNZ)  |
# ==============================================================================================================
# --- 1. START JZ TEST ---
# | 100000 |  00   | NOP            |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 110000 |  02   | NOP            |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 120000 |  03   | NOP            |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 130000 |  04   | IN             |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 140000 |  05   | NOP            |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 150000 |  06   | NOP            |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 160000 |  07   | NOP            |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 170000 |  08   | SUB            |    00     |    0b     |    00     |    ff    |    02     |    0000      |
# | 180000 |  09   | JZ             |    00     |    0b     |    00     |    ff    |    02     |    0001      |
# | 190000 |  0b   | NOP            |    00     |    0b     |    00     |    ff    |    02     |    0001      |
# | 200000 |  0c   | MOV            |    00     |    0b     |    00     |    ff    |    02     |    0001      |
# | 210000 |  0d   | NOP            |    00     |    0b     |    00     |    ff    |    02     |    0001      |
# | 220000 |  0e   | ???            |    00     |    0b     |    00     |    ff    |    02     |    0001      |
# | 230000 |  0f   | ???            |    00     |    0b     |    00     |    ff    |    02     |    0001      |
# | 240000 |  10   | ???            |    00     |    0b     |    0b     |    ff    |    02     |    0001      |
# | 250000 |  11   | ???            |    00     |    0b     |    0b     |    ff    |    02     |    0001      |
# | 260000 |  12   | ???            |    00     |    0b     |    0b     |    ff    |    02     |    0001      |
# | 270000 |  13   | ???            |    00     |    0b     |    0b     |    ff    |    02     |    0001      |
# | 280000 |  14   | ???            |    00     |    0b     |    0b     |    ff    |    02     |    0001      |
# --- TEST COMPLETE ---
```

Figure 19: JZ transcript

3.6 Testing JN

Code:

```
IN R1          ; Load jump target address into R1
NOP            ; Delay
NOP            ; Delay
NOP            ; Delay
NOT R0         ; Invert R0 (Result is negative, Sets N=1)
JN R1          ; Jump to address in R1 (Taken because N=1)
SETC (skipped) ; Skipped (Instruction flushed)
SETC (skipped) ; Skipped
MOV R2, R1     ; Execution resumes here (Jump Target)
```

Waveform:

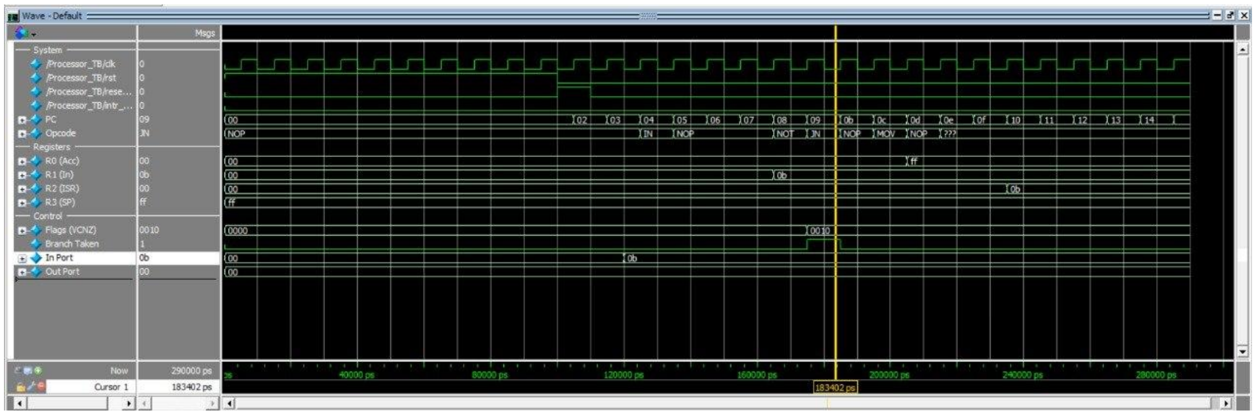
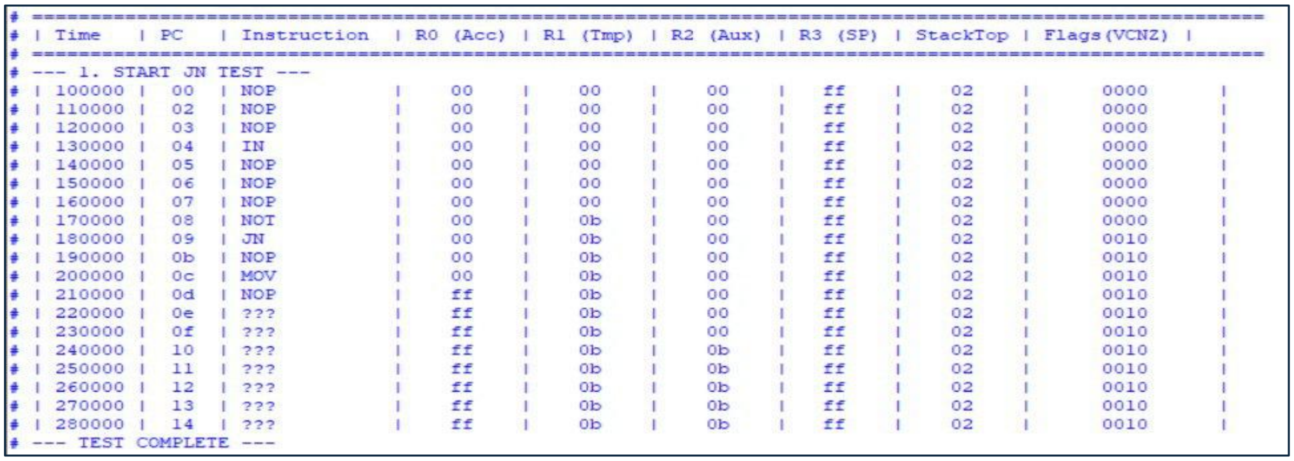


Figure 21: JN waveform

Transcript:



```
# ==============================================================================================
# | Time    | PC    | Instruction    | R0 (Acc) | R1 (Tmp) | R2 (Aux) | R3 (SP) | StackTop | Flags(VCNZ) |
# ==============================================================================================
# --- 1. START JN TEST ---
# | 100000 |  00   | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 110000 |  02   | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 120000 |  03   | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 130000 |  04   | IN             |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 140000 |  05   | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 150000 |  06   | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 160000 |  07   | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 170000 |  08   | NOT            |    00    |    0b    |    00    |   ff    |    02    |    0000     |
# | 180000 |  09   | JN             |    00    |    0b    |    00    |   ff    |    02    |    0010     |
# | 190000 |  0b   | NOP            |    00    |    0b    |    00    |   ff    |    02    |    0010     |
# | 200000 |  0c   | MOV            |    00    |    0b    |    00    |   ff    |    02    |    0010     |
# | 210000 |  0d   | NOP            |    ff    |    0b    |    00    |   ff    |    02    |    0010     |
# | 220000 |  0e   | ???            |    ff    |    0b    |    00    |   ff    |    02    |    0010     |
# | 230000 |  0f   | ???            |    ff    |    0b    |    00    |   ff    |    02    |    0010     |
# | 240000 |  10   | ???            |    ff    |    0b    |    0b    |   ff    |    02    |    0010     |
# | 250000 |  11   | ???            |    ff    |    0b    |    0b    |   ff    |    02    |    0010     |
# | 260000 |  12   | ???            |    ff    |    0b    |    0b    |   ff    |    02    |    0010     |
# | 270000 |  13   | ???            |    ff    |    0b    |    0b    |   ff    |    02    |    0010     |
# | 280000 |  14   | ???            |    ff    |    0b    |    0b    |   ff    |    02    |    0010     |
# --- TEST COMPLETE ---
```

Figure 22:JN transcript

3.7 Testing JC

Code:

```
IN R1        ; Load jump target address into R1
NOP          ; Delay
NOP          ; Delay
NOP          ; Delay
SETC         ; Set Carry Flag (C=1)
JN R1        ; Jump if Negative (Likely intended JC - Jump if Carry)
NOT R0       ; (Skipped instruction)
NOT R0       ; (Skipped instruction)
MOV R2, R1   ; Execution resumes here (Jump Target)
```

Waveform:

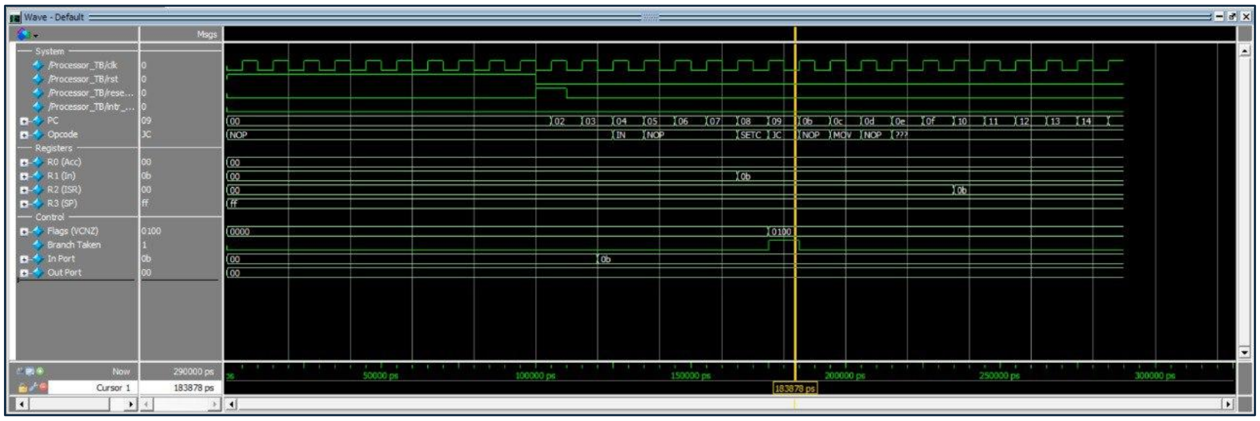
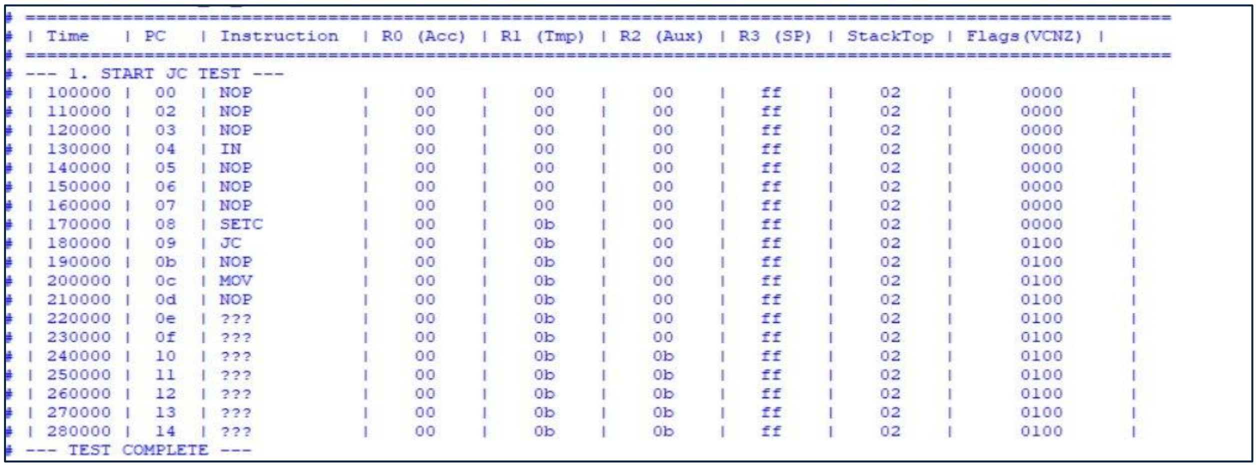


Figure 23:JC waveform

Transcript:



```
# ==================================================================================================
# | Time    | PC    | Instruction     | R0 (Acc) | R1 (Tmp) | R2 (Aux) | R3 (SP) | StackTop | Flags(VCNZ) |
# ==================================================================================================
# --- 1. START JC TEST ---
# | 100000 |  00   | NOP             |    00    |    00    |    00    |   ff    |    02    |     0000     |
# | 110000 |  02   | NOP             |    00    |    00    |    00    |   ff    |    02    |     0000     |
# | 120000 |  03   | NOP             |    00    |    00    |    00    |   ff    |    02    |     0000     |
# | 130000 |  04   | IN              |    00    |    00    |    00    |   ff    |    02    |     0000     |
# | 140000 |  05   | NOP             |    00    |    00    |    00    |   ff    |    02    |     0000     |
# | 150000 |  06   | NOP             |    00    |    00    |    00    |   ff    |    02    |     0000     |
# | 160000 |  07   | NOP             |    00    |    00    |    00    |   ff    |    02    |     0000     |
# | 170000 |  08   | SETC            |    00    |    0b    |    00    |   ff    |    02    |     0000     |
# | 180000 |  09   | JC              |    00    |    0b    |    00    |   ff    |    02    |     0100     |
# | 190000 |  0b   | NOP             |    00    |    0b    |    00    |   ff    |    02    |     0100     |
# | 200000 |  0c   | MOV             |    00    |    0b    |    00    |   ff    |    02    |     0100     |
# | 210000 |  0d   | NOP             |    00    |    0b    |    00    |   ff    |    02    |     0100     |
# | 220000 |  0e   | ???             |    00    |    0b    |    00    |   ff    |    02    |     0100     |
# | 230000 |  0f   | ???             |    00    |    0b    |    00    |   ff    |    02    |     0100     |
# | 240000 |  10   | ???             |    00    |    0b    |    0b    |   ff    |    02    |     0100     |
# | 250000 |  11   | ???             |    00    |    0b    |    0b    |   ff    |    02    |     0100     |
# | 260000 |  12   | ???             |    00    |    0b    |    0b    |   ff    |    02    |     0100     |
# | 270000 |  13   | ???             |    00    |    0b    |    0b    |   ff    |    02    |     0100     |
# | 280000 |  14   | ???             |    00    |    0b    |    0b    |   ff    |    02    |     0100     |
# --- TEST COMPLETE ---
```

Figure 24:JC transcript

3.8 Testing JV

Code:

```
IN R1        ; Load jump target address into R1
NOP          ; Delay
NOP          ; Delay
NOP          ; Delay
IN R0        ; Load value (e.g. 0x40) to cause overflow
ADD R0, R0   ; R0 + R0 causes Signed Overflow (V=1)
JV R1        ; Jump to address in R1 (Taken because V=1)
SETC R0      ; (Skipped instruction)
MOV R2, R1   ; Execution resumes here (Jump Target)
```

Waveform:

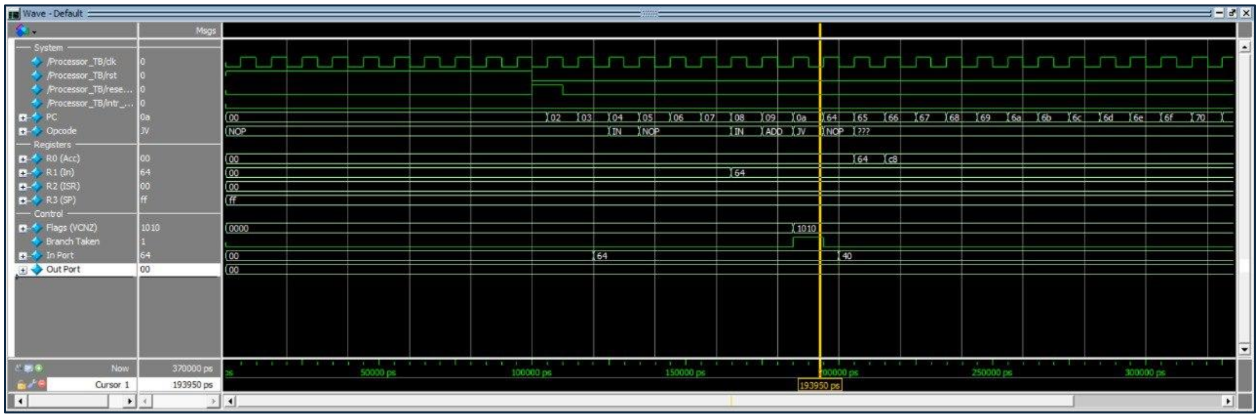
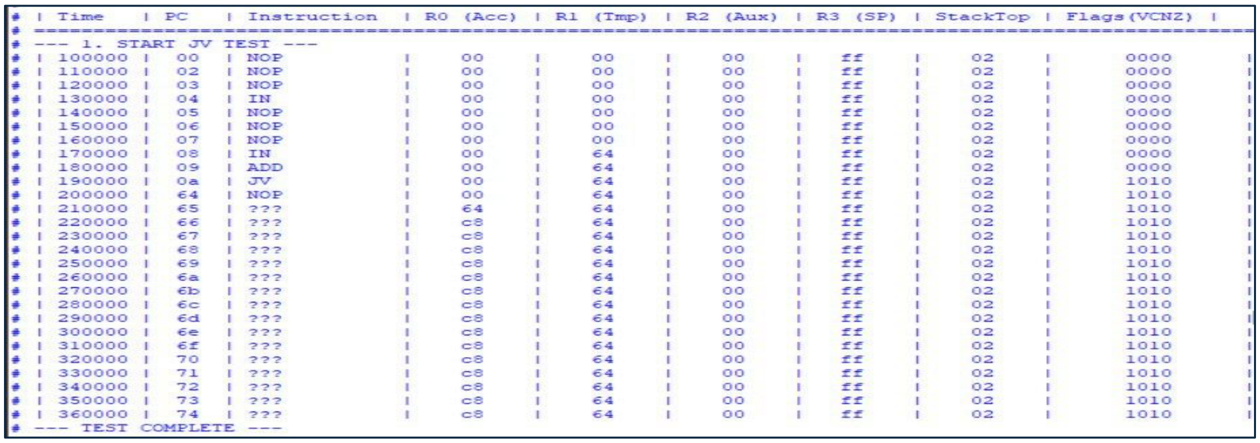


Figure 25:JV waveform

Transcript:



```
# | Time    | PC   | Instruction    | R0 (Acc) | R1 (Tmp) | R2 (Aux) | R3 (SP) | StackTop | Flags(VCNZ) |
# ==================================================================================================================
# --- 1. START JV TEST ---
# | 100000 |  00  | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 110000 |  02  | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 120000 |  03  | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 130000 |  04  | IN             |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 140000 |  05  | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 150000 |  06  | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 160000 |  07  | NOP            |    00    |    00    |    00    |   ff    |    02    |    0000     |
# | 170000 |  08  | IN             |    00    |    64    |    00    |   ff    |    02    |    0000     |
# | 180000 |  09  | ADD            |    00    |    64    |    00    |   ff    |    02    |    0000     |
# | 190000 |  0a  | JV             |    00    |    64    |    00    |   ff    |    02    |    1010     |
# | 200000 |  64  | NOP            |    00    |    64    |    00    |   ff    |    02    |    1010     |
# | 210000 |  65  | ???            |    64    |    64    |    00    |   ff    |    02    |    1010     |
# | 220000 |  66  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 230000 |  67  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 240000 |  68  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 250000 |  69  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 260000 |  6a  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 270000 |  6b  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 280000 |  6c  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 290000 |  6d  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 300000 |  6e  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 310000 |  6f  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 320000 |  70  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 330000 |  71  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 340000 |  72  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 350000 |  73  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# | 360000 |  74  | ???            |    c8    |    64    |    00    |   ff    |    02    |    1010     |
# --- TEST COMPLETE ---
```

Figure 26: JV transcript

3.9 Testing JMP

Code:

```
IN R1        ; Load jump target address into R1
NOP          ; Delay
NOP          ; Delay
NOP          ; Delay
JMP R1       ; Unconditionally Jump to address in R1
SETC         ; (Skipped instruction)
SETC         ; (Skipped instruction)
MOV R2, R1   ; Execution resumes here (Jump Target)
```

Waveform:

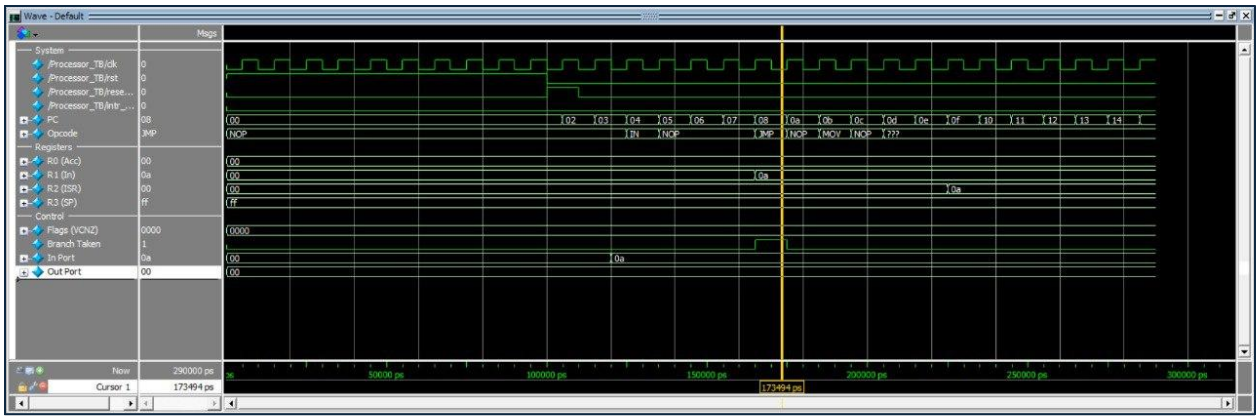
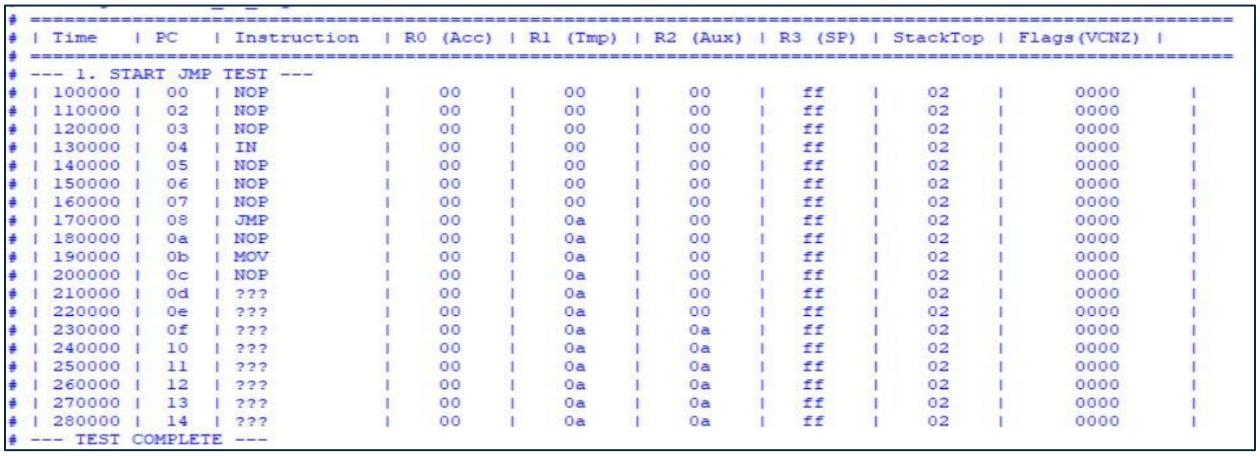


Figure 27:JMP waveform

Transcript:



```
# ==============================================================================================
# | Time    | PC    | Instruction   | R0 (Acc)  | R1 (Tmp)  | R2 (Aux)  | R3 (SP)  | StackTop  | Flags(VCNZ)  |
# ==============================================================================================
# --- 1. START JMP TEST ---
# | 100000 |  00   | NOP           |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 110000 |  02   | NOP           |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 120000 |  03   | NOP           |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 130000 |  04   | IN            |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 140000 |  05   | NOP           |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 150000 |  06   | NOP           |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 160000 |  07   | NOP           |    00     |    00     |    00     |    ff    |    02     |    0000      |
# | 170000 |  08   | JMP           |    00     |    0a     |    00     |    ff    |    02     |    0000      |
# | 180000 |  0a   | NOP           |    00     |    0a     |    00     |    ff    |    02     |    0000      |
# | 190000 |  0b   | MOV           |    00     |    0a     |    00     |    ff    |    02     |    0000      |
# | 200000 |  0c   | NOP           |    00     |    0a     |    00     |    ff    |    02     |    0000      |
# | 210000 |  0d   | ???           |    00     |    0a     |    00     |    ff    |    02     |    0000      |
# | 220000 |  0e   | ???           |    00     |    0a     |    00     |    ff    |    02     |    0000      |
# | 230000 |  0f   | ???           |    00     |    0a     |    0a     |    ff    |    02     |    0000      |
# | 240000 |  10   | ???           |    00     |    0a     |    0a     |    ff    |    02     |    0000      |
# | 250000 |  11   | ???           |    00     |    0a     |    0a     |    ff    |    02     |    0000      |
# | 260000 |  12   | ???           |    00     |    0a     |    0a     |    ff    |    02     |    0000      |
# | 270000 |  13   | ???           |    00     |    0a     |    0a     |    ff    |    02     |    0000      |
# | 280000 |  14   | ???           |    00     |    0a     |    0a     |    ff    |    02     |    0000      |
# --- TEST COMPLETE ---
```

Figure 28: JMP transcript

3.10 Testing Loop

This test program verifies the functionality of the LOOP instruction. It initializes a counter (R0), a jump target address (R1), and an accumulator (R2). Inside the loop, R2 is incremented and output to the port. The loop repeats 3 times until R0 decrements to zero.

Code:

```
IN R0        (Load Loop Count = 3)
IN R1        (Load Target Address = 05)
IN R2        (Initialize Accumulator = 0)
INC R2        (Start of Loop / Label: 05)
OUT R2        (Output current value)
LOOP R0, R1   (Decrement R0, Jump to R1 if R0 != 0)
OUT R0          (Loop Finished)
OUT R1
OUT R2
```

Waveform:

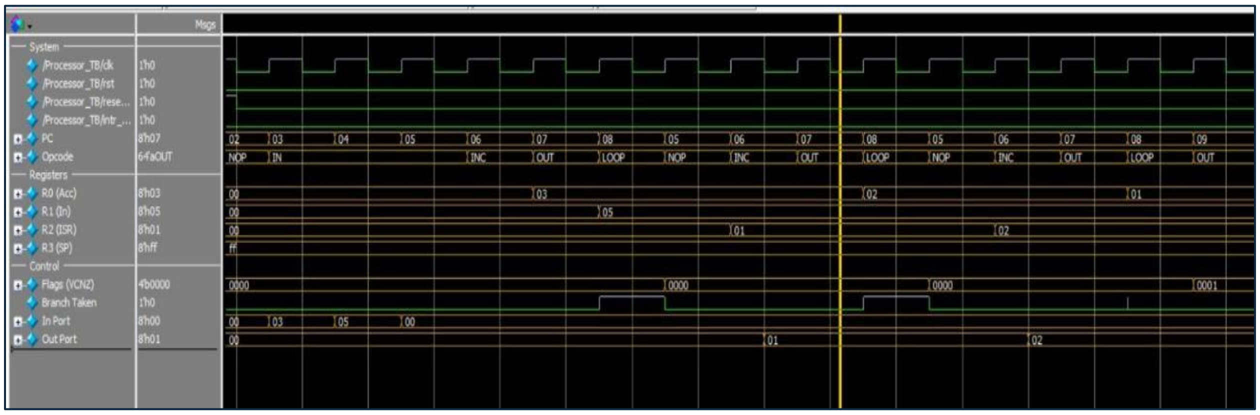
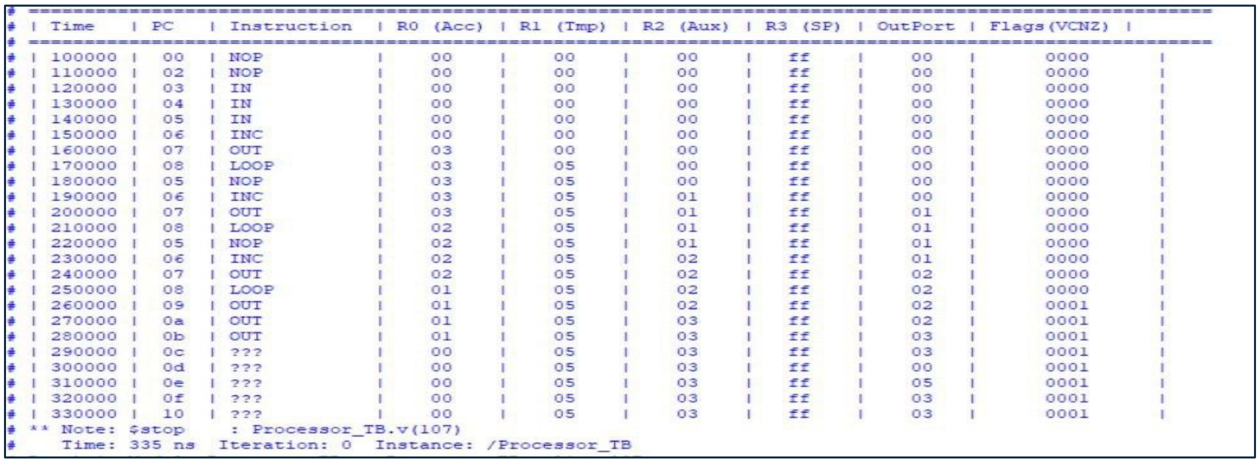


Figure 29:Loop waveform

Transcript:



```
# ==========================================================================================================
# | Time     | PC    | Instruction    | R0 (Acc) | R1 (Tmp) | R2 (Aux) | R3 (SP) | OutPort | Flags(VCNZ) |
# ==========================================================================================================
# | 100000 |  00  | NOP          |    00    |    00    |    00    |   ff    |   00    |      0000     |
# | 110000 |  02  | NOP          |    00    |    00    |    00    |   ff    |   00    |      0000     |
# | 120000 |  03  | IN           |    00    |    00    |    00    |   ff    |   00    |      0000     |
# | 130000 |  04  | IN           |    00    |    00    |    00    |   ff    |   00    |      0000     |
# | 140000 |  05  | IN           |    00    |    00    |    00    |   ff    |   00    |      0000     |
# | 150000 |  06  | INC          |    00    |    00    |    00    |   ff    |   00    |      0000     |
# | 160000 |  07  | OUT          |    03    |    00    |    00    |   ff    |   00    |      0000     |
# | 170000 |  08  | LOOP         |    03    |    05    |    00    |   ff    |   00    |      0000     |
# | 180000 |  05  | NOP          |    03    |    05    |    00    |   ff    |   00    |      0000     |
# | 190000 |  06  | INC          |    03    |    05    |    01    |   ff    |   00    |      0000     |
# | 200000 |  07  | OUT          |    03    |    05    |    01    |   ff    |   01    |      0000     |
# | 210000 |  08  | LOOP         |    02    |    05    |    01    |   ff    |   01    |      0000     |
# | 220000 |  05  | NOP          |    02    |    05    |    01    |   ff    |   01    |      0000     |
# | 230000 |  06  | INC          |    02    |    05    |    02    |   ff    |   01    |      0000     |
# | 240000 |  07  | OUT          |    02    |    05    |    02    |   ff    |   02    |      0000     |
# | 250000 |  08  | LOOP         |    01    |    05    |    02    |   ff    |   02    |      0000     |
# | 260000 |  09  | OUT          |    01    |    05    |    02    |   ff    |   02    |      0001     |
# | 270000 |  0a  | OUT          |    01    |    05    |    03    |   ff    |   02    |      0001     |
# | 280000 |  0b  | OUT          |    01    |    05    |    03    |   ff    |   03    |      0001     |
# | 290000 |  0c  | ???          |    00    |    05    |    03    |   ff    |   03    |      0001     |
# | 300000 |  0d  | ???          |    00    |    05    |    03    |   ff    |   00    |      0001     |
# | 310000 |  0e  | ???          |    00    |    05    |    03    |   ff    |   05    |      0001     |
# | 320000 |  0f  | ???          |    00    |    05    |    03    |   ff    |   03    |      0001     |
# | 330000 |  10  | ???          |    00    |    05    |    03    |   ff    |   03    |      0001     |
# ** Note: $stop    : Processor_TB.v(107)
#    Time: 335 ns  Iteration: 0  Instance: /Processor_TB
```

Figure 30: Loop transcript

3.11 Testing CALL RET

The NOPs appearing after RET (at addresses 0F, 10, 11) represent Pipeline Stalls (Bubbles). Since RET is a Control Flow instruction that requires reading from Memory (Stack) to retrieve the PC, the pipeline takes multiple cycles (Fetch, Decode, Execute, Memory) to resolve the correct Target Address. During these cycles, the processor blindly fetches the next sequential instructions. Once the RET address is ready, the Hazard Unit flushes these wrong instructions, creating the visible delay before jumping back to the correct address (06)

Code:

```
IN R0       (Load Data = 10)
IN R1       (Load Subroutine Address = 0A)
OUT R0      (Show Data '10' before Call)
CALL R1     (Push Return Address '06' to Stack, Jump to '0A')
NOP         (Wait/Delay Slot - *Instruction at Return Address 06*)
INC R0      (Back in Main: Increment 10 -> 11)
OUT R0      (Output 11 - Verifies successful return)
JMP R1      (End of Test Loop)
...
NOT R0      (Subroutine Start @0A: Invert 10 -> F5)
OUT R0      (Output F5 - Inside Subroutine)
NOT R0      (Invert back -> 10)
RET         (Pop Address '06' from Stack, Jump back)
```

Waveform:

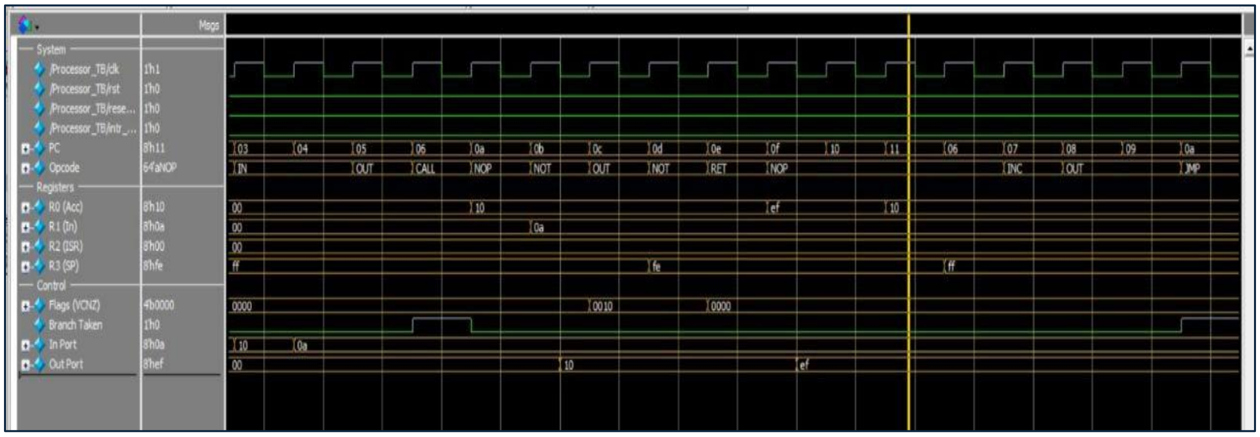
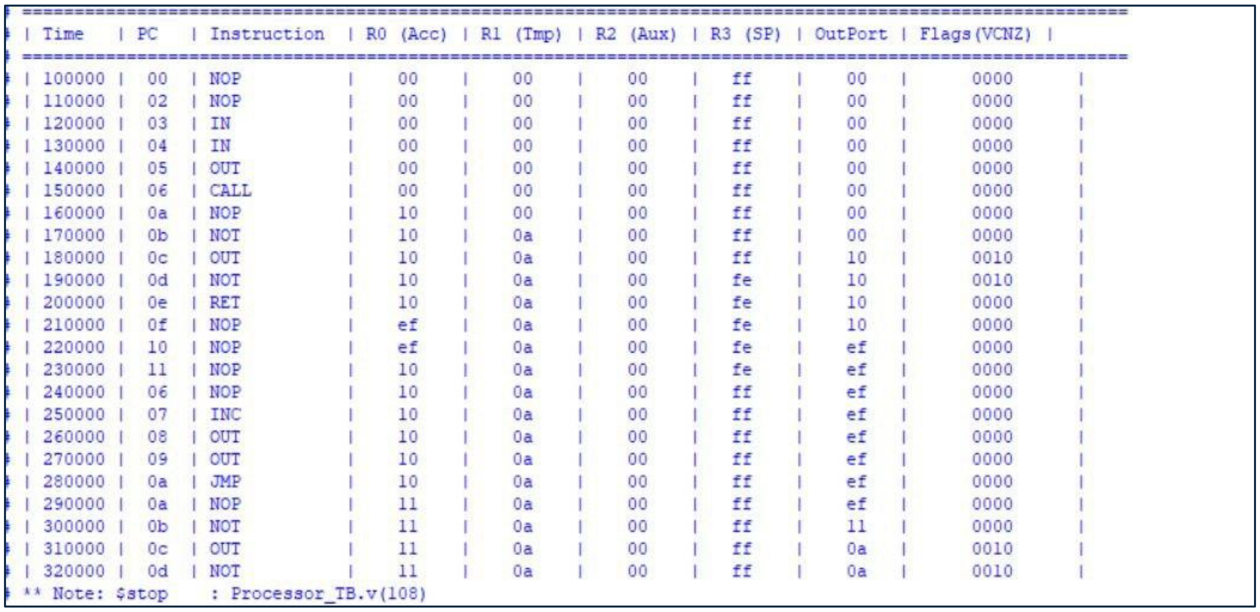


Figure 31: CALL RET waveform

Transcript:



```
 =========================================================================================================
 | Time    | PC   | Instruction   | R0 (Acc)  | R1 (Tmp)  | R2 (Aux)  | R3 (SP)  | OutPort | Flags(VCNZ) |
 =========================================================================================================
 | 100000 |  00  | NOP          |    00     |    00     |    00     |    ff    |    00   |      0000     |
 | 110000 |  02  | NOP          |    00     |    00     |    00     |    ff    |    00   |      0000     |
 | 120000 |  03  | IN           |    00     |    00     |    00     |    ff    |    00   |      0000     |
 | 130000 |  04  | IN           |    00     |    00     |    00     |    ff    |    00   |      0000     |
 | 140000 |  05  | OUT          |    00     |    00     |    00     |    ff    |    00   |      0000     |
 | 150000 |  06  | CALL         |    00     |    00     |    00     |    ff    |    00   |      0000     |
 | 160000 |  0a  | NOP          |    10     |    00     |    00     |    ff    |    00   |      0000     |
 | 170000 |  0b  | NOT          |    10     |    0a     |    00     |    ff    |    00   |      0000     |
 | 180000 |  0c  | OUT          |    10     |    0a     |    00     |    ff    |    10   |      0010     |
 | 190000 |  0d  | NOT          |    10     |    0a     |    00     |    fe    |    10   |      0010     |
 | 200000 |  0e  | RET          |    10     |    0a     |    00     |    fe    |    10   |      0000     |
 | 210000 |  0f  | NOP          |    ef     |    0a     |    00     |    fe    |    10   |      0000     |
 | 220000 |  10  | NOP          |    ef     |    0a     |    00     |    fe    |    ef   |      0000     |
 | 230000 |  11  | NOP          |    10     |    0a     |    00     |    fe    |    ef   |      0000     |
 | 240000 |  06  | NOP          |    10     |    0a     |    00     |    ff    |    ef   |      0000     |
 | 250000 |  07  | INC          |    10     |    0a     |    00     |    ff    |    ef   |      0000     |
 | 260000 |  08  | OUT          |    10     |    0a     |    00     |    ff    |    ef   |      0000     |
 | 270000 |  09  | OUT          |    10     |    0a     |    00     |    ff    |    ef   |      0000     |
 | 280000 |  0a  | JMP          |    10     |    0a     |    00     |    ff    |    ef   |      0000     |
 | 290000 |  0a  | NOP          |    11     |    0a     |    00     |    ff    |    ef   |      0000     |
 | 300000 |  0b  | NOT          |    11     |    0a     |    00     |    ff    |    11   |      0000     |
 | 310000 |  0c  | OUT          |    11     |    0a     |    00     |    ff    |    0a   |      0010     |
 | 320000 |  0d  | NOT          |    11     |    0a     |    00     |    ff    |    0a   |      0010     |
 ** Note: $stop    : Processor_TB.v(108)
```

Figure 32: CALL RET transcript

3.12 Testing INTERRUPT RTI

Code:

Technical Note: **on Interrupt Latency: You may observe approximately 2 NOPs (or bubbles) in the execution trace immediately after the interrupt signal is asserted. This is expected behavior caused by the Pipeline Flush. When an interrupt occurs, the processor must discard (flush) the instructions currently in the Fetch and Decode stages to safely vector to the ISR address (E0) without executing valid code from the previous stream.**

```
// --- Main Program (Starts at 02) ---
IN R0        (Load First Operand = 10)
IN R1        (Load Second Operand = 5)
ADD R0, R1    (Main Task: R0 = 10 + 5 = 15 -> Hex 0F)
OUT R0        (Output Result 0F - Before Interrupt)

// --- Interrupt Vulnerable Zone ---
NOP          (Wait for Interrupt...)
NOP          (Wait for Interrupt...)

// --- Return Point (Executes after RTI) ---
SUB R0, R1    (Resume Task: R0 = 15 - 5 = 10 -> Hex 0A)
OUT R0        (Output 0A - Verifies Successful Return)
```

OUT R1   (End of Program)
// ...

// --- ISR (Interrupt Service Routine at E0) ---
@E0
IN R2   (Load Interrupt Indicator = AA)
OUT R2   (Output AA - Proves we are inside ISR)
RTI   (Return from Interrupt - Pop PC & Flags)

Waveform:

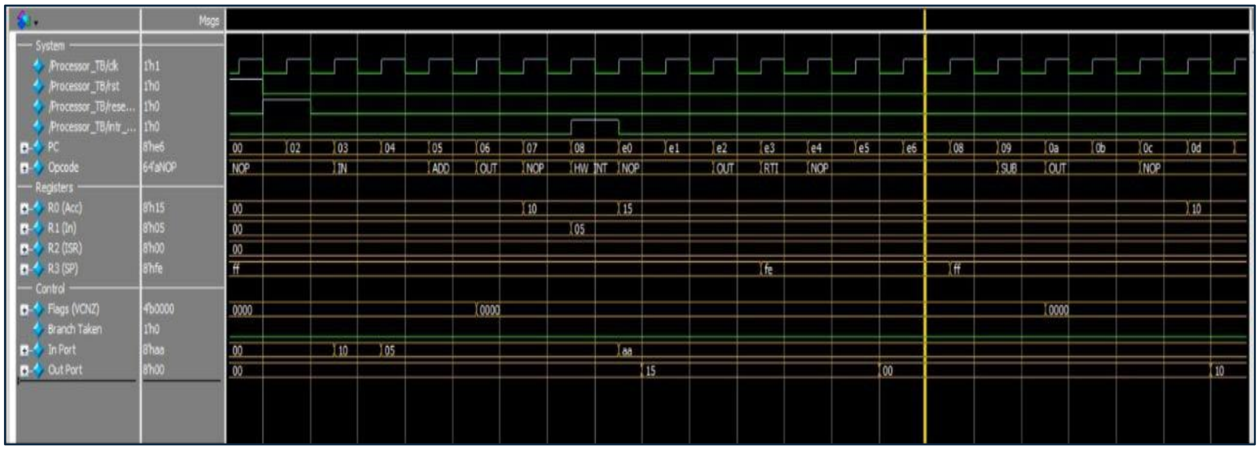
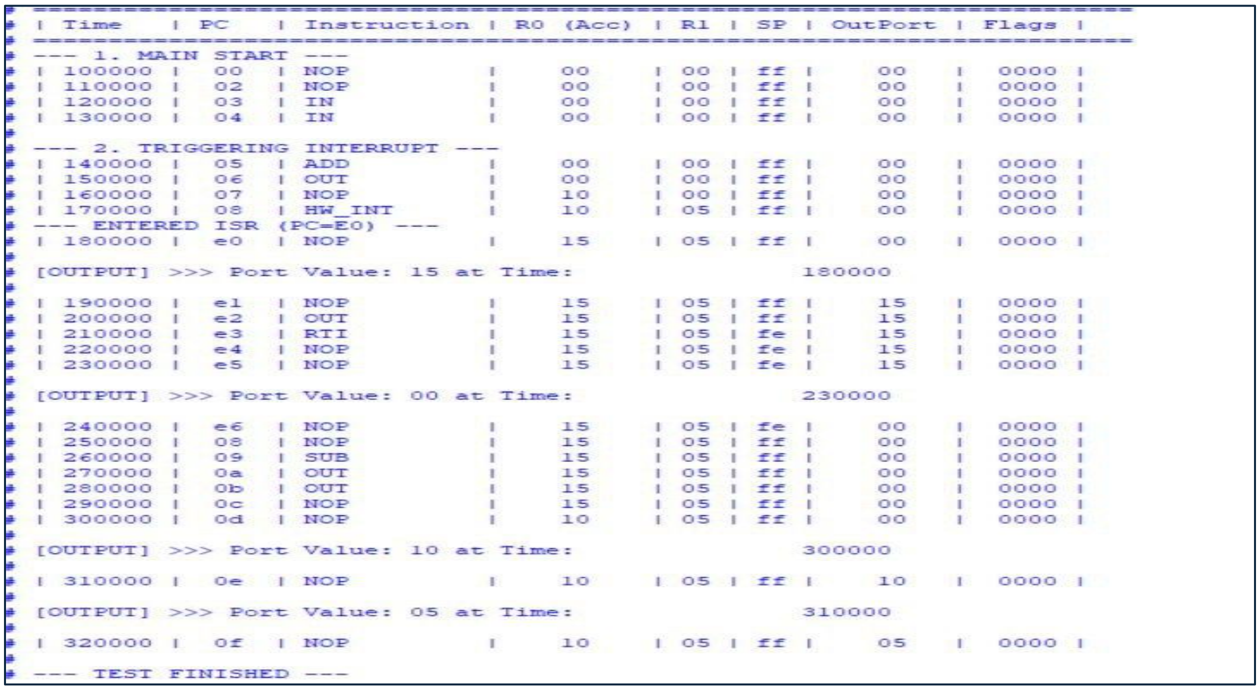


Figure 33: INTERRUPT RTI waveform

Transcript:



```
# ========================================================================
# | Time     | PC    | Instruction | R0 (Acc) | R1 | SP | OutPort | Flags |
# ========================================================================
# --- 1. MAIN START ---
# | 100000 |  00   | NOP          |     00    | 00 | ff |    00    |  0000 |
# | 110000 |  02   | NOP          |     00    | 00 | ff |    00    |  0000 |
# | 120000 |  03   | IN           |     00    | 00 | ff |    00    |  0000 |
# | 130000 |  04   | IN           |     00    | 00 | ff |    00    |  0000 |
#
# --- 2. TRIGGERING INTERRUPT ---
# | 140000 |  05   | ADD          |     00    | 00 | ff |    00    |  0000 |
# | 150000 |  06   | OUT          |     00    | 00 | ff |    00    |  0000 |
# | 160000 |  07   | NOP          |     10    | 00 | ff |    00    |  0000 |
# | 170000 |  08   | HW_INT       |     10    | 05 | ff |    00    |  0000 |
# --- ENTERED ISR (PC=E0) ---
# | 180000 |  e0   | NOP          |     15    | 05 | ff |    00    |  0000 |
#
# [OUTPUT] >>> Port Value: 15 at Time:                    180000
#
# | 190000 |  e1   | NOP          |     15    | 05 | ff |    15    |  0000 |
# | 200000 |  e2   | OUT          |     15    | 05 | ff |    15    |  0000 |
# | 210000 |  e3   | RTI          |     15    | 05 | fe |    15    |  0000 |
# | 220000 |  e4   | NOP          |     15    | 05 | fe |    15    |  0000 |
# | 230000 |  e5   | NOP          |     15    | 05 | fe |    15    |  0000 |
#
# [OUTPUT] >>> Port Value: 00 at Time:                    230000
#
# | 240000 |  e6   | NOP          |     15    | 05 | fe |    00    |  0000 |
# | 250000 |  08   | NOP          |     15    | 05 | ff |    00    |  0000 |
# | 260000 |  09   | SUB          |     15    | 05 | ff |    00    |  0000 |
# | 270000 |  0a   | OUT          |     15    | 05 | ff |    00    |  0000 |
# | 280000 |  0b   | OUT          |     15    | 05 | ff |    00    |  0000 |
# | 290000 |  0c   | NOP          |     15    | 05 | ff |    00    |  0000 |
# | 300000 |  0d   | NOP          |     10    | 05 | ff |    00    |  0000 |
#
# [OUTPUT] >>> Port Value: 10 at Time:                    300000
#
# | 310000 |  0e   | NOP          |     10    | 05 | ff |    10    |  0000 |
#
# [OUTPUT] >>> Port Value: 05 at Time:                    310000
#
# | 320000 |  0f   | NOP          |     10    | 05 | ff |    05    |  0000 |
#
# --- TEST FINISHED ---
```

Figure 34: INTERRUPT RTI transcript

3.14 Testing LDI STI

Code:

```
IN R1
IN R2
STI M[R1], R2
LDI R0, M[R1]
NOP
NOP
```

Waveform:

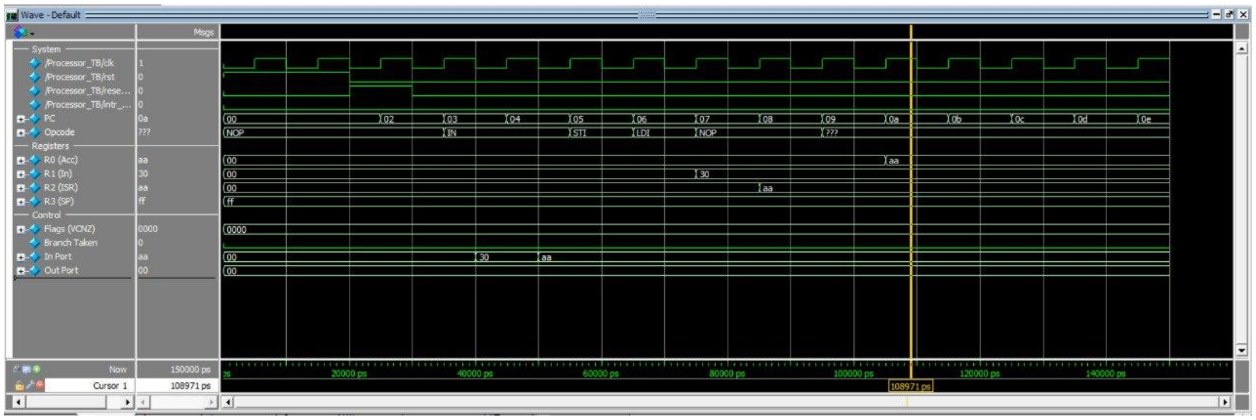
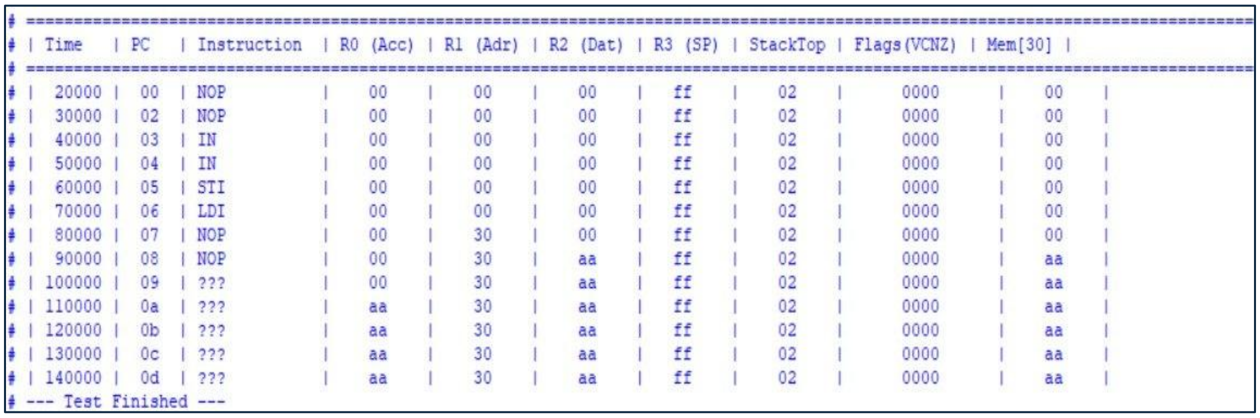


Figure 35: LDI STI waveform

Transcript:



```
# ===========================================================================================================
# | Time   | PC   | Instruction  | R0 (Acc) | R1 (Adr) | R2 (Dat) | R3 (SP) | StackTop | Flags(VCNZ) | Mem[30] |
# ===========================================================================================================
# |  20000 |  00  | NOP          |    00    |    00    |    00    |   ff    |    02    |    0000     |   00    |
# |  30000 |  02  | NOP          |    00    |    00    |    00    |   ff    |    02    |    0000     |   00    |
# |  40000 |  03  | IN           |    00    |    00    |    00    |   ff    |    02    |    0000     |   00    |
# |  50000 |  04  | IN           |    00    |    00    |    00    |   ff    |    02    |    0000     |   00    |
# |  60000 |  05  | STI          |    00    |    00    |    00    |   ff    |    02    |    0000     |   00    |
# |  70000 |  06  | LDI          |    00    |    00    |    00    |   ff    |    02    |    0000     |   00    |
# |  80000 |  07  | NOP          |    00    |    30    |    00    |   ff    |    02    |    0000     |   00    |
# |  90000 |  08  | NOP          |    00    |    30    |    aa    |   ff    |    02    |    0000     |   aa    |
# | 100000 |  09  | ???          |    00    |    30    |    aa    |   ff    |    02    |    0000     |   aa    |
# | 110000 |  0a  | ???          |    aa    |    30    |    aa    |   ff    |    02    |    0000     |   aa    |
# | 120000 |  0b  | ???          |    aa    |    30    |    aa    |   ff    |    02    |    0000     |   aa    |
# | 130000 |  0c  | ???          |    aa    |    30    |    aa    |   ff    |    02    |    0000     |   aa    |
# | 140000 |  0d  | ???          |    aa    |    30    |    aa    |   ff    |    02    |    0000     |   aa    |
# --- Test Finished ---
```

Figure 36: LDI STI transcript

3.15 Testing ADD SUB (focus on forwarding)

Code:

```
IN R0 (05)     ; Load 05 into R0
IN R1 (0A)     ; Load 10 (0xA) into R1
IN R2 (0A)     ; Load 10 (0xA) into R2
ADD R2, R1     ; R2 = 10 + 10 = 20 (0x14)
            ; Dependency: Result is in Pipeline, not yet in RegFile.
SUB R2, R0     ; R2 = 20 - 5 = 15 (0x0F)
            ; Forwarding: ALU uses result from previous ADD directly.
```

Waveform:

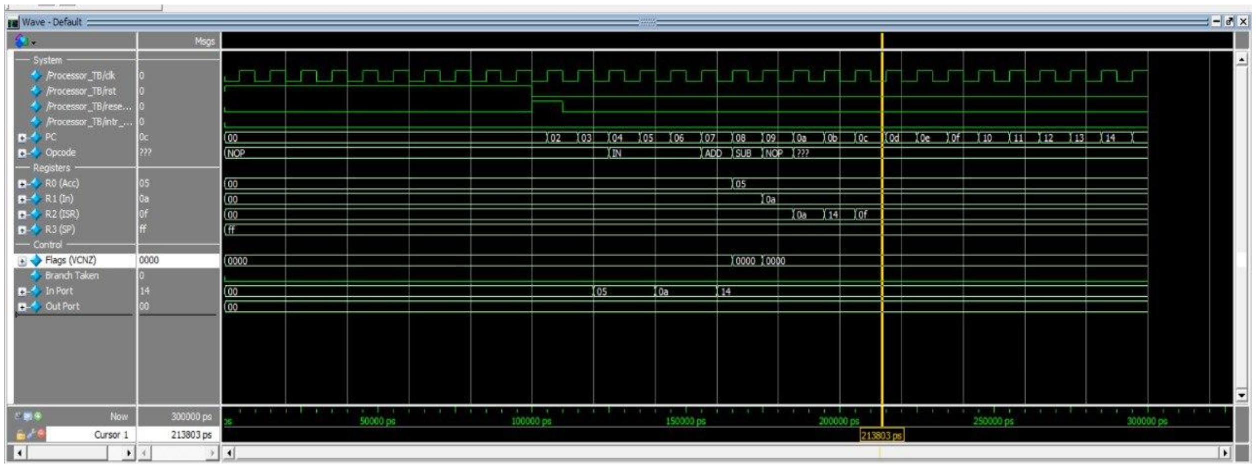
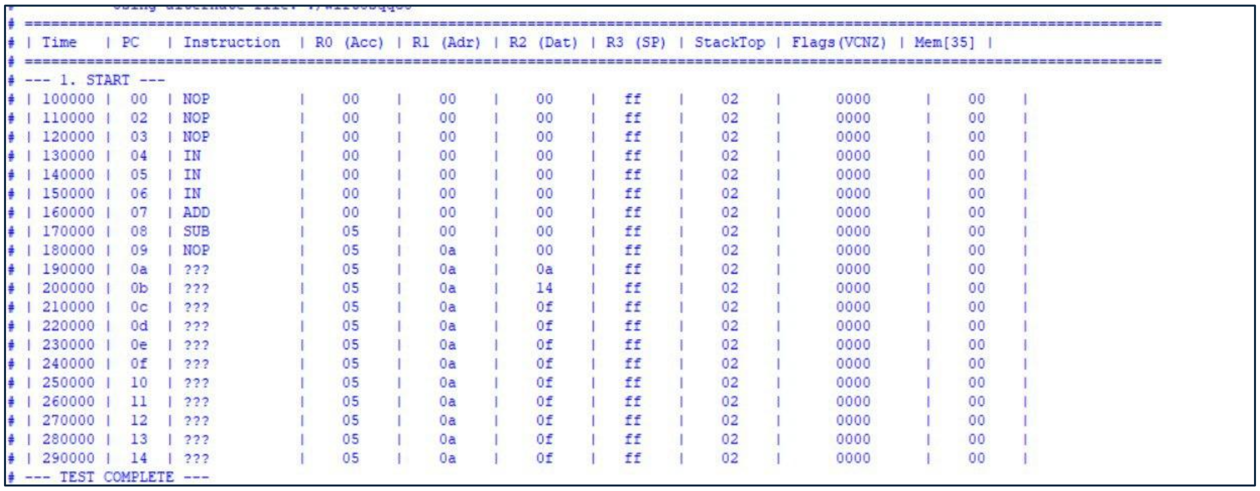


Figure 37:ADD SUB (focus on forwarding) waveform

Transcript:



```
# ==============================================================================================================
# | Time    | PC    | Instruction   | R0 (Acc) | R1 (Adr) | R2 (Dat) | R3 (SP) | StackTop | Flags(VCNZ) | Mem[35] |
# ==============================================================================================================
# --- 1. START ---
# | 100000 |  00   | NOP           |    00    |    00    |    00    |   ff    |    02    |     0000     |    00    |
# | 110000 |  02   | NOP           |    00    |    00    |    00    |   ff    |    02    |     0000     |    00    |
# | 120000 |  03   | NOP           |    00    |    00    |    00    |   ff    |    02    |     0000     |    00    |
# | 130000 |  04   | IN            |    00    |    00    |    00    |   ff    |    02    |     0000     |    00    |
# | 140000 |  05   | IN            |    00    |    00    |    00    |   ff    |    02    |     0000     |    00    |
# | 150000 |  06   | IN            |    00    |    00    |    00    |   ff    |    02    |     0000     |    00    |
# | 160000 |  07   | ADD           |    00    |    00    |    00    |   ff    |    02    |     0000     |    00    |
# | 170000 |  08   | SUB           |    05    |    00    |    00    |   ff    |    02    |     0000     |    00    |
# | 180000 |  09   | NOP           |    05    |    0a    |    00    |   ff    |    02    |     0000     |    00    |
# | 190000 |  0a   | ???           |    05    |    0a    |    0a    |   ff    |    02    |     0000     |    00    |
# | 200000 |  0b   | ???           |    05    |    0a    |    14    |   ff    |    02    |     0000     |    00    |
# | 210000 |  0c   | ???           |    05    |    0a    |    0f    |   ff    |    02    |     0000     |    00    |
# | 220000 |  0d   | ???           |    05    |    0a    |    0f    |   ff    |    02    |     0000     |    00    |
# | 230000 |  0e   | ???           |    05    |    0a    |    0f    |   ff    |    02    |     0000     |    00    |
# | 240000 |  0f   | ???           |    05    |    0a    |    0f    |   ff    |    02    |     0000     |    00    |
# | 250000 |  10   | ???           |    05    |    0a    |    0f    |   ff    |    02    |     0000     |    00    |
# | 260000 |  11   | ???           |    05    |    0a    |    0f    |   ff    |    02    |     0000     |    00    |
# | 270000 |  12   | ???           |    05    |    0a    |    0f    |   ff    |    02    |     0000     |    00    |
# | 280000 |  13   | ???           |    05    |    0a    |    0f    |   ff    |    02    |     0000     |    00    |
# | 290000 |  14   | ???           |    05    |    0a    |    0f    |   ff    |    02    |     0000     |    00    |
# --- TEST COMPLETE ---
```

Figure 38:ADD SUB (focus on forwarding) transcript

4.system synthesis from Vivado

Target FPGA: Xilinx Artix-7 XC7A35TICPG236-1L

Board: Digilent Basys3 Rev B

Technology: 7-Series FPGA (28nm process, industrial grade)

Clock: 100MHz input, 66.67MHz processor operation (15ns period*)***

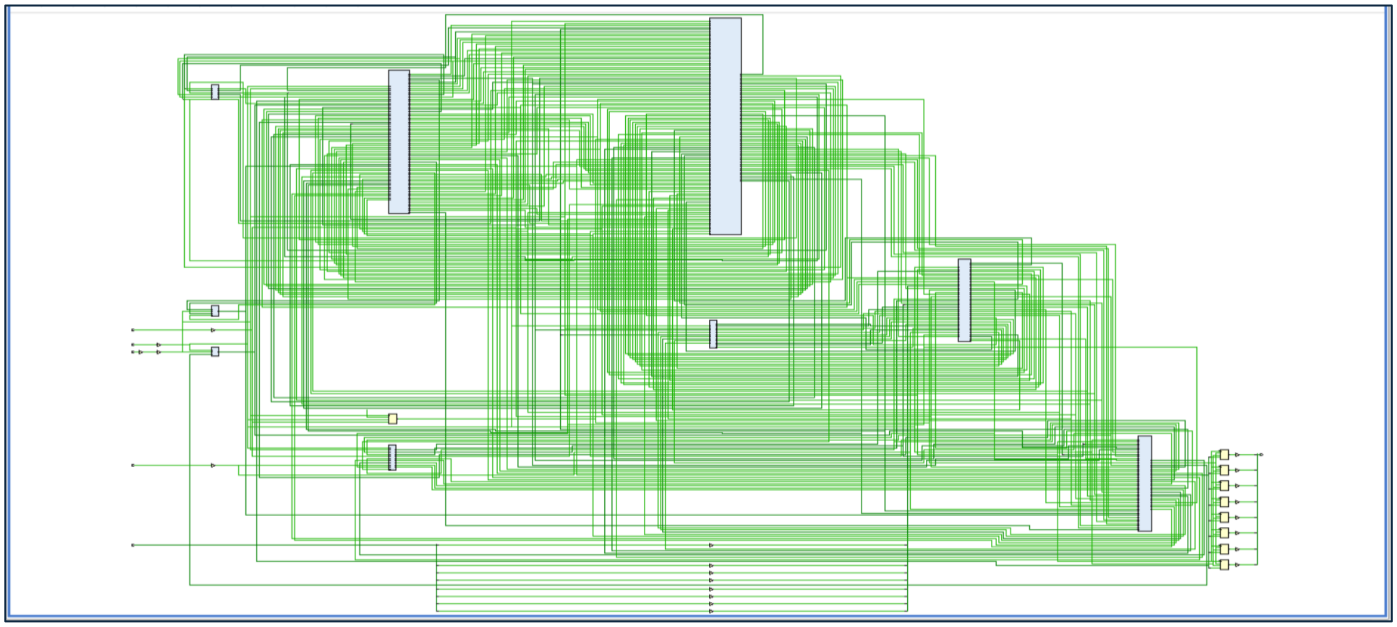
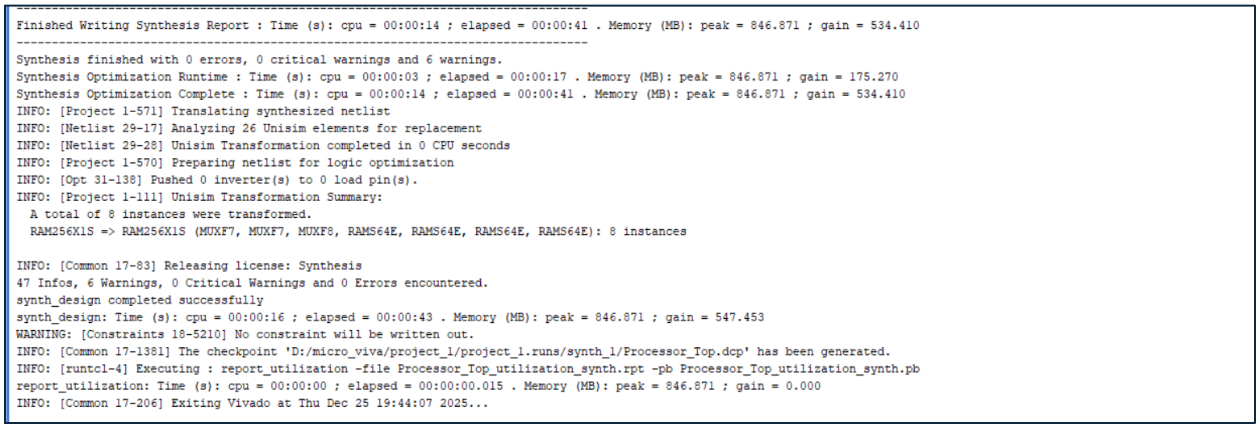


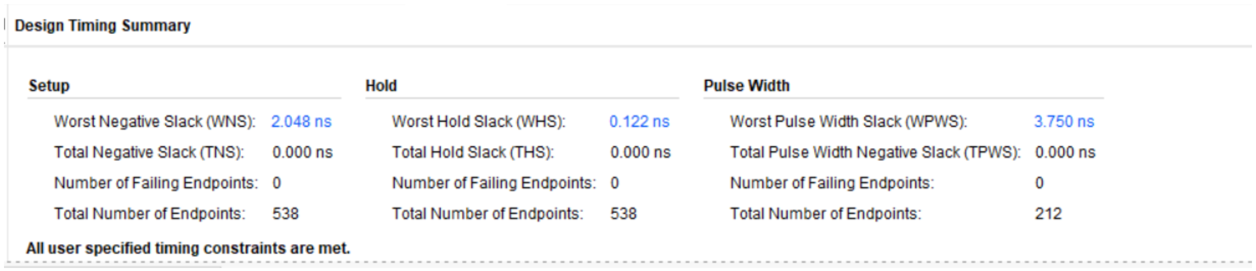
Figure 40:SCHEMATIC



```
--------------------------------------------------------------------------------
Finished Writing Synthesis Report : Time (s): cpu = 00:00:14 ; elapsed = 00:00:41 . Memory (MB): peak = 846.871 ; gain = 534.410
--------------------------------------------------------------------------------
Synthesis finished with 0 errors, 0 critical warnings and 6 warnings.
Synthesis Optimization Runtime : Time (s): cpu = 00:00:03 ; elapsed = 00:00:17 . Memory (MB): peak = 846.871 ; gain = 175.270
Synthesis Optimization Complete : Time (s): cpu = 00:00:14 ; elapsed = 00:00:41 . Memory (MB): peak = 846.871 ; gain = 534.410
INFO: [Project 1-571] Translating synthesized netlist
INFO: [Netlist 29-17] Analyzing 26 Unisim elements for replacement
INFO: [Netlist 29-28] Unisim Transformation completed in 0 CPU seconds
INFO: [Project 1-570] Preparing netlist for logic optimization
INFO: [Opt 31-138] Pushed 0 inverter(s) to 0 load pin(s).
INFO: [Project 1-111] Unisim Transformation Summary:
  A total of 8 instances were transformed.
  RAM256X1S => RAM256X1S (MUXF7, MUXF7, MUXF8, RAMS64E, RAMS64E, RAMS64E, RAMS64E): 8 instances

INFO: [Common 17-83] Releasing license: Synthesis
47 Infos, 6 Warnings, 0 Critical Warnings and 0 Errors encountered.
synth_design completed successfully
synth_design: Time (s): cpu = 00:00:16 ; elapsed = 00:00:43 . Memory (MB): peak = 846.871 ; gain = 547.453
WARNING: [Constraints 18-5210] No constraint will be written out.
INFO: [Common 17-1381] The checkpoint 'D:/micro_viva/project_1/project_1.runs/synth_1/Processor_Top.dcp' has been generated.
INFO: [runtcl-4] Executing : report_utilization -file Processor_Top_utilization_synth.rpt -pb Processor_Top_utilization_synth.pb
report_utilization: Time (s): cpu = 00:00:00 ; elapsed = 00:00:00.015 . Memory (MB): peak = 846.871 ; gain = 0.000
INFO: [Common 17-206] Exiting Vivado at Thu Dec 25 19:44:07 2025...
```

Figure 39:lOG message

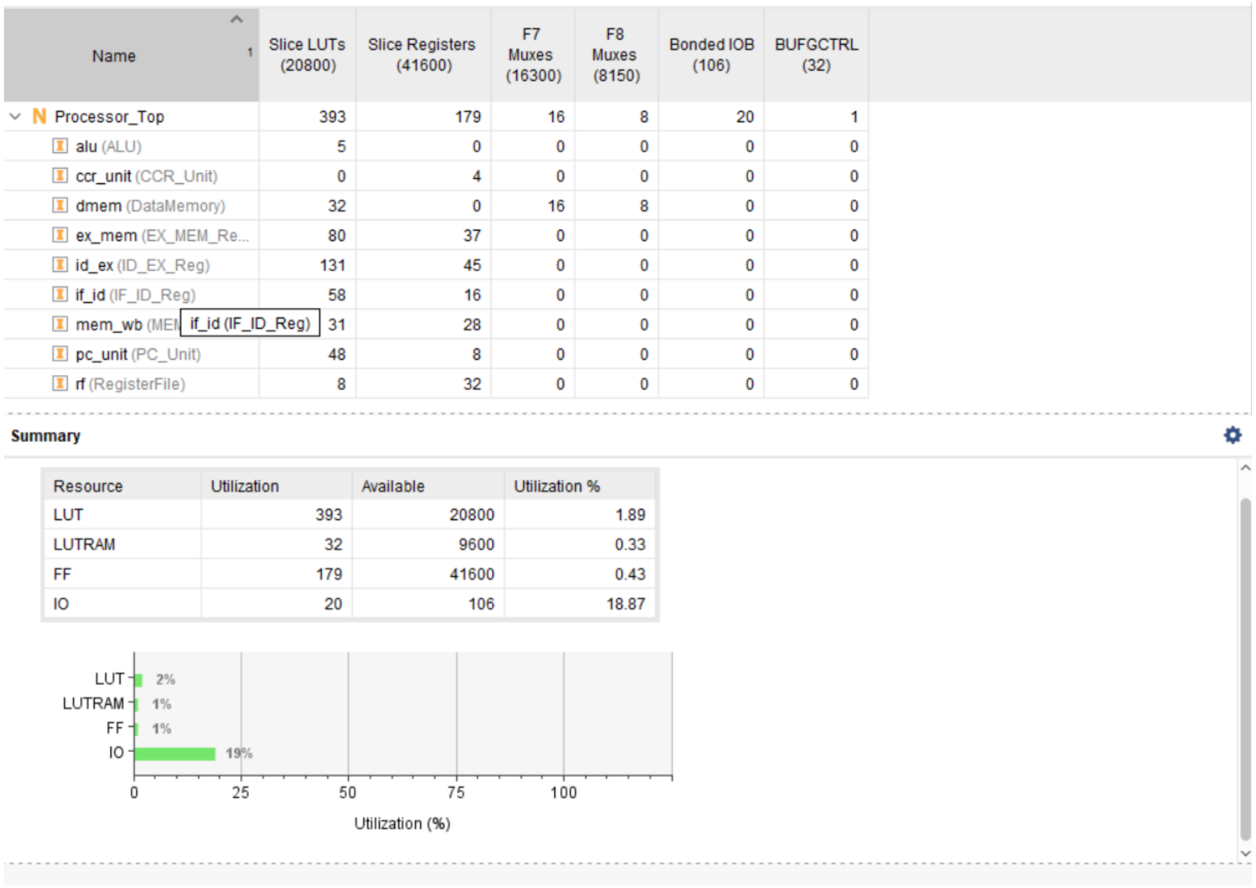


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2.048 ns	Worst Hold Slack (WHS):	0.122 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	538	Total Number of Endpoints:	538	Total Number of Endpoints:	212

All user specified timing constraints are met.

Figure 41: report time summary



Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (106)	BUFGCTRL (32)
Processor_Top	393	179	16	8	20	1
alu (ALU)	5	0	0	0	0	0
ccr_unit (CCR_Unit)	0	4	0	0	0	0
dmem (DataMemory)	32	0	16	8	0	0
ex_mem (EX_MEM_Re...	80	37	0	0	0	0
id_ex (ID_EX_Reg)	131	45	0	0	0	0
if_id (IF_ID_Reg)	58	16	0	0	0	0
mem_wb (MEI if_id (IF_ID_Reg)	31	28	0	0	0	0
pc_unit (PC_Unit)	48	8	0	0	0	0
rf (RegisterFile)	8	32	0	0	0	0

Summary

Resource	Utilization	Available	Utilization %
LUT	393	20800	1.89
LUTRAM	32	9600	0.33
FF	179	41600	0.43
IO	20	106	18.87

Figure 42:Utilization report

5.conclusion

This project successfully demonstrated the design and verification of an 8-bit RISC-like microprocessor based on **Harvard Architecture**. By decoupling instruction and data memory, the design effectively overcame structural hazards common in Von Neumann architectures, enabling efficient simultaneous memory access. The implementation of a five-stage pipeline (IF, ID, EX, MEM, WB), augmented by dedicated **Forwarding and Hazard detection units**, ensured high instruction throughput and robust handling of data dependencies.

The design was realized using Verilog HDL and validated through extensive simulation. The testing process confirmed the correct execution of a comprehensive instruction set, including arithmetic, logical, and control flow operations. The simulation results demonstrated that the processor functions correctly under various test scenarios, properly resolving pipeline hazards and executing interrupts as intended.

In summary, this project validates the feasibility of a custom lightweight RISC core, successfully bridging the gap between theoretical computer architecture concepts and practical hardware design.

6.Appendix

Link for our team video:

https://drive.google.com/drive/folders/1W-tap6-mNudc8pk-QNdkl7Do_5WeFi2l