Cairo University- Faculty of Engineering
Electronics and Communications Engineering Department
Advanced Processor Architecture (ELC 3030) – Winter 2025

Advanced Processor Architecture Project

8-bit Pipelined Von Neumann Processor

Bn	ID	Section	Name
36	9230242	1	امير اشرف لولى عباس
37	9230243	1	امير سامح فانوس عطيه فانوس
40	9230256	1	انس خالد السيد محمد السيد
25	9230664	3	كريم ايمن محمود يحيى
35	9231065	3	محمد احمد حمدى احمد
22	9230895	4	مصطفى محمد فرحان الخضرى

Under Supervision

Prof. Hossam A. H. Fahmy

Eng. Hassan M. F. El-Menier

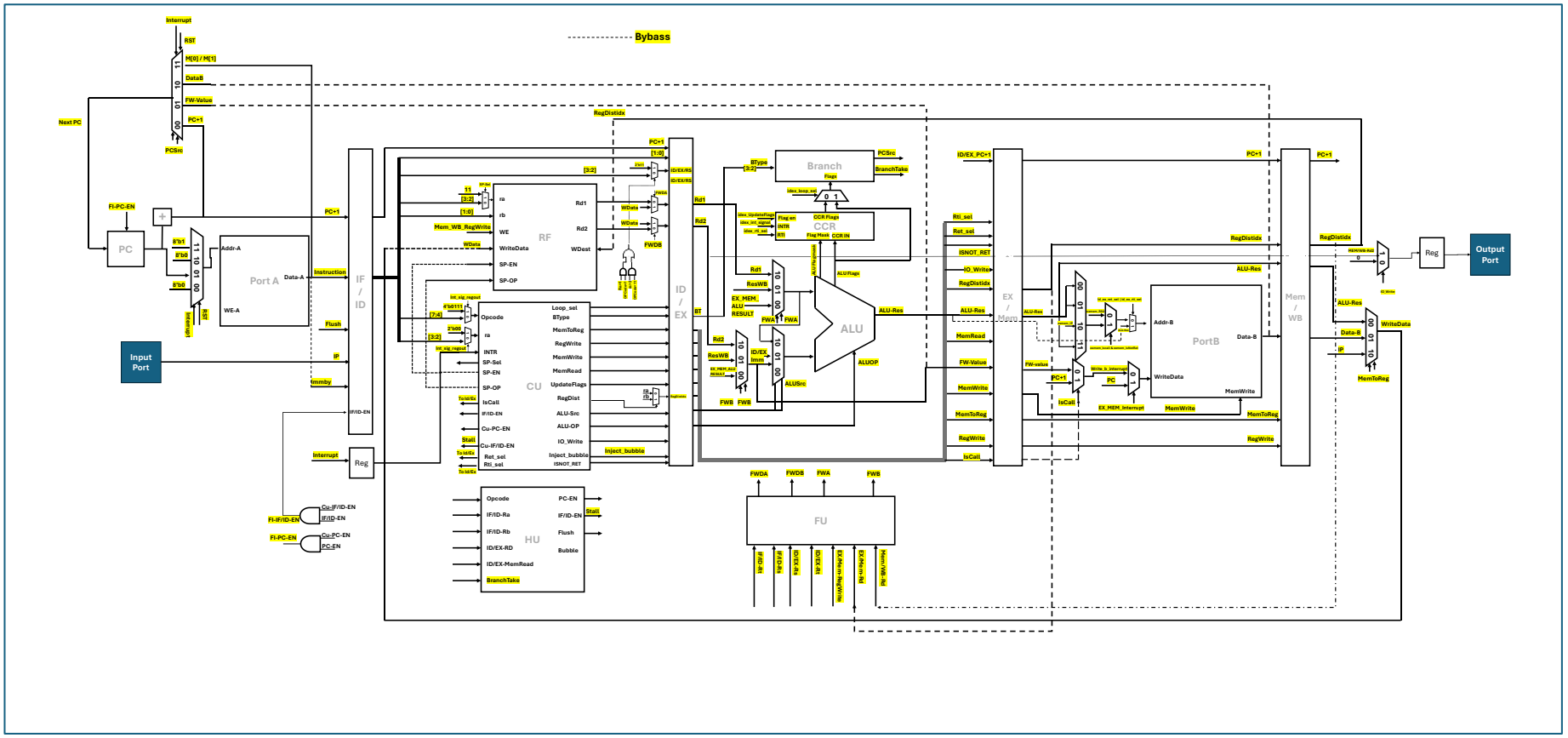
Demo Video: Demo Video

Table of Contents

I. Project Objective:

1) To design and implement an 8-bit pipelined Von Neumann processor based on the given ISA.
2) To realize a single shared memory system for instruction and data access.
3) To implement an FSM-based control unit for proper instruction sequencing.
4) To support pipeline execution with data forwarding for hazard resolution.
5) To verify correct operation through HDL simulation and waveform analysis.

II. Architecture:



This design implements a Five-Stage Pipelined Processor optimized for instruction throughput and hazard mitigation. The architecture follows a classic RISC-like structure, dividing execution into Instruction Fetch (IF), Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB) stages.

Key features of the design include:

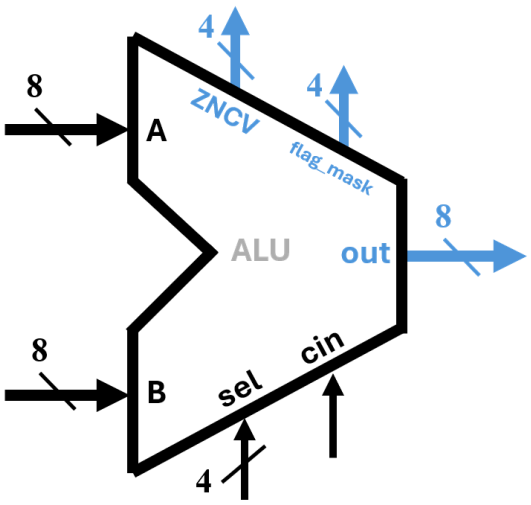
- Advanced Hazard Management: A dedicated Hazard Unit (HU) detects data dependencies and control hazards, automatically inserting stalls or flushing pipeline registers when necessary.
- Data Forwarding: A Forwarding Unit (FU) bypasses results from the MEM and WB stages directly to the ALU, minimizing stall cycles caused by Read-After-Write (RAW) dependencies.
- Integrated I/O: The processor includes dedicated Input and Output Ports, facilitating direct communication with external peripherals.
- Control Flow Handling: The design supports conditional branching and subroutine calls, with a Condition Code Register (CCR) managing status flags (Zero, Negative, Carry, Overflow) for logic decisions.
- Stack Support: The Control Unit includes logic for Stack Pointer (SP) operations, enabling nested subroutine calls and local variable management.

III. Main Modules

1. ALU

The Arithmetic Logic Unit (ALU) is a combinational execution unit responsible for performing arithmetic, logical, bit-manipulation, and flag-generation operations. In the proposed processor architecture, the ALU resides in the Execute (EX) stage of the 5-stage pipeline.

The ALU receives its operands and control signals from the ID/EX pipeline register, performs the selected operation within a single cycle, and forwards the result and updated flags to subsequent pipeline stages through the EX/MEM pipeline register.



A. Input Signals

Signal	Width	Description
A	8 bits	First operand, typically sourced from register file or forwarded result
B	8 bits	Second operand, typically sourced from register file, immediate, or forwarded result
sel	4 bits	ALU operation selector generated by the control unit
cin	1 bit	Carry-in input, mainly used for rotate operations

B. Output Signals

Signal	Width	Description
out	8 bits	ALU result
Z	1 bit	Zero flag
N	1 bit	Negative flag (MSB of result)
C	1 bit	Carry / borrow flag
V	1 bit	Signed overflow flag
flag_mask	4 bits	Indicates which flags are updated ([V C N Z])

The flag mask signal enables selective flag updates, allowing instructions that do not affect flags to preserve the previous flag state. This design choice is particularly suitable for a pipelined processor and simplifies flag forwarding and hazard handling.

C. Supported ALU Operations

- Data Movement Operations

Operation	Selector	Function	Flags Updated
PASS B	0001	out = B (MOV)	None
PASS A	1110	out = A	None
INC A	1111	out = A + 1	None

- Arithmetic Operations

Operation	Selector	Description	Flags
ADD	0010	Unsigned & signed addition	Z, N, C, V
SUB	0011	Subtraction (A - B)	Z, N, C, V
INC	1010	Increment B	Z, N, C, V
DEC	1011	Decrement B	Z, N, C, V
NEG	1001	Two’s complement of B	Z, N

Carry and overflow are calculated correctly using extended precision (temp_wide) and signed overflow detection logic.

- Logical Operations

Operation	Selector	Description	Flags
AND	0100	Bitwise AND	Z, N
OR	0101	Bitwise OR	Z, N
NOT	1000	Bitwise NOT of B	Z, N

- Bit Manipulation and Flag Control

Operation	Selector	Description	Flags
RLC	0110	Rotate left through carry	C
RRC	0111	Rotate right through carry	C
SETC	1100	Set carry flag	C
CLRC	1101	Clear carry flag	C

D. Flag Mask Mechanism

The flag_mask output is a 4-bit vector [V C N Z] that indicates which flags are modified by the current operation.

- 1111 → all flags updated
- 0011 → only Z and N updated
- 0100 → only C updated
- 0000 → no flags updated

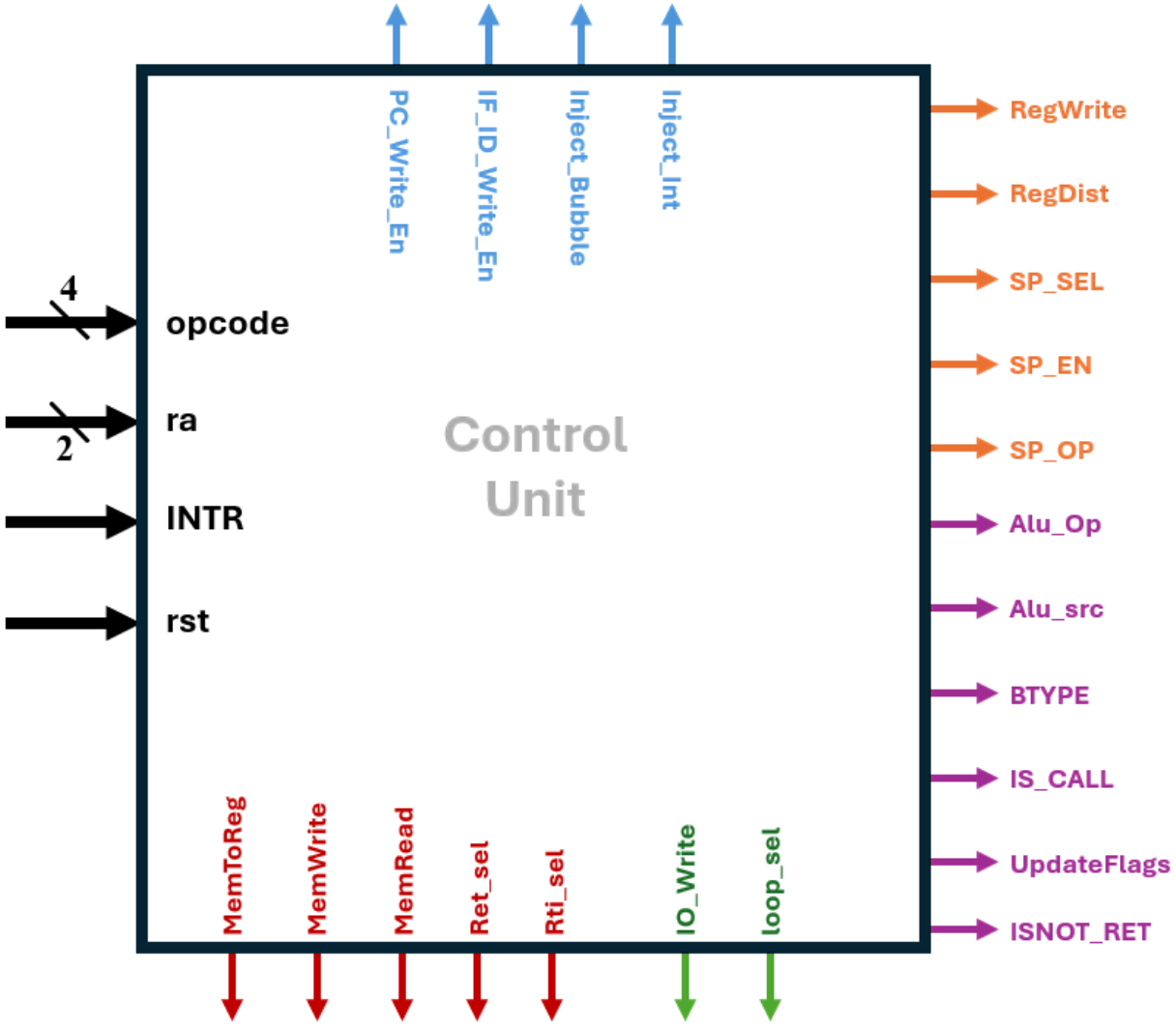
This mechanism enables:

- Flag preservation across instructions
- Clean separation between ALU logic and flag register logic
- Simplified pipeline flag forwarding

2. Control Unit

The Control Unit is the central decision-making module of the processor. It decodes the instruction opcode and auxiliary fields and generates all pipeline control signals required by the datapath.

Control signals are generated in the Instruction Decode (ID) stage. Signals are pipelined forward to EX, MEM, and WB stages. The Control Unit also manages: Instruction fetch sequencing, Immediate fetch cycles, Interrupt injection, Stack-based CALL/RET mechanisms, Branch classification.



A. Input Signals

Signal	Width	Description
clk	1	System clock
rst	1	Active-low synchronous reset
INTR	1	Interrupt request signal
opcode	4	Primary instruction opcode
ra	2	Sub-opcode / instruction modifier

B. Output Signals

- Fetch Stage Control

Signal	Description
PC_Write_En	Enables PC update
IF_ID_Write_En	Enables IF/ID pipeline register
Inject_Bubble	Injects a NOP into pipeline
Inject_Int	Forces PC to interrupt vector

- Decode / Register Stage

Signal	Description
RegWrite	Enables register writeback
RegDist	Selects destination register
SP_SEL	Select stack pointer as address
SP_EN	Enables stack pointer update
SP_OP	Stack pointer direction (inc/dec)

- Execute Stage

Signal	Description
Alu_Op	Direct ALU operation selector
Alu_src	ALU operand source selection
BTYPE	Branch type encoding
IS_CALL	Indicates CALL-like behavior
UpdateFlags	Enables flag register update
ISNOT_RET	Distinguishes RET/RTI

- Memory Stage

Signal	Description
MemRead	Enables memory read
MemWrite	Enables memory write
MemToReg	Write-back source selection
Ret_sel	Selects return address
Rti_sel	Selects RTI address
loop_sel	Enables loop branch handling

- Write-Back and I/O

Signal	Description
IO_Write	Enables output port write
loop_sel	Loop control selection

C. Internal Control FSM

The Control Unit contains a finite state machine responsible for instruction fetch sequencing and interrupting handling.

- **FSM States**

State	Function
Reset	Pipeline flush and initialization
FETCH	Normal instruction fetch
FETCH_IMM	Immediate operand fetch

FSM Behavior:

1. RESET State (2'b00)

• Trigger: Entered immediately when the active-low RST signal is asserted.

• Purpose: Initializes the pipeline and prevents "garbage" execution on startup.

Action:

– Inject Bubble = 1: Inserts a NOP (No-Operation) into the pipeline to clear it.

– PC_Write_En = 1: Allows the Program Counter to start counting.

Next State: Automatically transitions to FETCH.

2. FETCH State (2'b01)

• Trigger: This is the default "steady state" where normal single-cycle instructions (such as ADD, SUB, MOV) are processed.

• Purpose: Decodes the current opcode.

Transition Logic:

– Normal Instruction: If the opcode is not 12, the controller stays in FETCH.

– Immediate Instruction (Opcode = 12):

If the opcode is LDM, LDD, or STD (opcode value 4'b1100 or decimal 12), the controller recognizes that the next memory word is data, not an instruction, and transitions to FETCH_IMM.

3. FETCH_IMM State (2'b10)

• Trigger: Entered immediately after detecting an immediate instruction (Opcode 12).

• Purpose: Pauses the pipeline to consume the immediate value from memory without treating it as a new instruction.

Action:

– IF_ID_Write_En = 0: Stalls the Fetch/Decode register, keeping the LDM/LDD instruction in the Decode stage while the immediate data is fetched.

– Inject Bubble = 1: Sends a bubble to the Execute stage so the data byte is not mistakenly executed as an instruction.

Next State: Automatically returns to FETCH to process the next real instruction.

State	State Code	Condition	Action
Reset	00	!rst	Clears pipeline, goes to Fetch.
Fetch	01	Opcode != 12	Normal execution (Stay in Fetch).
Fetch	01	Opcode == 12	Immediate op detected (Go to Fetch_Imm).
Fetch_Imm	10	Always	Stalls pipeline to read data, returns to Fetch.

D. Instruction Decoding and Control Generation

- Data Transfer Instructions

Opcode	Instruction	Key Control Signals
0001	MOV	RegWrite=1, Alu_Op=MOV
1100	LDM	Alu_src=IMM, RegWrite=1
1100	LDD	MemRead=1, MemToReg=MEM
1100	STD	MemWrite=1

- Arithmetic and Logical Instructions

Opcode	Operation	Flags
0010	ADD	Z, N, C, V
0011	SUB	Z, N, C, V
0100	AND	Z, N
0101	OR	Z, N
1000	NOT / NEG / INC / DEC	Z, N, C, V (as applicable)

- Stack and I/O Instructions

Opcode	Subcode	Behavior
0111	PUSH	SP−−, MemWrite
0111	POP	SP++, MemRead, RegWrite
0111	OUT	IO_Write
0111	IN	RegWrite, MemToReg=IO

- Branch and Control Flow Instructions

Opcode	Type	Branch Encoding
1001	Conditional Branch	Z, N, C, V
1010	LOOP	Loop counter decrement
1011	JMP	Unconditional
1011	CALL	Stack push + PC save
1011	RET / RTI	Stack pop + PC restore

E. Interrupt Handling

Interrupts are handled as implicit CALL instructions:

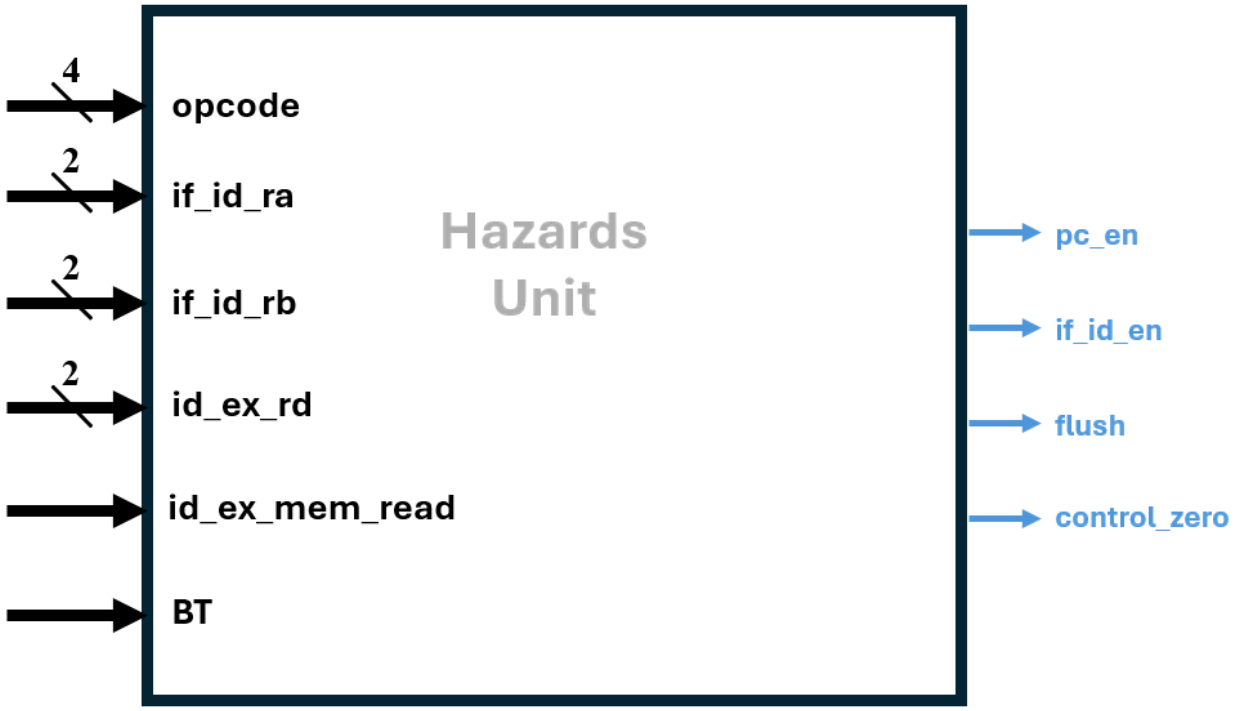
- Current PC is pushed onto the stack
- Control is redirected to an interrupt vector
- Pipeline bubbles are injected to maintain correctness

This design ensures precise interrupts in a pipelined environment.

3. Hazard Unit

The Hazard Detection Unit (HU) is a combinational control module responsible for preserving correct program execution in the presence of pipeline hazards. It operates primarily at the boundary between the Instruction Decode (ID) and Execute (EX) stages, monitoring instruction dependencies and control-flow changes.

The unit detects: Data hazards (specifically load-use hazards) and Control hazards arising from taken branches. Upon detecting a hazard, the HU either stalls the pipeline, injects a bubble, or flushes instructions, depending on the hazard type.



A. Input Signals

Signal	Width	Description
if_id_ra	2	Source register A of instruction in ID stage
if_id_rb	2	Source register B of instruction in ID stage
id_ex_rd	2	Destination register of instruction in EX stage
id_ex_mem_read	1	Indicates EX-stage instruction is a load
opcode	4	Opcode of instruction in ID stage
BT	1	Branch taken signal

B. Output Signals

Signal	Description
pc_en	Enables or stalls the Program Counter
if_id_en	Enables or stalls the IF/ID pipeline register
flush	Flushes the IF/ID pipeline register
control_zero	Forces ID/EX control signals to zero (bubble insertion)

C. Truth Table (Hazard-Level)

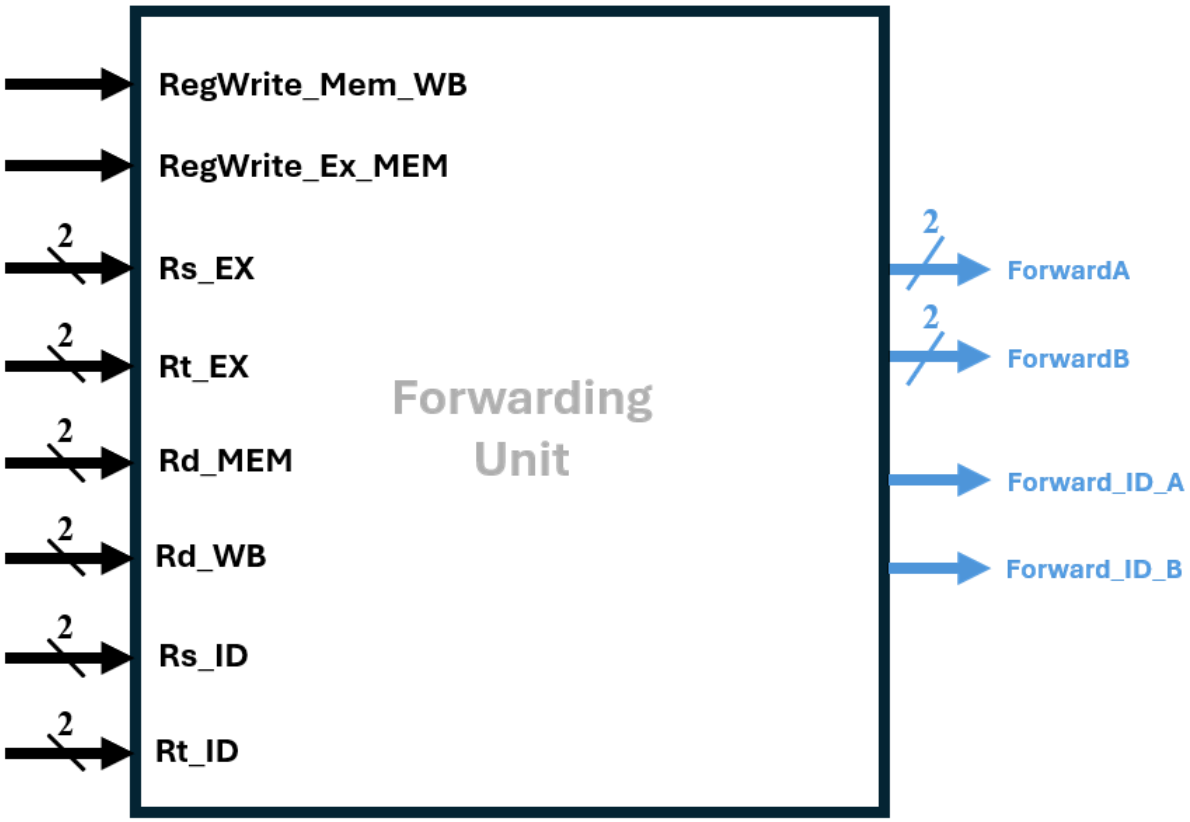
Condition	pc_en	if_id_en	flush	control_zero
No hazard	1	1	0	0
Load-use hazard	0	0	0	1
Branch taken	1	1	1	0
Load-use + branch	0	0	1	1

4. Forwarding Unit

The Forwarding Unit is a hazard resolution block whose main objective is to eliminate unnecessary pipeline stalls caused by Read-After-Write (RAW) data hazards. Our updated design correctly handles hazards across two pipeline regions:

1. EX stage hazards (classic ALU-to-ALU forwarding).
2. ID stage hazards (decode-time operand correctness).

This dual scope forwarding design is a strong architectural improvement and reflects a mature pipelined CPU design.



A. Input Signals

- Control Inputs

Signal	Description
RegWrite_Ex_MEM	Indicates that the instruction in EX/MEM will write to a register
RegWrite_Mem_WB	Indicates that the instruction in MEM/WB will write to a register

- Register Address Inputs

Signal	Width	Description
Rs_EX	2	Source register A address in EX stage
Rt_EX	2	Source register B address in EX stage
Rd_MEM	2	Destination register address in MEM stage
Rd_WB	2	Destination register address in WB stage
Rs_ID	2	Source register A during decode
Rt_ID	2	Source register B during decode

B. Output Signals

Signal	Width	Description
ForwardA	2	Select signal for ALU operand A multiplexer
ForwardB	2	Select signal for ALU operand B multiplexer
Forward_ID_A	1	Select signal for ID/EX Rs multiplexer
Forward_ID_B	1	Select signal for ID/EX Rt multiplexer

C. Forwarding Strategy

The Forwarding Unit (FU) resolves Read-After-Write (RAW) data hazards by dynamically selecting the most recent value of a register operand before it is written back to the register file.
This is achieved by comparing the source registers of the instruction in the EX-stage with the destination registers of instructions currently in the MEM and WB stages.

Forwarding Priority

To guarantee correctness, forwarding follows a strict priority order:

1. **EX/MEM → EX forwarding (highest priority)**
   The result from the immediately preceding instruction is the most recent and must be selected first.
2. **MEM/WB → EX forwarding (lower priority)**
   Used only when EX/MEM forwarding is not applicable.
3. **ID/EX register value (default)**
   Selected when no data hazard is detected.
4. **ID/WB → ID forwarding (lower priority)**

   Used when neither EX/MEM nor MEM/WB forwarding is applicable

This priority scheme ensures that the ALU always operates on the latest available data, preventing incorrect computations due to stale register values.

D. Forwarding Logic Description

- Forwarding for ALU Input A (Rs_EX)

Condition	Action
RegWrite_EX/MEM = 1 and (Rd_EX/MEM ≠ 0) and (Rd_EX/MEM = Rs_EX)	ForwardA = 10 (EX/MEM → EX)
RegWrite_MEM/WB = 1 and (Rd_MEM/WB ≠ 0) and (Rd_MEM/WB = Rs_EX) and no EX/MEM match	ForwardA = 01 (MEM/WB → EX)
Otherwise	ForwardA = 00 (ID/EX register value)

- Forwarding for ALU Input B (Rt_EX)

Condition	Action
RegWrite_EX/MEM = 1 and (Rd_EX/MEM ≠ 0) and (Rd_EX/MEM = Rt_EX)	ForwardB = 10 (EX/MEM → EX)
RegWrite_MEM/WB = 1 and (Rd_MEM/WB ≠ 0) and (Rd_MEM/WB = Rt_EX) and no EX/MEM match	ForwardB = 01 (MEM/WB → EX)
Otherwise	ForwardB = 00 (ID/EX register value)

E. Forwarding Select Encoding

Forward Code	Data Source
00	ID/EX register output
01	MEM/WB stage result
10	EX/MEM stage ALU output

5. Branch Unit

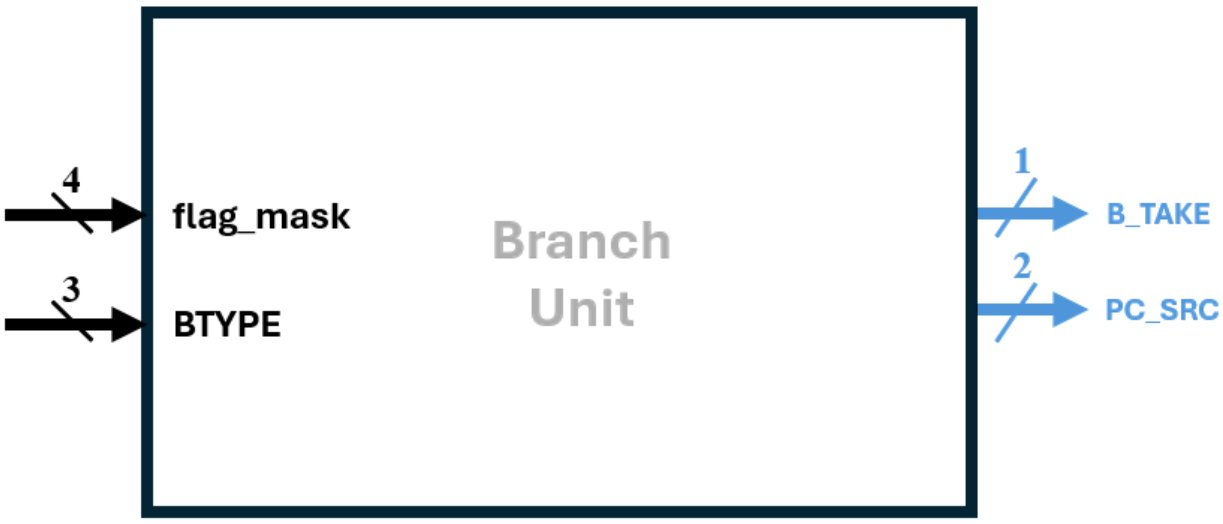
The Branch Unit is a combinational control-flow resolution block responsible for determining:

1. Whether a branch is taken
2. Which source updates the Program Counter (PC)

It evaluates branch conditions using the processor status flags generated by the ALU and the branch classification (BTYPE) provided by the Control Unit.

In the pipeline, the Branch Unit operates in the Execute (EX) stage, where branch conditions become resolvable, and its outputs are used by:

- The PC selection multiplexer
- The Hazard Detection Unit (via the branch-taken signal)



A. Input Signals

Signal	Width	Description
flag_mask	4	Current condition flags [Z, N, C, V]
BTYPE	3	Branch type encoding from Control Unit

B. Output Signals

Signal	Width	Description
B_TAKE	1	Indicates whether the branch is taken
PC_SRC	2	Selects the source used to update the PC

C. Branch Type Encoding

Encoding	Branch Type	Description
000	BR_NONE	No branch
001	BR_JZ	Jump if Zero
010	BR_JN	Jump if Negative
011	BR_JC	Jump if Carry
100	BR_JV	Jump if Overflow
101	BR_LOOP	Loop branch
110	BR_JMP	Unconditional jump / call
111	BR_RET	Return / RTI

D. PC Source Encoding

Code	PC Source
00	Normal PC increment
01	Forwarded branch target
10	Data operand (stack-based return)

E. Branch Decision Logic

- Conditional Branches

Branch	Condition	Flag Used
JZ	Z == 1	Zero
JN	N == 1	Negative
JC	C == 1	Carry
JV	V == 1	Overflow

- Loop Branch

Branch	Condition
LOOP	Z == 0

- Unconditional Control Flow

Branch	Behavior
JMP / CALL	Always taken, PC from forward path
RET / RTI	Always taken, PC loaded from stack

F. Truth Table (Branch-Level)

BTYPE	Condition	B_TAKE	PC_SRC
NONE	—	0	NORM
JZ	Z=1	1	FW
JN	N=1	1	FW
JC	C=1	1	FW
JV	V=1	1	FW
LOOP	Z=0	1	FW
JMP	—	1	FW

RET	—	1	DataB

6. Program Counter (PC)

The Program Counter (PC) is a sequential state-holding element responsible for tracking the address of the current instruction in the pipeline. It serves as the primary source for instruction fetch in the Instruction Fetch (IF) stage of the processor. Its value can be updated every cycle or stalled under pipeline hazard conditions, enabling precise control of instruction sequencing.

The PC interacts with:

- The IF stage memory interface to fetch instructions
- The Hazard Detection Unit to support stalls (pc_write)
- The Branch Unit or Control Unit for jumps, calls, and returns

A. Input Signals

Signal	Width	Description
rst	1	Active-low Synchronous reset; initializes the PC to pc_next
pc_write	1	Write enable; when low, the PC holds its value (pipeline stall)
pc_next	8	Next instruction address to load into the PC

B. Output Signals

Signal	Width	Description
pc_current	8	Current PC value, representing the address of the instruction being fetched

C. Operational Behavior

1. Synchronous Reset (Active Low):

- Unlike standard counters that reset to 0, this module loads the value from pc_next when rst is 0. This supports our Reset Vector design, allowing the PC to latch a specific start address (like 0x00) provided by the Next-PC logic during the reset cycle.

2. Pipeline Stalling (Write Enable):

- The PC only updates when the control signal pc write is High (1).
- If pc_write is Low (0), the PC ignores the clock and holds its current value, effectively freezing the fetch stage (Stall).

3. Normal Update:

- On every rising clock edge (while not stalled or resetting), it updates pc current with the new address provided on pc next.

7. Memory (Von Neuuman)

The Memory module implements dual port Von Neumann memory for the pipelined processor, supporting both instruction fetch and data memory access simultaneously. It provides:

1. **Port A** – Instruction Fetch (IF stage)
2. **Port B** – Data Memory access (MEM stage)

A. Input Signals

Signal	Width	Description
rst	1	Active-low asynchronous reset; initializes memory outputs
addr_a	8	Address for instruction fetch (Port A)
we_b	1	Write enable for data memory (Port B)
addr_b	8	Address for data memory access (Port B)
write_data_b	8	Data to write to data memory (Port B)

The memory is logically divided into two regions: addresses 0 to 127 are reserved for instruction memory and are not reset, while addresses 128 to 255 are used for data and stack memory and are cleared on reset.

B. Output Signals

Signal	Width	Description
instr_out	8	Instruction fetched from memory at addr_a
data_out_b	8	Data read from memory at addr_b

8. Register File

The Register File (RF) provides storage and rapid access for the processor's general-purpose registers. It is used in the ID stage to supply operands to the ALU and other functional units and receives write-back data from the MEM/WB stage. It also implements the Stack Pointer (SP) in a dedicated register, supporting stack operations.

The module supports:

- **Asynchronous read** for immediate operand access
- **Synchronous writing** on the rising edge of the clock
- **Stack pointer operations** (increment/decrement) for PUSH/POP instructions

A. Input Signals

Signal	Width	Description
rst	1	Active-low reset; clears registers and initializes SP
wenabel	1	General write enable for rd register
SP_EN	1	Stack Pointer enable; activates SP increment/decrement
SP_OP	1	SP operation selector; 1 = increment, 0 = decrement
ra	2	Address for read port A
rb	2	Address for read port B
rd	2	Destination register index for writing data
write_data	8	Data to write into R[rd]

B. Output Signals

Signal	Width	Description
ra_date	8	Current value of register R[ra] (asynchronous read)
rb_date	8	Current value of register R[rb] (asynchronous read)

C. Truth Table (Behavior-Level)

rst	W enabel	SP_EN	SP_OP	Operation
0	X	X	X	Reset R0-R2 = 0, SP = 255
1	0	0	X	Hold all registers
1	1	0	X	Write write_data → R[rd]
1	X	1	0	SP decrement → R [3] = R [3] -1
1	X	1	1	SP increment → R [3] = R [3] +1

9. IF/ID Pipeline Register

The IF/ID pipeline register sits between the Instruction Fetch (IF) stage and the Instruction Decode (ID) stage in a pipelined processor. Its primary purposes are:

1. Latch the fetched instruction from memory to pass to the decoder in the next stage.
2. Propagate PC+1 to support sequential instruction execution and branch calculations.
3. Latch immediate values or instruction pointer for use in ID or subsequent stages.
4. Support pipeline control such as stalling (hazard handling) and flushing (branch taken).

A. Input Signals

Signal	Width	Description
Clk	1	System clock; latches data on rising edge
Rst	1	Active-low asynchronous reset
IF_ID_EN	1	Enable signal for register update (stall when low)
Flush	1	Flush signal to clear register (branch taken)
PC_Plus_1_In	8	PC + 1 value from IF stage
Instruction_In	8	Instruction fetched from memory
Immby	8	Immediate byte from instruction fetch (if any)
IP	8	Instruction pointer value for tracking

B. Output Signals

Signal	Width	Description
`PC_Plus_1_Out`	8	Latched PC+1 value for use in ID stage
`Instruction_Out`	8	Latched instruction to decode stage
`immbyout`	8	Latched immediate byte to ID stage

ID/EX Pipeline Register

The ID/EX pipeline register serves as a synchronization stage between the Instruction Decode (ID) stage and the Execute (EX) stage in a pipelined processor. Its main purposes are:

1. Latching control signals generated in the ID stage for use in the EX-stage
2. Propagating operand data from the register file or immediate values to the EX-stage
3. Supporting pipeline hazard management via flush and inject bubble signals

This module ensures that each pipeline stage receives stable inputs every clock cycle while enabling hazard handling and instruction injection control.

A. Input Signals

Signal	Width	Description
rst	1	Active-low reset; clears all outputs
flush	1	Flush signal from hazard unit; clears outputs to insert NOP
inject_bubble	1	Inserts bubble (NOP) by disabling ALU operation
pc_plus1	8	PC incremented by 1 (for CALL/RET address storage)
IP	8	Instruction pointer or immediate program counter data
imm	8	Immediate operand from instruction
BType	3	Branch type control from ID stage
MemToReg	2	Controls data source for write-back stage
RegWrite	1	Enables writing to the register file
MemWrite	1	Enables writing to data memory
MemRead	1	Enables reading from data memory
UpdateFlags	1	Enables ALU flag update
RegDistidx	2	Selects destination register for write-back
ALU_src	2	Selects ALU operand source
ALU_op	4	ALU operation code
IO_Write	1	Enables writing to output ports

isCall	1	Marks CALL instruction for PC selection
loop_sel	1	Loop control signal
ra_val_in	8	Operand A value from register file
rb_val_in	8	Operand B value from register file
ra	2	Source register A address
rb	2	Source register B address

B. Output Signals

Signal	Width	Description
BType_out	3	Branch type propagated to EX stage
MemToReg_out	2	Memory-to-register control signal
RegWrite_out	1	Register write enable for EX stage
MemWrite_out	1	Memory write enable for EX stage
MemRead_out	1	Memory read enable for EX stage
UpdateFlags_out	1	ALU flag update enable
RegDistidx_out	2	Destination register index
ALU_src_out	2	ALU operand source selection
ALU_op_out	4	ALU operation code
IO_Write_out	1	Output port write enable
isCall_out	1	CALL instruction marker
loop_sel_out	1	Loop control propagated to EX
ra_val_out	8	Operand A value latched to EX stage
rb_val_out	8	Operand B value latched to EX stage
ra_out	2	Source register A address
rb_out	2	Source register B address
pc_plus1_out	8	PC + 1 latched for EX stage use
IP_out	8	Instruction pointer latched

imm_out	8	Immediate operand latched

10. EX/MEM Pipeline Register

The EX/MEM pipeline register acts as a synchronization stage between the Execute (EX) stage and the Memory (MEM) stage in a pipelined processor. Its key responsibilities are:

1. Latching ALU results and operand data from the EX-stage
2. Propagating control signals to the MEM stage for memory access and write-back
3. Maintaining instruction flow integrity across the pipeline

This module ensures that the MEM stage receives stable inputs every clock cycle, avoiding glitches and misaligned operations.

A. Input Signals

Signal	Width	Description
rst	1	Active-low reset; clears all outputs
pc_plus1	8	PC + 1 value propagated for CALL or branch operations
Rd2	8	Second operand value from ID/EX (typically for memory writes)
IO_Write	1	Enables writing to output ports
RegDistidx	2	Destination register index for write-back
ALU_res	8	ALU result from EX stage
FW_value	8	Forwarded value (for data hazard resolution)
MemWrite	1	Enables memory write operation
MemToReg	2	Memory-to-register selection signal
RegWrite	1	Enables writing to the register file
IP	8	Instruction pointer from EX stage
isCall	1	Indicates a CALL instruction (PC save)

B. Output Signals

Signal	Width	Description
pc_plus1_out	8	Latched PC + 1 for MEM stage usage
Rd2_out	8	Latched second operand value for memory write
IO_Write_out	1	Output port write enable latched to MEM stage
RegDistidx_out	2	Latched destination register index
ALU_res_out	8	Latched ALU result for memory operations and write-back
FW_value_out	8	Latched forwarded operand for MEM stage
MemWrite_out	1	Latched memory write enable
MemToReg_out	2	Latched memory-to-register selection
RegWrite_out	1	Latched register write enable
IP_out	8	Latched instruction pointer for MEM stage use
isCall_out	1	Latched CALL instruction marker

11. MEM/WB Pipeline Register

The MEM/WB pipeline register serves as the final pipeline stage connecting the Memory (MEM) stage to the Write-Back (WB) stage. Its key responsibilities are:

1. Latching results from MEM stage to ensure stable inputs for the WB stage
2. Propagating control signals such as register write enables and memory-to-register selection
3. Maintaining the correct register destination information for write-back

This stage ensures that the WB stage receives **synchronized, hazard-free signals** for writing results back to the register file.

A. Input Signals

Signal	Width	Description
clk	1	System clock; register updates on rising edge
rst	1	Active-low reset; clears all outputs
pc_plus1	8	PC + 1 value propagated for CALL/branch handling
RegDistidx	2	Destination register index for write-back
Rd2	8	Operand value (used for memory write or forwarding)
ALU_res	8	ALU result from EX/MEM stage
data_B	8	Data read from memory (for load instructions)
MemToReg	2	Selects data source for register write-back (ALU, MEM, etc.)
RegWrite	1	Enables writing to the register file
IP	8	Instruction pointer from MEM stage (optional for CALL/RET)

B. Output Signals

Signal	Width	Description
pc_plus1_out	8	Latched PC + 1 for WB stage
RegDistidx_out	2	Latched destination register index
Rd2_out	8	Latched operand data for WB stage
ALU_res_out	8	Latched ALU result for WB stage

data_B_out	8	Latched memory read data for load instructions
MemToReg_out	2	Latched memory-to-register selection
RegWrite_out	1	Latched register write enable
IP_out	8	Latched instruction pointer for WB stage use

12. CCR (Condition Code Register)

The CCR holds the processor status flags that reflect the outcome of ALU operations and special control events such as interrupts or return-from-interrupt (RTI). These flags are critical for:

1. Branch decisions (conditional jumps rely on Z, N, C, V flags).
2. Status reporting to other stages like the Branch Unit.
3. Interrupt handling (preserve and restore flag states).

The CCR acts as a 4-bit register with selective updating based on a mask and special operations for interrupts.

A. Input Signals

Signal	Width	Description
Z	1	Zero flag from ALU
N	1	Negative flag from ALU
C	1	Carry flag from ALU
V	1	Overflow flag from ALU
flag_en	1	Enable updating of flags
intr	1	Interrupt signal; triggers saving upper 4 bits
rti	1	Return from interrupt; restores upper 4 bits
flag_mask	4	Bitmask indicating which flags to update (Z, N, C, V)

B. Output Signals

Signal	Width	Description
CCR	4	Current 4-bit condition code value [V C N Z]

C. Truth Table (Flag Updates)

intr	rti	flag_en	flag_mask	Effect on CCR
0	0	0	xxxx	No change
0	0	1	0000	No change
0	0	1	0001	Update Z
0	0	1	0010	Update N
0	0	1	0100	Update C
0	0	1	1000	Update V
1	0	x	xxxx	Save flags to upper 4 bits, clear lower 4
0	1	x	xxxx	Restore flags from upper 4 bits

IV. Synthesizing the processor (Bonus)

1. (Getting frequency and FPGA utilization) FPGA Synthesis & Implementation

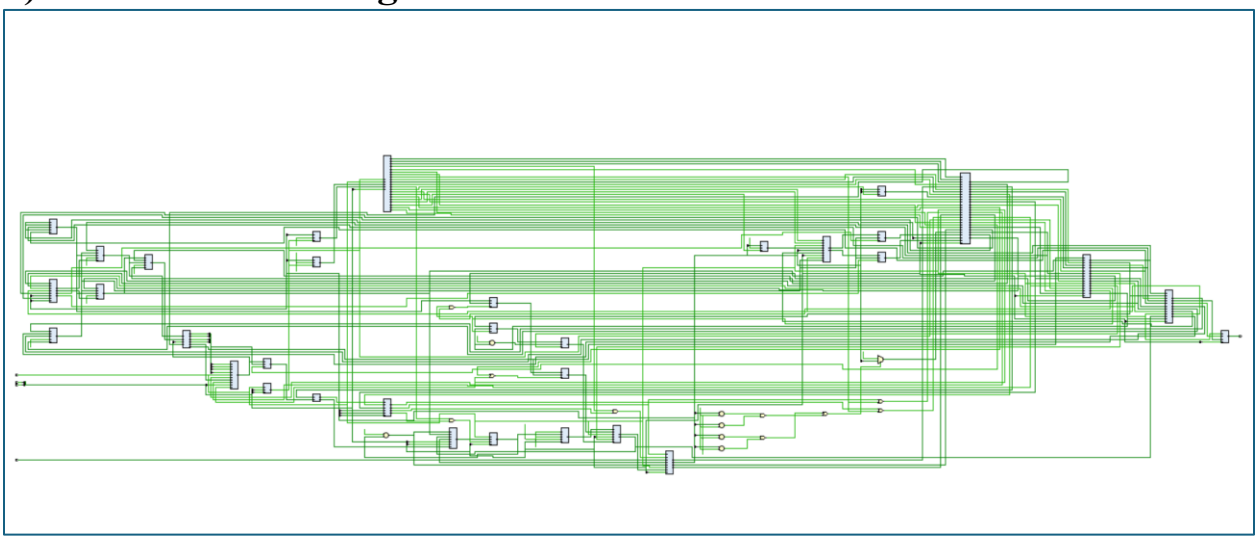
In this part, Xilinx Vivado was used to synthesize the processor on an FPGA and get its frequency and utilization. The FPGA used was AMD's XC7A35T-1CPG236C

Constraint File:

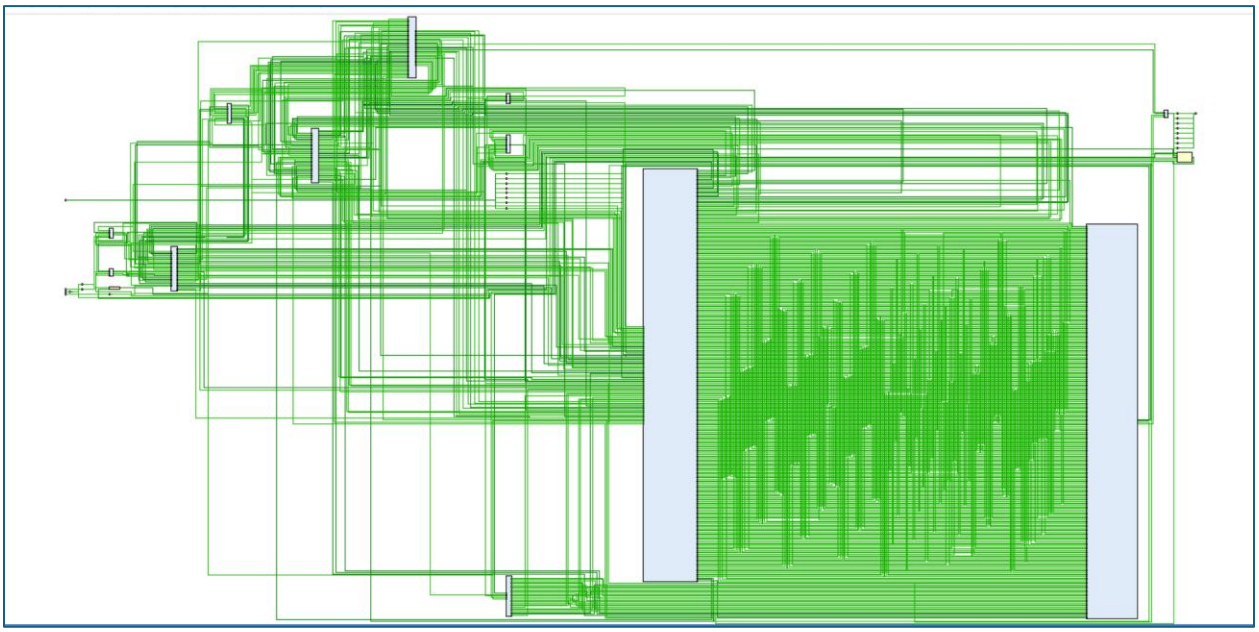
```
## Clock signal
set_property -dict {PACKAGE_PIN W5 IOSTANDARD LVCMOS33} [get_ports clk]
create_clock -period 20 -name sys_clk_pin -waveform {0.000 10} -add [get_ports clk]

## Switches (Mapped to I_Port [7:0])
set_property -dict {PACKAGE_PIN V17 IOSTANDARD LVCMOS33} [get_ports {I_Port[0]}]
set_property -dict {PACKAGE_PIN V16 IOSTANDARD LVCMOS33} [get_ports {I_Port[1]}]
set_property -dict {PACKAGE_PIN W16 IOSTANDARD LVCMOS33} [get_ports {I_Port[2]}]
set_property -dict {PACKAGE_PIN W17 IOSTANDARD LVCMOS33} [get_ports {I_Port[3]}]
set_property -dict {PACKAGE_PIN W15 IOSTANDARD LVCMOS33} [get_ports {I_Port[4]}]
set_property -dict {PACKAGE_PIN V15 IOSTANDARD LVCMOS33} [get_ports {I_Port[5]}]
set_property -dict {PACKAGE_PIN W14 IOSTANDARD LVCMOS33} [get_ports {I_Port[6]}]
set_property -dict {PACKAGE_PIN W13 IOSTANDARD LVCMOS33} [get_ports {I_Port[7]}]

## Switch for Interrupt Signal
set_property -dict {PACKAGE_PIN V2 IOSTANDARD LVCMOS33} [get_ports int_sig]

## LEDs (Mapped to O_Port [7:0])
set_property -dict {PACKAGE_PIN U16 IOSTANDARD LVCMOS33} [get_ports {O_Port[0]}]
set_property -dict {PACKAGE_PIN E19 IOSTANDARD LVCMOS33} [get_ports {O_Port[1]}]
set_property -dict {PACKAGE_PIN U19 IOSTANDARD LVCMOS33} [get_ports {O_Port[2]}]
set_property -dict {PACKAGE_PIN V19 IOSTANDARD LVCMOS33} [get_ports {O_Port[3]}]
set_property -dict {PACKAGE_PIN W18 IOSTANDARD LVCMOS33} [get_ports {O_Port[4]}]
set_property -dict {PACKAGE_PIN U15 IOSTANDARD LVCMOS33} [get_ports {O_Port[5]}]
set_property -dict {PACKAGE_PIN U14 IOSTANDARD LVCMOS33} [get_ports {O_Port[6]}]
set_property -dict {PACKAGE_PIN V14 IOSTANDARD LVCMOS33} [get_ports {O_Port[7]}]

## Buttons (Mapped to rstn)
# Note: Using the Center Button (U18) for Reset
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports rstn]

## Configuration options
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]
```

- **A frequency of 50MHz (Period of 20ns) was obtained.**

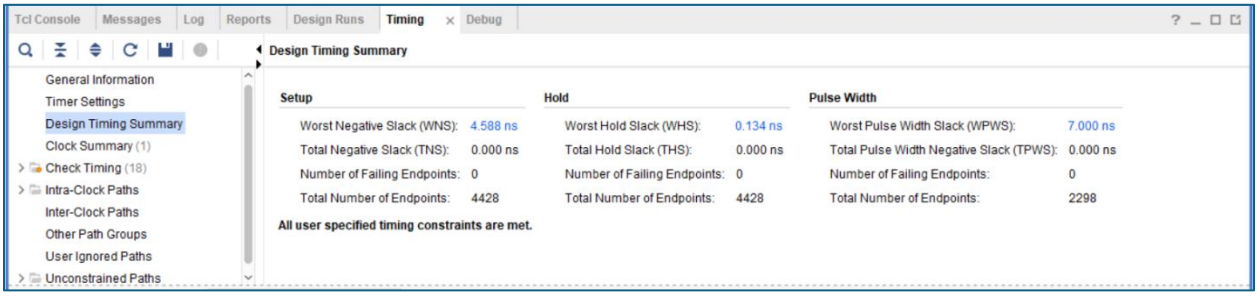
1) RTL Schematic Figure:



2) Synthesis Schematic Figure:



3) Synthesis Timing Report:

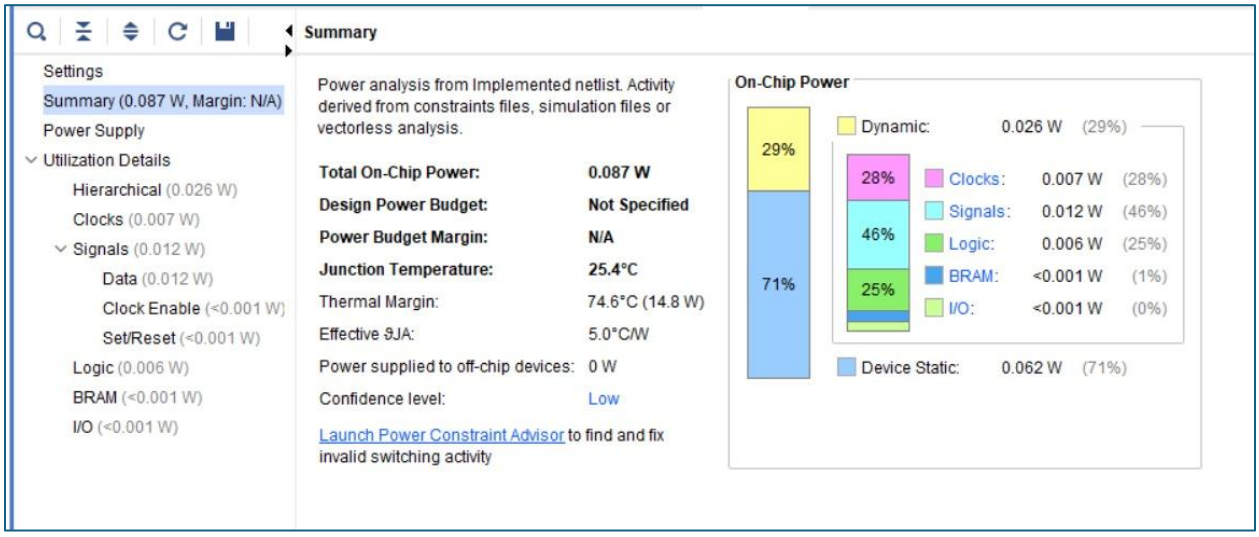


From the figure above, the design achieves a setup slack of 4.588 ns and a hold slack of 0.134 ns, indicating robust timing closure with comfortable margins for both setup and hold requirements.

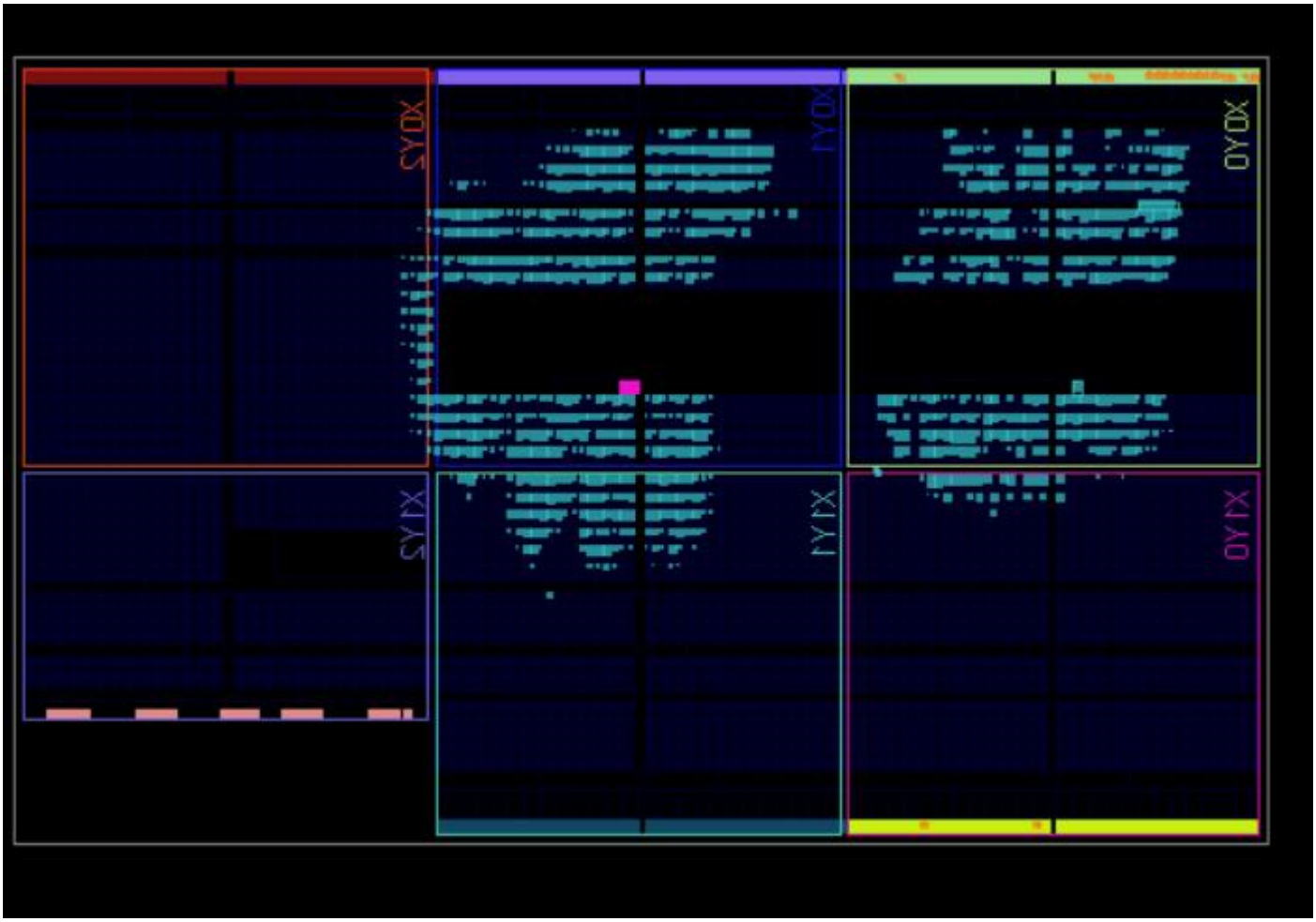
4) Synthesis Utilization Report:

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (106)	BUFGCTRL (32)
CPU_WrapperV3	1923	2297	342	168	19	1
alu_inst (ALU)	4	0	0	0	0	0
ccr_inst (CCR)	6	8	0	0	0	0
ctrl_inst (Control_unit)	2	2	0	0	0	0
dbg_hub (dbg_hub_CV)	0	0	0	0	0	0
ex_mem_reg_inst (EX...	60	53	0	0	0	0
id_ex_reg_inst (id_ex_...	447	67	3	0	0	0
if_id_reg_inst (IF_ID_...	116	32	3	0	0	0
mem_inst (memory)	1201	2048	336	168	0	0
mem_wb_reg_inst (M...	41	38	0	0	0	0
out_reg (OUT)	0	8	0	0	0	0
PC (Pc)	26	8	0	0	0	0
regfile_inst (Register_f...	20	32	0	0	0	0
u_ila_0 (u_ila_0_CV)	0	0	0	0	0	0
u_interrupt_reg (interru...	2	1	0	0	0	0

5) Power Report:



6) Implementation Device Figure:



7) Implementation Timing Report:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.104 ns	Worst Hold Slack (WHS):	0.045 ns	Worst Pulse Width Slack (WPWS):	8.750 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:	8335	Total Number of Endpoints:	8319	Total Number of Endpoints:	4503

All user specified timing constraints are met.

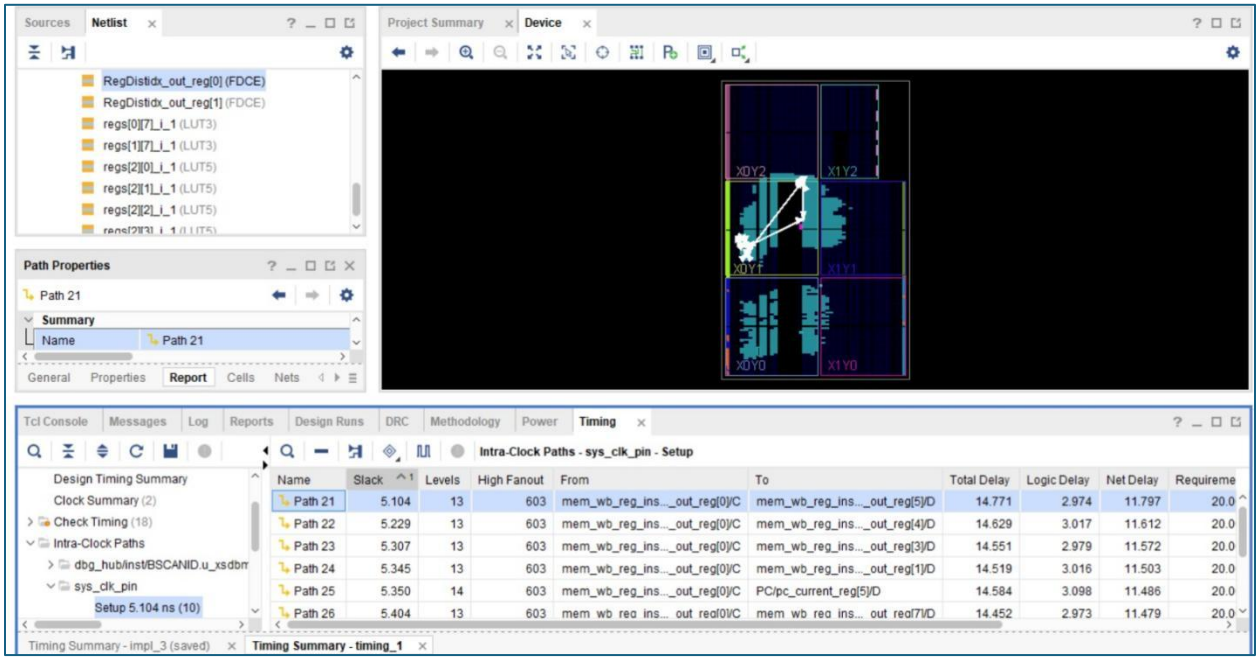
8) Implementation Utilization Report:

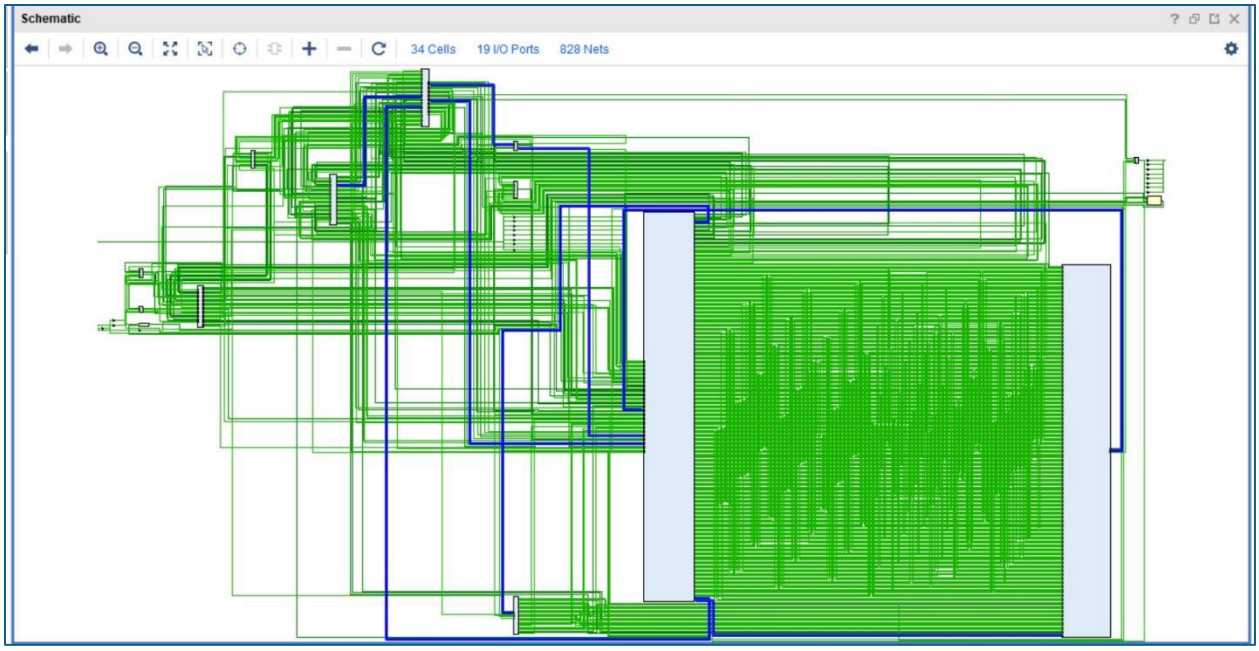
The successful generation of the bitstream confirms that the design is suitable for FPGA implementation. Nevertheless, hardware deployment was not performed due to the lack of access to a physical FPGA device

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)	BSCANE2 (4)
N CPU_WrapperV3	3183	4304	353	168	1715	3068	115	873	1	19	2	1
alu_inst (ALU)	4	0	0	0	5	4	0	0	0	0	0	0
ccr_inst (CCR)	6	8	0	0	4	6	0	6	0	0	0	0
ctrl_inst (Control_unit)	2	2	0	0	3	2	0	0	0	0	0	0
dbg_hub (dbg_hub)	475	727	0	0	234	451	24	315	0	0	1	1
ex_mem_reg_inst (EX...	60	53	0	0	60	60	0	2	0	0	0	0
id_ex_reg_inst (id_ex_...	447	67	3	0	215	447	0	0	0	0	0	0
if_id_reg_inst (IF_ID_...	116	32	3	0	54	116	0	0	0	0	0	0
mem_inst (memory)	1200	2048	336	168	945	1200	0	0	0	0	0	0
mem_wb_reg_inst (M...	39	38	0	0	37	39	0	0	0	0	0	0
out_reg (OUT)	0	8	0	0	2	0	0	0	0	0	0	0
PC (Pc)	26	8	0	0	18	26	0	0	0	0	0	0
regfile_inst (Register_f...	20	32	0	0	14	20	0	4	0	0	0	0
u_ila_0 (u_ila_0)	786	1280	11	0	409	695	91	452	1	0	0	0
u_interrupt_reg (interru...	2	1	0	0	3	2	0	0	0	0	0	0

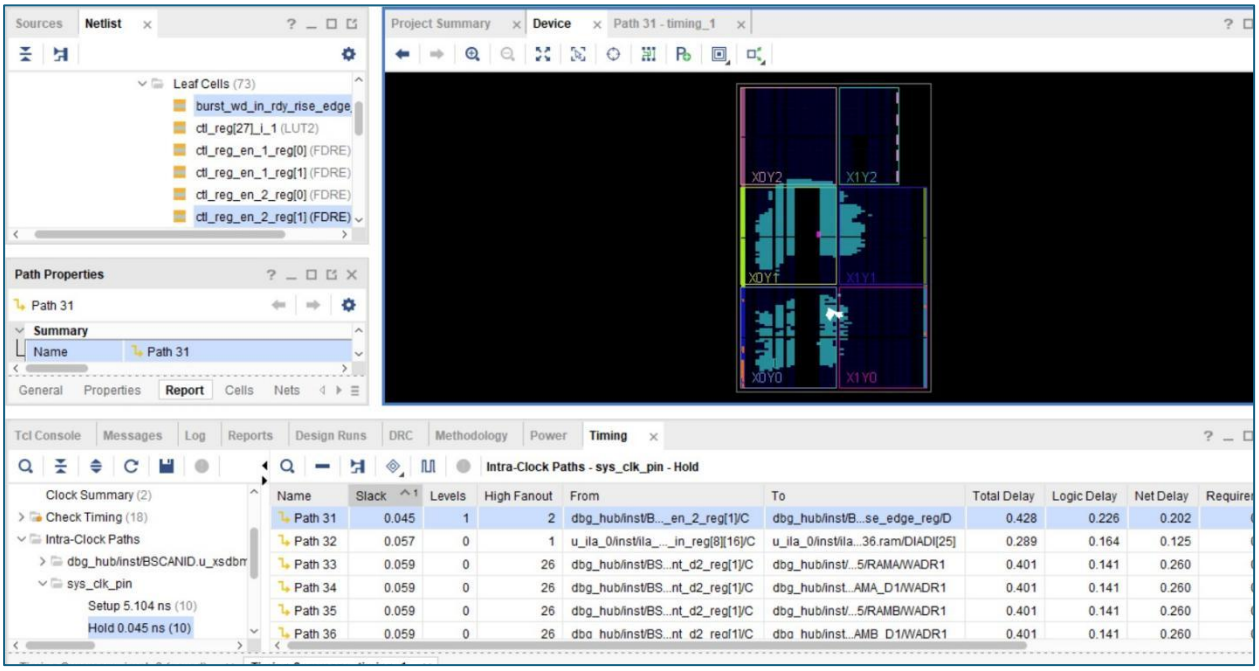
9) Worst Path:

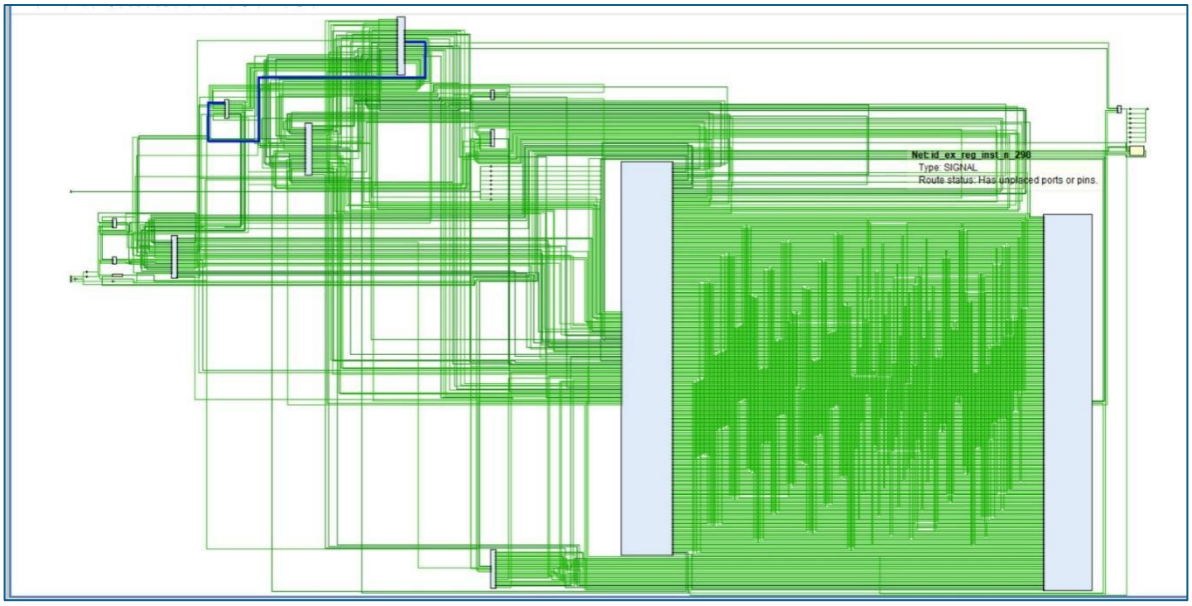
- Setup Time:




Schematic
34 Cells
19 I/O Ports
828 Nets

- Hold Time:


Sources
Netlist
Leaf Cells (73)
burst_wd_in_rdy_rise_edge
ctl_reg[27]_i_1 (LUT2)
ctl_reg_en_1_reg[0] (FDRE)
ctl_reg_en_1_reg[1] (FDRE)
ctl_reg_en_2_reg[0] (FDRE)
ctl_reg_en_2_reg[1] (FDRE)
Path Properties
Path 31
Summary
Name
Path 31
General
Properties
Report
Cells
Nets
Project Summary
Device
Path 31 - timing_1
X0Y2
X1Y2
X0Y1
X1Y1
X0Y0
X1Y0
Tcl Console
Messages
Log
Reports
Design Runs
DRC
Methodology
Power
Timing
Intra-Clock Paths - sys_clk_pin - Hold
Clock Summary (2)
Check Timing (18)
Intra-Clock Paths
dbg_hub/inst/BSCANID.u_xsdbm
sys_clk_pin
Setup 5.104 ns (10)
Hold 0.045 ns (10)
Name
Slack
Levels
High Fanout
From
To
Total Delay
Logic Delay
Net Delay
Path 31 0.045 1 2 dbg_hub/inst/B..._en_2_reg[1]/C dbg_hub/inst/B...se_edge_reg/D 0.428 0.226 0.202
Path 32 0.057 0 1 u_ila_0/inst/ila_..._in_reg[8][16]/C u_ila_0/inst/ila...36.ram/DIADI[25] 0.289 0.164 0.125
Path 33 0.059 0 26 dbg_hub/inst/BS...nt_d2_reg[1]/C dbg_hub/inst/...5/RAMA/WADR1 0.401 0.141 0.260
Path 34 0.059 0 26 dbg_hub/inst/BS...nt_d2_reg[1]/C dbg_hub/inst...AMA_D1/WADR1 0.401 0.141 0.260
Path 35 0.059 0 26 dbg_hub/inst/BS...nt_d2_reg[1]/C dbg_hub/inst/...5/RAMB/WADR1 0.401 0.141 0.260
Path 36 0.059 0 26 dbg_hub/inst/BS...nt_d2_reg[1]/C dbg_hub/inst...AMB_D1/WADR1 0.401 0.141 0.260



2. Test Cases and Results

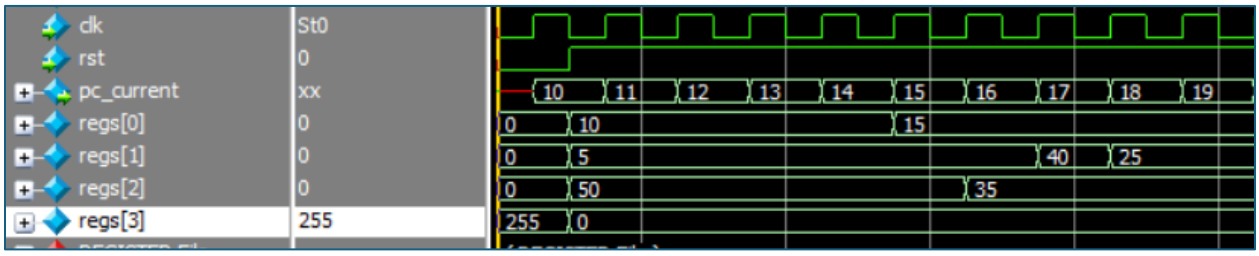
1. Program:

We in this simple program want to check pipeline is working and speed up = n

```
0x10 : ADD R0, R1
0x11 : SUB R2, R0
0x12 : ADD R1, R2
0x13 : SUB R1, R0
```

Prior to program execution, the register file is initialized such that R0 = 10, R1 = 5, R2 = 50, and R3 = 0.

Waveform:



Starting from PC = 0x10, the instruction progresses through the pipeline stages as follows: the Fetch stage occurs from PC = 0x10 to 0x11, the Decode stage from 0x11 to 0x12, the Execute stage from 0x12 to 0x13, the Memory stage from 0x13 to 0x14, and the Write-Back (WB) stage from 0x14 to 0x15.

As a result, after five clock cycles, the value 15 is written into R0. After one additional clock cycle, the value 35 is written into R2, followed by 40 being written into R1 in the next cycle. Finally, after another clock cycle, the value 25 is written into R1.

This timing behavior confirms that the processor operates correctly in a pipelined manner, and that the forwarding unit successfully resolves data hazards without introducing unnecessary pipeline stalls.

1 clk	2 clk	3 clk	4 clk	5 clk	6 clk	7 clk	8 clk
F	D	EX	MEM	WB			
	F	D	EX	MEM	WB		
		F	D	EX	MEM	WB	
			F	D	EX	MEM	WB

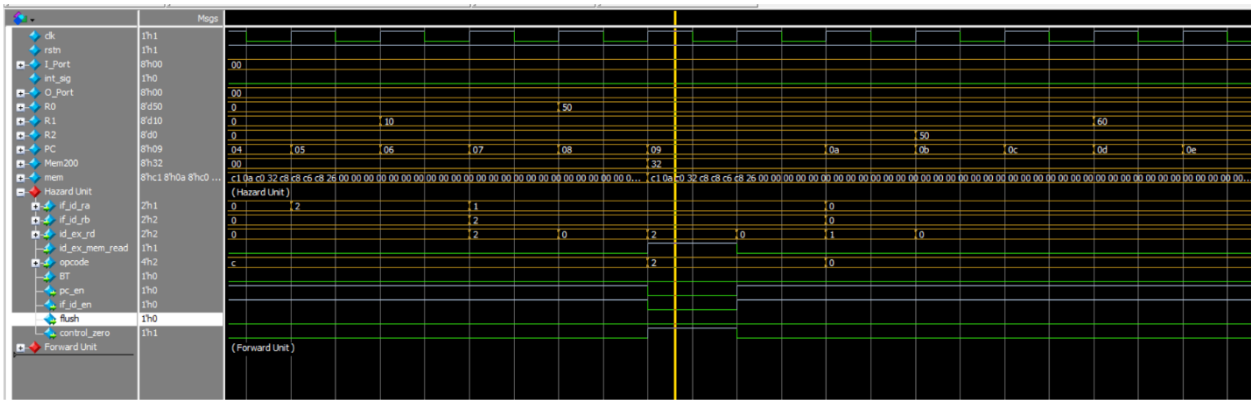
- So $Speed\ up\ = \frac{4 * n * \tau}{4\ \tau}\ =\ n\ (No.of\ Stages\ )\ =\ 5$
- Pipeline Latency = 5 clk cycle.
- CPI = 1 and due to stall in some of instructions CPI = 1 + stall cycles per instruction

2. Program:

```
LDM R1, 10
LDM R0, 50
STD R0, 200
LDD R2, 200
ADD R1, R2
```

There is clearly a raw hazard between instruction 4 & 5 so the hazard unit should initiate when these instructions are in play.

Waveform:



The hazard detection unit correctly identifies a data hazard when the destination register of the previous instruction matches the source register of the current instruction, and it appropriately stalls the pipeline to preserve correct execution.

- The first instruction is completed in 6 cycles.
- The second instruction requires 2 cycles after the sixth cycle and enters the pipeline at PC = 8.
- The third instruction also takes 2 cycles; however, due to asynchronous memory access, it completes at PC = 9.
- The fourth instruction completes in 2 cycles, finishing at PC = 11.
- The fifth instruction requires only 1 cycle, but due to the pipeline stall, it completes at PC = 13.

Transcript:

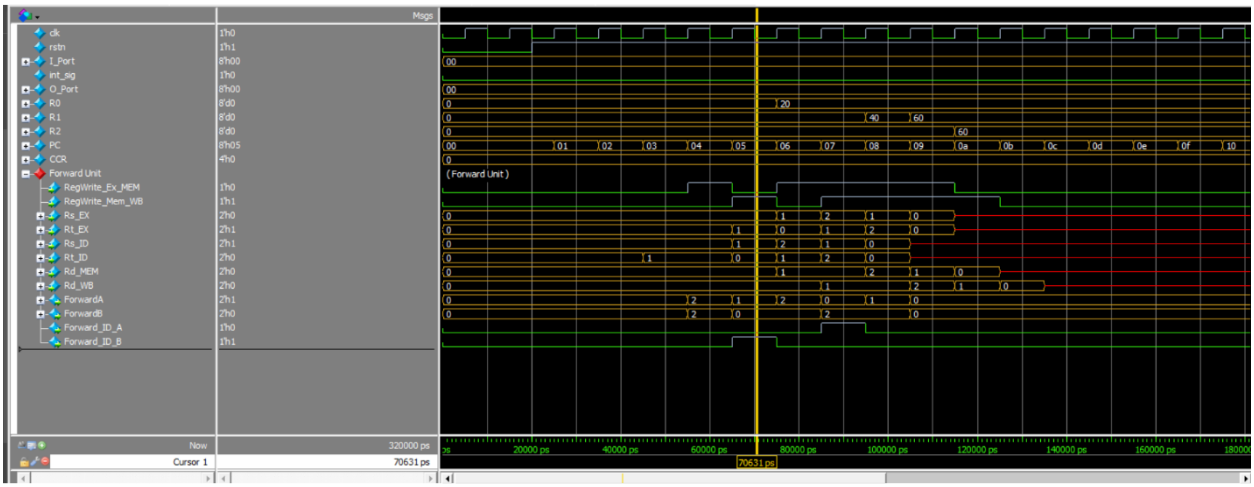
```
#
# ====================================
#  ADD / SUB PIPELINE TEST PASSED
# ====================================
# R0=15 R1=25 R2=35 R3=0 PC=2e
# ** Note: $stop    : F:/Third Year/First Term/Advance Micr
#    Time: 310 ns  Iteration: 0  Instance: /tb_OutputPort
```

3. Program:

```
LDM R0, 20
LDM R1, 40
ADD R1, R0
ADD R2, R1
AND R1, R2
```

For forwarding verification, a complex code sequence with several forwarding cases was chosen to thoroughly stress and validate the forwarding unit (FU).

Waveform:



The Forward_ID_B signal forwards the value of R0 (write-data content) into the ID/EX register, ensuring that the operand is available in the execute stage during the following cycle. Similarly, Forward_ID_A forwards the value of R1.

The forwarding control signals operate as follows:

- When Forward_A or Forward_B = 0, the normal operands from the ID/EX register are selected.
- When Forward_A or Forward_B = 1, the operand is forwarded from the write-back (WB) stage.
- When Forward_A or Forward_B = 2, the operand is forwarded from the memory (MEM) stage.

In the fifth cycle, instruction I3 resides in the IF/ID register while I1 is on the WB stage. Since R0 is a source register for I3 and a destination register for I1, bypassing occurs and Forward_ID_B is asserted. A similar forwarding scenario occurs between I2 and I4.

- Between I2 and I3, data is forwarded from the MEM stage to the EX-stage, resulting in Forward_A = 2.
- Between I4 and I2, data is forwarded from the decode (D) stage to the WB stage, causing Forward_ID_A to be asserted.
- Between I3 and I4, forwarding occurs from the MEM stage to the EX-stage, and thus Forward_B = 2.
- Between I5 and I4, bypassing again takes place between the MEM and EX stages.

Finally, between I5 and I3, data is forwarded from the EX-stage to the WB stage, resulting in Forward_A = 1 and Forward_B = 2.

During development, an issue was encountered when forwarding from two-byte instructions to one-byte instructions. After careful debugging and design adjustments, this issue was successfully resolved. The solution involved inserting two multiplexers at the inputs of the ID/EX register source operands. These multiplexers enable direct forwarding from the decode (D) stage to the write-back (WB) stage, allowing correct data bypassing between instructions of different sizes and ensuring proper operand availability in subsequent pipeline stages.

Transcript:

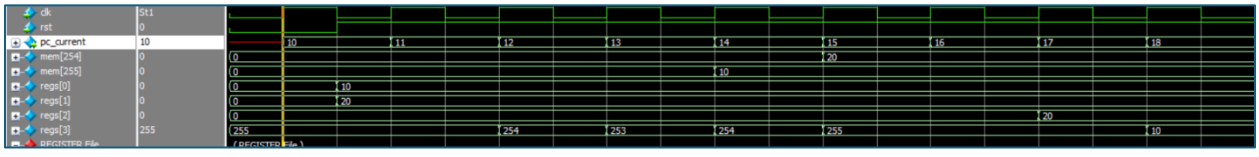
```
# ------------------------------------------------------------
# ALU SEQUENCE TEST REPORT
# ------------------------------------------------------------
# PC Final Value: 1e
# R0 (Expected 20):  20
# R1 (Expected 60):  60
# R2 (Expected 60):  60
# CCR (Flags)     : 0000
# [SUCCESS] ALU sequence executed correctly.
# ------------------------------------------------------------
```

4. Program:

```
0x10 : PUSH R0
0x11 : PUSH R1
0x12 : POP  R2
0x13 : POP  R3
0x14 : NOP
```

The register file is initialized such that R0 = 10, R1 = 20, R3 = 255 while the remaining registers are set to zero.

Waveform:



Upon reset, the PC is initialized to 0x10. After four clock cycles, the value of R0 is pushed onto the stack and written to memory location M[255]. In the fifth clock cycle, the value of R1 is written to M[254].

In the sixth clock cycle, the value 20 becomes available from memory. The value 20 is written into R2 in the seventh clock cycle. Finally, in the eighth clock cycle, the value 10 is written into R3.

1 clk	2 clk	3 clk	4 clk	5 clk	6 clk	7 clk	8 clk
F	D	EX	MEM	WB			
	F	D	EX	MEM	WB		
		F	D	EX	MEM	WB	
			F	D	EX	MEM	WB

Transcript:

```
#
# ====================================
#  PUSH PUSH POP POP TEST PASSED
# ====================================
# R0=10 R1=20 R2=20 R3=10
```

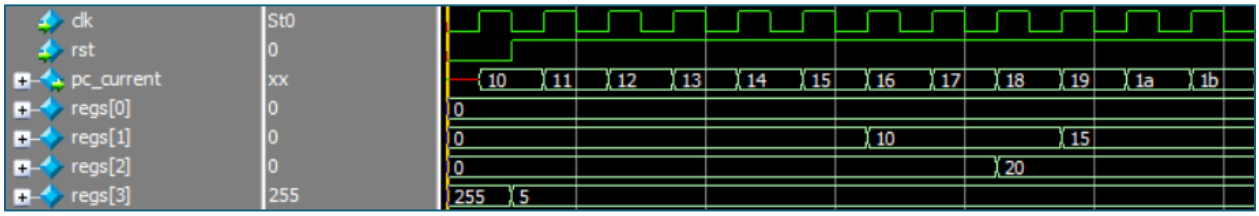
5. Program (Difficult Program)

This program was discussed with the TA during an offline meeting.

```
0x10 : LDD R1, [0x30]
0x12 : LDD R2, [0x31]
0x14 : ADD R1, R3
```

The processor is initialized with PC = 0x10, memory locations M[0x30] = 10 and M[0x31] = 20, and register values R0 = 0, R1 = 0, R2 = 0, and R3 = 5.

Waveform:



Starting from PC = 0x10, the instruction progresses through the pipeline stages as follows. The Fetch stage occurs from PC = 0x10 to 0x11, followed by the Decode stage from 0x11 to 0x13, during which one stall cycle is inserted to wait for the immediate value of the two-byte instruction. The instruction then enters the Execute stage from 0x13 to 0x14, the Memory stage from 0x14 to 0x15, and the Write-Back (WB) stage from 0x15 to 0x16.

As a result, after six clock cycles, the value 10 is written into R1. Due to the decode stall caused by the immediate operand, two additional clock cycles are required before the next result becomes available. Consequently, value 20 is written into R2, followed by the value 15 being written into R1 in the next cycle.

This scenario represents a challenging forwarding case, as it involves dependencies between a two-byte load instruction and a subsequent arithmetic instruction. The observed timing behavior confirms that the processor operates correctly in a pipelined manner, and that the forwarding unit, together with the decode stall mechanism, successfully resolves data hazards while preserving correct execution.

Transcript:

```
# ========================================
# Final State:
# R1 (Result) = 15 (Expected 15)
# R2 (Loaded) = 20 (Expected 20)
# R3 (Source) = 5 (Expected 5)
```

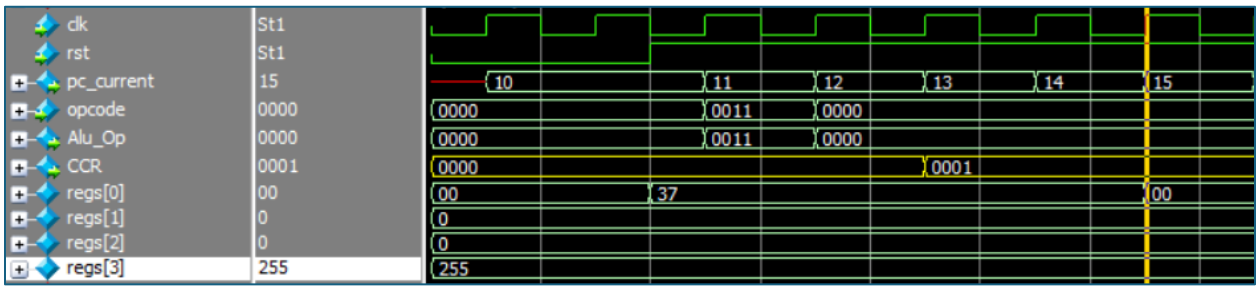
6. Program: Test Flags

Test 1: Zero Flag (Z)

0x10: SUB R0, R0 R0 = 55 (initial value)

Expected Result

- R0 = 0
- Z = 1



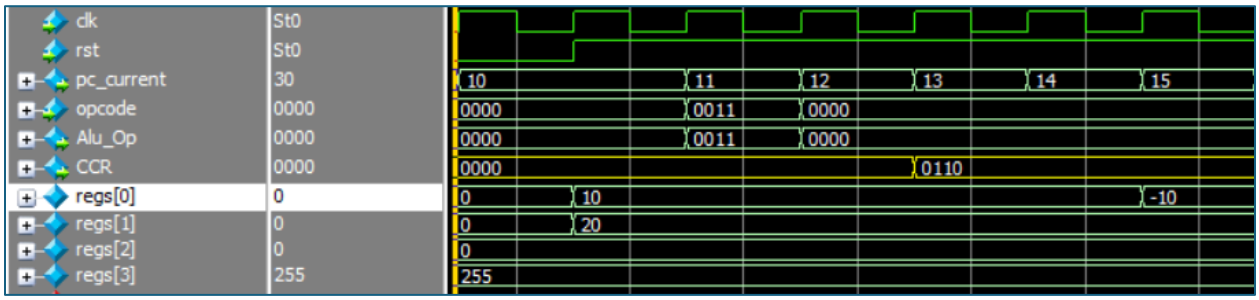
As expected, the Condition Code Register (CCR) is updated during the Execute stage. Accordingly, after the Execute stage, the Z flag is asserted (Z = 1). Due to the pipelined architecture, the computed result is written back to the register file after the remaining pipeline stages. Consequently, after two additional clock cycles, the value 0 is written into R0.

Test 2: Negative Flag (N)

SUB R0, R1 R0 = 10 (initial value) R1 = 20 (initial value)

Expected Result

- R0 = -10 (two's complement)
- N = 1
- Z = 0
- C = 1

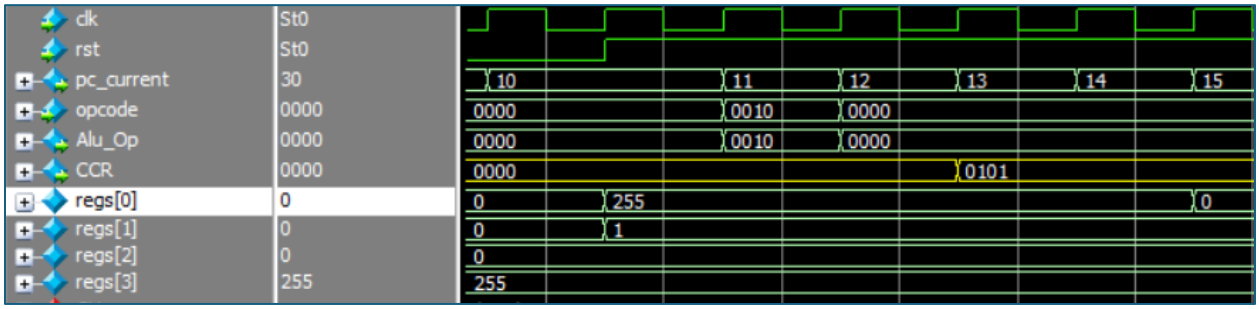


Test 3: Negative Flag (N)

ADD R0, R1 R0 = 255 (initial value) R1 = 1 (initial value)

Expected Result

- R0 = 0
- C = 1
- Z = 1



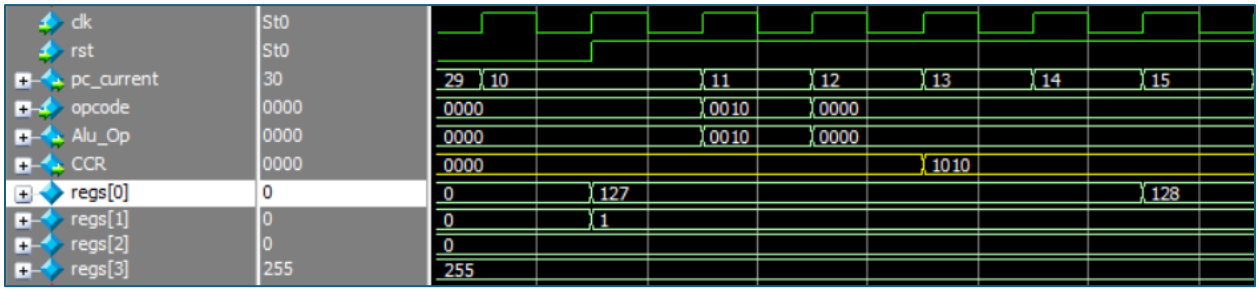
Test 4: Overflow Flag (V)

ADD R0, R1 R0 = 127 (initial value) R1 = 1 (initial value)

Expected Result

- R0 = 128 (−128 in two's complement)

- V = 1
- N = 1

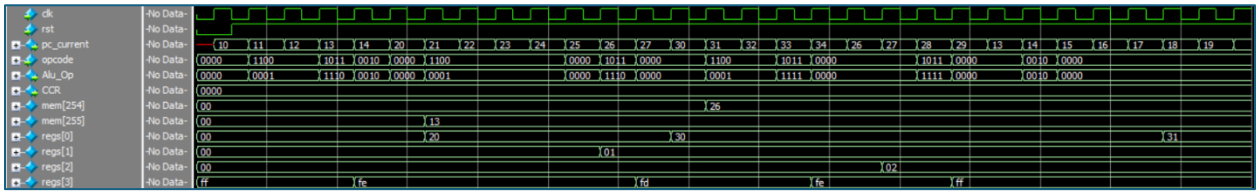


7. Program: CALL–CALL (nested subroutines)

Memory Address	Hex Value	Instruction / Description
0x00	0x10	Reset vector → PC starts at 0x10
0x10	0xC0	LDM R0, 0x20
0x11	0x20	Immediate value (Subroutine 1 address)
0x12	0xB4	CALL R0
0x13	0x21	ADD R0, R1
0x20	0xC1	LDM R1, 0x01
0x21	0x01	Immediate value
0x22	0xC0	LDM R0, 0x30
0x23	0x30	Immediate value (Subroutine 2 address)
0x24	0x00	NOP
0x25	0xB4	CALL R0
0x26	0x00	NOP
0x27	0xB8	RET
0x30	0xC2	LDM R2, 0x02
0x31	0x02	Immediate value
0x32	0xB8	RET

All general-purpose registers are cleared, while the stack pointer is initialized to its default value. Specifically, R0 = 0x00, R1 = 0x00, R2 = 0x00, and R3 = 0xFF.

Waveform:



The program is organized in memory to demonstrate correct execution of nested subroutine calls using a stack-based control flow.

Execution begins at the reset vector (0x00), which initializes the Program Counter (PC) to 0x10, the start address of the main program.

At address 0x10, the main program loads the address of Subroutine 1 (0x20) into R0 using an LDM instruction. The immediate value following this instruction is stored at 0x11. The CALL R0 instruction at 0x12 then transfers control to Subroutine 1(0x20) while pushing the return address onto the stack.

Subroutine 1, located at 0x20, begins by loading the constant value 0x01 into R1. It then loads the address of Subroutine 2 (0x30) into R0 and performs a nested CALL R0 at 0x25.

Subroutine 2, starting at 0x30, loads the value 0x02 into R2 and then executes a RET instruction at 0x32, returning control back to Subroutine 1 (0x26).

A RET instruction at 0x27 returns execution to the main program after Subroutine 2 completes.

After returning from the subroutine calls, execution resumes at 0x13, where an ADD R0, R1 (R0= R0 + R1= 0x31) instruction is executed to confirm correct return behavior and pipeline continuation.

The NOP instructions at 0x24 and 0x26 are used for instruction alignment and to simplify pipeline timing.

Transcript:

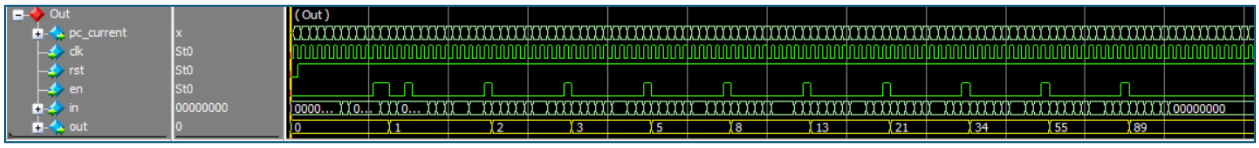
```
# ====================================
#  TWO CALLS TEST PASSED
# ====================================
# R0=31 R1=1 R2=2 R3=ff
# ** Note: $stop    : F:/Third Year/First
#    Time: 300 ns  Iteration: 0  Instance:
```

8. Program (Fibonacci sequence)

```
0x00 : 0x02    ; Reset vector → PC = 0x02
0x01 : 0x00

0x02 : LDM R0, 0x0A    ; Loop counter = 10
0x03 : 0x0A

0x04 : LDM R1, 0x00    ; First Fibonacci term = 0
0x05 : 0x00

0x06 : LDM R2, 0x01    ; Second Fibonacci term = 1
0x07 : 0x01

0x08 : OUT R1          ; Output first term
0x09 : OUT R2          ; Output second term

0x0A : MOV R3, R1      ; R3 = R1
0x0B : ADD R3, R2      ; R3 = R1 + R2
0x0C : OUT R3          ; Output next Fibonacci number
0x0D : MOV R1, R2      ; R1 = R2
0x0E : MOV R2, R3      ; R2 = R3

0x0F : LDM R3, 0x0A    ; Load loop start address
0x10 : 0x0A

0x11 : LOOP R0, R3     ; Decrement R0 and loop if not zero
0x12 : NOP
0x13 : NOP
```

PC = 0x02, R0 = 0x00, R1 = 0x00, R2 = 0x00, R3 = 0x00

Waveform:



Transcript:

```
--- Simulation Finished ---
Final Registers: R1= 55, R2= 89, R3= 10
** Note: $stop    : F:/Third Year/First Ter
   Time: 2010 ns  Iteration: 0  Instance: /
```

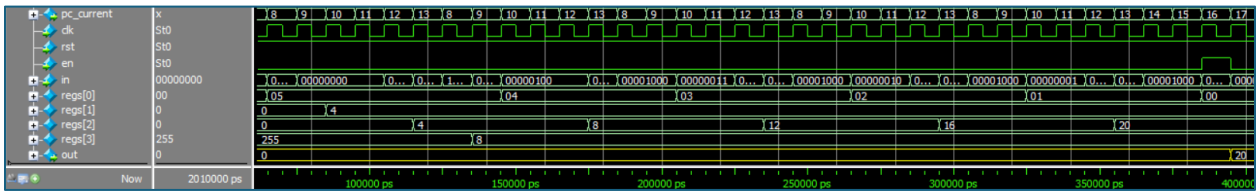
9. Program (Multiplication Using Repeated Addition)

This program computes the multiplication 5 × 4 = 20 (0x14) using repeated addition and a loop instruction. The result is accumulated in R2 and displayed through the output port.

```
0x00 : 0x02          ; Reset vector → PC = 0x02
0x01 : 0x00

0x02 : LDM R0, 0x05  ; R0 = 5 (loop counter / multiplier)
0x03 : 0x05

0x04 : LDM R1, 0x04  ; R1 = 4 (multiplicand)
0x05 : 0x04

0x06 : LDM R2, 0x00  ; R2 = 0 (result accumulator)
0x07 : 0x00

0x08 : ADD R2, R1    ; R2 = R2 + R1
0x09 : LDM R3, 0x08  ; Load loop start address
0x0A : 0x08

0x0B : LOOP R0, R3   ; Decrement R0 and jump to 0x08 if R0 ≠ 0
0x0C : OUT R2        ; Output result (20 / 0x14)
0x0D : NOP
```

PC = 0x02, R0 = 0x00, R1 = 0x00, R2 = 0x00, R3 = 0x00

Waveform:



Transcript:

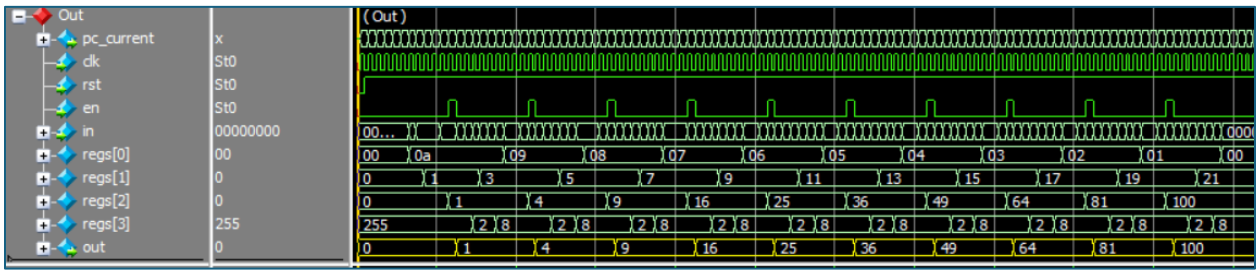
```
--- Simulation Finished ---
Final Registers: R1=  4, R2= 20, R3=  8
```

10. Program (SQUARE SERIES)

```
0x00 : 0x02          ; Reset vector → PC = 0x02
0x01 : 0x00

0x02 : LDM R0, 0x0A  ; R0 = 10 (number of squares)
0x03 : 0x0A

0x04 : LDM R1, 0x01  ; R1 = 1 (first odd number)
0x05 : 0x01

0x06 : LDM R2, 0x00  ; R2 = 0 (square accumulator)
0x07 : 0x00

0x08 : ADD R2, R1    ; R2 = R2 + R1  (compute next square)
0x09 : OUT R2        ; Output square value

0x0A : LDM R3, 0x02 ; Load constant 2
0x0B : 0x02

0x0C : ADD R1, R3    ; R1 = R1 + 2 (next odd number)

0x0D : LDM R3, 0x08 ; Load loop start address
0x0E : 0x08

0x0F : LOOP R0, R3  ; Decrement R0 and jump to 0x08 if R0 ≠ 0
0x10 : NOP
```

PC = 0x02, R0 = 0x00, R1 = 0x00, R2 = 0x00, R3 = 0x00

Waveform:



Transcript:

```
# --- Simulation Finished ---
# Final Registers: R1= 21, R2=100, R3=  8
# Test Case Passed
```

During testing:

ensure that M [0] is initialized before asserting Reset = 0 in the testbench, and that the memory reset logic is executed first. This prevents unknown (X) (not problem) values from appearing on signals during simulation.

```
integer i;
initial begin
    clk = 0;
    // ------------------------------------------------
    // Clear stack/data only
    // ------------------------------------------------
    for (i = 8'd0; i < 256; i = i + 1)
        uut.mem_inst.mem[i] = 8'h00;
    // ------------------------------------------------
    // Reset vector → PC = 0x10
    // ------------------------------------------------
    uut.mem_inst.mem[8'h00] = 8'h10;
    rstn = 0;
    I_Port = 0;
    int_sig = 0;
```

V. Conclusion

In conclusion, we have successfully designed and implemented an 8-bit pipelined processor based on the Von Neumann architecture. The design strictly adheres to the specified RISC-like ISA, effectively integrating a Finite State Machine (FSM) control unit to manage instruction execution and memory resource sharing. Key architectural features, including data forwarding and interrupt handling, were implemented to ensure robust operation and maximize instruction throughput.

Verification was a critical phase of this project. Although this report presents the main functional tests and waveform analysis, additional validation cases particularly interrupt handling, complex edge conditions, and extended instruction sequences are demonstrated in the demo video.