Faculty of Engineering
Department of Electronic and
Electrical Communications Engineering

Advanced Microprocessor (ELC3030) Project

NAME	SEC	BN	ID
Ahmed Mohamed Abd-El-Moneam	1	19	9230176
Amira Salah El-Din Mohamed	1	39	9230247
Aya Medhat Essam El-Dein	1	43	9230269
Sandra Atef Kamal	2	24	9230428
Kareem Hassan Atef	3	26	9230666
Mahmoud Hussieny Matar	4	14	9230832

Under the supervision of:

Dr. Hossam Fahmy

&

Eng. Hassan El-Monier

Table of Contents

1. Introduction

This project presents the design of an **8-bit pipelined processor** implemented using **Verilog HDL**. The processor supports arithmetic, logical, memory, control flow, stack, and interrupt instructions based on the given instruction set.

To improve performance, the processor uses a **five-stage pipeline**, allowing multiple instructions to be executed at the same time. Since pipelining can cause data and control hazards, the design includes a **hazard detection unit**, a **forwarding unit**, and mechanisms for **stalling and flushing** to ensure correct execution.

The processor follows a **Harvard architecture** with separate instruction and data memories. A stack pointer is implemented to support function calls, returns, and interrupt handling. This project demonstrates practical understanding of pipelined processor design and hardware implementation using Verilog.

2. Instruction Set Architecture Overview

The processor uses a simple 8-bit instruction set with 1 or 2-byte instructions. It has four 8-bit registers, with R3 acting as the stack pointer, an 8-bit program counter (PC), and four flags (Z, N, C, V) in the condition code register. The instruction set includes arithmetic, logical, data transfer, and control instructions, as well as stack and interrupt handling.

There are three instruction formats:

- **A-Format** for arithmetic, logical, and stack operations.
- **B-Format** for jumps, loops, calls, and returns.
- **L-Format** for load/store instructions.

3. Overall Processor Architecture

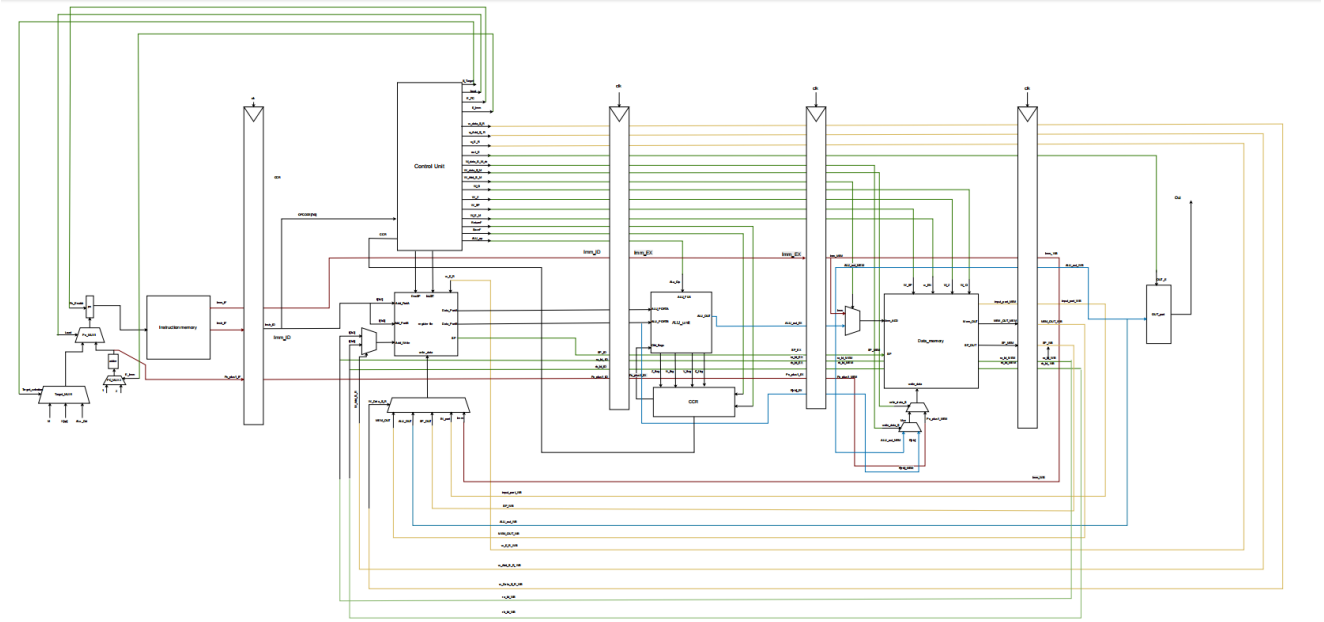


Figure 1: Processor Architecture

This is the Pipelined architecture for the detailed one please go for the PDF file attached or (Detailed Arch)

Full design with clear details can be found here.

The processor is organized as a **five-stage pipelined Datapath**. Each stage performs a specific function, and pipeline registers are used to pass data and control signals between stages.

The five stages are:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory Access (MEM)
5. Write Back (WB)

Between these stages, the following pipeline registers are used:

- IF/ID register
- ID/EX register
- EX/MEM register
- MEM/WB register

In addition to the datapath, the architecture includes several control and support units:

- Program Counter (PC) and PC Control Unit
- Instruction Memory
- Data Memory
- Register File
- Arithmetic Logic Unit (ALU)
- Condition Code Register (CCR)
- Hazard Detection Unit
- Forwarding Unit

The processor also supports **stack-based operations** and **interrupt handling**, which are implemented using the stack pointer and special memory locations.

4. Control Unit

The **Control Unit** is responsible for generating all control signals needed for correct execution of instructions in the pipelined processor. It controls the behavior of the datapath by decoding the instruction opcode and using status signals such as condition flags, reset, and interrupt signals.

In this design, the control unit is mainly **combinational**. This means that the control signals are generated directly from the instruction fields and processor status, without using a complex multi-cycle controller.

5. Datapath Description

- ## Instruction Fetch (IF) Stage

The **Instruction Fetch stage** is responsible for fetching instructions from instruction memory and updating the Program Counter (PC).

5.1.1 Program Counter and PC Control Unit

The **PC Control Unit** controls how the PC is updated. This unit is fully based on the instruction opcode, condition flags, reset signal, and interrupt signal.

The PC can be updated in several ways:

- **PC + 1** for normal instructions
- **PC + 2** for instructions that use an immediate value
- Loading a target address for jump, call, return, and interrupt instructions
- Stall

The PC target can be selected from:

- Memory location **M [0]** (used during reset)
- Memory location **M [1]** (used during interrupt)
- Stack pointer data (used for RET and RTI)
- Register R[rb] (used for jump and call instructions)

This detailed control allows correct execution of all control flow instructions implemented in the design.

5.1.2 Instruction Memory

Instruction memory is implemented as a **256 × 8-bit ROM**. Instructions are stored in a file named program.txt and loaded into memory at the start of simulation. Each instruction occupies one byte, and some instructions use an additional byte for an immediate value or effective address.

For every clock cycle, the instruction memory outputs:

- The current instruction at address PC
- The next byte at address PC + 1, which is used as an immediate value or address when needed

5.1.3 IF/ID Pipeline Register

The IF/ID pipeline register stores:

- The fetched instruction
- The immediate value
- The value of PC + 1
- Input port data
- Interrupt

This register supports:

- **Stall**, which holds the instruction when a data hazard is detected

- **Instruction Decode (ID) Stage**

The **Instruction Decode stage** prepares the instruction for execution.

5.1.4 Instruction Decoding

In this stage, the opcode field is decoded to determine:

- The instruction type
- ALU operation
- Memory access requirements
- Register write-back behaviour

Control signals are generated and passed to the next stages through the ID/EX pipeline register.

5.1.5 Register File Operation

The register file contains **four 8-bit registers**:

- R0, R1, R2: general-purpose registers
- R3: used as the **Stack Pointer (SP)**

The register file supports:

- Two simultaneous read operations
- One write operation
- Stack pointer increment and decrement (in case like push and pop)

On the beginning, the stack pointer is Forced to **255**, allowing the stack to grow downward.

5.1.6 Hazard Detection Unit

1. Data Hazards

When the current instruction depends on data that is being loaded from memory by the previous instruction, a **load-use hazard** occurs. In this case:

- The pipeline is stalled for one cycle
- The ID stage hold its value

2. Control Hazards

Control hazards occur due to:

- JMP, CALL and Conditional jumps (JZ, JN, JC, JV) - one stall (knowing the flags of the previous Instr and address)
- LOOP instruction - two stalls (knowing the flags of my Instr and address)
- RET, and RTI instructions - three stalls

When a control hazard is detected, a **Stall signal** is Stall the pc from updating until I Know the Address or the condition if I will branch or continue

- **Execute (EX) Stage**

The **Execute stage** performs all computations required by the instruction.

5.1.7 ALU Operations

The ALU supports:

- Arithmetic operations (ADD, SUB)
- Logical operations (AND, OR)
- Comparison operations

It also updates the **Condition Code Register (CCR)**, which contains:

- Zero flag
- Negative flag
- Carry flag
- Overflow flag

It stores the flags when needed

These flags are later used by conditional jump instructions.

5.1.8 Forwarding Unit

To reduce pipeline stalls, a **Forwarding Unit** is implemented.

The forwarding unit detects **Read After Write (RAW)** hazards and forwards data from:

- ALU output of the previous instruction
- Memory output
- Input port
- Immediate

This allows dependent instructions to execute without waiting for the write-back stage whenever possible.

5.1.9 ID/EX Pipeline Register

The ID/EX register stores:

- ALU control signals
- Memory control signals
- Register operands
- Immediate values
- Stack pointer value
- Instruction opcode

- **Memory Access (MEM) Stage**

The **Memory Access stage** handles all data memory operations.

5.1.10 Data Memory Design

Data memory is implemented as a **256 × 8-bit RAM**.

It supports:

- Normal memory read and write
- Stack-based access using the stack pointer

Memory writes are synchronous, while memory reads are combinational.

5.1.11 EX/MEM Pipeline Register

This register passes:

- ALU results
- Stack pointer value
- Register data
- Memory control signals

- **Write Back (WB) Stage**

The **Write Back stage** completes instruction execution.

The data written back to the register file can be selected from:

- ALU result
- Data memory output
- Stack pointer
- Immediate value
- Input port

The destination register is selected based on control signals, and the write operation completes the instruction.

6. Interrupt Handling

The processor supports interrupt handling using an **interrupt signal**, the **stack pointer**, and special memory locations.

When an interrupt occurs, the control unit:

1. Saves the current **Program Counter (PC)** on the stack.
2. Saves the current **condition flags (CCR)** on the stack.
3. Loads the interrupt service routine address from **M [1]** into the PC.

The **RTI (Return from Interrupt)** instruction is used to end the interrupt. It restores the saved flags and PC from the stack, then continues normal program execution.

This design ensures correct program execution before and after an interrupt.

We designed The Program on 3 Stages

- 1st : Designed the single Cycle Version and Tested it
- 2nd : Designed the pipelined Version and tested it without dependencies
- 3rd : Adding the Control Hazard and Forward Unit and handling the dependencies and Bypass

7. Bypassing Handling

- The forwarding unit detects RAW hazards by comparing the source registers of the current instruction with the destination registers of both the previous and the previous-previous instructions when write-back is enabled.
- Based on `w_Data_S_R`, the unit selects the correct bypass source (ALU result, memory output, immediate, or input port) and generates `forward_A` and `forward_B` to feed the ALU inputs directly.
- This allows bypassing for dependencies on the immediately preceding instruction as well as the one before it, eliminating unnecessary stalls.
- In addition, a separate PC bypass path (`forward_b_pc`) forwards the required register value for PC target selection, ensuring correct control flow without stalling.

8. Simulation and Verification

In the Testbench we force The R3=255, PC=0 to start the program from the beginning

- **Test 1**

Load, Arithmetic operation with bypass and jump with bypass

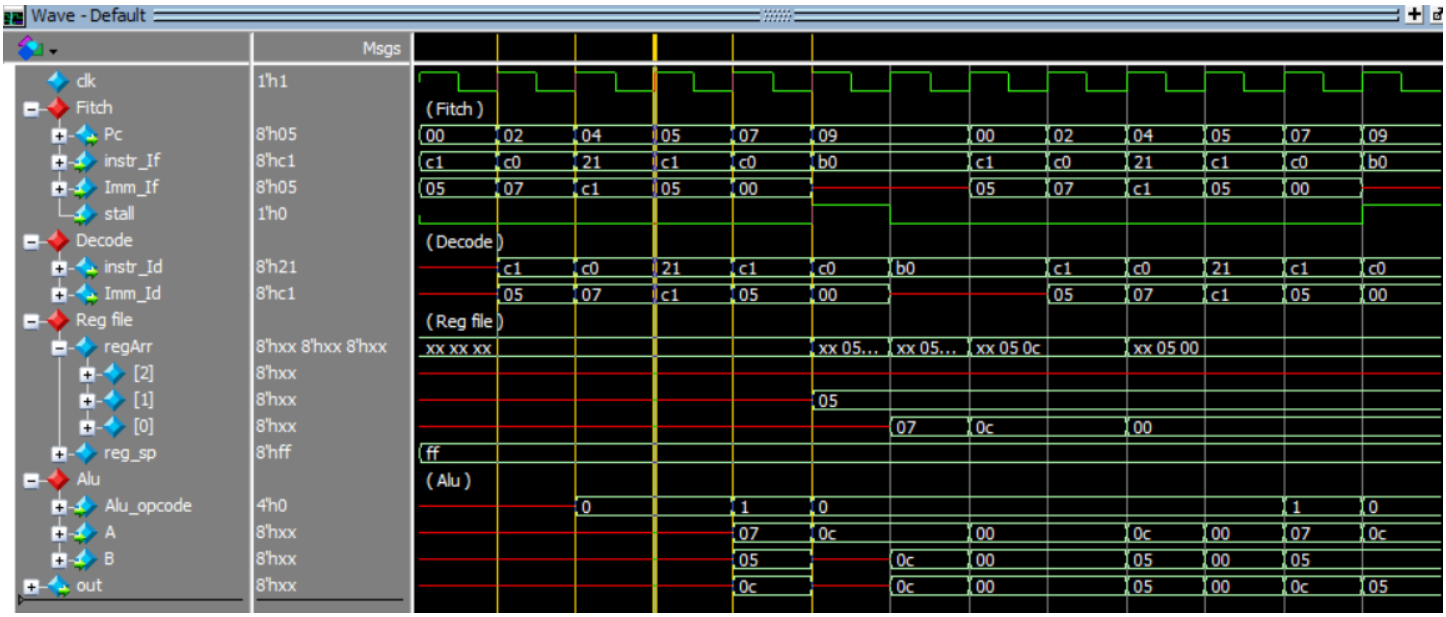
Operation:

Load R1 #05
Load R0 #07
Add R0 R1
Load R1 #05
Load R0 #00
JMP R0

Machine code:

```
C1
05
C0
07
21
C1
05
C0
00
B0
```

Simulation:



Comments:

Successful Load and addition and bypassing the loaded data to the Alu before writing them in the RegFile and unconditional jump with one Stall to know the address and bypassing the loaded Address to the Pc before writing it in the RegFile

- **Test 2**

Load, Arithmetic operation and store ,load from memory with bypass

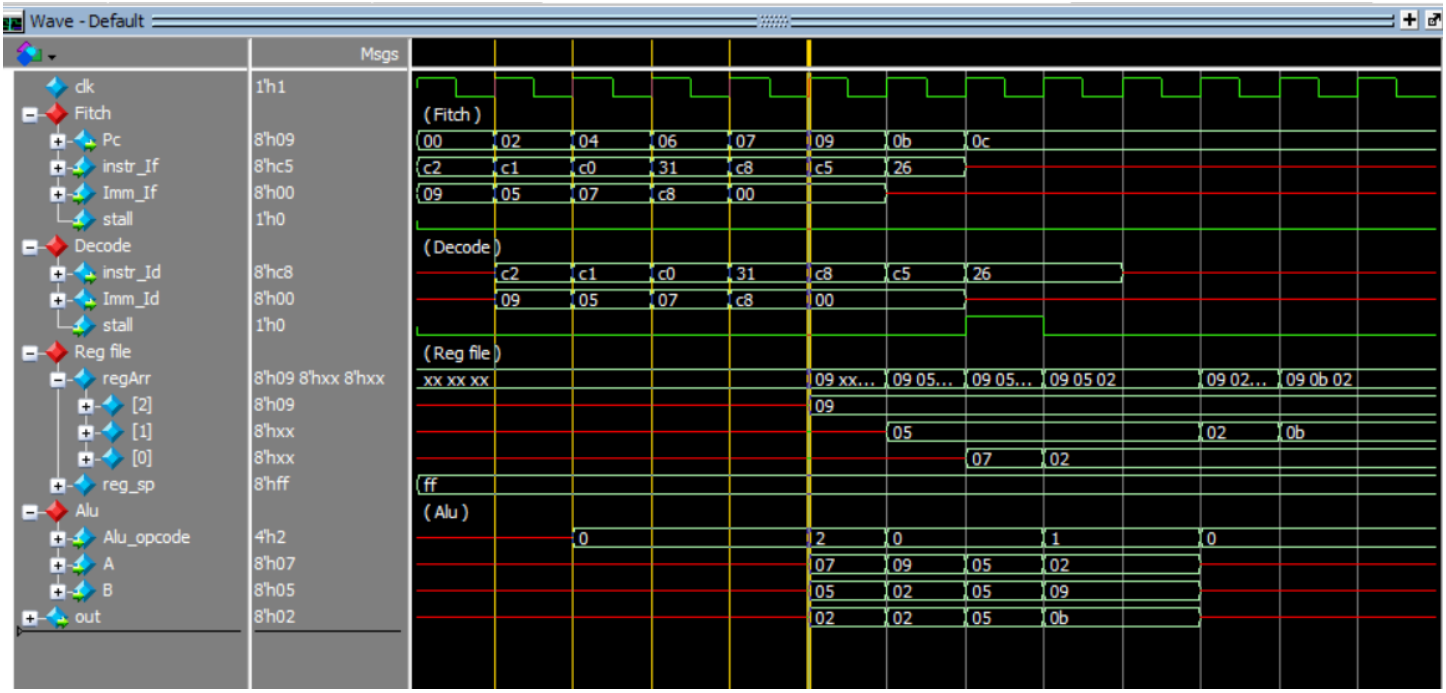
Operation:

Load R2 #09
Load R1 #05
Load R0 #07
Sub R0 R1
Store R0 M [#00]
Load R1 M [#00]
Add R1 R2

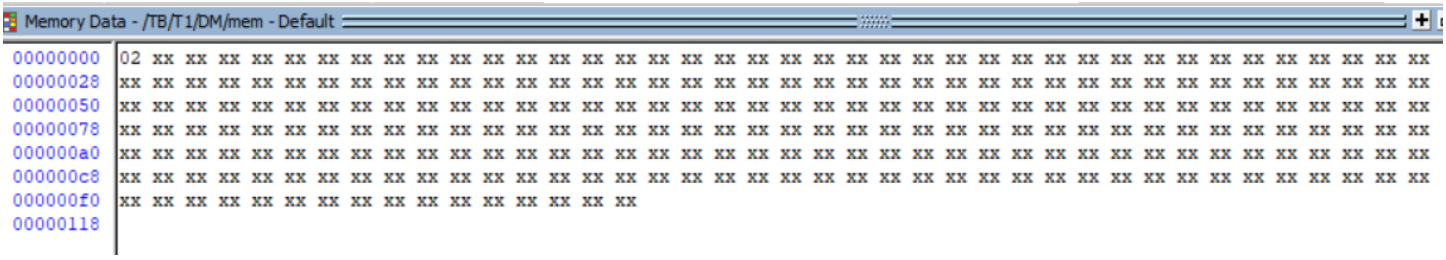
Machine code:

```
C2
09
C1
05
C0
07
31
C8
00
C5
00
26
```

Simulation:



Content of Memory:



Comments:

Successful Load and subtraction and bypassing the loaded data to the Alu before writing them in the RegFile and storing the result in the memory and loading it to the regfile and addition operation with one Stall and bypass to take the date from memory

- **Test 3**

Zero Flag Generation and Conditional Jump test

Operation:

LDM R0, #0a

LDM R1, #04

LDM R2, #4

SUB R1, R2

JZ R0

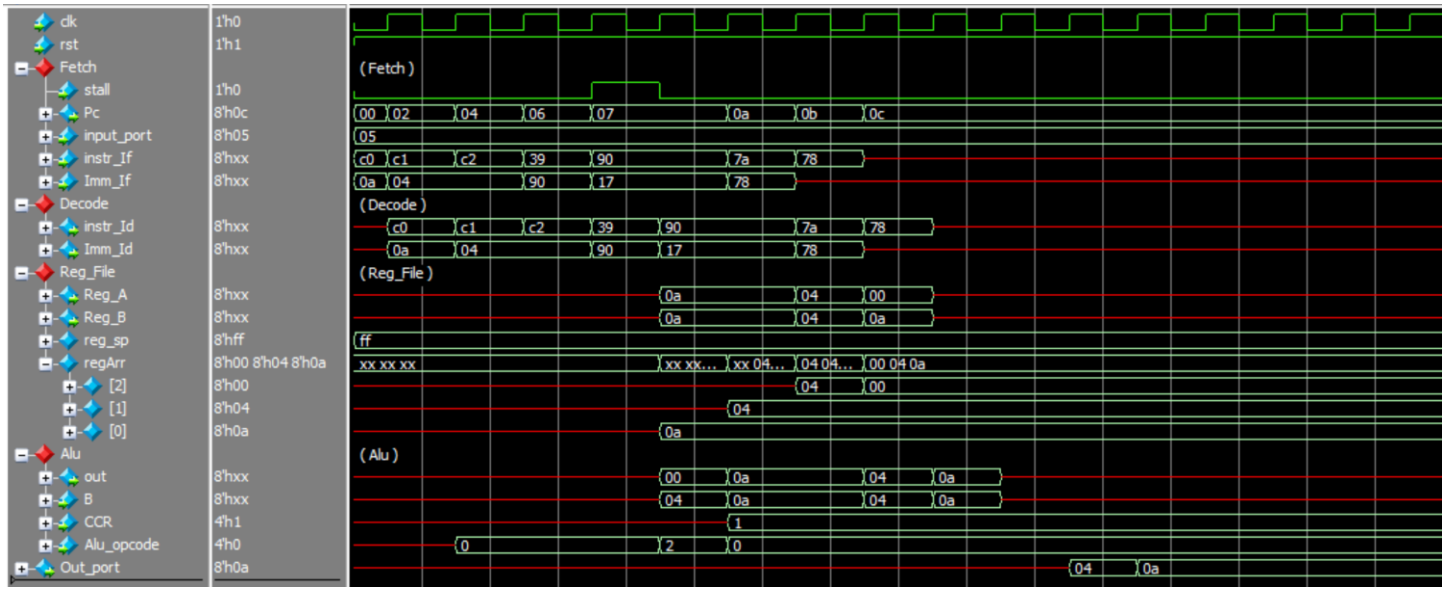
OUT R2

OUT R0

Machine code:

```
C0
0a
c1
04
c2
04
39
90
17
12
7a
78
```

Simulation:



Comment:

Successful load of operands and subtraction operation resulting in a zero value. The zero flag is generated correctly by the ALU and used immediately by the conditional jump instruction. The jump decision depends on a flag produced in the previous stage, and the control logic handles this dependency correctly by allowing the branch to be taken without executing the following instruction. This test confirms proper flag update, branch condition evaluation, and correct program counter redirection for the JZ instruction.

- **Test 4**

Load, Stack Operations, and Output with Bypassing

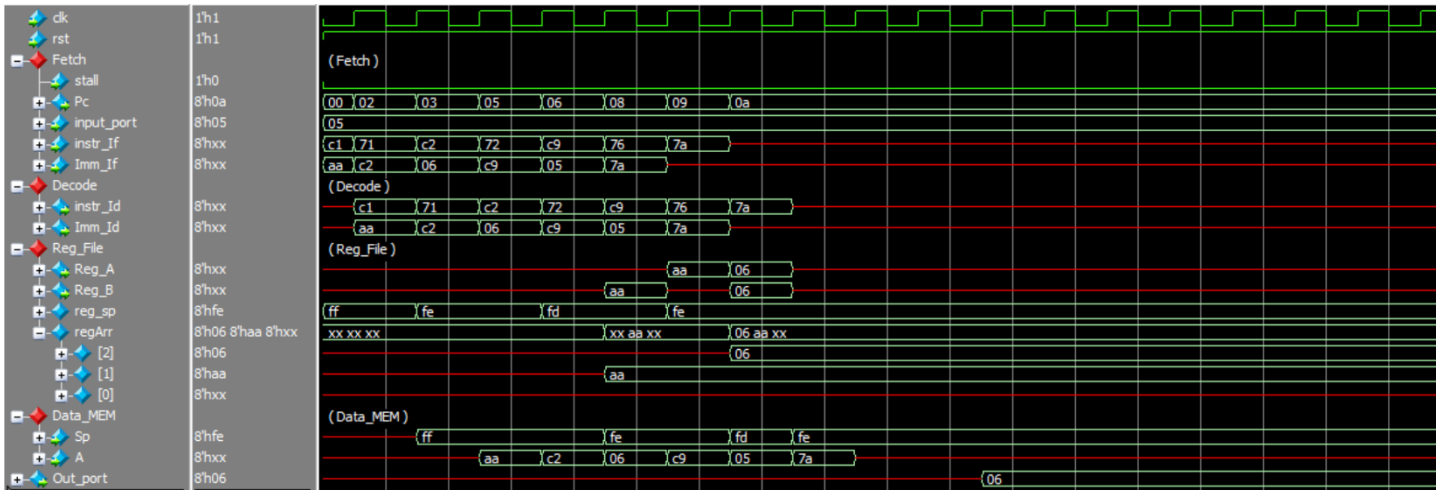
Operation:

LDM R1, #0xAA
PUSH R1
LDM R2, #0x06
PUSH R2
STD R1, #0x05
POP R2
OUT R2

Machine code:

```
c1
aa
71
c2
06
72
c9
05
76
7a
```

Simulation:



Content of Memory After Push R1:

```
sim:/TB/T1/DM/mem @ 107 ns
  0 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 16 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 32 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 48 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 64 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 80 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 96 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
112 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
128 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
144 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
160 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
176 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
192 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
208 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
224 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
240 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'haa
```

Content of Memory After Push R2:

```
sim:/TB/T1/DM/mem @ 139 ns
  0 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 16 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 32 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 48 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 64 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 80 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 96 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
112 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
128 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
144 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
160 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
176 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
192 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
208 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
224 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
240 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'h06 8'haa
```

Content of Memory After STD:

```
sim:/TB/T1/DM/mem @ 158 ns
  0 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'haa 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 16 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 32 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 48 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 64 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 80 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 96 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
112 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
128 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
144 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
160 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
176 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
192 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
208 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
224 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
240 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'h06 8'haa
```

Content of Memory After POP R2:

```
sim:/TB/T1/DM/mem @ 191 ns
  0 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'haa 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 16 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 32 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 48 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 64 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 80 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 96 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
112 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
128 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
144 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
160 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
176 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
192 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
208 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
224 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
240 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'haa
```

Comment:

This test successfully verifies **immediate load instructions** and **stack operations** (PUSH and POP).
The processor correctly handles stack pointer updates and memory access during stack operations.

- **Test 5**

We modify test bench to force interrupt to be high when opcode = FF

```
always @(*) begin
        if(T1.Opcode==8'hff)
            interrupt=1;
            else
            interrupt=0;
    end
```

Load, interrupt, load, INC, return from interrupt

Operation:

LDM R0,#0x09

LDM R1,#0x04

LDM R2,#0x03

Interrupt

LDM R2,#0x05

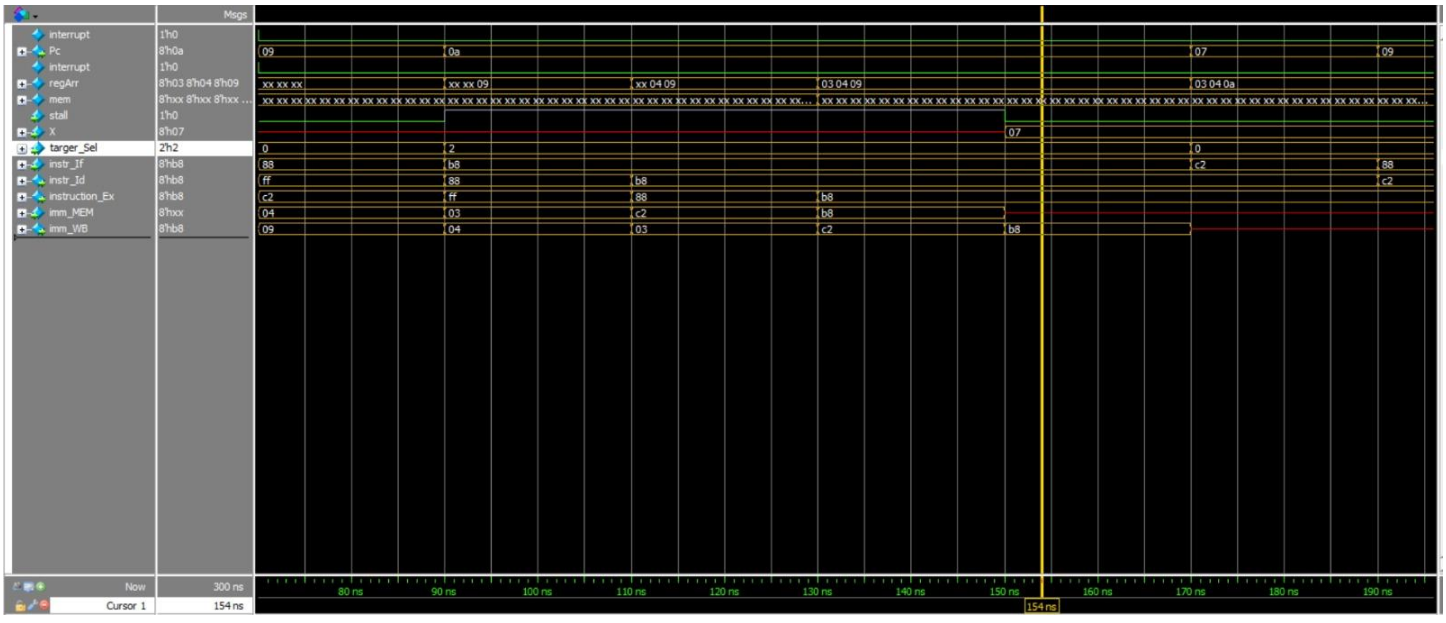
INC R0

RET

Machine code:

```
c0
09
c1
04
c2
03
FF
c2
05
88
b8
```

Simulation:



Memory after interrupt:

```
sim:/TB/T1/DM/mem @ 167 ns
  0 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 16 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 32 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 48 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 64 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 80 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
 96 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
112 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
128 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
144 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
160 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
176 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
192 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
208 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
224 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx
240 : 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'hxx 8'h07
```

Comment:

when interrupt(at pc =6) is high after 4 cycles pc+1 write in sp (7) and pc will be M[1] (09), then we load in register2(5) and add value in register 0 and register 1 and load in register 0 when inst_if=b8"RET" we stall until pc+1 to write in memory and to read it so stall three cycles after reach data memory , the pc will updated with last pc+1 before interrupt (7) . I add in wave value of immediate to show the validation of our pipeline architecture.

- Test 6

Load, Set curry, clear carry, LOOP

Operation:

LDM R0, #0x02

LDM R1, #0x01

LDM R2, #0x00

SETC

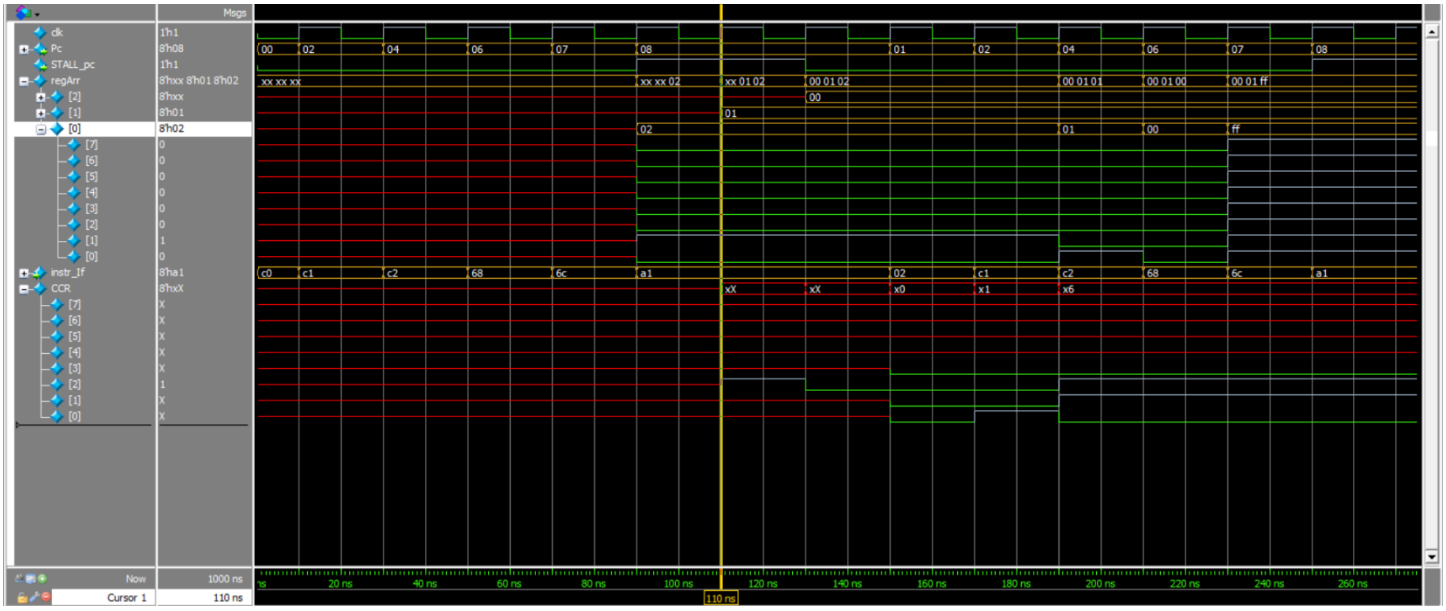
CLRC

LOOP

Machine code:

```
c0
02
c1
01
c2
00
68
6c
A1
```

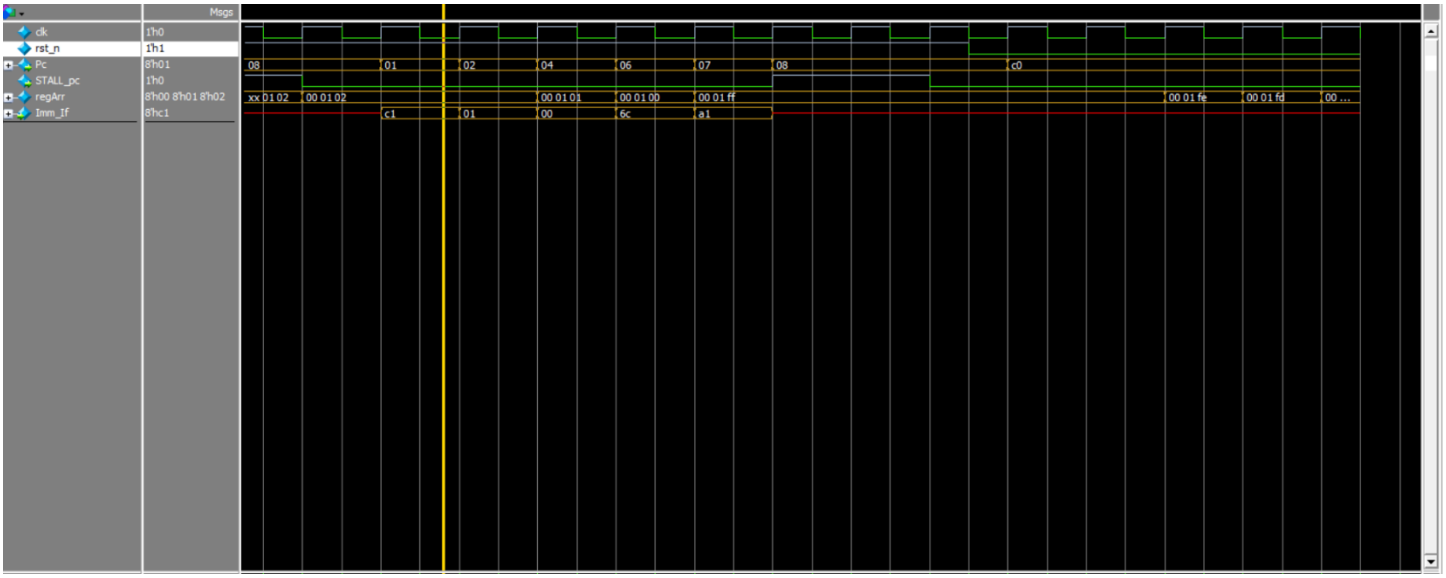
Simulation:



Comment: we stall in pc until execution is done after fetch loop instruction (A1), so we stall 2 cycles until (R[ra]-1) is calculated, then pc will be (R[rb] =01) which is shown in wave.

For same program we test if rst_n is low pc will be M [0] (c0)

Simulation:

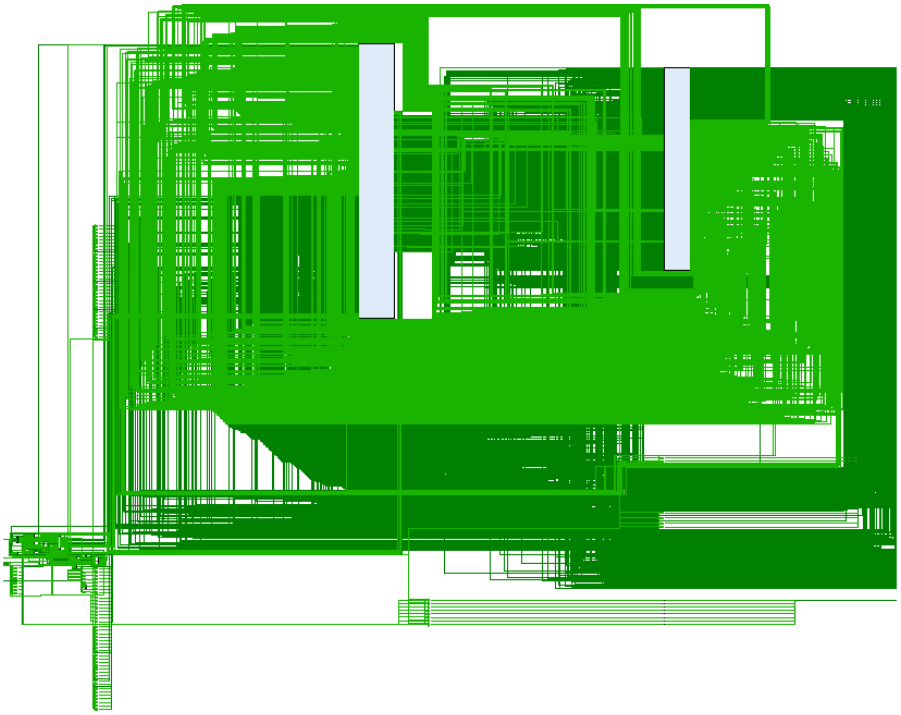


Comment:

Pc updated after one clock cycle due to M[0] is instruction memory and value read combinational and return to pc_mux in same time update after one clock for sequential update for PC register.

9. Synthesis (bonus part)

- **Schematic**



- **Timing Report**

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.144 ns	Worst Hold Slack (WHS):	0.142 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:	4353	Total Number of Endpoints:	4353	Total Number of Endpoints:	4331

All user specified timing constraints are met.

Clock Summary

Name	Waveform	Period (ns)	Frequency (MHz)
sys_clk_pin	{0.000 5.000}	10.000	100.000

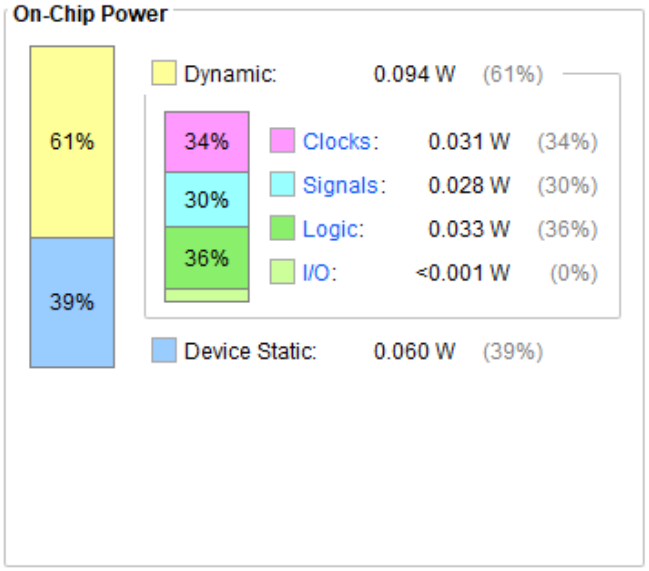
- **Power Report**

Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

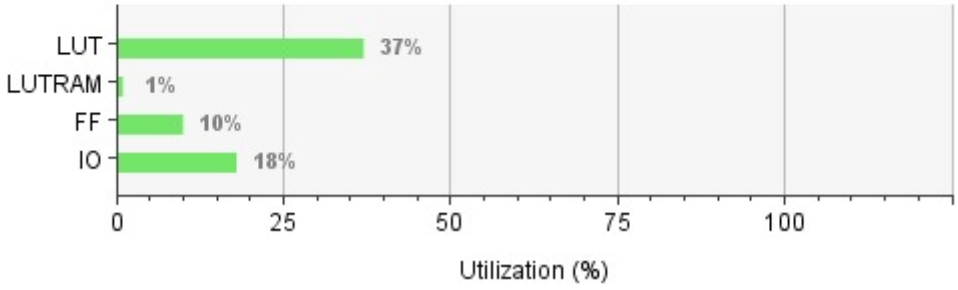
Total On-Chip Power:	**0.154 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.8°C**
Thermal Margin:	74.2°C (14.8 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



- **Utilization Report**

Resource	Utilization	Available	Utilization %
LUT	7704	20800	37.04
LUTRAM	9	9600	0.09
FF	4337	41600	10.43
IO	19	106	17.92



10. Conclusion

In this project, an **8-bit pipelined processor** was successfully designed and implemented using Verilog HDL. The processor supports multiple instruction types, pipelining, hazard handling, stack operations, and interrupts.

This project provided valuable experience in processor architecture design, pipeline control, and hardware description using Verilog. Future improvements could include adding more instructions, increasing data width, or improving performance.

11. Appendix (HDL Code)

- All project Codes Versions or material used (The final version Named Final Version Pip) (GitHub Link)
- Demo video explaining the project (Demo video)
- Final Architecture In high quality and detailed (Detailed Arch)