Cairo
University

# 8-Bit Pipelined RISC-Like Processor

Submitted to  Eng Hassan El Menier

ELC 3030 - Advanced Processor Architecture

# Overview

# Project Overview

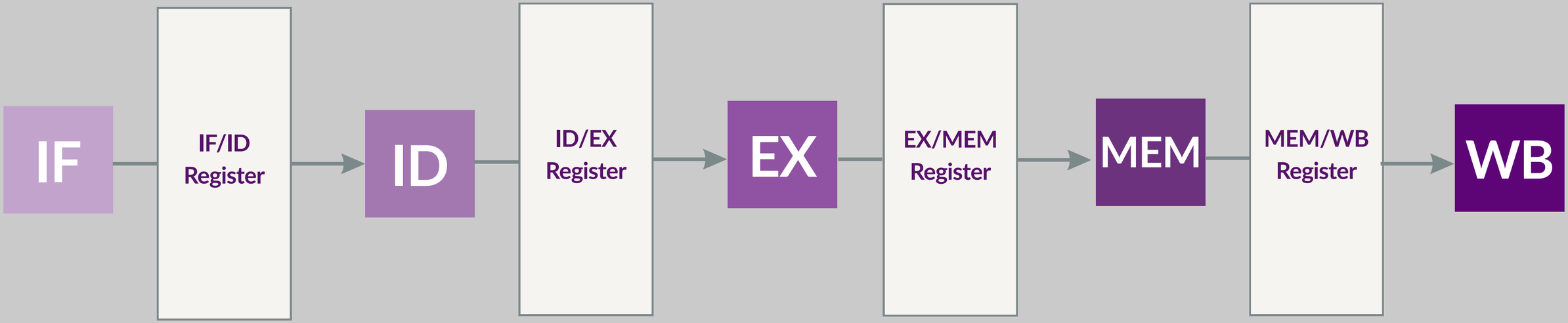
01 **Design of a simple 8-bit pipelined processor**

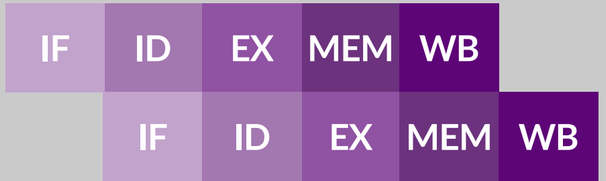
02 **RISC-like Instruction Set Architecture**

03 **Implemented using Verilog HDL**

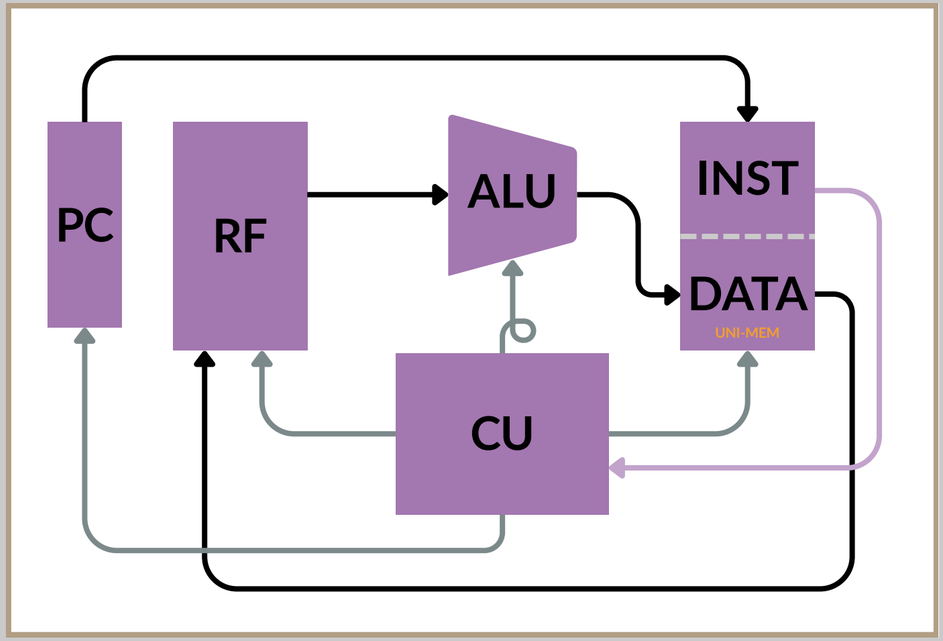
04 **Supports arithmetic, logic, memory, and control instructions**

# Pipeline Design



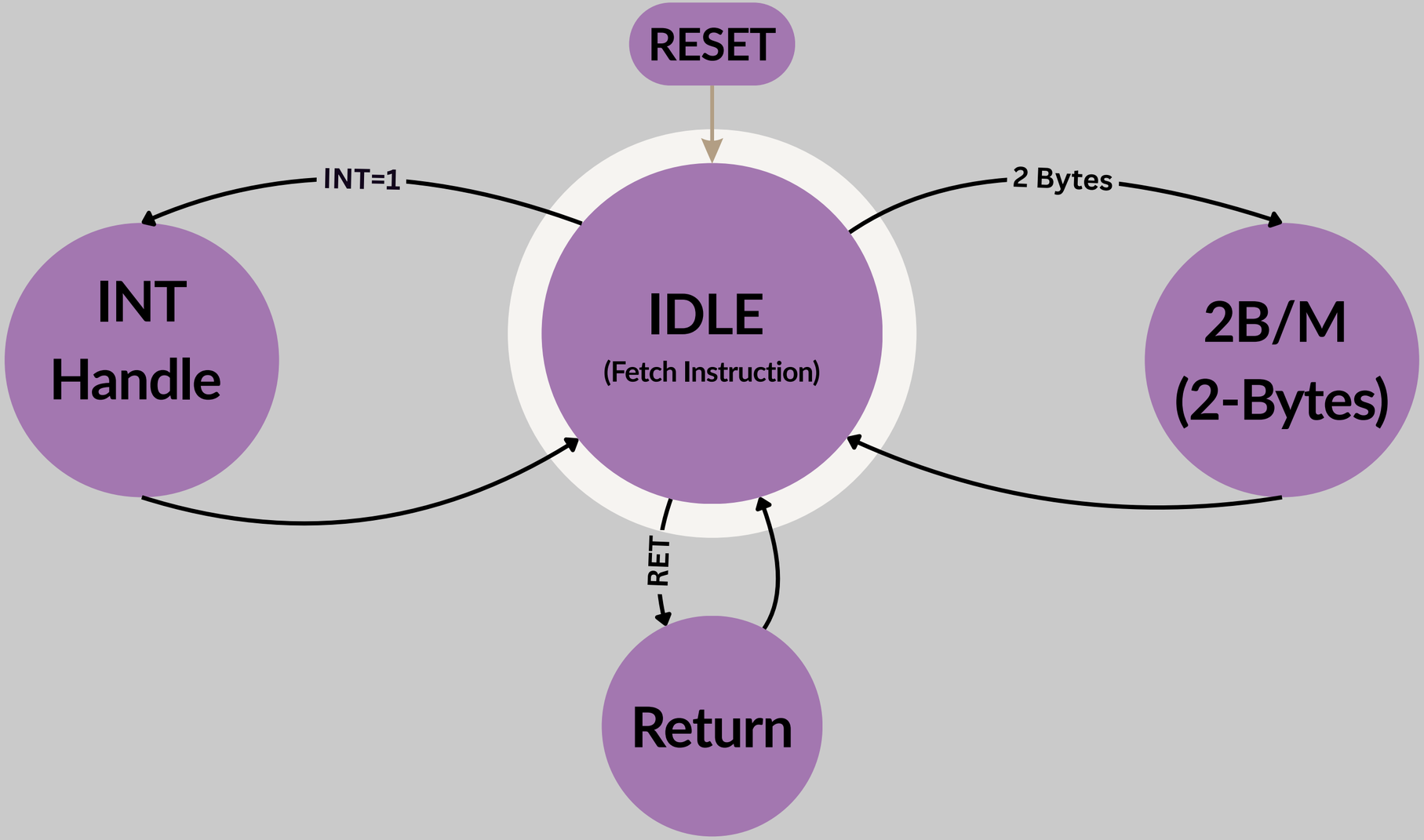



# System Architecture




- 8-bit Program Counter
- Four General Purpose Registers (R0–R3)
- Stack Pointer (R3)
- Implemented using both Harvard and Von Neumann architectures
- FSM-based Control Unit

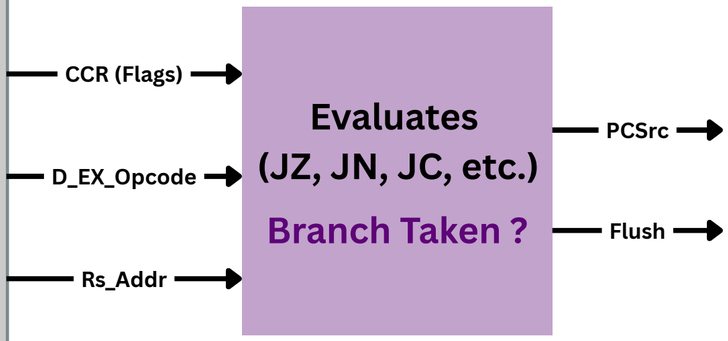
# Control Unit Design-FSM



# Key Features Implemented

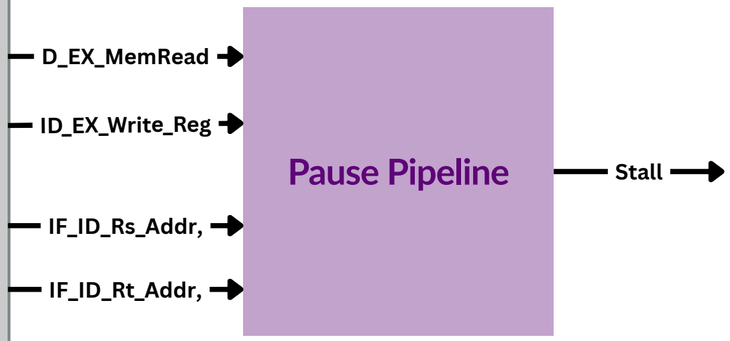
01

## Branch Unit

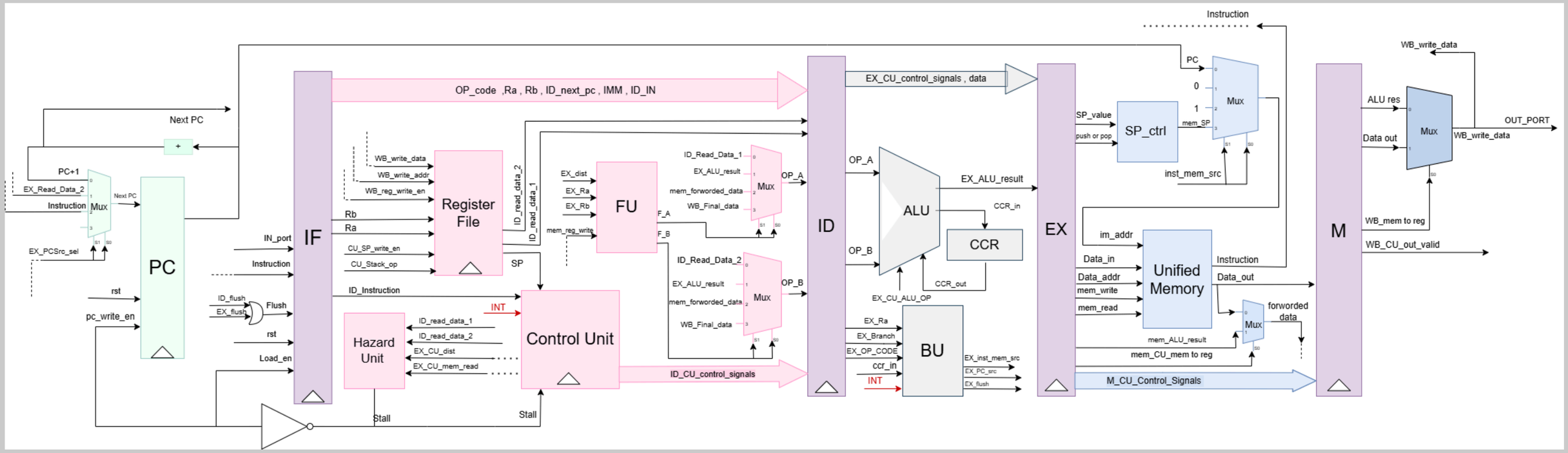



02

## Hazard Detection Unit



# Full Architecture

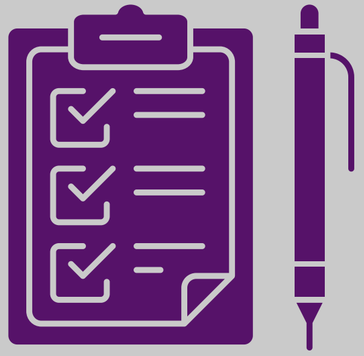


# Testing & Verification

- **General Instruction Set Testing**
- **Data Forwarding Testing**
- **Hazard Detection Testing**
- **Interrupt Handling Testing**

# Critical Edge Case Testing

## Test 1 Functionality



### Load-Use Hazard Detection

```
0xD1 :   LDD   R1, [R0]  ; Load data from memory
0x25 :   ADD   R1, R1
; Immediate use → load use hazard (stall)
```

### Multi-Stage Forwarding

```
0x25   ADD   R1, R1
; EX → ID forwarding
0x24   ADD   R1, R0
; MEM / WB →ID forwarding
```

### Dual-Source Forwarding

```
0x24   ADD   R1, R0
; Both ALU operands forwarded
```

### Flag-Based Branching

```
0x35   SUB   R1, R1      ; Sets Zero flag (Z = 1)
0x96   JZ    R2          ; Branch taken if Z = 1
0xC0   LDM   R0, #0xFF
; Branch target    executed
```

# Test 1 — Expected Results

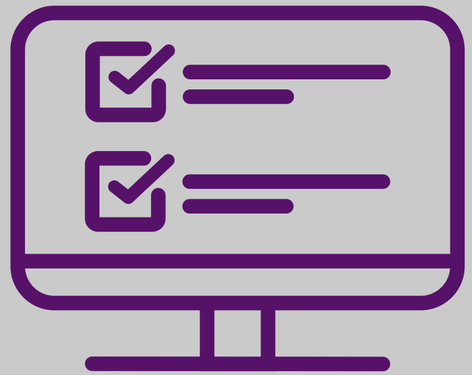
الله ينور

R0 = 0xFF ➤ Branch target reached

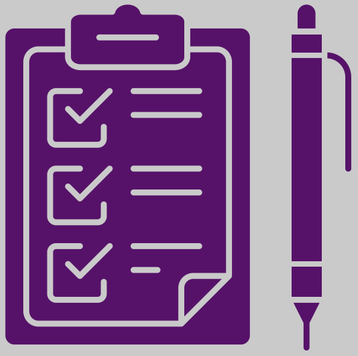
R1 = 0x00 ➤ Forwarding chain completed correctly

Zero Flag = 1 ➤ After SUB operation

Stall correctly inserted ➤ For load-use hazard



# Critical Edge Case Testing



## Test 2 Functionality

### Main Program: Test Flag Setting

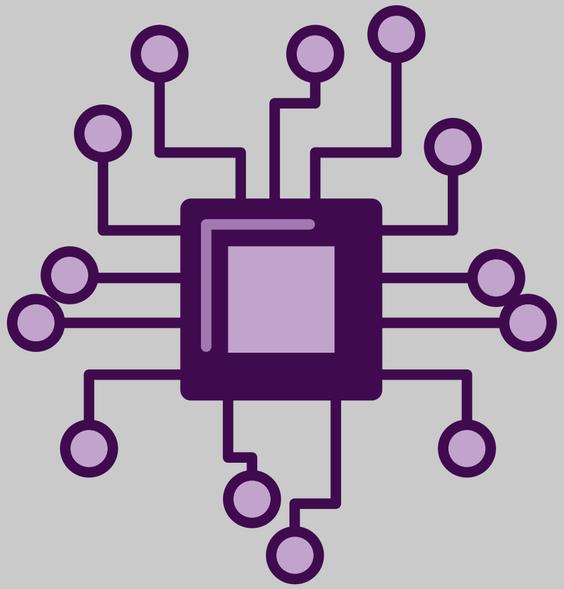
```
0xC0   LDM   R0, #0x7F
; Load 0x7F into R0
0xC1   LDM   R1, #0x01
 ; Load 0x01 into R1
0x21   ADD   R0, R1
 ; Add R1 to R0 → Sets V=1, N=1

0x00   NOP              ; No operation
0x00   NOP              ; No operation
0x00   NOP              ; No operation
, interrupt triggered here
0xC2   LDM   R2, #0xBB
; Load 0xBB into R2 (after RTI)
0x00   NOP              ; Continue execution
0x00   NOP              ; Continue execution
```

### Interrupt Service Routine: Test Nested CALL/RET

```
0xC0   LDM   R0, #0x51
; Load nested subroutine address into R0
0xB4   CALL  R0
; Call nested subroutine → pushes return
address to stack
0xC1   LDM   R1, #0xAA
; Load 0xAA into R1 after nested return
0xBC   RTI
; Return from interrupt → restores PC &
flags
```

# Nested Subroutine: Test Flag Modification & Return

```
0xC0   LDM   R0, #0xFF
 ; Load 0xFF into R0
0x88   INC   R0
 ; Increment R0 → modifies flags (Z=1, C=1)
0xB8   RET
 ; Return to ISR
```



# Test 2 — Expected Results

R1 = 0xAA ➤ ISR executed successfully

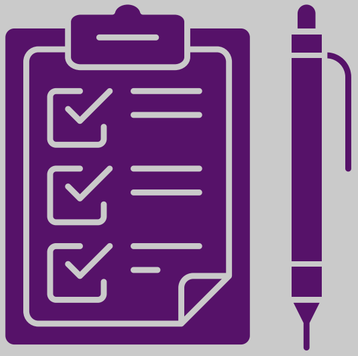
R2 = 0xBB ➤ Main program continued after RTI

SP = 0xFF ➤ Stack integrity maintained

Flags restored to pre-interrupt state

Nested CALL/RET return addresses handled correctly

# Critical Edge Case Testing



## Test 3 Functionality

### Stack Operations: Test PUSH

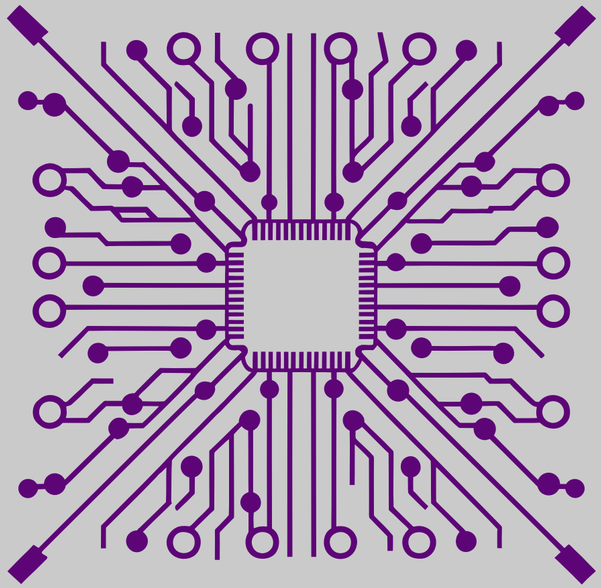
```
0xC0   LDM   R0, #0x11    ; Load 0x11 into R0 (initial value)
0xC1   LDM   R1, #0x22    ; Load 0x22 into R1 (initial value)
0xC2   LDM   R2, #0x33    ; Load 0x33 into R2 (initial value)
0x70   PUSH  R0           ; Push R0 → M[0xFF]=0x11, SP=0xFE
0x71   PUSH  R1           ; Push R1 → M[0xFE]=0x22, SP=0xFD
0x72   PUSH  R2           ; Push R2 → M[0xFD]=0x33, SP=0xFC
```

### Indirect Memory Access

```
0xC0   LDM   R0, #0xC0
 ; Load 0xC0 into R0 (address for indirect ops)
0xE2   STI   R0, R2
 ; Store R2 into memory pointed by R0 →
M[0xC0]=0x33
0xD1   LDI   R1, R0
 ; Load memory at address R0 into R1 → R1=0x33
```

## Multi-Source Forwarding

```
0x21   ADD   R0, R1
; ADD R0 + R1 → operands forwarded from EX/MEM/WB
0x21   ADD   R0, R1
 ; ADD R0 + R1 again (three-way dependency resolution)
```



## Stack Integrity Verification: Test POP

```
0x76   POP   R2
 ; Pop value into R2 → R2=0x33, SP=0xFD
0x75   POP   R1
 ; Pop value into R1 → R1=0x22, SP=0xFE
0x74   POP   R0
; Pop value into R0 → R0=0x11, SP=0xFF
0x20   ADD   R0, R0
 ; Final verification → check final R0 value
```

# Test 3 — Expected Results

M[0xC0] = 0x33 ➤ Indirect store successful

R0 = 0x22 ➤ Stack integrity maintained, final ADD operation

R1 = 0x22 ➤ Loaded via LDI, then popped

R2 = 0x33 ➤ Stack integrity maintained

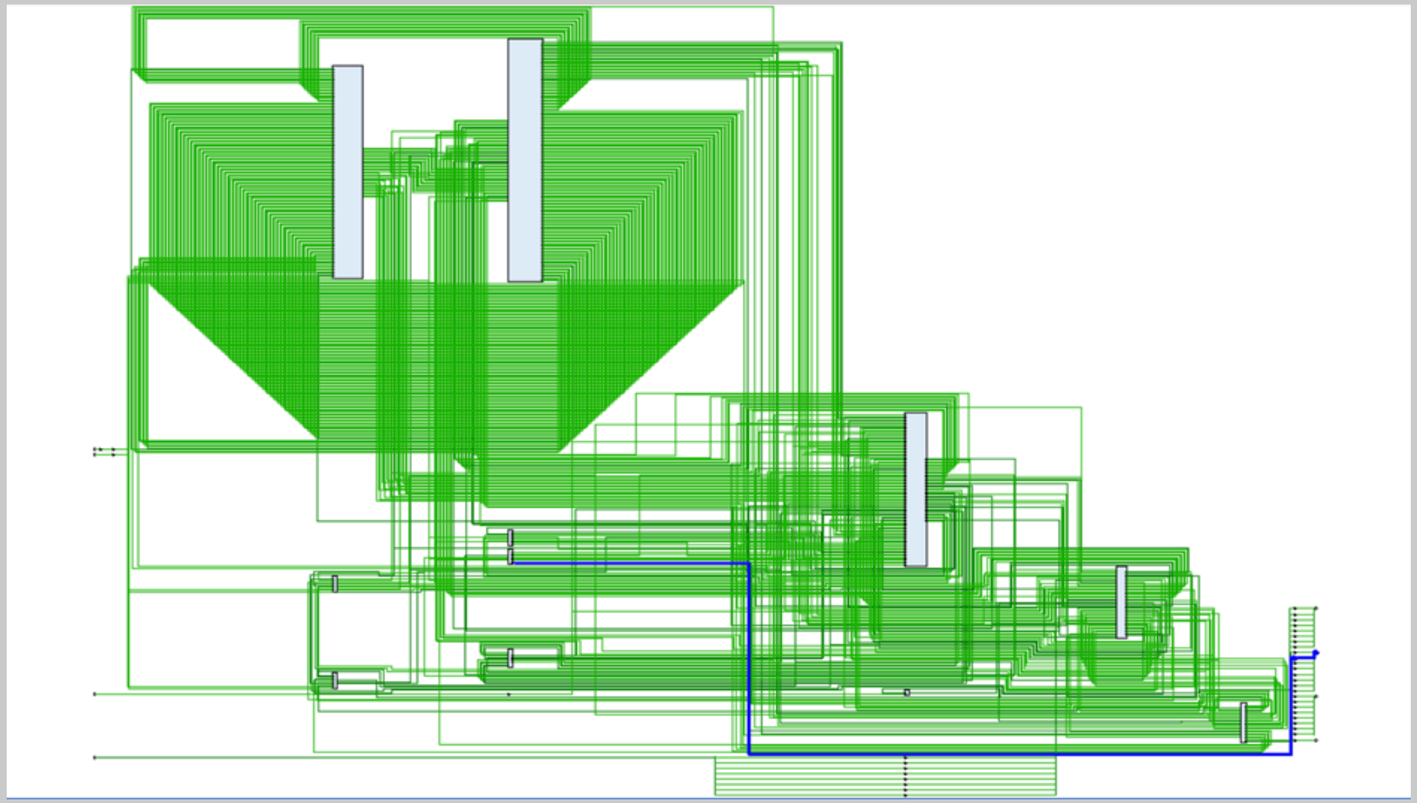
SP = 0xFF ➤ Returned to initial value

M[0xFF]=0x11, M[0xFE]=0x22, M[0xFD]=0x33 ➤ Stack memory preserved

Multi-source forwarding handled correctly

# Hardware Implementation Results

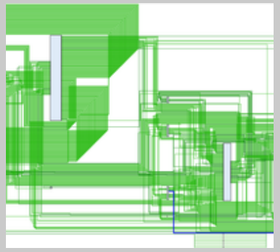
- Resource Utilization
- Timing  Analysis
- Power Analysis
- Synthesis Summary



Critical Path highlighted in the schematic

# Conclusion

01



**Implementation Success**

Fully functional 8-bit Von Neumann processor

5-stage pipeline Architecture

02


**Advanced Features**

Hardware-based hazard detection

Automated interrupt handling

03

**Verification & Testing**

100% testbench pass rate

All design objectives achieved

Cairo
University

# Thank You

## Presented By


- Nourhan Mohammad
- Huda Ehab
- Hagar Abdelkareem
- Mohammed Nasr
- Amr Hamdy
- Mohamad Abdallah