Laboratory #3: Pipelined Processor

Submission Date: April 4th 2025

CEG3156 - Computer Systems Design
Winter 2025

School of Electrical Engineering and Computer Science at the
University of Ottawa

Course Coordinator: Rami Abielmona, PhD, P. Eng
Teaching Assistant: Pavly Saleh

Nida Taj 300239050
August Zhang 300310509
Gavin Gao 300190846

Group 6

Table of Contents

Theoretical Part

Introduction

The objective of this lab was to be able to design, test and realize a pipelined processor and further design and implement a hazard detection unit along with other units such as a forwarding unit to handle and process pipeline hazards. The goal is to be able to design a datapath and realize the pipelined RISC processor in VHDL and finally simulate it on the board. Pipelined RISC processors are highly relevant to understand because they increase the efficiency and processing of instructions in modern day computing systems. The pipelined processor can process multiple instructions at once, significantly increasing throughput, which is crucial in many different computing applications such as embedded systems.

Pre – Lab

Hazard Detection Unit Equation:

```
If (ID/EX.MemRead & ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt =
IF/ID/RegisterRt)))
```

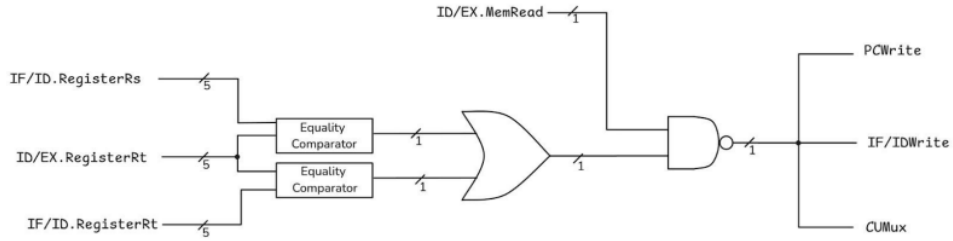

Figure 1.1 – Hazard detection unit

Forwarding Unit Equations:

- If (EX/MEM.RegWrite & (EX/MEM.RegisterRd ≠ 0) & (EX/MEM.RegisterRd=ID/EX.RegisterRs))
  ForwardA= 10
- If (EX/MEM.RegWrite & (EX/MEM.RegisterRd ≠ 0) & (EX/MEM.RegisterRd=ID/EX.RegisterRt))
  ForwardB= 10
- If (MEM/WB.RegWrite & (MEM/WB.RegisterRd ≠ 0) & (MEM/WB.RegisterRd=ID/EX.RegisterRs))
  ForwardA= 01
- If (MEM/WB.RegWrite & (MEM/WB.RegisterRd ≠ 0) & (MEM/WB.RegisterRd=ID/EX.RegisterRt))
  ForwardB= 01

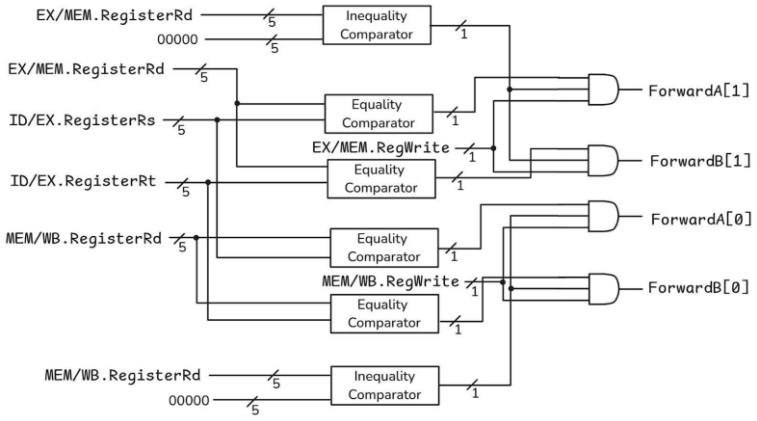

Figure 1.2 – Forwarding unit

First for Forwarding A (ALU Output A)

if (EX/MEM.RegWrite) AND (EX/MEM.RegisterRd ≠ 0) AND (EX/MEM.RegisterRd == ID/EX.RegisterRs) → ForwardA = 10
if (EX/MEM.RegWrite) AND (EX/MEM.RegisterRd ≠ 0) AND (EX/MEM.RegisterRd == ID/EX.RegisterRt) → ForwardB = 10

if (MEM/WB.RegWrite) AND (MEM/WB.RegisterRd ≠ 0) AND (MEM/WB.RegisterRd == ID/EX.RegisterRs) → ForwardA = 01
if (MEM/WB.RegWrite) AND (MEM/WB.RegisterRd ≠ 0) AND (MEM/WB.RegisterRd == ID/EX.RegisterRt) → ForwardB = 01

Else → 00 (no forwarding) // we will use value from regfil

Fwd value - mux - alu cs

EX/MEM.RegWrite	MEM/WB.RegWrite	EX/MEM.Rd == ID/EX.Rs	MEM/WB.Rd == ID/EX.Rs	EX/MEM.Rd == ID/EX.Rt	MEM/WB.Rd == ID/EX.Rt	ForwardA	ForwardB
1	X	1	X	0	0	10	00
1	X	0	X	1	X	00	10
0	1	1	X	0	0	01	00
0	1	0	X	1	X	00	01
0	0	0	0	0	0	00	00

Conditions to detect data hazard

EX/MEM.RegisterRd = ID/EX.RegisterRs ← type 1
EX/MEM.RegisterRd = ID/EX.RegisterRt ← type 1
MEM/WB.RegisterRd = ID/EX.RegisterRs ← type 2
MEM/WB.RegisterRd = ID/EX.RegisterRt ← type 2

datadep/hzd

If (ID/EX.MemRead) AND ((ID/EX.RegisterRt == IF/ID.RegisterRs) OR (ID/EX.RegisterRt == IF/ID.RegisterRt))
→ Then: Stall

If (ID/EX.MemRead) AND
((ID/EX.RegisterRt == IF/ID.RegisterRs)
OR (ID/EX.RegisterRt == IF/ID.RegisterRt))

stall

IO/EX MemRead	ID/EX RegRt	IF/IDReg Rs	IF/IDReg Rt	PCWrite	IF/IDWrite	MuxDetect
0	0	0	0	1	1	0
1	1	1	0	0	0	1
1	1	0	1	0	0	1
1	1	1	1	0	0	1

Figure 1.3 – Hazard detection conditions

Discussion of Problem

There are a few problems to be addressed over the course of this lab. The first being the successful design of both the hazard detection unit, and the forwarding unit. More specifically, the internals of these units, and successful Boolean realizations of both units. The next problem to be addressed is the pipelined processor itself. The single cycle datapath established during the previous laboratory will be amended in this laboratory to be a pipelined processor, which remains as an 8-bit datapath, that can process 32-bit instructions. The top-level entity of the processor should be able to take as input a 3-bit ValueSelect signal and output the correct values such as the PC value, ALU value, or read data values depending on the signal inputted.

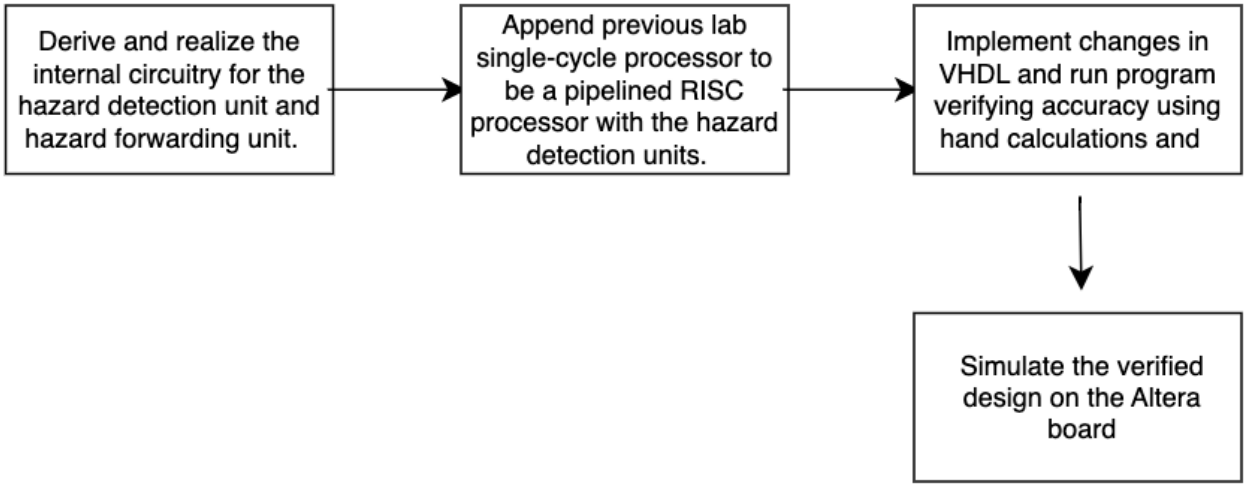

Figure 1.4 – Flowchart representation of proposed solution

Discussion of Algorithmic Solution

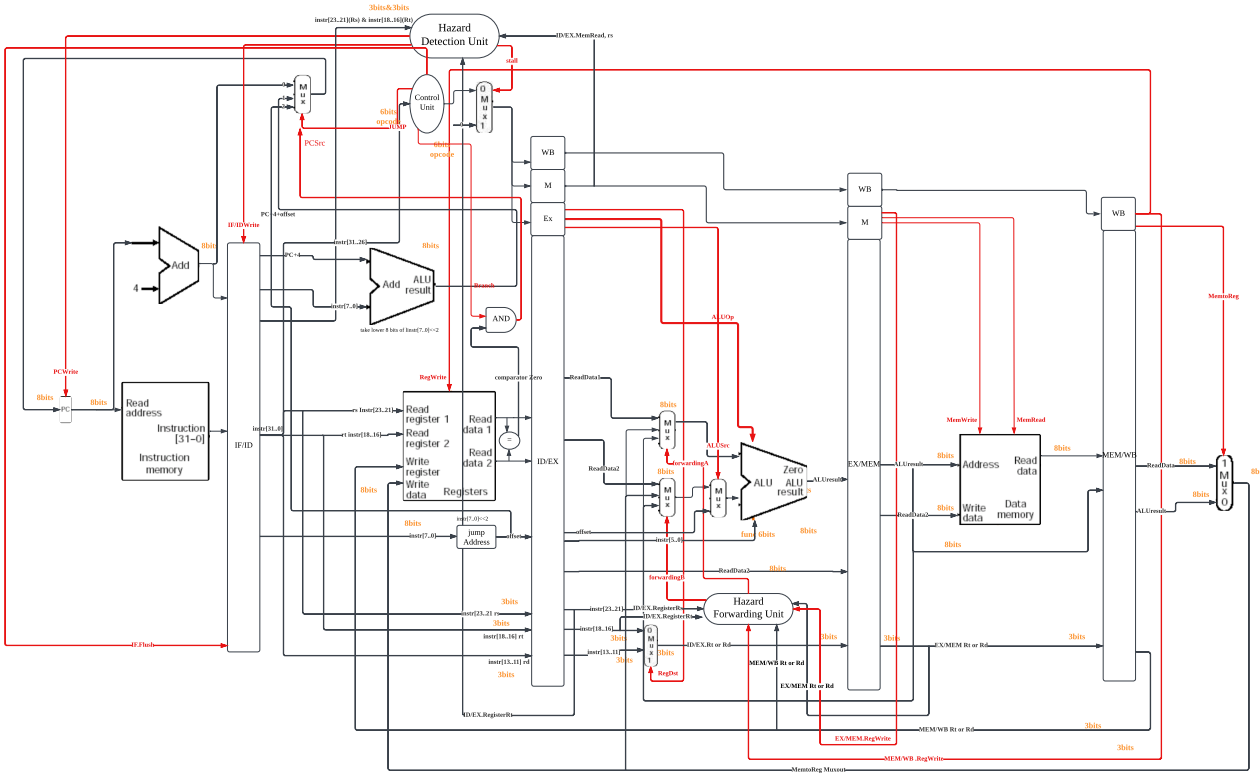

Figure 1.5 – Pipelined RISC processor 8-bit datapath

The figure 1.5 shows the RISC pipelined processor datapath. The datapath includes the five stages of the pipeline. The processor still follows an 8-bit datapath architecture, capable of handling 32-bit instructions. The datapath also includes the hazard forwarding and hazard detection units, which each take as input instr[23-21] in order to compare and detect hazards, and handle them in the appropriate manner, whether it be through flushing, stalling, or forwarding. This datapath compared to the single cycle processor also takes significantly more control signals, such as those for forwarding registers A and B as well as a stall signal and flush signal for the hazard detection. The PC in this datapath continues to take as input only 8-bits and output 32-bit instructions from the instruction memory. Furthermore, a branch can be detected in the initial stage of the pipeline to avoid branching hazards that may occur later on.

Design Part

Discussion of Used Components

Multiple components were used in the design of this pipelined processor:

PC – The program counter was used to fetch the current 8bit instruction address and take ss input the next 8bit instruction address and updates in the next rising clock cycle, sets to 00 when reset signal is on.

Instruction memory - Stores a total of 256 addresses that contain 32-bit instructions. In this lab, it contains a list of instructions from the verification part in the lab manual that must be fetched in sequence. Takes as input an 8-bit address and outputs a 32-bit instruction.

Pipelines – Being a five-stage pipeline, there is an IF/ID, ID/EX, EX/MEM, and MEM/WB pipeline. The IF/ID pipeline will take as input PC + 4 as well as the instruction to be executed, and if necessary, the flush signal is also sent to this pipeline. The ID/EX pipeline will take as the read data outputs from the register file as inputs, as well as the Jump address offset. This pipeline will output the read data and offset and instruction values into the execution stage. The Ex/MEM pipeline will receive the read data values and perform the necessary computations within its stage. The final pipeline will be the MEM/WB pipeline, which will take as input the read data from the data memory and ALU result.

ALU with shift - Calculate PC+4 + Offset address for branch instruction, by shift operand B(offset) to left by 2 bits then add with operand A(PC+4).

ALU - Performs different operations depending on the types of instruction. For lw and sw, performs A+B; for beq, perform A-B; for R type, based on 6bits funct, it performs Add, Subtract, And, OR, SLT(compare if A<B), or no operation in any other situations (output all 0s).

Adder +4 – Calculates the current instruction address + 4 and pass to Mux for deciding next instruction address.

Hazard detection Unit – the hazard detection unit takes 3 bits instr[23 - 21] from RS and 3 bits instr[18-16] from RT as well as ID/EX.MemRead all as input and processes it in order to detect whether or not a hazard will occur. This unit will then output a stall or flush control signal depending on the hazard and if one is detected.

Forwarding Unit – this unit is used to deal with hazards as well and performs a forwarding operation on A or B if a hazard is detected.

Data memory - using ram type, stores 256 registers that contain address of 8bits data. Initially holds addr 00 = 55 and addr 01 = AA values. The contents of the registers will be updated in the next rising clock cycle.

Comparator – this component is used to compare the read data outputs, in order to see if they

are the same values. The output of this is ANDed with the branch control signal and the output of that is set as one of the select values for a mux that will decide the address such as a jump.

Jump address - Calculates the target jump address by shifting the input address by 2 bits to the left.

Discussion of Actual Solution

All the pipelines and other units involving registers were designed using synch and enabled dflipflops. The design did not include a mux before the id/ex pipeline which is responsible for passing control signals and taking the stalling signal as input, since this caused errors that were not solvable and outputted incorrect results. The rest of the combinational units and registers worked as expected. The hazard detection units added new control signals to the datapath, not previously seen in the single cycle processor. The hazard detection unit was responsible for the flush and stall control signals, which performed their respective functions depending on whether a hazard was detected, such as a branch control hazard, or data hazard. The hazard forwarding unit added forwardingA and forwardingB control signals, which in the case of this lab helped solve a data hazard with the OR instruction from the provided benchmark code.

Discussion of Tool

In this lab, the main tool used was the Quartus II software. The Altera DE2 Board was not used in this lab. The Quartus II software was used to code the VHDL and design the hardware implemented in the pipelined processor datapath above. The verification and simulation of the VHDL code was performed using the waveform simulation functionality in the Quartus software itself.

Discussion of Challenging Problems

The major challenge faced during this lab was being able to implement a function that allows the processor to recover from stalling when a data hazard is detected. In the current design when a data hazard requiring a stall is detected, the processor performs the stall but is unable to stop stalling. The root cause of this is very likely due to a combinational loop or potential lower-level conflicts between the id/ex pipe and the detection unit. The stall signal is generated from the id/ex memRead and id/ex.RegisterRs values, which are derived from the id/ex pipe inputs, however this creates unstable behaviors for when the stall signal is read. Due to timing constraints, this issue could not be solved within the scope of this lab.

Real Implementation

Shown simulation/synthesis results

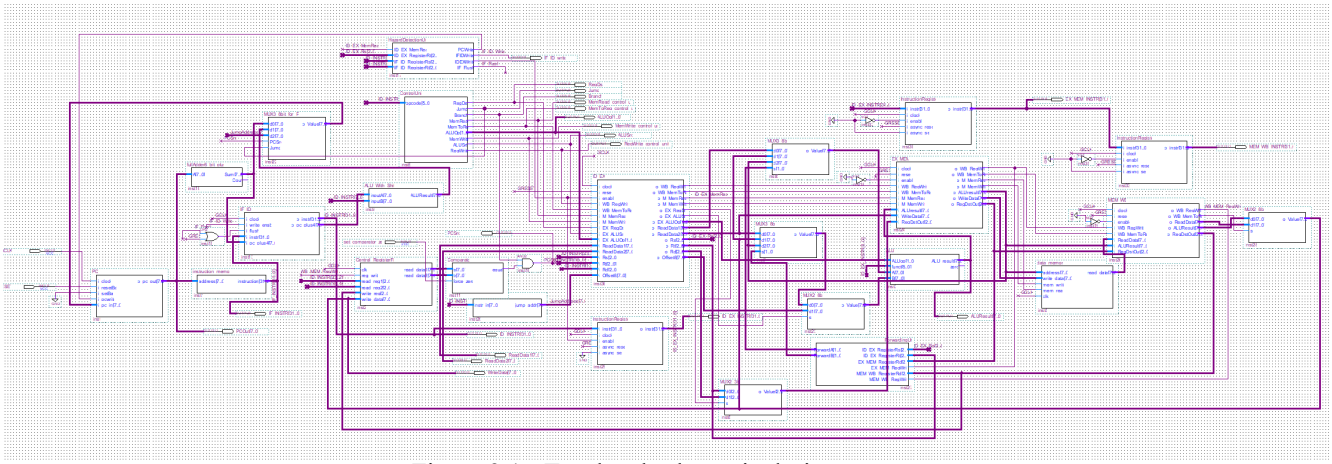

Figure 3.1 - Top-level schematic design

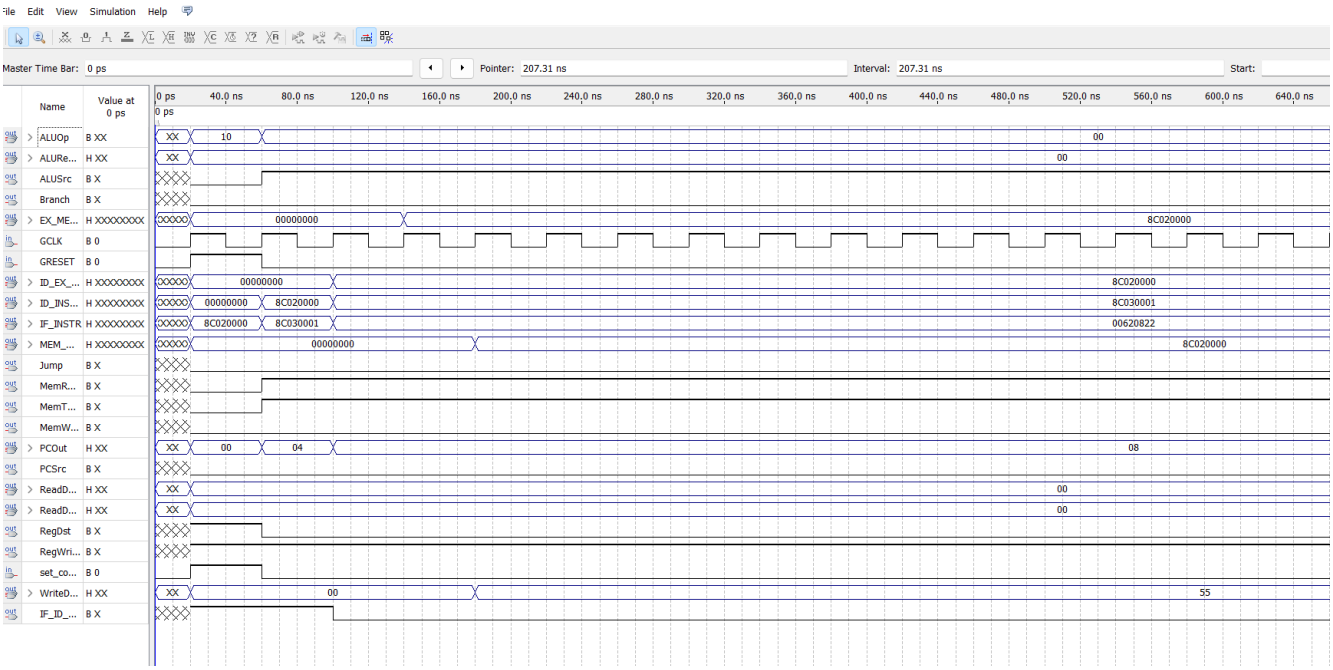

Figure 3.2 - Top-level Simulation result

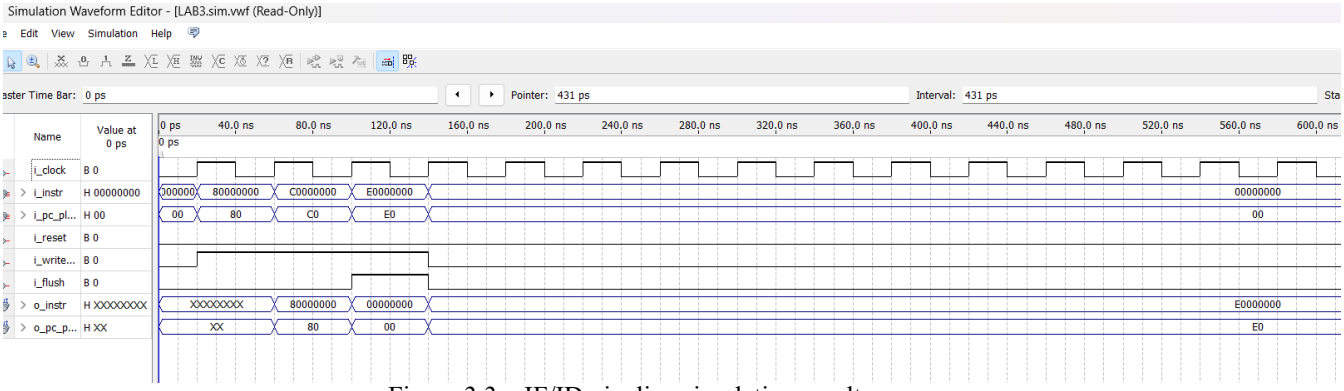

Figure 3.3 – IF/ID pipeline simulation result

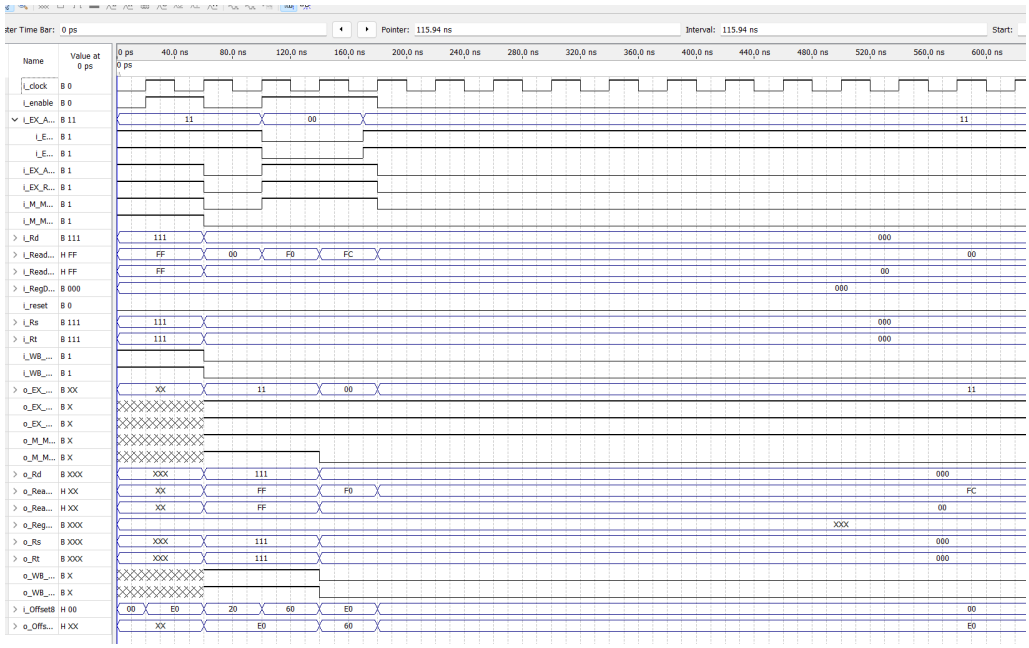

Figure 3.4 – ID/EX pipeline simulation result

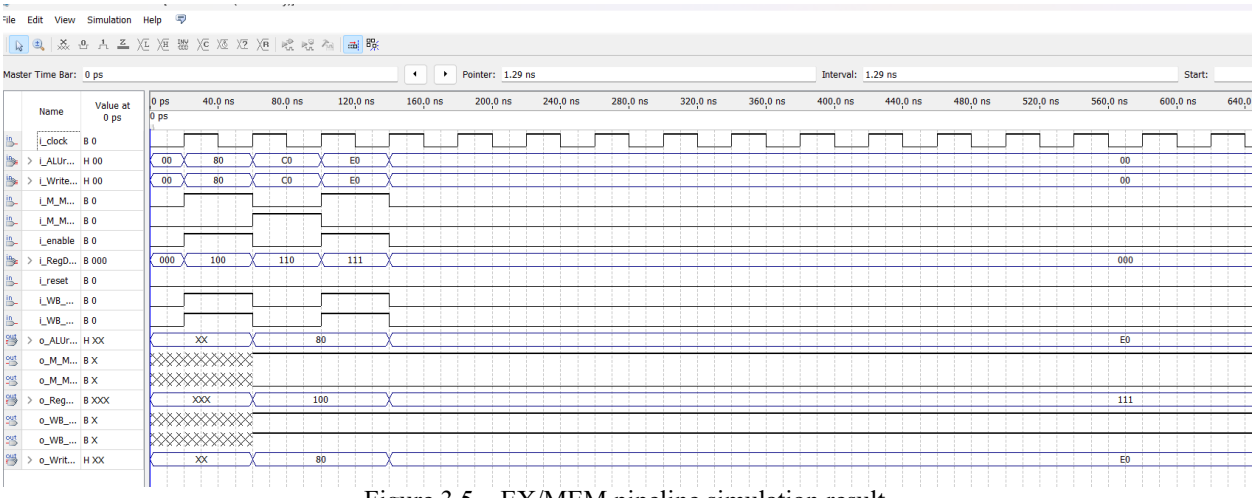

Figure 3.5 – EX/MEM pipeline simulation result

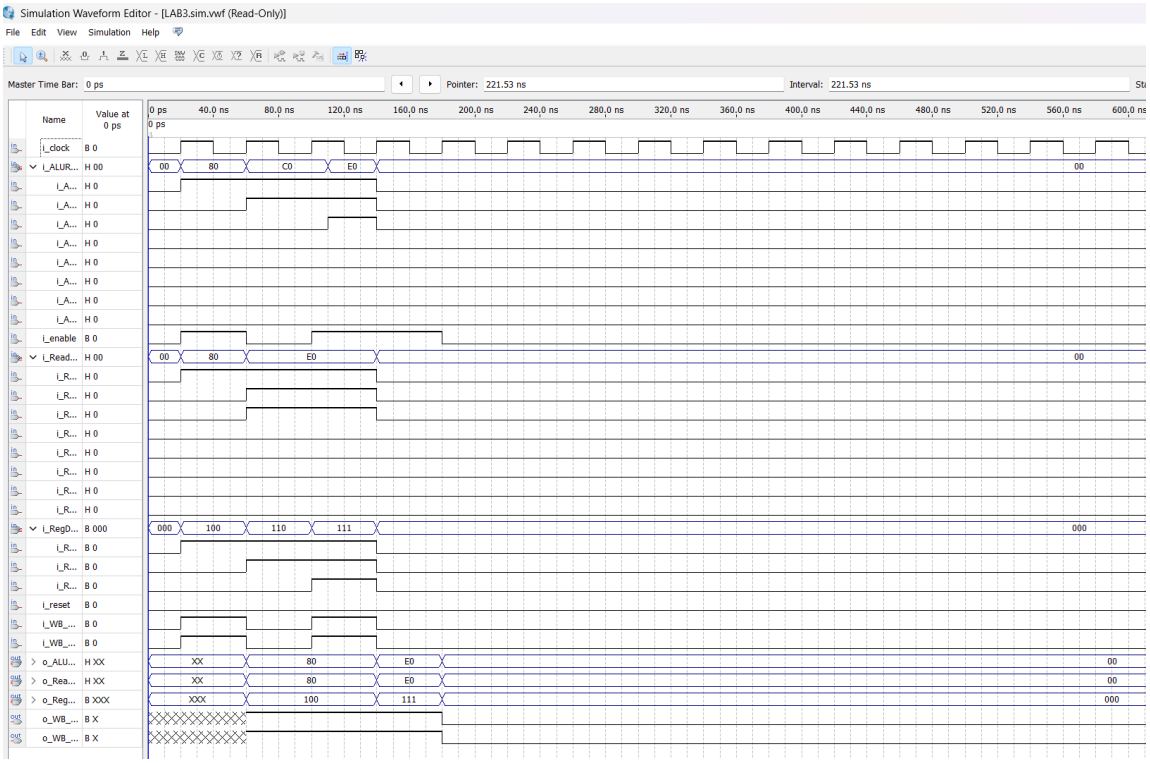

Figure 3.6 – MEM/WB pipeline simulation result

Verification

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
lw $2, 0	IF	ID	EX	MEM	WB												
lw $3, 1		IF	ID	EX	MEM	WB											
sub $1, $2, $3			IF	ID	EX	MEM	WB										
or $4, $1, $3				IF	ID	EX	MEM	WB									
%beq $1, $1, 20					IF	ID	EX	MEM	WB								
sw $4, 3						IF	ID	EX	MEM	WB							
add $1, $2, $3							IF	ID	EX	MEM	WB						
sw $1, 4								IF	ID	EX	MEM	WB					
lw $2, 3									IF	ID	EX	MEM	WB				
lw $3, 4										IF	ID	EX	MEM	WB			
j 11											IF	ID	EX	MEM	WB		
beq $1, $1, -44												IF	ID	EX	MEM	WB	
beq $1, $2, -8													IF	ID	EX	MEM	WB

Data hazard - SUB instruction requires values of register 2 and 3 in cycle 4, but these values are written in at cycle 5 and 6 - Stall would be the solution here
Data hazard - OR intrusion requires register 1 value in cycle 5, however register 1 is updated in cycle 7 - forwarding would be the solution here
Control hazard - Branch instruction in cycle 6 will always be true, since register 1 will always be equal to itself. Therefore branch must be taken, and because the instruction requires an offset of 20, next instruction will be lw $3, 4 therefore flush of the sw instruction right after the branch
Control hazard - Jump address will require jump to address 44, therefore the beq $1, $1, -44 address will need to be flushed.

Figure 3.7 – Hand calculation showing potential hazards in code execution

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
lw $2, 0	IF	ID	EX	MEM	WB												
lw $3, 1		IF	ID	EX	MEM	WB											
sub $1, $2, $3			bubble	bubble	IF	ID	EX →	MEM	WB								
or $4, $1, $3						IF	ID	→ EX	MEM	WB							
%beq $1, $1, 20							IF	ID	EX	MEM	WB						
sw $4, 3								nop	nop	nop	nop	nop					
lw $3, 4									IF	ID	EX	MEM	WB				
j 11											IF	ID	EX	MEM	WB		
beq $1, $1, -44												nop	nop	nop	nop	nop	
beq $1, $2, -8													IF	ID	EX	MEM	WB

Figure 3.8 – Hand calculation showing potential resolved hazards in code execution

Over the course of this lab, the datapath for a pipelined processor with hazard detection units was successfully derived. The datapath shown in figure 1.5 was then successfully translated into VHDL with a few bugs. The hazard detection units were also both realized in VHLD. The design was not simulated on the board; however, verification was performed using the waveform simulations along with the hand calculations derived for this lab. Unfortunately, the verification of the top-level entity waveform against the hand calculations shown in figure 3.8 was unsuccessful due to the stalling issue previously mentioned.

Discussion

8 bits×3 gates/bit=24 gates
24 gates×0.01ns/gate=0.24 ns
2 gate delays=2×0.01=0.02ns For Mux
2 mux so 0.04ns
1 gate delay=0.01ns (Simple Logic)
0.01ns For Zero Flag Logic
Worst case for ALU = Adder + 2 Mux _ Zero Flag Logic + Simple Logic = 0.04+0.24 +0.01 = **0.29ns**

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
Load Word (lw)	2ns	1ns	0.29	2ns	1ns	6.29
Store word (sw)	2ns	1ns	0.29	2ns		4.29
R-type (add, sub, OR, AND)	2ns	1ns	0.29		1ns	4.29
Branch (beq)	2ns	1ns	0.29			3.29
Jump	2ns	1ns	0.29			3.29

Table 1.1. - Single Cycle processor execution times

The single cycle processor from the previous lab had a worst-case clock cycle of 6.29ns. In this lab, because this is a pipelined processor, the worst-case clock cycle is 2ns in accordance with the fact that each instruction executes in stages, and the longest time taken by any one phase in the instruction execution is 2ns. In the single cycle processor, because the instructions execute one at a time, the processor must allow for the slowest instruction, which happened to be 6.29ns. Ultimately this means that the difference in execution time between the processors comes out to 4ns, making the pipelined processor significantly more efficient in execution time.

In terms of the solution for the pipelined processor itself, the datapath designed for this lab was implemented in VHLD, and all the components were successfully implemented as well. However, there were some challenges with the final pipelined processor design in VHDL. When the design was stimulated, only the first three instruction from the benchmark code were fetched correctly. Once the processor reached the sixth cycle, a data hazard was detected, requiring a stall, however once the processor stalled, it was unable to recover from that state, permanently stalling.

Conclusion

In conclusion, this lab provided a deeper understanding of the design and functionality of a pipelined processor. The lab also enhanced the understanding of the potential types of hazards that can be encountered in a pipelined processor, and how to resolve them. The datapath design in this lab as well as the hazard detection and forwarding units were all implemented in VHDL. Unfortunately, correct results were not obtained from the stimulated processor over the course of this lab

Appendix A

A-1 ALU VHDL Code

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY ALU IS
    PORT (
        ALUop      : IN std_logic_vector(1 DOWNTO 0);  -- ALU control signals from main control
        funct      : IN std_logic_vector(5 DOWNTO 0);  -- Function field from instruction
        A, B       : IN std_logic_vector(7 DOWNTO 0);  -- 8-bit ALU inputs
        ALU_result : OUT std_logic_vector(7 DOWNTO 0); -- 8-bit ALU result
        zero       : OUT std_logic                     -- Zero flag
    );
END ALU;

ARCHITECTURE behavioral OF ALU IS
    SIGNAL result : std_logic_vector(7 DOWNTO 0);
BEGIN

    PROCESS (ALUop, funct, A, B)
    BEGIN
        CASE ALUop IS
            WHEN "00" => -- For lw and sw (Addition)
                result <= A + B;

            WHEN "01" => -- For beq (Subtract)
                result <= A - B;

            WHEN "10" => -- R-type instructions
                CASE funct IS
                    WHEN "100000" => -- ADD
                        result <= A + B;

                    WHEN "100010" => -- SUBTRACT
                        result <= A - B;

                    WHEN "100100" => -- AND
                        result <= A AND B;

                    WHEN "100101" => -- OR
                        result <= A OR B;

                    WHEN "101010" => -- SET LESS THAN (SLT)
                        IF (A < B) THEN
                            result <= X"01";
                        ELSE
                            result <= X"00";
                        END IF;

                    WHEN OTHERS =>
                        result <= (OTHERS => '0');
                END CASE;

            WHEN OTHERS =>
                result <= (OTHERS => '0');
        END CASE;

        -- Set zero flag
        IF result = X"00" THEN
            zero <= '1';
        ELSE
            zero <= '0';
        END IF;

    END PROCESS;

    ALU_result <= result;

  END behavioral;
```

A-2 ALU with Shift VHDL Code

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY ALU_With_Shift IS
    PORT (
        inputA   : IN std_logic_vector(7 DOWNTO 0);  -- First ALU
operand (PC+4)
        inputB   : IN std_logic_vector(7 DOWNTO 0);  -- Second ALU
operand
        ALUResult: OUT std_logic_vector(7 DOWNTO 0)  -- Final ALU result
    );
END ALU_With_Shift;

ARCHITECTURE rtl OF ALU_With_Shift IS
    SIGNAL shifted_B : std_logic_vector(7 DOWNTO 0);
BEGIN
    -- Automatically shift inputB left by 2 before addition
    shifted_B <= inputB(5 DOWNTO 0) & "00";

    -- Perform addition (inputA + shifted_B)
    ALUResult <= inputA + shifted_B;
END rtl;
```

A-3 Comparator VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- Top-level 8-bit Comparator
ENTITY Comparator IS
    PORT(
        a     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        b     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        equal : OUT STD_LOGIC
    );
END Comparator;
ARCHITECTURE structural OF Comparator IS
    COMPONENT xnor_gate
        PORT(a, b : IN STD_LOGIC; y : OUT STD_LOGIC);
    END COMPONENT;
    COMPONENT and2_gate
        PORT(a, b : IN STD_LOGIC; y : OUT STD_LOGIC);
    END COMPONENT;
    SIGNAL x : STD_LOGIC_VECTOR(7 DOWNTO 0); -- outputs of xnor gates
    SIGNAL and1, and2_sig, and3, and4, and5, and6, and7 : STD_LOGIC;
BEGIN
    -- Instantiate 8 xnor gates
    x0: xnor_gate PORT MAP(a(0), b(0), x(0));
    x1: xnor_gate PORT MAP(a(1), b(1), x(1));
    x2: xnor_gate PORT MAP(a(2), b(2), x(2));
    x3: xnor_gate PORT MAP(a(3), b(3), x(3));
    x4: xnor_gate PORT MAP(a(4), b(4), x(4));
    x5: xnor_gate PORT MAP(a(5), b(5), x(5));
    x6: xnor_gate PORT MAP(a(6), b(6), x(6));
    x7: xnor_gate PORT MAP(a(7), b(7), x(7));
    -- AND reduction tree
    a1: and2_gate PORT MAP(x(0), x(1), and1);
    a2: and2_gate PORT MAP(x(2), x(3), and2_sig);
    a3: and2_gate PORT MAP(x(4), x(5), and3);
    a4: and2_gate PORT MAP(x(6), x(7), and4);
    a5: and2_gate PORT MAP(and1, and2_sig, and5);
    a6: and2_gate PORT MAP(and3, and4, and6);
    a7: and2_gate PORT MAP(and5, and6, equal);
END structural;
```

A-4 Data Memory VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY data_memory IS
    PORT (
        address  : IN  std_logic_vector(7 DOWNTO 0); -- 8-bit address for 256 locations
        write_data : IN  std_logic_vector(7 DOWNTO 0);
        read_data  : OUT std_logic_vector(7 DOWNTO 0);
        mem_write  : IN  std_logic;
        mem_read   : IN  std_logic;
        clk        : IN  std_logic
    );
END data_memory;
ARCHITECTURE rtl OF data_memory IS
    TYPE ram_type IS ARRAY (0 TO 255) OF std_logic_vector(7 DOWNTO 0);
    SIGNAL ram : ram_type := (
        -- Address 00: 55
        00 => X"55",
        -- Address 01: AA
        01 => X"AA",
        -- Fill remaining with 00
        OTHERS => X"00"
    );
BEGIN
    PROCESS (clk)
    BEGIN
        IF rising_edge(clk) THEN
            IF mem_write = '1' THEN
                ram(CONV_INTEGER(address)) <= write_data;
            END IF;
        END IF;
    END PROCESS;

    read_data <= ram(CONV_INTEGER(address)) WHEN mem_read = '1' ELSE (OTHERS => '0');
END rtl;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY data_memory IS
    PORT (
        address  : IN  std_logic_vector(7 DOWNTO 0); -- 8-bit address for 256 locations
        write_data : IN  std_logic_vector(7 DOWNTO 0);
        read_data  : OUT std_logic_vector(7 DOWNTO 0);
        mem_write  : IN  std_logic;
        mem_read   : IN  std_logic;
        clk        : IN  std_logic
    );
END data_memory;
ARCHITECTURE rtl OF data_memory IS
    TYPE ram_type IS ARRAY (0 TO 255) OF std_logic_vector(7 DOWNTO 0);
    SIGNAL ram : ram_type := (
        -- Address 00: 55
        00 => X"55",
        -- Address 01: AA
        01 => X"AA",
        -- Fill remaining with 00
        OTHERS => X"00"
    );
BEGIN
    PROCESS (clk)
    BEGIN
        IF rising_edge(clk) THEN
            IF mem_write = '1' THEN
                ram(CONV_INTEGER(address)) <= write_data;
            END IF;
        END IF;
    END PROCESS;

    read_data <= ram(CONV_INTEGER(address)) WHEN mem_read = '1' ELSE (OTHERS => '0');
END rtl;
```

A-5 EX_MEM VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY EX_MEM IS
    PORT (
        i_clock       : IN  STD_LOGIC;
        i_reset       : IN  STD_LOGIC;
        i_enable      : IN  STD_LOGIC;
        -- WB control signals
        i_WB_RegWrite  : IN  STD_LOGIC;
        i_WB_MemToReg  : IN  STD_LOGIC;
        -- M control signals
        i_M_MemRead    : IN  STD_LOGIC;
        i_M_MemWrite   : IN  STD_LOGIC;
        -- Data Inputs
        i_ALUresult    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        i_WriteData    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        i_RegDstOut    : IN  STD_LOGIC_VECTOR(2 DOWNTO 0); -- previously i_RegisterRd
        -- WB outputs
        o_WB_RegWrite  : OUT STD_LOGIC;
        o_WB_MemToReg  : OUT STD_LOGIC;
        -- M outputs
        o_M_MemRead    : OUT STD_LOGIC;
        o_M_MemWrite   : OUT STD_LOGIC;
        -- Data Outputs
        o_ALUresult    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        o_WriteData    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        o_RegDstOut    : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
    );
END EX_MEM;
BEGIN
    -- WB control signals
    wb_reg0 : OneBitRegister
        PORT MAP (
            i_input       => i_WB_RegWrite,
            i_enable      => i_enable,
            i_clock       => i_clock,
            i_async_reset => i_reset,
            i_async_set   => '0',
            o_q           => o_WB_RegWrite,
            o_qBar        => OPEN
        );
    wb_reg1 : OneBitRegister
        PORT MAP (
            i_input       => i_WB_MemToReg,
            i_enable      => i_enable,
            i_clock       => i_clock,
            i_async_reset => i_reset,
            i_async_set   => '0',
            o_q           => o_WB_MemToReg,
            o_qBar        => OPEN
        );
    -- M control signals
    m_reg0 : OneBitRegister
        PORT MAP (
            i_input       => i_M_MemRead,
            i_enable      => i_enable,
            i_clock       => i_clock,
            i_async_reset => i_reset,
            i_async_set   => '0',
            o_q           => o_M_MemRead,
            o_qBar        => OPEN
        );
    m_reg1 : OneBitRegister
        PORT MAP (
            i_input       => i_M_MemWrite,
            i_enable      => i_enable,
            i_clock       => i_clock,
            i_async_reset => i_reset,
            i_async_set   => '0',
            o_q           => o_M_MemWrite,
            o_qBar        => OPEN
        );
    -- ALU Result (8 bits)
    gen_ALU : FOR i IN 0 TO 7 GENERATE
        alu_reg : OneBitRegister
            PORT MAP (
                i_input       => i_ALUresult(i),
                i_enable      => i_enable,
                i_clock       => i_clock,
                i_async_reset => i_reset,
                i_async_set   => '0',
                o_q           => o_ALUresult(i),
                o_qBar        => OPEN
            );
    END GENERATE;
    -- Write Data (8 bits)
    gen_WriteData : FOR i IN 0 TO 7 GENERATE
        wd_reg : OneBitRegister
            PORT MAP (
                i_input       => i_WriteData(i),
                i_enable      => i_enable,
                i_clock       => i_clock,
                i_async_reset => i_reset,
                i_async_set   => '0',
                o_q           => o_WriteData(i),
                o_qBar        => OPEN
            );
    END GENERATE;
    -- RegDstOut (3 bits)
    gen_RegDst : FOR i IN 0 TO 2 GENERATE
        regdst_reg : OneBitRegister
            PORT MAP (
                i_input       => i_RegDstOut(i),
                i_enable      => i_enable,
                i_clock       => i_clock,
                i_async_reset => i_reset,
                i_async_set   => '0',
                o_q           => o_RegDstOut(i),
                o_qBar        => OPEN
            );
    END GENERATE;
END structural;
```

```
ARCHITECTURE structural OF EX_MEM IS
    COMPONENT OneBitRegister
        PORT (
            i_input       : IN  STD_LOGIC;
            i_enable      : IN  STD_LOGIC;
            i_clock       : IN  STD_LOGIC;
            i_async_reset : IN  STD_LOGIC;
            i_async_set   : IN  STD_LOGIC;
            o_q           : OUT STD_LOGIC;
            o_qBar        : OUT STD_LOGIC
        );
    END COMPONENT;
```

A-6 Forwarding Unit VHDL

```
ENTITY ForwardingUnit IS
    PORT (
        ID_EX_RegisterRs    : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        ID_EX_RegisterRt    : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        EX_MEM_RegisterRd   : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        EX_MEM_RegWrite     : IN  STD_LOGIC;
        MEM_WB_RegisterRd   : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        MEM_WB_RegWrite     : IN  STD_LOGIC;
        ForwardA            : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        ForwardB            : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
    );
END ForwardingUnit;

ARCHITECTURE rtl OF ForwardingUnit IS
BEGIN
    PROCESS (ID_EX_RegisterRs, ID_EX_RegisterRt,
             EX_MEM_RegisterRd, EX_MEM_RegWrite,
             MEM_WB_RegisterRd, MEM_WB_RegWrite)
    BEGIN
        -- Default: no forwarding
        ForwardA <= "00";
        ForwardB <= "00";

        -- EX hazard (EX/MEM.forward)
        IF (EX_MEM_RegWrite = '1') AND (EX_MEM_RegisterRd /= "000") THEN
            IF (EX_MEM_RegisterRd = ID_EX_RegisterRs) THEN
                ForwardA <= "10";
            END IF;
            IF (EX_MEM_RegisterRd = ID_EX_RegisterRt) THEN
                ForwardB <= "10";
            END IF;
        END IF;

        -- MEM hazard (MEM/WB.forward)
        IF (MEM_WB_RegWrite = '1') AND (MEM_WB_RegisterRd /= "000") THEN
            IF (MEM_WB_RegisterRd = ID_EX_RegisterRs) AND
               NOT(EX_MEM_RegWrite = '1' AND EX_MEM_RegisterRd = ID_EX_RegisterRs) THEN
                ForwardA <= "01";
            END IF;
            IF (MEM_WB_RegisterRd = ID_EX_RegisterRt) AND
               NOT(EX_MEM_RegWrite = '1' AND EX_MEM_RegisterRd = ID_EX_RegisterRt) THEN
                ForwardB <= "01";
            END IF;
        END IF;
    END PROCESS;
 END rtl;
```

A-7 Hazard Detection Unit VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY HazardDetectionUnit IS
    PORT (
        ID_EX_MemRead    : IN  STD_LOGIC;
        ID_EX_RegisterRt : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        IF_ID_RegisterRs : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        IF_ID_RegisterRt : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        PCWrite          : OUT STD_LOGIC;
        IFIDWrite        : OUT STD_LOGIC;
        IDEXWrite        : OUT STD_LOGIC;
        IF_Flush         : OUT STD_LOGIC
    );
END HazardDetectionUnit;

ARCHITECTURE rtl OF HazardDetectionUnit IS
BEGIN
    PROCESS (ID_EX_MemRead, ID_EX_RegisterRt, IF_ID_RegisterRs,
IF_ID_RegisterRt)
    BEGIN
        IF (ID_EX_MemRead = '1') AND
           ((ID_EX_RegisterRt = IF_ID_RegisterRs) OR
            (ID_EX_RegisterRt = IF_ID_RegisterRt)) THEN
            PCWrite   <= '0'; -- stall PC
            IFIDWrite <= '0'; -- stall IF/ID
            IDEXWrite <= '0'; -- stall ID/EX
            IF_Flush  <= '0'; -- don't flush yet
        ELSE
            PCWrite   <= '1';
            IFIDWrite <= '1';
            IDEXWrite <= '1';
            IF_Flush  <= '0';
        END IF;
    END PROCESS;
 END rtl;
```

A-8 Instruction Register VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY InstructionRegister IS
    PORT (
        i_instr       : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
        i_clock       : IN  STD_LOGIC;
        i_enable      : IN  STD_LOGIC;
        i_async_reset : IN  STD_LOGIC;
        i_async_set   : IN  STD_LOGIC;
        o_instr       : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END InstructionRegister;
ARCHITECTURE structural OF InstructionRegister IS
    COMPONENT OneBitRegister
        PORT (
            i_input       : IN  STD_LOGIC;
            i_enable      : IN  STD_LOGIC;
            i_clock       : IN  STD_LOGIC;
            i_async_reset : IN  STD_LOGIC;
            i_async_set   : IN  STD_LOGIC;
            o_q           : OUT STD_LOGIC;
            o_qBar        : OUT STD_LOGIC
        );
    END COMPONENT;
    SIGNAL dummy_qBar : STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
    -- Generate 32 one-bit registers
    gen_instr: FOR i IN 0 TO 31 GENERATE
        instr_bit: OneBitRegister
            PORT MAP (
                i_input       => i_instr(i),
                i_enable      => i_enable,
                i_clock       => i_clock,
                i_async_reset => i_async_reset,
                i_async_set   => i_async_set,
                o_q           => o_instr(i),
                o_qBar        => dummy_qBar(i)
            );
    END GENERATE;
 END structural;
```

A-9 ID_EX VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ID_EX IS
    PORT (
        i_clock       : IN  STD_LOGIC;
        i_reset       : IN  STD_LOGIC;
        i_enable      : IN  STD_LOGIC;

        -- WB control signals
        i_WB_RegWrite  : IN  STD_LOGIC;
        i_WB_MemToReg  : IN  STD_LOGIC;

        -- M control signals
        i_M_MemRead    : IN  STD_LOGIC;
        i_M_MemWrite   : IN  STD_LOGIC;

        -- EX control signals
        i_EX_RegDst    : IN  STD_LOGIC;
        i_EX_ALUSrc    : IN  STD_LOGIC;
        i_EX_ALUOp     : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);

        -- Data Inputs
        i_ReadData1    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        i_ReadData2    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        i_Rs           : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        i_Rt           : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        i_Rd           : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);

        i_Offset8      : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);  -- instr[7..0]

        -- WB outputs
        o_WB_RegWrite  : OUT STD_LOGIC;
        o_WB_MemToReg  : OUT STD_LOGIC;

        -- M outputs
        o_M_MemRead    : OUT STD_LOGIC;
        o_M_MemWrite   : OUT STD_LOGIC;

        -- EX outputs
        o_EX_RegDst    : OUT STD_LOGIC;
        o_EX_ALUSrc    : OUT STD_LOGIC;
        o_EX_ALUOp     : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);

        -- Data Outputs
        o_ReadData1    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        o_ReadData2    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        o_Rs           : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        o_Rt           : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        o_Rd           : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);

        o_Offset8      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END ID_EX;
```

```
ARCHITECTURE structural OF ID_EX IS
    COMPONENT OneBitRegister
        PORT (
            i_input       : IN  STD_LOGIC;
            i_enable      : IN  STD_LOGIC;
            i_clock       : IN  STD_LOGIC;
            i_async_reset : IN  STD_LOGIC;
            i_async_set   : IN  STD_LOGIC;
            o_q           : OUT STD_LOGIC;
            o_qBar        : OUT STD_LOGIC
        );
    END COMPONENT;

    SIGNAL dummy_qBar : STD_LOGIC_VECTOR(63 DOWNTO 0);
BEGIN
    -- WB control
    WB_RegWrite : OneBitRegister PORT MAP(i_WB_RegWrite, i_enable, i_clock, i_reset, '0',
o_WB_RegWrite, dummy_qBar(0));
    WB_MemToReg : OneBitRegister PORT MAP(i_WB_MemToReg, i_enable, i_clock, i_reset, '0',
o_WB_MemToReg, dummy_qBar(1));

    -- M control
    M_MemRead  : OneBitRegister PORT MAP(i_M_MemRead, i_enable, i_clock, i_reset, '0', o_M_MemRead,
dummy_qBar(2));
    M_MemWrite : OneBitRegister PORT MAP(i_M_MemWrite, i_enable, i_clock, i_reset, '0', o_M_MemWrite,
dummy_qBar(3));

    -- EX control
    EX_RegDst : OneBitRegister PORT MAP(i_EX_RegDst, i_enable, i_clock, i_reset, '0', o_EX_RegDst,
dummy_qBar(4));
    EX_ALUSrc : OneBitRegister PORT MAP(i_EX_ALUSrc, i_enable, i_clock, i_reset, '0', o_EX_ALUSrc,
dummy_qBar(5));
    EX_ALUOp0 : OneBitRegister PORT MAP(i_EX_ALUOp(0), i_enable, i_clock, i_reset, '0', o_EX_ALUOp(0),
dummy_qBar(6));
    EX_ALUOp1 : OneBitRegister PORT MAP(i_EX_ALUOp(1), i_enable, i_clock, i_reset, '0', o_EX_ALUOp(1),
dummy_qBar(7));

    -- ReadData1
    rd1: FOR i IN 0 TO 7 GENERATE
        reg1: OneBitRegister PORT MAP(
            i_input => i_ReadData1(i), i_enable => i_enable, i_clock => i_clock,
            i_async_reset => i_reset, i_async_set => '0', o_q => o_ReadData1(i), o_qBar => dummy_qBar(8
+ i)
        );
    END GENERATE;

    -- ReadData2
    rd2: FOR i IN 0 TO 7 GENERATE
        reg2: OneBitRegister PORT MAP(
            i_input => i_ReadData2(i), i_enable => i_enable, i_clock => i_clock,
            i_async_reset => i_reset, i_async_set => '0', o_q => o_ReadData2(i), o_qBar =>
dummy_qBar(16 + i)
        );
    END GENERATE;
```

```
    -- Rs, Rt, Rd
    rs: FOR i IN 0 TO 2 GENERATE
        reg_rs: OneBitRegister PORT MAP(
            i_input => i_Rs(i), i_enable => i_enable, i_clock => i_clock,
            i_async_reset => i_reset, i_async_set => '0', o_q => o_Rs(i), o_qBar => dummy_qBar(24 + i)
        );
    END GENERATE;

    rt: FOR i IN 0 TO 2 GENERATE
        reg_rt: OneBitRegister PORT MAP(
            i_input => i_Rt(i), i_enable => i_enable, i_clock => i_clock,
            i_async_reset => i_reset, i_async_set => '0', o_q => o_Rt(i), o_qBar => dummy_qBar(27 + i)
        );
    END GENERATE;

    rd: FOR i IN 0 TO 2 GENERATE
        reg_rd: OneBitRegister PORT MAP(
            i_input => i_Rd(i), i_enable => i_enable, i_clock => i_clock,
            i_async_reset => i_reset, i_async_set => '0', o_q => o_Rd(i), o_qBar => dummy_qBar(30 + i)
        );
    END GENERATE;


    -- Offset8
    offset: FOR i IN 0 TO 7 GENERATE
        reg_offset: OneBitRegister PORT MAP(
            i_input => i_Offset8(i), i_enable => i_enable, i_clock => i_clock,
            i_async_reset => i_reset, i_async_set => '0', o_q => o_Offset8(i), o_qBar => dummy_qBar(36
 + i)
        );
    END GENERATE;


  END structural;
```

A-10 IF_ID VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY IF_ID IS
    PORT (
        i_clock        : IN  STD_LOGIC;
        i_reset        : IN  STD_LOGIC;
        i_write_enable : IN  STD_LOGIC;
        i_flush        : IN  STD_LOGIC;

        -- Inputs
        i_instr        : IN  STD_LOGIC_VECTOR(31 DOWNTO 0); -- Fetched instruction
        i_pc_plus4     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);  -- PC + 4 value

        -- Outputs
        o_instr        : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        o_pc_plus4     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END IF_ID;
ARCHITECTURE structural OF IF_ID IS
    COMPONENT OneBitRegister
        PORT (
            i_input       : IN  STD_LOGIC;
            i_enable      : IN  STD_LOGIC;
            i_clock       : IN  STD_LOGIC;
            i_async_reset : IN  STD_LOGIC;
            i_async_set   : IN  STD_LOGIC;
            o_q           : OUT STD_LOGIC;
            o_qBar        : OUT STD_LOGIC
        );
    END COMPONENT;
    SIGNAL dummy_qBar : STD_LOGIC_VECTOR(39 DOWNTO 0); -- 32 bits for instr + 8 bits for pc+4
    -- Combined reset signal
    SIGNAL combined_reset : STD_LOGIC;
BEGIN
    -- Combine flush and reset
    combined_reset <= i_reset OR i_flush;
    -- Instruction bits (32)
    instr_regs: FOR i IN 0 TO 31 GENERATE
        instr_reg: OneBitRegister
            PORT MAP (
                i_input       => i_instr(i),
                i_enable      => i_write_enable,
                i_clock       => i_clock,
                i_async_reset => combined_reset,
                i_async_set   => '0',
                o_q           => o_instr(i),
                o_qBar        => dummy_qBar(i)
            );
    END GENERATE;

    -- PC + 4 bits (8)
    pc_regs: FOR i IN 0 TO 7 GENERATE
        pc_reg: OneBitRegister
            PORT MAP (
                i_input       => i_pc_plus4(i),
                i_enable      => i_write_enable,
                i_clock       => i_clock,
                i_async_reset => combined_reset,
                i_async_set   => '0',
                o_q           => o_pc_plus4(i),
                o_qBar        => dummy_qBar(i + 32)
            );
    END GENERATE;

 END structural;
```

A-11 Instruction Memory VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY instruction_memory IS
    PORT (
        address  : IN  std_logic_vector(7 DOWNTO 0); -- 8-bit address for 256 locations
        instruction : OUT std_logic_vector(31 DOWNTO 0)
    );
END instruction_memory;

ARCHITECTURE rtl OF instruction_memory IS
    TYPE rom_type IS ARRAY (0 TO 255) OF std_logic_vector(31 DOWNTO 0);
    CONSTANT rom : rom_type := (
    -- 0x00: lw $2, 0($0)
    0  => X"8C020000",
    -- 0x04: lw $3, 1($0)
    4  => X"8C030001",
    -- 0x08: sub $1, $3, $2
    8  => X"00620822",
    -- 0x0C: or $4, $1, $3
    12 => X"00232025",
    -- 0x10: inserted instr: beq $1, $1, 20 jump to address 36
    16 => X"10210004",
    -- 0x14: sw $4, 3($0)
    20 => X"AC040003",
    -- 0x18: add $1, $2, $3
    24 => X"00430820",
    -- 0x1C: sw $1, 4($0)
    28 => X"AC010004",
    -- 0x20: lw $2, 3($0)
    32 => X"8C020003",
    -- 0x24: lw $3, 4($0)
    36 => X"8C030004",
    -- 0x28: j 11
    40 => X"0800000B",
    -- 0x2C: beq $1, $1, -44 jump back to begining address 0
    44 => X"1021FFF4",
    -- 0x30: beq $1, $2, -8
    48 => X"1022FFFD",
    OTHERS => X"00000000"
);

BEGIN
    instruction <= rom(CONV_INTEGER(address));
 END rtl;
```

A-12 Jump Address VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY JumpAddress IS
    PORT (
        instr_in  : IN std_logic_vector(7 DOWNTO 0);  -- 8-bit jump target
from instruction

        jump_addr : OUT std_logic_vector(7 DOWNTO 0)  -- Final 8-bit jump
address
    );
END JumpAddress;

ARCHITECTURE structural OF JumpAddress IS
    SIGNAL shifted_addr : std_logic_vector(9 DOWNTO 0);
BEGIN
    -- Shift left by 2 (word alignment)
    shifted_addr <= instr_in & "00";

    -- Get lower 8 bits of shifted value
    jump_addr <= shifted_addr(7 DOWNTO 0);
END structural;
```

A-13 PC VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY PC IS
    PORT (
        i_clock       : IN  STD_LOGIC;
        i_resetBar    : IN  STD_LOGIC;
        i_setBar      : IN  STD_LOGIC;
        i_pcwrite     : IN  STD_LOGIC;                      -- PCWrite control
        i_pc_in       : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);   -- Next PC input
        o_pc_out      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)    -- Current PC output
    );
END PC;
ARCHITECTURE structural OF PC IS
    COMPONENT OneBitRegister
        PORT (
            i_input         : IN  STD_LOGIC;
            i_enable        : IN  STD_LOGIC;
            i_clock         : IN  STD_LOGIC;
            i_async_reset   : IN  STD_LOGIC;
            i_async_set     : IN  STD_LOGIC;
            o_q             : OUT STD_LOGIC;
            o_qBar          : OUT STD_LOGIC
        );
    END COMPONENT;
    SIGNAL dummy_qBar : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
    GEN_PC: FOR i IN 0 TO 7 GENERATE
        REG_BIT: OneBitRegister
            PORT MAP (
                i_input       => i_pc_in(i),
                i_enable      => i_pcwrite,
                i_clock       => i_clock,
                i_async_reset => i_resetBar,
                i_async_set   => i_setBar,
                o_q           => o_pc_out(i),
                o_qBar        => dummy_qBar(i)
            );
    END GENERATE;
 END structural;
```

A-14 MEM_WB VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY MEM_WB IS
    PORT (
        i_clock       : IN  STD_LOGIC;
        i_reset       : IN  STD_LOGIC;
        i_enable      : IN  STD_LOGIC;
        -- WB control signals
        i_WB_RegWrite : IN  STD_LOGIC;
        i_WB_MemToReg : IN  STD_LOGIC;
        -- Data inputs
        i_ReadData    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        i_ALUResult   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        i_RegDstOut   : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        -- WB control outputs
        o_WB_RegWrite : OUT STD_LOGIC;
        o_WB_MemToReg : OUT STD_LOGIC;
        -- Data outputs
        o_ReadData    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        o_ALUResult   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        o_RegDstOut   : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
    );
END MEM_WB;
ARCHITECTURE structural OF MEM_WB IS
    COMPONENT OneBitRegister
        PORT (
            i_input       : IN  STD_LOGIC;
            i_enable      : IN  STD_LOGIC;
            i_clock       : IN  STD_LOGIC;
            i_async_reset : IN  STD_LOGIC;
            i_async_set   : IN  STD_LOGIC;
            o_q           : OUT STD_LOGIC;
            o_qBar        : OUT STD_LOGIC
        );
    END COMPONENT;
    -- Dummy signals for o_qBar connections
    SIGNAL dummy_qBar : STD_LOGIC_VECTOR(19 DOWNTO 0);
BEGIN
    -- WB Control
    wb_regwrite: OneBitRegister
        PORT MAP (
            i_input       => i_WB_RegWrite,
            i_enable      => i_enable,
            i_clock       => i_clock,
            i_async_reset => i_reset,
            i_async_set   => '0',
            o_q           => o_WB_RegWrite,
            o_qBar        => dummy_qBar(0)
        );
    wb_memtoreg: OneBitRegister
        PORT MAP (
            i_input       => i_WB_MemToReg,
            i_enable      => i_enable,
            i_clock       => i_clock,
            i_async_reset => i_reset,
            i_async_set   => '0',
            o_q           => o_WB_MemToReg,
            o_qBar        => dummy_qBar(1)
        );
    -- Read Data
    rd0: OneBitRegister PORT MAP(i_input => i_ReadData(0), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ReadData(0), o_qBar => dummy_qBar(2));
    rd1: OneBitRegister PORT MAP(i_input => i_ReadData(1), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ReadData(1), o_qBar => dummy_qBar(3));
    rd2: OneBitRegister PORT MAP(i_input => i_ReadData(2), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ReadData(2), o_qBar => dummy_qBar(4));
    rd3: OneBitRegister PORT MAP(i_input => i_ReadData(3), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ReadData(3), o_qBar => dummy_qBar(5));
    rd4: OneBitRegister PORT MAP(i_input => i_ReadData(4), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ReadData(4), o_qBar => dummy_qBar(6));
    rd5: OneBitRegister PORT MAP(i_input => i_ReadData(5), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ReadData(5), o_qBar => dummy_qBar(7));
    rd6: OneBitRegister PORT MAP(i_input => i_ReadData(6), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ReadData(6), o_qBar => dummy_qBar(8));
    rd7: OneBitRegister PORT MAP(i_input => i_ReadData(7), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ReadData(7), o_qBar => dummy_qBar(9));
    -- ALU Result
    alu0: OneBitRegister PORT MAP(i_input => i_ALUResult(0), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ALUResult(0), o_qBar => dummy_qBar(10));
    alu1: OneBitRegister PORT MAP(i_input => i_ALUResult(1), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ALUResult(1), o_qBar => dummy_qBar(11));
    alu2: OneBitRegister PORT MAP(i_input => i_ALUResult(2), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ALUResult(2), o_qBar => dummy_qBar(12));
    alu3: OneBitRegister PORT MAP(i_input => i_ALUResult(3), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ALUResult(3), o_qBar => dummy_qBar(13));
    alu4: OneBitRegister PORT MAP(i_input => i_ALUResult(4), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ALUResult(4), o_qBar => dummy_qBar(14));
    alu5: OneBitRegister PORT MAP(i_input => i_ALUResult(5), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ALUResult(5), o_qBar => dummy_qBar(15));
    alu6: OneBitRegister PORT MAP(i_input => i_ALUResult(6), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ALUResult(6), o_qBar => dummy_qBar(16));
    alu7: OneBitRegister PORT MAP(i_input => i_ALUResult(7), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_ALUResult(7), o_qBar => dummy_qBar(17));
    -- RegDstOut
    r0: OneBitRegister PORT MAP(i_input => i_RegDstOut(0), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_RegDstOut(0), o_qBar => dummy_qBar(18));
    r1: OneBitRegister PORT MAP(i_input => i_RegDstOut(1), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_RegDstOut(1), o_qBar => dummy_qBar(19));
    r2: OneBitRegister PORT MAP(i_input => i_RegDstOut(2), i_enable => i_enable, i_clock => i_clock,
i_async_reset => i_reset, i_async_set => '0', o_q => o_RegDstOut(2), o_qBar => dummy_qBar(0)); --
wrapped reuse
 END structural;
```