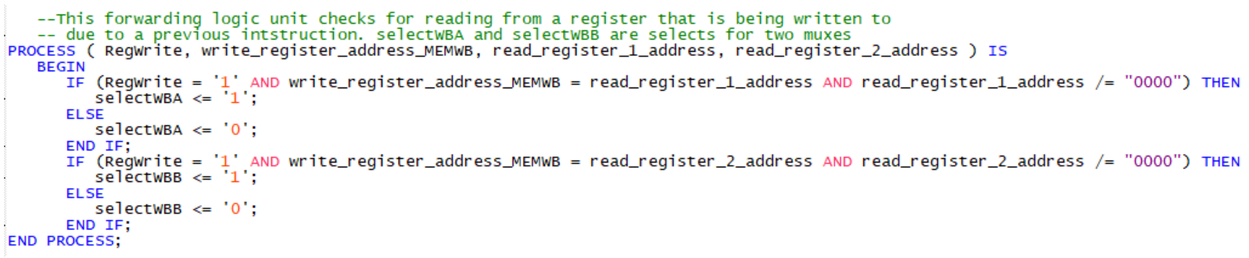
To: Dr. John Hutson
From: Jonathan Gusler
Date: 3/27/2019
Subject: Data Forwarding in the MIPS Processor

The previous lab, Lab 07, turned the single cycle processor used in prior labs into a pipelined processor. Pipelining introduces new problems, called hazards, that were addressed in this lab, particularly Data Hazards. To handle hazards, data is moved out of pipeline order so the data will be where it needs to be when it needs to be there. New hardware was added to detect hazards, route data, and select data inputs according to the specific hazard detected. This new hardware only detects hazards that affect AND, OR, SUB, and ADD instructions.

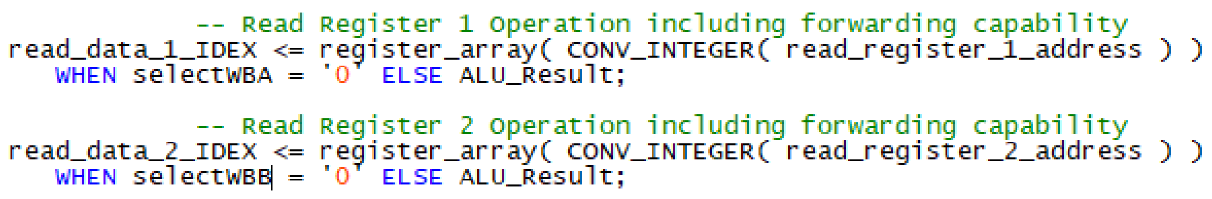
This lab builds continues from the processor used in Lab 7, [1]. This lab requires knowledge of pipelining and why and what kind of hazards are produced as a byproduct of pipelining a processor. This lab is based off Exercise #9 in Chapter 14 Section 12 of *Rapid Prototyping of Digital Systems, SOPC Edition* by Hamblen and Furman [2].

Two hazard detection units were implemented, each handling different hazards. One was put into the decode stage and detected the occurrence of reading from the same register that was being written to. The unit could change the data output from the decode stage to come from the result of an instruction 3 instructions ago. The code for the detection unit, *Figure 1*, and the multiplexers (muxes), *Figure 2*, are shown respectively.



```
   --This forwarding logic unit checks for reading from a register that is being written to
   -- due to a previous intstruction. selectWBA and selectWBB are selects for two muxes
PROCESS ( RegWrite, write_register_address_MEMWB, read_register_1_address, read_register_2_address ) IS
   BEGIN
      IF (RegWrite = '1' AND write_register_address_MEMWB = read_register_1_address AND read_register_1_address /= "0000") THEN
         selectWBA <= '1';
      ELSE
         selectWBA <= '0';
      END IF;
      IF (RegWrite = '1' AND write_register_address_MEMWB = read_register_2_address AND read_register_2_address /= "0000") THEN
         selectWBB <= '1';
      ELSE
         selectWBB <= '0';
      END IF;
END PROCESS;
```

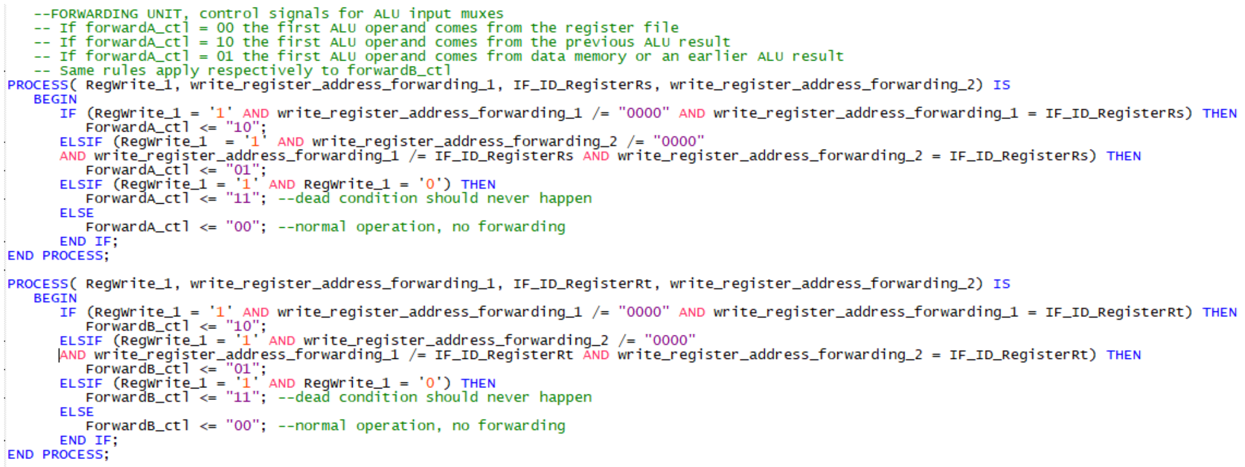
Figure 1: Detection unit in the decode stage outputting control signal to the muxes.



```
                 -- Read Register 1 Operation including forwarding capability
read_data_1_IDEX <= register_array( CONV_INTEGER( read_register_1_address ) )
   WHEN selectWBA = '0' ELSE ALU_Result;

                 -- Read Register 2 Operation including forwarding capability
read_data_2_IDEX <= register_array( CONV_INTEGER( read_register_2_address ) )
   WHEN selectWBB = '0' ELSE ALU_Result;
```

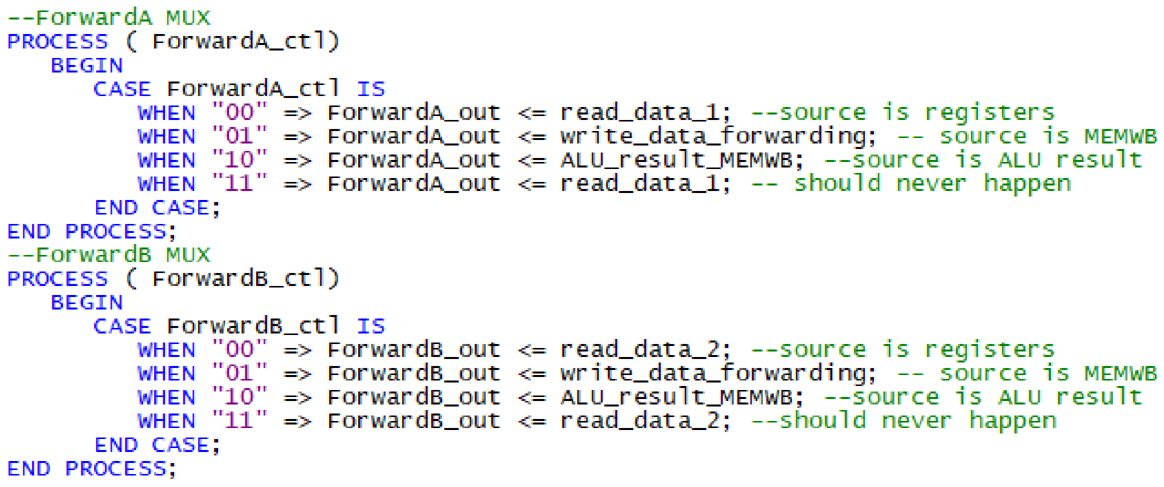
Figure 2: Code for muxes receiving control signal to select between registers and previous instruction result.

The second hazard detection unit was put into the execute stage, along with two muxes, one for each ALU input. This detection unit could change the data input to the ALU to either be from the decode stage, the previous instruction, or two instructions ago, depending on the specifics of the hazard detected. The code for the detection unit, *Figure 3*, and the multiplexers (muxes), *Figure 4*, are shown respectively.



```
   --FORWARDING UNIT, control signals for ALU input muxes
   -- If forwardA_ctl = 00 the first ALU operand comes from the register file
   -- If forwardA_ctl = 10 the first ALU operand comes from the previous ALU result
   -- If forwardA_ctl = 01 the first ALU operand comes from data memory or an earlier ALU result
   -- Same rules apply respectively to forwardB_ctl
PROCESS( RegWrite_1, write_register_address_forwarding_1, IF_ID_RegisterRs, write_register_address_forwarding_2) IS
   BEGIN
      IF (RegWrite_1 = '1' AND write_register_address_forwarding_1 /= "0000" AND write_register_address_forwarding_1 = IF_ID_RegisterRs) THEN
         ForwardA_ctl <= "10";
      ELSIF (RegWrite_1  = '1' AND write_register_address_forwarding_2 /= "0000"
      AND write_register_address_forwarding_1 /= IF_ID_RegisterRs AND write_register_address_forwarding_2 = IF_ID_RegisterRs) THEN
         ForwardA_ctl <= "01";
      ELSIF (RegWrite_1 = '1' AND RegWrite_1 = '0') THEN
         ForwardA_ctl <= "11"; --dead condition should never happen
      ELSE
         ForwardA_ctl <= "00"; --normal operation, no forwarding
      END IF;
END PROCESS;

PROCESS( RegWrite_1, write_register_address_forwarding_1, IF_ID_RegisterRt, write_register_address_forwarding_2) IS
   BEGIN
      IF (RegWrite_1 = '1' AND write_register_address_forwarding_1 /= "0000" AND write_register_address_forwarding_1 = IF_ID_RegisterRt) THEN
         ForwardB_ctl <= "10";
      ELSIF (RegWrite_1 = '1' AND write_register_address_forwarding_2 /= "0000"
      AND write_register_address_forwarding_1 /= IF_ID_RegisterRt AND write_register_address_forwarding_2 = IF_ID_RegisterRt) THEN
         ForwardB_ctl <= "01";
      ELSIF (RegWrite_1 = '1' AND RegWrite_1 = '0') THEN
         ForwardB_ctl <= "11"; --dead condition should never happen
      ELSE
         ForwardB_ctl <= "00"; --normal operation, no forwarding
      END IF;
END PROCESS;
```

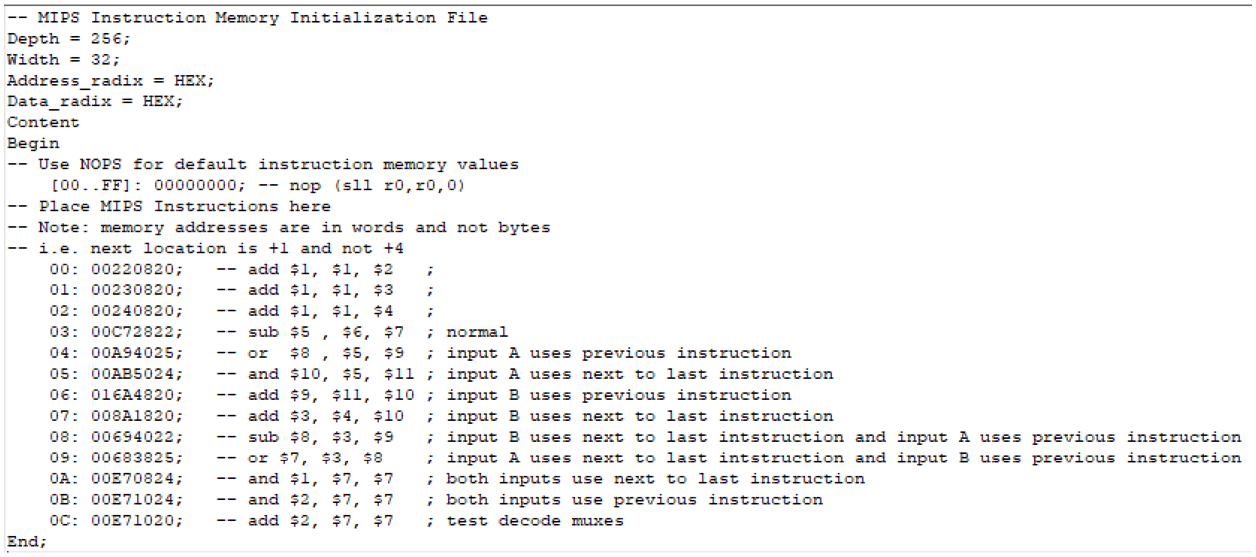
Figure 3: Detection unit in the execute stage outputting control signals to the muxes.



```
--ForwardA MUX
PROCESS ( ForwardA_ctl)
   BEGIN
      CASE ForwardA_ctl IS
         WHEN "00" => ForwardA_out <= read_data_1; --source is registers
         WHEN "01" => ForwardA_out <= write_data_forwarding; -- source is MEMWB
         WHEN "10" => ForwardA_out <= ALU_result_MEMWB; --source is ALU result
         WHEN "11" => ForwardA_out <= read_data_1; -- should never happen
      END CASE;
END PROCESS;
--ForwardB MUX
PROCESS ( ForwardB_ctl)
   BEGIN
      CASE ForwardB_ctl IS
         WHEN "00" => ForwardB_out <= read_data_2; --source is registers
         WHEN "01" => ForwardB_out <= write_data_forwarding; -- source is MEMWB
         WHEN "10" => ForwardB_out <= ALU_result_MEMWB; --source is ALU result
         WHEN "11" => ForwardB_out <= read_data_2; --should never happen
      END CASE;
END PROCESS;
```

Figure 4: Code for ALU input muxes selecting data from different pipeline stages.

To test the new hardware, a new program, *Figure 5*, was created involving only AND, SUB, OR, and ADD instructions that covers all the hazard possibilities that the current hardware can detect.



```
-- MIPS Instruction Memory Initialization File
Depth = 256;
Width = 32;
Address_radix = HEX;
Data_radix = HEX;
Content
Begin
-- Use NOPS for default instruction memory values
    [00..FF]: 00000000; -- nop (sll r0,r0,0)
-- Place MIPS Instructions here
-- Note: memory addresses are in words and not bytes
-- i.e. next location is +1 and not +4
    00: 00220820;    -- add $1, $1, $2    ;
    01: 00230820;    -- add $1, $1, $3    ;
    02: 00240820;    -- add $1, $1, $4    ;
    03: 00C72822;    -- sub $5 , $6, $7   ; normal
    04: 00A94025;    -- or  $8 , $5, $9   ; input A uses previous instruction
    05: 00AB5024;    -- and $10, $5, $11  ; input A uses next to last instruction
    06: 016A4820;    -- add $9, $11, $10  ; input B uses previous instruction
    07: 008A1820;    -- add $3, $4, $10   ; input B uses next to last instruction
    08: 00694022;    -- sub $8, $3, $9    ; input B uses next to last intstruction and input A uses previous instruction
    09: 00683825;    -- or $7, $3, $8     ; input A uses next to last intstruction and input B uses previous instruction
    0A: 00E70824;    -- and $1, $7, $7    ; both inputs use next to last instruction
    0B: 00E71024;    -- and $2, $7, $7    ; both inputs use previous instruction
    0C: 00E71020;    -- add $2, $7, $7    ; test decode muxes
End;
```

Figure 5: New program to test all hazard possibilities.

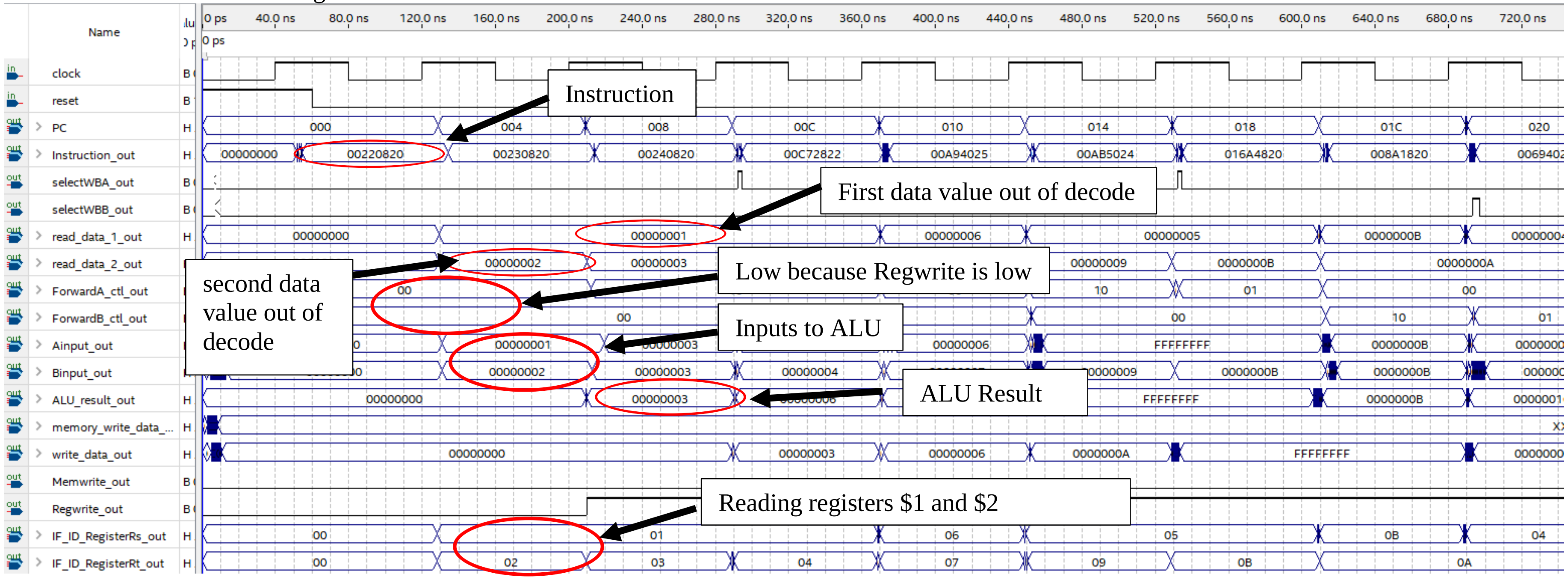
Error! Reference source not found. describes each of the relevant signals in the following analyses. An analysis of each instruction is shown starting after **Error! Reference source not found.**.

Signal Name	Descriptor
Clock	Clock signal, 80ns period
Reset	Reset signal to start system
PC	Program Counter
Instruction	MIPS instruction in hex
SelectWBA	Mux control signal for data 1 output from decode
SelectWBB	Mux control signal for data 2 output from decode
Read data 1	Data 1 output from decode to execute
Read data 2	Data 1 output from decode to execute
ForwardA ctl	Mux control signal for ALU input 1
ForwardB ctl	Mux control signal for ALU input 2
Ainput	Mux output into ALU input A
Binput	Mux output into ALU input B*
ALU result	ALU result
Write data	Data being written to register in decode
Regwrite	Regwrite signal to allow register writing
IF ID RegisterRs	MIPS instruction component Rs
IF ID RegisterRt	MIPS instruction component Rt

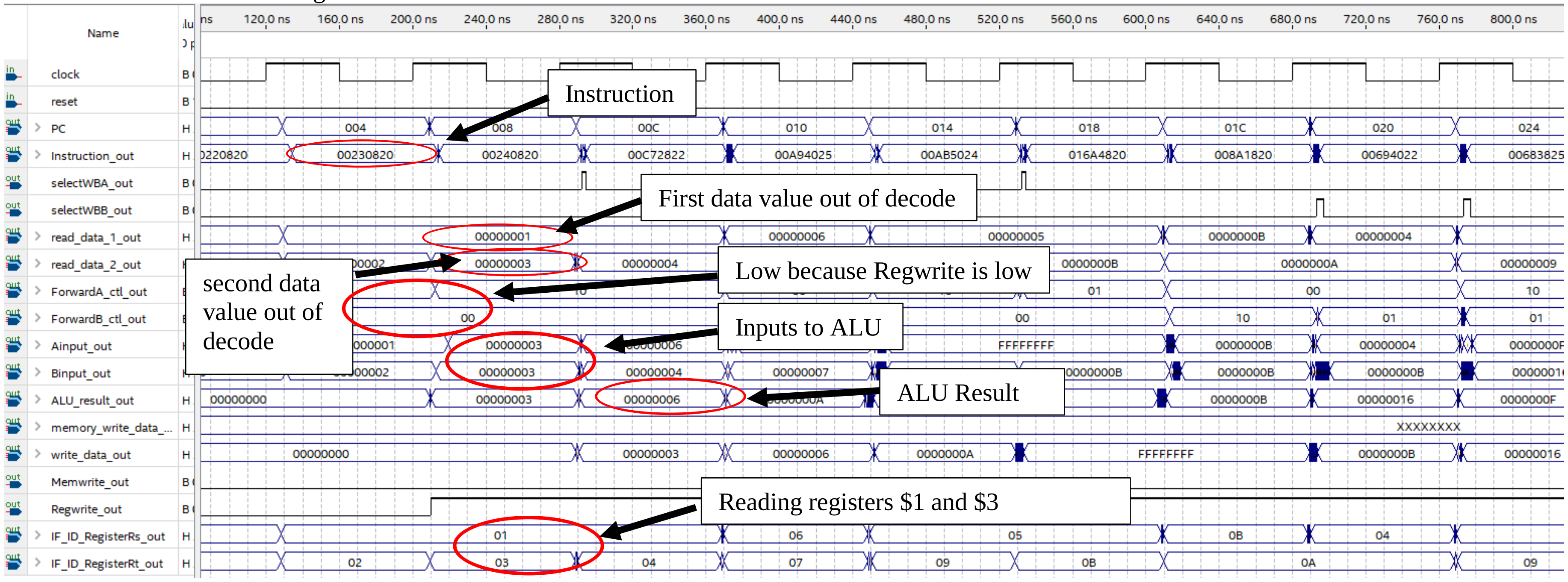
*This signal goes into another mux that selects between Binput and Sign extend which then goes into the ALU

Table 1: Relevant signals for hardware simulation.

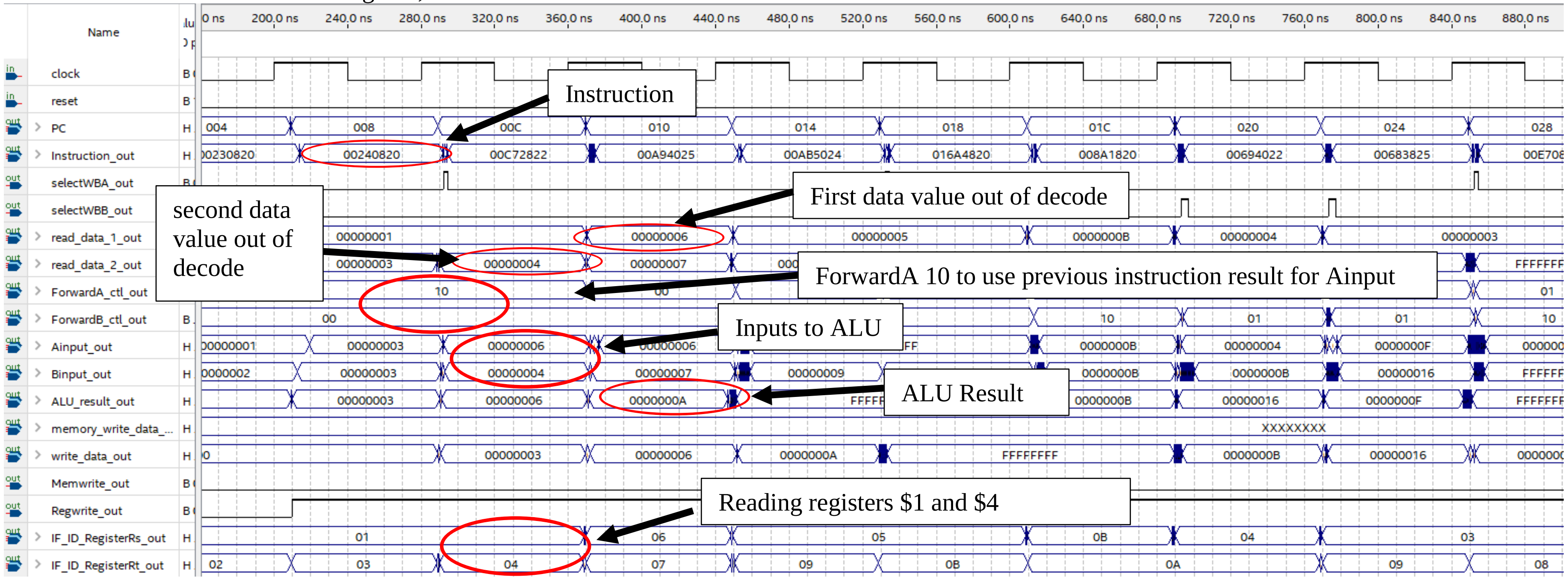
Instruction: add $1, $1, $2
Hazard: No hazard because RegWrite is low.



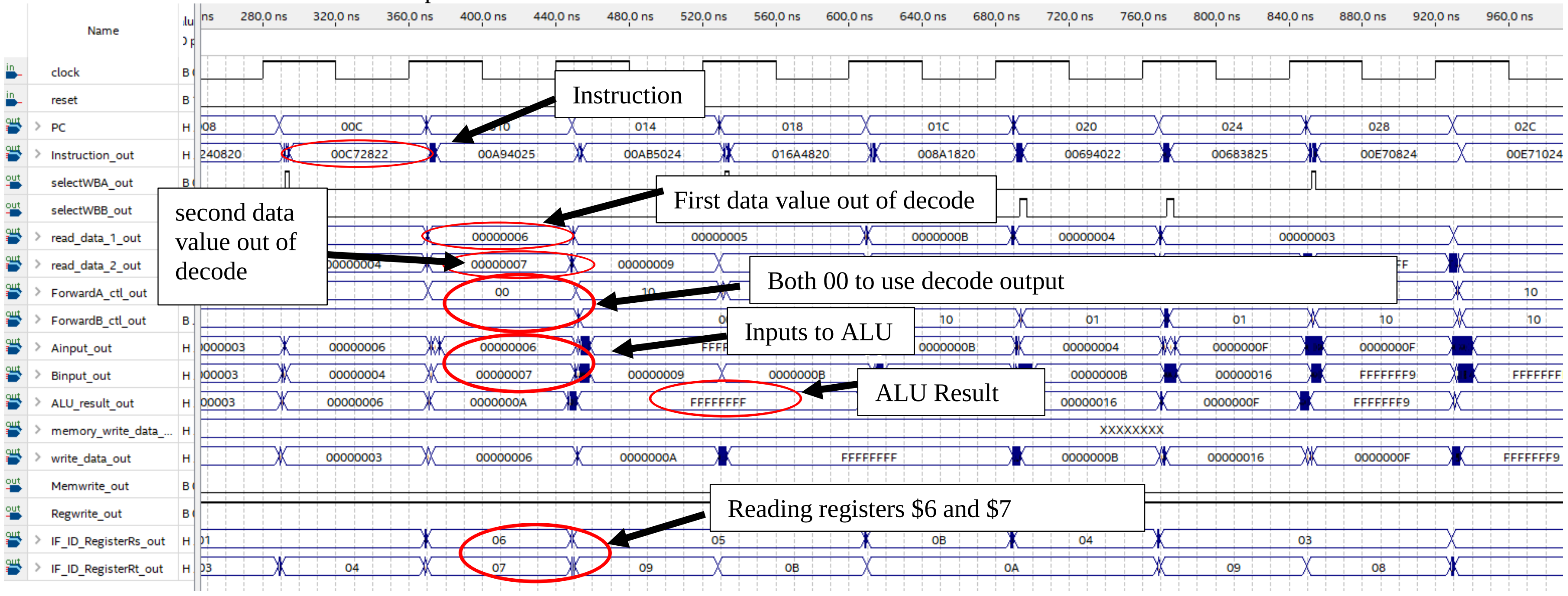
Instruction: add $1, $1, $3
Hazard: No hazard because Regwrite is low.



Instruction: add $1, $1, $4
Hazard: Read and write are same register, forward write data to read data.

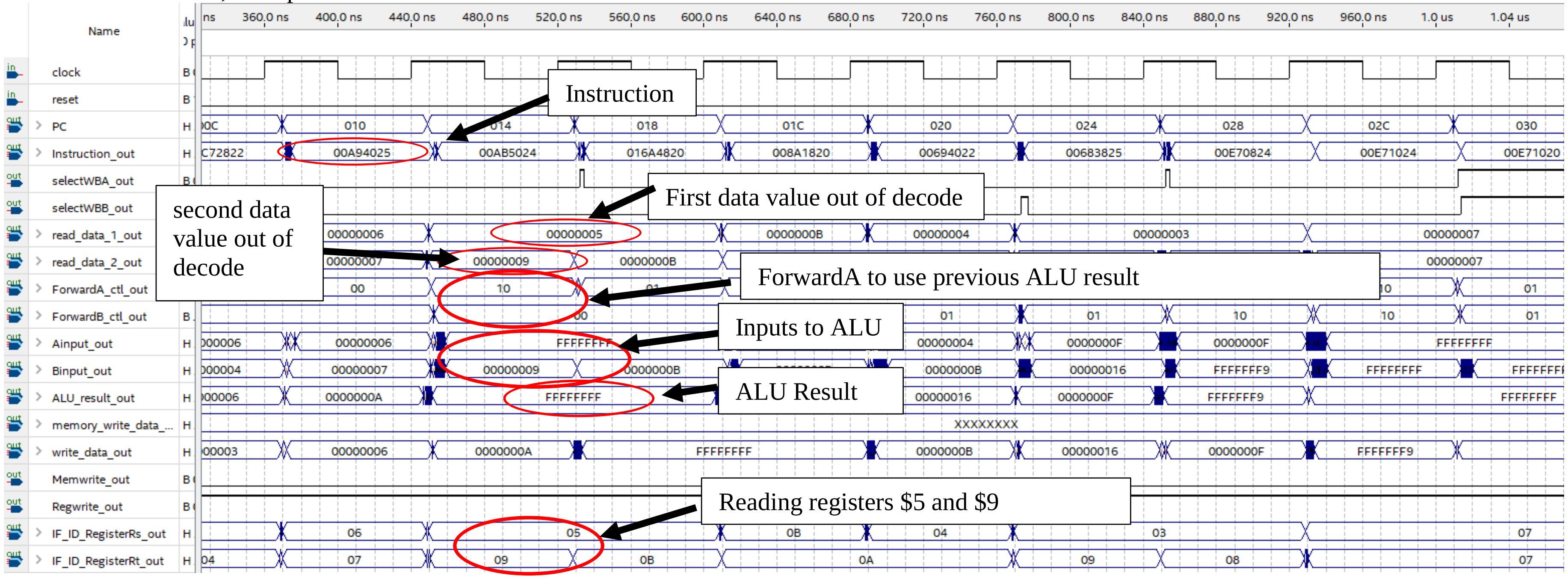


Instruction: sub $5, $6, $7
Hazard: No hazard because doesn't need previous instructions' results.



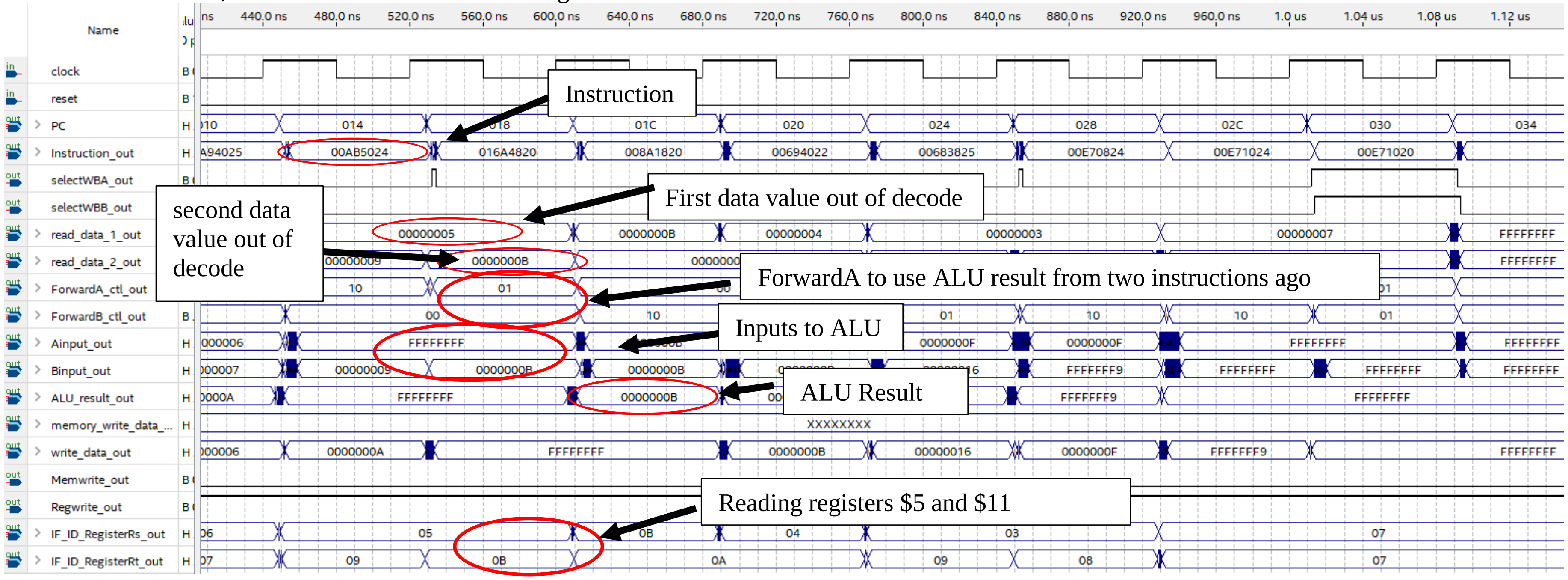
Instruction: or $8, $5, $9

Hazard: Hazard in Rs, need previous instruction result.



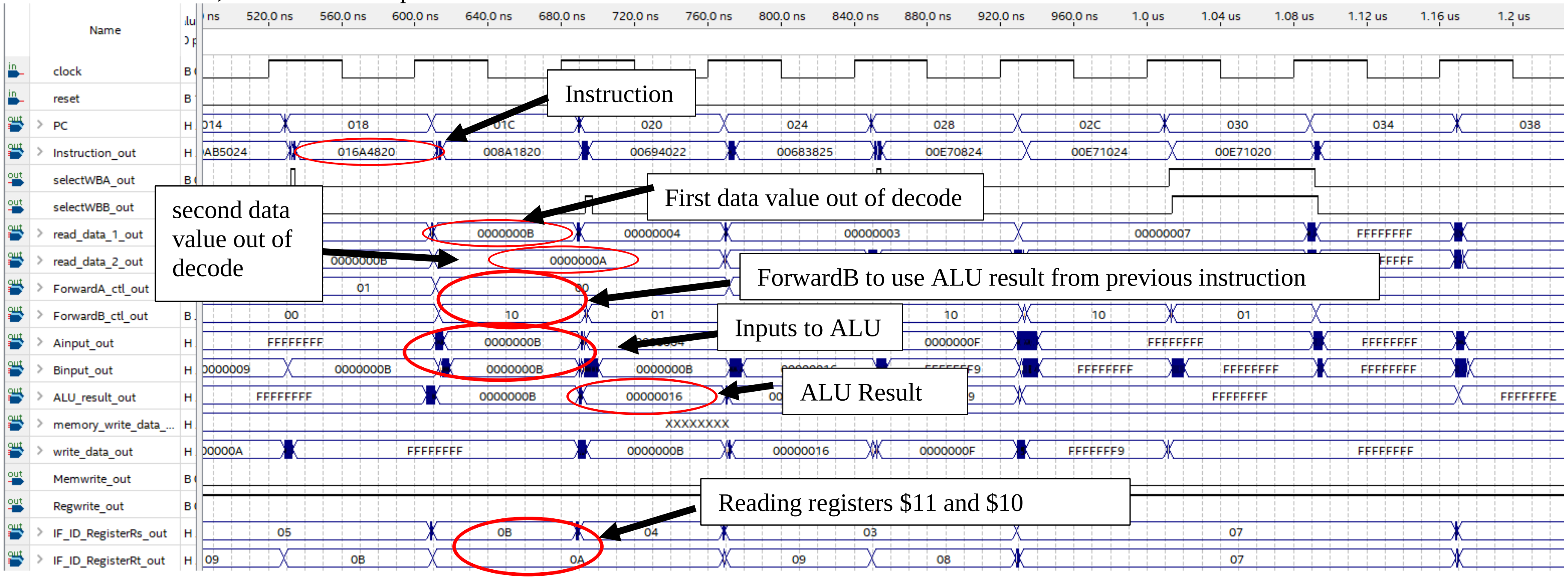
Instruction: and $10, $5, $11

Hazard: Hazard in Rs, need result from 2 instructions ago.



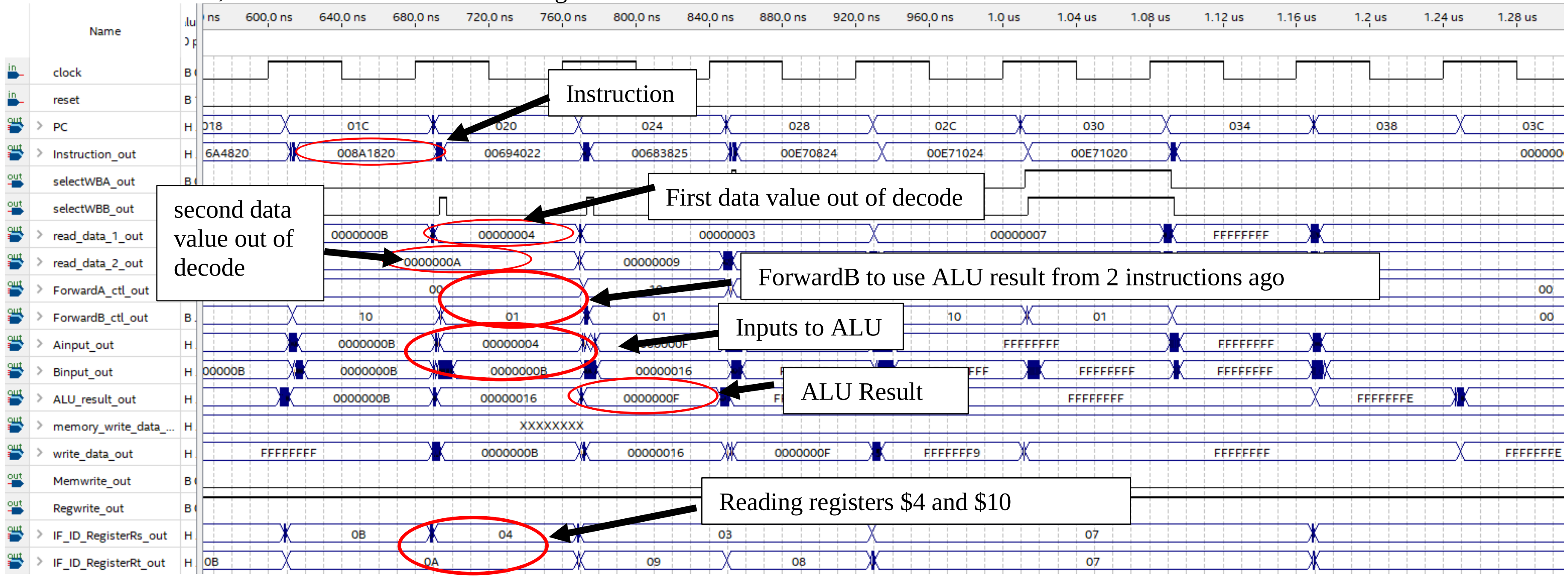
Instruction: add $9, $11, $10

Hazard: Hazard in Rt, need result from previous instruction.



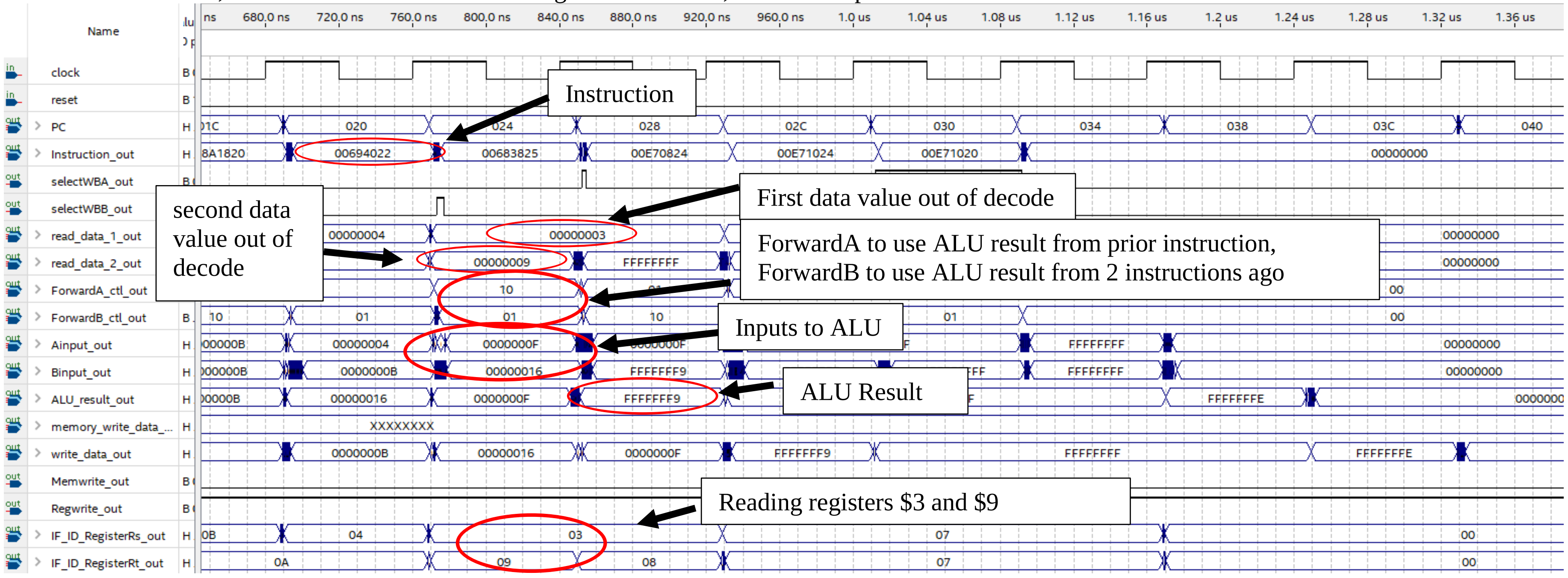
Instruction: add $3, $4, $10

Hazard: Hazard in Rt, need result from 2 instructions ago.



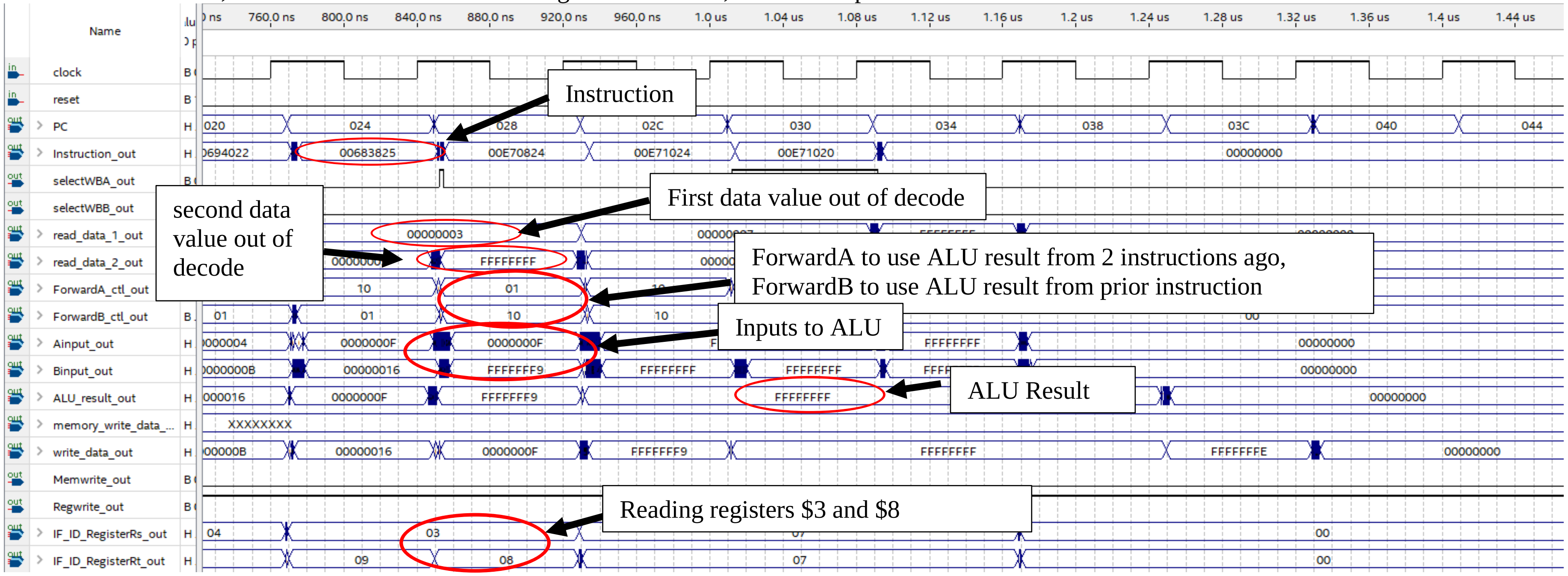
Instruction: sub $8, $3, $9

Hazard: Hazard in Rt, need result from 2 instructions ago. Hazard in Rs, need to use prior instruction's result.

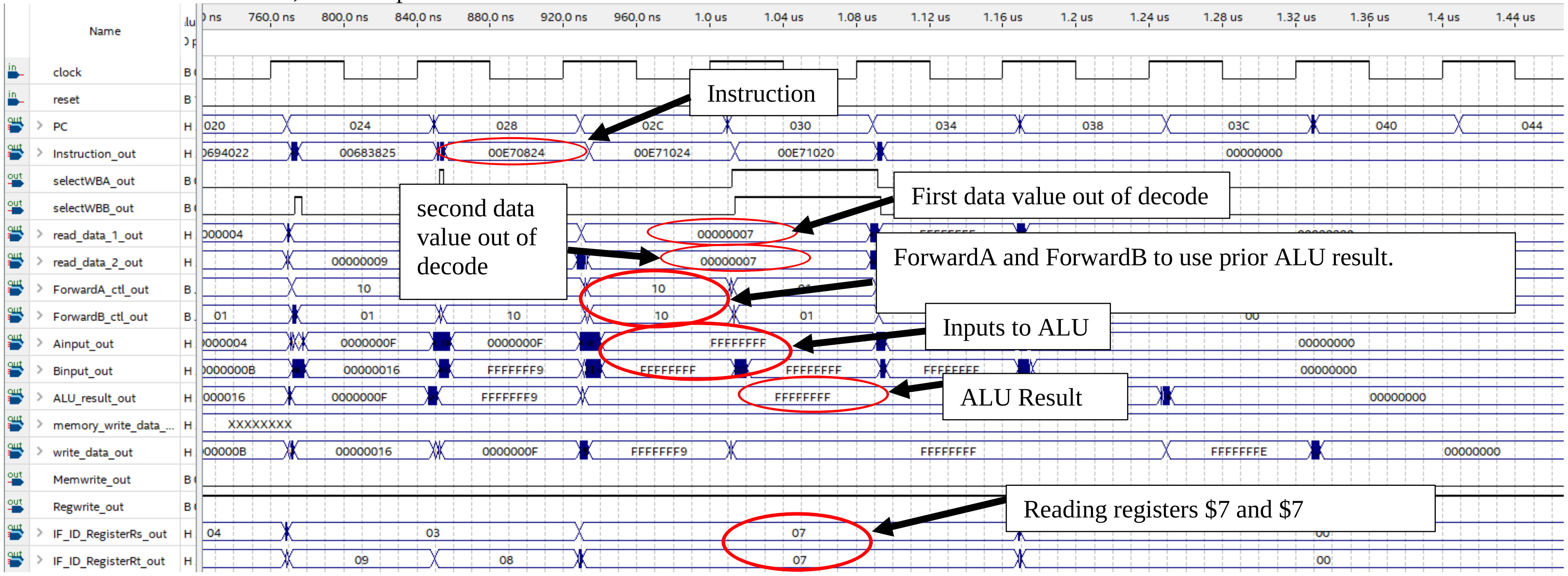


Instruction: or $7, $3, $8

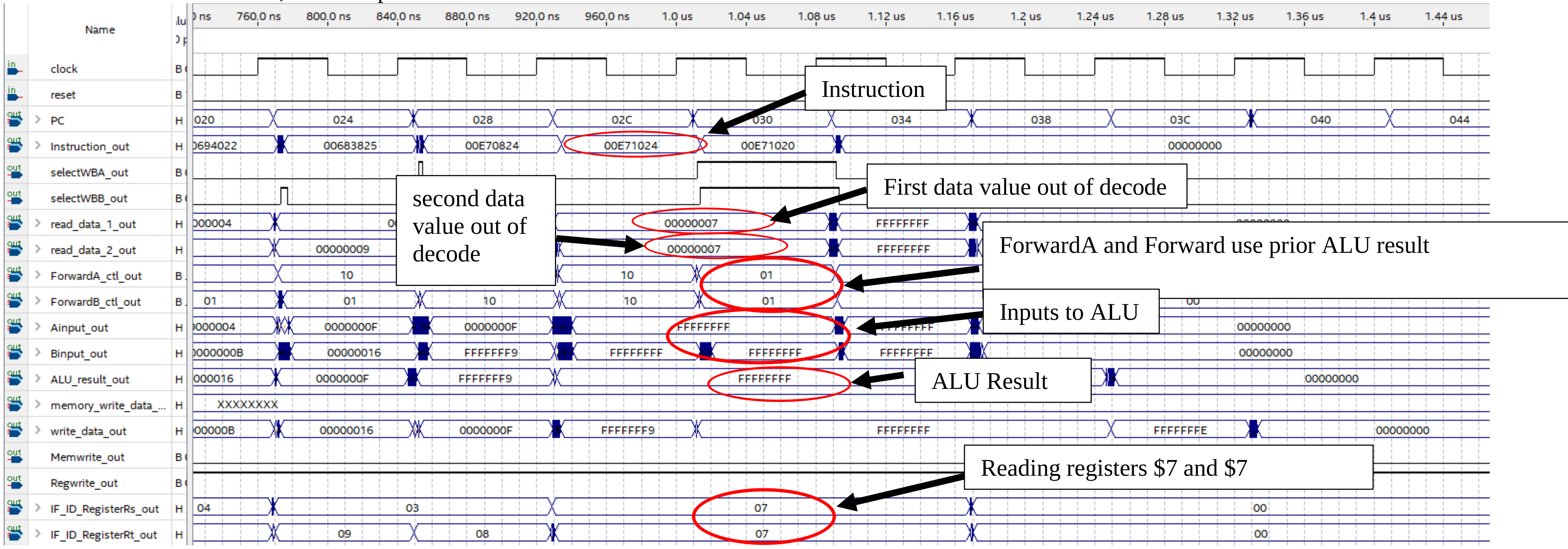
Hazard: Hazard in Rs, need result from 2 instructions ago. Hazard in Rt, need to use prior instruction's result.



Instruction: and $1, $7, $7
Hazard: Hazard in Rt and Rs, both use prior ALU result.

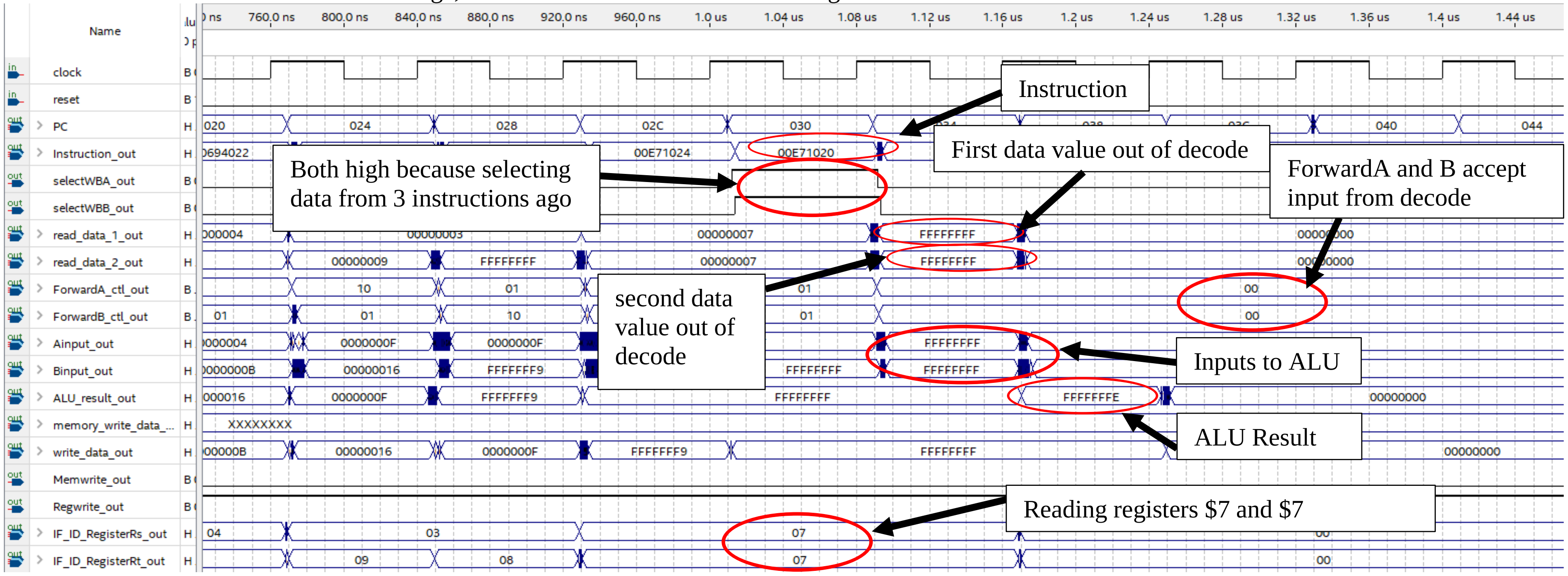


Instruction: and $2, $7, $7
Hazard: Hazard in Rt and Rs, both use prior ALU result.



Instruction: add $2, $7, $7

Hazard: Hazard in Rt and Rs in decode stage, both use result from 3 instructions ago.



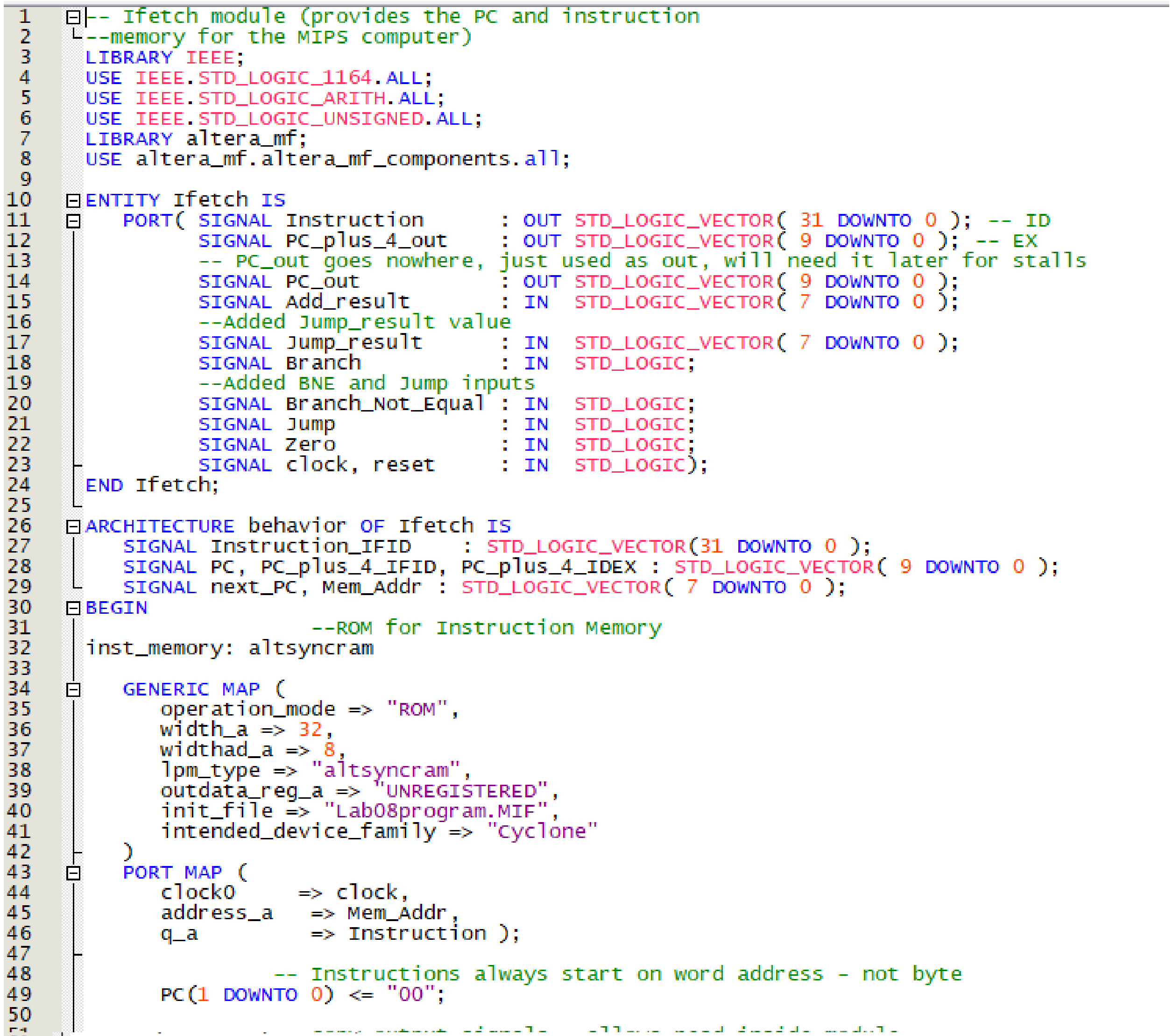
The analysis shows that the hazard detection units work properly, forwarding the correct data when it needs to be there. These detection units do not cover any branching or jumping, which will be addressed in the next lab. All the code is shown in Appendix A.

References

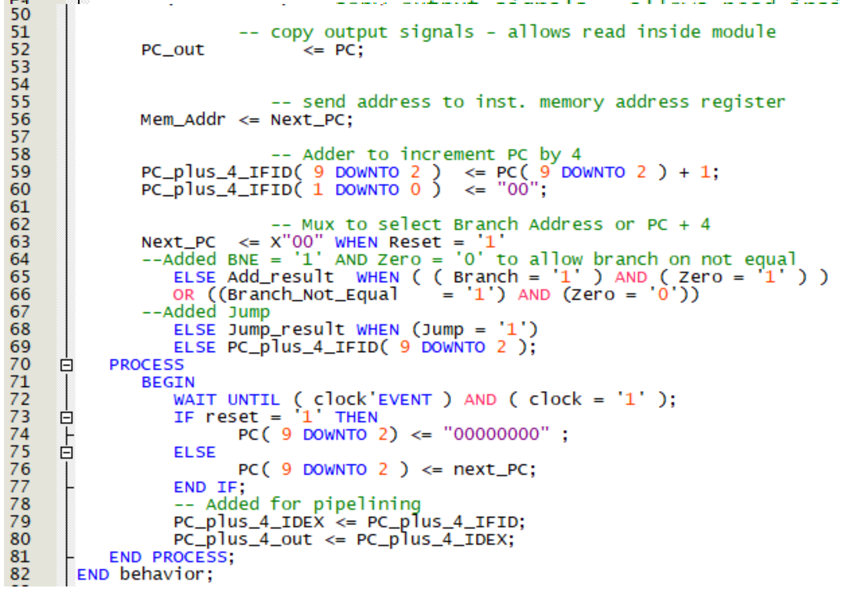
[1] J. Gusler, "Lab 07".

[2] H. F. Hamblen, in *Rapid Prototyping of Digital Systems, SOPC Edition*, p. Section 14.

Appendix A

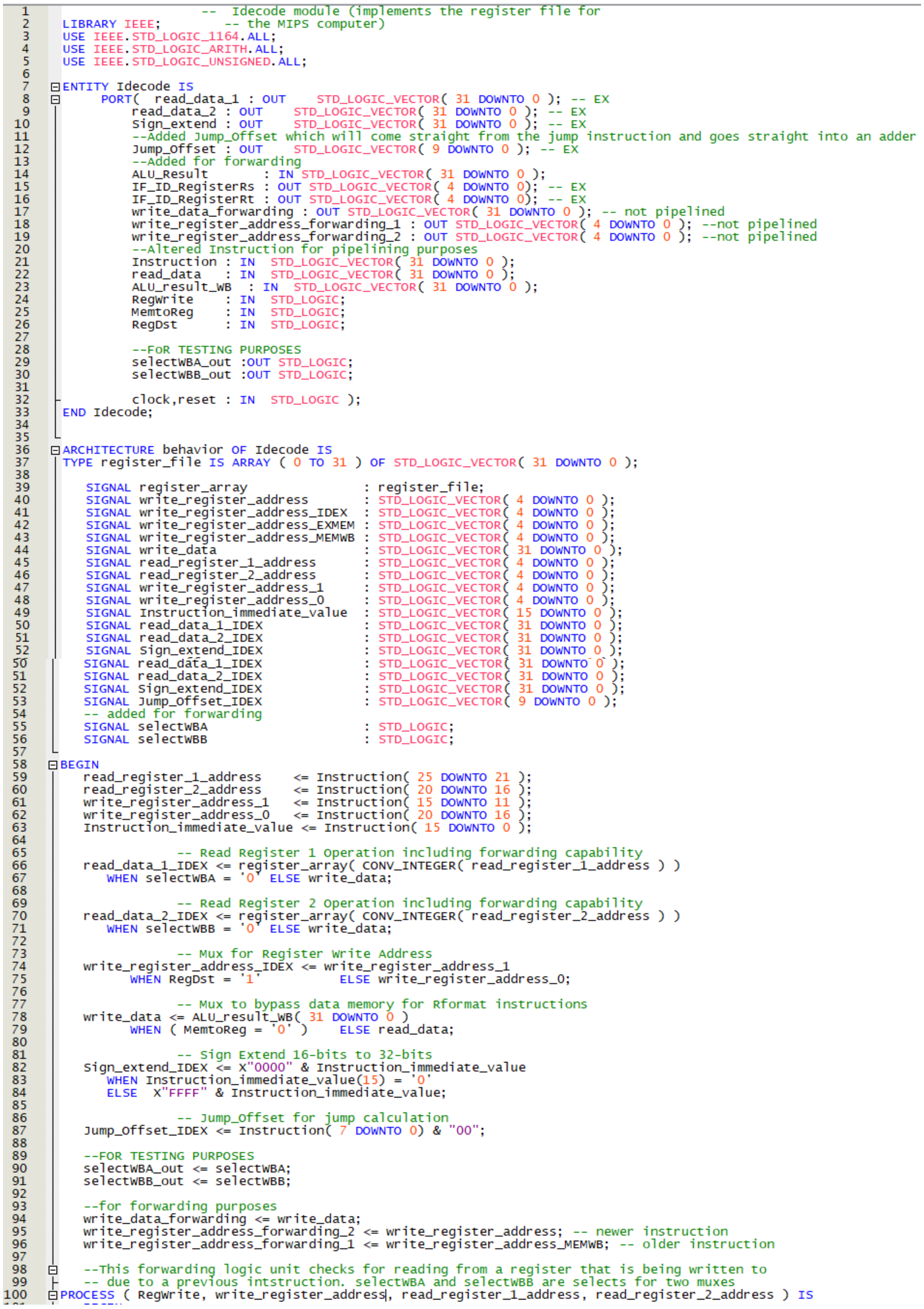


```
-- Ifetch module (provides the PC and instruction
--memory for the MIPS computer)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY Ifetch IS
   PORT( SIGNAL Instruction       : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- ID
         SIGNAL PC_plus_4_out     : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0 ); -- EX
         -- PC_out goes nowhere, just used as out, will need it later for stalls
         SIGNAL PC_out            : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0 );
         SIGNAL Add_result        : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         --Added Jump_result value
         SIGNAL Jump_result       : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         SIGNAL Branch            : IN  STD_LOGIC;
         --Added BNE and Jump inputs
         SIGNAL Branch_Not_Equal : IN  STD_LOGIC;
         SIGNAL Jump              : IN  STD_LOGIC;
         SIGNAL Zero              : IN  STD_LOGIC;
         SIGNAL clock, reset      : IN  STD_LOGIC);
END Ifetch;

ARCHITECTURE behavior OF Ifetch IS
   SIGNAL Instruction_IFID     : STD_LOGIC_VECTOR(31 DOWNTO 0 );
   SIGNAL PC, PC_plus_4_IFID, PC_plus_4_IDEX : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
   SIGNAL next_PC, Mem_Addr : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
BEGIN
                  --ROM for Instruction Memory
inst_memory: altsyncram

   GENERIC MAP (
      operation_mode => "ROM",
      width_a => 32,
      widthad_a => 8,
      lpm_type => "altsyncram",
      outdata_reg_a => "UNREGISTERED",
      init_file => "Lab08program.MIF",
      intended_device_family => "Cyclone"
   )
   PORT MAP (
      clock0      => clock,
      address_a    => Mem_Addr,
      q_a          => Instruction );

            -- Instructions always start on word address - not byte
      PC(1 DOWNTO 0) <= "00";
```

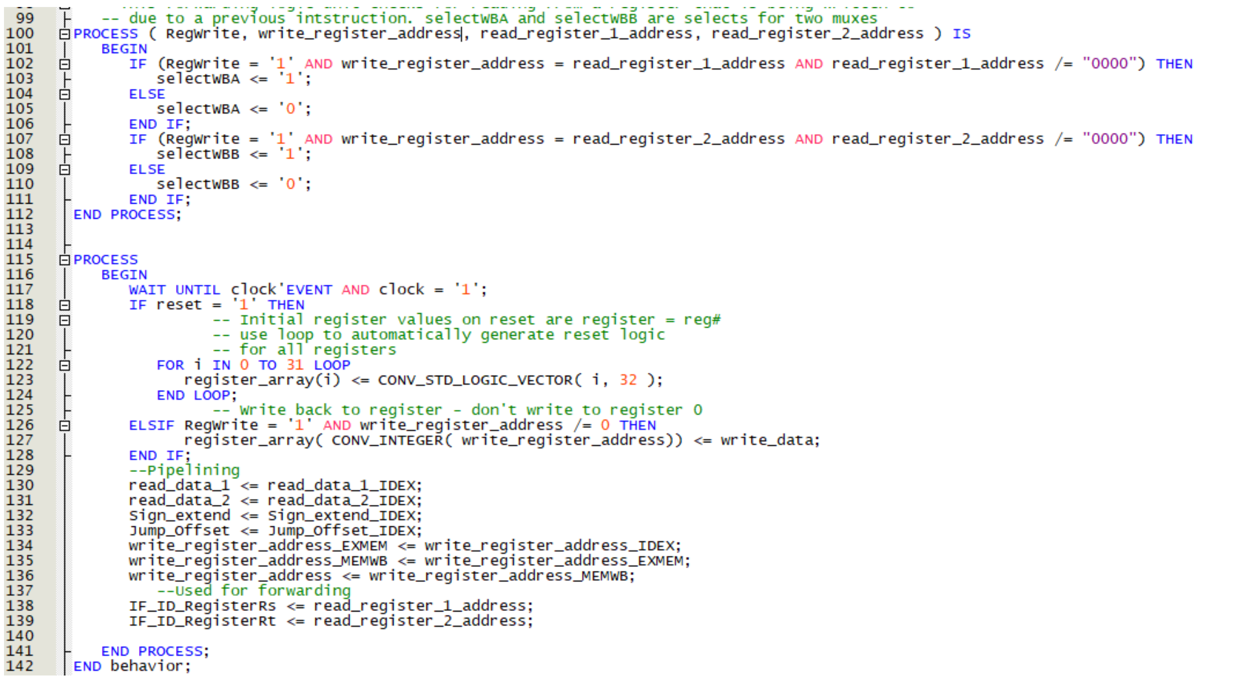


```
            -- copy output signals - allows read inside module
      PC_out         <= PC;


               -- send address to inst. memory address register
      Mem_Addr <= Next_PC;

               -- Adder to increment PC by 4
      PC_plus_4_IFID( 9 DOWNTO 2 )  <= PC( 9 DOWNTO 2 ) + 1;
      PC_plus_4_IFID( 1 DOWNTO 0 )  <= "00";

               -- Mux to select Branch Address or PC + 4
      Next_PC  <= X"00" WHEN Reset = '1'
      --Added BNE = '1' AND Zero = '0' to allow branch on not equal
         ELSE Add_result  WHEN ( ( Branch = '1' ) AND ( Zero = '1' ) )
         OR ((Branch_Not_Equal    = '1') AND (Zero = '0'))
      --Added Jump
         ELSE Jump_result WHEN (Jump = '1')
         ELSE PC_plus_4_IFID( 9 DOWNTO 2 );
   PROCESS
      BEGIN
         WAIT UNTIL ( clock'EVENT ) AND ( clock = '1' );
         IF reset = '1' THEN
               PC( 9 DOWNTO 2) <= "00000000" ;
         ELSE
               PC( 9 DOWNTO 2 ) <= next_PC;
         END IF;
         -- Added for pipelining
         PC_plus_4_IDEX <= PC_plus_4_IFID;
         PC_plus_4_out <= PC_plus_4_IDEX;
   END PROCESS;
END behavior;
```

Code for Fetch stage

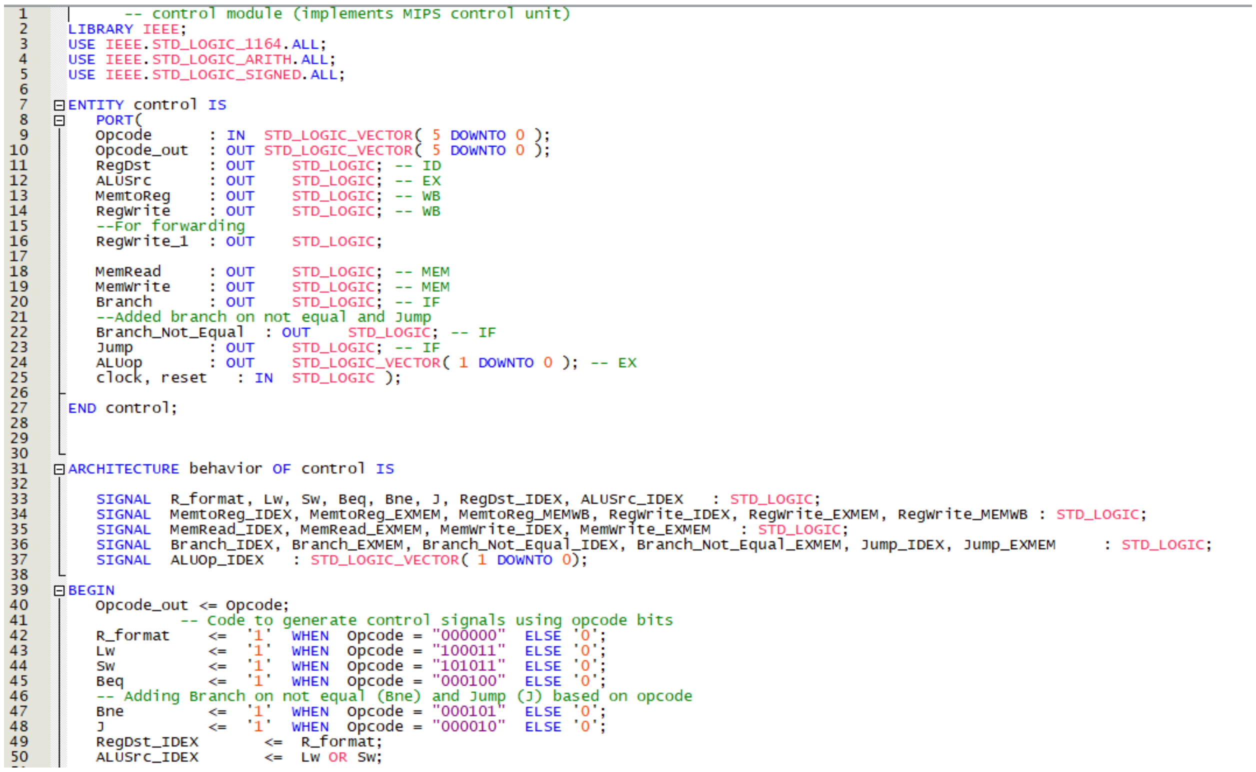


```vhdl
                    --  Idecode module (implements the register file for
LIBRARY IEEE;          -- the MIPS computer)
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY Idecode IS
     PORT(  read_data_1 : OUT    STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- EX
         read_data_2 : OUT    STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- EX
         Sign_extend : OUT    STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- EX
         --Added Jump_Offset which will come straight from the jump instruction and goes straight into an adder
         Jump_Offset : OUT    STD_LOGIC_VECTOR( 9 DOWNTO 0 ); -- EX
         --Added for forwarding
         ALU_Result        : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         IF_ID_RegisterRs : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0); -- EX
         IF_ID_RegisterRt : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0); -- EX
         write_data_forwarding : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- not pipelined
         write_register_address_forwarding_1 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
         write_register_address_forwarding_2 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
         --Altered Instruction for pipelining purposes
         Instruction : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         read_data   : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         ALU_result_WB  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         RegWrite    : IN  STD_LOGIC;
         MemtoReg    : IN  STD_LOGIC;
         RegDst      : IN  STD_LOGIC;

         --FOR TESTING PURPOSES
         selectWBA_out :OUT STD_LOGIC;
         selectWBB_out :OUT STD_LOGIC;

         clock,reset : IN  STD_LOGIC );
END Idecode;


ARCHITECTURE behavior OF Idecode IS
TYPE register_file IS ARRAY ( 0 TO 31 ) OF STD_LOGIC_VECTOR( 31 DOWNTO 0 );

   SIGNAL register_array                : register_file;
   SIGNAL write_register_address        : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
   SIGNAL write_register_address_IDEX   : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
   SIGNAL write_register_address_EXMEM  : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
   SIGNAL write_register_address_MEMWB  : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
   SIGNAL write_data                    : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
   SIGNAL read_register_1_address       : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
   SIGNAL read_register_2_address       : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
   SIGNAL write_register_address_1      : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
   SIGNAL write_register_address_0      : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
   SIGNAL Instruction_immediate_value   : STD_LOGIC_VECTOR( 15 DOWNTO 0 );
   SIGNAL read_data_1_IDEX              : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
   SIGNAL read_data_2_IDEX              : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
   SIGNAL Sign_extend_IDEX              : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
   SIGNAL read_data_1_IDEX              : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
   SIGNAL read_data_2_IDEX              : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
   SIGNAL Sign_extend_IDEX              : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
   SIGNAL Jump_Offset_IDEX              : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
   -- added for forwarding
   SIGNAL selectWBA                     : STD_LOGIC;
   SIGNAL selectWBB                     : STD_LOGIC;

BEGIN
   read_register_1_address     <= Instruction( 25 DOWNTO 21 );
   read_register_2_address     <= Instruction( 20 DOWNTO 16 );
   write_register_address_1    <= Instruction( 15 DOWNTO 11 );
   write_register_address_0    <= Instruction( 20 DOWNTO 16 );
   Instruction_immediate_value <= Instruction( 15 DOWNTO 0 );

               -- Read Register 1 Operation including forwarding capability
   read_data_1_IDEX <= register_array( CONV_INTEGER( read_register_1_address ) )
      WHEN selectWBA = '0' ELSE write_data;

               -- Read Register 2 Operation including forwarding capability
   read_data_2_IDEX <= register_array( CONV_INTEGER( read_register_2_address ) )
      WHEN selectWBB = '0' ELSE write_data;

               -- Mux for Register Write Address
   write_register_address_IDEX <= write_register_address_1
         WHEN RegDst = '1'         ELSE write_register_address_0;

               -- Mux to bypass data memory for Rformat instructions
   write_data <= ALU_result_WB( 31 DOWNTO 0 )
         WHEN ( MemtoReg = '0' )    ELSE read_data;

               -- Sign Extend 16-bits to 32-bits
   Sign_extend_IDEX <= X"0000" & Instruction_immediate_value
      WHEN Instruction_immediate_value(15) = '0'
      ELSE  X"FFFF" & Instruction_immediate_value;

               -- Jump_Offset for jump calculation
   Jump_Offset_IDEX <= Instruction( 7 DOWNTO 0) & "00";

   --FOR TESTING PURPOSES
   selectWBA_out <= selectWBA;
   selectWBB_out <= selectWBB;

   --for forwarding purposes
   write_data_forwarding <= write_data;
   write_register_address_forwarding_2 <= write_register_address; -- newer instruction
   write_register_address_forwarding_1 <= write_register_address_MEMWB; -- older instruction

   --This forwarding logic unit checks for reading from a register that is being written to
   -- due to a previous intstruction. selectWBA and selectWBB are selects for two muxes
PROCESS ( RegWrite, write_register_address, read_register_1_address, read_register_2_address ) IS
```

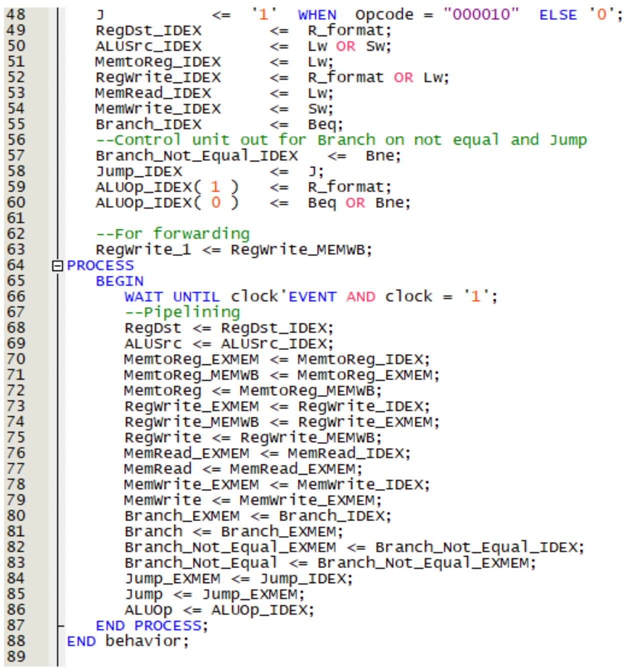


```
  -- due to a previous intstruction. selectWBA and selectWBB are selects for two muxes
PROCESS ( RegWrite, write_register_address, read_register_1_address, read_register_2_address ) IS
    BEGIN
        IF (RegWrite = '1' AND write_register_address = read_register_1_address AND read_register_1_address /= "0000") THEN
            selectWBA <= '1';
        ELSE
            selectWBA <= '0';
        END IF;
        IF (RegWrite = '1' AND write_register_address = read_register_2_address AND read_register_2_address /= "0000") THEN
            selectWBB <= '1';
        ELSE
            selectWBB <= '0';
        END IF;
END PROCESS;


PROCESS
    BEGIN
        WAIT UNTIL clock'EVENT AND clock = '1';
        IF reset = '1' THEN
                    -- Initial register values on reset are register = reg#
                    -- use loop to automatically generate reset logic
                    -- for all registers
            FOR i IN 0 TO 31 LOOP
                register_array(i) <= CONV_STD_LOGIC_VECTOR( i, 32 );
            END LOOP;
                    -- Write back to register - don't write to register 0
        ELSIF RegWrite = '1' AND write_register_address /= 0 THEN
                register_array( CONV_INTEGER( write_register_address)) <= write_data;
        END IF;
        --Pipelining
        read_data_1 <= read_data_1_IDEX;
        read_data_2 <= read_data_2_IDEX;
        Sign_extend <= Sign_extend_IDEX;
        Jump_Offset <= Jump_Offset_IDEX;
        write_register_address_EXMEM <= write_register_address_IDEX;
        write_register_address_MEMWB <= write_register_address_EXMEM;
        write_register_address <= write_register_address_MEMWB;
            --Used for forwarding
        IF_ID_RegisterRs <= read_register_1_address;
        IF_ID_RegisterRt <= read_register_2_address;

    END PROCESS;
END behavior;
```

Code for Decode stage

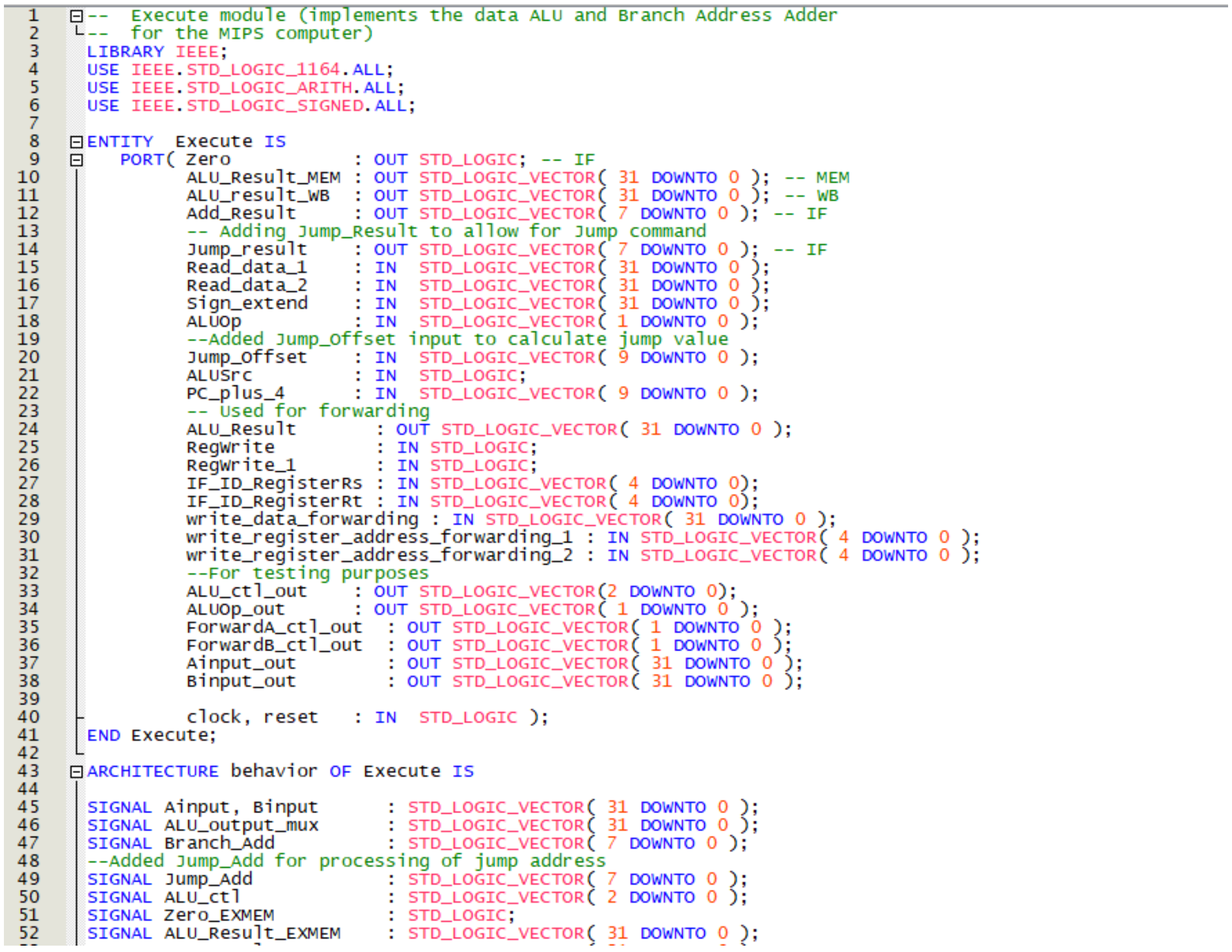


```
        -- control module (implements MIPS control unit)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY control IS
    PORT(
    Opcode        : IN  STD_LOGIC_VECTOR( 5 DOWNTO 0 );
    Opcode_out    : OUT STD_LOGIC_VECTOR( 5 DOWNTO 0 );
    RegDst        : OUT     STD_LOGIC; -- ID
    ALUSrc        : OUT     STD_LOGIC; -- EX
    MemtoReg      : OUT     STD_LOGIC; -- WB
    RegWrite      : OUT     STD_LOGIC; -- WB
    --For forwarding
    RegWrite_1    : OUT     STD_LOGIC;

    MemRead       : OUT     STD_LOGIC; -- MEM
    MemWrite      : OUT     STD_LOGIC; -- MEM
    Branch        : OUT     STD_LOGIC; -- IF
    --Added branch on not equal and Jump
    Branch_Not_Equal  : OUT     STD_LOGIC; -- IF
    Jump          : OUT     STD_LOGIC; -- IF
    ALUop         : OUT     STD_LOGIC_VECTOR( 1 DOWNTO 0 ); -- EX
    clock, reset    : IN  STD_LOGIC );

END control;



ARCHITECTURE behavior OF control IS

    SIGNAL  R_format, Lw, Sw, Beq, Bne, J, RegDst_IDEX, ALUSrc_IDEX    : STD_LOGIC;
    SIGNAL  MemtoReg_IDEX, MemtoReg_EXMEM, MemtoReg_MEMWB, RegWrite_IDEX, RegWrite_EXMEM, RegWrite_MEMWB : STD_LOGIC;
    SIGNAL  MemRead_IDEX, MemRead_EXMEM, MemWrite_IDEX, MemWrite_EXMEM    : STD_LOGIC;
    SIGNAL  Branch_IDEX, Branch_EXMEM, Branch_Not_Equal_IDEX, Branch_Not_Equal_EXMEM, Jump_IDEX, Jump_EXMEM      : STD_LOGIC;
    SIGNAL  ALUOp_IDEX    : STD_LOGIC_VECTOR( 1 DOWNTO 0);

BEGIN
    Opcode_out <= Opcode;
            -- Code to generate control signals using opcode bits
    R_format    <=  '1'  WHEN  Opcode = "000000"  ELSE '0';
    Lw          <=  '1'  WHEN  Opcode = "100011"  ELSE '0';
    Sw          <=  '1'  WHEN  Opcode = "101011"  ELSE '0';
    Beq         <=  '1'  WHEN  Opcode = "000100"  ELSE '0';
    -- Adding Branch on not equal (Bne) and Jump (J) based on opcode
    Bne         <=  '1'  WHEN  Opcode = "000101"  ELSE '0';
    J           <=  '1'  WHEN  Opcode = "000010"  ELSE '0';
    RegDst_IDEX       <=  R_format;
    ALUSrc_IDEX       <=  Lw OR Sw;
```

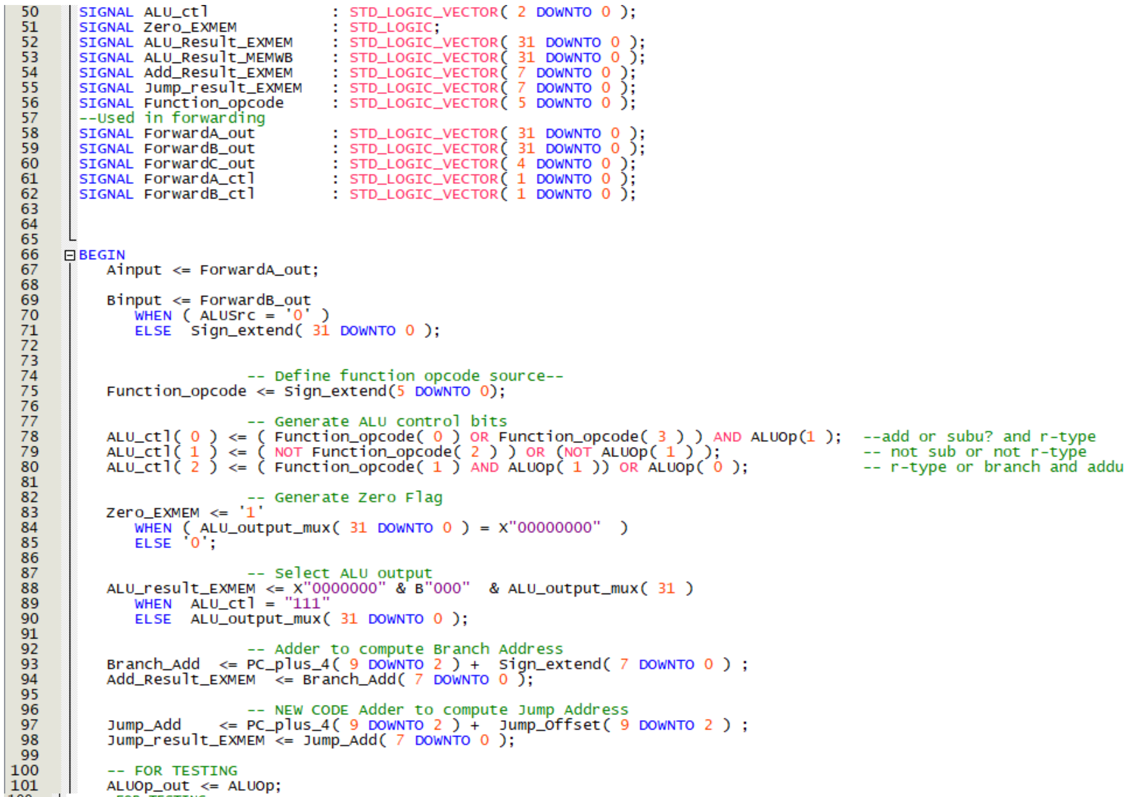


```vhdl
   J            <=  '1'  WHEN  Opcode = "000010"  ELSE '0';
   RegDst_IDEX        <=  R_format;
   ALUSrc_IDEX        <=  Lw OR Sw;
   MemtoReg_IDEX      <=  Lw;
   RegWrite_IDEX      <=  R_format OR Lw;
   MemRead_IDEX       <=  Lw;
   MemWrite_IDEX      <=  Sw;
   Branch_IDEX        <=  Beq;
   --Control unit out for Branch on not equal and Jump
   Branch_Not_Equal_IDEX    <=  Bne;
   Jump_IDEX          <=  J;
   ALUOp_IDEX( 1 )    <=  R_format;
   ALUOp_IDEX( 0 )    <=  Beq OR Bne;

   --For forwarding
   RegWrite_1 <= RegWrite_MEMWB;
PROCESS
   BEGIN
      WAIT UNTIL clock'EVENT AND clock = '1';
      --Pipelining
      RegDst <= RegDst_IDEX;
      ALUSrc <= ALUSrc_IDEX;
      MemtoReg_EXMEM <= MemtoReg_IDEX;
      MemtoReg_MEMWB <= MemtoReg_EXMEM;
      MemtoReg <= MemtoReg_MEMWB;
      RegWrite_EXMEM <= RegWrite_IDEX;
      RegWrite_MEMWB <= RegWrite_EXMEM;
      RegWrite <= RegWrite_MEMWB;
      MemRead_EXMEM <= MemRead_IDEX;
      MemRead <= MemRead_EXMEM;
      MemWrite_EXMEM <= MemWrite_IDEX;
      MemWrite <= MemWrite_EXMEM;
      Branch_EXMEM <= Branch_IDEX;
      Branch <= Branch_EXMEM;
      Branch_Not_Equal_EXMEM <= Branch_Not_Equal_IDEX;
      Branch_Not_Equal <= Branch_Not_Equal_EXMEM;
      Jump_EXMEM <= Jump_IDEX;
      Jump <= Jump_EXMEM;
      ALUOp <= ALUOp_IDEX;
   END PROCESS;
END behavior;
```

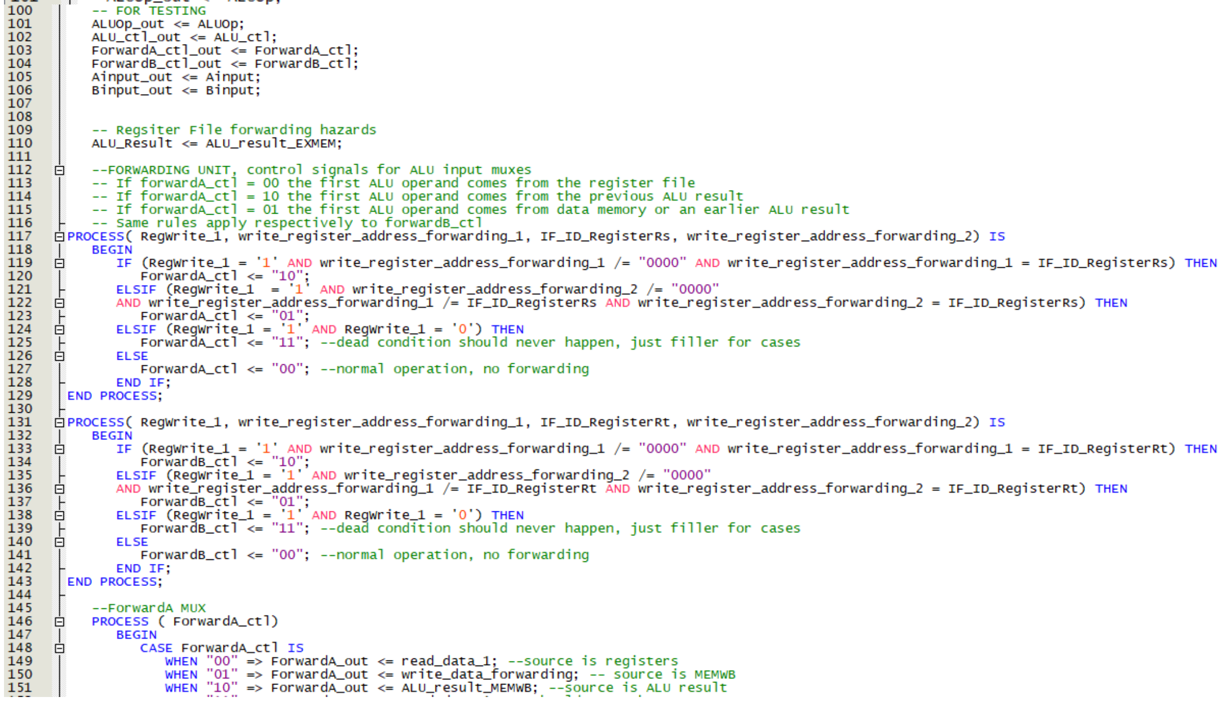
Code for Control stage



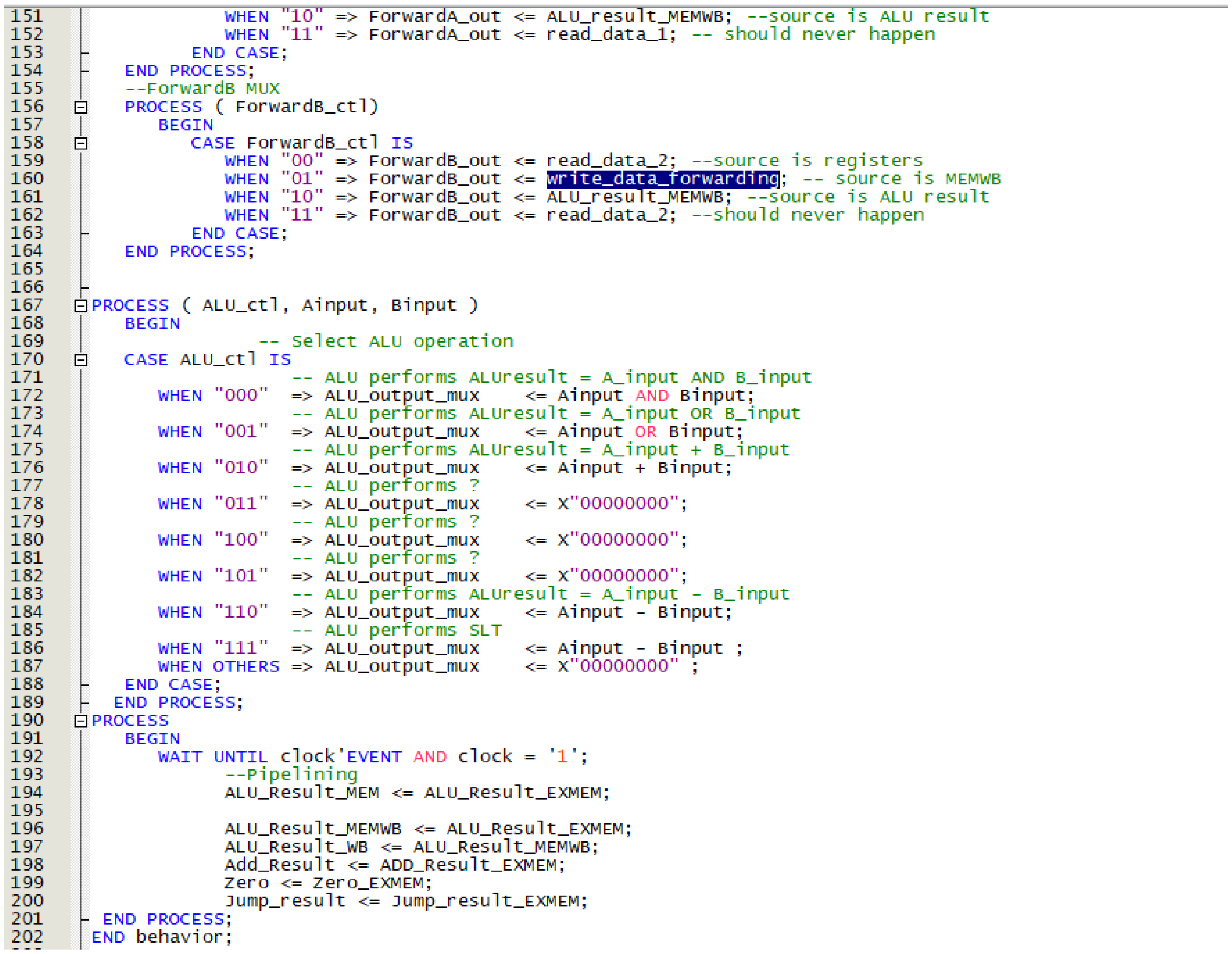
```vhdl
--   Execute module (implements the data ALU and Branch Address Adder
--   for the MIPS computer)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY  Execute IS
   PORT( Zero            : OUT STD_LOGIC; -- IF
         ALU_Result_MEM  : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- MEM
         ALU_result_WB   : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- WB
         Add_Result      : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 ); -- IF
         -- Adding Jump_Result to allow for Jump command
         Jump_result     : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 ); -- IF
         Read_data_1     : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         Read_data_2     : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         Sign_extend     : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         ALUOp           : IN  STD_LOGIC_VECTOR( 1 DOWNTO 0 );
         --Added Jump_Offset input to calculate jump value
         Jump_Offset     : IN  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
         ALUSrc          : IN  STD_LOGIC;
         PC_plus_4       : IN  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
         -- Used for forwarding
         ALU_Result        : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         RegWrite          : IN STD_LOGIC;
         RegWrite_1        : IN STD_LOGIC;
         IF_ID_RegisterRs : IN STD_LOGIC_VECTOR( 4 DOWNTO 0);
         IF_ID_RegisterRt : IN STD_LOGIC_VECTOR( 4 DOWNTO 0);
         write_data_forwarding : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         write_register_address_forwarding_1 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
         write_register_address_forwarding_2 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
         --For testing purposes
         ALU_ctl_out     : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
         ALUOp_out       : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
         ForwardA_ctl_out   : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
         ForwardB_ctl_out   : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
         Ainput_out         : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         Binput_out         : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );

         clock, reset    : IN  STD_LOGIC );
END Execute;

ARCHITECTURE behavior OF Execute IS

SIGNAL Ainput, Binput        : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL ALU_output_mux        : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Branch_Add            : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
--Added Jump_Add for processing of jump address
SIGNAL Jump_Add              : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL ALU_ctl               : STD_LOGIC_VECTOR( 2 DOWNTO 0 );
SIGNAL Zero_EXMEM            : STD_LOGIC;
SIGNAL ALU_Result_EXMEM      : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
```



```
SIGNAL ALU_ctl              : STD_LOGIC_VECTOR( 2 DOWNTO 0 );
SIGNAL Zero_EXMEM           : STD_LOGIC;
SIGNAL ALU_Result_EXMEM     : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL ALU_Result_MEMWB     : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Add_Result_EXMEM     : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL Jump_result_EXMEM    : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL Function_opcode      : STD_LOGIC_VECTOR( 5 DOWNTO 0 );
--Used in forwarding
SIGNAL ForwardA_out         : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL ForwardB_out         : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL ForwardC_out         : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
SIGNAL ForwardA_ctl         : STD_LOGIC_VECTOR( 1 DOWNTO 0 );
SIGNAL ForwardB_ctl         : STD_LOGIC_VECTOR( 1 DOWNTO 0 );



BEGIN
    Ainput <= ForwardA_out;

    Binput <= ForwardB_out
        WHEN ( ALUSrc = '0' )
        ELSE  Sign_extend( 31 DOWNTO 0 );


                    -- Define function opcode source--
    Function_opcode <= Sign_extend(5 DOWNTO 0);

                    -- Generate ALU control bits
    ALU_ctl( 0 ) <= ( Function_opcode( 0 ) OR Function_opcode( 3 ) ) AND ALUOp(1 );   --add or subu? and r-type
    ALU_ctl( 1 ) <= ( NOT Function_opcode( 2 ) ) OR (NOT ALUOp( 1 ) );                -- not sub or not r-type
    ALU_ctl( 2 ) <= ( Function_opcode( 1 ) AND ALUOp( 1 )) OR ALUOp( 0 );              -- r-type or branch and addu

                    -- Generate Zero Flag
    Zero_EXMEM <= '1'
        WHEN ( ALU_output_mux( 31 DOWNTO 0 ) = X"00000000"  )
        ELSE '0';

                    -- Select ALU output
    ALU_result_EXMEM <= X"0000000" & B"000"  & ALU_output_mux( 31 )
        WHEN  ALU_ctl = "111"
        ELSE  ALU_output_mux( 31 DOWNTO 0 );

                    -- Adder to compute Branch Address
    Branch_Add   <= PC_plus_4( 9 DOWNTO 2 ) +  Sign_extend( 7 DOWNTO 0 ) ;
    Add_Result_EXMEM  <= Branch_Add( 7 DOWNTO 0 );

                    -- NEW CODE Adder to compute Jump Address
    Jump_Add     <= PC_plus_4( 9 DOWNTO 2 ) +  Jump_Offset( 9 DOWNTO 2 ) ;
    Jump_result_EXMEM <= Jump_Add( 7 DOWNTO 0 );

    -- FOR TESTING
    ALUOp_out <= ALUOp;
```

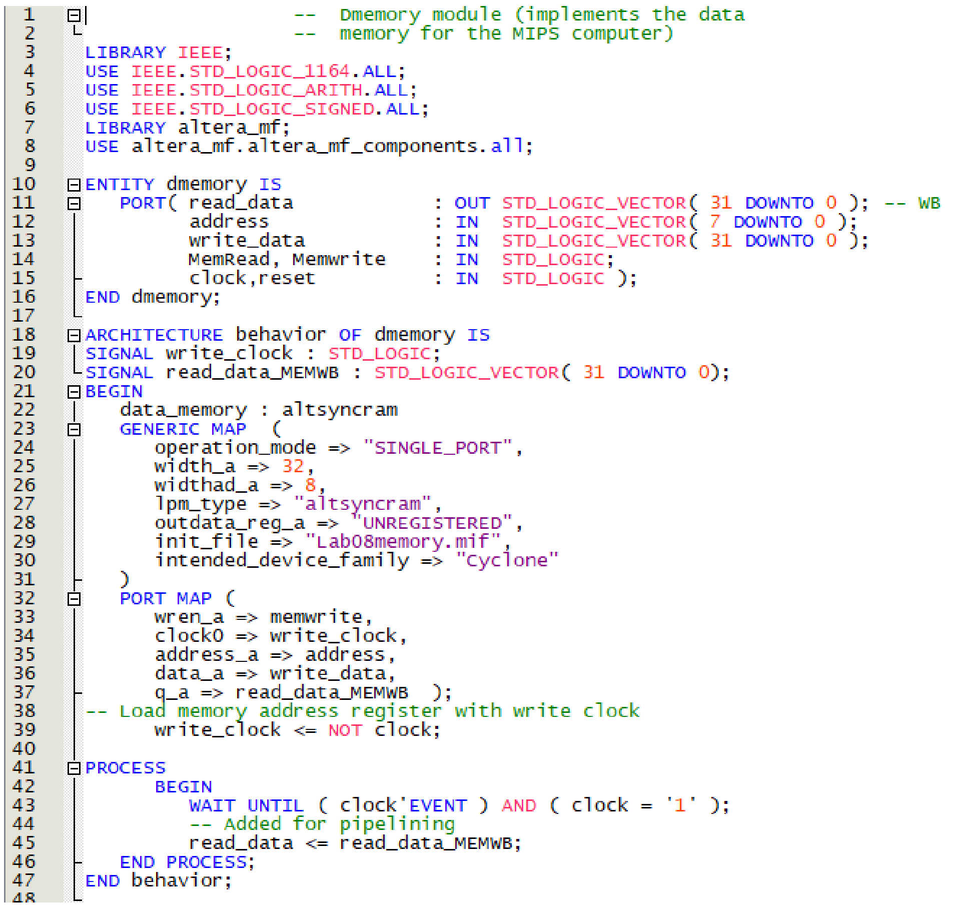


```
    -- FOR TESTING
    ALUOp_out <= ALUOp;
    ALU_ctl_out <= ALU_ctl;
    ForwardA_ctl_out <= ForwardA_ctl;
    ForwardB_ctl_out <= ForwardB_ctl;
    Ainput_out <= Ainput;
    Binput_out <= Binput;


    -- Regsiter File forwarding hazards
    ALU_Result <= ALU_result_EXMEM;

    --FORWARDING UNIT, control signals for ALU input muxes
    -- If forwardA_ctl = 00 the first ALU operand comes from the register file
    -- If forwardA_ctl = 10 the first ALU operand comes from the previous ALU result
    -- If forwardA_ctl = 01 the first ALU operand comes from data memory or an earlier ALU result
    -- Same rules apply respectively to forwardB_ctl
PROCESS( RegWrite_1, write_register_address_forwarding_1, IF_ID_RegisterRs, write_register_address_forwarding_2) IS
    BEGIN
        IF (RegWrite_1 = '1' AND write_register_address_forwarding_1 /= "0000" AND write_register_address_forwarding_1 = IF_ID_RegisterRs) THEN
            ForwardA_ctl <= "10";
        ELSIF (RegWrite_1  = '1' AND write_register_address_forwarding_2 /= "0000"
        AND write_register_address_forwarding_1 /= IF_ID_RegisterRs AND write_register_address_forwarding_2 = IF_ID_RegisterRs) THEN
            ForwardA_ctl <= "01";
        ELSIF (RegWrite_1 = '1' AND RegWrite_1 = '0') THEN
            ForwardA_ctl <= "11"; --dead condition should never happen, just filler for cases
        ELSE
            ForwardA_ctl <= "00"; --normal operation, no forwarding
        END IF;
END PROCESS;

PROCESS( RegWrite_1, write_register_address_forwarding_1, IF_ID_RegisterRt, write_register_address_forwarding_2) IS
    BEGIN
        IF (RegWrite_1 = '1' AND write_register_address_forwarding_1 /= "0000" AND write_register_address_forwarding_1 = IF_ID_RegisterRt) THEN
            ForwardB_ctl <= "10";
        ELSIF (RegWrite_1 = '1' AND write_register_address_forwarding_2 /= "0000"
        AND write_register_address_forwarding_1 /= IF_ID_RegisterRt AND write_register_address_forwarding_2 = IF_ID_RegisterRt) THEN
            ForwardB_ctl <= "01";
        ELSIF (RegWrite_1 = '1' AND RegWrite_1 = '0') THEN
            ForwardB_ctl <= "11"; --dead condition should never happen, just filler for cases
        ELSE
            ForwardB_ctl <= "00"; --normal operation, no forwarding
        END IF;
END PROCESS;

    --ForwardA MUX
    PROCESS ( ForwardA_ctl)
        BEGIN
            CASE ForwardA_ctl IS
                WHEN "00" => ForwardA_out <= read_data_1; --source is registers
                WHEN "01" => ForwardA_out <= write_data_forwarding; -- source is MEMWB
                WHEN "10" => ForwardA_out <= ALU_result_MEMWB; --source is ALU result
```



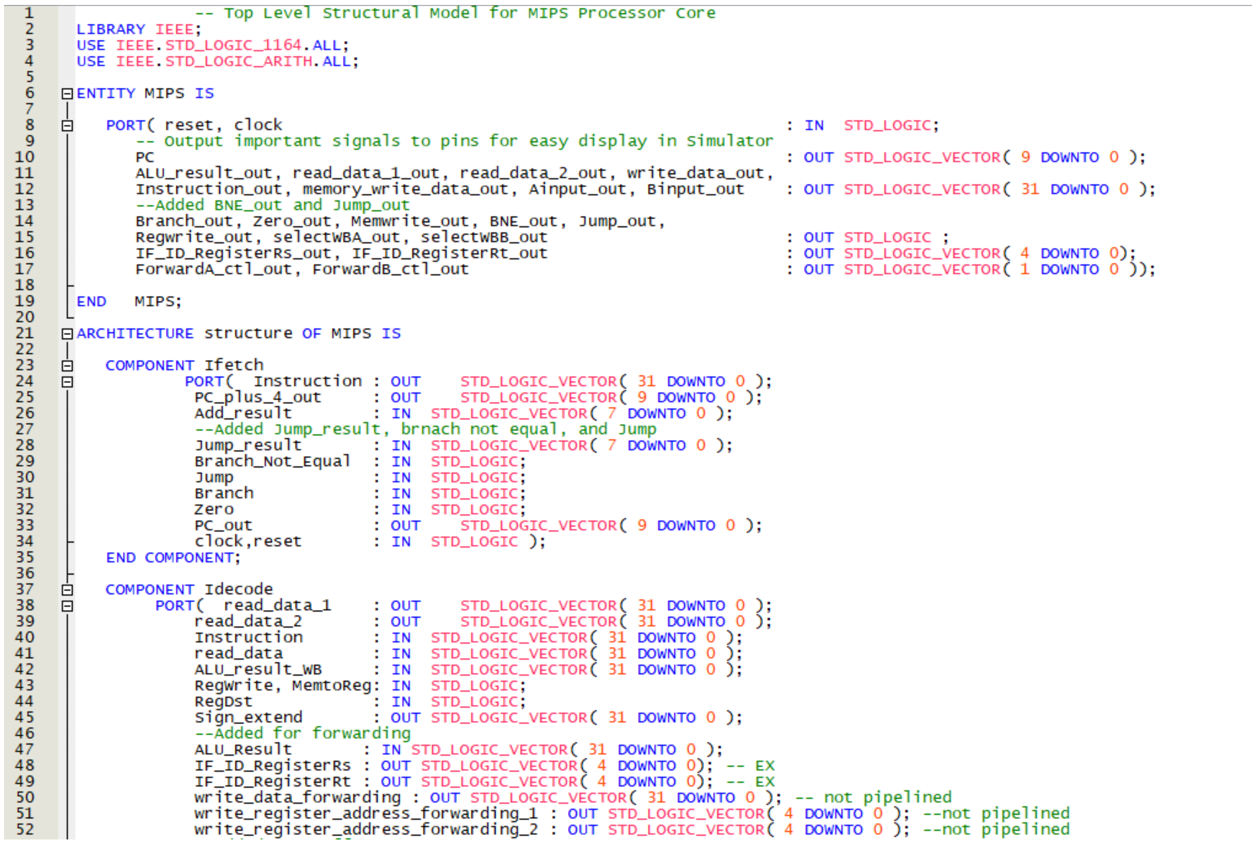
```vhdl
            WHEN "10" => ForwardA_out <= ALU_result_MEMWB; --source is ALU result
            WHEN "11" => ForwardA_out <= read_data_1; -- should never happen
         END CASE;
   END PROCESS;
   --ForwardB MUX
   PROCESS ( ForwardB_ctl)
      BEGIN
         CASE ForwardB_ctl IS
            WHEN "00" => ForwardB_out <= read_data_2; --source is registers
            WHEN "01" => ForwardB_out <= write_data_forwarding; -- source is MEMWB
            WHEN "10" => ForwardB_out <= ALU_result_MEMWB; --source is ALU result
            WHEN "11" => ForwardB_out <= read_data_2; --should never happen
         END CASE;
   END PROCESS;


PROCESS ( ALU_ctl, Ainput, Binput )
   BEGIN
               -- Select ALU operation
   CASE ALU_ctl IS
                  -- ALU performs ALUresult = A_input AND B_input
      WHEN "000"  => ALU_output_mux     <= Ainput AND Binput;
                  -- ALU performs ALUresult = A_input OR B_input
      WHEN "001"  => ALU_output_mux     <= Ainput OR Binput;
                  -- ALU performs ALUresult = A_input + B_input
      WHEN "010"  => ALU_output_mux     <= Ainput + Binput;
                  -- ALU performs ?
      WHEN "011"  => ALU_output_mux     <= X"00000000";
                  -- ALU performs ?
      WHEN "100"  => ALU_output_mux     <= X"00000000";
                  -- ALU performs ?
      WHEN "101"  => ALU_output_mux     <= X"00000000";
                  -- ALU performs ALUresult = A_input - B_input
      WHEN "110"  => ALU_output_mux     <= Ainput - Binput;
                  -- ALU performs SLT
      WHEN "111"  => ALU_output_mux     <= Ainput - Binput ;
      WHEN OTHERS => ALU_output_mux     <= X"00000000" ;
   END CASE;
  END PROCESS;
PROCESS
   BEGIN
      WAIT UNTIL clock'EVENT AND clock = '1';
            --Pipelining
            ALU_Result_MEM <= ALU_Result_EXMEM;

            ALU_Result_MEMWB <= ALU_Result_EXMEM;
            ALU_Result_WB <= ALU_Result_MEMWB;
            Add_Result <= ADD_Result_EXMEM;
            Zero <= Zero_EXMEM;
            Jump_result <= Jump_result_EXMEM;
 END PROCESS;
END behavior;
```

Code for Execute stage

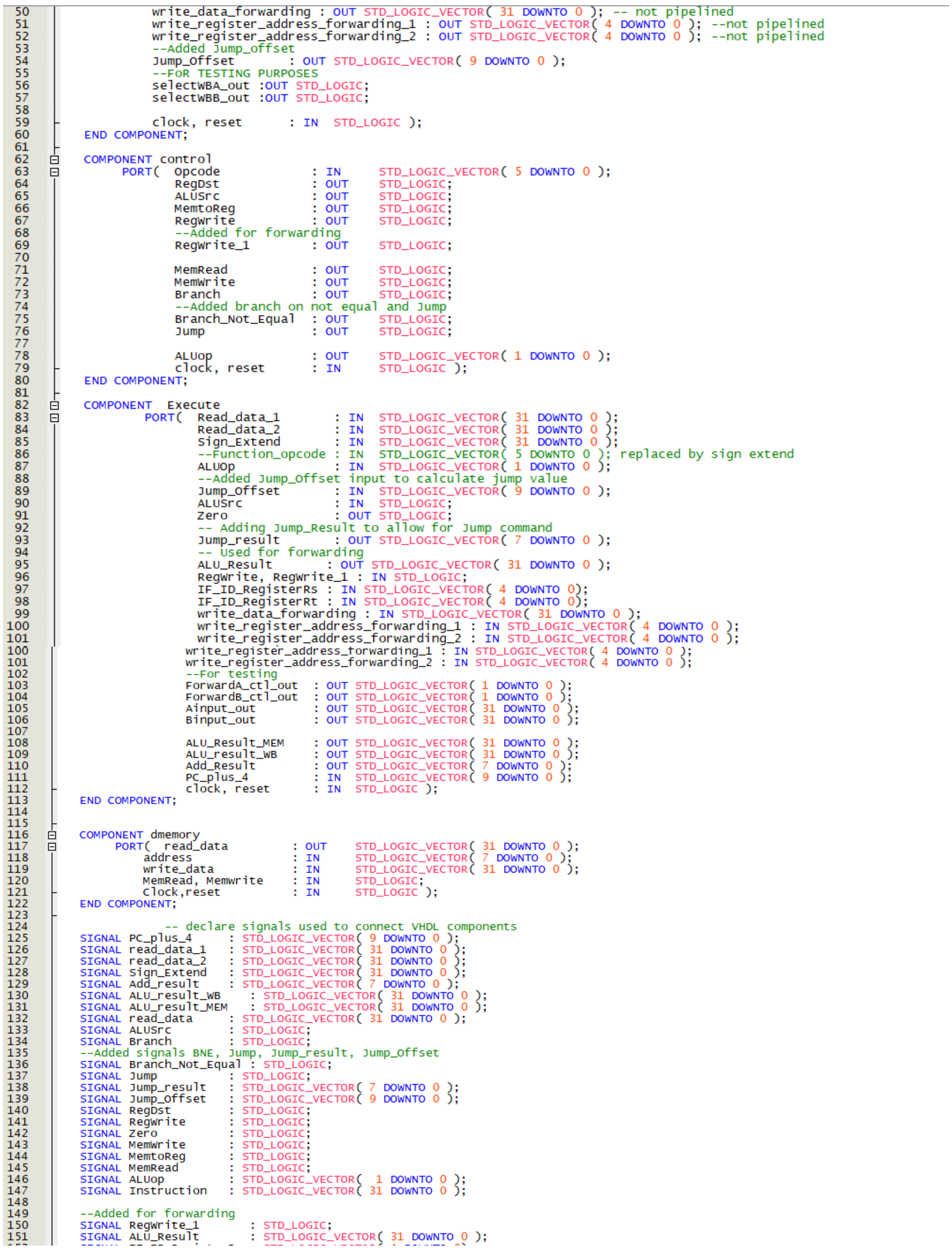


```vhdl
                    --  Dmemory module (implements the data
                    --  memory for the MIPS computer)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY dmemory IS
   PORT( read_data             : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- WB
         address               : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         write_data            : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         MemRead, Memwrite     : IN  STD_LOGIC;
         clock,reset           : IN  STD_LOGIC );
END dmemory;

ARCHITECTURE behavior OF dmemory IS
SIGNAL write_clock : STD_LOGIC;
SIGNAL read_data_MEMWB : STD_LOGIC_VECTOR( 31 DOWNTO 0);
BEGIN
   data_memory : altsyncram
   GENERIC MAP  (
      operation_mode => "SINGLE_PORT",
      width_a => 32,
      widthad_a => 8,
      lpm_type => "altsyncram",
      outdata_reg_a => "UNREGISTERED",
      init_file => "Lab08memory.mif",
      intended_device_family => "Cyclone"
   )
   PORT MAP (
      wren_a => memwrite,
      clock0 => write_clock,
      address_a => address,
      data_a => write_data,
      q_a => read_data_MEMWB  );
-- Load memory address register with write clock
      write_clock <= NOT clock;

PROCESS
      BEGIN
         WAIT UNTIL ( clock'EVENT ) AND ( clock = '1' );
         -- Added for pipelining
         read_data <= read_data_MEMWB;
   END PROCESS;
END behavior;
```

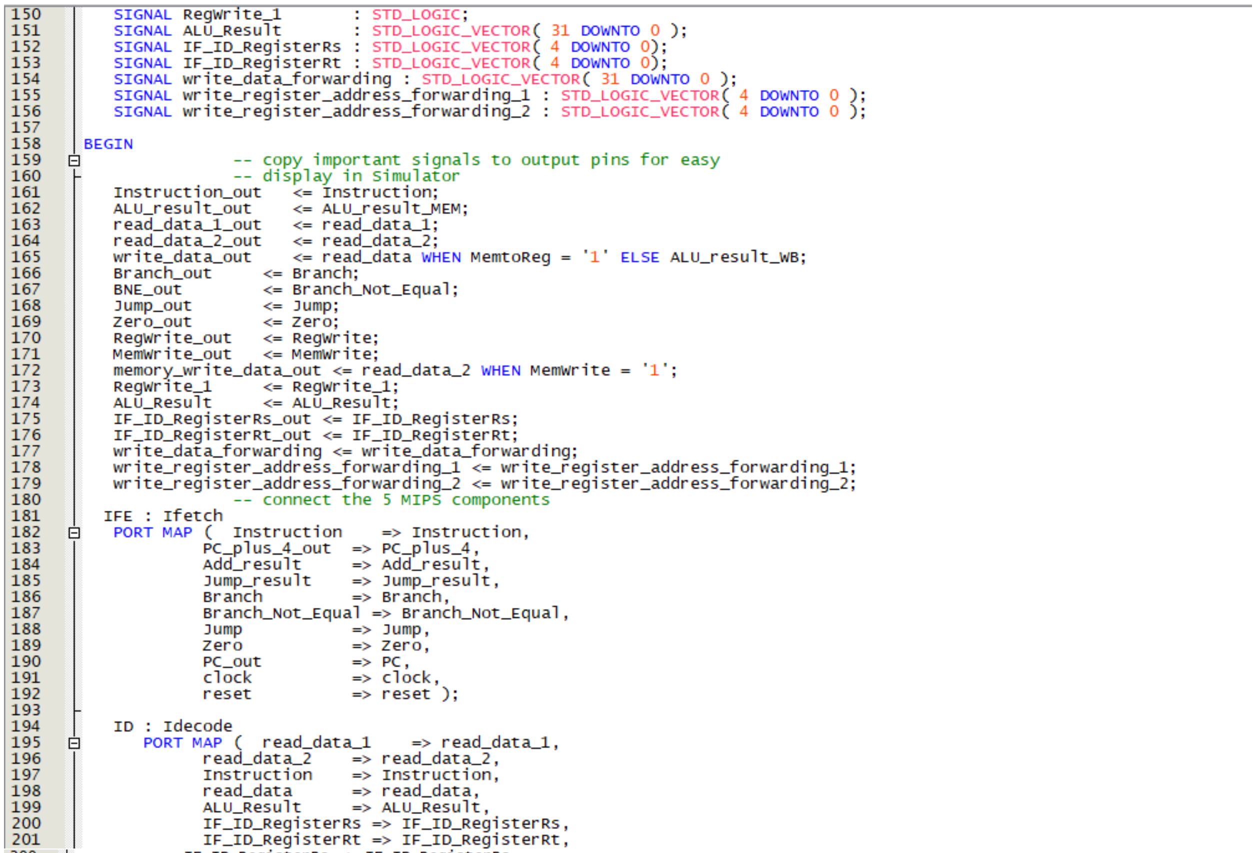
Code for Memory stage



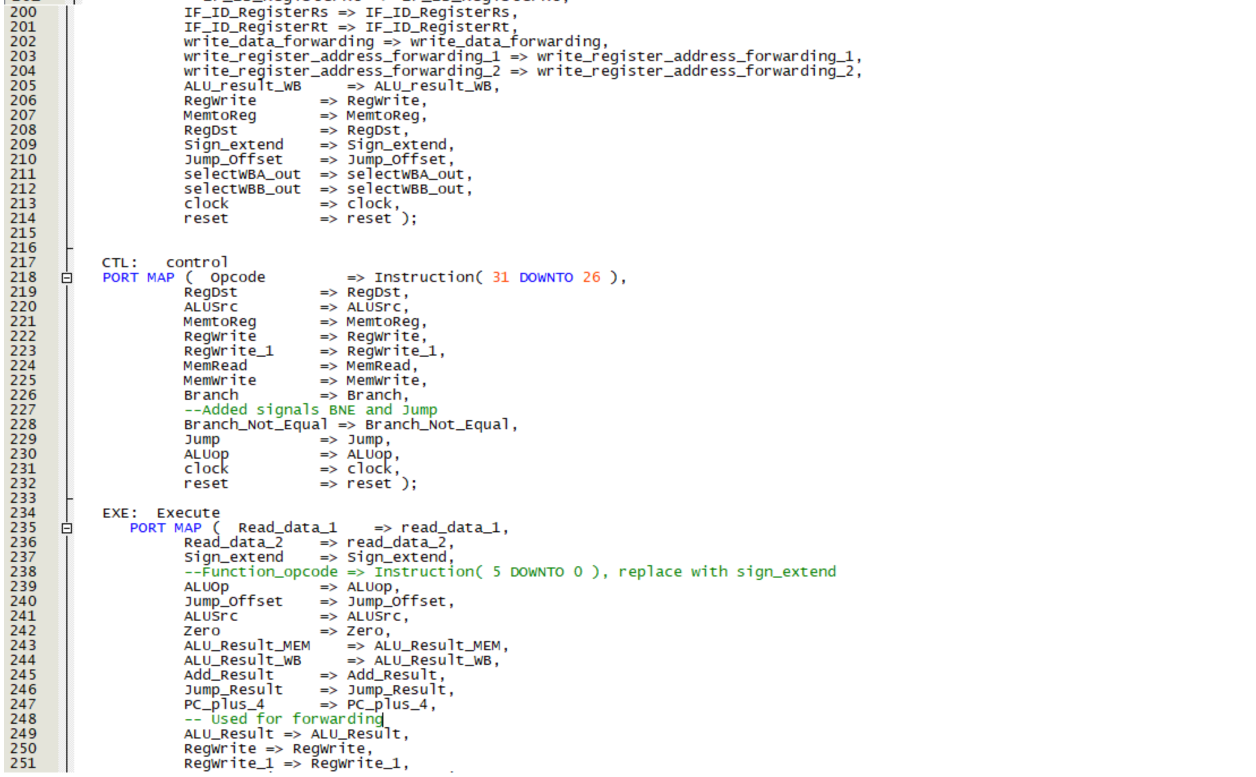
```vhdl
            -- Top Level Structural Model for MIPS Processor Core
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY MIPS IS

   PORT( reset, clock                                                : IN  STD_LOGIC;
      -- Output important signals to pins for easy display in Simulator
      PC                                                             : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0 );
      ALU_result_out, read_data_1_out, read_data_2_out, write_data_out,
      Instruction_out, memory_write_data_out, Ainput_out, Binput_out    : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
      --Added BNE_out and Jump_out
      Branch_out, Zero_out, Memwrite_out, BNE_out, Jump_out,
      Regwrite_out, selectWBA_out, selectWBB_out                     : OUT STD_LOGIC ;
      IF_ID_RegisterRs_out, IF_ID_RegisterRt_out                     : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0);
      ForwardA_ctl_out, ForwardB_ctl_out                             : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 ));

END   MIPS;

ARCHITECTURE structure OF MIPS IS

   COMPONENT Ifetch
            PORT(  Instruction : OUT    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
             PC_plus_4_out      : OUT    STD_LOGIC_VECTOR( 9 DOWNTO 0 );
             Add_result         : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
             --Added Jump_result, brnach not equal, and Jump
             Jump_result        : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
             Branch_Not_Equal   : IN  STD_LOGIC;
             Jump               : IN  STD_LOGIC;
             Branch             : IN  STD_LOGIC;
             Zero               : IN  STD_LOGIC;
             PC_out             : OUT    STD_LOGIC_VECTOR( 9 DOWNTO 0 );
             clock,reset        : IN  STD_LOGIC );
   END COMPONENT;

   COMPONENT Idecode
         PORT(  read_data_1     : OUT    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
             read_data_2        : OUT    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
             Instruction        : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
             read_data          : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
             ALU_result_WB      : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
             RegWrite, MemtoReg: IN  STD_LOGIC;
             RegDst             : IN  STD_LOGIC;
             Sign_extend        : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
             --Added for forwarding
             ALU_Result        : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
             IF_ID_RegisterRs : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0); -- EX
             IF_ID_RegisterRt : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0); -- EX
             write_data_forwarding : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- not pipelined
             write_register_address_forwarding_1 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
             write_register_address_forwarding_2 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
```



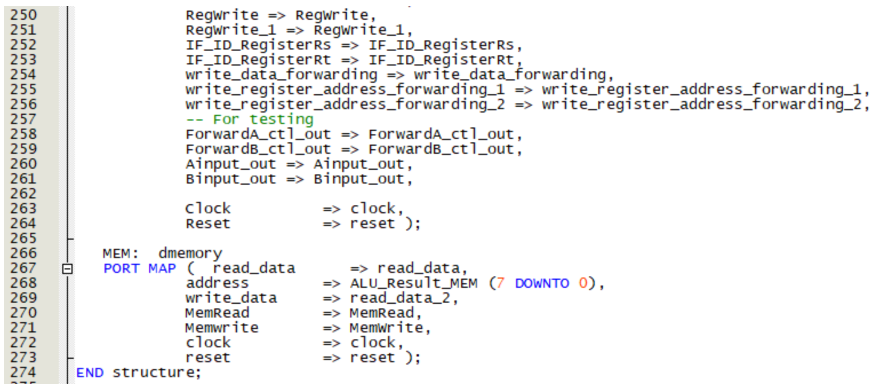
```
            write_data_forwarding : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- not pipelined
            write_register_address_forwarding_1 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
            write_register_address_forwarding_2 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
            --Added Jump_offset
            Jump_Offset        : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0 );
            --FoR TESTING PURPOSES
            selectWBA_out :OUT STD_LOGIC;
            selectWBB_out :OUT STD_LOGIC;

            clock, reset       : IN  STD_LOGIC );
    END COMPONENT;

    COMPONENT control
         PORT(  Opcode              : IN     STD_LOGIC_VECTOR( 5 DOWNTO 0 );
                RegDst              : OUT    STD_LOGIC;
                ALUSrc              : OUT    STD_LOGIC;
                MemtoReg            : OUT    STD_LOGIC;
                RegWrite            : OUT    STD_LOGIC;
                --Added for forwarding
                RegWrite_1          : OUT    STD_LOGIC;

                MemRead             : OUT    STD_LOGIC;
                MemWrite            : OUT    STD_LOGIC;
                Branch              : OUT    STD_LOGIC;
                --Added branch on not equal and Jump
                Branch_Not_Equal    : OUT    STD_LOGIC;
                Jump                : OUT    STD_LOGIC;

                ALUop               : OUT    STD_LOGIC_VECTOR( 1 DOWNTO 0 );
                clock, reset        : IN     STD_LOGIC );
    END COMPONENT;

    COMPONENT  Execute
            PORT(  Read_data_1        : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                   Read_data_2        : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                   Sign_Extend        : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                   --Function_opcode : IN  STD_LOGIC_VECTOR( 5 DOWNTO 0 ); replaced by sign extend
                   ALUOp              : IN  STD_LOGIC_VECTOR( 1 DOWNTO 0 );
                   --Added Jump_Offset input to calculate jump value
                   Jump_Offset        : IN  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
                   ALUSrc             : IN  STD_LOGIC;
                   Zero               : OUT STD_LOGIC;
                   -- Adding Jump_Result to allow for Jump command
                   Jump_result        : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
                   -- Used for forwarding
                   ALU_Result        : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                   RegWrite, RegWrite_1 : IN STD_LOGIC;
                   IF_ID_RegisterRs : IN STD_LOGIC_VECTOR( 4 DOWNTO 0);
                   IF_ID_RegisterRt : IN STD_LOGIC_VECTOR( 4 DOWNTO 0);
                   write_data_forwarding : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                   write_register_address_forwarding_1 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
                   write_register_address_forwarding_2 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
                 write_register_address_forwarding_1 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
                 write_register_address_forwarding_2 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
                 --For testing
                 ForwardA_ctl_out  : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
                 ForwardB_ctl_out  : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
                 Ainput_out        : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                 Binput_out        : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );

                 ALU_Result_MEM    : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                 ALU_result_WB     : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                 Add_Result        : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
                 PC_plus_4         : IN  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
                 clock, reset      : IN  STD_LOGIC );
    END COMPONENT;


    COMPONENT dmemory
         PORT(  read_data          : OUT    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                address            : IN     STD_LOGIC_VECTOR( 7 DOWNTO 0 );
                write_data         : IN     STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                MemRead, Memwrite  : IN     STD_LOGIC;
                Clock,reset        : IN     STD_LOGIC );
    END COMPONENT;

                  -- declare signals used to connect VHDL components
    SIGNAL PC_plus_4      : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
    SIGNAL read_data_1    : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
    SIGNAL read_data_2    : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
    SIGNAL Sign_Extend    : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
    SIGNAL Add_result     : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
    SIGNAL ALU_result_WB     : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
    SIGNAL ALU_result_MEM    : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
    SIGNAL read_data      : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
    SIGNAL ALUSrc         : STD_LOGIC;
    SIGNAL Branch         : STD_LOGIC;
    --Added signals BNE, Jump, Jump_result, Jump_offset
    SIGNAL Branch_Not_Equal : STD_LOGIC;
    SIGNAL Jump           : STD_LOGIC;
    SIGNAL Jump_result    : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
    SIGNAL Jump_Offset    : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
    SIGNAL RegDst         : STD_LOGIC;
    SIGNAL RegWrite       : STD_LOGIC;
    SIGNAL Zero           : STD_LOGIC;
    SIGNAL MemWrite       : STD_LOGIC;
    SIGNAL MemtoReg       : STD_LOGIC;
    SIGNAL MemRead        : STD_LOGIC;
    SIGNAL ALUop          : STD_LOGIC_VECTOR(  1 DOWNTO 0 );
    SIGNAL Instruction    : STD_LOGIC_VECTOR( 31 DOWNTO 0 );

    --Added for forwarding
    SIGNAL RegWrite_1        : STD_LOGIC;
    SIGNAL ALU_Result        : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
```



```
    SIGNAL RegWrite_1        : STD_LOGIC;
    SIGNAL ALU_Result        : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
    SIGNAL IF_ID_RegisterRs : STD_LOGIC_VECTOR( 4 DOWNTO 0);
    SIGNAL IF_ID_RegisterRt : STD_LOGIC_VECTOR( 4 DOWNTO 0);
    SIGNAL write_data_forwarding : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
    SIGNAL write_register_address_forwarding_1 : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
    SIGNAL write_register_address_forwarding_2 : STD_LOGIC_VECTOR( 4 DOWNTO 0 );

BEGIN
                -- copy important signals to output pins for easy
                -- display in Simulator
    Instruction_out    <= Instruction;
    ALU_result_out     <= ALU_result_MEM;
    read_data_1_out    <= read_data_1;
    read_data_2_out    <= read_data_2;
    write_data_out     <= read_data WHEN MemtoReg = '1' ELSE ALU_result_WB;
    Branch_out      <= Branch;
    BNE_out         <= Branch_Not_Equal;
    Jump_out        <= Jump;
    Zero_out        <= Zero;
    RegWrite_out    <= RegWrite;
    MemWrite_out    <= MemWrite;
    memory_write_data_out <= read_data_2 WHEN MemWrite = '1';
    RegWrite_1      <= RegWrite_1;
    ALU_Result      <= ALU_Result;
    IF_ID_RegisterRs_out <= IF_ID_RegisterRs;
    IF_ID_RegisterRt_out <= IF_ID_RegisterRt;
    write_data_forwarding <= write_data_forwarding;
    write_register_address_forwarding_1 <= write_register_address_forwarding_1;
    write_register_address_forwarding_2 <= write_register_address_forwarding_2;
                -- connect the 5 MIPS components
   IFE : Ifetch
    PORT MAP (  Instruction     => Instruction,
             PC_plus_4_out  => PC_plus_4,
             Add_result     => Add_result,
             Jump_result    => Jump_result,
             Branch         => Branch,
             Branch_Not_Equal => Branch_Not_Equal,
             Jump           => Jump,
             Zero           => Zero,
             PC_out         => PC,
             clock          => clock,
             reset          => reset );

    ID : Idecode
       PORT MAP (  read_data_1    => read_data_1,
             read_data_2    => read_data_2,
             Instruction    => Instruction,
             read_data      => read_data,
             ALU_Result     => ALU_Result,
             IF_ID_RegisterRs => IF_ID_RegisterRs,
             IF_ID_RegisterRt => IF_ID_RegisterRt,
```



```
             IF_ID_RegisterRs => IF_ID_RegisterRs,
             IF_ID_RegisterRt => IF_ID_RegisterRt,
             write_data_forwarding => write_data_forwarding,
             write_register_address_forwarding_1 => write_register_address_forwarding_1,
             write_register_address_forwarding_2 => write_register_address_forwarding_2,
             ALU_result_WB     => ALU_result_WB,
             RegWrite        => RegWrite,
             MemtoReg        => MemtoReg,
             RegDst          => RegDst,
             Sign_extend     => Sign_extend,
             Jump_Offset     => Jump_Offset,
             selectWBA_out   => selectWBA_out,
             selectWBB_out   => selectWBB_out,
             clock           => clock,
             reset           => reset );


   CTL:   control
   PORT MAP (  Opcode          => Instruction( 31 DOWNTO 26 ),
             RegDst          => RegDst,
             ALUSrc          => ALUSrc,
             MemtoReg        => MemtoReg,
             RegWrite        => RegWrite,
             RegWrite_1      => RegWrite_1,
             MemRead         => MemRead,
             MemWrite        => MemWrite,
             Branch          => Branch,
             --Added signals BNE and Jump
             Branch_Not_Equal => Branch_Not_Equal,
             Jump            => Jump,
             ALUop           => ALUop,
             clock           => clock,
             reset           => reset );

   EXE:  Execute
      PORT MAP (  Read_data_1     => read_data_1,
             Read_data_2     => read_data_2,
             Sign_extend     => Sign_extend,
             --Function_opcode => Instruction( 5 DOWNTO 0 ), replace with sign_extend
             ALUOp           => ALUop,
             Jump_Offset     => Jump_Offset,
             ALUSrc          => ALUSrc,
             Zero            => Zero,
             ALU_Result_MEM     => ALU_Result_MEM,
             ALU_Result_WB      => ALU_Result_WB,
             Add_Result      => Add_Result,
             Jump_Result     => Jump_Result,
             PC_plus_4       => PC_plus_4,
             -- Used for forwarding
             ALU_Result => ALU_Result,
             RegWrite => RegWrite,
             RegWrite_1 => RegWrite_1,
```



```
            RegWrite => RegWrite,
            RegWrite_1 => RegWrite_1,
            IF_ID_RegisterRs => IF_ID_RegisterRs,
            IF_ID_RegisterRt => IF_ID_RegisterRt,
            write_data_forwarding => write_data_forwarding,
            write_register_address_forwarding_1 => write_register_address_forwarding_1,
            write_register_address_forwarding_2 => write_register_address_forwarding_2,
            -- For testing
            ForwardA_ctl_out => ForwardA_ctl_out,
            ForwardB_ctl_out => ForwardB_ctl_out,
            Ainput_out => Ainput_out,
            Binput_out => Binput_out,

            Clock          => clock,
            Reset          => reset );

   MEM:  dmemory
   PORT MAP (  read_data      => read_data,
            address        => ALU_Result_MEM (7 DOWNTO 0),
            write_data     => read_data_2,
            MemRead        => MemRead,
            Memwrite       => MemWrite,
            clock          => clock,
            reset          => reset );
END structure;
```

Code for Mips definition file