

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 instruction. 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 instruction and input A uses previous instruction
09: 00683825; -- or $7, $3, $8 ; input A uses next to last instruction 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.**.

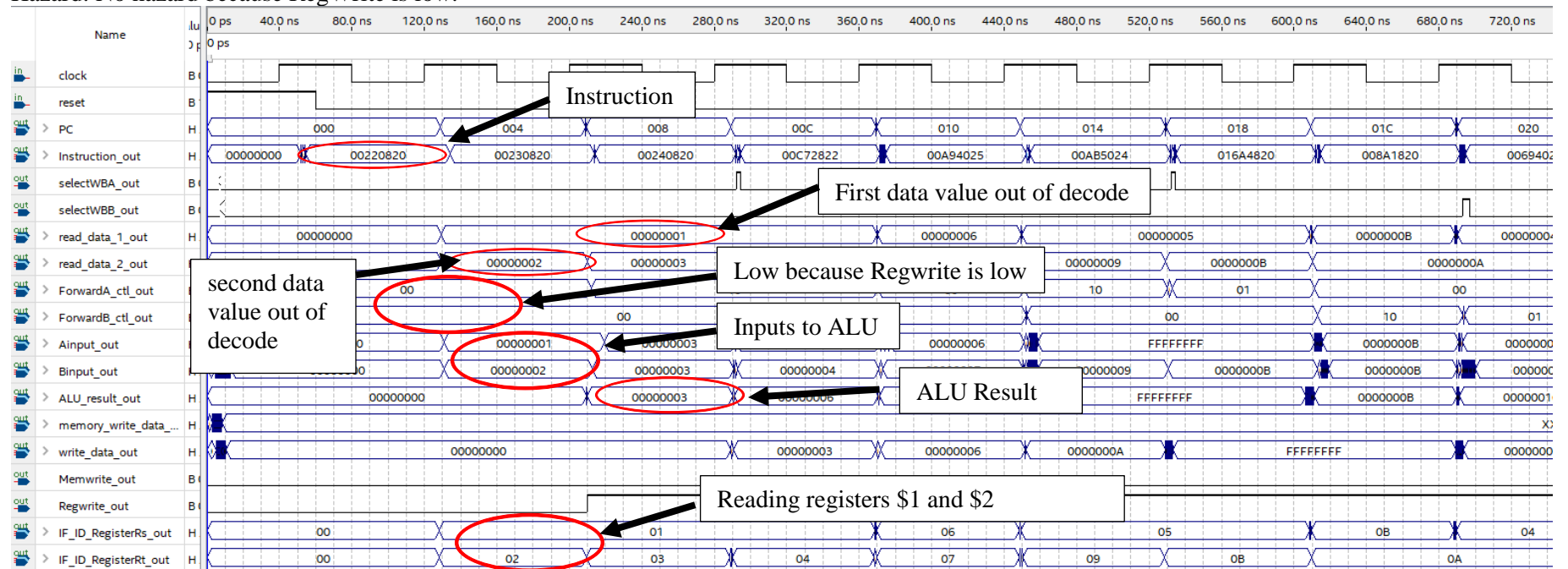
Signal Name	Descriptor
Clock	Clock signal, 80ns period
Reset	Reset signal to start system
PC	Program Counter
Instruction	MIPS instruction in hex
SelectWBA	Mux control signal for data 1 output from decode
SelectWBB	Mux control signal for data 2 output from decode
Read data 1	Data 1 output from decode 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.

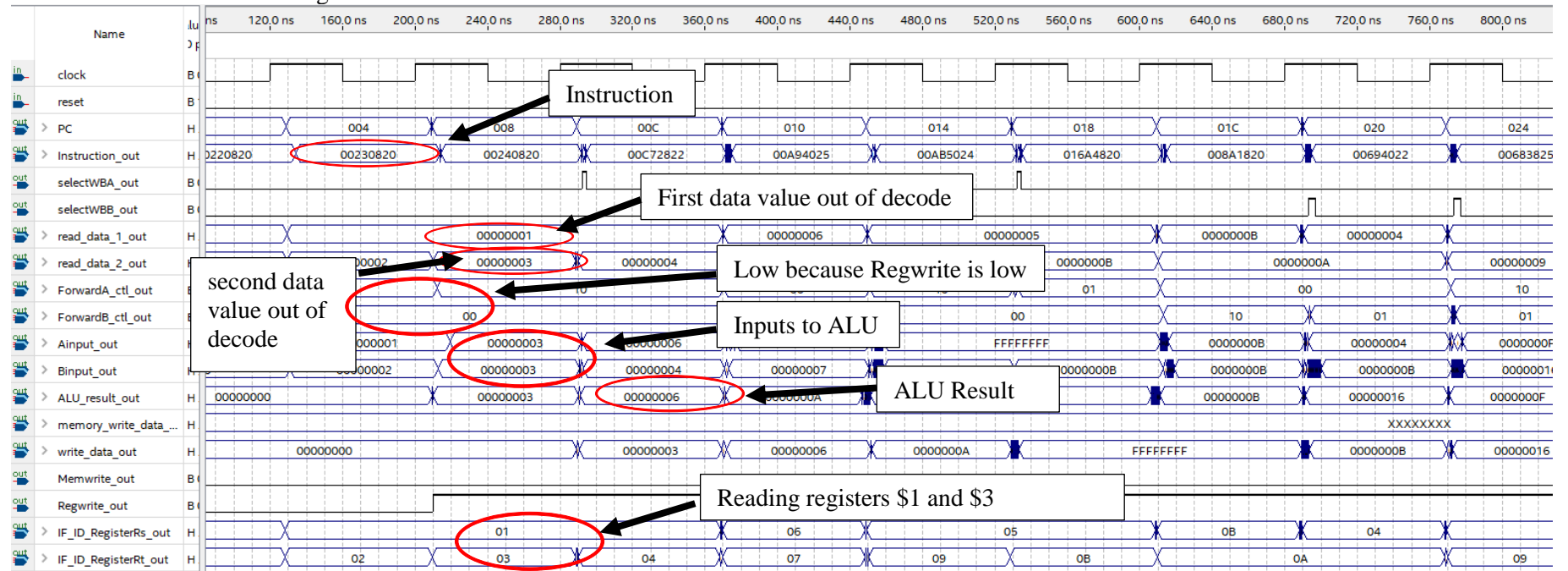
Instruction: add \$1, \$1, \$2

Hazard: No hazard because RegWrite is low.



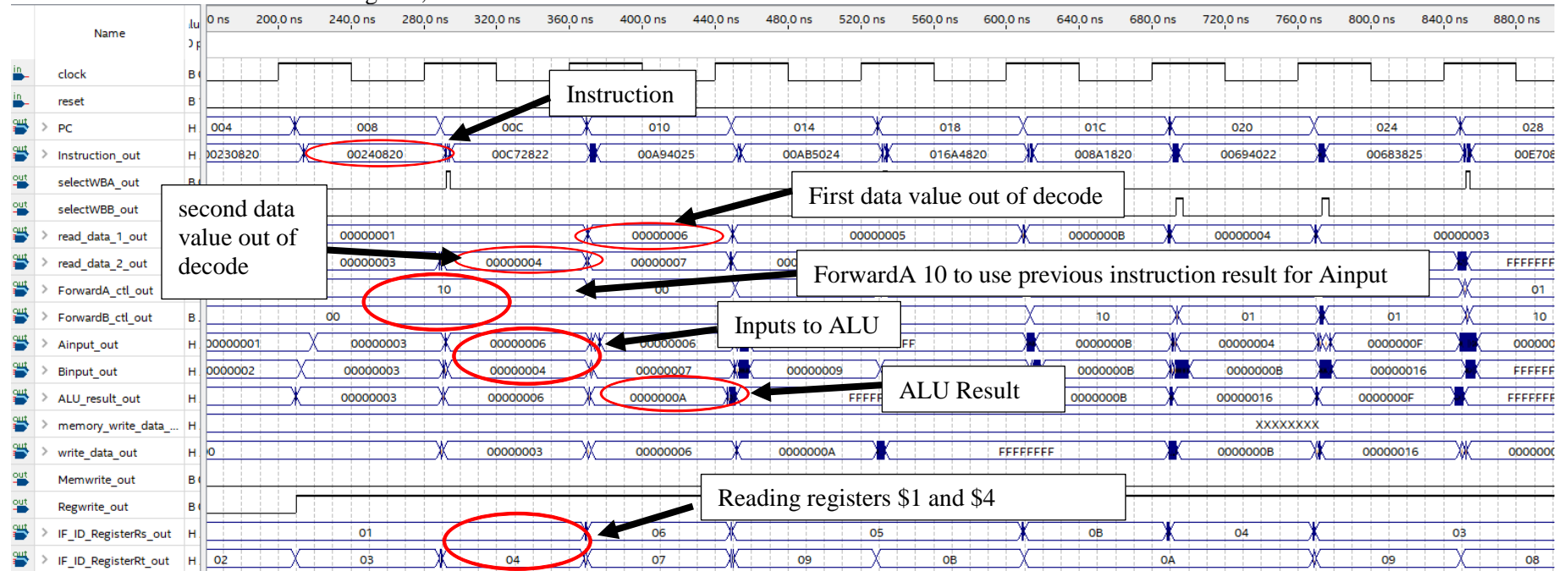
Instruction: add \$1, \$1, \$3

Hazard: No hazard because Regwrite is low.



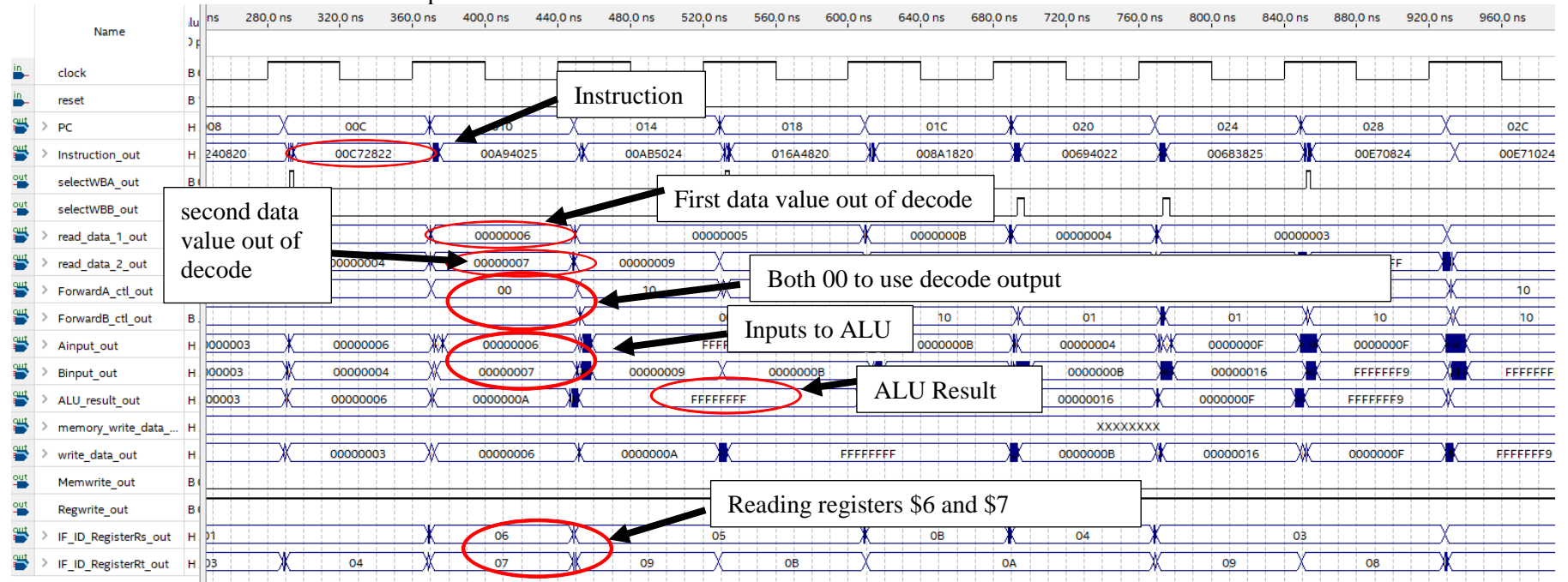
Instruction: add \$1, \$1, \$4

Hazard: Read and write are same register, forward write data to read data.



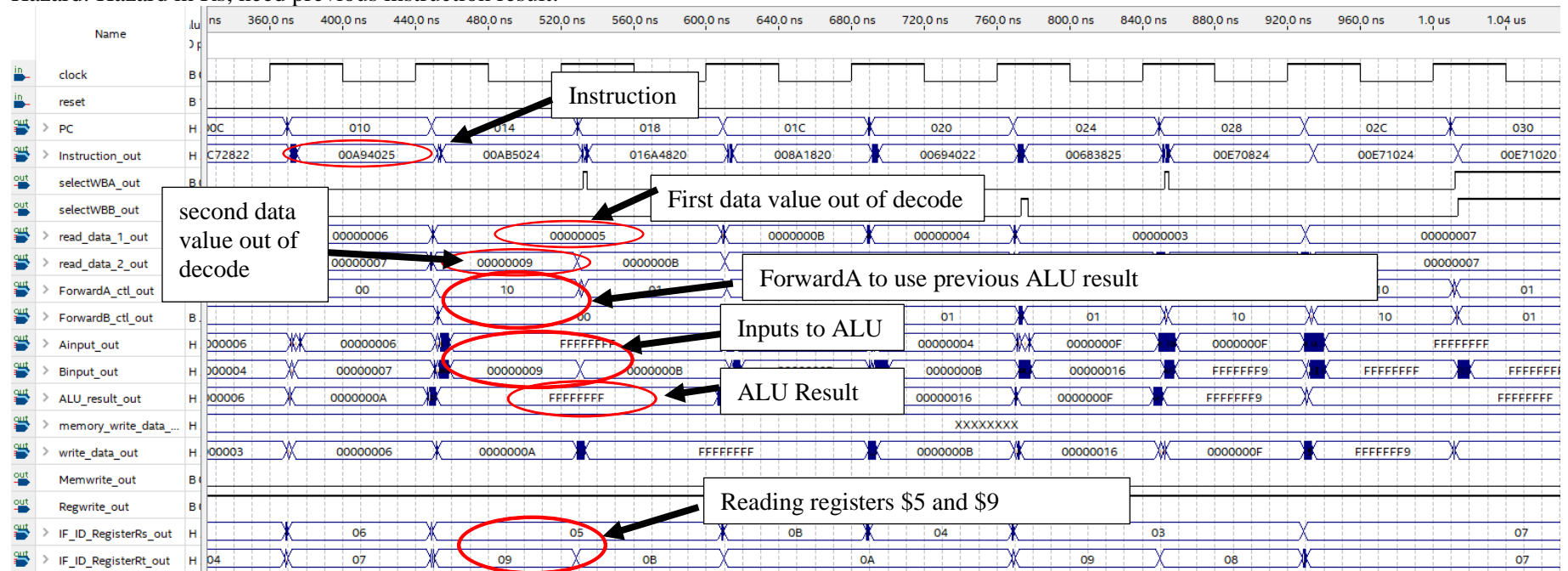
Instruction: sub \$5, \$6, \$7

Hazard: No hazard because doesn't need previous instructions' results.



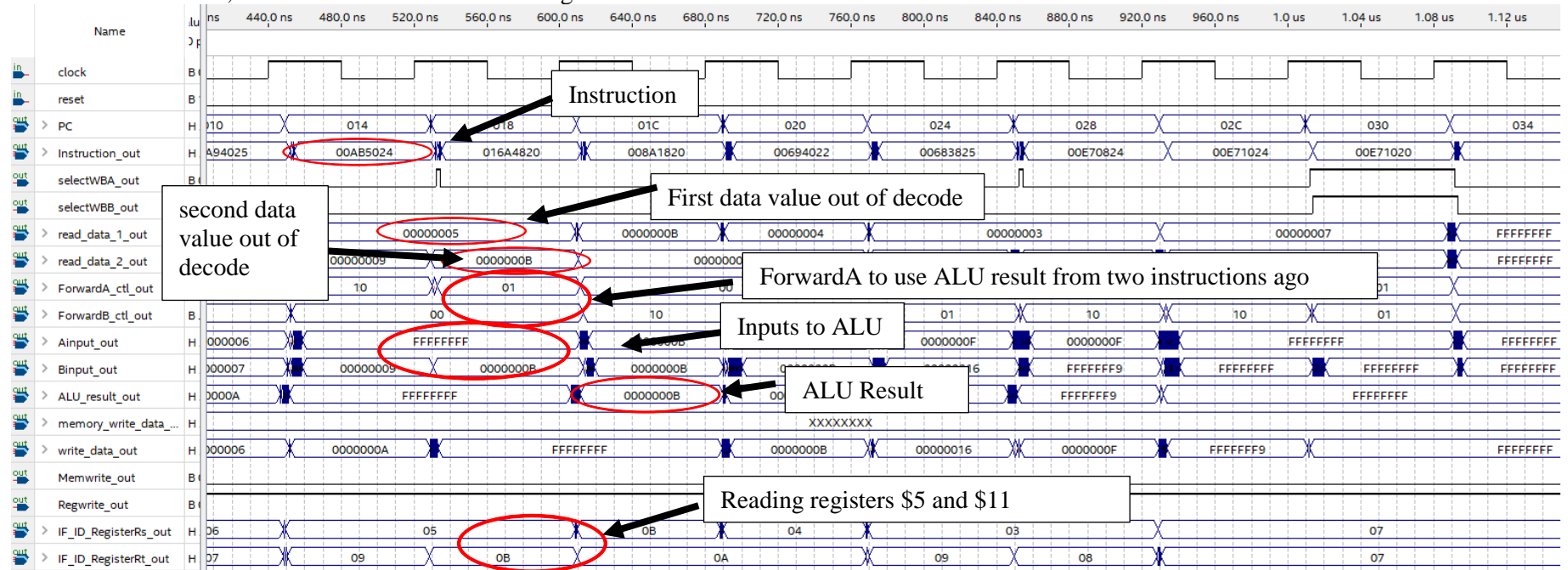
Instruction: or \$8, \$5, \$9

Hazard: Hazard in Rs, need previous instruction result.



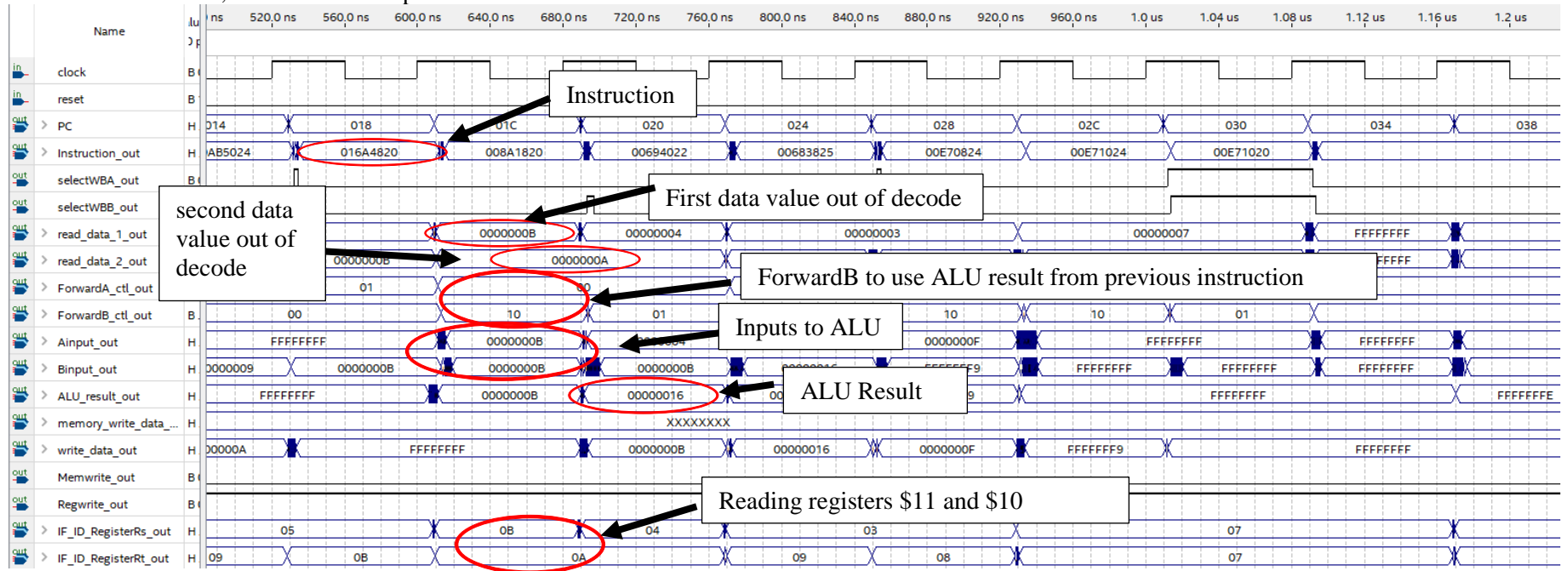
Instruction: and \$10, \$5, \$11

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



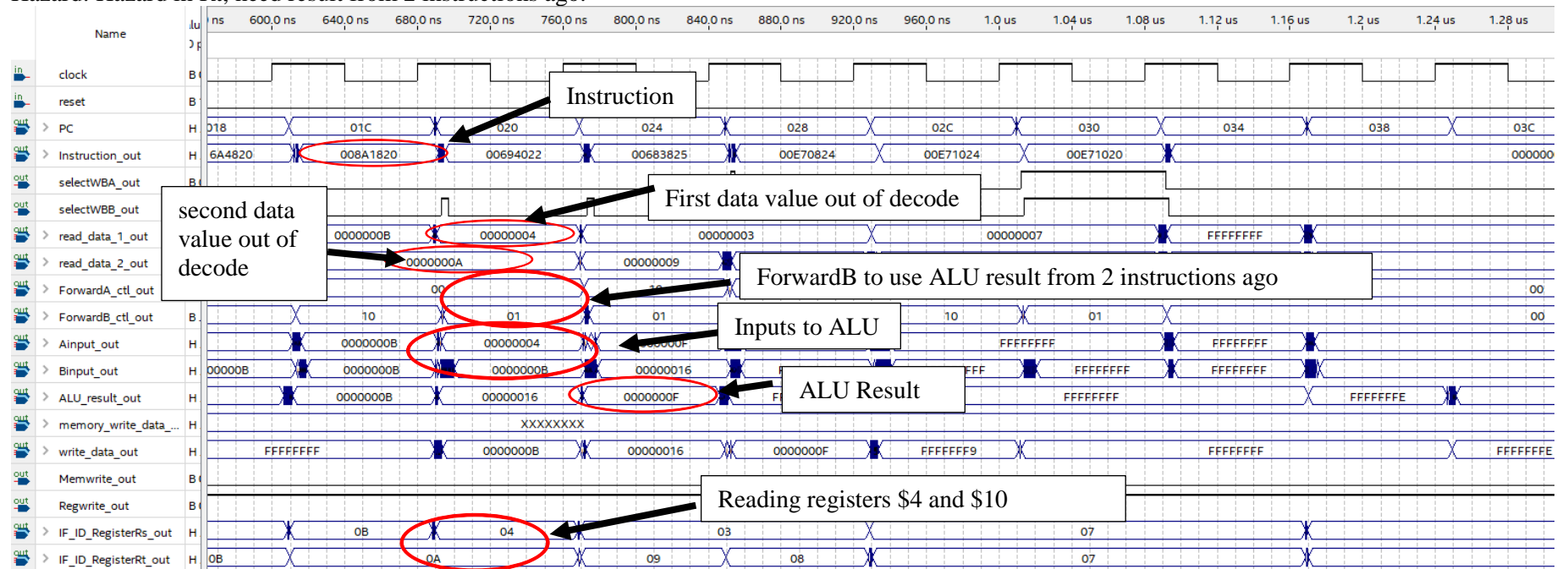
Instruction: add \$9, \$11, \$10

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



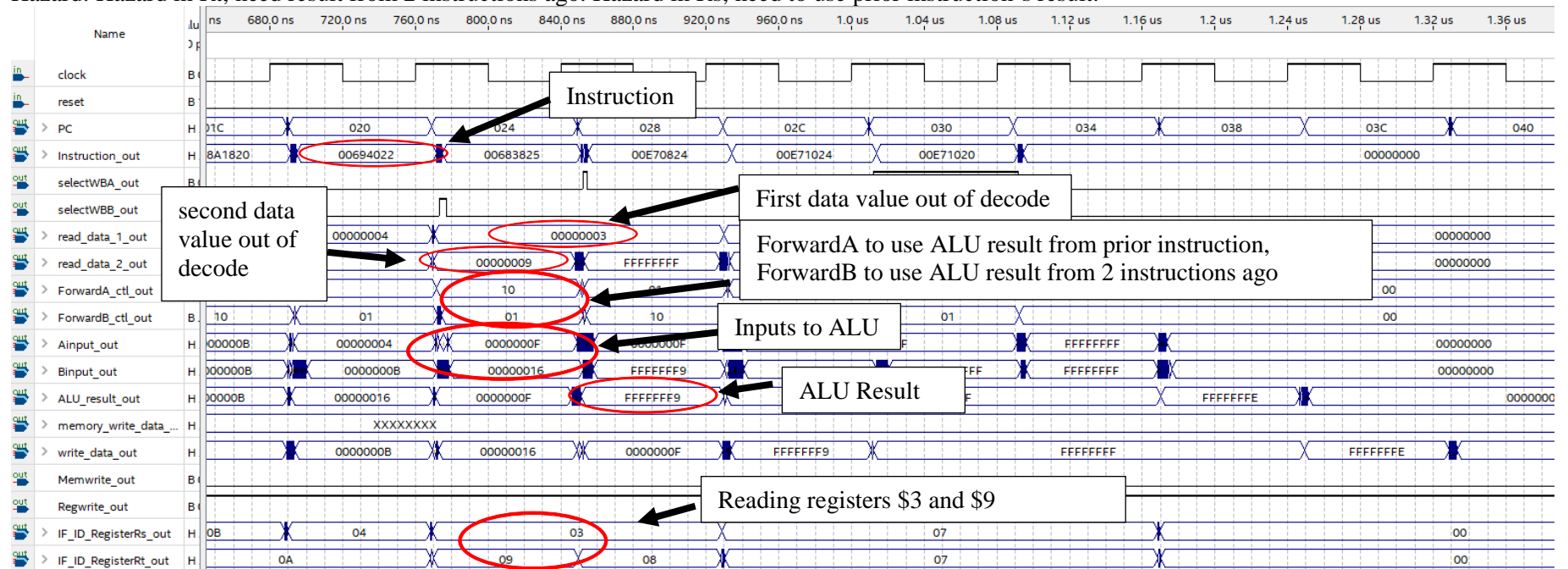
Instruction: add \$3, \$4, \$10

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



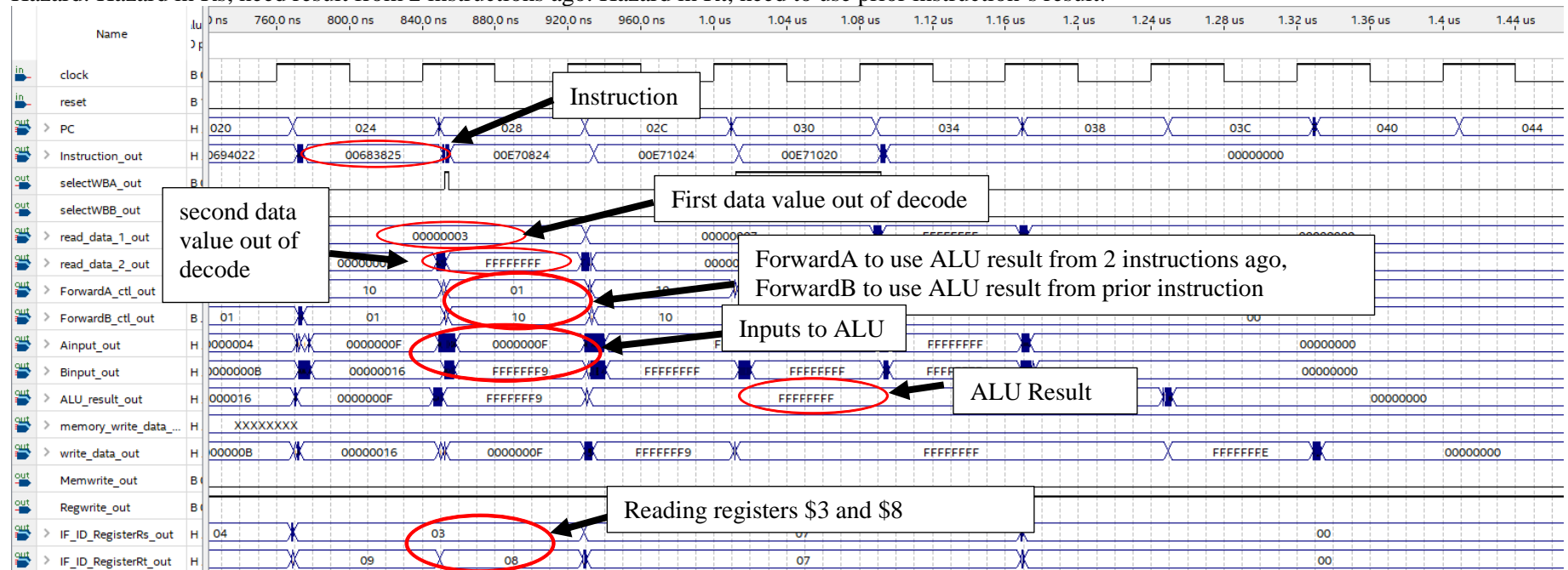
Instruction: sub \$8, \$3, \$9

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



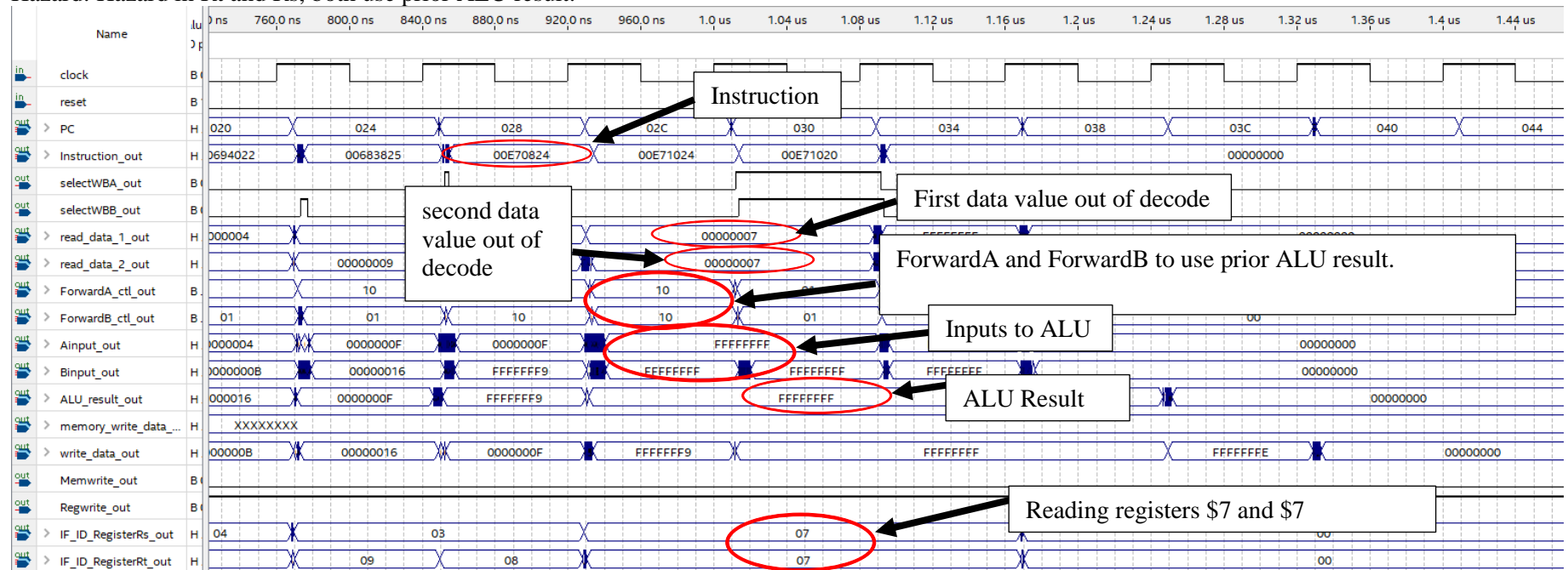
Instruction: or \$7, \$3, \$8

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



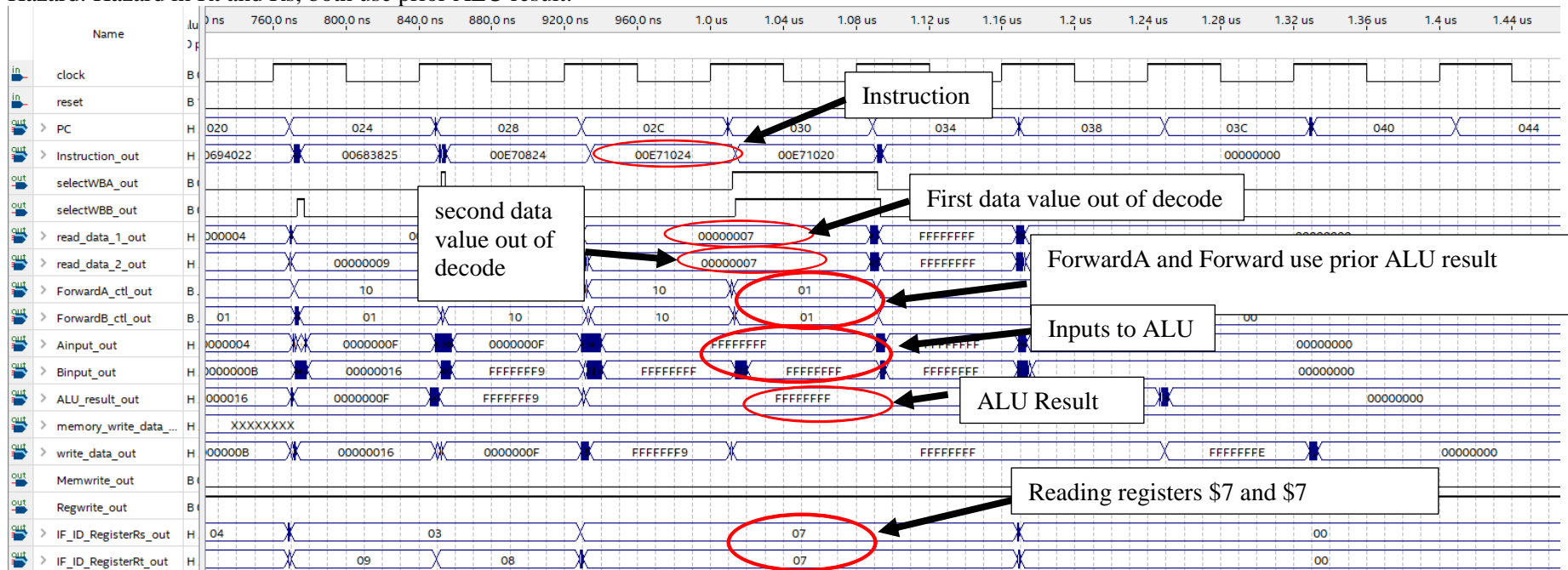
Instruction: and \$1, \$7, \$7

Hazard: Hazard in Rt and Rs, both use prior ALU result.



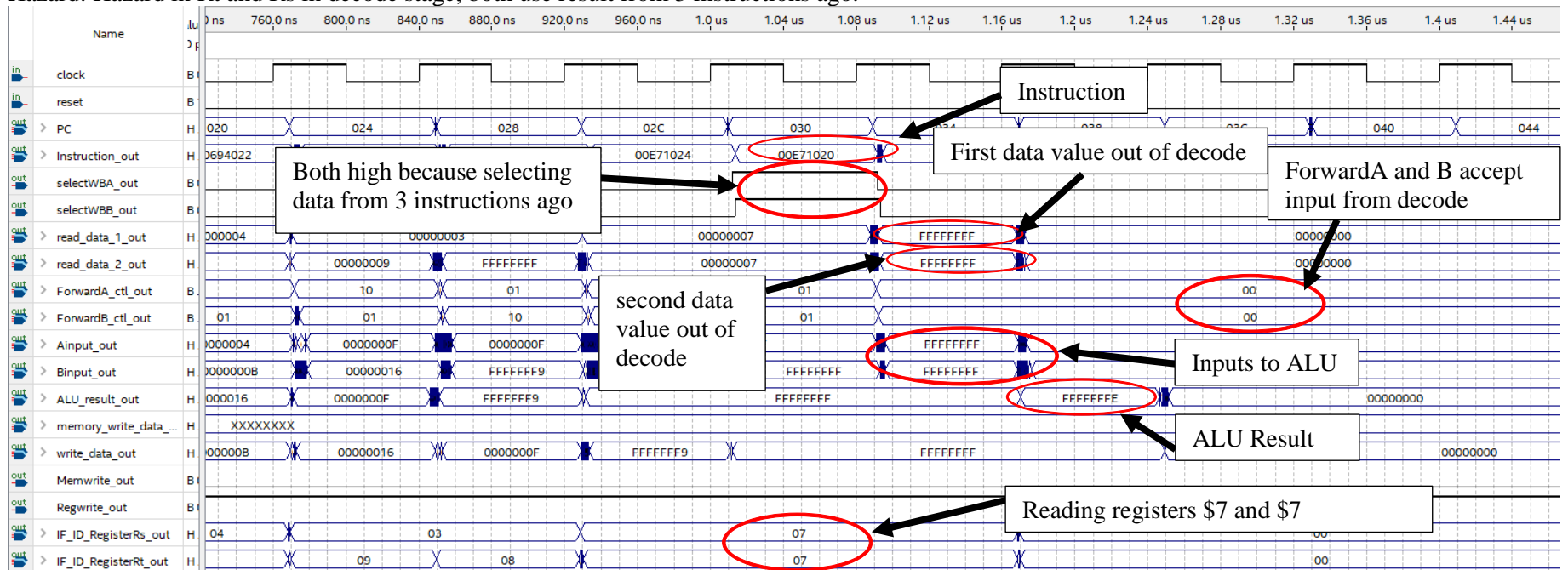
Instruction: and \$2, \$7, \$7

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

```

1  -- Ifetch module (provides the PC and instruction
2  -- memory for the MIPS computer)
3  LIBRARY IEEE;
4  USE IEEE.STD_LOGIC_1164.ALL;
5  USE IEEE.STD_LOGIC_ARITH.ALL;
6  USE IEEE.STD_LOGIC_UNSIGNED.ALL;
7  LIBRARY altera_mf;
8  USE altera_mf.altera_mf_components.all;
9
10 ENTITY Ifetch IS
11 PORT(
12     SIGNAL Instruction : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- ID
13     SIGNAL PC_plus_4_out : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0 ); -- EX
14     -- PC_out goes nowhere, just used as out, will need it later for stalls
15     SIGNAL PC_out : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0 );
16     SIGNAL Add_result : IN STD_LOGIC_VECTOR( 7 DOWNTO 0 );
17     --Added Jump_result value
18     SIGNAL Jump_result : IN STD_LOGIC_VECTOR( 7 DOWNTO 0 );
19     SIGNAL Branch : IN STD_LOGIC;
20     --Added BNE and Jump inputs
21     SIGNAL Branch_Not_Equal : IN STD_LOGIC;
22     SIGNAL Jump : IN STD_LOGIC;
23     SIGNAL Zero : IN STD_LOGIC;
24     SIGNAL clock, reset : IN STD_LOGIC);
25 END Ifetch;
26 ARCHITECTURE behavior OF Ifetch IS
27     SIGNAL Instruction_IFID : STD_LOGIC_VECTOR(31 DOWNTO 0 );
28     SIGNAL PC, PC_plus_4_IFID, PC_plus_4_IDEX : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
29     SIGNAL next_PC, Mem_Addr : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
30 BEGIN
31     --ROM for Instruction Memory
32     inst_memory: altsyncram
33
34     GENERIC MAP (
35         operation_mode => "ROM",
36         width_a => 32,
37         widthad_a => 8,
38         lpm_type => "altsyncram",
39         outdata_reg_a => "UNREGISTERED",
40         init_file => "Lab08program.MIF",
41         intended_device_family => "cyclone"
42     )
43     PORT MAP (
44         clock0 => clock,
45         address_a => Mem_Addr,
46         q_a => Instruction );
47
48     -- Instructions always start on word address - not byte
49     PC(1 DOWNTO 0) <= "00";
50
51     -- copy output signals - allows read inside module
52     PC_out <= PC;
53
54     -- send address to inst. memory address register
55     Mem_Addr <= next_PC;
56
57     -- Adder to increment PC by 4
58     PC_plus_4_IFID( 9 DOWNTO 2 ) <= PC( 9 DOWNTO 2 ) + 1;
59     PC_plus_4_IFID( 1 DOWNTO 0 ) <= "00";
60
61     -- Mux to select Branch Address or PC + 4
62     Next_PC <= X"00" WHEN Reset = '1'
63     --Added BNE = '1' AND Zero = '0' to allow branch on not equal
64     ELSE Add_result WHEN ( ( Branch = '1' ) AND ( Zero = '1' ) )
65     OR ( (Branch_Not_Equal = '1') AND (Zero = '0'))
66     --Added Jump
67     ELSE Jump_result WHEN (Jump = '1')
68     ELSE PC_plus_4_IFID( 9 DOWNTO 2 );
69
70 PROCESS
71 BEGIN
72     WAIT UNTIL ( clock'EVENT ) AND ( clock = '1' );
73     IF reset = '1' THEN
74         PC( 9 DOWNTO 2 ) <= "00000000" ;
75     ELSE
76         PC( 9 DOWNTO 2 ) <= next_PC;
77     END IF;
78     -- Added for pipelining
79     PC_plus_4_IDEX <= PC_plus_4_IFID;
80     PC_plus_4_out <= PC_plus_4_IDEX;
81 END PROCESS;
82 END behavior;

```

Code for Fetch stage

```

1      -- Idecode module (implements the register file for
2      -- the MIPS computer)
3      USE IEEE.STD_LOGIC_1164.ALL;
4      USE IEEE.STD_LOGIC_ARITH.ALL;
5      USE IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7      ENTITY Idecode IS
8      PORT( read_data_1 : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- EX
9            read_data_2 : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- EX
10           Sign_extend : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- EX
11           --Added Jump_Offset which will come straight from the jump instruction and goes straight into an adder
12           Jump_offset : OUT   STD_LOGIC_VECTOR( 9 DOWNTO 0 ); -- EX
13           --Added for forwarding
14           ALU_Result   : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
15           IF_ID_RegisterRs : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); -- EX
16           IF_ID_RegisterRt : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); -- EX
17           write_data_forwarding : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- not pipelined
18           write_register_address_forwarding_1 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
19           write_register_address_forwarding_2 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
20           --Altered Instruction for pipelining purposes
21           Instruction : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
22           read_data   : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
23           ALU_result_wb : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
24           Regwrite     : IN    STD_LOGIC;
25           MemtoReg     : IN    STD_LOGIC;
26           RegDst       : IN    STD_LOGIC;
27
28           --FOR TESTING PURPOSES
29           selectwBA_out : OUT STD_LOGIC;
30           selectwBB_out : OUT STD_LOGIC;
31
32           clock,reset : IN    STD_LOGIC );
33      END Idecode;
34
35      ARCHITECTURE behavior OF Idecode IS
36      TYPE register_file IS ARRAY ( 0 TO 31 ) OF STD_LOGIC_VECTOR( 31 DOWNTO 0 );
37
38      SIGNAL register_array          : register_file;
39      SIGNAL write_register_address  : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
40      SIGNAL write_register_address_IDEX : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
41      SIGNAL write_register_address_EXMEM : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
42      SIGNAL write_register_address_MEMWB : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
43      SIGNAL write_data              : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
44      SIGNAL read_register_1_address : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
45      SIGNAL read_register_2_address : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
46      SIGNAL write_register_address_1 : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
47      SIGNAL write_register_address_0 : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
48      SIGNAL Instruction_immediate_value : STD_LOGIC_VECTOR( 15 DOWNTO 0 );
49      SIGNAL read_data_1_IDEX          : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
50      SIGNAL read_data_2_IDEX          : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
51      SIGNAL Sign_extend_IDEX          : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
52      SIGNAL read_data_1_IDEX          : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
53      SIGNAL read_data_2_IDEX          : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
54      SIGNAL Sign_extend_IDEX          : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
55      SIGNAL Jump_offset_IDEX          : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
56      -- added for forwarding
57      SIGNAL selectwBA : STD_LOGIC;
58      SIGNAL selectwBB : STD_LOGIC;
59
60      BEGIN
61      read_register_1_address <= Instruction( 25 DOWNTO 21 );
62      read_register_2_address <= Instruction( 20 DOWNTO 16 );
63      write_register_address_1 <= Instruction( 15 DOWNTO 11 );
64      write_register_address_0 <= Instruction( 20 DOWNTO 16 );
65      Instruction_immediate_value <= Instruction( 15 DOWNTO 0 );
66
67      -- Read Register 1 Operation including forwarding capability
68      read_data_1_IDEX <= register_array( CONV_INTEGER( read_register_1_address ) )
69      WHEN selectwBA = '0' ELSE write_data;
70
71      -- Read Register 2 Operation including forwarding capability
72      read_data_2_IDEX <= register_array( CONV_INTEGER( read_register_2_address ) )
73      WHEN selectwBB = '0' ELSE write_data;
74
75      -- Mux for Register Write Address
76      write_register_address_IDEX <= write_register_address_1
77      WHEN RegDst = '1' ELSE write_register_address_0;
78
79      -- Mux to bypass data memory for Rformat instructions
80      write_data <= ALU_result_wb( 31 DOWNTO 0 )
81      WHEN ( MemtoReg = '0' ) ELSE read_data;
82
83      -- Sign Extend 16-bits to 32-bits
84      Sign_extend_IDEX <= X"0000" & Instruction_immediate_value
85      WHEN Instruction_immediate_value(15) = '0'
86      ELSE X"FFFF" & Instruction_immediate_value;
87
88      -- Jump_Offset for jump calculation
89      Jump_offset_IDEX <= Instruction( 7 DOWNTO 0 ) & "00";
90
91      --FOR TESTING PURPOSES
92      selectwBA_out <= selectwBA;
93      selectwBB_out <= selectwBB;
94
95      --for forwarding purposes
96      write_data_forwarding <= write_data;
97      write_register_address_forwarding_2 <= write_register_address; -- newer instruction
98      write_register_address_forwarding_1 <= write_register_address_MEMWB; -- older instruction
99
100     --This forwarding logic unit checks for reading from a register that is being written to
101     -- due to a previous instruction. selectwBA and selectwBB are selects for two muxes
102     PROCESS ( Regwrite, write_register_address, read_register_1_address, read_register_2_address ) IS

```

```

99 -- due to a previous instruction, selectwBA and selectwBB are selects for two muxes
100 PROCESS ( Regwrite, write_register_address, read_register_1_address, read_register_2_address ) IS
101 BEGIN
102     IF (Regwrite = '1' AND write_register_address = read_register_1_address AND read_register_1_address /= "0000") THEN
103         selectwBA <= '1';
104     ELSE
105         selectwBA <= '0';
106     END IF;
107     IF (Regwrite = '1' AND write_register_address = read_register_2_address AND read_register_2_address /= "0000") THEN
108         selectwBB <= '1';
109     ELSE
110         selectwBB <= '0';
111     END IF;
112 END PROCESS;
113
114
115 PROCESS
116 BEGIN
117     WAIT UNTIL clock'EVENT AND clock = '1';
118     IF reset = '1' THEN
119         -- Initial register values on reset are register = reg#
120         -- use loop to automatically generate reset logic
121         -- for all registers
122         FOR i IN 0 TO 31 LOOP
123             register_array(i) <= CONV_STD_LOGIC_VECTOR( i, 32 );
124         END LOOP;
125         -- Write back to register - don't write to register 0
126     ELSEIF Regwrite = '1' AND write_register_address /= 0 THEN
127         register_array( CONV_INTEGER( write_register_address )) <= write_data;
128     END IF;
129     --Pipelining
130     read_data_1 <= read_data_1_INDEX;
131     read_data_2 <= read_data_2_INDEX;
132     sign_extend <= Sign_extend_INDEX;
133     Jump_Offset <= Jump_Offset_INDEX;
134     write_register_address_EXMEM <= write_register_address_INDEX;
135     write_register_address_MEMWB <= write_register_address_EXMEM;
136     write_register_address <= write_register_address_MEMWB;
137     --Used for forwarding
138     IF_ID_RegisterRs <= read_register_1_address;
139     IF_ID_RegisterRt <= read_register_2_address;
140
141 END PROCESS;
142 END behavior;

```

Code for Decode stage

```

1 -- control module (implements MIPS control unit)
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 USE IEEE.STD_LOGIC_ARITH.ALL;
5 USE IEEE.STD_LOGIC_SIGNED.ALL;
6
7 ENTITY control IS
8 PORT(
9     Opcode      : IN  STD_LOGIC_VECTOR( 5 DOWNTO 0 );
10    Opcode_out   : OUT STD_LOGIC_VECTOR( 5 DOWNTO 0 );
11    RegDst       : OUT STD_LOGIC; -- ID
12    ALUSrc       : OUT STD_LOGIC; -- EX
13    MemtoReg     : OUT STD_LOGIC; -- WB
14    Regwrite     : OUT STD_LOGIC; -- WB
15    --For forwarding
16    Regwrite_1   : OUT STD_LOGIC;
17
18    MemRead      : OUT STD_LOGIC; -- MEM
19    Memwrite     : OUT STD_LOGIC; -- MEM
20    Branch       : OUT STD_LOGIC; -- IF
21    --Added branch on not equal and Jump
22    Branch_Not_Equal : OUT STD_LOGIC; -- IF
23    Jump         : OUT STD_LOGIC; -- IF
24    ALUop        : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 ); -- EX
25    clock, reset : IN  STD_LOGIC );
26
27 END control;
28
29
30
31 ARCHITECTURE behavior OF control IS
32
33     SIGNAL R_format, Lw, Sw, Beq, Bne, J, RegDst_INDEX, ALUSrc_INDEX : STD_LOGIC;
34     SIGNAL MemtoReg_INDEX, MemtoReg_EXMEM, MemtoReg_MEMWB, Regwrite_INDEX, Regwrite_EXMEM, Regwrite_MEMWB : STD_LOGIC;
35     SIGNAL MemRead_INDEX, MemRead_EXMEM, Memwrite_INDEX, Memwrite_EXMEM : STD_LOGIC;
36     SIGNAL Branch_INDEX, Branch_EXMEM, Branch_Not_Equal_INDEX, Branch_Not_Equal_EXMEM, Jump_INDEX, Jump_EXMEM : STD_LOGIC;
37     SIGNAL ALUop_INDEX : STD_LOGIC_VECTOR( 1 DOWNTO 0 );
38
39 BEGIN
40     Opcode_out <= Opcode;
41     -- Code to generate control signals using opcode bits
42     R_format <= '1' WHEN opcode = "000000" ELSE '0';
43     Lw <= '1' WHEN opcode = "100011" ELSE '0';
44     Sw <= '1' WHEN opcode = "101011" ELSE '0';
45     Beq <= '1' WHEN opcode = "000100" ELSE '0';
46     -- Adding Branch on not equal (Bne) and Jump (J) based on opcode
47     Bne <= '1' WHEN opcode = "000101" ELSE '0';
48     J <= '1' WHEN opcode = "000010" ELSE '0';
49     RegDst_INDEX <= R_format;
50     ALUSrc_INDEX <= Lw OR Sw;

```

```

48     J      <= '1' WHEN Opcode = "000010" ELSE '0';
49     RegDst_INDEX <= R_format;
50     ALUSrc_INDEX <= Lw OR Sw;
51     MemtoReg_INDEX <= Lw;
52     Regwrite_INDEX <= R_format OR Lw;
53     MemRead_INDEX <= Lw;
54     Memwrite_INDEX <= Sw;
55     Branch_INDEX <= Beq;
56     --Control unit out for Branch on not equal and Jump
57     Branch_Not_Equal_INDEX <= Bne;
58     Jump_INDEX <= J;
59     ALUOp_INDEX( 1 ) <= R_format;
60     ALUOp_INDEX( 0 ) <= Beq OR Bne;
61
62     --For forwarding
63     Regwrite_1 <= Regwrite_MEMWB;
64
65     PROCESS
66     BEGIN
67         WAIT UNTIL clock'EVENT AND clock = '1';
68         --Pipelining
69         RegDst <= RegDst_INDEX;
70         ALUSrc <= ALUSrc_INDEX;
71         MemtoReg_EXMEM <= MemtoReg_INDEX;
72         MemtoReg_MEMWB <= MemtoReg_EXMEM;
73         MemtoReg <= MemtoReg_MEMWB;
74         Regwrite_EXMEM <= Regwrite_INDEX;
75         Regwrite_MEMWB <= Regwrite_EXMEM;
76         Regwrite <= Regwrite_MEMWB;
77         MemRead_EXMEM <= MemRead_INDEX;
78         MemRead <= MemRead_EXMEM;
79         Memwrite_EXMEM <= Memwrite_INDEX;
80         Memwrite <= Memwrite_EXMEM;
81         Branch_EXMEM <= Branch_INDEX;
82         Branch <= Branch_EXMEM;
83         Branch_Not_Equal_EXMEM <= Branch_Not_Equal_INDEX;
84         Branch_Not_Equal <= Branch_Not_Equal_EXMEM;
85         Jump_EXMEM <= Jump_INDEX;
86         Jump <= Jump_EXMEM;
87         ALUOp <= ALUOp_INDEX;
88     END PROCESS;
89 END behavior;

```

Code for Control stage

```

1  -- Execute module (implements the data ALU and Branch Address Adder
2  -- for the MIPS computer)
3  LIBRARY IEEE;
4  USE IEEE.STD_LOGIC_1164.ALL;
5  USE IEEE.STD_LOGIC_ARITH.ALL;
6  USE IEEE.STD_LOGIC_SIGNED.ALL;
7
8  ENTITY Execute IS
9  PORT(
10     Zero : OUT STD_LOGIC; -- IF
11     ALU_Result_MEM : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- MEM
12     ALU_result_WB : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- WB
13     Add_Result : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 ); -- IF
14     -- Adding Jump_Result to allow for jump command
15     Jump_result : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 ); -- IF
16     Read_data_1 : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
17     Read_data_2 : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
18     Sign_extend : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
19     ALUOp : IN STD_LOGIC_VECTOR( 1 DOWNTO 0 );
20     --Added Jump_Offset input to calculate jump value
21     Jump_Offset : IN STD_LOGIC_VECTOR( 9 DOWNTO 0 );
22     ALUSrc : IN STD_LOGIC;
23     PC_plus_4 : IN STD_LOGIC_VECTOR( 9 DOWNTO 0 );
24     -- Used for forwarding
25     ALU_Result : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
26     Regwrite : IN STD_LOGIC;
27     Regwrite_1 : IN STD_LOGIC;
28     IF_ID_RegisterRs : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
29     IF_ID_RegisterRt : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
30     write_data_forwarding : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
31     write_register_address_forwarding_1 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
32     write_register_address_forwarding_2 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
33     --For testing purposes
34     ALU_ctl_out : OUT STD_LOGIC_VECTOR( 2 DOWNTO 0 );
35     ALUOp_out : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
36     ForwardA_ctl_out : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
37     ForwardB_ctl_out : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
38     Ainput_out : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
39     Binput_out : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
40
41     clock, reset : IN STD_LOGIC );
42 END Execute;
43
44 ARCHITECTURE behavior OF Execute IS
45     SIGNAL Ainput, Binput : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
46     SIGNAL ALU_output_mux : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
47     SIGNAL Branch_Add : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
48     --Added Jump_Add for processing of jump address
49     SIGNAL Jump_Add : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
50     SIGNAL ALU_ctl : STD_LOGIC_VECTOR( 2 DOWNTO 0 );
51     SIGNAL Zero_EXMEM : STD_LOGIC;
52     SIGNAL ALU_Result_EXMEM : STD_LOGIC_VECTOR( 31 DOWNTO 0 );

```

```

50 SIGNAL ALU_ctl      : STD_LOGIC_VECTOR( 2 DOWNTO 0 );
51 SIGNAL Zero_EXMEM   : STD_LOGIC;
52 SIGNAL ALU_Result_EXMEM : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
53 SIGNAL ALU_Result_MEMWB : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
54 SIGNAL Add_Result_EXMEM : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
55 SIGNAL Jump_Result_EXMEM : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
56 SIGNAL Function_opcode : STD_LOGIC_VECTOR( 5 DOWNTO 0 );
57 --Used in forwarding
58 SIGNAL ForwardA_out    : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
59 SIGNAL ForwardB_out    : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
60 SIGNAL ForwardC_out    : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
61 SIGNAL ForwardA_ctl    : STD_LOGIC_VECTOR( 1 DOWNTO 0 );
62 SIGNAL ForwardB_ctl    : STD_LOGIC_VECTOR( 1 DOWNTO 0 );
63
64
65
66 BEGIN
67     Ainput <= ForwardA_out;
68
69     Binput <= ForwardB_out
70     WHEN ( ALUSrc = '0' )
71     ELSE sign_extend( 31 DOWNTO 0 );
72
73
74     -- Define function opcode source--
75     Function_opcode <= sign_extend(5 DOWNTO 0);
76
77     -- Generate ALU control bits
78     ALU_ctl( 0 ) <= ( Function_opcode( 0 ) OR Function_opcode( 3 ) ) AND ALUop(1 ); --add or subu? and r-type
79     ALU_ctl( 1 ) <= ( NOT Function_opcode( 2 ) ) OR (NOT ALUop( 1 )); -- not sub or not r-type
80     ALU_ctl( 2 ) <= ( Function_opcode( 1 ) AND ALUop( 1 )) OR ALUop( 0 ); -- r-type or branch and addu
81
82     -- Generate Zero Flag
83     Zero_EXMEM <= '1'
84     WHEN ( ALU_output_mux( 31 DOWNTO 0 ) = X"00000000" )
85     ELSE '0';
86
87     -- Select ALU output
88     ALU_result_EXMEM <= X"00000000" & B"000" & ALU_output_mux( 31 )
89     WHEN ALU_ctl = "111"
90     ELSE ALU_output_mux( 31 DOWNTO 0 );
91
92     -- Adder to compute Branch Address
93     Branch_Add <= PC_plus_4( 9 DOWNTO 2 ) + sign_extend( 7 DOWNTO 0 );
94     Add_Result_EXMEM <= Branch_Add( 7 DOWNTO 0 );
95
96     -- NEW CODE Adder to compute Jump Address
97     Jump_Add <= PC_plus_4( 9 DOWNTO 2 ) + jump_offset( 9 DOWNTO 2 );
98     Jump_Result_EXMEM <= Jump_Add( 7 DOWNTO 0 );
99
100     -- FOR TESTING
101     ALUop_out <= ALUop;
102     -- FOR TESTING
103     ALUop_out <= ALUop;
104     ALU_ctl_out <= ALU_ctl;
105     ForwardA_ctl_out <= ForwardA_ctl;
106     ForwardB_ctl_out <= ForwardB_ctl;
107     Ainput_out <= Ainput;
108     Binput_out <= Binput;
109
110     -- Register File forwarding hazards
111     ALU_Result <= ALU_result_EXMEM;
112
113     --FORWARDING UNIT, control signals for ALU input muxes
114     -- If forwardA_ctl = 00 the first ALU operand comes from the register file
115     -- If forwardA_ctl = 10 the first ALU operand comes from the previous ALU result
116     -- If forwardA_ctl = 01 the first ALU operand comes from data memory or an earlier ALU result
117     -- Same rules apply respectively to ForwardB_ctl
118     PROCESS( Regwrite_1, write_register_address_forwarding_1, IF_ID_RegisterRs, write_register_address_forwarding_2) IS
119     BEGIN
120         IF (Regwrite_1 = '1' AND write_register_address_forwarding_1 /= "0000" AND write_register_address_forwarding_2 = IF_ID_RegisterRs) THEN
121             ForwardA_ctl <= "10";
122         ELSIF (Regwrite_1 = '1' AND write_register_address_forwarding_2 /= "0000"
123             AND write_register_address_forwarding_1 /= IF_ID_RegisterRs AND write_register_address_forwarding_2 = IF_ID_RegisterRs) THEN
124             ForwardA_ctl <= "01";
125         ELSIF (Regwrite_1 = '1' AND Regwrite_1 = '0') THEN
126             ForwardA_ctl <= "11"; --dead condition should never happen, just filler for cases
127         ELSE
128             ForwardA_ctl <= "00"; --normal operation, no forwarding
129         END IF;
130     END PROCESS;
131
132     PROCESS( Regwrite_1, write_register_address_forwarding_1, IF_ID_RegisterRt, write_register_address_forwarding_2) IS
133     BEGIN
134         IF (Regwrite_1 = '1' AND write_register_address_forwarding_1 /= "0000" AND write_register_address_forwarding_2 = IF_ID_RegisterRt) THEN
135             ForwardB_ctl <= "10";
136         ELSIF (Regwrite_1 = '1' AND write_register_address_forwarding_2 /= "0000"
137             AND write_register_address_forwarding_1 /= IF_ID_RegisterRt AND write_register_address_forwarding_2 = IF_ID_RegisterRt) THEN
138             ForwardB_ctl <= "01";
139         ELSIF (Regwrite_1 = '1' AND Regwrite_1 = '0') THEN
140             ForwardB_ctl <= "11"; --dead condition should never happen, just filler for cases
141         ELSE
142             ForwardB_ctl <= "00"; --normal operation, no forwarding
143         END IF;
144     END PROCESS;
145
146     --ForwardA MUX
147     PROCESS ( ForwardA_ctl)
148     BEGIN
149         CASE ForwardA_ctl IS
150             WHEN "00" => ForwardA_out <= read_data_1; --source is registers
151             WHEN "01" => ForwardA_out <= write_data_forwarding; -- source is MEMWB
152             WHEN "10" => ForwardA_out <= ALU_result_MEMWB; --source is ALU result
153             WHEN "11" => ForwardA_out <= ALU_result_EXMEM; --source is ALU result
154         END CASE;
155     END PROCESS;

```



```

151         WHEN "10" => ForwardA_out <= ALU_result_MEMWB; --source is ALU result
152         WHEN "11" => ForwardA_out <= read_data_1; -- should never happen
153     END CASE;
154 END PROCESS;
155 --ForwardB MUX
156 PROCESS ( ForwardB_ctl)
157 BEGIN
158     CASE ForwardB_ctl IS
159         WHEN "00" => ForwardB_out <= read_data_2; --source is registers
160         WHEN "01" => ForwardB_out <= write_data_forwarding; -- source is MEMWB
161         WHEN "10" => ForwardB_out <= ALU_result_MEMWB; --source is ALU result
162         WHEN "11" => ForwardB_out <= read_data_2; --should never happen
163     END CASE;
164 END PROCESS;
165
166
167 PROCESS ( ALU_ctl, Ainput, Binput )
168 BEGIN
169     -- Select ALU operation
170     CASE ALU_ctl IS
171         -- ALU performs ALUresult = A_input AND B_input
172         WHEN "000" => ALU_output_mux <= Ainput AND Binput;
173         -- ALU performs ALUresult = A_input OR B_input
174         WHEN "001" => ALU_output_mux <= Ainput OR Binput;
175         -- ALU performs ALUresult = A_input + B_input
176         WHEN "010" => ALU_output_mux <= Ainput + Binput;
177         -- ALU performs ?
178         WHEN "011" => ALU_output_mux <= x"00000000";
179         -- ALU performs ?
180         WHEN "100" => ALU_output_mux <= x"00000000";
181         -- ALU performs ?
182         WHEN "101" => ALU_output_mux <= x"00000000";
183         -- ALU performs ALUresult = A_input - B_input
184         WHEN "110" => ALU_output_mux <= Ainput - Binput;
185         -- ALU performs SLT
186         WHEN "111" => ALU_output_mux <= Ainput - Binput ;
187         WHEN OTHERS => ALU_output_mux <= x"00000000" ;
188     END CASE;
189 END PROCESS;
190 PROCESS
191 BEGIN
192     WAIT UNTIL clock'EVENT AND clock = '1';
193     --Pipelining
194     ALU_Result_MEM <= ALU_Result_EXMEM;
195
196     ALU_Result_MEMWB <= ALU_Result_EXMEM;
197     ALU_Result_WB <= ALU_Result_MEMWB;
198     Add_Result <= ADD_Result_EXMEM;
199     Zero <= Zero_EXMEM;
200     Jump_result <= Jump_result_EXMEM;
201 END PROCESS;
202 END behavior;

```

Code for Execute stage


```

1  |
2  |      -- Dmemory module (implements the data
3  |      -- memory for the MIPS computer)
4  |  LIBRARY IEEE;
5  |  USE IEEE.STD_LOGIC_1164.ALL;
6  |  USE IEEE.STD_LOGIC_ARITH.ALL;
7  |  USE IEEE.STD_LOGIC_SIGNED.ALL;
8  |  LIBRARY altera_mf;
9  |  USE altera_mf.altera_mf_components.all;
10 |
11 | ENTITY dmemory IS
12 |     PORT( read_data      : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- WB
13 |           address       : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
14 |           write_data    : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
15 |           MemRead, Memwrite : IN  STD_LOGIC;
16 |           clock,reset    : IN  STD_LOGIC );
17 | END dmemory;
18 |
19 | ARCHITECTURE behavior OF dmemory IS
20 |     SIGNAL write_clock : STD_LOGIC;
21 |     SIGNAL read_data_MEMWB : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
22 | BEGIN
23 |     data_memory : altsyncram
24 |     GENERIC MAP (
25 |         operation_mode => "SINGLE_PORT",
26 |         width_a => 32,
27 |         widthad_a => 8,
28 |         lpm_type => "altsyncram",
29 |         outdata_reg_a => "UNREGISTERED",
30 |         init_file => "Lab08memory.mif",
31 |         intended_device_family => "cyclone"
32 |     )
33 |     PORT MAP (
34 |         wren_a => memwrite,
35 |         clock0 => write_clock,
36 |         address_a => address,
37 |         data_a => write_data,
38 |         q_a => read_data_MEMWB );
39 |     -- Load memory address register with write clock
40 |     write_clock <= NOT clock;
41 |
42 | PROCESS
43 | BEGIN
44 |     WAIT UNTIL ( clock'EVENT ) AND ( clock = '1' );
45 |     -- Added for pipelining
46 |     read_data <= read_data_MEMWB;
47 | END PROCESS;
48 | END behavior;

```

Code for Memory stage

```

1  |      -- Top Level Structural Model for MIPS Processor Core
2  |  LIBRARY IEEE;
3  |  USE IEEE.STD_LOGIC_1164.ALL;
4  |  USE IEEE.STD_LOGIC_ARITH.ALL;
5  |
6  | ENTITY MIPS IS
7  |
8  |     PORT( reset, clock
9  |         -- Output important signals to pins for easy display in simulator
10 |         : IN  STD_LOGIC;
11 |         PC
12 |         : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0 );
13 |         ALU_result_out, read_data_1_out, read_data_2_out, write_data_out,
14 |         Instruction_out, memory_write_data_out, Ainput_out, Binput_out
15 |         : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
16 |         --Added BNE_out and Jump_out
17 |         Branch_out, Zero_out, Memwrite_out, BNE_out, Jump_out,
18 |         Regwrite_out, selectWBA_out, selectWBB_out
19 |         : OUT STD_LOGIC ;
20 |         IF_ID_RegisterRs_out, IF_ID_RegisterRt_out
21 |         : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 );
22 |         ForwardA_ctl_out, ForwardB_ctl_out
23 |         : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 ));
24 | END MIPS;
25 |
26 | ARCHITECTURE structure OF MIPS IS
27 |
28 |     COMPONENT Ifetch
29 |     PORT( Instruction : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
30 |           PC_plus_4_out : OUT  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
31 |           Add_result   : IN   STD_LOGIC_VECTOR( 7 DOWNTO 0 );
32 |           --Added Jump_result, brnach not equal, and Jump
33 |           Jump_result  : IN   STD_LOGIC_VECTOR( 7 DOWNTO 0 );
34 |           Branch_Not_Equal : IN  STD_LOGIC;
35 |           Jump         : IN  STD_LOGIC;
36 |           Branch       : IN  STD_LOGIC;
37 |           Zero         : IN  STD_LOGIC;
38 |           PC_out       : OUT  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
39 |           clock,reset  : IN   STD_LOGIC );
40 | END COMPONENT;
41 |
42 |     COMPONENT Idcode
43 |     PORT( read_data_1 : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
44 |           read_data_2 : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
45 |           Instruction : IN   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
46 |           read_data   : IN   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
47 |           ALU_result_WB : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
48 |           Regwrite     : IN  STD_LOGIC;
49 |           Regdst       : IN  STD_LOGIC;
50 |           Sign_extend  : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
51 |           --Added for forwarding
52 |           ALU_Result   : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
53 |           IF_ID_RegisterRs : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); -- EX
54 |           IF_ID_RegisterRt : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); -- EX
55 |           write_data_forwarding : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- not pipelined
56 |           write_register_address_forwarding_1 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
57 |           write_register_address_forwarding_2 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined

```

```

50     write_data_forwarding : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 ); -- not pipelined
51     write_register_address_forwarding_1 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
52     write_register_address_forwarding_2 : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0 ); --not pipelined
53     --Added Jump_Offset
54     Jump_Offset : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0 );
55     --For TESTING PURPOSES
56     selectwBA_out : OUT STD_LOGIC;
57     selectwBB_out : OUT STD_LOGIC;
58
59     clock, reset : IN STD_LOGIC );
60 END COMPONENT;
61
62 COMPONENT control
63 PORT(
64     Opcode : IN STD_LOGIC_VECTOR( 5 DOWNTO 0 );
65     RegDst : OUT STD_LOGIC;
66     ALUSrc : OUT STD_LOGIC;
67     MemtoReg : OUT STD_LOGIC;
68     Regwrite : OUT STD_LOGIC;
69     --Added for forwarding
70     Regwrite_1 : OUT STD_LOGIC;
71
72     MemRead : OUT STD_LOGIC;
73     Memwrite : OUT STD_LOGIC;
74     Branch : OUT STD_LOGIC;
75     --Added branch on not equal and Jump
76     Branch_Not_Equal : OUT STD_LOGIC;
77     Jump : OUT STD_LOGIC;
78
79     ALUop : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
80     clock, reset : IN STD_LOGIC );
81 END COMPONENT;
82
83 COMPONENT Execute
84 PORT(
85     Read_data_1 : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
86     Read_data_2 : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
87     Sign_Extend : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
88     --Function_opcode : IN STD_LOGIC_VECTOR( 5 DOWNTO 0 ); replaced by sign extend
89     ALUop : IN STD_LOGIC_VECTOR( 1 DOWNTO 0 );
90     --Added Jump_Offset input to calculate jump value
91     Jump_Offset : IN STD_LOGIC_VECTOR( 9 DOWNTO 0 );
92     ALUSrc : IN STD_LOGIC;
93     Zero : OUT STD_LOGIC;
94     -- Adding Jump_Result to allow for Jump command
95     Jump_result : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
96     -- used for forwarding
97     ALU_Result : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
98     Regwrite, Regwrite_1 : IN STD_LOGIC;
99     IF_ID_RegisterRs : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
100     IF_ID_RegisterRt : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
101     write_data_forwarding : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
102     write_register_address_forwarding_1 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
103     write_register_address_forwarding_2 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
104     write_register_address_forwarding_1 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
105     write_register_address_forwarding_2 : IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
106     --For testing
107     ForwardA_ctl_out : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
108     ForwardB_ctl_out : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 );
109     Ainput_out : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
110     Binput_out : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
111
112     ALU_Result_MEM : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
113     ALU_result_WB : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
114     Add_Result : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
115     PC_plus_4 : IN STD_LOGIC_VECTOR( 9 DOWNTO 0 );
116     clock, reset : IN STD_LOGIC );
117 END COMPONENT;
118
119 COMPONENT dmemory
120 PORT(
121     read_data : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
122     address : IN STD_LOGIC_VECTOR( 7 DOWNTO 0 );
123     write_data : IN STD_LOGIC_VECTOR( 31 DOWNTO 0 );
124     MemRead, Memwrite : IN STD_LOGIC;
125     clock, reset : IN STD_LOGIC );
126 END COMPONENT;
127
128 -- declare signals used to connect VHDL components
129 SIGNAL PC_plus_4 : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
130 SIGNAL read_data_1 : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
131 SIGNAL read_data_2 : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
132 SIGNAL Sign_Extend : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
133 SIGNAL Add_result : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
134 SIGNAL ALU_result_WB : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
135 SIGNAL ALU_result_MEM : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
136 SIGNAL read_data : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
137 SIGNAL ALUSrc : STD_LOGIC;
138 SIGNAL Branch : STD_LOGIC;
139 --Added signals BNE, Jump, Jump_Result, Jump_Offset
140 SIGNAL Branch_Not_Equal : STD_LOGIC;
141 SIGNAL Jump : STD_LOGIC;
142 SIGNAL Jump_result : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
143 SIGNAL Jump_Offset : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
144 SIGNAL RegDst : STD_LOGIC;
145 SIGNAL Regwrite : STD_LOGIC;
146 SIGNAL Zero : STD_LOGIC;
147 SIGNAL Memwrite : STD_LOGIC;
148 SIGNAL MemtoReg : STD_LOGIC;
149 SIGNAL MemRead : STD_LOGIC;
150 SIGNAL ALUop : STD_LOGIC_VECTOR( 1 DOWNTO 0 );
151 SIGNAL Instruction : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
152
153 --Added for forwarding
154 SIGNAL Regwrite_1 : STD_LOGIC;
155 SIGNAL ALU_Result : STD_LOGIC_VECTOR( 31 DOWNTO 0 );

```

```

150 SIGNAL Regwrite_1      : STD_LOGIC;
151 SIGNAL ALU_Result      : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
152 SIGNAL IF_ID_RegisterRs : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
153 SIGNAL IF_ID_RegisterRt : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
154 SIGNAL write_data_forwarding : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
155 SIGNAL write_register_address_forwarding_1 : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
156 SIGNAL write_register_address_forwarding_2 : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
157
158 BEGIN
159     -- copy important signals to output pins for easy
160     -- display in Simulator
161     Instruction_out    <= Instruction;
162     ALU_result_out     <= ALU_result_MEM;
163     read_data_1_out    <= read_data_1;
164     read_data_2_out    <= read_data_2;
165     write_data_out     <= read_data WHEN MemtoReg = '1' ELSE ALU_result_WB;
166     Branch_out        <= Branch;
167     BNE_out           <= Branch_Not_Equal;
168     Jump_out          <= Jump;
169     Zero_out          <= Zero;
170     Regwrite_out      <= Regwrite;
171     Memwrite_out      <= Memwrite;
172     memory_write_data_out <= read_data_2 WHEN Memwrite = '1';
173     Regwrite_1        <= Regwrite_1;
174     ALU_Result        <= ALU_Result;
175     IF_ID_RegisterRs_out <= IF_ID_RegisterRs;
176     IF_ID_RegisterRt_out <= IF_ID_RegisterRt;
177     write_data_forwarding <= write_data_forwarding;
178     write_register_address_forwarding_1 <= write_register_address_forwarding_1;
179     write_register_address_forwarding_2 <= write_register_address_forwarding_2;
180     -- connect the 5 MIPS components
181
182     IFE : Ifetch
183     PORT MAP ( Instruction => Instruction,
184               PC_plus_4_out => PC_plus_4,
185               Add_result    => Add_result,
186               Jump_result   => Jump_result,
187               Branch        => Branch,
188               Branch_Not_Equal => Branch_Not_Equal,
189               Jump          => Jump,
190               Zero          => Zero,
191               PC_out        => PC,
192               clock         => clock,
193               reset         => reset );
194
195     ID : Idecode
196     PORT MAP ( read_data_1    => read_data_1,
197               read_data_2    => read_data_2,
198               Instruction     => Instruction,
199               read_data      => read_data,
200               ALU_Result     => ALU_Result,
201               IF_ID_RegisterRs => IF_ID_RegisterRs,
202               IF_ID_RegisterRt => IF_ID_RegisterRt,
203               IF_ID_RegisterRs => IF_ID_RegisterRs,
204               IF_ID_RegisterRt => IF_ID_RegisterRt,
205               IF_ID_RegisterRs => IF_ID_RegisterRs,
206               IF_ID_RegisterRt => IF_ID_RegisterRt,
207               write_data_forwarding => write_data_forwarding,
208               write_register_address_forwarding_1 => write_register_address_forwarding_1,
209               write_register_address_forwarding_2 => write_register_address_forwarding_2,
210               ALU_result_WB    => ALU_result_WB,
211               Regwrite         => Regwrite,
212               MemtoReg         => MemtoReg,
213               RegDst           => RegDst,
214               Sign_extend     => Sign_extend,
215               Jump_Offset     => Jump_Offset,
216               selectWBA_out    => selectWBA_out,
217               selectWBB_out    => selectWBB_out,
218               clock            => clock,
219               reset            => reset );
220
221     CTL: control
222     PORT MAP ( Opcode        => Instruction( 31 DOWNTO 26 ),
223               RegDst         => RegDst,
224               ALUSrc         => ALUSrc,
225               MemtoReg       => MemtoReg,
226               Regwrite       => Regwrite,
227               Regwrite_1     => Regwrite_1,
228               MemRead        => MemRead,
229               Memwrite       => Memwrite,
230               Branch         => Branch,
231               --Added signals BNE and Jump
232               Branch_Not_Equal => Branch_Not_Equal,
233               Jump           => Jump,
234               ALUop          => ALUop,
235               clock          => clock,
236               reset          => reset );
237
238     EXE: Execute
239     PORT MAP ( Read_data_1    => read_data_1,
240               Read_data_2    => read_data_2,
241               Sign_extend    => Sign_extend,
242               --Function_opcode => Instruction( 5 DOWNTO 0 ), replace with sign_extend
243               ALUop          => ALUop,
244               Jump_Offset    => Jump_Offset,
245               ALUSrc         => ALUSrc,
246               Zero           => Zero,
247               ALU_Result_MEM => ALU_Result_MEM,
248               ALU_Result_WB  => ALU_Result_WB,
249               Add_Result     => Add_Result,
250               Jump_Result    => Jump_Result,
251               PC_plus_4      => PC_plus_4,
252               -- used for forwarding
253               ALU_Result     => ALU_Result,
254               Regwrite       => Regwrite,
255               Regwrite_1     => Regwrite_1,

```

```

250         Regwrite => Regwrite,
251         Regwrite_1 => Regwrite_1,
252         IF_ID_RegisterRs => IF_ID_RegisterRs,
253         IF_ID_RegisterRt => IF_ID_RegisterRt,
254         write_data_forwarding => write_data_forwarding,
255         write_register_address_forwarding_1 => write_register_address_forwarding_1,
256         write_register_address_forwarding_2 => write_register_address_forwarding_2,
257         -- For testing
258         ForwardA_ctl_out => ForwardA_ctl_out,
259         ForwardB_ctl_out => ForwardB_ctl_out,
260         Ainput_out => Ainput_out,
261         Binput_out => Binput_out,
262
263         Clock          => clock,
264         Reset          => reset );
265
266     MEM: dmemory
267     PORT MAP ( read_data    => read_data,
268               address      => ALU_Result_MEM (7 DOWNTO 0),
269               write_data   => read_data_2,
270               MemRead      => MemRead,
271               Memwrite     => Memwrite,
272               clock        => clock,
273               reset        => reset );
274
275 END structure;

```

Code for Mips definition file