Lab 07
Jonathan Gusler
3/10/2019

**Overview**
This lab began the process of pipelining the MIPS processor used since Lab 3. The pipelining does not support branch and jump instructions, nor protects against any type of hazard that comes as a result of pipelining. The rudimentary pipelining system did successfully execute the code segment provided by the lab book.

**Background**
This lab used the same foundational hardware from Labs 3 [1], 4 [2], 5 [3], and 6 [4]. This lab requires familiarity with VHDL, MIPS assembly language and machine language, strong understanding of the processor simulated by the VHDL in the previous labs, and conceptual understanding of pipelining and how a pipelined processor behaves differently than a single-cycle processor.

**Discussion**
The processor from labs 3-6 was single-cycle, meaning only one instruction was being executed per clock cycle. Lab 07 altered this processor into a 4-stage pipelined processor, the stages in order being Fetch, Decode, Execute, Memory. In the diagram of the processor, *Figure 1*, that is simulated by the code there is a Writeback stage, but this was included in the Decode stage.

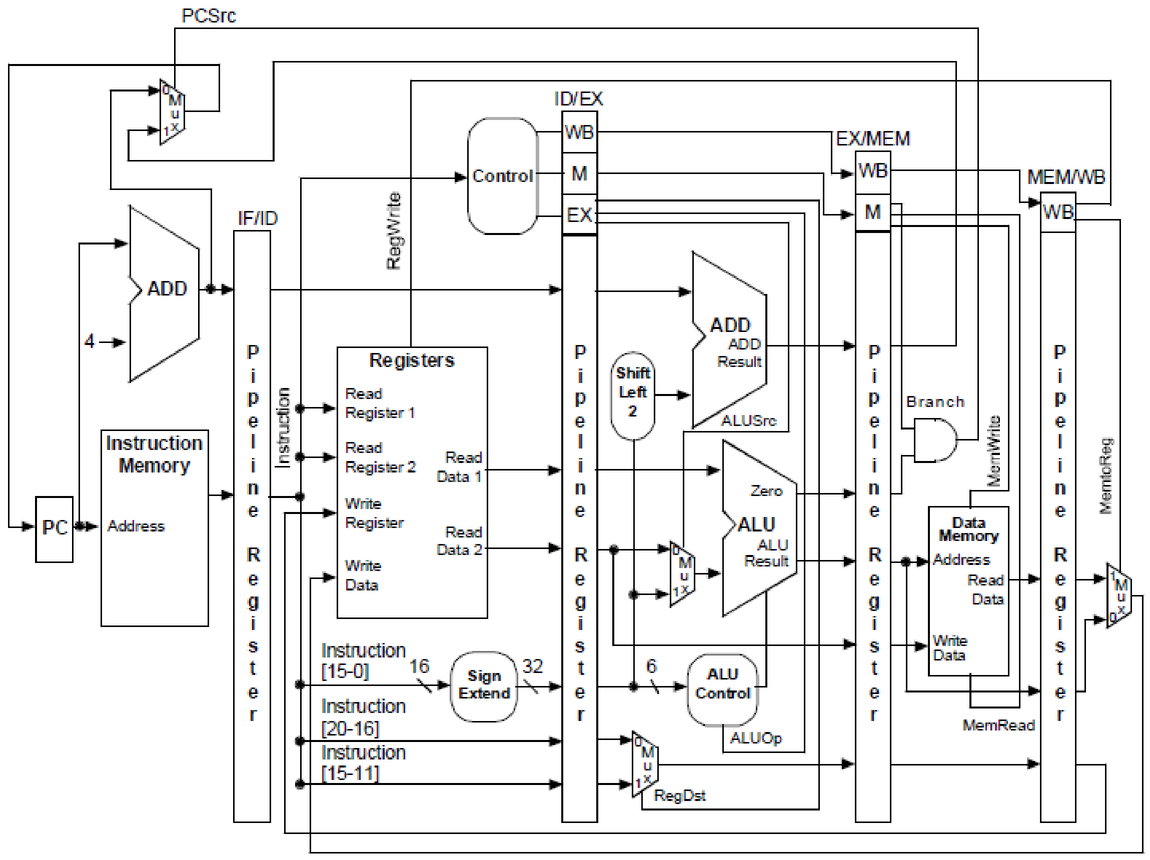
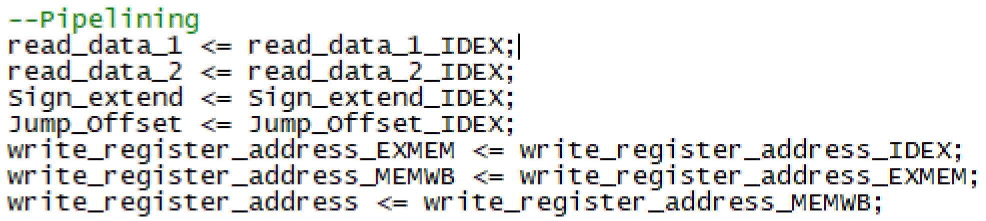


Figure 1: Diagram of original processor but doesn't show the added Jump and Branch hardware [5].

Pipelining the processor was accomplished by holding signals at their source stage until they were needed at their destination stage. For instance, if a signal was created in the decode stage and was next used in the memory stage, it was held at decode for two clock cycles and then sent out to go to memory. This “holding” was accomplished by creating, for each signal, a corresponding number of new signals based on how long it was going to be held and clocking the original signal through each of these signal stages until it was needed at another pipeline stage. To explain this using the previous example, assume a signal ‘A’ was needed at memory that came from decode. Decode would create the source signal ‘A_IDEX’, meaning signal A created at the decode stage and “going” to the execute stage. This signal, on the clock edge, would get sent to ‘A_EXMEM’, meaning signal A at the execute stage and then “going” to the memory, but in reality, always being at the decode stage. On the next clock edge, signal ‘A_EXMEM’ was sent to signal ‘A’ and sent to the memory stage. For a visual aid, *Figure 2* shows a code snippet from the Decode stage of the processor.

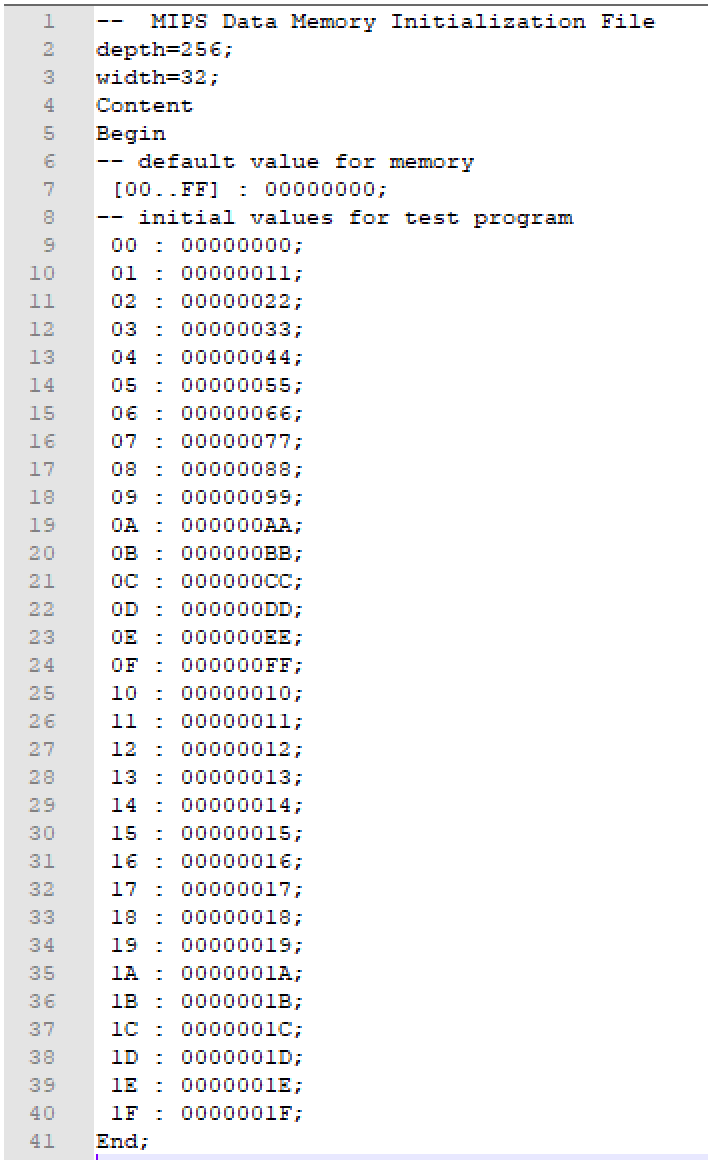


```
--Pipelining
read_data_1 <= read_data_1_IDEX;
read_data_2 <= read_data_2_IDEX;
Sign_extend <= Sign_extend_IDEX;
Jump_Offset <= Jump_Offset_IDEX;
write_register_address_EXMEM <= write_register_address_IDEX;
write_register_address_MEMWB <= write_register_address_EXMEM;
write_register_address <= write_register_address_MEMWB;
```

Figure 2: Code snippet demonstrating the holding of a signal until it is needed elsewhere.

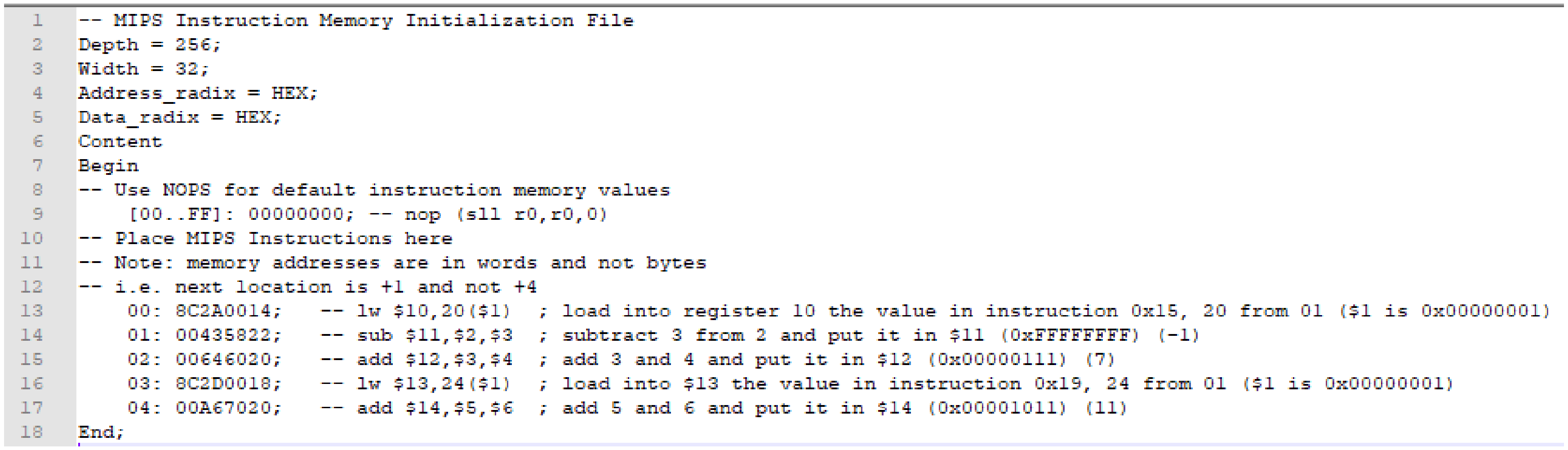
In the figure, ‘read_data_1_IDEX’ starts in the Decode stage and goes to the Execute stage. On the rising clock-edge the signal gets transferred to ‘read_data_1’, which is the signal that is the named input to the Execute stage. The ‘IDEX’ portion of the name tells where the signal is currently, Instruction Decode, and where it is going next in the pipeline, Execute. That does not mean that Execute is the final destination, it just happens to be in this case.

To test the new processor, a new memory file and program were used, shown in *Figure 3* and *Figure 4* respectively. The code for each of the stages is shown in Appendix A.



```
--  MIPS Data Memory Initialization File
depth=256;
width=32;
Content
Begin
-- default value for memory
 [00..FF] : 00000000;
-- initial values for test program
 00 : 00000000;
 01 : 00000011;
 02 : 00000022;
 03 : 00000033;
 04 : 00000044;
 05 : 00000055;
 06 : 00000066;
 07 : 00000077;
 08 : 00000088;
 09 : 00000099;
 0A : 000000AA;
 0B : 000000BB;
 0C : 000000CC;
 0D : 000000DD;
 0E : 000000EE;
 0F : 000000FF;
 10 : 00000010;
 11 : 00000011;
 12 : 00000012;
 13 : 00000013;
 14 : 00000014;
 15 : 00000015;
 16 : 00000016;
 17 : 00000017;
 18 : 00000018;
 19 : 00000019;
 1A : 0000001A;
 1B : 0000001B;
 1C : 0000001C;
 1D : 0000001D;
 1E : 0000001E;
 1F : 0000001F;
End;
```

Figure 3: New initial memory values used for Lab 07.



```
-- 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: 8C2A0014;   -- lw $10,20($1)  ; load into register 10 the value in instruction 0x15, 20 from 01 ($1 is 0x00000001)
    01: 00435822;   -- sub $11,$2,$3  ; subtract 3 from 2 and put it in $11 (0xFFFFFFFF) (-1)
    02: 00646020;   -- add $12,$3,$4  ; add 3 and 4 and put it in $12 (0x00000111) (7)
    03: 8C2D0018;   -- lw $13,24($1)  ; load into $13 the value in instruction 0x19, 24 from 01 ($1 is 0x00000001)
    04: 00A67020;   -- add $14,$5,$6  ; add 5 and 6 and put it in $14 (0x00001011) (11)
End;
```

Figure 4: Program used to test new pipelined processor.

An in-depth explanation of each of the instructions:

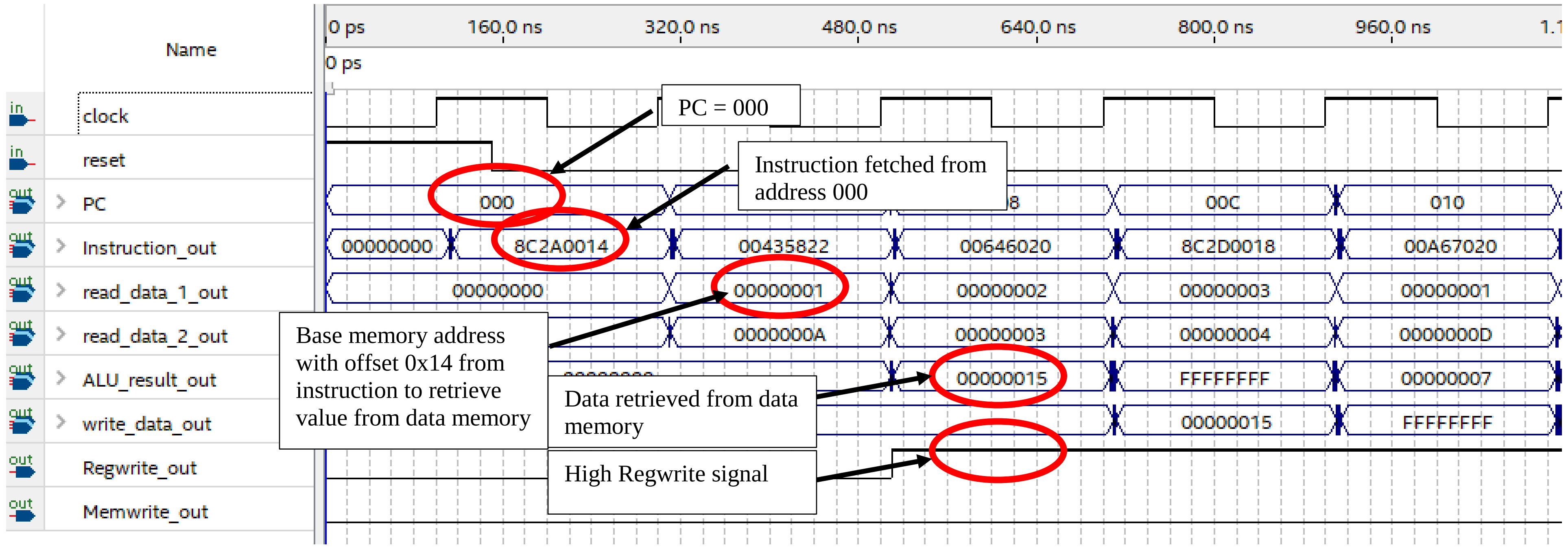
1) Load into register $10 the value in data memory register 0x15 (0x00000015), which is 20 away from location 1 in decimal
2) Subtract 3 from 2 (-1), (0xFFFFFFFF), and put the value in register $11
3) Add 3 to 4 (7), (0x00000007), and put the value in register $12
4) Load into register $13 the value in data memory register 0x19 (0x00000019), which is 24 away from location 1 in decimal
5) Add 5 to 6 (11), (0x0000000B), and put the value in register $14

**Analysis and Results**
The annotated figures that start on page 6 show a step by step walkthrough of the instructions being executed as shown in the simulation waveform. *Table 1* describes the relevant signals in the waveform.

| Signal Name | Description |
|---|---|
| clock | The clock for the system |
| reset | Resets system back to instruction on falling edge |
| PC | Program Counter |
| Instruction_out | Instruction fetched from IM |
| read_data_1_out | Rs output from registers |
| read_data_2_out | Rd output from registers |
| ALU_result_out | Output of ALU |
| write_data_out | Data to write to registers on high |
| Regwrite_out | Enable write to registers on high |
| Memwrite_out | Enable write to memory on high |

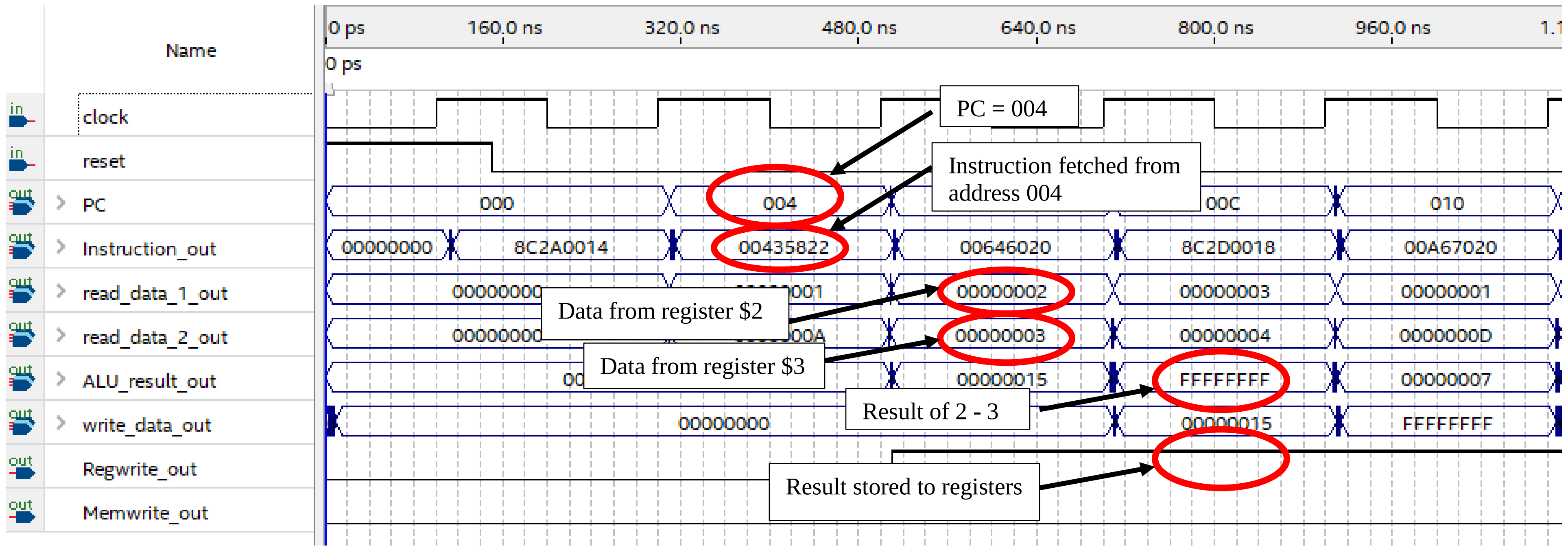
Table 1: Relevant signal names and descriptions for the following analyses.

00: 8C2A0014; -- lw $10,20($1)
100011 | 00001 | 01010 | 0000000000010100
Opcode | $s = 1 | $t = 10 | offset



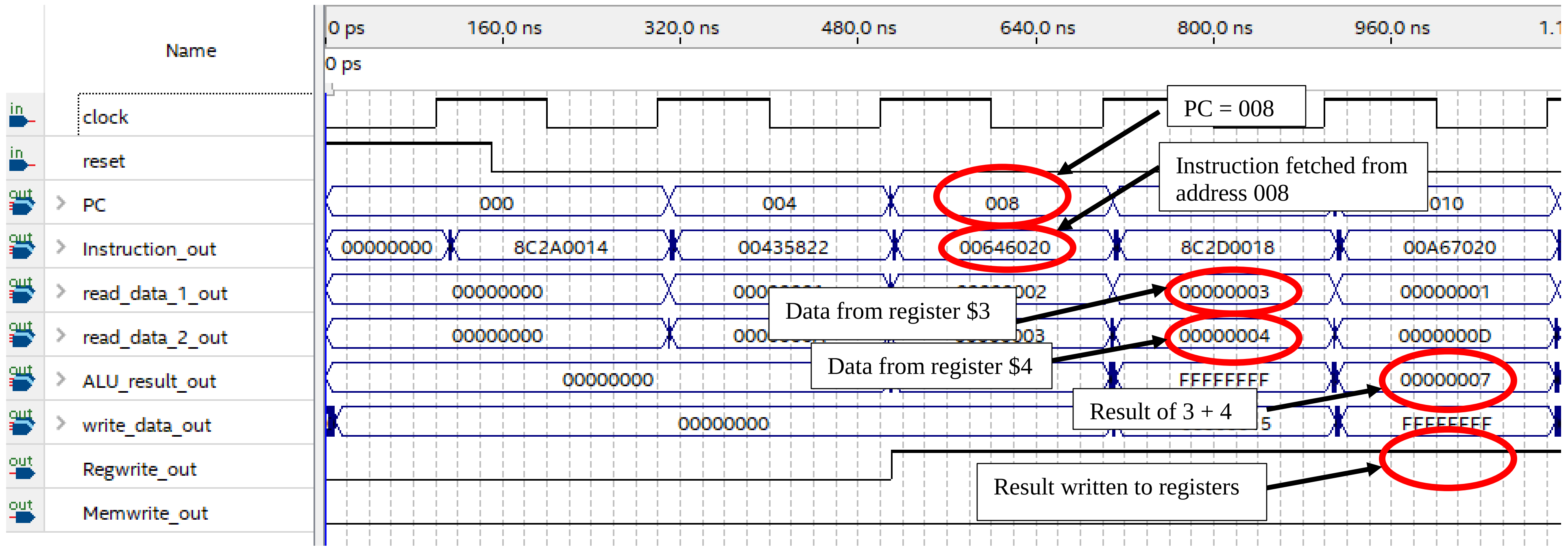
This instruction loads the value 0x00000015 into register $10 from data memory location 0x00000015.

01: 00435822; -- sub $11, $2, $3
000000 | 00010 | 00011 | 010110 |0000100010
Opcode | $s = 2 | $t = 3 | $d = 11 | function code subtract



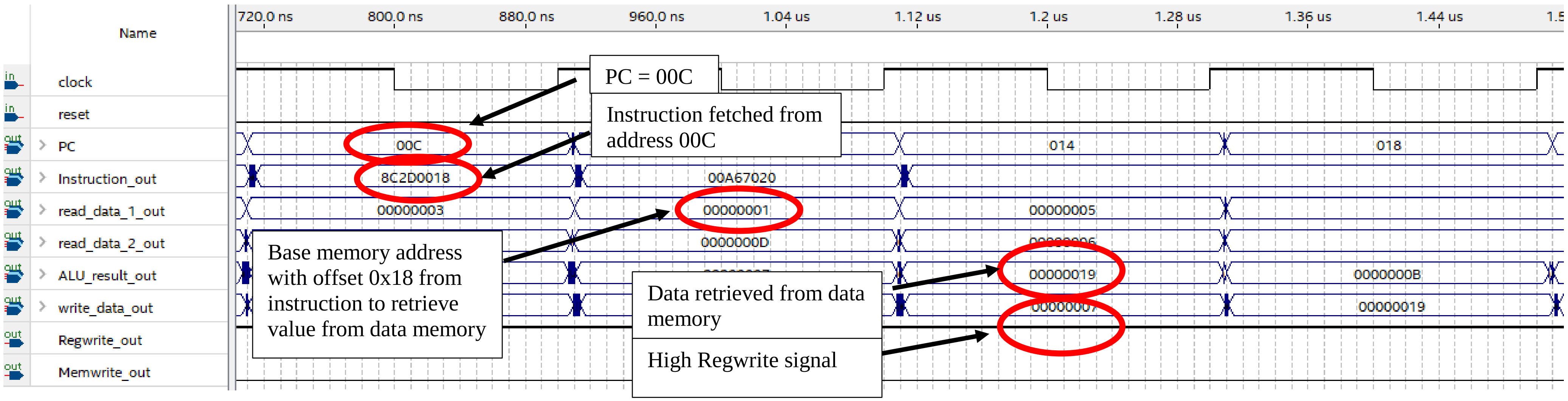
This instruction subtracts the value in register $3 from register $2 and stores the result in register $11.

02: 00646020;   -- add $12, $3, $4
000000 | 00011 | 00100  | 011000 |0000100000
Opcode | $s = 3 | $t = 4 | $d = 12  | function code addition



This instruction adds the value in register $3 and register $4 and stores the result in register $12.

03: 8C2D0018; -- lw $13, 24($1)
100011 | 00001 | 01101 | 0000000000011000
Opcode | $s = 1 | $t = 13 | offset



PC = 00C

Instruction fetched from address 00C

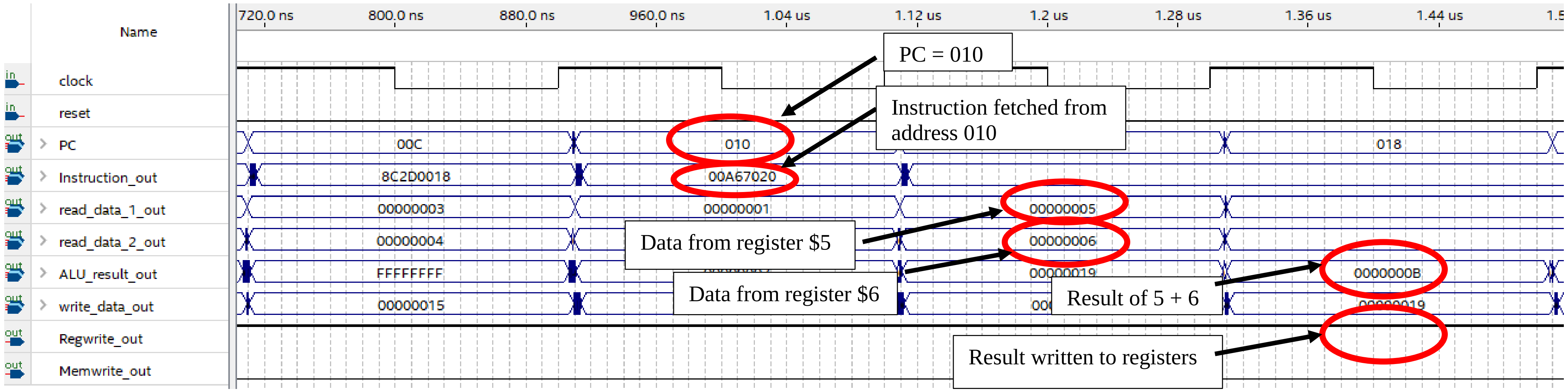
Base memory address with offset 0x18 from instruction to retrieve value from data memory

Data retrieved from data memory

High Regwrite signal

This instruction loads the value 0x00000019 into register $13 from data memory location 0x00000019.

04: 00A67020 -- add $14, $5, $6
000000 | 00101 | 00110 | 01110 | 00000100000
Opcode | $s = 5 | $t = 6 | $d = 14 | function code addition



This instruction adds the value in register $5 and register $6 and stores the result in register $14.

**Conclusion**
As shown in the analysis, the code successfully simulated a 4-stage pipeline according to the program tested, the number of clock cycles used, and the various actions taking place during each clock cycle. The next step is to add protection against hazards that arise from using a pipelined processor.

## Works Cited

[1] J. Gusler, "Lab 03," 2019.

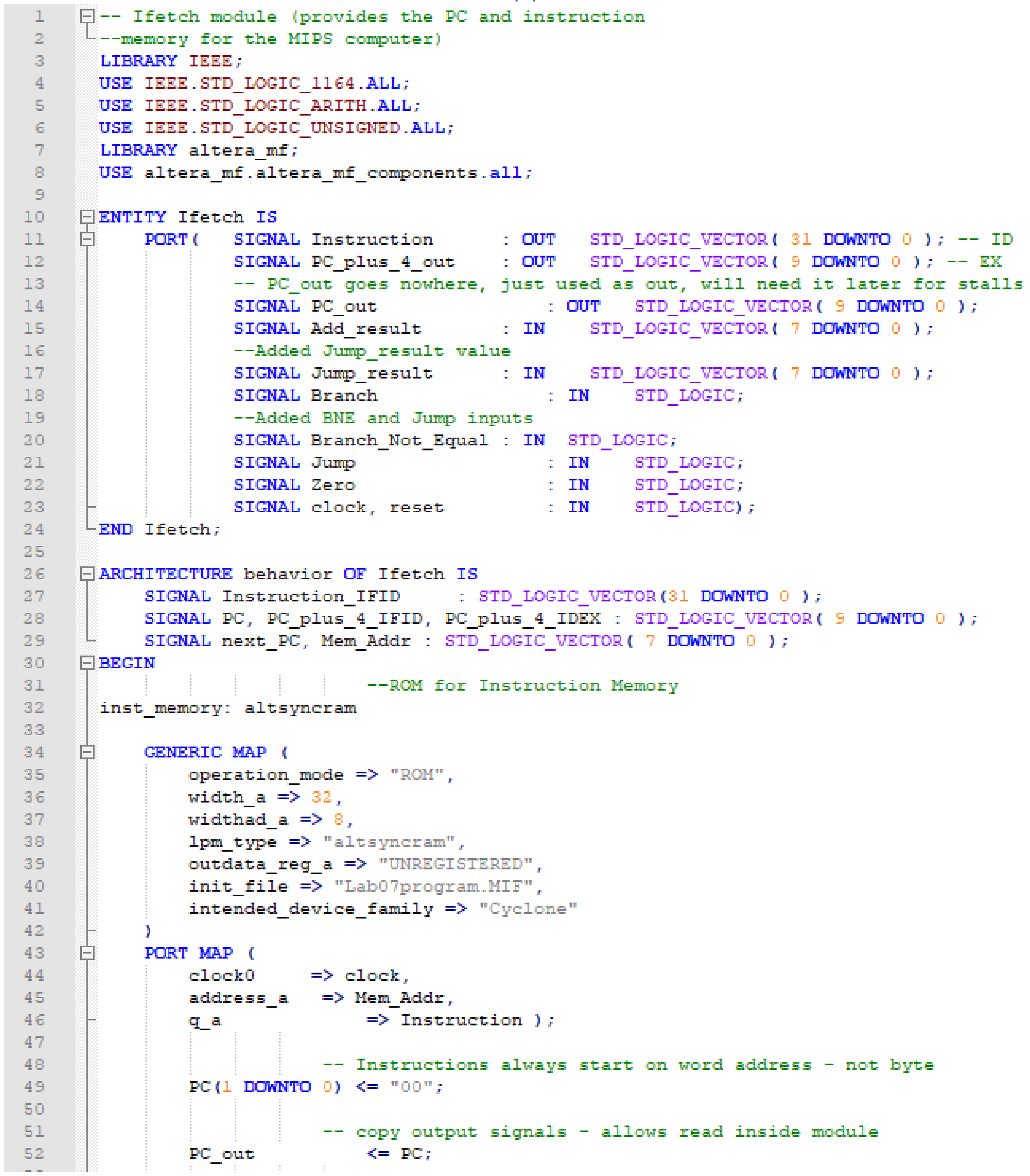
[2] J. Gusler, "Lab 04," 2019.

[3] J. Gusler, "Lab 05," 2019.

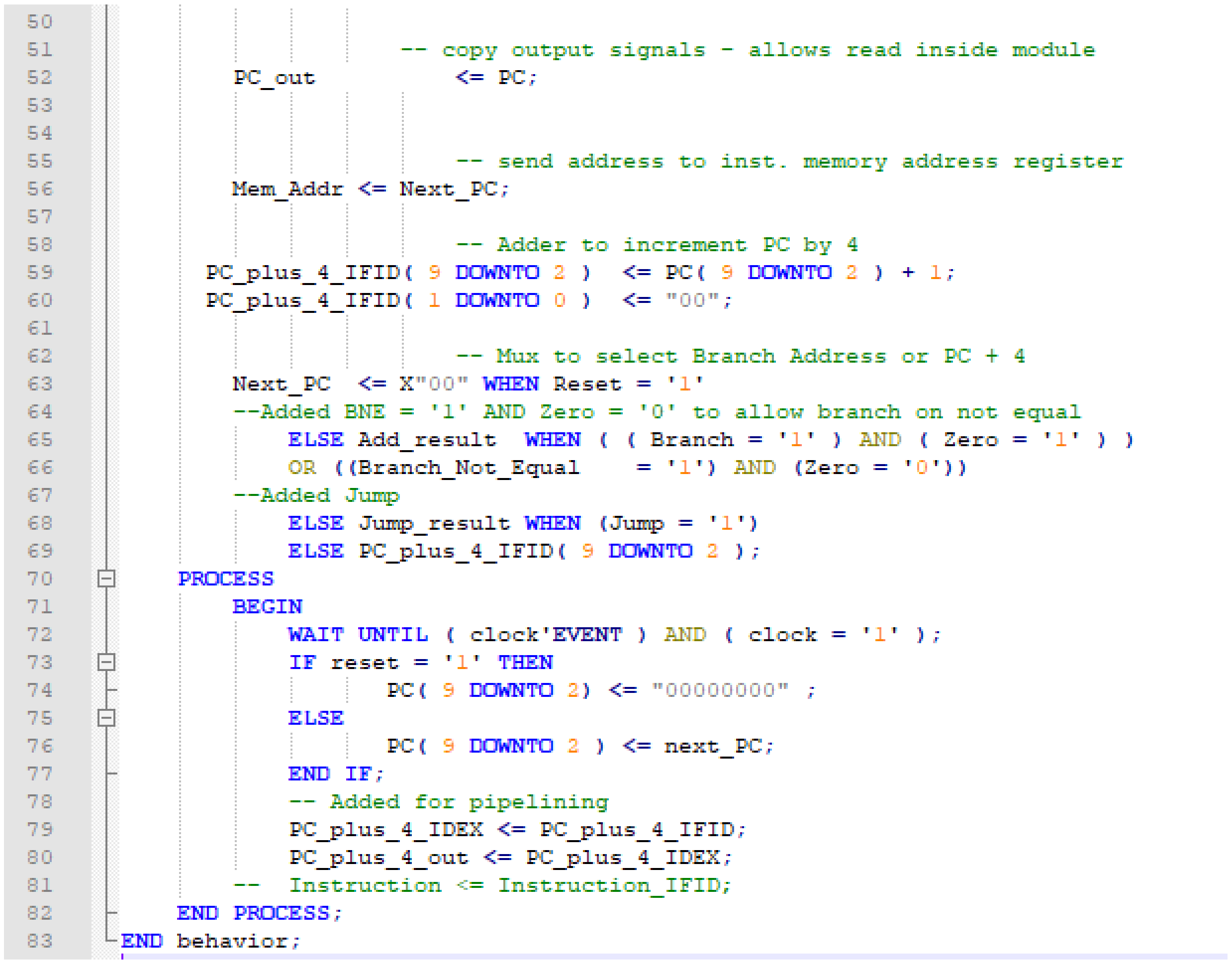
[4] J. Gusler, "Lab 06," 2019.

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

## Appendix A

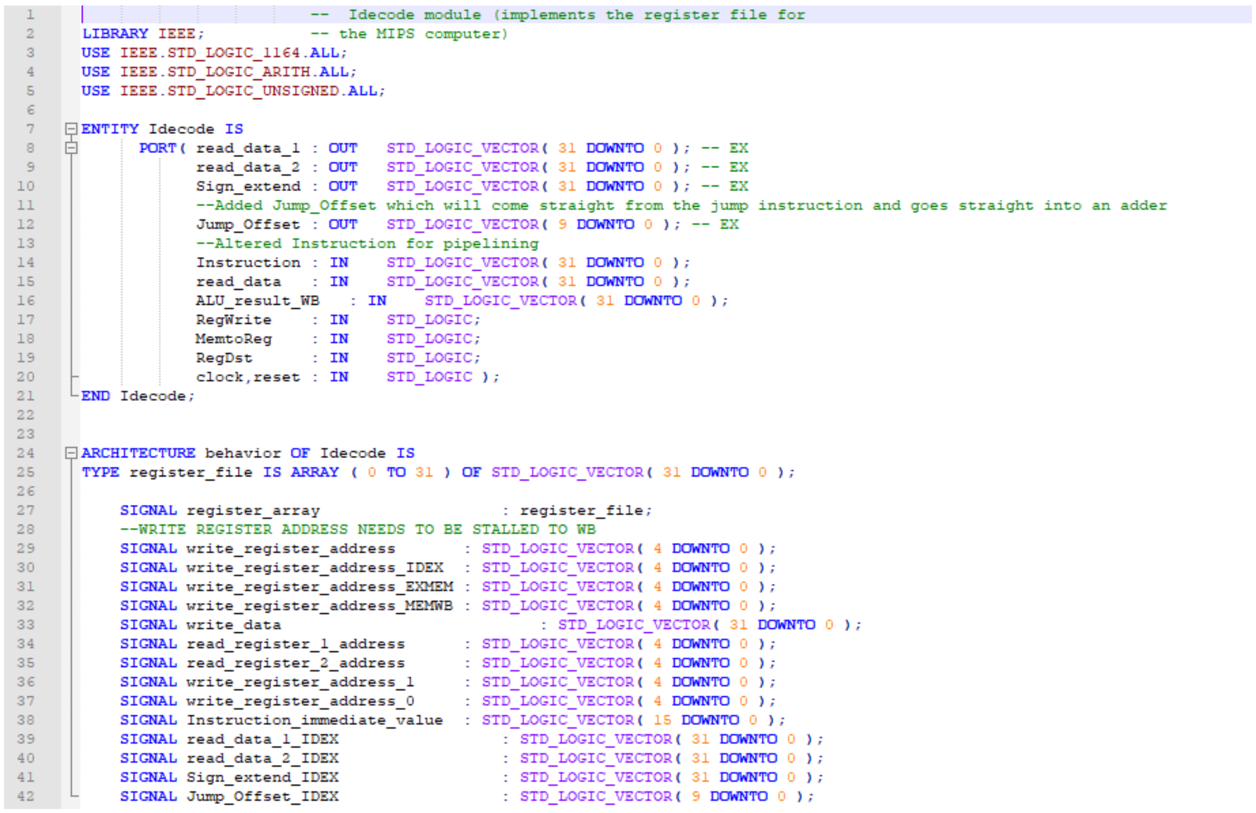


```vhdl
-- Ifetch module (provides the PC and instruction
--memory for the MIPS computer)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

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 => "Lab07program.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;
```

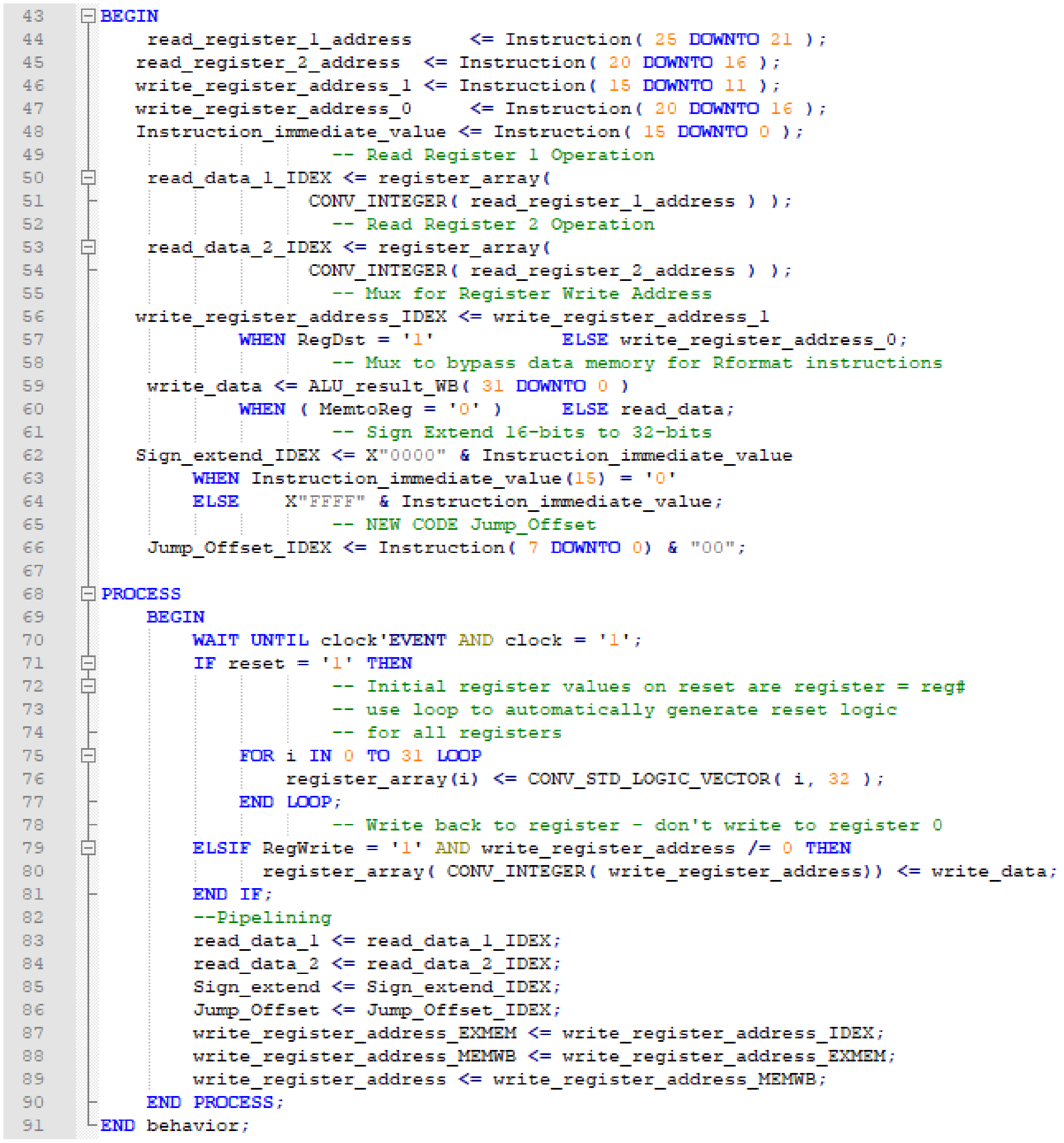


```

                        -- 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;
        --  Instruction <= Instruction_IFID;
    END PROCESS;
END behavior;
```

Code for Fetch stage of pipeline.

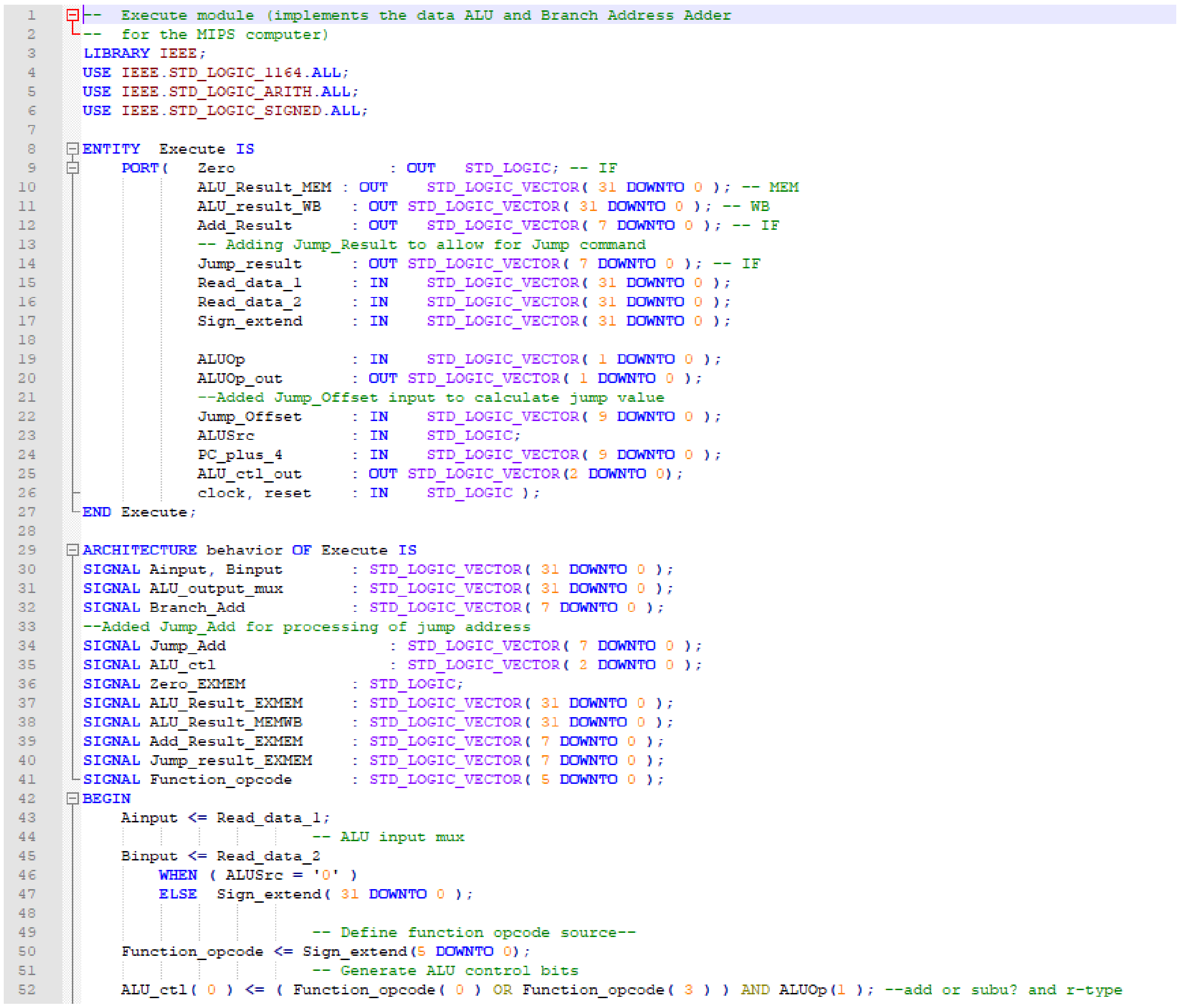


```
                        --  Idecode module (implements the register file for
LIBRARY IEEE;           -- the MIPS computer)
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

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
            --Altered Instruction for pipelining
            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;
            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;
    --WRITE REGISTER ADDRESS NEEDS TO BE STALLED TO WB
    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 Jump_Offset_IDEX                 : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
```

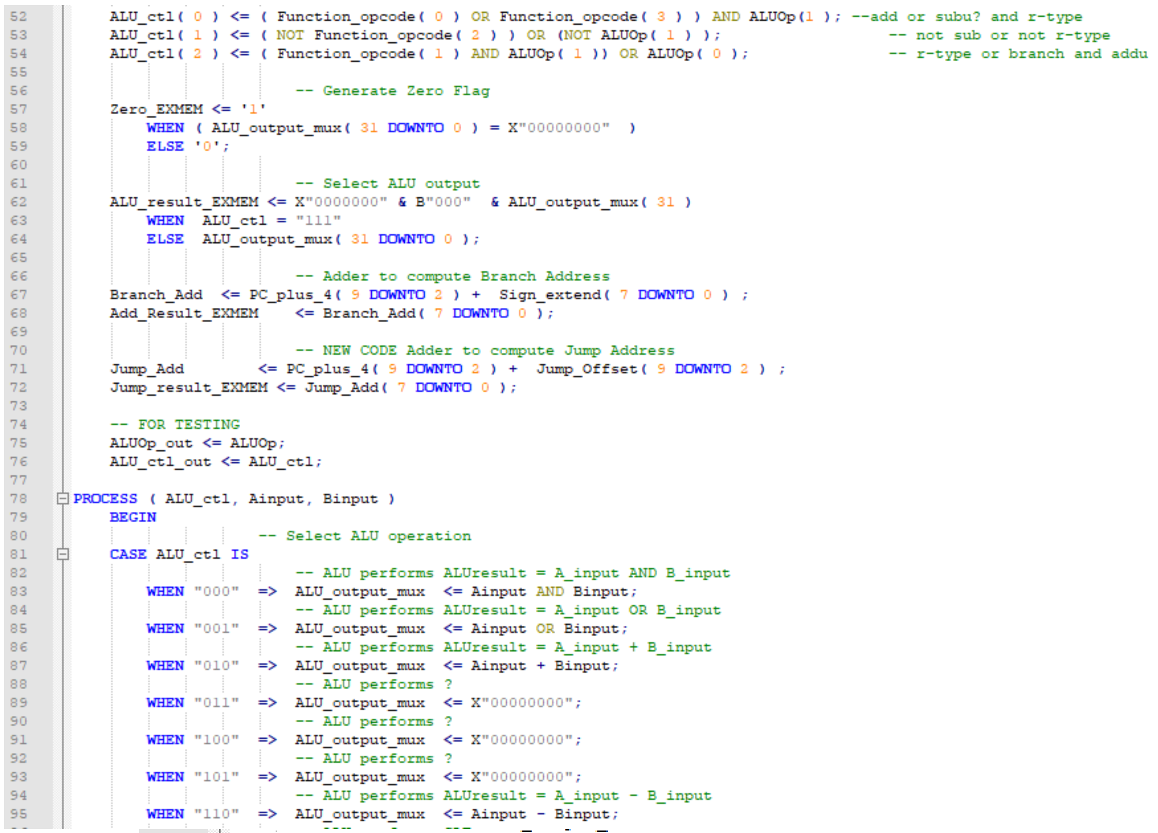


```vhdl
BEGIN
    read_register_1_address     <= Instruction( 25 DOWNTO 21 );
   read_register_2_address  <= Instruction( 20 DOWNTO 16 );
   write_register_address_1 <= Instruction( 15 DOWNTO 11 );
   write_register_address_0     <= Instruction( 20 DOWNTO 16 );
   Instruction_immediate_value <= Instruction( 15 DOWNTO 0 );
                    -- Read Register 1 Operation
    read_data_1_IDEX <= register_array(
                  CONV_INTEGER( read_register_1_address ) );
                    -- Read Register 2 Operation
    read_data_2_IDEX <= register_array(
                  CONV_INTEGER( read_register_2_address ) );
                    -- 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;
                    -- NEW CODE Jump_Offset
    Jump_Offset_IDEX <= Instruction( 7 DOWNTO 0) & "00";

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;
    END PROCESS;
END behavior;
```

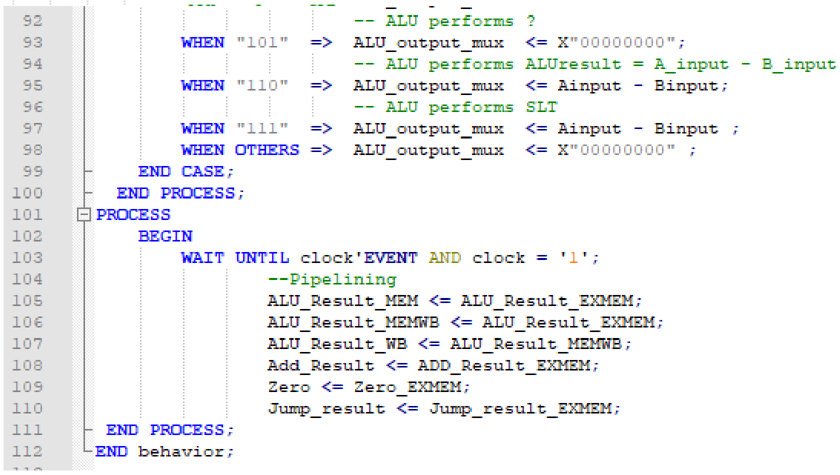
Code for Decode stage of the pipeline.



```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 );
            ALUOp_out       : OUT 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 );
            ALU_ctl_out     : OUT STD_LOGIC_VECTOR(2 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_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 );
BEGIN
    Ainput <= Read_data_1;
                    -- ALU input mux
    Binput <= Read_data_2
        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
```

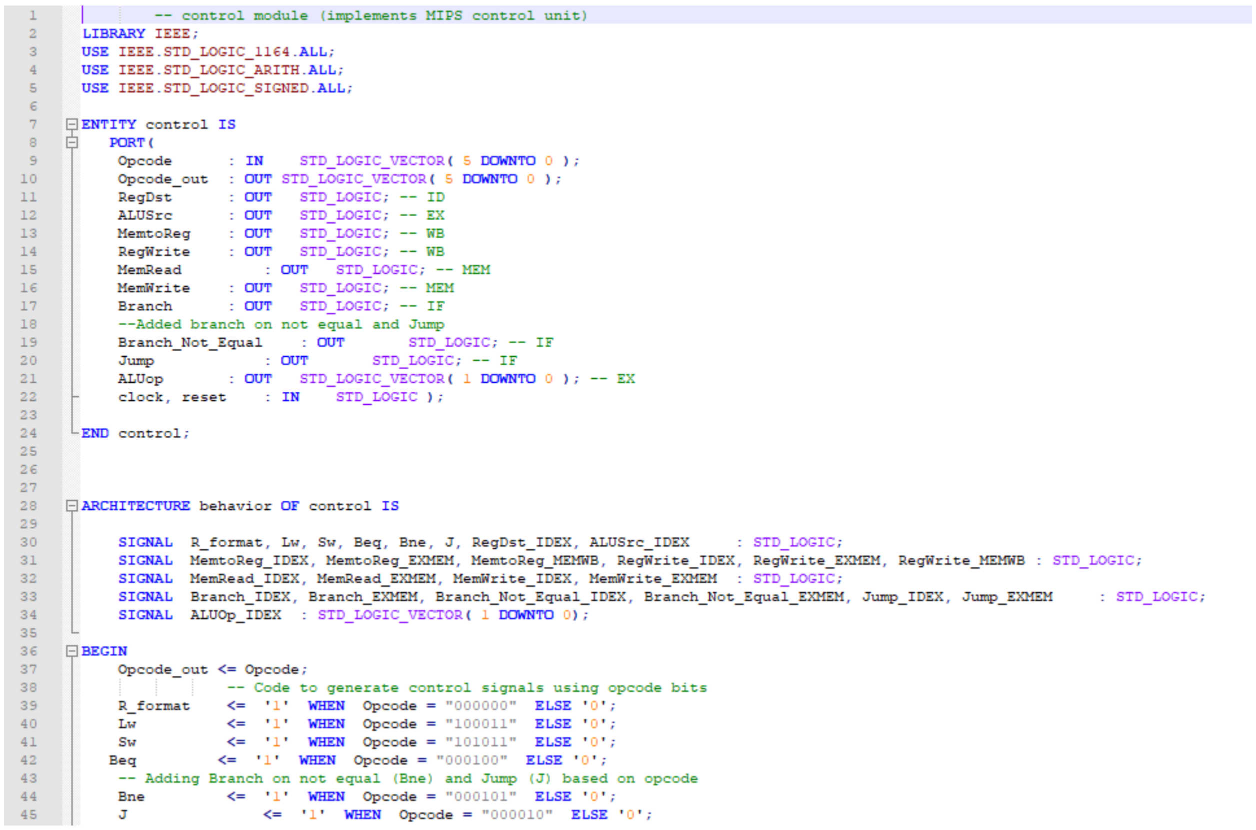


```
    ALU_ctl( 0 ) <= ( Function_opcode( 0 ) OR Function_opcode( 3 ) ) AND ALUOp(1 ); --add or subu? and r-type
    ALU_ctl( 1 ) <= ( NOT Function_opcode( 2 ) ) OR (NOT ALUOp( 1 ) );                  -- not sub or not r-type
    ALU_ctl( 2 ) <= ( Function_opcode( 1 ) AND ALUOp( 1 )) OR ALUOp( 0 );                -- 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;
    ALU_ctl_out <= ALU_ctl;

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;
```

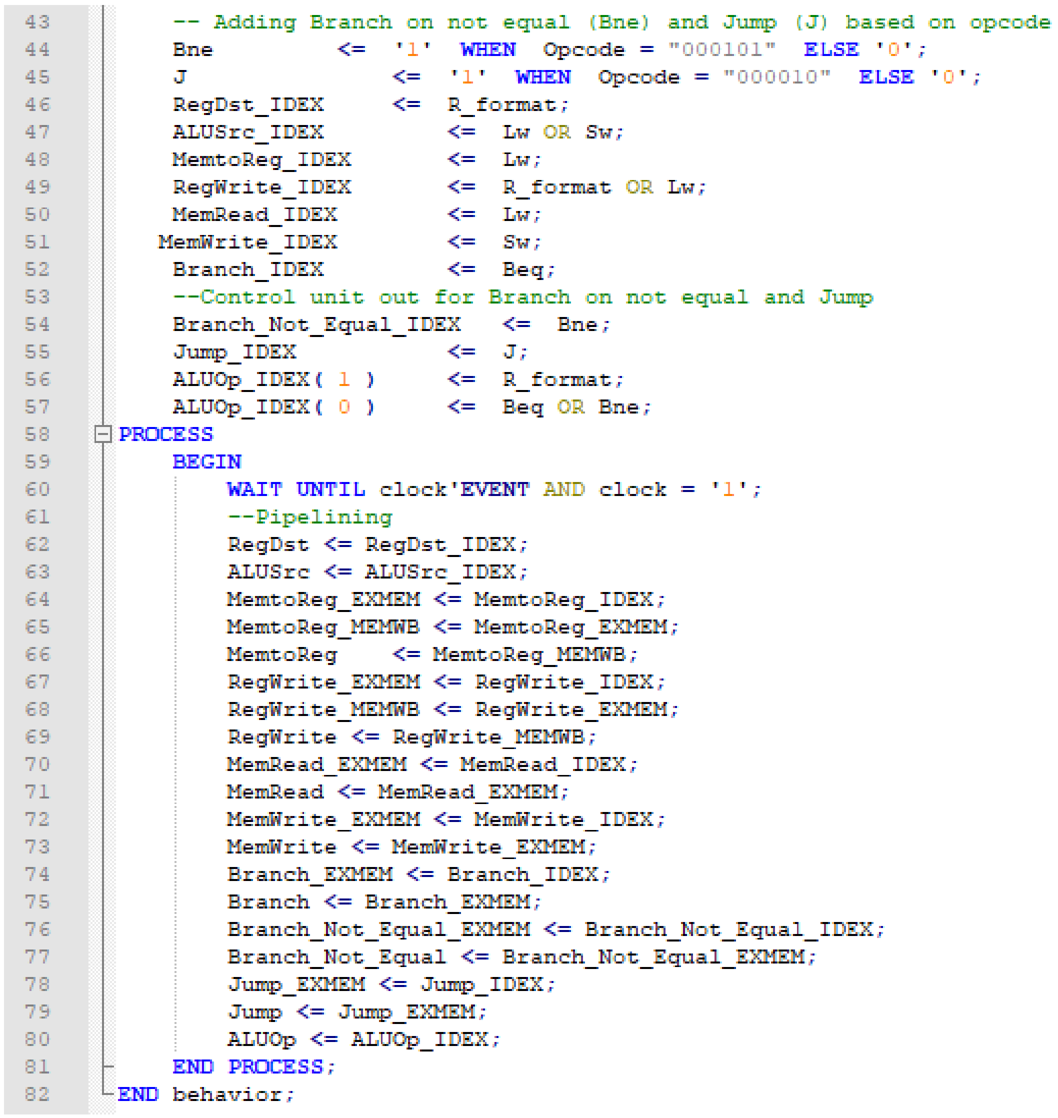


```
                        -- ALU performs ?
        WHEN "101"  =>  ALU_output_mux  <= X"00000000";
                        -- ALU performs ALUresult = A_input - B_input
        WHEN "110"  =>  ALU_output_mux  <= Ainput - Binput;
                        -- ALU performs SLT
        WHEN "111"  =>  ALU_output_mux  <= 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 of the pipeline.

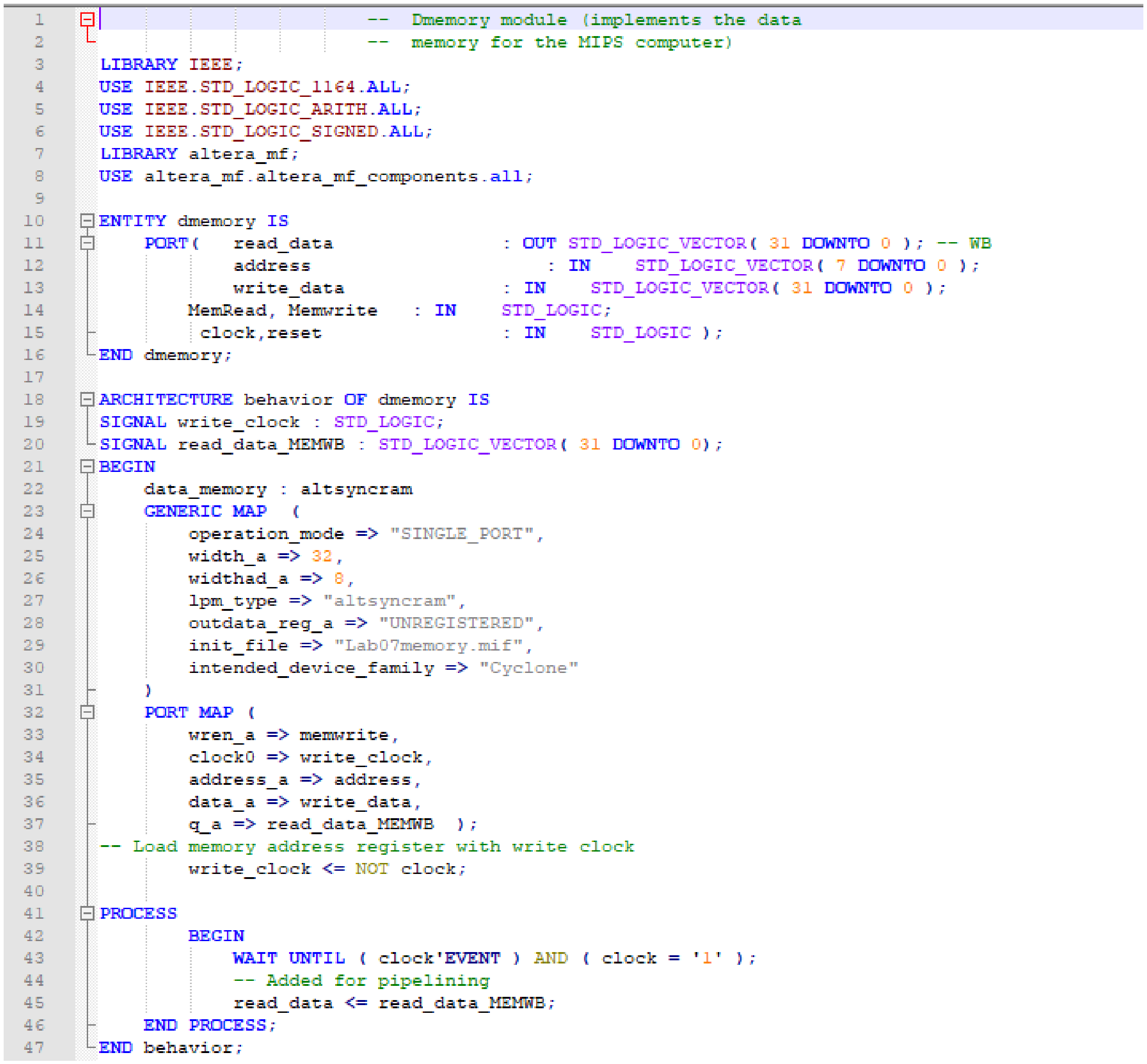


```vhdl
        -- control module (implements MIPS control unit)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

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
    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';
```



```
    -- 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;
    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;
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, part of the Execute stage of the pipeline.



```
                                        -- 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 => "Lab07memory.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 the Memory portion of the pipeline.