MIPS CPU with Multicycle Datapath Final Report for Computer Organization and Design (Fall 2017)

Hector Hernandez, Sicheng Wu

Contact Info: hhernandeztrivino@hawk.iit.edu, swu48@hawk.iit.edu

CWID: A20391413, A20365924

I. INTRODUCTION

In this project, we are supposed to design our own custom version of 32-bit RISC ISA based MIPS processor, which essentially utilize the VHDL hardware descriptive language to achieve low-level of computer design. The design principle of MIPS processor follows the rule of ***COMPUTER ORGANIZATION AND DESIGN IN FIFTH EDITION***, which supports three different types of instruction format: R-type, I-type, and J-type; each memory unit is 32-bit addressable. To better illustrate the data flow of CPU design, we implement a small part of actual MIPS ISA, as well as demonstrate CPU datapath in a nutshell.

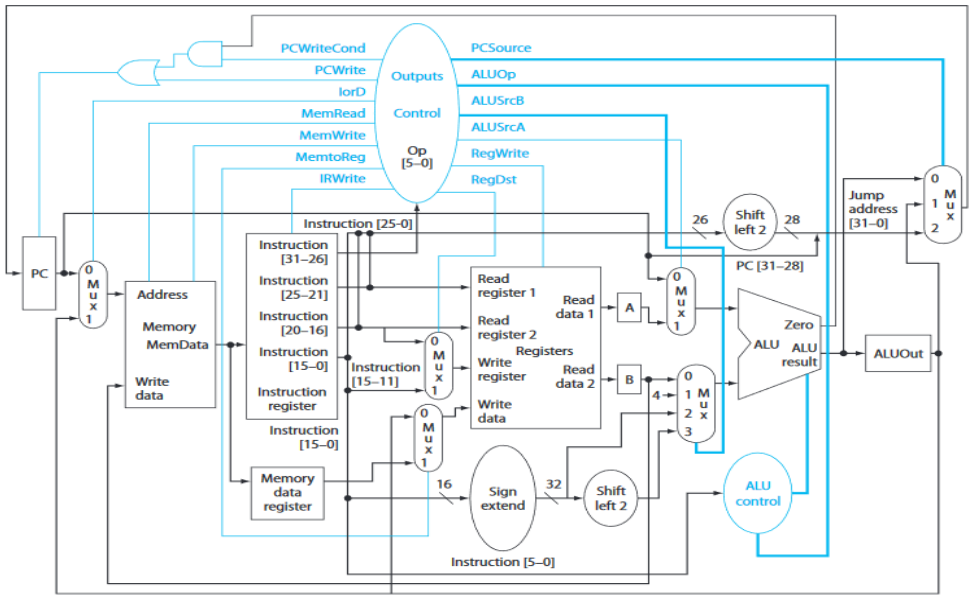

Figure 1. Multicycle Processor Datapath

II. IMPLEMENTATION AND CONTROL LOGIC

MIPS CPU can perform various of arithmetic and logic operations based on compiled program file or operating system generated machine code. It reads instruction and does a bunch of ALU operations on data; also, its working flow can be logically abstracted as a big finite state machine with various of working transitions. To ensure the whole working process can run properly to read and write instructions among different function blocks that we need introduce an external rising edge clock signal as well as build up a self-control finite state machine system.

In general, modern MIPS CPU requires 5-stage pipeline to complete an entire instruction. Even if we have not issued a pipelined version of CPU in this project, there still requires 5 stages or 5 functional building blocks to perform a complete CPU datapath. Initially, instructions are fetched from instruction memory and it will be decoded and recognized which type of instruction it belongs to. (Transition from 0 to 2) The next transition after instruction decode are dependent on opcode, which R-Type, I- Type, J-Type function has their own transitions to work with. In this project, to incorporate with immediate value related function, we added a transition state 10 in control unit which accepts the addi, andi, ori operations, which works as a control logic transition buffer between s2 and s0. The difficulty for implement this state is to transfer the new state back to the starting point s0 without conflicting others as well as arithmetic and logic operations datpath. When implement the control unit of Multicycle datapath, we found out an error provided by the reference where in state s4, MemReg = 1 instead of 0; for this part, we debugged for a pretty long time and finally correct this error.

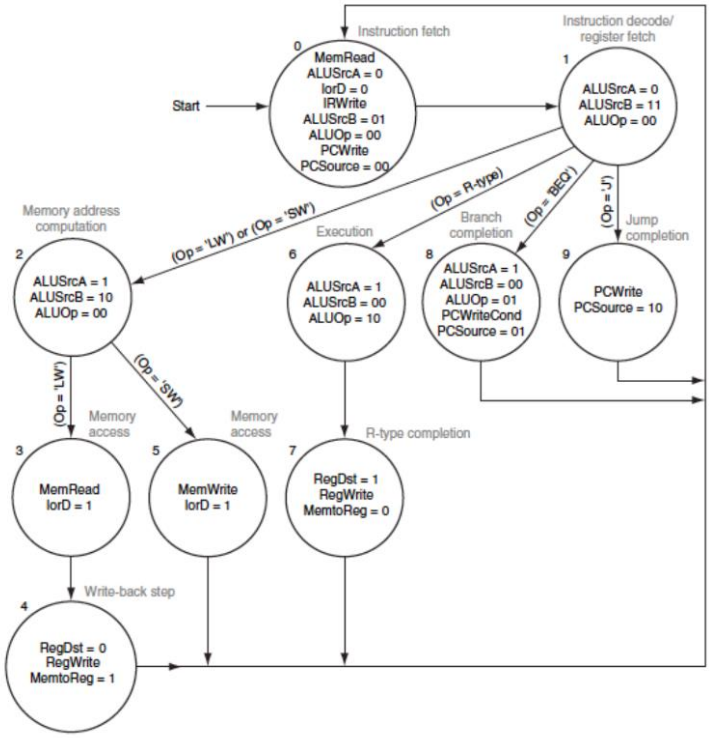

Figure 2. Contorl Logic Finite State Machine

About the implementation of Multicycle datapath, we divided the whole entire structure into 5 main parts (aka IF ID EX MEM WB) which is somewhat like we have learnt from class, also we dig deeper into some part of the element within the building block such as memory data register that we have re-described it in another VHDL file as it sometimes accepts the data from Register B; register B is also rewritten in a single file which sends out data when it receives the external clock cycle. The whole CPU functional descriptive files are enclosed in Appendix, if readers are interested how they were built just go down to take a close look.

III. Single File Testbench

After completing each part of VHDL coding, we wrote a corresponding test script for each relevant part. It provides an enormous help that we have eliminated minor error before the CPU synthesizing step.

IV. Test file and simulation

In this phase, after assembling all the element into a whole structure. We wrote a MIP.asm test script to see if our CPU works fine as we expected. Right this phase, it is trivial to show the waveform of each single file.

```
        PROCESS IS
                BEGIN
                --  PC <- PC +4
                WAIT FOR 10 ns;
                ALUSrcA         <= '0';
                ALUSrcB         <= "01";
                IR              <= "100000";
                ALUOp           <= "00";
                PCin            <= x"00000000";
                Input_4         <= x"00000004";

                --  Branch Instruction
                 WAIT FOR 10 ns;
                 ALUSrcA <= '1';
                 ALUSrcB <= "10";
                 ALUOp  <= "00";
                 Sign_extend    <= x"00000064";
                 Reg_A  <= x"00000000";


                -- R-type arithmetic
                WAIT FOR 10 ns;
                ALUSrcA         <= '1';
                ALUSrcB         <= "00";
                IR              <= "100000";
                ALUOp           <= "00";
                Reg_A           <= x"11235813";
                Reg_B           <= x"00000000";

                -- Address Calculation
                WAIT FOR 10 ns;
                ALUSrcA         <= '1';
                ALUSrcB         <= "10";
                IR              <= "100000";
                ALUOp           <= "00";
                Reg_A           <= x"00000008";
                Sign_extend     <= x"00008000";
                WAIT FOR 10 ns;
        WAIT;
END PROCESS;
END ARCHITECTURE;
```

Figure 3. ALU Testbench Code

= Multicycle Datapath CPU Test Code =

= = = = = = = = = = = = = Asm Code Demo = = = = = = =Result in Register =

```
        Line1:   addi $t6, $zero, 0x00FA      # $t6 = 0xFA
        Line2:   ori  $t1, $zero, 0x1000      # $t1 = 0x1000
        Line3:   addi $t0, $zero, 0x1000      # $t0 = 0x1000
        Line4:   add  $t2, $t0, $t1           # $t2 = 0x2000
        Line5:   sub  $t3, $t2, $t1           # $t3 = 0x1000
        Line6:   or   $t5, $t6, $zero         # $t5 = 0xFA
        Line7:   and  $t7, $t1, $t0           # $t7 = 0x1000
        Line8:   nor  $t9, $zero, $zero       # $t9 = 0xFFFF
        Line9:   lw   $t8, $t6($zero)         # $t8 = 0xFA
        Line10:  slt  $t4, $t3, $t2           # $t4 = 0x0001
Loop:   Line11:  addi $t0, $zero, 0x01        # for(int i = 0x1001; i < 0x2000; i++)
        Line12:  sw   $t0,  $s0($zero)        # count += 1;
        Line13:  bne  $t0, $t1, Loop          # $t0 = 0x2000, $t10 = 0x2000 // end of loop
```

= =
= =

Document 1. Assmbly Code for Multicylce.vhdl Test

```
Line 1:  0x00000000|0x200e00fa| addi $t6, $zero, 0x00FA
Line 2:  0x00000004|0x34091000| ori  $t1, $zero, 0x1000
Line 3:  0x00000008|0x20081000| addi $t0, $zero, 0x1000
Line 4:  0x0000000c|0x01095020| add  $t2, $t0, $t1
Line 5:  0x00000010|0x012a5822| sub  $t3, $t2, $t1
Line 6:  0x00000014|0x01c06825| or     $t5, $t6, $zero
```

```
Line 7:  0x00000018|0x01287824| and  $t7, $t1, $t0
Line 8:  0x0000001c|0x01e08025| nor  $t9, $zero, $zero
Line 9:  0x00000020|0x8c180000| lw   $t8, $t6($zero)
Line 10: 0x00000024|0x016a602a| slt  $t4, $t3, $t2
Line 11: 0x00000028|0x20080001| addi $t0, $zero, 0x01
Line 12: 0x0000002c|0xac080000| sw   $t0,  $s0($zero)
Line 13: 0x00000030|0x1509000C| bne  $t0, $t1, Loop
```

Document 2. Assmbly Code and Hexdecimal Code Conversion

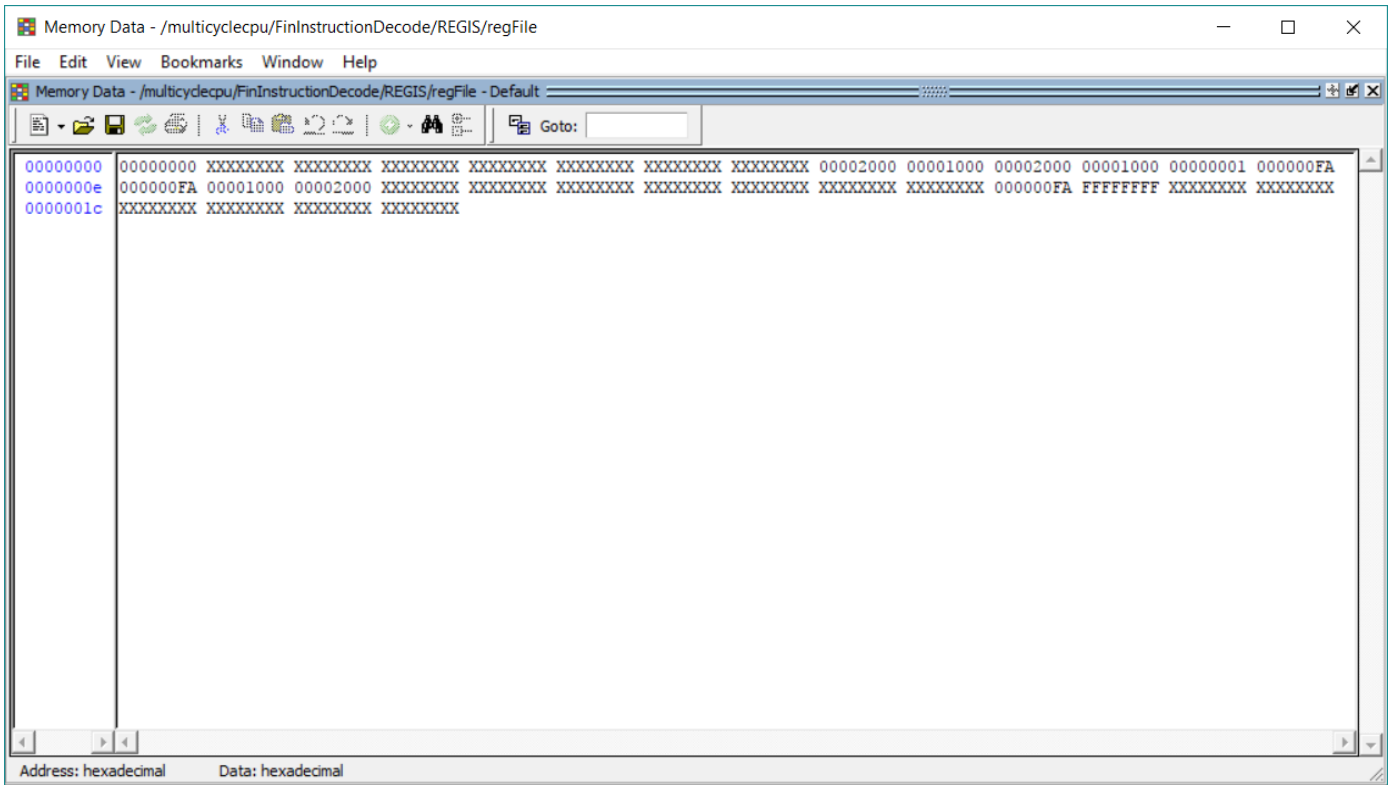

Figure 3. Register Memory Result after Simulation

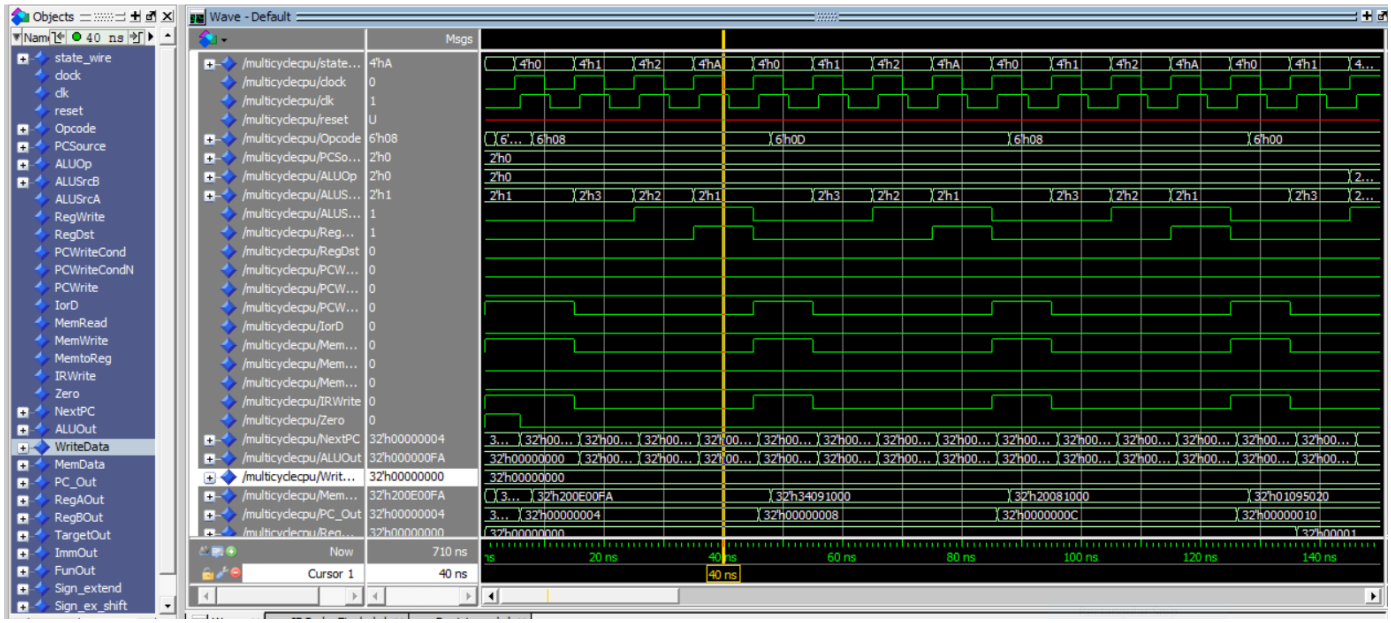

Figure 4. Waveform of CPU

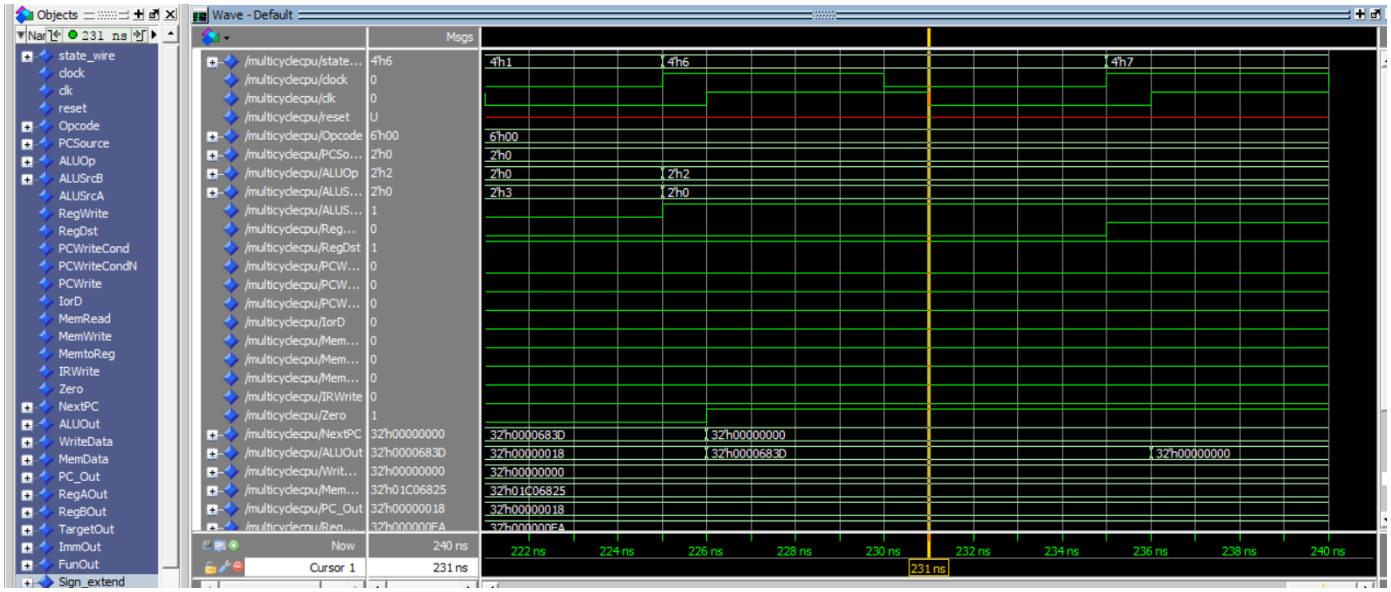

Figure 5. Waveform of CPU 2

V. Issue Analysis And Future Plan

From the beginning, we met issues with PC counter that cannot properly increment counter, which requires us to change the control unit logic to collaborate with an AND Gate of PCwriteN, PC combination as well as change the OR Gate at PC Counter Part. Also, the CPU clock rising edge and writing signals in different vhdl files are always the pain to debug, it is hard to decide whether the signal should rise or keep in the same format as clock. In the future, we consider the pipelined version of CPU and ore instruction sets will be implemented.

VI. Contribution for Group Member

The whole project, both of team members pretty much have done the 50-50 task and both Sicheng and Hector co-write this report. EX, ID VHDL design and MIPS.asm testing script were written by Sicheng. IF, MEM, Control logic, Shiftlogic were written by Hector. We collaborated each other well and have no dissatisfaction to each other.

VII. Acknowledgement

Hereby we cannot give to much thanks to our dedicated professor Erdal Orukulu who has always contributes his time to students and dedicated with his work. Also, we need appreciate our TAs who have always graded our homework in silent and must deserve the thank for hardworking.

APPENDIX

```
--*************************************************************************************************************
--*   Instruction Fetch Final Block
--*************************************************************************************************************

library ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.numeric_std.ALL;

ENTITY IFetchGroup IS
   PORT(

-- PC Register including combinational logic
      clk             :IN  std_logic;
      Zero            :IN  std_logic;
      PCWriteCond     :IN  std_logic;
      PCWriteCondN    :IN  std_logic;
      PCWrite         :IN  std_logic;
      NextPC          :IN  std_logic_vector( 31 DOWNTO 0 );

-- PC->MUX input
      IorD            :IN  std_logic;
      ALUOut          :IN  std_logic_vector( 31 DOWNTO 0 );


-- Memory inputs
      MemRead         :IN  std_logic;
      MemWrite        :IN  std_logic;
      WriteData       :IN  std_logic_vector( 31 DOWNTO 0 );


-- Memory Outputs
      MemData         :OUT  std_logic_vector( 31 DOWNTO  0 ):=x"00000000";
      PC_Out          :OUT  std_logic_vector( 31 DOWNTO  0 ):=x"00000000"
      );
END IFetchGroup;


ARCHITECTURE final OF IFetchGroup IS

   SIGNAL MuxOut     :  std_logic_vector( 31 DOWNTO  0 ):=x"00000000";
   SIGNAL pc_write   :  std_logic; -- W Signal
   SIGNAL PC         :  std_logic_vector( 31 DOWNTO  0 ):=x"00000000"; -- What type of Signal



BEGIN
   --pc_write   <= PCWrite OR  (Zero AND PCWriteCond);
   PC_Out      <= PC;
   MuxOut      <= ALUOut WHEN (IorD = '1') ELSE PC;



   DMemory: entity work.memory(multycycle) port map(
      MemRead  =>  MemRead,
```

```
        MemWrite =>   MemWrite,
        DataIn   =>   WriteData,
        AddressIn =>  MuxOut,
        DataOut =>    MemData
    );

  PROCESS(NextPC)
    BEGIN
    --if(rising_edge(clk)) then
       if NextPC(31) ='0' then
       IF ((PCWrite   OR  (Zero AND PCWriteCond) or ((not Zero) AND PCWriteCondN)) = '1' )  THEN
          PC <= NextPC ;
       ELSE
          PC <= PC;
       END IF;

    end if;
  END PROCESS;

  END ARCHITECTURE;

  --*************************************************************************************************************
  --*   Instruction Decode Final Block
  --*************************************************************************************************************
  LIBRARY ieee;
  USE ieee.std_logic_1164.ALL;
  USE ieee.std_logic_arith.ALL;
  USE ieee.std_logic_signed.ALL;
  USE ieee.numeric_std.ALL;

  ENTITY IDecodeGroup IS
    PORT(
       clk             :  IN   std_logic;
       InstrIn         :  IN   std_logic_vector(31 downto 0);
       IRWrite         :  IN   std_logic;
       RegDst          :  IN   std_logic;
       RegWrite        :  IN   std_logic;
       MemtoReg        :  IN   std_logic;
       AluResIn        :  in   std_logic_vector(31 downto 0);
       OpcodeOut       :  out  std_logic_vector(5 downto 0):="000000";
       RegAOut         :  out  std_logic_vector(31 downto 0):=x"00001234";
       RegBOut         :  out  std_logic_vector(31 downto 0):=x"00001234";
       TargetOut       :  out  std_logic_vector(25 downto 0):="00000000000000000000000100";
       ImmOut          :  out  std_logic_vector(15 downto 0):=x"1234";
       FunOut          :  out  std_logic_vector(5 downto 0):="000000"
       );
  END IDecodeGroup;

  ARCHITECTURE final OF IDecodeGroup IS
  -- I-type & R-type
       signal Rs     :  std_logic_vector(4 downto 0):="00000";
       signal Rt     :  std_logic_vector(4 downto 0):="00000";
  -- Exclusive R-type
       signal Rd     :  std_logic_vector(4 downto 0):="00000";
       signal SAOut      :  std_logic_vector(4 downto 0):="00000";

  -- Exclusive I-type
```

```
-- Exclusive J-type
        signal MemDataIn  :  std_logic_vector(31 downto 0):=x"00000000";
        signal RegAWire   :  std_logic_vector(31 downto 0):=x"00000000";
        signal RegBWire   :  std_logic_vector(31 downto 0):=x"00000000";
        SIGNAL RegAOutWire :  std_logic_vector(31 downto 0):=x"00000000";
        SIGNAL RegBOutWire :  std_logic_vector(31 downto 0):=x"00000000";
begin
    RegBOut <=RegBOutWire;
    RegAOut <=RegAOutWire;
    INSTREG: entity work.instructionregister(multycycle) port map(
        IRWrite     =>  IRWrite,
        InstrIn     =>  InstrIn,
        OpcodeOut   =>  OpcodeOut,
-- I-type & R-type
        RsOut       =>  Rs,
        RtOut       =>  Rt,
-- Exclusive R-type
        RdOut       =>  Rd,
        SAOut       =>  SAOut,
        FunOut      =>  FunOut,
-- Exclusive I-type
        ImmOut      =>  ImmOut,
-- Exclusive J-type
        TargetOut   =>  TargetOut
);

    MemDatReg: entity work.memorydataregister(multicycle) port map (
        clk         =>  clk,
        MemDataIn   =>  InstrIn,
        MemDataOut  =>  MemDataIn
    );

    REGIS:   entity work.Registers(multycycle) port map(

-- MUX Reg input
        clk         =>  clk,
        RegDst      =>  RegDst,
        RdIn        =>  Rd,

-- MUX data input and output
        MemtoReg    =>  MemtoReg,
        AluResIn    =>  AluResIn,
        MemDataIn   =>  MemDataIn,

-- Register input and outputs
        RegWrite    =>  RegWrite,

        RsIn        =>  Rs,
        RtIn        =>  Rt,
        RegAOut     =>  RegAWire,
        RegBOut     =>  RegBWire
);

    RegisterA: entity work.regA(multicycle) port map (
        clk         => clk,
        Data1In     => RegAWire,
```

```
      Data1Out    => RegAOutWire
   );
   RegisterB: entity work.regB(multicycle) port map (
      clk         => clk,
      Data2In     => RegBWire,
      Data2Out    => RegBOutWire
   );

END ARCHITECTURE;

--*************************************************************************************************************
--*   ALU Computing Final Block
--*************************************************************************************************************

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.numeric_std.ALL;

ENTITY ALU IS
   PORT(

--MUX_2, Reg_A, input and MUX_2_output
      clk             :IN   std_logic;
      ALUSrcA         :IN   std_logic;
      Reg_A           :IN   std_logic_vector( 31 DOWNTO 0 );
      PCin            :IN   std_logic_vector( 31 DOWNTO 0 );

--MUX_4, Reg_B, input and MUX_4_output
      ALUSrcB         :IN   std_logic_vector(  1 DOWNTO 0 );
      Reg_B           :IN   std_logic_vector( 31 DOWNTO 0 );

      Sign_extend     :IN   std_logic_vector( 31 DOWNTO 0 );
      Sign_ex_shift   :IN   std_logic_vector( 31 DOWNTO 0 );

--ALU Control and ALU Output
      IR              :IN   std_logic_vector(  5 DOWNTO  0 );
      ALUOp           :IN   std_logic_vector(  1 DOWNTO  0 );

      ALUResult       :OUT   std_logic_vector( 31 DOWNTO  0 ):=x"00000000";
      ALUOut          :OUT   std_logic_vector( 31 DOWNTO  0 ):=x"00000000";
      Zero            :OUT   std_logic :='0' );
END ALU;

ARCHITECTURE final OF ALU IS

SIGNAL  MUX_A_INPUT, MUX_B_INPUT  :   std_logic_vector( 31 DOWNTO 0 ):=x"00000000";
SIGNAL  ALU_MUX_OUTPUT            :   std_logic_vector( 31 DOWNTO 0 ):=x"00000000";
SIGNAL  ALU_CTR                   :   std_logic_vector(  2 DOWNTO 0 ):="000";
Signal  Input_4                   :std_logic_vector( 31 DOWNTO 0 ):=x"00000004";
BEGIN
   Input_4 <=  X"00000004";

--MUX Reg A Logic
   MUX_A:
   process(ALUSrcA,Reg_A,PCin)
```

```
    begin
    if ALUSrcA = '0' then

       MUX_A_INPUT <= PCin;
    ELSE
          MUX_A_INPUT <= Reg_A;
    end if;
    end process MUX_A;

  --MUX Reg B Logic
    MUX_B:
    PROCESS(ALUSrcB, Reg_B, Input_4, Sign_extend, Sign_ex_shift)
    BEGIN
       IF(ALUSrcB = "00" ) THEN
          MUX_B_INPUT <= Reg_B;
       ELSIF(ALUSrcB = "01") THEN
          MUX_B_INPUT <=Input_4; -- PC + 4
       ELSIF(ALUSrcB = "10") THEN
          MUX_B_INPUT <= Sign_extend;
       ELSIF(ALUSrcB = "11") THEN
          MUX_B_INPUT <= Sign_ex_shift;
       END IF;
    END PROCESS MUX_B;

  --ALU Control Logic
    ALU_CTR(0) <= ( IR(0) OR IR(3)) AND ALUOp(1); -- STILL NEED WORK CHECK
    ALU_CTR(1) <= ( NOT IR(2)) OR (NOT IR(1));
    ALU_CTR(2) <= ( IR(1) AND ALUOp(1)) OR ALUOp(0);

    Zero <= '1'   -- logic signal for ctr unit
     WHEN(ALU_MUX_OUTPUT( 31 DOWNTO 0 ) = (X"00000000"))
     ELSE '0';

    ALUResult <= ALU_MUX_OUTPUT( 31 DOWNTO 0 );

  --ALU Out
  -- Flip flop register
    ALUOutReg    : entity work.ALUOut(multicycle) port map(
                clk=>clk,
                ALUResIn      =>ALU_MUX_OUTPUT,
                ALUResOut    => ALUOut
  );

  --ALU Computation

    ALU_Compute:
    PROCESS (clk)--( MUX_A_INPUT, MUX_B_INPUT)

    BEGIN
    if(rising_edge(clk)) then
    CASE ALU_CTR IS

        WHEN "000" =>   ALU_MUX_OUTPUT   <= MUX_A_INPUT AND MUX_B_INPUT;    -- AND, ANDI
        WHEN "001" =>   ALU_MUX_OUTPUT   <= MUX_A_INPUT OR  MUX_B_INPUT; -- OR,  ORI
        WHEN "010" =>   ALU_MUX_OUTPUT   <= MUX_A_INPUT + MUX_B_INPUT;  -- ADD, ADDI
        WHEN "011" =>   ALU_MUX_OUTPUT   <= (others => '0'); -- new command
        WHEN "100" =>   ALU_MUX_OUTPUT   <= MUX_A_INPUT NOR MUX_B_INPUT; -- NOR
```

```
        WHEN "101" =>   ALU_MUX_OUTPUT   <= (others => '0'); -- new command
        WHEN "110" =>   ALU_MUX_OUTPUT   <= (MUX_A_INPUT - MUX_B_INPUT);-- SUB, SUBI
        WHEN "111" =>   IF MUX_A_INPUT < MUX_B_INPUT THEN    -- SLT
             ALU_MUX_OUTPUT   <= (X"00000001");
             ELSE
             ALU_MUX_OUTPUT   <= (X"00000000");
             END IF;

        WHEN OTHERS => ALU_MUX_OUTPUT   <= (others => '0');
    END CASE;
    end if;
      END PROCESS ALU_Compute;
END architecture;

--*************************************************************************************************************
--*   Jump Function Final Block
--*************************************************************************************************************
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;

entity jumpBlock is
    port(
        clk       : in   std_logic;
        target      : In    std_logic_vector(25 downto 0);
        pccont     : in   std_logic_vector(3 downto 0);
        ALUOut      : in   std_logic_vector(31 downto 0);
        ALUResult  : in   std_logic_vector(31 downto 0);
        PCSource: in   std_logic_vector( 1 DOWNTO 0 );
        NextPC       : out std_logic_vector(31 downto 0) :=X"00000000"
    );
end entity;
architecture final of jumpBlock is
    signal   JMPAddress    : std_logic_vector(31 downto 0):=X"00000000";
begin

    JMPAddress(31 DOWNTO  28) <= pccont;
    JMPAddress(27 DOWNTO  2)  <= target;
    JMPAddress(1 DOWNTO  0)   <= "00";
    process(ALUResult,ALUOut,JMPAddress,PCSource) begin
    --if(rising_edge(clk)) then
        case PCSource is
            when "00"   => NextPC <= ALUResult;
            when "01"   => NextPC <= ALUOut;
            when "10"   => NextPC <= JMPAddress;
            when others => report "Unreachable!" severity FAILURE;
        end case;
    --end if;
    end process;


end architecture;

--*************************************************************************************************************
--*   ALU Reg A Final Block
--*************************************************************************************************************
library ieee;
```

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

entity regB is
   port(
      clk         :  in  std_logic;
      Data2In          :  in  std_logic_vector(31 downto 0);
      Data2Out    :  out std_logic_vector(31 downto 0):=x"00000000"
   );
end entity;
architecture multicycle of regB is
   begin
      process (Data2In) begin--(clk) begin
         --if(rising_edge(clk)) then
            Data2Out <= Data2In;
         --end if;
      end process;
end architecture;

--*************************************************************************************************************
--*   ALU Reg B Final Block
--*************************************************************************************************************
library ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

entity regB is
   port(
      clk         :  in  std_logic;
      Data2In          :  in  std_logic_vector(31 downto 0);
      Data2Out    :  out std_logic_vector(31 downto 0):=x"00000000"
   );
end entity;
architecture multicycle of regB is
   begin
      process (Data2In) begin--(clk) begin
         --if(rising_edge(clk)) then
            Data2Out <= Data2In;
         --end if;
      end process;
end architecture;

--*************************************************************************************************************
--*   ALU Result Output Final Block
--*************************************************************************************************************
library ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

entity ALUOut is
   port(
      clk         :  in std_logic;
      ALUResIn      :  in  std_logic_vector(31 downto 0):=x"00000000";
      ALUResOut       :  out std_logic_vector(31 downto 0):=x"00000000"
   );
end entity;
architecture multicycle of ALUOut is
```

```
    begin
        process(clk) begin
            if(rising_edge(clk)) then
                ALUResOut    <= ALUResIn;
            end if;
        end process;
end architecture;

--*************************************************************************************************************
--*   Memory Data Register Final Block
--*************************************************************************************************************

library ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

entity memoryDataRegister is
    port(
        clk       :   in std_logic;
        MemDataIn :   in  std_logic_vector(31 downto 0);
        MemDataOut  :   out std_logic_vector(31 downto 0):=x"00000000"
    );
end entity;
architecture multicycle of memoryDataRegister is
    signal MemDataOutWire    : std_logic_vector(31 downto 0):=x"00000000";
    begin
    MemDataOut  <=    MemDataOutWire;
        process(clk) begin
            if(rising_edge(clk)) then
                MemDataOutWire  <= MemDataIn;
            end if;
        end process;
end architecture;


--*************************************************************************************************************
--*   CPU Control Final Block
--*************************************************************************************************************
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY Control IS
    PORT(
        clock           :IN   std_logic;
        reset           :IN   std_logic;
        Opcode          :IN   std_logic_vector( 5 DOWNTO 0 );

        PCSource        :OUT   std_logic_vector( 1 DOWNTO 0 ):="00";
        ALUOp           :OUT   std_logic_vector( 1 DOWNTO 0 ):= "00";
        ALUSrcB         :OUT   std_logic_vector( 1 DOWNTO 0 ):="01";
        ALUSrcA         :OUT   std_logic:= '0';
        RegWrite        :OUT   std_logic:= '0';
        RegDst          :OUT   std_logic:= '0';
        PCWriteCondN    :OUT   std_logic:='0';
        PCWriteCond     :OUT   std_logic:= '0';
```

```
        PCWrite          :OUT   std_logic:= '0';
        IorD             :OUT   std_logic:= '0';
        MemRead          :OUT   std_logic:= '0';
        MemWrite         :OUT   std_logic:= '0';
        MemtoReg         :OUT   std_logic:= '0';
        IRWrite          :OUT   std_logic:= '0';
        state_wire       : OUT  std_logic_vector(3 downto 0)
);
END Control;

ARCHITECTURE final OF Control IS

COMPONENT LCELL              -- How about get rid off theses lines of code?
   PORT( a_in  :IN  std_logic;
      a_out    :OUT std_logic);
   END COMPONENT;

TYPE State IS(si,s0, s1, s2, s3, s4, s5, s6, s7,s8, s9,s10);

SIGNAL next_state :State;
signal wake: std_logic:='0';
SIGNAL current_state:State:=si;
SIGNAL Opcode_out :std_logic_vector( 5 DOWNTO 0 ); --

BEGIN
   OP_BUf0: LCELL  PORT MAP( a_in => Opcode(0), a_out => Opcode_out(0));
   OP_BUf1: LCELL  PORT MAP( a_in => Opcode(1), a_out => Opcode_out(1));
   OP_BUf2: LCELL  PORT MAP( a_in => Opcode(2), a_out => Opcode_out(2));
   OP_BUf3: LCELL  PORT MAP( a_in => Opcode(3), a_out => Opcode_out(3));
   OP_BUf4: LCELL  PORT MAP( a_in => Opcode(4), a_out => Opcode_out(4));
   OP_BUf5: LCELL  PORT MAP( a_in => Opcode(5), a_out => Opcode_out(5));

state_trans:  PROCESS(current_state,Opcode, next_state,wake ) --, Opcode_out, next_state,wake)

BEGIN
   CASE current_state IS
--initialize
WHEN si =>
       state_wire<= X"F";
      MemRead <= '1';
      ALUSrcA <= '0';
      IorD     <= '0';
      IRWrite<= '1';
      ALUSrcB <= "01";
      ALUOp<= "00";
      PCWrite   <= '1';
      PCSource <= "00";

      RegWrite  <= '0';
      MemWrite    <= '0';
      PCWriteCond   <= '0';
      PCWriteCondN    <= '0';
      next_state <= s0; --Instruction Fetch
   WHEN s0 =>
       state_wire<= X"0";
      MemRead <= '1';
      ALUSrcA <= '0';
```

```
    IorD     <= '0';
    IRWrite<= '1';
    ALUSrcB <= "01";
    ALUOp<= "00" after 10 ps;
    PCWrite   <= '1';
    PCSource <= "00";

    RegWrite  <= '0';
    MemWrite    <= '0';
    PCWriteCond   <= '0';
    PCWriteCondN    <= '0';
    next_state <= s1; --Instruction Fetch

  WHEN s1 =>
     state_wire<= X"1";
    ALUSrcA <= '0';
    ALUSrcB <= "11";
    ALUOp<= "00";

    RegWrite  <= '0';
    MemRead <= '0';
    MemWrite    <= '0';
    IRWrite<= '0';
    PCWrite   <= '0';
    PCWriteCond   <= '0'; -- Block Instruction Compilation
     PCWriteCondN   <= '0';


    --   ORIGINALLY Opcode_out
    IF      Opcode = "100011"THEN next_state <= s2; -- lw opcode
    ELSIF   Opcode = "101011"THEN next_state <= s2; -- sw opcode
     ELSIF  Opcode = "001000"THEN next_state <= s2; -- addi opcode
     ELSIF  Opcode = "001100"THEN next_state <= s2; -- andi opcode
     ELSIF  Opcode = "001101"THEN next_state <= s2; -- ori opcode
    ELSIF   Opcode = "000000"THEN next_state <= s6; -- R-opcode
    ELSIF   Opcode = "000100"THEN next_state <= s8; -- beq opcode
     ELSIF  Opcode = "000101"THEN next_state <= s8; -- beq opcode
    ELSIF   Opcode = "000010"THEN next_state <= s9; -- j opcode
    ELSE        next_state <= next_state;
    END IF; --Instruction decode/ Register Fetch

  WHEN s2 =>
     state_wire<= X"2";
    ALUSrcA <= '1';
    ALUSrcB <= "10";
    ALUOp<= "00";

    RegWrite  <= '0';
    MemRead <= '0';
    MemWrite    <= '0';
    IRWrite<= '0';
    PCWrite   <= '0';
    PCWriteCond   <= '0';
     PCWriteCondN   <= '0';

    --   ORIGINALLY Opcode_out
    IF      Opcode = "100011"THEN next_state <= s3;
```

```
    ELSIF Opcode = "001000" THEN next_state <= s10; -- addi opcode
    ELSIF Opcode = "001100" THEN next_state <= s10; -- andi opcode
    ELSIF Opcode = "001101" THEN next_state <= s10; -- ori opcode
   ELSIF  Opcode = "101011" THEN next_state <= s5;
   ELSE       next_state <= next_state;  -- Stable Purpose
   END IF;

  WHEN s3 =>
    state_wire<= X"3";
   MemRead <= '1';
   IorD    <= '1';

   RegWrite  <= '0';
   MemWrite   <= '0';
   IRWrite<= '0';
   PCWrite   <= '0';
   PCWriteCond   <= '0';
    PCWriteCondN   <= '0';
   if Opcode ="100011" then   next_state    <= s4;
    else  next_state    <= s10;
    end if;
  WHEN s4 =>
    state_wire<= X"4";
   RegDst <= '0';  --Report Indicates  RegDst  <= '0'
   RegWrite  <= '1';
   MemtoReg   <= '1';

   MemRead <= '0';
   MemWrite   <= '0';
   IRWrite<= '0';
   PCWrite   <= '0';
   PCWriteCond   <= '0';
   PCWriteCondN    <= '0';
   next_state <= s0;
  WHEN s10 =>
    state_wire<= X"A";
   RegDst <= '0';  --Report Indicates  RegDst  <= '0'
   RegWrite  <= '1';
   MemtoReg   <= '0';
   ALUSrcB <= "01";
   ALUOp<= "00";
   MemRead <= '0';
   MemWrite   <= '0';
   IRWrite<= '0';
   PCWrite   <= '0';
   PCWriteCond   <= '0';
   PCWriteCondN    <= '0';
   next_state <= s0;
  WHEN s5 =>
    state_wire<= X"5";
   MemWrite   <= '1';
   IorD    <= '1';

   RegWrite  <= '0';
   MemRead <= '0';
   IRWrite<= '0';
   PCWrite   <= '0';
```

```
    PCWriteCond   <= '0';
    PCWriteCondN    <= '0';
    next_state <= s0;

  WHEN s6 =>
     state_wire<= X"6";
    ALUSrcA <= '1';
    ALUSrcB <= "00";
    ALUOp<= "10";

    RegWrite  <= '0';
    MemRead <= '0';
    IRWrite<= '0';
    PCWrite   <= '0';
    PCWriteCond   <= '0';
     PCWriteCondN   <= '0';
    MemWrite    <= '0';

    next_state <= s7;

  WHEN s7 =>
     state_wire<= X"7";
    RegDst <= '1';
    RegWrite  <= '1';
    MemtoReg   <= '0';

    MemRead <= '0';
    MemWrite    <= '0';
    IRWrite<= '0';
    PCWrite   <= '0';
    PCWriteCond   <= '0';
     PCWriteCondN   <= '0';
    next_state <= s0;

  WHEN s8 =>
    state_wire<= X"8";
    ALUSrcA <= '1';
    ALUSrcB <= "00";
    ALUOp<= "01";
    PCSource <= "01";
    if Opcode  ="000100" then
        PCWriteCond      <= '1';
     else
        PCWriteCondN   <= '1';
     end if;
    RegWrite  <= '0';
    MemRead <= '0';
    MemWrite    <= '0';
    IRWrite<= '0';
    PCWrite   <= '0';

    next_state <= s0;
  WHEN s9 =>
    state_wire<= X"9";
    PCSource <= "10";
    PCWrite   <= '1';
```

```
        RegWrite  <= '0';
        MemRead <= '0';
        MemWrite    <= '0';
        IRWrite<= '0';
        PCWriteCond   <= '0';
        PCWriteCondN    <= '0';
        next_state <= s0;
      END CASE;
  END PROCESS state_trans;

  state_clock:PROCESS(clock,reset)
  BEGIN
     IF reset='1' THEN
        current_state <= s0;

     ELSIF clock'EVENT and clock='1' THEN
     current_state <= next_state;
     wake <= not wake;
     END IF;
  END PROCESS state_clock;

  END architecture;

  --*************************************************************************************************************
  --*   Instruction Register Final Block
  --*************************************************************************************************************
  library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
  entity instructionRegister is
     port(
        IRWrite       : in std_logic:='1';
        InstrIn     : in std_logic_vector(31 downto 0):=x"31291234";
        OpcodeOut  : out std_logic_vector(5 downto 0):="000000";
  -- I-type & R-type
        RsOut      : out std_logic_vector(4 downto 0):="01001";
        RtOut      : out std_logic_vector(4 downto 0):="01001";
  -- Exclusive R-type
        RdOut      : out std_logic_vector(4 downto 0):="01001";
        SAOut      : out std_logic_vector(4 downto 0):="01001";
        FunOut       : out std_logic_vector(5 downto 0):="100000";
  -- Exclusive I-type
        ImmOut       : out std_logic_vector(15 downto 0):=x"0004";
  -- Exclusive J-type
        TargetOut    : out std_logic_vector(25 downto 0):="00000000000000000000000100"
     );
  end entity;
  architecture multycycle of instructionRegister is
     signal OpCodeOutWire   : std_logic_vector(5 downto 0):="000000";
  -- I-type & R-type
     signal RsOutWire    : std_logic_vector(4 downto 0):="01001";
     signal RtOutWire    : std_logic_vector(4 downto 0):="01001";
  -- Exclusive R-type
     signal RdOutWire    : std_logic_vector(4 downto 0):="01001";
     signal SAOutWire    : std_logic_vector(4 downto 0):="01001";
     signal FunOutWire  : std_logic_vector(5 downto 0):="000000";
  -- Exclusive I-type
```

```
    signal ImmOutWire: std_logic_vector(15 downto 0):=x"0004";
-- Exclusive J-type
    signal TargetOutWire : std_logic_vector(25 downto 0):="00000000000000000000000100";

begin

    OpcodeOut  <= OpCodeOutWire;

    RsOut      <= RsOutWire;
    RtOut      <= RtOutWire;

    RdOut      <= RdOutWire;
    SAOut      <= SAOutWire;
    FunOut       <= FunOutWire;

    ImmOut       <= ImmOutWire;

    TargetOut    <= TargetOutWire;


    process (IRWrite,InstrIn) begin
        if(IRWrite ='1') then
            OpCodeOutWire     <= InstrIn(31 downto 26);

            RsOutWire  <= InstrIn(25 downto 21);
            RtOutWire  <= InstrIn(20 downto 16);

            RdOutWire  <= InstrIn(15 downto 11);
            SAOutWire <= InstrIn(10 downto 6 );
            FunOutWire    <= InstrIn(5  downto 0 );

            ImmOutWire   <= InstrIn(15 downto 0 );

            TargetOutWire    <= InstrIn(25 downto 0);
        end if;
    end process;
end architecture;

--*************************************************************************************************************
--*   Memory Buffer
--*************************************************************************************************************
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity memory is generic(ADDRESSBITS : integer:=10);
    port(
        MemRead       : In std_logic;
        MemWrite   : In std_logic;
        DataIn     : In std_logic_vector(31 downto 0);
        AddressIn    : in std_logic_vector(31 downto 0);
        DataOut      : out std_logic_vector(31 downto 0):=x"00000000"
    );
end entity;

architecture multycycle of memory is
```

```
    signal dataOut1          : std_logic_vector(31 downto 0) :=x"31291234";
    signal addressIn1    : std_logic_vector(9 downto 0) :="0000000000";
    constant RAM_LINES       : integer := 2**ADDRESSBITS;
    type RAM is array(integer range<>) of std_logic_vector(31 downto 0);
    signal Mem         : RAM (0 to RAM_LINES-1);

begin
DataOut <=dataOut1;
-- addressIn1 <=AddressIn(9 downto 0);



Mem_Write:
    process (AddressIn)
        begin
            if(MemWrite='1') then
                mem(conv_integer(AddressIn)/4) <=DataIn ;

            end if;
    end process;


Mem_Read:
    process (MemWrite,DataIn,AddressIn,MemRead)
        begin
            if(MemRead='1') then
                dataOut1 <= mem(conv_integer(AddressIn)/4) after 2 ns;
            end if;
    end process;
end architecture;


--*********************************************************************************************************
--*   Core CPU
--*********************************************************************************************************
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
use ieee.numeric_std.all;
entity multicycleCPU is
end entity;

Architecture summary of multicycleCPU is
        signal  state_wire      :  std_logic_vector(3 downto 0);
        signal  clock         :  std_logic :='0';
        signal  clk         :  std_logic :='0';
        signal  reset       :  std_logic;
        signal  Opcode          :  std_logic_vector( 5 DOWNTO 0 ):= "001000";

        signal  PCSource    :  std_logic_vector( 1 DOWNTO 0 ):="00";
        signal  ALUOp           :  std_logic_vector( 1 DOWNTO 0 ):="00";
        signal  ALUSrcB             :  std_logic_vector( 1 DOWNTO 0 ):="00";
        signal  ALUSrcA             :  std_logic:='0';
        signal  RegWrite    :  std_logic:='0';
        signal  RegDst      :  std_logic:='0';
```

```
        signal  PCWriteCond     :  std_logic:='0';
        signal  PCWriteCondN     :  std_logic:='0';
        signal  PCWrite         :  std_logic:='0';
        signal  IorD        :  std_logic:='0';
        signal  MemRead         :  std_logic:='0';
        signal  MemWrite     :  std_logic:='0';
        signal  MemtoReg     :  std_logic:='0';
        signal  IRWrite         :  std_logic:='0';

    --Instruction Fetch signals
        signal  Zero        :  std_logic:='0';
        signal  NextPC          :  std_logic_vector( 31 DOWNTO 0 ):=X"00000000";
        signal  ALUOut          :  std_logic_vector( 31 DOWNTO 0 ):=X"00000000";
        signal  WriteData       :  std_logic_vector( 31 DOWNTO 0 ):=X"00000000";

    -- out
        signal  MemData         :  std_logic_vector( 31 DOWNTO  0 ):=X"00000000";
        signal  PC_Out          :  std_logic_vector( 31 DOWNTO  0 ):=X"00000000";

    --Instruction Decode signals
    -- out
        signal  RegAOut         :  std_logic_vector(31 downto 0):=X"00000000";
        signal  RegBOut         :  std_logic_vector(31 downto 0):=X"00000000";
        signal  TargetOut       :  std_logic_vector(25 downto 0):="00000000000000000000000000";

        signal  ImmOut          :  std_logic_vector(15 downto 0):=x"0000";
        signal  FunOut          :      std_logic_vector(5 downto 0):="000000";

        signal  Sign_extend     :  std_logic_vector(31 downto 0):=x"00000000";
        signal  Sign_ex_shift       :  std_logic_vector(31 downto 0):=x"00000000";
        signal  ALUResult       :  std_logic_vector(31 downto 0):=x"00000000";

    begin
       FinControlUnit    : entity work.control(final) port map(
          clock          =>       clock,
          reset          =>       reset,
          Opcode             =>       Opcode,
          PCSource       =>       PCSource,
          ALUOp          =>       ALUOp,
          ALUSrcB        =>       ALUSrcB,
          ALUSrcA        =>       ALUSrcA,
          RegWrite       =>       RegWrite,
          RegDst         =>       RegDst,
          PCWriteCond      =>       PCWriteCond,
          PCWriteCondN       =>       PCWriteCondN,
          PCWrite      =>       PCWrite,
          IorD           =>       IorD,
          MemRead        =>       MemRead,
          MemWrite       =>       MemWrite,
          MemtoReg       =>       MemtoReg,
          IRWrite        =>       IRWrite,
          state_wire         =>    state_wire
    );

       FinInstructionFetch : entity work.IFetchGroup(final) port map(
          clk          =>    clk,
          Zero         =>    Zero,
```

```
        PCWriteCond     =>  PCWriteCond,
        PCWriteCondN    =>  PCWriteCondN,
        PCWrite         =>  PCWrite,
        NextPC          =>  NextPC,

    --PC->MUX input

        IorD        =>  IorD,
        ALUOut      =>  ALUOut,

    --Memory inputs

        MemRead         =>  MemRead,
        MemWrite    =>  MemWrite,
        WriteData   =>  WriteData,

    --Memory Outputs

        MemData     =>  MemData,
        PC_Out      =>  PC_Out
    );
       FinInstructionDecode : entity work.idecodeGroup(final) port map(
        clk         =>  clk,
        InstrIn     =>  MemData,
        IRWrite     =>  IRWrite,
        RegDst      =>  RegDst,
        RegWrite    =>  RegWrite,
        MemtoReg    =>  MemtoReg,
        AluResIn    =>  ALUOut,
        OpcodeOut   =>  Opcode,
    --      out
        RegAOut     =>  RegAOut,
        RegBOut     =>  RegBOut,
        TargetOut   =>  TargetOut,
        ImmOut      =>  ImmOut,
        FunOut      =>  FunOut
    );
       Sign_extend <= std_logic_vector(resize(signed(ImmOut), Sign_extend'length));
       Sign_ex_shift <="00000000000000" & ImmOut &"00";

        FinALU  : entity work.ALU(final) port map(

    --MUX_2, Reg_A, input and MUX_2_output

        clk         =>  clk,
        ALUSrcA     =>  ALUSrcA,
        Reg_A       =>  RegAOut,
        PCin        =>  PC_Out,

    --MUX_4, Reg_B, input and MUX_4_output
        ALUSrcB     =>  ALUSrcB,
        Reg_B       =>  RegBOut,
        Sign_extend =>  Sign_extend,
        Sign_ex_shift   =>  Sign_extend,

    --ALU Control and ALU Output
        IR      =>  FunOut,
```

```
        ALUOp          =>   ALUOp,

        ALUResult      =>   ALUResult,
        ALUOut         =>   ALUOut,
        Zero       =>   Zero
    );
    FinJump  : entity work.jumpBlock(final) port map(
        clk          =>   clk,
        target           =>   TargetOut,
        pccont       =>   PC_Out(31 downto 28),
        ALUOut       =>   ALUOut,
        ALUResult    =>   ALUResult,
        PCSource     =>   PCSource,
        NextPC           =>   NextPC
    );

       clk_process :process
      begin

        clock <= '0';
          wait for 1 ns;
            clk <='0';
        wait for 4 ns;  --for 0.5 ns signal is '0'.
          clock <= '1';
        wait for 1 ns;  --for next 0.5 ns signal is '1'.
          clk <='1';
          wait for 4 ns;  --for 0.5 ns signal is '0'.

      end process;

    end architecture;
    --*************************************************************************************************************
    --*   Registers
    --*************************************************************************************************************

    library ieee;
       USE ieee.std_logic_1164.ALL;
       USE ieee.std_logic_unsigned.ALL;

    entity registers  is
       port(

    --*************************************************
    --*             MUX reg input
    --*************************************************
        clk          : in std_logic;
        RegDst     : in   std_logic;
        RdIn         : in std_logic_vector(4 downto 0);

    --*************************************************
    --*             MUX data input and output
    --*************************************************
        MemtoReg       : in   std_logic;
        AluResIn     : in   std_logic_vector(31 downto 0);
        MemDataIn : in   std_logic_vector(31 downto 0);

    --*************************************************
```

```
    --*              Register input and output
    --************************************************
          RegWrite    :   in std_logic:='0';

          RsIn        :   in std_logic_vector(4  downto 0);
          RtIn        :   in std_logic_vector(4  downto 0);

          RegAOut     :  out std_logic_vector(31 downto 0):=x"00000000";
          RegBOut     :  out std_logic_vector(31 downto 0):=x"00000000"
       );
    end entity;

    architecture multycycle of registers is
    --regFile as memory 32 registers 32 bits each in MIPS
       type RAM is array (integer range<>) of std_logic_vector(31 downto 0);
       signal regFile      : RAM(0 to 31);
       signal RegAOutWire  : std_logic_vector(31 downto 0):=x"00000000";
       signal RegBOutWire  : std_logic_vector(31 downto 0):=x"00000000";
       signal MuxRegIn     : std_logic_vector(4  downto 0):="00000";
       signal MuxDataIn       : std_logic_vector(31 downto 0):=X"00000000";

    begin
       RegAOut <=RegAOutWire;
       RegBOut <=RegBOutWire;

    -- Destination Register Mux
       process(RegDst,RtIn,RdIn) begin
          case RegDst is
             when '0'   => MuxRegIn <= RtIn;
             when '1' => MuxRegIn <= RdIn;
             when others => report "Unreachable!" severity FAILURE;
          end case;
       end process;
    -- Data write Mux
       process(MemtoReg,AluResIn) begin
          case MemtoReg is
             when '0'   => MuxDataIn <= AluResIn;
             when '1' => MuxDataIn <= MemDataIn;
             when others => report "Unreachable!" severity FAILURE;
          end case;
       end process;
    --Read Function
       Reg_Read:
       process(clk,RsIn,RtIn) begin
          if(rising_edge(clk)) then
             RegAOutWire     <=   regFile(conv_integer(RsIn));
             RegBOutWire     <=   regFile(conv_integer(RtIn));
          end if;
       end process;
    --Read Function
       Reg_Write:
       process(MuxRegIn,MuxDataIn) begin
          if (RegWrite ='1') then
             regFile(conv_integer(MuxRegIn)) <= MuxDataIn;
          end if;
       end process;
    end architecture;
```

REFERENCES

[1] ECE 485 Final Project Design and Implementation of a MIPS CPU with Multicycle Datapath (version 1.0) Dr. Erdal Oruklu
[2] ECE 485 VHDL Introduction Dr. Erdal Oruklu
[3] Compter Organization design Design and Implementation of MIPS CPU with Multicycle Datapath https://github.com/AdalbertoCq/Design.and.Implementation.of.a.MIPS.CPU.with.Multicycle.Datapath/blob/master/Design%20and%20Implementation.of.a.MIPS.CPU.with.Multicycle.Datapath.pdf
[4] MIPS Instruction Reference http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html
[5] Single Cycle MIPS Processor http://www.cs.columbia.edu/~martha/courses/3827/sp11/slides/9_singleCycleMIPS.pdf
[6] A Multicycle Data Path Implementation https://blackboard.iit.edu/bbcswebdav/pid-568908-dt-content-rid-6552256_1/courses/X9101475.201810/ece485project.pdf