EE309

Microprocessor

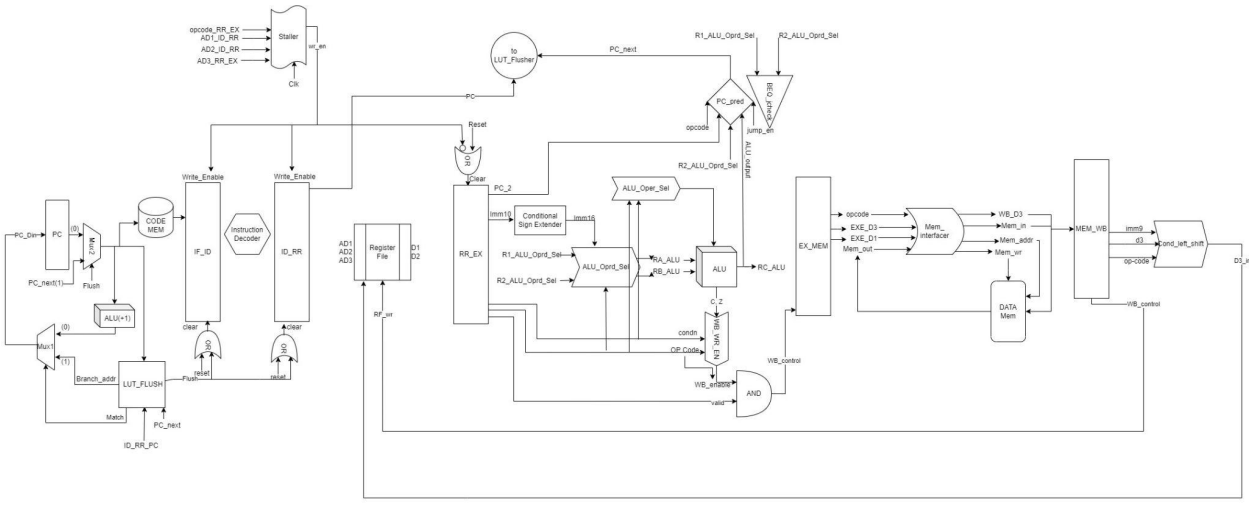
Project 2: Pipelined Processor

Team Members:

Aayush M Gopal	200020004
Navneet	200070048
Tanmay Dokania	200070083
Vansh Kapoor	200100164

Table Of Contents:

Datapath



Component Documentation

Write Enable

This is used to control the write back based on the following conditions.

1) For ADC/ADZ/NDC/NDZ disable if C or Z flag is 0 else enable.
2) Unconditionally disable if BEQ or JRI
3) Else unconditionally enable.

```
entity write_enable is
     port(
          opcode: in std_logic_vector(3 downto 0);
          CZ: in std_logic_vector(1 downto 0);
                C,Z: in std_logic;
                WB_enable: out std_logic
                );
end entity;
```

BEQ j_check

This is used to check if we have to jump or not by comparing RA and RB registers

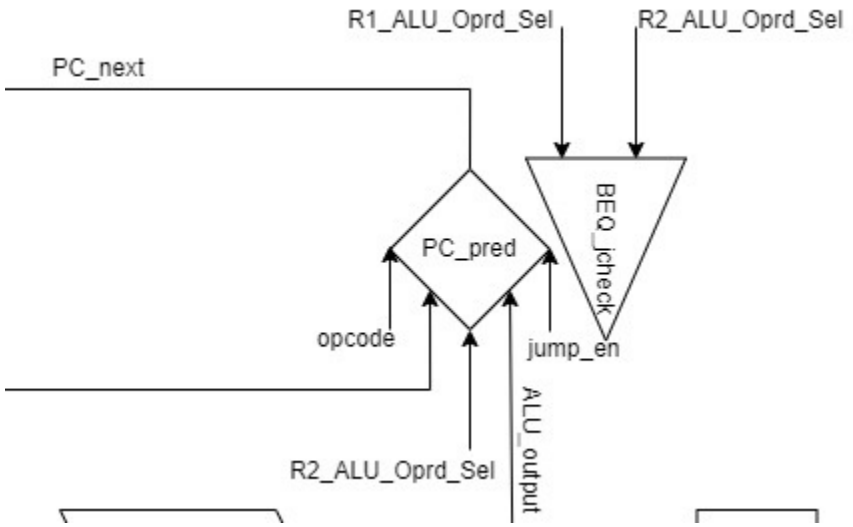
```
entity BEQ_jcheck is
     port(
          RA: in std_logic_vector(15 downto 0);
        RB: in std_logic_vector(15 downto 0);
        jump_enable: out std_logic
        );
end entity;
```

pc_pred

This is used to predict the new value to be loaded in pc based on following condition.

1) JLR -> pc_next = RB
2) BEQ -> based on the condition either pc_next=alu_out or pc_next=pc_2
3) JRI/JAL -> pc_next=alu_out
4) For other instruction pc_next is pc_2

```
entity pc_pred is
port(
  pc_2, RB, alu_out: in std_logic_vector(15 downto 0);
  opcode: in std_logic_vector(3 downto 0);
  jump_enable: in std_logic;
  pc_next: out std_logic_vector(15 downto 0)
                );
end entity;
```



cond_left_shift

This is used such then when an LHI instruction is encountered it takes the immediate value and returns out a left_shift_7 value else it returns back d3 as output.

```
entity cond_left_shift is
port(
     immediate: in std_logic_vector(8 downto 0);
     opcode: in std_logic_vector(3 downto 0);
     d3: in std_logic_vector(15 downto 0);
      d3_out: out std_logic_vector(15 downto 0)
               );
end entity;
```

Addr-cmp

This is used to compare the input (comparator) , output is 1 if both are same is output is 0.

```
entity Addr_cmp is
     port(
          addr1: in std_logic_vector(2 downto 0);
                addr2: in std_logic_vector(2 downto 0);
```

```
                output_match: out std_logic
                );
end entity;
```

alu_oper_sel

This is used to select the operands that are going into the alu based on the instructions as following

1) LHI/JRI -> OPR1 <= RA OPR2 <= immediate
2) LW/SW -> OPR1 <= RB OPR2 <= immediate
3) BEQ/JAL -> OPR1 <= PC OPR2 <= immediate
4) OTHERS -> OPR1 <= RA OPR2 <= RB

```
entity alu_oper_sel is
     port(
          opcode: in std_logic_vector(3 downto 0);
          RA: in std_logic_vector(15 downto 0);
         RB: in std_logic_vector(15 downto 0);
        immediate: in std_logic_vector(15 downto 0);
        PC: in std_logic_vector(15 downto 0);
        OPR1: out std_logic_vector(15 downto 0);
        OPR2: out std_logic_vector(15 downto 0)
                );
end entity;
```

sel_sign_extend

This is used to conditionally extends the immediate value which is given as input (10 bits) in the format (c123456789 where 1-9 immediate c- condition bit). The MSB decides the format.

1) c=1 then the immediate is considered as 9 bit immediate and sign extended to 16 bit
2) c=0 immediate is considered as 6bit immediate (cxxx123456) and sign extended to 16 bit.

```
entity sel_sign_extender is
     generic (
          inp_width : integer := 10;
          outp_width : integer := 16);
     port (
     inp : in std_logic_vector(inp_width - 1 downto 0);
     outp : out std_logic_vector(outp_width - 1
downto 0)
     );
end entity;
```

priority_mux

This is used to generate output based on the select key provided as follows

1) sel=00 ->output = input1
2) sel=01 ->output = input2
3) sel=10 ->output = input3

```
entity priority_mux is
    port(inp1,  inp2, inp3: in std_logic_vector(15 downto 0),
        sel: in std_logic_vector(1 downto 0),
        output: std_logic_vector(15 downto 0));
end entity;
```

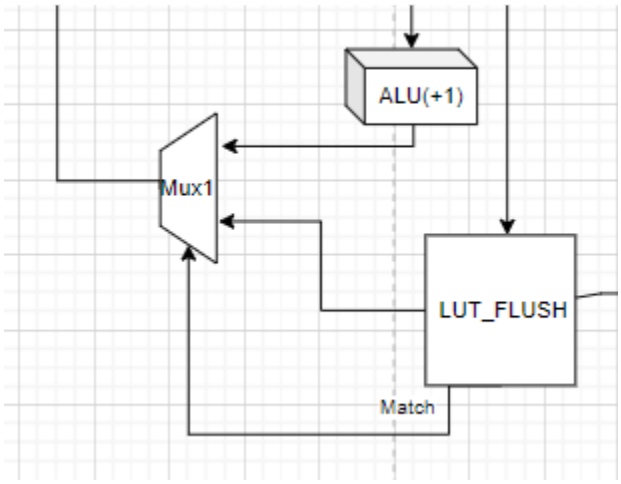
LUT

This is used to predict and store the decision values based on which branching prediction is made. It also nullifies incorrectly predicted instructions by preventing write-back to registers and Data memory and also “flushing” out the information carried by the pipeline registers regarding them.

It first checks if the address of the next instruction is present in the lookup table using the current PC, and generates the select input to the mux(match), Branch address accordingly.

It also compares the PC_PRED with PC_ID_RR and generates the “Flush” output =1 if both of them don’t match and according to updates/registers the correct Branch address for the current address(PC_RR_EXE) in the Lookup table.

```
entity lut is
     port (
          if_m1_out, pc_exe, pc_pred, pc_rr : in
std_logic_vector(15 downto 0);
          clk : in std_logic;
          match, clr : out std_logic;
          branch_addr : out std_logic_vector(15 downto 0)
     );
end entity;
```



pipeline_register

This is used to generate various pipelining registers required to be used and what parameters goes inside and what comes out of these registers.

```
entity pipe_reg is
     generic ( data_width : integer);
     port(
          clk, wr_enable, clr: in std_logic;
          Din: in std_logic_vector(data_width-1 downto 0);
          Dout: out std_logic_vector(data_width-1 downto    0));
```

```
end entity;

entity pipe_bit is
     generic ( data_width : integer);
     port(
          clk, wr_enable, clr: in std_logic;
          Din: in std_logic;
          Dout: out std_logic);
end entity;
```

Pipeline Register's Package

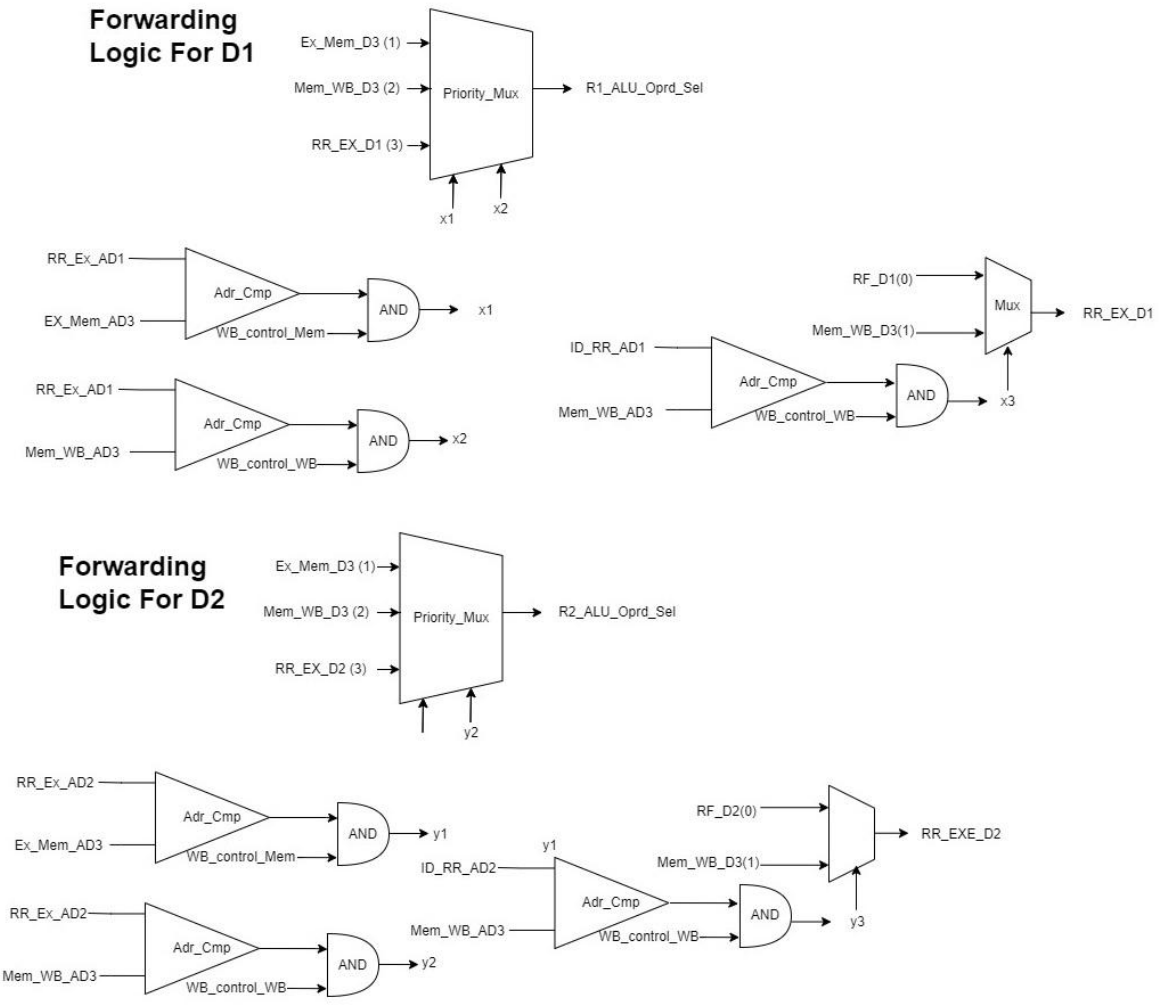
```
package pipeline_register is

         component pipe_IFD is
          port(
               pc, pc_2, inst : in std_logic_vector(15 downto
0);
               valid: in std_logic;
               clk: in std_logic;
                  clear: in std_logic;
               write_enable: in std_logic;
               valid_out: out std_logic;
                     pc_out, pc_2_out, inst_out : out
std_logic_vector(15 downto 0)
                     );

       end component;




       component pipe_IDRR is
          port(
               pc, pc_2, inst : in std_logic_vector(15 downto
0);
               valid: in std_logic;
               clk: in std_logic;
               cond: in std_logic_vector(1 downto 0);
```

```
                AD1, AD2, AD3: in std_logic_vector(2 downto 0);
                          write_enable: in std_logic;
                clear: in std_logic;
                valid_out: out std_logic;
                cond_out: out std_logic_vector(1 downto 0);
                AD1_out, AD2_out, AD3_out: out std_logic_vector(2
downto 0);
                        pc_out, pc_2_out, inst_out : out
std_logic_vector(15 downto 0)
                        );

        end component;



      component pipe_RREX is
          port(
                pc, pc_2, inst : in std_logic_vector(15 downto
0);
                D1, D2 : in std_logic_vector(15 downto 0);
                immd: in std_logic_vector(9 downto 0); --- this
is immediate data    MSB(9-1) merged with selector bit LSB(0)
                valid: in std_logic;
                clk: in std_logic;
                cond: in std_logic_vector(1 downto 0);
                AD1, AD2, AD3: in std_logic_vector(2 downto 0);
                clear: in std_logic;
                          write_enable: in std_logic;
                valid_out: out std_logic;
                cond_out: out std_logic_vector(1 downto 0);
                AD1_out, AD2_out, AD3_out: out std_logic_vector(2
downto 0);
                D1_out, D2_out : out std_logic_vector(15 downto
0);
                immd_out: out std_logic_vector(9 downto 0);
                pc_out, pc_2_out, inst_out : out
std_logic_vector(15 downto 0)
                );

        end component;
```

```
     component pipe_EXMOP is
          port(
               pc, pc_2, inst : in std_logic_vector(15 downto
0);
               D1, D3 : in std_logic_vector(15 downto 0);
               immd: in std_logic_vector(9 downto 0); --- this
is immediate data MSB(9-1) merged with selector bit LSB(0)
               valid, C, Z, wb_control: in std_logic;
               clk: in std_logic;
               cond: in std_logic_vector(1 downto 0);
                  AD3: in std_logic_vector(2 downto 0);
               clear: in std_logic;
                        write_enable: in std_logic;
               valid_out, C_out, Z_out, wb_control_out: out
std_logic;
               cond_out: out std_logic_vector(1 downto 0);
                  AD3_out: out std_logic_vector(2 downto 0);
               D1_out, D3_out : out std_logic_vector(15 downto
0);
               immd_out: out std_logic_vector(9 downto 0);
               pc_out, pc_2_out, inst_out : out
std_logic_vector(15 downto 0)
               );

       end component;



        component pipe_MOPWB is
          port(
               pc, pc_2, inst : in std_logic_vector(15 downto
0);
               D3: in std_logic_vector(15 downto 0);
               immd: in std_logic_vector(9 downto 0); --- this
is immediate data MSB(9-1) merged with selector bit LSB(0)
               valid, C, Z, wb_control: in std_logic;
               clk: in std_logic;
```

```
               cond: in std_logic_vector(1 downto 0);
                  AD3: in std_logic_vector(2 downto 0);
               clear: in std_logic;
                        write_enable: in std_logic;
               valid_out, C_out, Z_out, wb_control_out: out
std_logic;
               cond_out: out std_logic_vector(1 downto 0);
                  AD3_out: out std_logic_vector(2 downto 0);
               D3_out: out std_logic_vector(15 downto 0);
               immd_out: out std_logic_vector(9 downto 0);
               pc_out, pc_2_out, inst_out : out
std_logic_vector(15 downto 0)
               );
      end component;
end package;
```

Feed-Forwarding



Necessity:-For dependent instructions we require Feed-Forwarding to give correct inputs to the various stages of the pipeline. Due to dependent instructions, the value of the inputs of the ALU/Register writeback/memory writeback may differ from that of the values stored in the corresponding Pipeline register.

Resolution:-We conditionally select the corresponding values stored in the corresponding and successive pipeline registers

RR_EX_D1/D2, EX-MEM_D3, MEM_WB_D3 using “Priority Mux” (giving priority to immediately dependent instructions) by checking the relative positions of the dependent instructions.

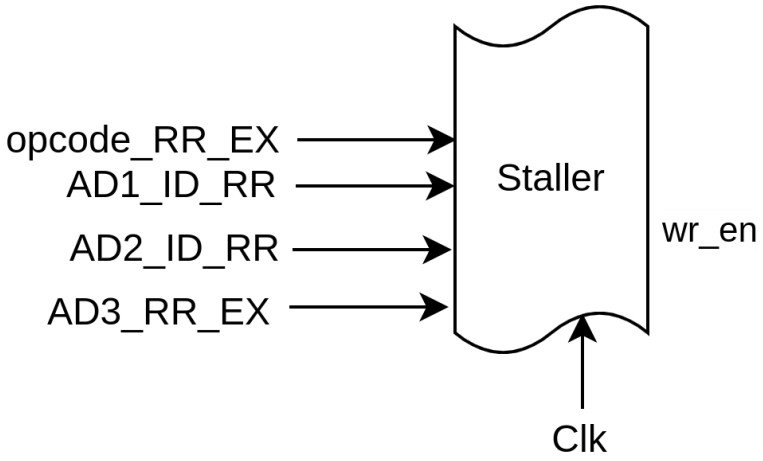
1) **Dependent I1, I2 Hazard:** We compare the RR_EX_AD1/2 (operand1/2 of I2) with EX_MEM_AD3 (destination of I1) and if they are equal then conditioned on the “validity”, i.e., if the instruction I1 is valid and writebacks to the register file (The instruction could be invalid if the flusher decides that it needs to be flushed). We provide the updated output stored in EX_MEM_D3 as input to the priority mux.
2) **Dependent I1, I3 Hazard:** We compare the RR_EX_AD1/2 (operand1/2 of I3) with MEM_WB_AD3 (destination of I1) conditioned on the “validity” of instruction I1.We provide the updated output stored in MEM_WB_D3 as input to the priority mux.
3) **Dependent I1, I4 Hazard:** We compare the ID_RR_AD1/2 (operand1/2 of I4) with MEM_WB_AD3 (destination of I1) conditioned on the “validity” of instruction I1.We provide the updated output stored in MEM_WB_D3 as input to mux.

Staller

It checks if the load instruction is immediately followed by a dependent instruction(the loaded register is the operand of the immediate instruction).If it is so it prevents/disables write-back to the IF_ID and ID_RR pipeline registers and also clears the RR_EX pipeline register(also making it’s valid bit=’0’) for 1 cycle, essentially stalling it for 1 cycle so that Feed-Forwarding can rectify the hazard in the upcoming cycle, i.e., essentially converting the given hazard into a I1,I3 type hazard and rectifying it by Feed-Forwarding.

```
entity staller is
     port(   opcode: in std_logic_vector(3 downto 0);
                AD1_ID_RR:    in std_logic_vector(2 downto 0);
                AD2_ID_RR:    in std_logic_vector(2 downto 0);
                AD3_RR_EX:    in std_logic_vector(2 downto 0);
                clk, reset:    in std_logic;
```

```
                wr_en:  out std_logic

        );

end entity;
```

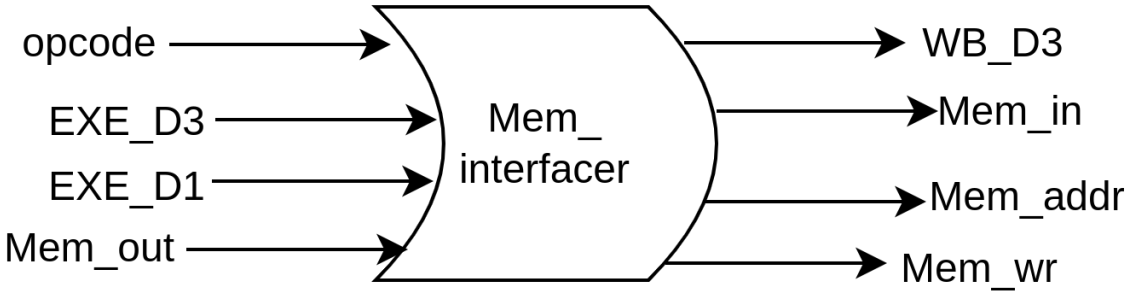


Memory Interfacer

This is used to define the interface with the memory as follows:-
LW: D3 of EXE goes to Mem_addr and Mem_out goes to WB_D3
SW: D3 of EXE goes to mem_addr and D1 of exe goes to mem in, turn mem write enable on.
Else: Forward exe_d3 to wb_d3.

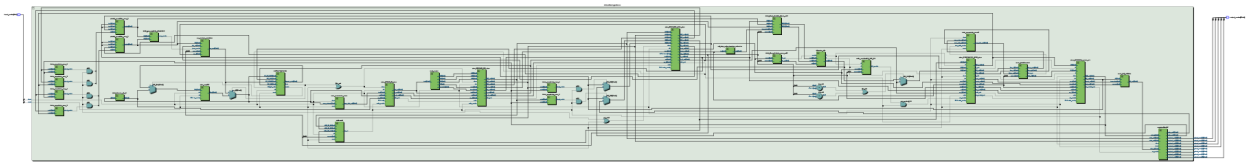
```
entity mem_interfacer is
     port(
          opcode: in std_logic_vector(3 downto 0);
          Exe_d3: in std_logic_vector(15 downto 0);
          Exe_d1: in std_logic_vector(15 downto 0);
                Mem_out: in std_logic_vector(15 downto 0);
          WB_d3: out std_logic_vector(15 downto 0);
          Mem_in: out std_logic_vector(15 downto 0);
          Mem_wr: out std_logic;
```

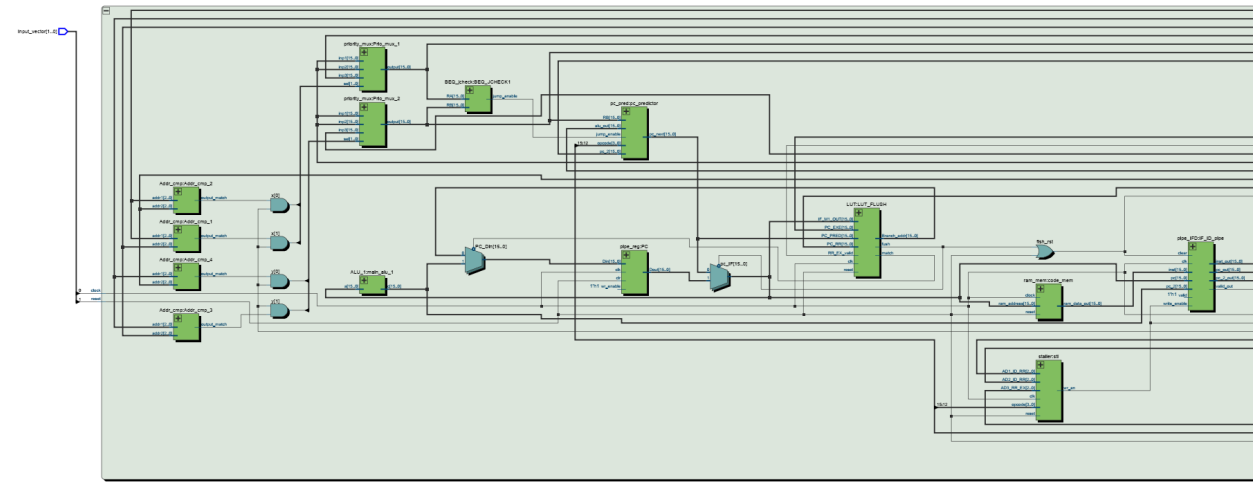
```
            Mem_addr: out std_logic_vector(15 downto 0)
            );
end entity;
```

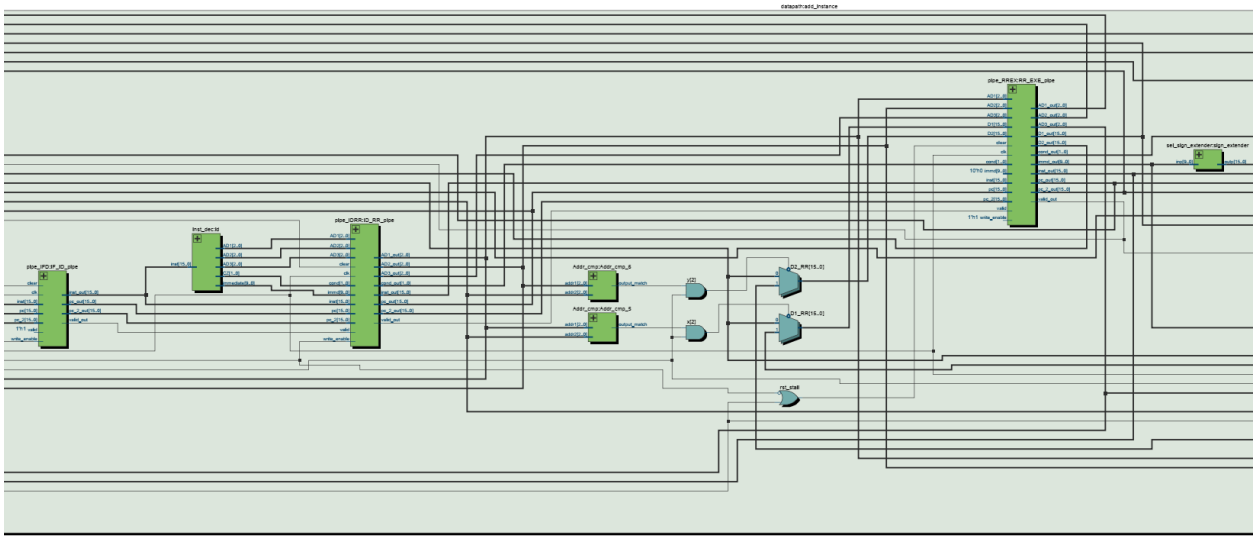


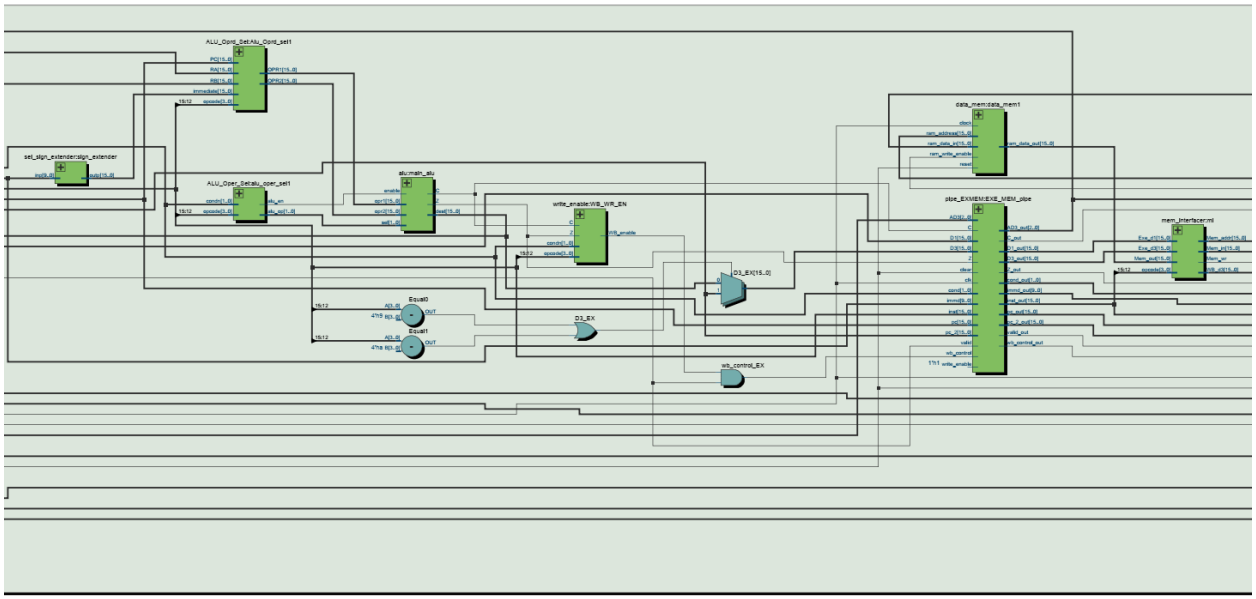
Simulation Results

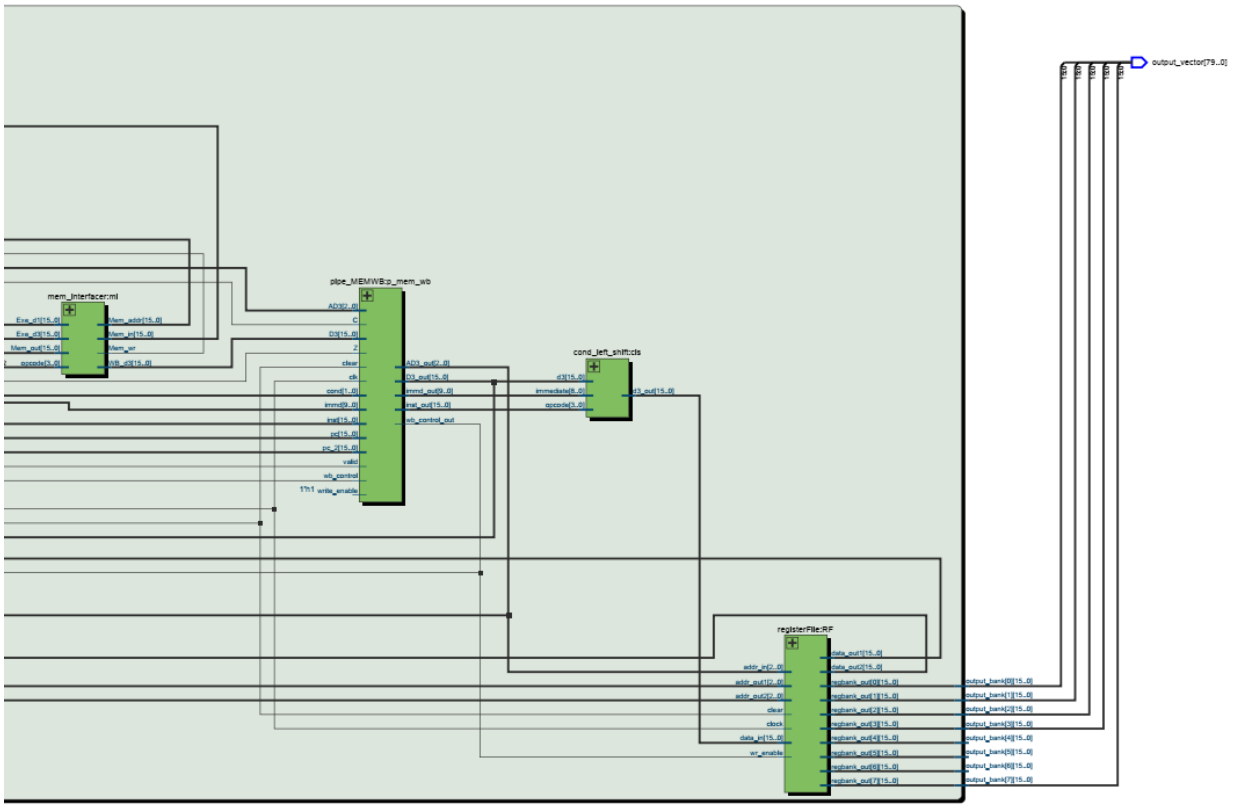
RTL View



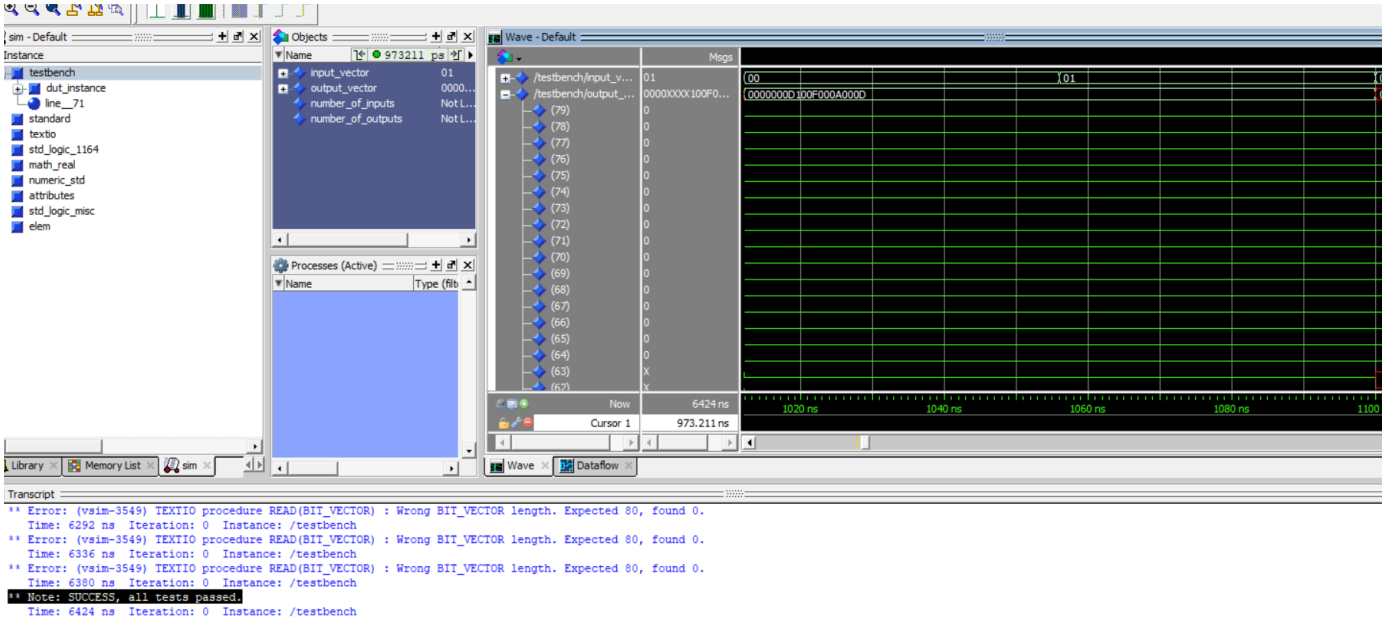








RTL Simulation



We have run different instructions and on running ADD with RA (R0) = 0003 and RB (R1) = 000A, we get RC as 000D which is stored in R3. Hence the working is verified.