Final project: Part III

Design of a (parameterizable) single-cycle datapath

Top entity: Parameterizable Single-Cycle Datapath

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
use work.DigEng.ALL;
-- Architecture B Parametrizable Single Cycle Data Path
-- Synchronous read Register Bank connected to an Asynchronous
-- Data Path.
-- The Data path contains 3 multiplexers, a single ALU and
-- a simulated ram Address output.
-- Multiplexer 1 controls the Input B of the ALU it either inputs
-- the immediate value or the value from Read B input from the register selected.
-- Multiplexer 2 controls what value is output to the RAM simulated address,
--it is either the output of the ALU or the input memory address.
--Multiplexer 3 controls the data that is input to the register bank input,
--either the output of the ALU is input or the value that is contained within
--address of the simulated RAM.
-- the simulated ram output is made up of a value to write to the ram
-- controlled by a tristate buffer an input for ram value to simulate the
-- value at that address and an output for the Address.
-- The ALU performs +1, -1, A+B, A-B, And, Or, Xor,Not operations,
--shift left & right, rotate left & right and the flags from
--the results are below
-- note only one of the above ALU operations is able to be performed per clk cycle
-- Flags(0): OUT = 0
-- Flags(1): OUT ? 0
-- Flags(2): OUT = 1
-- Flags(3): OUT < 0
-- Flags(4): OUT > 0
-- Flags(5): OUT ? 0
-- Flags(6): OUT ? 0
-- Flags(7): Overflow
entity Parameterizable_Single_Cycle_Datapath is
GENERIC ( data_size : natural := 16;
reg_size  : natural := 32);

Port ( REG_A           : in STD_LOGIC_VECTOR ((log2(reg_size))-1 downto 0);
REG_B           : in STD_LOGIC_VECTOR ((log2(reg_size))-1 downto 0);
WRITE_EN        : in STD_LOGIC;
-- ENABLES REGISTERS TO BE WRITTEN TO WHEN VALUE 1
CLK             : in STD_LOGIC;
RST             : in STD_LOGIC;
WRITE_REG_ADDR  : in STD_LOGIC_VECTOR ((log2(reg_size))- 1 downto 0);
-- REGISTER NUMBER TO BE WRITTEN TO
IMMED           : in STD_LOGIC_VECTOR ((data_size- 1) downto 0);
MEM_ADD         : in STD_LOGIC_VECTOR ((data_size- 1) downto 0);
-- MEMORY OF SIMULATED RAM TO BE CHOSEN
Sel             : in STD_LOGIC_VECTOR (2 downto 0);
-- BIT 0 CONTROLS mux 3, -- BIT 1 CONTROLS mux 2, -- BIT 2 CONTROLS mux 1
ALU             : in STD_LOGIC_VECTOR (3 downto 0);
SHIFT           : in STD_LOGIC_VECTOR ((log2(data_size))- 1 downto 0);
-- number of shifts/rotations the ALU should infer on the data
RAM_DATA        : in STD_LOGIC_VECTOR ((data_size - 1) downto 0);
-- data from address of the RAM simulation
OEN             : in STD_LOGIC;
-- output enable to write to Ram simulation values.
Flags           : out STD_LOGIC_VECTOR (7 downto 0);
MEM_DATA_WRITE  : out STD_LOGIC_VECTOR ((data_size - 1) downto 0);
--data to be written to the RAM simulation
```

```
MEM_DATA_ADD    : out STD_LOGIC_VECTOR ((data_size - 1) downto 0));
            -- address of the RAM simulation to be read or written to
end Parameterizable_Single_Cycle_Datapath;

architecture Behavioral of Parameterizable_Single_Cycle_Datapath is

signal REG_A_DATA_INT        : STD_LOGIC_VECTOR((data_size-1) downto 0);
signal REG_B_DATA_INT        : STD_LOGIC_VECTOR((data_size-1) downto 0);
signal ALU_SHIFT_OUT_INT     : STD_LOGIC_VECTOR((data_size-1) downto 0);
signal SHIFT_OUT_INT         : STD_LOGIC_VECTOR((data_size-1) downto 0);
signal MUX_1_OUT_INT         : STD_LOGIC_VECTOR((data_size-1) downto 0);
signal MUX_2_OUT_INT         : STD_LOGIC_VECTOR((data_size-1) downto 0);
signal MUX_3_OUT_INT         : STD_LOGIC_VECTOR((data_size-1) downto 0);
signal Flags_INT              : STD_LOGIC_VECTOR(7 downto 0);

begin

Reg_Bank: entity work.reg_bank
            Generic map( data_size => data_size,
                         reg_size  => reg_size)

              port map ( -- inputs
                         CLK        => clk ,
                         rst        => RST,
                         Write_en   => WRITE_EN,
                         Write_addr => WRITE_REG_ADDR,
                         Reg_A      => REG_A,
                         Reg_B      => REG_B,
                         DATA_IN    => MUX_3_OUT_INT,
                         -- outputs
                         DATA_OUT_1 => REG_A_DATA_INT,
                         DATA_OUT_2 => REG_B_DATA_INT);

ALU_Parameterizable: entity work.Parameterizable_ALU
            Generic map( data_size => data_size)

               Port map (  -- inputs
                           A       => REG_A_DATA_INT,
                           B       => MUX_1_OUT_INT,
                           X       => SHIFT,
                           OpCode  => ALU,
                           -- outputs
                           ALU_Out => ALU_SHIFT_OUT_INT,
                           Flags   => Flags_INT);

  --MULTIPLEXER 1
 MUX_1_OUT_INT  <= IMMED when Sel(2) = '1' else--
                   REG_B_DATA_INT when Sel(2) = '0' else--
                   (others => 'U');
  --MULTIPLEXER 2
 MEM_DATA_ADD  <=  MEM_ADD when Sel(1) = '1' else--
                   ALU_SHIFT_OUT_INT when Sel(1) = '0' else--
                   (others => 'U');
   --MULTIPLEXER 3
 MUX_3_OUT_INT  <= RAM_DATA when Sel(0) = '1' else--
                   ALU_SHIFT_OUT_INT when Sel(0) = '0' else--
                   (others => 'U');

   --TRI-STATE BUFFER OUTPUT ENABLE TO SIMULATED RAM
 MEM_DATA_WRITE <= REG_B_DATA_INT when (OEN = '1') else
                  (others => 'Z'); -- Z = high-impedance

 Flags <= Flags_INT;

end Behavioral;
```

Reg_Bank code found in Part 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.DigEng.ALL;

-- Parameterizable register bank
-- One Write per clock cycle, dual read
-- two generics for data size and number of registers
-- 2D Array of register outputs [reg_X][data_size]
--reg 0 is added to the start of reg array, reg 0 = phantom reg & value always = 0

entity reg_bank is
    GENERIC ( data_size : natural := 16;
              reg_size  : natural := 8);
    Port(
          -- inputs
          clk : in STD_LOGIC;
          rst : in STD_LOGIC;
          Write_en   : in STD_LOGIC;
          Write_addr : in STD_LOGIC_VECTOR (log2(reg_size)-1 downto 0);
          Reg_A : in STD_LOGIC_VECTOR (log2(reg_size)-1 downto 0);
          Reg_B : in STD_LOGIC_VECTOR (log2(reg_size)-1 downto 0);
          DATA_IN : in STD_LOGIC_VECTOR (data_size -1 downto 0);
          -- outputs
          DATA_OUT_1: out STD_LOGIC_VECTOR (data_size-1 downto 0);
          DATA_OUT_2: out STD_LOGIC_VECTOR (data_size-1 downto 0));
end reg_bank;
architecture Behavioral of reg_bank is
-- Internal signals
-- 2D array of register outputs
type reg_bank is array (reg_size-1  downto 0)
    of STD_LOGIC_VECTOR(data_size -1  downto 0);
signal int_reg_bank_out : reg_bank;  -- array
-- internal from decoder to registers
signal int_reg_addr     : STD_LOGIC_VECTOR ( reg_size -1 downto 0);
-- internal buffer B enable
signal int_buff_B_en    : STD_LOGIC_VECTOR( reg_size -1 downto 0);
-- internal buffer A enable
signal int_buff_A_en    : STD_LOGIC_VECTOR( reg_size -1 downto 0);
-- internal Data_in for register 0
signal int_data_in_reg0 : STD_LOGIC_VECTOR(data_size -1 downto  0);

begin
Reg0: entity work.reg_bits
            Generic map( data_size => data_size)
            port map ( clk => clk,
                       rst => rst,
                       en => int_reg_addr(0),
                       DATA_IN => int_data_in_reg0,
                       DATA_OUT => int_reg_bank_out(0));

int_reg_addr(0) <= '0';
DATA_OUT_1 <= int_reg_bank_out(0) when (int_buff_A_en(0) = '1')
                                  else (others => 'Z'); -- Z = high-impedance
DATA_OUT_2 <= int_reg_bank_out(0) when (int_buff_B_en(0) = '1')
                                  else (others => 'Z'); -- Z = high-impedance
```

```
-- starting from 1 as reg 0 (above) is a phantom register in this case
REG: for i in 1 to reg_size - 1 generate
   one_bit: entity work.reg_bits
            Generic map( data_size => data_size)
            port map ( clk => clk,
                       rst => rst,
                       en => int_reg_addr(i),
                       DATA_IN => DATA_IN,
                       DATA_OUT => int_reg_bank_out(i));

            -- buffer implementation
            DATA_OUT_1 <= int_reg_bank_out(i) when (int_buff_A_en(i) = '1')
                                              else (others => 'Z'); -- Z = high-
impedance
            DATA_OUT_2 <= int_reg_bank_out(i) when (int_buff_B_en(i) = '1')
                                              else (others => 'Z'); -- Z = high-
impedance

--Write address reg decoder
int_reg_addr(i) <= '1' when (i = unsigned(Write_addr) and Write_en = '1')
                   else 'Z';
    end generate;

--reg A/B address Decoder
DECODER: for i in 0 to reg_size - 1 generate
int_buff_A_en(i)<= '1' when (i = unsigned(Reg_A))
                       else 'Z';
int_buff_B_en(i)<= '1' when (i = unsigned(Reg_B))
                       else 'Z';

end generate;

end Behavioral;
```

ALU_Parameterizable code found in Part 2:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
use work.DigEng.ALL;

-- Asynchronous Parametrizable Arithmatic Logic Unit
-- allows for identity, bitwise, arithmaetic and shift logic
-- outputs results and flags for results output
-- each flag bit represents a different value event more detail below

entity Parameterizable_ALU is
generic (Data_Size : natural := 16);


    Port ( A : in STD_LOGIC_VECTOR ((Data_Size-1) downto 0);
           B : in STD_LOGIC_VECTOR ((Data_Size-1) downto 0);
           X : in STD_LOGIC_VECTOR (log2(Data_Size)-1 downto 0);
           OpCode : in STD_LOGIC_VECTOR (3 downto 0);
           ALU_Out : out STD_LOGIC_VECTOR (Data_Size-1 downto 0);
           Flags : out STD_LOGIC_VECTOR (7 downto 0));
end Parameterizable_ALU;


architecture Behavioral of Parameterizable_ALU is

signal MUX_Output : STD_LOGIC_VECTOR((Data_Size-1) downto 0);
signal X_Integer : integer ;

begin
X_Integer <= to_integer(unsigned(X));


            -- identity(Value in A unchanged to output)
 MUX_Output <= A when (OpCode  = "0000") else--
  (others => '0')when (OpCode  = "0001") else--
  (others => '0')when (OpCode  = "0010") else--
  (others => '0')when (OpCode  = "0011") else--

            -- bitwise logic
       (A and B) when(OpCode  = "0100") else--
        (A or B) when(OpCode  = "0101") else--
       (A xor B) when(OpCode  = "0110") else--
         (Not A) when(OpCode  = "0111") else-


 -- Arithmetic
 STD_LOGIC_VECTOR        (((signed(A)) + 1))            when(OpCode  = "1000") else--
 STD_LOGIC_VECTOR        (((signed(A)) - 1))            when(OpCode  = "1001") else--
 STD_LOGIC_VECTOR        (((signed(A)) + (signed(B))))  when(OpCode  = "1010") else--
 STD_LOGIC_VECTOR        (((signed(A)) - (signed(B))))  when(OpCode  = "1011") else-

 --shift
 STD_LOGIC_VECTOR(shift_left  (signed(A),   X_Integer)) when (OpCode = "1100") else--
 STD_LOGIC_VECTOR(shift_right (signed(A),   X_Integer)) when (OpCode = "1101") else--
 STD_LOGIC_VECTOR(rotate_left (unsigned(A), X_Integer)) when (OpCode = "1110") else--
 STD_LOGIC_VECTOR(rotate_right(unsigned(A), X_Integer)) when (OpCode = "1111") else--
                                                        (others => 'U');
```

```
   Flags(0) <= '1' when (signed(MUX_Output)  = 0) else '0'; -- equal to 0 check
    Flags(1) <= '1' when (signed(MUX_Output) /= 0) else '0'; -- not equal to 0 check
    Flags(2) <= '1' when (signed(MUX_Output)  = 1) else '0'; -- equal to 1 check
    Flags(3) <= '1' when (signed(MUX_Output)  < 0) else '0'; -- less than 0 check
    Flags(4) <= '1' when (signed(MUX_Output)  > 0) else '0'; -- greater than 0 check
    Flags(5) <= '1' when (signed(MUX_Output) <= 0) else '0'; -- less than or equal to 0 check
    Flags(6) <= '1' when (signed(MUX_Output) >= 0) else '0'; -- greater than or equal to 0 check

    Flags(7) <= --overflow conditions below
    -- checks when over flow can potentially occur via the opcode
    -- then checks the individual circumstances for each opcode that can potentially cause an overflow

    -- signed(A) + 1 overflows when A MSB = 0 and Output value MSB = 1
    '1' when (OpCode = "1000" and A(Data_Size-1)= '0' and MUX_Output(Data_Size-1)= '1' ) else
    -- signed(A) - 1 overflows when A MSB = 1 and Output value MSB = 0
    '1' when (OpCode = "1001" and A(Data_Size-1)= '1' and MUX_Output(Data_Size-1)= '0')  else

    -- signed(A) + signed(B) overflows when A MSB = 0 and when B MSB = 0 and Output value MSB = 1
    -- also over flows when
    -- signed(A) + signed(B) overflows when A MSB = 1 and when B MSB = 1 and Output value MSB = 0
    '1' when (OpCode = "1010" and A(Data_Size-1)= '0'
                              and B(Data_Size-1)= '0'
                              and MUX_Output(Data_Size-1)= '1') else

    '1' when (OpCode = "1010" and A(Data_Size-1)= '1'
                              and B(Data_Size-1)= '1'
                              and MUX_Output(Data_Size-1)= '0') else

    -- signed(A) - signed(B) overflows when A MSB = 1 and when B MSB = 0 and Output value MSB = 0
    -- also over flows when
    -- signed(A) - signed(B) overflows when A MSB = 0 and when B MSB = 1 and Output value MSB = 1
    '1' when (OpCode = "1011" and A(Data_Size-1)= '1'
                              and B(Data_Size-1)= '0'
                              and MUX_Output(Data_Size-1)= '0') else

    '1' when (OpCode = "1011" and A(Data_Size-1)= '0'
                              and B(Data_Size-1)= '1'
                              and MUX_Output(Data_Size-1)= '1') else
    '0';

 ALU_Out <= MUX_Output;

end Behavioral;
```

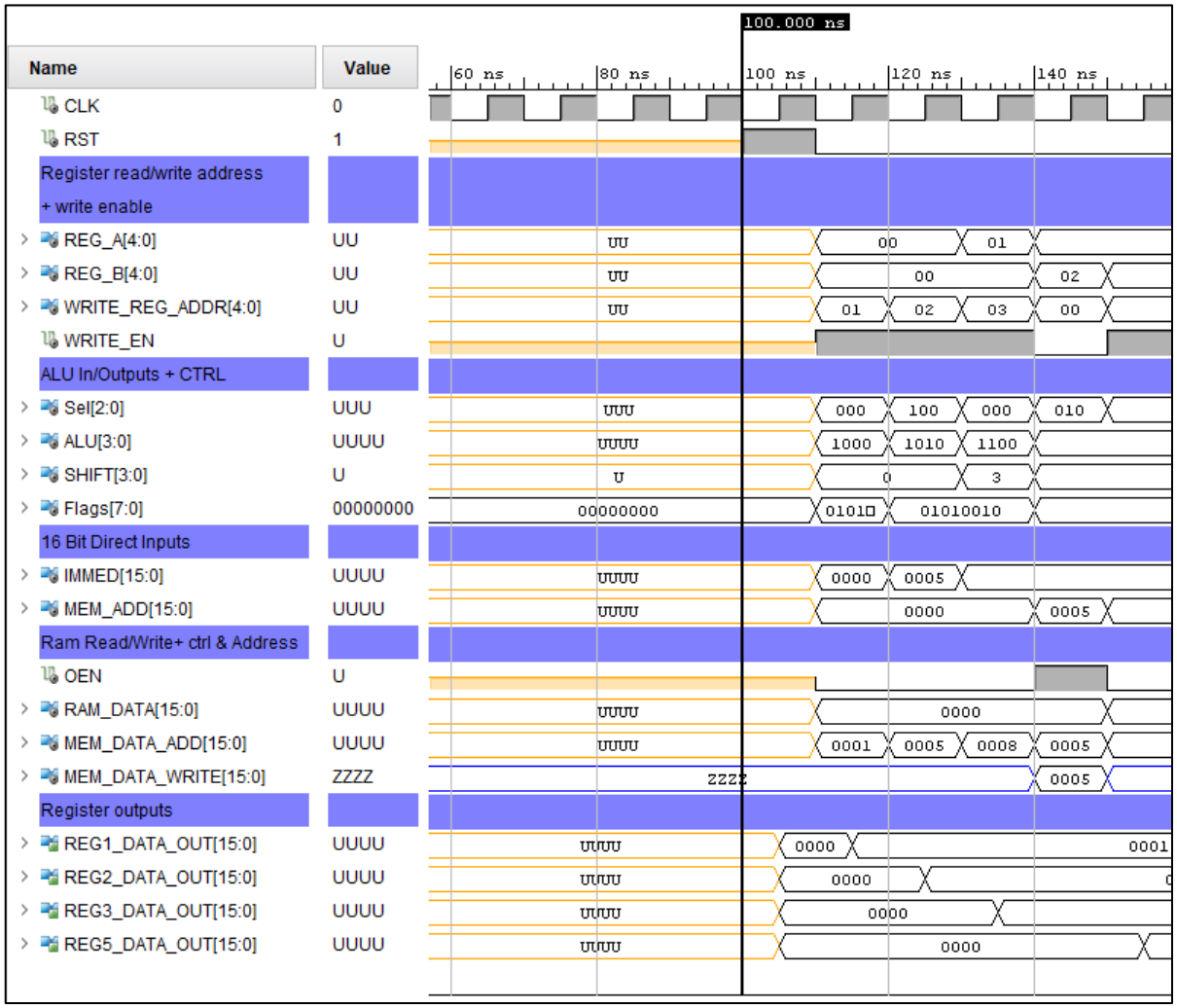
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.DigEng.ALL;

entity Parameterizable_Single_Cycle_Datapath_TB is

end Parameterizable_Single_Cycle_Datapath_TB;

architecture Behavioral of Parameterizable_Single_Cycle_Datapath_TB is

-- Constants
constant Data_size  : natural := 16;
constant reg_size   : natural := 32;
constant clk_period : time := 10ns;

-- Inputs
signal REG_A          : STD_LOGIC_VECTOR((log2(reg_size)) - 1 downto 0);
signal REG_B          : STD_LOGIC_VECTOR((log2(reg_size)) - 1 downto 0);
signal WRITE_EN       : STD_LOGIC;
signal CLK            : STD_LOGIC;
signal RST            : STD_LOGIC;
signal WRITE_REG_ADDR: STD_LOGIC_VECTOR ((log2(reg_size))- 1 downto 0);
signal IMMED          : STD_LOGIC_VECTOR(Data_size - 1 downto 0);
signal MEM_ADD        : STD_LOGIC_VECTOR(Data_size - 1 downto 0);
signal Sel            : STD_LOGIC_VECTOR(2 downto 0);
signal ALU            : STD_LOGIC_VECTOR(3 downto 0);
signal SHIFT          : STD_LOGIC_VECTOR ((log2(data_size))- 1 downto 0);
signal RAM_DATA       : STD_LOGIC_VECTOR(Data_size - 1 downto 0);
signal OEN            : STD_LOGIC;

-- Outputs
signal MEM_DATA_WRITE: STD_LOGIC_VECTOR(Data_size - 1 downto 0);
signal MEM_DATA_ADD  : STD_LOGIC_VECTOR ((data_size - 1) downto 0);
signal Flags          : STD_LOGIC_VECTOR (7 downto 0);

type test_vector is record
-- inputs
REG_A             : STD_LOGIC_VECTOR((log2(reg_size)) - 1 downto 0);
REG_B             : STD_LOGIC_VECTOR((log2(reg_size)) - 1 downto 0);
WRITE_EN          : STD_LOGIC;
WRITE_REG_ADDR    : STD_LOGIC_VECTOR ((log2(reg_size))- 1 downto 0);
IMMED             : STD_LOGIC_VECTOR(Data_size - 1 downto 0);
MEM_ADD           : STD_LOGIC_VECTOR(Data_size - 1 downto 0);
Sel               : STD_LOGIC_VECTOR(2 downto 0);
ALU               : STD_LOGIC_VECTOR(3 downto 0);
SHIFT             : STD_LOGIC_VECTOR ((log2(data_size))- 1 downto 0);
RAM_DATA          : STD_LOGIC_VECTOR(Data_size - 1 downto 0);
OEN               : STD_LOGIC;

-- Outputs
MEM_DATA_WRITE    : STD_LOGIC_VECTOR(Data_size - 1 downto 0);
MEM_DATA_ADD      : STD_LOGIC_VECTOR ((data_size - 1) downto 0);
Flags             : STD_LOGIC_VECTOR (7 downto 0);

end record;
```

```
type test_vector_array is array (natural range <>) of test_vector;
constant test_vectors : test_vector_array := (
    --REG_A,   REG_B, WRITE_EN,WRITE_REG_ADDR, IMMED,   MEM_ADD, Sel,  ALU,   SHIFT , RAM_DATA, OEN, Flags,     MEM_DATA_write,    MEM_DATA_ADD
    --inc R1, R0; [store the value 1 into register R1]
  ("00000",  "00000", '1',     "00001",        X"0000",X"0000", "000", "1000","0000", X"0000",  '0', "01010110","ZZZZZZZZZZZZZZZZ",X"0001"),

    -- addi R2, R0, 0005; [store value 5 into register R2]
  ("00000",  "00000", '1',     "00010",        X"0005",X"0000", "100", "1010","0000", X"0000",  '0', "01010010","ZZZZZZZZZZZZZZZZ",X"0005"),

    --shl R3, R1, 3; [store value 8 into register R3]
  ("00001",  "00000", '1',     "00011",        X"0000",X"0000", "000", "1100","0011", X"0000",  '0', "01010110", "ZZZZZZZZZZZZZZZZ",X"0001"),

    -- storr R2, R3; [store 5 into memory at address 8]
  ("00000",  "00010", '0',     "00000",        X"0000",X"0005",  "010", "0000","0000", X"0000",  '1', "01100001", X"0005"           ,X"0005"),


    --loadi R5, 1f1f; [load into R5 the contents of the memory at address 1F1]
  ("00000",  "00000", '1',     "00101",        X"0000",X"1F1F",  "011", "0000","0000", X"0007",  '0', "01100001", "ZZZZZZZZZZZZZZZZ",X"1F1F"));
```

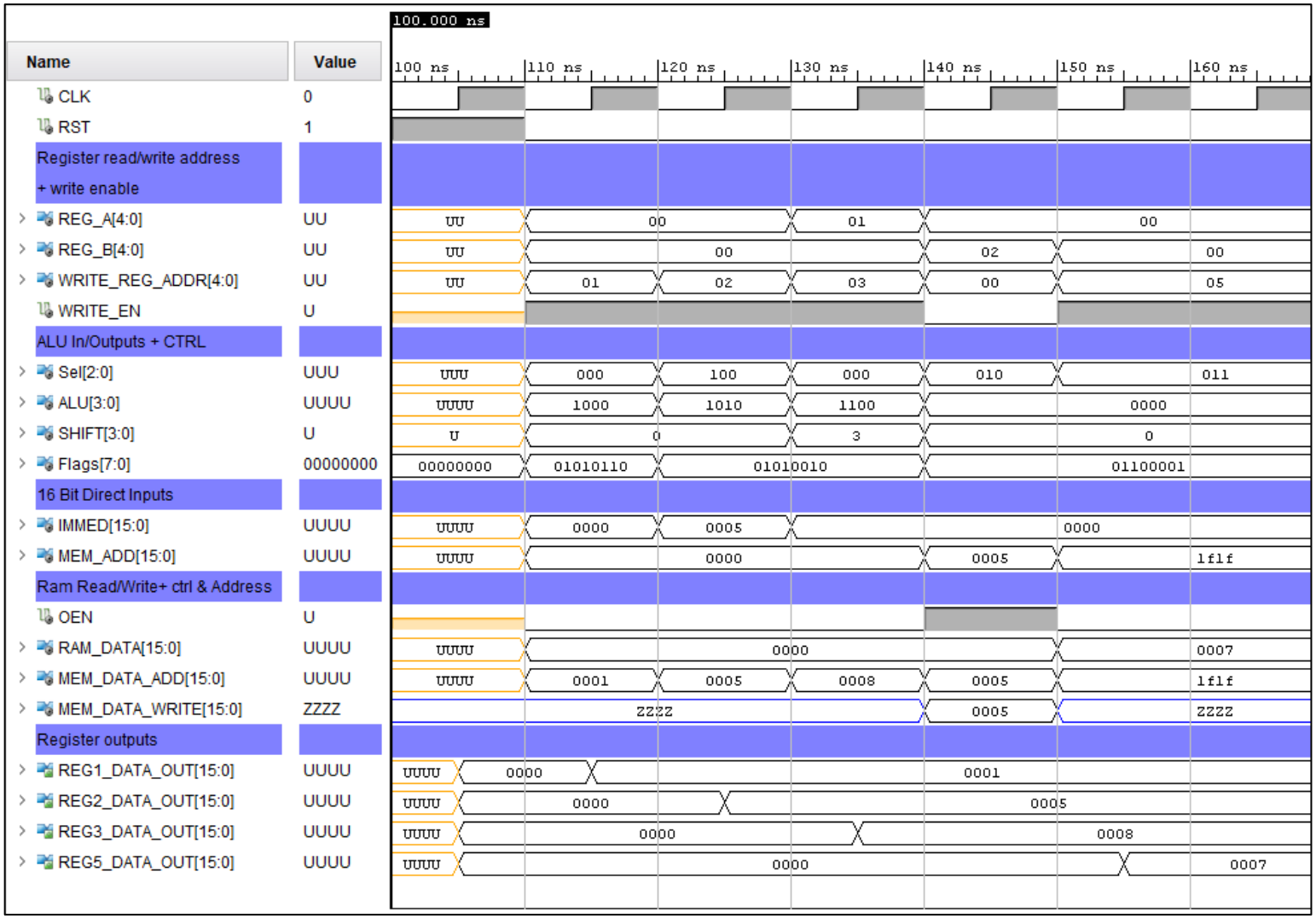
```
begin
UUT: entity work.Parameterizable_Single_Cycle_Datapath
-- map top level entity pins to testbench
GENERIC MAP( data_size => data_size,
             reg_size  => reg_size)

   PORT MAP( REG_A          => REG_A,
             REG_B          => REG_B,
             WRITE_EN       => WRITE_EN,
             CLK            => CLK,
             RST            => RST,
             WRITE_REG_ADDR => WRITE_REG_ADDR,
             IMMED          => IMMED,
             MEM_ADD        => MEM_ADD,
             Sel            => Sel,
             ALU            => ALU,
             SHIFT          => SHIFT,
             RAM_DATA       => RAM_DATA,
             OEN            => OEN,
             Flags          => Flags,
             MEM_DATA_WRITE => MEM_DATA_WRITE,
             MEM_DATA_ADD   => MEM_DATA_ADD);

 -- Clock process
 clk_process :process
 begin
 CLK <= '0';
 wait for clk_period/2;
 CLK <= '1';
 wait for clk_period/2;
 end process;

-- Test procedure
TEST: process
begin
wait for 100 ns; --wait for initialization
wait until falling_edge(CLK);-- input signals change at falling edge
-- reset registers
RST <= '1';
wait for 20ns;
RST <= '0';
 -- test all input vectors

for i in test_vectors'range loop

      REG_A          <= test_vectors(i).REG_A;
      REG_B          <= test_vectors(i).REG_B;
      WRITE_EN       <= test_vectors(i).WRITE_EN;
      WRITE_REG_ADDR <= test_vectors(i).WRITE_REG_ADDR;
      IMMED          <= test_vectors(i).IMMED;
      MEM_ADD        <= test_vectors(i).MEM_ADD;
      Sel            <= test_vectors(i).Sel;
      ALU            <= test_vectors(i).ALU ;
      SHIFT          <= test_vectors(i).SHIFT ;
      RAM_DATA       <= test_vectors(i).RAM_DATA;
      OEN            <= test_vectors(i).OEN;


      wait for 60 ns;
```

```
--Report error for every wrong test case of Flags, MEM_DATA_WRITE
-- and MEM_DATA_ADD.
assert  (
        (Flags = test_vectors(i).Flags)
        and
        (MEM_DATA_WRITE = test_vectors(i).MEM_DATA_WRITE)
        and
        (MEM_DATA_ADD = test_vectors(i).MEM_DATA_ADD)
        )
        -- report the values of all inputs and outputs in the case of wrong predicted
        -- values  with formatting.
report   CR &
"test_vectors value "    & integer'image(i) & " FAILED " & CR & CR &
"for inputs"             & CR &
"Reg A value: "          & integer  'image(to_integer(unsigned(REG_A))) & CR &
"Reg B value: "          & integer  'image(to_integer(unsigned(REG_B))) & CR &
"WRITE_EN : "            & std_logic'image(WRITE_EN) & CR &
"WRITE_REG_ADDR : "      & integer  'image(to_integer(signed(WRITE_REG_ADDR))) & CR &
"IMMED : "               & integer  'image(to_integer(unsigned(IMMED))) & CR &
"MEM_ADD : "             & integer  'image(to_integer(unsigned(MEM_ADD))) & CR &
"Sel: "                  & integer  'image(to_integer(unsigned(Sel))) & CR &
"ALU: "                  & integer  'image(to_integer(unsigned(ALU))) & CR &
"SHIFT: "                & integer  'image(to_integer(unsigned(SHIFT))) & CR &
"RAM_SIM: "              & integer  'image(to_integer(unsigned(RAM_SIM))) & CR &
"OEN: "                  & std_logic'image(OEN) & CR & CR &

"for outputs"            & CR &
"Flags: "                & integer  'image(to_integer(unsigned(Flags))) & CR &
"MEM_DATA_WRITE: "      & integer  'image(to_integer(unsigned(MEM_DATA_WRITE))) & CR &
"MEM_DATA_ADD : "        & integer  'image(to_integer(unsigned(MEM_DATA_ADD))) & CR


 severity error; -- only reports errors does NOT stop the program if errors detected

  end loop;

wait; -- waits forever

end process;

end Behavioral;
```

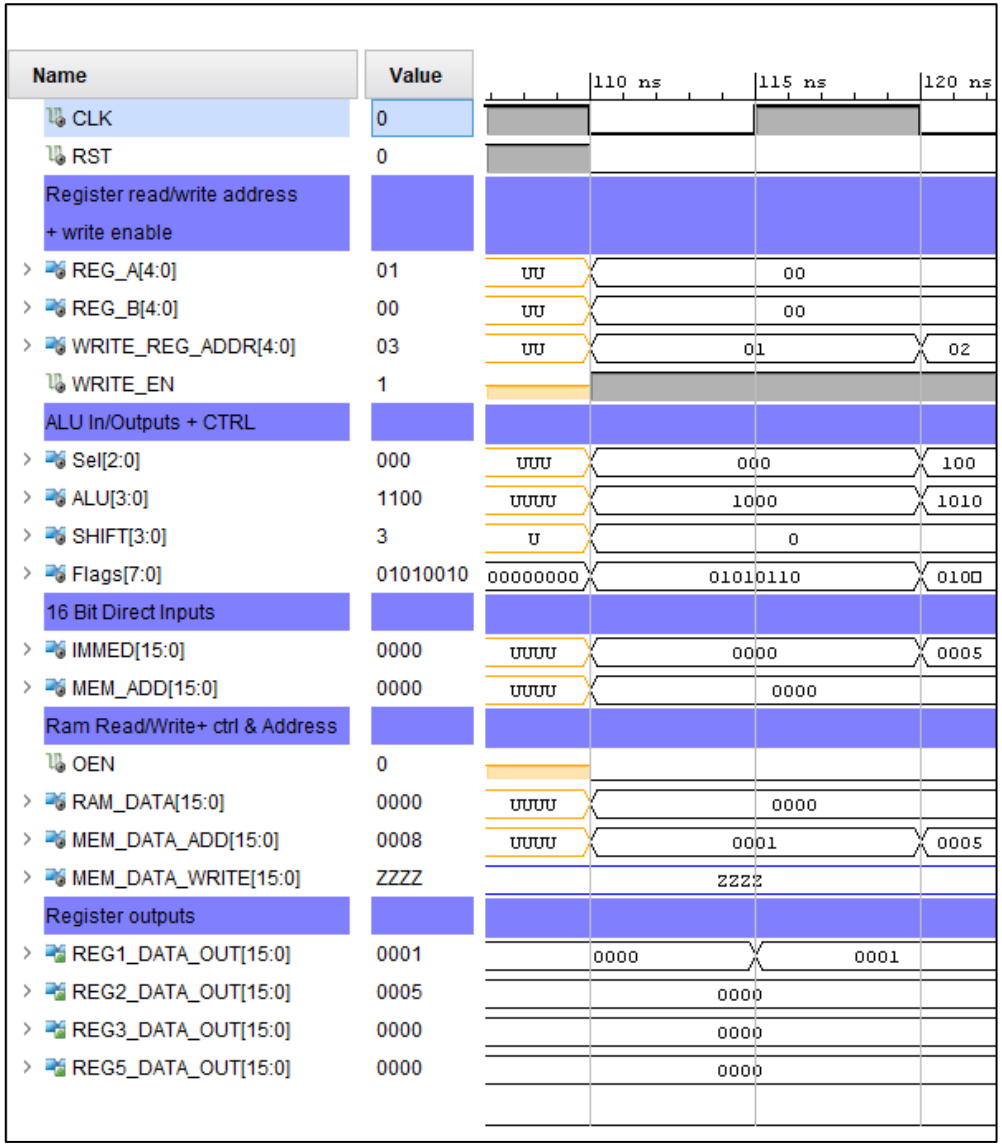
Register bank reset



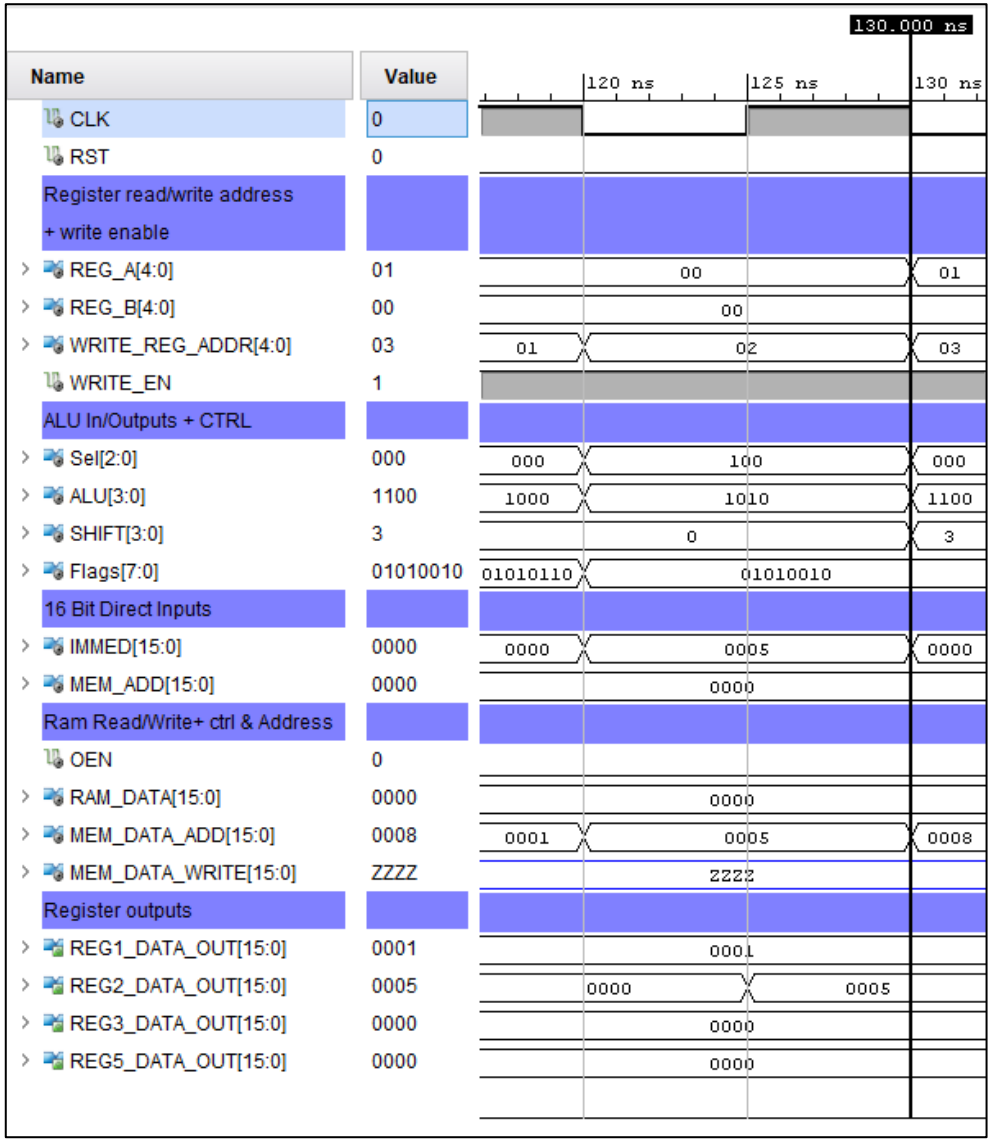
Results of all 5 instructions



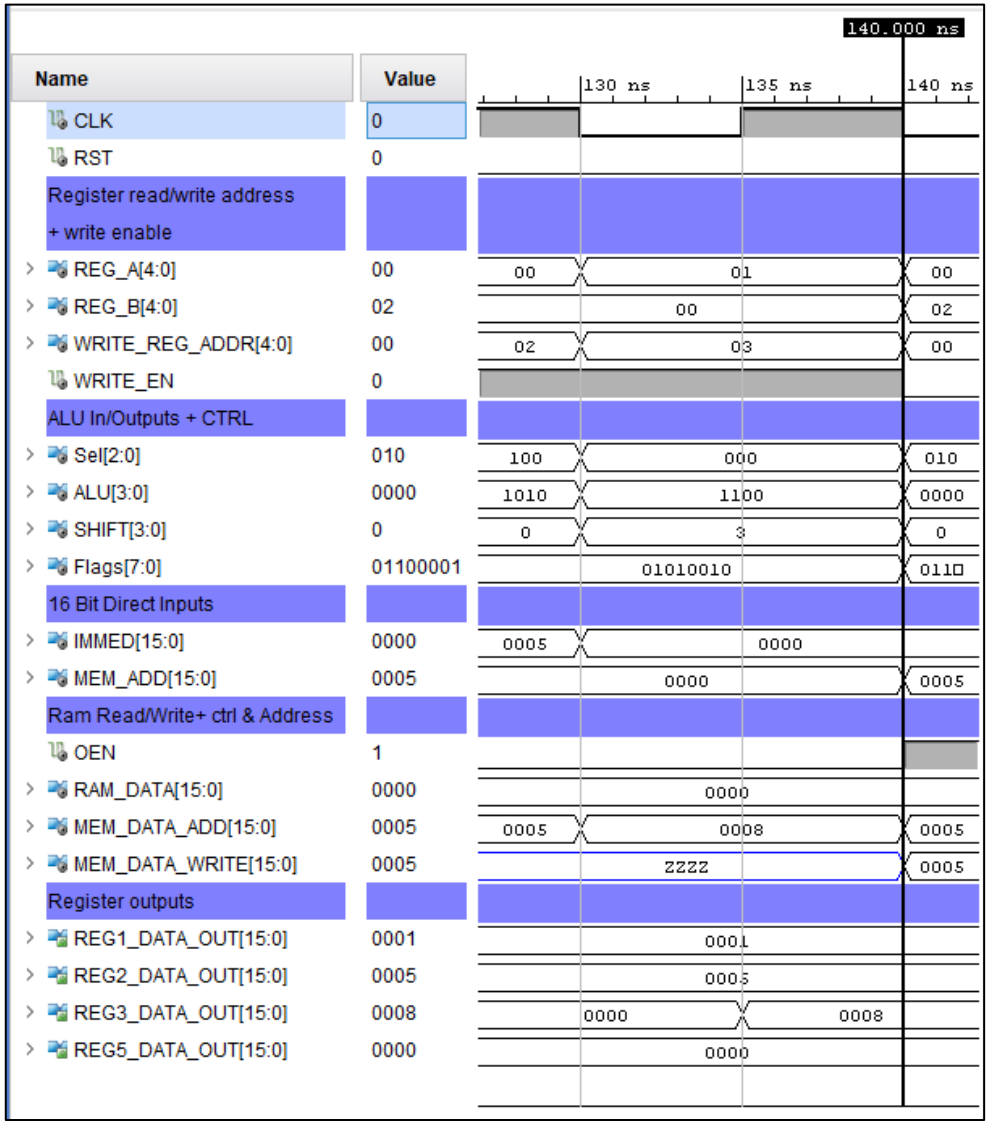
Instruction 1 inc R1, R0; [store the value 1 into register R1]



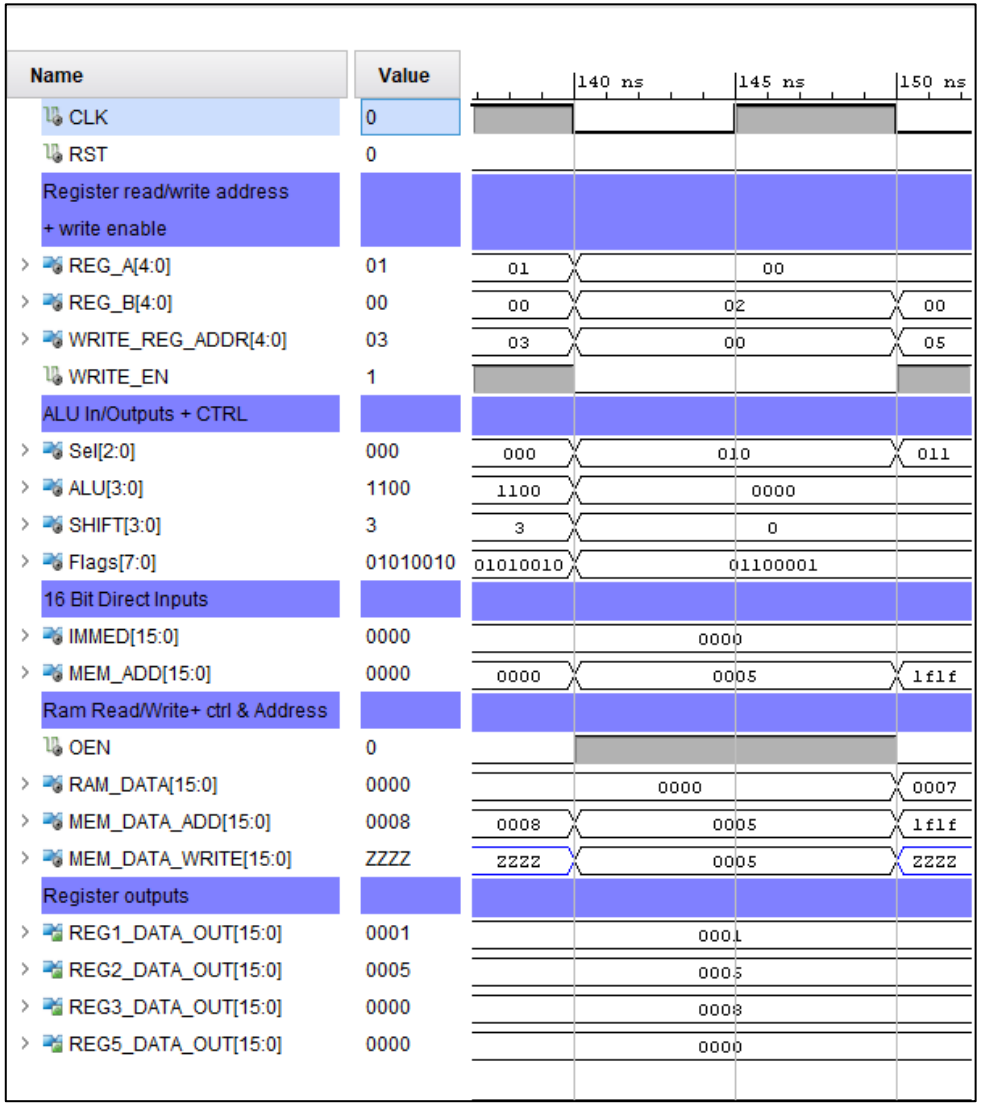
Instruction 2 addi R2, R0, 0005; [store value 5 into register R2]



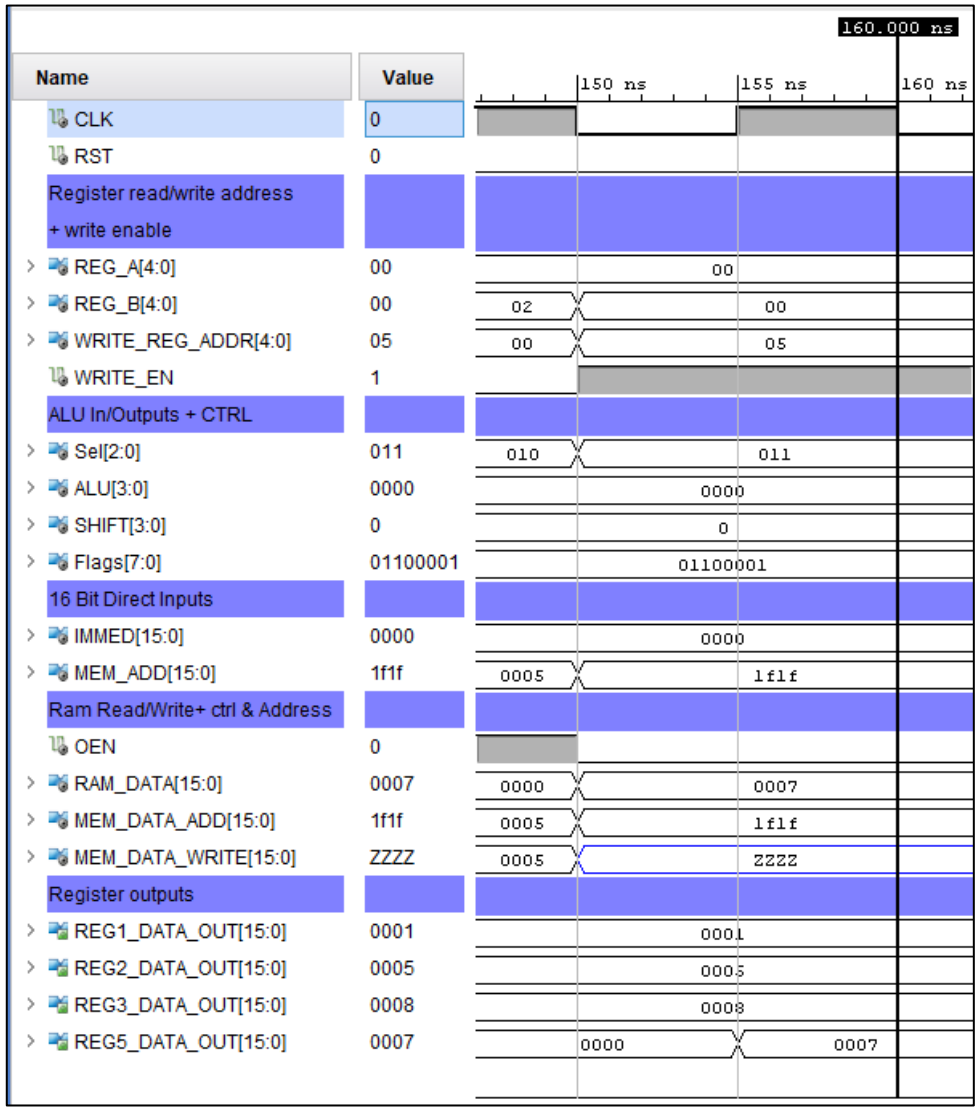
Instruction 3 shl R3, R1, 3; [store value 8 into register R3]



Instruction 4 storr R2, R3; [store 5 into memory at address 8]



Instruction 5 [load into R5 the contents of the memory at address 1F1F – assume that the value in question is 7 and provide it in your testbench]



```
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
         2 Input     16 Bit       Adders := 3
         3 Input     16 Bit       Adders := 1
+---XORs :
         2 Input     16 Bit         XORs := 1
+---Registers :
                     16 Bit    Registers := 32
+---Muxes :
         2 Input     16 Bit        Muxes := 3
         2 Input      1 Bit        Muxes := 66
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
```