COL216 Lab Assignment-2 (S1)

- Mayank Mangla (2020CS50430)

1 Introduction

The aim of this project is to design hardware for implementing a processor that can execute a subset of ARM instructions described below. Starting with a skeleton design, the hardware is built in several stages, adding some functionality at every stage. The designs are to be expressed in VHDL and then simulated and synthesized.

Stage 1: This stage is designing and testing the basic modules of the processor. The module set includes ALU, Register File, Program Memory and Data Memory.

The report for the stage 1 is given below:

2 Program Information

This program has been test on "eda playground" using "-2019 -o" flags.
The program is in four parts or modules: ALU (arithmetic and logic unit), Register File, Program Memory and Data Memory. Each module consists of 3 files. One for design, one for test bench and one for run.do.
All the files have some same part of the code:

```
library IEEE;
use IEEE.numeric_std.all;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

These are the header files that are used in the program. Other than this, each file has an entity declaration and its architecture implementation.
Run.do file has a common code that is the command to synthesis the module.

```
setup_design -manufacturer Xilinx -family Artix-7 -part 7A100TCSG324
foreach arg $::argv {
  add_input_file $arg
}
compile
synthesize
auto_write precision.v
report_output_file_list
report_area
report_timing
#exec cat precision.v
```

Next is the module wise understanding of the code:

2.1 ALU

This module does all the data processing instructions. This module responds to 16 different instructions.

```
entity ALU is
    port (op1,op2: in std_logic_vector(31 downto 0);    -- 2 operands
          opc:in std_logic_vector(3 downto 0);          -- operation code
          cin: in std_logic;                            -- carry in
          cout: out std_logic;                          -- carry out
          result: out std_logic_vector(31 downto 0));   -- final output
end ALU;
```

The inputs to this are:

1. two operand, each 32 bit unsigned integer
2. 4 bit instruction
3. carry in (single bit)

The outputs are:

1. carry out (single bit)
2. result of the instruction on the two input operands giving 32 bit unsigned integer

The different instructions have different 4 bit standard op code that is followed by the arm instruction set. Accordingly we perform the operation on the operands.

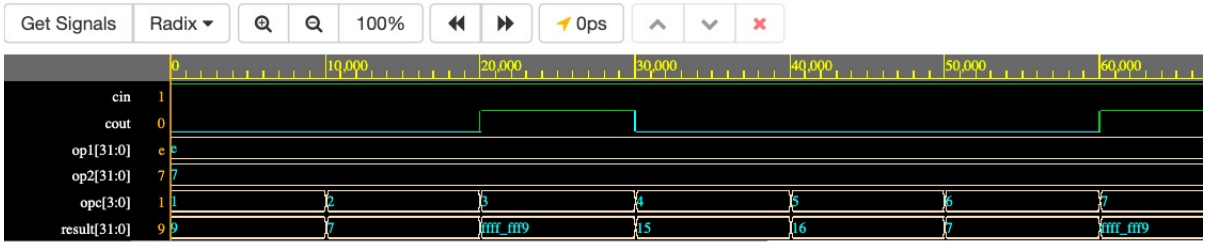
```
architecture implement_alu of ALU is
begin
    process(op1,op2,cin,opc)                              -- triggered by the change in inputs
    variable temp: std_logic_vector(32 downto 0);         -- temporary variables
    variable tempbit: std_logic := '0';                   -- defined for sequential statements
    begin
        case(opc) is                                      -- case switches for operations
            when "0000" => temp := (tempbit & op1) and (tempbit & op2);
            when "0001" => temp := (tempbit & op1) xor (tempbit & op2);
            when "0010" => temp := (tempbit & op1) + (not (tempbit & op2)) + 1;
            when "0011" => temp := (tempbit & op2) + (not (tempbit & op1)) + 1;
            when "0100" => temp := (tempbit & op1) + (tempbit & op2);
            when "0101" => temp := (tempbit & op1) + (tempbit & op2) + cin;
            when "0110" => temp := (tempbit & op1) + (not (tempbit & op2)) + cin;
            when "0111" => temp := (tempbit & op2) + (not (tempbit & op1)) + cin;
            when "1000" => temp := (tempbit & op1) and (tempbit & op2);
            when "1001" => temp := (tempbit & op1) xor (tempbit & op2);
            when "1010" => temp := (tempbit & op1) + (not (tempbit & op2)) + 1;
            when "1011" => temp := (tempbit & op1) + (tempbit & op2);
            when "1100" => temp := (tempbit & op1) or (tempbit & op2);
            when "1101" => temp := tempbit & op2;
            when "1110" => temp := (tempbit & op1) and (not (tempbit & op2));
            when "1111" => temp := (not (tempbit & op2));
            when others => temp := "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        end case;
        result <= temp(31 downto 0);                          -- updating the result
        cout <= temp(32);                                     -- and carry out
    end process;                                              -- ending process
end implement_alu;                                            -- ending implementation
```

Now the test bench for the ALU module test the design for different inputs which are hard coded in the test bench code. Its code can be displayed as (excluding something that is very trivial)

```
process
    variable v: std_logic_vector(3 downto 0);
begin -- input 1
a_in <= "00000000000000000000000000001110";
b_in <= "00000000000000000000000000000111";
c1_in <= '1';   opc_in <= "1111";
-- loop checking for all opcodes
for I in 0 to 15 loop
    v := opc_in;
    opc_in <= v + "0001";
    wait for 10 ns;
end loop;
wait for 10 ns;
```

```
-- input 2
a_in <= "11111111111111111111111111111111";
b_in <= "00000000000000000000000000000000";
c1_in <= '1';   opc_in <= "1111";
-- loop checking for all opcodes
for I in 0 to 15 loop
    v := opc_in;
    opc_in <= v + "0001";
    wait for 10 ns;
end loop;
wait for 10 ns;
-- input 3
a_in <= "10101010101010101010101010101010";
b_in <= "11111111111111110000000000000000";
c1_in <= '1';   opc_in <= "1111";
-- loop checking for all opcodes
for I in 0 to 15 loop
    v := opc_in;
    opc_in <= v + "0001";
    wait for 10 ns;
end loop;
wait for 10 ns;
wait;
end process;
```

The output signal to these test case are:

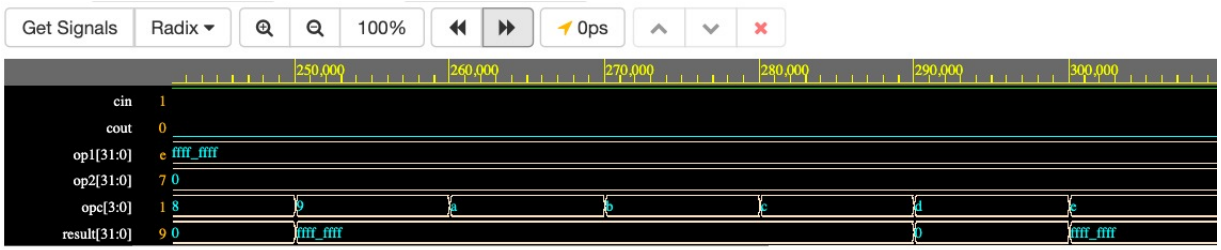


Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Note: To revert to EPWave opening in a new browser window, set that option on your user page.

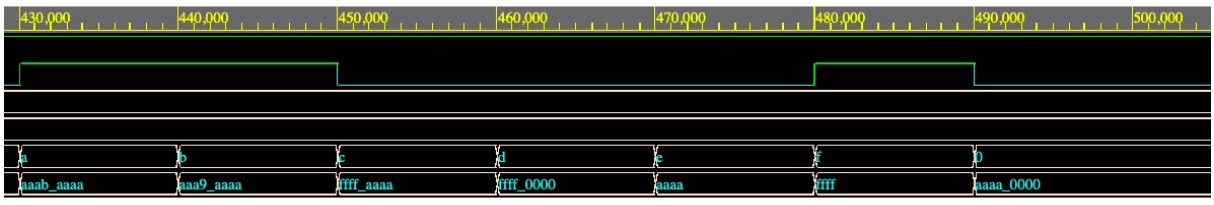
Note: To revert to EPWave opening in a new browser window, set that option on your user page.



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Note: To revert to EPWave opening in a new browser window, set that option on your user page.



The logs of the synthesis of ALU module are provided here:

```
# Info: ************************************************************
# Info: Device Utilization for 7A100TCSG324
# Info: ************************************************************
# Info: Resource                          Used    Avail   Utilization
# Info: -------------------------------------------------------------
# Info: IOs                               102     210      48.57%
# Info: Global Buffers                    0       32        0.00%
# Info: LUTs                              133     63400     0.21%
```

```
# Info: CLB Slices                          32      15850     0.20%
# Info: Dffs or Latches                     0       126800    0.00%
# Info: Block RAMs                          0       135       0.00%
# Info: DSP48E1s                            0       240       0.00%
# Info: ------------------------------------------------------------
# Info: ************************************************************
# Info: Library: work    Cell: ALU    View: implement_alu
# Info: ************************************************************
# Info:  Number of ports :                            102
# Info:  Number of nets :                             369
# Info:  Number of instances :                        300
# Info:  Number of references to this view :            0
# Info: Total accumulated area :
# Info:  Number of LUTs :                             133
# Info:  Number of LUTs with LUTNM/HLUTNM :             4
# Info:  Number of MUX CARRYs :                        32
# Info:  Number of accumulated instances :            300
# Info: ************************************************************
```

2.2 Register File

This module contains all the registers' addresses. This module enables us to read and write data from or to registers.

```
entity RegisterFile is
    port (rad1, rad2, wad: in std_logic_vector(3 downto 0);     -- read and write addresses
        data: in std_logic_vector(31 downto 0);                 -- data that is to be written
        wen,clk: in std_logic;                                  -- write enable and clock
        dout1,dout2: out std_logic_vector(31 downto 0));        -- data that is being read
end RegisterFile;
```

The inputs to this are:

1. two addresses of the register of which the data is to be read, each 4 bit unsigned integer
2. one address of the register which is to be overwritten, each 4 bit unsigned integer
3. clock and write enable (single bit)

The outputs are:

1. two data vectors that have been read from the two addresses (32 bit each)

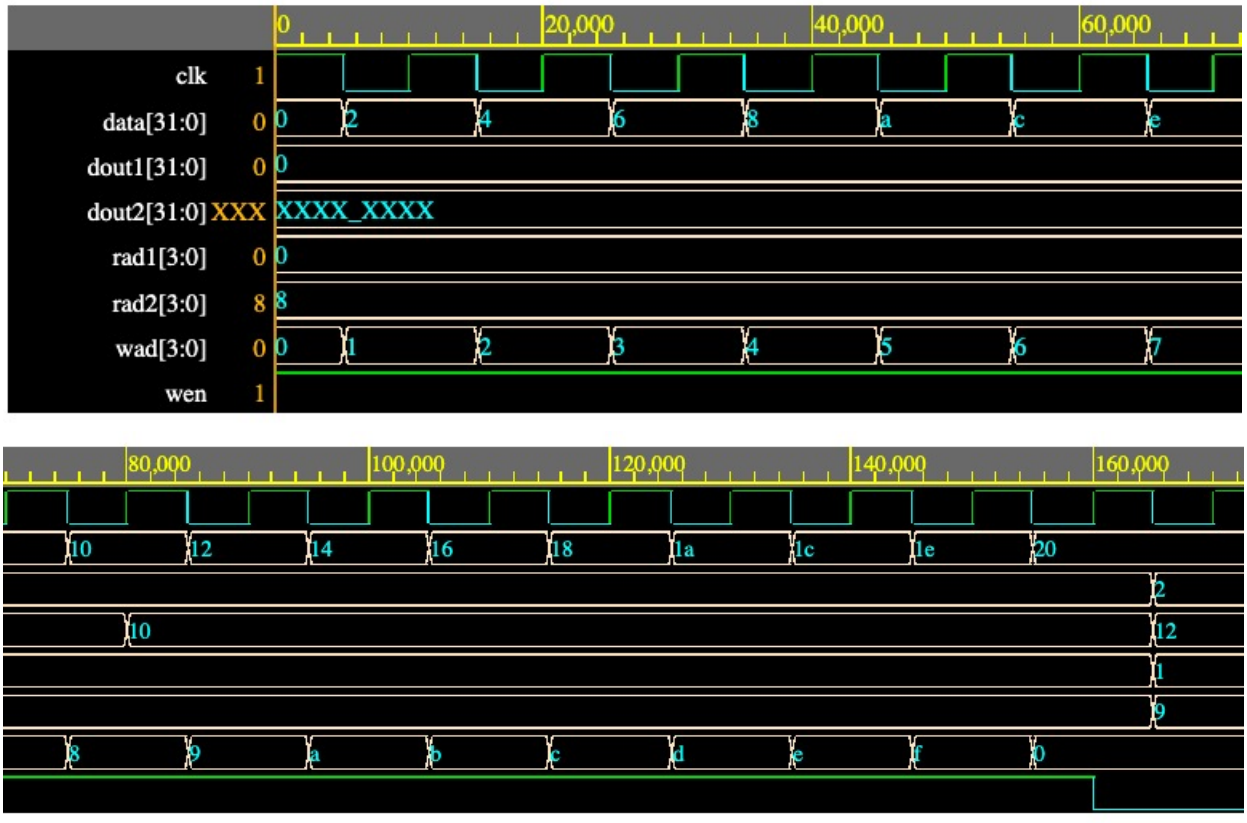
Here we have a memory data type that is an array of length 16 each element is of 32 bit. It stores the registers data.Then we read the data ignoring the condition of the clock. Then on the rising edge of the clock we write the data on the memory

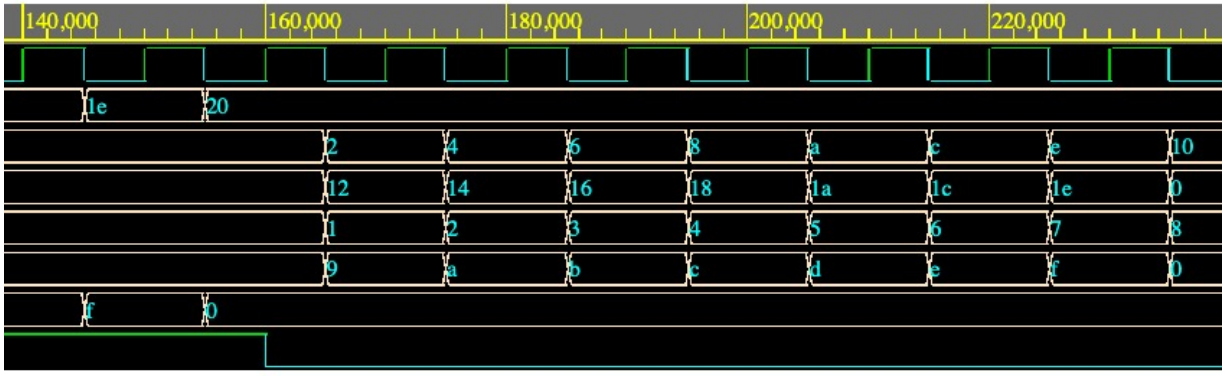
```
architecture implement_rf of RegisterFile is
    type mem is array (0 to 15) of std_logic_vector(31 downto 0);
    -- defining a type that is array of 16 std_logic_vector (32 bits)
    signal memory: mem;                                 -- internal signal that is of mem type
begin
    dout1 <= memory(conv_integer(rad1));                -- reading the register and outputing the same
    dout2 <= memory(conv_integer(rad2));                -- without any intervention of clock
    process(clk)                                        -- process when clock or writing is triggered
    begin
        if (rising_edge(clk)) then                      -- at rising edge of the clock
            if wen = '1' then                           -- and write enable set
                memory(conv_integer(wad)) <= data;      -- write the data to the register address
            end if;
        end if;
    end process;                                        -- end process
end implement_rf;                                       -- end implementation
```

Now the test bench for the Register File module test the design for different inputs which are hard coded in the test bench code. First we write some values in the register and then read them. Its code can be displayed as (excluding something that is very trivial)

```
process
begin
    clock <= '1';       r1 <= "0000";       r2 <= "1000";
    w <= "0000";    d <= "00000000000000000000000000000000";
    we <= '1';      -- write in memory
    for a in 0 to 15 loop
        clock <= not clock;     wait for 5 ns;
        w <= w + 1;             d <= d + 2;
        clock <= not clock;     wait for 5 ns; -- rising edge of the clock
    end loop;
    we <= '0';
    -- read from memory
    for b in 0 to 7 loop
        clock <= not clock;     wait for 5 ns;
        r1 <= r1 + 1;           r2 <= r2 + 1;
        clock <= not clock;     wait for 5 ns; -- rising edge of the clock
    end loop;
    wait;
end process;
```

The output signal to these test case are:





The logs of the synthesis of Register File module are provided here:

```
# Info: ********************************************************************
# Info: Device Utilization for 7A100TCSG324
# Info: ********************************************************************
# Info: Resource                          Used    Avail   Utilization
# Info: --------------------------------------------------------------------
# Info: IOs                               110     210      52.38%
# Info: Global Buffers                    1       32        3.12%
# Info: LUTs                              48      63400     0.08%
# Info: CLB Slices                        12      15850     0.08%
# Info: Dffs or Latches                   0       126800    0.00%
# Info: Block RAMs                        0       135       0.00%
# Info: Distributed RAMs
# Info:    RAM32M                         10
# Info:    RAM64M                         2
# Info: DSP48E1s                          0       240       0.00%
# Info: --------------------------------------------------------------------
# Info: ********************************************************************
# Info: Library: work    Cell: RegisterFile    View: implement_rf
# Info: ********************************************************************
# Info:  Number of ports :                          110
# Info:  Number of nets :                           220
# Info:  Number of instances :                      111
# Info:  Number of references to this view :          0
# Info: Total accumulated area :
# Info:  Number of LUTs :                            48
# Info:  Number of accumulated instances :          123
# Info: ********************************************************************
```

2.3 Program Memory

This module contains all the instructions stored in the form of 32 bit number each. We read the data from the this module and perform the instruction that is read. This is Read Only Memory and hence there is no port for writing into this memory.

```
entity ProgramMemory is
    port(radd: in std_logic_vector(5 downto 0);             -- reading address
        dout: out std_logic_vector(31 downto 0));           -- the data that is read
end ProgramMemory;
```

The inputs to this are:

1. one address of the register of which the data is to be read (4 bit unsigned integer)

The outputs are:

1. one data vector that has been read from the addresses (32 bit vector)

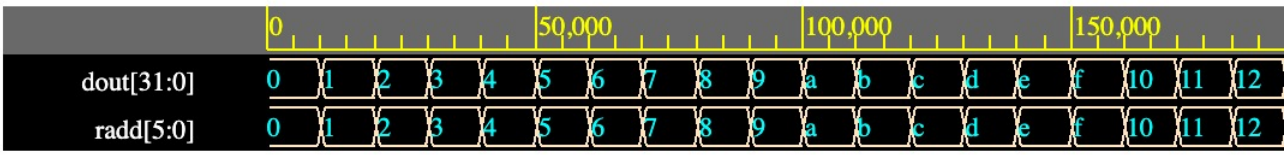
Here we have a memory data type that is an array of length 64 each element is of 32 bits *(This memory s hard coded in the code, can be modified according to the testing by changing its initialization in the code).*

```
architecture implement_pm of ProgramMemory is
    type mem is array (0 to 63) of std_logic_vector(31 downto 0);
    -- mem is array of instruction in program memory
    signal memory: mem:=
    ("00000000000000000000000000000000","00000000000000000000000000000001",
    "00000000000000000000000000000010","00000000000000000000000000000011",
    "00000000000000000000000000000100","00000000000000000000000000000101",
    "00000000000000000000000000000110","00000000000000000000000000000111",
    "00000000000000000000000000001000","00000000000000000000000000001001",
    ...);                                          -- signal that is accessing the instruction array
begin
    process(radd)                                  -- process on read address
    begin
        dout <= memory(conv_integer(radd));        -- getting the data at the given address
    end process;
end implement_pm;
```

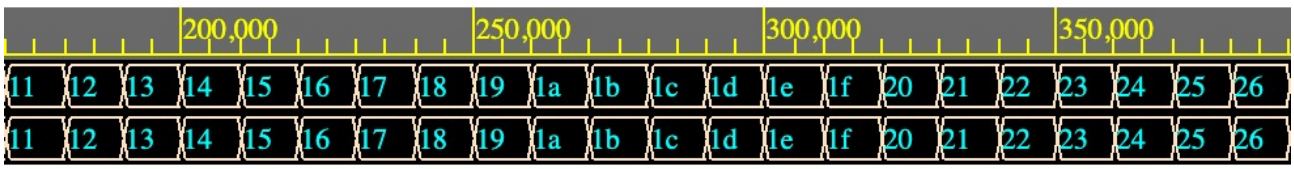
Now the test bench for the Program Memory module test the design for different inputs which are hard coded in the test bench code. We read the instruction codes that have been coded in the program. Its code can be displayed as (excluding something that is very trivial)

```
process
begin
    -- reading the data initialized
    readaddress <= "000000";
    for a in 0 to 63 loop
        wait for 10 ns;
        readaddress <= readaddress + "000001";
    end loop;
    wait;
end process;
```

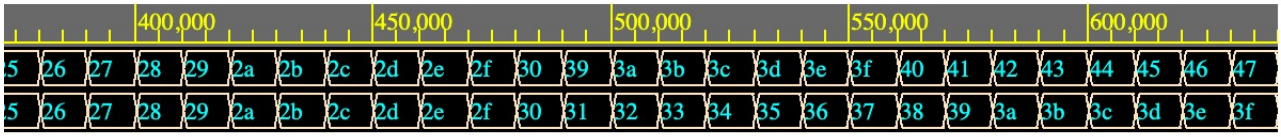
The output signal to these test case are:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.



age.



The logs of the synthesis of Program memory module are provided here:

```
# Info: ************************************************************************
# Info: Device Utilization for 7A100TCSG324
# Info: ************************************************************************
# Info: Resource                          Used    Avail   Utilization
# Info: ------------------------------------------------------------------------
# Info: IOs                               38      210      18.10%
# Info: Global Buffers                    0       32        0.00%
# Info: LUTs                              4       63400     0.01%
# Info: CLB Slices                        0       15850     0.00%
# Info: Dffs or Latches                   0       126800    0.00%
# Info: Block RAMs                        0       135       0.00%
# Info: DSP48E1s                          0       240       0.00%
# Info: ------------------------------------------------------------------------
# Info: ************************************************************************
# Info: Library: work    Cell: ProgramMemory    View: implement_pm
# Info: ************************************************************************
# Info:  Number of ports :                            38
# Info:  Number of nets :                             49
# Info:  Number of instances :                        43
# Info:  Number of references to this view :           0
# Info: Total accumulated area :
# Info:  Number of LUTs :                              4
# Info:  Number of LUTs with LUTNM/HLUTNM :            2
# Info:  Number of accumulated instances :            43
# Info: ************************************************************************
```

2.4 Data Memory

This module stores the data. This is different from program memory in a way that the data in data memory can be overwritten. This has one read and one write port.

```
entity DataMemory is
    port(add: in std_logic_vector(5 downto 0);              -- read and write addresses
        clk: in std_logic;                                  -- clock
        data: in std_logic_vector(31 downto 0);             -- input data
        dout: out std_logic_vector(31 downto 0);            -- output data
        wen: in std_logic_vector(3 downto 0));              -- write enable
end DataMemory;
```

The inputs to this are:

1. data that is to be written, 32 bit vector
2. address where the data is to be written and read, 6 bit unsigned number
3. clock (single bit)
4. a 4 bit write enable

The outputs are:

1. data that is being read, 32 bit vector

The different enables have different 4 bit code which is used to write in the memory to any one byte, half word or full word. Accordingly we write to the memory.

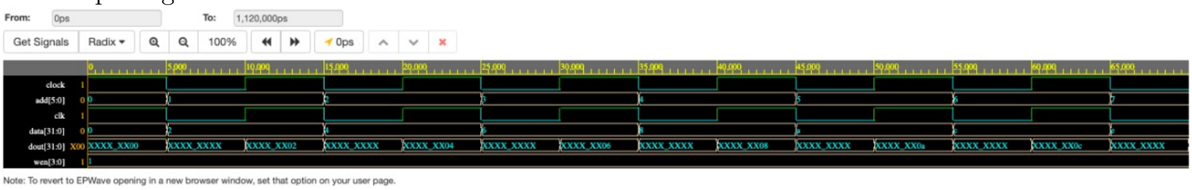
```
architecture implement_dm of DataMemory is
    type mem is array (0 to 63) of std_logic_vector(31 downto 0); -- mem array of data
    signal memory: mem;                                          -- signal for the data in data memory
begin
    process(clk)                                                 -- process with change in clock
    begin
        if(rising_edge(clk)) then
        -- if the rising edge of the clock then we do as write enables
            if wen= "0001" then memory(conv_integer(add))(7 downto 0) <= data(7 downto 0);
            elsif wen="0010" then memory(conv_integer(add))(15 downto 8) <= data(7 downto 0);
            elsif wen="0100" then memory(conv_integer(add))(23 downto 16) <= data(7 downto 0);
            elsif wen="1000" then memory(conv_integer(add))(31 downto 24) <= data(7 downto 0);
            elsif wen = "0011" then memory(conv_integer(add))(15 downto 0) <= data(15 downto 0);
            elsif wen = "1100" then memory(conv_integer(add))(31 downto 16) <= data(15 downto 0);
            elsif wen = "1111" then memory(conv_integer(add)) <= data ;
            end if;
        end if;
    end process;
    dout <= memory(conv_integer(add));                          -- read the data at given index
end implement_dm;
```

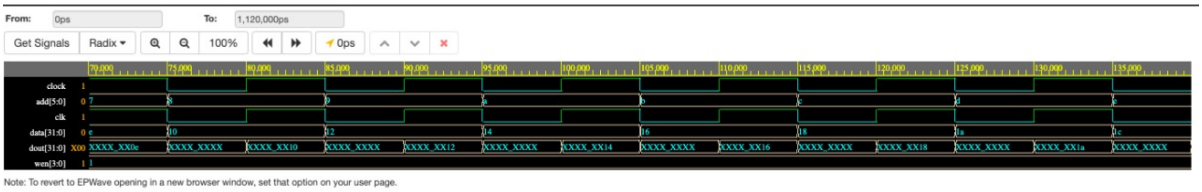
Now the test bench for the Data Memory module test the design for different inputs which are hard coded in the test bench code. Its code can be displayed as (excluding something that is very trivial)

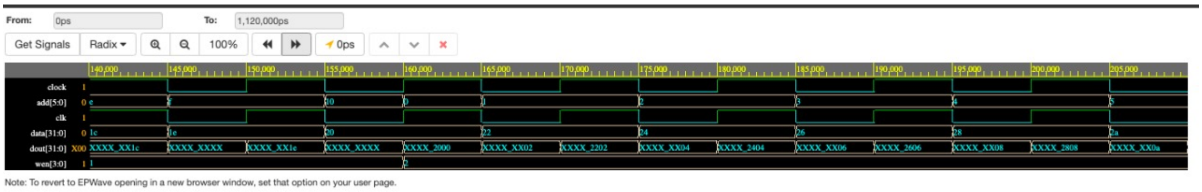
```
process
begin
    clock <= '1';   address <= "000000";
    data_in <= "00000000000000000000000000000000";
    wenable <= "0001"; -- write a byte in memory in left most byte
    for a in 0 to 15 loop
        clock <= not clock;      wait for 5 ns;
        address <= address + "000001";
        data_in <= data_in + "00000000000000000000000000000010";
        clock <= not clock;      wait for 5 ns; -- rising edge of the clock
    end loop;
    wenable <= "0010";  address <= "000000";
    -- write a byte in memory in second least significant byte
    for a in 0 to 15 loop
        clock <= not clock;      wait for 5 ns;
        address <= address + "000001";
        data_in <= data_in + "00000000000000000000000000000010";
        clock <= not clock;      wait for 5 ns; -- rising edge of the clock
    end loop;
    wenable <= "0100"; address <= "000000";
    -- write a byte in memory in second most significant byte
    for a in 0 to 15 loop
        clock <= not clock;     wait for 5 ns;
        address <= address + "000001";
        data_in <= data_in + "00000000000000000000000000000010";
        clock <= not clock;      wait for 5 ns; -- rising edge of the clock
    end loop;
    wenable <= "1000"; address <= "000000";
    -- write a byte in memory in most significant byte
    for a in 0 to 15 loop
        clock <= not clock;     wait for 5 ns;
        address <= address + "000001";
        data_in <= data_in + "00000000000000000000000000000010";
        clock <= not clock;      wait for 5 ns; -- rising edge of the clock
```

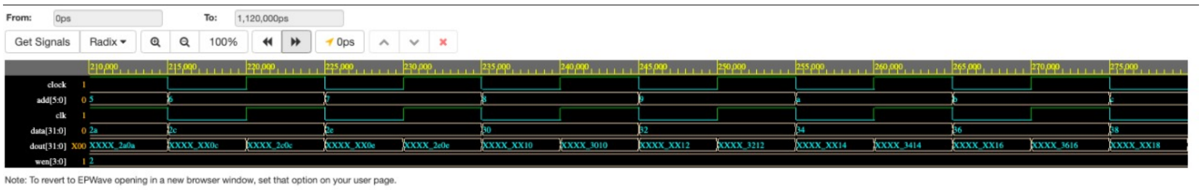
```
        end loop;
        wenable <= "0011"; -- write a byte in memory in half word (least significant)
        for a in 0 to 15 loop
            clock <= not clock;      wait for 5 ns;
            address <= address + "000001";
            data_in <= data_in + "00000000000000000000000000000010";
            clock <= not clock;      wait for 5 ns; -- rising edge of the clock
        end loop;
        wenable <= "1100"; address <= "010000";
        -- write a byte in memory in half word (most significant)
        for a in 0 to 15 loop
            clock <= not clock;      wait for 5 ns;
            address <= address + "000001";
            data_in <= data_in + "00000000000000000000000000000010";
            clock <= not clock;      wait for 5 ns; -- rising edge of the clock
        end loop;
        wenable <= "1111"; -- write a byte in memory in word
        for a in 0 to 15 loop
            clock <= not clock;      wait for 5 ns;
            address <= address + "000001";
            data_in <= data_in + "00000000000000000000000000000010";
            clock <= not clock;      wait for 5 ns; -- rising edge of the clock
        end loop;
        wait;
    end process;
```

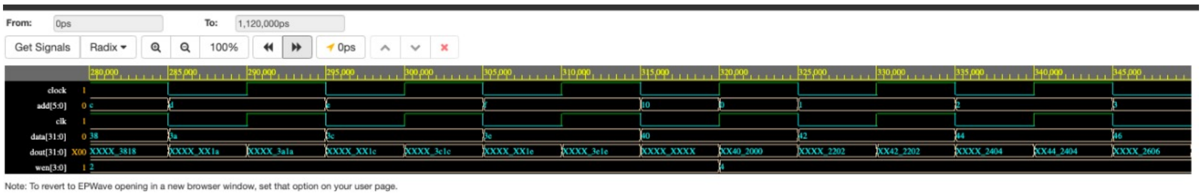
The output signal to these test case are:

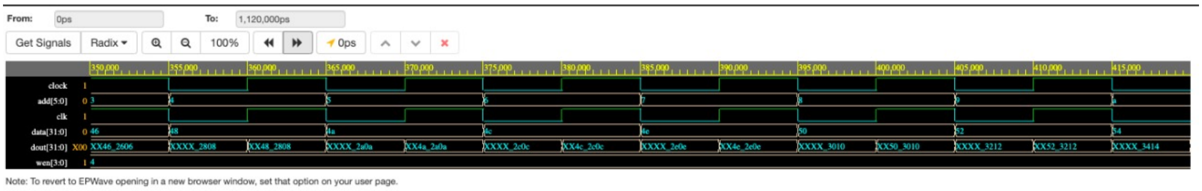




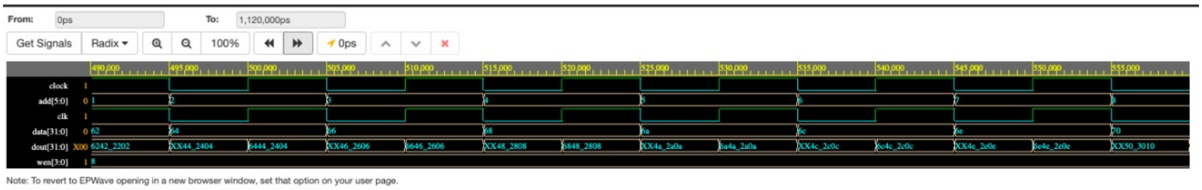


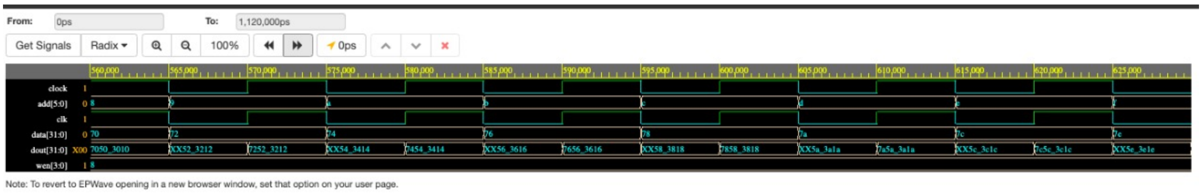


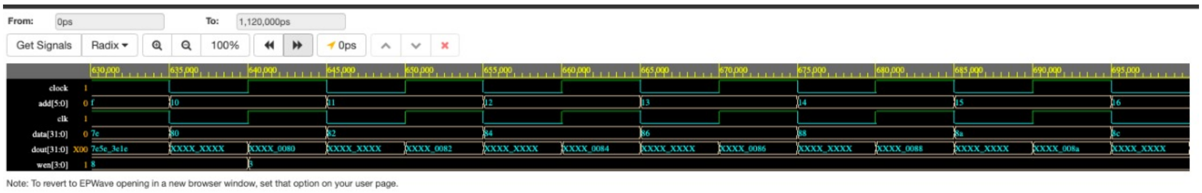

From: 0ps
To: 1,120,000ps
Get Signals
Radix
100%
0ps
clock
add[5:0]
clk
data[31:0]
dout[31:0]
wen[3:0]
Note: To revert to EPWave opening in a new browser window, set that option on your user page.


From: 0ps
To: 1,120,000ps
Get Signals
Radix
100%
0ps
clock
add[5:0]
clk
data[31:0]
dout[31:0]
wen[3:0]
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

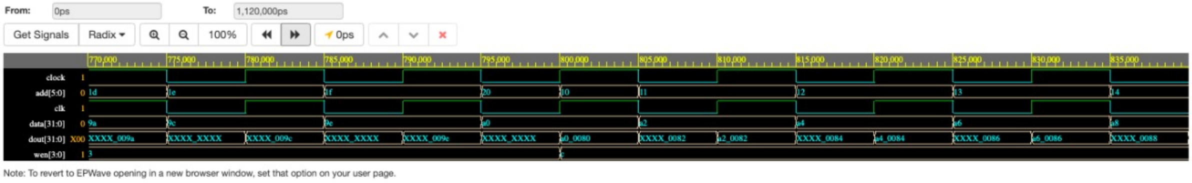

From: 0ps
To: 1,120,000ps
Get Signals
Radix
100%
0ps
clock
add[5:0]
clk
data[31:0]
dout[31:0]
wen[3:0]
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

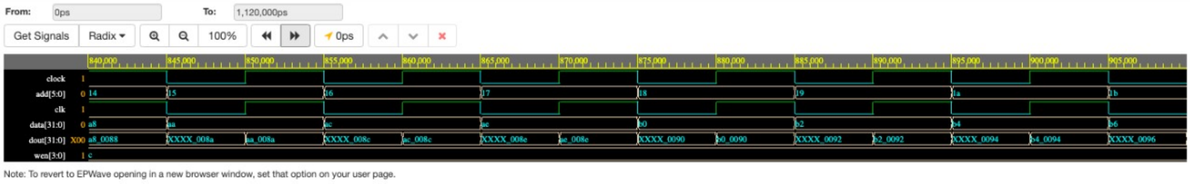

From: 0ps
To: 1,120,000ps
Get Signals
Radix
100%
0ps
clock
add[5:0]
clk
data[31:0]
dout[31:0]
wen[3:0]
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

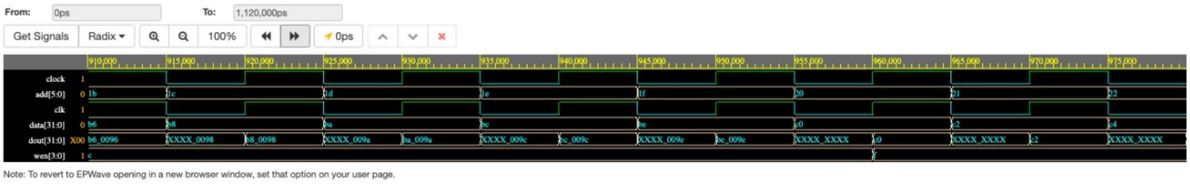

From: 0ps
To: 1,120,000ps
Get Signals
Radix
100%
0ps
clock
add[5:0]
clk
data[31:0]
dout[31:0]
wen[3:0]
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

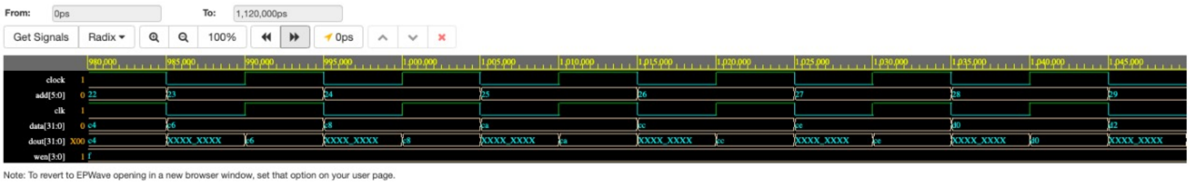

From: 0ps
To: 1,120,000ps
Get Signals
Radix
100%
0ps
clock
add[5:0]
clk
data[31:0]
dout[31:0]
wen[3:0]
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

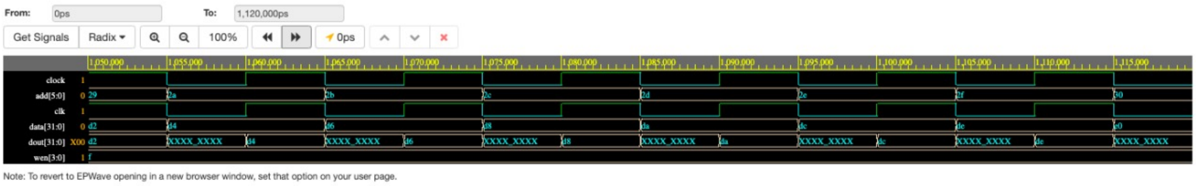

From: 0ps
To: 1,120,000ps
Get Signals
Radix
100%
0ps
clock
add[5:0]
clk
data[31:0]
dout[31:0]
wen[3:0]
Note: To revert to EPWave opening in a new browser window, set that option on your user page.











The logs of the synthesis of data Memory module are provided here:

```
# Info: *****************************************************************
# Info: Device Utilization for 7A100TCSG324
# Info: *****************************************************************
# Info: Resource                          Used    Avail   Utilization
# Info: ---------------------------------------------------------------
# Info: IOs                               75      210      35.71%
# Info: Global Buffers                    1       32        3.12%
# Info: LUTs                              62      63400     0.10%
# Info: CLB Slices                        13      15850     0.08%
# Info: Dffs or Latches                   0       126800    0.00%
# Info: Block RAMs                        0       135       0.00%
# Info: Distributed RAMs
# Info:    RAM64X1S                       32
# Info: DSP48E1s                          0       240       0.00%
# Info: ---------------------------------------------------------------
# Info: *****************************************************************
# Info: Library: work    Cell: DataMemory    View: implement_dm
# Info: *****************************************************************
# Info:  Number of ports :                          75
# Info:  Number of nets :                          180
```

```
# Info:  Number of instances :                      106
# Info:  Number of references to this view :          0
# Info: Total accumulated area :
# Info:  Number of LUTs :                            62
# Info:  Number of LUTs with LUTNM/HLUTNM :           6
# Info:  Number of accumulated instances :          137
# Info: ************************************************************
```

3 Conclusion

The program has been tested on different inputs and the results have been attached in the report. The submission file (.zip) contains 9 files other than this report file.
Two files each for each module. One contains the design (e.g. alu_design.vhd) and other contains the test bench for the module (e.g. alu_testbench.vhd). And one is run.do file that has been used to synthesis.