Università di Pisa

Electronics and communications systems - Project: Implementation of AMBA-APB communication protocol

Fabio Piras, Giacomo Volpi

Academic Year: 2022/2023

Contents

Chapter 1

Introduction and specifications

1 AMBA APB

Advanced Microcontroller Bus Architecture (AMBA) bus protocols are a set of industry-standard communication protocols for System-on-Chip memory-mapped interconnections. The Advanced Peripheral Bus (APB) is used to access low-speed peripherals and is the simplest one. It is required to design a digital circuit for implementing a communication based on AMBA-APB standard. The system is composed of one APB-Master which can send read-/write transactions to two APB-Slaves, a 64x16 ROM and a 128x8 RAM. The APB-Master coordinates the reading and writing transactions on the bus by handling the control signals of the communication interface. The APB-Slaves reply to the master's requests. The APB interface is described as follows:

- P_clk
- P_sel[0:K-1]: slave selection (K = number of slaves).
- P_write: 0 for a read transaction, 1 for a write transaction
- P_enable: enable signal
- P_rdata[0:N-1]: data read from peripheral
- P_wdata[0:N-1]: data to write to peripheral
- P_addr[0:M-1]: address (both for read and write transactions)

2 Possible applications

AMBA was introduced by ARM in 1996, this family of protocols is largely used for embedded processor bus architectures, because it is well documented and can be used without royalties. APB (Advanced Peripheral Bus) is designed for low bandwidth control accesses, for example register interfaces on system peripherals. However, even if it was designed by ARM, there are example of AMBA buses for non-ARM designs, like the Infineon ADM5120 SoC.

3 Architecture

3 .1 Input

- P_clk : the clock signal of the system

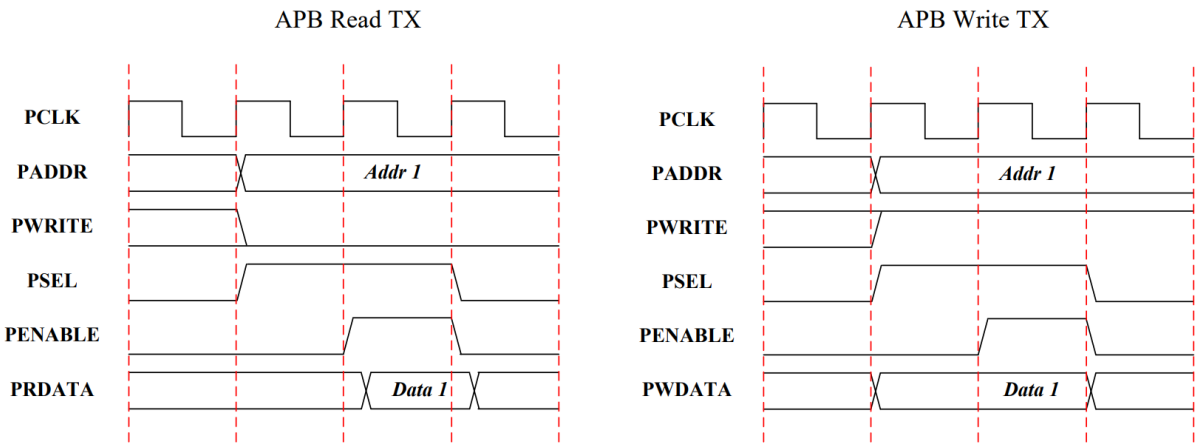


Figure 1.1: Timing of the digital circuit

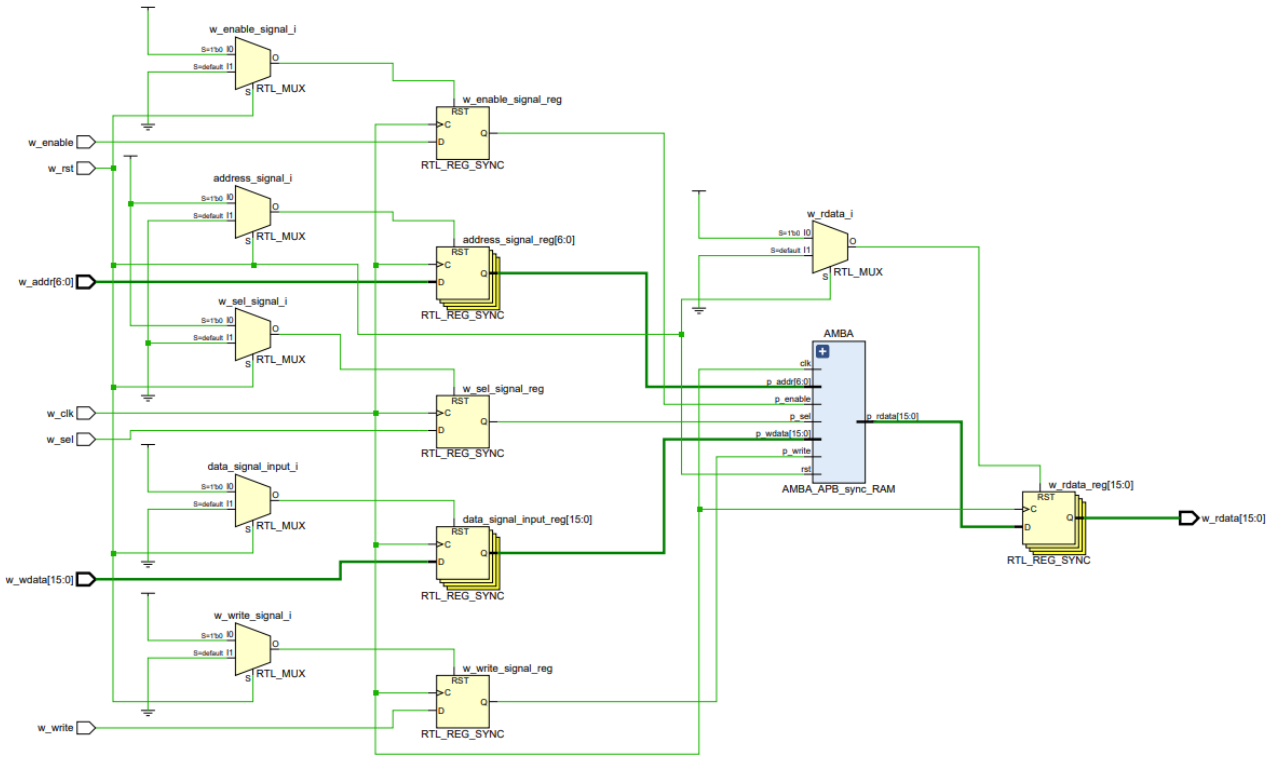


Figure 1.2: General design of the circuit

- P_reset : an asynchronous low-active reset signal
- P_sel : the slave selection signal, 0 in case of the ROM selection, 1 in case of RAM
- P_write: the write signal, 0 in case of read, 1 otherwise.
- P_enable: enable signal
- P_wdata [0:15]: data to write in peripheral
- P_addr[0:6]: address wires

3 .2 Output

- P_rdata [0:15]: read output from the system

3 .3 Components

Our digital circuits is based on 3 main components. A 64x16 ROM, a 128x8 RAM and a multiplexer.

RAM

The RAM represents a read/write memory. It has 7 wires of input for the address and 16 output wires; since the input data in our circuit is 2 byte wide and the RAM stores a byte per address, we choose to store the least significant byte of the word in the first address and the most significant byte in the next one. The read process is the same, so in order to obtain a word we read from address and address+1. Please note that our circuit does not support unaligned memory accesses, more on that in the section with Figure 2.2

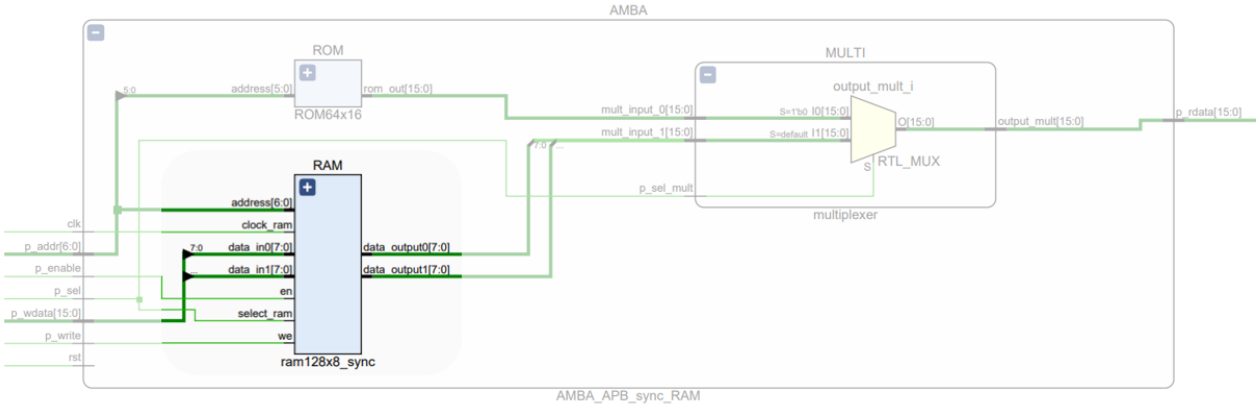


Figure 1.3: RAM

ROM

The ROM is a read-only component memory, it has 6 input wires for the address and 16 for the output. It stores a word.

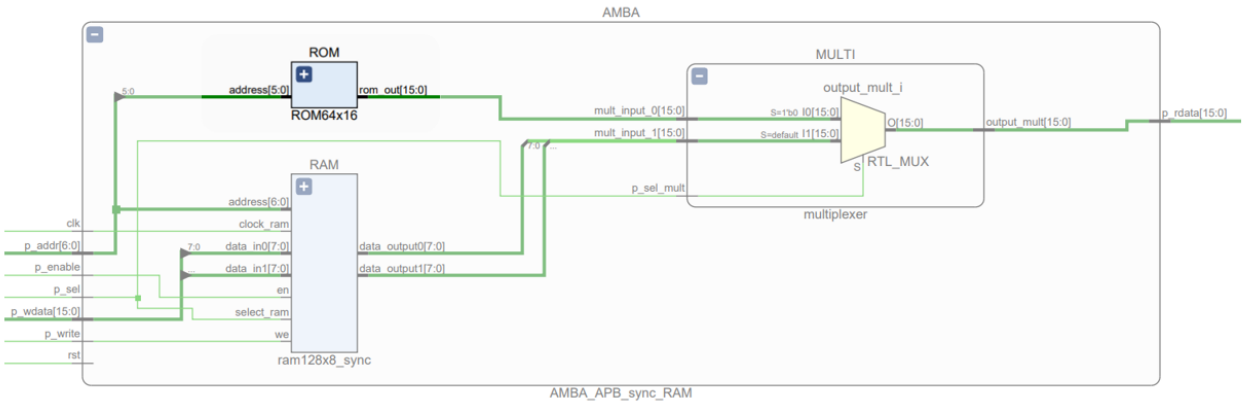


Figure 1.4: ROM

Multiplexer

The multiplexer is used to commute output between RAM and ROM, according to the selection signal. If the select is equal to 1 we read the output from the first, otherwise from the second one.

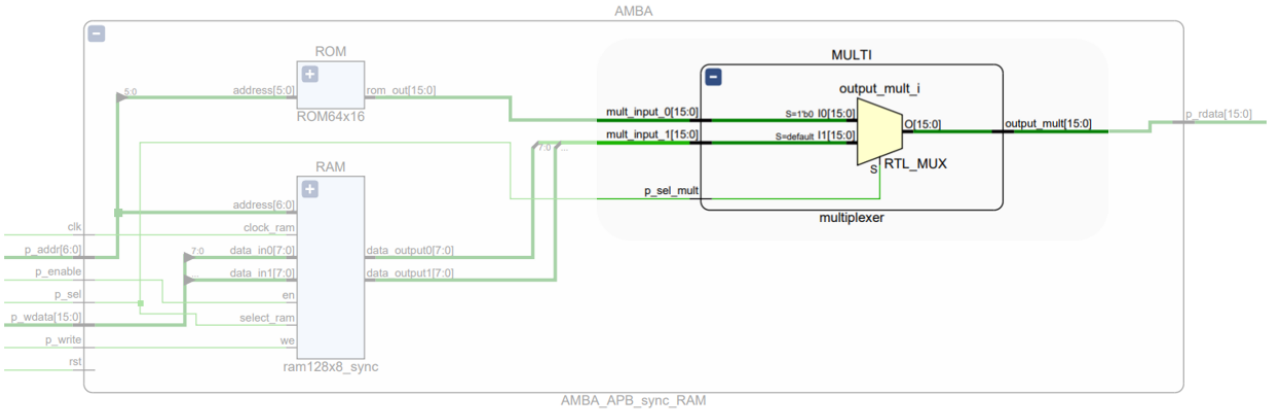


Figure 1.5: Multiplexer

Chapter 2

Verification and testing

The goal of this chapter is to verify the behavior of the designed circuit through the help of *ModelSim*, the chapter is divided in three sections the first one regarding the correct piloting of the network, the second one regarding what happens in case of wrong inputs and the third one regarding the *Vivado* wrapper.

1 Correct piloting

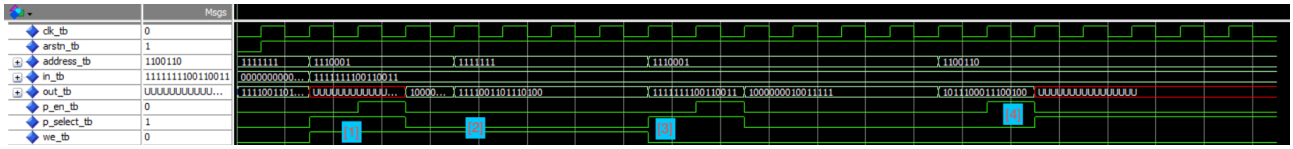


Figure 2.1: Correct piloting waveform in ModelSim

First we try to model the correct behavior of the circuit. We follow the temporization schemes in the project specification. After the reset we start with a writing operation in RAM [1]. At the second clock period we provide the address (1110001 | 113) and the input (11111111/00110011 | -205), *p_ select* and *we* (write enable) are set to 1. At the third rising edge of the clock *p_ enable* goes to 1 and the write is made. Note that the output is not significant due to the fact that we have not initialize/reset the RAM. For clock periods from 4 to 8 we have the ROM output (*p_ select* is set to 0) even if *we* is set to 1 (this means that the master does not actually read the ROM output) [2], note that this is only due to the fact that the multiplexer only commutes between the RAM and the ROM output based on the value of *p_ select*. Then we have a read operation in RAM, to test if the writing operation has been successfull [3]. The output value is the same (11111111/00110011 | -205) as before, and is present one clock before the enable arrives to respect the timing requirements form the Master point of view, this behaviour is achieved through the combinatorial data flow adopted for readings. The final test we perform is the ROM reading operation [4] that presents a behaviour equal to the RAM as previously described.

2 Wrong piloting

In this section we want to show what happen when master does not respect the temporization scheme, or in case of wrong commands (e.g. write to ROM command). First we start with a correct writing operation in RAM [1], this is used to emphasize the following incorrect operations. Then we want to test what happens in case of *p_ select* set to 0 (ROM selected) and

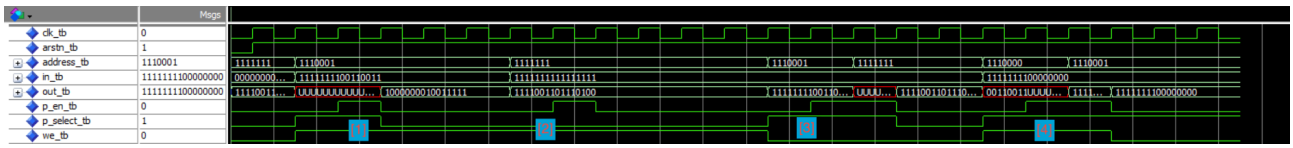


Figure 2.2: Wrong piloting waveform in ModelSim

we set to 1 (write operation); in this case no writing operation is performed on the ROM since naturally it does not take data wires as input, no writing is also performed in the RAM due to *p_ select* being set to 0. At point [3] we perform the case of a RAM reading and address change while p_enable is set to 1. In this case the output depends if the cell has a previous content or not (like in this case). For the last case [4] we consider a writing operation in RAM with address changing. In this particular case is also possible to see the non-alignment issue presented before, in this case the output wires contain the LSB of the first write [1] as the MSB and unknown values as LSB due to the RAM not being initialized/resetted at start.

3 Wrapper behaviour

For a complete case we also included the testbench on the wrapper implementation. As expected the output is presented after two clock cycles since the presence of the input and output registers.

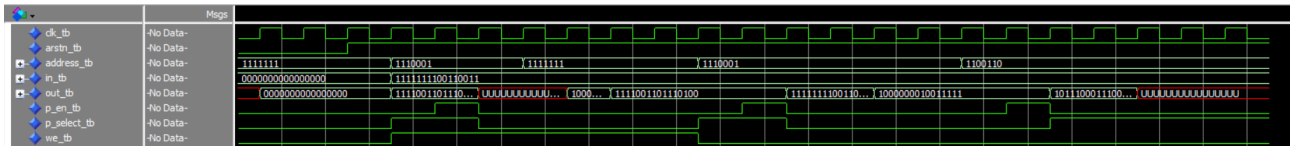


Figure 2.3: Correct piloting waveform for the wrapper in ModelSim

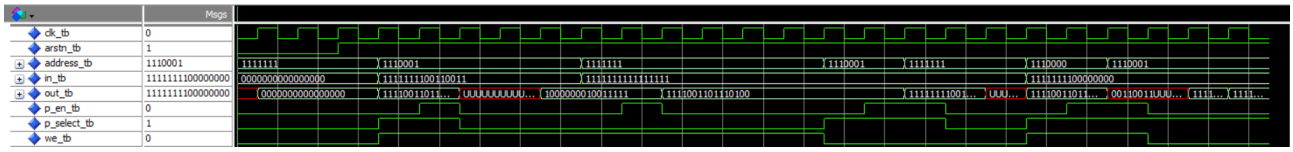


Figure 2.4: Wrong piloting waveform for the wrapper in ModelSim

Chapter 3

Vivado synthesis and implementation

In this chapter we will report the result from the synthesis and implementation of the circuit performed on *Vivado*. Naturally for the correct use of the tool we need to apply the paradigm of register-logic-register to the network, this is done by the use of the wrapper reported in section 5 . The operation were performed in out-of-context mode to avoid specifying the I/O pin mapping.

1 Warning

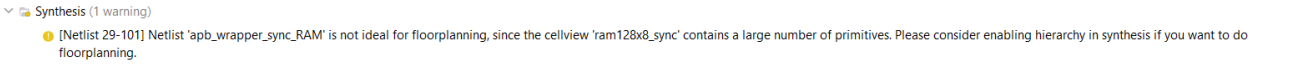


Figure 3.1: Warning produced by the Synthesis

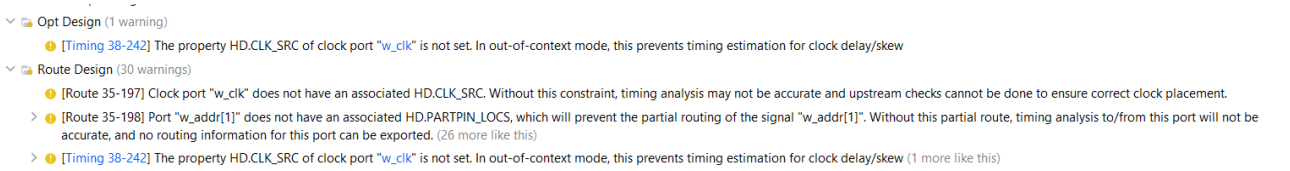


Figure 3.2: Warning produced by the Implementation

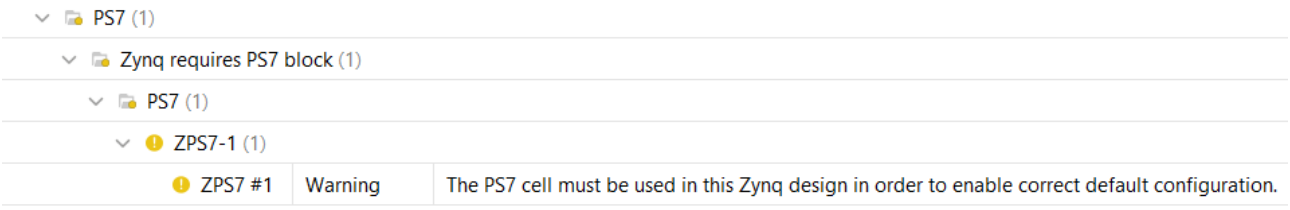


Figure 3.3: Warning produced by the DRC

The first one is caused by the way the RAM was described in VHDL, given the fact that we use an array of 127 location of one byte each, this result in a loss of time performance at the moment of synthesis and implementation on the *Vivado* tool, but it does not appear to have any more effects.

The second batch of warnings is caused by the mode out-of-context used during the *Vivado* implementation. Similarly the DRC warning are cause also by the out-of-context mode.

2 Resource utilization

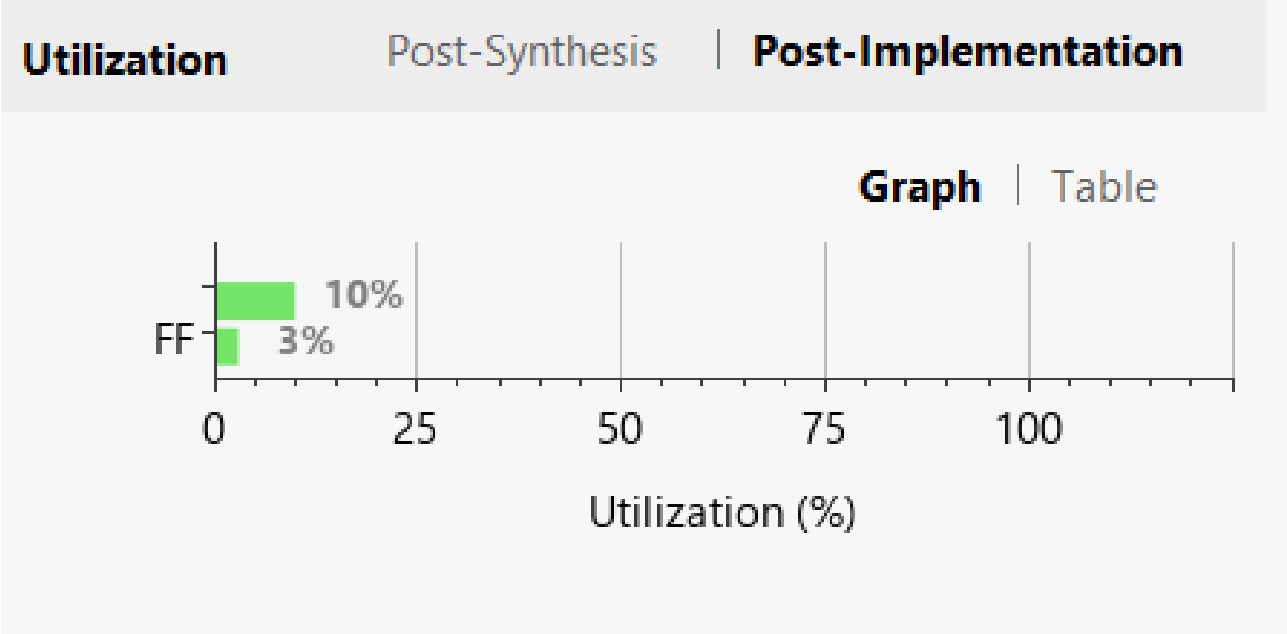
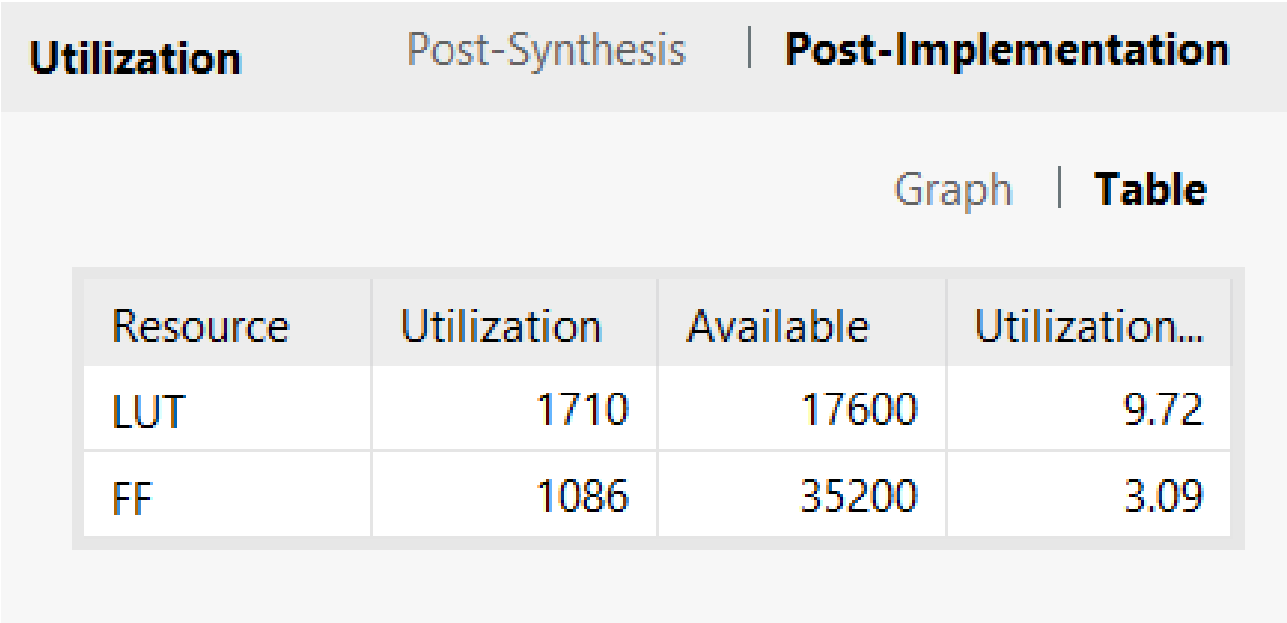


Figure 3.4: Utilization graph



Resource	Utilization	Available	Utilization...
LUT	1710	17600	9.72
FF	1086	35200	3.09

Figure 3.5: Utilization table

Given the simplicity of the operation performed, the total utilization of the resources is low, as expected.

3 Power consumption

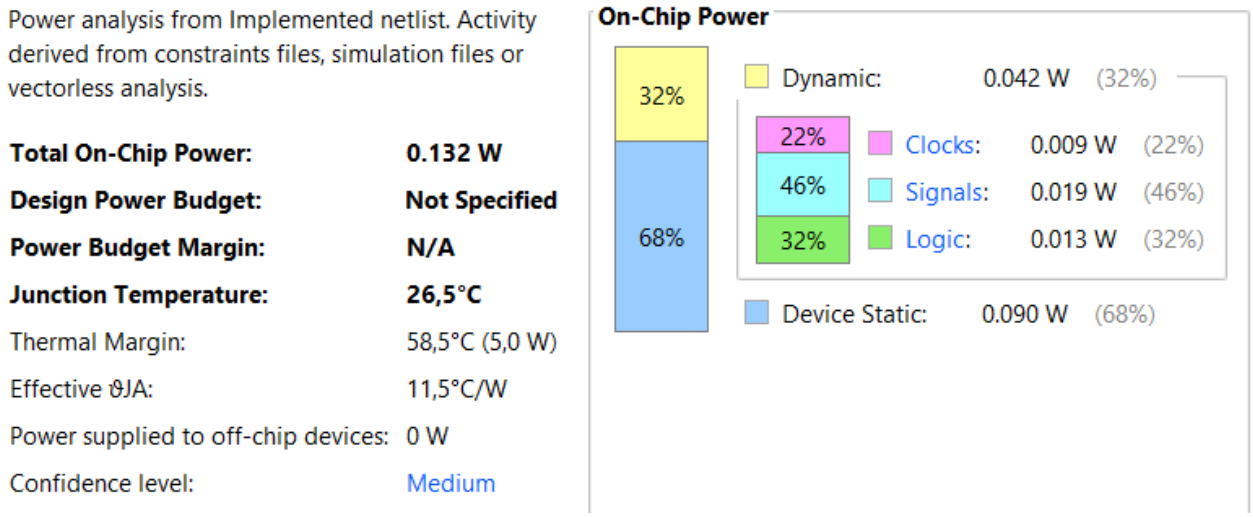


Figure 3.6: Power consumption

4 Maximum clock frequency

We start with the default clock timing of 10 ns, after synthesis we get the following result.

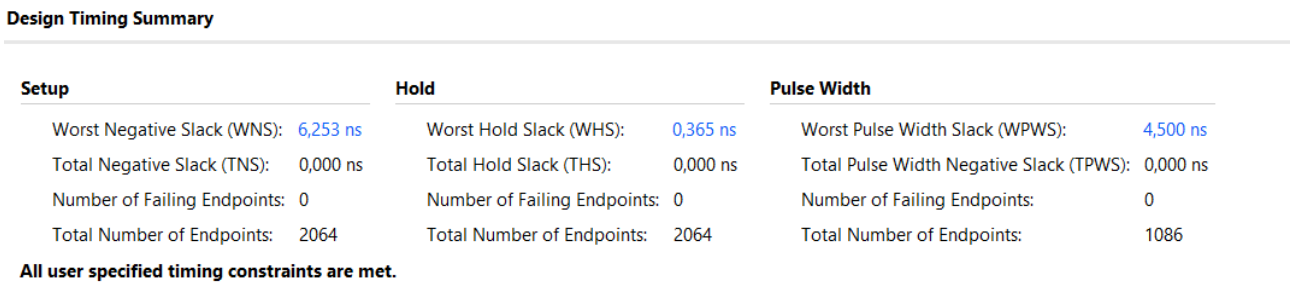


Figure 3.7: Timing for 10 ns clock

We proceed by reducing the clock to 3.747 ns following the previous result.

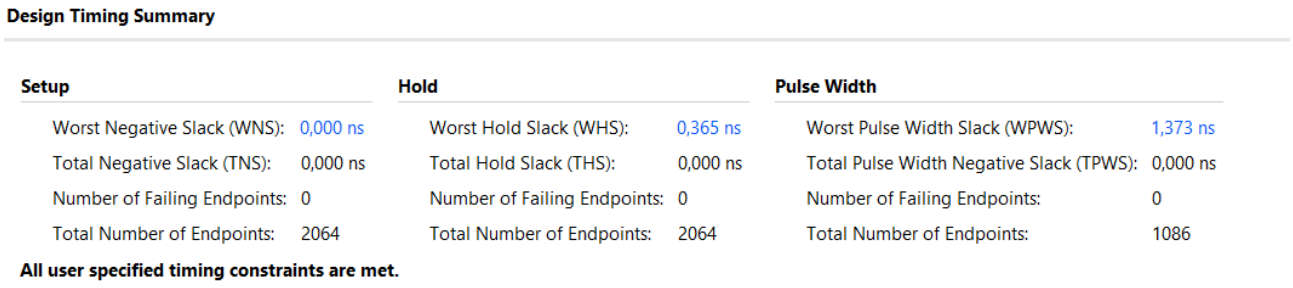


Figure 3.8: Timing for 3.747 ns clock

This results in a maximum clock frequency of approximately 266 MHz.

5 Critical path

By reducing by 0.001 ns the clock we naturally get a clock timing violation with the following critical paths.

Name	Slack	Levels	Routes	High F...	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	-0.001	5	6	2	AMBA/RAM/mem_reg[55][0]/C	w_rdata_reg[0]/D	3.725	1.583	2.142	3.7
Path 2	-0.001	5	6	2	AMBA/RAM/mem_reg[56][2]/C	w_rdata_reg[10]/D	3.725	1.583	2.142	3.7
Path 3	-0.001	5	6	2	AMBA/RAM/mem_reg[56][3]/C	w_rdata_reg[11]/D	3.725	1.583	2.142	3.7
Path 4	-0.001	5	6	2	AMBA/RAM/mem_reg[56][4]/C	w_rdata_reg[12]/D	3.725	1.583	2.142	3.7
Path 5	-0.001	5	6	2	AMBA/RAM/mem_reg[56][5]/C	w_rdata_reg[13]/D	3.725	1.583	2.142	3.7
Path 6	-0.001	5	6	2	AMBA/RAM/mem_reg[56][6]/C	w_rdata_reg[14]/D	3.725	1.583	2.142	3.7

Figure 3.9: Critical path for 3.746 ns clock

In the following is reported one of the schematic for the critical path, in particular the one for Path1

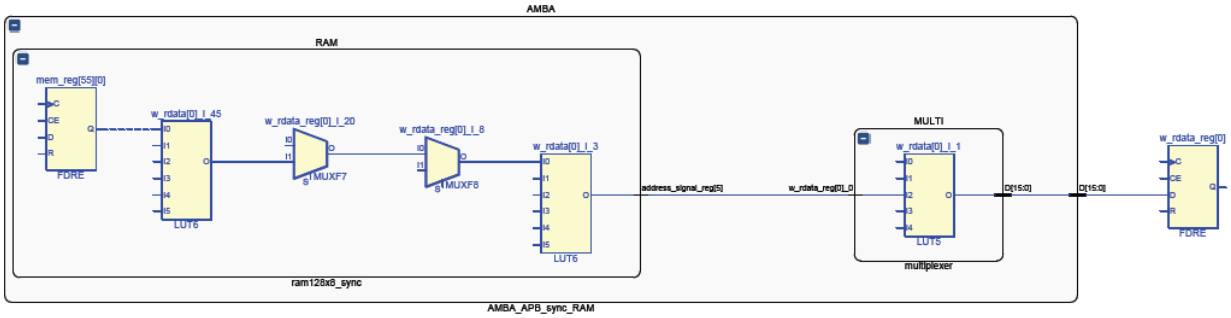


Figure 3.10: Schematic for critical path

It is interesting to notice that we get many critical path with the same slack, this is to be expected since, as shown in Figure 3.9 all of them originate from the RAM and, since the module is composed by 127 equal locations, is only natural for them to have the same slack.

6 Design

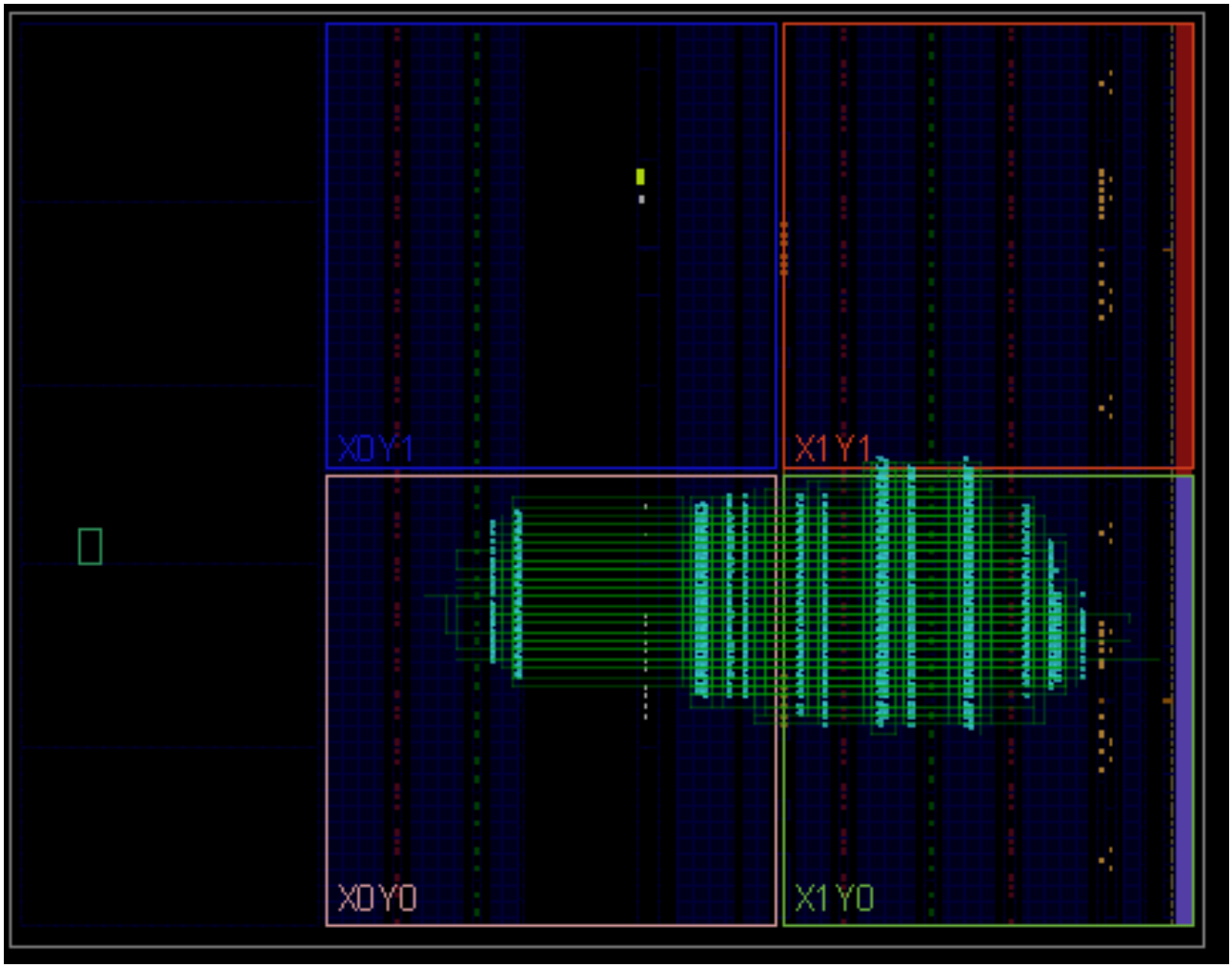


Figure 3.11: Schematic of the design

Chapter 4

Conclusion

AMBA-APB is a wildly used protocol with many design structure and implementation. In our design we managed to reach very high frequency for the clock at the expense of the power consumption. Other possible implementation could benefit from the use of a slave-master structure inside the circuit to make the network more flexible and more resistant to wrong piloting by sacrificing design simplicity and probably maximum clock frequency. As previously stated we decided not to handle operation not align to multiple of 16 regarding the address, this means that the master is responsible for handling it in the most proper way in base of its own design.

Chapter 5

VHDL code

1 Multiplexer

```
1  library IEEE;
2    use IEEE.std_logic_1164.all;
3    use IEEE.numeric_std.all;
4
5  entity multiplexer is
6    generic(
7      DATA_WIDTH_MULTI :integer := 16
8    );
9
10   port (
11     mult_input_0  : in  std_logic_vector(DATA_WIDTH_MULTI-1
     downto 0);     -- input from the ROM
12     mult_input_1  : in  std_logic_vector(DATA_WIDTH_MULTI-1
     downto 0);     -- input form the RAM
13     output_mult   : out std_logic_vector(DATA_WIDTH_MULTI-1
     downto 0);     -- output of the multiplexer
14     p_sel_mult    : in  std_logic
              -- input piloting the multiplexer
15   );
16 end entity;
17
18 architecture my_multiplexer of multiplexer is
19 begin
20
21   -- PROC describe the process for generating the output of the
      multiplexer
22   -- based upon the selection input
23   PROC : process (mult_input_0, mult_input_1, p_sel_mult)
24   begin
25     if (p_sel_mult = '0') then     -- read from ROM
26       output_mult <= mult_input_0;
27     else                           -- read from RAM
28       output_mult <= mult_input_1;
29     end if;
30   end process;
```

```
31
32 end architecture;
```

2 RAM

```
1  library ieee;
2      use ieee.std_logic_1164.all;
3      use ieee.std_logic_unsigned.all;
4      use ieee.numeric_std.all;
5
6  entity ram128x8_sync is
7      generic (
8          DATA_WIDTH_RAM:integer := 8;
9          ADDR_WIDTH_RAM :integer := 7
10     );
11     port (
12         address          :in std_logic_vector (ADDR_WIDTH_RAM-1
       downto 0);    -- address input
13         data_in0         :in std_logic_vector (DATA_WIDTH_RAM-1
       downto 0);    -- data input for lower byte
14         data_in1         :in std_logic_vector (DATA_WIDTH_RAM-1
       downto 0);    -- data input for upper byte
15         data_output0     :out std_logic_vector (DATA_WIDTH_RAM-1
       downto 0);  -- data output for lower byte
16         data_output1     :out std_logic_vector (DATA_WIDTH_RAM-1
       downto 0);  -- data output for upper byte
17         en               :in std_logic;
                -- enable input
18         we               :in std_logic;
                -- write = 1 | read = 0
19         select_ram       :in std_logic;
                -- select for the operation
20         clock_ram        :in std_logic
                -- clock
21
22     );
23 end entity;
24 architecture RAM_test of ram128x8_sync is
25     -- the RAM is model as an array of std_logic_vector of 1 byte
26     -- this implementation is likely the cause of warning "
       Netlist 29-101" in Vivado
27     type RAM is array(0 to 127) of std_logic_vector(7 downto 0);
28     signal mem : RAM;
29
30 begin
31     -- ram_work describe the synchronous process of writing
32     ram_work: process (clock_ram)
33     begin
34         if (rising_edge(clock_ram)) then         --RAM
       synchronization with the system clock
```

```
35
36          -- this loop can be used to set all location of the RAM
    to 0
37          -- this avoid seeing the UUUUUUUU output in the ModelSim
    timewave
38      -- if reset_ram= '0' then        --reset condition
39      --  for i in 0 to 127 loop         -- loop to reset to 0 all
    RAM location
40          --      mem(i) <= (others=>'0');
41      --  end loop;
42
43              if (we = '1' and en='1' and select_ram='1') then
    -- write condition:
44                  mem(to_integer(unsigned(address))) <= data_in0;

45                  mem(to_integer(unsigned(address+1))) <= data_in1;
     -- data is written in the location according to little endian
     paradigm
46
47              end if;
48          end if;
49          end process;
50
51          data_output0 <= mem(to_integer(unsigned(address)));
52          data_output1 <= mem(to_integer(unsigned(address+1)));
       -- data is read from the location accordimg to little
    endian paradigm
53
54  end architecture;
```

3 ROM

```
1  library IEEE;
2    use IEEE.std_logic_1164.all;
3    use IEEE.numeric_std.all;
4
5  -- the ROM entity is model after LUT
6  entity ROM64x16 is
7    generic(
8      DATA_WIDTH_ROM : natural := 16;
9      ADDR_WIDTH_ROM : natural := 6
10   );
11   --ROM acts in a combinatorial way, so it only need the addres
     as input and 16 wire for the output
12   --since the ROM is a read only memory it does not need other
     input to determin the operation
13   port (
14     address    : in  std_logic_vector(ADDR_WIDTH_ROM-1 downto 0);
       -- address input
15     rom_out    : out std_logic_vector(DATA_WIDTH_ROM-1 downto 0)
```

```
    -- data output
  );
end entity;

architecture rtl of ROM64x16 is

  type LUT_t is array (natural range 0 to 63) of integer;
  constant LUT: LUT_t := (
    0 => 0,
    1 => 3212,
    2 => 6393,
    3 => 9512,
    4 => 12539,
    5 => 15446,
    6 => 18204,
    7 => 20787,
    8 => 23170,
    9 => 25329,
    10 => 27245,
    11 => 28898,
    12 => 30273,
    13 => 31356,
    14 => 32137,
    15 => 32609,
    16 => 32767,
    17 => 32609,
    18 => 32137,
    19 => 31356,
    20 => 30273,
    21 => 28898,
    22 => 27245,
    23 => 25329,
    24 => 23170,
    25 => 20787,
    26 => 18204,
    27 => 15446,
    28 => 12539,
    29 => 9512,
    30 => 6393,
    31 => 3212,
    32 => 0,
    33 => -3212,
    34 => -6393,
    35 => -9512,
    36 => -12539,
    37 => -15446,
    38 => -18204,
    39 => -20787,
    40 => -23170,
    41 => -25329,
```

```
65      42 => -27245,
66      43 => -28898,
67      44 => -30273,
68      45 => -31356,
69      46 => -32137,
70      47 => -32609,
71      48 => -32767,
72      49 => -32609,
73      50 => -32137,
74      51 => -31356,
75      52 => -30273,
76      53 => -28898,
77      54 => -27245,
78      55 => -25329,
79      56 => -23170,
80      57 => -20787,
81      58 => -18204,
82      59 => -15446,
83      60 => -12539,
84      61 => -9512,
85      62 => -6393,
86      63 => -3212
87    );
88
89  begin
90    rom_out <= std_logic_vector(to_signed(LUT(to_integer(unsigned(
       address))),16));
91  end architecture;
```

4 AMBA-APB

```
1   library IEEE;
2     use IEEE.std_logic_1164.all;
3     use IEEE.numeric_std.all;
4
5   --Main architecture of the AMBA-APB component
6   entity AMBA_APB_sync_RAM is
7     generic(
8       DATA_WIDTH_APB :integer := 16;
9       ADDR_WIDTH_APB :integer := 7
10    );
11    port (
12      p_addr    : in  std_logic_vector(ADDR_WIDTH_APB-1 downto 0);
         -- address input
13      p_rdata   : out std_logic_vector(DATA_WIDTH_APB-1 downto 0);
         -- data output (for reading operations)
14      p_wdata   : in std_logic_vector(DATA_WIDTH_APB-1 downto 0);
         -- data input (for writing operations)
15      p_enable  : in std_logic;
         -- enable piloting input
```

```
16      p_write    : in std_logic;
         -- write piloting input
17      p_sel      : in std_logic;
         -- selection piloting input
18      clk        : in std_logic;
         -- clock
19      rst        : in std_logic
         -- reset
20    );
21  end entity;
22
23  architecture my_AMBA_APB of AMBA_APB_sync_RAM is
24    signal ROM_OUTPUT_TO_MULTIPLEXER : std_logic_vector(
       DATA_WIDTH_APB-1 downto 0);    -- signal connecting the output
       of the ROM to the input of the multiplexer
25    signal RAM_OUTPUT_TO_MULTIPLEXER : std_logic_vector (
       DATA_WIDTH_APB-1 downto 0);  -- signal connecting the output
       of the RAM to the input of the multiplexer
26    signal RAM_OUT_LSB : std_logic_vector(DATA_WIDTH_APB/2-1 downto
       0);                -- signal tracking the least significant
       byte in output from the RAM
27    signal RAM_OUT_MSB : std_logic_vector(DATA_WIDTH_APB/2-1 downto
       0);                -- signal tracking the most significant
       byte in output from the RAM
28
29    component multiplexer is
30      generic(
31        DATA_WIDTH_MULTI :integer := 16
32      );
33
34      port (
35        mult_input_0   : in  std_logic_vector(DATA_WIDTH_MULTI-1
       downto 0);     -- input from the ROM
36        mult_input_1   : in  std_logic_vector(DATA_WIDTH_MULTI-1
       downto 0);     -- input form the RAM
37        output_mult    : out std_logic_vector(DATA_WIDTH_MULTI-1
       downto 0);     -- output of the multiplexer
38        p_sel_mult     : in  std_logic
                  -- input piloting the multiplexer
39      );
40    end component;
41
42    component ram128x8_sync is
43      generic (
44        DATA_WIDTH_RAM :integer := 8;
45        ADDR_WIDTH_RAM :integer := 7
46      );
47      port (
48        address        : in    std_logic_vector (ADDR_WIDTH_RAM-1
       downto 0);    -- address input
```

```
49      data_in0       : in    std_logic_vector (DATA_WIDTH_RAM-1
    downto 0);    -- data input for lower byte
50      data_in1       : in    std_logic_vector (DATA_WIDTH_RAM-1
    downto 0);    -- data input for upper byte
51      data_output0   : out   std_logic_vector (DATA_WIDTH_RAM-1
    downto 0);    -- data output for lower byte
52      data_output1   : out   std_logic_vector (DATA_WIDTH_RAM-1
    downto 0);    -- data output for upper byte
53      en             : in    std_logic;
              -- enable input
54      we             : in    std_logic;
              -- write = 1 | read = 0
55      select_ram     : in    std_logic;
              -- select for the operation
56      clock_ram      : in    std_logic
              -- clock
57    );
58  end component;
59
60  component ROM64x16 is
61    generic(
62      DATA_WIDTH_ROM :integer := 16;
63      ADDR_WIDTH_ROM :integer := 6
64    );
65    port (
66      address    : in  std_logic_vector(ADDR_WIDTH_ROM-1 downto 0)
    ;  -- address input
67      rom_out    : out std_logic_vector(DATA_WIDTH_ROM-1 downto 0)
      -- data output
68    );
69  end component;
70
71  begin
72
73    ROM : ROM64x16
74    generic map(
75      DATA_WIDTH_ROM => DATA_WIDTH_APB,
76      ADDR_WIDTH_ROM => ADDR_WIDTH_APB-1
77    )
78
79    port map(
80        address => p_addr(ADDR_WIDTH_APB-2 downto 0),    -- ROM
    only need 6 bits of input as addres
81        rom_out => ROM_OUTPUT_TO_MULTIPLEXER             -- signal
     is used to pass data from the ROM to the multiplexer
82    );
83
84    RAM : ram128x8_sync
85    generic map(
86      DATA_WIDTH_RAM=>DATA_WIDTH_APB/2,
```

```
87        ADDR_WIDTH_RAM=>ADDR_WIDTH_APB
88      )
89
90      port map(
91           address => p_addr,
92           data_output0 => RAM_OUT_LSB,                    -- map
    the least significant byte from the output of the RAM
93           data_output1 => RAM_OUT_MSB,                    -- map
    the most significant byte from the output of the RAM
94           data_in0 => p_wdata(7 downto 0),                -- map
    the least significant byte to input of the RAM
95           data_in1 => p_wdata(15 downto 8),               -- map
    the most significant byte to input of the RAM
96           en => p_enable,
97           we => p_write,
98           clock_ram => clk,
99           select_ram => p_sel
100     );
101
102     MULTI: multiplexer
103     generic map(
104       DATA_WIDTH_MULTI=>DATA_WIDTH_APB
105     )
106
107     port map(
108          mult_input_0 => ROM_OUTPUT_TO_MULTIPLEXER,       -- map
    the input of the multiplexer coming from the ROM
109          mult_input_1 =>RAM_OUTPUT_TO_MULTIPLEXER,        -- map
    the input of the multiplexer coming from the RAM
110          output_mult => p_rdata,
111          p_sel_mult => p_sel
112     );
113
114     RAM_OUTPUT_TO_MULTIPLEXER <= RAM_OUT_MSB & RAM_OUT_LSB;   --
    recombining the two signals into one
115
116
117 end architecture;
```

5 Vivado wrapper

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  -- the wrapper is used since Vivado only evaluates timing on
     Register-Logic-Register paths
5  entity apb_wrapper_sync_RAM is
6    generic(
7          DATA_WIDTH_WRAPPER :natural := 16;
8          ADDR_WIDTH_WRAPPER :natural := 7
```

```
9   );
10
11   port(
12     w_addr       : in  std_logic_vector(ADDR_WIDTH_WRAPPER-1
     downto 0);
13     w_rdata      : out std_logic_vector(DATA_WIDTH_WRAPPER-1
     downto 0);
14     w_wdata      : in std_logic_vector(DATA_WIDTH_WRAPPER-1 downto
      0);
15     w_enable     : in std_logic;
16     w_write      : in std_logic;
17     w_sel        : in std_logic;
18     w_clk        : in std_logic;
19     w_rst        : in std_logic
20   );
21 end entity;
22
23
24
25
26 architecture struct of apb_wrapper_sync_RAM is
27     -- the signal are used to pass the data from the register to
    the AMBA_APB logic
28     signal address_signal : std_logic_vector(ADDR_WIDTH_WRAPPER-1
     downto 0);
29     signal data_signal_input : std_logic_vector(
    DATA_WIDTH_WRAPPER-1 downto 0);
30     signal data_signal_output : std_logic_vector(
    DATA_WIDTH_WRAPPER-1 downto 0);
31     signal w_enable_signal : std_logic;
32     signal w_sel_signal : std_logic;
33     signal w_write_signal : std_logic;
34
35 component AMBA_APB_sync_RAM is
36     generic(
37         DATA_WIDTH_APB :natural := 16;
38         ADDR_WIDTH_APB :natural := 7
39     );
40     port (
41         p_addr       : in  std_logic_vector(ADDR_WIDTH_APB-1
    downto 0);
42         p_rdata      : out std_logic_vector(DATA_WIDTH_APB-1
    downto 0);
43         p_wdata      : in std_logic_vector(DATA_WIDTH_APB-1 downto
      0);
44         p_enable     : in std_logic;
45         p_write      : in std_logic;
46         p_sel        : in std_logic;
47         clk          : in std_logic;
48         rst          : in std_logic
```

```
49      );
50  end component;
51
52  begin
53
54      AMBA: AMBA_APB_sync_RAM
55      generic map(
56          DATA_WIDTH_APB => DATA_WIDTH_WRAPPER,
57          ADDR_WIDTH_APB => ADDR_WIDTH_WRAPPER
58      )
59
60      port map(
61          p_addr => address_signal,
62          p_rdata => data_signal_output,
63          p_wdata => data_signal_input,
64          p_enable => w_enable_signal,
65          p_write => w_write_signal,
66          p_sel => w_sel_signal,
67          clk => w_clk,
68          rst => w_rst
69      );
70
71
72      -- the process mimic the register to implement the paradigm
     of register-logic-register for Vivado
73      wrapper: process (w_clk)
74    begin
75      if (rising_edge (w_clk)) then
76        if(w_rst = '0') then
77          w_sel_signal <= '0';
78          w_write_signal <= '0';
79          data_signal_input <= (others => '0');
80          address_signal <= (others => '0');
81                  w_rdata <= (others => '0');
82          w_enable_signal <= '0';
83        else
84                  w_sel_signal <= w_sel;
85          w_write_signal <= w_write;
86          data_signal_input <= w_wdata;
87                  w_rdata<=data_signal_output;
88          address_signal <= w_addr;
89          w_enable_signal <= w_enable;
90        end if;
91      end if;
92      end process;
93
94  end architecture;
95
96
97
```

6 Testbench

6 .1 Correct tb

```
library IEEE;
use IEEE.std_logic_1164.all;

entity AMBA_APB_tb_correct is
    end entity;

architecture testbench of AMBA_APB_tb_correct is

    constant DATA_WIDTH_tb :natural :=16;
    constant ADDR_WIDTH_tb :natural := 7;
    constant T_CLK : time := 10 ns;
    constant T_RESET : time := 5 ns;

    signal clk_tb : std_logic := '0';
    signal arstn_tb : std_logic := '0';
    signal address_tb : std_logic_vector(ADDR_WIDTH_tb-1 downto 0)  := (others => '1');
    signal in_tb : std_logic_vector (DATA_WIDTH_tb-1 downto 0) := (others => '0');
    signal out_tb : std_logic_vector (DATA_WIDTH_tb-1 downto 0);
    signal p_en_tb : std_logic := '0';
    signal p_select_tb : std_logic := '0';
    signal we_tb    :std_logic :='0';
    signal sim_stop : std_logic := '1';

    component AMBA_APB_sync_RAM is
        generic(
        DATA_WIDTH_APB :natural := 16;
        ADDR_WIDTH_APB :natural := 7
    );
    port (
        p_addr      : in  std_logic_vector(ADDR_WIDTH_APB-1 downto 0);
        p_rdata     : out std_logic_vector(DATA_WIDTH_APB-1 downto 0);
        p_wdata     : in std_logic_vector(DATA_WIDTH_APB-1 downto 0);
        p_enable    : in std_logic;
        p_write     : in std_logic;
        p_sel       : in std_logic;
        clk         : in std_logic;
        rst         : in std_logic
    );
    end component;

begin
```

```
42      clk_tb <= not(clk_tb) and sim_stop after T_CLK/2;
43      arstn_tb <= '1' after T_RESET;
44
45      AMBA: AMBA_APB_sync_RAM
46          generic map(
47              DATA_WIDTH_APB =>DATA_WIDTH_tb,
48              ADDR_WIDTH_APB =>ADDR_WIDTH_tb
49          )
50
51          port map(
52              p_addr=>address_tb,
53              p_rdata=>out_tb,
54              p_wdata=>in_tb,
55              p_enable => p_en_tb,
56              p_write => we_tb,
57              p_sel => p_select_tb,
58              clk => clk_tb,
59              rst => arstn_tb
60          );
61
62      STIMULI : process(clk_tb, arstn_tb)
63          variable t : integer := 0;
64      begin
65          if arstn_tb = '0' then
66              t :=0;
67              elsif rising_edge(clk_tb) then
68                  case(t) is
69                      when 1 => address_tb <="1110001"; in_tb <=
    "1111111100110011"; we_tb<='1'; p_select_tb<='1'; -- WRITE IN
    RAM
70                      when 2 => p_en_tb<='1';
71                      when 3 => p_en_tb<='0'; p_select_tb <= '0';
72                      when 4 => address_tb <="1111111";
73
74
75                      when 8 => address_tb <="1110001"; we_tb<='0';
     p_select_tb<='1'; -- READ IN RAM
76                      when 9 => p_en_tb<='1';
77                      when 10 => p_en_tb <='0';  p_select_tb<='0';
78
79                      when 14 => address_tb <="1100110"; we_tb
    <='0'; p_select_tb<='0'; -- READ IN ROM
80                      when 15 => p_en_tb <= '1';
81                      when 16 => p_en_tb <='0';  p_select_tb<='1';
82
83                      when 20 => sim_stop <= '0';
84                      when others => null;
85                  end case;
86                  t := t+1;
87              end if;
```

```
88      end process;
89  end architecture;
```

6 .2 Incorrect tb

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity AMBA_APB_tb_incorrect is
5       end entity;
6
7   architecture testbench of AMBA_APB_tb_incorrect is
8
9       constant DATA_WIDTH_tb :natural :=16;
10      constant ADDR_WIDTH_tb :natural := 7;
11      constant T_CLK : time := 10 ns;
12      constant T_RESET : time := 5 ns;
13
14      signal clk_tb : std_logic := '0';
15      signal arstn_tb : std_logic := '0';
16      signal address_tb : std_logic_vector(ADDR_WIDTH_tb-1 downto
    0)  := (others => '1');
17      signal in_tb : std_logic_vector (DATA_WIDTH_tb-1 downto 0)
    := (others => '0');
18      signal out_tb : std_logic_vector (DATA_WIDTH_tb-1 downto 0);
19      signal p_en_tb : std_logic := '0';
20      signal p_select_tb : std_logic := '0';
21      signal we_tb    :std_logic :='0';
22      signal sim_stop : std_logic := '1';
23
24      component AMBA_APB_sync_RAM is
25          generic(
26          DATA_WIDTH_APB :natural := 16;
27          ADDR_WIDTH_APB :natural := 7
28      );
29      port (
30          p_addr      : in  std_logic_vector(ADDR_WIDTH_APB-1
    downto 0);
31          p_rdata     : out std_logic_vector(DATA_WIDTH_APB-1
    downto 0);
32          p_wdata     : in std_logic_vector(DATA_WIDTH_APB-1 downto
     0);
33          p_enable    : in std_logic;
34          p_write     : in std_logic;
35          p_sel       : in std_logic;
36          clk         : in std_logic;
37          rst         : in std_logic
38      );
39      end component;
40
41  begin
```

```
42      clk_tb <= not(clk_tb) and sim_stop after T_CLK/2;
43      arstn_tb <= '1' after T_RESET;
44
45      AMBA: AMBA_APB_sync_RAM
46          generic map(
47              DATA_WIDTH_APB =>DATA_WIDTH_tb,
48              ADDR_WIDTH_APB =>ADDR_WIDTH_tb
49          )
50
51          port map(
52              p_addr=>address_tb,
53              p_rdata=>out_tb,
54              p_wdata=>in_tb,
55              p_enable => p_en_tb,
56              p_write => we_tb,
57              p_sel => p_select_tb,
58              clk => clk_tb,
59              rst => arstn_tb
60          );
61
62      STIMULI : process(clk_tb, arstn_tb)
63          variable t : integer := 0;
64      begin
65          if arstn_tb = '0' then
66              t :=0;
67              elsif rising_edge(clk_tb) then
68                  case(t) is
69
70                      when 1 => address_tb <="1110001"; in_tb <=
    "1111111100110011"; we_tb<='1'; p_select_tb<='1'; -- WRITE IN
    RAM
71                      when 2 => p_en_tb<='1';
72                      when 3 => p_en_tb<='0'; p_select_tb <= '0';
73
74                      when 6 => address_tb <="1111111"; in_tb <=
    "1111111111111111"; we_tb<='1'; p_select_tb<='0'; -- WRITE IN
    ROM (wrong behaviour)
75                      when 7 => p_en_tb<='1';
76                      when 8 => p_en_tb<='0';
77
78                      when 12 => address_tb <="1110001"; we_tb
    <='0'; p_select_tb<='1'; -- READ CYCLE FROM RAM WITH ADDR
    CHANGE (wrong behaviour)
79                      when 13=> p_en_tb<='1';
80                      when 14 => address_tb <="1111111";
81                      when 15 => p_en_tb<='0'; p_select_tb<='0';
82
83
84                      -- READ CYCLE FOLLOWING A WRITE CYCLE WITH
    ADDR CHANGE AND NO RESET OF VARIABLES (wrong behaviour)
```

```
85                      when 17 => address_tb <="1110000"; in_tb <=
    "1111111100000000"; we_tb<='1'; p_select_tb<='1';
86                      when 18 => p_en_tb<='1';
87                      when 19 => address_tb <="1110001";
88                      when 20 => we_tb<='0'; p_en_tb<='0';
89
90                      when 22 => sim_stop <= '0';
91                      when others => null;
92                  end case;
93                  t := t+1;
94              end if;
95      end process;
96  end architecture;
```

6 .3 Wrapper

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity AMBA_Wrapper_tb_v2 is
5       end entity;
6
7   architecture testbench of AMBA_Wrapper_tb_v2 is
8
9       constant DATA_WIDTH_tb :natural :=16;
10      constant ADDR_WIDTH_tb :natural := 7;
11      constant T_CLK : time := 10 ns;
12      constant T_RESET : time := 25 ns;
13
14      signal clk_tb : std_logic := '0';
15      signal arstn_tb : std_logic := '0';
16      signal address_tb : std_logic_vector(ADDR_WIDTH_tb-1 downto
    0)  := (others => '1');
17      signal in_tb : std_logic_vector (DATA_WIDTH_tb-1 downto 0)
    := (others => '0');
18      signal out_tb : std_logic_vector (DATA_WIDTH_tb-1 downto 0);
19      signal p_en_tb : std_logic := '0';
20      signal p_select_tb : std_logic := '0';
21      signal we_tb    :std_logic :='0';
22      signal sim_stop : std_logic := '1';
23
24      component apb_wrapper_sync_RAM is
25          generic(
26              DATA_WIDTH_WRAPPER :natural := 16;
27              ADDR_WIDTH_WRAPPER :natural := 7
28        );
29
30        port(
31          w_addr      : in  std_logic_vector(ADDR_WIDTH_WRAPPER-1
    downto 0);
32          w_rdata     : out std_logic_vector(DATA_WIDTH_WRAPPER-1
```

```
    downto 0);
33          w_wdata      : in std_logic_vector(DATA_WIDTH_WRAPPER-1
    downto 0);
34          w_enable     : in std_logic;
35          w_write      : in std_logic;
36          w_sel        : in std_logic;
37          w_clk        : in std_logic;
38          w_rst        : in std_logic
39        );
40      end component;
41
42  begin
43      clk_tb <= not(clk_tb) and sim_stop after T_CLK/2;
44      arstn_tb <= '1' after T_RESET;
45
46      WRAPPER: apb_wrapper_sync_RAM
47          generic map(
48              DATA_WIDTH_WRAPPER =>DATA_WIDTH_tb,
49              ADDR_WIDTH_WRAPPER =>ADDR_WIDTH_tb
50          )
51
52          port map(
53              w_addr=>address_tb,
54              w_rdata=>out_tb,
55              w_wdata=>in_tb,
56              w_enable => p_en_tb,
57              w_write => we_tb,
58              w_sel => p_select_tb,
59              w_clk => clk_tb,
60              w_rst => arstn_tb
61          );
62
63      STIMULI : process(clk_tb, arstn_tb)
64          variable t : integer := 0;
65      begin
66          if arstn_tb = '0' then
67              t :=0;
68              elsif rising_edge(clk_tb) then
69                  case(t) is
70                      when 2 => address_tb <="1110001"; in_tb <=
    "0000000011111111"; we_tb<='1'; p_select_tb<='1'; -- WRITE
    CYCLE
71                      when 3 => p_en_tb<='1';
72                      when 4 => p_en_tb<='0'; p_select_tb<='0';
73
74                      when 7 => we_tb<='0'; p_select_tb<='1'; --
    READ CYCLE FROM RAM
75                      when 8 => p_en_tb<='1';
76                      when 9 => p_en_tb<='0';
77
```

```
78                      when 12 => address_tb <="0000001";
    p_select_tb<='0'; -- READ CYCLE FROM ROM
79                      when 13 => p_en_tb<='1';
80                      when 14 => p_en_tb<='0';
81
82                      when 16 => address_tb <="1110101"; in_tb <=
    "1111111111111111"; we_tb<='1'; p_select_tb<='0'; -- WRITE IN
    ROM (wrong behaviour)
83                      when 17 => p_en_tb<='1';
84                      when 18 => p_en_tb<='0';
85
86                      when 20 => we_tb<='0'; p_select_tb<='1'; --
    READ CYCLE FROM RAM
87                      when 21 => p_en_tb<='1';
88                      when 22 => p_en_tb<='0';
89
90                      when 23 => address_tb <="1110001";
91
92                      when 25 => we_tb<='0'; p_select_tb<='1'; --
    READ CYCLE FROM RAM WITH ADDR CHANGE (wrong behaviour)
93                      when 26 => p_en_tb<='1';
94                      when 27 => address_tb <="1110101";
95                      when 28 => p_en_tb<='0';
96
97
98                      -- READ CYCLE FOLLOWING A WRITE CYCLE WITH
    ADDR CHANGE AND NO RESET OF VARIABLES (wrong behaviour)
99                      when 30 => address_tb <="1111000"; in_tb <=
    "1111111100000000"; we_tb<='1'; p_select_tb<='1';
100                     when 31 => p_en_tb<='1';
101                     when 32 => address_tb <="1110001";
102                     when 33 => we_tb<='0';
103
104                     when 50 => sim_stop <= '0';
105                     when others => null;
106                 end case;
107                 t := t+1;
108             end if;
109     end process;
110 end architecture;
```