ECEN 5863 Homework Assignment 2

Matt Hartnett

Q1)

This was implemented in VHDL using a behavioral comparison.

```
library ieee;
use ieee.std_logic_1164.all;

entity comparator2 is port (
    A, B: in std_logic_vector(1 downto 0);
    Equals: out std_logic);
end comparator2;

architecture Behavioral of comparator2 is
begin
   equals <= '1' when (A=B) else '0';
end Behavioral;
```

Whenever the two vectors, A and B, are equivalent, the output Equals is 1.

Q2)
Line 4 is missing the keyword port.
Line 5 doesn't declare the direction of a.
Line 7 bit_vector needs to either be 0 to 5 or 5 downto 0, not 5 to 0.
Line 7 requires a semicolon after the port declaration.
Line 10 is missing the keyword "is".
Line 12 requires a sensitivity list for the process statement.
Line 14 has a length mismatch between c and x"F"
Line 15 doesn't accept the type bit_vector for a.
Line 17 requires the binary to be in double quotes.
Line 18 is missing a semicolon.
Line 20 requires a semicolon.

Q3)
Here is the code with the proper corrections listed in Q2. This compiled successfully, indicated that all errors have been fixed. Additionally, the simulation screenshot below shows the correct behavior.

```
library ieee;                                   -- line 1
use ieee.std_logic_1164.all;                    -- line 2
                                                -- line 3
entity find_errors is port (                        -- line 4
    a: in bit_vector(0 to 3);                     -- line 5
    b: out std_logic_vector(3 downto 0);           -- line 6
    c: in bit_vector(5 downto 0));               -- line 7
end find_errors;                                -- line 8
                                                -- line 9
architecture not_good of find_errors is            -- line 10
   begin                                        -- line 11
   my_label: process (a,c)                            -- line 12
      begin                                     -- line 13
      if c = "111111" then                          -- line 14
         b <= to_stdlogicvector(a);                                 -- line 15
      else                                      -- line 16
         b <= "0101";                             -- line 17
      end if;                                   -- line 18
   end process;                                 -- line 19
end not_good;                                   -- line 20
```

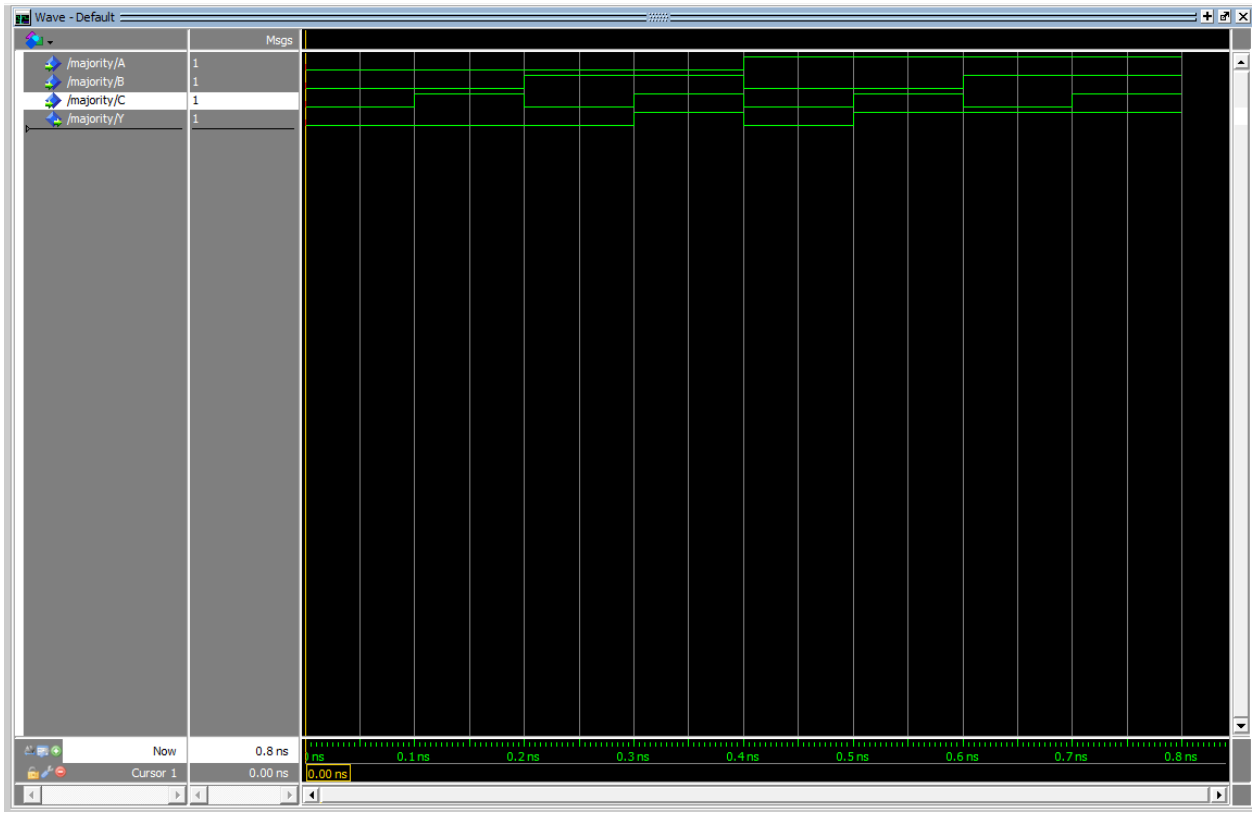
Simulation screenshot:

This simulation shows all of the behavior exhibited by this module. As you can see, when C is ”111111”, b follows the value of a, but when c is not “111111”, b is locked to 0101, regardless of the value of a.

Q4)
The design of my majority vote circuit is very simple, as shown with the logic statement below. By checking all the combinations of A, B, and C, you can very easily compute the majority using only 2 levels of logic.

```
architecture Behavorial of Majority is
begin
    Y <= (A and B) or (A and C) or (B and C);
end Behavorial;
```

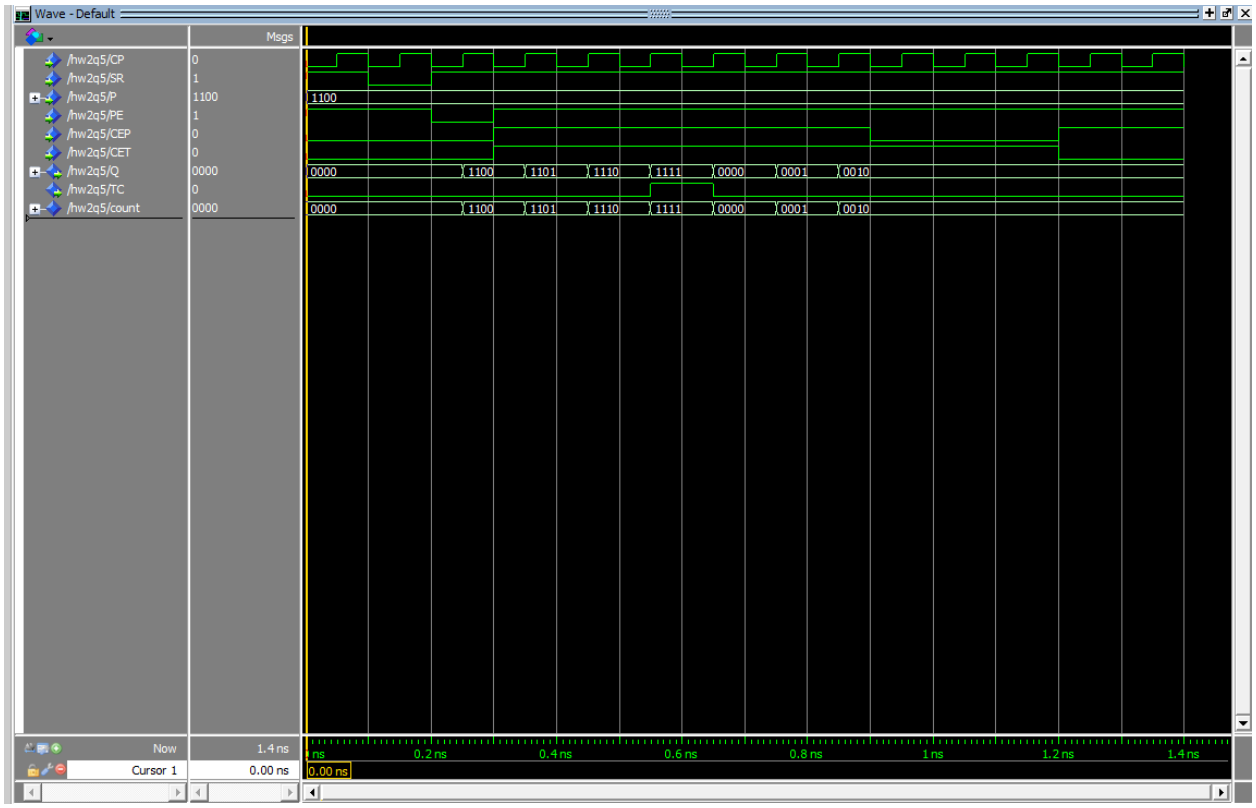
Here is a screenshot of the simulation showing every combination of A, B, and C. As you observe, only when 2 of the 3 inputs are 1 is the output 1.



Q5)
I implemented the counter as an individual module called HW2Q5. This module had the necessary IO as described in the problem statement. Fundamentally, the design works by clocking a counter that was represented by an unsigned 4 bit integer. This integer had a range of 0-15, and had a synchronous reset. The VHDL that described it is shown in the below screenshot. The synchronous reset is handled by the initial if statement within the rising edge that forces the count to 0 when the reset is asserted. This works off of a mux that sets the value of count based on a control vector that is set by the enables. There are three possible states: increment, set, and hold. The advantage of doing it in a single mux as opposed to several overlapping if statements is a lower data path latency, which allows for a higher clock. Since this is not a decadal counter, it can freely run and it automatically loops back once it hits its maximum value. The ripple out is a piece of combination logic that takes every bit of the counter value and ANDs them together, along with the CET, which acts as an enable for the roll over.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_misc.all;

entity HW2Q5 is port (
   CP: in std_logic; -- clock
   SR: in std_logic; -- Active low, synchronous reset
   P: in std_logic_vector(3 downto 0); -- Parallel input
   PE: in std_logic; -- Parallel Enable (Load)
   CEP: in std_logic; -- Count enable parallel input
   CET: in std_logic; -- Count enable trickle input
   Q: out std_logic_vector(3 downto 0);
   TC: out std_logic -- Terminal Count
);
end HW2Q5;

architecture Behavorial of HW2Q5 is
   signal count: unsigned(3 downto 0) := "0000";
   signal control:std_logic_vector(1 downto 0);

begin

   Q <= std_logic_vector(count);
   control(0) <= CEP and CET;
   control(1) <= not PE;

   process (CP)
   begin
      if rising_edge(CP) then
         if( SR = '0') then
            count <= "0000";
         else
            case control is
               when "00" => count <= count;         -- If the enables are off, hold value
               when "01" => count <= count + 1;     -- If enables are on, increment
               when "10" => count <= unsigned(P);   -- If parallel input is on, force value
               when "11" => count <= unsigned(P);   -- If parallel input is on, force value
               when others => count <= count;       -- Default behavior is hold value
            end case;
         end if;
      end if;
   end process;

   TC <= and_reduce(std_logic_vector(count)) and CET;  -- Roll over signal


end Behavorial;
```

Here is a screenshot of the simulation that recreates the timing diagram from the datasheet of the 74LS163. This diagram highlights all of the functions of the design, including the synchronous reset, the data loading, the counting (with the carry out ripple) as well as the two count enables. This timing simulation shows all of the possible combinations of the various inputs (although simplified slightly to reduce complexity).

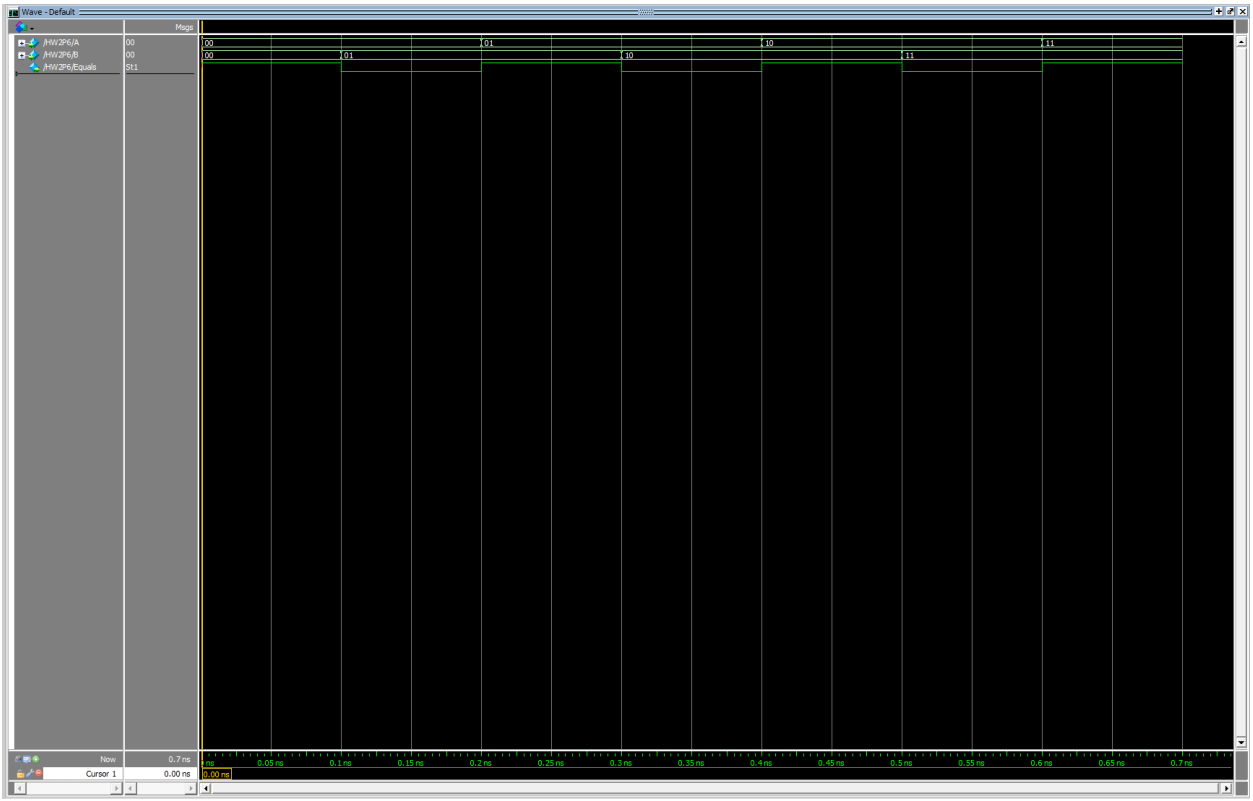


Once compiled, we can see the Fmax of the design, which will allow us to compare the FPGA design with the IC we're trying to replicate. From the datasheet, we can see that the maximum clock is 32MHz for the LS163 model. From the timing analyzer menu, we can see an Fmax of 120.22 MHz for this design. Running this on a Max 10 FPGA certainly has its advantages besides reprogrammability, as we're able to count almost 4 times as quickly.

Q6)
The design for a 2 bit comparator is very simple in Verilog, as shown below. Using a ternary operator, I compared the two vectors with each other and set the output accordingly.

```
module HW2P6(
    input[1:0] A, B,
    output Equals
);

assign Equals = (A == B) ? 1'b1 : 1'b0;


endmodule
```

Here is a screenshot of the simulation that shows the comparator's operation.

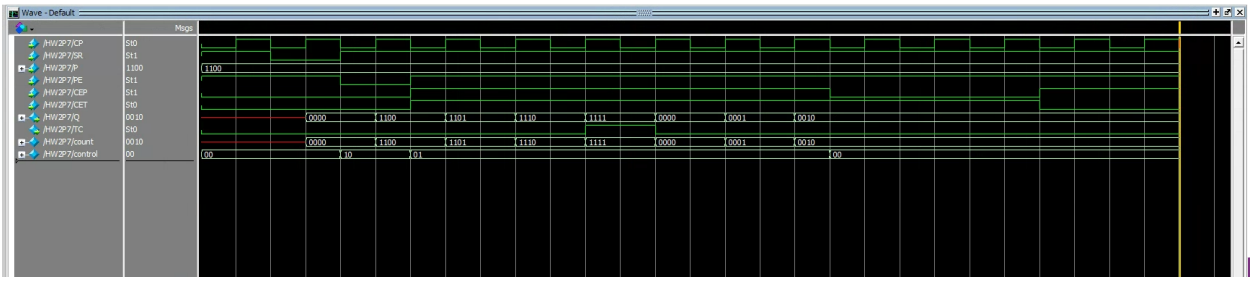


This design doesn't require a clock, so timing analysis isn't very important. There isn't an Fmax to report since a combinational design doesn't require a clock or a frequency. If there was a need to observe an Fmax, it would be possible to do so by putting the output through a flip flop.

Q7)
I designed this module in a very similar fashion to Q5, but obviously in Verilog. I once again used a single mux operated by a control vector to handle all of the operations of the counter. The control vector is determined by the values of the synchronous reset, CEP, CET, and PE using combinational logic. This control vector is used in a case statement to assign the count register to one of its four possible states. As before, the TC output is simply an AND reduce of the Q output vector and the CET signal.

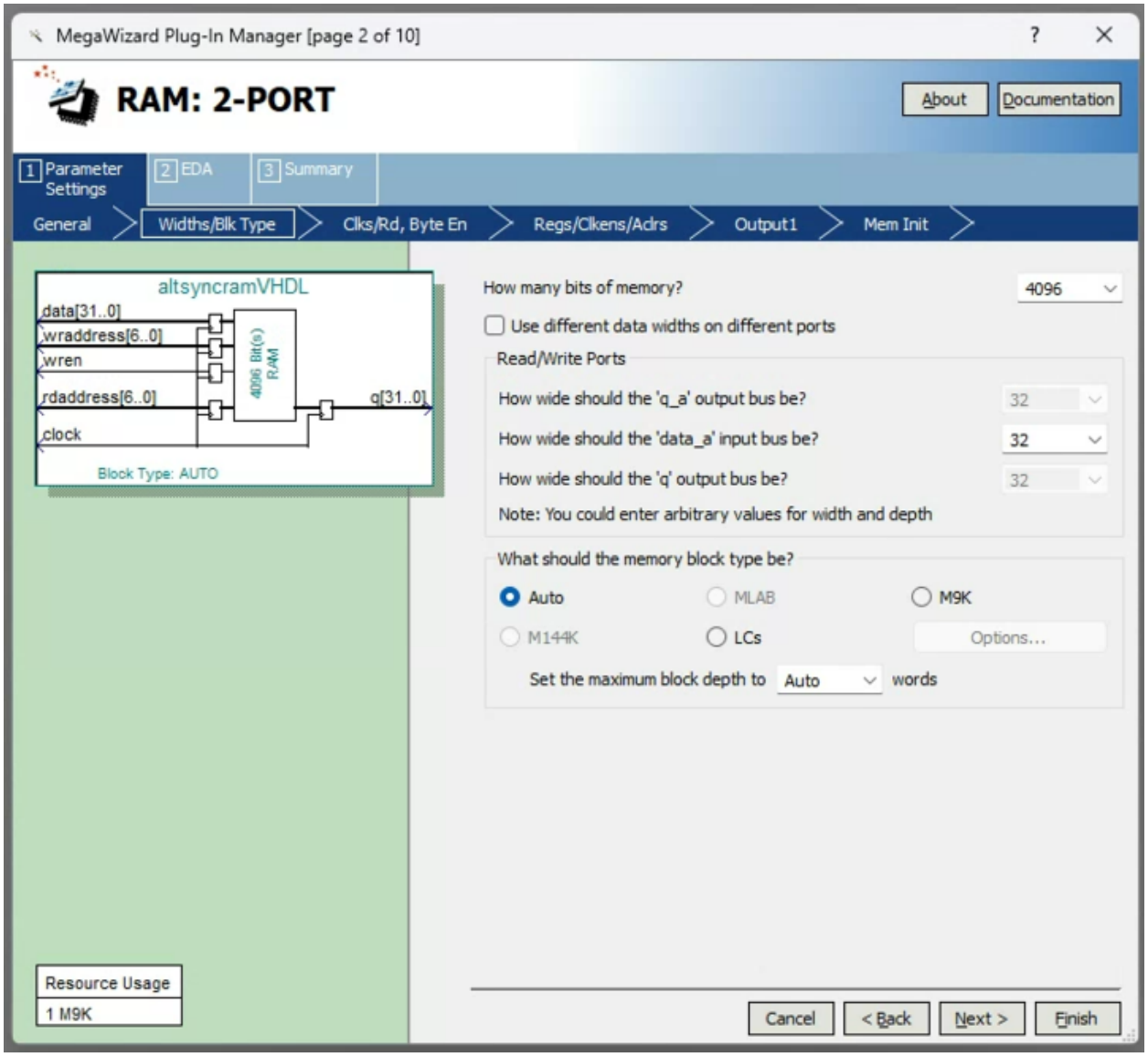
```
module HW2P7 (P, CP, SR, PE, CEP, CET, Q, TC);
   // Define input and output
   input [3:0] P; // Parallel Input
   input CP; // Clock
   input SR; // Active Low Synchronous Reset
   input PE; // Count Enable Parallel Input
   input CEP; // Count Enable Parallel
   input CET; // Count Enable Trickle
   output [3:0]Q; // Parallel Output
   output TC; // Terminal Count

   reg [3:0] count;
   wire [1:0] control;

   assign Q = count;
   assign control[0] = CEP & CET;    // Count enables
   assign control[1] = ~PE;       // Parallel input enable
   assign TC = &Q & CET;


   always @(posedge CP)
   begin
      if( SR == 1'b0) begin // Active low synchronous reset
         count <= 4'b0000;
      end else begin
         case (control)
            3'b00    :  count <= count;       // If both bits are 0, then counting is disabled
            3'b01    :  count <= count + 1;   // If the enables are high and the parallel input is low, count up
            3'b10    :  count <= P;           // If parallel input is high, force in the input
            3'b11    :  count <= P;           // If parallel input is high, force in the input
            default  :  count <= count;       // Disable counting is default behavior
         endcase
      end
   end

endmodule
```

Similarly to before, I recreated the timing diagram from the datasheet of the counter IC. Shown below is a screenshot of that simulation.



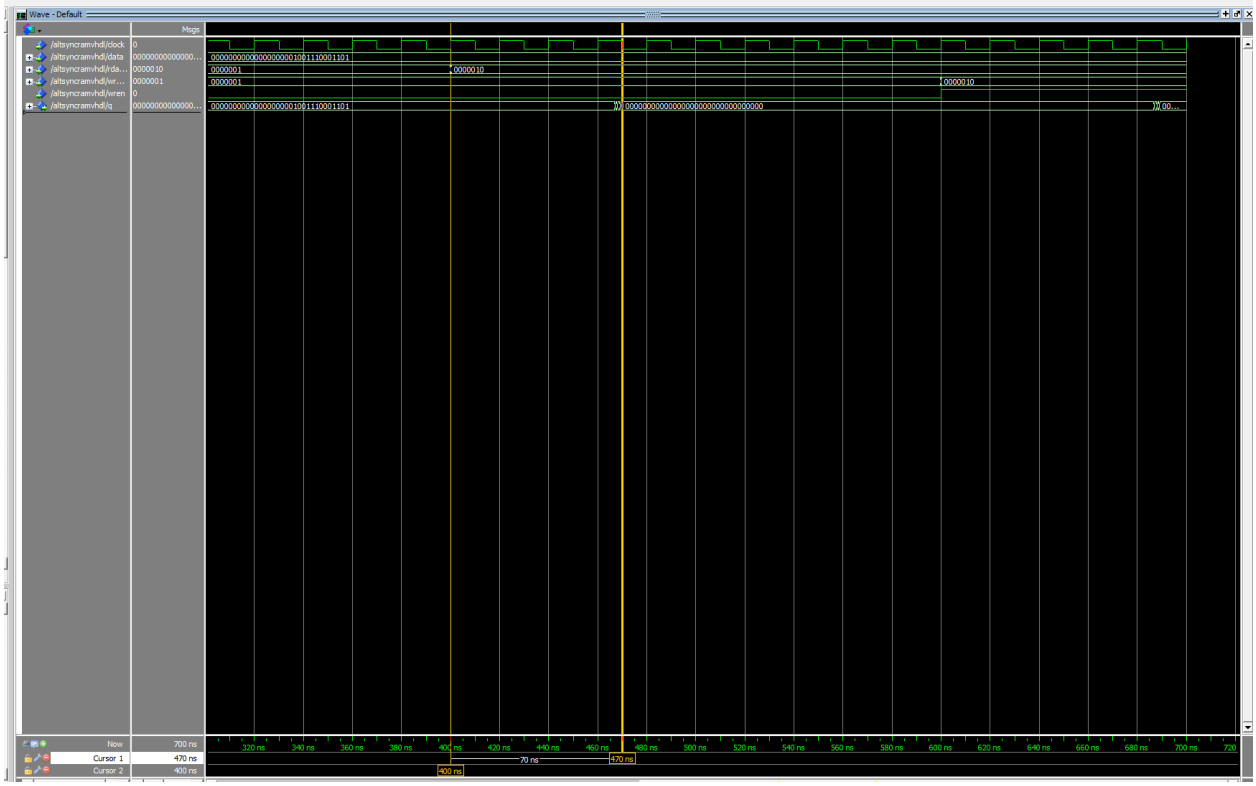
When comparing this to the IC, we can use Timing Analyzer to find that the Fmax is 155.16MHz, which is both substantially faster than the IC that has been emulated, and it's faster than the same design running on the MAX10. This is to be expected, as the family of FPGA is higher in the product tree, costs more, and should (does) run faster than the MAX10.

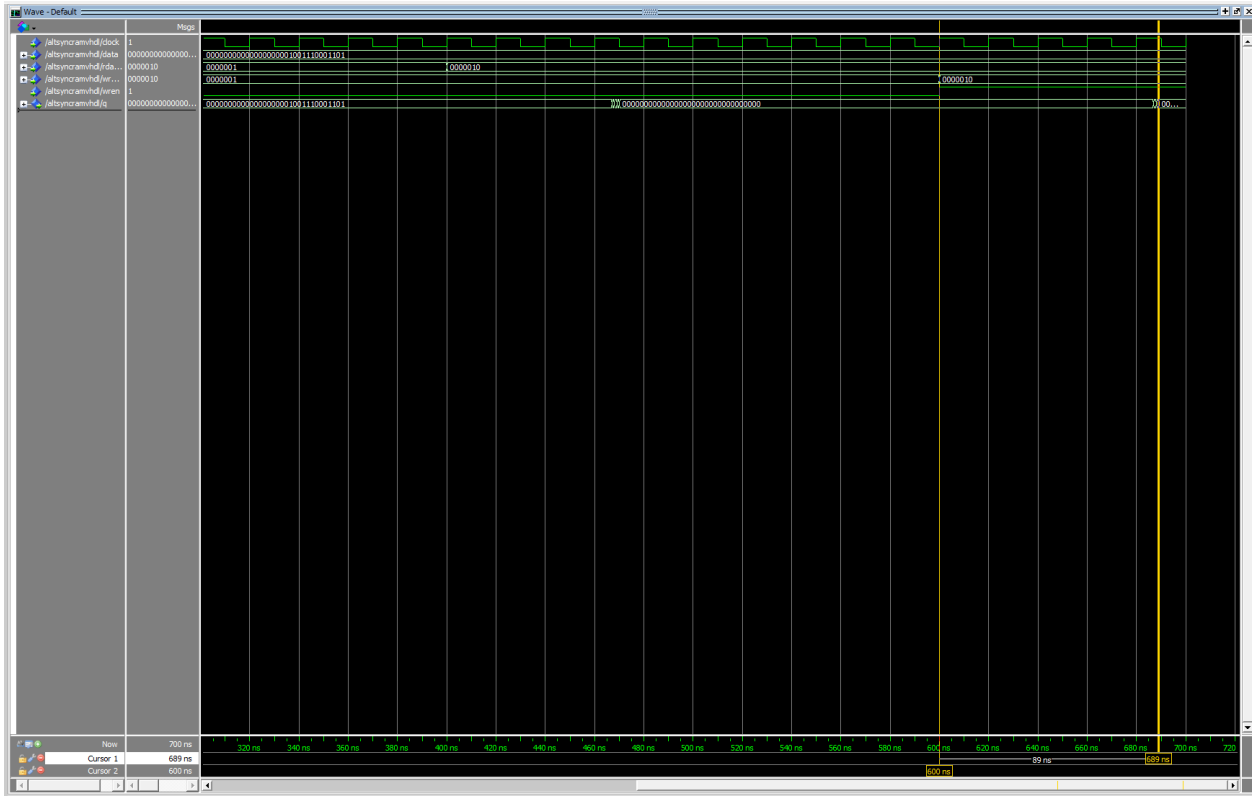
Q8)
In order to implement the altsyncram megafunction, I used the IP Catalog to instantiate a 2-port RAM module. I used the following configuration to get the memory to be the correct size



I also made sure to check the checkbox that generated a netlist so that I could run an RTL simulation for the next step of the problem. Once the VHDL was generated, I set the .qip file as my top level module (for simplicity's sake) and then compiled and ran timing analysis to get an Fmax of 302.66 MHz.

In order to compute the read and write time, I simulated how many clock cycles it would take between a write enable and a read enable to the data arriving.

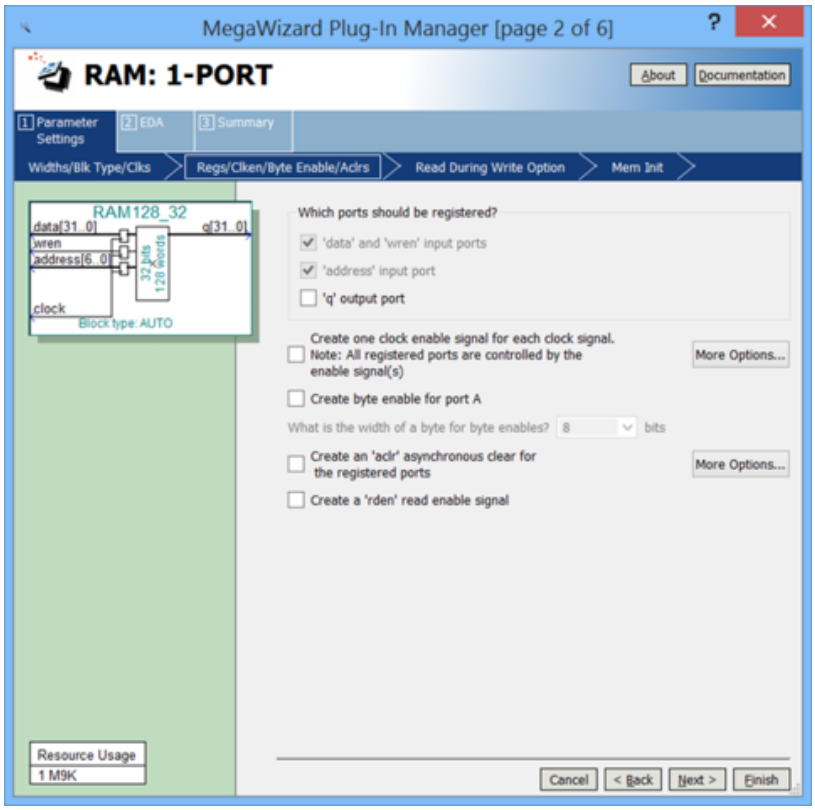


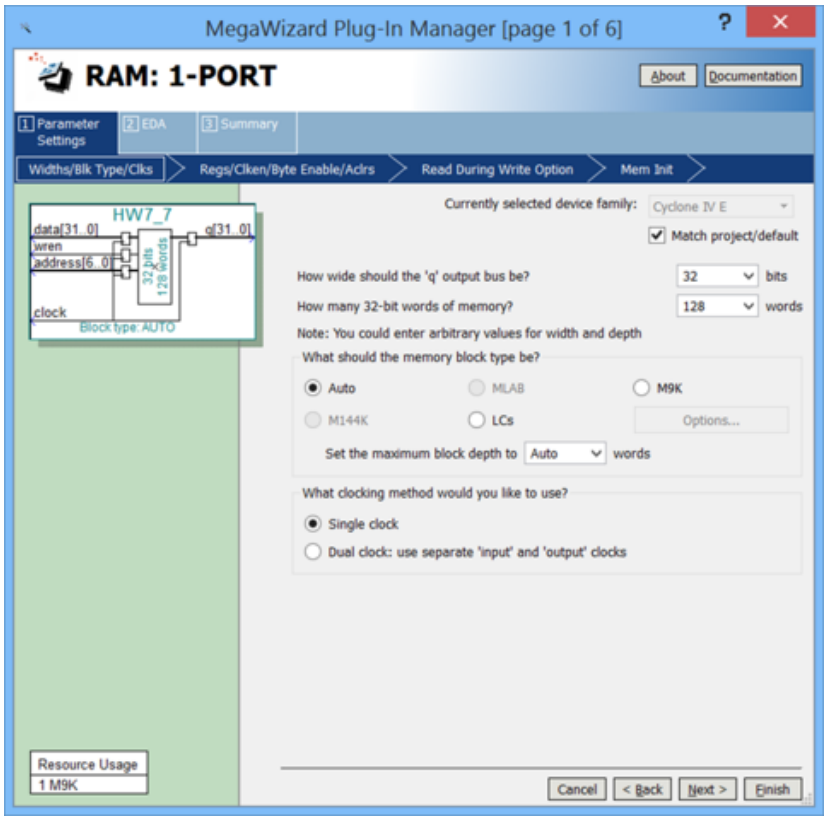


These two simulation screenshots show the time to read and time to read/write respectively. As shown by the first screenshot, a 70ns delay occurs between the read address being changed

and the data showing up on the bus. From the second screenshot, it takes 89ns between the write enable being active and the data showing up on the bus. From this information, we can subtract the two to determine that the read delay is 70ns and the write delay is 19ns. This is using a 50MHz clock signal.

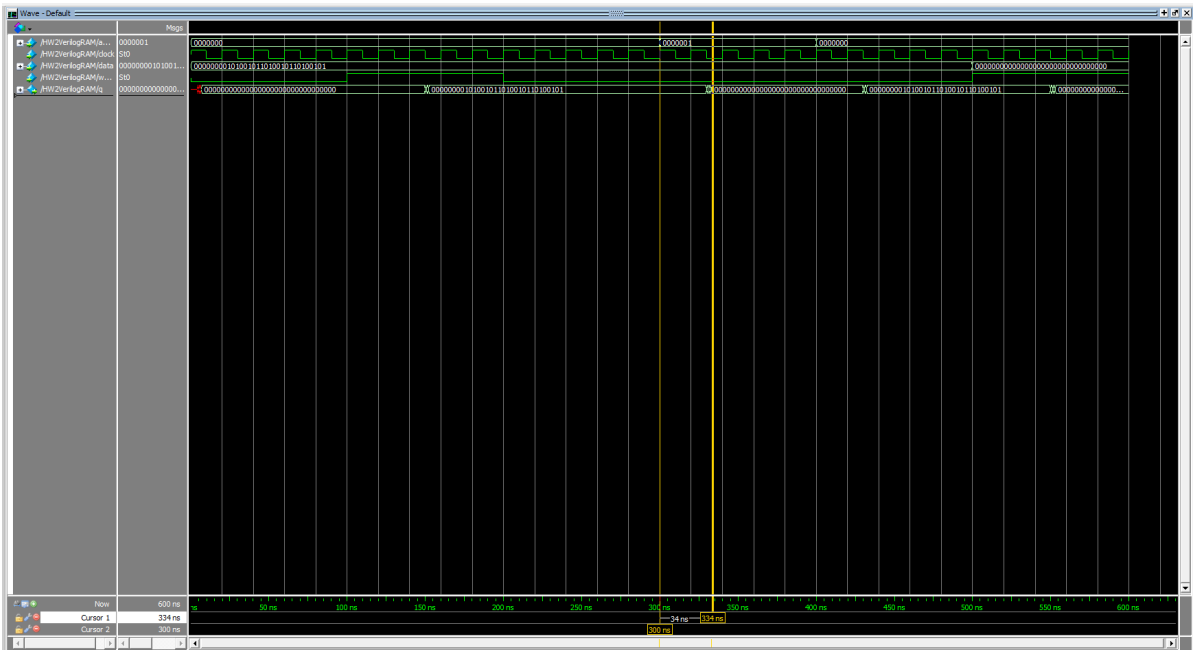
Q9)
For this one, I used the single port RAM, with the following settings, as recommended by Daniel.



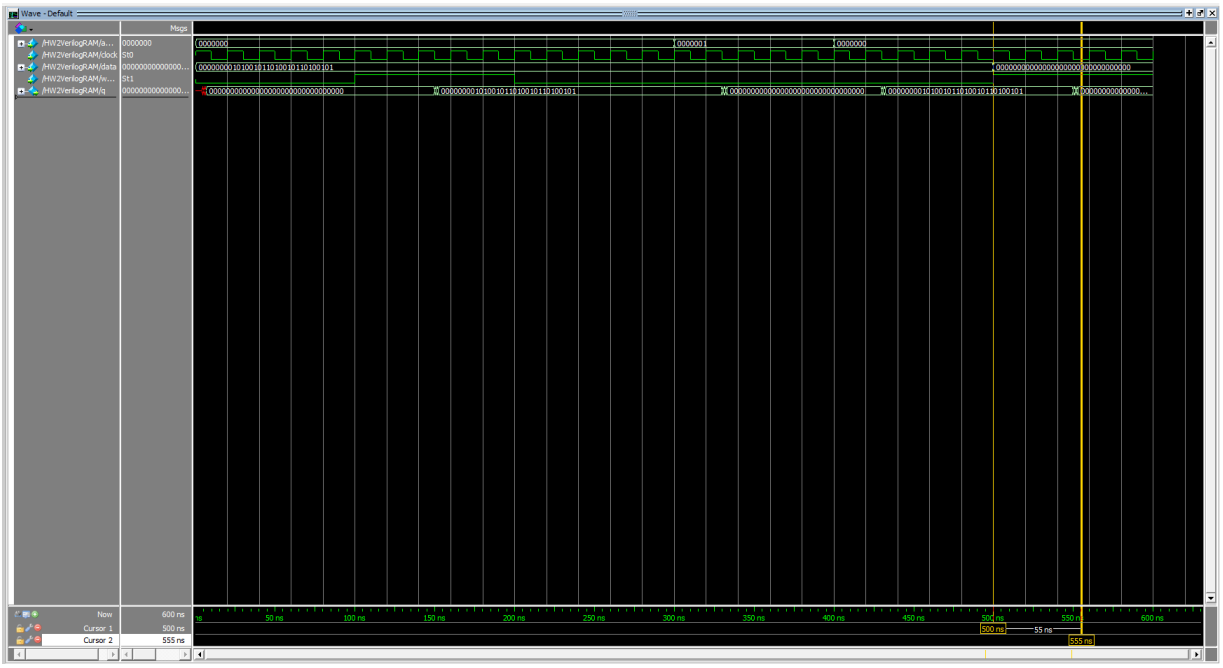


Once the IP was implemented, I set it as the top level module and compiled. I had to create a 50MHz clock constraint to pass the timing analyzer. After compilation, the Fmax listed is: 302.66MHz.

Similarly to question 8, I simulated the design using the gate level simulation to find the read and write delays.



This screenshot shows a 34ns read delay.



This screenshot shows a 55ns read and write delay.

From the two measured delays, we can see that the read delay is 34ns and the write delay is 21 ns.