CMPE 260 Laboratory Exercise 6

SRAM with Built in Memory Controller

Brian Landy
Performed April 20th, 2017
Submitted May 4th, 2017

Lab Section L2
Instructor: Tejaswini Ananthanarayana
TA: Barry Wu

Lecture Section 2
Professor: Richard Cliver

Table of Contents

Abstract

The purpose of this exercise was build an SRAM and that could be written to and have data read from it. This exercise also implemented a state machine so these processes could be handled by the hardware and not the user. This exercise also added a focus on inout signals and tri state buffers. This exercise was constructed and tested successfully. The functionality of the overall IO bus included the ability to read from the SRAM one data element at a time or 4 sequentially. The SRAM could also be written to by changing a read/write bit.

Design Methodology

To implement this design, a wrapper class was used to be the IO bus. All other components were built and ported in structurally. Those include a custom 4-bit tristate buffer, which is used twice, in the design. The purpose of the buffer is to make sure the data line is not altered simultaneously. Data is either being written or read in the machine and the data line must reflect that, as it is used to show data being read or written. The overall IO bus figure is seen in figure 1.

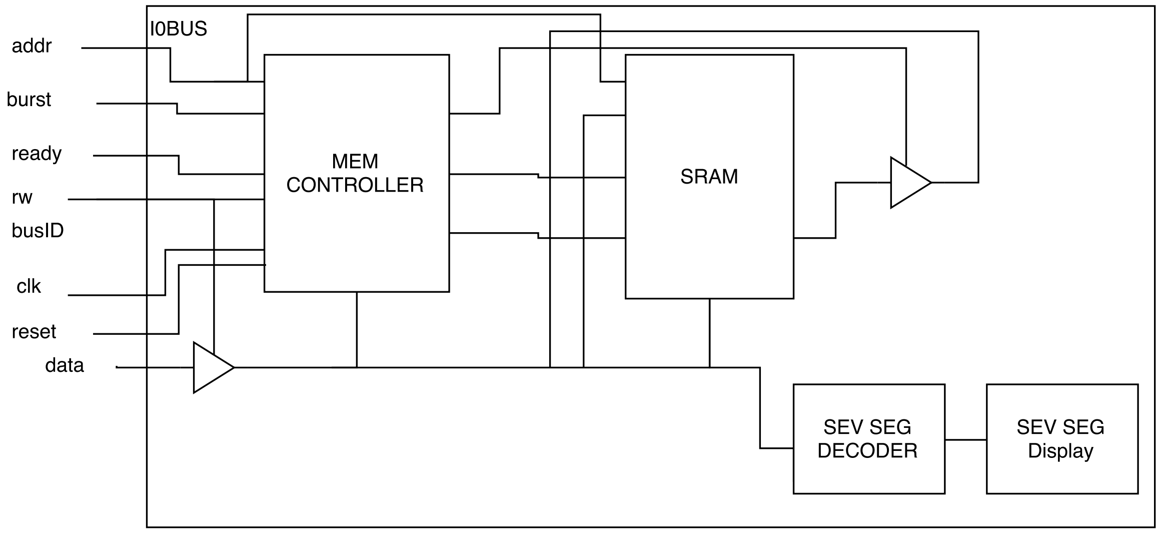


Figure 1 - IOBUS DIAGRAM

The inputs from the overall design flow into the SRAM as well as the memory controller. The purpose of the memory controller is to simply determine whether the memory unit is being written to or read from as well as provide the addresses to do such actions. The SRAM is even simpler, data is stored in an integer array of std logic vectors. The SRAM is written to when RW is equal to 0 and read from when RW is equal to 1.

The memory controller is an important component and functions in a simple fashion. Inside the component, there is a state machine that determines functionality of the SRAM. The state machine diagram that defines the memory controller is seen in figure 2.

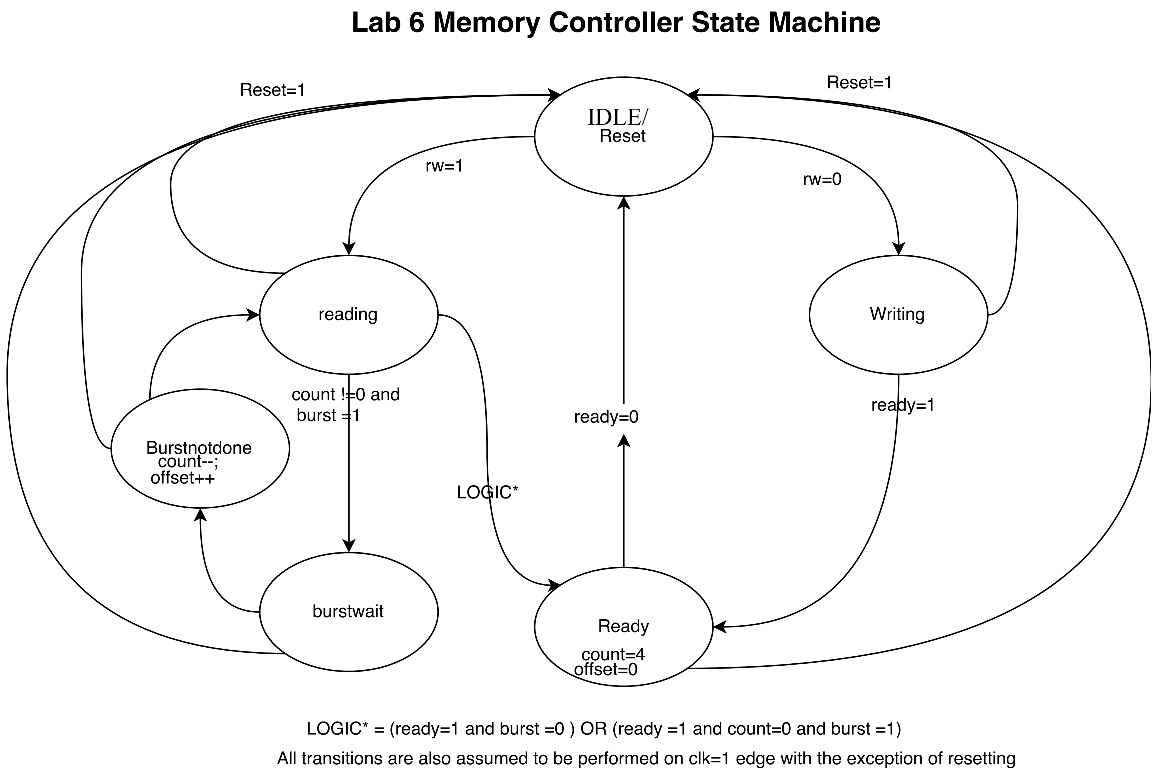

Lab 6 Memory Controller State Machine
Reset=1
Reset=1
IDLE/
Reset
rw=1
rw=0
reading
Writing
count !=0 and
burst =1
ready=0
ready=1
Burstnotdone
count--;
offset++
LOGIC*
burstwait
Ready
count=4
offset=0
LOGIC* = (ready=1 and burst =0) OR (ready =1 and count=0 and burst =1)
All transitions are also assumed to be performed on clk=1 edge with the exception of resetting

Figure 2 – SRAM Memory Controller State Machine

In figure 2, the memory controller state machine is present. Starting at the reset/Idle state the machine will start to read or write depending on the RW bit input. When the machine is in the write state, the oe is 0 and we is 1 because it will tell the SRAM to receive and store information. The machine will go to the ready state when the user is ready (when they enter ready =1). Then upon turning the ready bit back off, the machine will go back to the idle state and continue the process. So the ready state acts as a wait state. Alternatively, when RW=1, the machine will start to read from memory. The oe is set to 1 and we is 0. Provided burst is 0, the machine will move to the ready state as soon as ready goes high. If burst is high, functionality is different. If burst is high, then the burst count gets set to 4 and the machine loops through the burst not done branch instead for 3 times. Then when the last read is performed, the count is now at 0 and ready is the only input that determines whether the machine goes to the next state. To do this, there is a "burstnotdone" state that increments the offset to read the next address and decrements the burst count to keep track of the cycle. The burst wait will just continue in the satte it is in until ready is set to 0. That way ready will always have to be set to one for the reading state to move on and it will not just blow past it extremely fast and cause issues. With all of this in place, the machine functions properly and correctly writes to the SRAM. The machine was also written with attention and care. Since it will be simulated on an fpga, each internal signal had its own process and only one signal was touched in a specific process. They also used minimal logic per process. All transitions in the machine take place assuming it is the rising edge of a clock and reset is 0.

Results

The circuit was implemented ideally and needed testing. A test bench was set up with a handful of simple tests. The rest needed testing, the machine needed to be written to, read from and read from in burst format. The first 2 tests were the read and write. The machine was written to and read from in reverse order of the registers written to. The figure for this test is shown in figure 3.

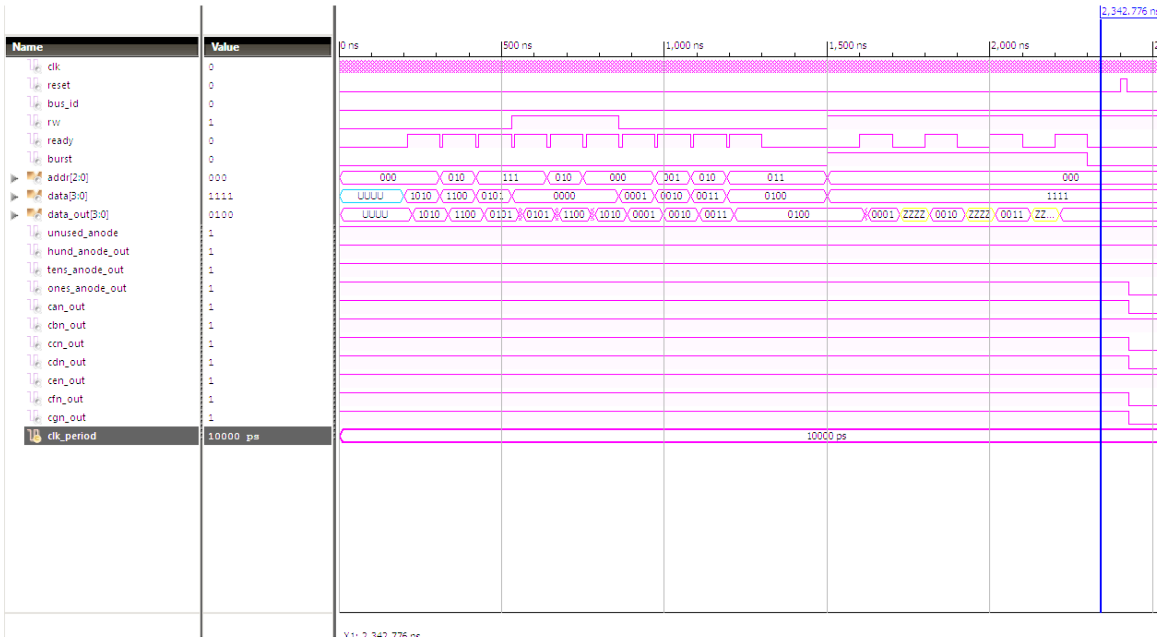


Figure 3 – All Test Cases

In figure 3, from 0 to about 510 ns, the machine is being written to. 3 unique values are being written to the first, last and a middle position. Then, they are being read from the SRAM in revers from about 510 to 850 ns. This demonstrates the correctness of reading and writing. Then, from 850 to 1300 ns, the machine is being written to ascending positions for the purpose of a burst read. The values are also in an ascending order. The burst read is then performed and all values are seen in the proper order, demonstrating the correctness of the burst read. This takes place at 1500ns. At the end of the simulation, a reset is performed. This is seen by looking to the right of the cursor. The peak goes high.

After successfully testing the circuit, the post route simulation was run and had similar results. The proper expected values can be observed at specific point where they are supposed to occur.

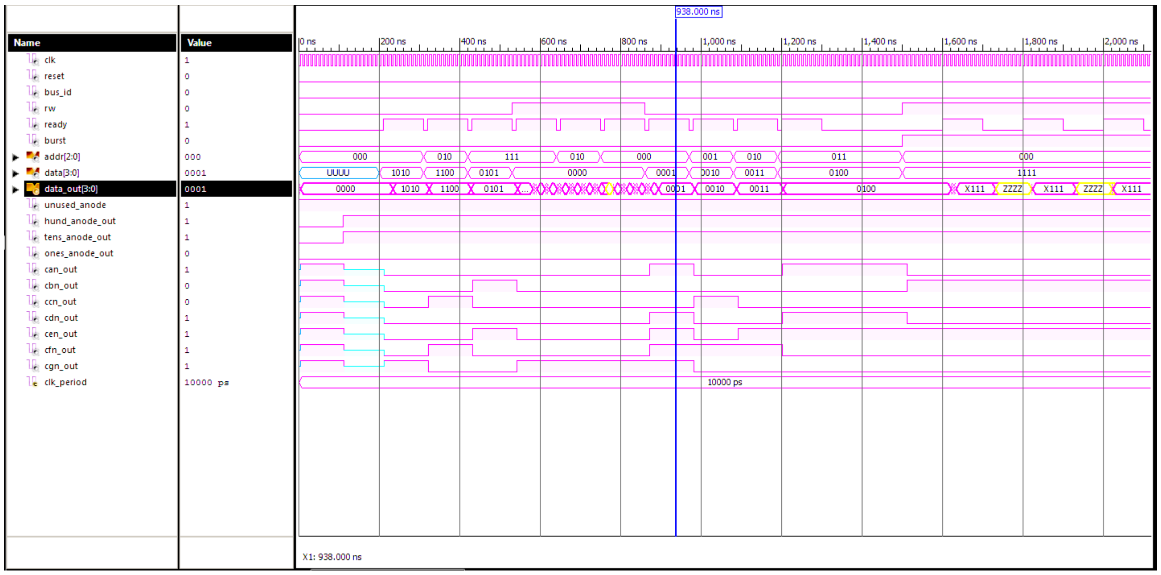


Figure 4 - read and write post route

After checking reading and writing, the burst was simulated. This was observed in figure 5.

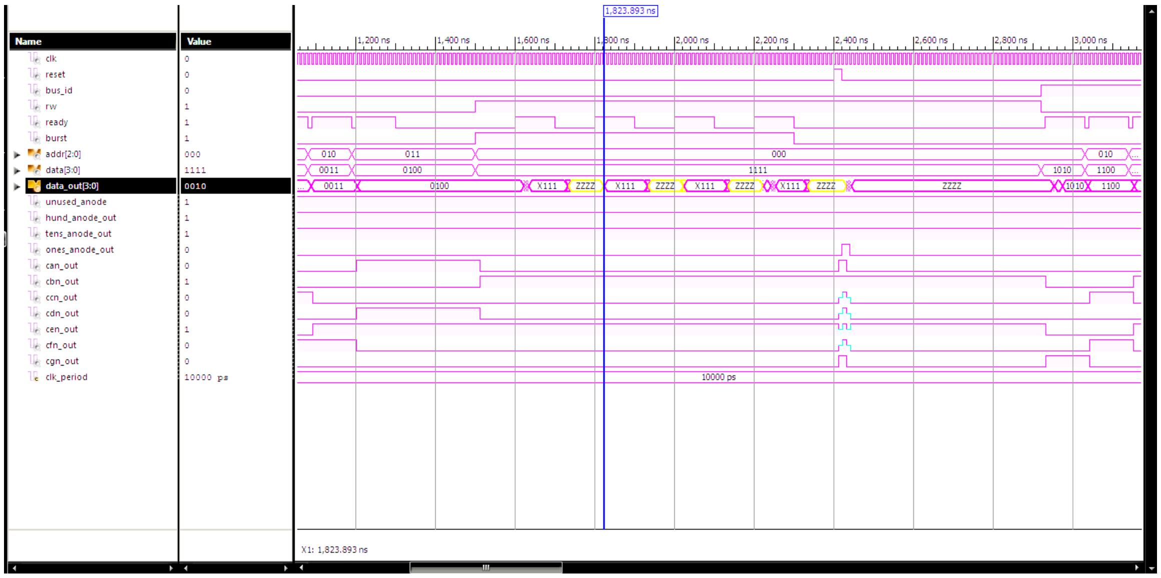


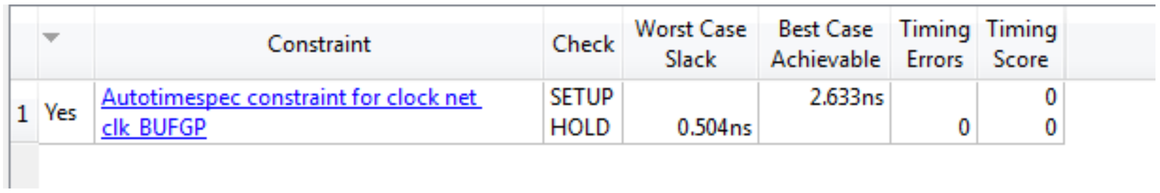
Figure 5 – post route burst / reset

The reset test case can also be observed at this point.

This should run properly on an FPGA provided all anodes are correct, which they are.

This circuit successfully simulates all tests for correctness and functioned on the Nexys 3 Board.

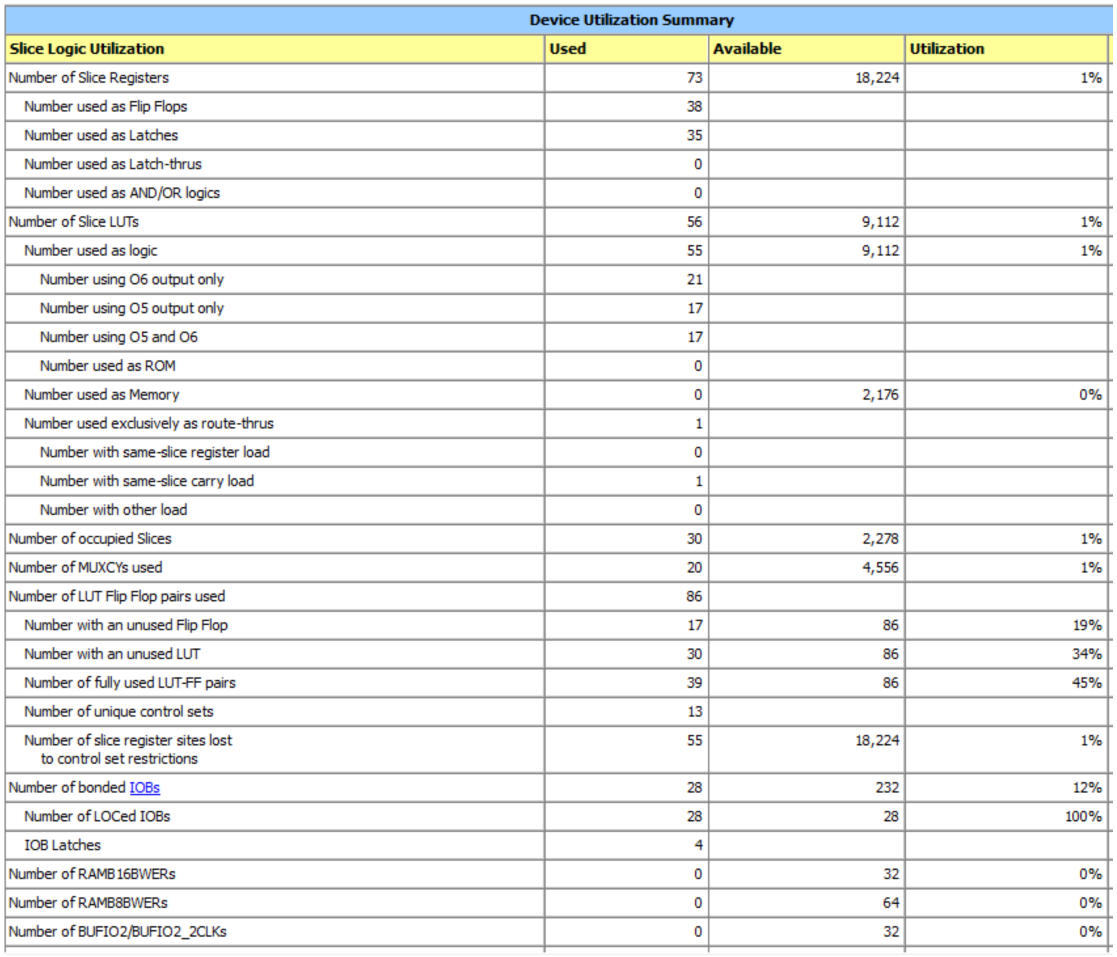
Figure 6 is the timing analysis for this circuit.



		Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
1	Yes	Autotimespec constraint for clock net clk_BUFGP	SETUP		2.633ns		0
			HOLD	0.504ns		0	0

Figure 6 – Timing results

As seen in figure 6, the best case achievable for this circuit is 2.633 nanoseconds. Figure 7 holds all of the simulation slice size results provided by Xilinx.



Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	73	18,224	1%
Number used as Flip Flops	38		
Number used as Latches	35		
Number used as Latch-thrus	0		
Number used as AND/OR logics	0		
Number of Slice LUTs	56	9,112	1%
Number used as logic	55	9,112	1%
Number using O6 output only	21		
Number using O5 output only	17		
Number using O5 and O6	17		
Number used as ROM	0		
Number used as Memory	0	2,176	0%
Number used exclusively as route-thrus	1		
Number with same-slice register load	0		
Number with same-slice carry load	1		
Number with other load	0		
Number of occupied Slices	30	2,278	1%
Number of MUXCYs used	20	4,556	1%
Number of LUT Flip Flop pairs used	86		
Number with an unused Flip Flop	17	86	19%
Number with an unused LUT	30	86	34%
Number of fully used LUT-FF pairs	39	86	45%
Number of unique control sets	13		
Number of slice register sites lost to control set restrictions	55	18,224	1%
Number of bonded IOBs	28	232	12%
Number of LOCed IOBs	28	28	100%
IOB Latches	4		
Number of RAMB16BWERs	0	32	0%
Number of RAMB8BWERs	0	64	0%
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%

Figure 9 – slice and area results

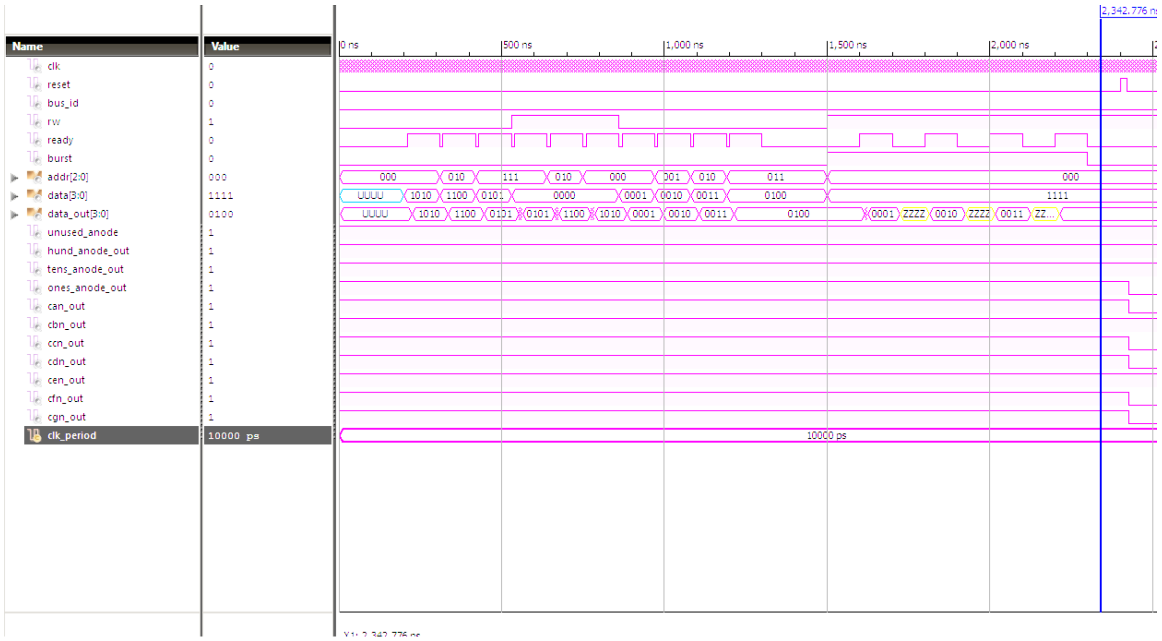
Overall, this circuit uses 38 flip flops total with 56 total LUTS used. These results all demonstrate the functional correctness of this exercise and circuit.

Conclusions

This exercise was a success in general. It demonstrated the importance understanding SRAM devices and using state machines to control the process. This combination of components ultimately led to a system that read and write values to memory. This circuit shows that there is a nice solution to storing information. This is an extremely important concept when it comes to digital systems. And this exercise demonstrated that.

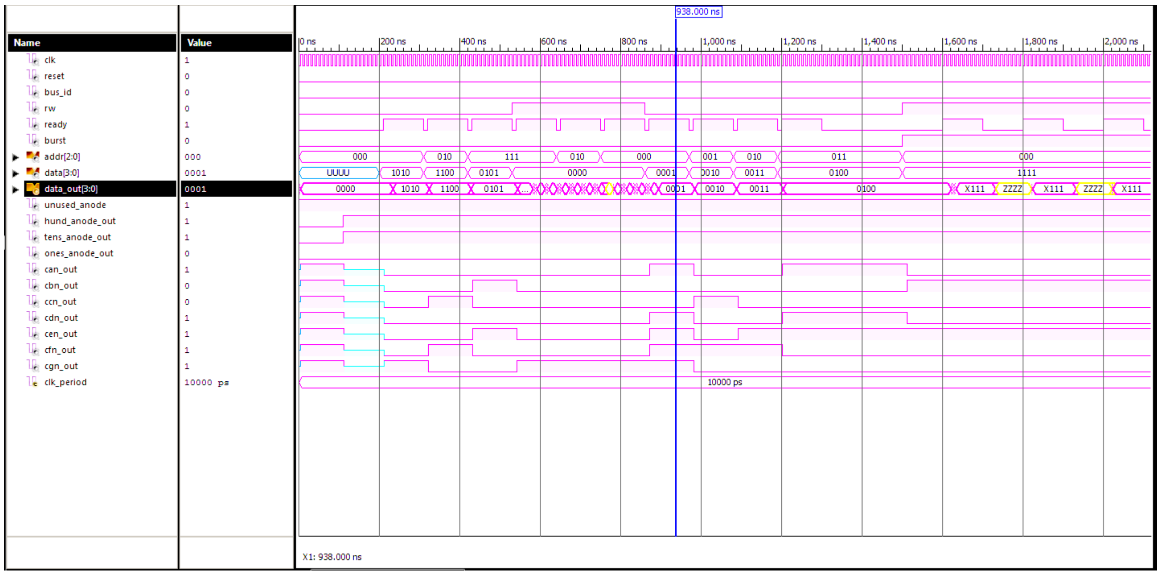
Appendix

Waveforms



Name	Value
clk	0
reset	0
bus_id	0
rw	1
ready	0
burst	0
addr[2:0]	000
data[3:0]	1111
data_out[3:0]	0100
unused_anode	1
hund_anode_out	1
tens_anode_out	1
ones_anode_out	1
can_out	1
cbn_out	1
ccn_out	1
cdn_out	1
cen_out	1
cfn_out	1
cgn_out	1
clk_period	10000 ps

Figure 3 – All Test Cases



Name	Value
clk	1
reset	0
bus_id	0
rw	0
ready	1
burst	0
addr[2:0]	000
data[3:0]	0001
data_out[3:0]	0001
unused_anode	1
hund_anode_out	1
tens_anode_out	1
ones_anode_out	0
can_out	1
cbn_out	0
ccn_out	0
cdn_out	1
cen_out	1
cfn_out	1
cgn_out	1
clk_period	10000 ps

Figure 4 - read and write post route

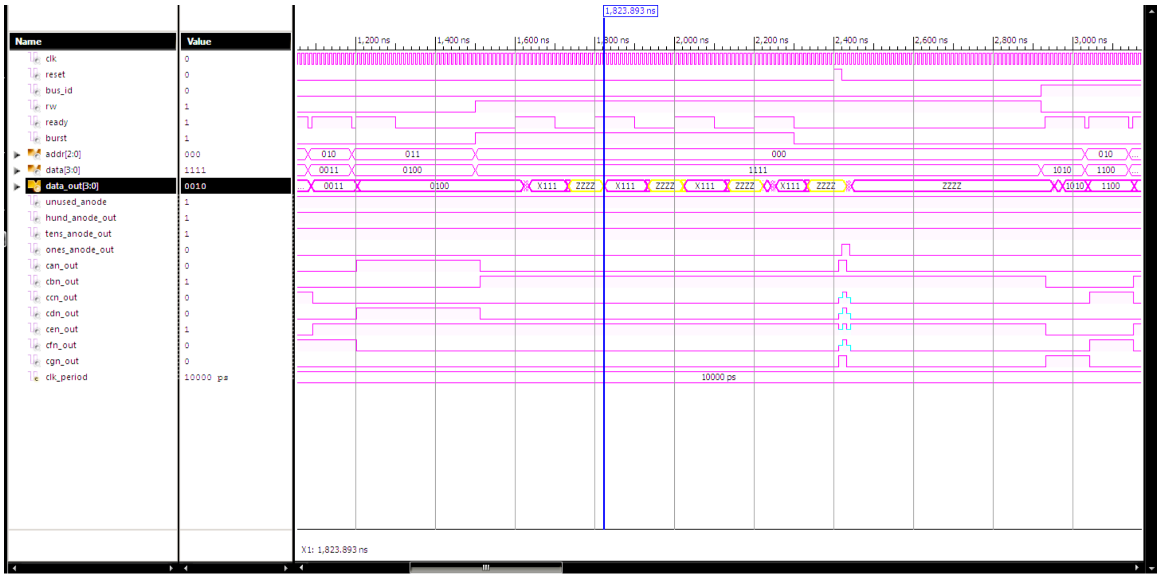


Figure 5 – post route burst / reset

Code

TRISTATE.VHD

```
entity tristate is
Port ( inputport    : in  STD_LOGIC_vector(3 downto 0);
          enable   : in  STD_LOGIC;
          outputport    : out STD_LOGIC_vector(3 downto 0)
                                          );
end tristate;

architecture Behavioral of tristate is

begin

outputport <= inputport when (enable = '0') else "ZZZZ";
end Behavioral;
```

SRAM.VHD

```
entity SRAM is
   Port ( oe : in  STD_LOGIC;
          we : in  STD_LOGIC;
          addr : in  STD_LOGIC_VECTOR (2 downto 0);
          data_in : in  STD_LOGIC_VECTOR (3 downto 0);
                                          bus_id : in std_logic;
          data_out : out  STD_LOGIC_VECTOR (3 downto 0));
end SRAM;

architecture Behavioral of SRAM is
--size (2^3)-1 = 8-1
type mem_data_type is array (7 downto 0) of std_logic_vector(3 downto 0);

signal mem_data: mem_data_type;
signal oe_we: std_logic_vector(1 downto 0);
signal data_temp : std_logic_vector(3 downto 0);


begin

              oe_we<=oe & we;
```

```
        process(oe_we,mem_data,addr,data_in,data_temp,bus_id) is begin
        --if bus_id='0' then

                case(oe_we) is
                        when "10" => data_temp<=mem_data(to_integer(unsigned(addr)));
                        when "01" => data_temp<=data_in;
                        when others => data_temp<=data_temp;
                end case;
        --end if;
        end process;


        data_out<=data_temp;

        process(oe_we, addr,mem_data,data_in,bus_id) is begin
        --if bus_id='0' then
        if oe_we = "01" then
                mem_data(to_integer(unsigned(addr))) <= data_in;
        end if;
--end if;
        end process;
end Behavioral;
```

SEVEN SEGMENT DECODER.VHD

```
entity seven_segment_decoder is
   Port ( bcd_Vec : in  STD_LOGIC_VECTOR(11 downto 0);
        hund_disp_n : out STD_LOGIC_VECTOR (6 downto 0);
        tens_disp_n : out  STD_LOGIC_VECTOR (6 downto 0);
        ones_disp_n : out   STD_LOGIC_VECTOR (6 downto 0));
end seven_segment_decoder;

architecture Behavioral of seven_segment_decoder is
signal topThree : STD_LOGIC_VECTOR(3 downto 0);
signal midThree : STD_LOGIC_VECTOR(3 downto 0);
signal botThree : STD_LOGIC_VECTOR(3 downto 0);



begin


topThree<=bcd_Vec(11 downto 8);
midThree <= bcd_Vec(7 downto 4);
botThree<=bcd_Vec(3 downto 0);



process (topThree)
BEGIN

case  topThree is
when "0000"=> hund_disp_n <="0000001";  -- '0'
when "0001"=> hund_disp_n <="1001111";  -- '1'
when "0010"=> hund_disp_n <="0010010";  -- '2'
when "0011"=> hund_disp_n <="0000110";  -- '3'
when "0100"=> hund_disp_n <="1001100";  -- '4'
when "0101"=> hund_disp_n <="0100100";  -- '5'
when "0110"=> hund_disp_n <="0100000";  -- '6'
when "0111"=> hund_disp_n <="0001111";  -- '7'
when "1000"=> hund_disp_n <="0000000";  -- '8'
when "1001"=> hund_disp_n <="0000100";  -- '9'
 --nothing is displayed when a number more than 9 is given as input.
when others=> hund_disp_n <="1111111";
end case;


end process;
```

```
process (midThree)
BEGIN

case  midThree is
when "0000"=> tens_disp_n <="0000001";  -- '0'
when "0001"=> tens_disp_n <="1001111";  -- '1'
when "0010"=> tens_disp_n <="0010010";  -- '2'
when "0011"=> tens_disp_n <="0000110";  -- '3'
when "0100"=> tens_disp_n <="1001100";  -- '4'
when "0101"=> tens_disp_n <="0100100";  -- '5'
when "0110"=> tens_disp_n <="0100000";  -- '6'
when "0111"=> tens_disp_n <="0001111";  -- '7'
when "1000"=> tens_disp_n <="0000000";  -- '8'
when "1001"=> tens_disp_n <="0000100";  -- '9'
 --nothing is displayed when a number more than 9 is given as input.
when others=> tens_disp_n <="1111111";
end case;


end process;



process (botThree)
BEGIN

case  botThree is
when "0000"=> ones_disp_n <="0000001";  -- '0'
when "0001"=> ones_disp_n <="1001111";  -- '1'
when "0010"=> ones_disp_n <="0010010";  -- '2'
when "0011"=> ones_disp_n <="0000110";  -- '3'
when "0100"=> ones_disp_n <="1001100";  -- '4'
when "0101"=> ones_disp_n <="0100100";  -- '5'
when "0110"=> ones_disp_n <="0100000";  -- '6'
when "0111"=> ones_disp_n <="0001111";  -- '7'
when "1000"=> ones_disp_n <="0000000";  -- '8'
when "1001"=> ones_disp_n <="0000100";  -- '9'
 --nothing is displayed when a number more than 9 is given as input.
when others=> ones_disp_n <="1111111";
end case;


end process;
end Behavioral;
```

SEVENSEGMENTDISPLAY.VHD

```
entity seven_seg_disp is
   Port ( hund_disp_n  : in  STD_LOGIC_VECTOR (6 downto 0);
                      tens_disp_n  : in  STD_LOGIC_VECTOR (6 downto 0);
           ones_disp_n  : in  STD_LOGIC_VECTOR (6 downto 0);
                      clk          : in  STD_LOGIC; -- digilent board generated clock
                                                reset_n      : in  STD_LOGIC; -- switch input
                                                unused_anode : out STD_LOGIC; -- unused an3
                                                hund_anode   : out STD_LOGIC; -- digilent an2
                      tens_anode   : out STD_LOGIC; -- digilent an3
                      ones_anode   : out STD_LOGIC; -- digilent an4
                                                CAn,CBn,CCn,CDn,CEn,CFn,CGn : out STD_LOGIC); -- digilent cathode - used for all
displays
end seven_seg_disp;

architecture Behavioral of seven_seg_disp is
-- Clock can be set at 50 or 100MHz clock
-- This counter rolls over when max value is achieved
--   if 50MHz then 7FFFF(hex) * (1/50MHz) and the time for the counter to roll over (period) is 10.49ms.
--     Each segment is on for 1/4 of the full period or 2.62ms.
--   if 100MHz then 7FFFF(hex) * (1/100MHz) and the time for the counter to roll over (period) is 5.25ms.
--     Each segment is on for 1/4 of the full period or 1.31ms.
-- Digilent Nexys 3 Board Reference Manual states "In order for each of the four digits to appear bright
--   and continuously illuminated, all four digits should be driven once every 1 to 16 ms,..."
-- This design refreshes all four digits every 10.49ms or 5.25ms, which is within the 1 to 16ms spec when the clock is 50 or 100
MHz.

         constant max_cnt_size : integer := 19; --19
```

```
		signal counter : STD_LOGIC_VECTOR (max_cnt_size-1 downto 0);
begin

  unused_anode <= '1';

  mux_disp: process (clk) begin
		if (clk'event and clk = '1') then
		  if (reset_n = '1') then
		      ones_anode <= '1';
		      tens_anode <= '1';
						hund_anode <= '1';
				CAn <= '1';
				CBn <= '1';
				CCn <= '1';
				CDn <= '1';
				CEn <= '1';
				CFn <= '1';
				CGn <= '1';
     else
						case counter(max_cnt_size-1 downto max_cnt_size-2) is
						  when "00" =>		-- display ones
								ones_anode <= '0';
								tens_anode <= '1';
								hund_anode <= '1';
								CAn <= ones_disp_n(6);
								CBn <= ones_disp_n(5);
								CCn <= ones_disp_n(4);
								CDn <= ones_disp_n(3);
								CEn <= ones_disp_n(2);
								CFn <= ones_disp_n(1);
								CGn <= ones_disp_n(0);
						  when "01" => -- display tens
								ones_anode <= '1';
								tens_anode <= '0';
								hund_anode <= '1';
								CAn <= tens_disp_n(6);
								CBn <= tens_disp_n(5);
								CCn <= tens_disp_n(4);
								CDn <= tens_disp_n(3);
								CEn <= tens_disp_n(2);
								CFn <= tens_disp_n(1);
								CGn <= tens_disp_n(0);
						  when "10" => -- display hundreds
								ones_anode <= '1';
								tens_anode <= '1';
								hund_anode <= '0';
								CAn <= hund_disp_n(6);
								CBn <= hund_disp_n(5);
								CCn <= hund_disp_n(4);
								CDn <= hund_disp_n(3);
								CEn <= hund_disp_n(2);
								CFn <= hund_disp_n(1);
								CGn <= hund_disp_n(0);
						  when others => -- blank display
								ones_anode <= '1';
								tens_anode <= '1';
								hund_anode <= '1';
								CAn <= '1';
								CBn <= '1';
								CCn <= '1';
								CDn <= '1';
								CEn <= '1';
								CFn <= '1';
								CGn <= '1';
						end case;
		  end if;
		end if;
  end process mux_disp;

  div_down_cntr : process (clk) begin
		if (clk'event and clk = '1') then
		  if (reset_n = '1') then
		    counter <= (OTHERS => '0');
		  else
		    counter <= counter + '1';
```

```
                end if;
               end if;
  end process div_down_cntr;

end Behavioral;
```

2:1 mux

```
entity mux21 is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Sel : in  STD_LOGIC;
           output : out  STD_LOGIC_VECTOR (3 downto 0));
end mux21;

architecture Behavioral of mux21 is

begin
output <= A when (Sel = '0') else B;

end Behavioral;
```

MEMCONTROLLER.VHD

```
entity memController is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           bus_id : in  STD_LOGIC;
           rw : in  STD_LOGIC;
           ready : in  STD_LOGIC;
           burst : in  STD_LOGIC;
           addr : in  STD_LOGIC_VECTOR (2 downto 0);
           oe : out  STD_LOGIC;
           we : out  STD_LOGIC;
           offset : out  STD_LOGIC_VECTOR (1 downto 0));
end memController;

architecture Behavioral of memController is
type state_type is (idle, writing, readystate, reading, burstnotdone,burstwait);
        --make state type
        signal state, next_state : state_type;

        signal burstcnt : Integer range 0 to 3;
        signal offset_internal : Integer range 0 to 3;
        --std_logic_vector(1 downto 0);
begin
--upon not resetting, state will assume the next state
        SYNC_PROC : process(clk, reset)
        begin
                if(clk'event and clk='1') then
                        if (reset = '1') then
                                state<=idle;
                        else
                                state<= next_state;
                        end if;
                end if;
        end process;


--move between states based on conditions
next_state_decode : process(clk, reset, state, next_state, bus_id, rw, ready, burst, addr,burstcnt)
begin

                --next_state<=state;
                case (state) is

                        when (idle) =>
                        if ready='1' then
                                if rw='0' then
                                        next_state<=writing;

                                elsif rw='1' then
                                        next_state<=reading;
                                end if;
                        end if;
```

```
            when (writing) =>
                if ready = '1' then
                    next_state<=readystate;
                elsif ready ='0' then
                    next_state<=writing;
                end if;

            when (reading)=>
                if burst ='0' then
                --normal read
                    if ready = '1' then
                        next_state<=readystate;
                    elsif ready ='0' then
                        next_state<=reading;
                    end if;

                elsif burst='1' then
                    if ready = '1' AND burstcnt=0 then
                        next_state<=readystate;
                    elsif ready ='1' AND burstcnt>0 then
                        next_state<=burstwait;
                    elsif ready='0' then
                        next_state<=reading;
                        end if;
                end if;

                when(burstwait)=>
                    if ready='0' then

                    --next_state<=burstwait;   else

                    next_state<=burstnotdone;
                    end if;


                when(burstnotdone) =>
                    next_state<=reading;

                when(readystate) =>
                if ready = '0' then
                    next_state<=idle;
                    end if;


            when others =>
            --should not reach this case, but if it goes here unexpectedly, ill edit this to diagnose.
            next_state<=idle;
            end case;


end process;

oe_proc : process(clk,reset) begin
if(clk'event and clk='1') then
    if reset ='0' then
        case (state) is
            when (reading) =>
            oe<='1';
            when others =>
            oe<='0';
            end case;
            end if;
            end if;

end process;

we_proc : process(clk,reset) begin
if(clk'event and clk='1') then
    if reset ='0' then
        case (state) is
            when (writing) =>
            we<='1';
            when others =>
            we<='0';
            end case;
```

```
                                        end if;
                                        end if;

end process;

offset_proc : process(clk,reset) begin
if(clk'event and clk='1') then
                if reset ='0' then
                        case (state) is
                                when (burstnotdone) =>

                                offset_internal<=offset_internal+1;

                                when (idle)=>
                                offset_internal<=0;

                                when others =>


                                end case;
                                end if;
                                end if;

end process;

burstcntproc_proc : process(clk,reset) begin
if(clk'event and clk='1') then
                if reset ='0' then
                        case (state) is
                                when (burstnotdone) =>
                                burstcnt<=burstcnt-1;

                                when (idle) =>
                                burstcnt<=3;


                                when others =>

                                end case;
                                end if;
                                end if;

end process;

offset<=std_logic_vector(to_unsigned(offset_internal,2));



end Behavioral;
```

IOBUS.VHD

```
entity IoBus is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           bus_id : in  STD_LOGIC;
           rw : in  STD_LOGIC;
           ready : in  STD_LOGIC;
           burst : in  STD_LOGIC;
           addr : in  STD_LOGIC_VECTOR (2 downto 0);


           data : in  STD_LOGIC_VECTOR (3 downto 0);

                                         data_out : out  STD_LOGIC_VECTOR (3 downto 0);

                                          unused_anode      : out STD_LOGIC; -- unused an3
                                         hund_anode_out    : out STD_LOGIC; -- digilent an2
                                         tens_anode_out    : out STD_LOGIC; -- digilent an3
           ones_anode_out    : out STD_LOGIC; -- digilent an4
           CAn_out           : out STD_LOGIC;
                                         CBn_out           : out STD_LOGIC;
                                         CCn_out           : out STD_LOGIC;
                               CDn_out           : out STD_LOGIC;
```

```
                                                    CEn_out           : out STD_LOGIC;
                                                    CFn_out           : out STD_LOGIC;
                                                    CGn_out            : out STD_LOGIC);
end IoBus;

architecture Behavioral of IoBus is

component tristate is
Port ( inputport    : in  STD_LOGIC_vector(3 downto 0);
          enable   : in  STD_LOGIC;
          outputport    : out STD_LOGIC_vector(3 downto 0)
                                                    );
end component;

component mux21 is
   Port ( A : in  STD_LOGIC_VECTOR(3 downto 0);
          B : in  STD_LOGIC_VECTOR(3 downto 0);
          Sel : in  STD_LOGIC;
          output : out  STD_LOGIC_VECTOR(3 downto 0));
end component;

component SRAM is
                 Port ( oe : in  STD_LOGIC;
          we : in  STD_LOGIC;
          addr : in  STD_LOGIC_VECTOR (2 downto 0);
          data_in : in  STD_LOGIC_VECTOR (3 downto 0);
                                                     bus_id : in std_logic;
          data_out : out  STD_LOGIC_VECTOR (3 downto 0));
                                                   end component;

component memController is
   Port ( clk : in  STD_LOGIC;
          reset : in  STD_LOGIC;
          bus_id : in  STD_LOGIC;
          rw : in  STD_LOGIC;
          ready : in  STD_LOGIC;
          burst : in  STD_LOGIC;
          addr : in  STD_LOGIC_VECTOR (2 downto 0);
          oe : out  STD_LOGIC;
          we : out  STD_LOGIC;
          offset : out  STD_LOGIC_VECTOR (1 downto 0));
end component;

component seven_seg_disp is
   Port ( hund_disp_n  : in  STD_LOGIC_VECTOR (6 downto 0);
                        tens_disp_n  : in  STD_LOGIC_VECTOR (6 downto 0);
          ones_disp_n  : in  STD_LOGIC_VECTOR (6 downto 0);
                        clk          : in  STD_LOGIC; -- digilent board generated clock
                                                    reset_n      : in  STD_LOGIC; -- switch input
                                                    unused_anode : out STD_LOGIC; -- unused an3
                                                    hund_anode   : out STD_LOGIC; -- digilent an2
                        tens_anode   : out STD_LOGIC; -- digilent an3
                        ones_anode   : out STD_LOGIC; -- digilent an4
                                                    CAn,CBn,CCn,CDn,CEn,CFn,CGn : out STD_LOGIC); -- digilent cathode - used for all displays
end component;

component seven_segment_decoder is
   Port ( bcd_Vec : in  STD_LOGIC_VECTOR(11 downto 0);
          hund_disp_n : out STD_LOGIC_VECTOR (6 downto 0);
          tens_disp_n : out  STD_LOGIC_VECTOR (6 downto 0);
          ones_disp_n : out   STD_LOGIC_VECTOR (6 downto 0));
end component;

Signal displayvec : std_logic_vector(3 downto 0);

Signal ones_disp_n_wire : std_logic_vector(6 downto 0);

Signal tens_disp_n_wire : std_logic_vector(6 downto 0);

Signal hund_disp_n_wire : std_logic_vector(6 downto 0);

signal boi : std_logic_vector(11 downto 0);


signal tri_dat_in : std_logic_vector(3 downto 0);
signal tri_mem_dat_out : std_logic_vector(3 downto 0);

signal mem_oe,mem_we : std_logic;

signal offset_int : std_logic_vector(1 downto 0);

signal SRAM_out_int : std_logic_vector(3 downto 0);


signal SRAM_out_int2 : std_logic_vector(3 downto 0);

signal mux_data_out : std_logic_vector(3 downto 0);

--signal addrandoffset : std_logic_vector(2 downto 0);

begin

instance0_mem_cont : memController
port map(clk=>clk,reset=>reset,bus_id
=>bus_id,rw=>rw,ready=>ready,burst=>burst,addr=>addr,oe=>mem_oe,we=>mem_we,offset=>offset_int);


--buss id 0 sram

instance0_sram : SRAM
port map(oe => mem_oe, we=>mem_we, addr=>std_logic_vector( unsigned(addr) + unsigned(offset_int)),data_in=>tri_dat_in,
bus_id=>bus_id, data_out=>SRAM_out_int);

--bus id 1 sram

--instance1_sram : SRAM
--port map(oe => mem_oe, we=>mem_we, addr=>std_logic_vector( unsigned(addr) + unsigned(offset_int)),data_in=>tri_dat_in,
bus_id=>NOT bus_id,data_out=>SRAM_out_int2);



--instanceselectbus_mux : mux21
--port map (A=>SRAM_out_int,B=>SRAM_out_int2,Sel=>bus_id,output=>mux_data_out);

instance0_tri : tristate
port map (inputport=>data, enable=>rw,outputport=>tri_dat_in);

instance1_tri : tristate
port map (inputport=>SRAM_out_int, enable=>mem_oe,outputport=>data_out);



instance0_mux : mux21
port map (A=>data,B=> data,Sel=>rw,output=>displayvec);

boi<="00000000"&displayvec; --displayvec is 4 bits - data out






seven_seg0 : seven_segment_decoder
port map     ( bcd_Vec => bin_to_bcd(boi),
           hund_disp_n =>hund_disp_n_wire,
           tens_disp_n =>tens_disp_n_wire,
           ones_disp_n =>ones_disp_n_wire);

instance0_seven_seg_dsip : seven_seg_disp
port map(  hund_disp_n       => hund_disp_n_wire,
   tens_disp_n       => tens_disp_n_wire,
   ones_disp_n       => ones_disp_n_wire,
   clk               => clk,
   reset_n           => reset,
   unused_anode      => unused_anode,
   hund_anode        => hund_anode_out,
   tens_anode        => tens_anode_out,
   ones_anode        => ones_anode_out,
  CAn              => CAn_out,
               CBn              => CBn_out,
               CCn              => CCn_out,
               CDn              => CDn_out,
               CEn              => CEn_out,
               CFn              => CFn_out,
               CGn              => CGn_out
               );

end Behavioral;
```

IOTB.VHD

```
ENTITY IO_tb IS
END IO_tb;

ARCHITECTURE behavior OF IO_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT IoBus
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
         bus_id : IN  std_logic;
         rw : IN  std_logic;
         ready : IN  std_logic;
         burst : IN  std_logic;
         addr : IN  std_logic_vector(2 downto 0);
         data : IN  std_logic_vector(3 downto 0);
         data_out : OUT  std_logic_vector(3 downto 0);
                                             unused_anode : OUT  std_logic;
         hund_anode_out : OUT  std_logic;
         tens_anode_out : OUT  std_logic;
         ones_anode_out : OUT  std_logic;
         CAn_out : OUT  std_logic;
         CBn_out : OUT  std_logic;
         CCn_out : OUT  std_logic;
         CDn_out : OUT  std_logic;
         CEn_out : OUT  std_logic;
         CFn_out : OUT  std_logic;
         CGn_out : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
   signal reset : std_logic := '0';
   signal bus_id : std_logic := '0';
   signal rw : std_logic := '0';
   signal ready : std_logic := '0';
   signal burst : std_logic := '0';
   signal addr : std_logic_vector(2 downto 0) := (others => '0');

               --BiDirs
   signal data : std_logic_vector(3 downto 0);
  signal data_out : std_logic_vector(3 downto 0);

               --Outputs
   signal unused_anode : std_logic;
   signal hund_anode_out : std_logic;
   signal tens_anode_out : std_logic;
   signal ones_anode_out : std_logic;
   signal CAn_out : std_logic;
   signal CBn_out : std_logic;
   signal CCn_out : std_logic;
   signal CDn_out : std_logic;
   signal CEn_out : std_logic;
   signal CFn_out : std_logic;
   signal CGn_out : std_logic;

   -- Clock period definitions
   constant clk_period : time := 10 ns;
```

```
BEGIN

                -- Instantiate the Unit Under Test (UUT)
  uut: IoBus PORT MAP (
        clk => clk,
        reset => reset,
        bus_id => bus_id,
        rw => rw,
        ready => ready,
        burst => burst,
        addr => addr,
        data => data,

                                                data_out=> data_out,

        unused_anode => unused_anode,
        hund_anode_out => hund_anode_out,
        tens_anode_out => tens_anode_out,
        ones_anode_out => ones_anode_out,
        CAn_out => CAn_out,
        CBn_out => CBn_out,
        CCn_out => CCn_out,
        CDn_out => CDn_out,
        CEn_out => CEn_out,
        CFn_out => CFn_out,
        CGn_out => CGn_out
      );

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


  -- Stimulus process
  stim_proc: process
  begin
    -- hold reset state for 100 ns.
    wait for 100 ns;

    wait for clk_period*10;

                        bus_id<='0';
                        rw<='0';
                        addr<="000";
                        data<="1010";
                        burst<='0';
                        wait for 10ns;
                        ready<='1';
                        wait for 100ns;
                        ready<='0';

                        bus_id<='0';
                        rw<='0';
                        addr<="010";
                        data<="1100";
                        burst<='0';
                        wait for 10ns;
                        ready<='1';
                        wait for 100ns;
                        ready<='0';

                        bus_id<='0';
                        rw<='0';
                        addr<="111";
                        data<="0101";
                        burst<='0';
                        wait for 10ns;
                        ready<='1';
                        wait for 100ns;
```

```
ready<='0';

----------readding now

bus_id<='0';
rw<='1';
addr<="111";
data<="0000";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='0';
rw<='1';
addr<="010";
data<="0000";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='0';
rw<='1';
addr<="000";
data<="0000";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';
-----------prep for burst by writing registers full
bus_id<='0';
rw<='0';
addr<="000";
data<="0001";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='0';
rw<='0';
addr<="001";
data<="0010";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='0';
rw<='0';
addr<="010";
data<="0011";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='0';
rw<='0';
addr<="011";
data<="0100";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

------------test burst read from 0
```

```
wait for 200ns;
bus_id<='0';
rw<='1';
addr<="000";
data<="1111";
burst<='1';

--read first
wait for 100ns;
ready<='1';

wait for 100ns;
ready<='0';


--read second
wait for 100ns;
ready<='1';

wait for 100ns;
ready<='0';

--read third
wait for 100ns;
ready<='1';

wait for 100ns;
ready<='0';

--read 4th
wait for 100ns;
ready<='1';

wait for 100ns;
ready<='0';

---------------done
burst<='0';
wait for 100ns;
reset<='1';
wait for 20ns;
reset<='0';
wait for 500ns;
------------------------------------------------------------------------------------------
--write second sram
bus_id<='1';
rw<='0';
addr<="000";
data<="1010";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='1';
rw<='0';
addr<="010";
data<="1100";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='1';
rw<='0';
addr<="111";
data<="0101";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';
```

```
--------read now
bus_id<='1';
rw<='1';
addr<="111";
data<="0000";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='1';
rw<='1';
addr<="010";
data<="0000";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='1';
rw<='1';
addr<="000";
data<="0000";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';
-----------prep for burst by writing registers full
bus_id<='1';
wait for 100ns;
rw<='0';
addr<="000";
data<="0001";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='1';
rw<='0';
addr<="001";
data<="0010";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='1';
rw<='0';
addr<="010";
data<="0011";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

bus_id<='1';
rw<='0';
addr<="011";
data<="0100";
burst<='0';
wait for 10ns;
ready<='1';
wait for 100ns;
ready<='0';

------------test burst read from 0
wait for 200ns;
bus_id<='1';
```

```
                        rw<='1';
                        addr<="000";
                        data<="1111";
                        burst<='1';

                        --read first
                        wait for 100ns;
                        ready<='1';

                        wait for 100ns;
                        ready<='0';


                        --read second
                        wait for 100ns;
                        ready<='1';

                        wait for 100ns;
                        ready<='0';

                        --read third
                        wait for 100ns;
                        ready<='1';

                        wait for 100ns;
                        ready<='0';

                        --read 4th
                        wait for 100ns;
                        ready<='1';

                        wait for 100ns;
                        ready<='0';

                        ----------done
                        burst<='0';

      -- insert stimulus here

      wait;
   end process;

END;
```

BIN_BCD.VHD

```
use IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.ALL;
use ieee.numeric_std.all;

package bin_bcd is

FUNCTION bin_to_bcd (bin :std_logic_vector(11 DOWNTO 0)) return std_logic_vector;


end bin_bcd;

package body bin_bcd is

FUNCTION Bin_to_BCD (bin :std_logic_vector(11 DOWNTO 0)) return std_logic_vector IS
-- if 100's, 10's, 1's >= 0101
   -- add 3
-- shift left 1
-- if 10 shifts, done
                variable i : integer:=0;
                variable bcd : std_logic_vector(23 downto 0);
                variable bint : std_logic_vector(11 DOWNTO 0) := bin;
                BEGIN
                        bcd := (others => '0');
                        bcd(11 downto 0) := bint;
                        for i in 0 to 11 loop
                                -- Left shift one bit
                                bcd(23 downto 0) := bcd(22 downto 0) & '0';
                                -- Check phase
                                if(i < 11 and bcd(15 downto 12) > "0100") then
                                        bcd(15 downto 12) := bcd(15 downto 12) + "0011";
                                end if;
                                if(i < 11 and bcd(19 downto 16) > "0100") then
                                        bcd(19 downto 16) := bcd(19 downto 16) + "0011";
                                end if;
                                if(i < 11 and bcd(23 downto 20) > "0100") then
                                        bcd(23 downto 20) := bcd(23 downto 20) + "0011";
                                end if;
                        end loop;
                        RETURN BCD(23 downto 12);
                END FUNCTION;

end bin_bcd;
```

Lab Report Demo Sheet

Name: Brian Landy Lab 06

Laboratory Sign Off	
Section	**Signature**
Pre-Lab	4/20 [illegible]
Simulation	[illegible] [illegible] 5/4
Code Critique	[illegible] [illegible] 5/4
Post – Route Simulation	[illegible] 5/4 [illegible]
Working Board	5/4 [illegible] [illegible]

Extra Credit: [illegible] 5/5