Lab 2: Static Random-Access Memory (SRAM)

Brandon Chin

CSC343 - Instructor: Prof. Izidor Gertner

3/16/2015

Objective

We must design a static random-access memory chip (SRAM) using D-Latches. The SRAM chip must be able to store 4-bit strings of information at any of its 16 address locations (16x4 SRAM). We must then display the contents of our memory device in the 7-segment HEX displays on the DE2 Circuit Board.

Functionality and Specifications

An SRAM device is composed of multiple address locations which can each store strings of information. There are two operations that the SRAM can perform, read and write. We can first specify the address location, then we can choose to write to the address (this will overwrite any existing information that may have previously been stored at the location), or we can read from the location (this will display the contents of the address through the 7-segment HEX display on the DE2 Board).

In order to design the SRAM, a few components must be designed first:

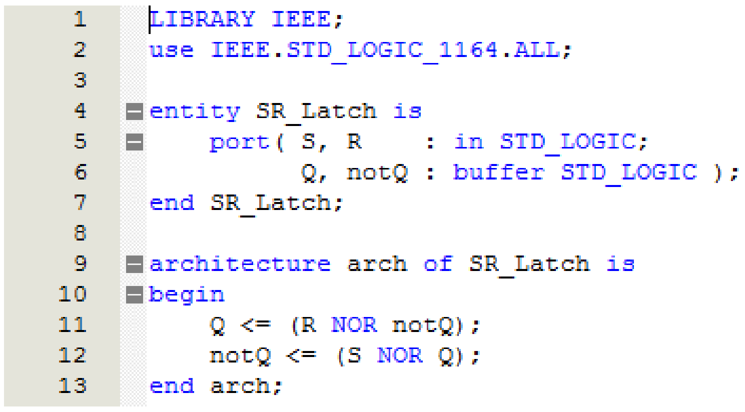
Latches: A very basic storage element which can maintain a binary state until it is signaled to change states. Latches store one bit of information at a time and are classified as level-triggered, meaning the outputs respond to the inputs only during the period that the clock, or enable, is on a particular level, high or low.

Flip-flops: Another storage element which can store one bit of information at a time and can maintain this binary state until it is signaled to change. However, instead of being level-triggered, flip-flops are considered edge-triggered, meaning the outputs respond to the inputs at the moment the clock, or enable, transitions from one state to another.

- Positive Edge-Triggered - Data is read on the rising edge of the clock/enable (transition from 0 to 1).
- Negative Edge-Triggered - Data is read on the falling edge of the clock/enable (transition from 1 to 0).

Registers: Another storage element which is usually composed of several other storage elements (typically flip-flops) and can store multiple bits of information at a time (a register of *n* number of flip-flops can store *n* number of bits).

Component: SR-Latch



```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity SR_Latch is
    port( S, R     : in STD_LOGIC;
          Q, notQ : buffer STD_LOGIC );
end SR_Latch;

architecture arch of SR_Latch is
begin
    Q <= (R NOR notQ);
    notQ <= (S NOR Q);
end arch;
```

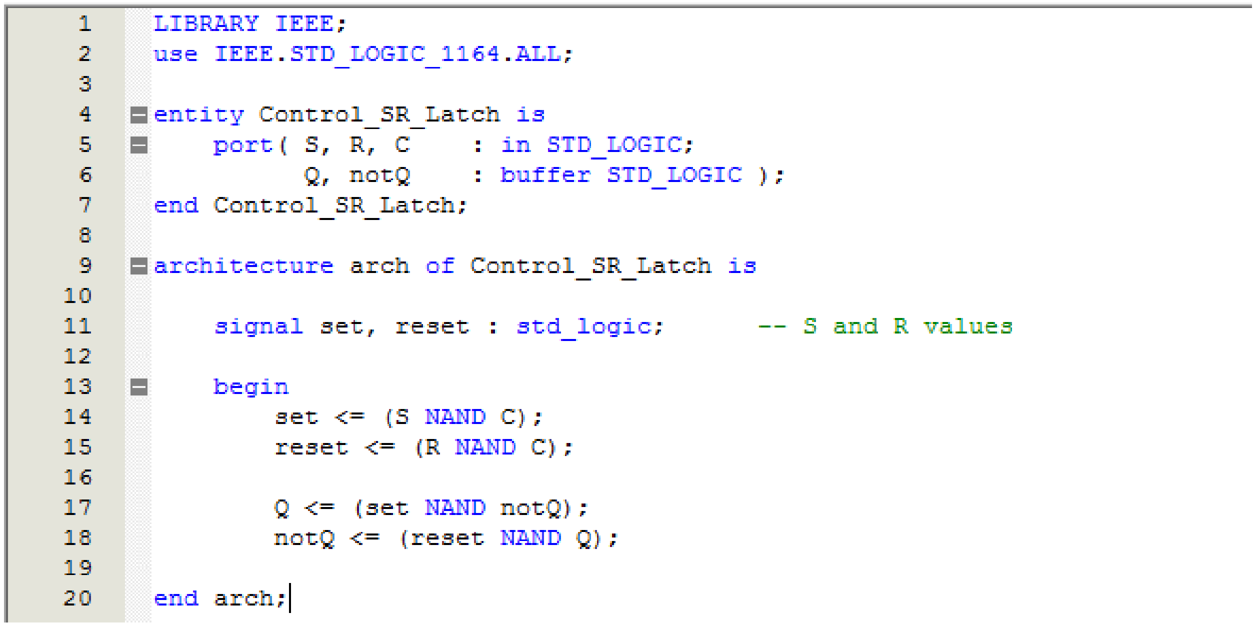
(SR_Latch.vhdl)

An SR-Latch is composed of two inputs S and R, or Set and Reset respectively, and two outputs, Q and its complement Q'. There are two NOR gates used, and the output of one is redirected back into the input of the other. This is called **feedback** and is how the SR-Latch is able to retain memory.

S	R	Q	Q'	State
1	0	1	0	Set
0	0	1	0	Set
0	1	0	1	Reset
0	0	0	1	Reset
1	1	0	0	Undefined

(SR-Latch Logic Table)

Component: Control SR-Latch



```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Control_SR_Latch is
    port( S, R, C     : in STD_LOGIC;
            Q, notQ     : buffer STD_LOGIC );
end Control_SR_Latch;

architecture arch of Control_SR_Latch is

    signal set, reset : std_logic;      -- S and R values

    begin
        set <= (S NAND C);
        reset <= (R NAND C);

        Q <= (set NAND notQ);
        notQ <= (reset NAND Q);

end arch;
```

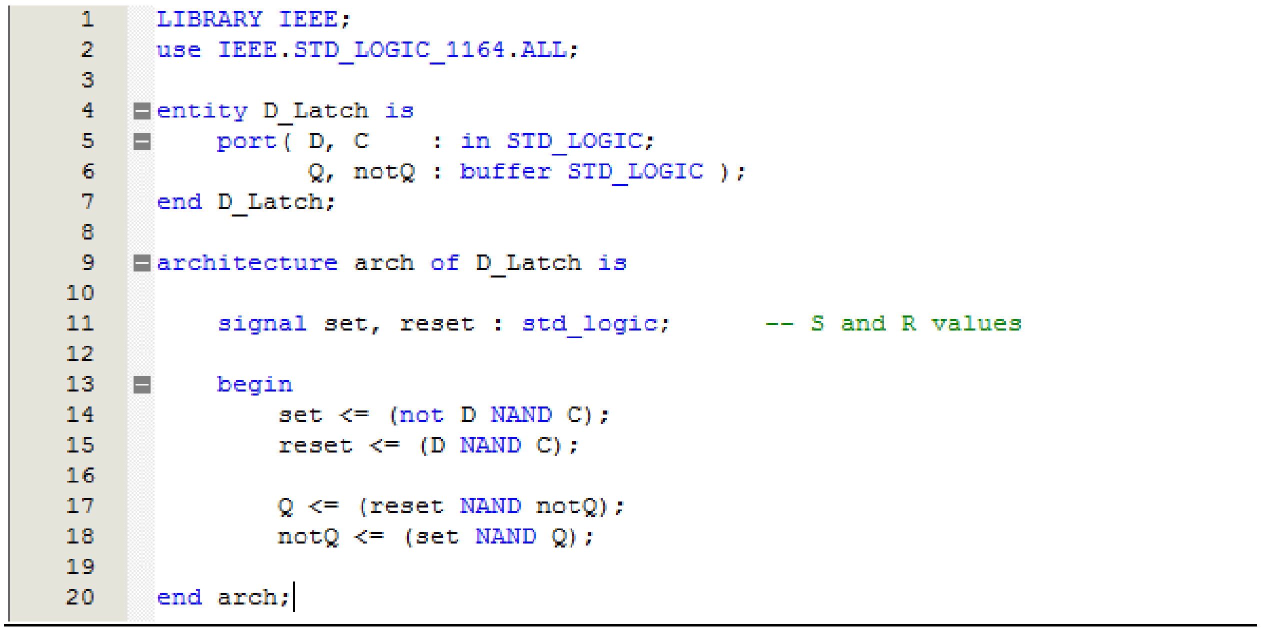
(Control_SR_Latch.vhdl)

Similar to the SR-Latch, but the control SR-Latch includes NAND gates, as opposed to NOR, and a control input C. The latch will only respond when C is 1, otherwise there will be no change, regardless of the values of S and R.

C	S	R	State
0	X	X	No Change
1	0	0	No Change
1	0	1	Reset (Q=0)
1	1	0	Set (Q=1)
1	1	1	Undefined

(Control SR-Latch Logic Table)

Component: D-Latch



```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_Latch is
    port( D, C     : in STD_LOGIC;
          Q, notQ : buffer STD_LOGIC );
end D_Latch;

architecture arch of D_Latch is

    signal set, reset : std_logic;      -- S and R values

    begin
        set <= (not D NAND C);
        reset <= (D NAND C);

        Q <= (reset NAND notQ);
        notQ <= (set NAND Q);

end arch;
```

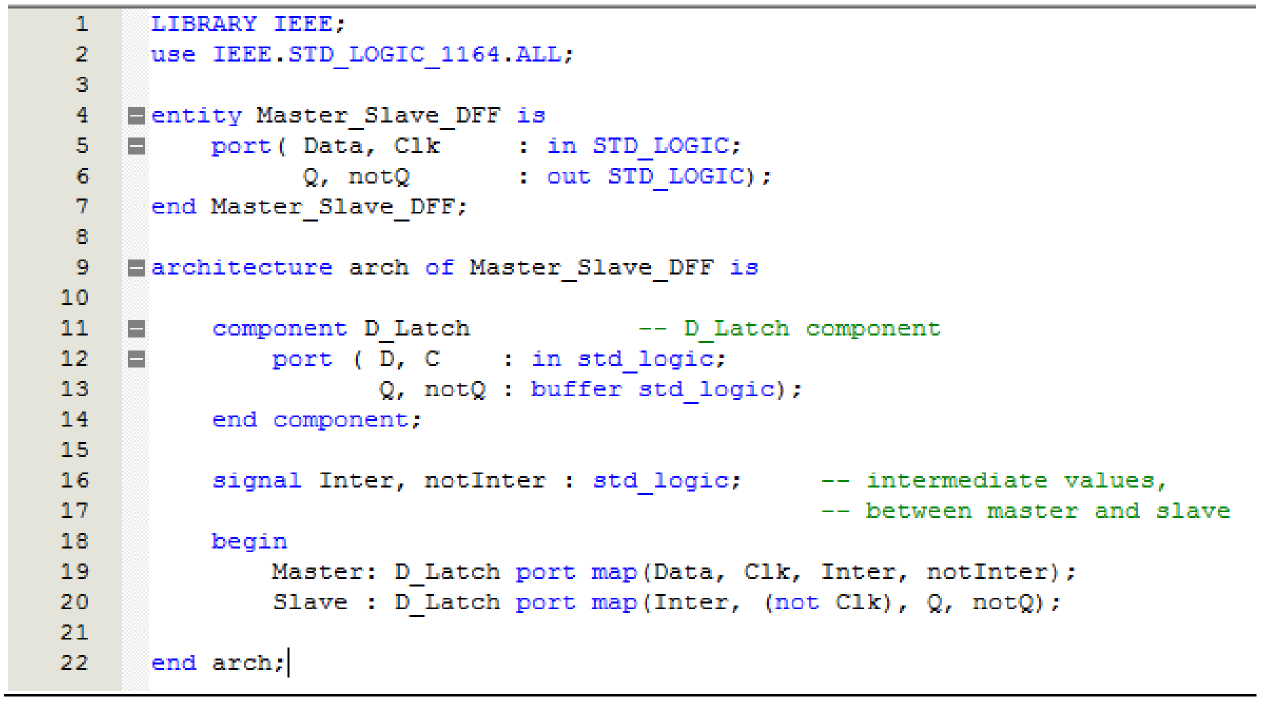
(D_Latch.vhdl)

The D-Latch eliminates the unwanted possibility of having S and R equal to 1 at the same time, thus preventing its undefined state. This is done so by replacing S and R with D, and simply passing D through one gate, and negating D through the other.

C	D	State
0	X	No Change
1	0	Reset (Q=0)
1	1	Set (Q=1)

(D-Latch Logic Table)

<u>Component: Master-Slave D-Flip-flop</u>

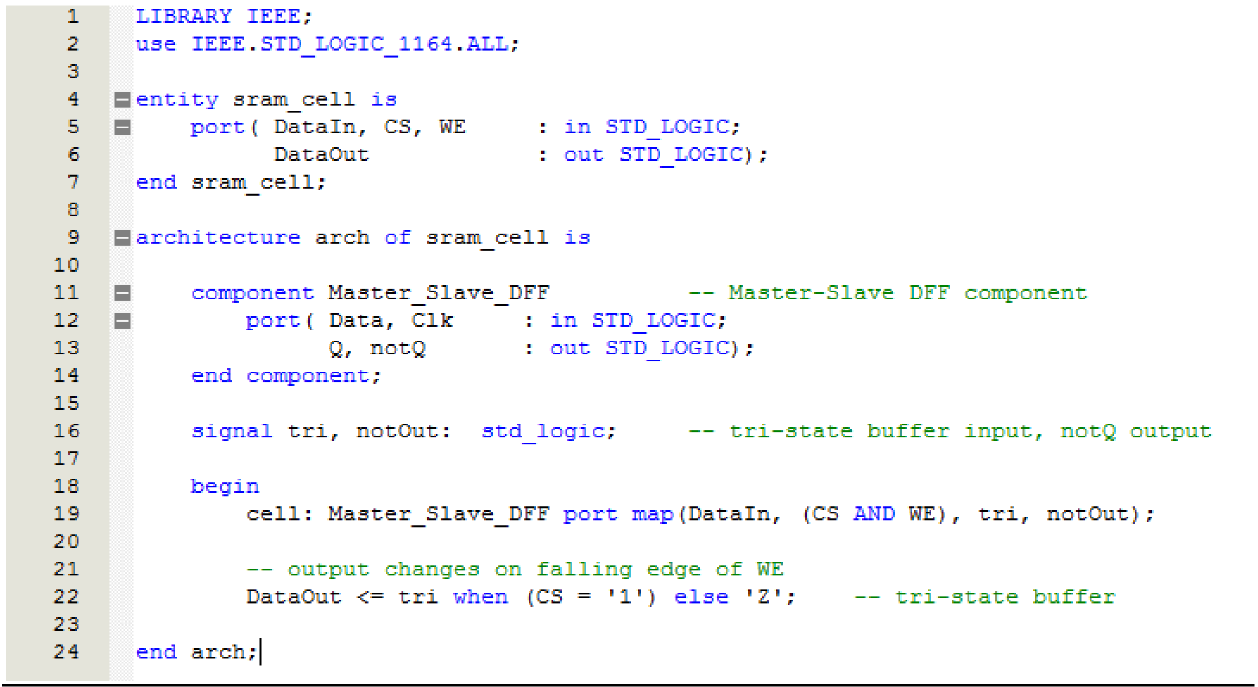


```vhdl
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Master_Slave_DFF is
    port( Data, Clk     : in STD_LOGIC;
          Q, notQ       : out STD_LOGIC);
end Master_Slave_DFF;

architecture arch of Master_Slave_DFF is

    component D_Latch               -- D_Latch component
        port ( D, C     : in std_logic;
               Q, notQ : buffer std_logic);
    end component;

    signal Inter, notInter : std_logic;     -- intermediate values,
                                            -- between master and slave
    begin
        Master: D_Latch port map(Data, Clk, Inter, notInter);
        Slave : D_Latch port map(Inter, (not Clk), Q, notQ);

end arch;
```

(Master_Slave_DFF.vhdl)

Composed of two D-latches connected in sequence. The output of the first latch is fed into the input of the second latch. The clock is connected to both latches, however, it is inverted on the second latch. By doing this, we create a negative-edge triggered flip-flop, and so information is stored on the falling edge of the clock input.

Component: SRAM Cell



```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sram_cell is
    port( DataIn, CS, WE      : in STD_LOGIC;
          DataOut             : out STD_LOGIC);
end sram_cell;

architecture arch of sram_cell is

    component Master_Slave_DFF          -- Master-Slave DFF component
        port( Data, Clk      : in STD_LOGIC;
              Q, notQ        : out STD_LOGIC);
    end component;

    signal tri, notOut:  std_logic;     -- tri-state buffer input, notQ output

    begin
        cell: Master_Slave_DFF port map(DataIn, (CS AND WE), tri, notOut);

        -- output changes on falling edge of WE
        DataOut <= tri when (CS = '1') else 'Z';    -- tri-state buffer

end arch;
```

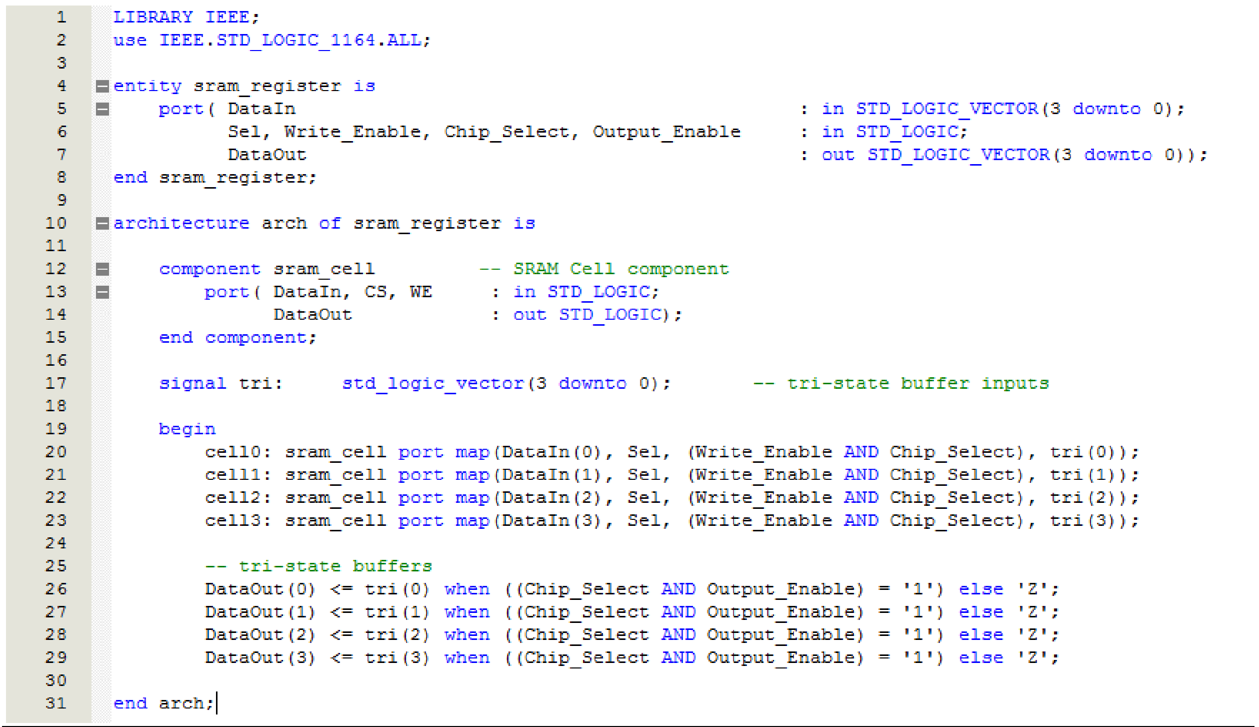
(sram_cell.vhdl)

An SRAM cell is created by extending the functionality of a master-slave D-Flip-flop. We have included a chip select (CS) and a write enable (WE) that must both be on in order to write to the flip-flop. The chip select activates the cell when it is high, otherwise you cannot write nor output data. The Write enable allows the cell to accept input, which is stored on the falling edge of the write enable input. The tri signal represents a tri-state buffer, which is a component that allows input to pass only if the chip select is high.

A	B	Out
0	1	0
1	1	1
X	0	Z

(Tri-State Buffer Logic Table)

Component: SRAM Register

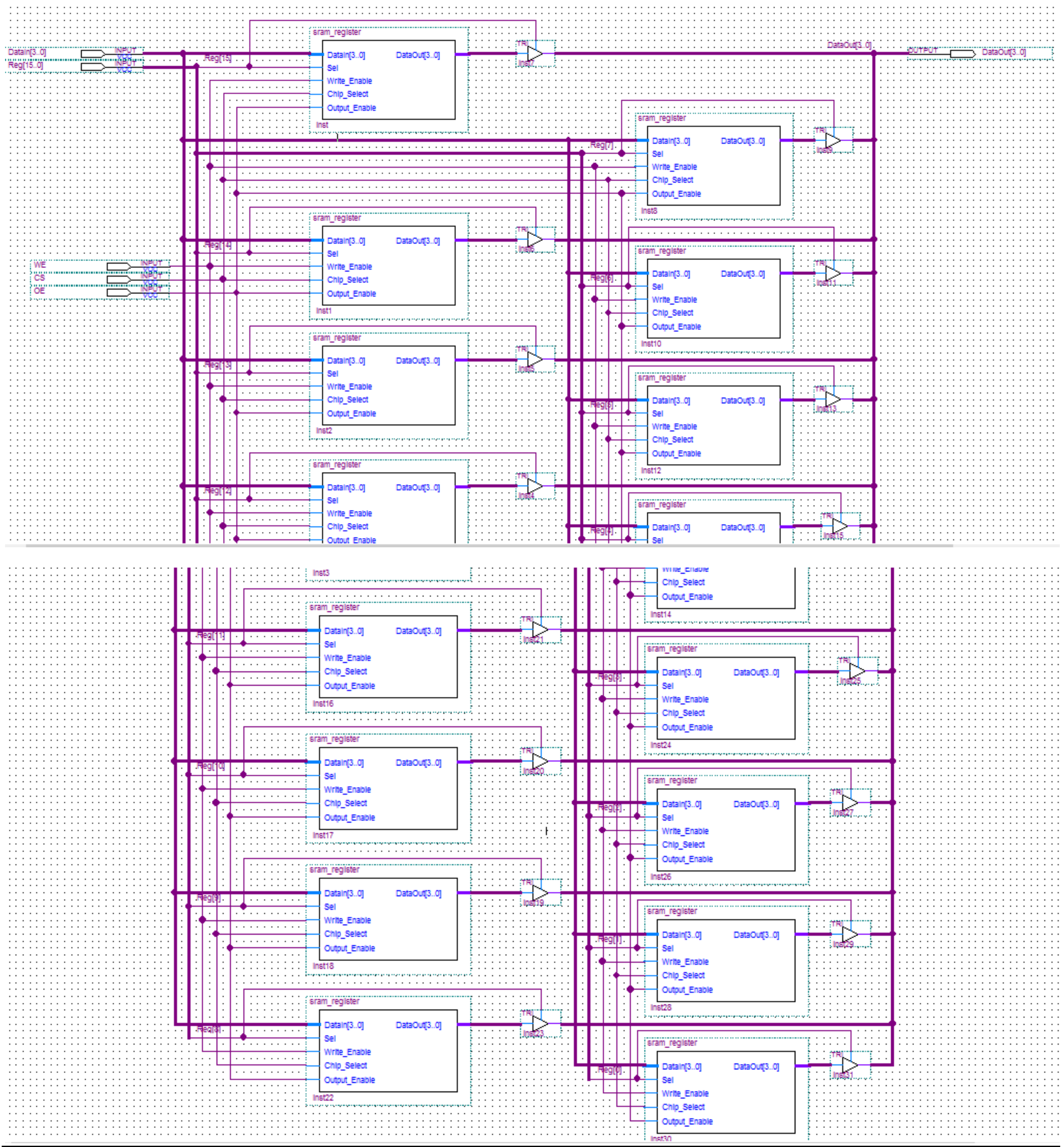


```vhdl
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sram_register is
    port( DataIn                                               : in STD_LOGIC_VECTOR(3 downto 0);
          Sel, Write_Enable, Chip_Select, Output_Enable        : in STD_LOGIC;
          DataOut                                              : out STD_LOGIC_VECTOR(3 downto 0));
end sram_register;

architecture arch of sram_register is

    component sram_cell          -- SRAM Cell component
        port( DataIn, CS, WE     : in STD_LOGIC;
              DataOut            : out STD_LOGIC);
    end component;

    signal tri:     std_logic_vector(3 downto 0);       -- tri-state buffer inputs

    begin
        cell0: sram_cell port map(DataIn(0), Sel, (Write_Enable AND Chip_Select), tri(0));
        cell1: sram_cell port map(DataIn(1), Sel, (Write_Enable AND Chip_Select), tri(1));
        cell2: sram_cell port map(DataIn(2), Sel, (Write_Enable AND Chip_Select), tri(2));
        cell3: sram_cell port map(DataIn(3), Sel, (Write_Enable AND Chip_Select), tri(3));

        -- tri-state buffers
        DataOut(0) <= tri(0) when ((Chip_Select AND Output_Enable) = '1') else 'Z';
        DataOut(1) <= tri(1) when ((Chip_Select AND Output_Enable) = '1') else 'Z';
        DataOut(2) <= tri(2) when ((Chip_Select AND Output_Enable) = '1') else 'Z';
        DataOut(3) <= tri(3) when ((Chip_Select AND Output_Enable) = '1') else 'Z';

end arch;
```

(sram_register.vhdl)

This register is composed of 4 SRAM cells, allowing it to store up to 4 bits of information. Once again, the output of each cell must pass through a tri-state buffer which allows the output to be projected only if both the chip select and the output enable is high. All contents will always be stored in the register, even if all inputs were to cut off.

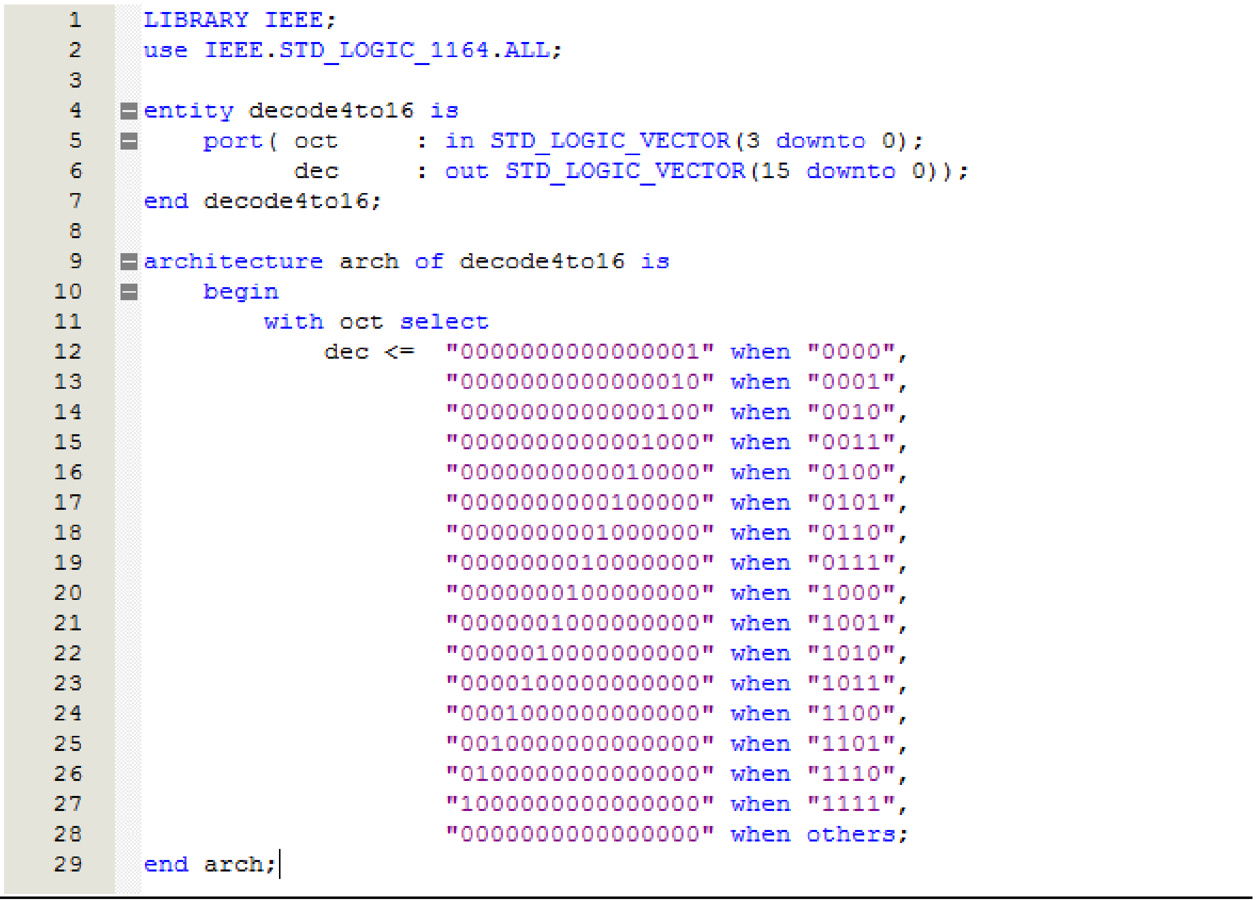
Component: 16x4 SRAM



(sram.bdf)

Here, we have connected 16 registers together, and the Reg input will determine which register will be written or read from. This makes up our entire SRAM unit.

<u>Component: 4-to-16 Decoder</u>

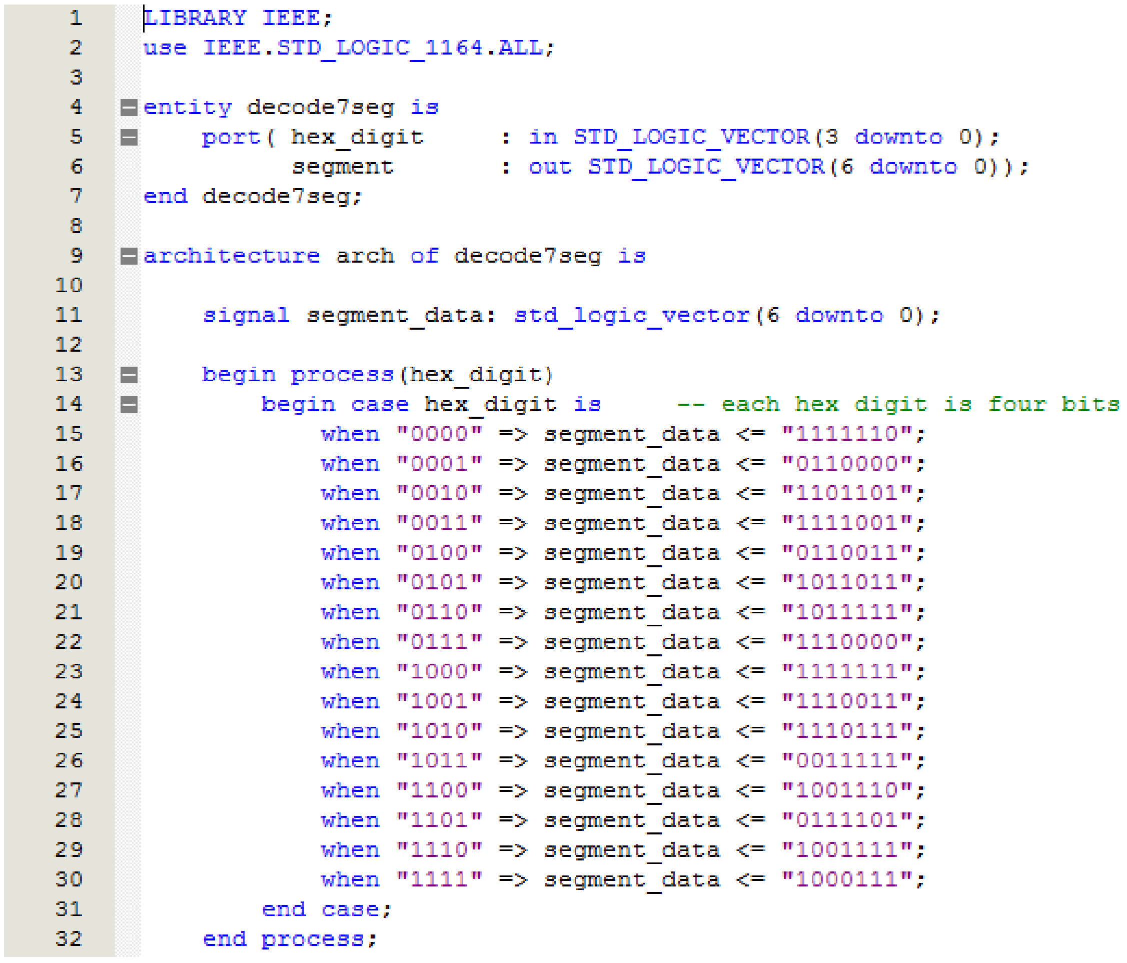


```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decode4to16 is
    port( oct      : in STD_LOGIC_VECTOR(3 downto 0);
          dec      : out STD_LOGIC_VECTOR(15 downto 0));
end decode4to16;

architecture arch of decode4to16 is
    begin
        with oct select
            dec <=  "0000000000000001" when "0000",
                    "0000000000000010" when "0001",
                    "0000000000000100" when "0010",
                    "0000000000001000" when "0011",
                    "0000000000010000" when "0100",
                    "0000000000100000" when "0101",
                    "0000000001000000" when "0110",
                    "0000000010000000" when "0111",
                    "0000000100000000" when "1000",
                    "0000001000000000" when "1001",
                    "0000010000000000" when "1010",
                    "0000100000000000" when "1011",
                    "0001000000000000" when "1100",
                    "0010000000000000" when "1101",
                    "0100000000000000" when "1110",
                    "1000000000000000" when "1111",
                    "0000000000000000" when others;
end arch;
```

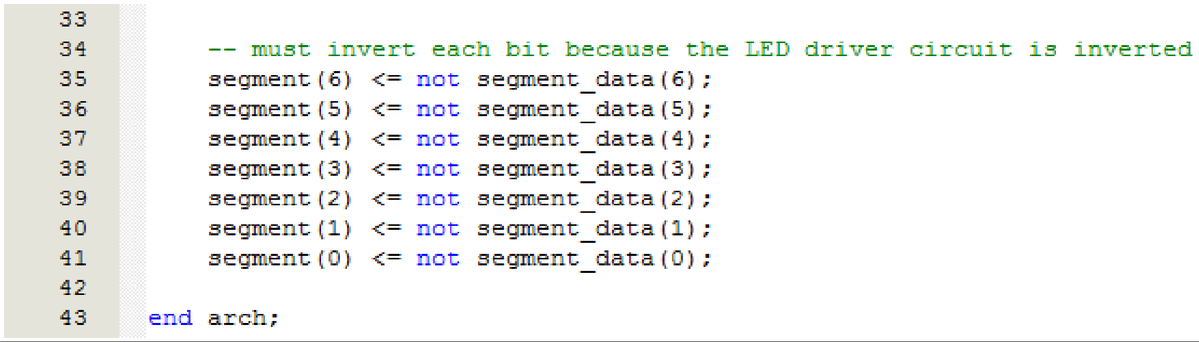
(decode4to16.vhdl)

This decoder takes in a 4 bit string and returns a 16 bit string which will be connected to the Reg input of the SRAM. There are 16 possible outputs for this decoder meant for each of the 16 registers in the SRAM.

Component: Seven-Segment Hex Display Decoder



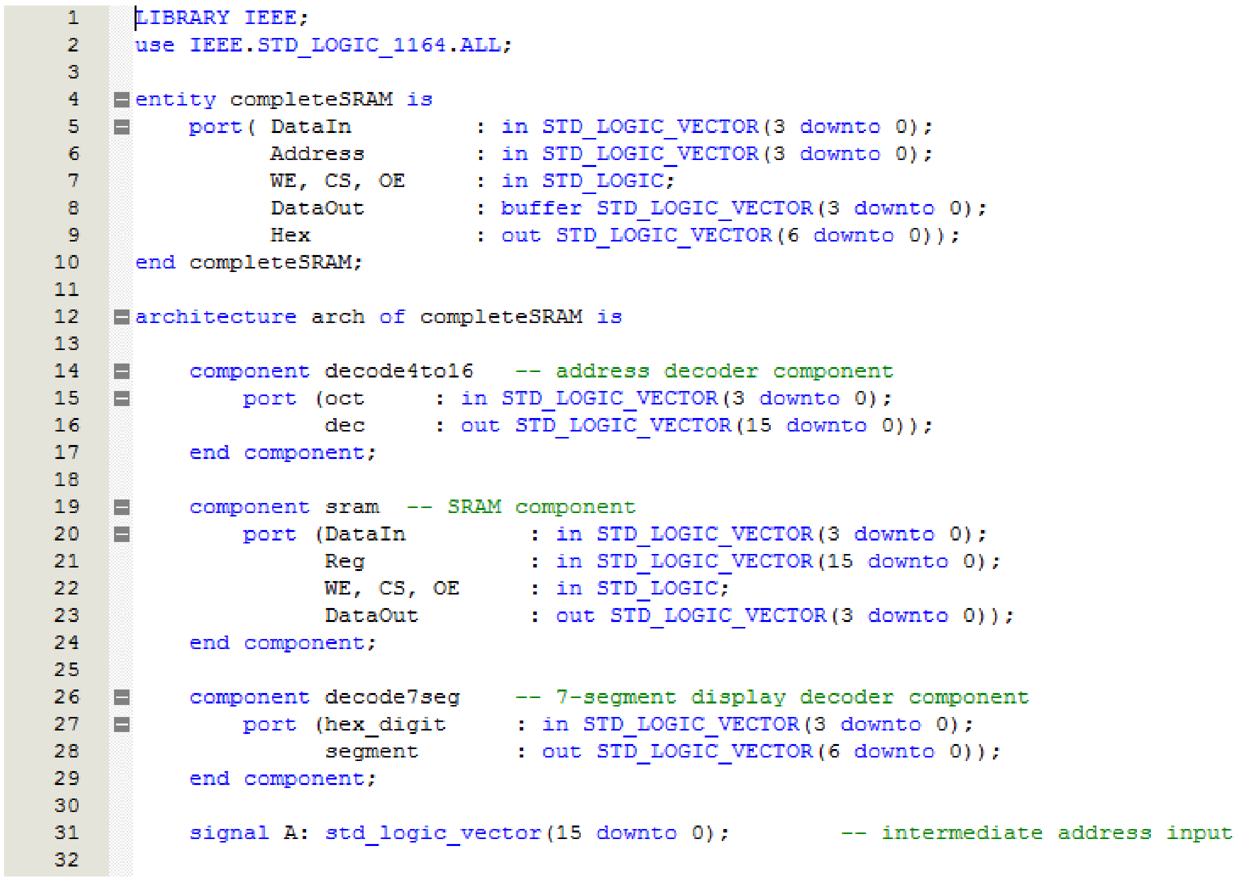
```vhdl
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decode7seg is
    port( hex_digit     : in STD_LOGIC_VECTOR(3 downto 0);
          segment       : out STD_LOGIC_VECTOR(6 downto 0));
end decode7seg;

architecture arch of decode7seg is

    signal segment_data: std_logic_vector(6 downto 0);

    begin process(hex_digit)
        begin case hex_digit is     -- each hex digit is four bits
            when "0000" => segment_data <= "1111110";
            when "0001" => segment_data <= "0110000";
            when "0010" => segment_data <= "1101101";
            when "0011" => segment_data <= "1111001";
            when "0100" => segment_data <= "0110011";
            when "0101" => segment_data <= "1011011";
            when "0110" => segment_data <= "1011111";
            when "0111" => segment_data <= "1110000";
            when "1000" => segment_data <= "1111111";
            when "1001" => segment_data <= "1110011";
            when "1010" => segment_data <= "1110111";
            when "1011" => segment_data <= "0011111";
            when "1100" => segment_data <= "1001110";
            when "1101" => segment_data <= "0111101";
            when "1110" => segment_data <= "1001111";
            when "1111" => segment_data <= "1000111";
        end case;
    end process;
```



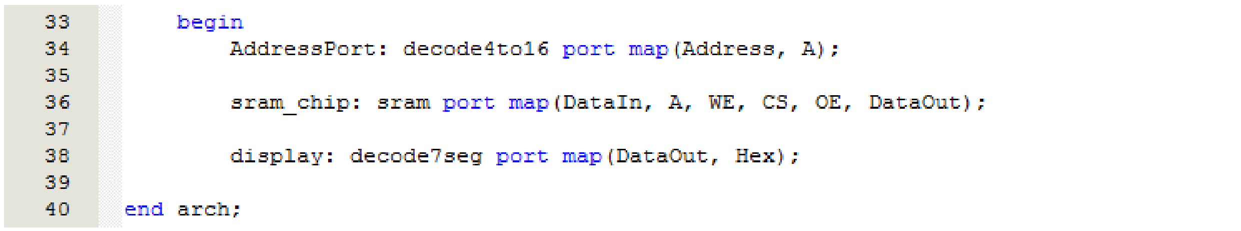
```vhdl

    -- must invert each bit because the LED driver circuit is inverted
    segment(6) <= not segment_data(6);
    segment(5) <= not segment_data(5);
    segment(4) <= not segment_data(4);
    segment(3) <= not segment_data(3);
    segment(2) <= not segment_data(2);
    segment(1) <= not segment_data(1);
    segment(0) <= not segment_data(0);

end arch;
```

(decode7seg.vhdl)

Complete Design

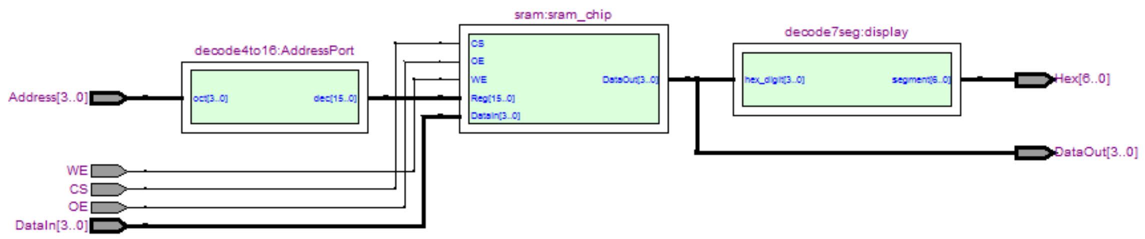


```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity completeSRAM is
    port( DataIn         : in STD_LOGIC_VECTOR(3 downto 0);
          Address        : in STD_LOGIC_VECTOR(3 downto 0);
          WE, CS, OE     : in STD_LOGIC;
          DataOut        : buffer STD_LOGIC_VECTOR(3 downto 0);
          Hex            : out STD_LOGIC_VECTOR(6 downto 0));
end completeSRAM;

architecture arch of completeSRAM is

    component decode4to16   -- address decoder component
        port (oct     : in STD_LOGIC_VECTOR(3 downto 0);
              dec     : out STD_LOGIC_VECTOR(15 downto 0));
    end component;

    component sram  -- SRAM component
        port (DataIn         : in STD_LOGIC_VECTOR(3 downto 0);
              Reg            : in STD_LOGIC_VECTOR(15 downto 0);
              WE, CS, OE     : in STD_LOGIC;
              DataOut        : out STD_LOGIC_VECTOR(3 downto 0));
    end component;

    component decode7seg    -- 7-segment display decoder component
        port (hex_digit     : in STD_LOGIC_VECTOR(3 downto 0);
              segment       : out STD_LOGIC_VECTOR(6 downto 0));
    end component;

    signal A: std_logic_vector(15 downto 0);        -- intermediate address input

```



```
    begin
        AddressPort: decode4to16 port map(Address, A);

        sram_chip: sram port map(DataIn, A, WE, CS, OE, DataOut);

        display: decode7seg port map(DataOut, Hex);

end arch;
```

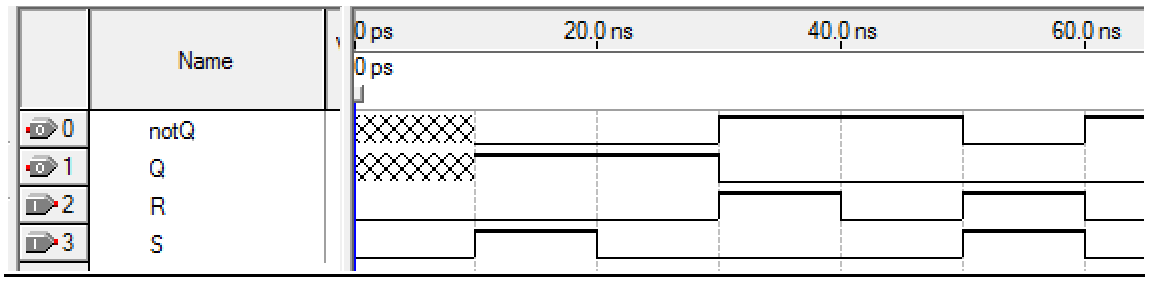
(completeSRAM.vhdl)



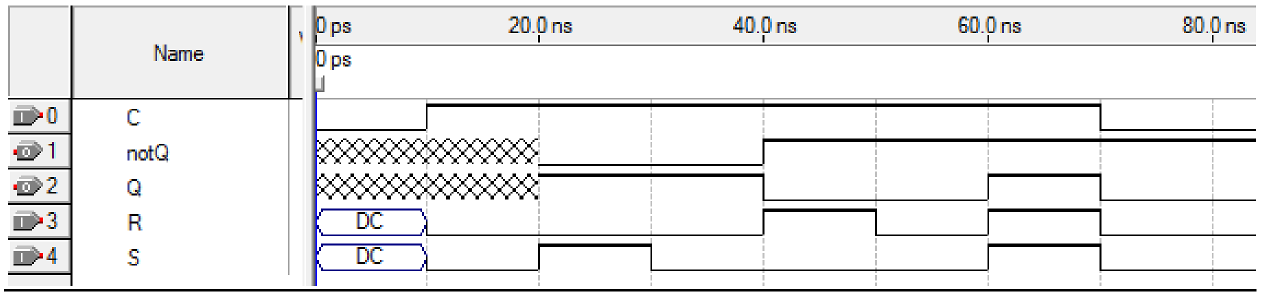
(RTL View of completeSRAM.vhdl)

Simulation

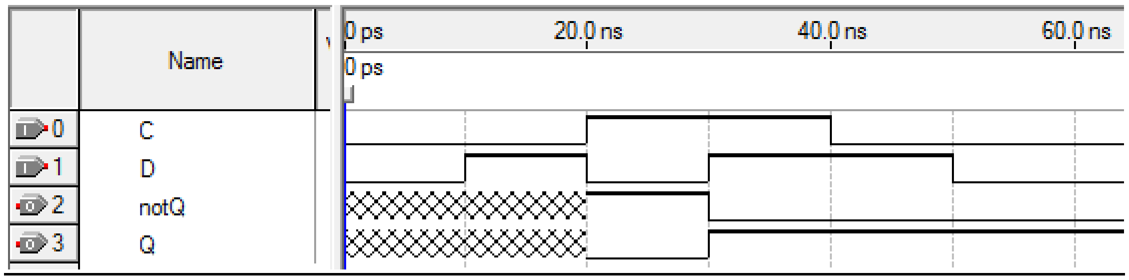
Vector Waveform: SR-Latch



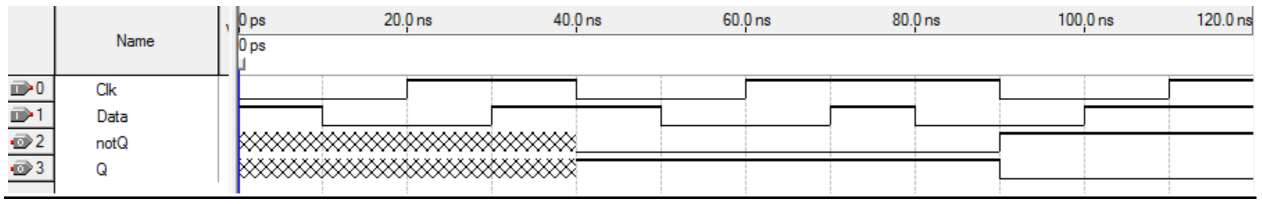
Vector Waveform: Control SR-Latch



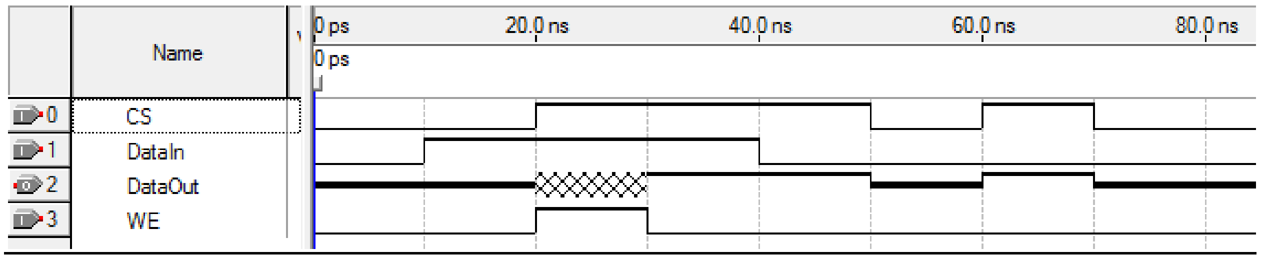
Vector Waveform: D-Latch



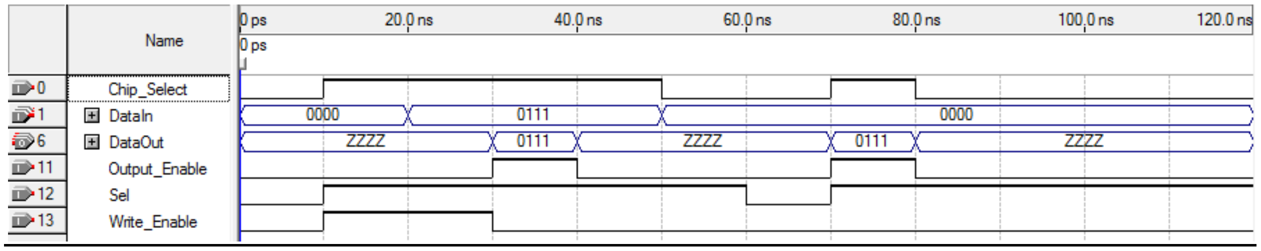
Vector Waveform: Master-Slave D-Flip-flop



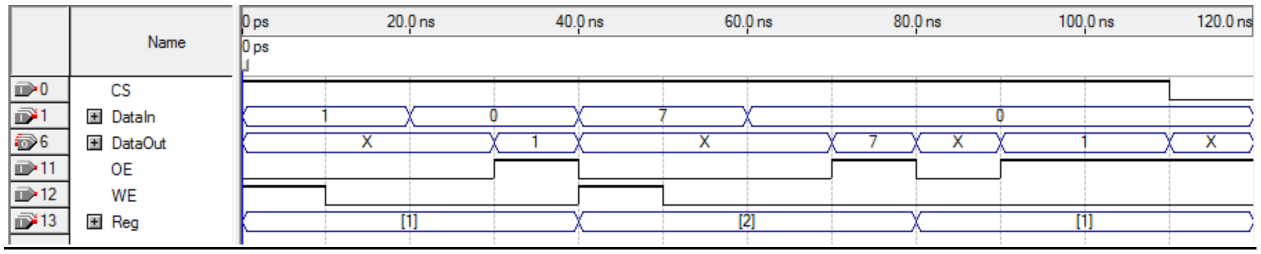
Vector Waveform: SRAM Cell



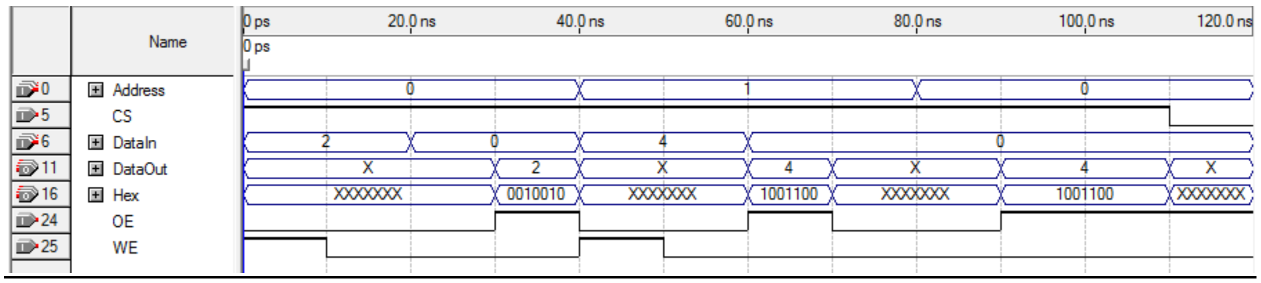
Vector Waveform: SRAM Register



Vector Waveform: 16x4 SRAM



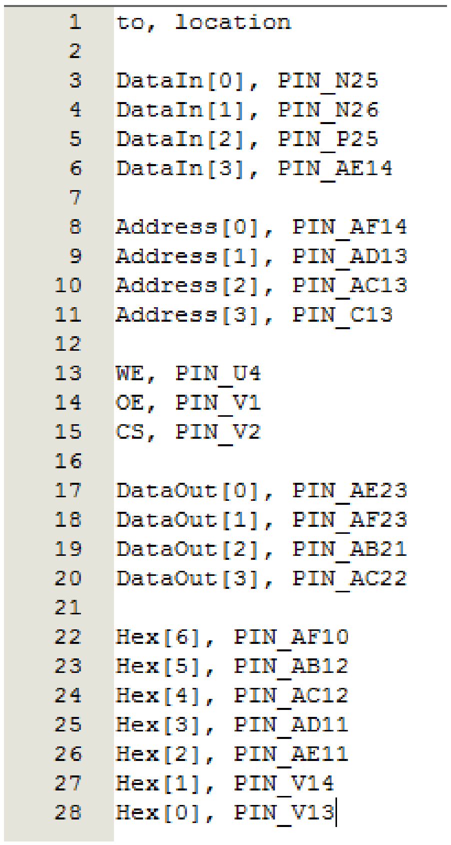
Vector Waveform: Complete Design



DE2 Circuit Board Test

Before connecting to the DE2 board, we must first assign the correct pins to each component of the circuit. The inputs (DataIn, Address, WE, OE, and CS) will be assigned to the board's toggle switches, the outputs (DataOut) will be assigned to the red LED lights, and the outputs (Hex) will be assigned to the first seven-segment hex display.

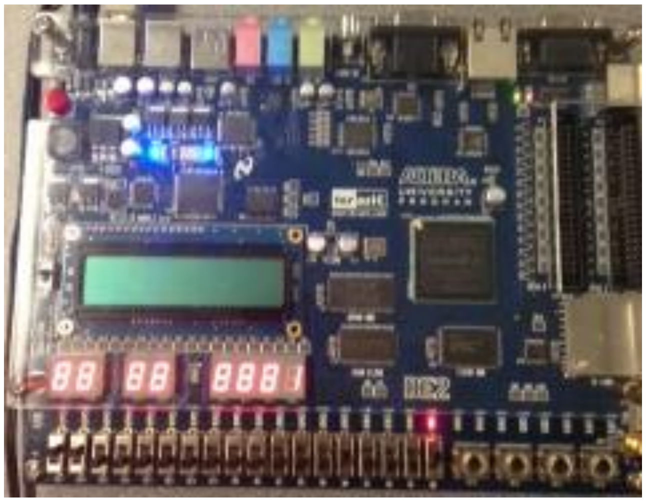
Pin assignments text file:



```
to, location

DataIn[0], PIN_N25
DataIn[1], PIN_N26
DataIn[2], PIN_P25
DataIn[3], PIN_AE14

Address[0], PIN_AF14
Address[1], PIN_AD13
Address[2], PIN_AC13
Address[3], PIN_C13

WE, PIN_U4
OE, PIN_V1
CS, PIN_V2

DataOut[0], PIN_AE23
DataOut[1], PIN_AF23
DataOut[2], PIN_AB21
DataOut[3], PIN_AC22

Hex[6], PIN_AF10
Hex[5], PIN_AB12
Hex[4], PIN_AC12
Hex[3], PIN_AD11
Hex[2], PIN_AE11
Hex[1], PIN_V14
Hex[0], PIN_V13
```

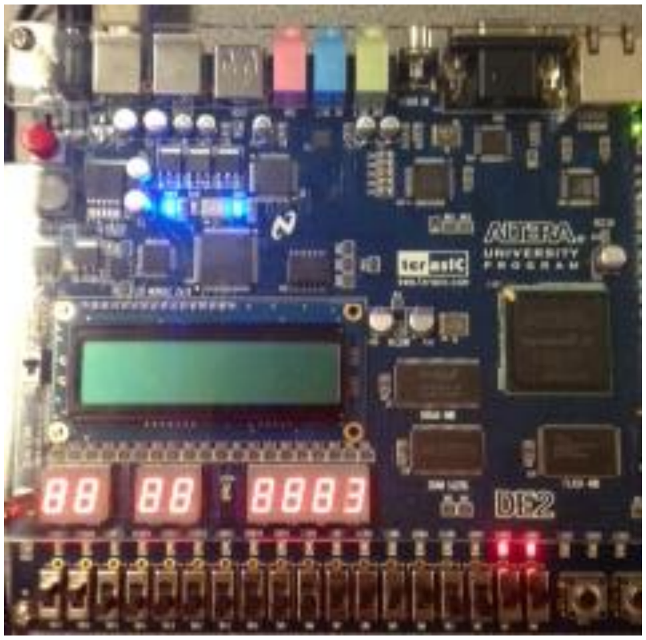
Now we can begin board testing:

Instance 1:



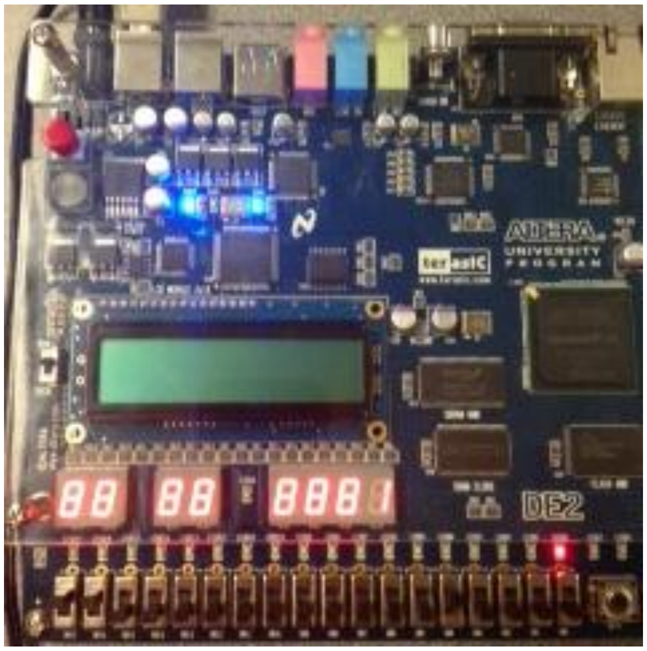
First we are storing the value 1 into register 1.

Instance 2:



Then we are storing the value 3 into the register 15.

Instance 3:



Finally, we are reading the value that we previously stored into register 1. The display shows the value 1, despite the input data being 0. This means that we have successfully stored the data into a register without the need to always be inputting that data.

Conclusion

Beginning with basic components, we continued to extend upon the functionality of our circuit. The major component was the SRAM device composed of multiple address locations which can each store strings of information. There are two operations that the SRAM can perform, read and write. We have demonstrated that after specifying an address location, we can choose to write to the address (this will overwrite any existing information that may have previously been stored at the location), or we can read from the location (this will display the contents of the address through the 7-segment HEX display on the DE2 Board).