Project 4: SRAM

Computer Organization CSC 34300 - EF

By: Sebastian Grygorczuk

Tabel of Content:

Objective:

The objective of this lab is to create a 16x32 SRAM and display the results of it on eight seven segment displays using eight bit input, four bit address operation code, chip select input, write enable input and output enable input and four keys to select which eight bits to of the address to write to.

Fundamentals:

Before we start working on building a memory we have to go over some basic elements that will help us to build towards the SRAM, these elements are SR-Latch, Control SR-Latch, D-Latch, and the Master Slave Flip Flop.

SR-Latch

The SR-Latch, is a basic memory device, it has two inputs Set and Reset which define the output if Q and Q' to be [1,0] when S is 1 and R is 0, and [0,1] when S is 0 and R is 1. When both are zero the SR-Latch will hold the value that was computed before, and when the two are both 1 the output is unpredictable and for such reason SR-Latch requires improvement we will see with the following devices. Below is the implementation of SR-Latch in Block Diagram and it's waveform.

S	R	Q	Q'	State
0	0	Q	Q'	No Change
0	1	0	1	Reset
1	0	1	0	Set
1	1	?	?	Undefined

Tabel 1. SR-Latch Truth Table

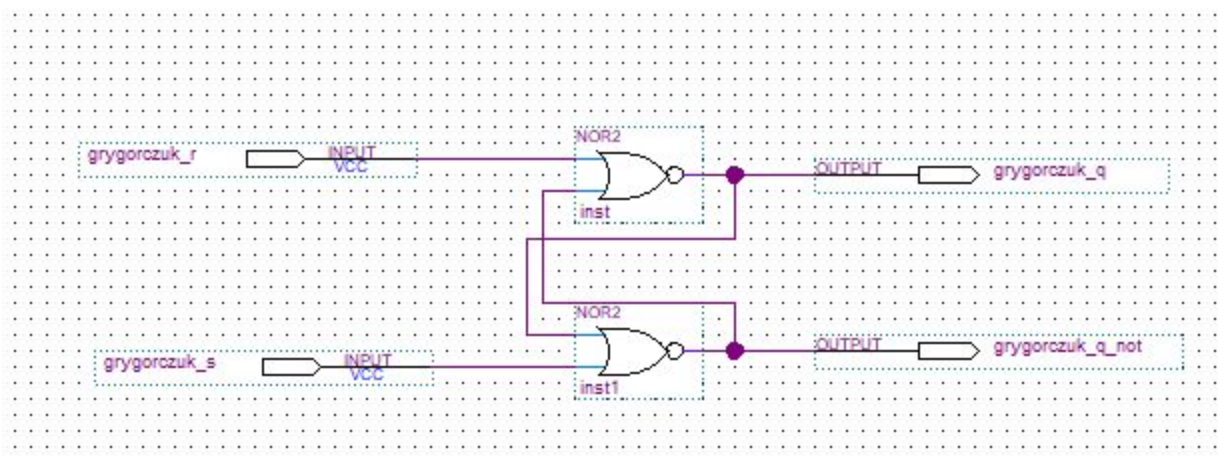

Figure 1. SR-Latch

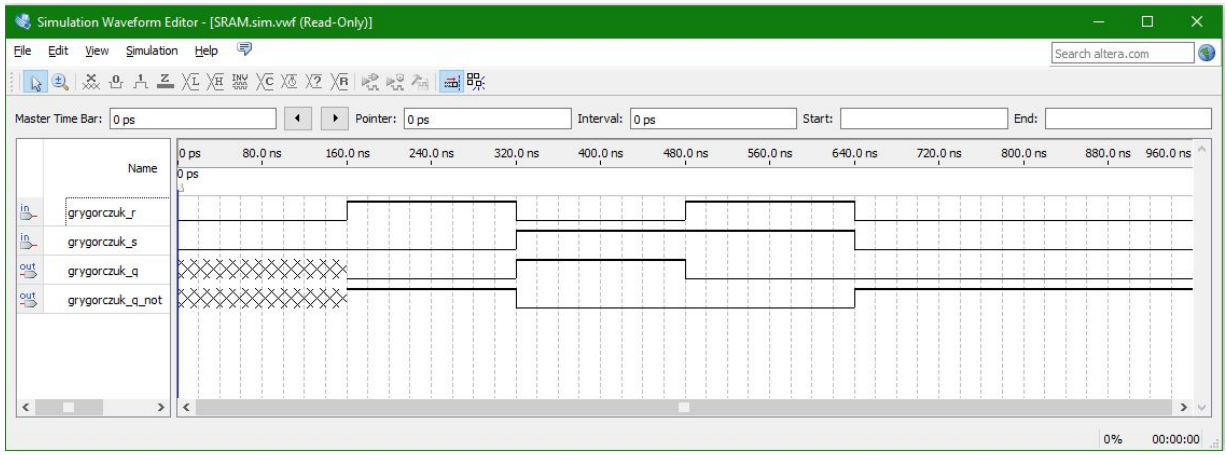

Figure 2. SR-Latch Waveform

As seen in Figure 2, when the first input is both zeros then the output is unknown, then it processed as expected till both inputs are one giving us an output of both zeroes yet when the next input is both zeroes the output is different, this is the reason why SR-Latch is inadequate for percisie usage, we just don't know what could happen when both S and R are set to one. Lastly we see that the SR-Latch holds onto the value in the very last segment as intended serving as a basic memory unit.

Control SR-Latch

Next we look at a Control SR-Latch, it's an improvement on the SR-Latch in the sense that it gives us more control over when the state can change however it's still has the issue of when both S and R are set to 1 we have an undefined state. Below is the block diagram implementation and waveform of the Control SR-Latch

C	S	R	Q	Q’	State
0	X	X	Q	Q’	No Change
1	0	0	Q	Q’	No Change
1	0	1	0	1	Reset
1	1	0	1	0	Set
1	1	1	?	?	Undefined

Tabel 2. Control SR-Latch

The X in S and R rows means that no matter what state S or R is the it all depends on C.

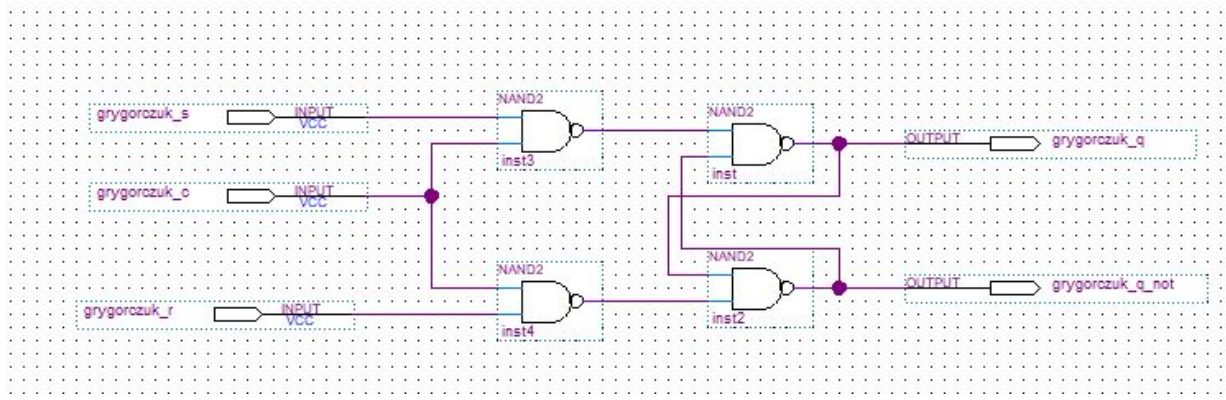

Figure 3. Control SR-Latch

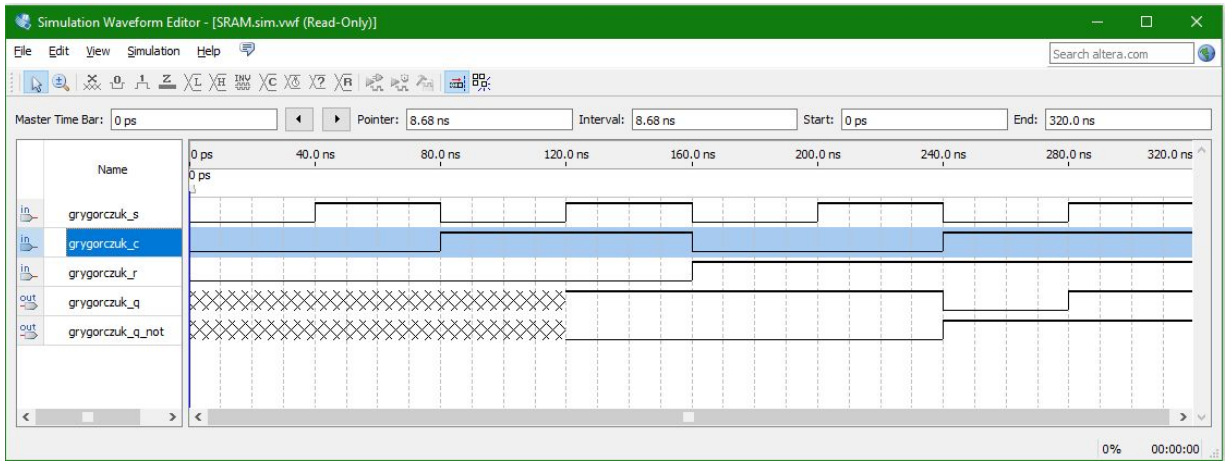

Figure 4. Control SR-Latch Waveform

In Figure 4, we see that the first three states redefined because there has been no input placed in them yet and C is either 0 or S and R are both 0. Then we see the S be turned on, followed by C being 0 and holding the set, next we then have R turned on, and then we see all of them turned on which gives us gibberish.

D-Latch

D-Latch is an improvement on the Control SR-Latch as it finally takes care of the problem of S and R being 1 at the same time, we just set them to be the negative of the other. Below is the implementation and the waveform of the D-Latch.

C	D	Q	Q’	State
0	X	Q	Q’	No Change
1	0	0	1	Reset
1	1	1	0	Set

Table 3. D-Latch Truth Table

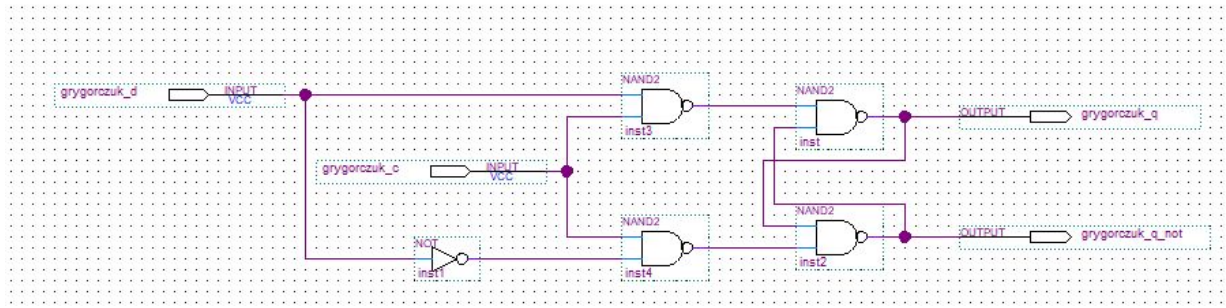

Figure 5. D-Latch

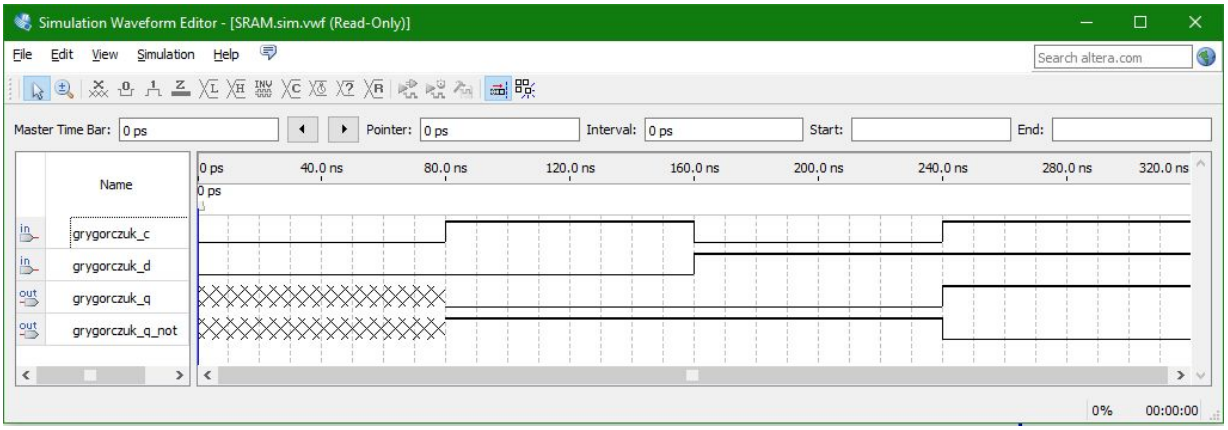

Figure 6. D-Latch Waveform

As seen above first the D-Latch holds the state, the sets it to Reset, holds it and then sets it to Set. There is no issue of having unknown outputs once the memory has been set.

Master Slave Flip Flop:

Now that we have a reliable memory device we have to make sure that it grabs the right data at the right time and not any random data that might come its way. The solution to that is the Master Slave Flip Flop which will only change states during the rising edge in our situation, meaning that whatever the state D is it will be saved if the clock is going from low to high state.

C	D	State
Low to High	X	Change
High to Low	X	No Change

Tabel 4. Master Slave Flip Flop Truth Table

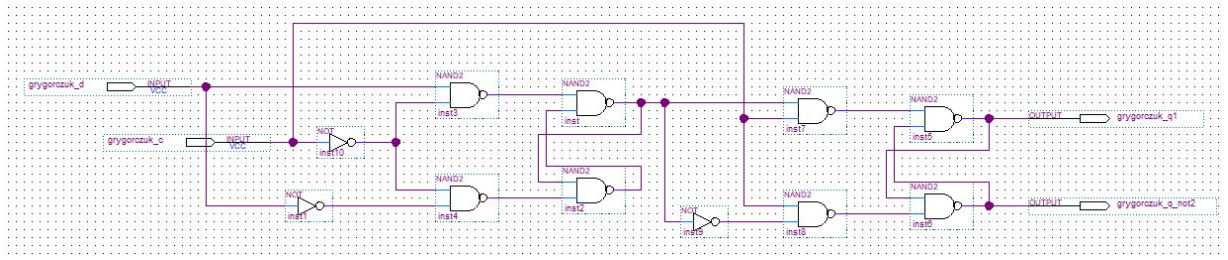

Figure 7. Master Slave Flip Flop

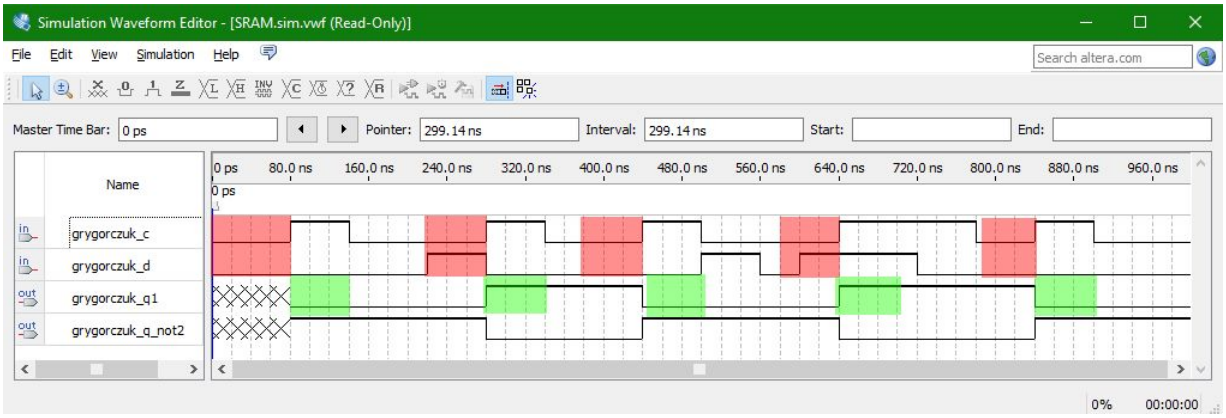

Figure 8. Master Slave Flip Flop Waveform

In figure 8 we look at a random wavepatter given to C and D. As we can see Q becomes whatever whatever D was before the C went from low to high. I highlighted the rising edge moments in red and the saved data after in green to show that it indeed saves the data from the previous period of rising edge. Everywhere else there is no change occuring.

SRAM:

Now that the basics are complete it's time to build the SRAM, we will have to do few things to get there, build a single cell of an SRAM, create a 16x1 SRAM module, create an address decoder which will allow us to switch between sixteen different SRAM locations, put all of that together to create a 16x4 SRAM, and finally create a decoder for a seven segment display.

SRAM Cell:

To create an SRAM Cell we will use the Master Slave Flip Flop we showed in previous section with the addition of new inputs that will serve as our C and D. We will have Input, Select Chip, and Write Enable. Input is straightforward it's the data we are trying to save. Select Chip will make the device operational, if it's high it will save, if it's low it will make the device give out a buffer state. The Write Enable allows data from the Input to be saved.

Select Chip	Write Enable	Input	State
0	X	X	Z
1	0	X	Output
1	1	X	Output and Save

Tabel 5. SRAM Cell

When both Select Chip is 0, Write Enable can be either 0 or 1 the output will only give out the buffer Z. If Select Chip is 1 and Write Enable is 0 the output will be whatever is stored in the SRAM Cell. If both the Select Chip and Write Enable are on it will give the output and save the input.

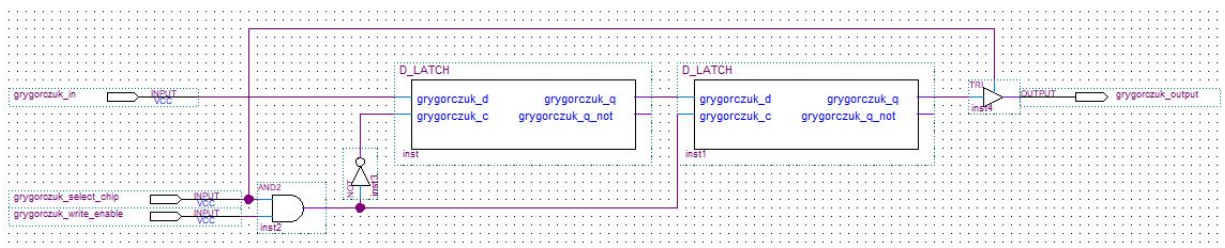

Figure 9. SRAM Cell

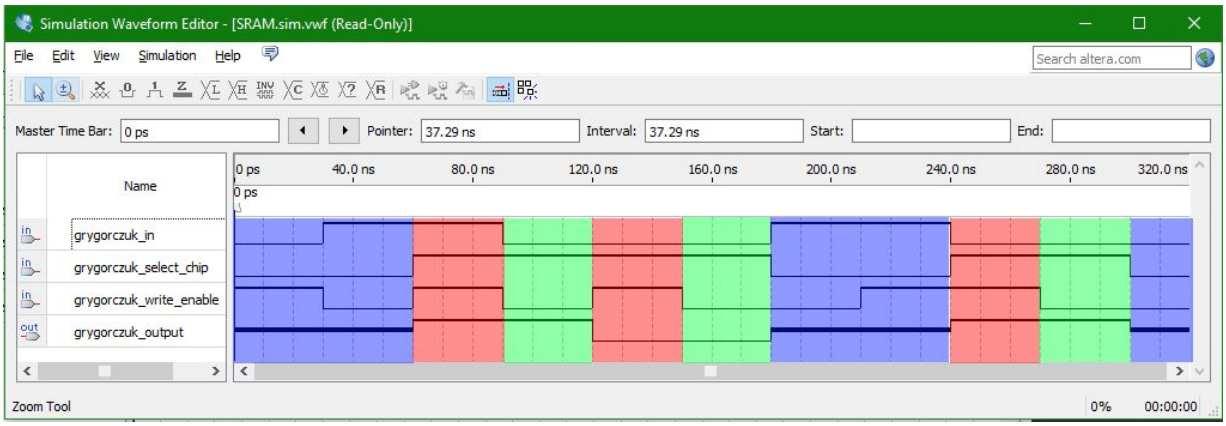

Figure 10. SRAM Cell Waveform

In figure 10 I show that when Select Chip is 0 the output is the buffer Z in blue. That when both Select chip and Write Enable are on they save and display the value of input of the value that it was during the rising edge in red and in green I show when only Select Chip is 1 it displays the stored value.

SRAM 16x1 Module:

Next step to getting a SRAM that is 16x4 is to make a 16x1 modula that will help make the job easier. The modula will have three inputs, the address decoder value which will tell the module which of the cells to read/write from (This is the Select Chip for a Cell). Only the chosen cell will have some kind of output all the others will give off the Z buffer. The input that will be stored in the chosen cell, and the write enable which will say that you can overwrite what's inside the cell. It will have output which take the output from chosen cell and pass it onto the seven segment display. Below are the block diagram of the Module and a waveform showing how it functions.

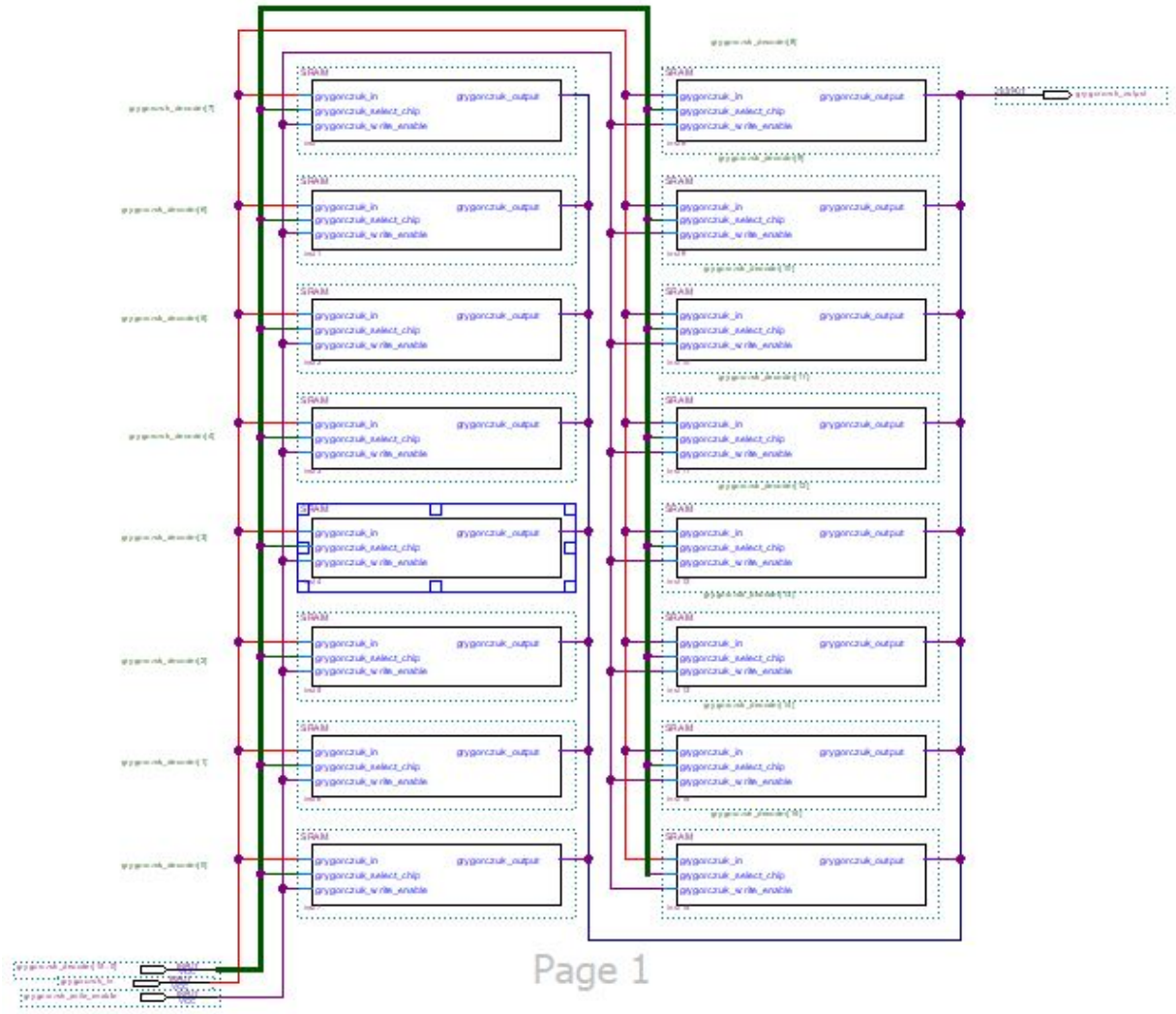

Figure 11. SMRA 16x1

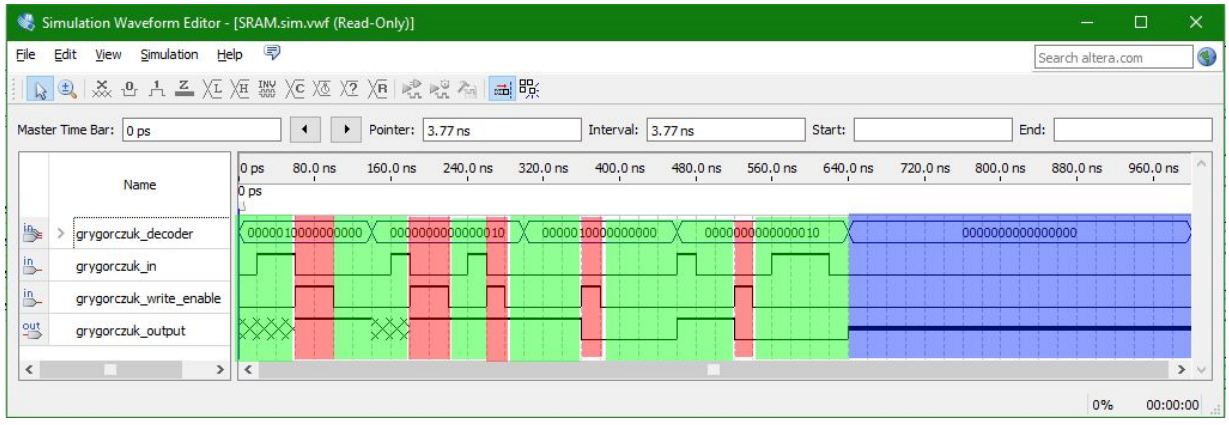

Figure 12. SRAM 16x1 Waveform

In Figure 12, we see that we can access any of that 16 cells using the decoder input, you see me access 10th cell and the 2nd cell multiple time save and read their values they had every time the waveform is red. Anytime the waveform is green its us accessing the cell to read from it and anytime the input from the decoder is all zeroes the output will give us the buffer signal. This works exactly like a Master Slave Flip Flop but now we can acesse many of them not just one.

Address Decoder:

To be able to switch between the different cells we need a easy way to do so, we want to use only four inputs since that gives us sixteen options, so we create a four to sixteen decoder, shown in the code below.

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

entity decoder_four_to_sixteen is
	port(
		grygorczuk_address: in std_logic_vector(3 downto 0);
		grygorczuk_decoder_output : out std_logic_vector(15 downto 0)
		);
	end decoder_four_to_sixteen;

	architecture decoder_four_to_sixteen_logic of decoder_four_to_sixteen is

	begin

	with grygorczuk_address select
		grygorczuk_decoder_output <= "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 decoder_four_to_sixteen_logic;
```

Code 1. Four to Sixteen Decoder

This code will allow us to use four switches to accesses sixteen different SRAM cells.

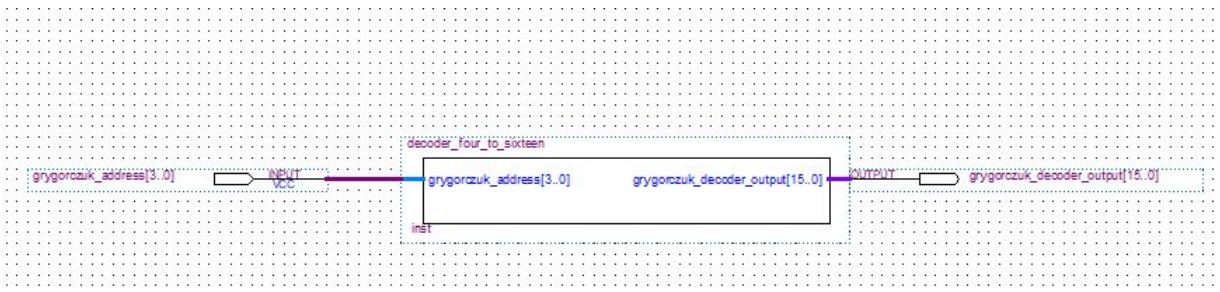

Figure 13. Four to Sixteen Decoder

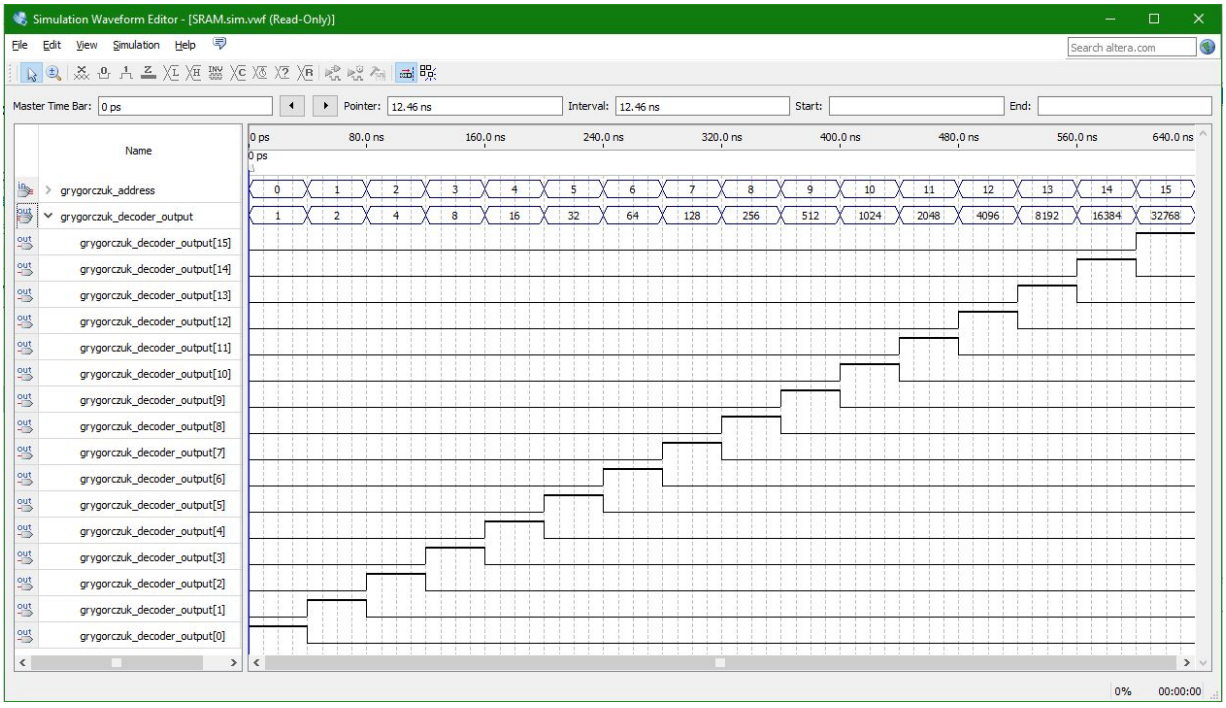

Figure 14. Four to Sixteen Decoder Waveform

SRAM 16x4:

Once we have the decoder and the SRAM 16x1 completed we can put the two together to create the SRAM 16x4. This will allow us to store an array of four bits, and recall that array at any moment. Let's look over the inputs we have now, there's the four bits of input that will be stored in the four SRAM cells, there the four bit address operation code which selects which row of the cells to save/read the input form/to. There is the Write Enable that lets the chips save the input. The Select Chip which works in conjunction to the write enable. And finally there is the output enable which allows the output to be sent from the SRAM to some other component. Below is the block diagram and the waveform that show its functionality.

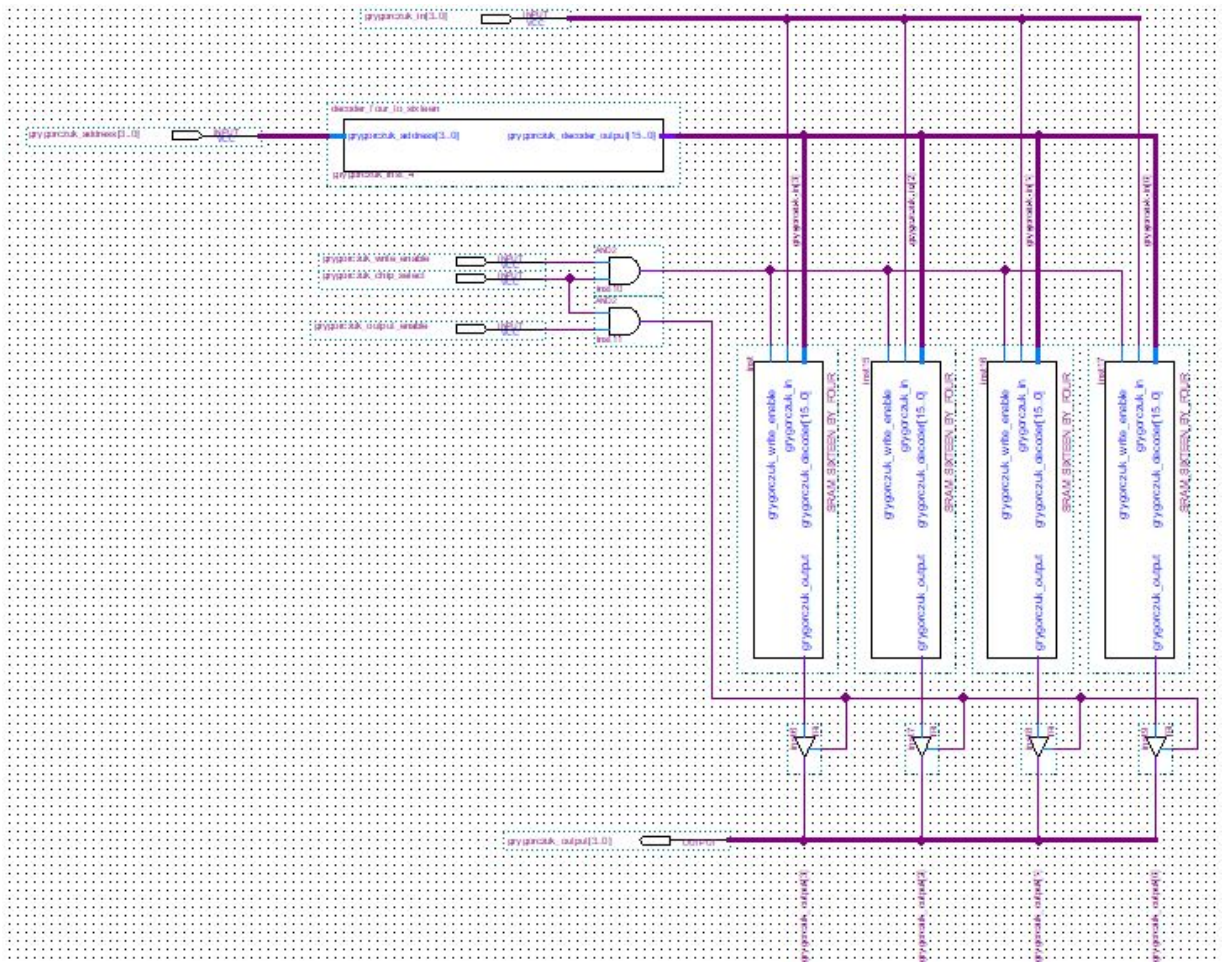

Figure 15. SRAM 16x4

As shown in Figure 13, the system takes zeroth bit from the left and the nth bit from the right, n being third in this case.

Figure 16. SRAM 16x4 Waveform

In Figure 16 you can see all the times that the memory gets saved in red, any time it's being displayed in green. Anytime the buffer is on because the Chip Select is off in dark blue, anytime it's off because the output enable is off in yellow and teal when both Chip Select and Output Enable are off.

In Figure 16, I access two different rows, 4 and 7 and I save different values into them and read them without modifying the memory stored. It works just like a Master Slave Flip Flop but with four different cells assessed at the same time and with ability to choose an array of cells.

Seven Segment Display:

Next we had to create a way to display the results of these actions on the board, we are going to do that by displaying them on a seven segment display in hexadecimal. Below is the code used to do that.

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

entity decoder_and_seven_segment_display is
	port(
		grygorczuk_input: in std_logic_vector(3 downto 0);
		grygorczuk_a, grygorczuk_b, grygorczuk_c, grygorczuk_d, grygorczuk_e, grygorczuk_f, grygorczuk_g: out
std_logic
		);
end decoder_and_seven_segment_display;

architecture decoder_and_seven_segment_display_logic of decoder_and_seven_segment_display is

signal grygorczuk_data : std_logic_vector(6 downto 0);

begin

	process (grygorczuk_input)
		begin
			case grygorczuk_input is
			when "0000" =>
				grygorczuk_data <= "0000001";
			when "0001" =>
				grygorczuk_data <= "1001111";
			when "0010" =>
				grygorczuk_data <= "0010010";
```

```
                        when "0011" =>
                                grygorczuk_data <= "0000110";
                        when "0100" =>
                                grygorczuk_data <= "1001100";
                        when "0101" =>
                                grygorczuk_data <= "0001000";
                        when "0110" =>
                                grygorczuk_data <= "0100000";
                        when "0111" =>
                                grygorczuk_data <= "0001111";
                        when "1000" =>
                                grygorczuk_data <= "0000000";
                        when "1001" =>
                                grygorczuk_data <= "0001100";
                        when "1010" =>
                                grygorczuk_data <= "0001000";
                        when "1011" =>
                                grygorczuk_data <= "1100000";
                        when "1100" =>
                                grygorczuk_data <= "0110001";
                        when "1101" =>
                                grygorczuk_data <= "1000010";
                        when "1110" =>
                                grygorczuk_data <= "0110000";
                        when "1111" =>
                                grygorczuk_data <= "0111000";
                        when others =>
                                NULL;
                end case;
        end process;

        grygorczuk_a <= grygorczuk_data(6);
        grygorczuk_b <= grygorczuk_data(5);
        grygorczuk_c <= grygorczuk_data(4);
        grygorczuk_d <= grygorczuk_data(3);
        grygorczuk_e <= grygorczuk_data(2);
        grygorczuk_f <= grygorczuk_data(1);
        grygorczuk_g <= grygorczuk_data(0);

end decoder_and_seven_segment_display_logic;
```

Code 2. Seven Segment Display

ABCDEFG	Seven Segment Display
0000001	0
1001111	1
0010010	2
0000110	3
1001100	4
0001000	5

0100000	6
0001111	7
0000000	8
0001100	9
0001000	A
1100000	b
0110001	C
1000010	D
0110000	E
0111000	F

Tabel 6. Seven Segment Hexadecimal

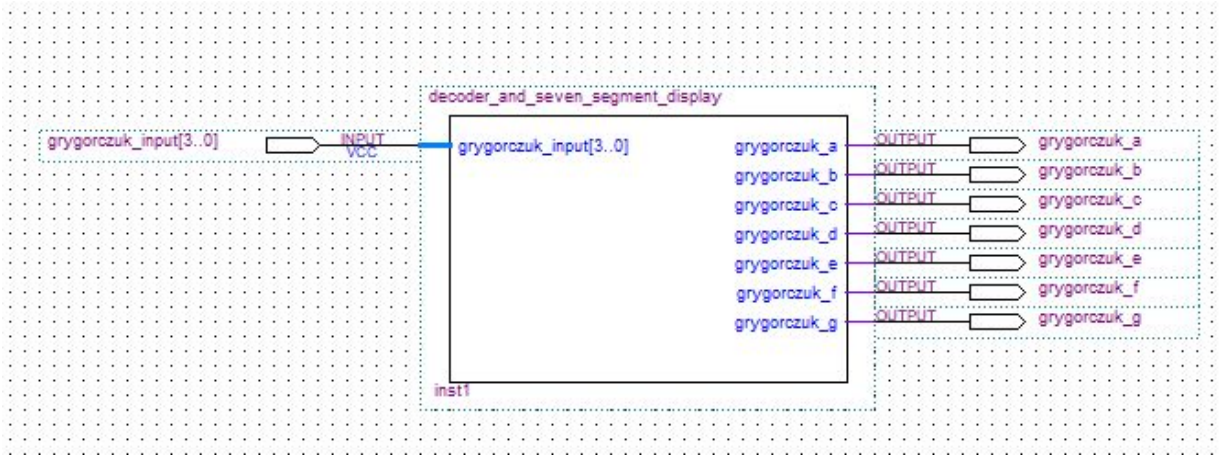

Figure 17. Seven Segment Display

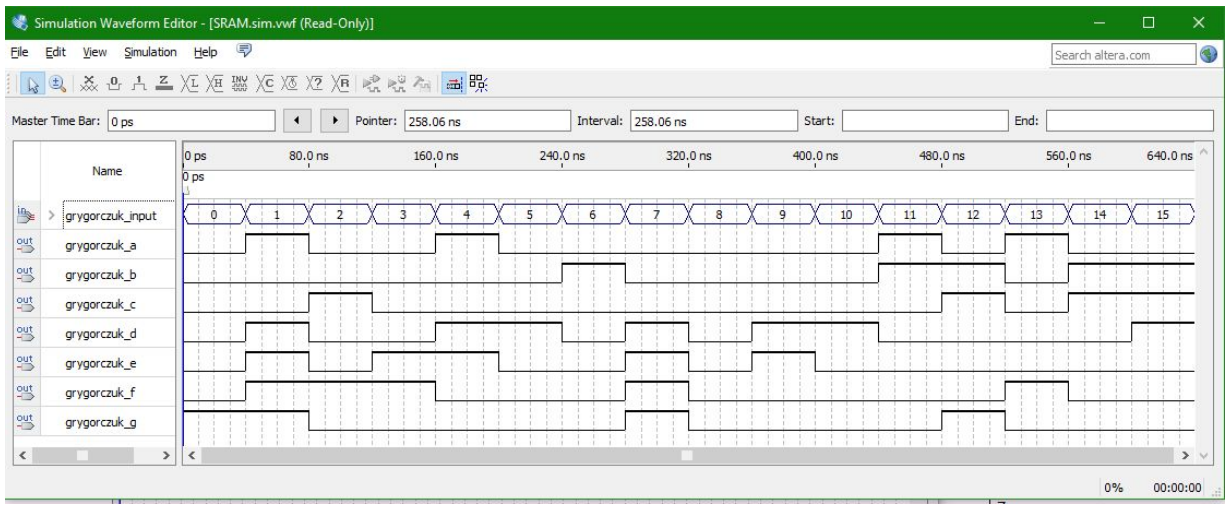

Figure 18. Seven Segment Display Waveform

You can compare Figure 18 to Table 6 to see that the outputs are giving out appropriate symbols to the seven segment display.

SRAM 16x4 and 16x8 With Seven Segment Display:

Below is what one SRAM 16x4 looks like when set up to the seven segment display.

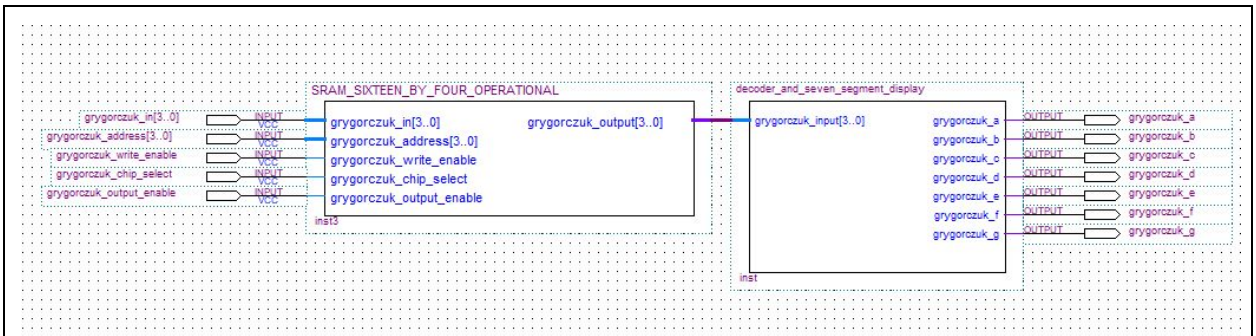

Figure 19. SRAM 16x4

Next step we take to build up towards SRAM 16x32 is we take two of these SRAM 16x4 to make a 16x8. Shown below

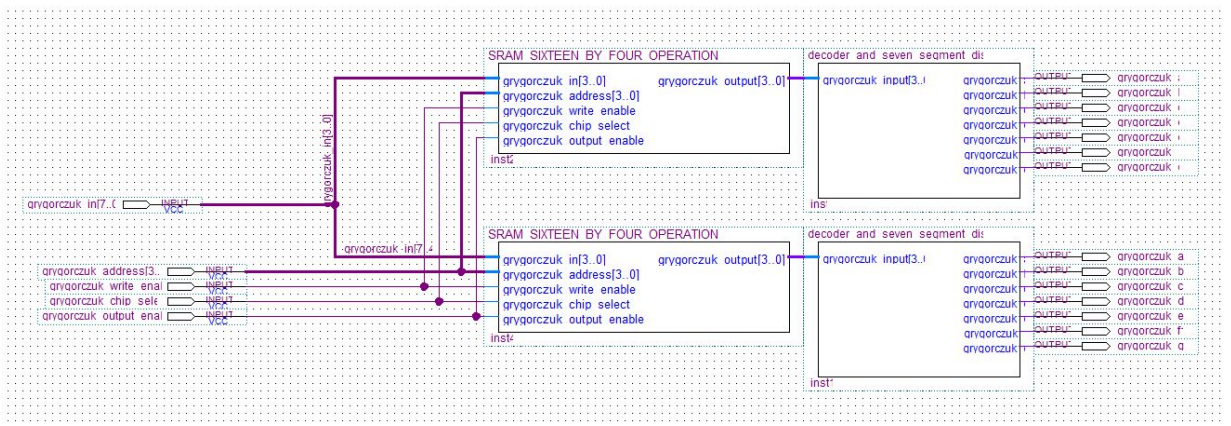

Figure 21. SRAM 16x8

SRAM 16x32:

Once I have the 16x8 I use four of those to create the final build of the 16x32. Since we have such a large number of bits to be placed we're going to save them eight bits at a time. To do this I've set up signals for the keys and when the key is pressed and write enable is on it will write to the 0-7 range, 8-15 range, 16-23 range, and 24-31 range. The outputs on the side is to make sure that the LEDs are off during the program's run time.

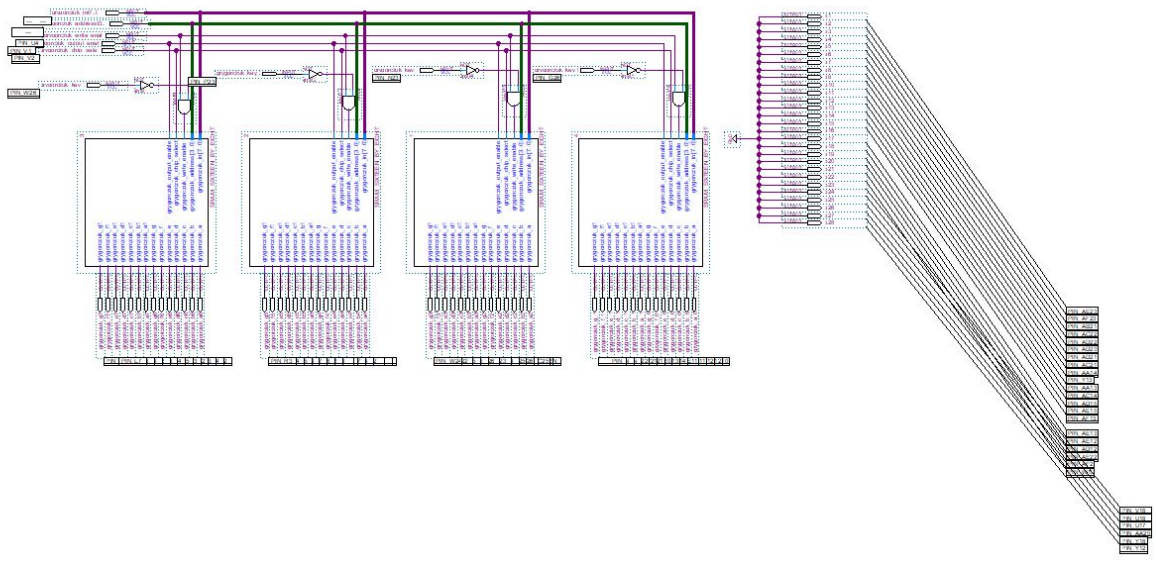

Figure 22. SRAM 16x32

Pinouts

To, Location
grygorczuk_in[0], PIN_N25
grygorczuk_in[1], PIN_N26
grygorczuk_in[2], PIN_P25
grygorczuk_in[3], PIN_AE14
grygorczuk_in[4], PIN_AF14
grygorczuk_in[5], PIN_AD13
grygorczuk_in[6], PIN_AC13
grygorczuk_in[7], PIN_C13
grygorczuk_key_0, PIN_G26
grygorczuk_key_1, PIN_N23
grygorczuk_key_2, PIN_P23
grygorczuk_key_3, PIN_W26
grygorczuk_address[0], PIN_B13
grygorczuk_address[1], PIN_A13
grygorczuk_address[2], PIN_N1
grygorczuk_address[3], PIN_P1
grygorczuk_write_enable, PIN_U4
grygorczuk_chip_select, PIN_V2
grygorczuk_output_enable, PIN_V1
grygorczuk_a_0, PIN_AF10
grygorczuk_b_0, PIN_AB12
grygorczuk_c_0, PIN_AC12
grygorczuk_d_0, PIN_AD11
grygorczuk_e_0, PIN_AE11
grygorczuk_f_0, PIN_V14
grygorczuk_g_0, PIN_V13
grygorczuk_a_1, PIN_V20
grygorczuk_b_1, PIN_V21
grygorczuk_c_1, PIN_W21
grygorczuk_d_1, PIN_Y22
grygorczuk_e_1, PIN_AA24
grygorczuk_f_1, PIN_AA23
grygorczuk_g_1, PIN_AB24
grygorczuk_a2, PIN_AB23
grygorczuk_b2, PIN_V22
grygorczuk_c2, PIN_AC25
grygorczuk_d2, PIN_AC26
grygorczuk_e2, PIN_AB26
grygorczuk_f2, PIN_AB25
grygorczuk_g2, PIN_Y24
grygorczuk_a3, PIN_Y23
grygorczuk_b3, PIN_AA25
grygorczuk_c3, PIN_AA26
grygorczuk_d3, PIN_Y26
grygorczuk_e3, PIN_Y25
grygorczuk_f3, PIN_U22
grygorczuk_g3, PIN_W24
grygorczuk_a4, PIN_U9
grygorczuk_b4, PIN_U1
grygorczuk_c4, PIN_U2
grygorczuk_d4, PIN_T4
grygorczuk_e4, PIN_R7
grygorczuk_f4, PIN_R6

```
grygorczuk_g4, PIN_T3

grygorczuk_a5, PIN_T2
grygorczuk_b5, PIN_P6
grygorczuk_c5, PIN_P7
grygorczuk_d5, PIN_T9
grygorczuk_e5, PIN_R5
grygorczuk_f5, PIN_R4
grygorczuk_g5, PIN_R3

grygorczuk_a6, PIN_R2
grygorczuk_b6, PIN_P4
grygorczuk_c6, PIN_P3
grygorczuk_d6, PIN_M2
grygorczuk_e6, PIN_M3
grygorczuk_f6, PIN_M5
grygorczuk_g6, PIN_M4

grygorczuk_a7, PIN_L3
grygorczuk_b7, PIN_L2
grygorczuk_c7, PIN_L9
grygorczuk_d7, PIN_L6
grygorczuk_e7, PIN_L7
grygorczuk_f7, PIN_P9
grygorczuk_g7, PIN_N9

o1, PIN_AE23
o2, PIN_AF23
o3, PIN_AB21
o4, PIN_AC22
o5, PIN_AD22
o6, PIN_AD23
o7, PIN_AD21
o8, PIN_AC21
o9, PIN_AA14
o10, PIN_Y13
o11, PIN_AA13
o12, PIN_AC14
o13, PIN_AD15
o14, PIN_AE15
o15, PIN_AF13
o17, PIN_AE13
o18, PIN_AE12
o19, PIN_AD12
o20, PIN_AE22
o21, PIN_AF22
o22, PIN_W19
o23, PIN_V18
o24, PIN_U18
o25, PIN_U17
o26, PIN_AA20
o27, PIN_Y18
o28, PIN_Y12
```

Code. Pin Out

Conclusion:

This project showed basic memory can be created and how you can access and store the bits at different addresses with the help of two D-Latches set up in a Master Slave Flip Flop. It showed the difference between a Latch and a Flip Flop, where the first changes state based on current state, high or low, while the other changes state based on rising or falling edge. It also challenged us to create a interesting way to store bytes using a limited input space having to use only eight inputs to dictate thirty two outputs that are started on the SRAM 16x32.