Report for 8-bit RAM design and spice

IN5260 - Low Power IoT nodes

Author: Kun Zhu

Supervisor: Snorre Aunet

Spring 2022

Contents

1. Abstract

This project is to build an 8-bit RAM and investigate SPICE simulation of 1-bit cell of the memory circuit about the leakage power consumption. The project is divided into 2 main parts, the SPICE simulation and Verilog wave diagram simulation.

The SPICE simulation is built upon subcircuit to the transistor level.

The Verilog/HDL output is achieved by a state machine, which produces the expected read and write functionalities of the circuit.

2. Introduction

Both the SPICE and Verilog are made from the transistor level subcircuits.

2.1 Spice

During operation, or standby mode, IoT devices will always have some power leakage due to current leakage of the transistors. The project rebuilds from the transistor level to make a bit cell for an 8-bit RAM circuit. This part aims to SPICE for the 1 bit cell leakage current.

Circuits built are listed as the following:

- Inverter subcircuit.
- AND gate subcircuit.
- NOR gate subcircuit.
- The bit cell built upon the above mentioned subcircuits.

2.2 HDL

By implementing Verilog code, this part is replicating the 8-bit RAM circuit using HDL, as well starting from the transistor level.

The modules built to achieve the goal is listed as the following:

- The decoder module, which has 3 bits input and 8 bits output. This is connected and controlling the RAM address lines.
- The bit cell module, which is composed has 1 input bit (in), 1 read and write bit (rw) , 1 select bit (sel) and 1 output bit (out).
- The RAM module, which is capable of taking in 8 bit inputs and 8 bits outputs, the rw bit controls read or write state, and the select bits controls the address of read/ write data.
- The state module, which controls the state of read or write that is synchronized with at clock signal.

3. Methods

The tools used for this project are Automatic Integrated Circuit Modelling Spice (AIMSpice) for power consumption and active-hdl for wave diagram simulation.

3.1 AIM-Spice

This tool is very similar to hardware description language, in the tool we need to define inputs, outputs and wires. Then we connect them accordingly to the circuit design. In below, there are the circuit designs made in AIM-Spice. The transistor dimensions are taken from samples provided.

3.1.1 Inverter

The inverter consists of 2 transistors, 1 pMOS and 1 nMos, the input and the output. The circuit and truth table is as below.

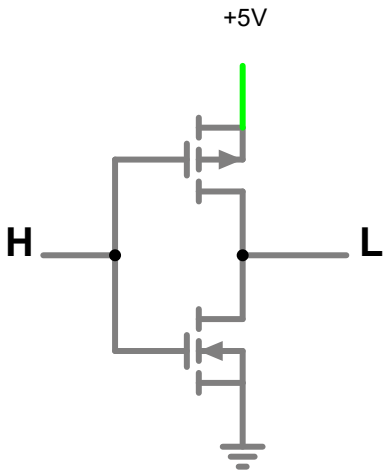


Figure 1 The inverter circuit. (Inverter, n.d.)

Input	output
0	1
1	0

Table 1 The inverter truth table.

In the above figure 1, input connected to the gates of pMOS and nMOS on the left side and output from the drain of the pMOS on the right, the circuit is supplied with vdd to the source of the pMOS and vss connected to the source of the nMOS. The logic relations is shown as Table 1 The inverter truth table.

As in description language, we can see the connection in AIM-Spice as below.

```
1.  *    Drain Gate Source Bulk
2.  *    D   G  S   B
3.  xmp out in vdd vdd pmos1v l=lp w=wp
4.  xmn out in 0 0 nmos1v l=ln w=wn
```

3.1.2 AND gate

The AND gate consists of 3 pMOS and 3 nMOS, there are 2 inputs and 1 output. The circuit and truth table is as below.

Input1	Input2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Table 2 The AND gate truth table.

The circuit design is as below.

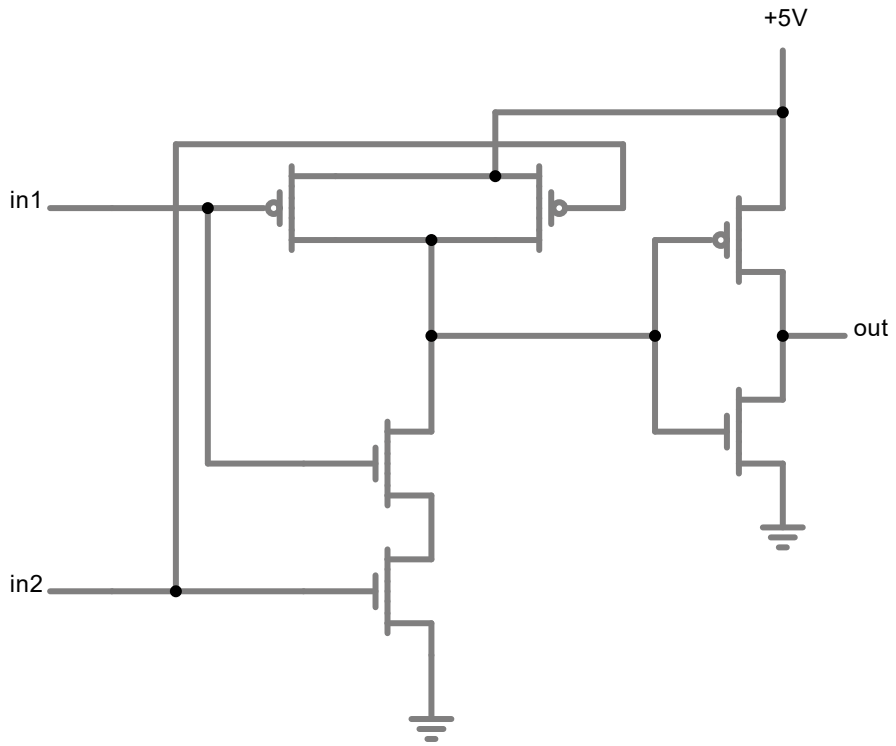


Figure 2 The AND gate circuit.

In the above circuit, there are 2 inputs in1 to pMOS gate and in2 to nMOS gate on the left, and output on the right, vdd is supplied to all the pMOS on the source, and vss connected to source of 2 nMOS. The logic relations is shown as Table 2 The AND gate truth table. above.

The above AND gate circuit is described as below in AIM-Spice.

```
1.  *    Drain Gate Source Bulk
2.  *    D   G  S   B
3.  xmp1 out w1 vdd vdd pmos1v l=lp w=wp
4.  xmn1 out w1 vss vss nmos1v l=ln w=wn
5.
6.  xmp2 w1 in2 vdd vdd pmos1v l=lp w=wp
7.  xmp3 w1 in1 vdd vdd pmos1v l=lp w=wp
8.
9.  xmn2 w1 in1 w2 w2 nmos1v l=ln w=wn
10. xmn3 w2 in2 vss vss nmos1v l=ln w=wn
```

3.1.3 NOR gate

The NOR gate consists of 2 pMOS and 2 nMOS, the truth table and circuit design is show as below.

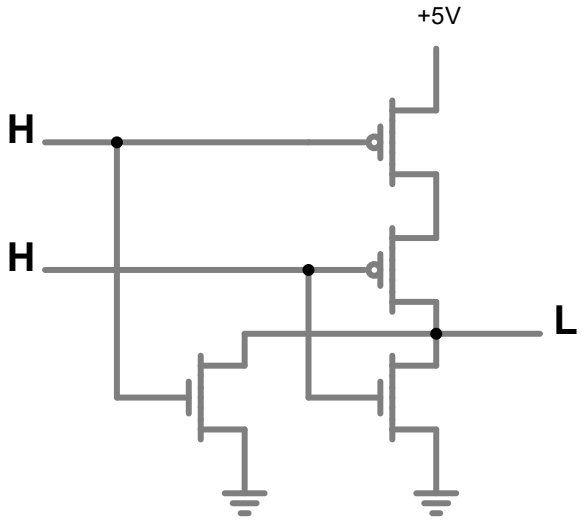


Figure 3 The NOR gate circuit. (NOR gate, n.d.)

Input1	Input2	Output
0	0	1
0	1	0
1	0	0
1	1	0

Table 3 The NOR gate truth table.

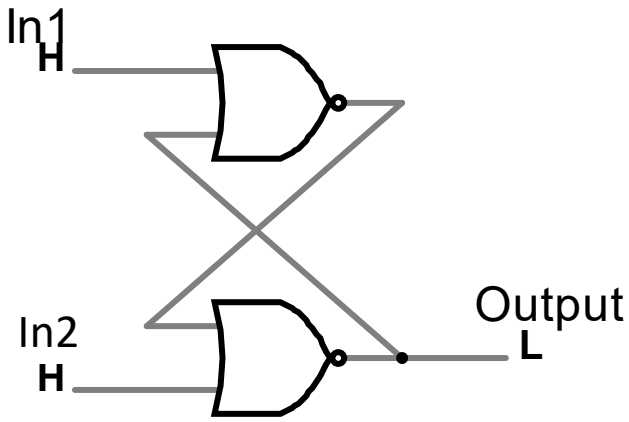
The above figure shows a NOR gate circuit, the circuit performs NOR logic operation, having 2 inputs connected to the gate of the 2 pMOS on the left and output from the pMOS drain on the right. The logic relations is shown as in Table 3 The NOR gate truth table.

The circuit is described as below in AIM-Spice.

```
1.   *    D   G   S   B
2.  xmp1 w1 in2 vdd vdd pmos1v l=lp w=wp
3.  xmp2 out in1 w1 w1 pmos1v l=lp w=wp
4.
5.  xmn1 out in1 vss vss nmos1v l=ln w=wn
6.  xmn2 out in2 vss vss nmos1v l=ln w=wn
```

3.1.4 SR latch

The SR latch consists of 2 NOR gates, 2 inputs and 1 output. The truth table and circuit design is shown as below.



S	R	Q	$\bar{Q}$
0	0	L	L
0	1	0	1
1	0	1	0
1	1	0	0

Table 4 The SR latch truth table.

Figure 4 Gate implementation of the SR latch circuit.

As shown above, the 2 inputs are on the left and output on the right, the logic relations are shown as the in Table 4 The SR latch truth table.

3.1.5 The bit cell

A bit cell consists of several AND gates and SR latch units, there are 1 input, 1 select line (sel) enables the cell, 1 read and write line (rw) controls the mode and 1 output.

When rw is low, the bit cell is in 'remember' state, which 'stores' the value from input, the sel is in the meantime controlling whether to enable the operation.

Similarly, when rw is high, the bit cell outputs the value that was remember in the 'storing' state.

The bit cell circuit is shown as below.

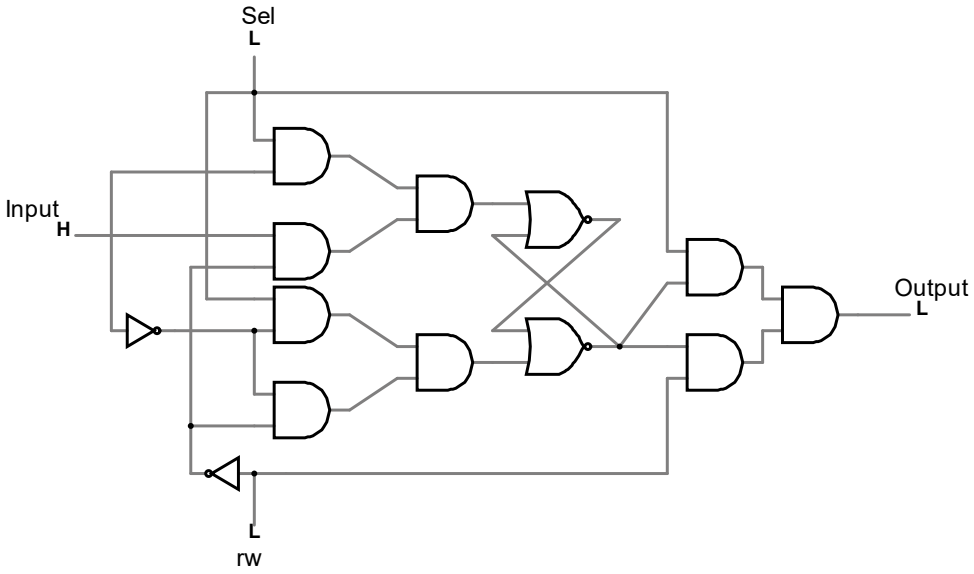


Figure 5 Gate implementation the bit cell circuit. (M. Morris Mano, MEMORY DECODING, 2011)

The above figure shows the gated circuit of 1 bit cell, which has 1 input on the left, sel the select line on the top, rw line the read and write enable on the bottom and the output on the right.

The circuit is described as below in AIM-Spice.

```
1.  * in out vdd vss
2.  xno1 in nin vdd 0 invertersub
3.  xno2 rw nrw vdd 0 invertersub
4.
5.  *   in  in out vdd vss
6.  xa11 sel in w11 vdd 0 andsub
7.  xa12 nrw in w12 vdd 0 andsub
8.  xa13 w11 w12 w13 vdd 0 andsub
9.
10. xa21 sel nin w21 vdd 0 andsub
11. xa22 nrw nin w22 vdd 0 andsub
12. xa23 w21 w22 w23 vdd 0 andsub
13.
14. * sr latch
15. *   in  in out vdd vss
16. xnr1 w13 w4 w3 vdd 0 norsub
17. xnr2 w23 w3 w4 vdd 0 norsub
18.
19. * final and Gate
20. *   in  in out vdd vss
21. xa31 sel w4 w31 vdd 0 andsub
22. xa32 rw w4 w32 vdd 0 andsub
23. xa33 w31 w32 out vdd 0 andsub
24.
```

3.2 Active-HDL

The Verilog design is a complete 8*8 RAM array, which can remember 8 bits input data in writing mode and output when in reading mode by selected address. Thus, a decoder is introduced to decode 3 bits address input and output 8 bits, the output and is directly connected to the sel line in a bit cell and can choose the corresponding bit cells to read or write data.

The idea is to build modules and combine them into the RAM structure. The modules includes:

- The 3 – 8 decoder module.
- The bit cell module.
- The byte module, this is a row of 8 bit cells.
- The RAM module, this is an 8 rows of the byte module.
- The state machine module, this is controlling the synchronized read or write signal.

3.2.1 3-8 decoder in Verilog

The 3 – 8 decoder consists of only 2 types of logic gates, the inverter and AND agte. Each the AND gates produces 1 bit of output, that is 8 outputs in total.

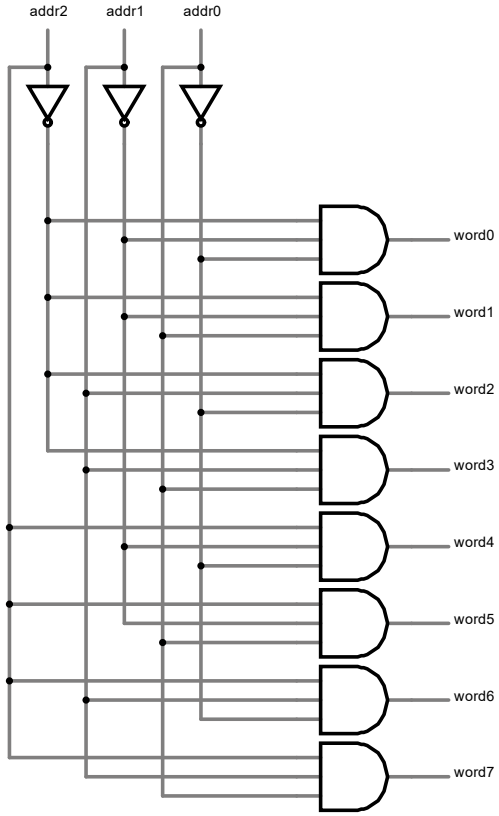


Figure 6 Gate implementation of the 3 - 8 decoder circuit.

As shown above, the input bits are addr0, addr1 and addr2, these bits are the usually controlled by cpu if there is any. On the right we see word0 to word7, these bits select which row in the 8*8 RAM matrices to perform read/ write operations.

In the Verilog syntax, the decoder is described as such.

```
1.  module decoder(id, od);
2.
3.    input [2:0]id;
4.    output [7:0]od;
5.    and a0(od[0], ~id[0], ~id[1], ~id[2]);
6.    and a1(od[1], ~id[0], ~id[1], id[2]);
7.    and a2(od[2], ~id[0], id[1], ~id[2]);
8.    and a3(od[3], ~id[0], id[1], id[2]);
9.    and a4(od[4], id[0], ~id[1], ~id[2]);
10.   and a5(od[5], id[0], ~id[1], id[2]);
11.   and a6(od[6], id[0], id[1], ~id[2]);
12.   and a7(od[7], id[0], id[1], id[2]);
13.
14. endmodule
```

3.2.2 The bit cell in Verilog

The circuit is the same as in Figure 5 Gate implementation the bit cell circuit.

The code describes the circuit in Verilog is as below.

```
1.   module bc(in, rw, sel, out);
2.
```

```
3.    input in, rw, sel;
4.    output out;
5.
6.    wire w1, w2, w3, w4;
7.    and a1(w1, sel, in, ~rw);
8.    and a2(w2, sel, ~rw, ~in);
9.    nor nr1(w3, w1, w4);
10.   nor nr2(w4, w2, w3);
11.   and a3(out, sel, w4, rw);
12.
13. endmodule
```

3.2.3 The byte module

The byte module is a row array of 8 bit cells which can take 8 bits input and store it or output it. The address line is controlled by the decoder and, read or write mode is controlled by the rw line.

The circuit design is shown as below.

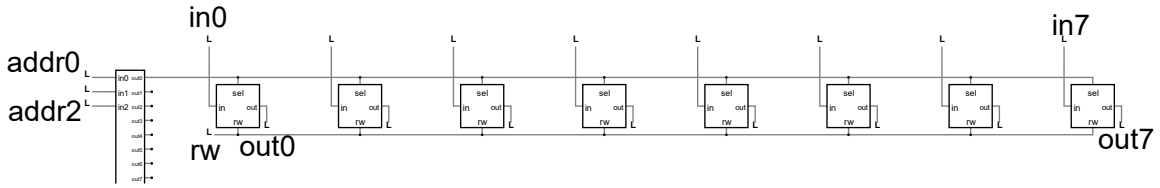


Figure 7 The byte module.

The above figure is the byte module, which consists of 8 bit cells, the bit cells are arrange in a row, the left side is the output of the decoder, when the relevant bits, in this case, when word0 is high this row of bit cells will be 'selected' to perform read or write operation depending on the rw. The read or write operation will either write all the 8 input bits from in0 to in7 to the row or read the pervious stored value from the input. The initial should be 0.

The code in Verilog is structured as below.

```
1.   module bt(inb, rwb, selb, outb);
2.
3.    input [7:0]inb;
4.    input selb;
5.    input rwb;
6.    output [7:0]outb;
7.
8.    // module bc(in, rw, sel, out);
9.    bc clm0(.out(outb[0]), .in(inb[0]), .rw(rwb), .sel(selb));
10.   bc clm1(.out(outb[1]), .in(inb[1]), .rw(rwb), .sel(selb));
11.   bc clm2(.out(outb[2]), .in(inb[2]), .rw(rwb), .sel(selb));
12.   bc clm3(.out(outb[3]), .in(inb[3]), .rw(rwb), .sel(selb));
13.   bc clm4(.out(outb[4]), .in(inb[4]), .rw(rwb), .sel(selb));
14.   bc clm5(.out(outb[5]), .in(inb[5]), .rw(rwb), .sel(selb));
15.   bc clm6(.out(outb[6]), .in(inb[6]), .rw(rwb), .sel(selb));
16.   bc clm7(.out(outb[7]), .in(inb[7]), .rw(rwb), .sel(selb));
17.
18. endmodule
```

3.2.4 The RAM circuit and design Verilog implementation

The RAM is composed of 2 parts, the 3-8 decoder and the 8*8 bit-cell arrays.

Each input pin is connected to its own columns. All sel pins of the same row is connected to the word (output bit of the decoder). All the rw pins are connected to rw to enable or disable read and write

for the memory unit. All bits from the same column are passing through the OR gates, and produce 1 bit output, in total of 8 bits output.

When rw is low, the circuit is in the write mode, which takes the 8 bits input and write it into 8 of the bit cell controlled by address/ decoder. For rw is high, the circuit turns into the read mode, which outputs 8 bits output and the data comes from the bit cells that previously were written to, the selection is controlled by the address/ decoder.

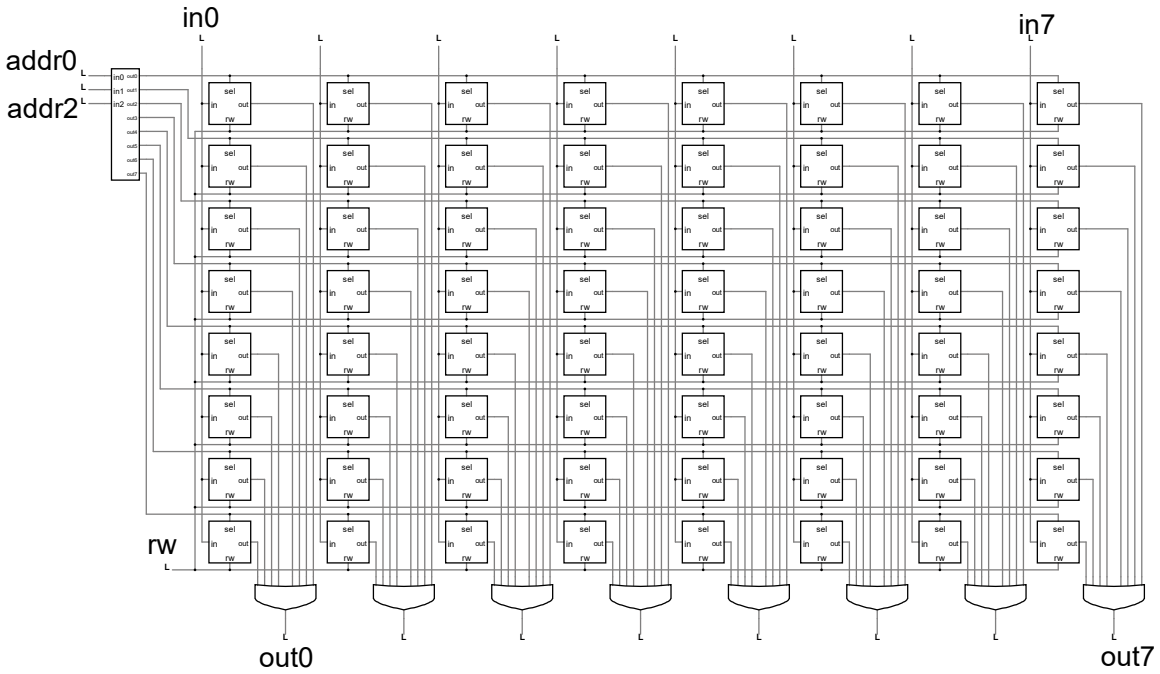


Figure 8 The RAM circuit. (M. Morris Mano, Digital Design, 2011)

In Figure 8 The RAM circuit. , we can see 8 rows of the byte module from Figure 7 The byte module. the input can now write to and read from different row, but one row at a time, the row to operate is selected by the address input. We want the output to show as 1 bit, so all 8 outputs of each column are connected to OR gates (there supposed to be 2 levels of OR operation and 3 OR gates, I used one 8-pin OR gate for simplicity). The Verilog implementation is an implementation of 8 rows of the byte module. Due to the long lines, the code is shown in the Appendices.

3.2.5 Finite state machine

The state machine design is based on the state truth table.

enable	current_state	rwin	next_state	rwout
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

The idle state is always ‘read’, when enable line is toggled low, the state machine stays in read mode. By using Karnaugh map method, the logic relations are

$$\text{next\_state} = rwin + \overline{enable}$$

$$\text{rwout} = rwin * \overline{enable}$$

The state machine diagram is shown as below, there are 2 sates: 0, 1.

The rwin pin to control read or write and output is always high when ~enabled. To synchronize the clock signal, a D flipflop is introduced, the design is shown as below.

The d-flipflop circuit and state machine circuit.

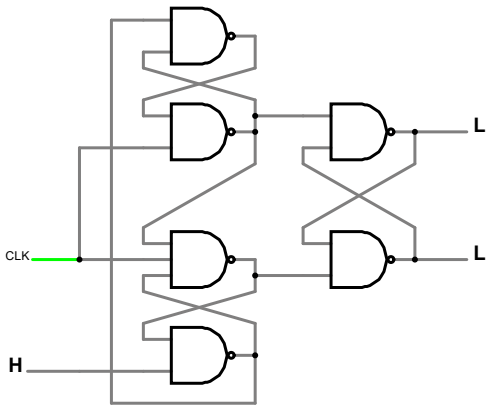


Figure 9 Gate implementation of the D flipflop.

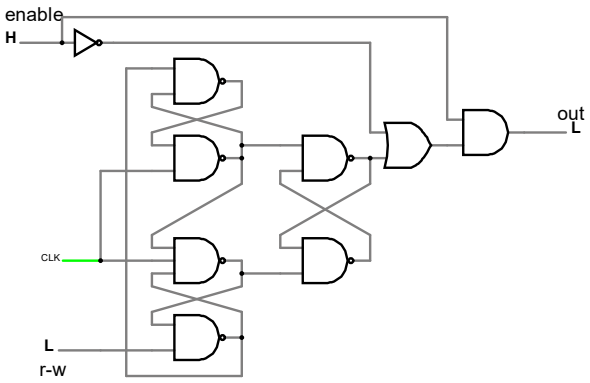


Figure 10 Gate implementation of the state machine.

The above figure 9 shows a circuit design for a D flipflop, with the help of the D flipflop we are able to synchronize the r-w signal and put it in the ‘states’. I added an enable line to turn on and off the state machine. In Figure 10 is the state machine circuit, we are seeing the enable, r-w and clk signal, the output is on the right. The result will simply be clock rising edge triggered signal that follows the r-w.

The code for D flipflop module:

```
1.   module dflipflop(data, clk, qata);
2.
3.    input data, clk;
4.    output qata;
5.
6.    wire w1, w2, w3, w4, w5, w6;
7.    nand a1(w1, w2, w4);
8.    nand a2(w2, w1, clk);
9.    nand a3(w3, w2, clk, w4);
10.   nand a4(w4, w3, data);
11.   nand a5(qata, w2, w6);
12.   nand a6(w6, w3, qata);
13.
14. Endmodule
```

The state machine module:

```
1.  module state(rwin, sclk, enable, datain, saddr, dataout);
```

```
2.    input sclk, enable, rwin;
3.    input [0:7]datain;
4.    input [2:0]saddr;
5.    output [7:0]dataout;
6.    wire ddata, dout, sout;
7.
8.    dflipflop states(.qata(dout), .clk(sclk), .data(rwin));
9.    or dd(sout, dout, ~enable);
10.   and fsm(ddata, sout, enable);
11.   ram ss(.outs(dataout), .ins(datain), .rws(dout), .addr(saddr));
12.
13. endmodule
```

4. Results

The results include 2 parts, the Spice simulation and wave diagram of the state machine from the Verilog solution.

4.1 The spice simulation

There are 5 corner data: tt, ff, fs, ss, sf at different input voltage level. There is 1 reference simulation set at the max voltage and input signals are static.

4.1.1 The static input simulation.

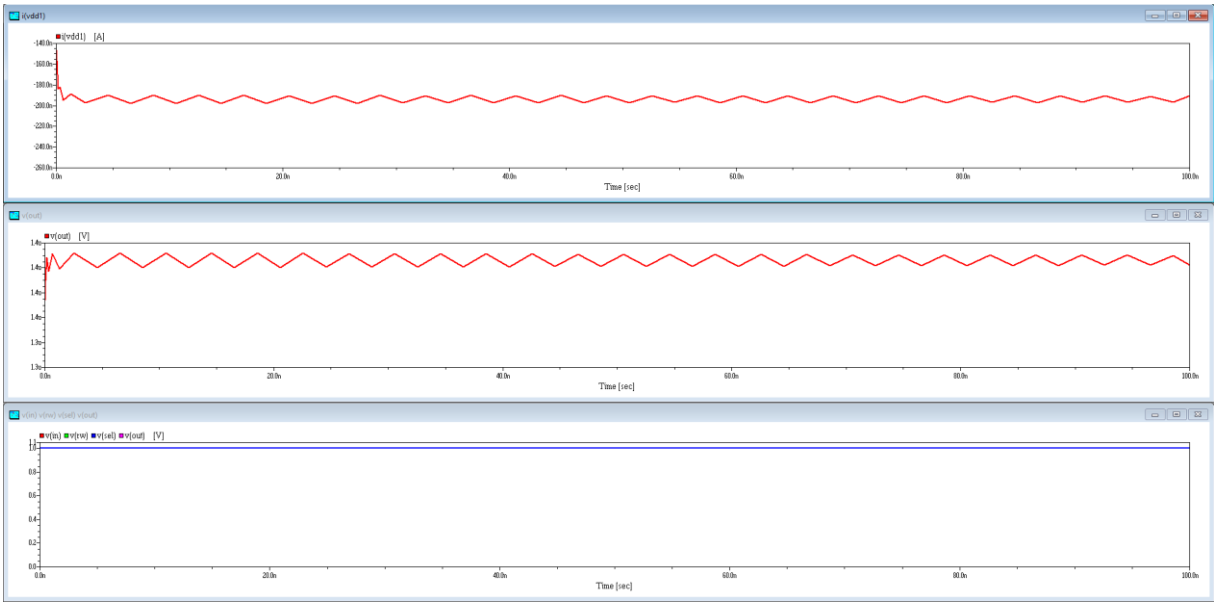


Figure 11 The simulation results for tt corner test with static inputs.

This figure 11, shows the tt corner test results, the plotted values are voltage of all inputs and output, as well as the current of from the power source (vdd). The circuit is simulating the operation under static input values at supply voltage at 1 V.

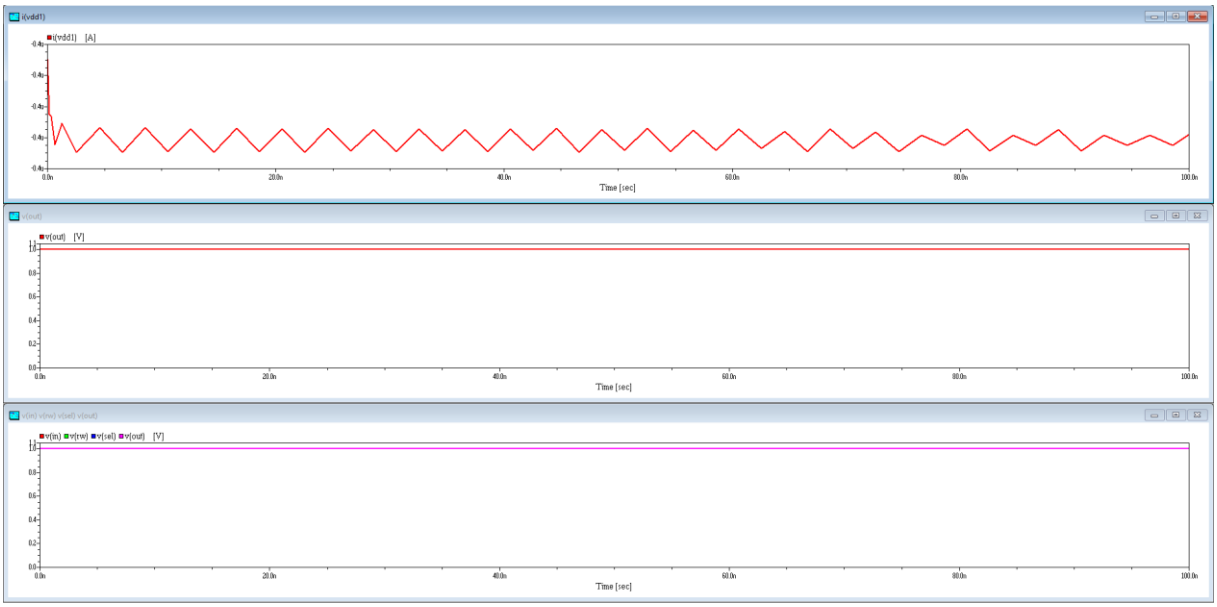


Figure 12 The simulation results for ff corner test with static inputs.

This figure 12, shows the ff corner test results, the plotted values are voltage of all inputs and output, as well as the current of from the power source (vdd). The circuit is simulating the operation under static input values at supply voltage at 1 V. Spice data of the leakage current with input, rw, sel signal line at static 1V.

Corner	I(vdd1)/ A
tt	-190n to -200n
ff	-0.4u
fs	-0.4u
ss	-100n
sf	-106n to -108n

4.1.2 Corner tests at 1V with pulsating inputs

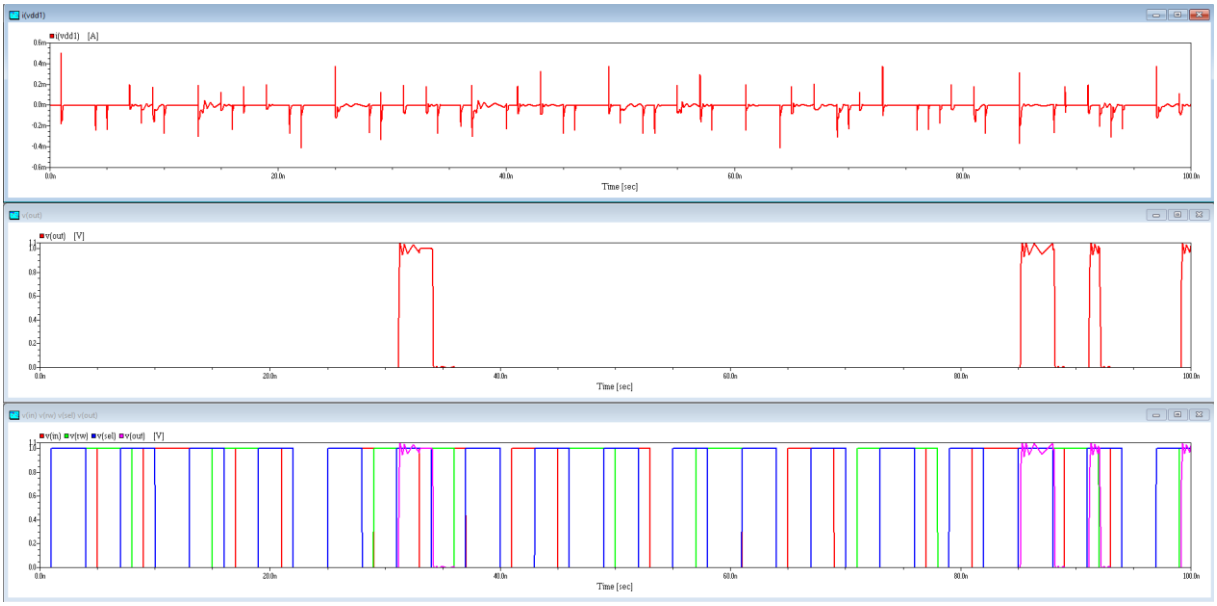


Figure 13 The plots for leakage current, inputs and outputs with the tt corner.

This figure 13, shows the tt corner test results, the plotted values are voltage of all inputs and output, as well as the current of from the power source (vdd). The circuit is simulating the operation under pulsating input values with different frequency at supply voltage at 1V.

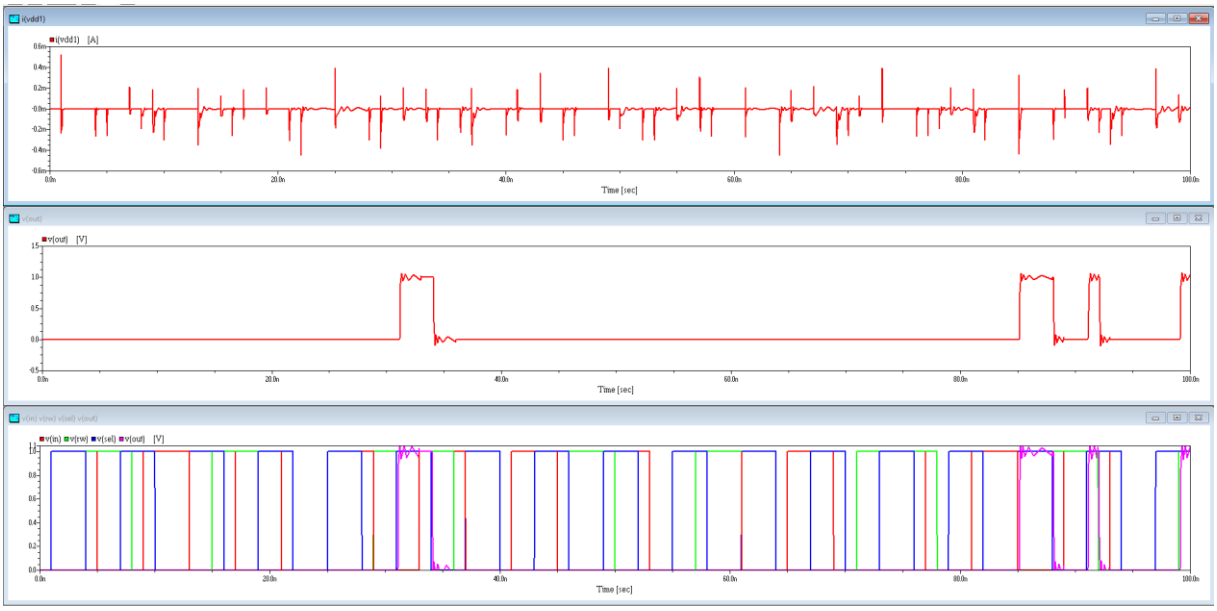


Figure 14 The plots for leakage current, inputs and outputs with the ff corner.

This figure 14, shows the ff corner test results, the plotted values are voltage of all inputs and output, as well as the current of from the power source (vdd). The circuit is simulating the operation under pulsating input values with different frequency at supply voltage at 1V. The leakage current is taken and read from the flat line sections from the I(vdd) plot. Spice data of the leakage current with input, rw, sel signal line at static 1V.

Corner	I(vdd1)/ A
tt	-1.0u
ff	-5.1u
fs	-0.3u
ss	1u
sf	1u to 2u

4.1.3 Corner tests at 0.5V with pulse input

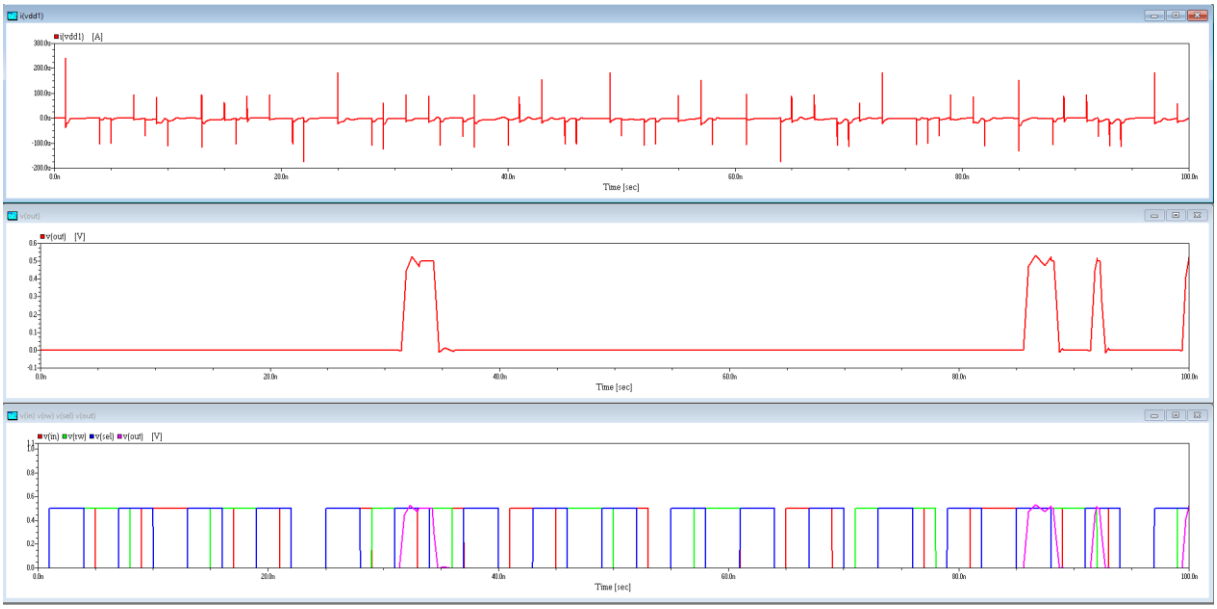


Figure 15 The tt corner test at minimum effective voltage.

This figure 15, shows the tt corner test results, the plotted values are voltage of all inputs and output, as well as the current of from the power source (vdd). The circuit is simulating the operation under pulsating input values with different frequency at supply voltage at 0.5V, the minimum effective input voltage as tested. The leakage current is taken and read from the flat line sections from the I(vdd) plot.

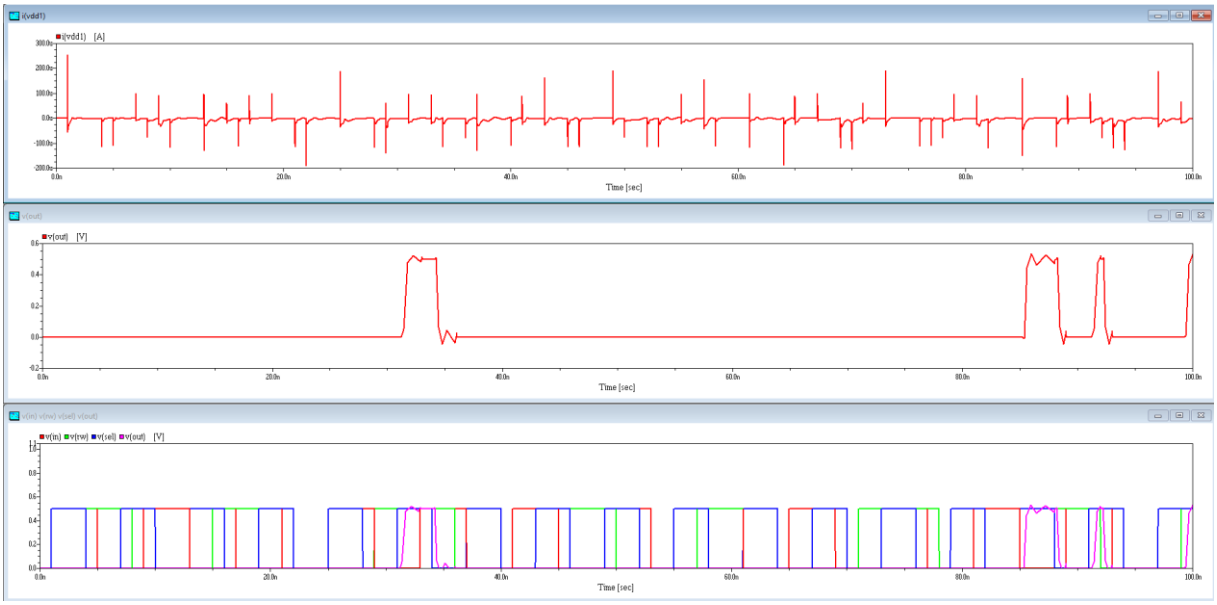


Figure 16 The ff corner test at minimum effective voltage.

This figure 16, shows the ff corner test results, the plotted values are voltage of all inputs and output, as well as the current of from the power source (vdd). The circuit is simulating the operation under pulsating input values with different frequency at supply voltage at 0.5V, the minimum effective input voltage as tested. The leakage current is taken and read from the flat line sections from the I(vdd) plot. Spice data of the leakage current with input, rw, sel signal line at static 0.5V.

Corner	I(vdd1)/ A
Tt	-0.5u to 0.5u
Ff	-0.5u to 0
Fs	0 to 0.2u
Ss	0 to 0.4u
Sf	0.1u

4.1.4 Corner tests at 0.4V with pulsating inputs

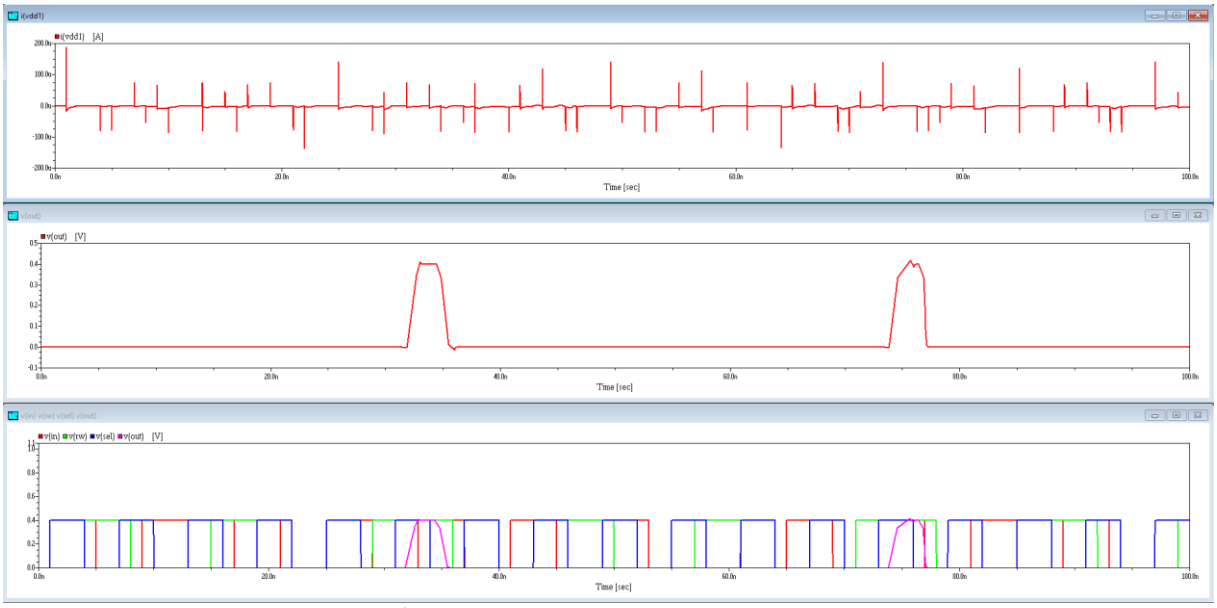


Figure 17 The tt corner test with missing/ inefficient outputs.

This figure 17, shows the tt corner test results, the plotted values are voltage of all inputs and output, as well as the current of from the power source (vdd). The circuit is simulating the operation under pulsating input values with different frequency at supply voltage at 0.4V, the minimum effective input voltage as tested. The leakage current is taken and read from the flat line sections from the I(vdd) plot. In this figure, we can see that we are experiencing missing outputs, these cases are seen as inefficient results. So we say that 0.5V is the minimum we can test for.

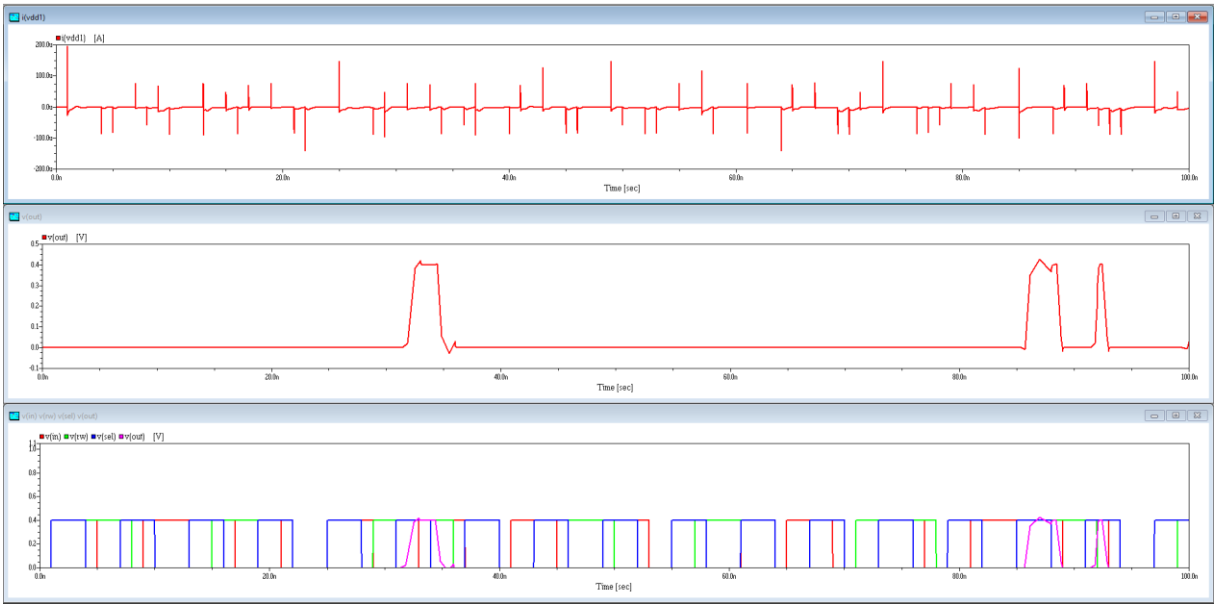


Figure 18 The ff corner with missing/ inefficient outputs.

This figure 18, shows the ff corner test results, the plotted values are voltage of all inputs and output, as well as the current of from the power source (vdd). The circuit is simulating the operation under pulsating input values with different frequency at supply voltage at 0.4V, the minimum effective input voltage as tested. The leakage current is taken and read from the flat line sections from the I(vdd) plot. In this figure, we can see that we are experiencing missing outputs, these cases are seen as inefficient results. So we say that 0.5V is the minimum we can test for.

As shown above, we can see that when input voltage is as low as 0.4V, we will experience inefficient / missing output. The below table shows the other corner test data.

Corner	I(vdd1)/ A
tt	Not valid/ missing output
ff	Not valid
fs	-0.6u to -0.4u
ss	N/A
sf	N/A

4.2 The state machine wave diagram simulation

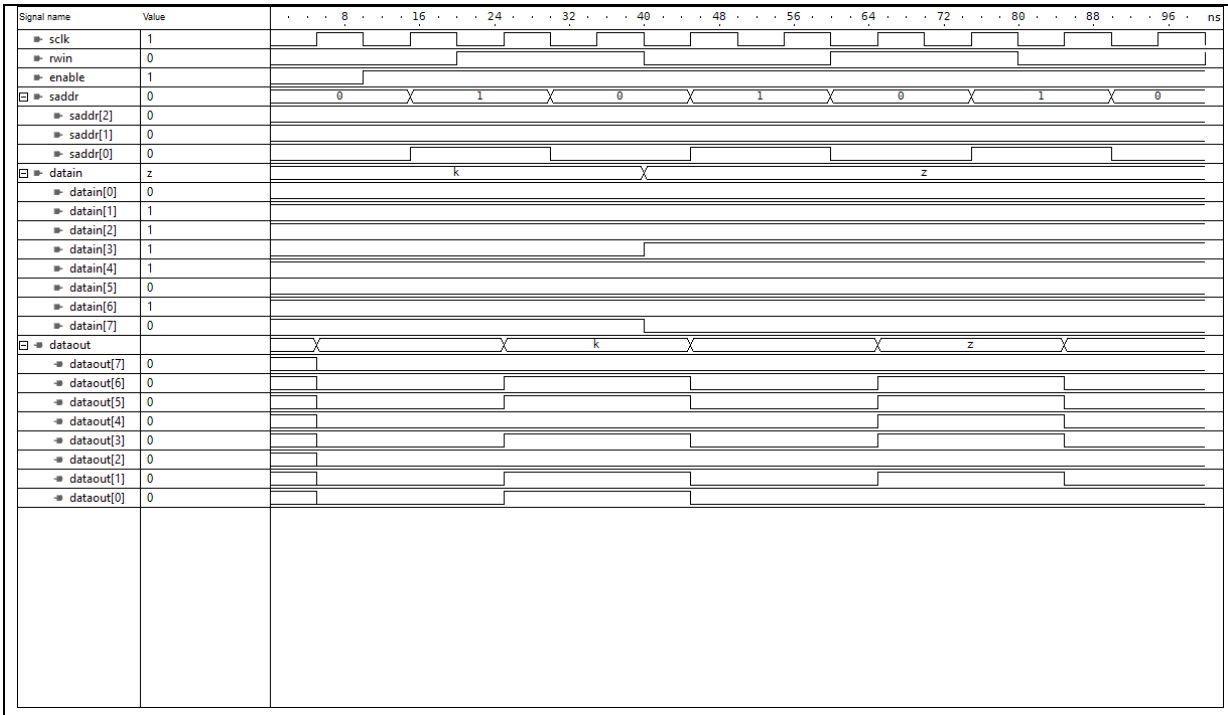


Figure 19 The wave diagram showing the writing and reading from initials from certain address line.

Here is the wave diagram that shows the state machine-controlled input and output in relevant to enable, clock (sclk) and read or write (rwin) control pins. The state machine is having input ‘k’ and ‘z’, the output is giving out ‘k’ and ‘z’.

5. Discussion

As shown in the results from the Spice simulation, it states that in the effective voltage range from 0.5V – 1V, the leakage current is positively proportional to the voltage.

	voltage/V	Leakage current at different corner tests/ A				
		tt	ff	fs	ss	sf
static	1	-190n to -200n	-0.4u	-0.4u	-100n	-106n to -108n
pulse	1	-1.0u	-5.1u	-0.3u	1u	1u to 2u
	0.5	-0.5u to 0.5u	-0.5u to 0	0 to 0.2u	0 to 0.4u	0.1u
	0.4	N/A	N/A	-0.6u to -0.4u	N/A	N/A

Table 5 The Spice simulation.

There are some imperfections with the AIM-Spice tool, when zooming to the leakage current the values are reading by approximation. In the meantime, when zooming to the values, different levels of zooming give fluctuating results at the same time points. So, the result listed above might not be very accurate, but we can still use the conclusion.

With regarding to the Verilog simulation, the wave diagram also has some imperfections, a test bench could have been made, but due to the difficulty of syntax in structural model, I made the test manually. In a testbench, there could be more wave patterns and more tests.

6. Summary

The project achieved the initial goal to spice simulate 1 bit cell leakage power consumption and simulate the wave diagram of the RAM circuit.

As results show, when operating, it would be a good approach to set the input power of an IoT device at a reasonably low voltage, in this way, it can reduce the leakage power consumption, thus reduce the general power consumption. It will also be nice to make a SPICE consumption simulation as an extension to this project.

For the state machine, it can be good to test more cases, such as write different values to the bit cells and more patterns of input bits. It will be also nice to try a 64-bit RAM, this will be also a good extension to this project. Or add an 8-bit computer to process the address line, thus, to make a microcontroller and simulate the power consumption of the MCU and the RAM as well.

References

Inverter. (n.d.). Retrieved from falstad.com:
https://www.falstad.com/circuit/circuitjs.html?cct=$+1+0.000005+10.20027730826997+53+5+50%0Aw+304+64+304+128+0%0Ad+304+128+368+192+2+default%0Ad+304+256+368+192+2+default%0Ad+240+192+304+128+2+default%0Ad+240+192+304+256+2+default%0Aw+304+256+304+352+0%0A

M. Morris Mano, M. D. (2011). In *Digital Design* (p. 301).

M. Morris Mano, M. D. (2011). Digital Design. In *Morris_textbook-317-330* (p. 308).

NOR gate. (n.d.). Retrieved from falstad.com:
https://www.falstad.com/circuit/circuitjs.html?cct=$+1+0.000005+10.20027730826997+53+5+50%0Aw+304+64+304+128+0%0Ad+304+128+368+192+2+default%0Ad+304+256+368+192+2+default%0Ad+240+192+304+128+2+default%0Ad+240+192+304+256+2+default%0Aw+304+256+304+352+0%0A

Appendices

1. The RAM Verilog active-HDL workspace.

ram.v

```
1.   module ram(ins, rws, addr, outs);
2.
3.    input [7:0]ins;
4.    input [2:0]addr;
5.    input rws;
6.    output [7:0]outs;
7.    reg [7:0]rowout[0:7];
8.    wire [7:0]word;
9.    wire [15:0]halfout;
10.
11.   //   bt(inb, rwb, selb, outb);
12.   decoder sel(.od(word), .id(addr));
13.
14.   // module bt(inb[8], rwb, selb, outb[8]);
15.   bt row0(.outb(rowout[0]), .inb(ins), .rwb(rws), .selb(word[0]));
16.   bt row1(.outb(rowout[1]), .inb(ins), .rwb(rws), .selb(word[1]));
17.   bt row2(.outb(rowout[2]), .inb(ins), .rwb(rws), .selb(word[2]));
18.   bt row3(.outb(rowout[3]), .inb(ins), .rwb(rws), .selb(word[3]));
19.   bt row4(.outb(rowout[4]), .inb(ins), .rwb(rws), .selb(word[4]));
20.   bt row5(.outb(rowout[5]), .inb(ins), .rwb(rws), .selb(word[5]));
21.   bt row6(.outb(rowout[6]), .inb(ins), .rwb(rws), .selb(word[6]));
22.   bt row7(.outb(rowout[7]), .inb(ins), .rwb(rws), .selb(word[7]));
23.
24.   //   or outbit0(outs[0], rowout[0][0], rowout[1][0], rowout[2][0], rowout[3][0],
25.   //              rowout[4][0], rowout[5][0], rowout[6][0], rowout[7][0]);
26.   // output bit 0
27.   or outh00(halfout[0],  rowout[0][0], rowout[1][0], rowout[2][0], rowout[3][0]);
28.   or outh01(halfout[1], rowout[4][0], rowout[5][0], rowout[6][0], rowout[7][0]);
29.   or outbit0(outs[0], halfout[0], halfout[1]);
30.
31.
32.   //   or outbit1(outs[1], rowout[0][1], rowout[1][1], rowout[2][1], rowout[3][1],
33.   //              rowout[4][1], rowout[5][1], rowout[6][1], rowout[7][1]);
34.   //   outout bit 1
35.   or outbit10(halfout[2], rowout[0][1], rowout[1][1], rowout[2][1], rowout[3][1]);
36.   or outbit11(halfout[3], rowout[4][1], rowout[5][1], rowout[6][1], rowout[7][1]);
37.   or outbit1(outs[1], halfout[2], halfout[3]);
38.
39.   //   or outbit2(outs[2], rowout[0][2], rowout[1][2], rowout[2][2], rowout[3][2],
40.   //              rowout[4][2], rowout[5][2], rowout[6][2], rowout[7][2]);
41.   or outbit20(halfout[4], rowout[0][2], rowout[1][2], rowout[2][2], rowout[3][2]);
42.   or outbit21(halfout[5], rowout[4][2], rowout[5][2], rowout[6][2], rowout[7][2]);
43.   or boutbit2(outs[2], halfout[4], halfout[5]);
44.
45.
46.   //   or outbit3(outs[3], rowout[0][3], rowout[1][3], rowout[2][3], rowout[3][3],
47.   //              rowout[4][3], rowout[5][3], rowout[6][3], rowout[7][3]);
48.   or outbit30(halfout[6], rowout[0][3], rowout[1][3], rowout[2][3], rowout[3][3]);
49.   or outbit31(halfout[7], rowout[4][3], rowout[5][3], rowout[6][3], rowout[7][3]);
50.   or outbit3(outs[3], halfout[6], halfout[7]);
51.
52.   //   or outbit4(outs[4], rowout[0][4], rowout[1][4], rowout[2][4], rowout[3][4],
53.   //              rowout[4][4], rowout[5][4], rowout[6][4], rowout[7][4]);
54.   or outbit40(halfout[8], rowout[0][4], rowout[1][4], rowout[2][4], rowout[3][4]);
55.   or outbit41(halfout[9], rowout[4][4], rowout[5][4], rowout[6][4], rowout[7][4]);
56.   or outbit4(outs[4], halfout[8], halfout[9]);
57.
58.   //   or outbit5(outs[5], rowout[0][5], rowout[1][5], rowout[2][5], rowout[3][5],
59.   //              rowout[4][5], rowout[5][5], rowout[6][5], rowout[7][5]);
60.   or outbit50(halfout[10], rowout[0][5], rowout[1][5], rowout[2][5], rowout[3][5]);
61.   or outbit51(halfout[11], rowout[4][5], rowout[5][5], rowout[6][5], rowout[7][5]);
62.   or outbit5(outs[5], halfout[10], halfout[11]);
```

```
63.
64.
65.   //   or outbit6(outs[6], rowout[0][6], rowout[1][6], rowout[2][6], rowout[3][6],
66.   //              rowout[4][6], rowout[5][6], rowout[6][6], rowout[7][6]);
67.   or outbit60(halfout[12], rowout[0][6], rowout[1][6], rowout[2][6], rowout[3][6]);
68.   or outbit61(halfout[13], rowout[4][6], rowout[5][6], rowout[6][6], rowout[7][6]);
69.   or outbit6(outs[6], halfout[12], halfout[13]);
70.
71.
72.   //   or outbit7(outs[7], rowout[0][7], rowout[1][7], rowout[2][7], rowout[3][7],
73.   //              rowout[4][7], rowout[5][7], rowout[6][7], rowout[7][7]);
74.   or outbit70(halfout[14], rowout[0][7], rowout[1][7], rowout[2][7], rowout[3][7]);
75.   or outbit71(halfout[15], rowout[4][7], rowout[5][7], rowout[6][7], rowout[7][7]);
76.   or outbit7(outs[7], halfout[14], halfout[15]);
77.
78. Endmodule
```

2. The state machine Verilog implementation

state.v

```
1.   module state(rwin, sclk, enable, datain, saddr, dataout);
2.    input sclk, enable, rwin;
3.    input [0:7]datain;
4.    input [2:0]saddr;
5.    output [7:0]dataout;
6.    wire ddata, dout, sout;
7.
8.    dflipflop states(.qata(dout), .clk(sclk), .data(rwin));
9.    or dd(sout, dout, ~enable);
10.   and fsm(ddata, sout, enable);
11.   ram ss(.outs(dataout), .ins(datain), .rws(dout), .addr(saddr));
12.
13. endmodule
```

3. The byte module Verilog implementation

```
1.   module bt(inb, rwb, selb, outb);
2.
3.    input [7:0]inb;
4.    input selb;
5.    input rwb;
6.    output [7:0]outb;
7.
8.    // module bc(in, rw, sel, out);
9.    bc clm0(.out(outb[0]), .in(inb[0]), .rw(rwb), .sel(selb));
10.   bc clm1(.out(outb[1]), .in(inb[1]), .rw(rwb), .sel(selb));
11.   bc clm2(.out(outb[2]), .in(inb[2]), .rw(rwb), .sel(selb));
12.   bc clm3(.out(outb[3]), .in(inb[3]), .rw(rwb), .sel(selb));
13.   bc clm4(.out(outb[4]), .in(inb[4]), .rw(rwb), .sel(selb));
14.   bc clm5(.out(outb[5]), .in(inb[5]), .rw(rwb), .sel(selb));
15.   bc clm6(.out(outb[6]), .in(inb[6]), .rw(rwb), .sel(selb));
16.   bc clm7(.out(outb[7]), .in(inb[7]), .rw(rwb), .sel(selb));
17.
18. endmodule
```

4. The bit cell Verilog implementation

```
1.   module bc(in, rw, sel, out);
2.
3.    input in, rw, sel;
4.    output out;
5.
6.    wire w1, w2, w3, w4;
7.
```

```
8.    and a1(w1, sel, in, ~rw);
9.    and a2(w2, sel, ~rw, ~in);
10.   nor nr1(w3, w1, w4);
11.   nor nr2(w4, w2, w3);
12.   and a3(out, sel, w4, rw);
13.
14. endmodule
```

5. The D flipflop Verilog implementation

```
1.   module dflipflop(data, clk, qata);
2.
3.    input data, clk;
4.    output qata;
5.
6.    wire w1, w2, w3, w4, w5, w6;
7.    nand a1(w1, w2, w4);
8.    nand a2(w2, w1, clk);
9.    nand a3(w3, w2, clk, w4);
10.   nand a4(w4, w3, data);
11.   nand a5(qata, w2, w6);
12.   nand a6(w6, w3, qata);
13.
14. endmodule
```

6. The decoder Verilog implementation

```
1.   module decoder(id, od);
2.
3.    input [2:0]id;
4.    output [7:0]od;
5.    and a0(od[0], ~id[0], ~id[1], ~id[2]);
6.    and a1(od[1], ~id[0], ~id[1], id[2]);
7.    and a2(od[2], ~id[0], id[1], ~id[2]);
8.    and a3(od[3], ~id[0], id[1], id[2]);
9.    and a4(od[4], id[0], ~id[1], ~id[2]);
10.   and a5(od[5], id[0], ~id[1], id[2]);
11.   and a6(od[6], id[0], id[1], ~id[2]);
12.   and a7(od[7], id[0], id[1], id[2]);
13.
14. endmodule
```

7. The inverter subcircuit

```
1.   .subckt invertersub in out vdd vss
2.
3.  *.param ln = 0.1u
4.  *.param lp = 0.1u
5.  *.param wn = 0.1u
6.  *.param wp = 0.36u
7.  *.param ld = 280n
8.
9.  *    D   G  S   B
10. xmp out in vdd vdd pmos1v l=lp w=wp
11. xmn out in 0 0 nmos1v l=ln w=wn
12.
13. .ends
```

8. The AND gate subcircuit

```
1.   .subckt andsub in1 in2 out vdd vss
2.
```

```
3.  * .param ld = 280n
4.
5.  *    Drain Gate Source Bulk
6.  *    D   G  S   B
7.  xmp1 out w1 vdd vdd pmos1v l=lp w=wp
8.  xmn1 out w1 vss vss nmos1v l=ln w=wn
9.
10. xmp2 w1 in2 vdd vdd pmos1v l=lp w=wp
11. xmp3 w1 in1 vdd vdd pmos1v l=lp w=wp
12.
13. xmn2 w1 in1 w2 w2 nmos1v l=ln w=wn
14. xmn3 w2 in2 vss vss nmos1v l=ln w=wn
15.
16. .ends
```

9. The NOR gate subcircuit

```
1.   .subckt norsub in1 in2 out vdd vss
2.
3.  *    D   G   S   B
4.  xmp1 w1 in2 vdd vdd pmos1v l=lp w=wp
5.  xmp2 out in1 w1 w1 pmos1v l=lp w=wp
6.
7.  xmn1 out in1 vss vss nmos1v l=ln w=wn
8.  xmn2 out in2 vss vss nmos1v l=ln w=wn
9.
10. .ends
```

10. The bit cell circuit

```
1.   ** bc
2.  * ports: in out rw addr
3.  VDD1 vdd 0 mv
4.  *VDD2 addr 0 1
5.
6.  *.include D:\ProgramFiles\AIM-Software\AIM-Spice\Canvas\gpdk90nm_tt.cir
7.  *.include D:\ProgramFiles\AIM-Software\AIM-Spice\Canvas\gpdk90nm_ff.cir
8.  *.include D:\ProgramFiles\AIM-Software\AIM-Spice\Canvas\gpdk90nm_fs.cir
9.  *.include D:\ProgramFiles\AIM-Software\AIM-Spice\Canvas\gpdk90nm_ss.cir
10. .include D:\ProgramFiles\AIM-Software\AIM-Spice\Canvas\gpdk90nm_sf.cir
11.
12. .include D:\ProgramFiles\AIM-Software\AIM-Spice\project\andsub.txt
13. .include D:\ProgramFiles\AIM-Software\AIM-Spice\project\invertersub.txt
14. .include D:\ProgramFiles\AIM-Software\AIM-Spice\project\norsub.txt
15.
16. .param time = 10p
17. .param mv = 0.4
18. .param ln = 0.1u
19. .param lp = 0.1u
20. .param wn = 0.1u
21. .param wp = 0.36u
22.
23. *               I1 I2 TimeDelay TRise TFall p.W  period)
24. Vgate1 in 0  mv PULSE(0 mv 1ns time time 4nS 8nS)
25. Vgate2 sel 0 mv PULSE(0 mv 1ns time time 3nS 6nS)
26. Vgate3 rw 0  mv PULSE(0 mv 1ns time time 7nS 14nS)
27.
28. * in out vdd vss
29. xno1 in nin vdd 0 invertersub
30. xno2 rw nrw vdd 0 invertersub
31.
32. *   in  in out vdd vss
33. xa11 sel in w11 vdd 0 andsub
34. xa12 nrw in w12 vdd 0 andsub
35. xa13 w11 w12 w13 vdd 0 andsub
36.
```

```
xa21 sel nin w21 vdd 0 andsub
xa22 nrw nin w22 vdd 0 andsub
xa23 w21 w22 w23 vdd 0 andsub

* sr latch
*   in  in out vdd vss
xnr1 w13 w4 w3 vdd 0 norsub
xnr2 w23 w3 w4 vdd 0 norsub

* final and Gate
*   in  in out vdd vss
xa31 sel w4 w31 vdd 0 andsub
xa32 rw w4 w32 vdd 0 andsub
xa33 w31 w32 out vdd 0 andsub

.plot V(in) V(rw) V(sel) V(out) ! 1.05
.plot V(out) ! 1.05
.plot I(VDD1) ! 1.05
```