NTNU | Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

TFE4152 Design of Integrated Circuits

Term project fall 2024

8x8-bit Memory Module

Authors:
L. Strand & S.W. Kalland
Group 3

Abstract

This report contains the design of an 8x8-bit memory module for low-power applications. It achieved a simulated leakage current of 2 to 6 nA at a supply voltage of 0.99 V. The memory module consists of a Finite State Machine acting as a control unit to a clock-independent SRAM. It has a stable analog read/write time of less than 3 ns, but requires two clock cycles for write operations, as opposed to a single clock cycle for read operations. The entire system is implemented at a logic gate level, with appendixes containing sufficient Verilog code to digitally simulate the system.

Table of Contents

1 Introduction

This report assignment was given to us in the course TFE4152 at NTNU. The task is to design a memory unit that will be used in an IoT device. The memory unit shall have a total of 64 bits of storage. Furthermore, the system shall have a word length of 8 bits and a read/write time of no longer than 3 ns. All further specifications are listed in the assignment sheet [1], with necessary theory presented in Section 2. Our implementation of the solution is presented in Section 3 with following results in Section 4. We discuss our design in greater detail in Section 5 and make some concluding remarks in Section 6.

2 Theory

2.1 Memory Units

Every digital memory unit is built up hierarchically by several word cells. The size of each word may vary from system to system (8-bit, 16-bit, 32-bit, etc.), but no matter the architecture, the smallest unit of memory, the bitcell, is present. The total available bits in the memory is the product of the number of bits per word times number of words in the unit. Every bit is addressable by a unique address that points to a single word in the unit. When reading or writing a word, the whole word is accessed, and should individual bits be of interest, then those must be parsed from the word.

The bitcells within each word are designed differently depending on the level of transparency required. There are several options for storing a single digital value, such as latches and flip-flops. Latches are transparent in the sense that once enabled, the output will follow the input immediately (with some negligible delay). Flip-flops on the other hand are not transparent: The output must always be overwritten by a rising or falling clock edge, thus delaying the output compared to the input.

2.2 Power Reduction

Total power consumption of a system is usually calculated as the sum of the static power consumption and the dynamic power consumption. The static power consumption is always present, even when the system is idle, meaning no inputs or outputs are changing. The dynamic power consumption is the extra power required to change the states of the system.

$$P_{total} = P_{static} + P_{dynamic} \tag{1}$$

Static power consumption is calculated by the average number of transistors that are in the active region, and how much each transistor leaks. The dynamic power consumption depends on many factors, but it can be reduced by lowering any of the system factors it is proportional to, see Equation 2 [2].

$$P_{dynamic} \propto f_{clk} \cdot C_{load} \cdot V_{DD}^2 \tag{2}$$

In recent years, with clock frequencies stagnating, contrary to Moore's law, and supply voltages being minimized, it is the static power consumption that ends up contributing the most to the overall power consumption. In order to reduce power consumption in a 'System on a Chip' (SoC) there are many factors to consider, but some very common ones are to:

- Reduce the number of transistors in the system as a whole.

- Reduce the current through each transistor when in the active region, which can be estimated with equation 3 [3].
- Reduce the supply voltage
- Reduce the switching frequency / clock frequency.
- Reduce the width-to-length ratio.
- Minimize the difference between voltage supply and threshold voltage of the transistors (V_{eff}).

To achieve the last bullet point, one may choose to either lower the voltage supply directly or increase the threshold voltage by tuning the bulk voltage of each transistor.

$$I_D = \frac{\mu C_{ox}}{2} \frac{W}{L} (V_{GS} - V_T)^2 \tag{3}$$

2.3 Process variation corners

When producing SoC's, the transistors on the wafer are prone to be slightly different from each other depending on a number of factors. For instance, transistors made in proximity on the wafer are more likely to be similar to each other than transistors placed further apart. These process variations can make some transistors switch on/off faster than others in unforeseen ways. In order to sufficiently simulate a system, we divide the transistor variations into five different simulation "corners" according to how fast they are switching.

- FF *Fast-Fast* Fast operating NMOS and Fast operating PMOS
- SS *Slow-Slow* Slow operating NMOS and Slow operating PMOS
- FS *Fast-Slow* Fast operating NMOS and Slow operating PMOS
- SF *Slow-Fast* Slow operating NMOS and Fast operating PMOS
- TT *Typical-Typical* Typical operating NMOS and Typical operating PMOS

The advantages of fast-operating transistors include faster switching times and lower threshold voltages, meaning that they can operate on a lower voltage supply. However they have higher power consumptions due to the higher leakage current and faster switching. The leakage current is a result of the shorter channel lengths meaning less capacitance between the channels. The lowered threshold voltage V_T will also provide higher leakage current. When V_T is lowered, it means that even a small voltage at the gate can cause the transistor to start conducting, making it more susceptible to noise-induced leakage. Additionally, sub-threshold leakage occurs when the gate voltage is below V_T, but still close enough to allow small currents to flow through the channel. In contrast to fast transistors, slow transistors have a higher V_T and thus lower power consumption, which makes them ideal for low energy implementations. However, these simulation corners are not chosen, they are simulated to make sure the system can operate with transistors that are slightly different to what was intended. The ideal design would be the TT corner, but the system has to be designed to work within all of the corners.

3 Method

3.1 System overview

Our system consists of a Finite State Machine and a (Static) Random Access Memory, as can be seen in Figure 1. The FSM serves as a control unit to make sure all operations on the RAM are performed in a stable manner. The FSM is connected to a clock signal, but the RAM is not.

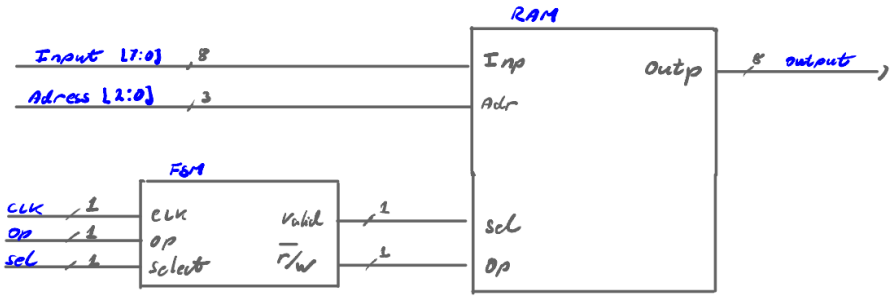


Figure 1: Block diagram of the system

3.2 RAM

The RAM subsystem consists of eight bitcells forming eight bytecells that form the full memory. It has these inputs and outputs:

- *adr* - 3-bit address signal
- *inp* - 8-bit input data to be stored in address
- *outp* - 8-bit output data to be read from address
- *op* - either 0 for read or 1 for write
- *sel* - must be 1 for *op* to affect the circuit

When the circuit is reading, meaning *op* is low, the output is what is stored in the currently addressed bytecell. When it is writing, the *outp* is Z (high impediance), meaning it can be easily overwritten, and should not be trusted if the circuit is not in read mode with *sel* high. The RAM's circuit is implemented as can be observed in Figure 2. It consists of eight bytecell modules. The entire output bus is connected to all the bytecells in parallel, making it possible to only read from one of the bytecells at a time.

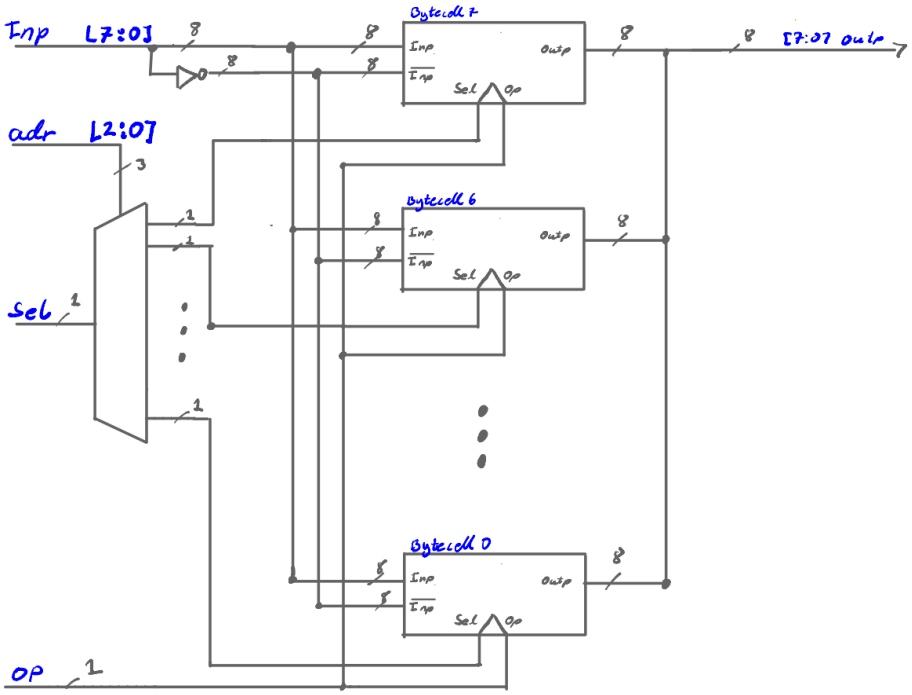


Figure 2: Schematic for the RAM subsystem

3.2.1 The bitcell

The bitcell concists of a D-latch and a tri-state-buffer. The logic schematic of the D-latch, connected to the transistor schematic of a tri-state-buffer, can be seen in Figure 3.

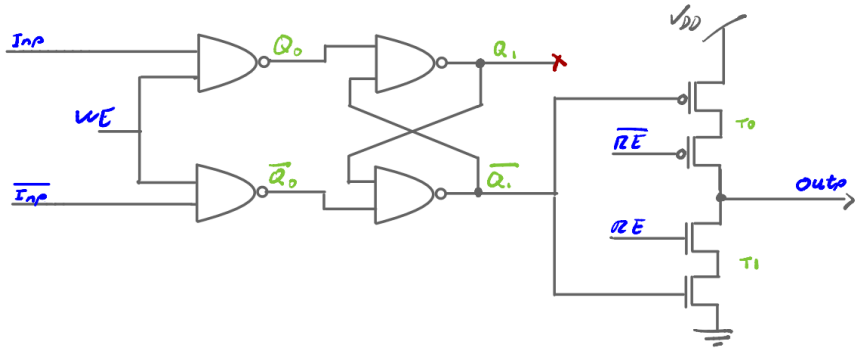


Figure 3: Wiring schematic for the bitcell module

The bitcell has inputs and outputs:

- *inp* - The input bit to be stored
- *inpn* - (input not) The inverse of the input bit

- *outp* - The output when the circuit is read
- *WE* - Write Enable, must be high to store a new value
- *RE* - Read Enable, must be high to set outp to the stored value
- *REN* - (Read Enable Not) inverted of *RE*

The inverse stored value sits in the node $\overline{Q_1}$. We choose to implement the bitcell with a tri-state-buffer in order to make the output high impedance (Z) when RE is low. This is to ensure that the shared output bus in the RAM module can be overwritten by the selected bytecell module if RE is high. This simplifies the rest of the design, since there is no need for a multiplexer to choose which address to read data from.

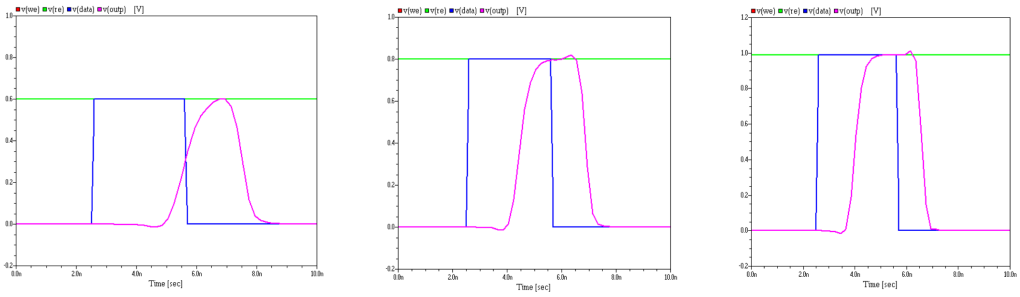
3.2.2 Bitcell leakage

The leakage current of each bitcell is a sum of the leakage current of all its transistors. In total there are 20 transistors in each bitcell, 10 NMOS and 10 PMOS. The leakage current of one NMOS can be estimated from equation 3. To reduce the leakage current, the right side of this equation can be reduced by tuning the width-to-length ratio. Hence, our choice of transistor sizes, as shown in table 1, are chosen to reduce this ratio (within the allowed specifications of the assignment).

Table 1: Chosen size of P- & N-MOS transistor.

W	101 nm
L	299 nm

In the next section the slow-slow (SS) corner has the slowest operation time of all the simulation corners as seen in Figure 11. The voltage supply was chosen to be $V_{DD} = 0.99\,\text{V}$ based on early simulations. This value provided us with the most stable and functional bitcell for all of the operational corners within the task's read/write requirements of 3 ns as can be seen in Figure 4.



(a) $Vdd = 0.6\,\text{V}$ (b) $Vdd = 0.8\,\text{V}$ (c) $Vdd = 0.99\,\text{V}$

Figure 4: Write times for the *SS* corner at different supply voltages, V_{dd}. The input pulse is exactly 3 ns long.

3.2.3 The bytecell

The bytecell consists of 8 bitcells and an F-block (function block) meant to translate the input signals of the bytecell to the required inputs of each of its bitcells. The bitcells all write their outputs to 8 different wires, which we refer to as the databus. The bytecell schematic can be seen in Figure 5.

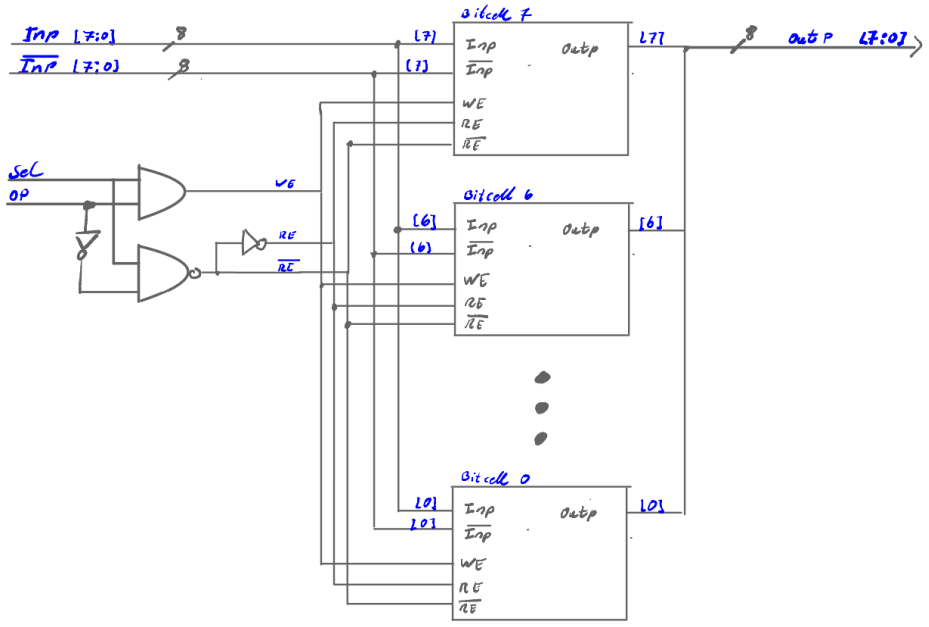


Figure 5: Wiring schematic for the bytecell module

The inputs and outputs of the bytecell are almost identical to those of the RAM, with the exception of there being no address signal, and the input having to be written both normally and inverted. Having the inverse signal bus at this level means saving transistors, as there will be one inverter for all of the 8 databus wires one level higher, instead of one inverter for every bitcell. The downside to this solution is that we need more wires connecting each module, which in turn can create more noise and disruptions internally. If the inverted wires are printed as differential pairs, the noise issue will be minimized.

The control logic on the left was designed to set *WE* high only when *op* and *sel* are high, and to set *RE* high only when *op* is low and *sel* is high. Otherwise, both *WE* and *RE* are set to low. The truth table can be seen in Table 2.

Table 2: The truth table for the control logic in Figure 5.

op	s*sel*	*WE*	*RE*
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	0

3.2.4 Demux

The demultiplexer (demux) that can be seen on the left in Figure 2 sends *inp* (*sel* in Figure 2) to a single addressed *outp* (a bytecell), according to the address in *adr*. All the other *outp* are set to low, except for the one that is addressed. The demux schematic is as shown in Figure 6.

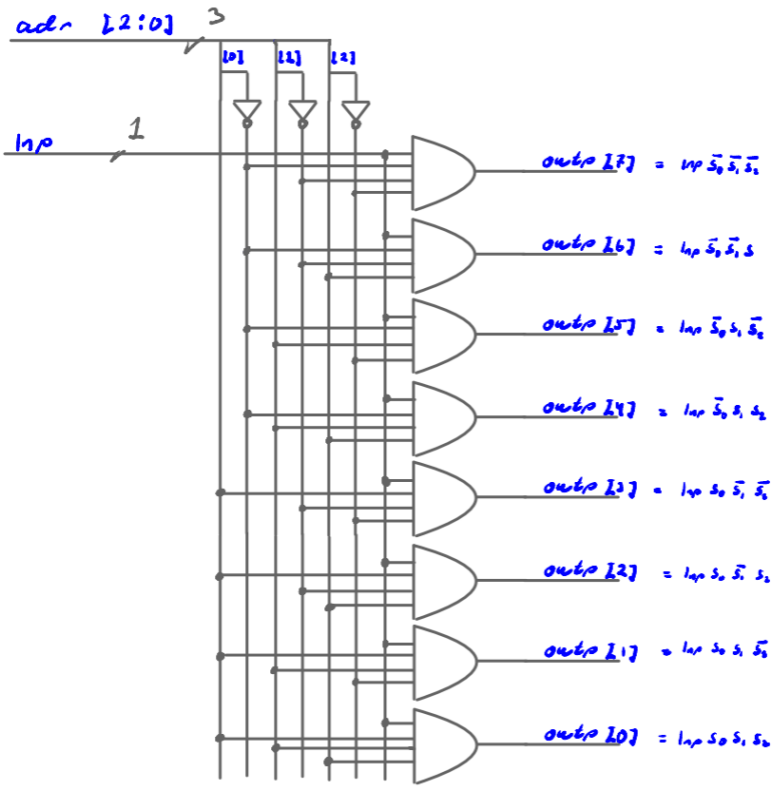


Figure 6: A schematic of the demux

3.3 FSM

The finite state machine is meant to work as an interface for stable and secure use of the RAM subsystem. Its design is based on what was demanded in the project description [1]. The required design can be seen in Figure 7.

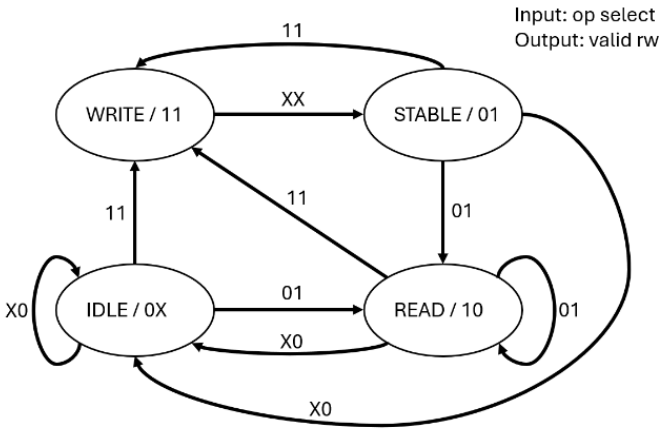


Figure 7: Desired state diagram [1]

In order to design the aforementioned system state codes were made as can be seen in Table 3. Then a truth table was created in order to find the logical relations between inputs and outputs in Table 4, where:

- A - current state of register $R0$.
- B - current state of register $R1$.
- C - input *op*.
- D - input *sel*.
- E - next state of register $R0$.
- F - next state of register $R1$.
- G - output *valid*.
- H - output *rw*.

Table 3: State codes.

State	Code
IDLE	0 X
STABLE	0 1
READ	1 0
WRITE	1 1

Table 4: Truth table for Finite State Machine.

State		Input		State nxt		Output	
		op	*sel*			*valid*	*rw*
A	B	C	D	E	F	G	H
0	0	0	0	0	X	0	X
0	0	0	1	1	0	0	X
0	0	1	0	0	X	0	X
0	0	1	1	1	1	0	X
0	1	0	0	0	X	1	0
0	1	0	1	1	0	1	0
0	1	1	0	0	X	1	0
0	1	1	1	1	1	1	0
1	0	0	0	0	X	0	1
1	0	0	1	1	0	0	1
1	0	1	0	0	X	0	1
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	1
1	1	0	1	0	1	1	1
1	1	1	0	0	1	1	1
1	1	1	1	0	1	1	1

Using the logical relations between the inputs, outputs, current states and next states in Equation 4, we designed an circuit implementation of the state machine in Figure 8.

$$\begin{aligned} E &= D \cdot \overline{A \cdot B} \\ F &= A \cdot B + C \cdot D \\ G &= B \\ H &= A \end{aligned} \tag{4}$$

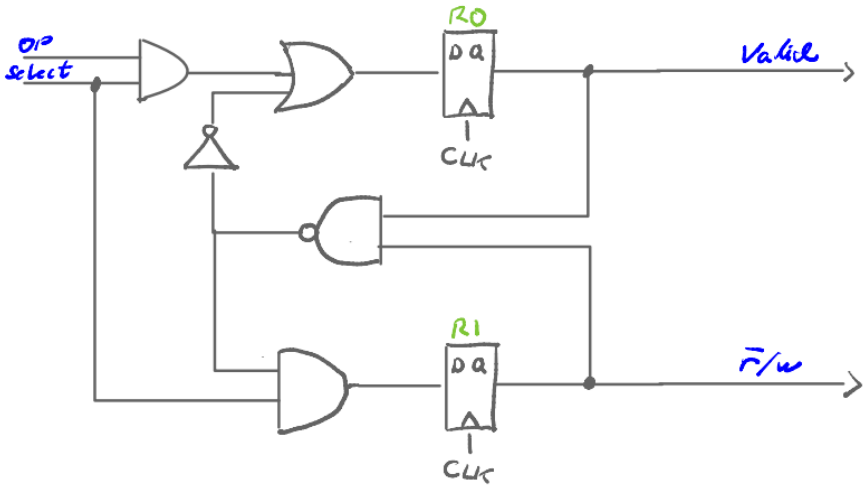


Figure 8: Wiring schematic for the FSM logic.

4 Simulations and Results

All analog simulations are performed in the language of *AIMSpice* in the *AIM-Spice* computer program. All digital simulations are performed in the language *Verilog*, with the editor *VSCode*, compiler *Icarus Verilog* and waveform plotter *gtkwave*. All codes used to simulate can be found in the appendix, Section 7.

4.1 Analog bitcell

The analog behavior of the bitcell is simulated to show its functionality at different temperatures, and its read-time, write-time and leakage current at the *SS*, *TT* and *FF* transistors operating corners.

4.1.1 Leakage current

The leakage currents at different configurations are shown in Figure 9. The current is negative, so keep in mind that it is the absolute value of the current which is of interest. It would ideally be as close to zero as possible to keep power consumption down.

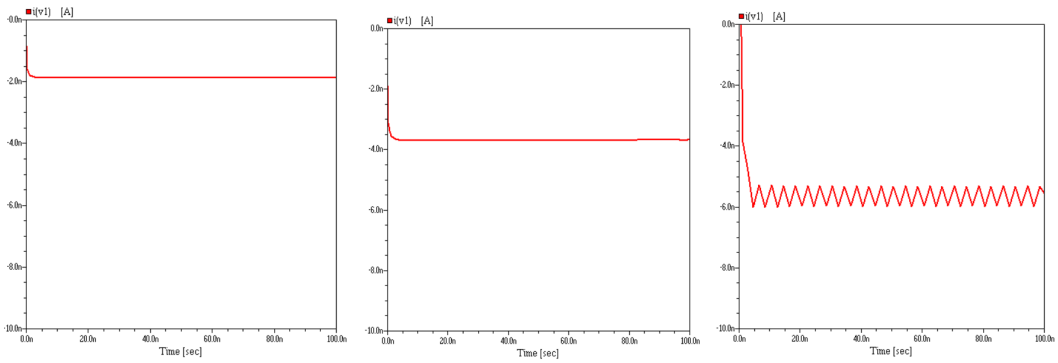


Figure 9: Leakage currents at *SS*, *TT* and *FF* (from left to right).

4.1.2 Read time

The read times can be seen in Figure 10. They are all shorter than 3 ns, which is the length of the *RE* pulse, as was required in the project description.

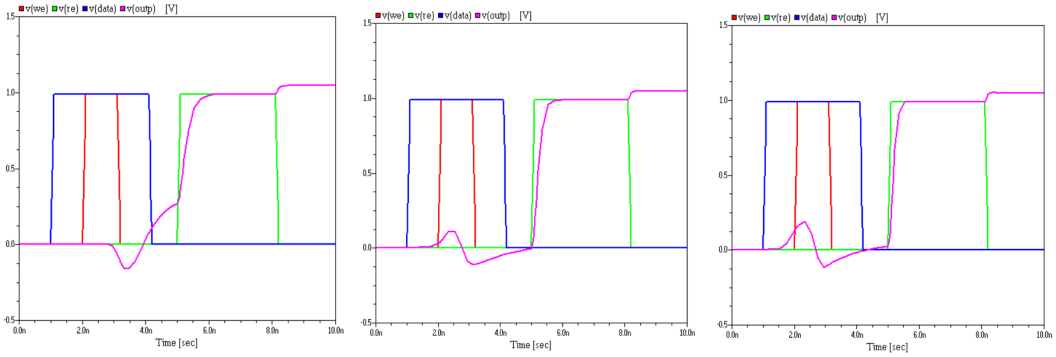


Figure 10: The read times to the bitcell at *SS*, *TT* and *FF* (from left to right) can be read as the time it takes v(outp) to reach v(re).

4.1.3 Write time

The write times can be seen in Figure 10. They are also all shorter than 3 ns, which is the length of the *inp* pulse, as was required in the project description.

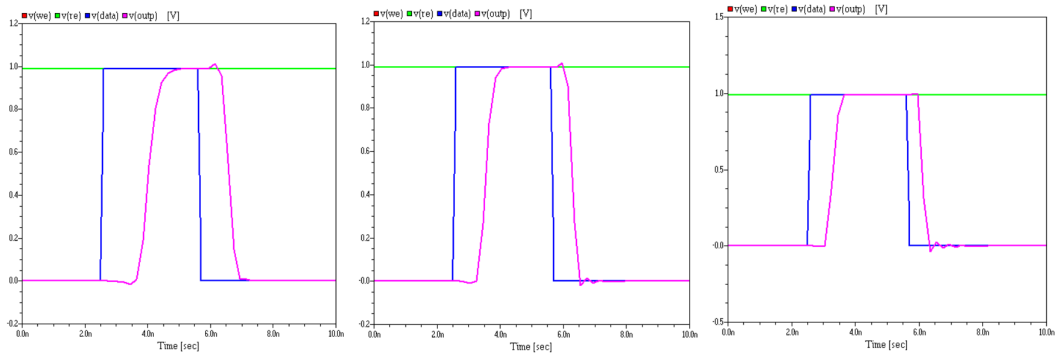


Figure 11: The write times to the bitcell at *SS*, *TT* and *FF* (from left to right) can be read as the time it takes *v(outp)* to reach *v(data)*.

4.1.4 Temperature Functionality

The circuit is tested at three different temperatures (−20 C, 27 C, 50 C) for five different circuit configurations (*FF, FS, SF, SS, TT*). The test is comprised of writing, storing and reading a logical low, and then a logical high. The functionality tests perform as can be seen in Figure 12.

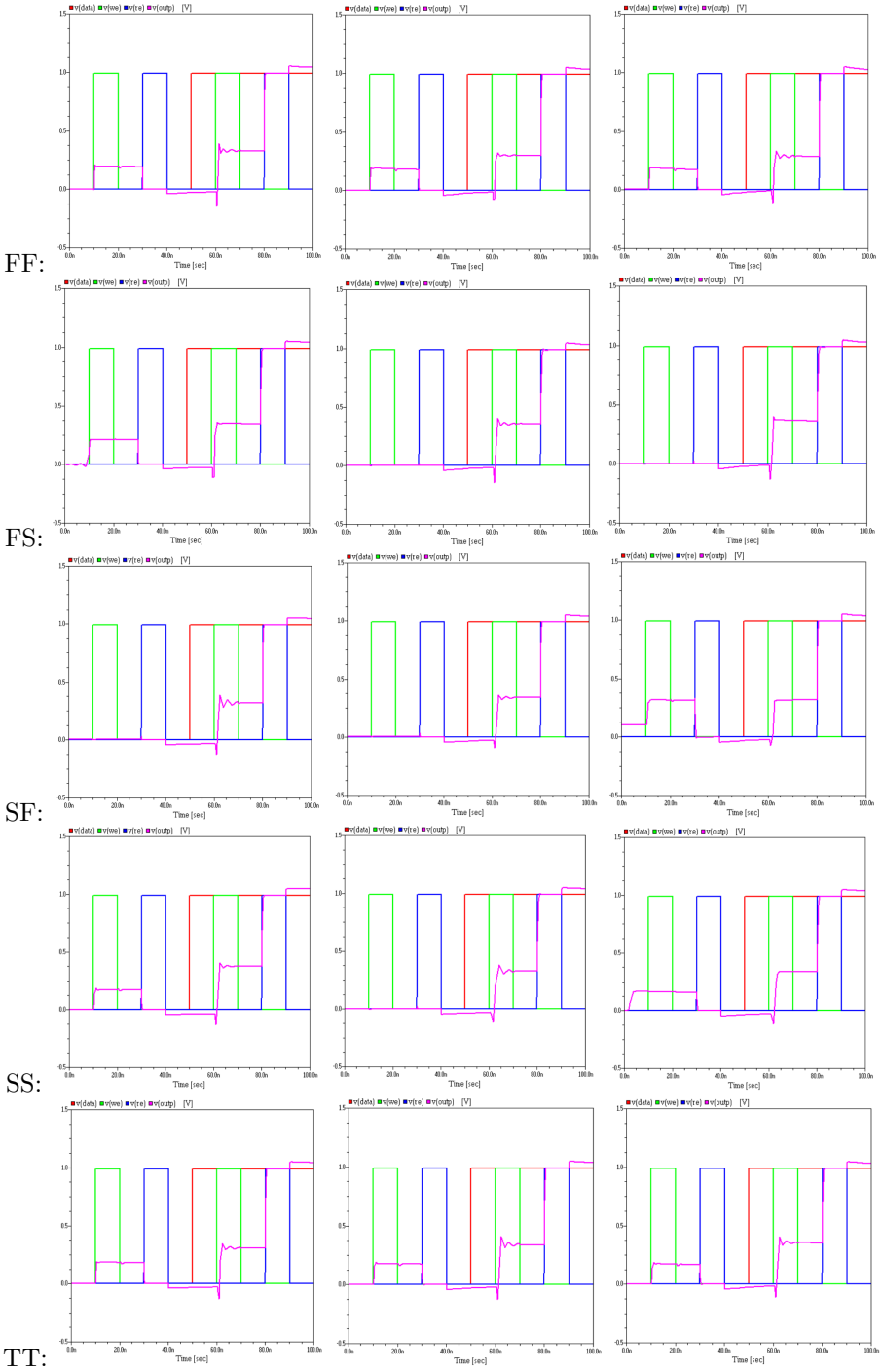


Figure 12: The functionality tests of the bitcell. The left column is for -20°C, the middle column is for +27°C, and the right column is for +50°C. All tests meet the system requirements.

4.2 Digital bitcell

The bitcell's behaviour is digitally simulated being given inputs and showing outputs according to what can be observed in Figure 13.

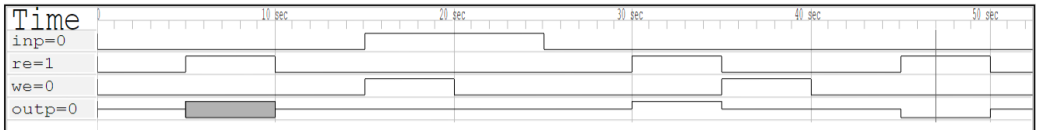


Figure 13: Testbench of the bitcell

In the testbench we read the bitcell before any value has been set, expecting a undefined output, X (gray area). Then we write a logic high, read the logic high, write a logic low, and read that too. When RE is low, we expect to see a high-impedance value, Z, which is observed as the *outp* line being between logical low and high. REN is not shown in the figure, since it is trivial. The subsystem performed exactly as expected.

4.3 Demux

The demux' behaviour is digitally simulated being given inputs and showing outputs according to what can be observed in Figure 14.

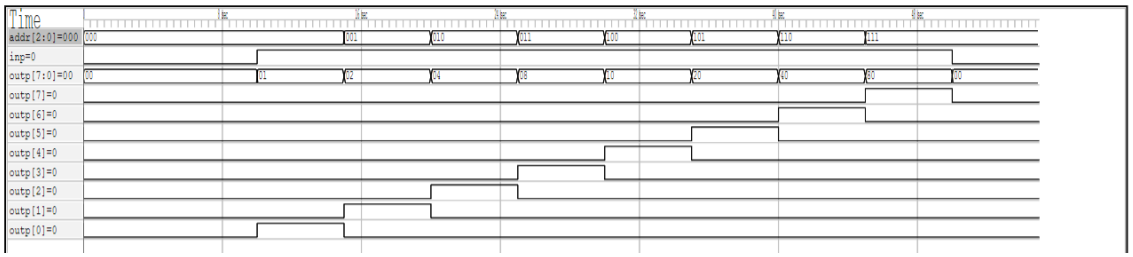


Figure 14

In the testbench we cycle through all of the outputs, checking that each of them can be set high, and that all the other ones are low if that is the case. The subsystem performed exactly as expected.

4.4 RAM

The RAM subsystem's behaviour is digitally simulated being given inputs and showing outputs according to what can be observed in Figure 15.

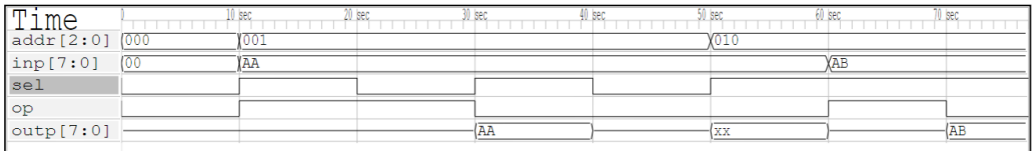


Figure 15: Testbench of the RAM

In the testbench we write the 8-bit hex value AA into address 001, and read that value. Then, we go to address 010 and read what is stored there, which should be undefined, XX, since nothing has been written to this address yet. We then write the hex value AB to this address and read to observe that it has changed from XX to AB. The subsystem performed exactly as expected.

4.5 FSM

The FSM subsystem's behaviour is digitally simulated being given inputs and showing outputs according to what can be observed in Figure 16.

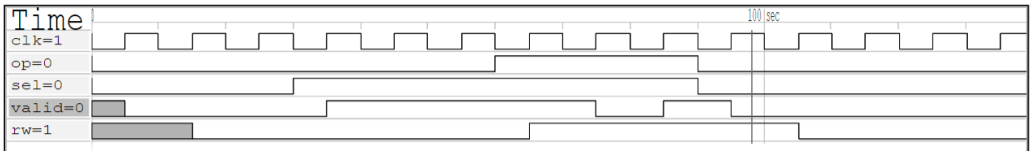


Figure 16: Testbench of the FSM

In the testbench we check that the FSM can enter all of its four states:

- We start by waiting two clock cycles for it to settle into the *IDLE* state.
- Then we set *sel* high, which causes the FSM to go into the *READ* state on the next rising edge, which can be seen by the FSM's outputs.
- After making sure the state is stable, we set *op* high as well, causing the FSM to go into the *WRITE* state.
- Here it will go to the *STABLE* state on the next clock cycle, no matter the input, but we keep the inputs as before to make sure it then goes back to *WRITE* again.
- Lastly, we observe the FSM going back to *IDLE* when both inputs are low.

The FSM performed as expected.

4.6 Complete memory module

The behaviour of the complete memory module system, with the FSM subsystem controlling the inputs to the RAM subsystem, is digitally simulated being given inputs and showing outputs according to what can be observed in Figure 17.

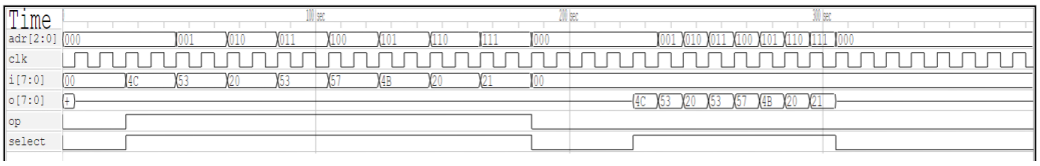


Figure 17: Testbench of the complete memory module

In this testbench we write our initials into the memory module, and then read them back. The ASCII string to store is *"LS SWK !"*, which converted to hex is $[4C, 53, 20, 53, 57, 4B, 20, 21]$. Writing

takes two clock cycles per word, since the FSM has to go into the *STABLE* state, and reading takes one clock cycle. The completed system performed as expected.

5 Discussion

5.1 Volatile bitcell outputs

Since our bitcells all write to a shared databus, what they do when not selected for reading is quite important. In the functionality test in Figure 12, it can be observed that when *RE* is low the bitcell is still outputting values. These values changed a lot when *WE* changed as well.

Tests that were not included in the results have been performed, where a pull-down resistor was attached to the output to see how "strong" the bitcell output was. With a fairly weak pull-down of 100 kΩ, the output was forced low in the span of a few ns when *RE* was low. These tests are not included due to lack of rigidity, but performing more tests on how an entire bytecell output behaves is recommended.

The reason these effects are observed is also of interest. We believe it is due to the transistors's capacitance. When the output is not actively driven, the tri-state buffer's capacitance could leak some current, creating a voltage difference on the output. In Figure 12, a typical low-pass filtered signal curve is observed at time 60 ns, indicating that capacitive or inductive effects are involved.

5.2 Databus or multiplexer

In our implementation we chose to have the output be a shared databus, as opposed to having a multiplexer (mux) select which bitcell to read from. When designing we assumed this would require less transistors, lead to a smaller footprint, and thus a lower power consumption. Since this memory is to be implemented in an IoT device, lowering footprint and power consumption should be a primary goal.

However, we never actually tested how a mux performed when compared to a databus. Considering what was discussed in the previous subsection about the bitcell's outputs, this point of discussion is even more relevant. A databus might have more capacitive effects, leading to the read-time of the module being significantly longer than the read-time of a single bitcell. Thus, a mux might have been more stable, or faster. Implementing both systems and testing them would have let us have a stronger argument on what design should be chosen.

5.3 Further testing

The testbenches that were performed were not all-encompassing. For instance, the FSM testbench checked that the FSM could reach all of its states, and give correct outputs in each of those states, but it did not check that all of the state transitions worked as intended. Every state should have been reached, and then received every possible input. The completed memory module was also not tested to its limits, as no reading before writing was performed, and not every combination of stored values were checked.

5.4 Supply voltage

For our system we chose to optimize the transistor dimensions for low leakage current, since this is an IoT application, and is thus likely dependent on a battery. As such, we had to adjust our supply voltage to be higher in order to have a fast enough write time. What we failed to consider was

that if the memory is to be used frequently, the dynamic power usage will suffer from our choice of a higher supply voltage. The dynamic power is related to the supply voltage squared, as can be seen in Equation 2, so minimizing leakage current is not necessarily equivalent to minimizing power consumption. We could have explored that further. It is also not obvious that we even have minimized leakage current, as the drain current, Equation 3, is also proportional to the voltage squared.

5.5 Scaleability

One aspect that was not explored, but which is worth mentioning, is the scalability of the system. Should a larger memory unit be desired, scaling the RAM module to include more bytecells only requires a larger demux and address bus; all other systems would be identical to how they are designed in this document.

6 Conclusion

In this report we have designed an 8x8-bit memory module for an IoT device. It has been implemented in Verilog with hierarchical modules. The memory unit operates as expected as shown by the testbenches in Section 4. Additionally, the bitcell has been simulated in the simulation tool *AIM-Spice*, and the simulation results are well within what the project demanded. We are especially fond of the small leakage current achieved, 2 nA, for the SS variation corner, as shown in Figure 9. These results confirm that our implimentation is viable solution for a low energy memory unit. Based on the results, we would recommend using this system for low-power applications, and certainly not for any cache close to a system CPU. It would be more suited for third level cache or as a main memory unit.

References

[1] Asta Skirbekk. Project in tfe4152 design of integrated circuits. Project description given to us by the course TFE4152, Fall 2024.

[2] Snorre Aunet. Lecture "domino logic, dram, power" in course tfe4152. Lecture given in the course TFE4152, Fall 2024.

[3] Martin Ken Carusone Tony C, Johns David A. *Analog integrated circuit design.* John Wiley & Sons, 2011.

7 Appendix

7.1 AIMSpice Code

funcTest.cir

```
*Test of functionality



.include .\subcircuits.cir



V1 1 0 dc Vdd
Vdata Data 0 PULSE(0 Vdd 50ns 0.1ns 0.1ns 50ns 100ns)
VdataN DataN 0 PULSE(Vdd 0 50ns 0.1ns 0.1ns 50ns 100ns)
Vwe WE 0 PULSE(0 Vdd 10ns 0.1ns 0.1ns 10ns 50ns)
Vre RE 0 PULSE(0 Vdd 30ns 0.1ns 0.1ns 10ns 50ns)
VreN REN 0 PULSE(Vdd 0 30ns 0.1ns 0.1ns 10ns 50ns)



xBitcell Data DataN WE RE REN outp 1 0 Bitcell
*R1 outp 0 1000k



.tran 10n 100n
*.plot v(Data) v(DataN)
*.plot v(WE) v(RE) v(REN)
*.plot v(outp)
*.plot i(v1)
.plot v(Data) v(WE) v(RE) v(outp)
```

leakTest.cir

```
*Test of leakage current



.include .\subcircuits.cir



*Test signals
V1 1 0 dc Vdd
Vdata Data 0 dc Vdd
VdataN DataN 0 dc 0
```

```
Vwe WE 0 dc Vdd
Vre RE 0 dc Vdd
VreN REN 0 dc 0



*Static bitcell for leakage measurement
xBitcell Data DataN WE RE REN outp 1 0 Bitcell
.tran 10n 100ns
*.op xBitcell
.plot i(V1)



*RESULTAT: -6.9e-09 A
```

readTest.cir

```
*Test of read time



.include .\subcircuits.cir



*Test signals
V1 1 0 dc Vdd
Vdata Data 0 PULSE(0 Vdd 1ns 0.1ns 0.1ns 3ns 10ns)
VdataN DataN 0 PULSE(Vdd 0 1ns 0.1ns 0.1ns 3ns 10ns)
Vwe WE 0 PULSE(0 Vdd 2ns 0.1ns 0.1ns 1ns 10ns)
Vre RE 0 PULSE(0 Vdd 5ns 0.1ns 0.1ns 3ns 10ns)
VreN REN 0 PULSE(Vdd 0 5ns 0.1ns 0.1ns 3ns 10ns)

*Testing functionality
xBitcell Data DataN WE RE REN outp 1 0 Bitcell
*R1 outp 0 1000k

.tran 1n 10n
*.plot v(Data) v(DataN)
*.plot v(WE) v(RE) v(REN)
*.plot v(outp)
*.plot i(v1)
.plot v(WE) v(RE) v(Data) v(outp)
```

writeTest.cir

```
*Test of write time



.include .\subcircuits.cir



*Test signals
V1 1 0 dc Vdd
Vdata Data 0 PULSE(0 Vdd 2.5ns 0.1ns 0.1ns 3ns 10ns)
VdataN DataN 0 PULSE(Vdd 0 2.5ns 0.1ns 0.1ns 3ns 10ns)
Vwe WE 0 dc Vdd
Vre RE 0 dc Vdd
VreN REN 0 dc 0

*Testing functionality
xBitcell Data DataN WE RE REN outp 1 0 Bitcell
*R1 outp 0 1000k

.tran 1n 10n
*.plot v(Data) v(DataN)
*.plot v(WE) v(RE) v(REN)
*.plot v(outp)
*.plot i(v1)
.plot v(WE) v(RE) v(Data) v(outp)
```

subcircuits.cir

```
*bitcell

.include .\gpdk90nm_ss.cir



*Parameters
.param Vdd=0.99
.options temp=27
.param L1=299n
.param W1=101n



*NAND
.subckt 2inNAND in1 in2 out1 Vdd Vss
xmp1 Vdd in1 out1 Vdd pmos1v L=L1 W=W1
xmp2 Vdd in2 out1 Vdd pmos1v L=L1 W=W1
```

```
xmn1 out1 in1 dn2 Vss nmos1v L=L1 W=W1
xmn2 dn2 in2 Vss Vss nmos1v L=L1 W=W1
.ends



*D-LATCH
.subckt Dlatch Data DataN WE outpQ1N Vdd Vss
xNAND1 Data WE Q0 Vdd Vss 2inNAND
xNAND2 DataN WE Q0N Vdd Vss 2inNAND
xNAND3 Q0 outpQ1N Q1 Vdd Vss 2inNAND
xNAND4 Q0N Q1 outpQ1N Vdd Vss 2inNAND
.ends



*Tri-state buffer
.subckt TSbuffer inp RE REN outp Vdd Vss
xmp1 T0 REN Vdd Vdd pmos1v L=L1 W=W1
xmp2 outp inp T0 T0 pmos1v L=L1 W=W1

xmn1 outp inp T1 T1 nmos1v L=L1 W=W1
xmn2 T1 RE Vss Vss nmos1v L=L1 W=W1
.ends



**Tri-state Test signals
*V1 1 0 dc Vdd
*Vinp inp 0 PULSE(0 Vdd 10ns 0.1ns 0.1ns 10ns 20ns)
*Vre RE 0 PULSE(0 Vdd 5ns 0.1ns 0.1ns 5ns 10ns)
*VreN REN 0 PULSE(Vdd 0 5ns 0.1ns 0.1ns 5ns 10ns)
*
*xTS1 inp RE REN outp 1 0 TSbuffer
*
*.tran 10n 20n
*.plot v(inp) v(RE) v(outp)



*Bitcell
.subckt Bitcell Data DataN WE RE REN outp Vdd Vss
xDL1 Data DataN WE Q1N Vdd Vss Dlatch
xTS1 Q1N RE REN outp Vdd Vss TSbuffer
.ends
```

```
**Test signals
*V1 1 0 dc Vdd
*Vdata Data 0 PULSE(0 Vdd 10ns 0.1ns 0.1ns 30ns 100ns)
*VdataN DataN 0 PULSE(Vdd 0 10ns 0.1ns 0.1ns 30ns 100ns)
*Vwe WE 0 PULSE(0 Vdd 20ns 0.1ns 0.1ns 10ns 100ns)
*Vre RE 0 PULSE(0 Vdd 50ns 0.1ns 0.1ns 10ns 100ns)
*VreN REN 0 PULSE(Vdd 0 50ns 0.1ns 0.1ns 10ns 100ns)

**Testing functionality
*xBitcell Data DataN WE RE REN outp 1 0 Bitcell
*R1 outp 0 1000k
*.tran 10n 100n

*.plot v(Data) v(DataN)
*.plot v(WE) v(RE) v(REN)
*.plot v(outp)
*.plot i(v1)

*.plot v(Data) v(WE) v(RE) v(outp)

*Static bitcell for leakage measurement
*xBitcell Data DataN WE RE REN Y Vdd 0 Bitcell
*.op xBitcell
```

7.2 Verilog Code

flipflop.v

```
module flipflop
    (
        input   inp,
        input   clk,
        output  outp
    );

    wire P0;
    wire P1;
    wire inpn;
    wire outpn;
    wire clkN;
    wire clkPE;

    not(clkN, clk);
    and(clkPE, clk, clkN);

    not(inpn, inp);
    nand(P0, inp, clkPE);
```

```
    nand(P1, inpn, clkPE);
    nand(outp, P0, outpn);
    nand(outpn, P1, outp);
endmodule
```

flipflop_tb.v

```
`include "flipflop.v"

module flipflop_tb;
    reg inp;
    reg clkPE;
    wire outp;

    flipflop ff1 (
        .inp(inp),
        .clkPE(clkPE),
        .outp(outp)
    );

    always begin
        #9 clkPE = 1;
        #1 clkPE = 0;
    end

    initial begin
        $dumpfile("flipflop_waveform.vcd");
        $dumpvars(1, flipflop_tb);

        inp = 0;
        clkPE = 0;

        #5 inp <= 1;
        #10 inp <= 0;
        #10 inp <= 1;
        #10 inp <= 0;
        #100 $finish;
    end
endmodule
```

bitcell.v

```
/*
Se vedlegg for kretskjema
Enkel bitcelle i SRAM modul.
WriteEnable og ReadEnable kan aldri være høy (1) samtidig. Ref modul ByteCell. */

module bitcell
    (
```

```
        input   inp,
        input   inpn,
        input   we,
        input   re,
        input   ren,
        output  outp
    );

    wire Q0;
    wire Q0n;
    wire Q1;
    wire Q1n;
    wire T0;
    wire T1;

    // supply rails. Logical 1
    supply1 VDD;
    supply0 GND;

    // D-latch
    nand(Q0, inp, we);
    nand(Q0n, inpn, we);
    nand(Q1, Q0, Q1n);
    nand(Q1n, Q0n, Q1);

    // Tri-state buffer
    pmos(T0, VDD, Q1n);
    nmos(T1, GND, Q1n);
    pmos(outp, T0, ren);
    nmos(outp, T1, re);
endmodule
```

bitcell_tb.v

```
`include "bitcell.v"        // include the module

module bitcell_tb;
    reg     inp;
    reg     inpn;
    reg     re;
    reg     ren;
    reg     we;
    wire    outp;

    bitcell bitcell_inst1
    (
        .inp    (inp),
```

```
        .inpn   (inpn),
        .we     (we),
        .re     (re),
        .ren    (ren),
        .outp   (outp)
    );

    initial begin
        $dumpfile("bitcell_waveform.vcd");
        $dumpvars(1, bitcell_tb);

        inp     = 0;
        inpn    = 1;
        re      = 0;
        ren     = 1;
        we      = 0;

        #5 re  <= 1; ren <= 0;
        #5 re  <= 0; ren <= 1;
        #5 inp <= 1; inpn <= 0; we <= 1;
        #5 we  <= 0;
        #5 inp <= 0; inpn <= 1;
        #5 re  <= 1; ren <= 0;
        #5 re  <= 0; ren <= 1; we <= 1;
        #5 we  <= 0;
        #5 re  <= 1; ren <= 0;
        #5 re  <= 0; ren <= 1;
        #5;
    end
endmodule
```

bytecell.v

```
/*
Se vedlegg for kretskjema.
En bytecelle bestående av 8 stk bitceller med enkel logikk for kontroll av
utgangsignalene WriteEnable og ReadEnable fra inngangsignalene sel(ect) og op(eration). .
Tar inn 8 bit inp(ut) signal og fordeler informasjonen til de tilhørende 8 bitcellene.
gir tilsvarende outp(ut) fra de 8 innbyrdes bitcellene.
Modul Bitcelle er konstruert i filen bitcelle.v
 */

`include "bitcell.v"                // include bitcell module to the script

module bytecell
    (
        input   [7:0]   inp,        // input
        input   [7:0]   inpn,       // input not
```

```
        input           op,         // operation, write = 1, read = 0
        input           sel,        // select. 1 if read/write operation is valid for one bytecell
        output  [7:0]   outp        // output
    );

    wire re;                    // read enable
    wire ren;                   // read enable not
    wire we;                    // write enable
    wire opn;

    // F Block. Se vedlegg for kretskjema
    not(opn, op);                   // opn = ~op
    and(we, sel, op);               // we = sel & op
    nand(ren, sel, opn);            // ren = sel & ~op
    not(re, ren);                   // re = ~ren

    // 8 stk bitceller
    genvar i;   generate
        for(i = 0; i < 8; i = i + 1) begin : bitcell_insts_i
            bitcell bitcells
            (
                .inp    (inp[i]),
                .inpn   (inpn[i]),
                .we     (we),
                .re     (re),
                .ren    (ren),
                .outp   (outp[i])
            );
        end
    endgenerate
endmodule
```

bytecell_tb.v

```
`include "bytecell.v"                   // include the module


// Test Benche for the ByteCell module
module bytecell_tb;
    reg     [7:0]   inp;
    reg     [7:0]   inpn;
    reg             op;
    reg             sel;
    wire    [7:0]   outp;

    bytecell bytecell_insts1
    (
        .inp    (inp),
```

```
        .inpn   (inpn),
        .op     (op),
        .sel    (sel),
        .outp   (outp)
    );

    initial begin
        $dumpfile("bytecell_waveform.vcd");
        $dumpvars(1, bytecell_tb);

        inp = 8'b00000000;
        inpn = 8'b11111111;
        op = 0;
        sel = 0;

        #5 sel  <= 1;
        #5 op   <= 0;               inp <= 8'b10101010; inpn <= 8'b01010101;
        #5 op   <= 1;
        #5 inp  <= 8'b11001100;     inpn <= 8'b00110011;
        #5 op   <= 0;               inp <= 8'b11110000; inpn <= 8'b00001111;
        #5 sel  <= 0;
        #5 op   <= 1; sel <= 1;
        #5 op   <= 0;               inp <= 8'b01010101; inpn <= 8'b10101010;
        #5 op   <= 1; sel <= 0;
        #5 op   <= 0;               inp <= 8'b00000000; inpn <= 8'b11111111;
        #5 sel  <= 1;
        #5;
    end
endmodule
```

demux1to8bit.v

```
/*
8 to 1 demux module
Se vedlegg for kretskjema
*/
module demux1to8bit
    (
        input           inp,
        input   [2:0]   adr,
        output  [7:0]   outp
    );

    wire    [2:0]   not_adr;

    not(not_adr[0], adr[0]);
    not(not_adr[1], adr[1]);
    not(not_adr[2], adr[2]);
```

```
    and(outp[0],    inp,    not_adr[2],    not_adr[1],    not_adr[0]);
    and(outp[1],    inp,    not_adr[2],    not_adr[1],    adr[0]);
    and(outp[2],    inp,    not_adr[2],    adr[1],        not_adr[0]);
    and(outp[3],    inp,    not_adr[2],    adr[1],        adr[0]);
    and(outp[4],    inp,    adr[2],        not_adr[1],    not_adr[0]);
    and(outp[5],    inp,    adr[2],        not_adr[1],    adr[0]);
    and(outp[6],    inp,    adr[2],        adr[1],        not_adr[0]);
    and(outp[7],    inp,    adr[2],        adr[1],        adr[0]);

endmodule
```

demux1to8bit_tb.v

```
`include "demux1to8bit.v"                    // include the module

// Test bench for 1 to 8 dembux
module demux1to8bit_tb;
    reg             inp;
    reg     [2:0]   adr;
    wire    [7:0]   outp;

    demux1to8bit demux1to8bit_inst1 (
        .inp    (inp),
        .adr   (adr),
        .outp   (outp)
    );

    initial begin
        $dumpfile("demux1to8bit_wavefile.vcd");
        $dumpvars(1, demux1to8bit_tb);

        inp     = 1'b0;
        adr    = 3'b000;

        #5;
        #5  inp  <= 1'b1;
        #5  adr <= 3'b001;
        #5  adr <= 3'b010;
        #5  adr <= 3'b011;
        #5  adr <= 3'b100;
        #5  adr <= 3'b101;
        #5  adr <= 3'b110;
        #5  adr <= 3'b111;
        #5  inp  <= 1'b0;
        #5;
```

```
    end
endmodule
```

ram.v

```
`include "bytecell.v"
`include "demux1to8bit.v"

module ram
    (
        input   [7:0]   inp,
        input   [2:0]   adr,
        input           op,
        input           sel,
        output  [7:0]   outp
    );

    wire    [7:0]   outp_demux;
    wire    [7:0]   inpn;

    not(inpn[0], inp[0]);
    not(inpn[1], inp[1]);
    not(inpn[2], inp[2]);
    not(inpn[3], inp[3]);
    not(inpn[4], inp[4]);
    not(inpn[5], inp[5]);
    not(inpn[6], inp[6]);
    not(inpn[7], inp[7]);

    demux1to8bit demux1to8bit_inst1
    (
        .inp    (sel),
        .adr    (adr),
        .outp   (outp_demux)
    );

    // instantiate 8 bytecell modules
    genvar i; generate
        for (i = 0; i < 8; i = i + 1) begin : bytecell_insts1
            bytecell bytecell_inst (
                .inp    (inp),
                .inpn   (inpn),
                .op     (op),
                .sel    (outp_demux[i]),
                .outp   (outp)
            );
        end
    endgenerate
```

```
endmodule
```

ram_tb.v

```
`include "ram.v"                // include the module

// Test Bench for RAM modul
module ram_tb;
    reg     [7:0]   inp;
    reg     [2:0]   adr;
    reg             op;
    reg             sel;
    wire    [7:0]   outp;

    ram ram_inst1 (
        .inp    (inp),
        .adr   (adr),
        .op     (op),
        .sel    (sel),
        .outp   (outp)
    );

    initial begin
        $dumpfile("ram_waveform.vcd");
        $dumpvars(1, ram_tb);

        inp  = 8'b00000000;
        adr = 3'b000;
        op   = 1'b0;
        sel  = 1'b0;

                #10 sel <= 1'b1;    op <= 1'b1;     inp <= 8'b10101010;     adr <= 3'b001;
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b0;     adr <= 3'b001;                              // read
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b1;     inp <= 8'b11001100;     adr <= 3'b010;      // writ
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b0;     adr <= 3'b010;                              // read
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b1;     inp <= 8'b11110000;     adr <= 3'b011;      // writ
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b0;     adr <= 3'b011;                              // read
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b1;     inp <= 8'b00001111;     adr <= 3'b100;      // writ
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b0;     adr <= 3'b100;                              // read
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b1;     inp <= 8'b00110011;     adr <= 3'b101;      // writ
```

```
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b0;     adr <= 3'b101;                                  // read
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b1;     inp <= 8'b01010101;     adr <= 3'b110;          // writ
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b0;     adr <= 3'b110;                                  // read
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b1;     inp <= 8'b10011001;     adr <= 3'b111;          // writ
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b0;     adr <= 3'b111;                                  // read
        #10 sel <= 1'b0;

        #10;

/*          #10 sel <= 1'b1;    op <= 1'b1;     inp <= 8'b10101010;     adr <= 3'b001;          // w
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b0;     adr <= 3'b001;                                  // read
        #10 sel <= 1'b0;
        #10 sel <= 1'b1;    op <= 1'b0;     adr <= 3'b010;                                  // read
        #10 op  <= 1'b1;    inp <= 8'b10101011;                                              // wri
        #10 op  <= 1'b0;
        #10 adr <= 3'b001;

        #10; */
    end
endmodule
```

FSM.v

```
`include "flipflop.v"

module fsm
    (
        input op,
        input sel,
        input clk,
        output valid,
        output rw
    );

    wire P0;
    wire P1;
    wire P2;
    wire P3;
    wire P4;


    and(P0, op, sel);
```

```
    or(P3, P0, P1);
    not(P1, P2);
    nand(P2, valid, rw);
    and(P4, P2, sel);


    flipflop ff1 (
        .inp(P4),
        .clk(clk),
        .outp(valid)
    );

    flipflop ff2 (
        .inp(P3),
        .clk(clk),
        .outp(rw)
    );
endmodule
```

FSM_tb.v

```
`include "FSM.v"

module FSM_tb;
    reg clk;
    reg op;
    reg sel;
    wire valid;
    wire rw;

    fsm fsm1 (
        .op(op),
        .sel(sel),
        .clk(clk),
        .valid(valid),
        .rw(rw)
    );

    always begin
        #5 clk = 1;
        #5 clk = 0;
    end

    initial begin
        $dumpfile("FSM_waveform.vcd");
        $dumpvars(1, FSM_tb);

        clk = 0;
```

```
        op  = 0;
        sel = 0;

        #30 op <= 0; sel <= 1; // read, valid = 1, rw = 0
        #20;
        #10 op <= 1; sel <= 1; // write, valid = 1, rw = 1
        #20; // occilate between write and stable
        #10 op <= 0; sel <= 0;

        #50 $finish;
    end
endmodule
```

mem_system.v

```
`include "ram.v"
`include "FSM.v"

module mem_system
    (
        input clk,
        input [7:0] i,
        input [2:0] adr,
        input op, select,
        output [7:0] o
    );

    wire valid_connect;
    wire rw_connect;

    fsm fsm1 (
        .op(op),
        .sel(select),
        .clk(clk),
        .valid(valid_connect),
        .rw(rw_connect)
    );

    ram ram1 (
        .inp(i),
        .adr(adr),
        .op(rw_connect),
        .sel(valid_connect),
        .outp(o)
    );
endmodule
```

mem_system_tb.v

```
`include "mem_system.v"

module mem_system_tb;

    reg clk;
    reg [7:0] i;
    reg [2:0] adr;
    reg op, select;
    wire [7:0] o;

    // Instantiate the Unit Under Test (UUT)
    mem_system mem_system1 (
        .clk(clk),
        .i(i),
        .adr(adr),
        .op(op), .select(select),
        .o(o)
    );

    always begin
        #5 clk = 1;
        #5 clk = 0;
    end

    initial begin
        $dumpfile("mem_system_waveform.vcd");
        $dumpvars(1, mem_system_tb);

        clk = 0;
        i = 0;
        adr = 0;
        op = 0; select = 0;

        #25;
        //Write "LS SWK !" to memory
        op = 1; select = 1;
        adr = 3'b000; i = 8'b01001100; // ASCII 'L'
        #20 adr = 3'b001; i = 8'b01010011; // ASCII 'S'
        #20 adr = 3'b010; i = 8'b00100000; // ASCII space
        #20 adr = 3'b011; i = 8'b01010011; // ASCII 'S'
        #20 adr = 3'b100; i = 8'b01010111; // ASCII 'W'
        #20 adr = 3'b101; i = 8'b01001011; // ASCII 'K'
        #20 adr = 3'b110; i = 8'b00100000; // ASCII space
        #20 adr = 3'b111; i = 8'b00100001; // ASCII '!'
        #20;
```

```
        op = 0; select = 0; adr=3'b000; i = 8'b00000000;
        #30;

        //Read from memory
        #10 op = 0; select = 1; adr = 3'b000;
        #10 adr = 3'b001;
        #10 adr = 3'b010;
        #10 adr = 3'b011;
        #10 adr = 3'b100;
        #10 adr = 3'b101;
        #10 adr = 3'b110;
        #10 adr = 3'b111;

        #10 op = 0; select = 0; adr=3'b000;

        #100 $finish;
    end

endmodule
```