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

TFE4152 Design of Integrated Circuits

Course project fall 2024

Design of 64bit memory unit for an IoT sensor using 90nm CMOS technology

Authors:
Adam Jonas
and
Md. Toushif Islam Shoueb
Group: 26

Abstract

This project focuses on the design, simulation, and implementation of an 8x8-bit memory unit aimed at meeting the specific needs of IoT applications. The design is optimized for low power usage and efficient data storage using 90 nm technology. The main goals of the project are to create a 1-bit memory cell using basic logic gates, then build an 8x8 memory array, develop an address decoder for correct memory access, and design a finite-state machine (FSM) to control read and write operations. The cell is also tested through simulations using AIMSpice to verify its robustness and ensure that it works well under different conditions and temperatures.

The process starts with designing a 1-bit memory cell using an SR latch with logic gates, control signal generation block, and a tristate buffer. This memory cell is then arranged into 8-bit words and 8 words are combined into rows, creating the 8x8 memory array that can store 64 bits of data. An FSM is developed to manage the read and write operations based on external input signals. The address decoder is used to ensures the right memory word is selected for reading or writing. The design is validated at a higher abstraction level through Verilog simulations using ActiveHDL, ensuring the correct functionality of the memory unit design.

The final output is a fully simulated and verified memory unit capable of efficient data storage and retrieval. The report provides a detailed overview of the design approach, simulation results, and challenges faced during the project, offering valuable information on memory system development for IoT applications.

1 Introduction

We are employed in a start-up IoT-company that is working on designing its own sensor unit. The unit needs a small memory array containing eight words, each word having eight bits, and it is our task to design it. This equates to an 8x8 bit memory, which means that it should contain a total of 64 bits (each bit is stored by bitcell). The gpdk 90-nm technology is going to be used. Both AimSpice and Verilog simulations in ActiveHDL are required to complete the task. Further specifications are required:

Table 1: Required specifications for memory unit

Parameter	Value
Supply voltage	0.3V $\leq$ 1V
Transistor width	$100nm \leq W \leq 1500nm$
Transistor length	$100nm \leq L \leq 300nm$
Technology	90nm technology
Power consumption	minimize
Maximum read/write time	$\leq$3ns

After figuring out the design schematics, we need to transfer it into Verilog code (gate level code only) and verify functionality of our design. Hierarchical design is recommended. No logic gate is allowed to have fan-in above 4.

First, we introduce Section 2 Theory, in which we present necessary theory facts used for our design, it includes basic information on logic gates, latches, static CMOS, decoder and FSM. It is followed by chapter 3 Method, where we present schematic design implemented in KiCad and the function principle of the memory. The same components that were described in the Theory section are now described in the way we designed it. After this chapter, the fourth and most important

chapter comes in Simulations and results. Here we present how we implemented our theoretical design into Verilog code and verified its functionality. AIM-Spice simulations were conducted, described, and results are shown. We finish our work with discussion of results, further improvements, optimization of design, and conclusion of what we achieved. In the very end are attached screens from the codes.

2 Theory

Memory systems are a key part of digital circuits, allowing devices to store and use data. They help systems keep information temporarily or permanently, depending on the application. Without memory, most digital systems could not perform their tasks effectively.

Memory Array: The core of data storage is a memory array. It is a structured way to store data in digital systems. It consists of multiple memory cells arranged in rows and columns. A single bit of information, either a 0 or a 1, which is the basic binary language of computers, can be stored in each memory cell, which is a tiny unit.

The rows and columns in a memory array allow for efficient access to the data. A specific data point is located using row and column addresses, much like finding a cell in a spreadsheet. This type of organization makes data reading and writing faster and more manageable.

In this project, the focus is on designing a static memory array, a type of memory that is often used in embedded systems and IoT devices. As long as power is available, static memory arrays store data, ensuring faster access and reliability. These arrays form the backbone of cache memory, microcontrollers, and other performance-critical applications.

To successfully design such a system, it is essential to understand key concepts such as memory architecture, the role of decoders in addressing, and the mechanics of read and write operations. Additionally, familiarity with digital logic components, such as flip-flops and latches, provides the necessary groundwork to implement the fundamental building blocks of the memory array.

2.1 Logic gates and Latch

Logic gates are electronic circuits that can be used to implement the most elementary logic expressions, also known as Boolean expressions. The logic gate is the most basic building block of combinatorial logic. There are three basic logic gates, namely the OR gate, the AND gate and the NOT gate. Other logic gates derived from these basic gates are the NAND gate, the NOR gate, the EXCLUSIVE OR gate, and the EXCLUSIVE-NOR gate.

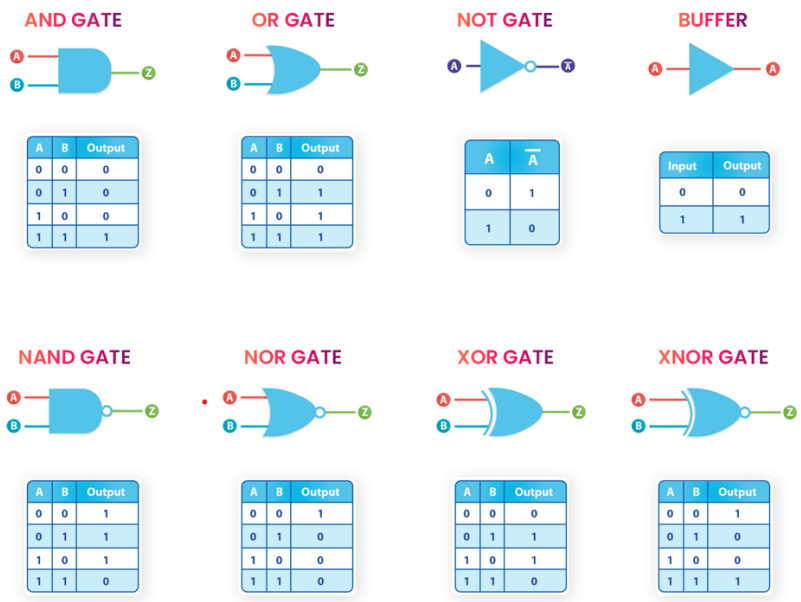


Figure 1: Diagram and Truth Table of Logic Gates

In fact, it is possible to use either only NAND gates or only NOR gates to implement any Boolean expression. This is possible because laws of boolean algebra allow it, therefore NAND or OR logic gates are enough to create any logic function. For this reason, NAND and NOR gates are called universal gates.

Boolean algebra is a branch of algebra that deals with variables and operations that have two possible values: true and false (commonly represented as 1 and 0). It is the mathematical foundation of digital logic and computer science, used to describe and analyze logical operations and digital circuits.

Some important theorem of Boolean algebra :
1. Identity Laws: $A + 0 = A; A \cdot 1 = A;$
2. Null Laws: $A + 1 = 1; A \cdot 0 = 0;$
3. Complement Laws: $A + A' = 1; A \cdot A' = 0;$
4. Distributive Laws: $A \cdot (B + C) = (A \cdot B) + (A \cdot C); A + (B \cdot C) = (A + B) \cdot (A + C);$
5. De Morgan's Theorems: $(A \cdot B)' = A' + B'; (A + B)' = A' \cdot B';$
6. Double Negation Law: $(A')' = A;$

SR Latch: The SR latch (Set-Reset latch) is a fundamental sequential logic circuit that can store a single bit of information. It has with two inputs, Set (S) and Reset (R), and two outputs, Q and its complement Q'. It stores inputs when inputs are 01 or 10. When both inputs are low (S = 0, R = 0), the latch retains its previous state, while the condition S = 1, R = 1 creates an undefined state, leading to logical inconsistency. Typically implemented using NOR or NAND gates. While it is simple in design and efficient for basic applications, the undefined state requires careful handling to prevent errors.

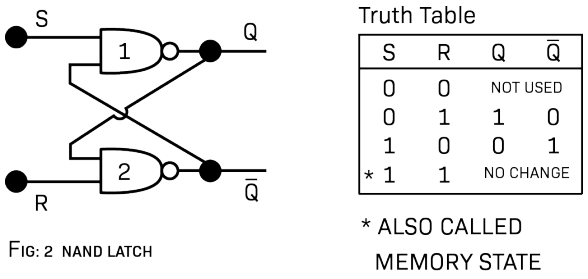


Figure 2: NAND-based SR latch and its truth table

2.2 Static CMOS

Static CMOS logic is the most widely used logic family in digital circuit design due to its robustness, simplicity, and low power consumption. It operates by using complementary nMOS and pMOS transistors to create logic gates that provide well-defined and stable output levels. The nMOS (pull-down network) connects the output to ground (0), while the pMOS (pull-up network) connects the output to the supply voltage (1). These networks are designed such that one is ON while the other is OFF for any input pattern, ensuring robust logic levels. Any Boolean expression and logic gates can be represented by a number of CMOS transistors. For example, the diagram below is the CMOS representation of a NOT gate [1].

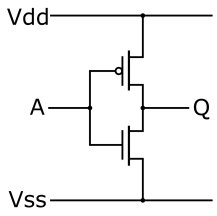


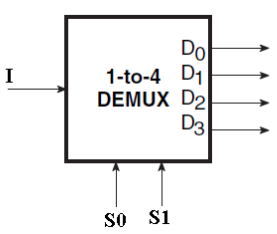
Figure 3: Inverter: CMOS representation of NOT gate

Here, when the input is high (1), the NMOS is activated and the output connects to V_{ss}, which is ground (0). On the other hand, when the input is low (0), the pMOS is activated and the output is connected to V_{dd}, which is high(1). Similarly, NAND gates can be employed by series transistors for the pull-down network and parallel transistors for the pull-up network. More complex gates utilize combinations of series and parallel transistors.

The output of a CMOS logic gate can be in one of four states: logic 1, logic 0, high impedance (Z), or contention (X). Logics 1 and 0 occur when one network is ON and the other is OFF. High impedance arises when both networks are OFF, which is crucial for devices like multiplexers and memory elements. Contention occurs when both networks are on, leading to indeterminate output and unwanted power dissipation. Despite potential issues, static CMOS gates remain the most widely used technology due to their efficiency and reliability.

2.3 Demultiplexer and Decoder

A demultiplexer is a combinatorial logic circuit that routes a single input to one of the 2^2 output lines, based on the value of the n selected lines. It effectively acts as a data distributor, where the active output line is determined by the selection line configuration. For example, in Figure 4, a 1-to-4 demultiplexer routes the input to one of four output lines, as shown in its circuit diagram and truth table.



I	Select		O/P			
	S0	S1	D0	D1	D2	D3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Figure 4: 1-to-4 demultiplexer

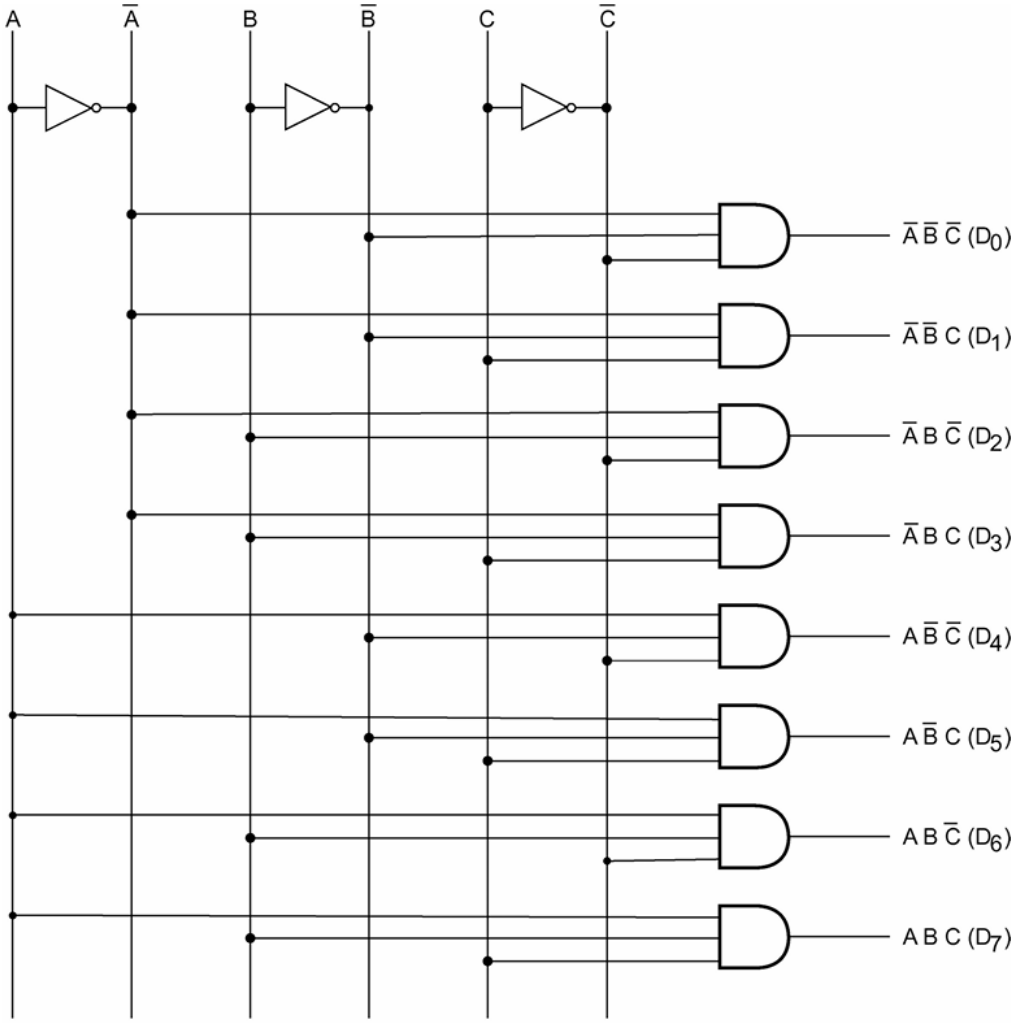
A decoder, on the other hand, is a specific type of demultiplexer without an input line. It decodes n input lines into a maximum of 2^n unique outputs. For instance, a 3-to-8 line decoder has three inputs and eight outputs, with only one output active for any given input combination. [2] A decoder is commonly used to generate minterms, where each output represents a specific minterm corresponding to the input binary code. For example, a 3-to-8 decoder produces eight minterms:

$$D_0 = A'B'C', D_1 = A'B'C, D_2 = A'BC', D_3 = A'BC, D_4 = AB'C', D_5 = AB'C, D_6 = ABC', D_7 = ABC.$$

These minterms can be combined to implement any Boolean function. For instance, the Boolean function:

$$Y = ABC + A'BC + AB'C + A'B'C$$

Shown logic can be realized using a 3-to-8 decoder, where the outputs are summed using an external OR gate.



INPUTS			OUTPUTS							
A	B	C	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Figure 5: Logic diagram and truth table of a 3-to-8 line decoder.

2.4 Finite State Machine

A Finite-State Machine (FSM) is a fundamental concept in digital circuit design and plays a crucial role in the implementation of control logic for sequential systems. FSMs are used to model and implement systems that transition between predefined states based on input signals and internal conditions. They are especially useful in the design of hardware such as processors, controllers, and communication protocols.

An FSM operates with a finite number of states and transitions between them based on inputs. It is made up of two primary types:

Combinatorial Logic: Determines the next state and output based on the current state and inputs. Memory Elements: Typically implemented with flip-flops, these elements store the current state information.

There are two common FSM models:

Mealy Machine: The output depends solely on the current state. Figure 7 shows the Mealy machine state diagram, where q0 and q1 are two states, (x1,x2) is the input set, and (y1,y2) is the output set.

Moore machine: The output depends on both the current state and the input. Figure 6 shows the state diagram of Moore machine, where q0 and q1 are two states, (x1,x2) is the input set, (y1,y2) is the output set.

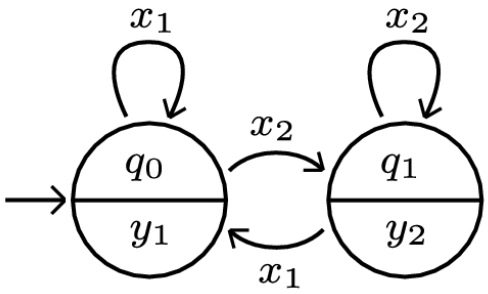


Figure 6: Moore Machine

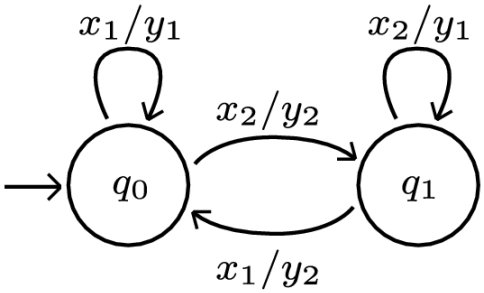


Figure 7: Mealy Machine

FSMs are integral to the design of sequential circuits, such as counters, registers, and communication interfaces, where precise control of state transitions is essential. Using FSMs, complex behaviors can be effectively managed and implemented in hardware.

3 Method

This section outlines the methodology used to implement the system, describing the steps from the design phase to the final circuit realization. The process involved the development of key components, including memory, the Finite State Machine (FSM), and other essential blocks. Design choices are justified with references to both theoretical concepts and practical considerations.

The design process began with a clear understanding of the functional requirements of the system, followed by the design of individual components. We have created a detailed block diagram of our whole design with appropriate connections and logic gates using a KiCad electronic scheme design and simulation software. Then transferred the design into Verilog gate level code. For the FSM, a state diagram was given, which was translated into a truth table and implemented using flip-flops and combinatorial logic. Memory elements were designed using static CMOS logic, with careful attention given to transistor sizing to achieve optimal performance. Once each component was designed, simulations were performed to test and verify the overall functionality.

3.1 64 bit memory

The design process begins with understanding how individual bitcells are arranged to form a larger memory structure. By analyzing how these bitcells are connected and integrated with FSM and Decoder. This section explores how the design was expanded from a single bitcell to a complete 64-bit memory block.

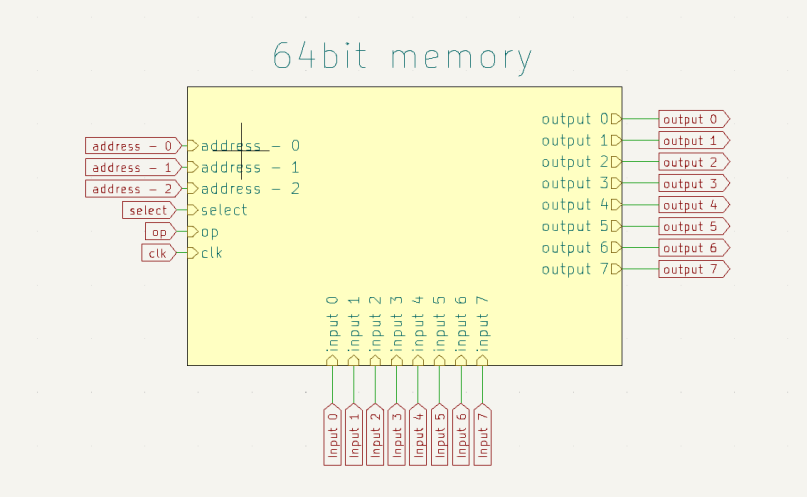


Figure 8: Memory: Top layer view

Figure 8 shows the view of the top layer of our designed memory. Inputs here are: - 3 bit address, named address-0, address-1, and address-2. This address line is used to select the corresponding memory word for operation, i.e. reading or writing. - 8 bit input, named input i; i=0,1,2,3,4,5,6,7; - op; defines the mode of operations. If op=0, it will perform the read operation and if op=1, it will perform write operation to the bitcell.

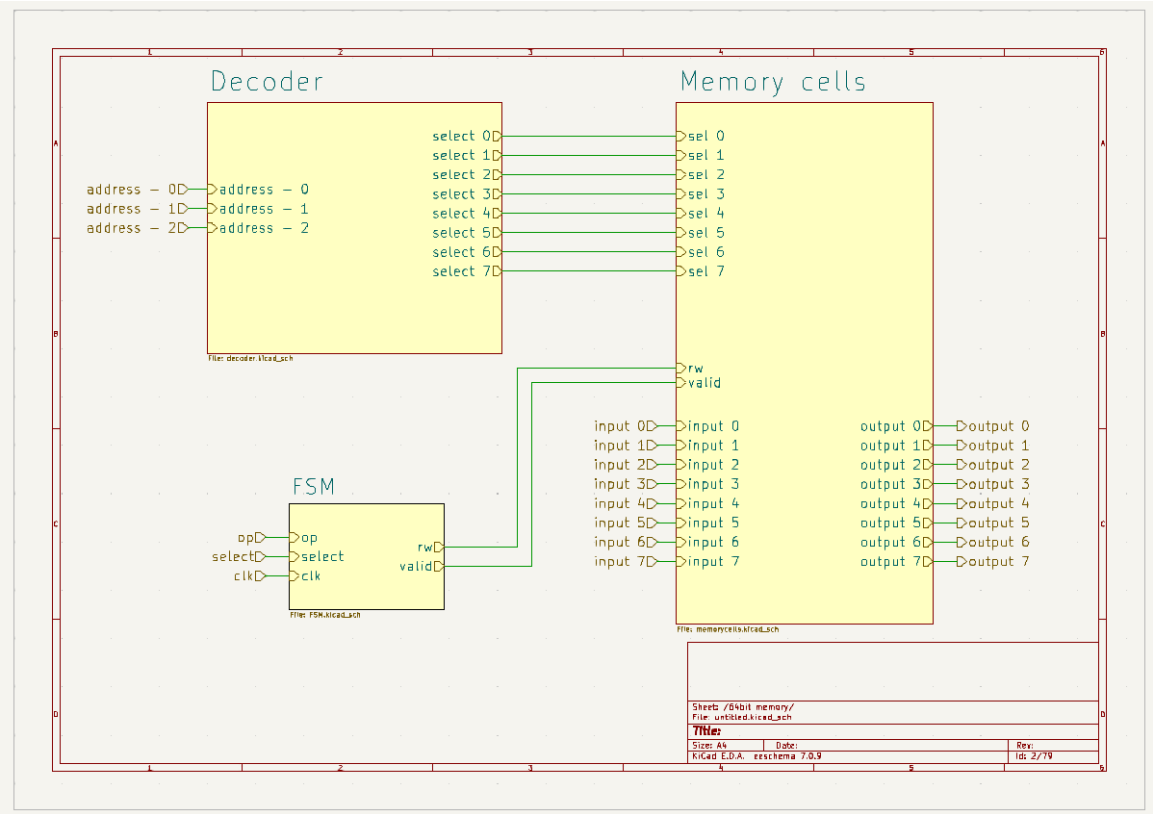


Figure 9: Memory: second layer view, placement of all integrated parts

How the FSM, decoder, and memory cells are integrated within the memory system is illustrated in Figure 9. The decoder decodes the address bits and enables one of the eight select lines. These select lines are connected to each word in the memory, enabling access to the corresponding word. The finite state machine (FSM) defines the mode of operation and synchronizes with the clock signal to generate a valid signal, ensuring proper execution of the operation.

In Figure 10, the connections of the eight words within the memory cell are illustrated in detail. The selection of a specific word is controlled by a 3-bit address, which determines the high state of the corresponding select line. This select line is then fed into an AND gate, along with another input signal, 'sel' which is generated by the FSM. The 'sel' signal indicate if read or write operation is allowed. If sel = 1, the AND gate output [see section 2.1] enables the selected word for a write/read operation, allowing it to take the input data and store them within the memory cell. This coordinated mechanism ensures accurate word selection and reliable data storage based on input address and operating mode.

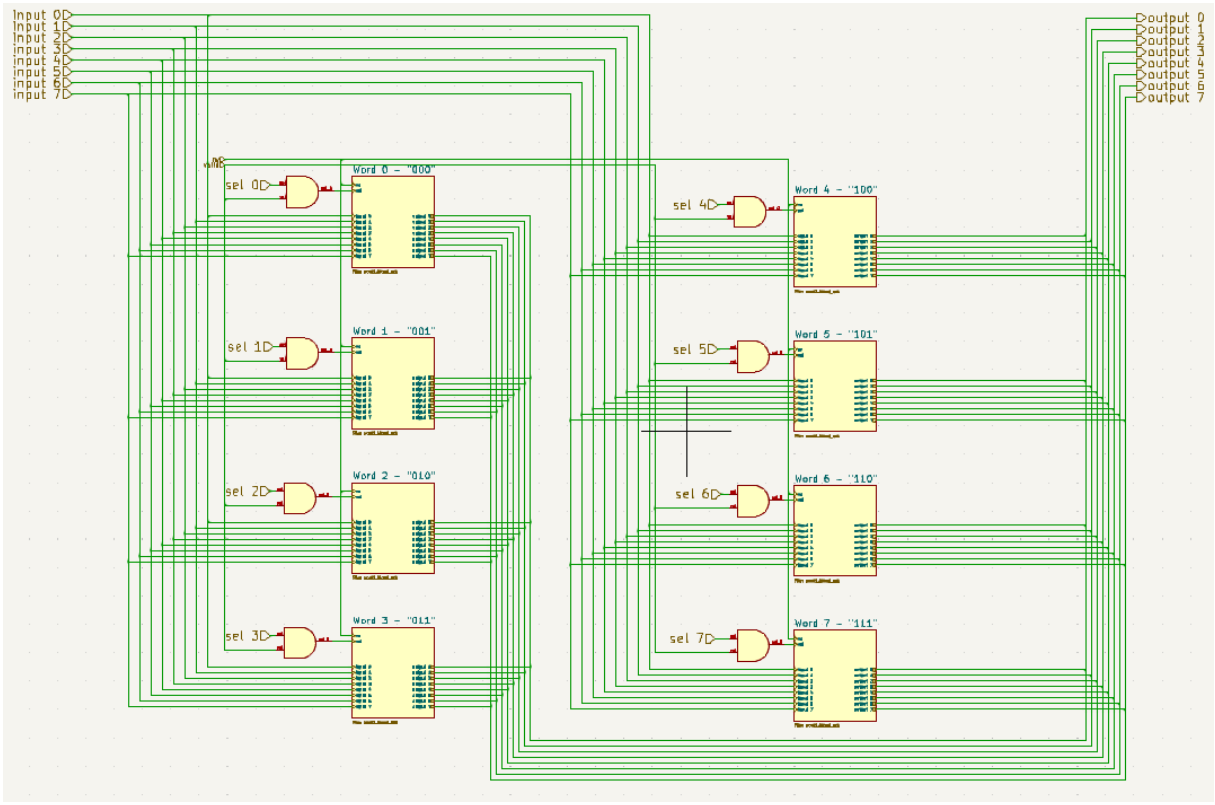


Figure 10: Inside the Memory Cell: representation of words

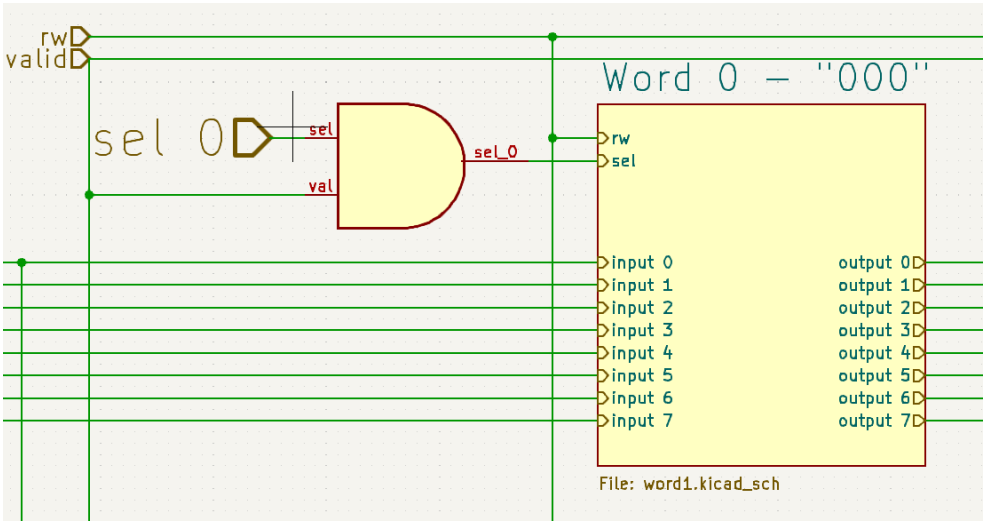


Figure 11: Detailed screenshot of the word

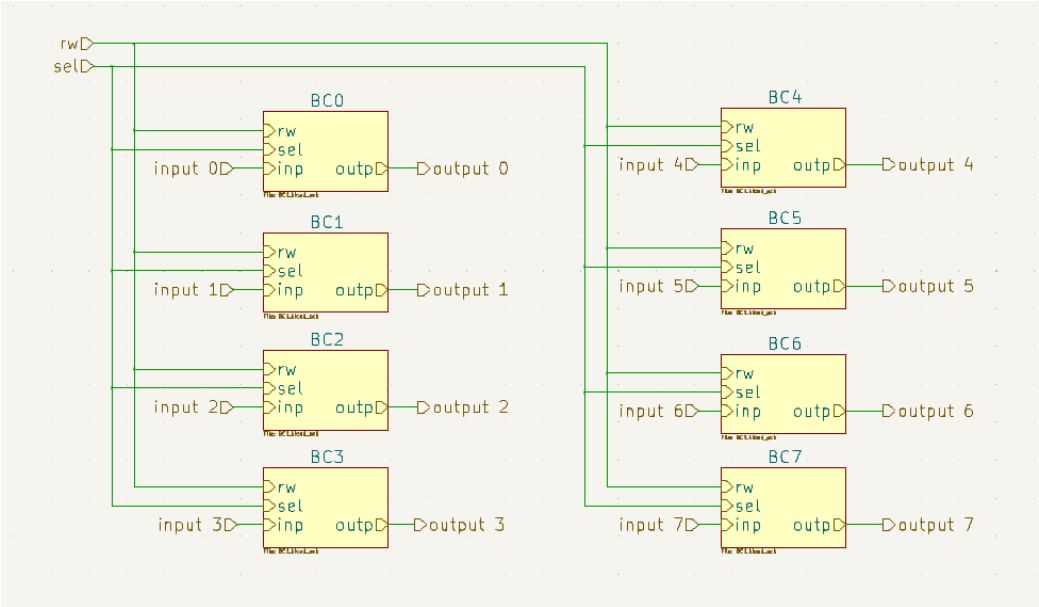


Figure 12: Inside a single Word: how bitcells are placed

Figure 12 illustrates the structure of a single word in the memory, which consists of eight bitcells arranged in sequence. Each bitcell is responsible for storing a single bit of data, making the entire word capable of holding 8 bits. The sel signal, when high, activates all bitcells in the word, enabling them for either reading or writing operations.

In write mode (rw = 1), the eight input bits are fed into the bitcells simultaneously. These bits are stored in the bitcells as per their respective positions, effectively completing the write operation. In read mode (rw = 0), the data stored in the bitcells are retrieved and transmitted to the output lines, providing an 8 bit output.

This configuration ensures that data flow smoothly in both directions: inputs are written precisely and outputs are read efficiently. By combining the sel signal and rw control, the memory can selectively access specific words while maintaining reliable operation for reading and writing processes.

3.2 The bitcell

In this project, the bitcell serves as the core building block of the memory circuit, playing a critical role in the storage and retrieval of individual bits of data. This section provides a detailed explanation of how the bitcell was implemented, starting with its design at the logic gate level and then exploring its structure at the transistor level with supporting figures.

The choice of the specific circuit topology for the bitcell is explained, highlighting its efficiency and suitability for this memory design. Key design decisions, such as selecting the width and length of the transistors, are discussed in detail, along with the factors that influenced these choices, including performance requirements and power optimization. Additionally, the rationale for selecting the supply voltage is provided, focusing on how it balances operational stability and energy efficiency.

At the very beginning of our project, we designed the bitcell. To designing it, we used SR latch, which can hold one bit data [see section 2.1]. Our designed bitcell is represented in Figure 13 using

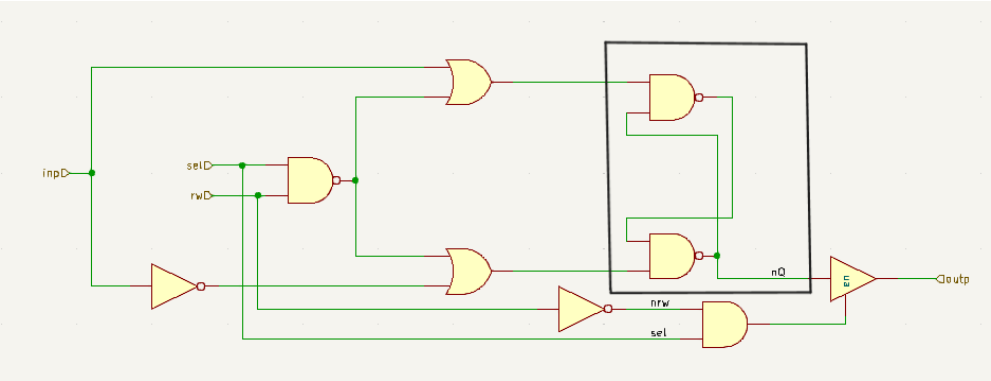


Figure 13: Logic gate-level design of bitcell

logic gates. In the gray rectangle is marked the NAND base of the SR latch. The sel (select) and rw (read/write) signals are fed into a NAND gate. If either of these signals is low (0), the NAND gate outputs 1. This output is passed through an OR gate and directed to the S and R inputs of the SR latch. As a result, when the NAND gate outputs 1, both inputs of the SR latch (S and R) become 1 [see the truth table of logic gates in Section 2.1]. Referring to the SR latch truth table [see Section 2.1], this condition causes the latch to hold its previous state, preserving the last stored value.

When sel and rw are both 1(write mode): The NAND gate outputs 0, enabling the current input value to be passed to the S input and its inverted value to the R input. Based on the SR latch truth table, this set-up updates the latch, with the output Q' reflecting the same value as the current input.

When sel = 1 and rw = 0 (Read Mode): The bitcell is selected for a read operation. In this case, the logic ensures that both S and R are set to 1, keeping the previously stored value intact (as the SR latch retains its state). Simultaneously, sel and the inverted rw signal are passed to another AND gate located below the black box. This AND gate outputs a high signal (1) to activate the tristate buffer.

When the tristate buffer is enabled (high), it connects the Q' output of the latch to the output line.

When sel = 0 or rw = 1 (Write mode or Unselected State): The tristate buffer is disabled, producing a high-impedance state (Z) on the output line. This prevents the bitcell from interfering with the output line during write operations or when the cell is not selected.

Role of the Tristate Buffer: The tristate buffer plays a key role in the proper functioning of the memory circuit. It allows the bitcell to be connected to the output line only when it is selected (sel = 1) and is in read mode (rw = 0). This ensures that only one bitcell at a time can influence the output, preventing multiple bitcells from driving the output line simultaneously and causing errors. By using the tristate buffer, we ensure that the memory operates reliably and efficiently, allowing for correct read and write operations without interference from unselected or nonactive bitcells.

The logic gate-level design is then converted into gate level design using CMOS circuits. As we discussed earlier about the conversion of any logic gate to CMOS representation in Section 2.1, we used that technique to convert all logic gates into transistor. This design needs 44 transistors in total. In Figure 14 yellow rectangle marks the SR-latch, the blue control signal generation block, and the red tristate buffer.

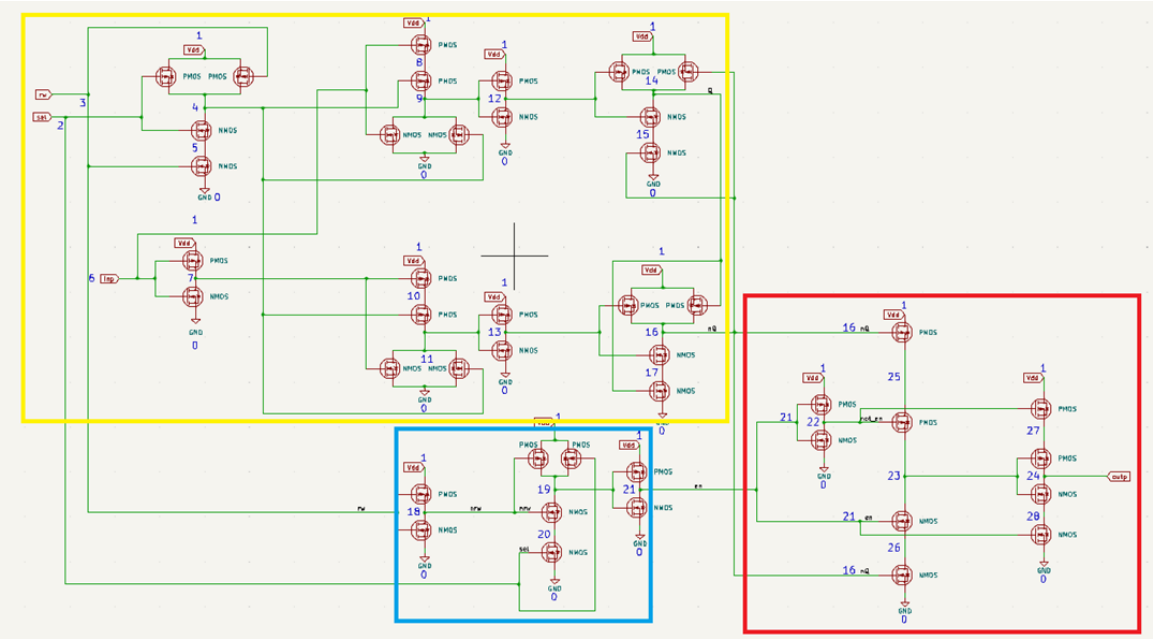


Figure 14: Transistor level design of bitcell.

Choice of design parameters: The design parameters for the transistors and the supply voltage were carefully chosen to balance power efficiency and performance.

For the dimensions of the transistor, we selected values to optimize power consumption. A lower W/L ratio results in a lower drain current, thereby reducing overall power usage. Initially, we estimated $W = 3\mu m$ and $L = 1\mu m$ but later adjusted these to $W = 2\mu m$ and $L = 2\mu m$ after conducting simulations in AIM-Spice. These adjusted values performed well in the bitcell design, so we adopted them for the final implementation. Although further optimization of the W/L ratio was theoretically possible, it would have required additional time for refinements.

For the supply voltage, we chose a value of 1V, as it struck a balance between practicality and performance. This voltage was sufficient to provide an effective gate-to-source voltage, ensuring that the transistors could fully switch to clear signal conduction. In addition, a 1V supply simplified power leakage calculations and made it easier to distinguish between logic high and low signals during simulations. It also provided enough power to counteract parasitic effects and drive the correct values to the desired output nodes. Although using a 1V supply does increase power consumption slightly, it was a practical choice to ensure reliable operation, with room for further optimization if necessary.

To implement this design, the initial step involved simulating the circuit using AIM-Spice to evaluate its performance characteristics, including reaction time and power consumption. This simulation was carried out at all four process corners—Fast-Fast (FF), Slow-Slow (SS), Fast-Slow (FS), and Slow-Fast (SF)—to ensure that the design performs reliably under varying manufacturing conditions. Furthermore, the simulations were repeated at different operating temperatures, specifically -27°C, 20°C, and 50°C, to verify the robustness and stability of the design under a range of environmental conditions. By thoroughly testing across these process corners and temperature variations, we were able to identify and address any potential worst-case scenarios, ensuring the design's reliabil-

ity, consistency, and adaptability in diverse operational contexts. These tests were crucial in refining the circuit and optimizing its performance before proceeding to the next stages of development.

After completing the initial simulations, we translated the logic gate design into Verilog code to facilitate digital implementation and testing. Using Active-HDL, we simulated the Verilog design with a comprehensive testbench to verify its functionality. This simulation ensured that the circuit could correctly take input, process them as expected, and retain the values according to the design requirements. The use of Verilog allowed us to validate the logical behavior of the circuit in a simulated digital environment, providing confidence in its ability to function as intended before moving on to physical implementation.

3.3 FSM

This section details the design and implementation process of the Finite State Machine (FSM), a critical component of the circuit. The FSM controls the operational flow by defining various states and transitions required to manage memory operations. Beginning with the state diagram, the development process is systematically described, highlighting how the logical structure was translated into a functional circuit. In addition, considerations for synchronization, timing, and design trade-offs are discussed, along with the challenges encountered during implementation.

Firstly, we took the given state diagram. This state diagram has four different states. these are write stable idle and read. The given state diagram is a Moore machine. This machine has two input bits: op and select and two output signals valid anf rw.

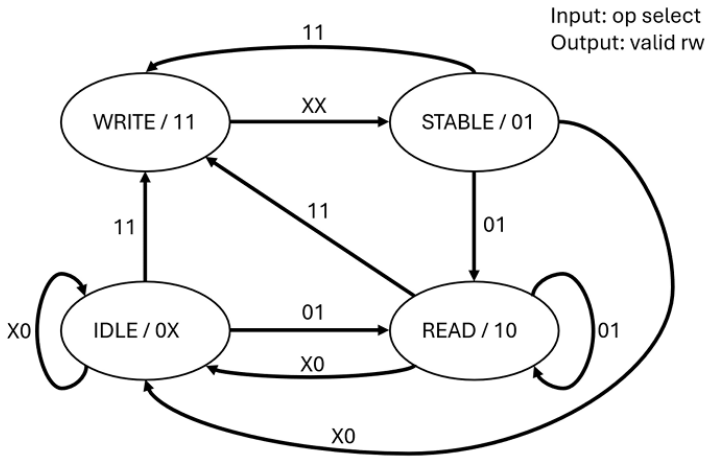


Figure 15: State diagram with four different states

The state diagram shown in Figure 15 is then translated into state table as shown in Table 2. Output of idle state is chosen to be *00*.

Table 2: Truth table for the FSM

Present State		Input		Next State		Output	
state	binary	op	select	state	binary	valid	rw
idle	00	0	0	idle	00	0	0
idle	00	0	1	read	01	0	0
idle	00	1	0	idle	00	0	0
idle	00	1	1	write	11	0	0
read	01	0	0	idle	00	1	0
read	01	0	1	read	01	1	0
read	01	1	0	idle	00	1	0
read	01	1	1	write	11	1	0
stable	10	0	0	idle	00	0	1
stable	10	0	1	read	01	0	1
stable	10	1	0	idle	00	0	1
stable	10	1	1	write	11	0	1
write	11	0	0	stable	10	1	1
write	11	0	1	stable	10	1	1
write	11	1	0	stable	10	1	1
write	11	1	1	stable	10	1	1

Following the standard procedure, the Karnaugh maps were then derived and logic functions made from them as shown in Figure 16.

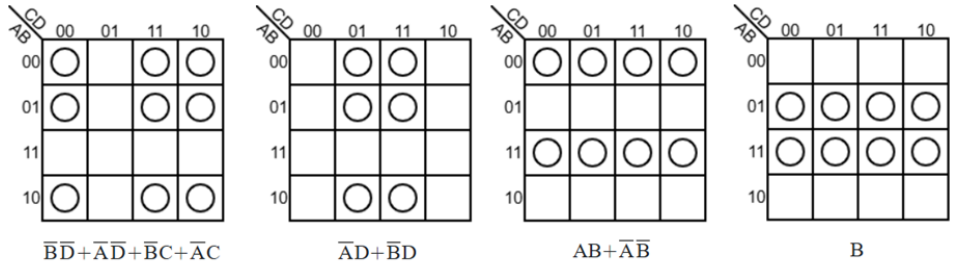


Figure 16: Karnaugh maps for FSM

The FSM design was coded and simulated in ActiveHDL using a testbench. Since the start, the FSM did not function well during simulation, especially for the write state (op=1, select=1). At this stage, the simulation encountered error 1660: delta count overflow, signaling that the FSM had entered a loop state. This occurred because, after a write operation, the FSM failed to transition into a stable state and instead oscillated repeatedly between states due to unchanged inputs.

After several unsuccessful attempts to debug and simulate the design, the FSM was reconstructed in a different way to resolve the problem. The updated implementation incorporated two D flip-flops and additional combinatorial logic derived from the corrected logic functions, as shown in Figure 17. This redesign effectively eliminated the looping issue, ensuring that the FSM transitioned to a stable state after the write operation and responded correctly to subsequent inputs. Moreover, this design required fewer transistors, making it simpler and more efficient.

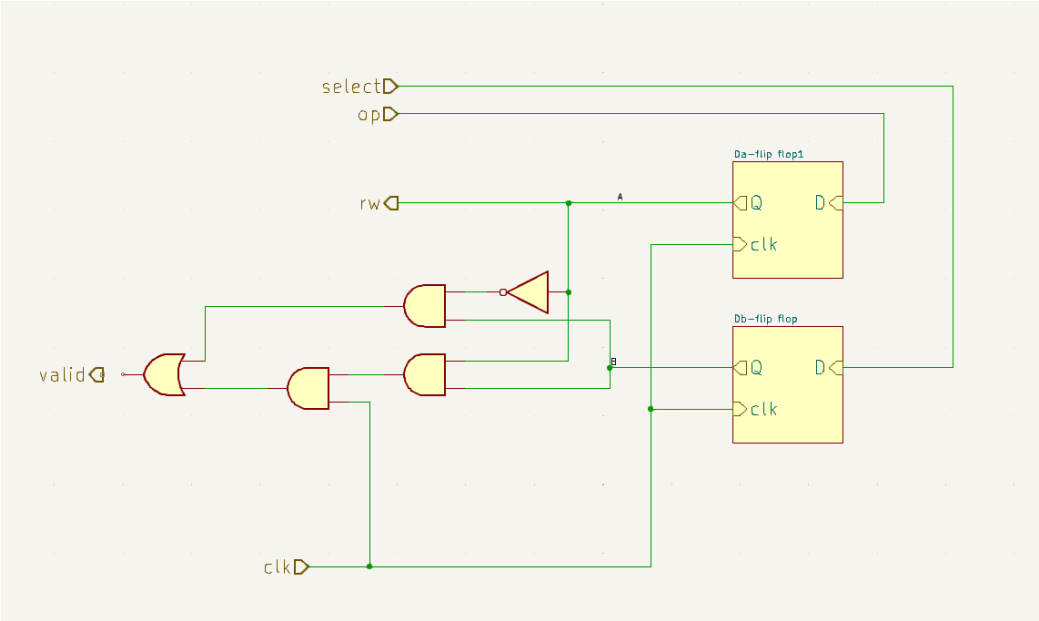


Figure 17: Logic gate-level design of FSM

This design outputs rw as the same value as the input op, which defines the operating mode (read or write). The valid signal is mirroring the select signal on the input but with an exception in case of write mode (op=1, select=1). In this case a valid signal is synchronized with clk signal to alter between write and stable state until inputs are changed. After each write operation is completed, the FSM transitions to a stable state, ready for the next set of inputs.

3.4 The decoder

The decoder plays a critical role in memory design by allowing the selection of specific memory words for read and write operations. This section provides an in-depth explanation of the implementation of the decoder, including its logic design and integration within the overall system. The decoder translates the address inputs into select signals that activate the appropriate memory word. Design considerations are discussed, such as the choice of circuit topology and optimization for speed and power efficiency. In addition, the challenges faced during the design process and the methods used to address them are highlighted. This ensures a comprehensive understanding of the decoder's functionality and its contribution to the system.

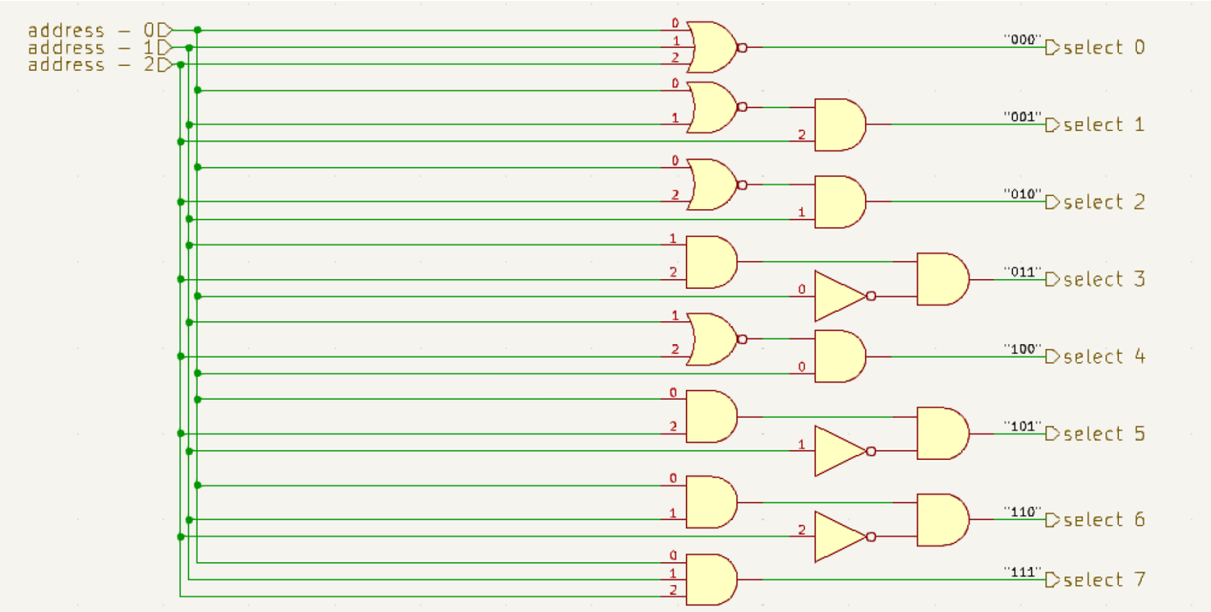


Figure 18: 3-to-8 address Decoder

Figure 18 illustrates our custom-designed address decoder. In this design, we aimed to create something different from the traditional decoders. Although our design requires more logic gates compared to the decoder described in Section 2.3, it performs reliably and accurately selects the memory word based on the given input. This approach provided an opportunity to explore alternative decoding methods while maintaining functionality. The Boolean expressions for all eight inputs in Figure 18 are given as follows:

$$D_0 = \overline{a+b+c}, \quad D_1 = \overline{a+b} \cdot c, \quad D_2 = \overline{a+c} \cdot b, \quad D_3 = \overline{a} \cdot b \cdot c, \quad D_4 = \overline{b+c} \cdot a, \quad D_5 = \overline{b} \cdot a \cdot c, \quad D_6 = \overline{c} \cdot a \cdot b, \quad D_7 = a \cdot b \cdot c$$

Table 3: Truth Table for the Decoder

add0	add1	add2	select0	select1	select2	select3	select4	select5	select6	select7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

The chosen decoder topology comes from the basic principle of signal decoding. For a system with 3 input signals and 8 unique input combinations, the key is to construct 8 distinct logic gate blocks.

Each block is designed to process one specific input combination and generate a corresponding select-line signal. This straightforward approach allows us to achieve the desired decoding functionality efficiently.

Choosing this topology, we considered factors such as ease of implementation, clarity of the logic structure, and maintainability of the design. The simplicity of the idea ensures that the design is easy to debug and modify if necessary. Furthermore, this design ensures accurate word selection, which is crucial for the overall operation of the system. The trade-off of requiring more logic gates is acceptable given the simplicity and reliability achieved with this topology.

4 Simulations and Results

In this section, we present the results obtained from the simulations performed on our design. The primary objective was to verify the functionality and performance of each component of the memory cell, ultimately ensuring that the entire memory cell functions as expected under various input conditions. The simulations were carried out using Aim-Spice and ActiveHDL, and the results are analyzed through result tables, waveforms, and performance graphs, which help validate the correctness and reliability of the design.

4.1 Bitcell Aim-spice simulation:

The gate-level design of the bitcell is shown in Figure 13, Section 3.2. From this setup, the AIM-Spice netlist for the bitcell was written. This process involves a straightforward transition from the gate-level design to the transistor netlist representation.

```
Bitcell - 1 bit memory - tt

vdd 1 0 dc 1
vrw 3 0 dc 0
vsel 2 0 dc 1
vinp 6 0 dc 1
*vinp 6 0 dc 1 pulse(0 1 1n 1p 1p 3n)

*bitcell logic based on SR latch
xmp1 4 2 1 1 pmoslv l=lp w=wp
xmp2 4 3 1 1 pmoslv l=lp w=wp
xmp3 4 2 5 5 nmoslv l=ln w=wn
xmp4 5 3 0 0 nmoslv l=ln w=wn
xmp5 7 6 1 1 pmoslv l=lp w=wp
xmp6 7 6 0 0 nmoslv l=ln w=wn
xmp7 8 6 1 1 pmoslv l=lp w=wp
xmp8 9 4 8 8 pmoslv l=lp w=wp
xmp9 9 6 0 0 nmoslv l=ln w=wn
xmp10 9 4 0 0 nmoslv l=ln w=wn
xmp11 10 7 1 1 pmoslv l=lp w=wp
xmp12 11 4 10 10 pmoslv l=lp w=wp
xmp13 11 7 0 0 nmoslv l=ln w=wn
xmp14 11 4 0 0 nmoslv l=ln w=wn
xmp15 12 9 1 1 pmoslv l=lp w=wp
xmp16 12 9 0 0 nmoslv l=ln w=wn
xmp17 13 11 1 1 pmoslv l=lp w=wp
xmp18 13 11 0 0 nmoslv l=ln w=wn
xmp19 14 12 1 1 pmoslv l=lp w=wp
xmp20 14 16 1 1 pmoslv l=lp w=wp
xmp21 14 12 15 15 nmoslv l=ln w=wn
xmp22 15 16 0 0 nmoslv l=ln w=wn
xmp23 16 13 1 1 pmoslv l=lp w=wp
xmp24 16 14 1 1 pmoslv l=lp w=wp
xmp25 16 13 17 17 nmoslv l=ln w=wn
xmp26 17 14 0 0 nmoslv l=ln w=wn

*nrw signal generation
xmp27 18 3 1 1 pmoslv l=ln w=wn
xmp28 18 3 0 0 nmoslv l=lp w=wp

*enable signal generation for tristate buffer
xmp29 19 18 1 1 pmoslv l=ln w=wn
xmp30 19 2 1 1 pmoslv l=ln w=wn
xmp31 19 18 20 20 nmoslv l=lp w=wp
xmp32 20 2 0 0 nmoslv l=lp w=wp
xmp33 21 19 1 1 pmoslv l=ln w=wn
xmp34 21 19 0 0 nmoslv l=lp w=wp

*tristate buffer
xmp35 22 21 1 1 pmoslv l=ln w=wn
xmp36 22 21 0 0 nmoslv l=lp w=wp
xmp37 25 16 1 1 pmoslv l=ln w=wn
xmp38 23 22 25 25 pmoslv l=ln w=wn
xmp39 23 21 26 26 nmoslv l=lp w=wp
xmp40 26 16 0 0 nmoslv l=lp w=wp
xmp41 24 23 27 27 pmoslv l=lp w=wp
xmp42 24 23 28 28 nmoslv l=lp w=wp
xmp43 27 22 1 1 pmoslv l=lp w=wp
xmp44 28 21 0 0 nmoslv l=lp w=wp

.param ln = 0.2u
.param wp = 0.2u
.param lp = 0.2u
.param wn = 0.2u
.include C:\Users\Adam\Desktop\gpdk90nm_tt.cir
```

Figure 19: Aim-spice netlist of bitcell

At the top of the netlist shown in Figure 19 are written DC voltage sources used to indicate the signal. The names are pretty self-explanatory. Commented source is a source used in upcoming simulations and will be explained further in text. Below in Figure 20 are shown DC operation points with node voltages in situations when rw=0 or rw=1. Taking into account the KiCad bitcell screenshot, it is easy to open it next to the shown operating points and derive the node voltages and their validity. For example, in both cases the output node should be in logic "0" (in case rw=1, sel=1 → en=0; in case rw=1, sel=1, output is ready for reading but there was no input to SR latch in write state, so its output is in logic "0"). This node corresponds to node n.24. On left is case rw=0, on right rw=1.

Bitcell - 1 bit ...		Bitcell - 1 bit ...	
Variables in cir...	Values	Variables in cir...	Values
v(1)	1 V	v(1)	1 V
v(3)	0 V	v(3)	1 V
v(2)	1 V	v(2)	1 V
v(6)	1 V	v(6)	1 V
v(4)	0.999981 V	v(4)	1.93894E-06 V
v(5)	0.947393 V	v(5)	9.6947E-07 V
v(7)	4.84736E-07 V	v(7)	4.84736E-07 V
v(8)	0.961414 V	v(8)	0.0381398 V
v(9)	4.85933E-08 V	v(9)	4.58546E-07 V
v(10)	0.999999 V	v(10)	0.999958 V
v(11)	4.85062E-07 V	v(11)	0.999917 V
v(12)	0.999979 V	v(12)	0.999979 V
v(13)	0.999979 V	v(13)	4.86396E-07 V
v(14)	0.999981 V	v(14)	1.93983E-06 V
v(16)	1.94052E-06 V	v(16)	0.999998 V
v(15)	0.947375 V	v(15)	9.69909E-07 V
v(17)	9.70261E-07 V	v(17)	0.0523466 V
v(18)	0.999979 V	v(18)	4.84736E-07 V
v(19)	1.93977E-06 V	v(19)	0.999979 V
v(20)	9.69876E-07 V	v(20)	9.09177E-06 V
v(21)	0.999979 V	v(21)	4.8515E-07 V
v(22)	4.8515E-07 V	v(22)	0.999979 V
v(25)	0.999981 V	v(25)	0.961463 V
v(23)	0.999962 V	v(23)	0.0017359 V
v(26)	0.947372 V	v(26)	9.71406E-08 V
v(24)	9.70926E-07 V	v(24)	0.0207459 V
v(27)	0.999999 V	v(27)	0.045384 V
v(28)	4.8546E-07 V	v(28)	0.00890449 V

Figure 20: Node voltages corresponding to the nodes in Figure 14

Leakage power: In the assignment was required to determine the leakage power for the TT, SS and FF corners at a temperature of 27 ° C. This was done by turning off all input voltages except V_{DD}. To get the power value, we need to calculate it from voltage and leakage currents. We can

obtain current from DC operating point simulation when we look for the value i(vdd). Current as a function of supply voltage. This value was written in the table for each corner. Calculation is very easy $\rightarrow P = V \cdot I$. Because $V_{DD} = 1\text{V}$, the power leakage is equal to the leakage currents. They are shown in Table 4 below.

Table 4: Leakage Power in Different Corners

Corner	$\mathbf{I_{VDD}}$ **[A]**	$\mathbf{V_{DD}}$ **[V]**	$\mathbf{P_{leakage}}$ **[W]**
FF	2.1134×10^{-5}	1	21.134 μ
SS	7.8168×10^{-9}	1	7.8168 n
TT	1.7291×10^{-8}	1	17.291 n

Write/Read time: To determine read/write, it is necessary to adjust the setup and run a transient simulation. This is supposed to be done for TT, SS and FF corners as well. To get the signal through the entire bitcell in the output, we need to open the whole way. To do this, the bitcell is set in write mode (rw=1, sel=1) and rw is disconnected from the input to the control signal block. To this block is connected DC source with 0V, which secures opened tristate buffer. Now the path is clear for the input signal in the form of pulse. This pulse is defined by the DC source 'vtest' shown in the netlist change. It is a pulse of 1V that is 3ns long. It is sent to the circuit at time 1ns. To depict the delay, transient simulation is run with a duration of 10ns. The voltage at node 24 is plotted as a function of time. The requirement is that the delay be shorter than 3ns. In the figures below are shown the delays for each corner with marked send time and ideal output time.

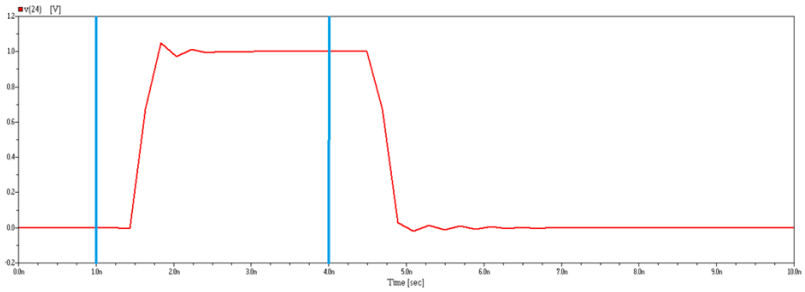


Figure 21: Reaction delay simulation for TT corner

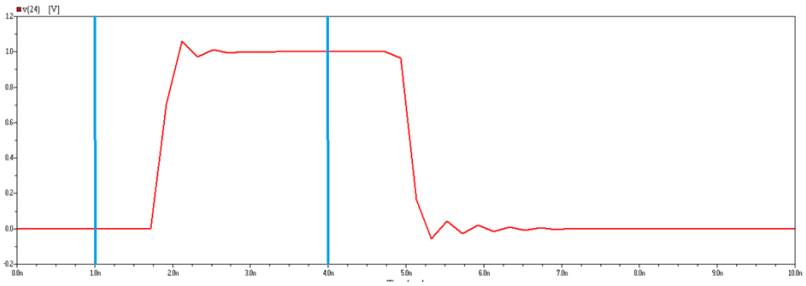


Figure 22: Reaction delay simulation for SS corner

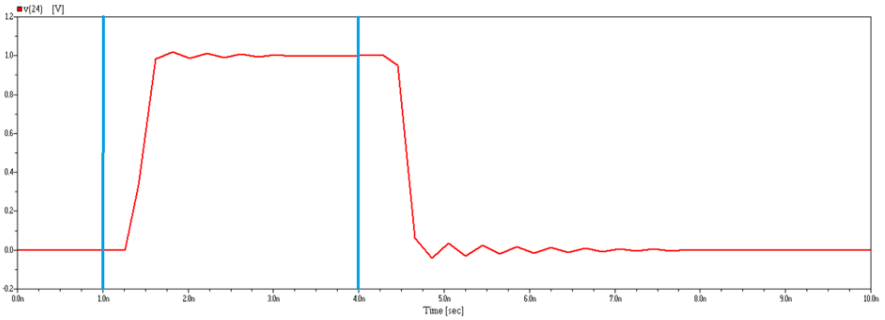


Figure 23: Reaction delay simulation for FF corner

From Figures 21 - 23, it is clear that the delay is slightly larger than 1ns only in the SS corner and that is still more than enough margin before the critical delay is achieved.

Corners and temperatures simulations: The simulation of bitcell is done in different corners at different temperatures. All corner variations were used, TT, SS, FF, FS, and SF. Each corner was simulated under temperatures of -20°C, 27°C and 50°C. The results are in the form of DC operating points. Changes in voltage values can be seen in important nodes. It is important to check and evaluate received parameters, decide what is happening, and adjust the design if needed. In our case, it seems that transistors are behaving variously. In extreme temperatures like -20°C values do not correspond to right functioning as well as in fast corner during high temperatures. These changes can be examined, for example, on output node n.24. There should always be low voltage in this node, but simulations show differently. Therefore, design should be changed or usage conditions should be defined precisely for correct working state. Figures are shown in order TT,SS,FF,FS,SF. In every figure is the lowest temperature leftmost and the highest is rightmost. The voltages are set for read state (rw=0, sel=1).

Bitcell - 1 bit ...		Bitcell - 1 bit ...		Bitcell - 1 bit ...	
Variables in cir...	Values	Variables in cir...	Values	Variables in cir...	Values
v(1)	1 V	v(1)	1 V	v(1)	1 V
v(3)	0 V	v(3)	0 V	v(3)	0 V
v(2)	1 V	v(2)	1 V	v(2)	1 V
v(6)	1 V	v(6)	1 V	v(6)	1 V
v(4)	0.999996 V	v(4)	0.999981 V	v(4)	0.999967 V
v(5)	0.948229 V	v(5)	0.947393 V	v(5)	0.946848 V
v(7)	6.46573E-08 V	v(7)	4.84736E-07 V	v(7)	1.08037E-06 V
v(8)	0.956735 V	v(8)	0.961414 V	v(8)	0.961259 V
v(9)	5.08508E-09 V	v(9)	4.85933E-08 V	v(9)	1.19317E-07 V
v(10)	1 V	v(10)	0.999999 V	v(10)	0.999998 V
v(11)	6.46692E-08 V	v(11)	4.85062E-07 V	v(11)	1.08153E-06 V
v(12)	0.999995 V	v(12)	0.999979 V	v(12)	0.999964 V
v(13)	0.999995 V	v(13)	0.999979 V	v(13)	0.999964 V
v(14)	2.58684E-07 V	v(14)	0.999981 V	v(14)	0.999967 V
v(16)	0.999996 V	v(16)	1.94052E-06 V	v(16)	4.32724E-06 V
v(15)	1.29342E-07 V	v(15)	0.947375 V	v(15)	0.946815 V
v(17)	0.948225 V	v(17)	9.70261E-07 V	v(17)	2.16362E-06 V
v(18)	0.999995 V	v(18)	0.999979 V	v(18)	0.999964 V
v(19)	2.58658E-07 V	v(19)	1.93977E-06 V	v(19)	4.32448E-06 V
v(20)	1.29329E-07 V	v(20)	9.69876E-07 V	v(20)	2.16221E-06 V
v(21)	0.999995 V	v(21)	0.999979 V	v(21)	0.999964 V
v(22)	6.46719E-08 V	v(22)	4.8515E-07 V	v(22)	1.08188E-06 V
v(25)	0.0377639 V	v(25)	0.999981 V	v(25)	0.999967 V
v(23)	1.21282E-07 V	v(23)	0.999962 V	v(23)	0.999933 V
v(26)	6.06412E-08 V	v(26)	0.947372 V	v(26)	0.946811 V
v(24)	0.99999 V	v(24)	9.70926E-07 V	v(24)	2.16603E-06 V
v(27)	0.999995 V	v(27)	0.999999 V	v(27)	0.999998 V
v(28)	1.84063E-06 V	v(28)	4.8546E-07 V	v(28)	1.083E-06 V

Figure 24: Simulation for TT corner under -20°C, 27°C and 50°C

Bitcell - 1 bit ...		Bitcell - 1 bit ...		Bitcell - 1 bit ...	
Variables in cir...	Values	Variables in cir...	Values	Variables in cir...	Values
v(1)	1 V	v(1)	1 V	v(1)	1 V
v(3)	0 V	v(3)	0 V	v(3)	0 V
v(2)	1 V	v(2)	1 V	v(2)	1 V
v(6)	1 V	v(6)	1 V	v(6)	1 V
v(4)	0.999997 V	v(4)	0.999987 V	v(4)	0.999975 V
v(5)	0.949072 V	v(5)	0.948333 V	v(5)	0.947816 V
v(7)	4.24335E-08 V	v(7)	3.27948E-07 V	v(7)	7.5503E-07 V
v(8)	0.938917 V	v(8)	0.960265 V	v(8)	0.96081 V
v(9)	3.41092E-09 V	v(9)	3.22824E-08 V	v(9)	8.18882E-08 V
v(10)	1 V	v(10)	0.999999 V	v(10)	0.999998 V
v(11)	4.24383E-08 V	v(11)	3.28106E-07 V	v(11)	7.55631E-07 V
v(12)	0.999997 V	v(12)	0.999985 V	v(12)	0.999973 V
v(13)	0.999997 V	v(13)	0.999985 V	v(13)	0.999973 V
v(14)	0.999997 V	v(14)	0.999987 V	v(14)	0.432459 V
v(16)	1.69756E-07 V	v(16)	1.31256E-06 V	v(16)	0.432459 V
v(15)	0.949069 V	v(15)	0.94832 V	v(15)	0.36402 V
v(17)	8.48781E-08 V	v(17)	6.56277E-07 V	v(17)	0.36402 V
v(18)	0.999997 V	v(18)	0.999985 V	v(18)	0.999973 V
v(19)	1.69746E-07 V	v(19)	1.31219E-06 V	v(19)	3.02168E-06 V
v(20)	8.48727E-08 V	v(20)	6.56091E-07 V	v(20)	1.51082E-06 V
v(21)	0.999997 V	v(21)	0.999985 V	v(21)	0.999973 V
v(22)	4.24394E-08 V	v(22)	3.28148E-07 V	v(22)	7.55811E-07 V
v(25)	0.999997 V	v(25)	0.999987 V	v(25)	0.588247 V
v(23)	0.999994 V	v(23)	0.999973 V	v(23)	0.340085 V
v(26)	0.949066 V	v(26)	0.948318 V	v(26)	0.286299 V
v(24)	8.48876E-08 V	v(24)	6.56602E-07 V	v(24)	0.888646 V
v(27)	1 V	v(27)	0.999999 V	v(27)	0.962539 V
v(28)	4.24438E-08 V	v(28)	3.28299E-07 V	v(28)	0.0174106 V

Figure 25: Simulation for SS corner under -20°C, 27°C and 50°C

Bitcell - 1 bit ...		Bitcell - 1 bit ...		Bitcell - 1 bit ...	
Variables in cir...	Values	Variables in cir...	Values	Variables in cir...	Values
v(1)	1 V	v(1)	1 V	v(1)	1 V
v(3)	0 V	v(3)	0 V	v(3)	0 V
v(2)	1 V	v(2)	1 V	v(2)	1 V
v(6)	1 V	v(6)	1 V	v(6)	1 V
v(4)	0.999993 V	v(4)	0.999974 V	v(4)	0.999955 V
v(5)	0.947782 V	v(5)	0.946865 V	v(5)	0.946292 V
v(7)	1.00244E-07 V	v(7)	7.01529E-07 V	v(7)	1.51045E-06 V
v(8)	0.960931 V	v(8)	0.962005 V	v(8)	0.961578 V
v(9)	8.11072E-09 V	v(9)	7.25202E-08 V	v(9)	1.71626E-07 V
v(10)	1 V	v(10)	0.999998 V	v(10)	0.999997 V
v(11)	1.00273E-07 V	v(11)	7.02182E-07 V	v(11)	1.51261E-06 V
v(12)	0.999993 V	v(12)	0.999971 V	v(12)	0.99995 V
v(13)	0.999993 V	v(13)	0.999971 V	v(13)	0.99995 V
v(14)	4.01109E-07 V	v(14)	2.80928E-06 V	v(14)	0.999955 V
v(16)	0.999993 V	v(16)	0.999974 V	v(16)	6.05257E-06 V
v(15)	2.00554E-07 V	v(15)	1.40464E-06 V	v(15)	0.946247 V
v(17)	0.947776 V	v(17)	0.946839 V	v(17)	3.02628E-06 V
v(18)	0.999993 V	v(18)	0.999971 V	v(18)	0.99995 V
v(19)	4.01047E-07 V	v(19)	2.80777E-06 V	v(19)	6.04743E-06 V
v(20)	2.00523E-07 V	v(20)	1.40387E-06 V	v(20)	3.02365E-06 V
v(21)	0.999993 V	v(21)	0.999971 V	v(21)	0.99995 V
v(22)	1.00279E-07 V	v(22)	7.02361E-07 V	v(22)	1.51327E-06 V
v(25)	0.0371173 V	v(25)	0.0379855 V	v(25)	0.999955 V
v(23)	1.88439E-07 V	v(23)	1.33081E-06 V	v(23)	0.99991 V
v(26)	9.42193E-08 V	v(26)	6.65405E-07 V	v(26)	0.94624 V
v(24)	0.999985 V	v(24)	0.999942 V	v(24)	3.03077E-06 V
v(27)	0.999993 V	v(27)	0.999971 V	v(27)	0.999997 V
v(28)	2.73502E-06 V	v(28)	1.25311E-05 V	v(28)	1.51536E-06 V

Figure 26: Simulation for FF corner under -20°C, 27°C and 50°C

Bitcell - 1 bit ...		Bitcell - 1 bit ...		Bitcell - 1 bit ...	
Variables in cir...	Values	Variables in cir...	Values	Variables in cir...	Values
v(1)	1 V	v(1)	1 V	v(1)	1 V
v(3)	0 V	v(3)	0 V	v(3)	0 V
v(2)	1 V	v(2)	1 V	v(2)	1 V
v(6)	1 V	v(6)	1 V	v(6)	1 V
v(4)	0.999986 V	v(4)	0.999944 V	v(4)	0.999903 V
v(5)	0.947781 V	v(5)	0.946861 V	v(5)	0.946285 V
v(7)	1.95682E-08 V	v(7)	1.4922E-07 V	v(7)	3.41511E-07 V
v(8)	0.938911 V	v(8)	0.960226 V	v(8)	0.960756 V
v(9)	1.57302E-09 V	v(9)	1.4694E-08 V	v(9)	3.70628E-08 V
v(10)	1 V	v(10)	0.999999 V	v(10)	0.999998 V
v(11)	1.95794E-08 V	v(11)	1.49548E-07 V	v(11)	3.42715E-07 V
v(12)	0.999985 V	v(12)	0.999938 V	v(12)	0.999894 V
v(13)	0.999985 V	v(13)	0.999938 V	v(13)	0.999894 V
v(14)	0.999986 V	v(14)	5.98326E-07 V	v(14)	1.37139E-06 V
v(16)	7.83219E-08 V	v(16)	0.999944 V	v(16)	0.999903 V
v(15)	0.947768 V	v(15)	2.99163E-07 V	v(15)	6.85694E-07 V
v(17)	3.91609E-08 V	v(17)	0.94681 V	v(17)	0.946195 V
v(18)	0.999985 V	v(18)	0.999938 V	v(18)	0.999894 V
v(19)	7.82988E-08 V	v(19)	5.97638E-07 V	v(19)	1.36884E-06 V
v(20)	3.91492E-08 V	v(20)	2.98812E-07 V	v(20)	6.84395E-07 V
v(21)	0.999985 V	v(21)	0.999938 V	v(21)	0.999894 V
v(22)	1.95812E-08 V	v(22)	1.496E-07 V	v(22)	3.42911E-07 V
v(25)	0.999986 V	v(25)	0.0382884 V	v(25)	0.0387263 V
v(23)	0.999973 V	v(23)	2.82715E-07 V	v(23)	6.50223E-07 V
v(26)	0.947767 V	v(26)	1.41357E-07 V	v(26)	3.25111E-07 V
v(24)	3.91823E-08 V	v(24)	0.999876 V	v(24)	0.999788 V
v(27)	1 V	v(27)	0.999938 V	v(27)	0.999894 V
v(28)	1.95911E-08 V	v(28)	1.25294E-05 V	v(28)	2.26974E-05 V

Figure 27: Simulation for FS corner under -20°C, 27°C and 50°C

Bitcell - 1 bit ...		Bitcell - 1 bit ...		Bitcell - 1 bit ...	
Variables in cir...	Values	Variables in cir...	Values	Variables in cir...	Values
v(1)	1 V	v(1)	1 V	v(1)	1 V
v(3)	0 V	v(3)	0 V	v(3)	0 V
v(2)	1 V	v(2)	1 V	v(2)	1 V
v(6)	1 V	v(6)	1 V	v(6)	1 V
v(4)	0.999999 V	v(4)	0.999994 V	v(4)	0.999989 V
v(5)	0.949072 V	v(5)	0.948334 V	v(5)	0.947818 V
v(7)	2.17379E-07 V	v(7)	1.54179E-06 V	v(7)	3.33937E-06 V
v(8)	0.960935 V	v(8)	0.962021 V	v(8)	0.961603 V
v(9)	1.75873E-08 V	v(9)	1.59353E-07 V	v(9)	3.79323E-07 V
v(10)	1 V	v(10)	0.999998 V	v(10)	0.999997 V
v(11)	2.17388E-07 V	v(11)	1.54202E-06 V	v(11)	3.3401E-06 V
v(12)	0.999998 V	v(12)	0.999993 V	v(12)	0.999988 V
v(13)	0.999998 V	v(13)	0.999993 V	v(13)	0.999988 V
v(14)	8.69574E-07 V	v(14)	0.999994 V	v(14)	0.999989 V
v(16)	0.999999 V	v(16)	6.1688E-06 V	v(16)	1.33634E-05 V
v(15)	4.34787E-07 V	v(15)	0.948322 V	v(15)	0.947797 V
v(17)	0.94907 V	v(17)	3.0844E-06 V	v(17)	6.6817E-06 V
v(18)	0.999998 V	v(18)	0.999993 V	v(18)	0.999988 V
v(19)	8.69545E-07 V	v(19)	6.168E-06 V	v(19)	1.33606E-05 V
v(20)	4.34772E-07 V	v(20)	3.08399E-06 V	v(20)	6.68022E-06 V
v(21)	0.999998 V	v(21)	0.999993 V	v(21)	0.999988 V
v(22)	2.17394E-07 V	v(22)	1.54223E-06 V	v(22)	3.34095E-06 V
v(25)	0.0371134 V	v(25)	0.999994 V	v(25)	0.999989 V
v(23)	4.0853E-07 V	v(23)	0.999987 V	v(23)	0.999977 V
v(26)	2.04265E-07 V	v(26)	0.948322 V	v(26)	0.947795 V
v(24)	0.999997 V	v(24)	3.08495E-06 V	v(24)	6.68364E-06 V
v(27)	0.999998 V	v(27)	0.999998 V	v(27)	0.999997 V
v(28)	1.23914E-06 V	v(28)	1.54247E-06 V	v(28)	3.3418E-06 V

Figure 28: Simulation for SF corner under -20°C, 27°C and 50°C

4.2 Bitcell ActiveHDL simulation

This is perhaps the most important section of the whole project. In the figures below, we demonstrate the functionality of our Verilog design. Documentation of every part would be a long process, so we show only the most important parts of the codes and simulations. To find all the details of the whole scheme, codes and simulations of separate components and whole memory are shown in the section Appendices at the bottom of this document. The ideal result and goal of this simulation should aim at writing a whole word in a memory and then reading it out.

Below we show screenshots of the top memory layer, testbench and simulation result. We decided to simulate the writing and reading of a word *"engineer"*, which has exactly 8 letters and therefore uses and tests all the 8 words and 64 bits. To obtain a binary code for the intended word, we used the online conversion tool [3].

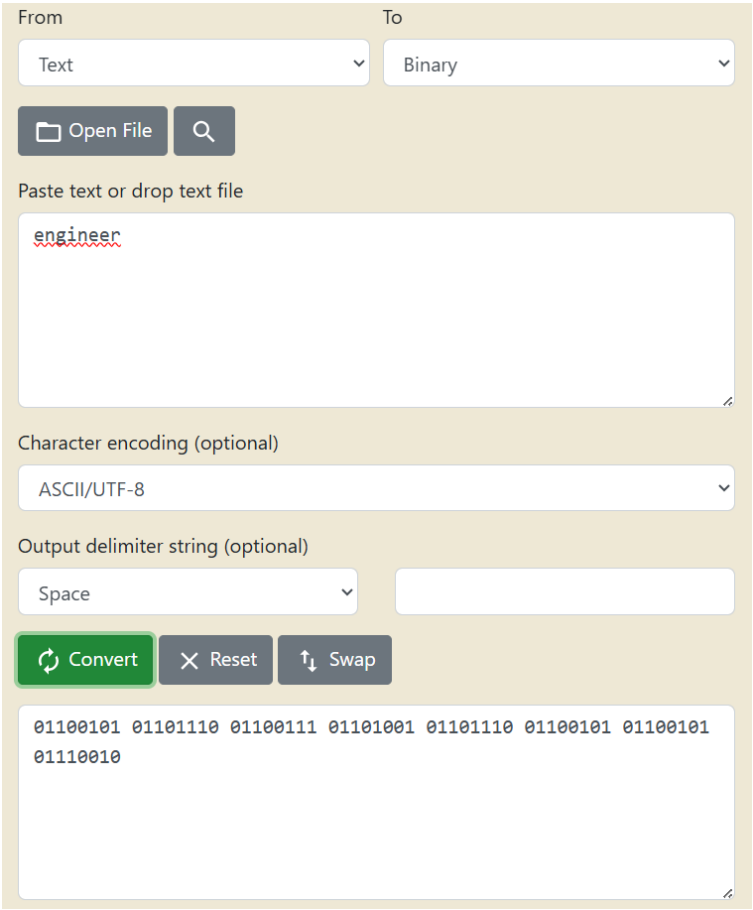


Figure 29: Conversion from written text to binary code

To write in words, we set the memory in the write mode (op=1, select=1). Then we start to write in data, we always input address of the word on address lines, and at the same time the data that should come in on input lines. After writing the first 8 bits, we first change the address of a word and add a little delay before changing the input. This is because we experienced problems with simulation results when both inputs and the address were changed at the same time. If this was done, the information in the previous word changed from previous input to new input. We think that this is not a major problem because in reality, the inputs incoming to the physical circuit would need to defined to first change the address and then the inputs. Otherwise, it would be just the process of overwriting old data to new data. After we have finished with the write operation, we switch to read mode (op=0, select=1) to get information we just wrote. Then we just change addresses as shown in the testbench, and we are getting saved data on the outputs. After verifying the result, we end up with the same binary code as we sent on input.

```
1  `timescale 1ps / 1ps
2  module memory (
3      // Address inputs
4      input adr0, adr1, adr2,
5
6      // Control signals
7       input select, op, clk,
8
9      // Data inputs (8 bits)
10      input i0, i1, i2, i3, i4, i5, i6, i7,
11
12     // Data outputs (8 bits)
13     output o0, o1, o2, o3, o4, o5, o6, o7
14 );
15
16     // Internal signals
17      wire valid, rw;
18      wire sel0, sel1, sel2, sel3, sel4, sel5, sel6, sel7;
19
20     // Concatenate inputs into a single 8-bit bus
21
22     // Logic and module instantiations would go here
23
24
25 fsm3 fsm3(
26     .op(op),
27     .select(select),
28     .clk(clk),
29     .valid(valid),
30     .rw(rw));
31
```

Figure 30: Top layer memory code in Verilog - first part

```
32 decoder decoder(
33     .adr0(adr0),
34     .adr1(adr1),
35     .adr2(adr2),
36     .sel0(sel0),.sel1(sel1),.sel2(sel2),.sel3(sel3),.sel4(sel4),.sel5(sel5),.sel6(sel6),.sel7(sel7)
37     );
38
39 memorycells memorycells(
40         .valid(valid),
41             .rw(rw),
42         .sel0(sel0),.sel1(sel1),.sel2(sel2),.sel3(sel3),.sel4(sel4),.sel5(sel5),.sel6(sel6),.sel7(sel7),
43         .i0(i0), .i1(i1),.i2(i2),.i3(i3),.i4(i4),.i5(i5),.i6(i6), .i7(i7),
44         .o0(o0), .o1(o1), .o2(o2), .o3(o3), .o4(o4), .o5(o5), .o6(o6), .o7(o7)
45         );
46 endmodule
```

Figure 31: Top layer memory code in Verilog - second part

```
module tb_memory;

    reg op,select,clk,i0 ,i1 ,i2 ,i3 ,i4 ,i5 ,i6 ,i7,adr0,adr1,adr2;
    wire o0 ,o1 ,o2 ,o3 ,o4 ,o5 ,o6 ,o7;
    memory uut (
    .clk(clk),
    .op(op),
    .select(select),
    .adr0(adr0),.adr1(adr1),.adr2(adr2),
    .i0(i0),.i1(i1),.i2(i2),.i3(i3),.i4(i4),.i5(i5),.i6(i6),.i7(i7),
    .o0(o0),.o1(o1),.o2(o2),.o3(o3),.o4(o4),.o5(o5),.o6(o6),.o7(o7)
    );

    initial begin
     clk = 0;


//writing of word engineer into memory
     adr0=0; adr1=0; adr2=0; select=1; op=1;
     i0=0; i1=1; i2=1; i3=0; i4=0; i5=1; i6=0; i7=1; #20;
     adr0=0; adr1=0; adr2=1; #10;
     i0=0; i1=1; i2=1; i3=0; i4=1; i5=1; i6=1; i7=0; #20;
     adr0=0; adr1=1; adr2=0; #10;
     i0=0; i1=1; i2=1; i3=0; i4=0; i5=1; i6=1; i7=1; #20;
     adr0=0; adr1=1; adr2=1; #10;
     i0=0; i1=1; i2=1; i3=0; i4=1; i5=0; i6=0; i7=1; #20;
     adr0=1; adr1=0; adr2=0; #10;
     i0=0; i1=1; i2=1; i3=0; i4=1; i5=1; i6=1; i7=0; #20;
     adr0=1; adr1=0; adr2=1; #10;
     i0=0; i1=1; i2=1; i3=0; i4=0; i5=1; i6=0; i7=1; #20;
     adr0=1; adr1=1; adr2=0; #10;
     i0=0; i1=1; i2=1; i3=0; i4=0; i5=1; i6=0; i7=1; #20;
     adr0=1; adr1=1; adr2=1; #10;
     i0=0; i1=1; i2=1; i3=1; i4=0; i5=0; i6=1; i7=0; #20;
```

Figure 32: Testbench code - first part

```
36  //read-out
37       select=1; op=0;
38       adr0=0; adr1=0; adr2=0; #10;
39       adr0=0; adr1=0; adr2=1; #10;
40       adr0=0; adr1=1; adr2=0; #10;
41       adr0=0; adr1=1; adr2=1; #10;
42       adr0=1; adr1=0; adr2=0; #10;
43       adr0=1; adr1=0; adr2=1; #10;
44       adr0=1; adr1=1; adr2=0; #10;
45       adr0=1; adr1=1; adr2=1; #10;
46       select=0; op=0;
47      end
48  //clock generation
49      always begin
50      #5 clk = ~clk;
51      end
52
53  endmodule
```

Figure 33: Testbench code - second part

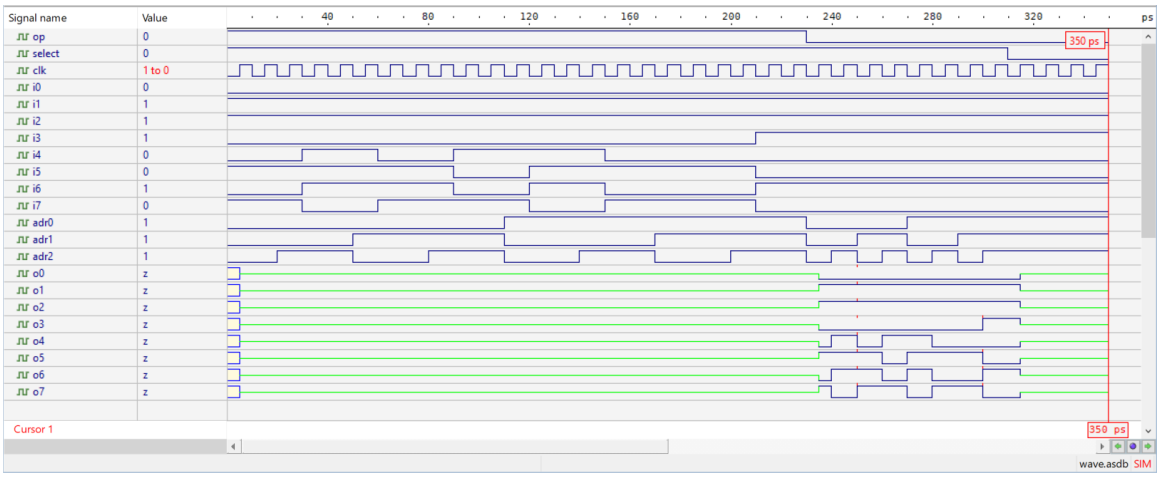


Figure 34: Simulation of the whole memory - write-in and read-out of the word "engineer"

In the figure above, we can see that the output bits are changing with a change of the address input bit, that is a good sign. After verifying the result, we end up with the same binary code as we sent on input.

5 Discussion

Our final result is that we finished a fully working 64 bit memory, but optimizations would be needed. During the simulations in ActiveHDL it was proved that memory is working as it should. The results of AIM-Spice simulations showed that the bitcell is not stable in extreme conditions (e.g., temperature or corner variations). During standard conditions everything works fine, so the design would probably require a slight change of the transistor parameters. But maybe not, that solely depends on conditions in which the sensor with memory would work. We think it is worth mentioning that we did not need much help to create the circuits. Except for the SR latch used as bitcell and D-flipflop, we made up the whole scheme by ourselves. Of course, the Internet helped a lot, but in most cases it was a help related to program functioning (AIM-Spice, ActiveHDL, etc.). It is fair to also mention that the books CMOS VLSI Design, Analog Integrated Circuits [4] and others helped as well. To create snippets of code, we used an overleaf template for listings. [5]

Surprising results are from AIM-Spice simulations when some of the transistors got open when they should not in extreme conditions. The explanation for this occurrence is simply because of transistor type and design. Using a different type or size of transistor would most probably solve the problem. ActiveHDL simulations were a long and exhausting process, but in the end, results turned out as they should. If the circuit would not work properly, we would not expect to have problem in this part of the project. FSM construction is still a mystery to us and we did not figure out how to get proper Moore machine as intended when following the steps in the lecture. However, we think that the FSM we implemented is good enough in simulations. We tried the mentioned FSM creating steps many times, but we do not know where the wrong step is made.

There are definitely some things that would change if more chances were given, but time is limited. The most important thing is that the design works. Changes could be made in Verilog that could be written much more efficiently. Also, transistor optimizations could be done. In the logical design (e.g. FSM, decoder), some changes could be implemented as well to reduce the number of transistors used in design, while maintaining the same logic functions. The transistor design is also related to the power supply and power consumption, so the overall power dissipation could be adjusted. We would be ready to pass the design to colleagues, but we would need to know more about conditions and environmental factors that would be in contact with the sensor. If the conditions are not within the borders of stable working borders, we would need to adjust the physical design of transistors.

6 Conclusion

As was already mentioned, we can conclude that the memory is working as shown in Active-HDL simulations. To give clear image of our design parameters, we created a Table 5 that summarizes the important values. We achieved a working 64-bit memory unit which is able to hold written data and output them as needed. It is controlled by op and selects signals that are responsible for different working states. Address lines are responsible for choosing the correct word, and then we have 8 input lines where data are sent in. If going to read mode, by choosing an address of required word, information is sent to output pins. Our goal aligns with what we have set at the beginning of this course pretty well, providing a fully functional design of the memory unit. Similarly as before, we mention that many things could be adjusted, and some more work would be needed to achieve an optimal design, ready to manufacture without regard to working conditions. The only thing we would like to highlight is the nice looking hierarchical memory scheme in KiCad which gave us a solid

concept of what we want to achieve. There was still a long way to go from the scheme to the Verilog implementation, but the idea was very clear and we knew what we needed to achieve. KiCad is a free software that can create circuit schemes for students to better understand and imagination. It helped us a lot. To conclude this work with final words, we are thankful for this project as it brought us many new, useful skills for the upcoming years and helped us to create a solid base of knowledge in the domain of integrated circuit design. Now we are more prepared to face new challenges in our field of studies and to make progress towards successful careers as electronic engineers.

Table 5: Summarization of important parameters

Parameter	Value
V_{DD}	1V
Transistor width	2 μm
Transistor length	2 μm
Power leakage	17.291nW
Read/write time delay	$\approx$ 1ns

References

[1] Neil H.E. Weste and David Money Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, Boston, MA, 4th edition, 2010.

[2] Anil K. Maini. *Digital Electronics: Principles, Devices and Applications*. Wiley, Chichester, England, 2007.

[3] RapidTables. Text conversion tool. `https://www.rapidtables.com/convert/number/ascii-to-binary.html`, 2024. Accessed: 2024-11-22.

[4] K.W.Martin T.C.Carusone, D.A.Johns. *Analog Integrated Circuit Design*. Wiley, United States of America, 2011.

[5] OverLeaf. Code listing. `https://www.overleaf.com/learn/latex/Code_listing`, 2024. Accessed: 2024-11-22.

Appendices

A AIMSpice Code

```
Bitcell - 1 bit memory - tt

vdd 1 0 dc 1
vrw 3 0 dc 0
vsel 2 0 dc 1
vinp 6 0 dc 1
*vinp 6 0 dc 1 pulse(0 1 1n 1p 1p 3n)

*bitcell logic based on SR latch
xmp1 4 2 1 1 pmos1v l=lp w=wp
xmp2 4 3 1 1 pmos1v l=lp w=wp
xmp3 4 2 5 5 nmos1v l=ln w=wn
xmp4 5 3 0 0 nmos1v l=ln w=wn
xmp5 7 6 1 1 pmos1v l=lp w=wp
xmp6 7 6 0 0 nmos1v l=ln w=wn
xmp7 8 6 1 1 pmos1v l=lp w=wp
xmp8 9 4 8 8 pmos1v l=lp w=wp
xmp9 9 6 0 0 nmos1v l=ln w=wn
xmp10 9 4 0 0 nmos1v l=ln w=wn
xmp11 10 7 1 1 pmos1v l=lp w=wp
xmp12 11 4 10 10 pmos1v l=lp w=wp
xmp13 11 7 0 0 nmos1v l=ln w=wn
xmp14 11 4 0 0 nmos1v l=ln w=wn
xmp15 12 9 1 1 pmos1v l=lp w=wp
xmp16 12 9 0 0 nmos1v l=ln w=wn
xmp17 13 11 1 1 pmos1v l=lp w=wp
xmp18 13 11 0 0 nmos1v l=ln w=wn
xmp19 14 12 1 1 pmos1v l=lp w=wp
xmp20 14 16 1 1 pmos1v l=lp w=wp
xmp21 14 12 15 15 nmos1v l=ln w=wn
xmp22 15 16 0 0 nmos1v l=ln w=wn
xmp23 16 13 1 1 pmos1v l=lp w=wp
xmp24 16 14 1 1 pmos1v l=lp w=wp
xmp25 16 13 17 17 nmos1v l=ln w=wn
xmp26 17 14 0 0 nmos1v l=ln w=wn

*nrw signal generation
xmp27 18 3 1 1 pmos1v l=ln w=wn
xmp28 18 3 0 0 nmos1v l=lp w=wp

*enable signal generation for tristate buffer
xmp29 19 18 1 1 pmos1v l=ln w=wn
xmp30 19 2 1 1 pmos1v l=ln w=wn
xmp31 19 18 20 20 nmos1v l=lp w=wp
xmp32 20 2 0 0 nmos1v l=lp w=wp
xmp33 21 19 1 1 pmos1v l=ln w=wn
xmp34 21 19 0 0 nmos1v l=lp w=wp

*tristate buffer
xmp35 22 21 1 1 pmos1v l=ln w=wn
xmp36 22 21 0 0 nmos1v l=lp w=wp
xmp37 25 16 1 1 pmos1v l=ln w=wn
xmp38 23 22 25 25 pmos1v l=ln w=wn
xmp39 23 21 26 26 nmos1v l=lp w=wp
xmp40 26 16 0 0 nmos1v l=lp w=wp
xmp41 24 23 27 27 pmos1v l=lp w=wp
xmp42 24 23 28 28 nmos1v l=lp w=wp
xmp43 27 22 1 1 pmos1v l=lp w=wp
```

```
xmp44 28 21 0 0 nmos1v l=lp w=wp

.param ln = 0.2u
.param wp = 0.2u
.param lp = 0.2u
.param wn = 0.2u
.include C:\Users\Adam\Desktop\gpdk90nm_tt.cir
```

Listing 1: AIM-Spice bitcell netlist

B Verilog Code

All modules of Verilog codes are below.

```
// Place your code here exactly as intended.
module memory (
    // Address inputs
    input adr0, adr1, adr2,

    // Control signals
    input select, op, clk,

    // Data inputs (8 bits)
    input i0, i1, i2, i3, i4, i5, i6, i7,

    // Data outputs (8 bits)
    output o0, o1, o2, o3, o4, o5, o6, o7
);

    // Internal signals
    wire valid, rw;
    wire sel0, sel1, sel2, sel3, sel4, sel5, sel6, sel7;

    // FSM
    fsm3 fsm3(.op(op), .select(select), .clk(clk), .valid(valid), .rw(rw));

    // Decoder
    decoder decoder(
        .adr0(adr0),
        .adr1(adr1),
        .adr2(adr2),
        .sel0(sel0), .sel1(sel1), .sel2(sel2), .sel3(sel3), .sel4(sel4), .sel5(sel5),
     .sel6(sel6), .sel7(sel7)
    );
    // Memory cells
    memorycells memorycells(
        .valid(valid),
        .rw(rw),
        .sel0(sel0), .sel1(sel1), .sel2(sel2), .sel3(sel3), .sel4(sel4), .sel5(sel5),
     .sel6(sel6), .sel7(sel7),
        .i0(i0), .i1(i1), .i2(i2), .i3(i3), .i4(i4), .i5(i5), .i6(i6), .i7(i7),
        .o0(o0), .o1(o1), .o2(o2), .o3(o3), .o4(o4), .o5(o5), .o6(o6), .o7(o7)
    );
endmodule
```

Listing 2: Memory Module Code

```
`timescale 1ps / 1ps

module fsm3 (
    input  op,
    input  select,
    input  clk,
    output valid,
    output rw
);

    // Internal Wires
    wire nop, s, r, nq, s2, r2, nq2, q2, nsel, nrw, val1, val2, val3;

    // DA Flip-Flop
    not (nop, op);
    nand (s, op, clk);
    nand (r, nop, clk);
    nand (nq, r, rw);
    nand (rw, s, nq);

    // DB Flip-Flop
    not (nsel, select);
    nand (s2, select, clk);
    nand (r2, nsel, clk);
    nand (nq2, r2, q2);
    nand (q2, s2, nq2);

    // Stable State Logic
    not (nrw, rw);
    and (val1, q2, nrw);
    and (val2, rw, q2);
    and (val3, val2, clk);
    or (valid, val3, val1);

endmodule
```

Listing 3: Finite State Machine Module Code

```
module decoder (
    input  adr0, adr1, adr2,         // Input address lines
    output sel0, sel1, sel2, sel3,  // Output select lines
           sel4, sel5, sel6, sel7
);

    // Internal Wires
    wire nadr01, nadr02, nadr12;
    wire adr01, adr02, adr12;
    wire nadr0, nadr1, nadr2;

    // Output Logic
    // sel0
    nor(sel0, adr0, adr1, adr2);

    // sel1
    nor(nadr01, adr0, adr1);
    and(sel1, nadr01, adr2);

    // sel2
    nor(nadr02, adr0, adr2);
    and(sel2, nadr02, adr1);

    // sel3
    and(adr12, adr1, adr2);
    not(nadr0, adr0);
    and(sel3, adr12, nadr0);

    // sel4
    nor(nadr12, adr1, adr2);
    and(sel4, nadr12, adr0);

    // sel5
    and(adr02, adr0, adr2);
    not(nadr1, adr1);
    and(sel5, adr02, nadr1);

    // sel6
    and(adr01, adr0, adr1);
    not(nadr2, adr2);
    and(sel6, adr01, nadr2);

    // sel7
    and(sel7, adr0, adr1, adr2);

endmodule
```

Listing 4: Decoder Module Code

```
`timescale 1ps / 1ps

module memorycells (
    input  [7:0] i,                // 8-bit input data
    input        rw,               // Read/Write signal
    input        valid,            // Valid signal
    input        sel0, sel1, sel2, sel3, sel4, sel5, sel6, sel7, // Select signals
    output [7:0] o                 // 8-bit output for selected word
);
    // Internal signals for word selection
    wire en0, en1, en2, en3, en4, en5, en6, en7;
    // Generate enable signals
    and(en0, valid, sel0);
    and(en1, valid, sel1);
    and(en2, valid, sel2);
    and(en3, valid, sel3);
    and(en4, valid, sel4);
    and(en5, valid, sel5);
    and(en6, valid, sel6);
    and(en7, valid, sel7);

    // Instantiate word modules
    word0 word_inst0 (.i(i), .o(o), .sel(en0), .rw(rw));
    word0 word_inst1 (.i(i), .o(o), .sel(en1), .rw(rw));
    word0 word_inst2 (.i(i), .o(o), .sel(en2), .rw(rw));
    word0 word_inst3 (.i(i), .o(o), .sel(en3), .rw(rw));
    word0 word_inst4 (.i(i), .o(o), .sel(en4), .rw(rw));
    word0 word_inst5 (.i(i), .o(o), .sel(en5), .rw(rw));
    word0 word_inst6 (.i(i), .o(o), .sel(en6), .rw(rw));
    word0 word_inst7 (.i(i), .o(o), .sel(en7), .rw(rw));

endmodule
```

Listing 5: Memory Cells Module Code

```
module word0 (
    input  [7:0] i,    // 8-bit input
    output [7:0] o,    // 8-bit output
    input        sel,  // Select signal
    input        rw    // Read/Write signal
);
    // Generate block to instantiate 8 bitcells
    genvar j;
    generate
        for (j = 0; j < 8; j = j + 1) begin : bitcell_gen
            bitcell0 bc (
                .sel(sel),
                .rw(rw),
                .inp(i[7-j]),
                .outp(o[7-j])
            );
        end
    endgenerate
endmodule
```

Listing 6: 8-bit Word Module Using Generate Loop

```
module bitcell0 (
    input  sel,     // Select signal
    input  rw,      // Read/Write signal
    input  inp,     // Input data
    output outp     // Output data
);
    // Internal signal declarations
    wire w2, r, s, ninp, q, nq, nrw, en;

    // Latch logic
    not (ninp, inp);
    nand (w2, sel, rw);
    or (r, w2, inp);
    or (s, w2, ninp);
    nand (q, r, nq);
    nand (nq, s, q);

    // Tristate logic
    not (nrw, rw);
    and (en, nrw, sel);

    // Built-in tristate buffer
    tristate4 tristate_inst (
        .nq(nq),
        .en(en),
        .outp(outp)
    );
endmodule

// Tristate buffer module
module tristate4 (
    input  nq,     // Input signal
    input  en,     // Enable signal
    output outp    // Output signal
);
    tri outp;      // Tristate signal declaration
    bufif1 b1(outp, nq, en); // Buffered tristate logic
endmodule
```

Listing 7: Single Bitcell Module with Tristate Logic

```
module tb_memory;
    // Testbench signals
    reg op, select, clk;
    reg i0, i1, i2, i3, i4, i5, i6, i7;
    reg adr0, adr1, adr2;
    wire o0, o1, o2, o3, o4, o5, o6, o7;
    // Instantiate the memory module
    memory uut (
        .clk(clk),
        .op(op),
        .select(select),
        .adr0(adr0), .adr1(adr1), .adr2(adr2),
        .i0(i0), .i1(i1), .i2(i2), .i3(i3),
        .i4(i4), .i5(i5), .i6(i6), .i7(i7),
        .o0(o0), .o1(o1), .o2(o2), .o3(o3),
        .o4(o4), .o5(o5), .o6(o6), .o7(o7)
    );
    // Initialization block
    initial begin
        clk = 0;

        // Writing "ENGINEER" into memory
        adr0 = 0; adr1 = 0; adr2 = 0; select = 1; op = 1;
        i0 = 0; i1 = 1; i2 = 1; i3 = 0; i4 = 0; i5 = 1; i6 = 0; i7 = 1; #20;
        adr0 = 0; adr1 = 0; adr2 = 1; #10;
        i0 = 0; i1 = 1; i2 = 1; i3 = 0; i4 = 1; i5 = 1; i6 = 1; i7 = 0; #20;
        adr0 = 0; adr1 = 1; adr2 = 0; #10;
        i0 = 0; i1 = 1; i2 = 1; i3 = 0; i4 = 0; i5 = 1; i6 = 1; i7 = 1; #20;
        adr0 = 0; adr1 = 1; adr2 = 1; #10;
        i0 = 0; i1 = 1; i2 = 1; i3 = 0; i4 = 1; i5 = 0; i6 = 0; i7 = 1; #20;
        adr0 = 1; adr1 = 0; adr2 = 0; #10;
        i0 = 0; i1 = 1; i2 = 1; i3 = 0; i4 = 1; i5 = 1; i6 = 1; i7 = 0; #20;
        adr0 = 1; adr1 = 0; adr2 = 1; #10;
        i0 = 0; i1 = 1; i2 = 1; i3 = 0; i4 = 0; i5 = 1; i6 = 0; i7 = 1; #20;
        adr0 = 1; adr1 = 1; adr2 = 0; #10;
        i0 = 0; i1 = 1; i2 = 1; i3 = 0; i4 = 0; i5 = 1; i6 = 0; i7 = 1; #20;
        adr0 = 1; adr1 = 1; adr2 = 1; #10;
        i0 = 0; i1 = 1; i2 = 1; i3 = 1; i4 = 0; i5 = 0; i6 = 1; i7 = 0; #20;

        // Read-out process
        select = 1; op = 0;
        adr0 = 0; adr1 = 0; adr2 = 0; #10;
        adr0 = 0; adr1 = 0; adr2 = 1; #10;
        adr0 = 0; adr1 = 1; adr2 = 0; #10;
        adr0 = 0; adr1 = 1; adr2 = 1; #10;
        adr0 = 1; adr1 = 0; adr2 = 0; #10;
        adr0 = 1; adr1 = 0; adr2 = 1; #10;
        adr0 = 1; adr1 = 1; adr2 = 0; #10;
        adr0 = 1; adr1 = 1; adr2 = 1; #10;
        select = 0; op = 0;
    end
    // Clock generation
    always begin
        #5 clk = ~clk;
    end
endmodule
```

Listing 8: Testbench for Memory Module

C Optional - Additional screenshots of the Verilog and AIM-Spice setups

In this section we add supportive materials that help to understand the whole topology of the Verilog code memory design. We show different setups and simulations of separate modules like fsm, decoder or bitcell. There is always shown code, testbench and simulation waveform for each component to see its functionality. Additional screenshots of AIM-Spice code are added to show setups for described simulations.

C.1 AIM-Spice additional screenshots

```
Bitcell - 1 bit memory - tt

vdd 1 0 dc 1
vrw 3 0 dc 0
vsel 2 0 dc 1
vinp 6 0 dc 1
*vinp 6 0 dc 1 pulse(0 1 1n 1p 1p 3n)

*bitcell logic based on SR latch
xmp1 4 2 1 1 pmos1v l=1p w=wp
xmp2 4 3 1 1 pmos1v l=1p w=wp
xmp3 4 2 5 5 nmos1v l=1n w=wn
xmp4 5 3 0 0 nmos1v l=1n w=wn
xmp5 7 6 1 1 pmos1v l=1p w=wp
xmp6 7 6 0 0 nmos1v l=1n w=wn
xmp7 8 6 1 1 pmos1v l=1p w=wp
xmp8 9 4 8 8 pmos1v l=1p w=wp
xmp9 9 6 0 0 nmos1v l=1n w=wn
xmp10 9 4 0 0 nmos1v l=1n w=wn
xmp11 10 7 1 1 pmos1v l=1p w=wp
xmp12 11 4 10 10 pmos1v l=1p w=wp
xmp13 11 7 0 0 nmos1v l=1n w=wn
xmp14 11 4 0 0 nmos1v l=1n w=wn
xmp15 12 9 1 1 pmos1v l=1p w=wp
xmp16 12 9 0 0 nmos1v l=1n w=wn
xmp17 13 11 1 1 pmos1v l=1p w=wp
xmp18 13 11 0 0 nmos1v l=1n w=wn
xmp19 14 12 1 1 pmos1v l=1p w=wp
xmp20 14 16 1 1 pmos1v l=1p w=wp
xmp21 14 12 15 15 nmos1v l=1n w=wn
xmp22 15 16 0 0 nmos1v l=1n w=wn
xmp23 16 13 1 1 pmos1v l=1p w=wp
xmp24 16 14 1 1 pmos1v l=1p w=wp
xmp25 16 13 17 17 nmos1v l=1n w=wn
xmp26 17 14 0 0 nmos1v l=1n w=wn

*nrw signal generation
xmp27 18 3 1 1 pmos1v l=1n w=wn
xmp28 18 3 0 0 nmos1v l=1p w=wp

*enable signal generation for tristate buffer
xmp29 19 18 1 1 pmos1v l=1n w=wn
xmp30 19 2 1 1 pmos1v l=1n w=wn
xmp31 19 18 20 20 nmos1v l=1p w=wp
xmp32 20 2 0 0 nmos1v l=1p w=wp
xmp33 21 19 1 1 pmos1v l=1n w=wn
xmp34 21 19 0 0 nmos1v l=1p w=wp

*tristate buffer
xmp35 22 21 1 1 pmos1v l=1n w=wn
xmp36 22 21 0 0 nmos1v l=1p w=wp
xmp37 25 16 1 1 pmos1v l=1n w=wn
xmp38 23 22 25 25 pmos1v l=1n w=wn
xmp39 23 21 26 26 nmos1v l=1p w=wp
xmp40 26 16 0 0 nmos1v l=1p w=wp
xmp41 24 23 27 27 pmos1v l=1p w=wp
xmp42 24 23 28 28 nmos1v l=1p w=wp
xmp43 27 22 1 1 pmos1v l=1p w=wp
xmp44 28 21 0 0 nmos1v l=1p w=wp

.param ln = 0.2u
.param wp = 0.2u
.param lp = 0.2u
.param wn = 0.2u
.include C:\Users\Adam\Desktop\gpdk90nm_tt.cir
```

Figure 35: Aim-spice netlist of bitcell

```
vdd 1 0 dc 1
vrw 3 0 dc 1
vsel 2 0 dc 1
*vinp 6 0 dc 1
vinp 6 0 dc 1 pulse(0 1 1n 1p 1p 3n)
vtest 30 0 dc 0
```

Figure 36: DC source setup for delay simulation

```
*nrw signal generation - vtest as input
xmp27 18 30 1 1 pmos1v l=ln w=wn
xmp28 18 30 0 0 nmos1v l=lp w=wp
```

Figure 37: Code setup change for delay simulation

```
vdd 1 0 dc 1
vrw 3 0 dc 0
vsel 2 0 dc 0
vinp 6 0 dc 0
*vinp 6 0 dc 1 pulse(0 1 1n 1p 1p 3n)
```

Figure 38: DC source setup change for leakage simulation

C.2 Verilog additional screenshots

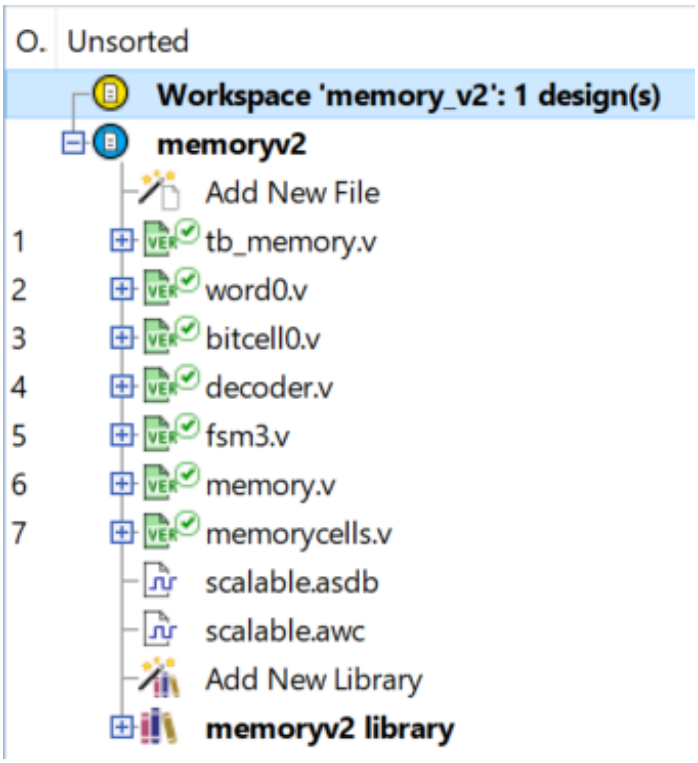


Figure 39: Summary tree of all used files for memory

```
1  `timescale 1ps / 1ps
2  module memory (
3      // Address inputs
4      input adr0, adr1, adr2,
5
6      // Control signals
7       input select, op, clk,
8
9      // Data inputs (8 bits)
10      input i0, i1, i2, i3, i4, i5, i6, i7,
11
12     // Data outputs (8 bits)
13     output o0, o1, o2, o3, o4, o5, o6, o7
14 );
15
16     // Internal signals
17      wire valid, rw;
18      wire sel0, sel1, sel2, sel3, sel4, sel5, sel6, sel7;
19
20     // Concatenate inputs into a single 8-bit bus
21
22     // Logic and module instantiations would go here
23
24
25 fsm3 fsm3(
26     .op(op),
27     .select(select),
28     .clk(clk),
29     .valid(valid),
30     .rw(rw));
31
```

Figure 40: Top layer memory code in Verilog - first part

```
32 decoder decoder(
33     .adr0(adr0),
34     .adr1(adr1),
35     .adr2(adr2),
36     .sel0(sel0),.sel1(sel1),.sel2(sel2),.sel3(sel3),.sel4(sel4),.sel5(sel5),.sel6(sel6),.sel7(sel7)
37     );
38
39 memorycells memorycells(
40         .valid(valid),
41              .rw(rw),
42         .sel0(sel0),.sel1(sel1),.sel2(sel2),.sel3(sel3),.sel4(sel4),.sel5(sel5),.sel6(sel6),.sel7(sel7),
43         .i0(i0), .i1(i1),.i2(i2),.i3(i3),.i4(i4),.i5(i5),.i6(i6), .i7(i7),
44         .o0(o0), .o1(o1), .o2(o2), .o3(o3), .o4(o4), .o5(o5), .o6(o6), .o7(o7)
45         );
46 endmodule
```

Figure 41: Top layer memory code in Verilog - second part

```
1  `timescale 1ps / 1ps
2
3  module fsm3 ( op ,select ,clk ,valid ,rw );
4
5  input op;
6  wire op;
7  input select;
8  wire select;
9  input clk;
10 wire clk;
11 output valid;
12 wire valid;
13 output rw;
14 wire rw;
15
16 //wires
17 wire nop, s, r, nq, s2, r2, nq2, q2, nsel,nrw,val1,val2,val3;
18
19 //}} End of automatically maintained section
20 //da flipflop
21 not(nop,op);
22 nand(s,op,clk);
23 nand(r,nop,clk);
24 nand(nq,r,rw);
25 nand(rw,s,nq);
26
27 //db flipflop
28 not(nsel, select);
29 nand(s2, select, clk);
30 nand(r2, nsel, clk);
31 nand(nq2, r2, q2);
32 nand(q2, s2, nq2);
33
34 //stable state
35 not(nrw,rw);
36 and(val1,q2,nrw);
37 and(val2,rw,q2);
38 and(val3,val2,clk);
39 or(valid,val3,val1);
40
41 endmodule
```

Figure 42: Finite state machine code

```
27  module decoder ( adr0 ,adr1 ,adr2 ,sel0 ,sel1 ,sel2 ,sel3 ,sel4 ,sel5 ,sel6 ,sel7 );
28
29  input adr0;
30  wire adr0;
31  input adr1;
32  wire adr1;
33  input adr2;
34  wire adr2;
35  output sel0;
36  wire sel0;
37  output sel1;
38  wire sel1;
39  output sel2;
40  wire sel2;
41  output sel3;
42  wire sel3;
43  output sel4;
44  wire sel4;
45  output sel5;
46  wire sel5;
47  output sel6;
48  wire sel6;
49  output sel7;
50  wire sel7;
51
52  //wires
53  wire nadr01;
54  wire adr01;
55  wire nadr02;
56  wire adr02;
57  wire adr12;
58  wire nadr0;
59  wire nadr12;
60  wire nadr1;
61  wire nadr2;
```

Figure 43: Decoder code - part one

```
63  //sel0
64  nor(sel0,adr0,adr1,adr2);
65
66  //sel1
67  nor(nadr01,adr0,adr1);
68  and(sel1,nadr01,adr2);
69
70  //sel2
71  nor(nadr02,adr0,adr2);
72  and(sel2,nadr02,adr1);
73
74  //sel3
75  and(adr12,adr1,adr2);
76  not(nadr0,adr0);
77  and(sel3,adr12,nadr0);
78
79  //sel4
80  nor(nadr12,adr1,adr2);
81  and(sel4,nadr12,adr0);
82
83  //sel5
84  and(adr02,adr0,adr2);
85  not(nadr1,adr1);
86  and(sel5,adr02,nadr1);
87
88  //sel6
89  and(adr01,adr0,adr1);
90  not(nadr2,adr2);
91  and(sel6,adr01,nadr2);
92
93  //sel7
94  and(sel7,adr0,adr1,adr2);
95
96  endmodule
```

Figure 44: Decoder code - second part

```
    `timescale 1ps / 1ps
module memorycells (
    input i0, i1, i2, i3, i4, i5, i6, i7,     // 8-bit input data
    input rw,             // Read/Write signal
    input valid,          // Valid signal
    input sel0, sel1, sel2, sel3, sel4, sel5, sel6, sel7, // Select signals for each word
    output o0, o1, o2, o3, o4, o5, o6, o7        // 8-bit output for selected word
);
    // Internal signals for word selection
    wire en0, en1, en2, en3, en4, en5, en6, en7;

    // Generate select signals
    and(en0, valid, sel0);
    and(en1, valid, sel1);
    and(en2, valid, sel2);
    and(en3, valid, sel3);
    and(en4, valid, sel4);
    and(en5, valid, sel5);
    and(en6, valid, sel6);
    and(en7, valid, sel7);

    // Instantiate word0 modules for each word
    word0 word_inst0 (.i({i0, i1, i2, i3, i4, i5, i6, i7}), .o({o0, o1, o2, o3, o4, o5, o6, o7}), .sel(en0), .rw(rw));
    word0 word_inst1 (.i({i0, i1, i2, i3, i4, i5, i6, i7}), .o({o0, o1, o2, o3, o4, o5, o6, o7}), .sel(en1), .rw(rw));
    word0 word_inst2 (.i({i0, i1, i2, i3, i4, i5, i6, i7}), .o({o0, o1, o2, o3, o4, o5, o6, o7}), .sel(en2), .rw(rw));
    word0 word_inst3 (.i({i0, i1, i2, i3, i4, i5, i6, i7}), .o({o0, o1, o2, o3, o4, o5, o6, o7}), .sel(en3), .rw(rw));
    word0 word_inst4 (.i({i0, i1, i2, i3, i4, i5, i6, i7}), .o({o0, o1, o2, o3, o4, o5, o6, o7}), .sel(en4), .rw(rw));
    word0 word_inst5 (.i({i0, i1, i2, i3, i4, i5, i6, i7}), .o({o0, o1, o2, o3, o4, o5, o6, o7}), .sel(en5), .rw(rw));
    word0 word_inst6 (.i({i0, i1, i2, i3, i4, i5, i6, i7}), .o({o0, o1, o2, o3, o4, o5, o6, o7}), .sel(en6), .rw(rw));
    word0 word_inst7 (.i({i0, i1, i2, i3, i4, i5, i6, i7}), .o({o0, o1, o2, o3, o4, o5, o6, o7}), .sel(en7), .rw(rw));

endmodule
```

Figure 45: Word creation code - this code corresponds to the Figure 10

```
// 8-bit word using generate loop
`timescale 1ps / 1ps
module word0 (
    input [7:0] i,   // 8-bit input
    output [7:0] o, // 8-bit output
    input sel,           // Select signal
    input rw             // Read/Write signal
);

    // Generate block to instantiate 8 bitcells
    genvar j;
    generate
        for (j = 0; j < 8; j = j + 1) begin : bitcell_gen
            bitcell0 bc (
                .sel(sel),
                .rw(rw),
                .inp(i[7-j]),
                .outp(o[7-j])
            );
        end
    endgenerate

endmodule
```

Figure 46: Code for creating 8 bitcells in one word

```
1  `timescale 1ps / 1ps
2
3  module bitcell0 (
4      input sel,      // Select signal
5      input rw,       // Read/Write signal
6      input inp,      // Input data
7      output outp     // Output data
8  );
9      wire w2, r, s, ninp, q, nq, nrw, en;
10
11     not(ninp, inp);
12     nand(w2, sel, rw);
13     or(r, w2, inp);
14     or(s, w2, ninp);
15     nand(q, r, nq);
16     nand(nq, s, q);
17
18     // Tristate logic
19     not(nrw, rw);
20     and(en, nrw, sel);
21
22     // Built-in tristate buffer
23     tristate4 tristate4(
24                    .nq(nq),
25                    .en(en),
26                    .outp(outp));
27 endmodule
28
29
30 module tristate4 (nq, en, outp);
31     input   nq, en;
32     output  outp;
33     tri     outp;
34     bufif1  b1(outp, nq, en);
35 endmodule
```

Figure 47: Design of the bitcell in Verilog

```
module tb_memory;

    reg op,select,clk,i0 ,i1 ,i2 ,i3 ,i4 ,i5 ,i6 ,i7,adr0,adr1,adr2;
    wire o0 ,o1 ,o2 ,o3 ,o4 ,o5 ,o6 ,o7;
    memory uut (
    .clk(clk),
    .op(op),
    .select(select),
    .adr0(adr0),.adr1(adr1),.adr2(adr2),
    .i0(i0),.i1(i1),.i2(i2),.i3(i3),.i4(i4),.i5(i5),.i6(i6),.i7(i7),
    .o0(o0),.o1(o1),.o2(o2),.o3(o3),.o4(o4),.o5(o5),.o6(o6),.o7(o7)
    );

    initial begin
     clk = 0;


//writing of word engineer into memory
     adr0=0; adr1=0; adr2=0; select=1; op=1;
     i0=0; i1=1; i2=1; i3=0; i4=0; i5=1; i6=0; i7=1; #20;
     adr0=0; adr1=0; adr2=1; #10;
     i0=0; i1=1; i2=1; i3=0; i4=1; i5=1; i6=1; i7=0; #20;
     adr0=0; adr1=1; adr2=0; #10;
     i0=0; i1=1; i2=1; i3=0; i4=0; i5=1; i6=1; i7=1; #20;
     adr0=0; adr1=1; adr2=1; #10;
     i0=0; i1=1; i2=1; i3=0; i4=1; i5=0; i6=0; i7=1; #20;
     adr0=1; adr1=0; adr2=0; #10;
     i0=0; i1=1; i2=1; i3=0; i4=1; i5=1; i6=1; i7=0; #20;
     adr0=1; adr1=0; adr2=1; #10;
     i0=0; i1=1; i2=1; i3=0; i4=0; i5=1; i6=0; i7=1; #20;
     adr0=1; adr1=1; adr2=0; #10;
     i0=0; i1=1; i2=1; i3=0; i4=0; i5=1; i6=0; i7=1; #20;
     adr0=1; adr1=1; adr2=1; #10;
     i0=0; i1=1; i2=1; i3=1; i4=0; i5=0; i6=1; i7=0; #20;
```

Figure 48: Testbench code - first part

```
36  //read-out
37       select=1; op=0;
38       adr0=0; adr1=0; adr2=0; #10;
39       adr0=0; adr1=0; adr2=1; #10;
40       adr0=0; adr1=1; adr2=0; #10;
41       adr0=0; adr1=1; adr2=1; #10;
42       adr0=1; adr1=0; adr2=0; #10;
43       adr0=1; adr1=0; adr2=1; #10;
44       adr0=1; adr1=1; adr2=0; #10;
45       adr0=1; adr1=1; adr2=1; #10;
46       select=0; op=0;
47      end
48  //clock generation
49      always begin
50      #5 clk = ~clk;
51      end
52
53  endmodule
```

Figure 49: Testbench code - second part

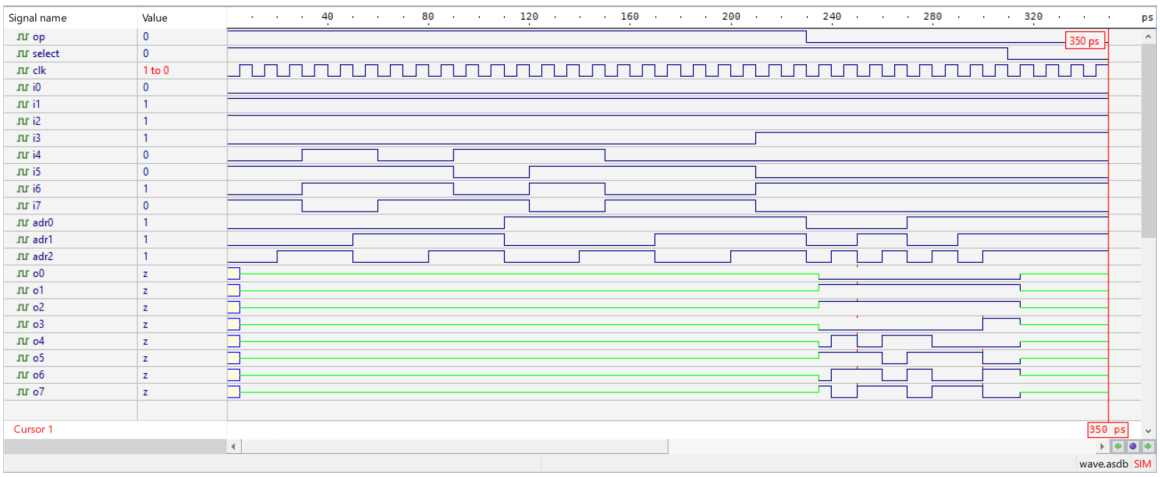


Figure 50: Simulation of the whole memory - write-in and read-out of the word "engineer"

```
3  module testbench;
4
5      reg inp, rw, sel;
6      wire outp;
7
8      bitcell uut (
9      .inp(inp),
10     .rw(rw),
11     .sel(sel),
12     .outp(outp));
13
14     initial begin
15     $monitor("Time=%0d inp=%b rw=%b, sel=%b, outp=%b", $time, inp,rw,sel,outp);
16
17     inp = 0; rw=0; sel=0; #10;
18     inp = 0; rw=0; sel=1; #10;
19     inp = 0; rw=1; sel=0; #10;
20     inp = 0; rw=1; sel=1; #10;
21     inp = 1; rw=0; sel=0; #10;
22     inp = 1; rw=0; sel=1; #10;
23     inp = 1; rw=1; sel=0; #10;
24     inp = 1; rw=1; sel=1; #10;
25     inp = 1; rw=0; sel=1; #10;
26     inp = 0; rw=1; sel=1; #10;
27     inp = 1; rw=0; sel=1; #10;
28     $finish;
29     end
30 endmodule
```

Figure 51: Testbench code - bitcell

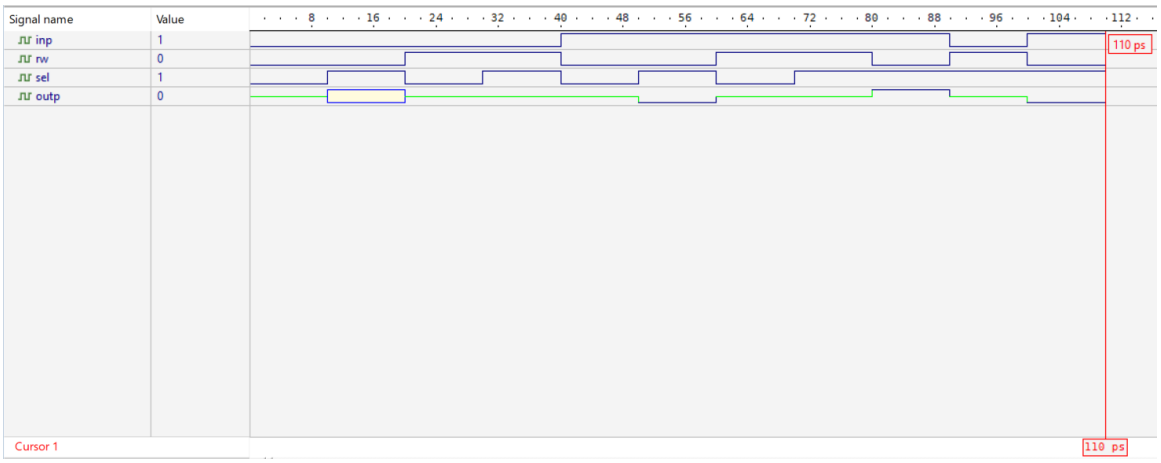


Figure 52: Simulation - bitcell

```
module testbench;

    reg adr0,adr1,adr2;
    wire sel0,sel1,sel2,sel3,sel4,sel5,sel6,sel7;

    decoder uut (
    .sel0(sel0),
    .sel1(sel1),
    .sel2(sel2),
    .sel3(sel3),
    .sel4(sel4),
    .sel5(sel5),
    .sel6(sel6),
    .sel7(sel7),
    .adr0(adr0),
    .adr1(adr1),
    .adr2(adr2));

    initial begin
$monitor("Time=%0d sel0=%b, sel1=%b, sel2=%b, sel3=%b, sel4=%b, sel5=%b, sel6=%b, sel7=%b, adr0=%b,adr1=%b,adr2=%b", $time,sel0,sel1,sel2,sel3,sel4,sel5,sel6,sel7,adr0,adr1,adr2);

    adr0 = 0; adr1=0; adr2=0; #10;
    adr0 = 0; adr1=0; adr2=1; #10;
    adr0 = 0; adr1=1; adr2=0; #10;
    adr0 = 0; adr1=1; adr2=1; #10;
    adr0 = 1; adr1=0; adr2=0; #10;
    adr0 = 1; adr1=0; adr2=1; #10;
    adr0 = 1; adr1=1; adr2=0; #10;
    adr0 = 1; adr1=1; adr2=1; #10;


    end
endmodule
```

Figure 53: Testbench code - decoder

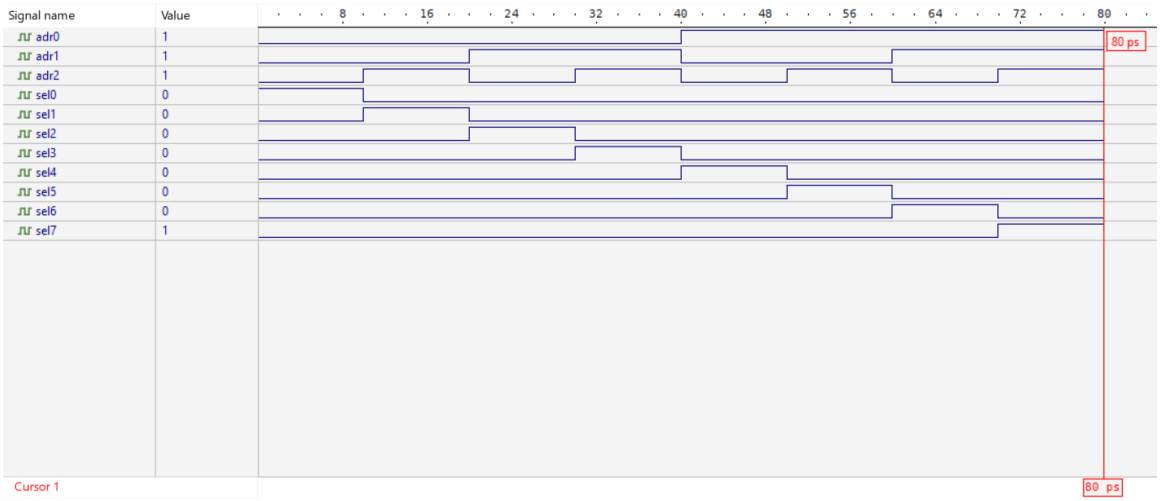


Figure 54: Simulation - decoder

```
1  module testbenchv2;
2
3      reg op,select,clk;
4      wire rw,valid;
5
6      fsm3v2 uut (
7      .clk(clk),
8      .op(op),
9      .select(select),
10     .rw(rw),
11     .valid(valid));
12
13     initial begin
14     clk = 0;
15     op = 0; select = 0; #10;
16     op = 0; select = 1; #10;
17     op = 1; select = 0; #10;
18     op = 1; select = 1; #10;
19     op = 0; select = 1; #10;
20     op = 1; select = 1; #10;
21     op = 1; select = 0; #10;
22     op = 0; select = 0; #10;
23     op = 1; select = 1; #10;
24     end
25
26
27
28     always begin
29     #5 clk = ~clk;
30     end
31
32 endmodule
```

Figure 55: Testbench code - FSM

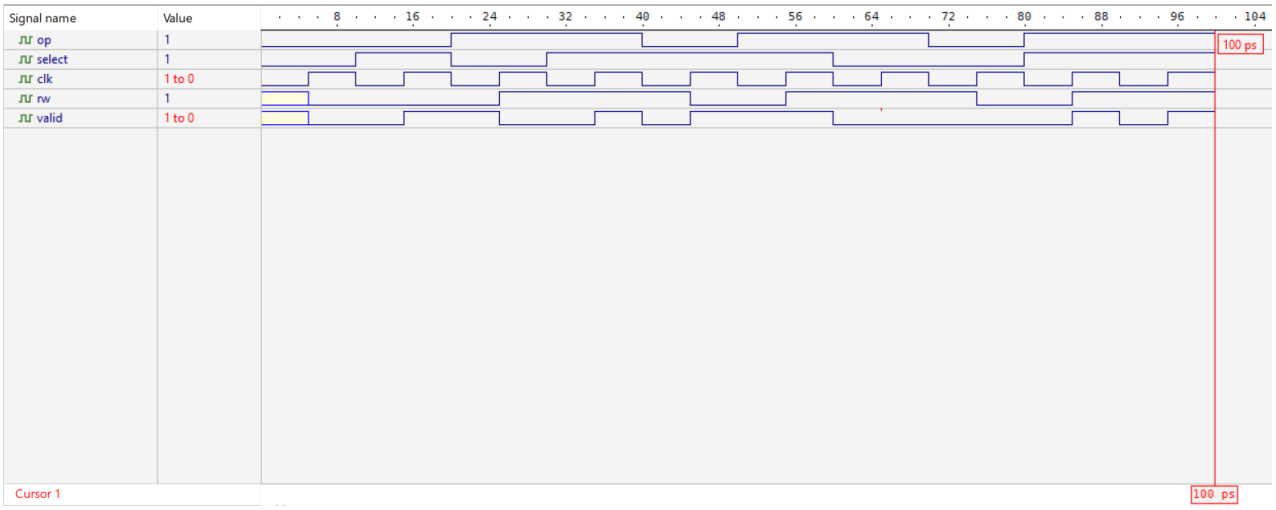


Figure 56: Simulation - FSM