ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 242E

DIGITAL CIRCUITS LABORATORY HOMEWORK REPORT

HOMEWORK NO : 3

LAB SESSION : FRIDAY - 16.00

GROUP NO : G18

GROUP MEMBERS:

150200916 : Denıs Iurıe Davıdoglu

150220770 : Onur Baylam

SPRING 2023

Contents

1 INTRODUCTION

In this homework, three-state logic is explored for designing data buses and basic memory. Along low and high states, a net can also get high impedance state, which behaves as if not connected to the circuit at all. By connecting a zero or an one to high impedance wire, there is no risk of creating short circuit, and the wire gets the same voltage as the connected input. This allows turning outputs of a logic circuit on and off, and connecting many three-state units to a common bus.

2 MATERIALS AND METHODS

2.1 Part 1

- Before starting with the first part, the most basic component of three-state logic, the buffer, is designed. Buffers can be inverting and non-inverting, and with the goal of transmitting data to a bus, non-inverting type is considered. There are two inputs and one output; if enable input is low, output should be high impedance state, otherwise output will be equal to input data. Using a continuous assignment with ternary operator in Verilog, buffer is implemented and simulated:

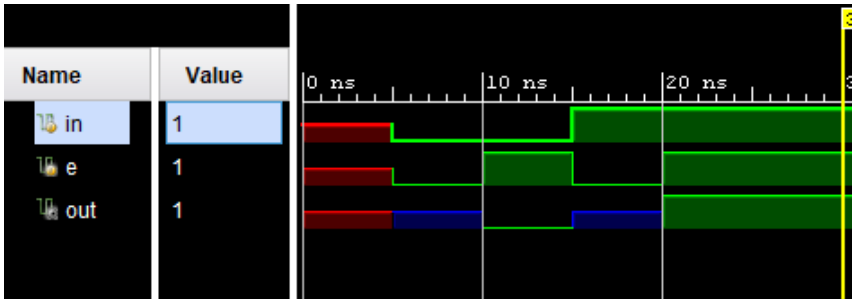

Figure 1: 3-state buffer simulation

Next, to test the capability of designed buffer to form a bus, the following circuit is tested:

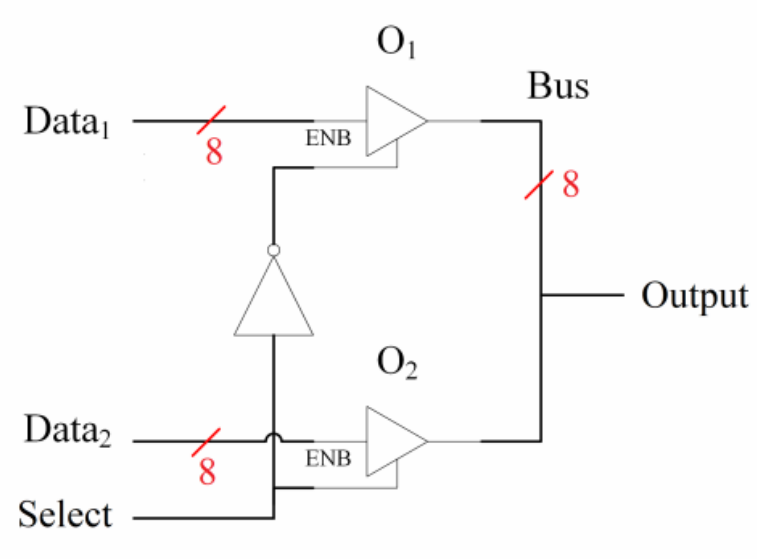

Figure 2: 8-bit data bus with 2 drivers with 3-state buffers

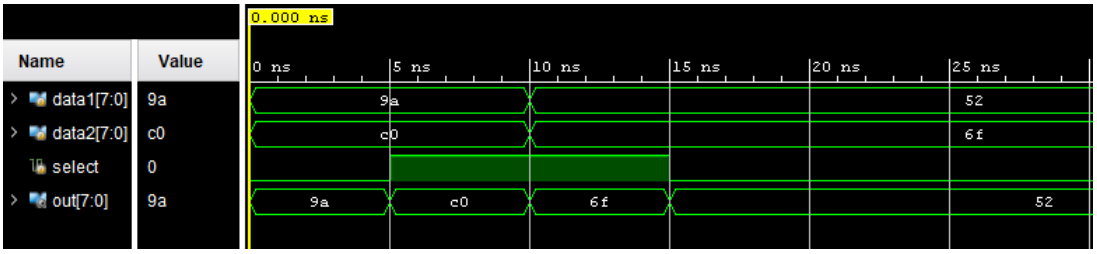

Figure 3: Part 1's simulation

As it is seen from simulation, we are capable of multiplexing data using select input, without actually using a multiplexer.

2.2 Part 2

- Buses are supposed to not only be written to, but also read from. By extending the module from previous part, two controlled outputs can be added, as in the diagram:

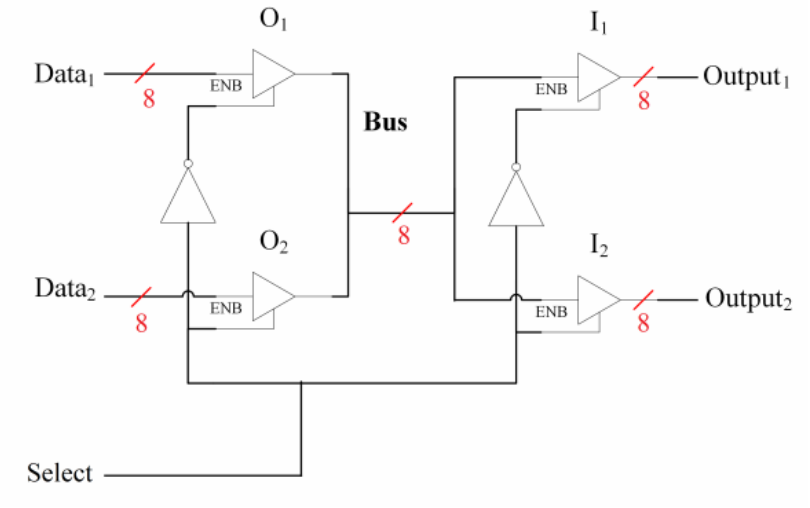

Figure 4: 8-bit data bus with 2 drivers and 2 readers

The behaviour of the outputs in this module slightly differs from the previous module. While the bus gets the same data, we do not pay attention to it anymore, and instead focus on the controlled outputs. When an output is not selected, it enters high impedance state:

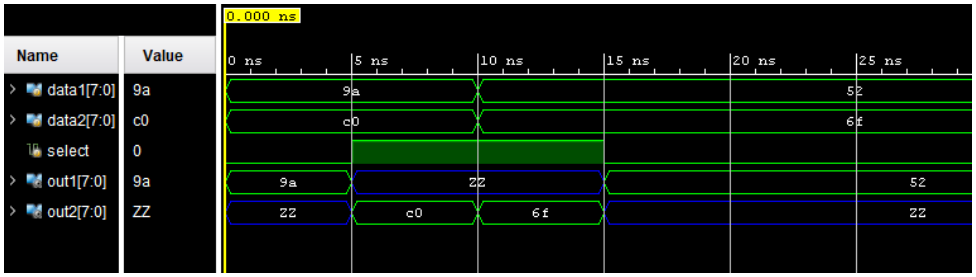

Figure 5: Part 2's simulation

2.3 Part 3

- Here an 8-bit memory line module is explained. In order to use registers on a common bus, their outputs should be tri-stated using buffers. They should have write enable and read enable inputs, and normally these will not be active both at the same time, because the both are meant to be connected to bus. Module has a positive edge triggered clock for loading data and a negative edge triggered reset. Outputting internal data is done asynchronously, and the advantage is that in a real-world application, data should be already stabilized on the bus upon being read.

Clock	Reset	LineSel	Write	Read	Data	Output
⎍	ϕ	1	1	ϕ	Input	?
ϕ	⎍	ϕ	ϕ	ϕ	0	?
ϕ	ϕ	1	ϕ	1	?	Data
ϕ	ϕ	0	ϕ	1	?	Z
ϕ	ϕ	1	ϕ	0	?	Z
ϕ	ϕ	0	ϕ	0	?	Z

Figure 6: 8-bit memory line's truth table

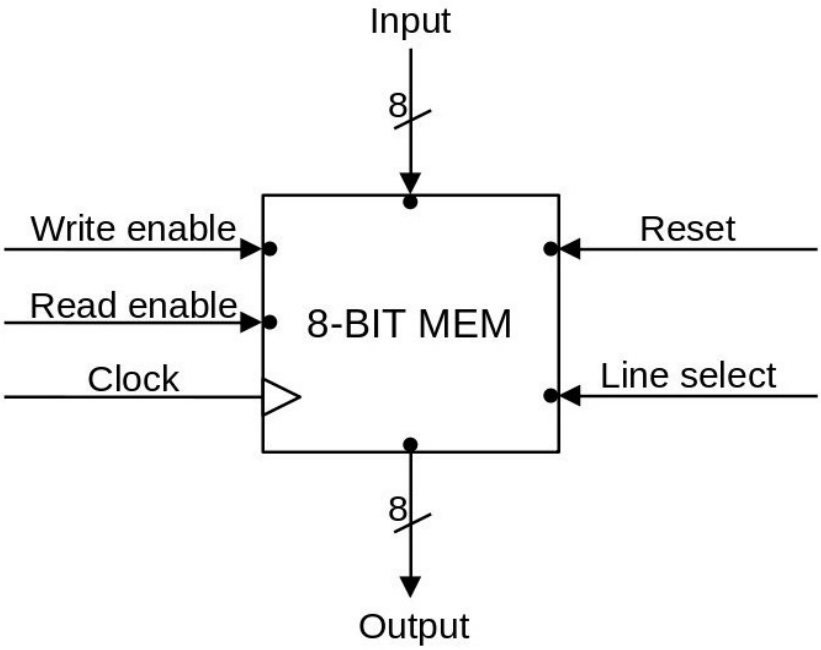

Figure 7: 8-bit memory line module

Memory line module's simulation results corresponds to the function table:

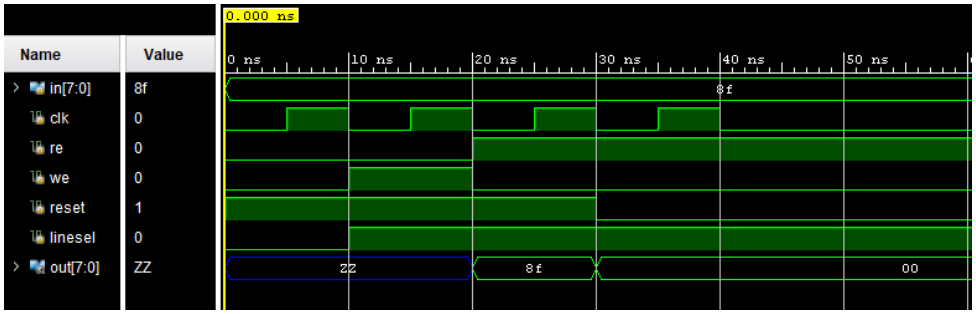

Figure 8: Part 3's simulation

2.4 Part 4

- Many 8-bit memories can be arranged to form an 8-byte memory. Input, clock, reset, read enable and write enable are common for all of 8-bit memories, and to perform individual read and write operations, line selects are utilized. 8-bit memory has extra 3-bit adress input and chip select, which ramify through a decoder to enable small memories one at a time:

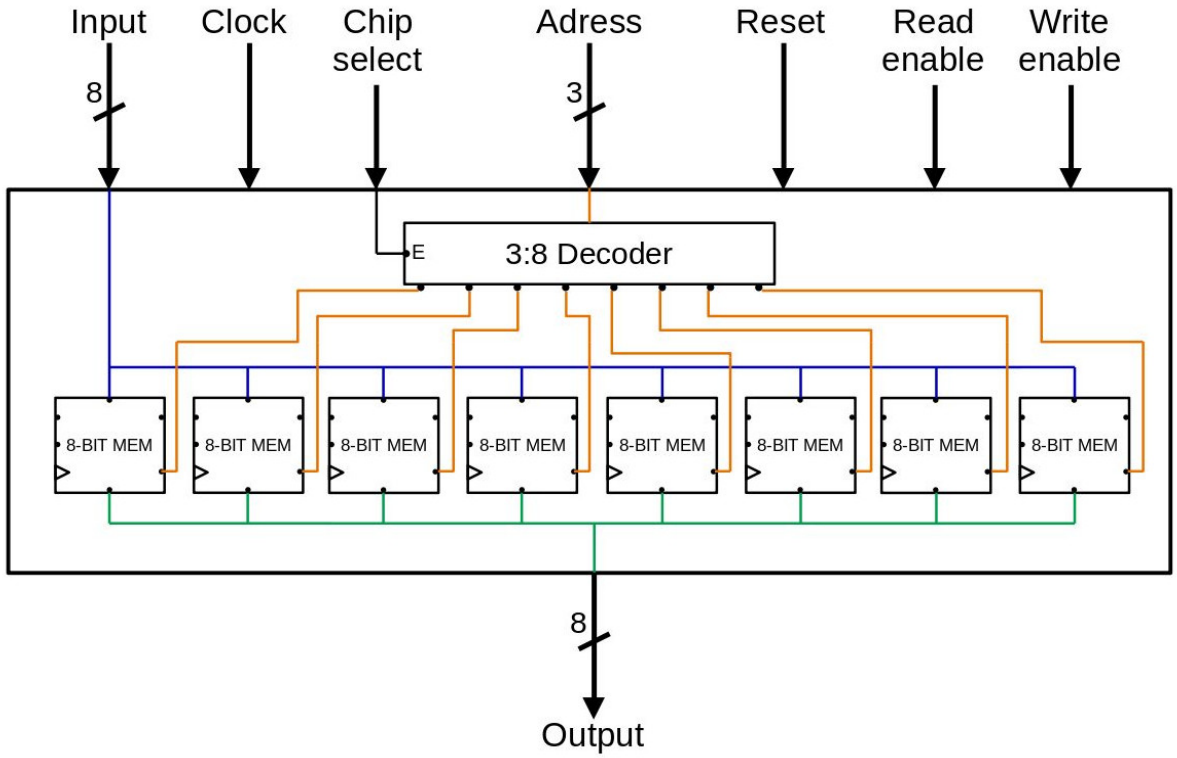

Figure 9: 8-byte memory using 8-bit memory line modules

Testbench for this memory checks the correctness of load, output and clear operations:

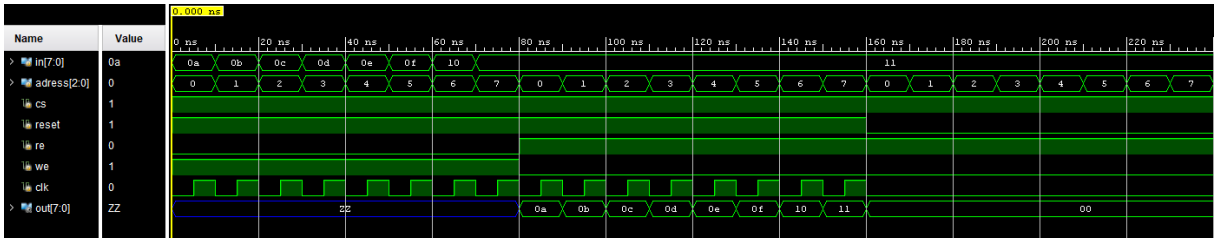

Figure 10: Part 4's simulation

2.5 Part 5

- In this part, we have implemented a 32-byte memory module using our 8-byte memory module which is implemented in the previous part. In addition, 32-byte memory module has following inputs: *clk* : Clock signal, *in* : 8-bit data input, *a* : 5-bit address, *reset* : reset signal, *re* : read enable, *we* : write enable, *out* : 8-bit memory line output. To validate our module, we simulated it with given cases.

Simulation Result of 32-byte Memory Module:

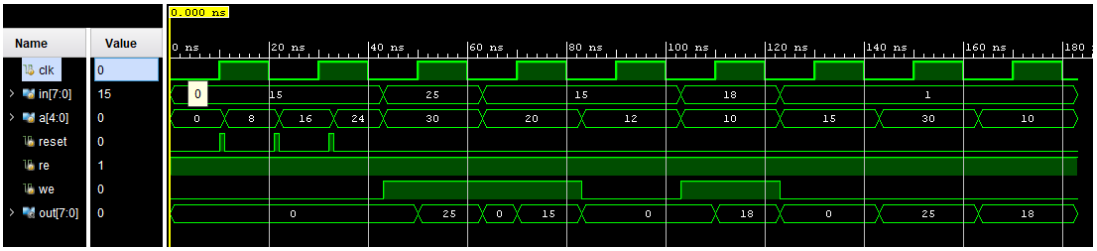

Figure 11: Inputs and outputs are in decimal format

As it is observable, we reset all memory lines until $40ns$ at the falling edges of *reset* signal. Afterwards, we set some values to specific address locations in the memory such that we write 25 to address 30 at $50ns$. In addition, to be able to write into memory lines, our *we* (write enable) signal should be high. Also, our *re* signal is set to 1 during simulation, to perform reading from the memory. Otherwise, our output will be high-impedance. For instance, in the second simulation result, which is realized with different input combinations, we can see that when *re* (read enable) is low at $80ns$, the output of the module is high-impedance.

Second Simulation Result of 32-byte Memory Module:

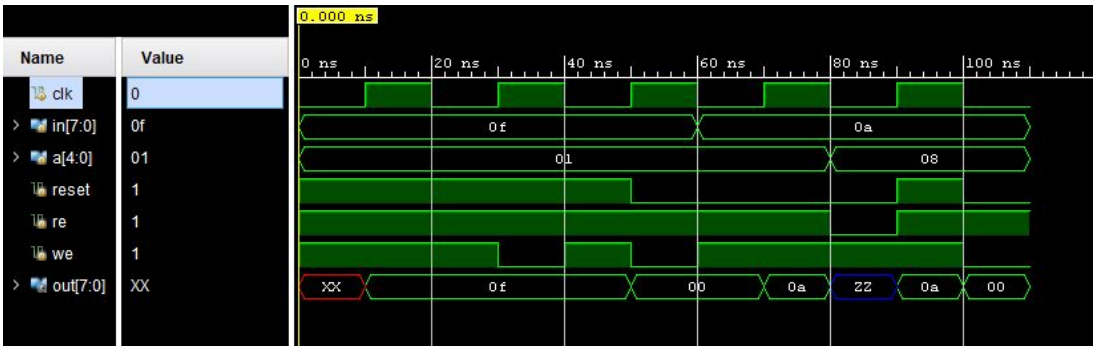

Figure 12: Inputs and outputs are in decimal format

Finally, we provide with RTL Schematic of the module below.

RTL Schematic of 32-byte Memory Module:

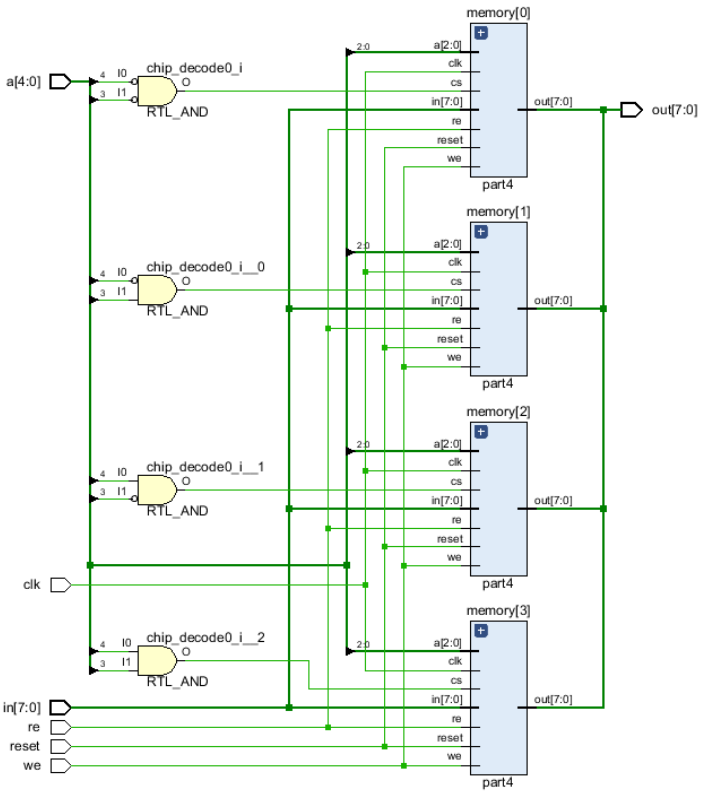

Figure 13: RTL Schematic of 32-byte memory module

2.6 Part 6

- In order to implement 128-byte memory module, we used our 32-byte memory module, which is implemented in the fifth part. Our 128-byte memory module has 32-bit data input and 32-bit data output and as similar to the 32-byte memory module, it has also following signals: a : 5-bit address input, re : read enable signal, we : write enable signal, and $reset$ signal. Moreover, it consists of four 32-byte memory module. To validate our design, we firstly simulate 128-byte memory module with the same operations as the previous part and expect same outputs, which should be the case.

Simulation Result of 128-byte Memory Module:

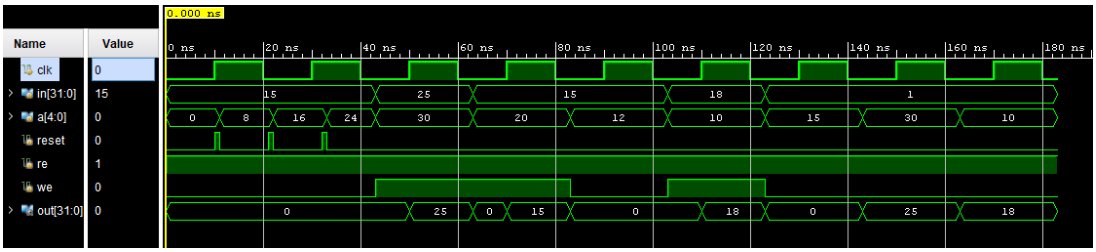

Figure 14: Inputs and outputs are in decimal format

NOTE: To make analyzing the results easier, inputs and outputs of the module are provided in decimal format.

As it is seen from the simulation results, firstly we reset memory lines, then set specific values to different memory locations. For example, at $70ns$ we write 15 to the memory location of 20. In addition, to be able to write into memory, our *we* signal should be high. Otherwise, we cannot write into memory. Also, to read from the memory *re* signal should be high if not, our output would be high-impedance. Finally, we provide with RTL Schematic of the module below.

RTL Schematic of 128-byte Memory Module:

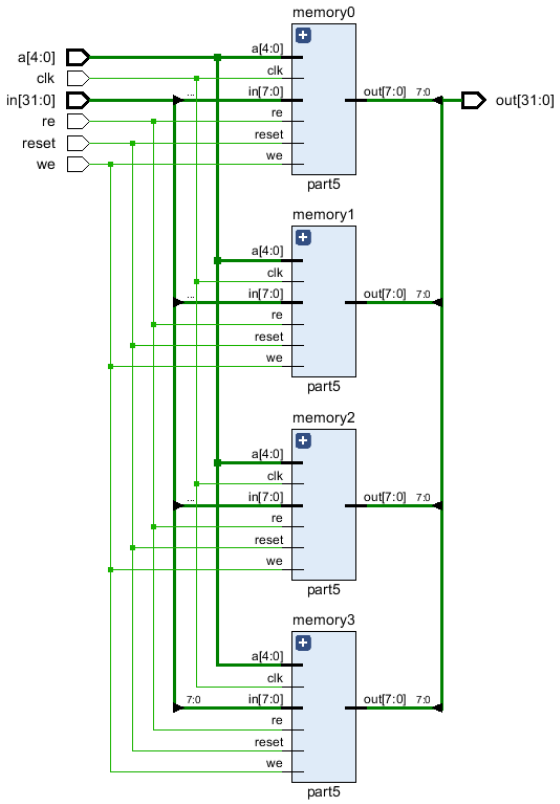

Figure 15: RTL Schematic of 128-byte memory module

3 CONCLUSION

As a consequence, firstly, we designed a three-state buffer module in Verilog HDL, which is used as a fundamental step to implement further modules like 8-bit data bus module. Also, we created more complex memory modules such as 8-bit memory line module, 8-byte memory module, 32-byte memory module, and 128-byte memory module using each implemented module in the next parts. For instance, 128-byte memory module consists of four 32-byte memory module. Afterwards, we simulated each module with different input combinations to ensure that our modules operate accurately. Instead of giving arbitrary input values in the simulations, we selected our input combinations carefully to observe all edge-case conditions. Finally, we tried to code our modules in the simplest way, which makes coding process easier and prevent us from having difficulties in detecting possible mistakes in the code.