Sequential Logic Circuits

Learning Outcomes

At the end of this lab, you should be able to:

1. differentiate combinational and sequential elements in a processor;
2. implement sequential logic circuit elements in VHDL;

Contents

1 Resources **1**

2 Discussion **1**
- 2.1 Clocks . . . 1
- 2.2 Latches . . . 3
- 2.3 Flip-Flops . . . 4
- 2.4 Register Files . . . 5
- 2.5 Static RAM . . . 7
- 2.6 Dynamic RAM . . . 10
- 2.7 Finite State Machines . . . 10
  - 2.7.1 Example: Traffic Lite . . . 11

3 Summary **12**

4 Learning Activities **13**

5 Deliverable **13**

6 Further Reading **13**

1 Resources

- Video: https://youtu.be/CzyXb_T-xgU.
- Source Codes: https://git.io/JU3al

2 Discussion

The discussion in this handout aims only to provide an outline of what is in the video lecture. It is recommended that you watch the video in its entirety.

Sequential elements are sometimes called memory elements because they store state information. The *output* of any memory element dependes on both the *input* and the *value stored* in it. An understanding of clocks is important in the study of sequential elements.

2.1 Clocks

Clocks determine when the current state of a sequential elements needs to be updated. It is a *signal* that transitions from low to high and high to low in a fixed cycle time or clock period. Figure 1 shows an example clock signal. It is a digital signal so it is represented by a square wave and the transition from low to high or high to low is abrupt unlike in analog signals. These abrupt transitions are called *clock* *edges* which may be a *rising edge* (low to high) or *falling edge* (high to low). The *clock period* is the time to complete a cycle and the number of cycles per second is called the *frequency* in *hertz*. The higher the frequency the higher the number of cycles per second. We will use these parameters later when we go to the performance evaluation lab.

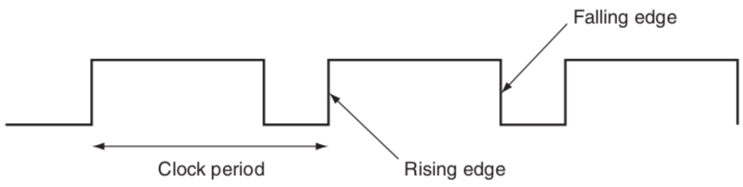


Figure 1: A clock signal showin the clock period, falling edge, and rising edge.

The updating of state elements is usually done at clock edge to ensure that the signals will be valid. Figure 2 shows a combinational element sandwiched between two state elements. Observe that the updating of the state elements is done at the rising edge. Using the clock edge as marker allows the same state element to be the input and output of the combinational element, without invalidating the signal.

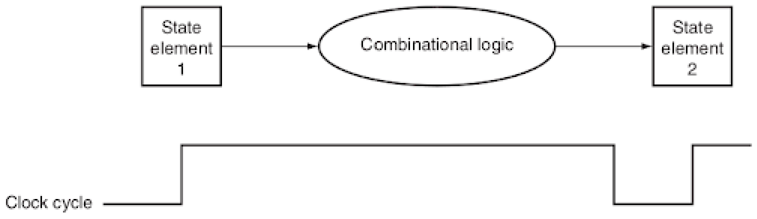


Figure 2: A combinational element is sandwiched between two state elements. The state elements are updated at the rising clock edge.

The VHDL code below shows how to generate a clock signal *clk* that can be used in a testbench as input to sequential elements being tested.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY clocker_tb IS
END clocker_tb;
ARCHITECTURE behavior OF clocker_tb IS
   --100Mhz
   CONSTANT frequency: integer := 100e6;
   CONSTANT period : time := 1000 ms / frequency;
   SIGNAL clk : std_logic := '0';
BEGIN
   clk <= not clk after period / 2;
   -- do some stuff here using clk as input
END ARCHITECTURE;
```

2.2 Latches

Latches are unclocked memory elements. Figure 3 shows a Set-Reset(SR) latch using two NOR gates. Q is the main output and Q bar is the complement of Q. When Q is true, if S is asserted then Q will be asserted. When R is asserted, then Q bar will be asserted.

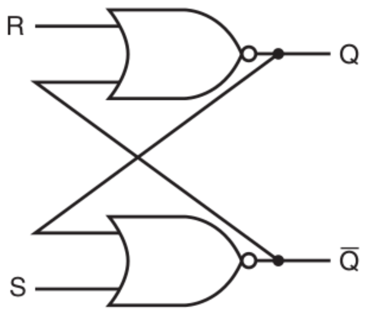


Figure 3: A Set-Reset latch.

The code VHDL code for the S-R latch is shown below.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------
ENTITY sr_latch IS
   PORT (R, S: IN STD_LOGIC;
   Q, Q_BAR: INOUT STD_LOGIC);
END sr_latch;
--------------------------------
ARCHITECTURE pure_logic OF sr_latch IS
BEGIN
   Q <= R NOR Q_BAR;
   Q_BAR <= S NOR Q;
END pure_logic;
```

When a clock is added to a latch, it is called a clocked latch. Figure 4 shows a D latch. The clock input is C and the data input is D. Q is the internal state. It is basically an extension of the S-R latch. An important thing to remember is that in clocked latches, *the state changes whenever the input change and the clock is asserted.*

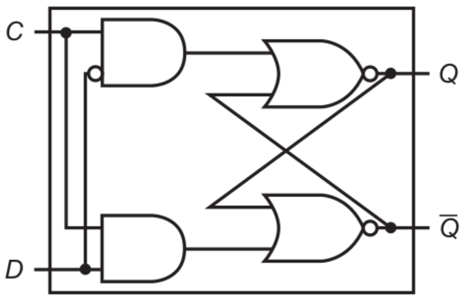


Figure 4: A D latch.

The VHDL code for the D latch is shown below.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------
ENTITY d_latch IS
   PORT (C, D: IN STD_LOGIC;
   Q, Q_BAR: INOUT STD_LOGIC);
END d_latch;
--------------------------------
ARCHITECTURE pure_logic OF d_latch IS
BEGIN
   Q <= (C AND NOT D) NOR Q_BAR;
   Q_BAR <= (D AND C) NOR Q;
END pure_logic;
```

2.3 Flip-Flops

Flip-flops are similar to clocked-latches. The main difference is the update of the state happens at the edge of the clock signal. Recall the advantage of edge-triggered clocking methodology above. Figure 5 shows a D flip-flop using D latches.

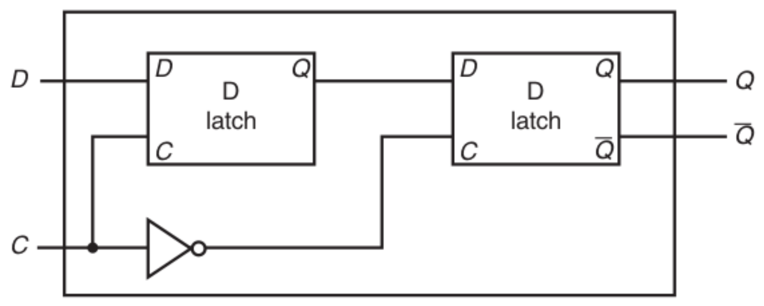


Figure 5: A D flip-flop created using two D latches in a master-slave configuration.

The VHDL code for this D flip-flop is shown below. There is a simpler code for this described in the video.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY dff is
   PORT( D: IN STD_LOGIC;
      C: IN STD_LOGIC;
      Q: INOUT STD_LOGIC;
      Q_BAR: INOUT STD_LOGIC);
END dff;
ARCHITECTURE behavioral OF dff IS
   COMPONENT d_latch IS
      PORT (C, D: IN STD_LOGIC;
      Q, Q_BAR: INOUT STD_LOGIC);
   END COMPONENT;
   SIGNAL u,v: STD_LOGIC;
   SIGNAL NOTC: STD_LOGIC := NOT C;
BEGIN
   master: d_latch port map (C, D, u, v);
   slave: d_latch port map (NOTC, u, Q, Q_BAR);
END behavioral;
```

2.4 Register Files

A register file consists of a set of registers that can be read and written by supplying a register number to be accessed. Figure 6 shows an example register file where two registers can be read and one register can be written into. In order to write to a register, the *Write* enable control signal should be asserted.

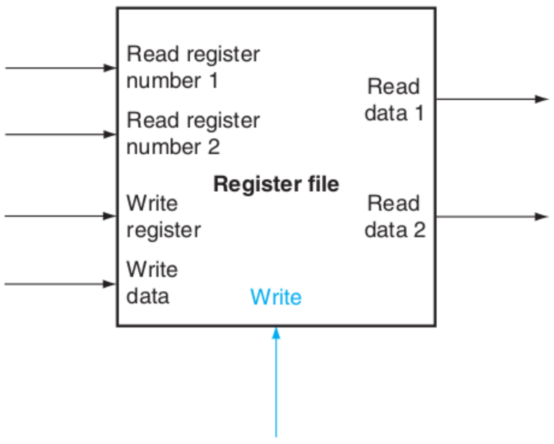


Figure 6: 2-to-1 Multiplexer.

Registers are implemented as flip-flops. A multiplexer is used to select which register output will be allowed to pass through during read. Figure 7 shows two multiplexers connected to the array of registers because the register file allows two registers to be read.

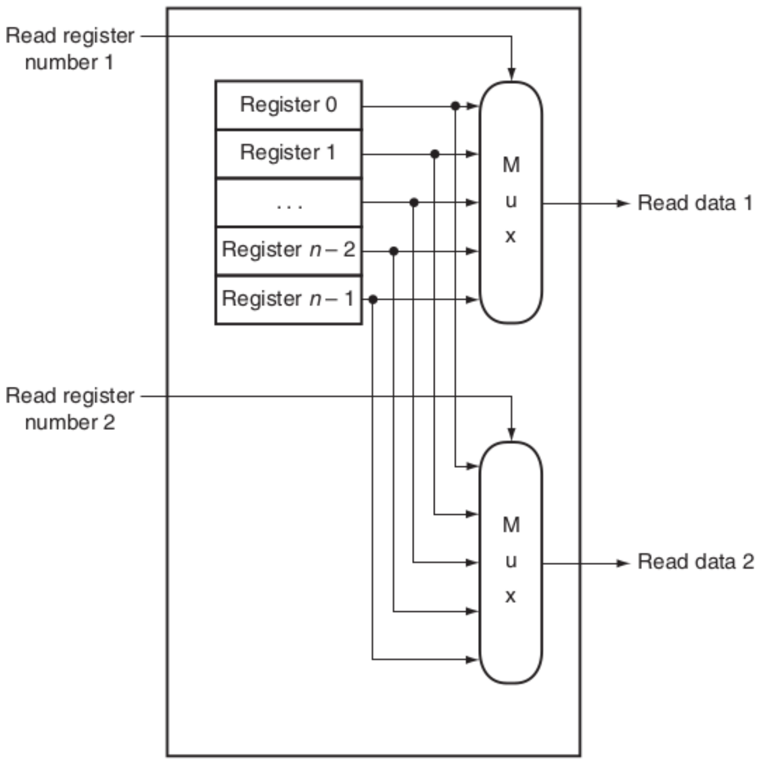


Figure 7: Reading in a register file.

Writing to a register file will require a decoder to select the register to write to. The output of the decoder is and-ed to the *Write* control signal. The result is used as the clock input to the register(flip-flop) as shown in Figure 8.

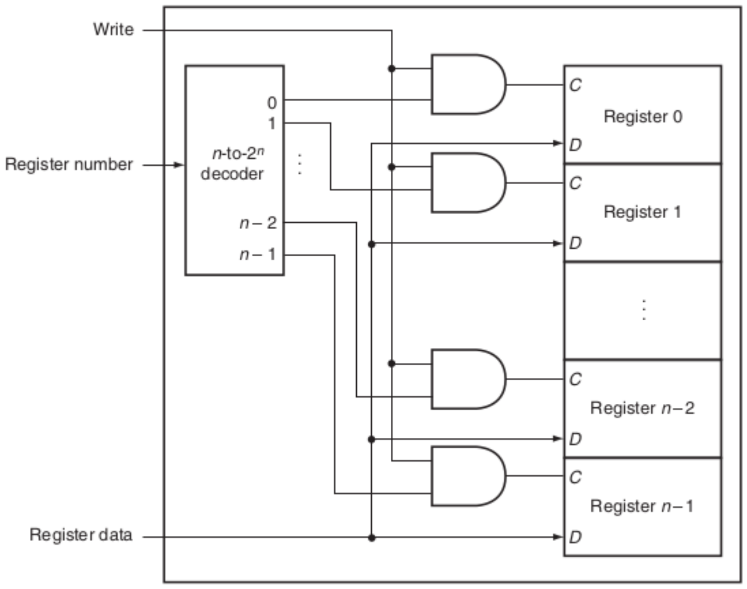


Figure 8: Writing to a register file.

The VHDL source code for a register file with 2 1-bit registers is shown below. The *registers* signal is defined as an array type. The register numbers are used as index into this array.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY reg_file2 is
   PORT(
      read_n1: IN STD_LOGIC;
      read_n2: IN STD_LOGIC;
      write_n: IN STD_LOGIC;
      write_data: IN STD_LOGIC;
      write: IN STD_LOGIC;
      clk: IN STD_LOGIC;
      read_data1: OUT STD_LOGIC;
      read_data2: OUT STD_LOGIC);
END reg_file2;
ARCHITECTURE behavioral OF reg_file2 IS
TYPE rf_type IS ARRAY(0 to 1) of STD_LOGIC;
SIGNAL registers : rf_type;
BEGIN
   rf: PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clk) THEN
         IF (write ='1') THEN
            registers(TO_INTEGER(UNSIGNED'('0' & write_n))) <= write_data;
         END IF;
      END IF;
      read_data1 <= registers(TO_INTEGER(UNSIGNED'( '0' & read_n1)));
      read_data2 <= registers(TO_INTEGER(UNSIGNED'( '0' & read_n2)));
   END PROCESS;
END behavioral;
```

Registers are fast memory elements but has limited storage. For example, in the Intel x86_64 processor, the size of the the registers is 64-bits. For larger storage, Random Access Memory (RAMs) is used.

2.5 Static RAM

Static RAM (SRAM) has a specific configuration in terms of the *number of addressable locations* and *width of each addressable location*. Figure 9 shows an SRAM with 2M addressable locations and each location is 16 bits. The address line is 21 bits and the data input and output is 16 bits.

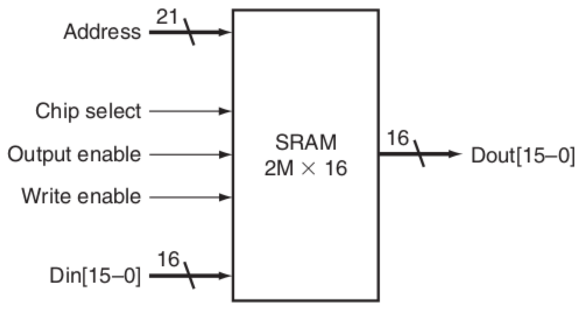


Figure 9: A 2Mx16 SRAM.

Since SRAMs can store large amount of information, using a multiplexer to select the location is not cost-effective. Instead, a shared output line, called a *bit line*, allows multiple memory cells in the memory array to assert. The circuitry to allow enable this is called the *three-state buffer* or *tristate buffer* (Figure 10).

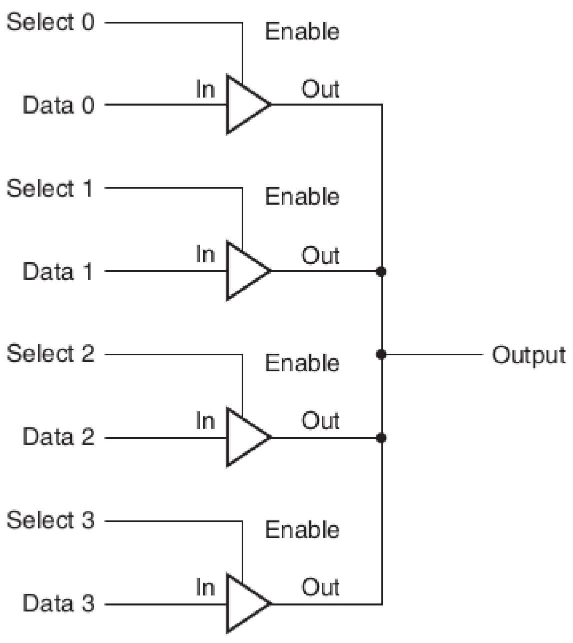


Figure 10: Tristate buffer allows multiple memory cells to share a bit line making it more cost-effective than using a multiplexer.

The VHDL code for the D flip-flop with the *Enable* line and *Reset* line is shown below.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY DFF is
PORT( din: IN STD_LOGIC;
      clk: IN STD_LOGIC;
      rst: IN STD_LOGIC;
      en: IN STD_LOGIC;
      dout: OUT STD_LOGIC);
END DFF;

ARCHITECTURE behavioral of DFF is
BEGIN
   PROCESS(rst,clk,din)
      BEGIN
         IF (rst='1') THEN
            dout<='0';
         ELSIF(RISING_EDGE(clk)) THEN
            IF (en='1') THEN
               dout<= din;
            ELSE
               dout<='Z';
            END IF;
      END IF;
   END PROCESS;
END behavioral;
```

A small 4x2 SRAM might be built using D latches with an *Enable* line.

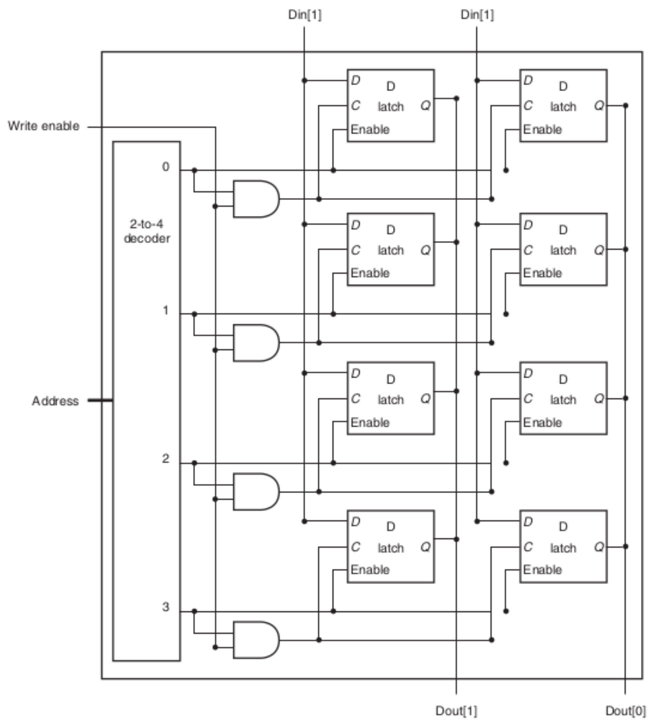


Figure 11: A 4x2 SRAM.

To reduce the size of the decoder, SRAMs use a two-step decoding method. In Figure 12, the first decoder generates the addresses for an eight 4K x 1024 arrays then a set of multiplexers is used to select 1 bit from each 1024-bit-wide array. The address lines are split into two groups.

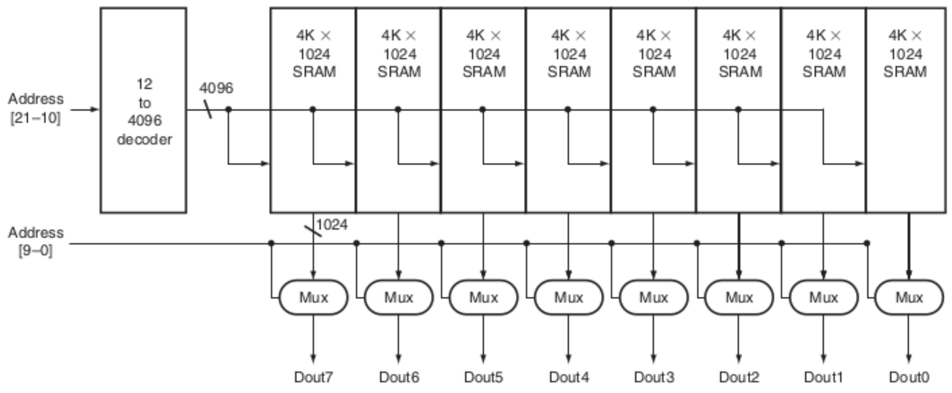


Figure 12: Two-step decoding to reduce the size of the decoder in SRAM.

2.6 Dynamic RAM

The values in SRAM are stored in the flip-flops. As long as there is power, the values stored can be kept indefinitely. In Dynamic RAM (DRAM), the values are stored as charge in a capacitor. It uses a single transistor to access the the values. SRAMs require four to six transistors per bit, thus DRAMs are much cheaper for use in main memory. SRAMs are usually used for caches. The main drawback for DRAMs is that it must periodically be refreshed (read the values then write again), thus the word *dynamic*. DRAMs use two-level decoding consisting of a row access followed by a column access. Figure 13 shows a 4Mx1 DRAM. Notice that a single address line is used to for both the row and column access. A pair of signals are used to tell whether the value in the address line is for row or column. See Further Reading section on a simple implementation of RAM in VHDL.

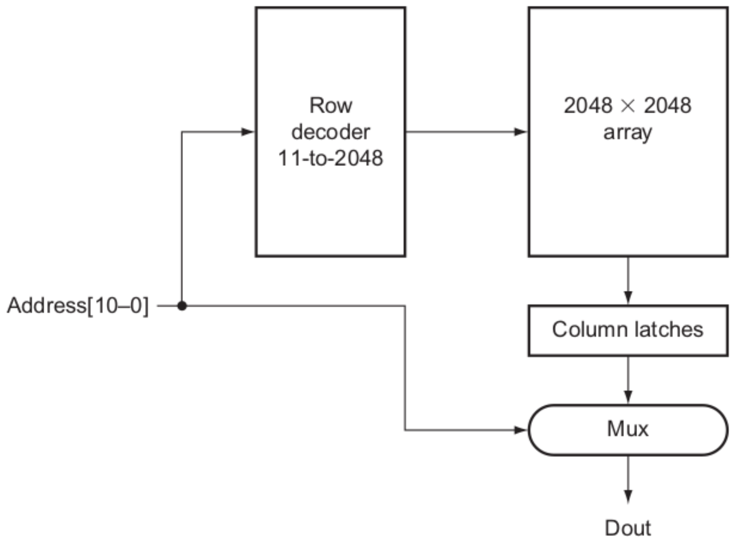


Figure 13: A 4Mx1 DRAM with 2048x2048 array.

2.7 Finite State Machines

Sequential systems are described using Finite State Machines (FSMs). Their behaviour depends on both the input and internal state thus using a truth table is not enough. Figure 14 shows a block diagram of the main components of an FSM.

- *Set of States* - corresponds to all the possible values of the internal storage
- *Current State* - corresponds to current value of the internal storage
- *Next-state Function* - a combinatinal function that, given the current state, determines the next state of the system.
- *Output function* - produces a set of outputs from the current state and the inputs

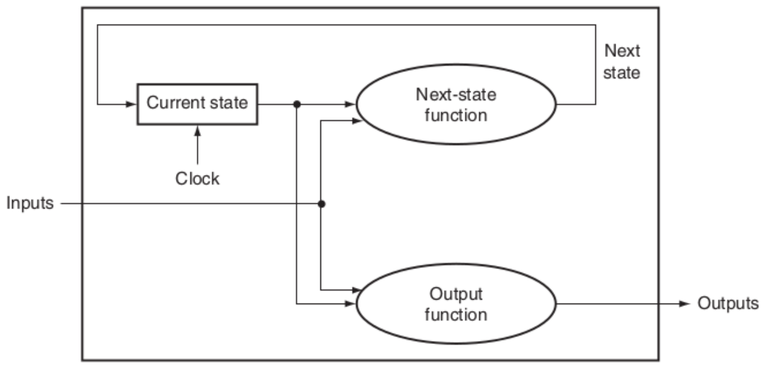


Figure 14: A Finite State Machine and its main components.

2.7.1 Example: Traffic Lite

Let us design a simple traffic light system. The following tables and figures contain the specification.

Table 1: Outputs.

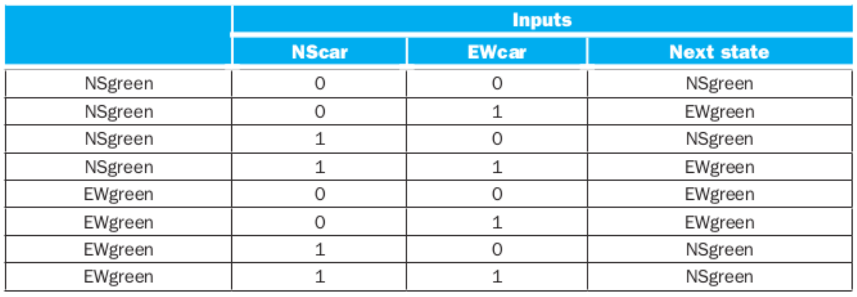
Signal	Description
NSLite	When this signal is asserted, the light on the north-south road is green; when this signal is deasserted, the light on the north-south road is red
EWLite	When this signal is asserted, the light on the east-west road is green when this signal is deasserted, the light on the east-west road is red

Table 2: Inputs.

Signal	Description
NScar	Indicates that a car is over the detector placed in the roadbed in front of the light on the north-south road (going north or south)
EWcar	Indicates that a car is over the detector placed in the roadbed in front of the light on the east-west road (going east or west) red

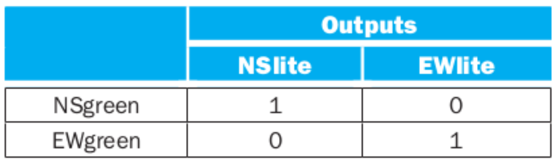
Table 3: States.

Signal	**Description**
NSgreen	The traffic light is green in the north-south direction
EWgreen	The traffic light is green in the east-west direction.



	Inputs		
	NScar	EWcar	Next state
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

Figure 15: Next-state function.



	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

Figure 16: Output function.

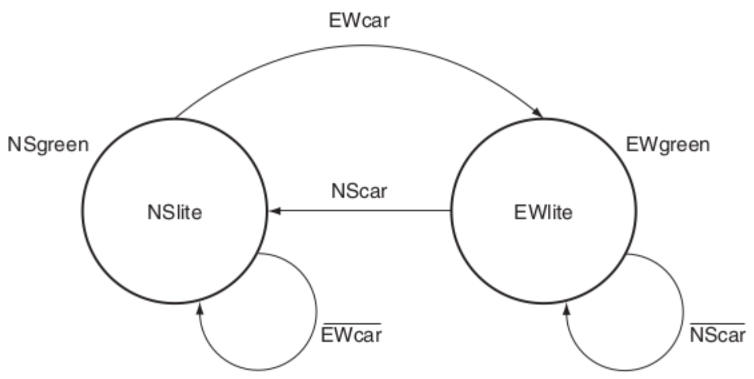


Figure 17: 2-to-1 Multiplexer.

3 Summary

We discussed some of the sequential elements that are useful in the design of a processor. We also showed the design and implementation of a simple traffic light system using finite state machines.

4 Learning Activities

Download the source codes for this lab then try experimenting by adding more test cases in the testbenches. Submit a PDF document that shows screenshots of your modifications and runs.

5 Deliverable

Your final deliverable for this lab is described in the accompanying exercise handout.

6 Further Reading

- https://www.doulos.com/knowhow/vhdl/simple-ram-model/

References

[1] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, ARM Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, arm edition, 2017.