## Contents


FPGA (Field Programmable Gate Array) ..... 2

Configurable Logic Blocks (CLBs) ..... 5

Configurable I/O Blocks ..... 6

Programmable Interconnect ..... 6

HDL (Hardware Description Language) ..... 8

Verilog HDL (Hardware Description Language) ..... 8

1. Gate- and switch-level modeling (Structural Model) ..... 8

Combinational Circuits ..... 9

Squential Circuits. ..... 9

2. Assignments (Dataflow Representation) ..... 9

3. Behavioral Modeling (Behavioral Representation) ..... 10

Coding with Verilog Language ..... 11

1. Half Adder Using Verilog ..... 11

2. Full Adder Using Verilog ..... 13

3. 2-bit Binary Adder ..... 15

Verilog Testbench ..... 16

4. Multiplexer ..... 17

5. Decoder ..... 19

6. 8 bit Adder/ Subtractor ..... 21

7. SR (Set / Reset) Latch ..... 22

8. D Latch ..... 23

Edge Triggering (Edge-sensitive) ..... 24

Level Triggering (Level-sensitive) ..... 24

9. Register ..... 27

Synchronous and Asynchronous Sequential Circuits ..... 27

Finite State Machines ..... 28

10. Counter ..... 30

11. Integrated Systems Architecture Exercise ..... 31

12. Carry Ripple Adder ..... 32

13. Carry-Lookahead Adder ..... 33

14. Unsigned Multiplication ..... 35

15. FSM with Synchronous Sequence Circuit Design ..... 36

References ..... 39

# FPGA (Field Programmable Gate Array)

If examined historically, it will be better understood why FPGA emerged and what it does.

As a programmable door series before 1985;

PLA (Programmable Logic Array), programmable AND and OR gates could be defined as bool functions in Sum of Product (SoP) form.

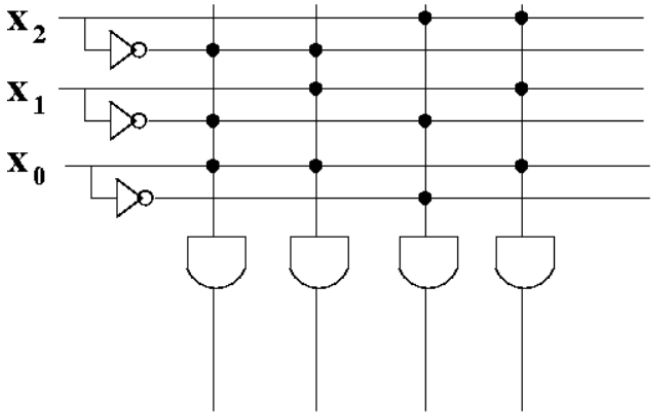
PLA implementation procedure is as follows.

When we examine the truth table of the given inputs and outputs, we should not forget that we will first perform the PLA process according to the minterms (SoP) of the outputs.

According to this truth table, there are 4 minterm cases. These minterm states:

$z1 = m1 + m3 + m4$; $z2 = m4 + m7$. This will appear on PLA as follows;

| $x_2$ | $x_1$ | $x_0$ | $z_1$ | $z_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 |




Once you have determined the minterms, they must show an exit. For this reason, the AND results of $x2x1x0$ should be output with the OR gate in the Sum of products form[1].

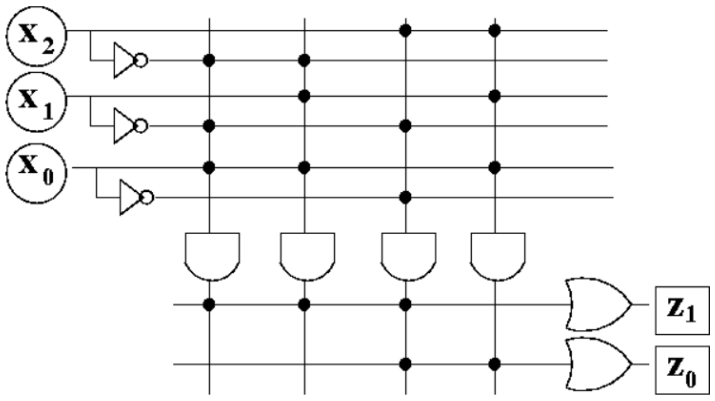



*Figure 1. PLA Architecture*

PLA is used to implement various combinational circuits using a buffer, AND gate, and OR gate. In PLA, not all minterms are implemented, only necessary minterms are implemented.

PLA provides more flexibility as it has a programmable AND gate array and a programmable OR gate array. However, its disadvantage is that it is not easy to use.

Due to significant time loss during the programming of AND and OR gate entrances, a version update has been implemented where the entrances of AND gates are programmed to reduce this time loss, but the entrances of OR gates are left unprogrammed. In this version, an attempt has been made to speed up compared to the previous version, and it has been successful.

The operation of a PLA can be summarized in three steps:

1. Programming: The user defines the logic function to be implemented by the PLA by programming the input and output configurations into the device.

2. Product term generation: The inputs are applied to the AND gate array to produce a set of product terms.

3. Sum term generation: The product terms are then applied to the OR gate array to generate the final output.

PLAs are often used in digital systems as they are versatile and allow complex functions to be implemented easily. They are particularly useful for implementing Boolean expressions with many variables as the arrays of AND gates and OR gates can be configured to handle large numbers of inputs[2].

The increasing requirements expected from PLAs and the market's need for faster programmable circuits have led to the emergence of CPLDs (Complex Programmable Logic Devices). In addition, the need for this architecture has increased as transistor technology has increased rapidly according to Moore's law and the area had to be reduced.

CPLD stands for Complex Programmable Logic Devices, and FPGA stands for Field-Programmable Gate Array. The functions of the two are essentially the same, and their programming and other processes are similar (although the programming files are different, they are automatically generated by the software). However, the internal implementation principles and chip structures differ slightly.

The programmable logic unit in CPLD serves a similar function to the basic I/O port of an FPGA. However, the application scope of CPLD is relatively limited, with differences in performance, I/O complexity, and support standards. CPLDs generally support fewer I/O standards and operate at lower frequencies compared to FPGAs.

In CPLD, the basic logic unit is the macrocell. A macrocell consists of AND or OR arrays along with flip-flops. The 'AND-OR' array within the macrocell completes the combinational logic function, while the flip-flops handle sequential logic. An important concept related to CPLD's basic logic unit is the 'product term.' This term refers to the output of the AND array within the macrocell, and its quantity indicates the CPLD's capacity. The product term array essentially functions as an 'AND-OR' array, with each intersection being a programmable fuse. When activated, it implements 'AND' logic. Following the 'AND' array, there is typically an 'OR' array, used to complete the 'or' relationship in the smallest logical expression.

The wiring resources in CPLDs are simpler compared to those in FPGAs, with relatively limited resources. Typically, CPLDs use a centralized wiring pool structure, which is essentially a switch matrix. This structure allows for connections between the input and output items of different macrocells by tying nodes. However, due to the limited interconnection resources within CPLD devices, routing may pose some challenges. Since the wiring pool structure in CPLDs is fixed, the

delay from an input pin to an output pin is also fixed, known as a Pin-to-Pin delay (Tpd). Tpd delay indicates the maximum frequency that CPLD devices can achieve and serves as a measure of the CPLD device's speed grade.

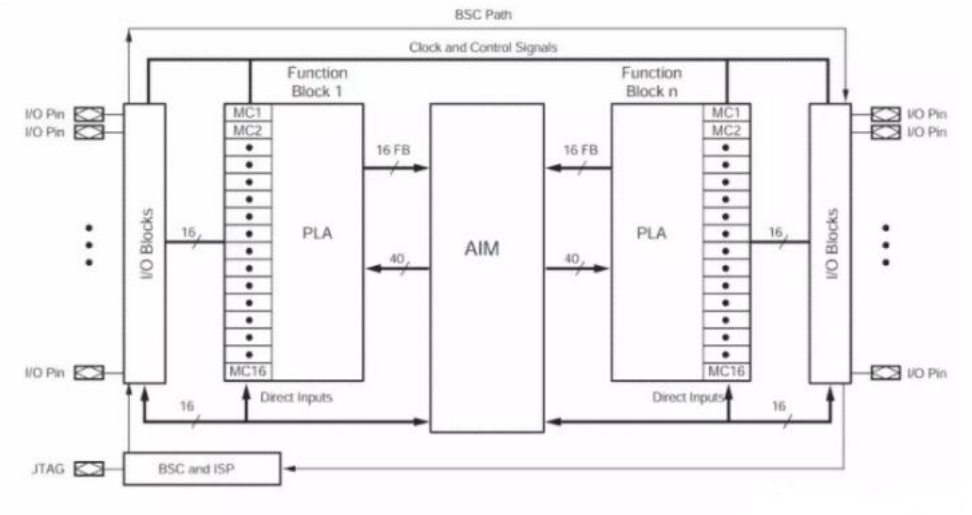



*Figure 2. CoolRunner-II CPLD Architecture*

CPLDs and FPGAs both have similar functions and programming processes but differ in their internal chip structures. CPLDs have a more limited application scope and support fewer I/O standards compared to FPGAs. In CPLDs, the basic logic unit is the macrocell, composed of AND or OR arrays and flip-flops, with 'product terms' indicating capacity. CPLDs have a simpler routing structure with a centralized wiring pool, which may pose challenges in routing due to limited resources. The Pin-to-Pin delay (Tpd) in CPLDs is fixed, indicating the device's maximum achievable frequency and speed grade[3].

The inadequacies or disadvantages of all these technologies led to the design of the FPGA device.

Field Programmable Gate Arrays (FPGAs), which closely resemble the 'gate array' format that Application-Specific Integrated Circuits (ASICs) are currently not using, actually ended up killing the ASIC gate array. Not too long ago, FPGAs were primarily marketed for two purposes: (a) prototyping ASICs and (b) deploying them in systems to achieve a quicker time to market, knowing they would eventually be replaced by an ASIC implementation.

Regarding this latter point, while FPGAs can be programmed on your desktop in a matter of minutes, ASICs require weeks to produce a new design. As FPGA speeds increased, power consumption decreased, and prices dropped, FPGAs began to be deployed in products without the need to replace them with equivalent ASICs. Of course, FPGAs are still good for prototyping ASICs and continue to be used in this way.

Each FPGA vendor has its own FPGA architecture, but in general they all consist of logic blocks, configurable I/O blocks, and programmable interconnect. There will also be clock circuitry to drive clock signals to each logic block. Additional logic resources such as ALUs, memory, and decoders may also be available. The three basic types of programmable elements for an FPGA are static RAM, anti-fuses, and flash EPROM[4], [5].

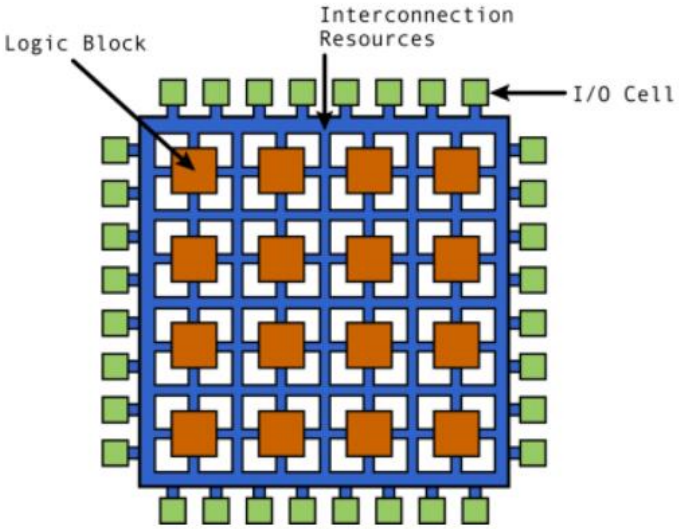



*Figure 3. FPGA Architecture*

## Configurable Logic Blocks (CLBs)

These blocks contain the logic for the FPGA. In the large-grain architecture used by all FPGA vendors today, these CLBs contain enough logic to create a small state machine as illustrated in Fig 4. The block contains RAM for creating arbitrary combinatorial logic functions, also known as lookup tables (LUTs). It also contains flip-flops for clocked storage elements, along with multiplexers to route the logic within the block and to and from external resources. The multiplexers also allow polarity selection and reset and clear input selection.

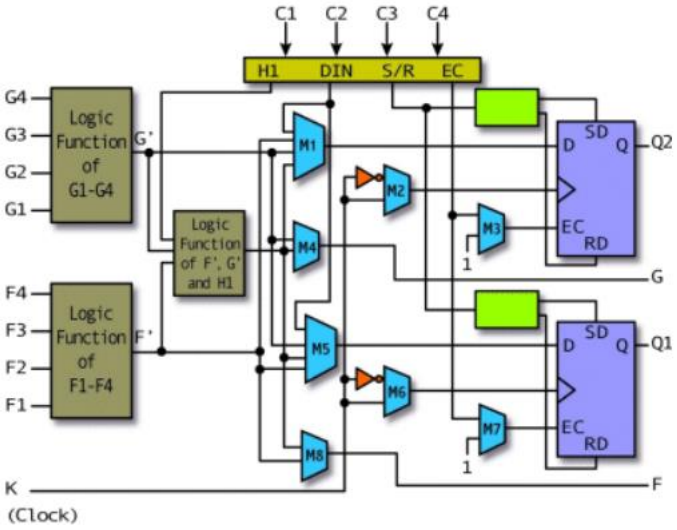



*Figure 4. FPGA Configurable logic block (CLB) (courtesy of Xilinx)*

## Configurable I/O Blocks

A Configurable input/output (I/O) Block, as shown in Fig 5., is used to bring signals onto the chip and send them back off again. It consists of an input buffer and an output buffer with three-state and open collector output controls. Typically there are pull-up resistors on the outputs and sometimes pull-down resistors that can be used to terminate signals and buses without requiring discrete resistors external to the chip.

The polarity of the output can usually be programmed for active high or active low output, and often the slew rate of the output can be programmed for fast or slow rise and fall times. There are typically flip-flops on outputs so that clocked signals can be output directly to the pins without encountering significant delay, more easily meeting the setup time requirement for external devices. Similarly, flip-flops on the inputs reduce the delay on a signal before reaching a flip-flop, thus reducing the hold time requirement of the FPGA[4].

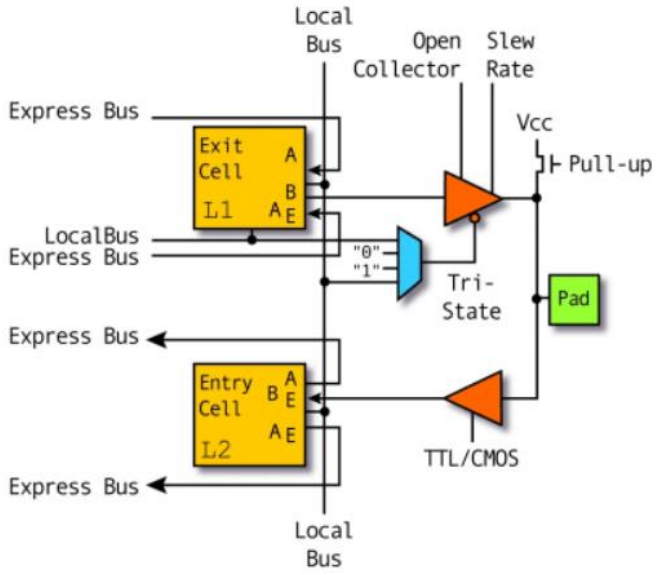



*Figure 5. FPGA Configurable I/O block (courtesy of Xilinx)*

## Programmable Interconnect

In Fig 6., a hierarchy of interconnected resources can be seen. There are long lines that can be used to connect critical CLBs that are physically far from each other on the chip without inducing much delay. These long lines can also be used as buses within the chip.

There are also short lines that are used to connect individual CLBs that are located physically close to each other. Transistors are used to turn on or off connections between different lines. There are also several programmable switch matrices in the FPGA to connect these long and short lines in specific, flexible combinations.

Three-state buffers are used to connect many CLBs to a long line, creating a bus. Special long lines, called global clock lines, are specially designed for low impedance and thus fast propagation times. These are connected to the clock buffers and each clocked element in each CLB. This is how the clocks are distributed throughout the FPGA, ensuring minimal skew between clock signals arriving at different flip-flops within the chip.

In an ASIC, the majority of the delay comes from the logic in the design, because logic is connected with metal lines that exhibit little delay. In an FGPA, however, most of the delay in the chip comes from the interconnect, because the interconnect – like the logic – is fixed on the chip. To connect one CLB to another CLB in a different part of the chip often requires a connection through many transistors and switch matrices, each of which introduces extra delay[4].

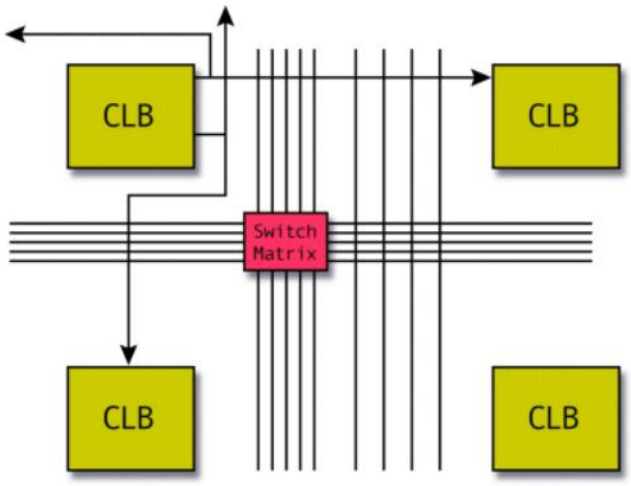
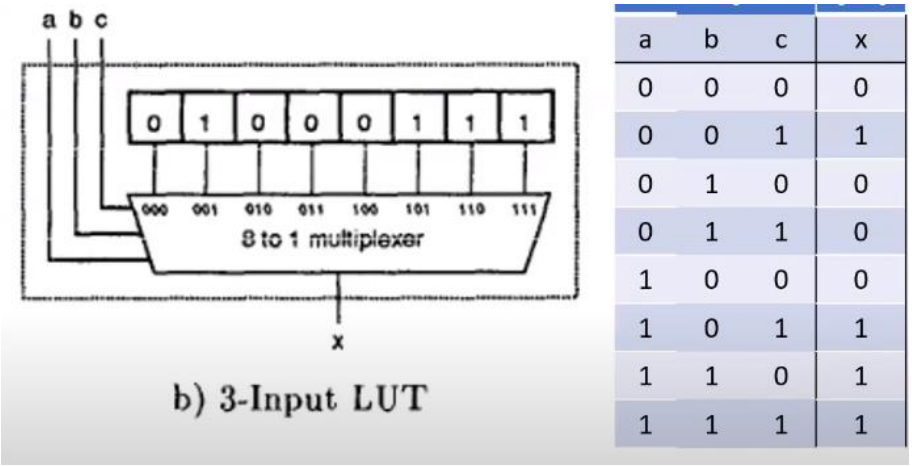



Figure 6. FPGA programmable interconnect diagram (courtesy of Xilinx)

Briefly, we can code both the truth tables in the logic cell and the cables in between with interconnect.

The building block of FPGAs is the Look Up Table. There are no and, or gates in the FPGA. There is a Look Up Table logic. Combinational circuits and sum of minterm structures are realized with look up tables.




| a | b | c | x |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

*Figure 7. LUT Structure*

Any Boolean function can be implemented with a LUT. For example, a 3-input truth table was implemented with an 8-bit LUT. During FPGA configuration, the contents of the RAM in each LUT used are loaded as initial values according to the truth table of the function to be performed.

In terms of FPGA resource usage, the Boolean function does not matter, any function can be defined in the LUT. Function input signals serve as address bits of the RAM content.

Combinational circuits are realized with LUT.

# HDL (Hardware Description Language)

It should not be thought of as sequential C logic. It is a hardware design language. Each code written corresponds to a LUT or a flip-flop.

In the history of digital computers, various notations have been developed to capture the logical behavior of digital circuits at different levels of abstraction. Examples of such representations include boolean equations, timing charts, state transition tables, diagrams, and hardware description languages.

Hardware Description Language (HDL) is a programming language used to describe the structure, behavior, and timing of electronic circuits, most commonly digital logic circuits. HDLs are used to design processors, motherboards, CPUs and various other Digital circuits. In addition to their use in circuit design, HDLs serve the purpose of simulating the circuit and verifying its response. There are many HDLs available, but the most popular HDLs by far are Verilog and VHDL.

HDLs are similar to, but not the same as, a traditional programming language. These are specifically designed to identify hardware.

Unlike traditional programming languages that primarily represent serial operations, HDL distinguishes itself by representing extensive parallel operations[6].

Verilog language was preferred within the scope of this resource.

## Verilog HDL (Hardware Description Language)

Verilog is also a Hardware Description Language. It uses a textual format to describe electronic systems and circuits. In the field of electronic design, Verilog is used for testability analysis, error grading, logic synthesis, and verification through simulation of timing analysis.

Verilog is also more compact because the language is more of a true hardware modeling language. As a result, you typically write fewer lines of code, providing a comparison with the C language. However, Verilog has a superior grasp of hardware modeling and an inferior programming structure.

Design can be done in Verilog in three different ways.

- Gate-level modeling using instantiations of predifined and use-defined primitive gates.
- Dataflow modeling using continuous assignment statements with the keyword **assign**.
- Behavioral modeling using prosedural assignment statements with the keyword **always**.

### 1. Gate- and switch-level modeling (Structural Model)

It is a way of describing the syntax and semantics of the built-in primitives of gate and switch level modeling and how a hardware design can be defined using these primitives[7].

Verilog HDL has 14 predefined logic gates and 12 switches for ease of modeling at the gate and switch level. Modeling with logic gates and switches has the following advantages:

- Gates provide a much closer one-to-one mapping between the actual circuit and the model.
- There is no permanent assignment equivalent to a bidirectional transfer gate.

A gate or switch instance declaration shall begin with the keyword that specifies the gate or switch primitive being used by the instances that follow in the declaration.

| n_input gates | n_output gates | Three-state gates | Pull gates | MOS switches | Bidirectional switches |
|---|---|---|---|---|---|
| and | buf | bufif0 | pulldown | cmos | rtran |
| nand | not | bufif1 | pullup | nmos | rtranif0 |
| nor | | notif0 | | pmos | rtranif1 |
| or | | notif1 | | rcmos | tran |
| xnor | | | | rnmos | tranif0 |
| xor | | | | rpmos | tranif1 |

*Figure 8. The keywords that shall begin a gate or a switch instance declaration.*

Before moving on to examples, it is necessary to understand the circuit structure well. Circuits are divided into two: combinational and squential circuits.

Combinational Circuits: The output signals of the circuit depend only on the value of the current input signals. No memory in the circuit indicates state. As soon as the input signal changes, it affects the output signal.

Squential Circuits: The output signals of the circuit depend on the input signals and the memory value specified in the circuit state. The change of the input signal does not always affect the output signal, it depends on the state of the circuit.

## 2. Assignments (Dataflow Representation)

Assignments are the basic mechanism for placing values on networks and variables. There are two basic types of appointment:

- Continuous assignment assigns values to networks.
- Procedural assignment assigns values to variables.

The assignment consists of two parts: the left-hand side and the right-hand side, separated by the equals ( = ) character; or, in non-procedural continuous assignments, the less than-equal ( <= ) character pair is used.

The right side can be any expression that evaluates the value. The left side specifies the variable to which the value on the right side will be assigned. The left side can take one of the forms given in Fig 9., depending on whether the assignment is continuous or procedural[7].

| Statement type | Left-hand side |
|---|---|
| Continuous assignment | Net (vector or scalar)<br>Constant bit-select of a vector net<br>Constant part-select of a vector net<br>Constant indexed part-select of a vector net<br>Concatenation or nested concatenation of any of the above left-hand side |
| Procedural assignment | Variables (vector or scalar)<br>Bit-select of a vector reg, integer, or time variable<br>Constant part-select of a vector reg, integer, or time variable<br>Indexed part-select of a vector reg, integer, or time variable<br>Memory word<br>Concatenation or nested concatenation of any of the above left-hand side |

*Figure 9. Legal left-hand forms in assignment statements*

## Assign Keywords

| | |
|---|---|
| {} {{}} | Concatenation, replication |
| unary + unary - | Unary operators |
| + - * / ** | Arithmetic |
| % | Modulus |
| > >= < <= | Relational |
| ! | Logical negation |
| && | Logical and |
\|\| | Logical or |
| == | Logical equality |
| != | Logical inequality |
| === | Case equality |
| !== | Case inequality |
| ~ | Bitwise negation |
| & | Bitwise and |
\| | Bitwise inclusive or |
| ^ | Bitwise exclusive or |
| ^~ or ~^ | Bitwise equivalence |
| & | Reduction and |
| ~& | Reduction nand |

| | |
|---|---|
\| | Reduction or |
~\| | Reduction nor |
| ^ | Reduction xor |
| ~^ or ^~ | Reduction xnor |
| << | Logical left shift |
| >> | Logical right shift |
| <<< | Arithmetic left shift |
| >>> | Arithmetic right shift |
| ? : | Conditional |

*Figure 10. Assign Keywords*

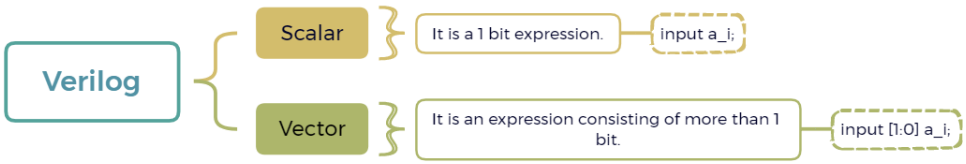



*Figure 11. Scalar and Vector*

## 3. Behavioral Modeling (Behavioral Representation)

The language constructs introduced so far allow hardware to be described at a relatively detailed level.

Modeling a circuit with logic gates and continuous assignments reflects quite closely the logic structure of the circuit being modeled; however, these constructs do not provide the power of abstraction necessary for describing complex high-level aspects of a system. The procedural constructs described in this clause are well suited to tackling problems such as describing a microprocessor or implementing complex timing checks.

It begins with a brief overview of a behavioral model to provide context for the many types of behavioral declarations in Verilog HDL.

Verilog behavioral models contain procedural statements that control the simulation and manipulate variables of the data types previously described. These statements are contained within procedures. Each procedure has an activity flow associated with it. The activity starts at the control constructs initial and always. Each initial construct and each always construct starts a separate activity flow. All of the activity flows are concurrent to model the inherent concurrence of hardware[7].

## Coding with Verilog Language

### 1. Half Adder Using Verilog

A half adder is a digital logic circuit that performs binary addition of two single-bit binary numbers. It has two inputs, A and B, and two outputs, sum and carry.

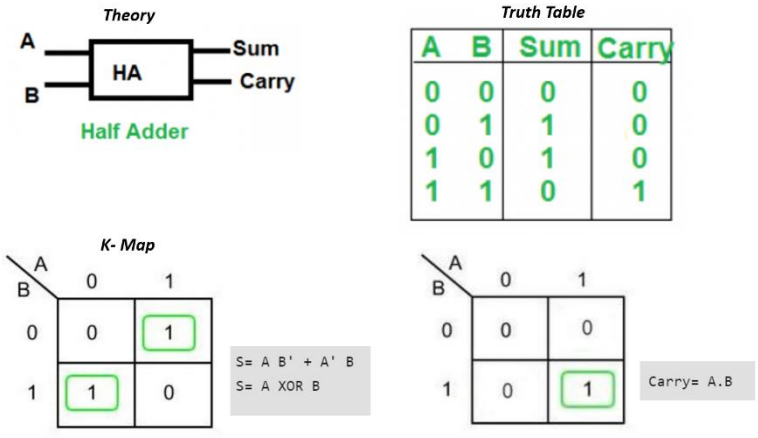



*Figure 12. (Half Adder) Sum is the XOR of inputs A&B, Carry is the AND of inputs A&B*

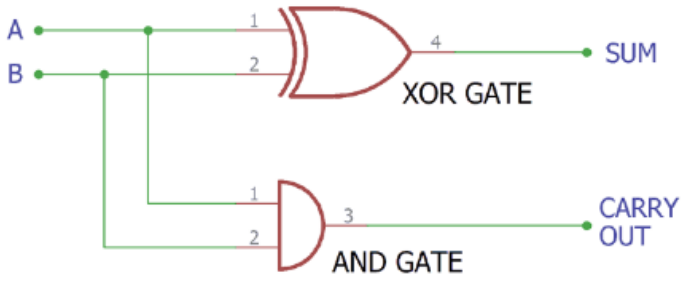


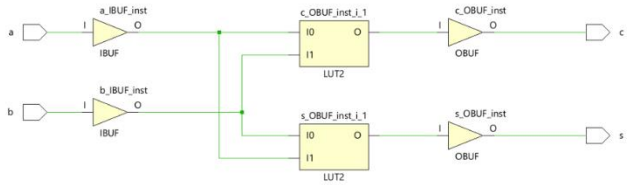

*Figure 13. Half Adder Implementation Using Logic Gates*

## Verilog Code

### Structural Method

```
module half_adder_structural(
input a,     // Input 'a'
input b,     // Input 'b'
output s,    // Output 's' (Sum)
output c     // Output 'c' (Carry)
);
xor gate_xor (s, a, b);  // XOR gate for sum
and gate_and (c, a, b);  // AND gate for carry

endmodule
```

### Schematic




### Utilization

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1 | 41000 | 0.00 |
| IO | 4 | 300 | 1.33 |

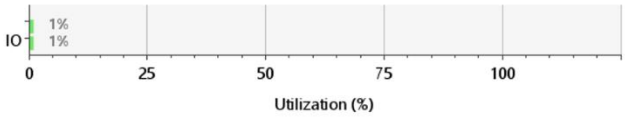


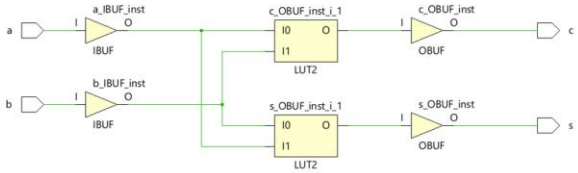

*Figure 14. Half Adder Verilog Structural Method*

## Verilog Code

### Dataflow Representation

```
module half_adder_dataflow(
  input a,     // Input 'a'
  input b,     // Input 'b'
  output s,    // Output 's' (Sum)
  output c     // Output 'c' (Carry)
    );
  assign s = a ^ b;  // Dataflow expression for sum
  assign c = a & b;  // Dataflow expression for carry
endmodule
```

### Schematic




### Utilization

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1 | 41000 | 0.00 |
| IO | 4 | 300 | 1.33 |

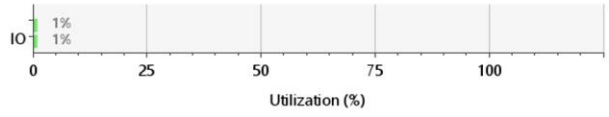


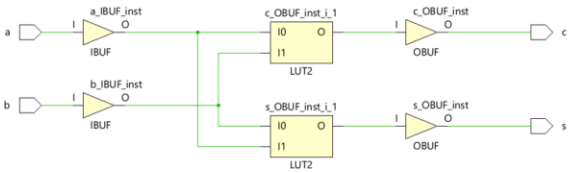

*Figure 15. Half Adder Verilog Dataflow Representation*

**Verilog Code**

Behavioral Representation

```
module half_adder_behavioral(
  input a,    // Input 'a'
  input b,    // Input 'b'
  output s,   // Output 's' (Sum)
  output c    // Output 'c' (Carry)
   );
    reg sum, carry;
    // Combinational logic equations for sum and carry
  always @(*) begin
    sum = a ^ b;  // XOR operation for sum
    carry = a & b;  // AND operation for carry
  end
  assign s = sum;
  assign c = carry;
endmodule
```

Schematic




Utilization

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1 | 41000 | 0.00 |
| IO | 4 | 300 | 1.33 |

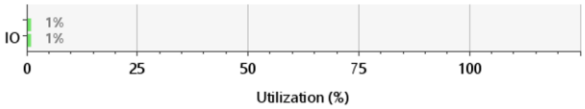
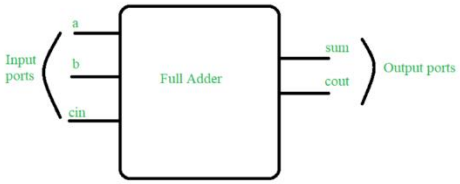



*Figure 16. Half Adder Verilog Behavioral Representation*

## 2. Full Adder Using Verilog

It takes two single-digit numbers plus an overflow input and calculates their sum. It is often used in digital computing and data processing systems. It is a fundamental building block for performing complex numerical operations. Computers, calculators, and digital signal processing systems include full adders.

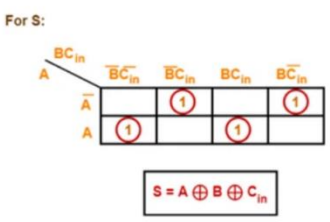
***Theory***




***Truth Table***

| INPUTS | | | OUTPUTS | |
|---|---|---|---|---|
| A | B | $C_{in}$ | SUM | CARRY$_{OUT}$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

***K- Map***



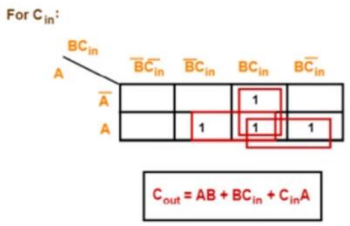




*Figure 17. Full Adder truth table and K-Map*

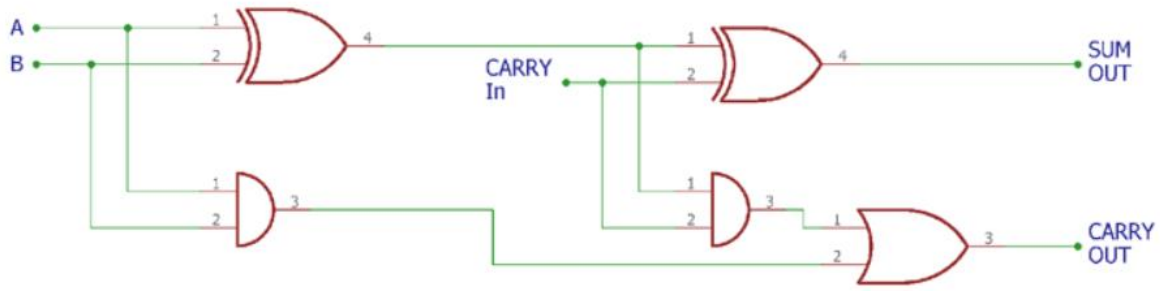


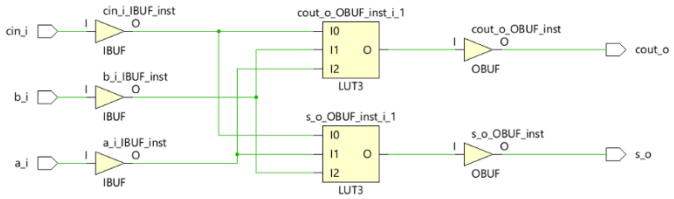

*Figure 18. Full Adder Implementation Using Logic Gates*

## Verilog Code

### Structural Method

```
module full_adder_struct(
    input a_i,
    input b_i,
    input cin_i,
    output s_o,
    output cout_o
    );
    wire xor1, and1, and2;
    xor G1(xor1, a_i, b_i);
    and G2(and1, a_i, b_i);
    xor G3(s_o, xor1, cin_i);
    and G4(and2, xor1, cin_i);
    or G5(cout_o, and2, and1);
endmodule
```

### Schematic




### Utilization

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1 | 41000 | 0.00 |
| IO | 5 | 300 | 1.67 |

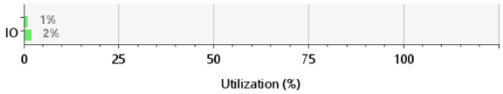



*Figure 19. Full Adder Verilog Structural Method*

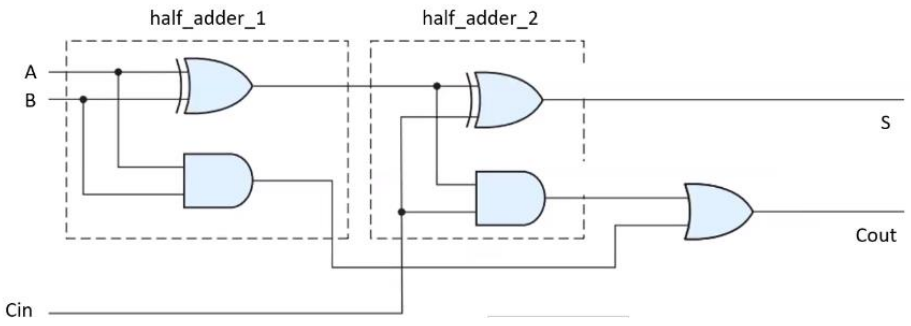



*Figure 20. Full Adder Design with Hierarchical Design*

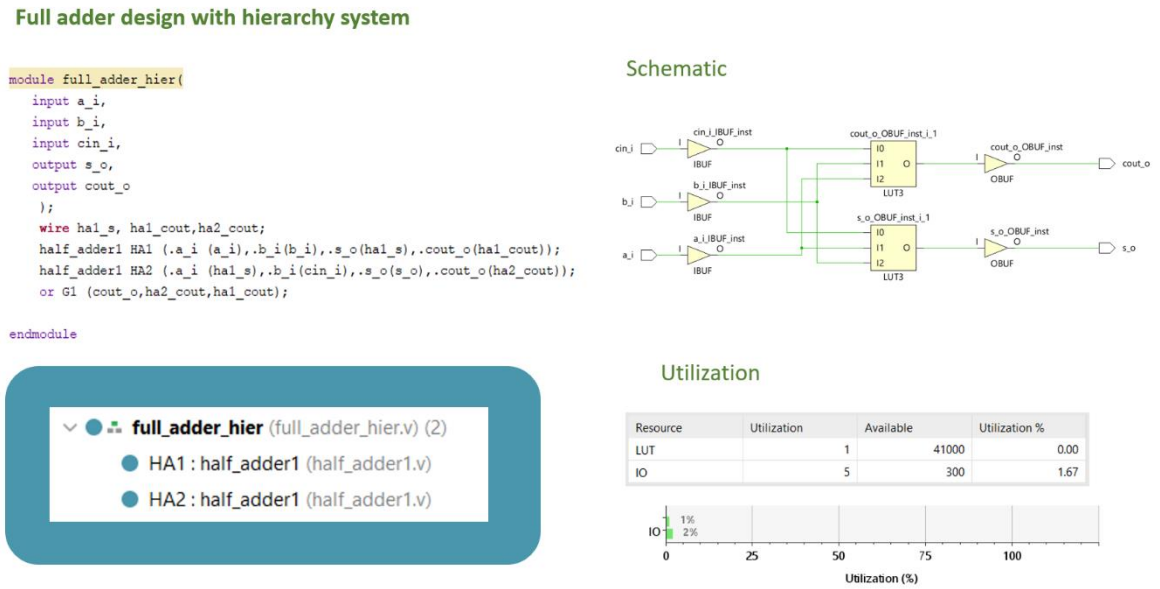
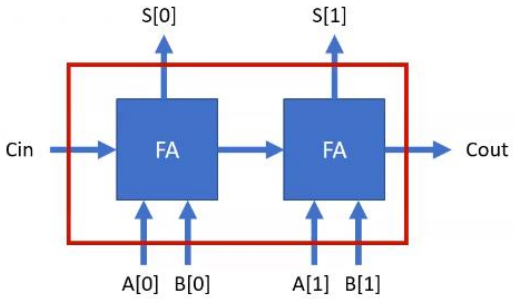



*Figure 21. Defining a full adder using half adder*

Using a module inside another module is called **instantiation**. To use a module in a design, this is done by calling the name of that module and providing the necessary connections. This is used to support hierarchical design and is useful for breaking complex designs into smaller, manageable modules. Connecting the modules is a crucial step. The Verilog program may not give an error, but our code may not work correctly due to our logical error. When we connect both modules, when we look at the code example in Figure 20, the definition .a_i refers to a term defined in the half-adder module. The term in parentheses (a_i) refers to the input in that program module[8].

## 3. 2-bit Binary Adder

In the examples, all scalar, i.e. one-bit, operations were performed. But now let's examine how operations with vector input are performed.



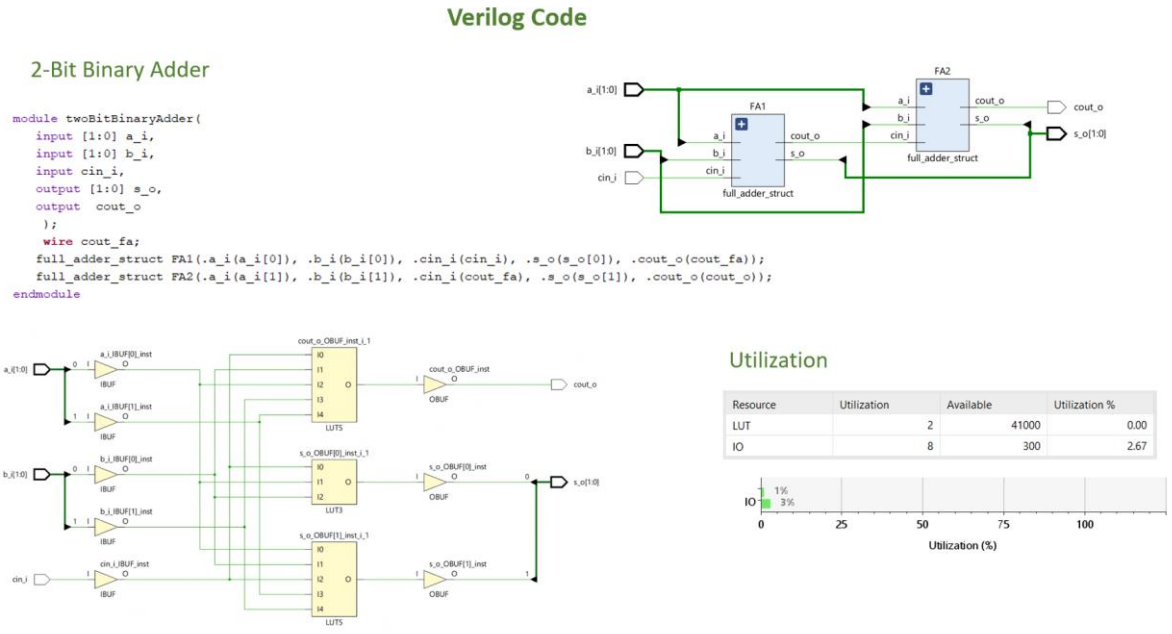



*Figure 22. Verilog Code of 2-Bit Binary Adder*

## Verilog Testbench

Used to test the correctness of a design, testbench is another Verilog module used to simulate and test the design module. A testbench provides data appropriate to the inputs of the design module, allowing us to verify by observing and checking the outputs. The purpose of a testbench is to provide a way to simulate the behavior of the design under various conditions, inputs, and scenarios before fabricating the physical hardware. It allows designers to catch bugs, validate functionality, and optimize designs without the cost and time associated with physical prototyping[9].

In Testbench, input signals must be defined as **reg** and output signals must be defined as **wire**.

**The Wire** is used to connect different elements. It can also be referred to as physical wires. It can be read or assigned. No values are stored in it. It must be routed through a continuous assignment statement or a module's port.

**The Reg** represents data storage elements in Verilog / SystemVerilog. It retains its value until the next value is assigned to it[10].

Testbenches are used only for simulation purposes and not for synthesis. Hence the full range of Verilog constructs like **initial** and system tasks clike **$display** an be used to help with simulation and debugging.

**Verilog Code**

Testbench Module of 2-Bit Binary Adder

```
module tb_twoBitBinaryAdder();
    reg [1:0] a_i, b_i;
    reg cin_i;    //define input of DUT reg
    wire [1:0] s_o; //define output of DUT wire
    wire cout_o;

twoBitBinaryAdder DUT //design Unit Test
(
.a_i     (a_i),
.b_i     (b_i),
.cin_i   (cin_i),
.s_o     (s_o),
.cout_o  (cout_o));
```

```
initial begin
a_i    =2'b00;
b_i    =2'b00;
cin_i  =1'b0;
#10;
a_i    =2'b01;
b_i    =2'b10;
cin_i  =1'b1;
#10;
a_i    =2'b11;
b_i    =2'b11;
cin_i  =1'b1;
#10;
a_i    =2'b11;
b_i    =2'b11;
cin_i  =1'b0;
#10;
a_i    =2'b01;
b_i    =2'b01;
cin_i  =1'b1;
#10;
$finish
end

endmodule
```

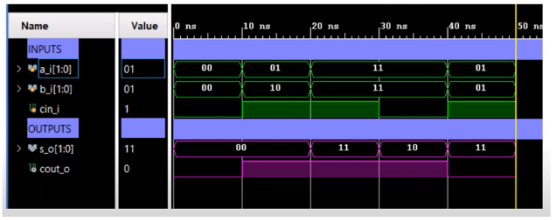
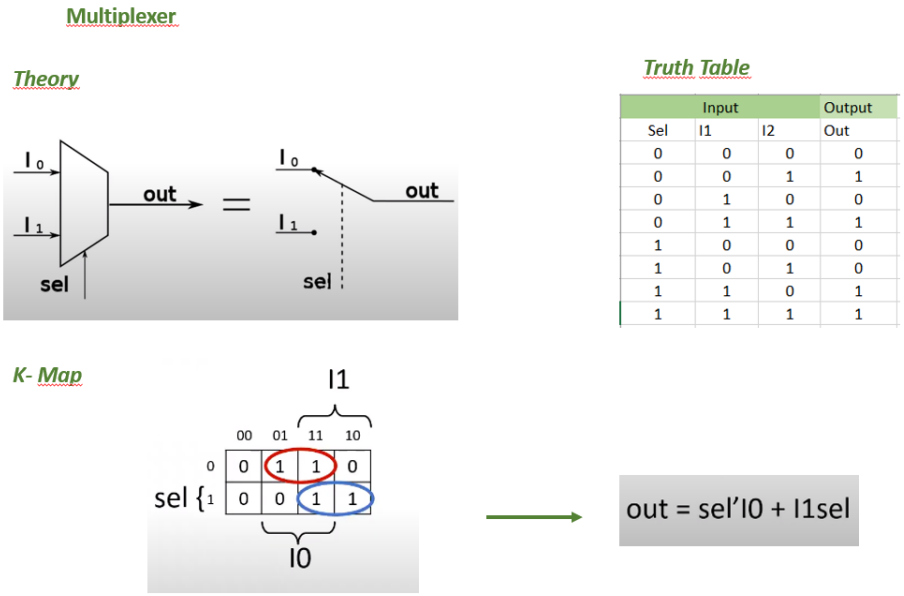



*Figure 23. Testbench Module of 2-bit Binary Adder*

## 4. Multiplexer

A multiplexer is a combinational type of digital circuit that transfers one of the available input lines to a single output and, which input has to be transferred to the output will be decided by the state(logic 0 or logic 1) of the select line signal. 2:1 Multiplexer has two inputs, one select line (to select one of the two inputs), and a single output.




| | Input | | Output |
|---|---|---|---|
| Sel | I1 | I2 | Out |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

*Figure 24. Mux Representation*

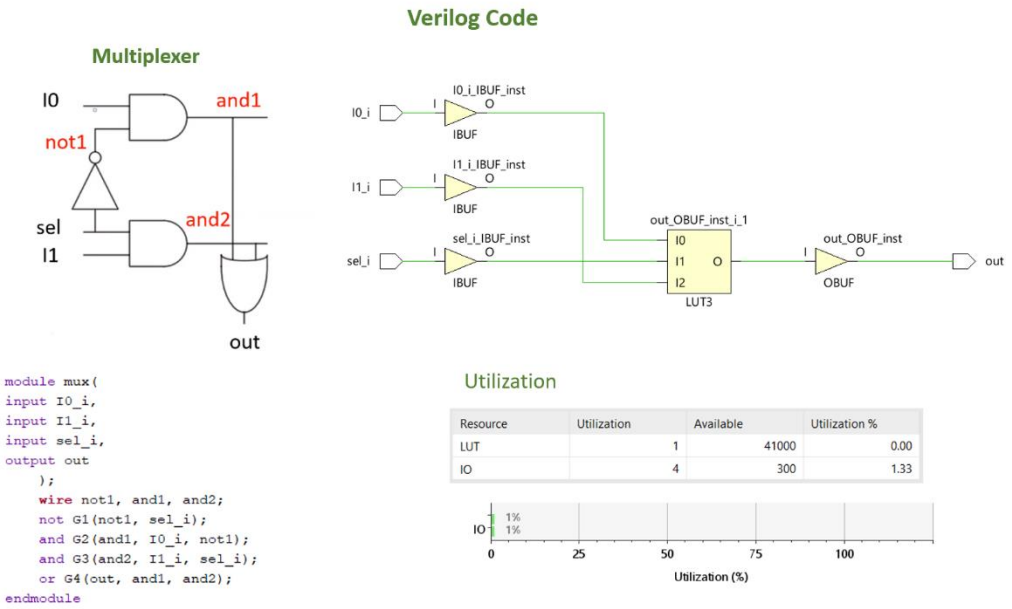



*Figure 25. Verilog Code of Multiplexer*

**Assign:** As its name suggests, it is the keyword that connects a circuit to a wire and, in its simplest form, provides a short circuit between two ends. Using it for any memory structure is inappropriate, so circuits that drive wire-type structures are implemented with assign. **assign** is usually written as a single line.

$$\textbf{assign y} = \textbf{a} \,\&\, ((\textbf{b} \mid \textbf{c}) \,\hat{}\, (\sim\textbf{d})); \text{ // Here y must be wire}$$

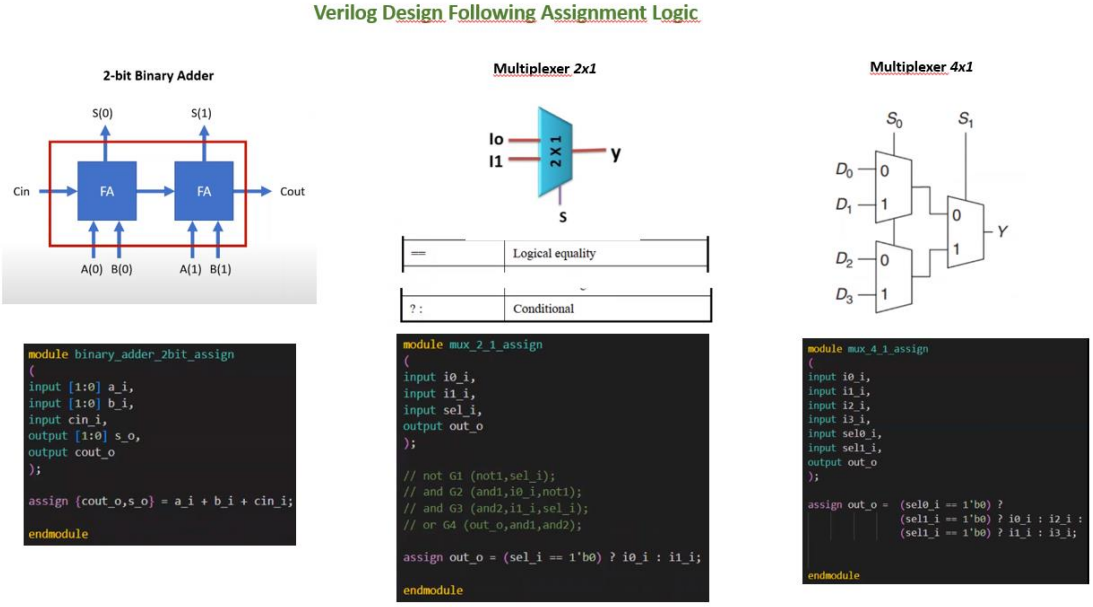


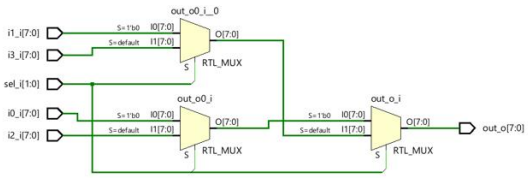

*Figure 26. Assign Code Structure Demonstration with Examples*

As shown in Fig.26, all three applications are in scalar input. When we examine how the process works when there is vector input,

**Multiplexer *4x1 8 Bit***

```
module mux_4_1_8bit_assign(
    input  [7:0] i0_i,
    input  [7:0] i1_i,
    input  [7:0] i2_i,
    input  [7:0] i3_i,
    input  [1:0] sel_i,
    output [7:0] out_o
    );
    assign out_o = (sel_i[0] == 1'b0) ?
                   (sel_i[1] == 1'b0) ? i0_i : i2_i:
                   (sel_i[1] == 1'b0) ? i1_i : i3_i;
endmodule
```

**Vivado Elaberation**




```
module mux_4_1_8bit_assign
(
input [7:0] i0_i,
input [7:0] i1_i,
input [7:0] i2_i,
input [7:0] i3_i,
input [1:0] sel_i,
output [7:0] out_o
);

//assign out_o =  (sel_i[0] == 1'b0) ?
//                (sel_i[1] == 1'b0) ? i0_i : i2_i :
//                (sel_i[1] == 1'b0) ? i1_i : i3_i;

assign out_o =  (sel_i == 2'b00) ? i0_i :
                (sel_i == 2'b01) ? i1_i :
                (sel_i == 2'b10) ? i2_i :
                                   i3_i;

endmodule
```

**Utilization**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 8 | 41000 | 0.02 |
| IO | 42 | 300 | 14.00 |

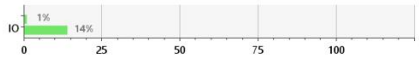


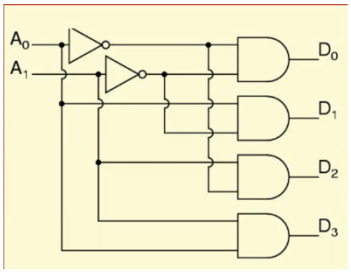

*Figure 27. Assign Code Structure Demonstration with Vector Input*

## 5. Decoder

A decoder is a combinational logic circuit that has ‘n’ input signal lines and 2n output lines. In the 2:4 decoder, we have 2 input lines and 4 output lines. In addition, we provide ‘enable‘ to the input to ensure the decoder is functioning whenever enable is 1 and it is turned off when enable is 0.

**Verilog Code**

***Decoder***




```
module decoder_2_4
(
input [1:0] a_i,
output [3:0] d_o
);


assign d_o =    (a_i == 2'b00) ? 4'b0001 :
                (a_i == 2'b01) ? 4'b0010 :
                (a_i == 2'b10) ? 4'b0100 :
                                 4'b1000;


endmodule
```

***Truth Table***

| Input | | Output | | | |
|---|---|---|---|---|---|
| A0 | A1 | D0 | D1 | D2 | D3 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

***Testbench***

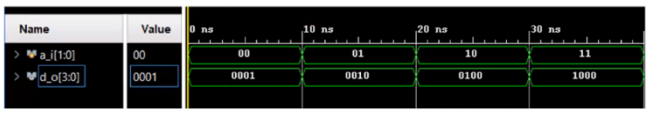



*Figure 28. Verilog Code of Decoder*

Always blocks are used to assign signals that need memory.

The operation of always blocks can be explained as follows, based on an example always code:

```
always @ (x,y) begin
t = a & b; //Here t must be of type reg.
z = x | t; //Here z must be of type reg.
end
```

The always keyword indicates that this circuit should always work. The @(…) expression following it explains the condition under which it will run. The signal list written in parentheses here is called the sensitivity list. Circuits are defined inside an always block written in this way

It works when any of the signals in the sensitivity list changes, other than these, any input changes do not change the outputs. So we can say always @ (x,y) as follows: Execute the assignments in this block when there is a change in any of the x or y signals. The word begin, which comes after it, is used in pairs with the word end, which comes after. Instead of curly brace pairs ({}) used to specify code blocks in high-level programming languages such as C and JAVA, the begin-end pair is used in Verilog HDL[11].

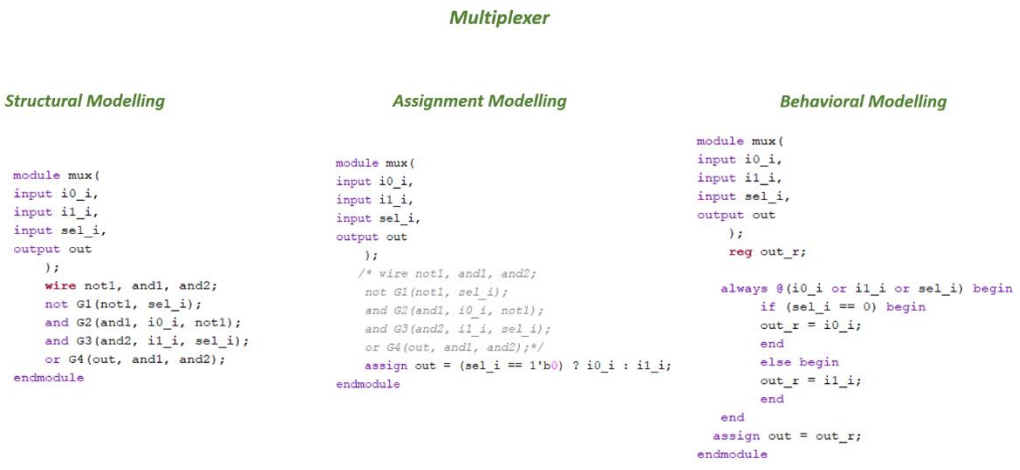



*Figure 29. Comparison of all Models and Demonstration of Verilog Codes*

## 6. 8 bit Adder/ Subtractor

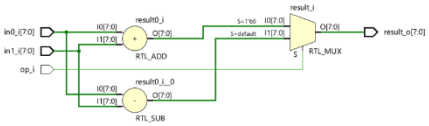
**8 bit Adder / Subtractor Verilog Code**

```
module add_sub_8bit(
    input [7:0] in0_i,
    input [7:0] in1_i,
    input op_i,
    output [7:0] result_o
    );
    reg [7:0] result;
    always @(in0_i, in1_i, op_i)begin
       if (op_i == 1'b0)begin
       result = in0_i + in1_i;
       end
       else begin
       result = in0_i - in1_i;
       end
    end
    assign result_o = result;
endmodule
```

**Testbench**

```
module add_sub_tb();
    reg [7:0] in0_i;
    reg [7:0] in1_i;
    reg op_i;
    wire [7:0] result_o;

    add_sub_8bit DUT(.in0_i(in0_i),
                     .in1_i(in1_i),
                     .op_i(op_i),
                     .result_o(result_o));
```

```
initial begin
in0_i = 8'd10;
in1_i = 8'd55;
op_i = 1'b0;
#10;
in0_i = 8'd10;
in1_i = 8'd55;
op_i = 1'b1;
#10;
in0_i = 8'd55;
in1_i = 8'd10;
op_i = 1'b1;
#10;
in0_i = 8'd100;
in1_i = 8'd55;
op_i = 1'b0;
#10;
in0_i = 8'd100;
in1_i = 8'd55;
op_i = 1'b1;
#10;
in0_i = 8'd55;
in1_i = 8'd100;
op_i = 1'b1;
#10;
$finish;
end
endmodule
```



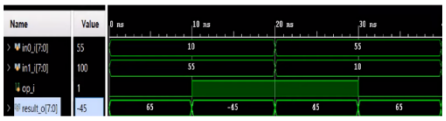




*Figure 29. Comparison of all Models and Demonstration of Verilog Codes*

If we repeat the definition of sequential circuits to remember, they are circuits whose output signal depends on both the current value and the past value of the input signal. They contain a memory unit.

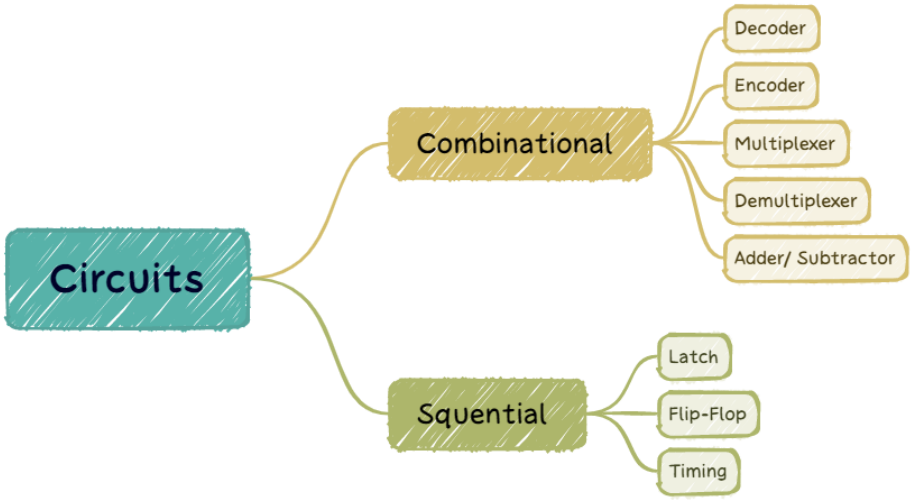



*Figure 30. Representation of circuits with Xmind Map*

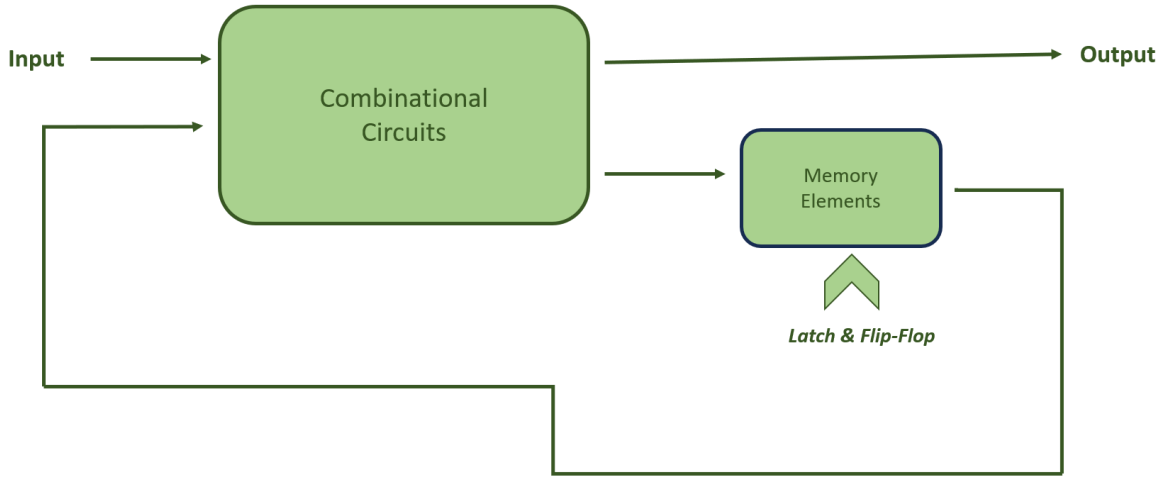



*Figure 31. Sequential Circuit Representation Drawing*

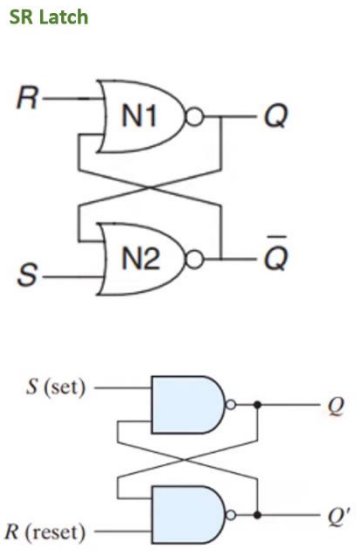
The latch is an undesirable structure in sequential circuits and usually gives errors.

## 7. SR (Set / Reset) Latch

Before starting with the S-R latch you need to know what a latch is. A latch is an asynchronous circuit (it doesn't require a clock signal to work), and it has two stable states, HIGH ("1") and LOW ("0"), that can be used for storing binary data. Many sequential circuits and larger storage devices, such as shift registers, use latches as their principal building block.

The simplest latch is the Set-Reset (S-R) latch. You can build one by connecting two NOR gates with a cross-feedback loop[12].

The feedback path is crucial to storing one bit of data as long as the circuit is powered. In this circuit, the upper gate has the S input and the main output Q, while the lower gate has the R input and the inverted output $\overline{Q}$.




| Case | S | R | Q | $\overline{Q}$ |
|---|---|---|---|---|
| IV | 0 | 0 | $Q_{prev}$ | $\overline{Q}_{prev}$ |
| I | 0 | 1 | 0 | 1 |
| II | 1 | 0 | 1 | 0 |
| III | 1 | 1 | 0 | 0 |

*Figure 32. The SR Latch Representation*

The biggest problem with SR Latch is that an undesirable situation occurs when S and R are 1 at the same time. Its design should be arranged so that S and R are never '1' at the same time. However, this may not be possible or may make the design difficult. For this reason, the best solution is to change the latch type and apply D-Latch[13].

## 8. D Latch

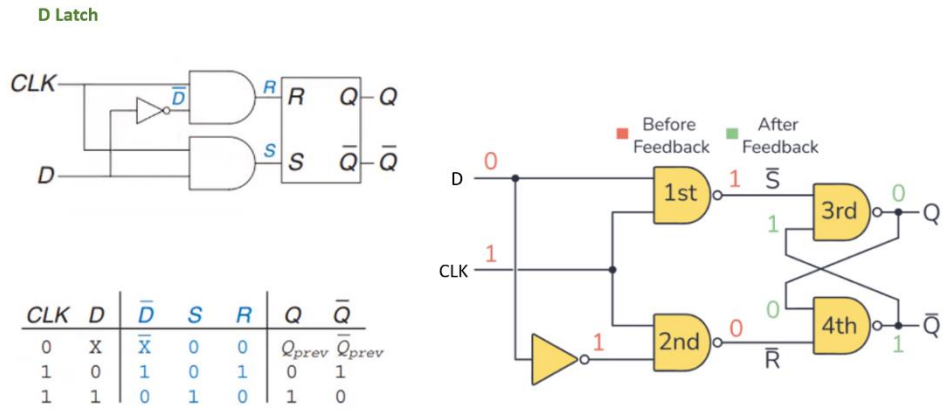
The D Latch is a logic circuit most frequently used for storing data in digital systems. It is based on the S-R latch, but it doesn't have an "undefined" or "invalid" state problem[14]. A D latch can store a bit value, either 1 or 0. When its Enable pin is HIGH, the value on the D pin will be stored on the Q output. It builds upon the design of the S-R latch, with a few added logic gates.

The inverter on the input makes sure the S and the R inputs are always opposites, to avoid the invalid state of both being 1.

When the CLK signal is '1', the Q value is equal to D. When the CLK signal is 'O', S and R become 'O' and the Q value maintains its previous state, that is, its value.

If the CLK signal is '1', the value D is assigned to the Q signal. Making assignments only at certain "moments" will be easier in terms of timing analysis and synchronization of the system.

We use Flip-Flops to assign values at certain "moments"[13].




| CLK | D | $\overline{D}$ | S | R | Q | $\overline{Q}$ |
|---|---|---|---|---|---|---|
| 0 | X | $\overline{X}$ | 0 | 0 | $Q_{prev}$ | $\overline{Q}_{prev}$ |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 |

*Figure 33. The D Latch Representation*

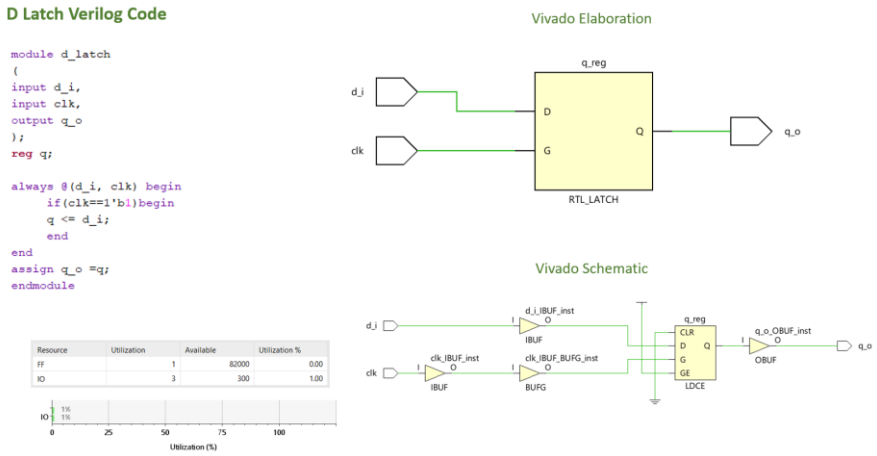



*Figure 34. Verilog Code of D Latch*

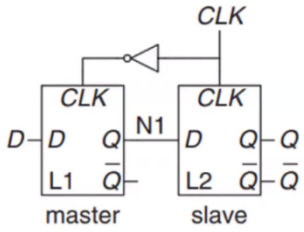
- BUFG, seen in Fig.33., is a large buffer within the FBGA. CLK signals must reach all flip-flops in the circuit.
- When we use Latch, vivado gives a warning in this case. This warning is called avoid unintended latch.

They usually work as **Latch: Level-sensitive** and **Flip-Flop: Edge-sensitive**. On closer inspection, in a sequential circuit, the output changes depending on the trigger. There are two types of triggering: edge and level triggering. There are two levels in the clock pulse or signal. One is high voltage (VH) and the other is low voltage (VL). Additionally, these voltage levels help determine the trigger type[15].

Edge Triggering (Edge-sensitive): In a sequential circuit, if the output changes when the signal transits from a high level to a low level or from a low level to a high level, we call it edge triggering. Here, the edge that changes the voltage from low level to the high level is called rising edge (positive edge). And, the edge that changes the voltage from high level to the low level is called falling edge (negative edge). Thus, when an event is triggered at the rising edge or falling edge, we call it edge triggering. For example, assume lighting an LED according to the edge triggering. In this scenario, the LED lights on every time the signal transits from low voltage to high voltage. Taking some examples; S-R flip flop, J-K flip flop and D flip flop are some common examples for flip flops with edge triggering[15].

Level Triggering (Level-sensitive): In the sequential circuit, if the output changes during the high voltage period or low voltage period, it is called level triggering. In other words, the output changes during either high voltage or low voltage period- not during the edges like in edge triggering. Thus, when an event triggers at the clock level, we call it level triggering. Assume lighting an LED according to level triggering. LED can turn on at any time during the high voltage. In other words, the event is triggered whenever a clock level is encountered. Considering examples; SR latch and D latch are some examples for latches with level triggering[15].
In brief, there are two types of triggering in sequential circuits. The triggering results can change the output of the circuit. The main difference between edge and level triggering is that, in edge triggering, the output of the sequential circuit changes during the high voltage period or low voltage period while in level triggering, the output of the sequential circuit changes during transits from the high voltage to low voltage or low voltage to high voltage.

**D Flip-Flop**



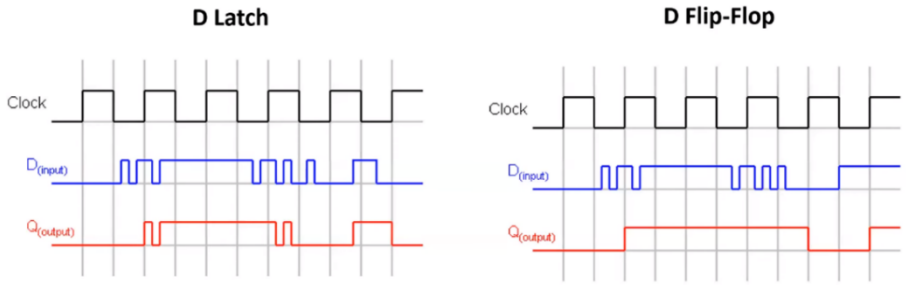
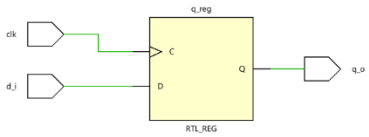



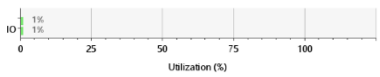

*Figure 35. Demonstration of D Flip-Flop and D Latch Trigger*

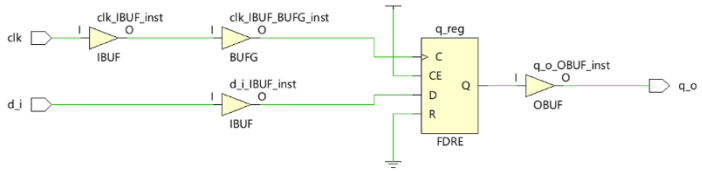
**D Flip-Flop Verilog Code**

```
module d_fflop
(
input d_i,
input clk,
output q_o
 );
 reg q;
 always @(posedge clk)begin
 q <= d_i;
 end
 assign q_o = q;
endmodule
```




| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| FF | 1 | 82000 | 0.00 |
| IO | 3 | 300 | 1.00 |



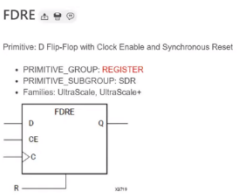




*Figure 36. Verilog code of D Flip-Flop*

According to Fig.35, when we give power to the digital circuit, we do not know what the initial state of this flip-flop is. However initial states are important because other states can be determined based on the initial state. This situation of defining the initial state is called initial value or reset[16].

**D Flip-Flop Active Low Reset**

```
module d_fflop
(
input d_i,
input clk,
input rst_n, //active-low reset
output q_o
 );
 reg q;
 always @(posedge clk)begin
    if (rst_n == 1'b0)begin
       q <= 1'b0;
    end
    else begin
       q <= d_i;
    end
 end
 assign q_o = q;
endmodule
```

**D Flip-Flop Active High Reset**

```
module d_fflop
(
input d_i,
input clk,
input rst, //active-high reset
output q_o
 );
 reg q;
 always @(posedge clk)begin
    if (rst == 1'b1)begin
       q <= 1'b0;
    end
    else begin
       q <= d_i;
    end
 end
 assign q_o = q;
endmodule
```

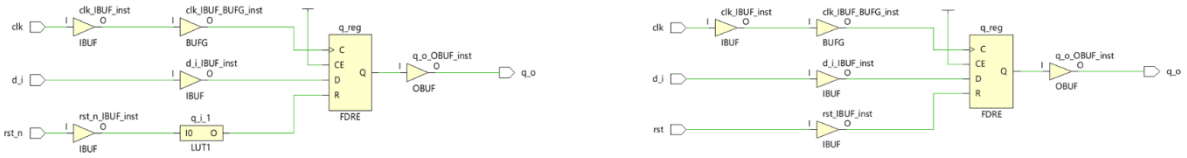



*Figure 37. Representation of Verilog code*

Performance is another critical aspect of circuit design. Synchronous reset typically offers better performance compared to its asynchronous counterpart. By utilizing the clock signal, synchronous reset ensures that the reset operation occurs at well-defined and predictable points in time. This synchronization facilitates tighter control over timing constraints, allowing for faster and more efficient circuit operation. On the other hand, asynchronous reset can introduce performance challenges due to its lack of synchronization with the clock signal. The reset operation can occur at any point, potentially disrupting the normal flow of circuit operation. Designers must carefully consider timing constraints and trade-offs when implementing asynchronous reset, ensuring the circuit meets the required performance specifications.

Overall, stability analysis and performance evaluation play vital roles in circuit design. Understanding the impact of synchronous reset and asynchronous reset on stability and performance allows designers to make informed decisions and create reliable and efficient circuits[17], [18].

**D Flip-Flop Asynchronous Verilog Code**

```
module d_fflop
(
input d_i,
input clk,
input rst_n, //active-low reset
output q_o
 );
 reg q;
 //always @(posedge clk) begin Synchronous reset
 always @(posedge clk, negedge rst_n)begin  //Asynchronous reset
    if (rst_n == 1'b0)begin
       q <= 1'b0;
    end
    else begin
       q <= d_i;
    end
 end
 assign q_o = q;
endmodule
```

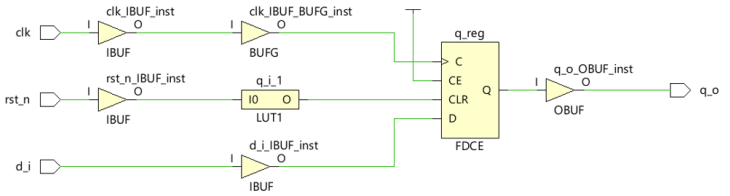
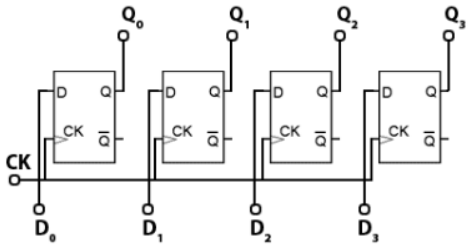



*Figure 38. Representation of Asynchronous Verilog Code*

## 9. Register

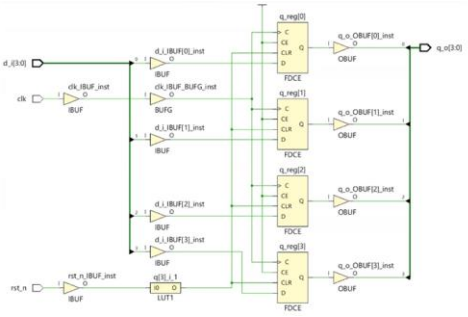



They are digital circuits consisting of grouped flip-flops and fed with the same clock signal.

**Register Verilog Code**

```
module reg4(
    input [3:0] d_i,
    input rst_n,
    input clk,
    input [3:0] q_o
    );
    reg [3:0] q;
    always @(posedge clk or negedge rst_n) begin
        if (rst_n == 1'b0)begin
           q <= 4'b0000;
        end
        else begin
           q <= d_i;
        end
    end
    assign q_o = q;
endmodule
```

```
module reg_param #(parameter N = 32)
(
input [N-1:0] d_i,
input rst_n,
input clk,
output [N-1:0] q_o
);
reg [N-1:0] q;
always @(posedge clk or negedge rst_n) begin
    if (rst_n == 1'b0) begin
    q <= {N{1'b0}};
    end
    else begin
    q <= d_i;
    end
end

assign q_o = q;
endmodule
```



| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1 | 41000 | 0.00 |
| FF | 32 | 82000 | 0.04 |
| IO | 66 | 300 | 22.00 |

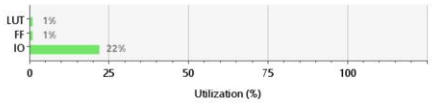



*Figure 39. Representation of Register Verilog Code*

Parameters are Verilog constructs that allow a module to be reused with a different specification. For example, a 4-bit adder can be parameterized to accept a value for the number of bits and new parameter values can be passed in during module instantiation. So, an N-bit adder can become a 4-bit, 8-bit or 16-bit adder. They are like arguments to a function that are passed in during a function call[19].

### Synchronous and Asynchronous Sequential Circuits

Asynchronous circuits do not have the clk signal and are faster. It consists of inputs at the gate level, but as you know, these gates consist of transistors. Asynchronous circuits, no matter how fast they are, are temperature dependent because transistors have a temperature threshold. It is a promising topic and academic studies are still ongoing.

Synchronous sequential circuits consist of combinational and sequential blocks. Combined circuits are circuits that react instantly to changes in partial input signals and calculate the result signals.

The sequential circuit part is a block that is synchronized with a clock signal, usually consisting of Flip-Flops and represents the current state of the circuit.

FlipFlops transmits the input signal value to the output signal on the rising or falling edge of the clock pulse.

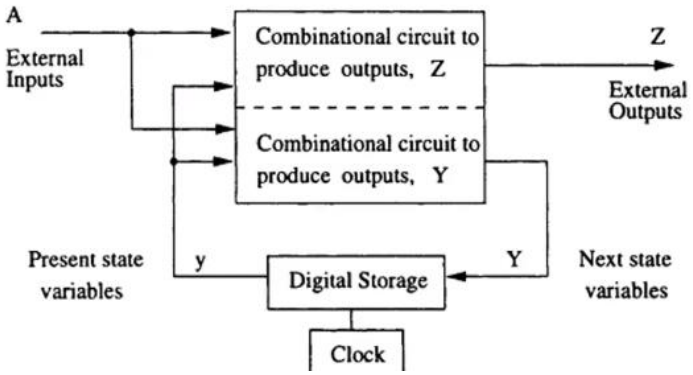
The rules of synchronous sequential circuit composition teach us that a circuit is a synchronous sequential circuit if it consists of interconnected circuit elements, such that

* Every circuit element is either a register or a combinational circuit

* At least one circuit element is a register

* All registers receive the same clock signal

* Every cyclic path contains at least one register[20].




## Finite State Machines

The finite state machine (FSM) is a software design pattern where a given model transitions to other behavioral states through external input.

Sequential logic systems finite state machines (FSMs). As FSMs, they consist of a set of states, some inputs, some outputs, and a set of rules for transitioning from state to state. When designing a digital system, it is very common to start by describing how the system works with a finite state machine model. This design step allows the designer to think about the design from a high-level perspective without having to think too much about what type of hardware the system will be implemented on or what design tools will be required to implement the design. Once the FSM is fully designed, if it is well designed, it is easy to write the design in a hardware description language (such as Verilog or VHDL) to implement it in a digital IC (integrated circuit)[21].

One of the most important issues in synchronous sequential circuits is the Finite state machine. The reason why it is called a finite state machine is that up to $2^n$ states can be created with n flip-flops.

Moore and Mealy machines are two common types of FSMs, and their main differences relate to their state outputs.

Moore Machine: In a Moore machine, there is an output associated with the state itself. That is, each of the states is assigned a fixed output. Transitions between states depend only on the inputs, and when the state changes the outputs change independently of the state.

Mealy Machine: In the Mealy machine, inputs as well as states determine outputs. That is, the outputs change depending on the combination of state and input.

Transitions between states still depend on inputs, but outputs vary depending on combinations of states and inputs.

To understand their basic differences, in Moore machines the outputs depend only on the state, whereas in Mealy machines they depend on both state and inputs. Both can be used in different applications and are preferred depending on the design requirements.

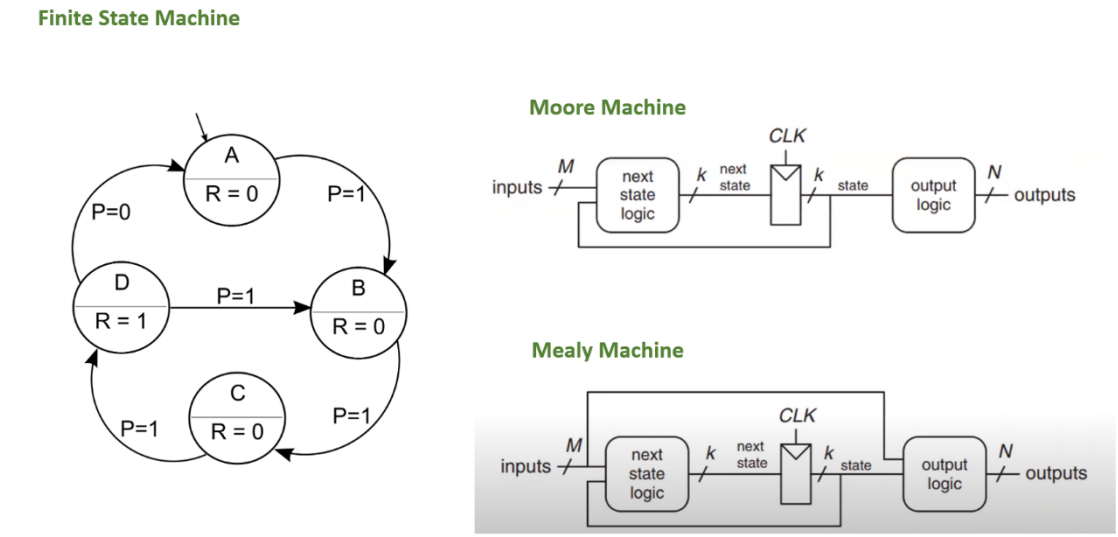



*Figure 40. Representation of Finite Machine and Its Types*

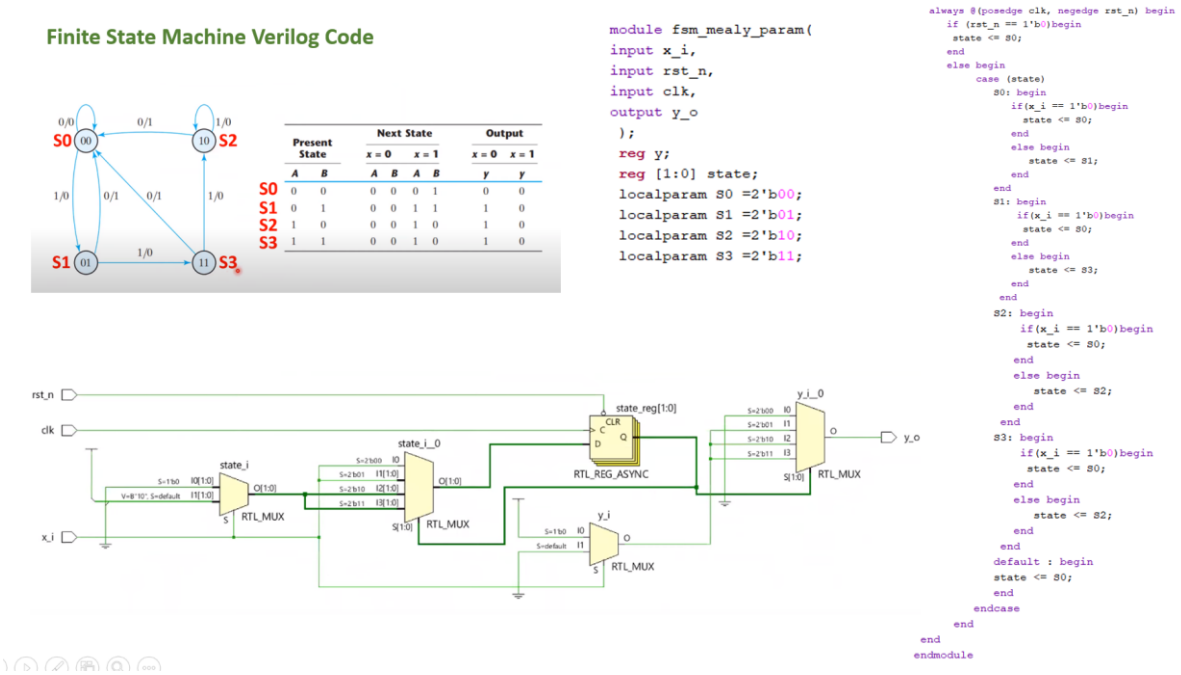



*Figure 41. Finite State Machine Verilog Code*

## Paramater:

* `Parameter` is typically used for defining constant values that are utilized inside or outside the module.

* A `parameter` defined within a module can be altered during the design of the module.

* `parameter` definitions are accessible both inside and outside the module.

* The value of a `parameter` remains the same across all instances of the module.

* `Parameter` values are commonly used for defining design parameters (such as sizes, delay times, constants, etc.) or general constant values (such as presets, defaults, etc.).

Localparam:

* Localparam is used only for literals defined inside the module.

*Localparam values cannot be changed outside of the module.

*Localparam definitions are only accessible within the module.

*Localparam values are often used within the design for temporary literals or calculated literals (e.g. widths, custom constants, etc.).

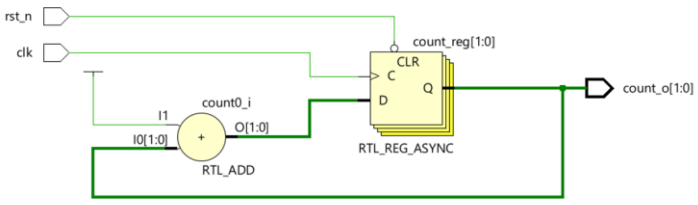
- 'Parameter' can be accessed inside and outside the module, while 'localparam' can only be accessed inside the module.
- 'Parameter' has the same value for all instances of the module, while 'localparam' has a constant value and is valid for all instances of the module.
- 'Parameter' is used to define design parameters and global constant values, while 'localparam' is generally used for temporary constants or calculated constant values within the design.

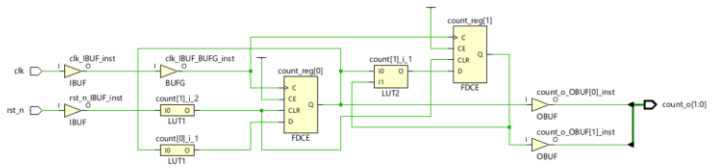
## 10. Counter

Counters are fundamental components in digital circuits that keep track of the number of clock cycles. They can be used to divide the frequency of a clock, generate timing signals, and count events in a system.

**Counter**

```
module counter_2bit(
input clk,
input rst_n,
output [1:0] count_o
);
reg [1:0]count;

always @(posedge clk or negedge rst_n)begin
     if (rst_n ==1'b0) begin
     count <= 2'b00;
     end
     else begin
     count <= count +1;
     end
end
assign count_o = count;
endmodule
```



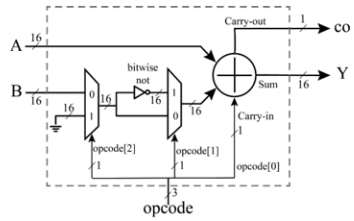

**Counter verilog code with 'parameter'**

```
module counter_Nbit#(parameter N = 16)
(
input clk,
input rst_n,
output [N-1:0] count_o
);
reg [N-1:0] count;
 always @(posedge clk, negedge rst_n)begin
      if(rst_n == 1'b0) begin
      count <= {N{1'b0}};
      end
      else begin
      count <= count +1;
      end
 end
 assign count_o = count;
endmodule
```

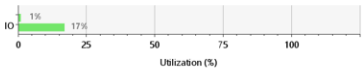
*Figure 42. Counter Verilog Code*

## 11. Integrated Systems Architecture Exercise

```
module ese01(
input [15:0]A,
input [15:0]B,
input [2:0]opcode,
output [15:0] Y,
output co);
 wire [15:0]X;
 reg [15:0]mux1;
 reg [15:0]mux2;
 always @(B or opcode)begin
  if (opcode[2] == 1'b0) begin
    mux1 <= B;
  end else begin
    mux1 <= X;
  end
  if (opcode [1] == 1'b1)begin
    mux2 <= ~mux1;
  end else begin
    mux2 <= mux1;
  end
 end

 assign {co, Y} = A + mux2 + opcode[0];

endmodule
```




| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 17 | 41000 | 0.04 |
| IO | 52 | 300 | 17.33 |




| Name | Slack | Levels | Routes | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 8 | 5 | 1 | B[0] | co | 5.578 | 4.164 | 1.414 | ∞ | input port clock |
| Path 2 | ∞ | 7 | 5 | 1 | B[0] | Y[13] | 5.540 | 4.126 | 1.414 | ∞ | input port clock |
| Path 3 | ∞ | 7 | 5 | 1 | B[0] | Y[15] | 5.497 | 4.083 | 1.414 | ∞ | input port clock |
| Path 4 | ∞ | 6 | 5 | 1 | B[0] | Y[9] | 5.482 | 4.068 | 1.414 | ∞ | input port clock |
| Path 5 | ∞ | 7 | 5 | 1 | B[0] | Y[12] | 5.469 | 4.055 | 1.414 | ∞ | input port clock |

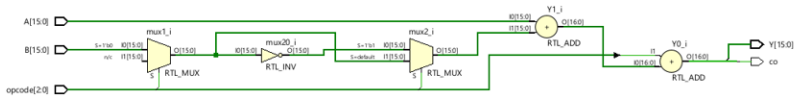



*Figure 43. Exercise Verilog Code and Synthesis*

The following code structure was created by adding pipes according to the desired data.

```
module ese01_pipe(
    input [15:0] A,
    input [15:0] B,
    input [2:0] opcode,
    input pipe,
    input wire clk,
    output reg [15:0] Y,
    output reg co
    );
    reg [15:0] X;
    reg [15:0] mux1;
    reg [15:0] mux2;
    reg [15:0] A_r;
    reg [15:0] pipeline_reg1, pipeline_reg2;

    always @(*) begin
        if (opcode[2] == 1'b0) begin
            mux1 = B;
        end else begin
            mux1 = X;
        end

        if (opcode[1] == 1'b1) begin
            mux2 = ~mux1;
        end else begin
            mux2 = mux1;
        end
    end

    always @(posedge clk) begin
        if (pipe == 2) begin
            pipeline_reg1 = A_r;
            pipeline_reg2 = mux2;
            {co, Y} = A_r + mux2 + opcode[0];
        end else begin
            {co, Y} = A_r + mux2 + opcode[0];
        end
    end

    // Pipeline register
    always @(posedge clk) begin
        if (pipe == 2) begin
            A_r <= pipeline_reg1;
            mux2 <= pipeline_reg2;
        end
    end
assign A = A_r;

endmodule
```

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 17 | 41000 | 0.04 |
| FF | 17 | 82000 | 0.02 |
| IO | 53 | 300 | 17.67 |

| Name | Slack | Levels | Routes | High Fanout | From | To | Total Delay | Logic Delay | Net Delay |
|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 2 | 2 | 1 | Y_reg[0]/C | Y[0] | 3.419 | 2.836 | 0.584 |
| Path 2 | ∞ | 2 | 2 | 1 | Y_reg[10]/C | Y[10] | 3.419 | 2.836 | 0.584 |
| Path 3 | ∞ | 2 | 2 | 1 | Y_reg[11]/C | Y[11] | 3.419 | 2.836 | 0.584 |
| Path 4 | ∞ | 2 | 2 | 1 | Y_reg[12]/C | Y[12] | 3.419 | 2.836 | 0.584 |
| Path 5 | ∞ | 2 | 2 | 1 | Y_reg[13]/C | Y[13] | 3.419 | 2.836 | 0.584 |

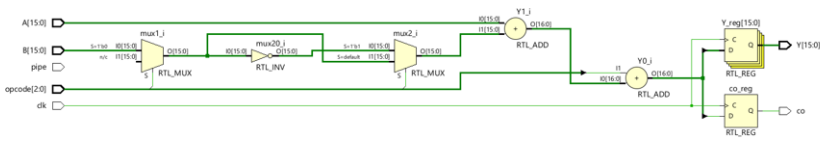



*Figure 44. Exercise Pipe Verilog Code and Synthesis*

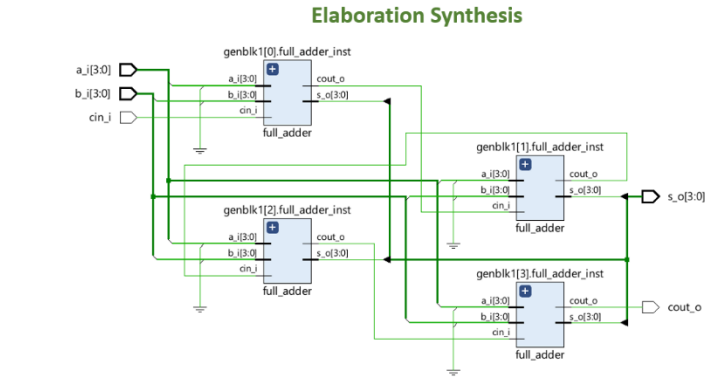
## 12. Carry Ripple Adder

Multiple full adder circuits can be cascaded in parallel to add an N-bit number. An N- bit parallel adder must have N number of full adder circuits. A ripple carry adder is a logic circuit in which the carry-out of each full adder is the carry-in of the succeeding next most significant full adder. It is called a ripple carry adder because each carry bit gets rippled into the next stage[22].

**Ripple Carry Adder**

```
module ripple_carry_adder
#(parameter N = 4)
(
input [N-1:0] a_i,
input [N-1:0] b_i,
input cin_i,
output [N-1:0]s_o,
output cout_o
);
wire [N:0] carry;

genvar i;
generate
for(i=0; i<N; i = i+1)begin
    full_adder full_adder_inst
    (
      .a_i(a_i[i]),
      .b_i(b_i[i]),
      .cin_i(carry[i]),
      .s_o(s_o[i]),
      .cout_o(carry[i+1])
    );
end
endgenerate
assign carry[0] = cin_i;
assign cout_o = carry[N];
endmodule
```

**Elaboration Synthesis**



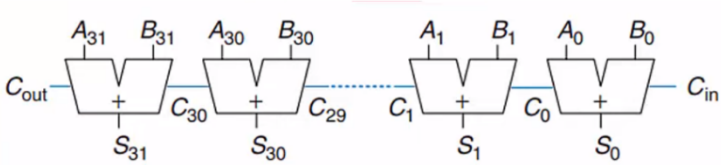
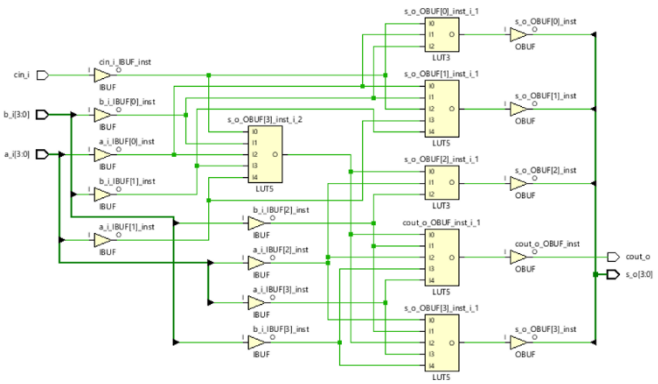
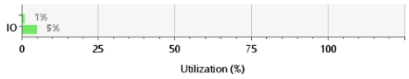




*Figure 45. Ripple Carry Adder Verilog Code*



| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 4 | 41000 | 0.01 |
| IO | 14 | 300 | 4.67 |




Design Runs | DRC | Power | **Timing** | Utilization

**Unconstrained Paths - NONE - NONE - Setup**

| Name | Slack | Levels | Routes | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Require |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 4 | 3 | 3 | a_i[0] | cout_o | 6.619 | 3.595 | 3.024 | |
| Path 2 | ∞ | 4 | 3 | 3 | a_i[0] | s_o[3] | 6.400 | 3.481 | 2.919 | |
| Path 3 | ∞ | 4 | 3 | 3 | a_i[0] | s_o[2] | 6.347 | 3.476 | 2.872 | |
| Path 4 | ∞ | 3 | 2 | 3 | a_i[0] | s_o[1] | 5.836 | 3.281 | 2.555 | |
| Path 5 | ∞ | 3 | 2 | 3 | a_i[0] | s_o[0] | 5.791 | 3.299 | 2.492 | |

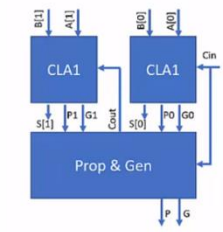
*Figure 46. Ripple Carry Adder Utilization and Timing*

## 13. Carry-Lookahead Adder

A carry-lookahead adder (CLA) or fast adder is a type of electronics adder used in digital logic. A carry-lookahead adder improves speed by reducing the time required to determine carry bits. It can be contrasted with the simpler, but usually slower, ripple-carry adder (RCA), for which the carry bit is calculated alongside the sum bit, and each stage must wait until the previous carry bit has been calculated to begin calculating its sum bit and carry. The carry-lookahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger-value bits of the adder[23].

**Carry - Lookahead Adder**

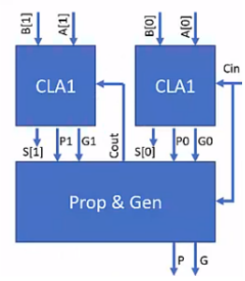
$$G_i = A_i \,\&\, B_i$$
$$P_i = A_i \,\wedge\, B_i = A_i \mid B_i$$
$$S_i = P_i \,\wedge\, C_i = (A_i \,\wedge\, B_i) \,\wedge\, C_i$$
$$C_{i+1} = G_i + P_i C_i$$
$$PG = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$GG = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$
$$PG = P_1 \,\&\, P_0$$
$$GG = G_1 \mid G_0 \,\&\, P_1$$
$$Cout = G_0 \mid P_0 \,\&\, Cin$$




```
module prop_gen
(
  input cin_i,
  input [1:0] p_i,
  input [1:0] g_i,
  output cout_o,
  output p_o,
  output g_o
);

assign cout_o = g_i[0] | p_i[0] & cin_i;
assign p_o = &p_i;
assign g_o = g_i[1] | g_i[0] & p_i[1];

endmodule
```

```
module cla1
(
   input a_i,
   input b_i,
   input cin_i,
   output p_o,
   output g_o,
   output s_o
 );

 assign p_o = a_i ^ b_i;
 assign g_o = a_i & b_i;
 assign s_o = a_i ^ b_i ^ cin_i;
endmodule
```

*Figure 47. CLA Verilog Code-1*




```
module cla2
(
   input [1:0] a_i,
   input [1:0] b_i,
   input cin_i,
   output p_o,
   output g_o,
   output [1:0] s_o
);
wire [1:0] p,g;
wire cout;
cla1 cla1_1
(
.a_i  (a_i[0]),
.b_i  (b_i[0]),
.cin_i (cin_i),
.p_o  (p[0]),
.g_o  (g[0]),
.s_o  (s_o[0])
);
```

```
cla1 cla1_2
(
.a_i  (a_i[1]),
.b_i  (b_i[1]),
.cin_i (cout),
.p_o  (p[1]),
.g_o  (g[1]),
.s_o  (s_o[1])
);

prop_gen pg
(
.cin_i  (cin_i),
.p_i  (p),
.g_i  (g),
.cout_o (cout),
.p_o  (p_o),
.g_o  (g_o)
);
} endmodule
```

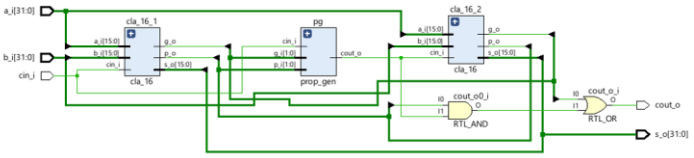
```
module cla32
(
   input [31:0] a_i,
   input [31:0] b_i,
   input cin_i,
   output cout_o,
   output [31:0] s_o
);

wire [1:0] p,g;
wire cout;

assign cout_o = g[1] | p[1] & cout;

cla_16 cla_16_1
(
  .a_i  (a_i[15:0]),
  .b_i  (b_i[15:0]),
  .cin_i (cin_i),
  .p_o  (p[0]),
  .g_o  (g[0]),
  .s_o  (s_o[15:0])

);
```

```
 cla_16 cla_16_2
(
  .a_i  (a_i[31:16]),
  .b_i  (b_i[31:16]),
  .cin_i (cout),
  .p_o  (p[1]),
  .g_o  (g[1]),
  .s_o  (s_o[31:16])

);

prop_gen pg
(
.cin_i  (cin_i),
.p_i  (p),
.g_i  (g),
.cout_o (cout),
.p_o  (),
.g_o  ()
);
endmodule
```

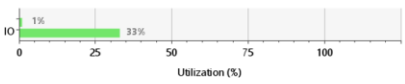
*Figure 48. CLA Verilog Code-2*


*Figure 49. CLA Testbench*

## Carry - Lookahead Adder Synthesis Results




| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 62 | 41000 | 0.15 |
| IO | 98 | 300 | 32.67 |




Design Runs | Utilization | Timing

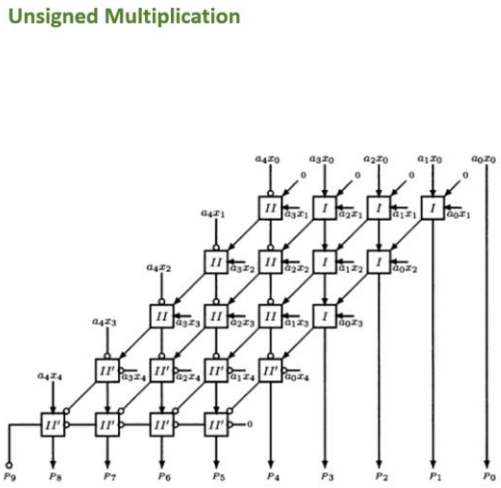
Unconstrained Paths - NONE - NONE - Setup

| Name | Slack | Levels | Routes | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 7 | 8 | 8 | a_i[5] | s_o[20] | 6.674 | 3.574 | 3.100 | ∞ | input port clock |
| Path 2 | ∞ | 7 | 8 | 8 | a_i[5] | s_o[21] | 6.671 | 3.571 | 3.100 | ∞ | input port clock |
| Path 3 | ∞ | 7 | 8 | 8 | a_i[5] | s_o[22] | 6.671 | 3.571 | 3.100 | ∞ | input port clock |
| Path 4 | ∞ | 7 | 8 | 8 | a_i[5] | s_o[19] | 6.670 | 3.577 | 3.093 | ∞ | input port clock |
| Path 5 | ∞ | 7 | 8 | 8 | a_i[5] | s_o[18] | 6.667 | 3.574 | 3.093 | ∞ | input port clock |
| Path 6 | ∞ | 7 | 8 | 8 | a_i[5] | s_o[23] | 6.659 | 3.577 | 3.082 | ∞ | input port clock |
| Path 7 | ∞ | 7 | 8 | 8 | a_i[5] | s_o[28] | 6.659 | 3.583 | 3.076 | ∞ | input port clock |
| Path 8 | ∞ | 7 | 8 | 8 | a_i[5] | s_o[29] | 6.659 | 3.583 | 3.076 | ∞ | input port clock |
| Path 9 | ∞ | 7 | 8 | 8 | a_i[5] | s_o[30] | 6.659 | 3.583 | 3.076 | ∞ | input port clock |
| Path 10 | ∞ | 7 | 8 | 8 | a_i[5] | s_o[31] | 6.659 | 3.583 | 3.076 | ∞ | input port clock |

*Figure 50. CLA Synthesis Results*

## 14. Unsigned Multiplication

Multiplication of two unsigned binary numbers of n bit size results into 2n bit result. In binary system, multiplication of the multiplicand by multiplier, if multiplier is 1 then multiplicand is entered in appropriate shifted position and if the multiplier is 0 then 0s are entered in appropriate shifted position[24].




```
module multu
#(parameter N = 32)
(
input [N-1:0]a_i,
input [N-1:0]b_i,
output [2*N-1:0] mul_o
 );

integer i;
integer j;
reg [2*N-1:0] temp;
reg [2*N-1:0] pp;

always @(*) begin
  temp = 0;
  for (i = 0; i < N; i = i+1)begin
     pp =0;
     for(j =0; j<N; j = j+1)begin
        pp[j] =a_i[j] & b_i[i];
     end
     pp = pp << i;
     temp = temp + pp;
  end
end

assign mul_o = temp;
endmodule
```

*Figure 51. Unsigned Multiplication Verilog Code*

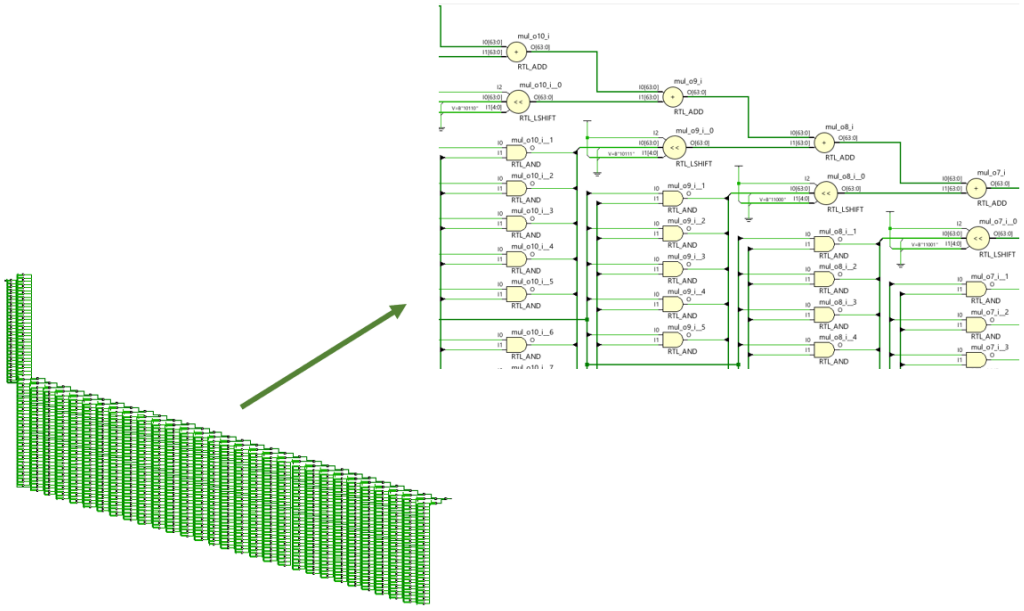
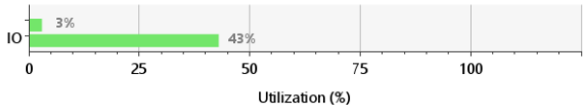


*Figure 52. Unsigned Multiplication Synthesis Result*

**Summary**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1069 | 41000 | 2.61 |
| IO | 128 | 300 | 42.67 |




**Unconstrained Paths - NONE - NONE - Setup**

| Name | Slack | Levels | Routes | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 16 | 8 | 63 | b_i[21] | mul_o[49] | 16.747 | 5.583 | 11.165 | ∞ | input port clock |
| Path 2 | ∞ | 14 | 8 | 63 | b_i[21] | mul_o[32] | 16.698 | 5.167 | 11.530 | ∞ | input port clock |
| Path 3 | ∞ | 14 | 8 | 63 | b_i[21] | mul_o[33] | 16.697 | 5.256 | 11.441 | ∞ | input port clock |
| Path 4 | ∞ | 16 | 8 | 63 | b_i[21] | mul_o[48] | 16.689 | 5.535 | 11.154 | ∞ | input port clock |
| Path 5 | ∞ | 15 | 8 | 63 | b_i[21] | mul_o[44] | 16.655 | 5.466 | 11.190 | ∞ | input port clock |
| Path 6 | ∞ | 16 | 8 | 63 | b_i[21] | mul_o[41] | 16.655 | 5.377 | 11.278 | ∞ | input port clock |
| Path 7 | ∞ | 18 | 8 | 63 | b_i[21] | mul_o[57] | 16.650 | 5.732 | 10.918 | ∞ | input port clock |
| Path 8 | ∞ | 15 | 8 | 63 | b_i[21] | mul_o[45] | 16.631 | 5.544 | 11.087 | ∞ | input port clock |
| Path 9 | ∞ | 17 | 8 | 63 | b_i[21] | mul_o[52] | 16.606 | 5.577 | 11.029 | ∞ | input port clock |
| Path 10 | ∞ | 17 | 8 | 63 | b_i[21] | mul_o[53] | 16.600 | 5.673 | 10.928 | ∞ | input port clock |

*Figure 53. Unsigned Multiplication Timing and Utilization Result*

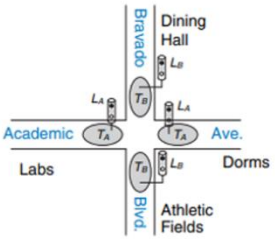
## 15. FSM with Synchronous Sequence Circuit Design

In this part, Harris & Harris book 3.4.1 FSM Example is discussed[25].

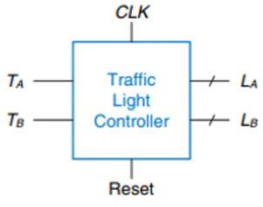
- ✓ Academic students shuttle back and forth between the Laboratories and Dormitories.
- ✓ Students from the Sports Academy move between the Cafeteria and the sports fields.
- ✓ Academic students are absorbed in their studies and do not look at the road, while sports students are focused on the ball and do not pay attention to the road, leading to collisions at intersections.

Objective: Design a traffic light to minimize these accidents.

- ➢ Sensors Ta and Tb check for the presence of students on two streets.
- ➢ Traffic lights La and Lb can display red, yellow, and green signals.
- ➢ There is a 5-second clock signal.
- ➢ To prevent student collisions, design an FSM (Finite State Machine) that includes sensor inputs and traffic light outputs, design the FSM diagram and state transition table, and write the Verilog code.




Campus Map




Black box view of finite state machine

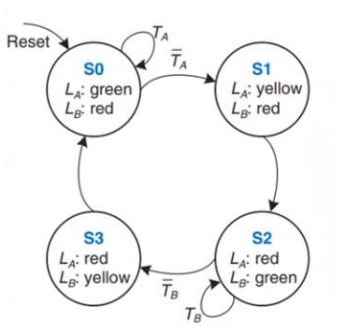



**Table 3.1** State transition table

| Current State $S$ | Inputs $T_A$ | Inputs $T_B$ | Next State $S'$ |
|---|---|---|---|
| S0 | 0 | X | S1 |
| S0 | 1 | X | S0 |
| S1 | X | X | S2 |
| S2 | X | 0 | S3 |
| S2 | X | 1 | S2 |
| S3 | X | X | S0 |

**Table 3.2** State encoding

| State | Encoding $S_{1:0}$ |
|---|---|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |
| S3 | 11 |

**Table 3.3** Output encoding

| Output | Encoding $L_{1:0}$ |
|---|---|
| green | 00 |
| yellow | 01 |
| red | 10 |

$$S_1' = S_1 \oplus S_0$$
$$S_0' = \overline{S}_1\overline{S}_0\overline{T}_A + S_1\overline{S}_0\overline{T}_B$$

$$L_{A1} = S_1$$
$$L_{A0} = \overline{S}_1 S_0$$
$$L_{B1} = \overline{S}_1$$
$$L_{B0} = S_1 S_0$$

**Table 3.4** State transition table with binary encodings

| Current State $S_1$ | Current State $S_0$ | Inputs $T_A$ | Inputs $T_B$ | Next State $S_1'$ | Next State $S_0'$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0 | 0 |

**Table 3.5** Output table

| Current State $S_1$ | Current State $S_0$ | Outputs $L_{A1}$ | Outputs $L_{A0}$ | Outputs $L_{B1}$ | Outputs $L_{B0}$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |

*Figure 54. FSM Demonstration and State Tables*

```
module fsm_exp
(
 input clk,
 input rst_n,
 input ta_i,
 input tb_i,
 output [1:0]la_o,
 output [1:0]lb_o
 );

 localparam S0 = 2'b00;
 localparam S1 = 2'b01;
 localparam S2 = 2'b10;
 localparam S3 = 2'b11;

 localparam green = 2'b00;
 localparam yellow = 2'b01;
 localparam red = 2'b10;

 reg [1:0] la,lb;
 reg [1:0] state;
```

```
always @(posedge clk, negedge rst_n)begin
    if(rst_n == 1'b0)begin
       state <= S0;
    end
    else begin
        case (state)
           S0 : begin
              if (ta_i == 1'b1)begin
                 state <= S0;
               end
               else begin
                  state <= S1;
               end
           end
           S1 :begin
              state <= S2;
           end
           S2 :begin
                if (tb_i == 1'b0) begin
                  state <= S3;
                end else begin
                  state <= S2;
                end
            end
           S3 :begin
              state <= S0;
           end
           default : begin
              state <= S0;
           end
        endcase

    end
end
```

```
always @(*) begin
   case (state)
      S0: begin
         la = green;
         lb = red;
      end
      S1: begin
         la = yellow;
         lb = red;
      end
      S2: begin
         la = red;
         lb = green;
      end
      S3: begin
         la = red;
         lb = yellow;
      end
      default : begin
         la = green;
         lb = red;
      end
   endcase
end

assign la_o = la;
assign lb_o = lb;
endmodule
```

*Figure 55. FSM Verilog Code*

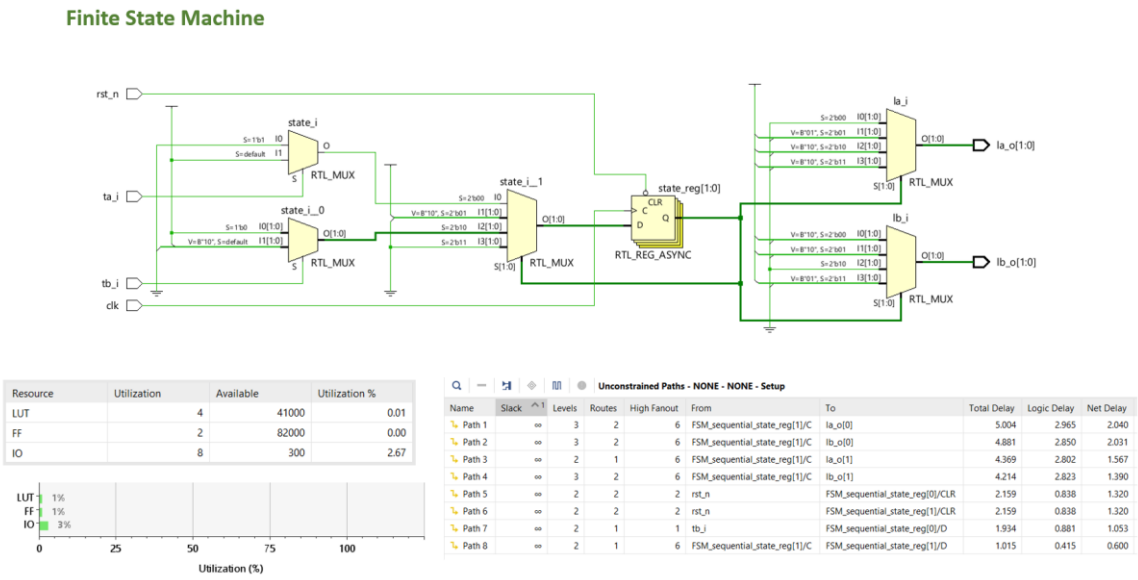



*Figure 56. FSM Verilog Code Utilization and Timing Synthesis*

Finally, we can say that the "encoding" part is an important issue in state machines. Encoding is still a matter of research today. How can we make a situational machine faster? In the example above, I used "binary encoding", but my code first says either "00" or "11" to go from "01" to "10", which may cause a loss of time, even in milliseconds. For this reason, if I had used "gray encoding", it would have been a little faster than my code.

| Binary | One-Hot | Gray |
|---|---|---|
| 00 | 0001 | 00 |
| 01 | 0010 | 01 |
| 10 | 0100 | 11 |
| 11 | 1000 | 10 |

# References


[1] ‘Programmable Logic Array’. Accessed: Apr. 20, 2024. [Online]. Available: https://www.geeksforgeeks.org/programmable-logic-array/

[2] ‘PLA (Programmable Logic Array)’. Accessed: Apr. 20, 2024. [Online]. Available: https://people.cs.pitt.edu/~wiebe/courses/CS447/lectures/pla.html

[3] ‘ CPLD (Complex Programmable Logic Device)’. Accessed: Apr. 20, 2024. [Online]. Available: https://www.easybom.com/blog/a/cpld-complex-programmable-logic-device-explained

[4] ‘All About FPGAs’. Accessed: Apr. 20, 2024. [Online]. Available: https://www.eetimes.com/all-about-fpgas/

[5] ‘FPGA Architecture’. Accessed: Apr. 20, 2024. [Online]. Available: https://www.seeedstudio.com/blog/2021/01/20/what-is-an-fpga-and-why/

[6] ‘Hardware Description Language’. Accessed: Apr. 20, 2024. [Online]. Available: https://www.geeksforgeeks.org/hardware-description-language/

[7] *IEEE Standard for Verilog Hardware Description Language.* [publisher not identified], 2006.

[8] ‘Verilog Module Instantiations’, https://www.chipverify.com/verilog/verilog-module-instantiations.

[9] ‘Verilog Testbench’, https://www.chipverify.com/verilog/verilog-testbench.

[10] ‘what-is-the-difference-between-reg-and-wire-in-a-verilog-module’, https://stackoverflow.com/questions/33459048/what-is-the-difference-between-reg-and-wire-in-a-verilog-module.

[11] ‘Verilog HDL ile Davranışsal Modelleme 1 (Temel Kavramlar)’.

[12] ‘SR Latch’, https://www.build-electronic-circuits.com/s-r-latch/.

[13] Mehmet Burak Aykenar, ‘Sequential Circuits | Flip-Flop ’, https://www.youtube.com/watch?v=1s8xiFdxcA4&list=PLZyLAHn50933YtB32ECJujIskuFJiz8AJ&index=13.

[14] Omar Muñoz Urias, ‘D-Latch’, https://www.build-electronic-circuits.com/d-latch/.

[15] ‘what-is-the-difference-between-edge-and-level-triggering’, https://pediaa.com/what-is-the-difference-between-edge-and-level-triggering/.

[16] AMD, ‘When-and-Where-to-Use-a-Reset’, https://docs.amd.com/r/en-US/ug949-vivado-design-methodology/When-and-Where-to-Use-a-Reset.

[17] ‘synchronous-reset-vs-asynchronous-reset’, https://vlsiweb.com/synchronous-reset-vs-asynchronous-reset/.

[18] Mehmet Burak Aykenar, ‘Logic Design & Computer Arhitecture’, https://www.youtube.com/@mehmetburakaykenar.

[19] Chip Verify, ‘verilog-parameters’, https://chipverify.com/verilog/verilog-parameters.

[20] Mehmet Burak Aykenar, ‘Synchronous Circuits’, https://www.youtube.com/watch?v=5iNGic0Cg8U&list=PLZyLAHn50933YtB32ECJujIskuFJiz8AJ&index=13.

[21] All About Circuits, ‘finite-state-machine’, https://www.allaboutcircuits.com/technical-articles/implementing-a-finite-state-machine-in-vhdl/.

[22] ‘ripple-carry-adder’, https://www.circuitstoday.com/ripple-carry-adder.

[23] ‘Carry-lookahead_adder’, https://en.wikipedia.org/wiki/Carry-lookahead_adder.

[24] ‘Multiplication’, https://math.libretexts.org/Bookshelves/Algebra/Elementary_Algebra_(Ellis_and_Burzynski)/03%3A_Basic_Operations_with_Real_Numbers/3.06%3A_Multiplication_and_Division_of_Signed_Numbers.

[25] D. Hung, ‘In Praise of Digital Design and Computer Architecture ARM ® Edition’.