

# **A SUMMER INTERNSHIP REPORT**

**On**

## **Electronic Circuit Analysis using PSPICE & Xilinx Vivado (ALARM CLOCK)**

**Submitted for the partial fulfilment  
of**

**B. Tech.**

**in**

## **ELECTRONICS AND COMMUNICATION ENGINEERING**



***SUBMITTED BY:***

**Pankhuri Panday  
(2100270310096)**

**5<sup>th</sup> Semester - Third Year  
Section- EC-2**

**SUBMITTED TO:**

**Dr. PANKAJ GOEL  
(ASST. PROF.)  
ECE DEPT.**

**Ajay Kumar Garg Engineering College, Ghaziabad**

**27<sup>th</sup> Km Milestone, Delhi-Meerut Expressway, P.O. Adhyatmik Nagar, Ghaziabad-201009**

**Dr. A. P. J. Abdul Kalam Technical University, Lucknow**

**NOVEMBER 2023**

# Acknowledgement

I want to express my sincere gratitude and thanks to **Prof. (Dr.) Neelesh Kumar Gupta (HoD, ECE Department), Ajay Kumar Garg Engineering College, Ghaziabad** for granting me permission for my industrial training in the field of “**Electronic Circuit Analysis using PSPICE & Xilinx Vivado**”.

I express my sincere thanks to **Asst. Prof. (Dr.) Pankaj Goel** for his cooperative attitude and consistence guidance, due to which I was able to complete my training successfully.

Finally, I pay my thankful regard and gratitude to the faculty members and lab technicians of **Ajay Kumar Garg Engineering College, Ghaziabad** for their valuable help, support and guidance.

**Pankhuri Panday**  
**2100270310096**  
**5<sup>th</sup> Sem-Third Year**  
**Section- EC-2**

# **TABLE OF CONTENTS**

<b>Chapter 1. Introduction to Verilog HDL</b>	<b>1-12</b>
1.1 Introduction	
1.2 Keywords and Identifiers	
1.3 Data Types and Modelling Styles	
1.4 Verilog programming for various combinational and Sequential circuits	
<b>Chapter 2. Introduction to PSPICE and Xilinx Vivado Design Suite</b>	<b>13-18</b>
2.1 Introduction	
2.2 Key features of PSPICE Simulator	
2.3 Xilinx Vivado Design flow	
<b>Chapter 3. Project Description</b>	<b>19-26</b>
3.1 Introduction to Alarm Clock	
3.2 Objectives	
3.3 Description of Project	
3.3.1 Block Diagram/ Flow Chart	
3.3.2 Algorithm Description	
<b>Chapter 4. Simulation and Synthesis Results</b>	<b>27-30</b>
4.1 Simulation Result	
4.2 Synthesis Results	
<b>Chapter 5. Conclusion and Future Scope</b>	<b>31-33</b>
5.1 Conclusion	
5.2 Future Scope	
<b>References</b>	<b>34</b>

# CHAPTER- 1

## INTRODUCTION TO VERILOG HDL

### 1.1 Introduction

Verilog Hardware Description Language (HDL) is a high-level programming language used for the design, simulation, and synthesis of digital electronic systems. It is widely used in the field of electronic design automation (EDA) to describe and model the behaviour of digital circuits and systems. Verilog is particularly popular for its role in designing and verifying digital integrated circuits, FPGA (Field-Programmable Gate Array) configurations, and other digital systems. Here's a brief introduction to Verilog HDL:

**1.Purpose:** Verilog HDL is primarily used for specifying the functionality and structure of digital systems. Designers use it to describe how various components (gates, registers, flip-flops, etc.) are connected and how they interact with one another.

#### **2.Two Main Uses:**

- a) **Simulation:** Verilog is used to create simulation models of digital circuits, allowing designers to verify the correctness of their designs before committing to hardware implementation. This helps in identifying and debugging potential issues early in the design process.
- b) **Synthesis:** Verilog can also be used for logic synthesis, where the Verilog code is transformed into an equivalent hardware description that can be implemented in physical devices, like ASICs (Application-Specific Integrated Circuits) or FPGAs.

**3.Structure:** Verilog code is organized into modules, which represent different components or subcomponents of a digital system. Modules contain inputs, outputs, registers, wires, and a description of how the components are interconnected.

**4.Behavioural and Structural:** Verilog supports both behavioural and structural modelling. Behavioural modelling focuses on the functionality and operations of a component, while structural modelling defines how components are connected and configured.

**5.Simulators:** Verilog can be used with simulators like ModelSim, XSIM, and many others to test and validate the behaviour of a digital design. These simulators allow designers to execute the Verilog code and observe how the circuit behaves over time.

**6.Testbenches:** Verilog testbenches are used to stimulate the design under test (DUT) and monitor its behaviour during simulation. Testbenches are written in Verilog as well and are crucial for design verification.

**7.Synthesis Tools:** For physical implementation, Verilog can be synthesized using tools like Xilinx Vivado, Synopsys Design Compiler, or Cadence Genus. These tools convert Verilog descriptions into a netlist that can be implemented in hardware.

**8. Standardization:** Verilog HDL has been standardized by the IEEE (Institute of Electrical and Electronics Engineers) as IEEE 1364. The standard has evolved over time, and SystemVerilog is an extension that includes additional features for system-level design and verification.

**9. VHDL vs. Verilog:** Verilog is one of the two most commonly used HDLs, the other being VHDL (VHSIC Hardware Description Language). Each has its own strengths and is favoured in different industries and applications.

Verilog HDL is a powerful and widely-used language for modelling, simulating, and synthesizing digital circuits and systems. It plays a crucial role in the design and verification of modern digital electronics, from microprocessors to complex FPGA-based systems. Designers use Verilog to describe the functionality and structure of digital components and systems, making it an essential tool in the field of electronic design and computer engineering.

## 1.2 Keywords and Identifiers

### *Keywords:*

In Verilog, like in most programming and hardware description languages, there are keywords and identifiers. Keywords are reserved words with specific meanings in the language, and identifiers are user-defined names for various elements in your Verilog code, such as modules, signals, and variables. Here's a list of some common keywords and guidelines for identifiers in Verilog:

Examples of Verilog keywords include:

- **module:** Used to define a module, which is the basic building block in Verilog.
- **endmodule:** Marks the end of a module definition.
- **input:** Declares an input port to a module.
- **output:** Declares an output port from a module.
- **inout:** Declares a bidirectional (input and output) port to a module.
- **wire:** Declares a wire data type for interconnecting signals.
- **reg:** Declares a register data type for sequential logic.
- **always:** Begins a block of code that specifies the behaviour of the circuit.
- **if, else:** Conditional statements used to control the flow of logic.
- **case, casez, casex:** Used for case statement constructs.
- **assign:** Used to drive a wire with a continuous assignment.
- **begin, end:** Define a block of statements.
- **always\_ff or always\_comb:** More specific versions of the always block for sequential and combinational logic, respectively.
- **initial:** Used for describing initial conditions in a design.
- **posedge, negedge:** Used in edge-sensitive always blocks.
- **module, endmodule:** Used to define and end module definitions.
- **function, endfunction:** Used to define and end functions.

### *Identifiers:*

- Identifiers are user-defined names for various elements in your Verilog code, such as modules, signals, variables, and instances.
- Identifiers must begin with an alphabetical letter (a-z or A-Z) or an underscore (\_).
- Following the first character, you can use letters, numbers, or underscores.
- Identifiers are case-sensitive. For example, mySignal and mysignal are treated as distinct identifiers.

- It's a good practice to use descriptive and meaningful names for clarity and maintainability.

Here are some examples of Verilog identifiers:

- **counter:** An identifier for a signal or variable.
- **ALU\_Control\_Unit:** An identifier for a module or instance.
- **data\_in:** An identifier for an input port.
- **result\_out:** An identifier for an output port.
- **clk:** An identifier for a clock signal.
- **\_enable\_signal:** Identifiers can start with an underscore.
- **counter2:** Identifiers can contain numbers.
- **state\_machine:** An identifier for a module.

## 1.3 Data Types and Modelling Styles

In Verilog, there are several data types that are used to represent different kinds of values and variables. These data types play a crucial role in describing the behaviour and structure of digital circuits. Here are some common data types in Verilog:

1. **Wire:** The ``wire`` data type represents a continuous signal or wire that carries values from one logic element to another. Wires are typically used for connecting the inputs and outputs of modules.

```
verilog Copy code  
  
wire data_bus;
```

2. **Register:** The ``reg`` data type represents registers, which are elements that store values and can be used for sequential logic. Registers are often used to hold state information in digital circuits.

```
verilog Copy code  
  
reg counter;
```

3. **Integer:** The ``integer`` data type represents integer values. It's often used for loop counters and other integer variables in behavioural descriptions.

```
verilog Copy code  
  
integer i;
```

4. **Real:** The ``real`` data type represents real numbers, which can have fractional parts. Real values are used for analog simulations and can also be used in certain mathematical operations in Verilog.

```
verilog Copy code  
  
real voltage;
```


5. **Time:** The ``time`` data type represents time values. It is used for specifying delays and time-related parameters in simulation models.

```
verilog Copy code  
  
time delay = 10ns;
```

6. **Parameter:** The ``parameter`` data type is used for constants that can be defined at the module level and provide flexibility to customize a module's behaviour without modifying the core code.




verilog

 Copy code

```
parameter DATA_WIDTH = 8;
```

7. **Enum:** Verilog supports enumerated data types, which allow you to create a set of named values. Enums are useful for improving code readability and maintainability.


verilog

 Copy code

```
typedef enum logic [1:0] { IDLE, RUNNING, STOPPED } State;  
State current_state;
```

8. **Arrays:** Verilog allows you to define arrays of various data types. For example, you can create arrays of wires, registers, or integers to represent multiple signals or variables of the same type.


verilog

 Copy code

```
wire [7:0] data_bus[3:0];
```

9. **Structs and Unions:** Verilog-2005 and later versions support user-defined data structures, including structs and unions, which are used for organizing and grouping related data.

verilog

 Copy code

```
typedef struct {  
    reg [7:0] value;  
    wire valid;  
} Data;  
Data data_packet;
```

These data types help in describing the behaviour of digital circuits and facilitate modelling and simulation in Verilog. The choice of data type depends on the purpose of the variable or signal, and using the appropriate data type is important for accurate and efficient modelling of digital systems.

In Verilog, there are several modelling styles that you can use to describe the behaviour and structure of digital circuits. The choice of modelling style depends on the complexity of your design, your design goals, and your personal preference. Here are some common modelling styles in Verilog:

### ***1. Behavioural Modelling:***

- **RTL (Register-Transfer Level):** RTL modelling describes the behaviour of a circuit in terms of registers and the data transfer between them. It is a high-level description that

focuses on the flow of data between registers and the combinational logic that connects them.

- **Algorithmic Level:** In algorithmic-level modelling, you describe the desired functionality of a digital circuit using high-level algorithms and operations. This style is often used for complex digital signal processing or control systems.

## ***2. Structural Modelling:***

- **Gate-Level Modelling:** Gate-level modelling describes the design using basic logic gates (AND, OR, NOT, etc.) and flip-flops. It is a low-level description that is often used for synthesis.
- **Switch-Level Modelling:** Switch-level modelling is even more detailed and describes the behaviour of transistors and switches within the gates. It is used in very low-level designs and for custom IC design.

## ***3. Dataflow Modelling:***

- **Continuous Assignment:** In this modelling style, you use continuous assignment statements to describe the flow of data in a circuit. It is commonly used for assigning values to wires based on combinational logic.
- **Procedural Assignment:** Procedural assignment is used to model the behaviour of sequential logic elements, such as registers and flip-flops. This style uses procedural constructs like ``always`` blocks.

## ***4. Mixed-Signal Modelling:***

- In some designs, you may need to model both digital and analog components. Verilog-A and Verilog-AMS are extensions of Verilog that allow for mixed-signal modelling. Verilog-A is used for analog modelling, while Verilog-AMS combines both analog and digital modelling.

## ***5. FSM (Finite State Machine) Modelling:***

- For designs that involve state machines, you can use a specific modelling style to describe the state transitions and state outputs. This often involves defining state variables and using ``case`` statements.

## ***6. Testbench Modelling:***

- A testbench is a part of your Verilog code that is not used for the actual implementation of the design but is essential for simulating and verifying the design. Testbenches include stimulus generation, monitoring, and checking for correctness.

## ***7. Parameterized and Generic Modelling:***

- Verilog allows you to create parameterized and generic modules that can be configured with different parameters or data widths to make your code more versatile and reusable.

## ***8. Structural Hierarchy:***

- You can model complex systems by breaking them down into smaller, more manageable modules. This hierarchical modelling style makes it easier to design, simulate, and understand large systems.

## ***9. Memory Modelling:***

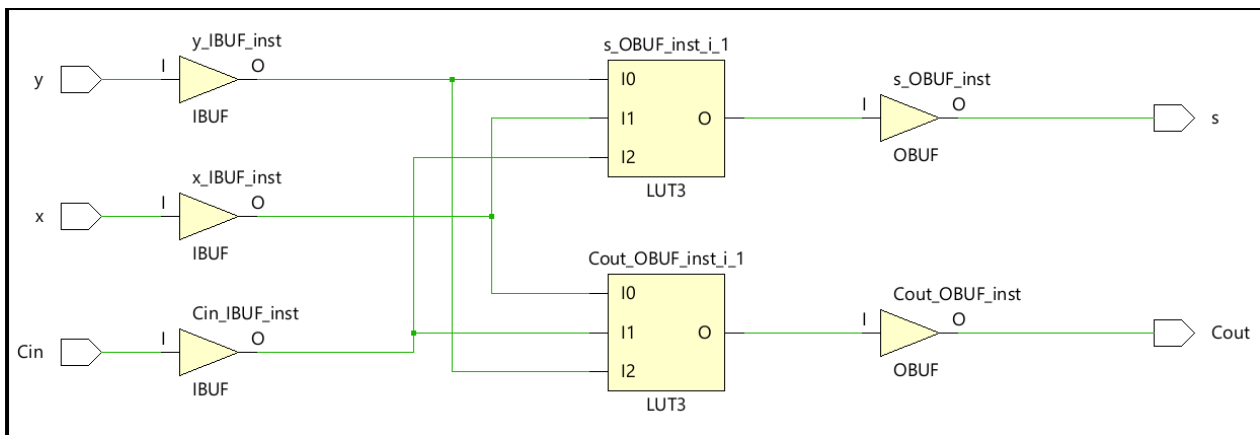
- In digital systems that involve memory elements (RAM, ROM, etc.), specific modelling styles and constructs are used to define and simulate memory modules.

Each of these modelling styles has its own advantages and is suited for different design scenarios. The choice of modelling style should consider the level of abstraction, the target hardware, and the complexity of the design. In practice, complex designs often use a combination of these styles to capture different aspects of the design.

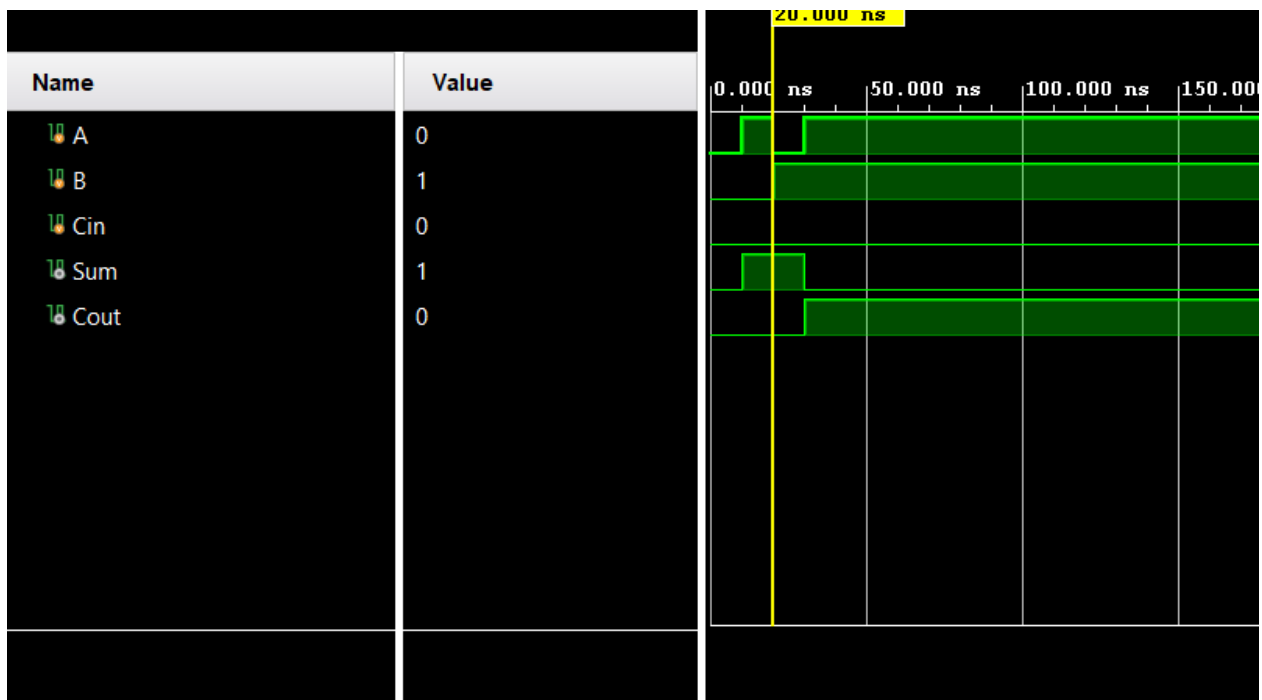
## 1.4 Verilog programming for various combinational and Sequential circuits

### *Full Adder:*

```
module fulladd (Cin, x, y, s, Cout);  
input Cin, x, y;  
output s, Cout;  
wire z1, z2, z3, z4;  
and And1 (z1, x, y);  
and And2 (z2, x, Cin);  
and And3 (z3, y, Cin);  
or Or1 (Cout, z1, z2, z3);  
xor Xor1 (z4, x, y);  
xor Xor2 (s, z4, Cin);  
endmodule
```



**Fig. 1.1** Schematic Diagram of Full Adder



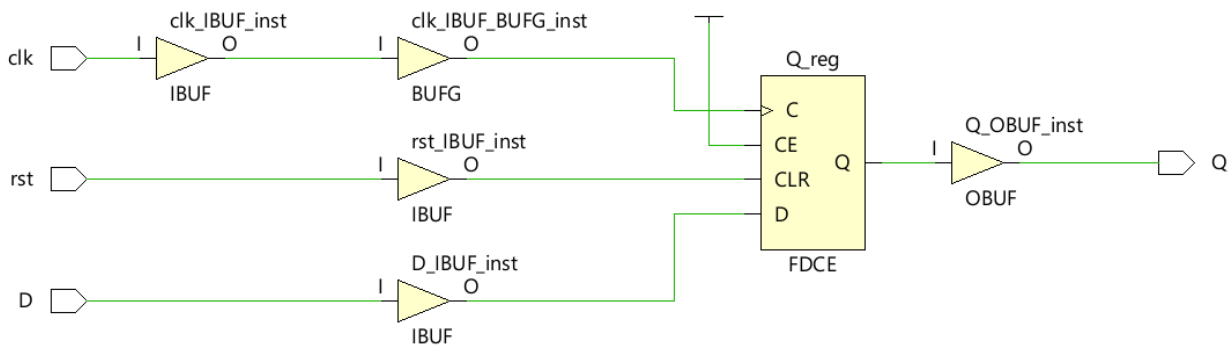
**Fig. 1.2** Stimulation of Full Adder

### *D flip-flop:*

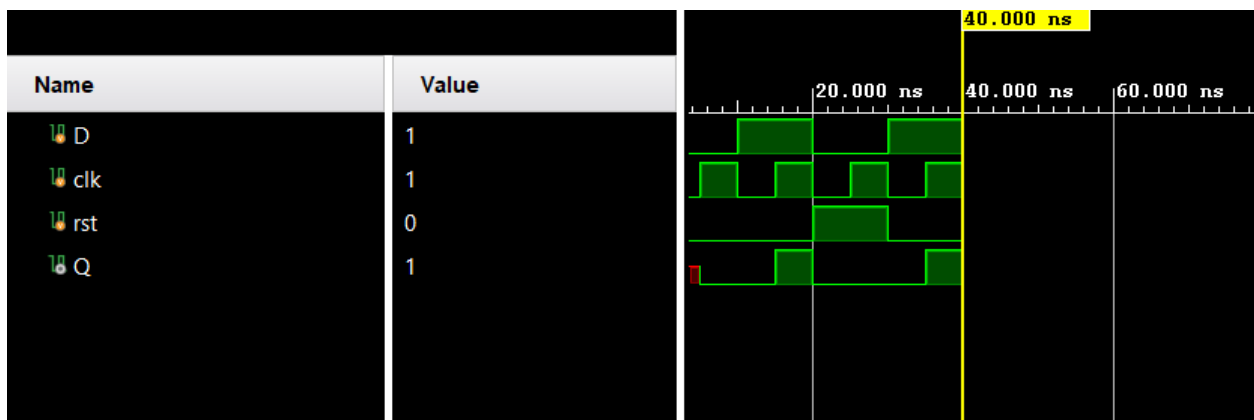
```

module d_flip_flop(
  input D,
  input clk,
  input rst,
  output reg Q
);
always @(posedge clk or posedge rst) begin
  if (rst) begin
    Q <= 0;
  end else begin
    Q <= D;
  end
end
endmodule

```



**Fig. 1.3** Schematic Diagram of D flip-flop with asynchronous reset



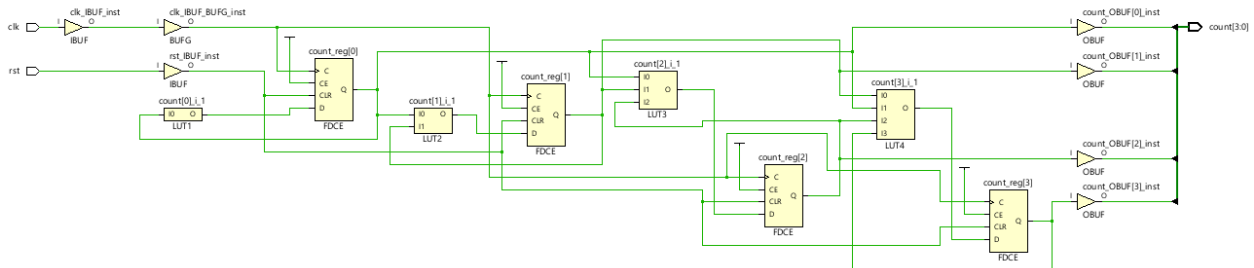
**Fig. 1.4** Stimulation of D flip-flop

## Four-bit up-counter

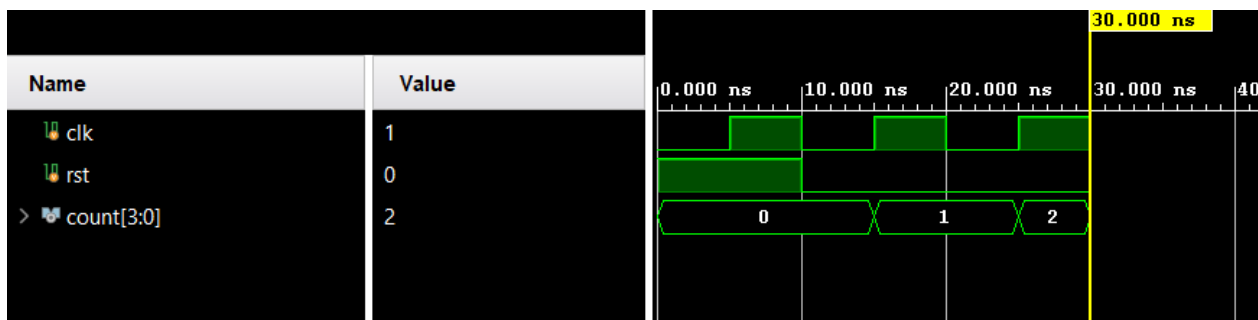
```
module up_counter_4bit(
    input wire clk, // Clock input
    input wire rst, // Reset input
    output reg [3:0] count // 4-bit counter output
);
```

```
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            count <= 4'b0000; // Reset the counter to 0
        end else begin
            count <= count + 1; // Increment the counter on the rising edge of the clock
        end
    end
end
```

```
endmodule
```



**Fig. 1.5** Schematic Diagram of Four-bit up-counter



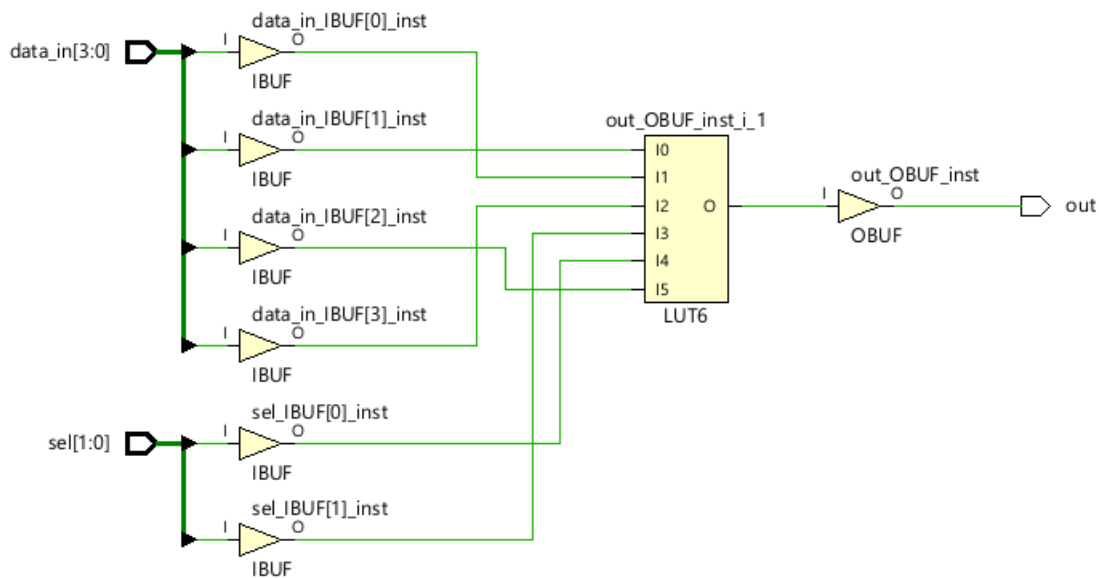
**Fig. 1.6** Stimulation of Four-bit up-counter

## 4-to-1 multiplexer specified using the conditional operator

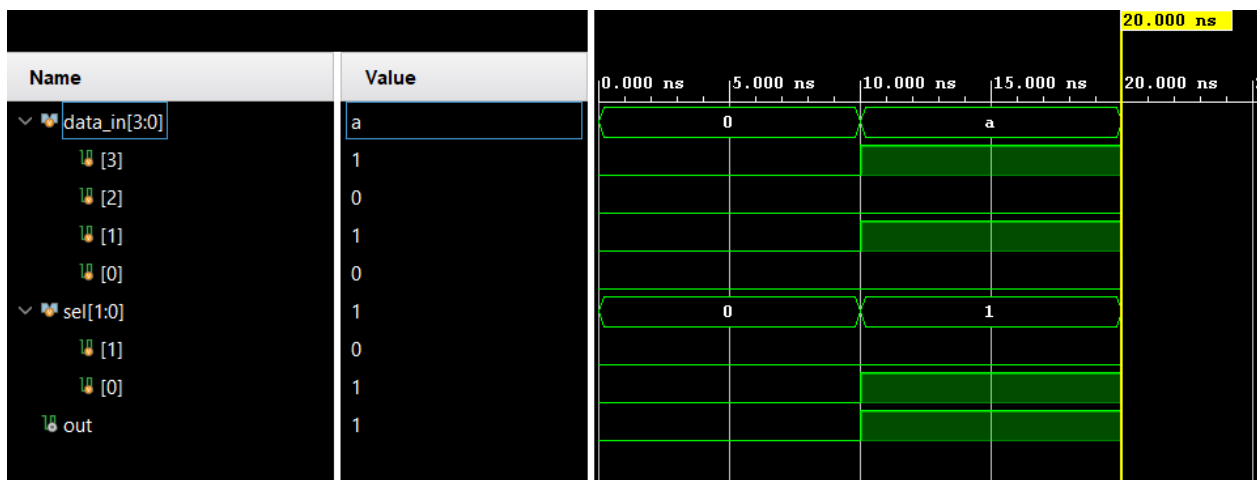
```

module mux_4to1(
  input wire [3:0] data_in,
  input wire [1:0] sel,
  output reg out
);
always @* begin
  case (sel)
    2'b00: out = data_in[0];
    2'b01: out = data_in[1];
    2'b10: out = data_in[2];
    2'b11: out = data_in[3];
    default: out = 4'bxxxx;
  endcase
end
endmodule

```



**Fig. 1.7** Schematic Diagram of 4-to-1 multiplexer



**Fig. 1.8** Stimulation of 4-to-1 multiplexer

## CHAPTER- 2

# INTRODUCTION TO XILINX VIVADO TOOLBOX

## 2.1 Introduction

PSPICE and Xilinx Vivado are two powerful software tools used in the field of electronic design and digital circuit development. They serve different purposes and are widely utilized in various stages of the electronic design process. Here's an introduction to each of these tools:

### *1. PSPICE (Program to Simulate Circuits Emulating Electronics):*

PSPICE is a widely used electronic circuit simulation software developed by Cadence Design Systems. It allows engineers and designers to model, simulate, and analyse electronic circuits before they are built in hardware. PSPICE is particularly popular for analog and mixed-signal circuit simulations. Here are some key features and applications of PSPICE:

- **Circuit Simulation:** PSPICE enables users to create and simulate electronic circuits by defining components, connections, and parameters. It can model both linear and nonlinear components, making it suitable for a wide range of applications.
- **AC and DC Analysis:** It can perform AC analysis for frequency response, DC analysis for bias point calculation, and transient analysis for time-domain simulations, among others.
- **Monte Carlo Analysis:** PSPICE allows for statistical analysis of circuit performance by varying component values within specified ranges.
- **Sensitivity Analysis:** Users can assess how changes in component values impact circuit performance.
- **Device Modelling:** PSPICE provides a library of device models, and users can create custom models for specialized components.
- **Waveform Visualization:** It generates output waveforms, plots, and graphs to help designers understand the behaviour of their circuits.
- **Education:** PSPICE is used in academia to teach electronics and circuit analysis.

### *2. Xilinx Vivado:*

Xilinx Vivado is an integrated development environment (IDE) developed by Xilinx, a leading FPGA (Field-Programmable Gate Array) and SoC (System-on-Chip) manufacturer. Vivado is primarily used for digital logic design, synthesis, implementation, and verification of FPGA and SoC-based projects. Here's an overview of Xilinx Vivado:

- **FPGA and SoC Design:** Vivado is the go-to tool for developing digital designs that target Xilinx FPGA and SoC devices. Designers use Vivado to create custom logic circuits, integrate processor cores (like ARM Cortex-A9), and develop complex embedded systems.



- **High-Level Synthesis:** Vivado HLS is an add-on tool that allows designers to create FPGA designs using high-level programming languages like C, C++, and SystemC, which are then synthesized into RTL (Register Transfer Level) code.
- **IP Integration:** Vivado includes a vast library of pre-designed IP (Intellectual Property) blocks, such as memory controllers, communication interfaces, and image processing cores, which can be easily integrated into custom designs.
- **Simulation and Verification:** Vivado provides built-in simulation tools for functional verification and debugging of FPGA designs.
- **Implementation:** It supports the synthesis and place-and-route processes to convert RTL code into a configuration bitstream that can be loaded onto the FPGA.
- **Programming and Debugging:** Vivado allows users to program the configured FPGA and provides debugging tools for hardware/software co-designs.
- **Power Analysis:** Designers can analyse power consumption at various stages of the design process.

Both PSPICE and Xilinx Vivado are indispensable tools in the world of electronic design and digital circuit development. While PSPICE focuses on electronic circuit simulation and analysis, Xilinx Vivado is tailored for FPGA and SoC design and development. Depending on your specific needs, you may use one or both tools in your projects.

## 2.2 Key features of PSPICE Simulator

PSPICE (Program to Simulate Circuits Emulating Electronics) is a widely used electronic circuit simulation software that offers a wide range of features for designing and analysing electronic circuits. Here are some of the key features of the PSPICE simulator:

1. **Circuit Simulation:** PSPICE allows you to create and simulate electronic circuits, including analog, digital, and mixed-signal circuits. It supports a wide variety of components and circuit types.
2. **Component Library:** PSPICE provides a comprehensive library of electronic components, including resistors, capacitors, transistors, operational amplifiers, and more. Users can access and use these components to build their circuits.
3. **Transient Analysis:** This feature allows you to simulate and analyse the time-domain behaviour of a circuit. It's useful for understanding how a circuit responds to changes in input over time.
4. **AC and DC Analysis:** PSPICE can perform AC analysis for frequency response (Bode plots) and DC analysis for calculating the bias point (DC operating point) of a circuit.
5. **Parameter Sweeps:** You can perform parameter sweeps to study how circuit performance varies with changes in component values. This is useful for sensitivity analysis and optimizing circuit design.
6. **Monte Carlo Analysis:** PSPICE supports statistical analysis by allowing you to perform Monte Carlo simulations. This feature helps assess the impact of component tolerances on circuit performance.
7. **Device Modelling:** PSPICE offers a wide range of device models, including standard models for common components and the capability to create custom device models for specialized components.
8. **Sensitivity Analysis:** It enables you to analyse how variations in component values affect the circuit's behaviour and performance.
9. **Complex Number Support:** PSPICE can handle complex numbers, making it suitable for analysing circuits with impedance, transfer functions, and phasor analysis.
10. **Interactive Design:** You can interactively design and modify circuits, making it easy to experiment with different configurations and component values.
11. **Integration with PCB Design Tools:** PSPICE is often integrated with PCB (Printed Circuit Board) design software, allowing designers to validate their circuit designs before moving to PCB layout.
12. **Custom Scripting:** Advanced users can use scripting and automation to create custom simulations and analyses.
13. **Library Management:** You can create and manage custom component libraries to store and reuse your design assets.

PSPICE is a versatile tool that is widely used by electronic engineers and designers for both educational and professional purposes. Its features make it a valuable resource for simulating and analysing electronic circuits, ensuring they function as intended before physical implementation.

## 2.3 Xilinx Vivado Design flow

Xilinx Vivado is a comprehensive integrated development environment (IDE) used for designing and programming Xilinx Field-Programmable Gate Arrays (FPGAs) and System-on-Chip (SoC) devices. The Vivado design flow outlines the steps involved in creating, implementing, and programming FPGA and SoC designs. Here's an overview of the typical Xilinx Vivado design flow:

### 1. Project Creation:

- Create a new Vivado project, specifying the project name, location, and target FPGA or SoC device.
- Define the project settings, including the top-level module, preferred language (VHDL or Verilog), and simulation tool preferences.

### 2. Design Entry:

- Designers can use different methods for entering their design:
- RTL Design: Write or import Register Transfer Level (RTL) code using VHDL, Verilog, or high-level languages.
- High-Level Synthesis (HLS): Create RTL designs from high-level languages such as C, C++, or SystemC using Vivado HLS.
- Schematic Entry: Use the graphical interface to draw schematics for simple designs.

### 3. Simulation and Verification:

- Verify the functionality of your design using Vivado's built-in simulator or third-party simulators such as ModelSim.
- Write and run testbenches to validate the design's correctness.

### 4. Synthesis:

- Perform synthesis to convert the RTL code into a gate-level netlist that can be implemented on the target FPGA.
- Optimize the design for area, speed, or power, based on your requirements.

### 5. Implementation:

- The implementation process involves several steps:
- Place and Route: Place the logical elements (LUTs, flip-flops, etc.) onto the FPGA and determine their physical locations. Establish routing connections between them.
- Bitstream Generation: Create the configuration bitstream file, which represents the FPGA programming file.
- Timing Analysis: Perform static timing analysis to ensure that the design meets the desired timing constraints.
- Report Generation: Review various reports generated by Vivado to assess resource utilization, performance, and critical paths.

### 6. Programming the FPGA:

- Program the configured bitstream onto the target FPGA device using a programming cable or onboard programming interface.
- Ensure the device is correctly configured, and your design is operational.

## **7. Hardware Debugging:**

- Vivado provides tools for debugging hardware designs using integrated logic analyser and ILA (Integrated Logic Analyzer) cores.
- You can also set up probes and triggers to monitor signals and diagnose issues in your design.

## **8. Software Development (SoC Only):**

- If you're working on an SoC design, you can develop software applications to run on the embedded processors (e.g., ARM Cortex-A9) in the device.
- Create, compile, and debug software code within the Vivado environment.

## **9. Testing and Validation:**

- After programming the FPGA or SoC, thoroughly test the hardware and software components to ensure they meet the design specifications.

## **10. Documentation and Project Management:**

- Create project documentation, including design specifications, reports, and any necessary user manuals.
- Manage version control and project files within Vivado.

## **11. Deployment:**

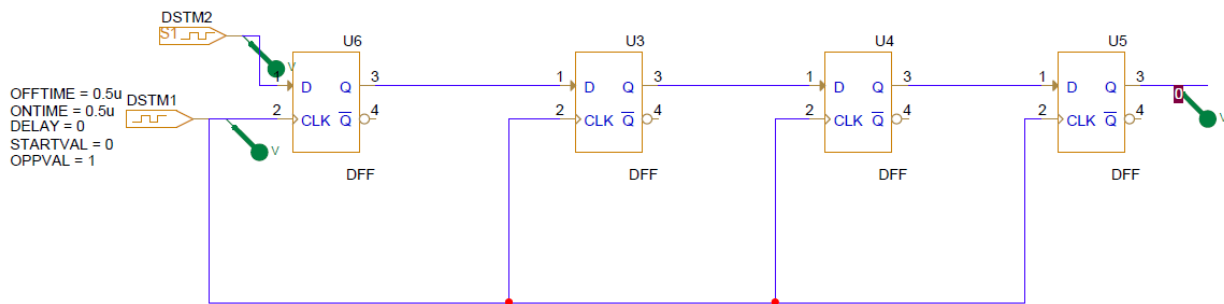
- If your design meets the required performance and functionality criteria, you can deploy it in your target application or product.

## **12. Optimization and Iteration:**

- If necessary, iterate through the design flow to make improvements or optimizations based on testing and validation results.

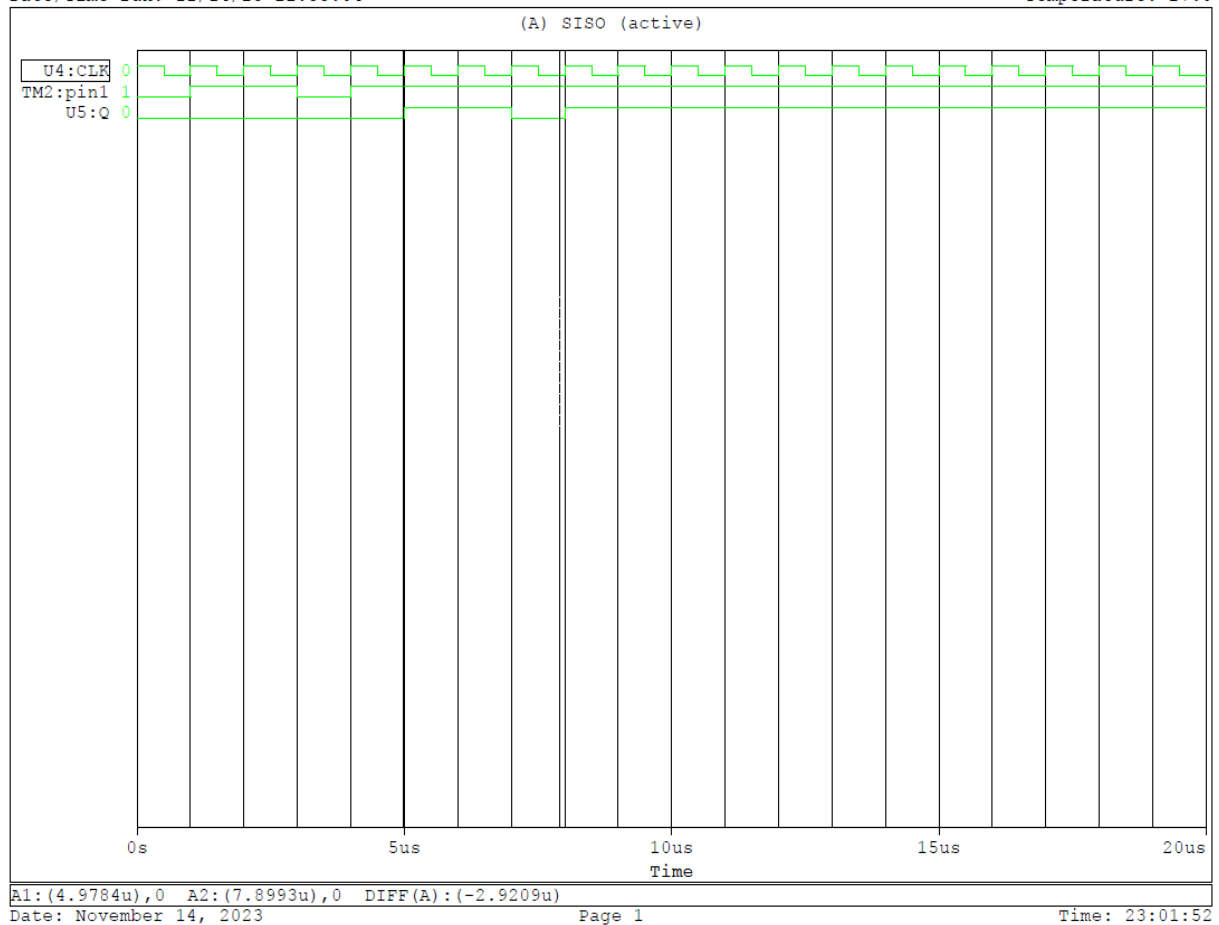
The Xilinx Vivado design flow provides a structured and comprehensive framework for creating FPGA and SoC designs, from initial concept to deployment. It is a powerful tool for hardware designers and embedded system developers working with Xilinx devices.

## 2.4 Stimulation of Series in Series out (SISO) Shift Register Using PSPICE



**Fig. 2.1** Schematic Diagram of SISO

\*\* Profile: "SCHEMATIC1-SISO" [ C:\Users\raghv\OneDrive\Desktop\Vivado\Design1-PSpiceFiles\SCHEMATIC1\S...  
Date/Time run: 11/14/23 22:55:04 Temperature: 27.0



**Fig. 2.2** Stimulation of SISO

# **CHAPTER- 3**

## **PROJECT DESCRIPTION**

### **3.1 Introduction to Alarm Clock**

An alarm clock is a device designed to awaken or alert individuals at a specific time, typically in the morning, to help them start their day on time. It is an essential tool for time management and ensuring punctuality. The primary function of an alarm clock is to produce a loud and attention-grabbing sound, often referred to as an alarm, at the predetermined time set by the user.

Traditional alarm clocks use various mechanisms to create sound, such as mechanical bells, chimes, or electronic beeps. Modern alarm clocks, however, often incorporate digital displays, radio receivers, and other features, allowing users to choose from a variety of alarm sounds or even wake up to music.

The basic components of an alarm clock typically include a clock face or digital display for timekeeping, buttons or controls for setting the alarm time, and the alarm sound mechanism. Many alarm clocks also come with additional features, such as snooze buttons, radio functionality, built-in ambient lighting, and even smartphone integration.

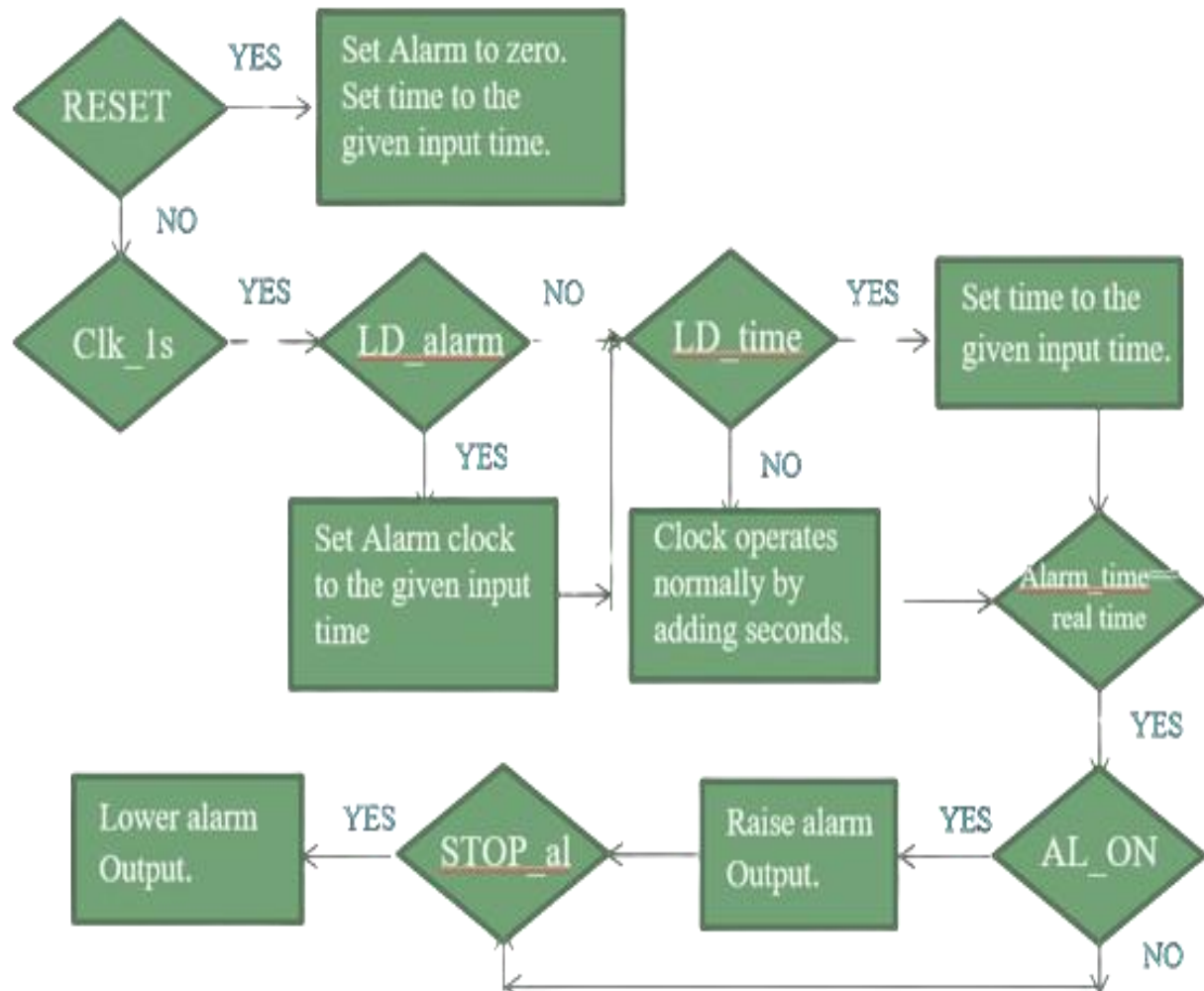
Over time, alarm clocks have evolved to meet changing consumer needs and preferences. While traditional standalone alarm clocks are still prevalent, many people now use their smartphones or other electronic devices as alarm clocks, thanks to their versatility and portability. Despite these changes, the fundamental purpose of alarm clocks remains the same – to provide a reliable and effective means of waking individuals at their desired time.

### **3.2 Features of Alarm Clock**

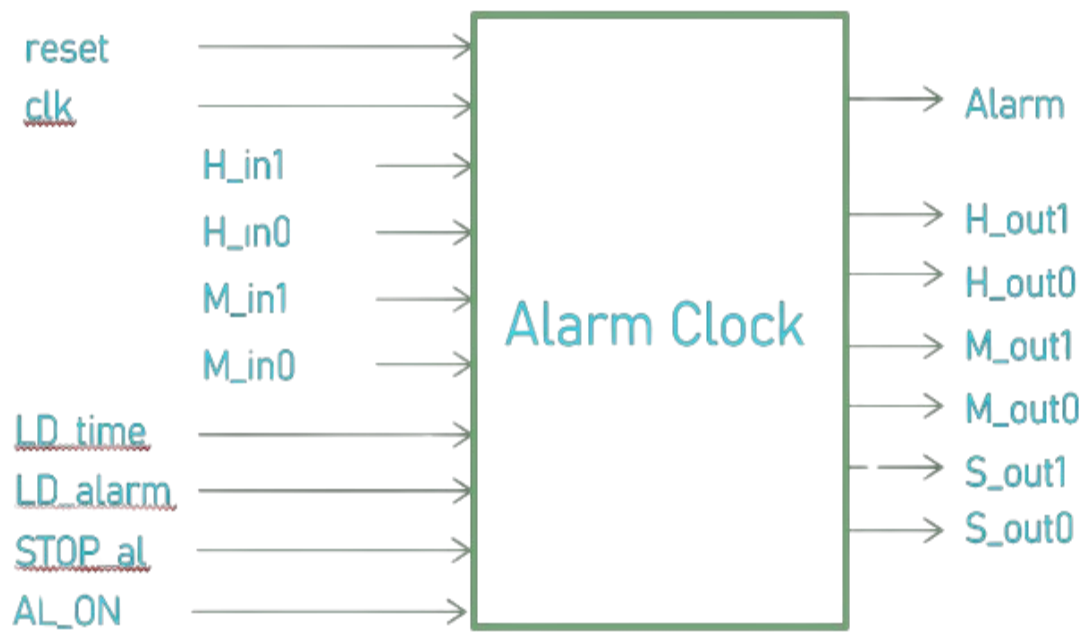
- Clock generation.
- Initializing clock time to a particular value.
- Setting time for alarm.
- Enabling and disabling alarm.
- Stopping alarm.

## 3.2 Description of Project

### 3.2.1 Flow Chart



### 3.2.2 Block Diagram





### 3.2.3 Verilog Code

```
module Aclock(  
input reset,  
input clk,  
input [1:0] H_in1,  
input [3:0] H_in0,  
input [3:0] M_in1,  
input [3:0] M_in0,  
input LD_time,  
input LD_alarm,  
input STOP_al,  
input AL_ON,  
output reg Alarm,  
output [1:0] H_out1,  
output [3:0] H_out0,  
output [3:0] M_out1,  
output [3:0] M_out0,  
output [3:0] S_out1,  
output [3:0] S_out0);
```

**Initialization**

```
reg clk_1s;  
reg [3:0] tmp_1s;  
reg [5:0] tmp_hour, tmp_minute, tmp_second;  
reg [1:0] c_hour1,a_hour1;  
reg [3:0] c_hour0,a_hour0;  
reg [3:0] c_min1,a_min1;  
reg [3:0] c_min0,a_min0;  
reg [3:0] c_sec1,a_sec1;  
reg [3:0] c_sec0,a_sec0;
```

```
function [3:0] mod_10;  
input [5:0] number;  
begin  
mod_10 = (number >= 50) ? 5 : ((number >= 40)? 4 : ((number >= 30)?  
3 : ((number >= 20)? 2 : ((number >= 10)? 1 : 0))));  
end  
endfunction
```

**MOD 10 function**

```
always @(posedge clk_1s or posedge reset )  
begin  
if(reset) begin  
a_hour1 <= 2'b00;  
a_hour0 <= 4'b0000;  
a_min1 <= 4'b0000;  
a_min0 <= 4'b0000;  
a_sec1 <= 4'b0000;  
a_sec0 <= 4'b0000;
```

```

tmp_hour <= H_in1*10 + H_in0;
tmp_minute <= M_in1*10 + M_in0;
tmp_second <= 0;
end
else begin
if(LD_alarm) begin
a_hour1 <= H_in1;
a_hour0 <= H_in0;
a_min1 <= M_in1;
a_min0 <= M_in0;
a_sec1 <= 4'b0000;
a_sec0 <= 4'b0000;
end
if(LD_time) begin
tmp_hour <= H_in1*10 + H_in0;
tmp_minute <= M_in1*10 + M_in0;
tmp_second <= 0;
end
else begin
tmp_second <= tmp_second + 1;
if(tmp_second >=59) begin
tmp_minute <= tmp_minute + 1;
tmp_second <= 0;
if(tmp_minute >=59) begin
tmp_minute <= 0;
tmp_hour <= tmp_hour + 1;
if(tmp_hour >= 24) begin
tmp_hour <= 0;
end
end
end
end

end
end
end

```

```

always @(posedge clk or posedge reset)
begin
if(reset)
begin
tmp_1s <= 0;
clk_1s <= 0;
end
else begin
tmp_1s <= tmp_1s + 1;
if(tmp_1s <= 5)
clk_1s <= 0;
else if (tmp_1s >= 10) begin

```

### Loading and Incrementing time

**Make 1s clock**

```
clk_1s <= 1;
tmp_1s <= 1;
end
else
clk_1s <= 1;
end
end
```

```
always @(*) begin
```

```
if(tmp_hour>=20) begin
c_hour1 = 2;
end
else begin
if(tmp_hour >=10)
c_hour1 = 1;
else
c_hour1 = 0;
end
c_hour0 = tmp_hour - c_hour1*10;
c_min1 = mod_10(tmp_minute);
c_min0 = tmp_minute - c_min1*10;
c_sec1 = mod_10(tmp_second);
c_sec0 = tmp_second - c_sec1*10;
end
```

```
always @(posedge clk_1s or posedge reset)
```

```
begin
if(reset)
Alarm <=0;
else begin
if({a_hour1,a_hour0,a_min1,a_min0}=={c_hour1,c_hour0,c_min1,c_min0})
begin
if(AL_ON) Alarm <= 1;
end
if(STOP_al) Alarm <=0;

end
end
```

```
assign H_out1 = c_hour1;
assign H_out0 = c_hour0;
assign M_out1 = c_min1;
assign M_out0 = c_min0;
assign S_out1 = c_sec1;
assign S_out0 = c_sec0;
```

**Output time**

```

always @(posedge clk_1s or posedge reset)
begin
  if(reset)
    Alarm <=0;
  else begin
    if({a_hour1,a_hour0,a_min1,a_min0,a_sec1,a_sec0}=={c_hour1,c_hour0,c_min1,
c_min0,c_sec1,c_sec0})
      begin / if(AL_ON) Alarm <= 1;
      end
    if(STOP_al)
      Alarm <=0;
    end
  end
end

endmodule

```

**Setting and  
Disabling alarm**

### 3.2.4 Test Bench Code

```
module Testbench;

reg reset;

reg clk;

reg [1:0] H_in1;
reg [3:0] H_in0;

reg [3:0] M_in1; reg [3:0] M_in0; reg LD_time; reg LD_alarm;

reg STOP_al;

reg AL_ON;

// Outputs
wire Alarm;

wire [1:0] H_out1;
wire [3:0] H_out0;

wire [3:0] M_out1;
wire [3:0] M_out0;

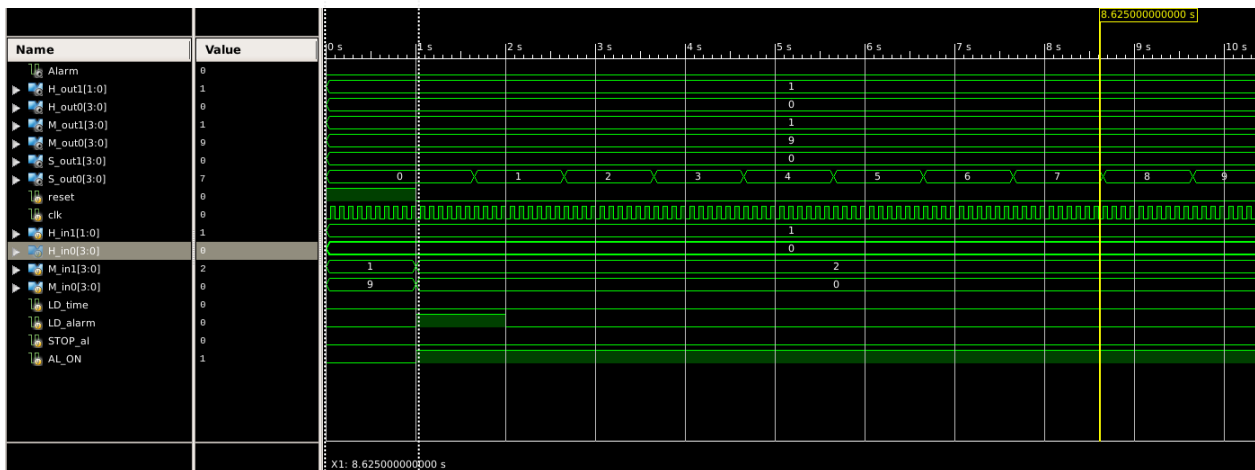
wire [3:0] S_out1;
wire [3:0] S_out0;

Aclock uut ( .reset(reset), .clk(clk), .H_in1(H_in1), H_in0(H_in0), .M_in1(M_in1),
.M_in0(M_in0), .LD_time(LD_time), .LD_alarm(LD_alarm), .STOP_al(STOP_al),
.AL_ON(AL_ON), .Alarm(Alarm), H_out1(H_out1), H_out0(H_out0), .M_out1(M_out1),
.M_out0(M_out0), .S_out1(S_out1), .S_out0(S_out0) );
```

# CHAPTER- 4

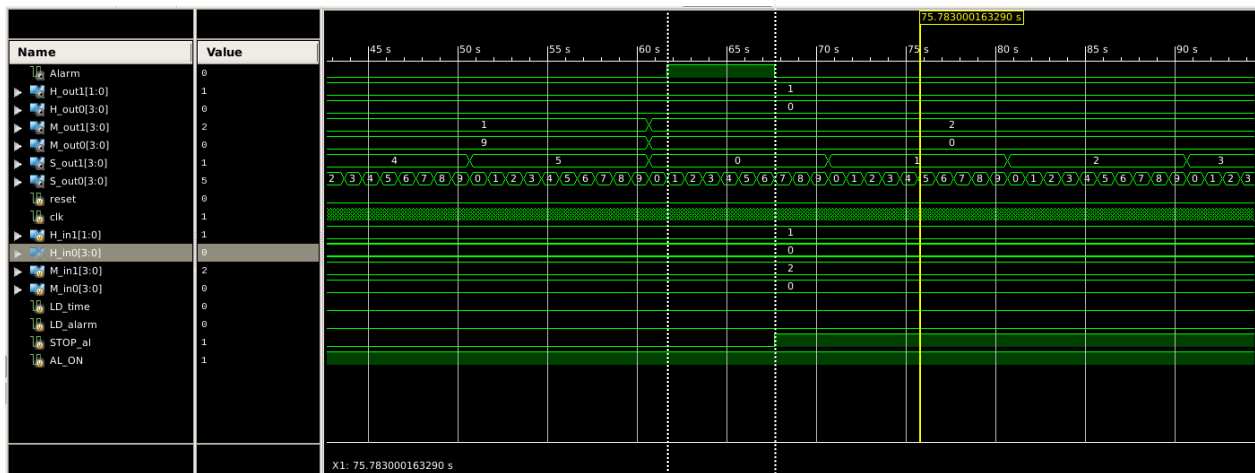
## SIMULATION & SYNTHESIS RESULTS

### 4.1 Stimulation



reset=1

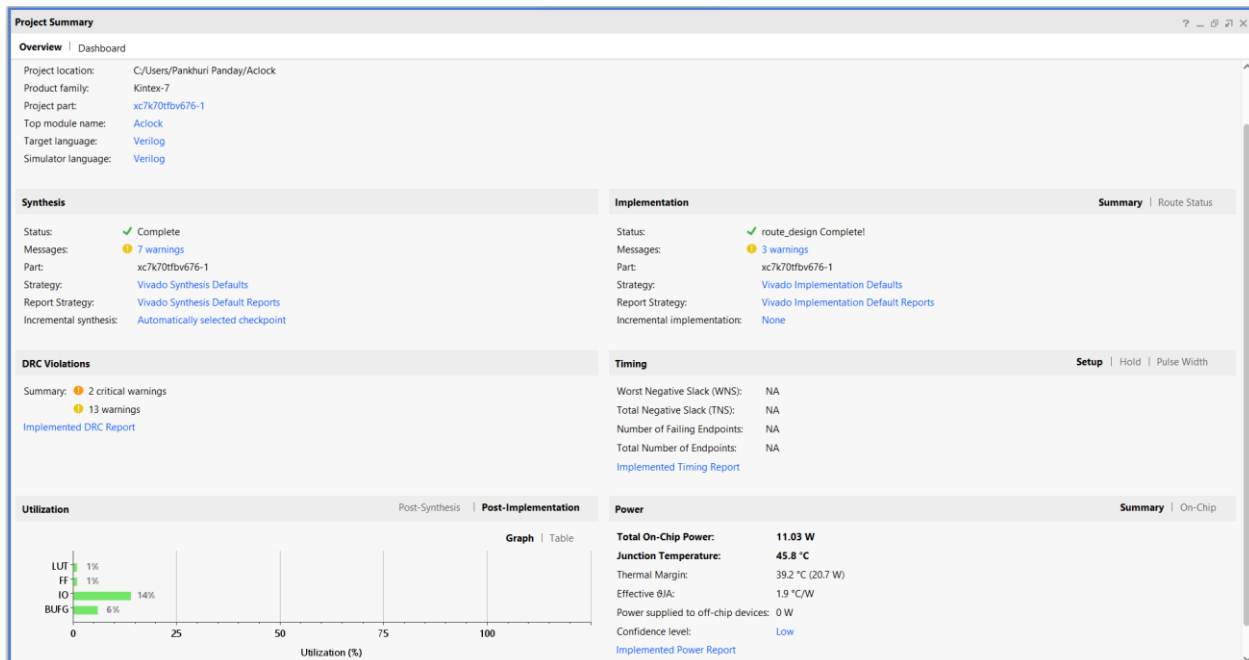
LD\_alarm=1



Alarm = 1

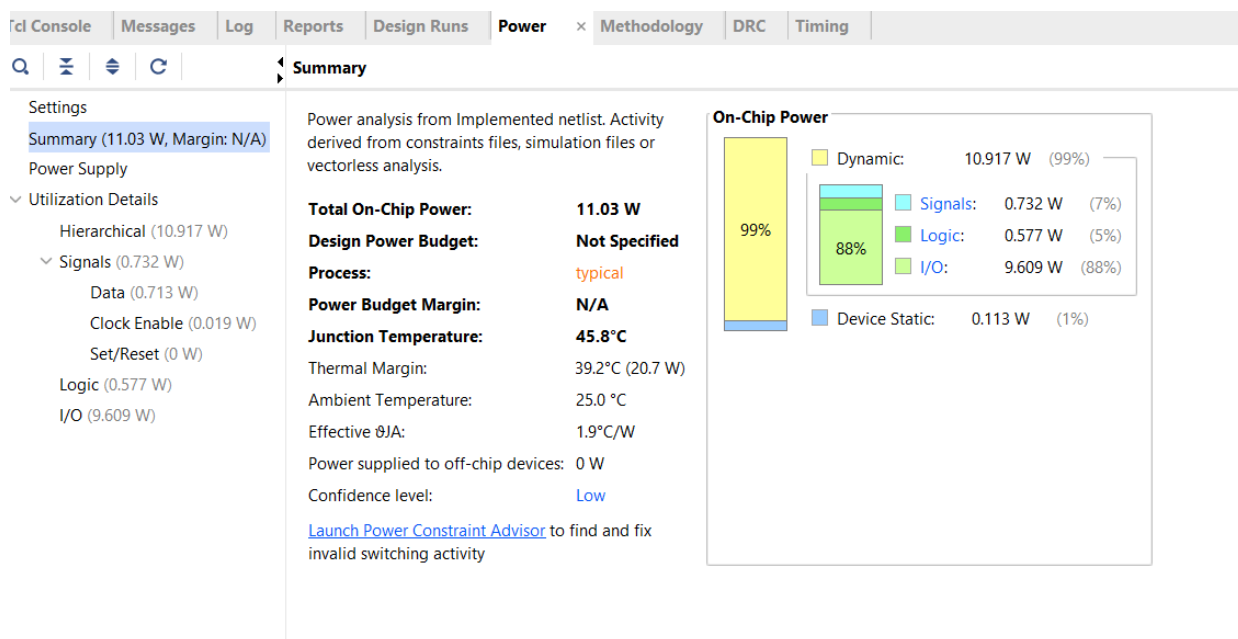
STOP\_al = 1

## 4.2 Project Summary



## 4.3 Reports

### ● Power



## ● *Synthesis report*

```

76 Finished RTL Component Statistics
77 -----
78
79 Start Part Resource Summary
80 -----
81 Part Resources:
82 DSPs: 240 (col length:80)
83 BRAMs: 270 (col length: RAMB18 80 RAMB36 40)
84 -----
85 Finished Part Resource Summary
86 -----
87 No constraint files found.
88 -----
89 Start Cross Boundary and Area Optimization
90 -----
91 WARNING: [Synth 8-7080] Parallel synthesis criteria is not met
92 WARNING: [Synth 8-3917] design Aclock has port M_out1[3] driven by constant 0
93 WARNING: [Synth 8-3917] design Aclock has port S_out1[3] driven by constant 0
94 -----
95 Finished Cross Boundary and Area Optimization : Time (s): cpu = 00:00:17 ; elapsed = 00:00:48 . Memory (MB): peak = 1220.352 ; gain = 735.898
96 -----
97 No constraint files found.
98 -----
99 Start Timing Optimization
100 -----
101
102 Finished Timing Optimization : Time (s): cpu = 00:00:17 ; elapsed = 00:00:48 . Memory (MB): peak = 1220.352 ; gain = 735.898
103 -----
104
105 Start Technology Mapping
106 -----

```

## ● *Utilization*

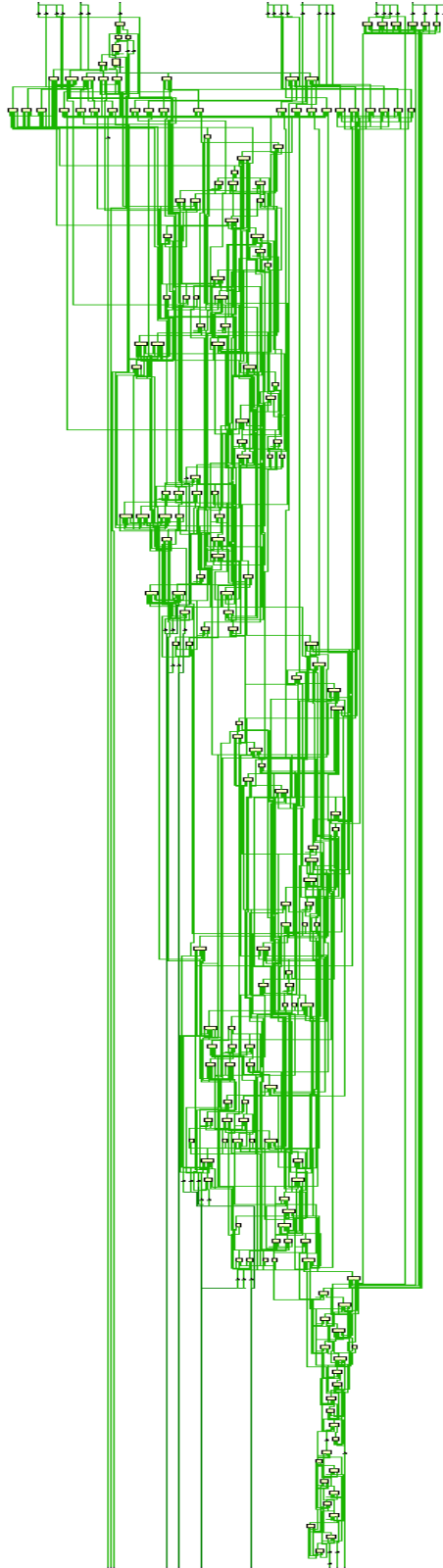
```

13 Utilization Design Information
14
15 Table of Contents
16 -----
17 1. Slice Logic
18 1.1 Summary of Registers by Type
19 2. Slice Logic Distribution
20 3. Memory
21 4. DSP
22 5. IO and GT Specific
23 6. Clocking
24 7. Specific Feature
25 8. Primitives
26 9. Black Boxes
27 10. Instantiated Netlists
28
29 1. Slice Logic
30 -----
31
32 +-----+-----+-----+-----+-----+-----+
33 | Site Type | Used | Fixed | Prohibited | Available | Util% |
34 +-----+-----+-----+-----+-----+-----+
35 | Slice LUTs | 105 | 0 | 0 | 41000 | 0.26 |
36 | LUT as Logic | 105 | 0 | 0 | 41000 | 0.26 |
37 | LUT as Memory | 0 | 0 | 0 | 13400 | 0.00 |
38 | Slice Registers | 62 | 0 | 0 | 82000 | 0.08 |
39 | Register as Flip Flop | 50 | 0 | 0 | 82000 | 0.06 |
40 | Register as Latch | 12 | 0 | 0 | 82000 | 0.01 |
41 | F7 Muxes | 0 | 0 | 0 | 20500 | 0.00 |
42 | F8 Muxes | 0 | 0 | 0 | 10250 | 0.00 |

```



## 4.4 Synthesis



# CHAPTER- 5

## Conclusion and Future Scope

### 5.1 CONCLUSION

In conclusion, the project has provided valuable insights into the world of digital design, FPGA programming, and hardware development. The use of Xilinx Vivado has demonstrated its efficiency in implementing complex functionalities and managing the synthesis, implementation, and programming processes seamlessly.

As we reflect on the journey of designing an alarm clock, we recognize the importance of understanding the hardware components, coding in HDL, and optimizing the design for the target FPGA device. The integration of clocking, signal processing, and user interface elements showcases the versatility of Vivado and the FPGA platform.

Furthermore, the project emphasizes the significance of debugging and troubleshooting in the development cycle. Overcoming challenges and refining the design not only enhances technical skills but also instills a problem-solving mindset crucial for any engineering endeavor.

In the end, this alarm clock project with Xilinx Vivado serves as a stepping stone for further exploration into digital design, FPGA applications, and real-world embedded systems. The skills acquired and lessons learned during this project lay a solid foundation for tackling more advanced and intricate projects in the realm of digital hardware development.

### 5.2 FUTURE SCOPE

The future scope of an alarm clock project using Xilinx Vivado, or any FPGA-based project, can involve various aspects of improvement, expansion, and integration with emerging technologies. Here are some potential directions for future development:

#### 1. *User Interface Enhancement:*

- Improve the user interface by incorporating touchscreens or graphical displays for a more interactive experience.
- Implement voice recognition for setting alarms and interacting with the alarm clock.

#### 2. *Wireless Connectivity:*

- Integrate Wi-Fi or Bluetooth modules to enable remote control and synchronization with other devices.

- Implement features such as syncing with online calendars or fetching weather information to provide additional context for the user.

### **3. *Advanced Alarm Features:***

- Implement intelligent alarm features, such as adjusting the alarm time based on the user's sleep cycle.
- Integrate sensors to monitor ambient conditions and wake the user up at an optimal time in their sleep cycle.

### **4. *Security and Authentication:***

- Integrate biometric authentication (fingerprint, facial recognition) for enhanced security.
- Implement features such as secure alarm setting to prevent unauthorized access.

### **5. *Integration with Smart Home Systems:***

- Connect the alarm clock to smart home systems to control lights, thermostats, or other smart devices upon waking up or going to sleep.
- Implement IoT (Internet of Things) protocols for seamless integration with other smart devices in the home.

### **6. *Customization and Personalization:***

- Allow users to customize alarm sounds, display themes, and other personal preferences.
- Implement machine learning algorithms to learn user preferences over time and adapt the alarm settings accordingly.

### **7. *Energy Efficiency:***

- Optimize power consumption for longer battery life (if applicable) or to reduce energy consumption when connected to the power grid.

### **8. *Health Monitoring:***

- Integrate health monitoring features, such as heart rate or sleep pattern analysis, to provide users with insights into their well-being.

### **9. *Cross-Platform Compatibility:***

- Develop companion mobile apps or web interfaces for remote control and monitoring of the alarm clock.
- Ensure compatibility with different operating systems and devices.

### **10. *Community and Social Features:***

- Implement social sharing features to allow users to share their waking up routines or achievements.
- Create a community platform where users can exchange custom alarm profiles and settings.

### **11. *Firmware and Software Updates:***

- Establish a mechanism for firmware and software updates to add new features, improve security, and fix bugs.

Remember to consider user feedback and market trends to stay relevant and meet the evolving needs of users. Additionally, advancements in FPGA technology and tools may also provide opportunities for optimization and new features in future versions of your alarm clock project.

## REFERENCES

- Wikipedia (The Free Encyclopedia)  
<https://en.wikipedia.org/wiki/Verilog>
- JavaPoint  
<https://www.javatpoint.com/verilog>
- Stephen Brown,& Zvonko Vranesic(2014).Fundamental of Digital Logic with Verilog Design
- YouTube  
<https://www.youtube.com/watch?v=pTk1H50e8b>
- Github  
<https://github.com/Arjun-Narula/Clock-with-Alarm>